Automatic verification of tasks schedulers

Josselin Giet

PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École normale supérieure

Vérification automatique d'ordonnanceurs de tâches

Automatic verification of tasks schedulers

Soutenue par
Josselin GIET
Le 26 septembre 2024

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Préparée au
Département d'informatique de l'École normale supérieure

Composition du jury :

Arthur CHARGUÉRAUD Chargé de recherche, INRIA, Nancy	*Rapporteur*
Jean-Christophe FILLIÂTRE Directeur de recherche, CNRS, Saclay	*Rapporteur*
Julia LAWALL Directrice de recherche, INRIA, Paris	*Examinatrice*
Matthieu LEMERRE Chargé de recherche, CEA, Saclay	*Examinateur*
Marc POUZET Professeur des Universités, ÉNS, Paris	*Examinateur*
Xavier RIVAL Directeur de recherche, INRIA, Paris	*Directeur de thèse*

ENS | PSL

Abstract

The aim of this thesis is the verification of task schedulers for operating systems through static analysis based on abstract interpretation. Operating systems are collections of software present on almost every computer. Their purpose is to allow other programs to run without having to manage low-level problems such as memory. Due to this central role, operating systems have become critical components of IT infrastructures: any error in the operating system can have consequences on other programs, potentially causing the entire computer to crash.

One component at the core of an operating system is the task scheduler. This component is responsible for determining, according to a predefined policy, which task can execute at what time. These components use unbounded dynamic data structures to store the necessary elements for their operation. These data structures allow elements to be easily moved between them. Verifying a task scheduler requires designing an analysis capable of accurately representing these data structures and their contents.

The first part of this thesis describes a toy imperative language that explicitly manipulates memory. We then provide the concrete semantics of this language, followed by a presentation of a numerical static analysis to determine the range of numerical variables and a shape analysis capable of reasoning about unbounded inductive data structures.

The second part is devoted to presenting a relational abstract domain capable of reasoning about symbolic sequences. This domain expresses constraints on the contents of these sequences, such as their lengths, extreme values, and sorted characteristics.

The third part presents the combination of the shape analysis described in the first part with the sequence domain. This combination enhances the expressiveness of the analysis. It is now capable of proving the partial functional correctness of complex algorithms, such as sorting algorithms on lists or binary trees, as well as list libraries drawn from real applications.

The final part of this thesis presents the application of the analysis to an instance of the FreeRTOS task scheduler. The first step in the verification process is formalizing the properties we seek to establish on the scheduler's functions. The second step aims to show that the specified properties are verified by the instance's functions using the analysis.

Keywords: Abstract interpretation, Operating system, scheduler, separation logic

Résumé

Le but de cette thèse est la vérification d'ordonnanceurs de tâches de systèmes d'exploitation par analyse statique basée sur l'interprétation abstraite. Les systèmes d'exploitation sont des ensembles de logiciels présents sur presque tout ordinateur. Leur but est de permettre aux autres programmes de s'exécuter sans avoir à gérer des spécificités bas niveau comme la mémoire. En conséquence de ce rôle central, les systèmes sont devenus des composants critiques des infrastructures informatiques : toute erreur au niveau du système d'exploitation peut avoir des conséquences sur les autres programmes allant jusqu'au plantage de l'ordinateur.

Un des composants au cœur d'un système d'exploitation est l'ordonnanceur de tâches. Ce dernier est chargé de déterminer quelle tâche peut s'exécuter à quel moment, en suivant une politique préétablie. Les ordonnanceurs de tâches utilisent des structures de données dynamiques non bornées afin de stocker les éléments nécessaires à leur fonctionnement. Ces structures de données permettent de déplacer facilement les éléments d'une structure vers l'autre. Par conséquent, la vérification d'un ordonnanceur de tâche nécessite de concevoir une analyse capable de représenter correctement ces structures de données et leur contenu.

La première partie de cette thèse décrit un langage impératif jouet semblable au C manipulant explicitement la mémoire. On donne ensuite la sémantique concret de ce langage, puis on présente une analyse statique numérique afin de déterminer la plage de valeur des variables ainsi qu'une analyse de forme capable de raisonner sur des structures de données inductives non bornées.

La seconde partie est consacrée à la présentation d'un domaine abstrait relationnel capable de raisonner sur des séquences symboliques. Ce domaine exprime des contraintes sur le contenu de ces séquences comme leurs longueurs, leurs valeurs extrémales et leurs caractères triés.

La troisième partie présente la combinaison de l'analyse de forme présentée dans la première partie avec le domaine de séquences. Cette combinaison augmente l'expressivité de l'analyse. Cette dernière est maintenant capable de prouver la correction fonctionnelle partielle d'algorithmes complexes comme des algorithmes de tris sur les listes ou les arbres binaires, ainsi que sur des bibliothèques de listes provenant d'applications réelles.

La dernière partie de cette thèse présente le travail d'application de l'analyse sur une instance de l'ordonnanceur de tâches de FreeRTOS. La première étape de la vérification est la formalisation des propriétés que nous cherchons à établir sur les fonctions de l'ordonnanceur. Cela inclut les invariants globaux de l'ordonnanceur. La seconde étape concerne le travail de validation pour montrer que ces propriétés spécifiées sont vérifiées par les fonctions de l'instance au moyen de l'analyse.

Mots-clés : Interpretation abstraite, Systèmes d'exploitation, Ordonnanceurs, Logique de séparation

Résumé étendu

Motivation

L'Importance des systèmes d'exploitations

Pour motiver ce travail, il est essentiel de souligner l'importance des systèmes d'exploitation. Un système d'exploitation, abrégé par la suite OS pour *Operating System*, forme l'interface entre les *applications utilisateurs* qui s'exécutent sur un ordinateur (comme un navigateur internet ou un navigateur de fichier) et le matériel, c'est-à-dire le processeur, la mémoire et autres composants physiques de l'ordinateur, sur lequel s'exécutent ces applications. En résumé, un système d'exploitation sert à faire tourner d'autres programmes.

Pour fournir un environnement d'exécution aux applications de l'utilisateur, un système d'exploitation remplissent des missions qui peuvent être réparties en deux grandes familles :

- **Abstraire le matériel** En l'absence de système d'exploitation, toute application devrait être conçue afin de s'exécuter sur un matériel spécifique. Concevoir une application *bare metal* demande de gérer les détails bas niveau du matériel comme la phase de démarrage de l'ordinateur ou les interruptions matérielles.
- **Répartir les ressources entre applications** Un ordinateur fournit des ressources pour permettre aux applications de s'exécuter. Il y a par exemple le temps d'exécution sur le processeur ou l'utilisation de la mémoire. Les applications peuvent faire des requêtes de ressources qui sont incompatibles. Il revient alors au système d'exploitation de décider comment allouer les ressources en cas de conflit.

Les systèmes d'exploitations sont donc des composants essentiels des infrastructures informatiques. Mais ils sont aussi des composants critiques. En effet, contrairement aux erreurs dans une application de l'utilisateur qui n'impacte que cette dernière, toute erreur au niveau du système d'exploitation peut impacter tout le reste de l'ordinateur. Cela peut aller de l'impossibilité d'utiliser un composant matériel à un plantage global de l'ordinateur.

Ce travail part donc du constat que les systèmes d'exploitation sont critiques. Il cherche dès lors comment améliorer la confiance dans les systèmes d'exploitation.

Améliorer la confiance dans les systèmes d'exploitation : But

La première partie de la réponse concerne le but : Quelle propriété cherchons-nous à établir ? Cette section présente, sans prétention d'exhaustivité, quelques propriétés sur les exécutions de programmes intéressantes dans le cadre des systèmes d'exploitation

- **Absence d'erreur à l'exécution** Cette propriété énonce que le système d'exploitation ne va pas planter. Dans le cas précis du langage C, principale langage utilisé pour la conception de systèmes d'exploitation, cela revient à montrer que le programme ne peut pas atteindre un état de *undefined behavior* [2218].
- **Préservation d'invariants** Cette propriété revient à dire qu'à certains instants de l'exécution du programme, son état doit rester dans un ensemble donné. Dans le cadre d'un programme manipulant des structures de données dynamiques (comme des listes chaînées) on peut distinguer les *invariants de structures* des *invariants fonctionnels*. Les premiers énoncent que la liste doit rester bien formée vis-à-vis de la disposition mémoire, tandis que les seconds expriment des propriétés concernant le contenu de la liste comme le fait que la liste est triée par ordre croissant.

- **Correction fonctionnelle partielle** La correction fonctionnelle partielle d'un programme énonce que si le programme termine sans erreurs, alors il a effectué les opérations attendues. Cette propriété est donc énoncée différemment pour chaque programme sous la forme de triplet de Hoare [Hoa69].

- **Terminaison et propriétés de vivacité** Une autre propriété important pour les systèmes est la terminaison. Cette propriété est dite de vivacité, car tout comportement fini d'un programme peut potentiellement satisfaire cette propriété. Parmi les autres propriétés de vivacité importantes dans le contexte des systèmes d'exploitation, on peut mentionner l'absence de famine : toute requête faite au système d'exploitation doit être traitée.

- **Concurrence** L'exécution du programme d'un système d'exploitation n'est pas complètement séquentiel. En effet, les interruptions matérielles ou les architectures multicœurs introduisent de la concurrence. Il faut donc veiller à ce que l'enchevêtrement des fils d'exécution n'a pas d'incidence sur le résultat, c'est l'*absence de course critique*. On peut aussi chercher à montrer que les mécanismes de verrouillage utilisés pour éviter que deux programmes accèdent en même temps à la même ressource ne puissent pas causer de blocage.

- **Sécurité** De manière informelle la sécurité d'un système d'exploitation correspond à une famille de propriétés afin de prévenir, résister ou bien affaiblir une menace malicieuse. Cela consiste, par exemple, à s'assurer qu'un attaquant ne peut pas apprendre d'information sur une donnée cachée en inspectant des données publiques.

Améliorer la confiance dans les systèmes d'exploitation : Moyens

Après avoir présenté les propriétés d'intérêt, on peut maintenant chercher quels sont les moyens existants afin d'établir ces propriétés. Cette section présente quelques méthodes utilisées pour chercher si un programme vérifie ou non une propriété.

Le premier point dans cette section concerne le théorème d'impossibilité de Rice [Ric53]. Ce dernier énonce que toute propriété sémantique non triviale est indécidable. Autrement dit pour toute propriété sur l'exécution d'un programme, il n'existe aucune méthode automatique (c'est-à-dire qui termine toujours et ne requiert aucune aide de l'utilisateur), correcte (sans faux négatifs) et complète (sans fausses alarmes) permettant de déterminer si un programme satisfait ou non cette propriété. Il faut donc relâcher les contraintes sur la méthode de vérification.

Si on cherche une méthode automatique et complète, on peut tester le programme en l'exécutant dans quelques scénarios possibles. Si une erreur est rencontrée, alors cette erreur correspond bien à une défaillance possible du programme. Cependant, comme il est soit trop coûteux soit tout simplement impossible de tester un programme sur tous les comportements possibles, on ne peut pas conclure par des tests qu'un programme vérifie une propriété : c'est une méthode incorrecte.

On peut aussi prouver des propriétés sur des programmes par des méthodes correctes, complètes, mais non automatiques. Dans ce cas, il revient à l'utilisateur d'écrire la preuve (potentiellement aidé par un solveur) pour montrer que le programme satisfait bien la propriété dans un assistant de preuve. Ces méthodes sont très expressives. En effet, l'utilisation d'un assistant de preuve permet de raisonner sur n'importe quelle propriété exprimable dans la logique de l'assistant, souvent une logique d'ordre supérieure. En revanche, le coût de la vérification par ces méthodes est important. L'utilisateur doit fournir un grand nombre d'étapes et toute modification du programme par la suite demande aussi de modifier la preuve.

La troisième famille de méthodes concernent les méthodes correctes et automatiques. Parmi ces dernières se trouve l'analyse statique par interprétation abstraite. Celle-ci cherche à calculer une sur-approximation des comportements du programme. Si cette approximation vérifie la propriété attendue, alors par implication on en déduit que toute exécution du programme satisfait cette propriété. En revanche dans le cas où l'approximation n'est pas incluse dans la propriété, alors il est impossible de savoir si cela est due à l'existence d'une exécution incorrecte du programme ou à une perte de précision dans le calcul de la sur-approximation des comportements d'un programme. Ces méthodes sont donc conçues dans le but de vérifier une propriété spécifique.

Ce travail cherche donc à concevoir une analyse statique par interprétation abstraite pour des propriétés plus riches que celles étudiées jusqu'à présent.

Comme la vérification d'un système d'exploitation entier est une tâche trop importante, nous nous concentrons ici sur un composant qui forme le cœur d'un système d'exploitation : l'ordonnanceur de

tâches. Ces programmes correspondent à la deuxième famille de missions d'un système d'exploitation, car il est chargé de répartir le temps d'utilisation du processeur en répondant à la question « Quel programme doit tourner maintenant ? »

Ces composants manipulent les processus en cours d'exécution dans des structures de données. Il existe deux groupes de structures de données. Le premier groupe concerne les structures de données statiques comme les tableaux. Pour ces structures, l'empreinte mémoire ne varie pas au cours du temps. Elles sont donc moins versatiles que celles du deuxième groupe : les structures de donnée dynamiques. Dans ces structures, un élément peut être ajouté ou supprimé facilement en modifiant seulement quelques valeurs.

Nous nous concentrons ici sur la vérification de la correction fonctionnelle partielle pour des programmes manipulant des structures de données dynamiques dans le but de vérifier des ordonnanceurs de tâches.

Dans cette thèse, nous faisons les contributions suivantes :

- Dans le chapitre 3 nous présentons un nouveau domaine abstrait capable de raisonner sur des séquences de valeurs. Ce domaine est capable d'exprimer des contraintes relationnelles sur le contenu des séquences comme le fait qu'une séquence est le résultat de la concaténation d'autres séquences.

  Ce domaine de séquence est paramétré par des domaines sous-jacents pour raisonner sur les attributs de ces séquences. En utilisant un domaine de multi-ensemble, le domaine de séquence est capable d'inférer des contraintes exprimant que deux séquences ont le même contenu. De plus, au moyen d'un domaine numérique, le domaine de séquence peut exprimer des contraintes sur les longueurs des séquences ainsi que sur les éléments extrêmes contenus dans ces séquences. Ce domaine est ainsi capable d'établir qu'une séquence est triée.

- Dans le chapitre 4, nous présentons comment utiliser le domaine de séquence introduit au chapitre précédent en le combinant avec une analyse de forme basée sur la logique de séparation [CR08] afin d'exprimer des contraintes sur le contenu stocké dans des structures de données dynamiques.

  La combinaison du domaine des séquences avec l'analyse de forme requiert une extension des prédicats inductifs de la logique de séparation, afin de représenter la séquence des éléments contenus dans une structure de données. De plus, cette combinaison nécessite également une adaptation des opérateurs utilisés pour approximer la sémantique du programme, dans le but d'inférer des contraintes précises sur les séquences lors des opérations effectuées.

  Cette analyse a été implémentée dans l'analyseur MemCAD, et nous l'avons utilisée pour vérifier la correction fonctionnelle partielle de programmes manipulant des structures de données dynamiques comme des algorithmes de tri. Cette analyse a aussi été utilisée pour vérifier des bibliothèques de listes issues d'applications industrielles.

- Enfin, dans le chapitre 5, nous utilisons l'analyse présentée dans le chapitre précédent afin de vérifier la correction fonctionnelle partielle d'une instance de l'ordonnanceur de tâches de FreeRTOS. Cette instance se concentre sur les contraintes temps-réel de l'ordonnanceur, comme « si l'ordonnanceur est en cours d'exécution, aucune tâche placée en attente ne doit rester dans cet état dès lors que son délai a expiré ».

  Cette vérification est constituée de deux étapes. La première est la spécification de l'instance que nous cherchons à vérifier. Cette spécification comprend les invariants globaux de l'ordonnanceur de tâches ainsi sur pour chaque fonction les prés- et postconditions possibles. Le deuxième étape est le travail nécessaire pour vérifier cette spécification sur l'instance au moyen de l'analyse. Elle demande un travail de pour améliorer la précision ainsi que les performances de l'analyse.

La vérification par analyse statique basée sur l'interprétation abstraite

Le chapitre 2 présente les bases théoriques de ce travail.

Un Langage jouet : `MemImp`

La première étape dans les méthodes de vérification formelles est de donner la sémantique du langage dans lequel est écrit le programme que l'on cherche à vérifier. Nous présentons ainsi dans la Section 2.1 un langage jouet appelé `MemImp` qui se veut être un sous-ensemble du C. Ce langage manipule explicitement le mémoire au moyen de pointeurs. Toutefois, la manipulation de la mémoire est limitée par certains aspects du langage : Il est impossible de faire des allocations dynamiques de tailles inconnues à la compilation et il est uniquement possible de faire des adressages mémoire de la taille d'une valeur entière. La syntaxe de `MemImp` est présentée dans la Figure 2.1.

La sémantique de ce langage, appelée *sémantique concrète*, est donnée sous forme dénotationnelle. Une instruction du langage `MemImp` est vue comme une fonction qui prend en entrée un ensemble d'états mémoires de `MemImp` (définis dans la définition 2.1) et qui renvoie un autre ensemble qui correspond aux états possibles après l'exécution de l'instruction en partant de n'importe quel état dans l'ensemble d'entrée. Il s'agit des fonctions présentées dans les Figures 2.2 et 2.3. Par ailleurs, comme ce travail se concentre sur la correction fonctionnelle partielle, nous utilisons une sémantique dite *angélique* selon la classification proposée par Cousot [Cou02]. Ainsi, toute trace d'exécution d'une instruction qui ne termine pas n'est pas prise en compte dans le résultat de la sémantique.

Un Exemple simple d'analyse statique : une analyse numérique

Afin d'illustrer les concepts importants de l'analyse statique par interprétation abstraite, la Section 2.2 présente une analyse numérique où la valeur de chaque variable de `MemImp` est approximés par l'intervalle des valeurs possibles et en utilisant une valeur « je ne sais pas », notée $\top^\sharp_n$, dans le cas où le programme utilise des pointeurs.

Cette section permet de présenter les opérateurs nécessaires pour définir une *sémantique abstraite*, c'est-à-dire pour être capable de calculer une sur-approximation des états possibles obtenus par la sémantique concrète. Elle présente aussi, pour tous ces opérateurs les propriétés suffisantes sur ces opérateurs pour que l'analyse résultante soit correcte par construction.

Une Analyse de forme basée sur la logique de séparation

La Section 2.3 décrit succinctement une analyse de forme utilisant un sous-ensemble de la logique de séparation tirée de [CR08]. La logique de séparation [Rey02] permet de décrire un ensemble d'états mémoires par des prédicats. L'analyse de forme en utilise trois types :

- le prédicat *points-to*, $\alpha \mapsto \beta$ qui représente une seule cellule de la mémoire à l'adresse α contenant une valeur β
- les prédicats *inductifs complets* comme le prédicat **list**, présenté dans l'exemple 2.3.2 qui décrit une liste simplement chaînée,
- les prédicats *de segments inductifs*, qui représente des structures de données incomplètes. Par exemple, un segment de liste, présenté dans l'exemple 2.7 et noté $\alpha.\mathbf{list} \mathrel{-\!\!*} \beta.\mathbf{list}$ correspond à une liste partielle dont la première cellule est à l'adresse α et dont la dernière cellule pointe sur l'adresse β.

Ces prédicats sont reliés entre eux par l'opérateur qui donne son nom à la logique de séparation : la *conjonction séparante*, notée $*$. Cet opérateur exprime que les régions de la mémoire représentées par ces différents prédicats sont deux-à-deux disjointes.

La Section 2.4 présente la sémantique abstraite de cette analyse. Le principal opérateur utilisé est l'opérateur de pliage qui permet de préciser un état abstrait en remplaçant un prédicat inductif par ses possibles définitions. Cette présentation se conclut par un exemple d'analyse d'un programme effectuant une insertion dans un arbre binaire de recherche. Cet exemple illustre comment l'analyse de forme est capable de montrer l'absence d'erreurs à l'exécution et la préservation des invariants de structure, mais comme les prédicats inductifs sont incapables d'exprimer une quelconque information sur leur contenu, cette analyse ne parvient pas à prouver la préservation des invariants fonctionnels et la correction fonctionnelle partielle.

Pour augmenter l'expressivité d'une telle analyse, nous proposons d'ajouter des nouveaux paramètres de séquence aux prédicats inductifs.

Le Domaine de séquence

Le chapitre 3 présente un domaine capable de raisonner sur des séquences de valeurs.

Expressivité du domaine

La Section 3.1 introduit les prédicats logiques manipulés par le domaine de séquence.

Afin de raisonner sur les longueurs ainsi que sur les valeurs minimums et maximums stockées dans une séquence, le domaine de séquence attribut à chaque variable de séquence S des *variables d'attributs numériques* $\texttt{len}_S, \texttt{min}_S$ et $\texttt{max}_S$ qui correspondent respectivement à la longueur, la valeur maximale et la valeur minimale de S. Par ailleurs, pour exprimer des contraintes sur le contenu de la séquence S, le domaine utilise une *variable d'attribut de contenu* $\texttt{mset}_S$ qui correspond au multiensemble des éléments apparaissant dans la séquence S.

La première famille de contraintes sont les *contraintes de définitions*, de la forme $S = E$. Dans cette contrainte, S est une *variable symbolique de séquence*, qui représente une séquence de valeur et E est une *expression de séquence*. La définition formelle d'une expression de séquence est donnée dans la Définition 3.2 et leur sémantique est définie dans la Figure 3.2.

Le deuxième type de contraintes sont les *contraintes d'unicité* de la forme $\textbf{unique}(S)$. Ces contraintes expriment que les éléments dans la séquence S sont tous deux à deux différents. Les définitions formelles des expressions et contraintes de séquences. La définition des contraintes de séquence et leur sémantique est présentée dans la Définition 3.3.

La Section 3.2 présente les éléments du domaine abstrait de séquence. À haut niveau, un élément du domaine abstrait de séquence $\mathbb{D}_s^\sharp$ est ou bien un élément $\bot_s^\sharp$ correspondant à un état incohérent, ou bien un triplet constitué d'une conjonction finie de contrainte de séquences, avec un élément d'un domaine abstrait numérique et un élément d'un domaine abstrait de multiensemble. Ceci correspond à la définition présentée dans la Définition 3.4. La concrétisation du domaine est donnée dans la Définition 3.5.

Comme le but d'un domaine abstrait est de disposer d'une représentation efficace des éléments du domaine, la Section 3.2.3, présente la représentation machine des éléments du domaine de séquence. Cette représentation demande que certains invariants soient maintenus par les opérateurs du domaine. Par exemple, les variables de séquences connues comme étant égales à la séquence vide doivent être absentes des définitions des autres séquences.

Opérateurs du domaine

Le reste du chapitre 3 présente les opérateurs du domaine de séquence.

La Section 3.3 introduit les opérateurs $\textbf{supp}_s^\sharp$ (dans la Définition 3.7) et $\textbf{prune}_s^\sharp$ (dans la Figure 3.7). Le premier détermine quelles sont les variables restreintes par un élément abstrait du domaine. Le second opérateur enlève toute occurrence d'une variable dans un état abstrait.

La Section 3.4 présente les principaux opérateurs du domaine de séquence. Le premier opérateur définit dans cette section est l'opérateur $\textbf{guard}_s^\sharp$ qui rajoute une nouvelle contrainte dans un élément abstrait. Cet opérateur fonctionne en plusieurs étapes. Les premières étapes rajoutent la nouvelle contrainte dans l'état abstrait tout en maintenant les invariants de la représentation machine. Les étapes suivantes utilisent la nouvelle contrainte pour en inférer d'autres. En particulier, chaque contrainte de séquences est traduite en contraintes de multi-ensemble et numériques traduisant la préservation du contenu ou les égalités entre les longueurs de séquence. D'autres heuristiques infèrent de nouvelles contraintes de séquences en comparant plusieurs définitions d'une même variable de séquence.

Le second opérateur, $\textbf{sat}_s^\sharp$, vérifie de manière conservative si une contrainte est impliquée par un état abstrait. Cet opérateur utilise les éléments des domaines numérique et de multiensemble pour traiter certaines contraintes.

En utilisant les opérateurs précédents on définit automatiquement le premier opérateur de treillis du domaine de séquence : le test d'inclusion $\sqsubseteq_s^\sharp$ est défini en appliquent $\textbf{sat}_s^\sharp$ pour chaque contrainte dans l'entrée de droite sur l'état abstrait de gauche (*cf.* Définition 3.9). L'union abstraite $\sqcup_s^\sharp$, utilise un opérateur d'unification $\textbf{unify}_s^\sharp$, présenté dans la Figure 3.15, qui étant donné deux définitions d'une même variable dans les deux entrées, essaie de trouver une expression commune. Pour calculer les composantes du domaine de multi-ensemble et du domaine numérique, le domaine de séquence

utilise l'opérateur correspondant dans les domaines sous-jacents. Enfin, l'opérateur d'élargissement $\nabla_s^\sharp$ est défini similairement à l'opérateur d'union à une exception : les composantes numériques et de multiensemble du résultat sont calculées en * utilisant l'élargissement des domaines sous-jacents.

Analyse de forme utilisant des prédicats inductifs avec paramètres de séquence

Le chapitre 4 présente la combinaison de la logique de forme présentée dans la Section 2.3 et le domaine de séquence introduit dans le chapitre précédent. Cette combinaison, prenant le forme d'un produit réduit entre les deux domaines, repose sur l'ajout d'un nouveau type de paramètres dans les prédicats inductif : des paramètres de séquences.

Extension des prédicats inductifs avec des paramètres de séquence

La Section 4.1 présente comment ajouter les paramètres de séquences aux prédicats inductifs. Cet ajout permet aussi d'insérer dans les définitions de prédicats inductifs des contraintes de séquence entre les paramètres de séquences et les variables désignant ou bien l'adresse de la structure ou bien les valeurs stockées dedans. Par exemple, dans le prédicat inductif $\alpha.\mathbf{tree}(\kappa_p, S)$, défini dans la Figure 4.2, le paramètre de séquence S désigne la séquence des éléments stockés dans l'arbre binaire à l'adresse α, selon un parcours infixe.

Cette section introduit aussi une classification des paramètres de séquences. La première classe de paramètres correspond aux *paramètres additifs* (*cf.* Définition 4.3). De tels paramètres permettent de définir des prédicats de segments à partir du prédicat complet. En effet, si un prédicat inductif ne possède que des paramètres additifs, alors on peut modifier l'algorithme dérivant les prédicats de segments à partir des prédicats inductifs complets. Pour ce faire, on double le nombre des paramètres de séquences. Les deux paramètres de séquences du prédicat de segment représentent chacun la séquence d'un côté de la partie de la structure de donnée qui n'est pas représentée dans le segment. Ainsi, dans le prédicat de segment d'arbre $\alpha.\mathbf{tree}(\kappa_p) \not\models \{S_l \boxdot S_r\} \models \beta.\mathbf{tree}(\kappa'_p)$, défini dans la Figure 4.4, la variable de séquence S_l correspond à la séquence d'éléments à gauche du point d'insertion du sous-arbre manquant et S_r à la séquence à droite (*cf.* Figure 4.3).

Une fois les paramètres de segments définis, on peut énoncer les *lemmes de concaténation* (Lemmes 4.2 et 4.3). Ces lemmes établissent que la conjonction d'un segment et d'un prédicat inductif complet ou d'un segment, peut être affaiblie respectivement en un prédicat complet ou un prédicat de segment. Par ailleurs ces lemmes donnent des contraintes liant les paramètres de séquence dans la conjonction et dans l'état abstrait affaibli.

La deuxième classe de paramètres de séquences est la classe des *paramètres de tête* (*cf.* Définition 4.4). Cette classe est une sous-classe des paramètres additifs, correspondant aux paramètres exprimant la séquence des adresses des nœuds dans une structure de donnée. Dans ce cas, le Lemme 4.4 énonce qu'une telle séquence est sans répétition.

Enfin, la dernière classe, est la classe des paramètres *gauches* ou *droits* (*cf.* Définition 4.5). Ces paramètres permettent d'inférer qu'un des paramètres du prédicat de segment est toujours vide et peut être enlevé.

La Section 4.2 présente la construction du domaine abstrait résultant du produit réduit dans la Définition 4.6 ainsi que la concrétisation de ce domaine dans la Définition 4.8. Une conséquence importante de la concrétisation est que les variables symboliques numériques qui ne désignent pas l'adresse d'une variable ainsi que les variables de séquences sont existentiellement quantifiées. Cela permet d'établir les *lemmes d'instanciation* (Lemmes 4.6 et 4.7) qui énoncent que rajouter une variable symbolique fraîche dans un état abstrait en lui adjoignant une définition ne modifie pas la concrétisation de cet état abstrait.

Opérateurs abstraits du produit réduit

Les Sections 4.3 et 4.4 présentent les opérateurs abstraits utilisés par l'analyse. La première famille d'opérateur sont les opérateurs de dépliage qui précisent l'état abstrait en remplaçant un état prédicat inductif par une disjonction d'états abstraits plus précis. L'opérateur de *dépliage avant* $\mathbf{unfold}_{\mathbb{S}}^\sharp$ (Définition 4.10) remplace un prédicat inductif par ses possibles définitions. Le *dépliage arrière*,

b-unfold$^\sharp_{\mathbb{S}}$, présenté dans la Figure 4.8 matérialise le dernier élément d'un segment inductif. Enfin, le *dépliage non-local* **nl-unfold**$^\sharp_{\mathbb{S}}$ (Définitions 4.12 et 4.13) coupe un prédicat inductif en deux quand il est certain qu'une cellule est présente dans ce prédicat inductif. Avec les opérateurs de pliage, on peut définir tous les opérateurs abstraits à l'exception des opérateurs de treillis.

Les opérateurs de treillis, définis dans la Section 4.4, reposent sur le même principe à trois étapes :

1. L'*étape de forme* calcule le résultat attendu entre les parties de forme des états abstraits. Cette étape fonctionne en affaiblissant les états abstrait en pliant plusieurs éléments de l'état abstrait en un prédicat inductif. Le pliage génère des contraintes de séquence qui doivent être valides pour que le pliage soit correcte. Les règle de calcul du test d'inclusion sont présentés dans la Figure 4.14 et celles de l'union et de l'élargissement sont données dans la Figure 4.19.

2. L'*étape d'instanciation* utilise les contraintes accumulées dans l'étape de forme pour enrichir les parties de séquence des états abstraits. Cela se fait grâce à l'opérateur **instantiate**$^\sharp_{\mathbb{S}}$, présenté dans l'Algorithme 1, et la correction de cette partie repose sur le lemme d'instanciation de la Section 4.2.

3. L'*étape de séquence* calcule le résultat entre les parties de séquence instanciées en utilisant l'opérateur correspondant du domaine de séquence.

Pour illustrer l'analyse, la Section 4.5 présente en détail l'analyse de l'insertion dans un arbre binaire de recherche.

Implémentation et Évaluation

La Section 4.6 présente l'implémentation de l'analyse dans l'analyseur statique MemCAD. Cette section discute aussi les performances ainsi que les résultats obtenus par l'analyse comme la preuve de la correction fonctionnelle partielle de plusieurs algorithmes de tri et la preuve de plusieurs bibliothèques de listes tirées de système d'exploitation. Les preuves de ces programmes, en particulier les algorithmes de tri, demandent que l'analyse infère des invariants de forme et de séquences précis En particulier cette section présente l'importance de bien choisir la spécification de structures de données ainsi que les heuristiques et directives utilisées pour guider l'analyse.

Pour conclure la présentation de l'analyse, la Section 4.7 discute les autres travaux en lien avec l'analyse de programme manipulant différents types de conteneurs, tels que les tableaux, les chaînes des caractères ou les structures de données dynamiques.

Analyse de FreeRTOS

Le Chapitre 5 présente le travail sur la vérification d'une instance de l'ordonnateur de tâche de FreeRTOS. La Section 5.1 présente FreeRTOS ainsi que le fonctionnement général de son ordonnanceur. En particulier, les différents états et priorités dans lesquelles peuvent être les tâches. Cette section présente aussi les spécificités de l'instance que nous cherchons à vérifier.

La Section 5.2 présente la méthode employée pour vérifier l'ordonnanceur. Notre modélisation de l'ordonnanceur suit le cycle d'exécution d'une application de FreeRTOS : On commence par vérifier les fonctions d'initialisation, puis une fois l'état de l'ordonnanceur en cours d'exécution atteint tous les autres appels systèmes sont vus comme des boucles sur cet état. Notre vérification de FreeRTOS est agnostique vis-à-vis de l'application : nous ne faisons aucune hypothèse sur le nombre de tâches ou sur leur code. Enfin, chaque point d'entrée de l'ordonnanceur est analysé séparément, mais tous les appels de fonction internes à l'ordonnanceur sont *inlinés* pour garder de la précision.

Spécification de l'ordonnanceur

Les Sections 5.3 et 5.4 présentent respectivement les spécifications des invariants de l'ordonnanceur et les spécifications propres à chaque fonction.

La présentation des invariants de l'ordonnanceur est faite en donnant d'abord les parties de l'ordonnanceur propre à chaque état de tâche puis en combinant le tout en une unique formule H. À chaque état des tâches de l'ordonnanceur correspond une liste qui stocke l'ensemble des tâches dans cet état. En plus des listes, chaque état utilise des variables globales pour stocker des informations importantes sur les tâches, par exemple dans le cas des tâches prêtes à être exécutées, l'ordonnanceur dispose

d'un pointeur qui marque la tâche en cours d'exécution. La formule H qui spécifie l'ordonnanceur est paramétrés par 14 variables. Certaines sont des variables de séquences décrivant les séquences présentes dans les listes, d'autres correspondent aux valeurs de variables globales.

Les fonctions de l'ordonnanceur sont spécifiées comme une conjonction de buts. Chaque but est un triplet de Hoare où la précondition et la postcondition sont de la forme $H(\vec{p}) \wedge \varphi(\vec{p})$. La première partie de la conjonction garantit que les invariants de l'ordonnanceur sont bien maintenus par la fonction. La seconde partie, la formule $\varphi(\vec{p})$ est la partie de la spécification propre au but que l'on cherche à vérifier. Elle contient des contraintes sur les paramètres $\vec{p}$, ainsi que sur les paramètres de la fonction et la valeur renvoyée.

Les Sections 5.4 et 5.4 se terminent par une discussion sur le coût de cette spécification. La spécification des invariants des différents états de l'ordonnanceur nécessitent moins de 200 lignes. La spécification des fonctions, elle, demande moins de 700 lignes de code. Ce cout est inégalement réparti entre les différents buts. Un tiers des buts sont spécifiés en moins de 10 lignes : ce sont les buts simples. Un autre tiers demande entre 10 et 20 lignes. Enfin, le dernier tiers correspond aux buts complexes qui peuvent demander jusqu'à 42 lignes de spécification. Finalement, le rapport entre le nombre de lignes de spécifications et le nombre de lignes de code vérifié est de 1.1.

Vérification de l'ordonnanceur

La Section 5.5 présente le travail effectué pour analyser les fonctions constituant l'analyse de FreeRTOS.

La première partie de cette section concerne les modifications apportées dans le code. Certaines sont de simples directives de l'analyseur pour guider le domaine de partitions. Une autre modification change la condition de sortie d'une boucle. La nouvelle condition est équivalent à la précédente, mais permet de déclencher les heuristiques de réduction du domaine de séquence. La dernière forme de modification est l'ajout de pointeurs fantômes dans le code dans le but de guider les opérateurs de treillis du domaine de forme.

La deuxième partie de la Section 5.5 présente les résultats expérimentaux obtenus. Ceux-ci sont synthétisés dans la Table 5.3. Tous les buts spécifiés sont prouvés à l'exception de deux, appartenant à la même fonction. Nous présentons ce qu'il faut ajouter à l'analyse pour analyser et prouver ces deux buts manquants

La dernière partie, présente la discussion sur les performances de l'analyse et tout particulièrement le temps nécessaire pour vérifier les fonctions de l'ordonnanceur. Le premier facteur est le coût du domaine de disjonction. En effet, le nombre de disjonctions peut atteindre jusqu'à 30. Le deuxième facteur est le coût du domaine numérique utilisé par notre analyse. Le domaine des inégalités linéaires, ou domaine des polyèdres, a un cout non borné dans le pire cas et exponentielle dans le nombre de variables *en pratique*. Enfin, le dernier facteur expliquant les performances de l'analyse est le manque d'optimisation dans la phase d'instanciation des opérateurs de treillis du produit réduit. Cela est illustré dans la Figure 5.9.

Discussion

La Section 5.6 présente les enseignements tirés travail de vérification. Le premier concerne le choix de la spécification. En effet, même si deux spécifications logiques sont logiquement équivalentes, il est possible qu l'une d'entre elles soit préférable pour l'analyse. On peut citer, entre autres, le fait de décomposer autant que possible les spécifications en buts.

Le deuxième enseignement porte sur le travail de débogage de l'analyse. Quand une analyse échoue à prouver un but, il est toujours possible d'inspecter les états abstraits calculés pour détecter l'instant où l'analyse perd en précision.

Le troisième enseignement est lié aux modifications apportées à l'analyse au cours de la vérification de FreeRTOS. Seulement deux modifications furent nécessaires pour prouver les fonctions de FreeRTOS : une dans le domaine de forme et une dans le domaine de séquence. Dans les deux cas, la modification ne représente que quelques lignes de code.

Le dernier enseignement concerne le coût de la vérification (9 mois de travail) ainsi que la répartition : un quart de ce coût correspond à l'écriture de la spécification ainsi qu'aux modifications apportées par la suite. L'amélioration des performances et de la précision de l'analyse représente 15 %. Les 60 % restant correspondent au travail d'inspection des journaux de l'analyse pour détecter les pertes de précision de l'analyse.

La Section 5.7 discute le coût de la vérification des différentes fonctionnalités de l'ordonnanceur de FreeRTOS qui ne sont pas analysées. Certaines comme la gestion de l'état de tâche suspendu peuvent être vérifiées sans nécessiter d'importantes modifications à l'analyse, tandis que d'autres nécessitent d'améliorer l'expressivité du domaine de séquence pour exprimer, par exemple, la construction par compréhension de séquences. Enfin certaines extensions, comme les listes d'événements, demandent de représenter des structures de données qui ne sont pas compatibles avec le domaine de forme basé sur la logique de séparation.

Enfin la Section 5.8 décrit les travaux de spécification et de vérification de FreeRTOS, et en particulier de son ordonnanceur, entrepris jusqu'à présent.

Conclusion

Le Chapitre 6 conclut ce manuscrit.

Dans cette thèse, nous avons examiné la vérification d'un composant critique d'un système d'exploitation, à savoir l'ordonnanceur de tâches, en utilisant une approche automatique et rigoureuse : l'analyse statique basée sur l'interprétation abstraite. Par vérification, nous entendons non seulement prouver que le programme est exempt d'erreurs d'exécution, mais aussi garantir la préservation des invariants de l'ordonnanceur et la correction fonctionnelle partielle de ses fonctions constitutives.

Pour atteindre cet objectif, nous avons conçu une analyse capable d'exprimer des contraintes complexes sur le contenu des structures de données inductives. La première étape a consisté à développer un domaine abstrait permettant de raisonner sur des séquences de valeurs. Ce domaine exprime des contraintes relationnelles sur des expressions construites avec des variables symboliques de séquence et numériques, composées par concaténation ou tri. Il s'appuie également sur deux domaines auxiliaires : un domaine numérique pour les contraintes sur les éléments extrêmes des séquences et leur longueur et un domaine de multi-ensemble pour raisonner sur le contenu des séquences indépendamment de la position des éléments.

Ensuite, nous avons étendu une analyse de forme basée sur la logique de séparation en combinant cette analyse avec le domaine des séquences. Cette extension introduit un nouveau type de paramètre dans les prédicats inductifs, un paramètre de séquence décrivant le contenu des structures de données résumées par ces prédicats. Nous avons proposé une classification de ces paramètres pour détecter ceux utilisables dans des prédicats de segment et pour inférer automatiquement des propriétés sur ces paramètres. Les fonctions de transfert abstrait du domaine de forme ont été adaptées pour tenir compte de ces paramètres de séquence.

Nous avons mis en œuvre cette analyse, démontrant qu'elle est suffisamment expressive pour prouver la correction fonctionnelle partielle de programmes complexes, comme des algorithmes de tri implémentés avec des listes ou des arbres binaires de recherche. Elle a également analysé avec succès des bibliothèques de listes issues d'applications réelles.

Enfin, nous avons appliqué cette analyse à une instance industrielle de système d'exploitation, FreeRTOS. Nous avons spécifié les états de l'ordonnanceur et les pré et post-conditions des fonctions. Comme notre analyse ne nécessite pas d'invariants de boucle, l'effort de spécification a été modeste. Nous avons ensuite vérifié l'instance, nécessitant quelques modifications du code source pour orienter l'analyse. Finalement, nous avons vérifié la correction fonctionnelle partielle de toutes les fonctions de l'ordonnanceur, sauf une.

Nous reconnaissons que nos résultats sont influencés par notre connaissance du fonctionnement de l'analyse. Par exemple, les modifications du code ne pourraient pas être reproduites par un utilisateur inexpérimenté. De plus, notre analyse présente des performances limitées en termes de temps d'exécution, dues à la complexité des contraintes numériques à exprimer. Néanmoins, ces résultats sont encourageants, car la majeure partie du raisonnement, notamment celui sur le contenu des structures de données, est effectuée automatiquement par notre analyse.

La conclusion se termine par une liste d'extension de ce travail. Certaines sont des améliorations directes de notre analyse tandis que d'autres permettraient de vérifier d'autres propriétés comme la terminaison ou les problèmes introduits par une gestion concurrente de la mémoire.

Remerciements

En premier lieu, je tiens à remercier Arthur Charguéraud et Jean-Christophe Filliâtre d'avoir accepté de rapporter mon manuscrit de thèse. Je suis aussi reconnaissant vis-à-vis de Julia Lawall, Matthieu Lemerre et Marc Pouzet d'avoir accepté d'être dans mon jury de thèse.

Merci beaucoup Xavier, d'avoir été mon superviseur de stage de master puis mon directeur de thèse. Tes conseils, tes remarques ainsi que tes idées de corrections m'ont été très précieux tout au long de mon travail de recherche et pour mener à bien mon travail de rédaction.

Merci Damien et Timothy pour vos précieux conseils lors du comité de suivi.

J'ai eu la chance de faire ma thèse dans un environnement fabuleux : l'équipe AnTique. Merci à tous ses membres permanents, Bernadette, Jérôme F., Caterina & Vincent pour toutes ces discussions, scientifiques ou non. Ces dernières furent sans aucun doute les plus utiles pour moi.

Je tiens aussi à remercier chaleureusement, les autres membres de l'équipe : Adam, Albin, Alessandro, Antoine, Aurélie, Denis, Guru, Ignacio, Jérôme B., Kevin, Louis, Luca, Marc, Marco, Matthieu, Naïm, Olivier, Patrizio, Serge, Stan, Valentin. Merci pour toutes nos discussions, bavardages, repas (au pôt ou dans la rue Mouffetard), et bières bues ensemble.

J'aimerai particulièrement remercier les deux stagiaires avec qui j'ai eu la chance de travailler durant cette thèse : Charles & Félix.

Je suis aussi reconnaissant vis-à-vis des membres de l'équipe APR : Antoine, d'avoir accepté que je prenne en charge les TPs du cours d'analyse statique, ainsi que ses doctorants, Francesco, Mamy, Marco et Milla.

Merci Thierry pour ton aide dans l'implémentation de MemCAD et du langage de spécification.

Merci aussi à tous ceux avec qui j'ai fait des soirées jeux, des randos, regarder des films et autres activités et qui m'ont sorti la tête de ma thèse. En particulier, merci beaucoup à tous mes nombreux colocs durant ces quatre dernières années : Mathilde, Charles, Louis, Robin L., Théophile de la *Coloc Jaurès* et Aron, Robin N.-D. et Victoire de la *Coloc Verdi*.

Merci à mes parents et à mes frères pour tout leur soutien durant ma scolarité et ma thèse. Sans vous, je ne serai pas là aujourd'hui.

Enfin, merci à tous ceux que j'ai oubliés et qui ne m'en tiennent pas rigueur.

Contents

1 Chapter Introduction

1.1 Operating systems are ubiquitous

1.1.1 What is an operating system?

The simplest definition one can find of an *operating system* (OS) is "the programs that make other programs run on a computer". These others programs being called *user applications*. However, this definition is sometimes too broad to distinguish, on a computer, what is part of the operating system, what is not. For example: is the dynamic linker part of the operating system? One cannot deny that

the goal of the dynamic linker is also to make programs run on a computer, though it is not part of the operating system.

The second more advanced definition concerns the task handled by the operating system:

- **OS provides a hardware abstraction for applications.** Without the OS, each programmer would be required to write their user applications to accommodate the specific hardware it is run on. Such a way to run an application is called *bare-metal*. This also means that the programmer must take into account all the low-level difficulties such as hardware interrupts. In contrast, the OS hides the low-level complexity of the hardware. It also provides a common interface for actions such as keyboard input regardless of the hardware involved in this action. As a consequence, operating systems allow programs to be more portable and simpler.

- **OS manages the resources for the applications.** A computer provides some resources for applications to use. The first of these being the use of the processor (CPU time). All the user applications request (part of) these resources to perform their own tasks. They often do so in a conflicting manner, for example when several applications try to read or write some data from the drive. This is the goal of the operating systems to manage all these requests and to serve them in an efficient and fair manner. This allows the programmer to design their user applications without worrying about other applications that may also be running on the same computer.

It is important to note at this point that the definition provided earlier does not take into account the level of privilege used by the operating system. Among all the features provided by an operating system, the core ones such as memory management and applications scheduling form what is called the *kernel*. The kernel part of the operating system often run in privileged mode compared to other *user-level programs*. Regarding the separation between the kernel and the rest of the operating system, there exist two main approaches. The first one consists in putting all hardware managing component inside the kernel. This approach, called **monolithic kernel** is followed by all consumer kernels such as Windows NT, the Linux kernel and the FreeBSD kernel. The second approach restricts the set of actions operated with high privilege level to the bare minimum to form a **microkernel**. In a micro-kernel, device drivers are not part of the kernel. They run as an unprivileged user-mode application. This distinction is often introduced to ensure security features such as prohibiting privilege escalation, but it does not change the set of features and actions performed by an operating system.

1.1.2 The critical aspect of operating systems

Because of the ease they give to programmers, operating systems are everywhere. Excepting a limited family of application such as embedded systems, every computer runs an operating system, from a tiny microcontroller with a single core and a few kilobytes of memory, to the supercomputer node using hundreds of gigabytes of memory with several multicore processors. But in order to fulfill these two goals, an operating system must accommodate a large variety of hardware component (CPU architectures, device drivers) and also a similarly large variety of applications. This has resulted in an increase in the size and complexity of operating systems. Therefore, the development and maintenance of OS become increasingly difficult.

*
* *

As a consequence of both the nature of the mission they fulfill and the ubiquitousness of their use, operating systems are critical components of modern computers. Therefore, it is crucial to ensure that an OS does not crash and behaves as expected. Indeed, a software crash at the kernel level results in a crash of the whole computer. For example, during a test on USS Yorktown of the smart ship program in 1996, a human error that was not detected by the Windows NT kernel led to the crash of all systems. The ship remained "dead in the water" for two hours and fourty-five minutes [Stu98]. In the case of an unexpected behavior, some functionalities of the OS, such as the possibility to use a device, could become inoperable. Moreover, the curse of operating systems is that the tools available to diagnostic an error (*e.g.* debugger) are limited compared to the size of the operating systems, and practically unusable in case of a crash. Furthermore, it is also difficult to replicate an error.

1.2 Possible goals to improve OS quality

In this section, we describe possible objectives to improve the quality of operating systems. These goals are, in fact, properties that we want to see verified by operating systems. Here we focus on semantic properties. That is to say, properties about the possible executions of the operating system. In particular, we do not consider syntactic properties such as "all defined variables must have distinct names". These properties are not unique to operating systems but are defined for any program. Nevertheless, we illustrate their importance in the context of operating systems.

1.2.1 Absence of run-time error

The first property we would like to check is the absence of run-time errors (ARTE). It ensures that the program does not perform any operations that could result in a "crash". Indeed, when a crash happens at the kernel level, it is very difficult if not impossible to catch it and handle it safely. In the `C` programming language, which is the most popular to write operating systems, the notion of run-time error is not defined as such. In the standardization of `C`, the authors uses the term *undefined behavior*, that is defined as [2218]:

> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.

Undefined behaviors range from division by zero, to signed integer overflow, and dereferencing a pointer to an object whose lifetime has ended.

In the Rust programming language, the authors distinguish two causes of run-time errors. The first one is called panic. When a panic occurs, the program is terminated abruptly, and a panic message is displayed along with relevant information about the error. Panicking is a mechanism designed to handle situations where the program encounters a state it cannot recover from. Example of such situations are division by zero, overflow during integer conversions, or performing an out of bound index access in a vector. In the case of panics the behavior is well-documented. This is not the case, for *undefined behavior*. These are similar to undefined behavior in the `C` programming language: in case of an undefined behavior, the programmer has no guarantee to what the program may do next. An example of undefined behavior in Rust is dereferencing a null or dangling pointer.

Among the undefined behaviors in the C standard, those related to improper memory manipulation are particularly dangerous. They can indeed be used as an attack vector to compromise the security of the system (see below). For example, Microsoft reported in 2019 that 70 % of the security bugs patched by Microsoft are related to memory mismanagement [Mil19]. Moreover, according to the MITRE corporation, 46 % of the known exploited vulnerabilities in 2023 correspond to memory safety issues [Cor23].

1.2.2 Preservation of invariants

The preservation of invariants property states that the current state of the program must satisfy a set of constraints. This set of constraints is called the *invariant of the system*. That is to say a property that must be preserved by any computing step of the program. This property generally allows the possibility for this invariant to be momentarily broken. In the case of an operating system, these properties range from numerical assertions such as "The value of this variable must always be in a given set", to invariants related to the memory layout, *e.g.* "this structure must always be a well-formed circular doubly-linked list whose node must be correctly initialized before insertion". Since the insertion into a doubly linked list requires assigning two pointers in the list, the structures is not maintained between these two assignments. But when once the insertion is performed, the list must again be properly formed.

When the operating system detects that some invariant does not hold in the current state, and that the operating system cannot recover from this state, the operating system panics. In the case of a panic, the operating system displays information to understand the cause of the panic and either enters an infinite loop or tries to reboot the computer. In cases where the system does not detect that the invariant is violated, certain components of the system, which operate assuming that the invariant is always upheld, may no longer function properly.

1.2.3 Partial functional correctness

Informally speaking, partial correctness says that the program, when it terminates and does not crash, indeed performs the expected operations. It means that the program is a correct implementation of some mathematical specification. Such specification are often expressed using Hoare triple [Hoa69] $\{H(\texttt{x},\texttt{y})\}\,\texttt{r = f(x,y)}\,\{P(\texttt{x},\texttt{y},\texttt{r})\}$.

One program may have several triples depending on the possible case disjunctions. For example, the partial functional correctness of the function removing the first element of a list can be expressed using two Hoare triples:

1. The first one states that if the list is empty, then the returned value is the null pointer, and the list is left untouched.

2. The second one state that if the list is non-empty, then the returned value is a pointer to the first element of the list when the function was called, and this element is removed from the list by side effect.

1.2.4 Liveness properties

One way to define *liveness* properties is to say that they are properties that cannot be refuted by looking at a partial execution of the program. Compared to ARTE and invariants preservation, liveness properties are generally harder to verify. Indeed, one can observe, in finite time, when the properties mentioned earlier are breached. So proving liveness properties requires looking at all possible infinite executions, to check if one of them breaches the property. A classical example of liveness property is termination. Liveness properties also correspond to properties expressing absence of starvation such as "a request will be served at some point".

1.2.5 Concurrency & Asynchronism related issues

Operating systems are not comprised of a single program but several that run simultaneously on separate processors or in an interleaved manner. Since these processes running in parallel on other CPUs or interrupting a process often have in common a shared state, such as device buffers, they can perform certain operations that modify the state of other processes. When the order of execution and the interleaving of processes change the possible executions of the operating systems, it is referred to as a *race condition*. The property stating that a program can be safely executed by multiple threads concurrently without race conditions is called *thread safety*. Race conditions can be challenging to detect and reproduce because they depend on specific timing conditions. They are a common source of bugs in operating systems.

To mitigate these race conditions, the programmer may either disable hardware interruptions temporarily to ensure that the process is run uninterrupted. But this only works for single core architectures. The more general solution is to use *mutual exclusion* mechanism (MutEx). It is a synchronization mechanism used to ensure that only one process at a time can execute a specific section of code. One example of MutEx is the use of lock [MCS91]. Each process that tries to perform an operation on a shared data structure must acquire the lock before doing so and should release it after. This prevents other processes from performing operations on this shared data structure. When a process tries to acquire a lock already owned by another process it blocks until this lock is released. But if the process possessing the lock is also waiting to acquire another lock possessed by the first process, then the whole system is stuck in a *deadlock*. Therefore, the synchronization mechanism must be designed to ensure that such a deadlock never arises.

Moreover, as processes compete to acquire some resources, the system may serve some processes at the expense of others. But it is crucial to ensure that each process can ultimately utilize the resources it requires. If a process cannot acquire a necessary resource, it starves and cannot perform the expected task. The *fairness* property ensures that each entity of the system will have its fair access to shared resources.

1.2.6 Security

The term of security encapsulates a large family of properties. This family of properties characterizes the ability of the program to prevent, resist to, or mitigate attacks from a malicious threat. This is called *confidentiality*. For example, this means that the attacker cannot learn any information about

the value of a confidential value by only looking at public information (this is called *noninterference*). Security properties also include the ability to maintain integrity of the system against unauthorized modification or tampering and to ensure that the system remains available despite the attacks.

But what constitutes a malicious threat, and what are the possible operations for the attacker is generally hard to define. Indeed, the attacker may be leveraging a large variety of public information such as time required by some computation [KHF+19, LSG+18] or the energy consumption [GN23].

1.3 Approaches followed to improve reliability of operating systems

This section gives a sketch of approaches followed to gain some confidence in the behavior of operating systems. It focuses on formal methods, *i.e.* methods that require reasoning on the mathematical formalization of the behaviors of the program. This excludes software engineering methods such as coding guideline [dev23], and use of version control software [Con05], though they play an essential part in the development of operating systems.

1.3.1 An impossibility theorem

The ideal way to ensure that a program operates as expected would be to create an algorithm that takes as input the program and its expected behavior and returns **true** if the program respect the expected behavior and **false** otherwise. However, we know since Rice's theorem [Ric53] that such algorithm cannot exist. This theorem states that "all non-trivial semantics properties of a Turing complete language are undecidable." All the above-mentioned properties are instances of non-trivial properties.

As a consequence of this impossibility theorem, we know that there cannot exist verification methods that satisfy the three following properties:

- **Completeness** A verification method is complete if it accepts any correct programs. This means that there cannot be any false alarms: if a program is rejected by the analysis, then we know for sure that it does not satisfy the required property.
- **Soundness** An approach is sound if all programs that do not satisfy the property are rejected by the analysis. That is to say, the analysis does not have any false negative. For example, if it proves that the program is free of run-time errors, then indeed it is the case.
- **Automation** Automation refers to approaches that do not require user intervention—except for specifying the expected behavior—and that always terminate.

Therefore, any verification method must drop, or weaken at least one of these properties.

One could argue that a real-world computer has a finite memory and therefore, the hypotheses of Rice's theorem are not satisfied. But in that case, the theoretical limit would become a practical one since the cardinality of the set of states is exponential in the size of the memory. As a consequence, it is not feasible to write a verification method that enumerates all the possible states.

1.3.2 Testing

Testing a part of an operating system and checking if it behaves as expected is the main way to detect bugs in modern development frameworks. To do so, programmers write alongside their code small programs that call parts of their code on specific input and checks that the returned value corresponds to the intended one.

Software tests can be categorized based on different levels of granularity. The smallest level of granularity concerns *unit testing*. At this level, components are tested independently, each function at a time. To test the interaction of several components together, we use *integration testing*. Finally, to test the whole program at once, we use *end-to-end testing*. It is important to note that the higher the scale, the more difficult it is to express the expected behavior using assertions checking. As all software components, operating systems use these tests in order to detect bugs [LLC19, Dev16].

For each test, the programmer must specify the specific input and the assertions to check in the output. This is very time-consuming. Of course, this can be accelerated with fuzzing: the inputs are automatically generated either totally randomly or by modifying already existing inputs. But in this

case, it is not possible to express what is the expected final state. Therefore, these methods are used to check absence of run-time errors and preservation of invariants. This problem is acknowledged, for all large and complex pieces of software, by some industrial actors such as Airbus [DS07]:

> Considering the steady increase of the size and complexity of this kind of software, classical validation and verification processes, based on massive testing campaigns and complementary intellectual analyses, hardly scale up within reasonable costs.

Furthermore, the main issue with tests is known since the late 70s. As Dijkstra stated [Dij70] "Testing shows the presence, not the absence of bugs." Indeed, if a test fails, then we are sure that the program does not satisfy the expected property. Therefore, testing is both complete and automatic, and as a consequence of Rice's theorem, it cannot be sound, unless the space of states is small enough for exhaustive testing. This means that, in the general case, testing may accept erroneous programs.

1.3.3 Methods on non Turing-complete languages

One way to circumvent the undecidability result is to limit the set of programs to be verified. In fact, if we assume that the programs do not contain loops, then we can design complete, sound and automatic approaches such as bounded model checking and symbolic execution. These approaches have been applied to OS verification. For example, Serval [NBG+19] uses symbolic evaluation to precisely translate the behavior of a program into a logic formula and thanks to SMT solvers, Serval checks that the produced formula ensures some property. The property verified by Serval ranges from absence of run-time errors to noninterference. Furthermore, by analyzing two programs, written in different languages, Serval can be applied to establish the equivalence of these programs and prove the correction of compilations. Bounded model checking [KT14] has also been leveraged to prove the memory safety of network utilities in FreeRTOS [Cho20].

This limit imposed is too strong to be applied to real world OS. Indeed, real world operating systems encounter a wide variety of unbounded behaviors (number of users, number of processes). Here again, circumventing these limitations by taking the physical limits of the system as the bounds of the exploration (e.g. the size of the memory for the number of processes) would result in an impractical verification approach.

1.3.4 Choosing programming languages

1.3.4.1 Using generic programming languages that enforce some properties

Due to recent developments in programming languages design, several general purposes languages have emerged, that guarantee some properties on programs written thanks to them. A famous example is the Rust [KNC15] programming language. The Rust compiler ensures memory safety by checking that if there exists several references to an object, then none of them is allowed to change values stored in it, and by computing at compile time the lifetime of each object, in order to know when it needs to be freed. If the program is compiled successfully, then the code is considered *safe* Rust, and "No matter what, safe Rust can't cause undefined behavior"[Tea15].

But safe Rust is too restrictive to write operating systems. Indeed, this requires to manipulate raw pointers, and to do low-level memory management. To tackle this issue, the Rust compiler, uses a keyword, `unsafe`, to encapsulate code that is not verified by the compiler. And all operating systems developed in Rust [BA20, Dev15, LAC+15] use unsafe Rust. It is up to the programmer to ensure that memory safety still holds. But, this not always the case, and Rudra [BKA+21], an analyzer that detect potential bugs in unsafe Rust, was able to leverage a memory safety bug to perform arbitrary read/write operations in private memories. Therefore, the authors of Rudra conclude that:

> It is not (yet) practical to build a security mechanism solely based on Rust 's safety guarantee".

1.3.4.2 Using Domain Specific Languages

Another approach used to ensure that some operating system components verify some property is to use a domain specific language (DSL) [GSL+17]. Unlike general-purpose programming languages, DSLs are tailored to address the needs and challenges of a specific application domain. The component

is written in a higher specification language that is then compiled into a lower level language, and finally inserted into the code source. The component is often expressed in a concise and efficient manner that hides some low-level difficulties to the programmer such as pointer arithmetic. Therefore, the obtained code enjoys, by construction, some properties such as memory-safety and invariants preservation.

Another advantage of DSLs, is that they may impose some restrictions to the operations allowed. For example, COGENT [AHC+16], a DSL designed for the conception of file systems, leverages a linear type systems to ensure that each value is used exactly once (excepted read-only values). A second advantage of this approach is that it allows reasoning on a higher level language in order to prove new properties. For example, COGENT outputs a high-level COQ specification of the program, and Lepers *et al.* [LGC+20] proved work-conservation on a scheduler designed with the IPANEMA DSL.

1.3.5 Deductive methods

1.3.5.1 Using a proof assistant

A possible approach to verify that Operating Systems behave as expected is to construct a manual proof that can be later automatically checked by a computer using a proof assistant. These methods are both sound and complete but not automatic. Deductive methods have been successfully applied on operating systems. The most noticeable example is the seL4 microkernel[KEH+09]. It is designed in three layers, from the most abstract one, written in Isabelle/HOL [NWP02], at the top, to its actual implementation in `C` at the bottom. The middle layer is composed of an executable specification written in Haskell. Each layer is a refinement from the one above itself: any behavior in a layer must correspond to a behavior in the upper layer. This ensures that each property verified in the upper layers also hold in the lower ones. The proof of seL4 states that its implementation is free of run-time errors, memory violation and that it matches the high-level specification. This approach has been also successfully implemented in the Coq [Tea23c] proof assistant to design CertikOS [GSC+16a]. CertikOS is a framework that uses up to 8 refinement layers to develop the mCertiKOS hypervisor [Tea23b] and the concurrent operating system mC2 [Tea23a].

But this approach is very expensive in terms of development effort: while the implementation of seL4 requires 8700 lines of `C`, the specification of all layers and the refinement proofs take 200,000 lines of Isabelle/HOL. The overall effort estimated to develop seL4 is estimated at 20 person/year. Another drawback of this approach is that it often require the code to be written alongside its proof as stated by the authors of seL4:

> “ The requirements of verification force the designers to think of the simplest and cleanest way of achieving their goals.

Finally, we may observe that software evolution requires to change the proof as well. For example, updating the access right system for the "Call" system call, changed only 30 lines of `C`, but the changes in the proof took more than 8000 lines of Isabelle [Pér18].

1.3.5.2 Using a dedicated prover

The process of writing the code's proof can be partially automated. Given an implementation, a verification condition generator automatically derives a formula expressing the absence of run-time errors, and the (partial) functional correctness of the code. This formula is later given to theorem provers such as Vampire [RV99] or SMT solvers such as Z3 [DMB08] as well as tools from the proof assistant (e.g. sledgehammer from Isabelle). If the solver validates the formula then we are sure that the program is free of bugs and behaves as expected. And if the prover provides a counterexample, then the verification condition generator can derive a possible execution of the program that violate the specification. However, in some cases, the solver timeouts: we do not have a specific answer. When this happens, the developer may guide the solver by adding other information such as invariants at some point in the program or some lemmas. These invariants and lemmas are proved using the solver (or they may be assumed if the user asks so), and later used by the prover for other formulas.

This approach has also been used to verify components of operating systems. For example, Blanchard *et al.* [BKL18] proved the functional correctness of the List module of the CONTIKI OS and Mangano *et al.* [MDK16] proved a memory allocation module using FRAMA-C [CCK+]. Moreover, the Hip/Sleek platform [CDNQ07] was used to prove the partial functional correctness of the task scheduler from FreeRTOS [FHQ12]. In both cases, the programmer had to provide intermediate lemmas to help the solvers to reason about inductive data structures. Moreover, using VeriFast [JSP10],

Chong *et al.* [CJ21] were able to prove the thread safety of the interprocess communication mechanism of FreeRTOS. This means that any operations in the queue library is free of data-races even if several tasks may try to add or remove some element in the same queue.

It is also possible to mix techniques to have the best of both worlds. For example, [DDK+15], proves the functional correctness of FreeRTOS's task Scheduler using refinement techniques, similar to seL4. To the original C implementation, three intermediary layers of specifications were added. The two uppermost layers specified in the Z language [Spi89]. And using VCC [CDH+09], parts of the proof of refinement were automatically written.

In addition to the fact that the external provers require loop invariants and additional lemmas, the main drawback of using an external prover is that it may be hard to understand when it fails to prove a formula. This is acknowledged by the authors of VCC [CDH+09]:

> Unfortunately, the ideal situation is encountered only seldomly during the process of verification engineering, where most time is spent debugging failed verification attempts. Due to the undecidability of the underlying problem, these failures can either be caused by a genuine error in either the code or the annotations, or by the inability of the SMT solver to prove or refute a verification condition within available resources like computer memory, time, or verification engineer's patience.

Furthermore, using solvers designed for specific types of tasks may result in poor performance on problems that were not envisaged by the developers [dMP13]. For instance, SMT solvers are known for not being able to perform complex inductive reasoning [FP23].

1.3.6 Automatic static analysis

The remaining family of approaches concern ones that are both sound and automatic. As a consequence of Rice's theorem they are necessarily incomplete. That is to say, they may reject programs that do satisfy the given property. One example of such methods is the static analysis by abstract interpretation [CC77]. Since the behavior of a program cannot be computed exactly, the analysis rely on an abstract domain to compute an approximation of the semantic of the program. An abstract domain provides a finite and efficient representation of the behaviors of the program as well as transfer functions that approximate the operations of the program on the memory states. If the approximation computed by the abstract domain corresponds to an over approximation and if the property we try to establish is subset close, then checking that the over approximation satisfies this property is enough to prove that all possible executions of the program satisfy the interest property.

For example, Nicole *et al.* [NLBR21] proved the absence of run-time errors and privilege escalation in small real-time operating systems ASTERIOS [KS19] and EducRTOS[Lem20]. To do so, they computed an invariant over-approximating the reachable states at any points of the program. As this invariant does not contain the error state, the analysis proved that the operating systems could not crash. Moreover, since not all states were reachable by the executions, the analysis proved the absence of privilege escalation. Furthermore, the Infer tool [CDOY11] has been used to analyze operating systems. Infer is able to prove the absence of run-time errors and thanks to the bi-abduction technique it can generate possible pre- and post-conditions of functions. Infer successfully analyzed and generated contracts for half of the procedures in the Linux kernel. Nevertheless, it should be noted that these generated pre- and post-conditions, though sound by construction, may not be relevant to prove the partial functional correctness of the analyzed functions.

Static analyses have also been used to prove some properties on OS components. For example, the SADA analyzer [OMLB16] verified that some drivers are free of run-time errors and that they respect the hardware specification, expressed as an automaton. This automaton specifies property such as "no data should be exchanged over the bus, until some register is set to some specific value". The MemCAD [CR08] tool was used to prove consistency property of OS components [LR17] relying on dynamic data structures stored in a static container (*e.g.* a linked list in the cells of an array). In the case of a memory allocator the consistency property is "any memory cell must be part of either the list of free cells or the list of allocated cells, and the two lists should be disjoint".

One other example of static analysis is the Coccinelle tool [LMU05]. It helps the developer to solve the problem of collateral evolution, that is to say, propagate API changes to all other components and external device drivers. To do so, the developer writes a *semantic patch*. In essence, a semantic patch is a rewriting rule together with information on the control-flow graph as well as constraints

Methods	Properties					
	ARTE	Invariants	Functional Correctness	Termination	Thread Safety	Security
Non Turing complete languages	[Cho20] [NBG+19]	[NBG+19]				
DSL	[LGC+20, AHC+16]					
Deductive methods (proof assistant)	[KEH+09]				[GSC+16b]	[KEH+09]
Deductive methods (external solver)	[BKL18, MDK16, FHQ12, CJ21]					[DHK21]
Abstract interpretation	[NLBR21, LR17] [OMLB16] [Stu08, CDOY11]		This thesis			[NLBR21]

Table 1.1: Comparison of OS and their components' verification efforts

on expressions to determine where the rule should be used. This semantic patch is then applied automatically to all source files. To know precisely where the semantic patch should be applied, Coccinelle relies on constant propagation, a lightweight form of abstract interpretation. Moreover, by writing specific semantic patches, Coccinelle was successfully used to find bugs in the Linux Kernel, such as buffer overflows [Stu08].

1.3.7 The case for automatic verification of tasks schedulers

To conclude this section, Table 1.1 presents a comparison of the verification works (*i.e.* excluding testing) presented above, by organizing them according to their approaches and the kind of properties established by the work. For example, deductive methods based over a proof assistant can prove high-level properties that require to reason on multiple execution paths simultaneously. In the opposite, abstract interpretation based approaches are limited to state properties (*i.e.* some state is not reachable).

In this thesis, we propose to design a static analysis based on abstract interpretation in order to prove more complex properties than ARTE, such as invariant preservation and partial functional correctness. Though verification of a whole operating system is out of reach of automated verification techniques, analyzing components of operating systems separately is possible. Therefore, we focus our efforts on what constitutes the core component of operating systems: the task scheduler. Even if we remove all device drivers, memory management abstractions, we still need a scheduler to answer the question "What process should we run now?" Furthermore, task scheduler are good candidates for automatic verification since they come in many flavors to handle the different use cases. For example, the CFS, used in the Linux kernel, aims to allocate to each process an execution time according to its priority, whereas in real-time operating systems, time constraints are paramount.

As a target of our analysis we consider an instance of FreeRTOS. It is a small real-time operating system available for up to 40 different architectures. FreeRTOS Is used by industrial actors such as Amazon or Espressif and aims primarily to be deployed on microcontroller. The scheduler of FreeRTOS consists of different levels of priority, and in each level, the tasks are scheduled using a Round-Robin policy. Moreover, tasks can be suspended to wait for an event to happen (*e.g.* a lock release) or to be delayed for some moment. Our instance mainly focuses on the second part, *i.e.* the real-time constraints. We consider an instance with a single level of priority, we aim to prove that a task cannot stay delayed longer than its specified delay.

1.4 Overview of our approach

This section illustrates our objective of task scheduler verification through abstract interpretation. First we present a simple task scheduler that will serve as an example throughout this thesis. Then, we describe the kind of required properties to establish the partial functional correctness of this scheduler.

1.4.1 The weighted fair scheduler

1.4.1.1 General presentation

This scheduler, called WFS for *Weighted Fair Scheduler* is a simplified version of Linux's CFS:

- A task is either running or ready to be run. There is no waiting task.
- A user task may be added or deleted at any time.
- To ensure that there is always a task to be run, an *idle task* is created at the start of the scheduler. This task must never be deleted.
- When added to the scheduler, each task is assigned a weight expressing the *cost* of running this task. The scheduler must ensure that a task with cost 2 is run twice less often than another task with cost 1. To ensure this, the scheduler computes for each task the *weighted service time* (WST), *i.e.* the actual time the task was run multiplied by its cost[1]. The scheduler simply selects, the task with the lowest weighted service time to be run.

As a consequence the scheduler must be able to perform smoothly the following operations. First, the scheduler should be able to pick the task with the lowest WST, *i.e.* a priority queue. This means that the scheduler must store all the tasks in a sorted data structure. Second, it should add new tasks and update WST of tasks after being run for some time. Therefore, the scheduler must be able to insert tasks at any place of the data structure. In the following, we assume that there exists a data structure library which implements these two actions using two functions: `insert` and `select`. Both functions should work by side effect on the data structure given in argument, the `select` function removes the task with lowest WST from the data structure and returns a pointer to it. Consequently, in these functions, the type of the input data structure is a double pointer.

A simplified implementation of the WFS is presented in Listing 1.1. The scheduler manipulates two objects. The first one, is a pointer to the Task Control Block (TCB) corresponding to the current task to be run, the second one is the sorted data structure containing all other tasks. After each unit of time, an interruption happens and the `task_update` function is called. The field of the task keeping track of the weighted service time is incremented according to the weight of the task. Then, the `current_task` is updated by performing one insertion followed by a removal in the `ready_tasks` data structure. To stop itself, a task calls the `task_delete` function. Since a task cannot delete another and the idle task is a simple infinite loop, we may safely assume that when `task_delete` is called, `current_task` never points to the idle task. This function simply deallocates the memory block corresponding to the task. The `current_task` becomes a dangling pointer: so we update it by performing a removal in `ready_tasks`. Finally, the `task_add` inserts the task given in argument in the `ready_tasks` data structure. There is no need to update the `current_task` pointer.

1.4.1.2 Choosing the right data structure

As stated above, the design of WFS imposes to choose a data structure that can easily insert tasks at an arbitrary location in the data structure to preserve its sortedness. This rules out static data structure that stores data in a contiguous memory region, such as arrays and ring-buffers. Indeed, inserting elements into the middle or beginning of such data structures requires shifting all subsequent elements to make space for the new element. So, the scheduler should keep tracks of all WST of tasks in a sorted dynamic data structure. There are, at least, two candidates for such data structures: the sorted doubly linked list and the binary search tree. In the case of the sorted doubly linked list picking the task with the lowest WST can be performed in constant time, whereas inserting a new element or updating the WST of the running task may require to compare it with all other elements in the

[1] For the sake of simplicity we ignore in this presentation behaviors caused by integer overflow.

Listing 1.1: Simplified code of Weighted Fair Scheduler

```
struct TCB {
  unsigned int wst;
  unsigned int weight;
  ...
};
typedef struct TCB  task;
typedef ...         task_container;

task              *current_task;
task_container    *ready_tasks:

void insert(task* new, task_container** container);
task *select(task_container** container);

void task_update() {
  current_task->wst += current_task->weight;
  insert(current_task, &ready_tasks);
  current_task = select(&ready_tasks);
}

void task_delete() {
  free(current_task);
  current_task = select(&ready_tasks);
}

void task_add(task* new){
  insert(new, &ready_tasks)
}
```

list, resulting in a linear complexity in the worst case. In comparison, the binary search tree performs both operations in logarithmic time if it is balanced.

Listing 1.2 shows the implementations of `insert` and `select` in the case where the scheduler uses an imperative binary search tree to store all ready tasks.

1.4.2 Expressiveness needed to prove the partial functional correctness of WFS

We now present the properties that the abstract domain needs to express to prove the absence of run-time error, the preservation of invariants, and the partial functional correctness of WFS.

1.4.2.1 Memory Safety

In WFS, the only possible cause of run-time error is invalid pointer dereference. For example, in Listing 1.2 in functions `select` and `insert`, the `container` argument is dereferenced without prior checking. This means that these functions assume that the pointer is valid, and that the caller always set this argument to a valid pointer. In some cases, pointers are set to the `null` value to signal that they are not valid. This is used in the `SET_PARENT` macro: if the value of `t` pointer is non-null, then the pointer is assumed to be valid. Therefore, the domain must express that some pointers are valid and that they are valid if and only if they are non-null.

Moreover, some statements assign new values to pointers. If the memory location corresponding to the old value is no longer reachable (meaning there are no other pointers to that cell), then it cannot be freed in the future. Therefore, that portion of memory will always be marked as in use, leading to a memory leak. Establishing memory safety requires to prove that all cells that are not freed are still reachable from the `ready_tasks` and `current_task` pointers.

1.4.2.2 Invariants preservation

The `insert` and `select` functions make implicit assumptions about the binary search tree. Examples of such properties are that each node contains a valid pointer to a task control block, and that two distinct nodes cannot point to the same TCB. Moreover, each node has two children pointed by the `left` and `right` fields. These pointers must either be null or point to a valid tree node. In this latter case, the `parent` pointer of the child must point to the former. Since the root of the tree has no

Listing 1.2: Binary Search Tree library for WFS

```
typedef struct node {
  task *content;
  struct node *left;
  struct node *right;
  struct node *parent;
} node;
typedef node        task_container;

#define SET_PARENT(t, p)   if(t) t->parent=p

void insert(task* new, task_container** container){
  node* node = malloc(sizeof(node));
  node->task = new;
  node->left = node->right = null;
  if(*container){ // Non-Empty Case
    struct node* c = *container;
    while(c->content->wst <= new->wst && c->left ||
          c->content->wst >  new->wst && c->right )
      c = c->content->wst <= new->wst ? c->left : c->right;
    node->parent = c;
    if( c->content->wst <= new->wst ){
      c->left = node;
    } else {
      c->right = node;
    }
  } else { // Empty Case
    *container = node->parent = node;
  }
}

task *select(task_container** container) {
  node* c = *container;
  while(c->left) c = c->left;
  if(c == c->parent){ // Root Case
    *container=c->right;
    SET_PARENT(c->right, c->right);
  } else { // Leaf Case
    c->parent->left = c->right;
    SET_PARENT(c->right, c->parent);
  }
  task *task = c->content;
  free(c);
  return task;
}
```

parent, its `parent` field points to itself. The root of the tree is the only node where the parent pointer points to itself. For example, the `select` function uses this property to check whether the node `c` to be removed corresponds to the root of the tree. All these properties are called *shape properties* since they correspond to the memory layout of the binary tree.

In addition to shape properties, the functions also make assumptions in the order of appearance of elements. They are sorted according to the value of the `wst` field of their TCB. Therefore, the minimum value stored in the tree corresponds to the leftmost leaf. To summarize, an example of a well-formed binary search tree is depicted in Figure 1.1. Structures of type `task` are depicted in gray, those of type `task_container` in black. We use a specific color, or style for each field pointer, *e.g.* `parent` pointers are dashed. For simplicity, `null` pointers are denoted by the empty field. Note that we do not require the tree to be balanced. Indeed, this only impacts the time complexity of the program, not its possible behaviors.

The `task_update`, `task_delete`, and `task_add` also assume that some invariants hold when they are called. For instance, the `current_task` pointer must always be a valid pointer. In addition, the TCB pointed by it must not occur in the `ready_tasks` container and its `wst` must be lower or equal than the weighted service times of tasks stored in the `ready_tasks`.

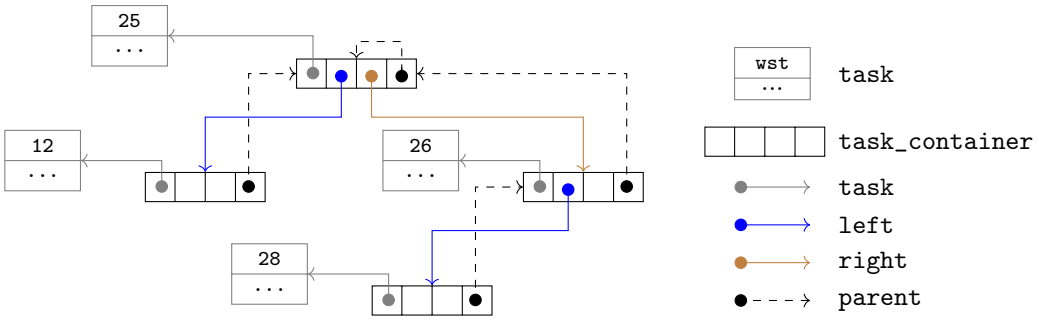


Figure 1.1: Example of a well-formed binary search tree

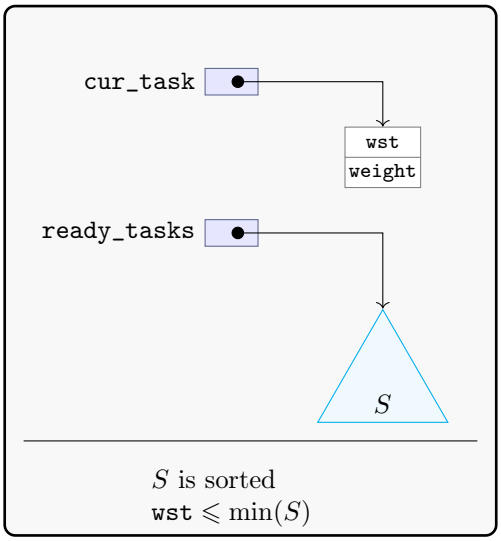


Figure 1.2: Invariant of WFS

1.4.2.3 Partial functional correctness

In the case of `insert`, partial functional correctness states that when the function returns the TCB pointed by the argument is indeed inserted in the tree pointed by `container`, at the right position corresponding to the value of its `wst`, and that all members of the tree when the function was called are still present in the tree.

Regarding `task_update`, the partial functional correctness means that at the end, the value of the `wst` field of the TCB pointed by `current_task` before the function call is correctly updated and that the `current_task` pointer now points to the TCB whose `wst` corresponds to the lowest among those in the tree at function's call and the updated `wst` value.

1.4.3 Abstraction and analysis

1.4.3.1 The Abstract domain

As a consequence, we need to build an abstract domain that is able to express properties about the memory layout such as "pointer p is valid and points to value v". Furthermore, since inductive data structures are unbounded, the abstract domain must be able to summarize them with finitely representable predicates. Nevertheless, this summarization should retain some information on both the memory layout of the binary tree (corresponding to the shape properties mentioned above), and the content stored in the tree such as its sortedness, the presence or absence of element.

For example, Figure 1.2 displays the abstract state corresponding to the invariant of WFS. An abstract state is composed of two parts. The first describes the memory layout. Here, it expresses that the variable `current_task` is a pointer to a valid TCB, and that `ready_tasks` points to a well-formed binary tree. The sequence of the `wst` field of each TCB obtained by an infix tree traversal is expressed with a sequence variable S. The second part expresses constraints over the variables occurring in the

Pre-condition	Post-condition
$H(wst, weight, S)$ $\wedge\, wst + weight \leqslant \min(S)$	$H(wst', weight, S)$ $\wedge\, wst' = wst + weight$
$H(wst, weight, S)$ $\wedge\, S = [wst'].S'$ $\wedge\, wst + weight > wst'$	$H(wst', weight', S_f)$ $\wedge\, S_f = \mathbf{sort}([wst + weight].S')$

Figure 1.3: A pair of Pre- and Post-conditions of `task_update`

first part. For examples, in the invariant, S must be sorted. This constraint, together with the fact that `ready_tasks` points to a well-formed binary tree, states that it is a binary search tree. Moreover, the value of the `wst` field of the TCB pointed must be a lower bound of the elements of S. We observe that the state of the WFS scheduler can be described using only three parameters:

- the weighted service time of the current task,
- the weight of the current task,
- the sequence of the weighted service time of the tasks in the container pointed by `ready_tasks`.

We write $H(wst, weight, S)$ for the abstract state corresponding to the invariant.

1.4.3.2 Analysis

To verify the properties mentioned in the previous section, we provide a set of contracts for each function of the program. A contract is a pair made of a pre- and post-condition expressed as abstract states. For example, Figure 1.3 presents the contracts of `task_update`. All the states involved are written as a call to H with additional numerical and sequence constraints. This ensures that the post-condition satisfies the invariants of WFS. To prove the partial functional correctness of `task_update` we consider two cases separately:

- The first case corresponds to the situation where the current task is still the task with the lowest weighted service time after it has been updated. This is expressed by the constraint $wst{+}weight \leqslant \min(S)$. In this case, the `ready_tasks` container as well as the $weight$ parameter, are unchanged. Only the wst parameter is updated according to the weight of the current task.
- The second one corresponds to the eventuality where the current task is no longer the one with the lowest WST. This means that there exists in the container a task whose weighted service time is lower than the updated wst of the current task. In that case, the current task is set to be the one with the lowest WST in `ready_tasks`. Furthermore, the previous current task (*i.e.* at the moment the `task_update` function is called) has its WST updated and is inserted at the right position in the container. This is done by restricting the final sequence of ready tasks to be the sorted sequence formed by appending the updated WST to the original sequence in `ready_tasks`, excluding its initial element.

Our approach works as follows: starting from the initial pre-condition as initial state, our analysis should compute an over approximation of the possible memory states for each point of the program using forward abstract interpretation. If none of these abstractions contains an erroneous memory state, this means that our analysis proved the absence of run-time errors. Additionally, if the abstract state corresponding to the `return` statement entails the post-condition, then our analysis also proved the partial functional correctness.

1.5 Contributions

The goal of this thesis is to develop a sound and automatic static analysis to prove partial functional correctness of programs manipulating inductive data structures and in particular task schedulers.

We make the following contributions:

- We define an abstract domain that is able to reason about sequences of values. This sequence abstract domain expresses properties such as their bounds, their length, and their sortedness.

- We combine the previously cited sequence abstract domain with another domain that can reason about the shape of inductive data structures. From this combination we derive a static analysis that can prove partial functional correctness of complex programs such as sorting algorithms as well as programs from real-world libraries.
- To demonstrate the feasibility of automatic verification of task schedulers, we apply this analysis on an instance of the FreeRTOS task scheduler. This instance mainly focuses on the real-time constraints. That is to say, in the absence of interruptions, the scheduler ensures that a delayed task does not exceed its waiting time.

1.6 Outline

The remaining of the thesis is structured as follows.

In Chapter 2, we lay the groundwork for this thesis. We present a toy programming language used throughout this thesis, and we define its semantics in Section 2.1. Section 2.2 covers the basics of the abstract interpretation-based static analysis. In Section 2.3, we describe how a subset of separation logic can be used as an abstract domain to summarize memory states containing unbounded inductive data structures. Finally, we outline an analysis based on this abstraction in Section 2.4.

Chapter 3 is dedicated to the presentation of our novel abstract domain to reason over sequences of values. We define the syntax of the constraints manipulated by this language in Section 3.1. The Sections 3.3, 3.4, and 3.5 present the various abstract operators of the sequence abstract domain.

Chapter 4 details the combination of the separation logic based shape analysis presented in Section 2.4 together with the sequence abstract domain defined in Chapter 3. This extension is constructed by adding sequence parameters to the predicates of separation logic. This is the objective of Section 4.1. The sequence parameters also require to extend the various abstract operator used by the analysis. We present these extensions in Sections 4.3 and 4.4. Finally, in Section 4.6 we present and discuss our experimental results.

Chapters 3 and 4 are a detailed presentation of our work published in [GRR23b].

In Chapter 5, we describe the work undertaken to verify a task scheduler taken from an instance of FreeRTOS. Sections 5.3 and 5.4 focus on the specification work to describe the invariants of the schedulers as well as the expected behaviors of the functions provided by the scheduler. In Section 5.5, we present the results obtained by our verifications. We also discuss the modification of the code required to obtain these results, as well as the different identified factors impacting the time spent by the analysis. Finally, in Section 5.6, we discuss various lessons learned during our attempt to verify the task scheduler.

Chapter 6 concludes and proposes future work opened by this thesis.

Chapter

2 A Small imperative language manipulating dynamic data structure and a corresponding shape analysis

The goal of this chapter is to present the foundation of static analysis by abstract interpretation. The first step is to formally define the semantics of the language in which the programs we seek to analyze are written. We introduce a toy imperative language `MemImp` that manipulates inductive data structures using memory pointers. Then, we present how a subset of separation logic can be used as an abstract domain to summarize memory states and a static analysis based on this abstract domain introduced by Chang *et al.* [CR08]. This analysis does not express any constraint on the content of data structures. It is mainly focused at proving absence of run-time error and preservation of shape invariants.

2.1 MemImp

2.1.1 Syntax

The `MemImp` toy language, used throughout this thesis, is a simple imperative language with functions, that primarily aims at directly manipulating memory. Though this language may be seen as a restricted dialect of `C`, it is expressive enough to write complex programs manipulating dynamic data structures such as the weighted fair scheduler presented in the introduction. Its syntax is presented in Figure 2.1. For the sake of simplicity we consider an untyped language that does not make any difference between expressions corresponding to memory addresses and unsigned integer values. We define $\mathbb{V}$ as the collection of all values. Additionally, we denote $\mathbb{A}$ as the subset of values representing all addresses (*i.e.* regardless of their validity). Unlike `C`, `MemImp` do not consider structures with two elements or more as values. As a result, it is unable to copy an entire structure at once or have it returned by a function.

The syntax of `MemImp` expressions is presented in figure 2.1a. An expression is either a constant from the set of values $\mathbb{V}$, the content of the memory stored at an address expressed by a left-value, the address of a left value itself, or the result of an arithmetical operation between two expressions. A left-value is either a variable, or a memory cell at an address expressed by some expression. Their syntax is presented in Figure 2.1b. To make things easier, we assume that `MemImp` programs only manipulate global variables. It is possible to add global variables that account for local variables. For instance, in Listing 1.2 the local variables named `c`in functions `insert` and `select` can be replaced by global variables `c_insert` and `c_select`, respectively. The finite set of all variables manipulated by a program is written $\mathbb{X}$.

Remark 2.1: Structures fields

The syntax for left-value does not explicitly support structure fields offset. It is indeed possible to obtain the same left-value using arithmetic operations. In the following we assume that there is a set of field names $\mathbb{F}$, as well as a function $\varphi_{\mathbb{F}}$ assigning to each name the corresponding offset. We use the notation l`.f` as syntactic sugar to denote the left-value `*(&`$l + \varphi_{\mathbb{F}}$`(f))`. Moreover, we use the arrow notations "e`->f`" that adds to the expression e the value of the offset corresponding to `f` and dereferences the result, *i.e.* `*(`$e + \varphi_{\mathbb{F}}$`(f))`. For example, since one structure field correspond to an offset equal to one, the left value `c->left` from Listing 1.2 corresponds to `*(c + 1)`.

In addition to simple assignments, `MemImp` features function calls as basic statements. In the context of `MemImp`, we consider that only functions are present, and there are no procedures. To avoid reasoning about the possible interleaving of evaluations of functions as well as the left-hand side of the assignment, in a function call, we enforce that the assigned left-value must be a program variable. Moreover, to support dynamic memory allocation, `MemImp` features the `malloc` function, the parameter of which must be a statically known value. Additionally, `MemImp` incorporates standard control-flow statements such as the no-operation statement, sequence of statements, conditional statements, and loops. The syntax of `MemImp` statements is depicted in Figure 2.1d.

To keep things simple, boolean expressions, presented in Figure 2.1c, are restricted to comparison of expressions. More complex expressions using boolean operators `&&` and `||` can easily be emulated using conditional statements.

Remark 2.2: Restrincting `malloc` to constant size allocation

Though `malloc` usually supports arbitrary size dynamic allocation, we restrict it to constant size allocation. This syntactic limitation is problematic only when we try to allocate a cell of unknown size at compile time, such as for arrays of variable size. But since we focus on dynamic data structure, we can safely assume that all of these data structures content cells are allocated separately and have a size known by the compiler.

Finally, a program in the `MemImp` language is simply a finite set of function declarations. The declaration of a function consists of a statement followed by a directive that returns an expression

$$
\begin{array}{rlll}
\langle \texttt{expr} \rangle ::= & c & c \in \mathbb{V} & \text{(constant expression)} \\
| & \langle \texttt{l-value} \rangle & & \text{(value stored at a memory address)} \\
| & \texttt{\&}\langle \texttt{l-value} \rangle & & \text{(value of a memory address)} \\
| & \langle \texttt{expr} \rangle \oplus \langle \texttt{expr} \rangle & \oplus \in \{+, -, \times, /, \%\} & \text{(binary operation)}
\end{array}
$$

(a) Syntax of expressions

$$
\begin{array}{rlll}
\langle \texttt{l-value} \rangle ::= & \texttt{x} & \texttt{x} \in \mathbb{X} & \text{(program variable)} \\
| & \texttt{*}\langle \texttt{expr} \rangle & & \text{(pointer dereference)}
\end{array}
$$

(b) Syntax of left-values

$$
\langle \texttt{bexpr} \rangle ::= \langle \texttt{expr} \rangle \bowtie \langle \texttt{expr} \rangle \qquad \bowtie \in \{=; \neq; \leqslant; >\}
$$

(c) Syntax of boolean expressions

$$
\begin{array}{rlll}
\langle \texttt{stmt} \rangle ::= & \texttt{skip} & & \text{(no-op)} \\
| & \langle \texttt{l-value} \rangle = \langle \texttt{expr} \rangle & & \text{(assignment)} \\
| & \texttt{x} = \texttt{f}() & \texttt{x} \in \mathbb{X} & \text{(function call)} \\
| & \texttt{x} = \texttt{malloc}(c) & \texttt{x} \in \mathbb{X}, c \in \mathbb{V} & \text{(dynamic memory allocation)} \\
| & \langle \texttt{stmt} \rangle ; \langle \texttt{stmt} \rangle & & \text{(sequence)} \\
| & \texttt{if}(\langle \texttt{bexpr} \rangle)\{\langle \texttt{stmt} \rangle\}\texttt{else}\{\langle \texttt{stmt} \rangle\} & & \text{(conditional)} \\
| & \texttt{while}(\langle \texttt{bexpr} \rangle)\{\langle \texttt{stmt} \rangle\} & & \text{(loop)}
\end{array}
$$

(d) Syntax of statements

$$
\langle \texttt{prog} \rangle ::= \langle \texttt{f}()\{\langle \texttt{stmt} \rangle ; \texttt{return}\ \langle \texttt{expr} \rangle ; \} \rangle^{+}
$$

(e) Syntax of programs

Figure 2.1: Syntax of the `MemImp` toy language

as the result. Moreover, functions must all have different names and any called function should be well-defined.

2.1.2 Semantics

In this subsection, we introduce the semantics of `MemImp` programs. Our objective is to demonstrate the absence of run-time errors and to verify properties concerning the program state after a function returns. To achieve this, we employ a *denotational semantics*. This approach involves modeling `MemImp` statements as functions that take a set of program states as input and return the set of states reachable after executing the statement.

Since our focus is on executions that terminate, and since we do not attempt to establish termination, we adopt an angelic denotational semantics. This implies that executions that diverge, meaning they neither terminate nor cause any runtime errors, are not considered in the semantics.

2.1.2.1 The memory model

The memory model used to define the semantics of `MemImp` is a simplification of models developed for the semantics of `C` [LABS12]. Though the size of the smallest addressable unit of memory, a *byte*, is often smaller than the size of integer values and memory addresses, we assume here that any memory address stores a complete value. As a consequence, we do not need to make any assumptions regarding the application binary interface such as architecture size or endianness.

Definition 2.1: Memory states

A *memory state* of the program is a tuple (ρ, m) such that

- $\rho : \mathbb{X} \to \mathbb{A}$ is a function, called the *stack*, assigning to each program variable its address,
- $m : \mathbb{A} \rightharpoonup \mathbb{V}$ is a partial function, called the *heap*, mapping each valid address to the value of its content. Its support[1], *i.e.*, the set of addresses that have an image by m, is written

$$
\begin{aligned}
\mathbb{E}[\![\bullet]\!] &: \langle \mathsf{expr} \rangle \times \mathbb{S}_\Omega \to \mathbb{V}_\Omega \\
\mathbb{E}[\![\_]\!](\Omega) &= \Omega \\
\mathbb{E}[\![c]\!](\rho, m) &= c \\
\mathbb{E}[\![\,\&\, l]\!](\rho, m) &= \mathbb{L}[\![l]\!](\rho, m) \\
\mathbb{E}[\![l]\!](\rho, m) &= \begin{cases} m\left(\mathbb{L}[\![l]\!](\rho, m)\right) & \text{if } \begin{cases} \mathbb{L}[\![l]\!](\rho, m) \neq \Omega \\ \mathbb{L}[\![l]\!](\rho, m) \in \mathbf{supp}(m) \end{cases} \\ \Omega & \text{otherwise} \end{cases} \\
\mathbb{E}[\![e \oplus e']\!](\rho, m) &= \begin{cases} \mathbb{E}[\![e]\!](\rho, m) \oplus_{\mathbb{V}} \mathbb{E}[\![e']\!](\rho, m) & \text{if } \begin{cases} \mathbb{E}[\![e]\!](\rho, m) \neq \Omega \\ \mathbb{E}[\![e']\!](\rho, m) \neq \Omega \\ \oplus \in \{/, \%\} \Rightarrow \mathbb{E}[\![e']\!](\rho, m) \neq 0 \end{cases} \\ \Omega & \text{otherwise} \end{cases}
\end{aligned}
$$

(a) Evaluation of arithmetic expressions

$$
\begin{aligned}
\mathbb{L}[\![\bullet]\!] &: \langle \mathsf{l\text{-}value} \rangle \times \mathbb{S}_\Omega \to \mathbb{V}_\Omega \\
\mathbb{L}[\![\_]\!](\Omega) &= \Omega \\
\mathbb{L}[\![\mathtt{x}]\!](\rho, m) &= \rho(\mathtt{x}) \\
\mathbb{L}[\![*e]\!](\rho, m) &= \mathbb{E}[\![e]\!](\rho, m)
\end{aligned}
$$

(b) Evaluation of left-values

$$
\begin{aligned}
\mathbb{C}[\![\bullet]\!] &: \langle \mathsf{bexpr} \rangle \times \wp(\mathbb{S}_\Omega) \to \wp(\mathbb{S}_\Omega) \\
\mathbb{C}[\![e \bowtie e']\!](\Sigma) &= \left\{ (\rho, m) \in \Sigma \;\middle|\; \begin{array}{l} \mathbb{E}[\![e]\!](\rho, m) \neq \Omega \\ \mathbb{E}[\![e']\!](\rho, m) \neq \Omega \\ \mathbb{E}[\![e]\!](\rho, m) \bowtie_{\mathbb{V}} \mathbb{E}[\![e']\!](\rho, m) \end{array} \right\} \\
&\quad \cup \left\{ \Omega \,\middle|\, \exists s \in \Sigma, \mathbb{E}[\![e]\!](s) = \Omega \vee \mathbb{E}[\![e']\!](s) = \Omega \right\}
\end{aligned}
$$

(c) Semantics of boolean expression

Figure 2.2: Expressions semantics of `MemImp`

$\mathbf{supp}(m)$.

A memory state is *well-formed* if all program variables have a valid address, *i.e.*:

$$\forall \mathtt{x} \in \mathbb{X}, \rho(\mathtt{x}) \in \mathbf{supp}(m)$$

We note $\mathbb{S}$ for the set of all well-formed memory states.

Given a heap m, an address a in its support and a value v, we write $m\,[a \mapsto v]$ for the updated heap with the image of a now being v. The support as well as the images of other addresses remain the same as for the original function. In the case where a is not in the support of m, we denote $m \uplus [a \mapsto v]$ as the heap whose support is $\mathbf{supp}(m) \uplus \{a\}$. The image of a is v and the image of other addresses remains unchanged. When a heap m contains exactly one relation from an address a to a value v, *i.e.* its support is the singleton $\{a\}$, we explicitly write the mapping $\{a \mapsto v\}$.

As one of the aims of our analysis is to prove the absence of run-time error, we explicitly mark states corresponding to an *undefined behavior* as error states. We use the symbol Ω to denote an error. For any set E, we write E_Ω, the set obtained by adding to E the error symbol: $E_\Omega := E \uplus \{\Omega\}$. For example the set of all values, including the error value, is $\mathbb{V}_\Omega$, and the set of all states is $\mathbb{S}_\Omega$.

Finally, we assume that there exist arithmetic operators, written $\oplus_{\mathbb{V}}$ for addition, subtraction, multiplication, division, and modulo, as well as comparison operators, $\bowtie_{\mathbb{V}}$, for equality, disequality and inequalities, and that $\mathbb{V}$ contains the value 0 which does not belong to $\mathbb{A}$.

2.1.2.2 Semantics of expressions

The semantics of an arithmetic expression e, presented in Figure 2.2a, is a function $\mathbb{E}[\![e]\!] : \mathbb{S}_\Omega \to \mathbb{V}_\Omega$, that takes a state as input and returns its value in the state. Note that the evaluation of expression is fully deterministic. The function $\mathbb{E}[\![e]\!]$ may also return an error value Ω if the input state is the error state or if an error occurred during the evaluation of e. This assures that $\mathbb{E}[\![e]\!]$ is Ω-tight: if an error

[1] Mathematically speaking, the term "domain" would be relevant here. However, it is intentionally avoided in order to prevent any confusion with the concrete and abstract domains defined later in this thesis.

$$\mathbf{assign} : \langle \text{l-value} \rangle \times \langle \text{expr} \rangle \times \mathbb{S}_\Omega \to \mathbb{S}_\Omega$$
$$\mathbf{assign}(\_, \_, \Omega) = \Omega$$
$$\mathbf{assign}(l, e, (\rho, m)) = \begin{cases} (\rho, m[\mathbb{L}[\![l]\!](\rho, m) \mapsto \mathbb{E}[\![e]\!](\rho, m)]) & \text{if } \begin{cases} \mathbb{E}[\![e]\!](\rho, m) \neq \Omega \\ \mathbb{L}[\![l]\!](\rho, m) \neq \Omega \\ \mathbb{L}[\![l]\!](\rho, m) \in \mathbf{supp}(m) \end{cases} \\ \Omega & \text{otherwise} \end{cases}$$

(a) Definition of **assign** operator

$$\mathbb{S}[\![\bullet]\!] : \langle \text{stmt} \rangle \times \wp(\mathbb{S}_\Omega) \to \wp(\mathbb{S}_\Omega)$$
$$\mathbb{S}[\![\texttt{skip}]\!](\Sigma) = \Sigma$$
$$\mathbb{S}[\![l = e]\!](\Sigma) = \mathbf{assign}(l, e, \Sigma)$$
$$\mathbb{S}[\![\texttt{x} = \texttt{f}()]\!](\Sigma) = \mathbf{assign}\,(\texttt{x}, e, \mathbb{S}[\![s]\!](\Sigma))$$
$$\text{where } \texttt{f}()\{s; \texttt{return } e;\} \text{ is the declaration of } \texttt{f}$$
$$\mathbb{S}[\![\texttt{x} = \texttt{malloc}(c)]\!](\Sigma) = \left\{ (\rho, m') \;\middle|\; \exists m, a, v_0, \ldots, v_{c-1}, \begin{array}{l} (\rho, m) \in \Sigma \\ m' = m[\rho(\texttt{x}) \mapsto a] \\ \qquad \uplus [a \mapsto v_0, \ldots, a + c - 1 \mapsto v_{c-1}] \\ [a, a + c - 1] \cap \mathbf{supp}(m) = \varnothing \end{array} \right\}$$
$$\cup\, \mathbf{assign}(\texttt{x}, 0, \Sigma)$$
$$\mathbb{S}[\![s; s']\!](\Sigma) = \mathbb{S}[\![s']\!] \circ \mathbb{S}[\![s]\!](\Sigma)$$
$$\mathbb{S}[\![\texttt{if}(b)\{s\}\texttt{else}\{s'\}]\!](\Sigma) = \mathbb{S}[\![s]\!] \circ \mathbb{C}[\![b]\!](\Sigma) \cup \mathbb{S}[\![s']\!] \circ \mathbb{C}[\![\neg b]\!](\Sigma)$$
$$\mathbb{S}[\![\texttt{while}(b)\{s\}]\!](\Sigma) = \mathbb{C}[\![\neg b]\!]\,(\mathbf{lfp}\, F)$$
$$\text{where } F : X \mapsto \Sigma \cup \mathbb{S}[\![s]\!] \circ \mathbb{C}[\![b]\!](X)$$

(b) Semantics of statements

Figure 2.3: Semantics of `MemImp`

is raised during the evaluation of any sub-expressions of e, then the outcome of the evaluation of e is Ω. For expressions, we consider two cases of errors: memory reading at an invalid address, *i.e.* an address that does not belong to $\mathbf{supp}(m)$, or division and modulo by zero.

To evaluate the address of a left-value, l, we rely on an evaluation function $\mathbb{L}[\![l]\!] : \mathbb{S}_\Omega \to \mathbb{V}_\Omega$, presented in Figure 2.2b. This function returns an error value, if either the input state is the error state, or the left-value l is a pointer dereference of some expression e, whose evaluation raises an error.

The semantics of a boolean expression $e \bowtie e'$ is a function $\mathbb{C}[\![e \bowtie e']\!] : \wp(\mathbb{S}_\Omega) \to \wp(\mathbb{S}_\Omega)$ that takes as input a set of states, Σ, and returns the set of memory states in Σ that verify the condition. Moreover, if the evaluation of e or e' in any state in Σ raises an error, then the error state is also in the returned set. This includes the case where the error state was already in Σ. The definition of $\mathbb{C}[\![e \bowtie e']\!]$ is presented in Figure c.

2.1.2.3 Semantics of statements

Given a left-value, l, an expression, e, and a state s, the **assign** operator, returns the state, obtained after assigning to l, the value of the expression e. This operator fails, *i.e.* returns an error state, if the evaluation of either e or l fails, or if the address corresponding to l is not in the heap's support. Otherwise, it updates the memory by assigning to the address corresponding to the evaluation of l, the value corresponding to the evaluation of e. The definition of **assign** is presented in Figure 2.3a. In the following, we automatically lift the **assign** operator to sets of states by considering the image set: $\mathbf{assign}(l, e, \Sigma) := \{\mathbf{assign}(l, e, s) \mid s \in \Sigma\}$.

The semantics of a statement s, defined in Figure 2.3b, is a function $\mathbb{S}[\![s]\!]$ that takes as input a set of states and returns the set of states reached at the end of the statement's execution.

The semantics of the `skip` statement is the identity function. For the assignment statements, the semantics boils down to calling the **assign** operator. In the case of a function call $\texttt{x} = \texttt{f}()$ the body of the function is executed, then the returned value is assigned to `x`.

For memory allocation, $\texttt{x} = \texttt{malloc}(c)$, there are two possible outcomes:

- Either, the memory is updated by allocating a fresh memory block of size c at an address a such that the whole block does not intersect the heap's domain. Consequently, the values stored in the block are initialized with indeterminate values $v_0, \ldots v_{c-1}$, and the variable `x` is assigned the address a.

- Or, the value 0, also written or `0x0`, is assigned to `x`. In that case no memory allocation is performed.

Semantics of compound statements are defined straightforwardly. The semantics of a sequence of statements is the composition of the semantics. For a conditional statement, its semantics is obtained by first filtering the two branches using the semantics of the boolean expressions then applying the semantics of the branches, and merging the two results. We write $\neg b$ for the negation of a boolean condition. Such negation is obtained by replacing the comparison operator by its opposite (*e.g.* "$=$" by "$\neq$", and "$\leqslant$" by "$>$").

Finally, to compute the semantics of a loop, we first compute the set of states that are reachable at the head of the loop, *i.e.* the loop invariant. This corresponds to the least fixed point of a function F that applies one loop iteration to its input and combines it with the set of states given as input of the loop semantics. This least-fixed-point is well-defined since F is upper-continuous on the complete partial order $(\wp(\mathbb{S}_\Omega), \subseteq)$. Once the loop invariant computed, it is filtered using the negation of the loop's condition.

To conclude, we observe that an error may be raised either when a division or a modulo by zero is performed or when the program tries to read or write at an address that is not in the heap's domain. Since all operators presented are Ω-tight, any raised error is propagated throughout the various semantics definitions. This ensures that executing a statement s starting from a memory state (ρ, m), does not cause any run-time error if and only if $\Omega \notin \mathbb{S}[\![s]\!](\rho, m)$.

2.2 A basic numerical analysis

This section describes the core features and results of abstract interpretation. Here, we consider a restriction of `MemImp`, where the only possible left-values are variables, and without dynamic memory allocations. Since the goal of this section is to introduce abstract interpretation to the reader that is not familiar to these concepts, we omit the proofs of theorems and we refer the reader to the tutorial written by Antoiné Miné [Min17].

2.2.1 Abstracting numerical values

As a consequence of this restriction, the only piece of information necessary to carry out the analysis is the values of each variable. Therefore, the memory states can be simplified as *numerical states* $\mathbb{S}_n := (\mathbb{X} \to \mathbb{V})$. To simplify further the presentation of the analysis, let us fix in this section the set of values $\mathbb{V}$ to be the set of integers $\mathbb{Z}$.

Given a set of memory states, one can compute the set of corresponding numerical ones. Specifically, given a memory state (ρ, m) the numerical state related is simply the composition of the heap together with the stack: $m \circ \rho$. Moreover, if a set of memory states contains the error state then so must the corresponding set of numerical states. In summary, the conversion of a set of memory states to a set of numerical ones is computed with the following function:

$$\begin{aligned} \alpha_n : \wp(\mathbb{S}_\Omega) &\longrightarrow \wp(\mathbb{S}_{n\Omega}) \\ \Sigma &\longmapsto \{\nu \mid \exists (\rho, m) \in \Sigma, \nu = m \circ \rho\} \cup (\Sigma \cap \{\Omega\}) \end{aligned}$$

Conversely, given a set of numerical states, Σ_n, we can calculate the set of corresponding memory states. That is, the memory states related to numerical states in Σ_n.

$$\begin{aligned} \gamma_n : \wp(\mathbb{S}_{n\Omega}) &\longrightarrow \wp(\mathbb{S}_\Omega) \\ \Sigma_n &\longmapsto \{(\rho, m) \mid \exists \nu \in \Sigma_n, \nu = m \circ \rho\} \cup (\Sigma_n \cap \{\Omega\}) \end{aligned}$$

This transformation from memory to numerical states is an example of abstraction. Specifically, the set of memory states and numerical states together with the function α_n and γ_n form a Galois connection.

Definition 2.2: Galois connection

Let $(A, \sqsubseteq)$ and $(C, \leqslant)$ be two partially ordered sets. A pair of functions $\gamma : A \to C$ and $\alpha : C \to A$ forms a *Galois connection* if and only if:

$$\forall a \in A, \forall c \in C, c \leqslant \gamma(a) \Leftrightarrow \alpha(a) \sqsubseteq c$$

In that case, we write $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$ and C is called the *concrete domain*, A the *abstract domain*, γ the *concretization function*, and α the *abstraction function*.

Theorem 2.1: Numerical translation is a Galois connection

$(\wp(\mathbb{S}_\Omega), \subseteq) \underset{\alpha_n}{\overset{\gamma_n}{\leftrightarrows}} (\wp(\mathbb{S}_{n\Omega}), \subseteq)$ forms a Galois connection

Proof. Let $\Sigma \in \wp(\mathbb{S}_\Omega)$, and $\Sigma_n \in \wp(\mathbb{S}_{n\Omega})$.

$$\begin{aligned}
& \Sigma \subseteq \gamma_n(\Sigma_n) \\
\Leftrightarrow\ & \Sigma \subseteq \{(\rho, m) \mid \exists \nu \in \Sigma_n, \nu = m \circ \rho\} \cup (\Sigma_n \cap \{\Omega\}) \\
\Leftrightarrow\ & \begin{cases} \forall (\rho, m) \in \Sigma, \exists \nu \in \Sigma_n, \nu = m \circ \rho \\ \Omega \in \Sigma \Rightarrow \Omega \in \Sigma_n \end{cases} \\
\Leftrightarrow\ & \{m \circ \rho \mid (\rho, m) \in \Sigma\} \cup (\Sigma \cap \{\Omega\}) \subseteq \Sigma_n \\
\Leftrightarrow\ & \alpha_n(\Sigma) \subseteq \Sigma_n
\end{aligned}$$

□

Definition 2.3: Sound and exact abstraction

In a Galois connection $(C, \leqslant) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, an abstract element $a \in A$ is a *sound abstraction* of a concrete element $c \in C$ if and only if $c \leqslant \gamma(a)$. It is an *exact abstraction* if $c = \gamma(a)$.

2.2.2 The interval abstract domain

Though $\wp(\mathbb{S}_{n\Omega})$ forms an abstraction of $\wp(\mathbb{S}_\Omega)$, it is not an effective abstract domain. It fails to offer a synthetic representation of program states. To further abstract the set of numerical states, one can simply consider, for each variable, the upper and lower bounds of the range of possible values. This is the *interval abstraction*.

Definition 2.4: Interval partially ordered set

The set of interval values[2] is defined as:

$$\mathbb{I}^\sharp := \{(a, b) \in (\mathbb{Z} \uplus \{-\infty\}) \times (\mathbb{Z} \uplus \{+\infty\}) \mid a \leqslant b\} \uplus \{\bot_\mathbb{I}\}$$

This set is equipped with an order relation $\sqsubseteq^\sharp_\mathbb{I}$ that satisfies:

- $\forall I \in \mathbb{I}^\sharp, \bot_\mathbb{I} \sqsubseteq^\sharp_\mathbb{I} I$
- $\forall (a, b), (c, d) \in \mathbb{I}^\sharp \setminus \{\bot_\mathbb{I}\}, (a, b) \sqsubseteq^\sharp_\mathbb{I} (c, d) \Leftrightarrow c \leqslant a \wedge b \leqslant d$

Moreover, given two interval values, their *least upper bound* is defined as:

$$\begin{aligned}
X \sqcup_\mathbb{I} \bot_\mathbb{I} &:= X \\
\bot_\mathbb{I} \sqcup_\mathbb{I} X &:= X \\
(a, b) \sqcup_\mathbb{I} (c, d) &:= (\min(a, c), \max(b, d))
\end{aligned}$$

The order relation of the interval poset is defined to correspond to the inclusion of intervals. Additionally, we use a specific value $\bot_\mathbb{I}$ to denote the empty set of values. Figure 2.4 depicts a schematic version of the interval poset. Dashed vertices denote order relation with intermediary elements that are not depicted in the diagram.

By employing the interval poset, we can construct the set of numerical abstract states where each program variable is assigned an interval value. To denote situations where the analysis answers "I don't know", we use an abstract value denoted as $\top^\sharp_n$. This value contains the case where the analysis encountered a possible run-time error.

[2] Note that interval abstract values are written with parenthesis to prevent confusing them with integer intervals, written with brackets.

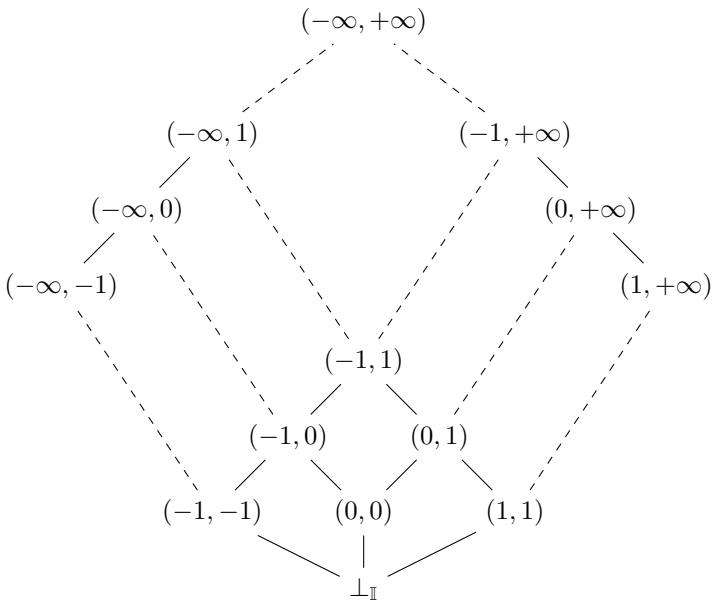


Figure 2.4: Simplified Hasse diagram of the interval poset $(\mathbb{I}^\sharp, \sqsubseteq^\sharp_{\mathbb{I}})$

Definition 2.5: Numerical abstract states

A *numerical abstract state* $\sigma^\sharp \in \mathbb{S}^\sharp_n$ is either:

- a function $\sigma^\sharp : \mathbb{X} \to \mathbb{I}^\sharp$, assigning to each variable the range of its value;
- an element, written $\top^\sharp_n$, expressing that we have no information.

The set of numerical abstract states is equipped with an ordering $\sqsubseteq^\sharp_n$

- $\forall \sigma^\sharp \in \mathbb{S}^\sharp_n, \sigma^\sharp \sqsubseteq^\sharp_n \top^\sharp_n$
- $\forall (\sigma^\sharp_1, \sigma^\sharp_2) \in (\mathbb{S}^\sharp_n \setminus \{\top^\sharp_n\})^2, \sigma^\sharp_1 \sqsubseteq^\sharp_n \sigma^\sharp_2 \Leftrightarrow \forall \mathtt{x} \in \mathbb{X}, \sigma^\sharp_1(\mathtt{x}) \sqsubseteq^\sharp_{\mathbb{I}} \sigma^\sharp_2(\mathtt{x})$

Given a set of numerical states one can compute the corresponding numerical abstract state as follows:

$$\begin{array}{rcll} \alpha_{\mathbb{I}} : \wp(\mathbb{S}_{n\Omega}) & \longrightarrow & \mathbb{S}^\sharp_n & \\ \varnothing & \longmapsto & (\lambda \mathtt{x}.\bot_{\mathbb{I}}) & \\ \Sigma_n & \longmapsto & \begin{cases} \top^\sharp_n & \text{if } \Omega \in \Sigma_n \\ \left(\lambda \mathtt{x}.(\inf_{\nu \in \Sigma_n} \nu(\mathtt{x}), \sup_{\nu \in \Sigma_n} \nu(\mathtt{x}))\right) & \text{otherwise} \end{cases} & \end{array}$$

Conversely, one can determine the set of numerical states that is abstracted by the current abstract store.

$$\begin{array}{rcl} \gamma_{\mathbb{I}} : \mathbb{S}^\sharp_n & \longrightarrow & \wp(\mathbb{S}_{n\Omega}) \\ \top^\sharp_n & \longmapsto & \mathbb{S}_{n\Omega} \\ \sigma^\sharp & \longmapsto & \begin{cases} \varnothing & \text{if } \exists \mathtt{x} \in \mathbb{X}, \sigma^\sharp(\mathtt{x}) = \bot_{\mathbb{I}} \\ \left\{\nu \mid \forall \mathtt{x} \in \mathbb{X}, a \leqslant \nu(\mathtt{x}) \leqslant b \text{ where } \sigma^\sharp(\mathtt{x}) = (a, b)\right\} & \end{cases} \end{array}$$

The numerical abstract states form an abstract domain of numerical states.

Theorem 2.2: Interval galois connection

$(\mathbb{S}_{n\Omega}, \subseteq) \underset{\alpha_{\mathbb{I}}}{\overset{\gamma_{\mathbb{I}}}{\leftrightarrows}} (\mathbb{S}^\sharp_n, \sqsubseteq^\sharp_n)$ forms a Galois Connection.

2.2.3 Interval analysis

This section outlines the process of constructing a sound static analysis using the interval domain defined in the previous section.

$$
\begin{aligned}
\mathbb{E}[\![\bullet]\!]^\sharp_n &: \langle \mathsf{expr} \rangle \times \mathbb{S}^\sharp_n \to \mathbb{I}^\sharp \uplus \{\top^\sharp_n\} \\
\mathbb{E}[\![\_]\!]^\sharp_n(\top^\sharp_n) &= \top^\sharp_n \\
\mathbb{E}[\![c]\!]^\sharp_n(\sigma^\sharp) &= (c, c) \\
\mathbb{E}[\![\mathtt{x}]\!]^\sharp_n(\sigma^\sharp) &= \sigma^\sharp(\mathtt{x}) \\
\mathbb{E}[\![*e]\!]^\sharp_n(\sigma^\sharp) &= \top^\sharp_n \\
\mathbb{E}[\![\&l]\!]^\sharp_n(\sigma^\sharp) &= \top^\sharp_n \\
\mathbb{E}[\![e \oplus e']\!]^\sharp_n(\sigma^\sharp) &= \begin{cases} \top^\sharp_n & \text{if } \mathbb{E}[\![e]\!]^\sharp_n(\sigma^\sharp) = \top^\sharp_n \vee \mathbb{E}[\![e']\!]^\sharp_n(\sigma^\sharp) = \top^\sharp_n \\ \mathbb{E}[\![e]\!]^\sharp_n(\sigma^\sharp) \oplus^\sharp_n \mathbb{E}[\![e']\!]^\sharp_n(\sigma^\sharp) & \text{otherwise} \end{cases}
\end{aligned}
$$

(a) Evaluation of numerical expression in the interval domain

$$
\begin{aligned}
\bot_{\mathbb{I}} \oplus^\sharp_{\mathbb{I}} X &= X \oplus^\sharp_{\mathbb{I}} \bot_{\mathbb{I}} = \bot_{\mathbb{I}} \\
(a,b) +^\sharp_{\mathbb{I}} (c,d) &= (a+c, b+d) \\
(a,b) -^\sharp_{\mathbb{I}} (c,d) &= (a-d, b-c) \\
(a,b) \times^\sharp_{\mathbb{I}} (c,d) &= (\min\Delta, \max\Delta) \qquad \text{where } \Delta := \{ac, bc, ad, bd\} \\
(a,b)/^\sharp_{\mathbb{I}}(c,d) &= \begin{cases} (\min(b/c, b/d), \max(a/c, a/d)) & \text{if } d < 0 \\ \top^\sharp_n & \text{if } c \leqslant 0 \leqslant d \\ (\min(a/c, a/d), \max(b/c, b/d)) & \text{if } 0 < c \end{cases} \\
(a,b)\%^\sharp_{\mathbb{I}}(c,d) &= \begin{cases} \top^\sharp_n & \text{if } c \leqslant 0 \leqslant d \\ (-l, l) & \text{where } l := \max(|c|, |d|) \end{cases}
\end{aligned}
$$

(b) Interval numerical operators

Figure 2.5: Abstract semantics of numerical expressions

2.2.3.1 Evaluation of expression

The abstract semantics of an expression e is defined as a function $\mathbb{E}[\![e]\!]^\sharp_n : \mathbb{S}^\sharp_n \to \mathbb{I}^\sharp \uplus \{\top^\sharp_n\}$, depicted in Figure 2.5a, which maps an abstract state to an interval. This interval over-approximates the potential values of e in the numerical states described by the abstract state. The function returns the default "I don't know" value, $\top^\sharp_n$, if we attempt to evaluate an expression that involves the memory layout such as pointer dereference and address evaluation. This abstract value is the only one that encapsulates the possibility of a memory error. Similar to the error state in the concrete semantics, if the default value is returned at some point, it is propagated through the remaining of the evaluation.

When the expression to evaluate is an arithmetic operation, the semantics relies on abstract interval operators $\oplus^\sharp_n$. The definition of interval operators is presented in Figure 2.5b. If the operation is a division or a modulo and the divisor is an interval containing the value 0, it implies that the abstract values contain numerical states that could lead to a division-by-zero error. Therefore, the value returned is also $\top^\sharp_n$.

The value returned by $\mathbb{E}[\![e]\!]^\sharp_n(\sigma^\sharp)$ is a sound over-approximation of the possible values of e in the states synthesized by $\sigma^\sharp$. Of course, if $\mathbb{E}[\![e]\!]^\sharp_n(\sigma^\sharp)$ returns $\top^\sharp_n$, this over-approximation trivially holds.

Theorem 2.3: Soundness of $\mathbb{E}[\![\bullet]\!]^\sharp_n$

For any expression e and abstract state $\sigma^\sharp$, if $\mathbb{E}[\![e]\!]^\sharp_n(\sigma^\sharp) = (a, b)$, then $\mathbb{E}[\![e]\!] \circ \alpha_n(\sigma^\sharp) \subseteq [a, b]$.

Example 2.1: Incompletness of $\mathbb{E}[\![\bullet]\!]^\sharp_n$

If we try to compute the abstract evaluation of the expression $1/(2 \times \mathtt{x} - 1)$ in the abstract state $\sigma^\sharp : \mathtt{x} \mapsto (0, 1)$, we obtain:

$$
\begin{aligned}
\mathbb{E}[\![1/(2 \times \mathtt{x} - 1)]\!]^\sharp_n(\sigma^\sharp) &= (1,1)/^\sharp_{\mathbb{I}}((2,2) \times^\sharp_{\mathbb{I}} (0,1) -^\sharp_{\mathbb{I}} (1,1)) \\
&= (1,1)/^\sharp_{\mathbb{I}}((0,2) -^\sharp_{\mathbb{I}} (1,1)) \\
&= (1,1)/^\sharp_{\mathbb{I}}(-1,1) \\
&= \top^\sharp_n
\end{aligned}
$$

This result is not precise at all. The evaluation of the expression when the value of $\mathtt{x}$ is either 0 or 1 does not cause any error and the corresponding values are either -1 or 1.

$$\begin{aligned}
\mathbf{assign}^\sharp_n &: \langle \text{l-value} \rangle \times \langle \text{expr} \rangle \times \mathbb{S}^\sharp_n \to \mathbb{S}^\sharp_n \\
\mathbf{assign}^\sharp_n(_, _, \top^\sharp_n) &= \top^\sharp_n \\
\mathbf{assign}^\sharp_n(\mathtt{x}, e, \sigma^\sharp) &= \begin{cases} \top^\sharp_n & \text{if } \mathbb{E}[\![e]\!]^\sharp_n(\sigma^\sharp) = \top^\sharp_n \\ \sigma^\sharp\left[\mathtt{x} \mapsto \mathbb{E}[\![e]\!]^\sharp_n(\sigma^\sharp)\right] & \text{otherwise} \end{cases} \\
\mathbf{assign}^\sharp_n(\ast e, _, _) &= \top^\sharp_n
\end{aligned}$$

(a) Definition of $\mathbf{assign}^\sharp_n$ operator

$$\begin{aligned}
\mathbf{malloc}^\sharp_n &: \mathbb{X} \times \mathbb{V} \times \mathbb{S}^\sharp_n \to \mathbb{S}^\sharp_n \\
\mathbf{malloc}^\sharp_n(_, _, \top^\sharp_n) &= \top^\sharp_n \\
\mathbf{malloc}^\sharp_n(\mathtt{x}, c, \sigma^\sharp) &= \sigma^\sharp\left[\mathtt{x} \mapsto (0, +\infty)\right]
\end{aligned}$$

(b) Definition of $\mathbf{malloc}^\sharp_n$ operator

$$\begin{aligned}
\mathbf{guard}^\sharp_n &: \langle \text{bexpr} \rangle \times \mathbb{S}^\sharp_n \to \mathbb{S}^\sharp_n \\
\mathbf{guard}^\sharp_n(_, \top^\sharp_n) &= \top^\sharp_n \\
\mathbf{guard}^\sharp_n(\mathtt{x} \leqslant \mathtt{y}, \sigma^\sharp) &= \begin{cases} \sigma^\sharp\begin{bmatrix} \mathtt{x} \mapsto (a, \min(b, d)) \\ \mathtt{y} \mapsto (\max(a, c), d) \end{bmatrix} & \text{if } \begin{cases} \sigma^\sharp(\mathtt{x}) = (a, b) \\ \wedge\, \sigma^\sharp(\mathtt{y}) = (c, d) \\ \wedge\, a \leqslant d \end{cases} \\ \lambda_.\bot_{\mathbb{I}} & \text{otherwise} \end{cases} \\
\mathbf{guard}^\sharp_n(\mathtt{x} = \mathtt{y}, \sigma^\sharp) &= \mathbf{guard}^\sharp_n(\mathtt{x} \leqslant \mathtt{y}, \mathbf{guard}^\sharp_n(\mathtt{y} \leqslant \mathtt{x}, \sigma^\sharp)) \\
\mathbf{guard}^\sharp_n(e \bowtie e, \sigma^\sharp) &= \begin{cases} \top^\sharp_n & \text{if } \mathbb{E}[\![e]\!]^\sharp_n(\sigma^\sharp) = \top^\sharp_n \vee \mathbb{E}[\![e']\!]^\sharp_n(\sigma^\sharp) = \top^\sharp_n \\ \sigma^\sharp & \text{otherwise} \end{cases}
\end{aligned}$$

(c) Definition of $\mathbf{guard}^\sharp_n$ operator

Figure 2.6: Definition of abstract operators

2.2.3.2 Abstract transfer function

The $\mathbf{assign}^\sharp_n$ operator over-approximates the **assign** operator defined in the concrete semantics of `MemImp`. Its definition is presented in Figure 2.6a. If the abstract state given as input is not $\top^\sharp_n$ and the assigned left-value is a variable, then $\mathbf{assign}^\sharp_n$ evaluates the expression. If the result of the evaluation is an interval, then the abstract state is returned after being updated with the new interval. In all other cases $\mathbf{assign}^\sharp_n$ returns $\top^\sharp_n$.

Theorem 2.4: Soundness of $\mathbf{assign}^\sharp_n$

For any expression e, left-value l, and abstract state $\sigma^\sharp$:

$$\mathbf{assign}\left(l, e, \gamma_n\left(\sigma^\sharp\right)\right) \subseteq \gamma_n\left(\mathbf{assign}^\sharp_n(l, e, \sigma^\sharp)\right)$$

Given a variable $\mathtt{x} \in \mathbb{X}$, a constant value $c \in \mathbb{V}$, and an abstract value $\sigma^\sharp$, $\mathbf{malloc}^\sharp_n(\mathtt{x}, c, \sigma^\sharp)$ simply assigns to $\mathtt{x}$ the interval $(0, +\infty)$. Indeed, whether a memory cell is actually allocated is not relevant in our analysis. It only has to consider the assignment of the allocated address (or the `null` value if no allocation happens) to $\mathtt{x}$.

Theorem 2.5: Soundness of $\mathbf{malloc}^\sharp_n$

For any variable $\mathtt{x}$, constant c, and abstract state $\sigma^\sharp$:

$$\left(\mathbb{S}[\![\mathtt{x} = \mathtt{malloc}(c)]\!] \circ \gamma_n\right)\left(\sigma^\sharp\right) \subseteq \gamma_n\left(\mathbf{malloc}^\sharp_n(\mathtt{x}, c, \sigma^\sharp)\right)$$

The goal of the $\mathbf{guard}^\sharp_n$ operator is to soundly over-approximate the semantics of boolean expression. The definition presented in Figure 2.6c is a naive version. If the constraint is a simple inequality constraint between two program variables, then we can squeeze the bound of the intervals of the variables using this constraint. If this inequality is not feasible with the current range of variables, then we return the abstract value where all program variables are mapped to $\bot_{\mathbb{I}}$. We extend the $\mathbf{guard}^\sharp_n$ to equalities constraints between program variables. For the other possible constraints, the operator applies a default strategy. If the evaluation of one of the two compared expressions returns $\top^\sharp_n$, then so does $\mathbf{guard}^\sharp_n$. Otherwise, the input abstract value is returned without modification.

$$
\begin{aligned}
\mathbb{S}[\![\bullet]\!]^\sharp_n &: \langle \mathsf{stmt} \rangle \times \mathbb{S}^\sharp_n \to \mathbb{S}^\sharp_n \\
\mathbb{S}[\![\texttt{skip}]\!]^\sharp_n(\sigma^\sharp) &= \sigma^\sharp \\
\mathbb{S}[\![l = e]\!]^\sharp_n(\sigma^\sharp) &= \mathbf{assign}^\sharp_n(l, e, \sigma^\sharp) \\
\mathbb{S}[\![\mathtt{x} = \texttt{malloc}(c)]\!]^\sharp_n(\sigma^\sharp) &= \mathbf{malloc}^\sharp_n(\mathtt{x}, c, \sigma^\sharp) \\
\mathbb{S}[\![\mathtt{x} = \texttt{f}()]\!]^\sharp_n(\sigma^\sharp) &= \top^\sharp_n \qquad \text{if there is a control flow from the body of } \texttt{f} \text{ to the current statement where } \texttt{f}()\{s; \texttt{return } e;\} \text{ is the declaration of } \texttt{f} \\
\mathbb{S}[\![\mathtt{x} = \texttt{f}()]\!]^\sharp_n(\sigma^\sharp) &= \mathbf{assign}^\sharp_n(\mathtt{x}, e, \mathbb{S}[\![s]\!]^\sharp_n(\sigma^\sharp)) \\
\mathbb{S}[\![s; s']\!]^\sharp_n(\sigma^\sharp) &= \left(\mathbb{S}[\![s']\!]^\sharp_n \circ \mathbb{S}[\![s]\!]^\sharp_n\right)(\sigma^\sharp) \\
\mathbb{S}[\![\texttt{if}(b)\{s\}\texttt{else}\{s'\}]\!]^\sharp_n(\sigma^\sharp) &= \mathbb{S}[\![s]\!]^\sharp_n(\mathbf{guard}^\sharp_n(b, \sigma^\sharp)) \sqcup^\sharp_n \mathbb{S}[\![s']\!]^\sharp_n(\mathbf{guard}^\sharp_n(\neg b, \sigma^\sharp)) \\
\mathbb{S}[\![\texttt{while}(b)\{s\}]\!]^\sharp_n(\sigma^\sharp) &= \mathbf{guard}^\sharp_n(\neg b)\left(\lim_k \sigma^\sharp_k\right) \\
&\text{where } \begin{cases} \sigma^\sharp_0 := \lambda \mathtt{x}.\bot_{\mathbb{I}} \\ \sigma^\sharp_{k+1} := \sigma^\sharp_k \nabla^\sharp_n F^\sharp\left(\sigma^\sharp_k\right) \end{cases} \\
&\text{and } F^\sharp(\sigma^\sharp_i) := \sigma^\sharp \sqcup^\sharp_n \mathbb{S}[\![s]\!]^\sharp_n(\mathbf{guard}^\sharp_n(b, \sigma^\sharp_i))
\end{aligned}
$$

Figure 2.7: Abstract semantics

A more precise result could be achieved using a variant of the HC4 algorithm [BGGP99]. This algorithm first evaluates the numerical expressions involved in the constraints, then it infers the necessary values of variables that satisfy the constraint.

Theorem 2.6: Soundness of $\mathbf{guard}^\sharp_n$

For any boolean expression b and abstract state $\sigma^\sharp$:

$$(\mathbb{C}[\![b]\!] \circ \gamma_n)\left(\sigma^\sharp\right) \subseteq \gamma_n\left(\mathbf{guard}^\sharp_n(b, \sigma^\sharp)\right)$$

2.2.3.3 Abstract semantics

The abstract semantics of a statement s is a function $\mathbb{S}[\![s]\!]^\sharp_n : \mathbb{S}^\sharp_n \to \mathbb{S}^\sharp_n$. Its definition is presented in Figure 2.7. It is important to note that the abstract semantics must be effectively computable. This means that, for any abstract value $\sigma^\sharp$, the evaluation of $\mathbb{S}[\![s]\!]^\sharp_n(\sigma^\sharp)$ terminates.

Similarly to the concrete semantics, the `skip` statement abstract semantics boils down to the identity function, and the abstract semantics of a sequence statement $s; s'$ is the composition of the abstract semantics of s' with the abstract semantics of s. Moreover, for the assign statements, the abstract semantics is equal to the $\mathbf{assign}^\sharp_n$ operator. Likewise, for dynamic memory allocation, the abstract semantics is defined with the $\mathbf{malloc}^\sharp_n$ operator.

For function calls, we consider two cases:

- The first case occurs when the analysis detects recursive functions. To ensure the analysis terminates, the analysis immediately returns $\top^\sharp_n$.
- The second case corresponds to well-founded function calls. In this scenario, the abstract semantics is similar to the concrete one.

For conditional statements, we analyze the two branches separately. First, the abstract semantics use the $\mathbf{guard}^\sharp_n$ operator to compute the abstract states at the beginning of the two branches. Next, the two branches are interpreted individually. Finally, the states at the end of the two branches are merged using the *join* operator.

Definition 2.6: Join operator

The join operator for the interval abstract domain $\sqcup^\sharp_n$ is defined as:

$$
\begin{aligned}
\bullet \sqcup^\sharp_n \bullet &: \mathbb{S}^\sharp_n \times \mathbb{S}^\sharp_n \longrightarrow \mathbb{S}^\sharp_n \\
\top^\sharp_n \sqcup^\sharp_n \_ &= \_ \sqcup^\sharp_n \top^\sharp_n = \top^\sharp_n \\
\sigma^\sharp_1 \sqcup^\sharp_n \sigma^\sharp_2 &= \lambda \mathtt{x}.\sigma^\sharp_1(\mathtt{x}) \sqcup_{\mathbb{I}} \sigma^\sharp_2(\mathtt{x})
\end{aligned}
$$

The purpose of the $\sqcup^\sharp_n$ operator is to conservatively merge two abstract states into one. It is designed in such a way that any concrete state synthesized in one input of $\sqcup^\sharp_n$ is also synthesized by

the output. In other words:

Theorem 2.7: Soundness of $\sqcup_n^\sharp$

For any abstract values $\sigma_1^\sharp$ and $\sigma_2^\sharp$, we have: $\gamma_n\left(\sigma_1^\sharp\right) \cup \gamma_n\left(\sigma_2^\sharp\right) \subseteq \gamma_n\left(\sigma_1^\sharp \sqcup_n^\sharp \sigma_2^\sharp\right)$

For loops, the analysis cannot compute an over-approximation of the invariant in the same manner as the concrete semantics does. Indeed, Kleene's fixpoint theorem may require an infinite number of iterations. To address this issue, the analysis constructs a sound over-approximation of the invariant using convergence acceleration, employing a *widening* operator.

Definition 2.7: Interval abstract domain widening operator

The widening operator for the interval abstract domain $\nabla_n^\sharp$ is defined as:

$$\begin{aligned} \bullet \nabla_n^\sharp \bullet \quad &: \mathbb{S}_n^\sharp \times \mathbb{S}_n^\sharp \longrightarrow \mathbb{S}_n^\sharp \\ \top_n^\sharp \nabla_n^\sharp \_ = \_ \nabla_n^\sharp \top_n^\sharp &= \top_n^\sharp \\ \sigma_1^\sharp \nabla_n^\sharp \sigma_2^\sharp &= \lambda \mathtt{x}.\sigma_1^\sharp(\mathtt{x}) \nabla_{\mathbb{I}} \sigma_2^\sharp(\mathtt{x}) \end{aligned}$$

where the interval values widening is defined as:

$$\begin{aligned} \bullet \nabla_{\mathbb{I}} \bullet \quad &: \mathbb{I}^\sharp \times \mathbb{I}^\sharp \longrightarrow \mathbb{I}^\sharp \\ \bot_{\mathbb{I}} \nabla_{\mathbb{I}} \_ \quad &= \_ \nabla_{\mathbb{I}} \bot_{\mathbb{I}} = \bot_{\mathbb{I}} \\ (a,b) \nabla_{\mathbb{I}} (c,d) &= \left(\begin{cases} a & \text{if } a \leqslant c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } d \leqslant b \\ +\infty & \text{otherwise} \end{cases} \right) \end{aligned}$$

The $\nabla_n^\sharp$ operator defined above is a correct widening operator. First, it computes a sound over-approximation of its inputs. Second, it ensures the convergence of iteration by removing unstable variable bounds.

Theorem 2.8: Soundness and termination property of $\nabla_n^\sharp$

For any abstract states $\sigma_1^\sharp$ and $\sigma_2^\sharp$, $\gamma_n\left(\sigma_1^\sharp\right) \cup \gamma_n\left(\sigma_2^\sharp\right) \subseteq \gamma_n\left(\sigma_1^\sharp \nabla_n^\sharp \sigma_2^\sharp\right)$

Additionally, for any sequence of abstract states $\left(\sigma_k^\sharp\right)_{k\in\mathbb{N}}$, the sequence of abstract states $\left(\sigma_k^{\sharp(\nabla)}\right)_{k\in\mathbb{N}}$ defined as $\sigma_0^{\sharp(\nabla)} := \sigma_0^\sharp$ and $\sigma_{k+1}^{\sharp(\nabla)} := \sigma_k^{\sharp(\nabla)} \nabla_n^\sharp \sigma_{k+1}^\sharp$ is ultimately stationary.

With the widening operator, we can define a sound and computable abstract loop invariant as the limit of a sequence of abstract states. The sequence is initialized with an empty abstract state. Each iteration step is computed by widening the previous element of the sequence with the result of applying function $F^\sharp$ to the same element. This function $F^\sharp$ is the abstract counterpart of function F defined in Figure 2.3b. Finally, the result of the abstract semantics of loops is obtained by applying the $\mathbf{guard}_n^\sharp$ operator to the abstract invariant with the negation of the loop condition.

Theorem 2.9: Soundness of the abstract semantics

For any numerical abstract value $\sigma^\sharp$, and any statement s,

$$\left(\mathbb{S}[\![s]\!] \circ \gamma_n\right)\left(\sigma^\sharp\right) \subseteq \left(\gamma_n \circ \mathbb{S}[\![s]\!]_n^\sharp\right)\left(\sigma^\sharp\right)$$

2.2.3.4 Wrapping up the analysis

Additionally, if a state property $P \subseteq \mathbb{S}$ can be exactly represented with an abstract value $\sigma_P^\sharp$ (*i.e.* $P = \gamma_n(\sigma_P^\sharp)$), then the analysis verifies that the final abstract state $\sigma_{final}^\sharp$ satisfies this property by checking whether $\sigma_{final}^\sharp \sqsubseteq_n^\sharp \sigma_P^\sharp$.

Theorem 2.10: Soundness of inclusion checking $\sqsubseteq_n^\sharp$

For any abstract states $\sigma_1^\sharp$ and $\sigma_2^\sharp$, $\sigma_1^\sharp \sqsubseteq_n^\sharp \sigma_2^\sharp \Longrightarrow \gamma_n\left(\sigma_1^\sharp\right) \subseteq \gamma_n\left(\sigma_2^\sharp\right)$

To conclude, given a function contract $\left\{\sigma_{pre}^\sharp\right\}\ \mathtt{x} = \mathtt{f}()\ \left\{\sigma_{post}^\sharp\right\}$, the analysis of this contract boils down to $\mathbb{S}[\![\mathtt{x} = \mathtt{f}()]\!]_n^\sharp\left(\sigma_{pre}^\sharp\right) \sqsubseteq_n^\sharp \sigma_{post}^\sharp$.

Theorem 2.11: Soundness of the analysis

Given a function `f` and a variable `x`, two states properties P_{pre} and P_{post}, and two abstract states $\sigma^\sharp_{pre}$ and $\sigma^\sharp_{post}$, if:

- $\sigma^\sharp_{pre}$ is a sound abstraction of P_{pre},
- $\sigma^\sharp_{post}$ is an exact abstraction of P_{post},
- $\mathbb{S}[\![\mathtt{x} = \mathtt{f}()]\!]^\sharp_n \left(\sigma^\sharp_{pre}\right) \sqsubseteq^\sharp_n \sigma^\sharp_{post}$,

then, the Hoare triple $\{P_{pre}\}\, \mathtt{x} = \mathtt{f}()\, \{P_{post}\}$ holds.

Example 2.2: Proving the absence of run-time error with the numerical analysis

For example, the absence of run-time error property can be exactly abstracted as $\sigma^\sharp_{ARTE} := \lambda \mathtt{x}.(-\infty, +\infty)$.

As a conclusion, we point out that this analysis can be defined for any abstract domain $\mathbb{D}^\sharp$ that implements the following operator:

- $\mathbf{assign}^\sharp : \langle \mathsf{l\text{-}value} \rangle \times \langle \mathsf{expr} \rangle \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\mathbf{malloc}^\sharp : \mathbb{X} \times \mathbb{V} \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\mathbf{guard}^\sharp : \langle \mathsf{bexpr} \rangle \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\sqcup^\sharp : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\nabla^\sharp : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$
- $\sqsubseteq^\sharp : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \{\mathbf{true}, \mathbf{false}\}$
- $\gamma : \mathbb{D}^\sharp \to \wp\left(\mathbb{S}_\Omega\right)$

Indeed, the first five operators are sufficient to define the abstract semantics of any statement s, $[\![s]\!]^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$. Thanks to the abstract inclusion test $\sqsubseteq^\sharp$, the analysis can establish whether the post condition is satisfied by the final state. The concretization function is defined in order to formalize the soundness of our analysis. That is to say, if the operators satisfy the soundness properties stated above, then the analysis is sound.

2.3 Abstracting the memory states with separation logic

This section presents how a subset of separation logic [Rey02] can be employed as an abstract domain for memory states. We first demonstrate how to represent a single memory state in separation logic. Then we show how inductive predicates can summarize unbounded data structures.

Contrary to the abstract domain defined in the previous section, abstracting the memory layout of several memory states require expressing constraints on the support of the heap. This corresponds to an unbounded number of values. To do so we introduce a countable set $\mathcal{V}_n$ of *numeric symbolic variables*,written with Greek letters $\alpha, \beta, \ldots$, taking values in $\mathbb{V}$. Furthermore, for each program variable `x`, we pick one symbolic variable, written $\underline{\mathtt{x}}$, to denote the address of `x`.

2.3.1 Abstracting simple memory state

An abstract memory state $m^\sharp$, is a finite collection of atomic separation logic predicates. The first kind of atomic predicate is the *points-to predicate*, written $\alpha.\mathtt{f} \mapsto \beta$. It characterizes a memory with a single cell at an address described by α, with an offset corresponding to the value of field `f`, containing a value described by β. The second kind of atomic predicate is the **emp** predicate that describes an empty memory. These predicates are combined using the separating conjunction $*$. The separating conjunction $m^\sharp_1 * m^\sharp_2$ expresses that the memory described by $m^\sharp_1$ is disjoint from the one described by $m^\sharp_2$.

Listing 2.1: Singly linked list definition

```
1  struct node {
2    struct node *next;
3    int data;
4  }
```

Definition 2.8: Abstract memory states

Abstract memory states are defined by the following grammar:

$$\begin{aligned} m^\sharp ::= {} & \alpha.\mathtt{f} \mapsto \beta \qquad \alpha, \beta \in \mathcal{V}_n, \mathtt{f} \in \mathbb{F} \\ \mid {} & \mathbf{emp} \\ \mid {} & m^\sharp * m^\sharp \end{aligned}$$

The set of abstract memory states is written $\mathbb{M}^\sharp$.

The semantics of abstract memory states is defined using a satisfiability relation $\vDash_m$. This relation expresses that an abstract memory state $m^\sharp$ indeed synthesizes a concrete heap m. To monitor the values taken by the symbolic variables present in the separation logic formula, we employ a *numerical valuation function* $\sigma_n : \mathcal{V}_n \to \mathbb{V}$. The heap m satisfies a points-to predicate if and only if m is a partial function with exactly one entry: the cell at address $\sigma_n(\alpha) + \varphi_{\mathbb{F}}(\mathtt{f})$ contains the value $\sigma_n(\beta)$. The **emp** predicate is satisfied if and only if the heap is the empty function. To satisfy a separating conjunction $m_1^\sharp * m_2^\sharp$, the heap m must be split into two disjunctive functions heaps m_1, m_2 satisfying $m_1^\sharp$ and $m_2^\sharp$, respectively. Note that in the case of separating conjunction, the numerical valuation is not split.

Definition 2.9: Memory satisfiability relation

The memory satisfiability relation $\vDash_m$ is defined inductively on the syntax of abstract memory heaps as:

$$\begin{aligned} m, \sigma_n \vDash_m \alpha.\mathtt{f} \mapsto \beta \quad & \text{iff } m = \{\sigma_n(\alpha) + \varphi_{\mathbb{F}}(\mathtt{f}) \mapsto \sigma_n(\beta)\} \\ m, \sigma_n \vDash_m \mathbf{emp} \quad & \text{iff } m = \varnothing \\ m, \sigma_n \vDash_m m_1^\sharp * m_2^\sharp \quad & \text{iff } \exists m_1, m_2, \begin{cases} \mathbf{supp}(m_1) \cap \mathbf{supp}(m_2) = \varnothing \\ m = m_1 \uplus m_2 \\ m_1, \sigma_n \vDash_m m_1^\sharp \\ m_2, \sigma_n \vDash_m m_2^\sharp \end{cases} \end{aligned}$$

From the satisfiability relation, we derive the concretization function, which maps an abstract memory state into a set of pairs consisting of a memory state along with a numerical valuation function. The concretization function not only checks that the concrete memory heap satisfies the abstract memory, it also ensures that the stack of the memory state is consistent with the valuation of symbolic variables denoting addresses of program variables.

Definition 2.10: Concretization of abstract memory states

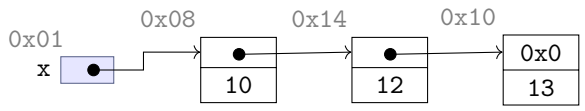
For an abstract memory state $m^\sharp$, the corresponding set of memory states with numerical valuation functions is defined by:

$$\begin{aligned} \gamma_m : \mathbb{M}^\sharp & \longrightarrow \wp\left(\mathbb{S} \times (\mathcal{V}_n \to \mathbb{V})\right) \\ m^\sharp & \longmapsto \left\{ ((\rho, m), \sigma_n) \;\middle|\; \begin{array}{l} m, \sigma_n \vDash_m m^\sharp \\ \forall \mathtt{x} \in \mathbb{X}, \rho(\mathtt{x}) = \sigma_n(\underline{\mathtt{x}}) \end{array} \right\} \end{aligned}$$

Example 2.3: Abstract shape of a singly linked list

Figure 2.8a shows a memory state with a singly linked list. The list type, with the corresponding field, is defined in Listing 2.1. An abstract memory for this concrete memory is presented in Figure 2.8b.

The symbolic valuation σ_n that links the abstract memory to the concrete one (*i.e.* such that the judgement $m, \sigma_n \vDash_m m^\sharp$ holds), is:



(a) A concrete singly linked list

$$\begin{aligned}\underline{\texttt{x}} \mapsto \alpha_0 * \alpha_0.\texttt{next} \mapsto \alpha_1 * \alpha_0.\texttt{data} \mapsto \beta_0\\ * \alpha_1.\texttt{next} \mapsto \alpha_2 * \alpha_1.\texttt{data} \mapsto \beta_1\\ * \alpha_2.\texttt{next} \mapsto \alpha_3 * \alpha_2.\texttt{data} \mapsto \beta_2\end{aligned}$$

(b) An abstract memory of a singly linked list

Figure 2.8: A concrete singly linked list and its abstract counterpart

$$\begin{array}{llll}\alpha_0 \mapsto \texttt{0x08} & \alpha_2 \mapsto \texttt{0x10} & \beta_0 \mapsto 10 & \beta_2 \mapsto 13\\ \alpha_1 \mapsto \texttt{0x14} & \alpha_3 \mapsto \texttt{0x00} & \beta_1 \mapsto 12 & \underline{\texttt{x}} \mapsto \texttt{0x01}\end{array}$$

Example 2.4: Abstract shape of WFS

The tree presented in Figure 1.1 (page 13) can be abstracted using the following separation logic formula:

$$\begin{array}{llll}\alpha_0.\texttt{task} \mapsto \tau_0 & * \alpha_1.\texttt{task} \mapsto \tau_1 & * \alpha_2.\texttt{task} \mapsto \tau_2 & * \alpha_3.\texttt{task} \mapsto \tau_3\\ * \alpha_0.\texttt{left} \mapsto \alpha_1 & * \alpha_1.\texttt{left} \mapsto \delta_0 & * \alpha_2.\texttt{left} \mapsto \alpha_3 & * \alpha_3.\texttt{left} \mapsto \delta_0\\ * \alpha_0.\texttt{right} \mapsto \alpha_2 & * \alpha_1.\texttt{right} \mapsto \delta_0 & * \alpha_2.\texttt{right} \mapsto \delta_0 & * \alpha_3.\texttt{right} \mapsto \delta_0\\ * \alpha_0.\texttt{parent} \mapsto \alpha_0 & * \alpha_1.\texttt{parent} \mapsto \alpha_0 & * \alpha_2.\texttt{parent} \mapsto \alpha_0 & * \alpha_3.\texttt{parent} \mapsto \alpha_2\\ * \tau_0.\texttt{wst} \mapsto \beta_0 & * \tau_1.\texttt{wst} \mapsto \beta_1 & * \tau_2.\texttt{wst} \mapsto \beta_2 & * \tau_3.\texttt{wst} \mapsto \beta_3\end{array}$$

For the sake of readability, points-to predicates that describe a contiguous memory cell are grouped. The node at address α_0 corresponds to the root of the tree. For each tree node with address α_i, the address of the corresponding task control block is τ_i.

2.3.2 Abstract memory states with unbounded data structures

The abstract memory states introduced so far can describe memory states containing a linked list of a fixed size as well as binary tree with a specific layout. However, it cannot represent memory states with singly linked lists of arbitrary lengths, or binary trees that do not share the same layout. To address this difficulty, we build upon the recursive pattern inherent in these data structures, which can be summarized through the use of an *inductive predicate*.

Figure 2.9 presents the syntax of inductive predicates. Intuitively, an instance of inductive predicate $\alpha.\mathbf{pred}(\overrightarrow{\kappa})$ states that α corresponds to the address of a well-formed data structure that can be defined recursively. An inductive predicate call is also parametrized by a set of symbolic variables $\overrightarrow{\kappa}$[3] which determine the value of certain elements within the data structure. An inductive predicate definition is a finite and non-empty disjunction of *cases* or *rules*. Each case is an existentially quantified formula that combines separation logic predicates (called the *shape part*) together with numerical constraints (called the *pure part*). In this context, we restrict the possible separation logic predicates in the shape part. The grammar defining these separation logic predicates, indexed by an ι, asserts for each symbolic variable what kind of variables can be used. We distinguish three kind of numerical symbolic variables. The variable α forms the main parameter kind. The second kind of variables are the predicate's parameters, denoted by the variable κ. The final kind concerns the existentially quantified symbolic variables, β. For instance, points-to predicate can only have the main parameter α as source. These predicates constitute the *cell part*. The shape part may also encompass calls to other inductive predicates (this is the *nested part*), as well as recursive calls to **pred** inductive predicate. These calls form the *recursive part*. For all inductive predicates calls, the main parameter must be an existentially quantified symbolic variable. The combination of the cell and nested parts is called the *local part*. Additionally, the pure part consists of a finite conjunction of comparisons of symbolic numeric expressions, *i.e.* expressions where all variables are symbolic variables. To avoid confusion with the set of expressions in the `MemImp` language, we index symbolic numeric expressions and constraints by the set $\mathcal{V}$ of symbolic variables.

[3] In this thesis, we write $\overrightarrow{\kappa}$ to denote a finite, possibly empty, sequence of symbolic variables: $\kappa_1, \ldots, \kappa_l$.

$$
\begin{array}{rlll}
m_\iota^\sharp ::= & \alpha.\mathtt{f} \mapsto \delta & \delta \in \{\alpha, \beta, \kappa\} & \text{(cell part)} \\
\mid & \beta.\mathbf{pred}(\overrightarrow{\delta}) & \delta \in \{\alpha, \beta, \kappa\} & \text{(recursive/nested calls)} \\
\mid & \mathbf{emp} & & \\
\mid & m_\iota^\sharp * m_\iota^\sharp & &
\end{array}
$$

(a) Syntax of memory part of inductive predicates

$$
\begin{array}{rlll}
e_\mathcal{V} ::= & c & c \in \mathbb{V}, & \text{(constants)} \\
\mid & \alpha & \alpha \in \mathcal{V} & \text{(symbolic variables)} \\
\mid & e_\mathcal{V} \oplus e_\mathcal{V} & \oplus \in \{+, -, \times\} & \text{(numeric operation)}
\end{array}
$$

(b) Syntax of symbolic expressions

$$
\begin{array}{rll}
\varphi_\mathcal{V} ::= & e_\mathcal{V} \bowtie e_\mathcal{V} & \bowtie \in \{=, \neq, \leqslant, <\} \quad \text{(expression comparison)} \\
\mid & \mathbf{true} & \text{(empty conjunction)} \\
\mid & \varphi_\mathcal{V} \wedge \varphi_\mathcal{V} & \text{(conjunction of constraints)}
\end{array}
$$

(c) Syntax of pure part of inductive predicates

$$\alpha.\mathbf{pred}(\overrightarrow{\kappa}) := \bigvee \exists \overrightarrow{\beta}, m_\iota^\sharp \wedge \varphi_\mathcal{V}$$

(d) Syntax of inductive predicates

Figure 2.9: Syntax of inductive predicates

To extend the satisfiability relation $\vDash_m$ to inductive predicates, we first need to define what it means for a numerical valuation to satisfy a constraint. Consequently, we introduce the numerical satisfiability relation.

Definition 2.11: Semantics of symbolic numeric expressions and constraints

The evaluation of a symbolic expression $e_\mathcal{V}$ is a function $\mathbb{E}[\![e_\mathcal{V}]\!]_\mathcal{V} : (\mathcal{V} \to \mathbb{V}) \to \mathbb{V}$ defined inductively:

$$
\begin{aligned}
\mathbb{E}[\![c]\!]_\mathcal{V}(\sigma_n) &:= c \\
\mathbb{E}[\![\alpha]\!]_\mathcal{V}(\sigma_n) &:= \sigma_n(\alpha) \\
\mathbb{E}[\![e_\mathcal{V} \oplus e'_\mathcal{V}]\!]_\mathcal{V}(\sigma_n) &:= \mathbb{E}[\![e_\mathcal{V}]\!]_\mathcal{V}(\sigma_n) \oplus \mathbb{E}[\![e'_\mathcal{V}]\!]_\mathcal{V}(\sigma_n)
\end{aligned}
$$

Conversely, the numerical satisfiability relation between a symbolic valuation σ_n and a symbolic constraint is defined as:

$$
\begin{array}{ll}
\sigma_n \vDash_n e_\mathcal{V} \bowtie e'_\mathcal{V} & \text{iff } \mathbb{E}[\![e_\mathcal{V}]\!]_\mathcal{V}(\sigma_n) \bowtie \mathbb{E}[\![e'_\mathcal{V}]\!]_\mathcal{V}(\sigma_n) \\
\sigma_n \vDash_n \mathbf{true} & \text{always} \\
\sigma_n \vDash_n \varphi_\mathcal{V} \wedge \psi_\mathcal{V} & \text{iff } \sigma_n \vDash_n \varphi_\mathcal{V} \text{ and } \sigma_n \vDash_n \psi_\mathcal{V}
\end{array}
$$

Definition 2.12: Satisfiability relation $\vDash_m$ for inductive predicates

For an inductive predicate $\alpha.\mathbf{pred}(\overrightarrow{\kappa}) = \bigvee_k \exists \overrightarrow{\beta}, m_{\iota,k}^\sharp \wedge \varphi_{\mathcal{V},k}$, we extend the satisfiability relation as follows:

$$
m, \sigma_n \vDash_m \alpha.\mathbf{pred}(\overrightarrow{\kappa}) \quad \text{iff} \quad \begin{array}{l} \text{for some rule index } k, \\ \text{and some values } \vec{c} \in \mathbb{V}, \end{array} \begin{cases} m, \sigma_n[\vec{\beta} \mapsto \vec{c}] \vDash_m m_{\iota,k}^\sharp \\ \sigma_n[\vec{\beta} \mapsto \vec{c}] \vDash_n \varphi_{\mathcal{V},k} \end{cases}
$$

As the concretization function is defined solely using the satisfiablity relation, we do not need to update its definition for inductive predicates.

Example 2.5: The singly linked list inductive predicate

The inductive predicate corresponding to singly linked lists is:

$$
\begin{aligned}
\alpha.\mathbf{list}() := & \mathbf{emp} \wedge \alpha = \mathtt{0x0} \\
& \vee \exists b_n, b_d, \alpha.\mathtt{next} \mapsto \beta_n * \alpha.\mathtt{data} \mapsto \beta_d * \beta_n.\mathbf{list}() \wedge \alpha \neq \mathtt{0x0}
\end{aligned}
$$

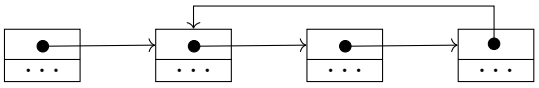
$$\alpha.\mathbf{task} := \exists \beta_t, \beta_w, \alpha.\texttt{wst} \mapsto \beta_t * \alpha.\texttt{weight} \mapsto \beta_w \wedge \alpha \neq \texttt{0x0}$$

$$\begin{array}{lll}
\alpha.\mathbf{tree}(\kappa_p) := \mathbf{emp} \wedge \alpha = \texttt{0x0} & & \\
\quad \vee \exists \beta_t, \beta_l, \beta_r, & \alpha.\texttt{task} \mapsto \beta_t * \alpha.\texttt{parent} \mapsto \kappa_p & \left.\begin{array}{l}\\\end{array}\right\} \text{cell} \\
& * \alpha.\texttt{left} \mapsto \beta_l * \alpha.\texttt{right} \mapsto \beta_r & \left.\begin{array}{l}\\\end{array}\right\} \text{part} \\
& * \beta_t.\mathbf{task} & \left.\begin{array}{l}\\\end{array}\right\} \text{nested part} \\
& * \beta_l.\mathbf{tree}(\alpha) & \left.\begin{array}{l}\\\end{array}\right\} \text{recursive} \\
& * \beta_r.\mathbf{tree}(\alpha) & \left.\begin{array}{l}\\\end{array}\right\} \text{part} \\
& \wedge \alpha \neq \texttt{0x0} &
\end{array}$$

Figure 2.10: Inductive predicates for WFS

The first rule states that if a symbolic variable is equal to the null pointer, then it is the address of well-formed singly linked list. In that case, the memory summarized in the predicate call is empty. The second rule corresponds to the scenario where there is at least one cell in the list, at address α. The `next` field of the cell contains a value β_n that corresponds to the address of the remaining of the list. The second field `data` contains a value β_d. In the second rule, the cell part contains the two points-to predicates. The predicate call $\beta_n.\mathbf{list}$ forms the recursive part. The nested part is empty.

The separating conjunction ensures that the list nodes summarized in the recursive call $\beta_n.\mathbf{list}$ are distinct from the node at address α. Therefore, this definition rules out ill-formed lists such as:



Thanks to the **list** predicate, the abstract state from Figure 2.8b, can be summarized further as $\underline{\texttt{x}} \mapsto \alpha * \alpha.\mathbf{list}$.

Example 2.6: Binary tree inductive predicate

Figure 2.10, depicts the inductive predicates used to summarize the binary tree of WFS.

The first one, $\alpha.\mathbf{task}$, simply states that α is the non-null address of a cell with two fields.

The $\alpha.\mathbf{tree}(\kappa_p)$ predicate has two rules. The first one corresponds to the empty case, similar to the singly linked list predicate. The second rule describes a cell of type `struct node`. The `task` field contains a pointer to a well-formed TCB, summarized thanks to a **task** predicate call. The `left` and `right` fields contain pointers to well-formed binary trees, expressed by recursive calls to **tree**. The `parent` field contains the value of the parameter κ_p that denotes the address of the node's parent. Therefore, in the recursive calls this parameter is set to the address of the current node, α.

Finally, the binary tree of Figure 1.1 (page 13) can be summarized as $\alpha_r.\mathbf{tree}(\alpha_r)$, where α_r is the symbolic variable denoting the address of the root of the tree. Setting the parent parameter to be the main parameter ensures that the root is its own parent.

Segment predicates The inductive predicates presented so far depict complete data structures. So, they cannot represent the situation where a pointer points to a sub-part of a data structure summarized by a predicate. To address this problem, we introduce *segment predicates*. In essence, for a given inductive predicate **pred**, a segment predicate $\alpha.\mathbf{pred} \mathrel{-\!\!*} \beta.\mathbf{pred}$, represents an inductively defined partial data structure, between addresses α and β. That is to say, if we add a call to inductive predicate **pred** at address β, we obtain a full data structure at address α. This is expressed by the following *concatenation principle*:

$$\gamma_m\left(\left(\alpha.\mathbf{pred}(\overrightarrow{\kappa}) \mathrel{-\!\!*} \beta.\mathbf{pred}(\overrightarrow{\delta})\right) * \beta.\mathbf{pred}(\overrightarrow{\delta})\right) \subseteq \gamma_m(\alpha.\mathbf{pred}(\overrightarrow{\kappa}))$$

An inductive predicate that is not an instance of a segment predicate, such as the **list**, **task**, **tree**, is called a *full predicate*. Given a full inductive predicate, its segment counterpart is derived automatically from its definition. Additionally, since a segment predicate is an instance of an inductive predicate, the satisfiability relation and the concretization function remain unchanged.

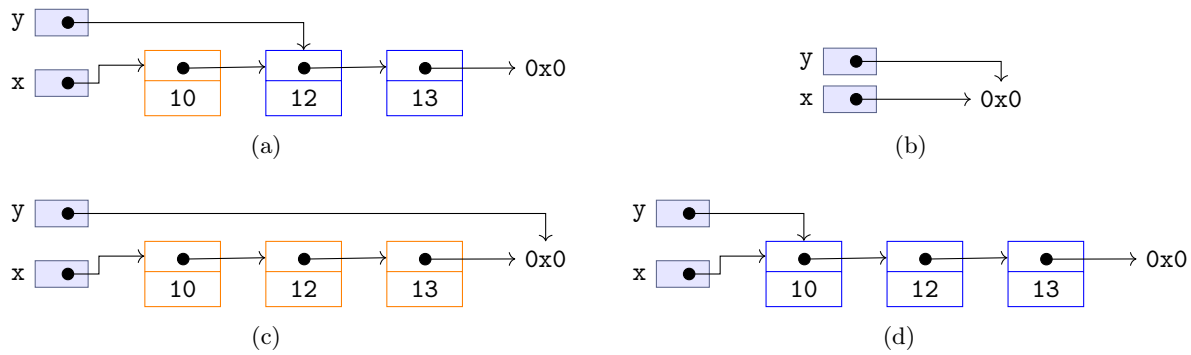


Figure 2.11: Four examples of memory states corresponding to an abstract state with a list segment and a full list

Remark 2.3: Comparison between segments and magic wand

To denote partial memory, separation logic uses the separating implication $-\!\!*$, introduced by Ishtiaq et O'Hearn [IO01]. Intuitively, a separating implication $m^\sharp -\!\!* m^{\sharp\prime}$ denotes any memory state such that if we add a disjoint memory abstracted by $m^\sharp$, the result is a memory abstracted by $m^{\sharp\prime}$. That is to say:

$$\sigma_n, m \vDash_m m^\sharp -\!\!* m^{\sharp\prime} \qquad \text{iff for any heap } m' \text{ if } \sigma_n, m' \vDash_m m^\sharp \text{ then } \sigma_n, m \uplus m' \vDash_m m^{\sharp\prime}$$

This definition is written so as to be able to simplify separation logic formula using the separation logic equivalent of the *modus ponens* principle. However, this definition does not give any insight regarding the content of the memory abstracted by $m^\sharp -\!\!* m^{\sharp\prime}$. Therefore, we restrict the expression of partial memories to segment predicates since they can themselves be expressed as inductive predicates.

Example 2.7: Singly linked list segment

The segment counter-part of the **list** predicate is defined as :

$$\begin{aligned}\alpha.\mathbf{list} \mathrel{-\!\!*} \alpha'.\mathbf{list} := {} & \mathbf{emp} \wedge \alpha = \alpha' \\ & \vee\ \exists \beta_n, \beta_d, \alpha.\mathtt{next} \mapsto \beta_n * \alpha.\mathtt{data} \mapsto \beta_d * \beta_n.\mathbf{list} \mathrel{-\!\!*} \alpha'.\mathbf{list} \wedge \alpha \neq \mathtt{0x0}\end{aligned}$$

The first rule corresponds to the scenario where the segment is empty: its extremities α and α' are equal. The second rule states that there is at least one node summarized in the segment. This rule is similar to the non-empty case of the **list** predicate, except that the recursive call β_n.**list** is replaced by a call to the segment predicate. The end of segment remains α'.

Figure 2.11 presents memory states that are possible concretization of the abstract memory state $\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta * \alpha.\mathbf{list} \mathrel{-\!\!*} \beta.\mathbf{list} * \beta.\mathbf{list}$. In each state, nodes that are summarized in the segment are drawn in orange, whereas nodes that correspond to the full predicate are in blue.

Example 2.8: Binary tree segment predicate

Figure 2.12 presents the definition of the partial binary tree predicate. Similarly to the list segment, the first rule corresponds to the empty segment. In this case, the two extreme addresses α ad α' are equal as well as the parameters corresponding to the parent field. The second and third rule are obtained by replacing recursive calls by a call to the segment predicate. In the second rule, the end of the segment α' is located in the left subtree. Whereas, in the third rule, the end is in the right subtree.

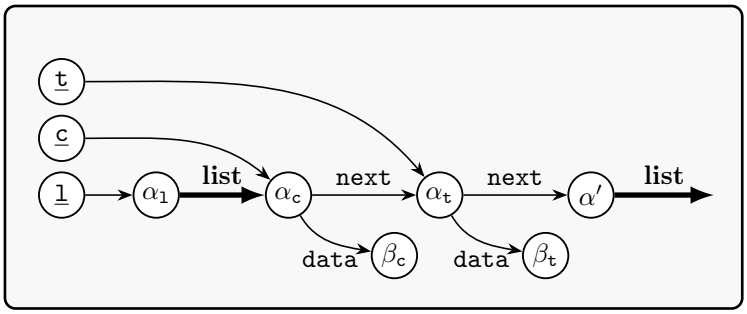
Given that the **task** predicate has no recursive call, defining a segment predicate makes no sense.

$$
\begin{aligned}
\alpha.\mathbf{tree}(\kappa_p) \mathrel{-\!\!*} \alpha'.\mathbf{tree}(\kappa'_p) := {} & \mathbf{emp} \wedge \alpha = \alpha' \wedge \kappa_p = \kappa'_p \\
& \vee \exists \beta_t, \beta_l, \beta_r, \quad \alpha.\mathtt{task} \mapsto \beta_t * \alpha.\mathtt{parent} \mapsto \kappa_p \\
& \qquad * \alpha.\mathtt{left} \mapsto \beta_l * \alpha.\mathtt{right} \mapsto \beta_r \\
& \qquad * \beta_t.\mathbf{task} \\
& \qquad * \beta_l.\mathbf{tree}(\alpha) \mathrel{-\!\!*} \alpha'.\mathbf{tree}(\kappa'_p) * \beta_r.\mathbf{tree}(\alpha) \\
& \qquad \wedge \alpha \neq \mathtt{0x0} \\
& \vee \exists \beta_t, \beta_l, \beta_r, \quad \alpha.\mathtt{task} \mapsto \beta_t * \alpha.\mathtt{parent} \mapsto \kappa_p \\
& \qquad * \alpha.\mathtt{left} \mapsto \beta_l * \alpha.\mathtt{right} \mapsto \beta_r \\
& \qquad * \beta_t.\mathbf{task} \\
& \qquad * \beta_l.\mathbf{tree}(\alpha) * \beta_r.\mathbf{tree}(\alpha) \mathrel{-\!\!*} \alpha'.\mathbf{tree}(\kappa'_p) \\
& \qquad \wedge \alpha \neq \mathtt{0x0}
\end{aligned}
$$

Figure 2.12: Binary tree segment predicate

$$
\begin{aligned}
& \underline{\mathtt{l}} \mapsto \alpha_{\mathtt{l}} * \underline{\mathtt{c}} \mapsto \alpha_{\mathtt{c}} * \underline{\mathtt{t}} \mapsto \alpha_{\mathtt{t}} \\
& * \alpha_{\mathtt{l}}.\mathbf{list} \mathrel{-\!\!*} \alpha_{\mathtt{c}}.\mathbf{list} \\
& * \alpha_{\mathtt{c}}.\mathtt{next} \mapsto \alpha_{\mathtt{t}} * \alpha_{\mathtt{c}}.\mathtt{data} \mapsto \beta_{\mathtt{c}} \\
& * \alpha_{\mathtt{t}}.\mathtt{next} \mapsto \alpha' * \alpha_{\mathtt{t}}.\mathtt{data} \mapsto \beta_{\mathtt{t}} \\
& * \alpha'.\mathbf{list}
\end{aligned}
$$

(a) Abstract memory state



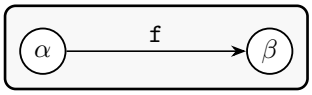
(b) Graph representation of an abstract state

Figure 2.13: An abstract memory state and its graph representation

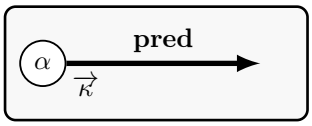
2.3.3 Representing abstract memories with graphs

The abstract memory domain does not encode abstract values as a set of separation logic predicates grouped with the separating conjunction. Instead, the domain manipulates *abstract memory graphs*. An abstract memory graph is a directed graph where vertices are symbolic variables, and each edge represents a specific separation logic predicate.

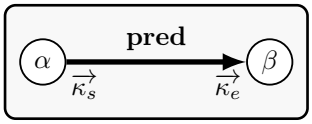
Points-to predicates A points-to predicate $\alpha.\mathtt{f} \mapsto \beta$ is represented by a simple edge from α to β. This edge is labeled with the offset `f`.



Full inductive predicates An inductive predicate $\alpha.\mathbf{pred}(\overrightarrow{\kappa})$ is depicted as a thick edge originating from α. This edge is labeled by the inductive predicate **pred**, and the arguments $\overrightarrow{\kappa}$ are shown at the start of the edge.



Segment predicates To represent instances of segment predicates $\alpha.\mathbf{pred}(\overrightarrow{\kappa_s}) \mathrel{-\!\!*} \beta.\mathbf{pred}(\overrightarrow{\kappa_e})$, the domain uses a thick edge from α to β. This edge is labeled by the inductive predicate used **pred**. Additionally, the arguments κ_s and κ_e are displayed respectively at the source and at the tail of the edge.



Example 2.9: Graph representation of a memory state

Figure 2.13 depicts an abstract memory state and its corresponding graph representation. This state corresponds to the exploration of a list, starting at an address stored in `l`. The two variables used as cursors for the list traversal, `c` and `t`, are pointers to consecutive cells of the list.

2.3.4 Combining the shape domain with a numerical domain

The shape domain defined above has a major drawback: it cannot express numeric constraints over the values of symbolic variables. For example, the abstract state of Figure 2.8b, does not exactly

correspond to the memory states containing a list of length 3. Indeed, it can be concretized into a singly linked list where the `next` field of the last element is not the null pointer. To address this issue, we combine the memory abstract domain $\mathbb{M}^\sharp$ together with a numerical domain.

Signature of the numerical abstract domain In the following, we assume a numerical abstract domain $\mathbb{D}_n^\sharp$ and a concretization function $\gamma_n : \mathbb{D}_n^\sharp \to \wp(\mathcal{V} \to \mathbb{V})$. Additionally, we assume that the abstract domain provides the following operators:

- $\mathbf{assign}_n^\sharp : \mathcal{V} \times \mathcal{E}_\mathcal{V} \times \mathbb{D}_n^\sharp \to \mathbb{D}_n^\sharp$
- $\mathbf{guard}_n^\sharp : \mathcal{B}_\mathcal{V} \times \mathbb{D}_n^\sharp \to \mathbb{D}_n^\sharp$
- $\sqcup_n^\sharp : \mathbb{D}_n^\sharp \times \mathbb{D}_n^\sharp \to \mathbb{D}_n^\sharp$
- $\nabla_n^\sharp : \mathbb{D}_n^\sharp \times \mathbb{D}_n^\sharp \to \mathbb{D}_n^\sharp$
- $\sqsubseteq_n^\sharp : \mathbb{D}_n^\sharp \times \mathbb{D}_n^\sharp \to \{\mathbf{true}, \mathbf{false}\}$
- $\top_n^\sharp : \mathbb{D}_n^\sharp$
- $\bot_n^\sharp : \mathbb{D}_n^\sharp$
- $\mathbf{sat}_n^\sharp : \mathcal{B}_\mathcal{V} \times \mathbb{D}_n^\sharp \to \{\mathbf{true}, \mathbf{false}\}$
- $\mathbf{supp}_n^\sharp : \mathbb{D}_n^\sharp \to \wp_{fin}(\mathcal{V})$
- $\mathbf{prune}_n^\sharp : \mathbb{D}_n^\sharp \times \mathcal{V} \to \mathbb{D}_n^\sharp$

Note that these operators do not manipulate the numeric and boolean expressions of the `MemImp` language, but rather the expressions using symbolic variables defined in Figures 2.9b and 2.9c. Additionally, we assume that these operators meet the soundness properties defined in section 2.2.3.

The last four elements of the list have not been introduced so far. The $\bot_n^\sharp$ element abstracts the empty set of numerical valuations, *i.e.* $\gamma_n(\bot_n^\sharp) = \varnothing$. The $\mathbf{sat}_n^\sharp$ operator inputs a symbolic numerical constraint $b_\mathcal{V} \in \mathcal{B}_\mathcal{V}$ and a numerical abstract value $\sigma_n^\sharp \in \mathbb{D}_n^\sharp$ and returns **true** if all symbolic valuations σ_n abstracted by $\sigma_n^\sharp$ satisfy the constraint $b_\mathcal{V}$, *i.e.*:

$$\mathbf{sat}_n^\sharp(b_\mathcal{V}, \sigma_n^\sharp) = \mathbf{true} \implies \forall \sigma_n \in \gamma_n(\sigma_n^\sharp), \sigma_n \vDash_n b_\mathcal{V}$$

The $\mathbf{supp}_n^\sharp$ operator computes for each numerical abstract value $\sigma_n^\sharp$ the finite set of symbolic variables it constrains. That is to say, if two numerical valuations coincide on $\mathbf{supp}_n^\sharp(\sigma_n^\sharp)$, then either both are abstracted by $\sigma_n^\sharp$ or neither are.

$$\forall \sigma_n, \sigma'_n \in \mathcal{V} \to \mathbb{V}, \left(\forall \alpha \in \mathbf{supp}_n^\sharp(\sigma_n^\sharp), \sigma_n(\alpha) = \sigma'_n(\alpha)\right) \implies \left(\sigma_n \in \gamma_n(\sigma_n^\sharp) \Leftrightarrow \sigma'_n \in \gamma_n(\sigma_n^\sharp)\right)$$

Finally, the $\mathbf{prune}_n^\sharp$ operator inputs an abstract numerical value $\sigma_n^\sharp$ and a symbolic variable α and rewrites the abstract state so that α is no longer constrained.

$$\begin{aligned} &\mathbf{supp}_n^\sharp(\mathbf{prune}_n^\sharp(\sigma_n^\sharp, \alpha)) = \mathbf{supp}_n^\sharp(\sigma_n^\sharp) \setminus \{\alpha\} \\ \wedge\ &\gamma_n\left(\sigma_n^\sharp\right) \subseteq \gamma_n\left(\mathbf{prune}_n^\sharp(\sigma_n^\sharp, \alpha)\right) \end{aligned}$$

Examples of such numerical domains are the interval domain presented before, the octagon domain [Min01], or the polyhedra abstract domain [CH78].

In the remaining of this thesis, we will employ a numerical domain consisting of linear inequalities represented using an element of the polyhedra domain, combined with a finite set disequalities amongst symbolic variables and constant values, *i.e.* $\alpha \neq \beta$ or $\alpha \neq c$.

Reduced product To combine the numerical abstract domain together with the memory domain we employ a reduced product [CC79]. This means that elements of our abstract domain are either a pair of elements from the memory and the numerical domains, or a specific value $\top_\mathbb{S}^\sharp$, expressing that the domain has no information on the possible states, or another specific value $\bot_\mathbb{S}^\sharp$ denoting the empty set of program states. Additionally, the product is $\bot$-coalescent. That is to say, if an abstract value is a pair where the numerical value is $\bot_n^\sharp$, then the whole abstract value must be transformed to $\bot_\mathbb{S}^\sharp$.

Definition 2.13: Combined abstract domain

The combined abstract domain is defined as $\mathbb{S}^\sharp := \left\{\top^\sharp_{\mathbb{S}}, \bot^\sharp_{\mathbb{S}}\right\} \uplus \left(\mathbb{M}^\sharp \times \mathbb{D}^\sharp_n \setminus \{\bot^\sharp_n\}\right)$. Additionally, the concretization $\gamma_{\mathbb{S}} : \mathbb{S}^\sharp \to \wp(\mathbb{S}_\Omega)$ is defined as :

$$\begin{aligned}\gamma_{\mathbb{S}}\left(\top^\sharp_{\mathbb{S}}\right) &:= \mathbb{S}_\Omega\\ \gamma_{\mathbb{S}}\left(\bot^\sharp_{\mathbb{S}}\right) &:= \varnothing\\ \gamma_{\mathbb{S}}\left((m^\sharp, \sigma^\sharp_n)\right) &:= \left\{(\rho, m) \mid \exists \sigma_n, ((\rho, m), \sigma_n) \in \gamma_m(m^\sharp) \wedge \sigma_n \in \gamma_n(\sigma^\sharp_n)\right\}\end{aligned}$$

Disjunctive abstract domain In order to perform case split, the analysis manipulates a finite disjunction of elements of the combined abstract domain $\bigvee_i s^\sharp_i$. This forms a disjunctive abstract domain: $\mathbb{D}^\sharp_d := \left(\mathbb{S}^\sharp\right)^*$. The concretization of a disjunctive abstract state is simply the union of the concretization of its elements.

$$\gamma_d\left(\bigvee_i s^\sharp_i\right) := \bigcup_i \gamma_{\mathbb{S}}\left(s^\sharp_i\right)$$

If one of the elements of the disjunction is $\top^\sharp_{\mathbb{S}}$, then the whole disjunction is rewritten into $\top^\sharp_{\mathbb{S}}$. If one of the elements is $\bot^\sharp_{\mathbb{S}}$, then it is removed from the disjunction.

2.4 Abstract Semantics

This section presents the abstract semantics for the abstract domain introduced in the previous section. To shorten the presentation, we will only define the abstract operators on simple abstract values of $\mathbb{S}^\sharp \setminus \{\top^\sharp_{\mathbb{S}}, \bot^\sharp_{\mathbb{S}}\}$. These operators can easily be extended to extreme abstract values (*i.e.* $\top^\sharp_{\mathbb{S}}$ and $\bot^\sharp_{\mathbb{S}}$), since they are strict, as well as to finite disjunctions of abstract values by lifting them pointwise.

Similarly to Section 2.2, this section presents an existing analysis. Consequently, we omit the proofs of soundness results, and we refer the reader to the original presentation of this analysis [CR08] for the proofs.

2.4.1 Evaluation of expressions

Figure 2.14 presents the abstract evaluation of left values and numeric expressions. Since the syntax of numerical expressions and left-hand values depend on each other, their abstract evaluations are mutually recursive functions.

2.4.1.1 Left-values

For a left-value l, its abstract evaluation is a function $\mathbb{L}[\![l]\!]^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to (\mathbb{S}^\sharp \times \mathcal{V} \times \mathbb{V}) \uplus \{\top\}$ that inputs an abstract state $s^\sharp$, and returns either $\top$ if the evaluation encountered a failure, or a tuple $(s^{\sharp\prime}, \alpha, c)$, such that the value of $\alpha + c$ in $s^{\sharp\prime}$ corresponds to the address of l in $s^\sharp$. The symbolic variable α expresses the address of the first element of the cell, and the constant value c denotes the offset of the address l in that cell.

When the left-value is a variable $\mathtt{x}$, the symbolic variable denoting to its address $\underline{\mathtt{x}}$ is returned and the corresponding offset is 0. Additionally, no modification is performed on the input abstract value $s^\sharp$. If the left-value corresponds to a pointer dereference $*e$, then its abstract evaluation is equal to the abstract evaluation of the expression, $\mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}}$.

2.4.1.2 Expressions

Likewise, for an expression e, its evaluation is a function $\mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to (\mathbb{S}^\sharp \times \mathcal{V} \times \mathbb{V}) \uplus \{\top\}$ that returns either $\top$ when a possible error is detected, or a tuple $(s^{\sharp\prime}, \alpha, c)$. In that case, the value of $\alpha + c$ in $s^{\sharp\prime}$ is equal to the value of e in $s^\sharp$.

For example, when the expression is a constant c, then the evaluation simply adds a novel symbolic variable $\alpha^\dagger$ to the numerical part of the abstract value together with the equality constraint that sets the value of $\alpha^\dagger$ to c. In that scenario the offset is set to 0. When the expression is the address of the left value $\&\, l$, then its abstract evaluation boils down to the abstract evaluation of the left value l.

$$
\begin{aligned}
\mathbb{L}[\![\ \bullet\ ]\!]^\sharp_{\mathbb{S}} &: \langle \mathsf{l\text{-}value} \rangle \times \mathbb{S}^\sharp \to \left(\mathbb{S}^\sharp \times \mathcal{V} \times \mathbb{V}\right) \uplus \{\top\} \\
\mathbb{L}[\![\mathtt{x}]\!]^\sharp_{\mathbb{S}}(s^\sharp) &= (s^\sharp, \underline{\mathtt{x}}, 0) \\
\mathbb{L}[\![*e]\!]^\sharp_{\mathbb{S}}(s^\sharp) &= \mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}}(s^\sharp)
\end{aligned}
$$

(a) Abstract evaluation of left-values

$$
\begin{aligned}
\mathbb{E}[\![\ \bullet\ ]\!]^\sharp_{\mathbb{S}} &: \langle \mathsf{expr} \rangle \times \mathbb{S}^\sharp \to \left(\mathbb{S}^\sharp \times \mathcal{V} \times \mathbb{V}\right) \uplus \{\top\} \\
\mathbb{E}[\![c]\!]^\sharp_{\mathbb{S}}((m^\sharp, \sigma^\sharp_n)) &= \left((m^\sharp, \mathbf{guard}^\sharp_n(\sigma^\sharp_n, \alpha^\dagger = c)), \alpha^\dagger, 0\right) \qquad \alpha^\dagger \text{ fresh} \\
\mathbb{E}[\![\,\&\, l]\!]^\sharp_{\mathbb{S}}(s^\sharp) &= \mathbb{L}[\![l]\!]^\sharp_{\mathbb{S}}(s^\sharp) \\
\mathbb{E}[\![l]\!]^\sharp_{\mathbb{S}}(s^\sharp) &= \begin{cases} \mathbf{read}^\sharp_{\mathbb{S}}(s^{\sharp\prime}, \alpha, c) & \text{if } \mathbb{L}[\![l]\!]^\sharp_{\mathbb{S}}(s^\sharp) = (s^{\sharp\prime}, \alpha, c) \\ \top & \text{otherwise} \end{cases} \\
\mathbb{E}[\![e \pm c]\!]^\sharp_{\mathbb{S}}(s^\sharp) &= \begin{cases} (s^{\sharp\prime}, \beta, c' \pm c) & \text{if } \mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}}(s^\sharp) = (s^{\sharp\prime}, \beta, c') \\ \top & \text{otherwise} \end{cases} \\
\mathbb{E}[\![e \oplus e']\!]^\sharp_{\mathbb{S}}(s^\sharp) &= \begin{cases} \left((m^{\sharp\prime\prime}, \sigma^{\sharp\prime\prime\prime}_n), \alpha^\dagger, 0\right) & \text{if } \begin{array}{l} \mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}}(s^\sharp) = (s^{\sharp\prime}, \alpha, c) \\ \wedge\, \mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}}(s^{\sharp\prime}) = (s^{\sharp\prime\prime}, \alpha', c') \\ \wedge \oplus \in \{/, \%\} \Rightarrow \mathbf{sat}^\sharp_n(\sigma^{\sharp\prime\prime}_n, \alpha' + c' \neq 0) = \mathbf{true} \\ \wedge\, \sigma^{\sharp\prime\prime\prime}_n = \mathbf{guard}^\sharp_n(\sigma^{\sharp\prime\prime}_n, \alpha^\dagger = (\alpha + c) \oplus (\alpha' + c')) \\ \wedge\, \alpha^\dagger \text{ fresh} \end{array} \\ \top & \text{otherwise} \end{cases}
\end{aligned}
$$

(b) Abstract evaluation of expressions

Figure 2.14: Abstract semantics of expressions

The most interesting case concerns expressions involving a memory reading at an address denoted by a left-value l. First, the left-value l is evaluated. If the outcome is a triple $(s^{\sharp\prime}, \alpha, c)$, and if there exists in the memory part of $s^{\sharp\prime}$ a points-to predicate $\alpha.\mathtt{f} \mapsto \beta$, for some field $\mathtt{f}$ with a value matching c [4], then the destination of this points-to predicate, β, is returned. The abstract value is $s^{\sharp\prime}$, and the offset is 0. The last step is performed by the abstract memory reading operator $\mathbf{read}^\sharp_{\mathbb{S}}$:

$$
\begin{aligned}
\mathbf{read}^\sharp_{\mathbb{S}} &: \mathcal{V} \times \mathbb{V} \times \mathbb{S}^\sharp \to \left(\mathbb{S}^\sharp \times \mathcal{V} \times \mathbb{V}\right) \uplus \{\top\} \\
\mathbf{read}^\sharp_{\mathbb{S}}\left(\alpha, c, (m^\sharp, s^\sharp)\right) &= \begin{cases} \left((m^\sharp, s^\sharp), \beta, 0\right) & \text{if } \begin{array}{l} m^\sharp = \alpha.\mathtt{f} \mapsto \beta * m^{\sharp\prime} \\ \wedge\, \varphi_{\mathbb{F}}(\mathtt{f}) = c \end{array} \\ \top & \text{otherwise} \end{cases}
\end{aligned}
$$

In other scenarios, the evaluation fails.
For expressions involving an operation, the evaluation distinguishes two cases.

- The first case corresponds to instances of addition or subtraction by a constant value c. The outcome of the evaluation is determined by either adding or subtracting the constant c to the offset of the evaluation of the left expression.
- The second case covers all remaining scenarios. The two sub-expressions are evaluated sequentially. If the operator is a division or a modulo, the evaluation ensures that the right expression cannot be null by invoking the $\mathbf{sat}^\sharp_n$ operator. Ultimately, the evaluation introduces a fresh symbolic variable $\alpha^\dagger$ into the numeric part of the abstract value. Here, the adjective *fresh* means that $\alpha^\dagger$ is not in the support of the abstract state. This variable is accompanied by a constraint indicating that it is equal to the result of the operation between the evaluations of the two sub-expressions.

[4] Recall that the value of a field $\mathtt{f}$ is computed using a function $\varphi_{\mathbb{F}} : \mathbb{F} \to \mathbb{V}$, introduced in Remark 2.1.

Remark 2.4: Constraining a fresh variable

In the general case, evaluating an expression containing a numerical operation introduces a fresh variable, and adds a constraint on this variable. This may seem unsound at first glance since the concretization of the numerical abstract value is strictly smaller after the addition of the constraint $\alpha^\dagger = (\alpha + c) \oplus (\alpha' + c')$. However, recall that in the concretization $\gamma_{\mathbb{S}}$ (Definition 2.13), numerical valuations are existentially quantified.

For instance, let (ρ, m) be a memory state in $(m^\sharp, \sigma_n^\sharp)$. This means that there exists a numerical valuation $\sigma_n \in \mathcal{V} \to \mathbb{V}$, such that $\sigma_n \in \gamma_n(\sigma_n^\sharp)$, and $(\sigma_n, m) \vDash_m m^\sharp$. Now let us define the numerical valuation $\sigma'_n := \sigma_n[\alpha^\dagger \mapsto \mathbb{E}[\![(\alpha + c) \oplus (\alpha' + c')]\!]_{\mathcal{V}}(\sigma_n)]$. That is to say, the numerical valuation that is equal to σ_n in all inputs excepts $\alpha^\dagger$, where it returns the result of the evaluation of the expression $(\alpha + c) \oplus (\alpha' + c')$ in σ_n. This implies that the restrictions of σ_n and σ'_n to the support of the abstract value $(m^\sharp, \sigma_n^\sharp)$ are equal, since $\alpha^\dagger$ is picked outside the support. By soundness of the support, we deduce that $m, \sigma'_n \vDash_m m^\sharp$ and $\sigma'_n \in \gamma_n(\sigma_n^\sharp)$. Additionally, by definition, the new valuation satisfies the constraint on $\alpha^\dagger$. Therefore, by soundness of $\mathbf{guard}_n^\sharp$, we infer that $\sigma'_n \in \gamma_n(\mathbf{guard}_n^\sharp(\sigma_n^\sharp, \alpha^\dagger = (\alpha + c) \oplus (\alpha' + c')))$. Finally, since $m, \sigma'_n \vDash_m m^\sharp$, we conclude that:

$$m \in \gamma_{\mathbb{S}}\left(m^\sharp, \mathbf{guard}_n^\sharp\left(\sigma_n^\sharp, \alpha^\dagger = (\alpha + c) \oplus (\alpha' + c')\right)\right)$$

This means that the introduction of $\alpha^\dagger$ and the addition of the constraint $\alpha^\dagger = (\alpha + c) \oplus (\alpha' + c')$ do not modify the concretization of the updated abstract value.

It is important to note that this soundness argument holds as long as the evaluation of the expression in σ_n has a result. If the evaluation of the expression $(\alpha + c) \oplus (\alpha' + c')$ performs a division by zero, then we cannot find a possible value for $\alpha^\dagger$ in σ'_n.

Example 2.10: Abstract evaluation of expression

To illustrate the abstract expression evaluation, let us consider the weighted service time update from Listing 1.1 (page 11) at line 16 in the abstract state $(m^\sharp, \alpha \neq \texttt{0x0})$ where:

$$m^\sharp := \underline{\texttt{task}} \mapsto \alpha * \alpha.\texttt{wst} \mapsto \beta_t * \alpha.\texttt{weight} \mapsto \beta_w$$

The expression `task->wst + task->weight` corresponds in `MemImp` to `*task + *(task + 1)`. The evaluation starts by the left sub-expression `*task`. It is a left-value: we first evaluate the corresponding address. But since the left-value is a pointer dereference, its address is equal to the value of the expression `task`. This expression is also a left value, so we must perform a second memory reading at the address corresponding to symbolic variable $\underline{\texttt{task}}$. To sum up, the evaluation of `*task` corresponds to the following computing steps:

$$\begin{aligned}
\mathbb{E}[\![\texttt{*task}]\!]_{\mathbb{S}}^\sharp(s^\sharp) &= \mathbf{read}_{\mathbb{S}}^\sharp \circ \mathbb{L}[\![\texttt{*task}]\!]_n^\sharp(s^\sharp) \\
&= \mathbf{read}_{\mathbb{S}}^\sharp \circ \mathbb{E}[\![\texttt{task}]\!]_n^\sharp(s^\sharp) \\
&= \mathbf{read}_{\mathbb{S}}^\sharp \circ \mathbf{read}_{\mathbb{S}}^\sharp \circ \mathbb{L}[\![\texttt{task}]\!]_n^\sharp(s^\sharp) \\
&= \mathbf{read}_{\mathbb{S}}^\sharp \circ \mathbf{read}_{\mathbb{S}}^\sharp(s^\sharp, \underline{\texttt{task}}, 0) \\
&= \mathbf{read}_{\mathbb{S}}^\sharp(s^\sharp, \alpha, 0) \\
&= (s^\sharp, \beta_t, 0)
\end{aligned}$$

The evaluation of the right sub-expression follows the same principle. However, since the address of the dereferenced pointer is $*(\texttt{task} + 1)$, the offset of the outcome of the first memory read is incremented by 1. Therefore, the second memory reading is $\mathbf{read}_{\mathbb{S}}^\sharp(s^\sharp, \alpha, 1)$ and yields $(s^\sharp, \beta_w, 0)$.

Since the expression is the sum of non-constant expressions, the evaluation picks a fresh symbolic variable β to denote the result. This variable is constrained to be equal to the sum of β_t and β_w in the numerical part of the returned abstract value. The outcome of the evaluation is: $((m^\sharp, \alpha \neq \texttt{0x0} \wedge \beta = \beta_t + \beta_w), \beta, 0)$

2.4.1.3 Unfolding

The memory reading operator, $\mathbf{read}_{\mathbb{S}}^\sharp$, has a major limitation: It fails when the address to read is not explicitly present in the abstract memory as the source of a points-to predicate. This means that

$\mathbf{read}^\sharp_{\mathbb{S}}$ does not work when applied to memory cells that are abstracted in an inductive predicate. For example $\mathbf{read}^\sharp_{\mathbb{S}}(\alpha, 0, (\underline{\mathtt{x}} \mapsto \alpha * \alpha.\mathbf{list}, \alpha \neq 0))$ fails. However, the constraint $\alpha \neq 0$ ensures that the list at address α is not empty: there exists a memory cell at address α.

To support these cases, the analysis unfolds the inductive predicate. The abstract value is rewritten into a disjunction of states. For each rule in the definition of the predicate, we add an element in the disjunction. The inductive call in the abstract memory is replaced by the shape part of the rule. Additionally, the numeric part of the abstract state is refined by assuming the pure part of the definition.

Definition 2.14: Predicate unfolding

For a symbolic value α, an abstract state $s^\sharp = (m^\sharp * \alpha.\mathbf{pred}(\overrightarrow{\kappa}), \sigma^\sharp_n)$, where α is the source of an inductive predicate $\mathbf{pred}(\overrightarrow{\kappa}) := \bigvee_i \exists \overrightarrow{\beta_i}, m^\sharp_i \wedge \varphi_i$ the predicate unfolding operator is defined as:

$$\mathbf{unfold}^\sharp_{\mathbb{S}}(\alpha, s^\sharp) := \bigvee_i \left(m^\sharp * m^\sharp_i[\overrightarrow{\delta}/\overrightarrow{\beta_i}], \mathbf{guard}^\sharp_n(\sigma^\sharp_n, \varphi_i[\overrightarrow{\delta}/\overrightarrow{\beta_i}]) \right) \qquad \overrightarrow{\delta} \text{ fresh}$$

Example 2.11: Unfolding of a full inductive predicate

Let us consider the case presented in the first paragraph of this subsection:

$$s^\sharp := (\underline{\mathtt{x}} \mapsto \alpha * \alpha.\mathbf{list}, \alpha \neq 0)$$

Since the **list** predicate contains two rules, the unfolding operator considers a disjunction of two abstract states:

$$s^\sharp_1 := (\underline{\mathtt{x}} \mapsto \alpha * \mathbf{emp}, \underbrace{\alpha \neq 0 \wedge \alpha = 0}_{=\bot^\sharp_n})$$

$$s^\sharp_2 := \left(\begin{array}{c} \underline{\mathtt{x}} \mapsto \alpha * \alpha.\mathtt{next} \mapsto \beta * \beta.\mathbf{list}, \alpha \neq 0 \\ * \alpha.\mathtt{next} \mapsto \delta \end{array} \right)$$

The first element $s^\sharp_1$ corresponds to the empty rule, and $s^\sharp_2$ to the rule where there is at least one node in the list. To enhance readability, we display points-to predicates describing one list cell in a single column. In each state, the part added in the abstract value is represented in green. We omit the constraint $\alpha \neq 0$ in $s^\sharp_2$ as it was already present in the numerical part of $s^\sharp$.

In $s^\sharp_1$ the pure part of the **list** predicate is inconsistent with the numerical part of $s^\sharp$. Therefore, $\mathbf{guard}^\sharp_n(\alpha \neq 0, \alpha = 0)$ yields the bottom value $\bot^\sharp_n$. This means that the empty rule is not feasible. As a consequence the whole abstract value $s^\sharp_1$ is rewritten into $\bot^\sharp_{\mathbb{S}}$, and removed from the disjunction. So, in the result of $\mathbf{unfold}^\sharp_{\mathbb{S}}(s^\sharp)$, only $s^\sharp_2$ remains.

Example 2.12: Unfolding of a segment predicate

Let us consider the segment unfolding in the abstract state from Example 2.7:

$$(\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta * \alpha.\mathbf{list} \mathrel{-\!\!*} \beta.\mathbf{list} * \beta.\mathbf{list}, \top^\sharp_n)$$

Applying the segment counter-part of the **list** predicate produces two abstract states:

$$s^\sharp_1 := (\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \alpha * \mathbf{emp} * \alpha.\mathbf{list}, \alpha = \beta)$$

$$s^\sharp_2 := \left(\begin{array}{c} \underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta * \alpha.\mathtt{next} \mapsto \alpha' * \alpha'.\mathbf{list} \mathrel{-\!\!*} \beta.\mathbf{list} * \beta.\mathbf{list}, \alpha \neq \mathtt{0x0} \\ * \alpha.\mathtt{next} \mapsto \delta \end{array} \right)$$

The first case corresponds to the empty rule of the segment definition. Note that the equality constraint $\alpha = \beta$ is used to replace all occurrences of β by α in the memory part of the abstract value. The second abstract value corresponds to the case where there is at least one element in the segment. Hence, the $\mathbf{unfold}^\sharp_{\mathbb{S}}$ operator materializes a list cell at address α, and assumes that this address is non-null in the numerical part.

The rewriting step carried out by the unfolding operator maintains soundness: the disjunction of the output over-approximates the input.

Theorem 2.12: Soundness of unfolding

For any abstract state $s^\sharp$ and symbolic variable α, if $\mathbf{unfold}^\sharp_{\mathbb{S}}(s^\sharp, \alpha) = \bigvee_i s_i^\sharp$, then

$$\gamma_{\mathbb{S}}(s^\sharp) \subseteq \bigcup_i \gamma_{\mathbb{S}}(s_i^\sharp)$$

With the aid of the $\mathbf{unfold}^\sharp_{\mathbb{S}}$ operator, we can now add another case to the memory reading operator, in the case where the content to read is in a cell summarized by an inductive predicate.

$$\mathbf{read}^\sharp_{\mathbb{S}}(\alpha, c, s^\sharp) = \mathbf{read}^\sharp_{\mathbb{S}}\left(\alpha, c, \mathbf{unfold}^\sharp_{\mathbb{S}}(\alpha, s^\sharp)\right) \quad \begin{array}{l}\text{when } \alpha \text{ is the source of} \\ \text{an inductive predicate in } s^\sharp\end{array}$$

Recall that in the memory part of an inductive predicate definition, all recursive predicates cannot originate from the same source as the main predicate call. This entails that the $\mathbf{read}^\sharp_{\mathbb{S}}$ requires at most one predicate unfolding in order to materialize the cell it attempts to read.

2.4.1.4 Soundness

Since the evaluation of an expression could alter the abstract state, its soundness cannot be straightforwardly expressed as with Theorem 2.3. In essence, the soundness of abstract evaluation functions states that if the abstract evaluation process does not fail and returns a disjunction $\bigvee_i(s_i^\sharp, \alpha_i, c_i)$, then for any concrete state abstracted by the input abstract state, there exists an element in the returned disjunction $(s_i^\sharp, \alpha_i, c_i)$, for some index value i. This element satisfies the condition that the result $\alpha_i + c_i$ can be evaluated to the concrete evaluation of the expression in the concrete state, according to some numeric valuation in the concretization of $s_i^\sharp$.

Theorem 2.13: Soundness of $\mathbb{L}[\![\bullet]\!]^\sharp_{\mathbb{S}}$ and $\mathbb{E}[\![\bullet]\!]^\sharp_{\mathbb{S}}$

For any abstract state $s^\sharp \in \mathbb{S}^\sharp$ and any left-value l, if $\mathbb{L}[\![l]\!]^\sharp_{\mathbb{S}}(s^\sharp) = \bigvee_i \left((m_i^{\sharp\prime}, \sigma_{n,i}^{\sharp\prime}), \alpha_i, c_i\right)$, then

$$\forall (\rho, m) \in \gamma_{\mathbb{S}}(s^\sharp), \exists i, \sigma_n \in \mathcal{V} \to \mathbb{V}, \begin{cases} \sigma_n \in \gamma_n(\sigma_{n,i}^{\sharp\prime}) \\ \wedge\, ((\rho, m), \sigma_n) \in \gamma_m(m_i^{\sharp\prime}) \\ \wedge\, \sigma_n(\alpha_i) + c_i = \mathbb{L}[\![l]\!](\rho, m) \end{cases}$$

Similarly, for any expression e, if $\mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}}(s^\sharp) = \bigvee_i \left((m_i^{\sharp\prime}, \sigma_{n,i}^{\sharp\prime}), \alpha_i, c_i\right)$, then

$$\forall (\rho, m) \in \gamma_{\mathbb{S}}(s^\sharp), \exists i, \sigma_n \in \mathcal{V} \to \mathbb{V}, \begin{cases} \sigma_n \in \gamma_n(\sigma_{n,i}^{\sharp\prime}) \\ \wedge\, ((\rho, m), \sigma_n) \in \gamma_m(m_i^{\sharp\prime}) \\ \wedge\, \sigma_n(\alpha_i) + c_i = \mathbb{E}[\![e]\!](\rho, m) \end{cases}$$

The first two conditions of the conjunction implicitly state that any concrete memory state abstracted by $s^\sharp$ is also abstract by some abstract element in the disjunction, *i.e.* the disjunction of the returned abstract states form a sound over-approximation of $\sigma^\sharp$. Additionally, the evaluation of l in any memory state abstracted by the disjunction (and also by $s^\sharp$) does not raise the error state.

2.4.2 Abstract transfer function

The abstract operators $\mathbb{E}[\![\bullet]\!]^\sharp_{\mathbb{S}}$ et $\mathbb{L}[\![\bullet]\!]^\sharp_{\mathbb{S}}$ introduced in the previous section over-approximate the semantics of the evaluation of expressions and left-values. We now present the abstract operators that over-approximate the semantics of statements of the `MemImp` language.

2.4.2.1 Assignment

The assignment operator of the combined abstract domain is presented in Figure 2.15b. In order to perform an assignment $l = e$, the operator first evaluates the address corresponding to the left-value and the value of the expression. Then, it relies on a second operator $\mathbf{write}^\sharp_{\mathbb{S}}$, to perform the actual memory write. The definition of this operator is presented in Figure 2.15a. It is similar to the definition of $\mathbf{read}^\sharp_{\mathbb{S}}$. To write the value β at address α and offset c, the operator checks if α.`f` is the source of some points-to predicate where field `f` matches the value of offset c. In this case, the writing is performed by replacing the destination of the predicate by β. If α is the source of an

$$\mathbf{write}^\sharp_{\mathbb{S}} : \mathcal{V} \times \mathbb{V} \times \mathcal{V} \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp \uplus \{\top\}$$

$$\mathbf{write}^\sharp_{\mathbb{S}}\left(\alpha, c, \beta, (m^\sharp, \sigma^\sharp_n)\right) = \begin{cases} \left((\alpha.\mathtt{f} \mapsto \beta * m^{\sharp\prime}, \sigma^\sharp_n), \beta, 0\right) & \text{if } \begin{array}{r} m^\sharp = \alpha.\mathtt{f} \mapsto \beta' * m^{\sharp\prime} \\ \wedge\, \varphi_{\mathbb{F}}(\mathtt{f}) = c \end{array} \\ \mathbf{write}^\sharp_{\mathbb{S}}\left(\alpha, c, \beta, \mathbf{unfold}^\sharp_{\mathbb{S}}(\alpha, (m^\sharp, \sigma^\sharp_n))\right) & \text{if } m^\sharp = \alpha.\mathbf{pred} * m^{\sharp\prime} \\ \top & \text{otherwise} \end{cases}$$

(a) Memory writing operator $\mathbf{write}^\sharp_{\mathbb{S}}$

$$\mathbf{assign}^\sharp_{\mathbb{S}} : \langle \mathsf{l\text{-}value} \rangle \times \langle \mathsf{expr} \rangle \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp$$

$$\mathbf{assign}^\sharp_{\mathbb{S}}\left(l, e, s^\sharp\right) = \begin{cases} \mathbf{write}^\sharp_{\mathbb{S}}\left(\alpha, c, \beta, (m^{\sharp\prime\prime}, \sigma^{\sharp\prime\prime}_n)\right) & \text{if } \begin{array}{r} (s^{\sharp\prime}, \alpha, c) = \mathbb{L}[\![l]\!]^\sharp_{\mathbb{S}} \\ \wedge\, (s^{\sharp\prime\prime}, \beta, 0) = \mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}} \end{array} \\ \mathbf{write}^\sharp_{\mathbb{S}}\left(\alpha, c, \beta^\dagger, (m^{\sharp\prime\prime}, \sigma^{\sharp\prime\prime\prime}_n)\right) & \text{if } \begin{array}{r} (s^{\sharp\prime}, \alpha, c) = \mathbb{L}[\![l]\!]^\sharp_{\mathbb{S}} \\ \wedge\, (s^{\sharp\prime\prime}, \beta, c') = \mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}} \\ \wedge\, \sigma^{\sharp\prime\prime\prime}_n = \mathbf{guard}^\sharp_n\left(\sigma^{\sharp\prime\prime}_n, \beta^\dagger = \beta + c\right) \end{array} \\ \top^\sharp_{\mathbb{S}} & \text{otherwise} \end{cases}$$

(b) Definition of $\mathbf{assign}^\sharp_{\mathbb{S}}$

Figure 2.15: Abstract transfer functions for assignment

inductive predicate, it must be unfolded before reapplying the $\mathbf{write}^\sharp_{\mathbb{S}}$ operator. In all other scenarios, the writing fails.

Note that since the left-value evaluation function returns a single address for any element of the disjunction, the analysis only performs strong update. Any ambiguity regarding the assigned left-value must be eliminated thanks to the disjunctions.

Example 2.13: Updating the CSt of the current task

To illustrate the $\mathbf{assign}^\sharp_{\mathbb{S}}$ operator, let us consider the assignment $\mathtt{x} = \mathtt{x} + 4 \times \mathtt{y}$ performed on the following abstract state:

$$s^\sharp = (\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta,\ \alpha \geqslant 0)$$

The outcome of the evaluation of the assigned expression is:

$$\mathbb{E}[\![\mathtt{x} + 4 \times \mathtt{y}]\!]^\sharp_{\mathbb{S}}(s^\sharp) = (\alpha'', 0, (\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta,\ \alpha \geqslant 0 \wedge \alpha' = 4 \times \beta \wedge \alpha'' = \alpha + \alpha'))$$

And after performing the memory writing, the operator outputs:

$$(\underline{\mathtt{x}} \mapsto \alpha'' * \underline{\mathtt{y}} \mapsto \beta,\ \alpha \geqslant 0 \wedge \alpha' = 4 \times \beta \wedge \alpha'' = \alpha + \alpha')$$

Abstract state pruning In order to evaluate sub-expressions, the analysis might introduce fresh symbolic variables to represent intermediate results. These symbolic variables extend the support of numeric part of the abstract value, potentially impeding the speed of the analysis. To address this issue, the analysis maintains the following invariant: "For each abstract state, the support of the numeric part must be included in the support of the memory part". Hence, following each analysis step, such as evaluating one statement, the analysis prunes the abstract value. It removes from the numerical part all symbolic variables that are not present in the memory part using the $\mathbf{prune}^\sharp_n$ operator.

Example 2.14: Continuating Example 2.13

In the result of assignment $\mathtt{x} = \mathtt{x} + 4 \times \mathtt{y}$, symbolic variables α and α' are only present in the numerical part. Therefore, they are eliminated. And the analysis simplifies the output abstract state to:

$$(\underline{\mathtt{x}} \mapsto \alpha'' * \underline{\mathtt{y}} \mapsto \beta,\ \alpha'' \geqslant 4 \times \beta)$$

Another case of abstract state pruning arises when some part of the abstract memory is no longer reachable from program variables. Such a case corresponds to a possible memory leak. The analysis reports it and continues after removing the unreachable memory parts and pruning the numerical part accordingly.

Example 2.15: Abstract state pruning

To demonstrate abstract state pruning, let us consider the assignment `x = x->next` in the following abstract state:

$$s^\sharp = (\underline{\mathtt{x}} \mapsto \alpha * \alpha.\mathbf{list},\ \alpha \neq 0)$$

To evaluate the assigned expression, the **list** predicate is unfolded. The empty case is inconsistent with the numeric constraint $\alpha \neq 0$. In the second case, the expression `x->next` boils down to the symbolic variable β that corresponds to the address of the remaining of the list. After the unfolding and the memory writing, the analysis yields:

$$s^{\sharp\prime} = \left(\begin{array}{c} \underline{\mathtt{x}} \mapsto \beta * \alpha.\mathtt{next} \mapsto \beta \\ * \alpha.\mathtt{next} \mapsto \delta \end{array} * \beta.\mathbf{list},\ \alpha \neq 0\right)$$

In this state, α is no longer reachable from $\underline{\mathtt{x}}$. Therefore, the two points-to predicates at address α are removed as well as the constraint $\alpha \neq 0$. To conclude, the outcome of the assignment is:

$$s^{\sharp\prime\prime} = (\underline{\mathtt{x}} \mapsto \beta * \beta.\mathbf{list},\ \top_n^\sharp)$$

Finally, similarly to Theorem 2.4, the $\mathbf{assign}_{\mathbb{S}}^\sharp$ operator is a sound abstract transfer function for the assignment statement.

2.4.2.2 Memory allocation

To perform a dynamic allocation $\mathtt{x} = \mathtt{malloc}(c)$, the analysis considers two cases. Each case forms an element of the returned disjunction. In the first one, no allocation is performed, and the pointer returned by the `malloc` function is the null pointer. It boils down to assigning 0 to the variable program variable `x`. In the second case, the analysis picks fresh variables $\alpha^\dagger, \delta_0^\dagger, \ldots, \delta_{c-1}^\dagger$ to denote the address and the content of the new cell. It adds c points-to predicate to the memory part of the abstract state. Finally, the symbolic variable α is assigned to `x`.

Definition 2.15: Asbtract memory allocation operator $\mathbf{malloc}_{\mathbb{S}}^\sharp$

The abstract dynamic memory allocation operator $\mathbf{malloc}_{\mathbb{S}}^\sharp : \mathbb{X} \times \mathbb{V} \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ is defined as :

$$\begin{aligned} \mathbf{malloc}_{\mathbb{S}}^\sharp\left(\mathtt{x}, c, (m^\sharp, \sigma_n^\sharp)\right) := \mathbf{write}_{\mathbb{S}}^\sharp\left(\underline{\mathtt{x}}, 0, \alpha^\dagger, (m^\sharp, \mathbf{guard}_n^\sharp(\sigma_n^\sharp, \alpha^\dagger = 0))\right) \\ \bigvee \mathbf{write}_{\mathbb{S}}^\sharp\left(\underline{\mathtt{x}}, 0, \alpha^\dagger, (m^{\sharp\prime}, \mathbf{guard}_n^\sharp(\sigma_n^\sharp, \alpha^\dagger \neq 0))\right) \qquad \alpha^\dagger, \delta_i^\dagger \text{ fresh} \\ \text{where } m^{\sharp\prime} := m^\sharp * \left(\circledast_{0 \leqslant i < c}\, \alpha^\dagger.\mathtt{f}_i \mapsto \delta_i^\dagger\right) \end{aligned}$$

The symbol $\circledast$ corresponds to the iterated separating conjunction: $\circledast_{0 \leqslant i < c}\, m_i^\sharp := m_0^\sharp * m_1^\sharp * \ldots * m_{c-1}^\sharp$.

Finally, following the soundness criteria defined in Theorem 2.5, $\mathbf{malloc}_{\mathbb{S}}^\sharp$ is sound.

$$\mathbf{guard}^\sharp_{\mathbb{S}} : \langle \mathsf{bexpr} \rangle \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp$$

$$\mathbf{guard}^\sharp_{\mathbb{S}}\left(e = e', s^\sharp\right) = \begin{cases} \bot^\sharp_{\mathbb{S}} & \begin{array}{l}(s^{\sharp\prime}, \alpha, c) = \mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}} \\ \text{if} \wedge (s^{\sharp\prime\prime}, \beta, c') = \mathbb{E}[\![e']\!]^\sharp_{\mathbb{S}} \\ \quad \wedge m^{\sharp\prime\prime} = \alpha.\mathtt{f}_c \mapsto \delta * \beta.\mathtt{f}_{c'} \mapsto \delta * m^\sharp_0 \end{array} \\ (m^{\sharp\prime\prime\prime}, \sigma^{\sharp\prime\prime\prime}_n) & \begin{array}{l}(s^{\sharp\prime}, \alpha, c) = \mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}} \\ \text{if} \wedge (s^{\sharp\prime\prime}, \beta, c) = \mathbb{E}[\![e']\!]^\sharp_{\mathbb{S}} \\ \quad \wedge \sigma^{\sharp\prime\prime\prime}_n = \mathbf{guard}^\sharp_n\left(\sigma^{\sharp\prime\prime}_n, \alpha + c \bowtie \beta + c\right) \\ \quad \wedge m^{\sharp\prime\prime\prime} = m^{\sharp\prime\prime}\left[\alpha/\beta\right] \end{array} \\ \top^\sharp_{\mathbb{S}} & \text{otherwise} \end{cases}$$

$$\mathbf{guard}^\sharp_{\mathbb{S}}\left(e \bowtie e', s^\sharp\right) = \begin{cases} (m^{\sharp\prime\prime}, \sigma^{\sharp\prime\prime\prime}_n) & \begin{array}{l}(s^{\sharp\prime}, \alpha, c) = \mathbb{E}[\![e]\!]^\sharp_{\mathbb{S}} \\ \text{if} \wedge (s^{\sharp\prime\prime}, \beta, c') = \mathbb{E}[\![e']\!]^\sharp_{\mathbb{S}} \\ \quad \wedge \sigma^{\sharp\prime\prime\prime}_n = \mathbf{guard}^\sharp_n\left(\sigma^{\sharp\prime\prime}_n, \alpha + c \bowtie \beta + c'\right) \end{array} \\ \top^\sharp_{\mathbb{S}} & \text{otherwise} \end{cases}$$

Figure 2.16: Conditional operator $\mathbf{guard}^\sharp_{\mathbb{S}}$

2.4.2.3 Conditional operator

Figure 2.16 presents the $\mathbf{guard}^\sharp_{\mathbb{S}}$ operator. In the general scenario, the operator functions as follows: it evaluates the two expressions and introduces a constraint between the results of these evaluations. If the constraint is an equality, the operator further modifies the memory part. First, if the two sub-expressions correspond to addresses of points-to predicates, then it implies that the separating conjunction no longer holds. The entire abstract value is reduced to $\bot^\sharp_{\mathbb{S}}$. Note that in this case, it is not necessary to check that the offsets computed are equal. The second case arises when the two offsets are identical. In this case, the symbolic variables in the results are equal. Therefore, the memory part undergoes a rewrite where one symbolic variable is replaced by the other.

Ultimately, in accordance with the soundness criteria from Theorem 2.6, $\mathbf{guard}^\sharp_{\mathbb{S}}$ guarantees soundness.

2.4.3 Lattice operators

In the combined abstract domain, the lattice operators (inclusion test, union, widening) follow the same pattern. Initially, the memory components of the input are adjusted to make them fit. Subsequently, the value domain operator is applied to the numerical part of the abstract value.

2.4.3.1 Predicate folding

The unfolding operator introduced previously replaces a single inductive predicate call by its definition. Lattice operators use the converse principle: *predicate folding*. It rewrites several parts of the abstract memory into a single inductive predicate call.

Folding predicate definition If some parts of the abstract memory matches a rule of some inductive predicate definition, then these parts can be grouped into a single inductive predicate. But before doing so, the analysis must enforce that the constraints corresponding to the pure part of the rule are satisfied in the abstract numerical value, using $\mathbf{sat}^\sharp_n$.

Example 2.16: Folding a singly linked list

In the abstract state $\left(\underline{\mathtt{x}} \mapsto \alpha * \alpha.\mathtt{next} \mapsto \beta * \alpha.\mathtt{data} \mapsto \delta * \beta.\mathbf{list}, \sigma^\sharp_n\right)$, the parts of the memory in blue match the non-empty rule of the **list** predicate. In this rule, the pure part is a constraint stating that α is non-null. So if $\mathbf{sat}^\sharp_n(\alpha \neq 0, \sigma^\sharp_n) = \mathbf{true}$, then the abstract state can be folded into $\left(\underline{\mathtt{x}} \mapsto \alpha * \alpha.\mathbf{list}, \sigma^\sharp_n\right)$

It is important to note that a special kind of predicate definition folding corresponds to empty rules. For instance, all segment predicates have an empty rule. Hence, it is always feasible to generate a segment seemingly out of nothing.

Example 2.17: Folding an inductive segment

In this example we consider the abstract state from the last example with one modification: `y` now points to the successor of the first node. That is to say:

$$\left(\begin{array}{c}\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta * \alpha.\mathtt{next} \mapsto \beta \ * \beta.\mathbf{list}, \sigma_n^\sharp \\ * \, \alpha.\mathtt{next} \mapsto \delta\end{array}\right)$$

In this abstract state it is possible to fold the cell at address α to obtain a full **list** predicate. However, this would entail forfeiting the information regarding `y` pointing somewhere within the list. The goal here is to fold the memory cell in blue into a segment predicate. The initial step involves appending an empty segment that start and ends in β. The resulting abstract memory is as follows:

$$\begin{array}{c}\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta * \alpha.\mathtt{next} \mapsto \beta \ * \beta.\mathbf{list} \mathrel{-\!\!*} \beta.\mathbf{list} * \beta.\mathbf{list} \\ * \, \alpha.\mathtt{next} \mapsto \delta\end{array}$$

The part in blue corresponds to a non-empty segment rule. The pure part of the rule enforces that α and β are different. If the constraint is verified in the numerical part of the abstract value, the abstract memory state can be folded once again. This yields the following:

$$\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta * \alpha.\mathbf{list} \mathrel{-\!\!*} \beta.\mathbf{list} * \beta.\mathbf{list}$$

Concatenating segments and other predicates The second type of folding corresponds to the concatenation principles discussed earlier. Given that a segment summarizes an incomplete inductive data structure, appending a full inductive predicate to it forms a full inductive predicate.

$$\alpha.\mathbf{pred}(\kappa) \mathrel{-\!\!*} \beta.\mathbf{pred}(\kappa') * \beta.\mathbf{pred}(\kappa') \overset{\mathcal{F}}{\rightsquigarrow} \alpha.\mathbf{pred}(\kappa)$$

Likewise, appending another segment at the tail of an existing segment forms a new segment spanning from the start of the first one to the end of the second.

$$\alpha.\mathbf{pred}(\kappa) \mathrel{-\!\!*} \beta.\mathbf{pred}(\kappa') * \beta.\mathbf{pred}(\kappa') \mathrel{-\!\!*} \delta.\mathbf{pred}(\kappa'') \overset{\mathcal{F}}{\rightsquigarrow} \alpha.\mathbf{pred}(\kappa) \mathrel{-\!\!*} \delta.\mathbf{pred}(\kappa'')$$

As explained in Example 2.17, folding may occasionally result in the loss of vital information from the abstract memory part. Moreover, as it is always feasible to create an empty segment, ensuring that the rewriting process does not deviate from the intended direction necessitates caution. We will not discuss these matters in the current chapter. For further insights into how these rules are invoked and executed, we direct the reader to [CR08].

2.4.3.2 Inclusion test

In order to prove that an abstract value $(m_l^\sharp, \sigma_{n,l}^\sharp)$ is included in $(m_r^\sharp, \sigma_{n,r}^\sharp)$, the memory part of the left input is folded until it matches the right input. If this succeeds, then the inclusion test is performed between the numerical parts of the abstract values. If the check in the numerical domain succeeds as well, it entails that the concretization of the left input is included in the concretization of the right one.

Following the soundness criteria from Theorem 2.10, this inclusion test is conservative: if it succeeds, then the concretization of the left input is included in the concretization of the right one.

Example 2.18: Inclusion test

To illustrate the inclusion test, let us consider the two abstract states:

$$s_l^\sharp = \left(\begin{array}{c}\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \alpha'' * \alpha.\mathbf{list} \mathrel{-\!\!*} \alpha'.\mathbf{list} * \alpha'.\mathtt{next} \mapsto \alpha'' \ * \alpha''.\mathbf{list}, \\ * \, \alpha'.\mathtt{next} \mapsto \delta \\ \alpha, \alpha', \alpha'' \neq 0 \wedge \alpha \neq \alpha' \wedge \alpha \neq \alpha'' \wedge \alpha' \neq \alpha''\end{array}\right)$$

$$s_r^\sharp = (\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta * \alpha.\mathbf{list} \mathrel{-\!\!*} \beta.\mathbf{list} * \beta.\mathbf{list}, \alpha, \beta \neq 0)$$

As seen in to Example 2.17, the cell at address α' can be folded into a list segment between

α' and α''. Moreover, the segments between α and α'' can be concatenated into a single one. To sum up, the rewriting steps can be summarized as follows:

$$\begin{aligned}
&\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \alpha'' * \alpha.\mathbf{list} \nrightarrow \alpha'.\mathbf{list} * \alpha'.\mathtt{next} \mapsto \alpha'' * \alpha''.\mathbf{list} \\
&\qquad\qquad * \alpha'.\mathtt{next} \mapsto \delta \\
\overset{\mathcal{F}}{\rightsquigarrow}\ &\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \alpha'' * \alpha.\mathbf{list} \nrightarrow \alpha'.\mathbf{list} * \underbrace{\alpha'.\mathtt{next} \mapsto \alpha'' * \alpha''.\mathbf{list} \nrightarrow \alpha''.\mathbf{list} * \alpha''.\mathbf{list}}_{} \\
&\qquad\qquad * \alpha'.\mathtt{next} \mapsto \delta \\
\overset{\mathcal{F}}{\rightsquigarrow}\ &\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \alpha'' * \underbrace{\alpha.\mathbf{list} \nrightarrow \alpha'.\mathbf{list} * \alpha'.\mathbf{list} \nrightarrow \alpha''.\mathbf{list}}_{} * \alpha''.\mathbf{list} \\
\overset{\mathcal{F}}{\rightsquigarrow}\ &\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \alpha'' * \alpha.\mathbf{list} \nrightarrow \alpha''.\mathbf{list} * \alpha''.\mathbf{list}
\end{aligned}$$

The final form matches the memory part of $s_r^\sharp$. So the inclusion holds between the memory parts. Additionally, the matching between the left and right memories produces a mapping between symbolic variables. In the right abstract state the symbolic variable α and β correspond respectively to α and α'' in the left input. The numerical abstract domain is used to check whether $\sigma_{n,l}^\sharp \sqsubseteq_n^\sharp \sigma_{n,r}^\sharp[\alpha''/\beta]$. Since this inclusion holds in the numerical abstract domain, the analysis concludes that $s_l^\sharp \sqsubseteq_{\mathbb{S}}^\sharp s_r^\sharp = \mathbf{true}$.

2.4.3.3 Upper bound

Join In essence, the upper bound between two abstract states folds the memory parts of the two inputs, until they fit with one another. Then the numerical parts of the inputs are joined to form the numerical part of the output. This produces a sound over-approximation of both inputs.

Example 2.19: Abstract join

To illustrate the abstract union in the combined abstract domain consider, let us consider the two following abstract states:

$$\begin{aligned}
s_l^\sharp &= (\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \alpha * \alpha.\mathbf{list}, \alpha \neq 0) \\
s_r^\sharp &= \left(\begin{array}{l} \underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta' * \alpha.\mathbf{list} \nrightarrow \beta.\mathbf{list} * \beta.\mathtt{next} \mapsto \beta' * \beta'.\mathbf{list}, \alpha, \beta, \beta' \neq 0 \\ \qquad\qquad * \beta.\mathtt{next} \mapsto \delta \end{array}\right)
\end{aligned}$$

Example 2.18 shows how $m_r^\sharp \overset{\mathcal{F}^*}{\rightsquigarrow} \underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \beta' * \alpha.\mathbf{list} \nrightarrow \beta'.\mathbf{list} * \beta'.\mathbf{list}$. Moreover, the left input can be folded to obtain a similar abstract memory. It suffices to add an empty segment that begins and ends in α. This means that the memory part of the output is:

$$\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \alpha' * \alpha.\mathbf{list} \nrightarrow \alpha'.\mathbf{list} * \alpha'.\mathbf{list}$$

In the output both α and α' correspond to symbolic variables in the left input. So the left numerical state is modified by asserting that α' is equal to α. For the right input, a simple rewriting suffices: β' is replaced by α'. Therefore, the numerical part of the output is computed as:

$$\begin{aligned}
\sigma_{n,o}^\sharp &= \mathbf{guard}_n^\sharp(\sigma_{n,l}^\sharp, \alpha' = \alpha) \sqcup_n^\sharp \sigma_{n,r}^\sharp[\alpha'/\beta'] \\
&= (\alpha' = \alpha \neq 0) \sqcup_n^\sharp (\alpha, \alpha', \beta \neq 0) \\
&= (\alpha, \alpha' \neq 0)
\end{aligned}$$

To conclude, the union of the two abstract states is:

$$s_l^\sharp \sqcup_{\mathbb{S}}^\sharp s_r^\sharp = (\underline{\mathtt{x}} \mapsto \alpha * \underline{\mathtt{y}} \mapsto \alpha' * \alpha.\mathbf{list} \nrightarrow \alpha'.\mathbf{list} * \alpha'.\mathbf{list}, \alpha, \alpha' \neq 0)$$

Widening Widening in the combined abstract domain resembles union, with two exceptions. First, the rewriting rules applicable to the memory of the left input are restricted to guarantee convergence. Second, the numerical part is derived by widening the left numerical input with the right one. The resulting widening is sound and ensures the termination properties stated in Theorem 2.8.

Listing 2.2: Insertion function from Listing 1.2

```
1  void insert(task* new, task_container** container){
2    node* node = malloc(sizeof(node));
3    node->task = new;
4    node->left = node->right = null;
5    if( *container ){ // Non-Empty Case
6      struct node* c = *container;
7      while(c->content->wst <= new->wst && c->left ||
8            c->content->wst >  new->wst && c->right )
9        c = c->content->wst <= new->wst ? c->left : c->right;
10     node->parent = c;
11     if( c->content->wst <= new->wst ){
12       c->left = node;
13     } else {
14       c->right = node;
15     }
16   } else { // Empty Case
17     *container = node->parent = node;
18   }
19 }
```

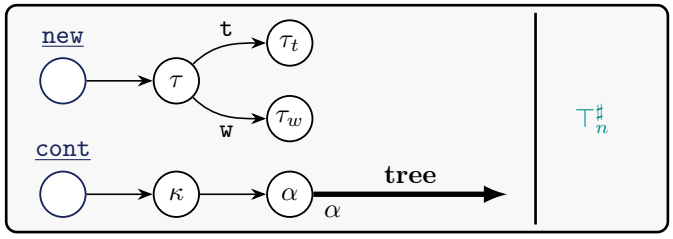


Figure 2.17: Pre-Condition

2.4.4 A final example

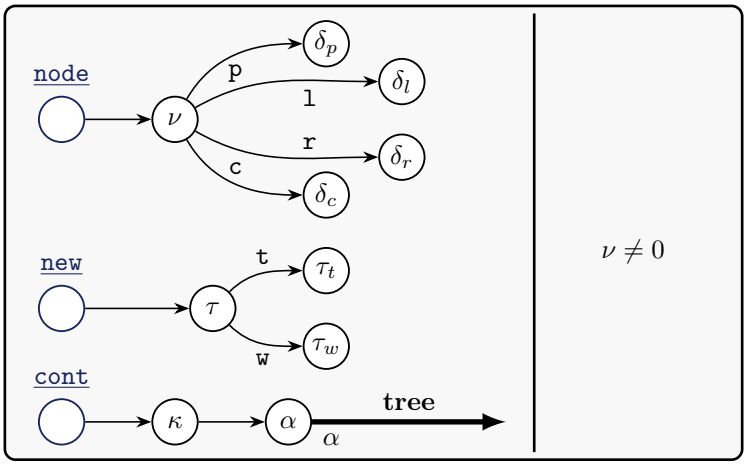
To conclude this chapter, let us sketch the analysis of the `insert` function from Listing 1.2. The pre-condition of this function states that `container` points to a pointer of a well-formed binary tree and `new` to a task. The abstract state corresponding to the precondition is depicted in Figure 2.17. To simplify the graph representation of abstract states, we shorten field names to one letter. All fields are abbreviated by their initial, excepted the weighted service time field `wst`, that is shortened by `t`. Similarly, variable `container` is shortened by `cont`.

2.4.4.1 Initialization

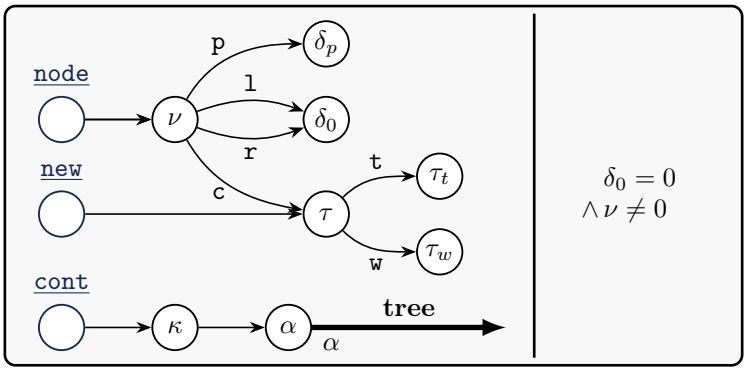
The function starts by allocating a tree node to store the task control block. In this example, we assume that `malloc` always performs dynamic memory allocation. Therefore, the analysis adds symbolic variables $\nu, \delta_l, \delta_r, \delta_p$, and δ_c to denote the address and the content of the new memory cell. Additionally, it guards in the numerical part of the abstract value the constraint expressing that ν is the non-null address. After the dynamic memory allocation, the abstract state computed by the analysis corresponds to the one depicted in Figure 2.18a.

Then, the analysis proceeds with the initialisation of the new node. The assignment `node->task = new` boils down to setting the destination of the edge from the node ν, labeled by `content` to τ. Additionally, to set the children pointer to the null pointer, the analysis adds a fresh symbolic variable δ_0, with a constraint in the numerical part of the abstract value stating that δ_0 has a null value. Then, it changes the destination of the `left` and `right` arrows to the new node δ_0. Since the vertexes δ_l and δ_r are no longer reachable from a symbolic variable denoting a program variable, they are removed from the abstract value. At the end of lines 4, the analysis computes the abstract value shown in Figure 2.18b.

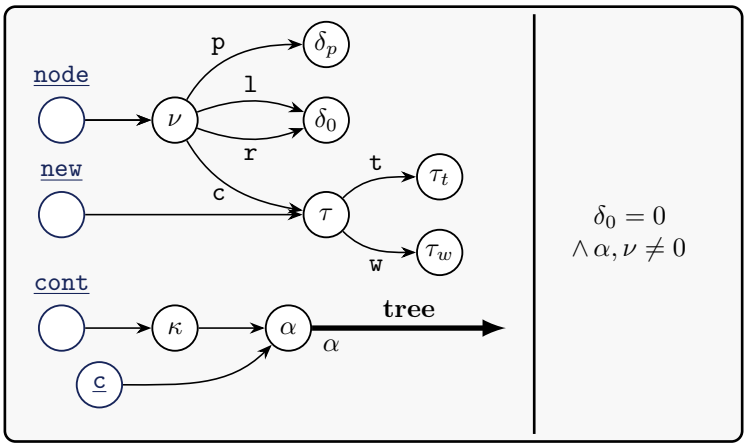
In this example, we focus on the non-empty case. The constraint of the conditional statement in line 5, corresponds to $\alpha \neq 0$. It is added in the numerical part of the abstract state. Furthermore, after performing the assignment of line 6, the analysis obtains the abstract state, written $s^\sharp_{l.6}$, presented in Figure 2.18c.



(a) Abstract state at the end of line 2



(b) Abstract state at the end of line 4



(c) Abstract state at the end of line 6

Figure 2.18: Abstract states computed during the initialization

2.4.4.2 Analysis of the loop

First iteration Then, the analysis proceeds with the loop. Given that the loop condition comprises a disjunction of two cases, each case is analyzed separately by the analysis. In the first one, the analysis must evaluate `c->content->wst`. This expression is summarized in the $\alpha.\mathbf{tree}(\alpha)$ inductive call. So the analysis unfolds it. The empty case is not feasible because it is inconsistent with the constraint of line 5, $\alpha \neq 0$. This entails that the memory reading `c->content` does not fail. At this moment, the state inferred by the analysis corresponds to the abstract value depicted in Figure 2.19a. Since the loop does not modify the parts of the memory pointed by `new` and `node`, we do not display them in the graph representation. However, we keep track of the numerical constraints involving these parts.

Additionally, the analysis unfolds further the task predicate that is pointed by field `content`, to read the field `wst` of the task stored at the root of the tree. After this two unfolding, the analysis computes abstract state presented in Figure 2.19b.

In the abstract state the two constraints in the first disjunction boil down to $\beta_t \leqslant \tau_t$ and $\beta_l \neq 0$. They are added in the numerical part of the abstract state. Then the analysis interprets the body of the loop. In the current abstract state, the assignment is performed by writing β_l as the destination of the points-to predicate from `c`. After the first iteration through the loop, the analysis generates the abstract state presented in Figure 2.19c.

Analyzing the second loop condition is carried out similarly, resulting in the state shown in Figure 2.19d.

First widening After the first iteration, the analysis widens the state at the head of the loop (presented in Figure 2.18c) with the result of the analysis of the body of the loop (presented in Figures 2.19c and 2.19d). As described earlier, the widening starts with the memory part. The parts corresponding to variables `new` and `node` are similar in all abstract states. Moreover, in all inputs, `c` points to a full inductive predicate. The remainder of the memories describe a (potentially empty) tree segment extending from the address indicated by `cont` to `c`. Figure 2.20 illustrates, for each input, what parts of the memories are summarized by an inductive predicate in the output. Then the widening is performed between the numerical parts of the inputs. It retains that `container`, `c` are non-null pointers. The outcome of the first widening $s_1^{\sharp(\nabla)}$ is the abstract state presented at the bottom of Figure 2.20.

Second iteration Since $s_{l.6}^{\sharp}$ differs from $s^{\sharp(1)}$, the loop invariant is not stable. Therefore, the analysis continues with another iteration. The second iteration follows the same principle as the first one, given that `c` corresponds to the non-null address of a well-formed binary tree. It is unfolded, as well as the task stored at the corresponding node. Once again, we obtain a disjunction consisting of two elements presented in Figure 2.21.

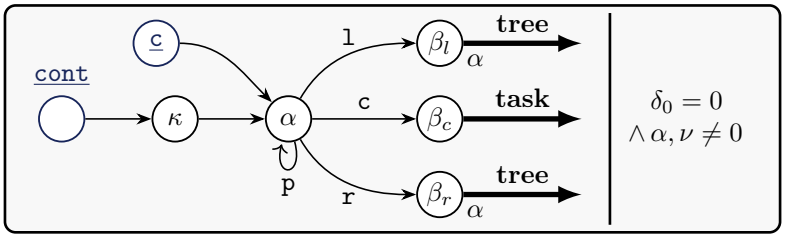
Second widening After the second iteration, the analysis performs a widening between the previous widened abstract state, and the result of the iteration. The widening follows the same pattern as the previous one: `c` points to a full inductive predicate and the part of tree between `container` and `c` is summarized in a segment predicate. Figure 2.22 displays the parts of the abstract states that are matched together, as well as the result of the second widening $s_2^{\sharp(\nabla)}$. The sole difference between $s_1^{\sharp(\nabla)}$ and $s_2^{\sharp(\nabla)}$ lies in the fact that the address α pointed by `container` is no longer the backward pointer of the tree pointed by `c`. Instead, it is some non-null address π.

Third iteration The invariant remains unstable. Therefore, the analysis performs a third iteration. Following the widening of $s_2^{\sharp(\nabla)}$ with the result of the third iteration, we observe that $s_2^{\sharp(\nabla)}$ is stable.

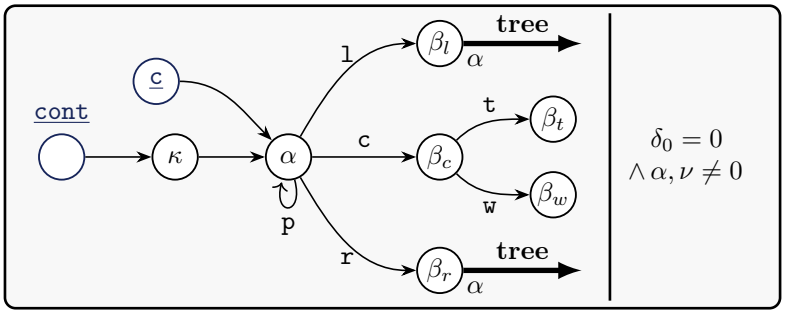
2.4.4.3 Insertion

Ultimately, after inserting the `new` node in the tree, and taking into account the empty case, the analysis computes the final state presented in Figure 2.23.

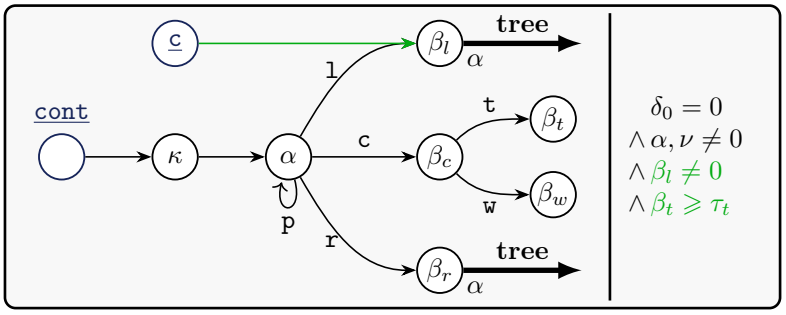
This final state entails that no run-time error is reachable starting from a memory state abstracted by $s_{pre}^{\sharp}$ and that the tree pointed by `container` remains a well-formed binary tree. Additionally, it states that this tree contains a leaf pointing to the `new` task. However, this abstract state imposes no



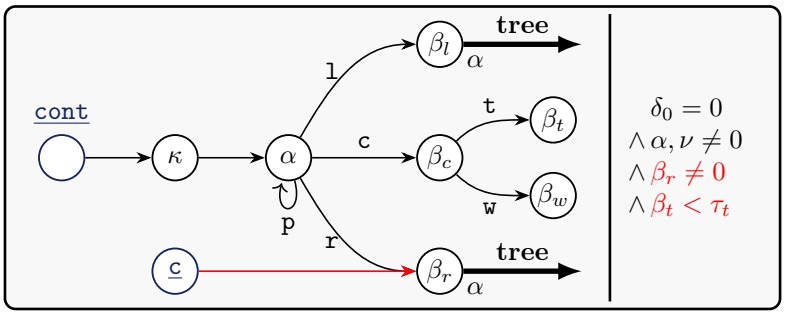
(a) Abstract state obtained after unfolding α.**tree**



(b) Abstract state obtained after unfolding β_c.**task**



(c) Abstract state computed at the end of the first iteration (left case)



(d) Abstract state computed at the end of the first iteration (right case)

Figure 2.19: Abstract states computed during the first iteration

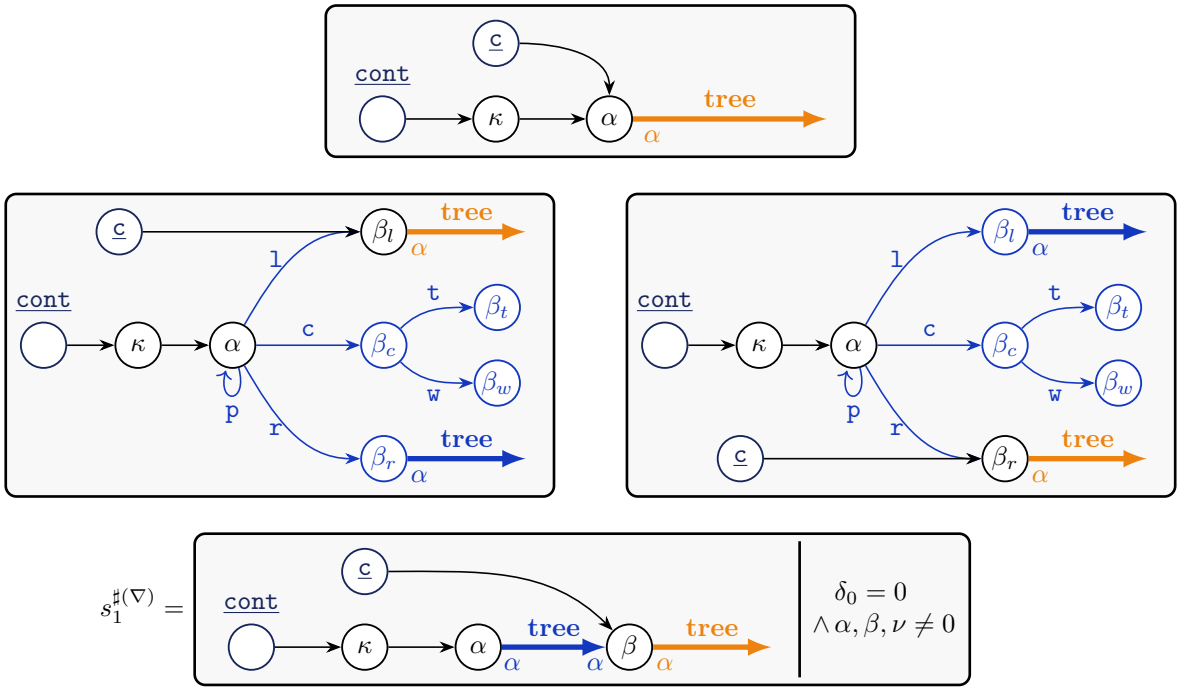
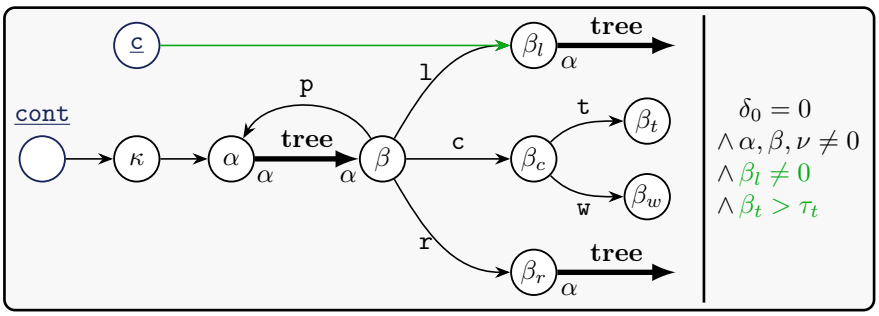
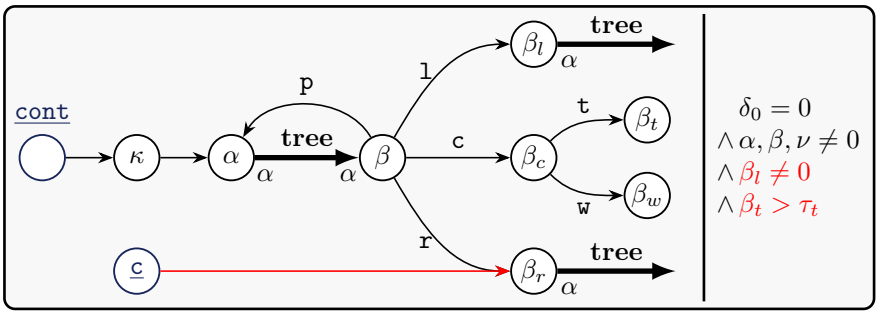


Figure 2.20: First widening



(a) Abstract state computed at the end of the second iteration (left case)



(b) Abstract state computed at the end of the second iteration (right case)

Figure 2.21: Abstract states computed during the second iteration

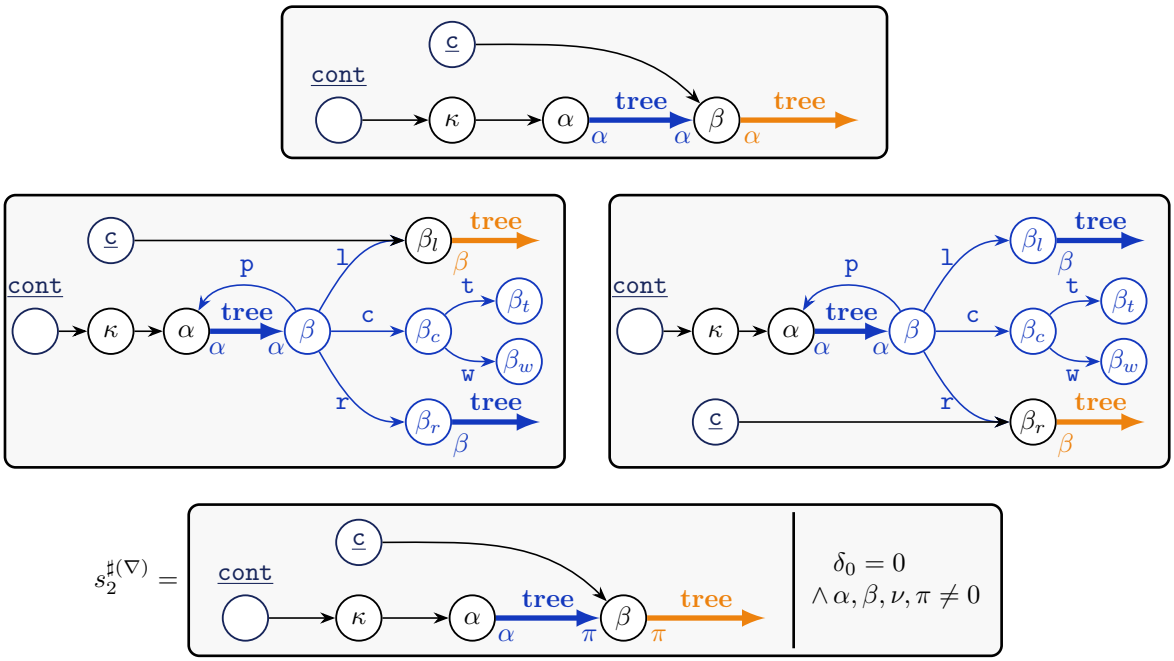


Figure 2.22: Second widening

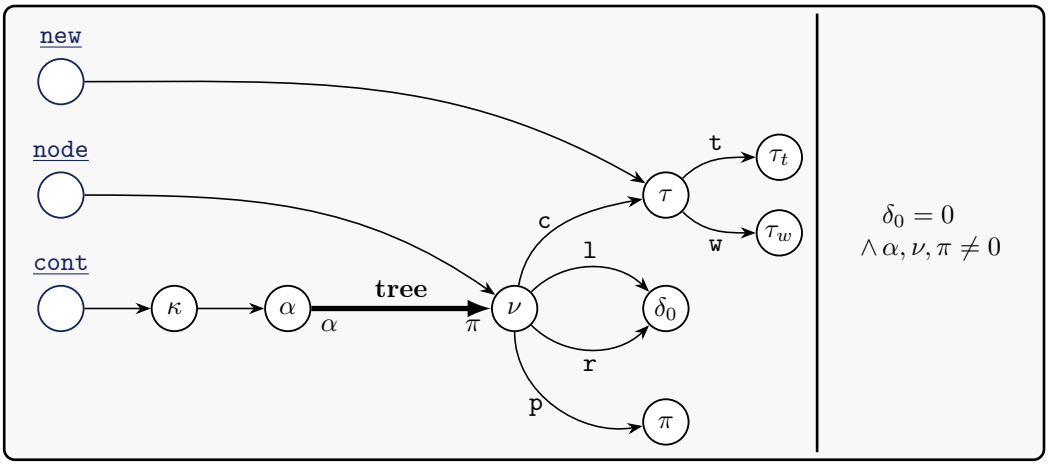
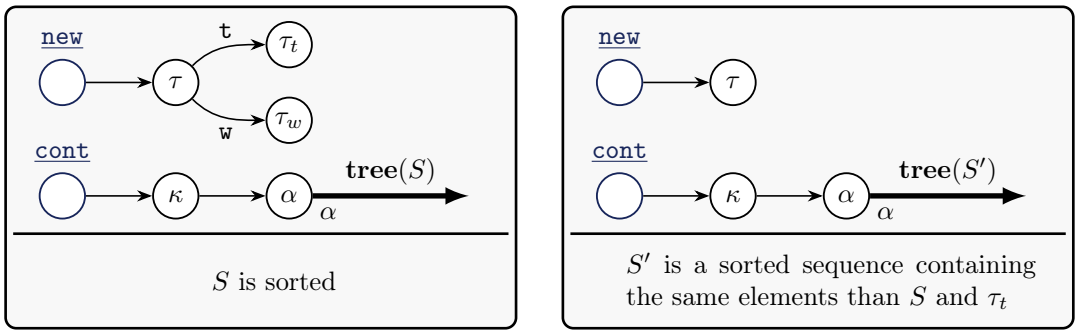


Figure 2.23: Final state

constraints regarding the insertion point of the new element or the preservation of content. Consequently, it lacks the expressiveness required to demonstrate the partial functional correctness of the insertion in a binary search tree.

To address this problem, we propose to strengthen the definition of inductive predicate with a *sequence parameter*. For instance, $\alpha.\mathbf{tree}(\alpha, S)$ asserts that α is the address of a well-formed binary tree. Furthermore, it specifies that the sequence of weighted service times stored in the tree conforms to a specified sequence S. If we additionally require the elements of S to be sorted, then the tree qualifies as a binary search tree. Therefore, with such an inductive parameter extended with a sequence parameter we can express the partial functional correctness of the `insert` function as follows:



However, in order to prove such constraints on the content of the binary tree, the analysis needs to be able to reason over sequences of numerical values, their content and their sortedness. Additionally, the analysis must be able to extend the definition of inductive predicates to support such sequence parameters. Supporting sequence parameters implies to accommodate the unfolding and folding abstract operators to derive the corresponding sequence constraints to either add or verify.

3 Chapter
A sequence abstract domain

This chapter presents a relational abstract domain dedicated to reason over sequence of values. This domain is able to reason over the content, the length, the extreme elements, and the sortedness of sequences. We describe the domain, the constraints it manipulates, its concretization. And we present the abstract transfer functions.

3.1 Sequence Predicates

This section presents the type of sequence constraints that should be expressed by the abstract domain. The goal of this domain is to express constraints on $\mathbb{V}^*$ the set of (possibly empty) words over the alphabet $\mathbb{V}$.

$$\sigma_s : \begin{Bmatrix} S \mapsto 1\,2\,5\,8 \\ S_0 \mapsto 1\,2\,3\,5\,8 \\ S_1 \mapsto 1\,2\,5\,8\,9 \\ S_2 \mapsto \varepsilon \end{Bmatrix} \quad \sigma_m : \begin{Bmatrix} \texttt{mset}_S \mapsto \{\!\{ 1;2;5;8 \}\!\} \\ \texttt{mset}_{S_0} \mapsto \{\!\{ 1;2;3;5;8 \}\!\} \\ \texttt{mset}_{S_1} \mapsto \{\!\{ 1;2;5;8;9 \}\!\} \\ \texttt{mset}_{S_2} \mapsto \{\!\{ \}\!\} \end{Bmatrix} \quad \sigma_n : \begin{Bmatrix} \texttt{min}_S \mapsto 1 & \texttt{max}_S \mapsto 8 & \texttt{len}_S \mapsto 4 \\ \texttt{min}_{S_0} \mapsto 1 & \texttt{max}_{S_0} \mapsto 8 & \texttt{len}_{S_0} \mapsto 5 \\ \texttt{min}_{S_1} \mapsto 1 & \texttt{max}_{S_1} \mapsto 9 & \texttt{len}_{S_1} \mapsto 5 \\ \texttt{min}_{S_2} \mapsto +\infty & \texttt{max}_{S_2} \mapsto -\infty & \texttt{len}_{S_2} \mapsto 0 \\ \alpha \mapsto 3 & \beta \mapsto 9 & \end{Bmatrix}$$

Figure 3.1: Example of a sequence concrete state

3.1.1 Three types of symbolic variables

Given that the goal of this abstract domain is to express constraints on sequence of values, it must do so on both sequences and the values of their elements, *i.e.* numerical values. Therefore, the domain seeks to reason about two types: numerical sequences and numerical values.

Recall that $\mathcal{V}$ forms a set of symbolic numerical variables α, β denoting values in $\mathbb{V}$.

Sequence variables Alongside the set of numerical variables $\mathcal{V}$ previously introduced, we introduce a separate set $\mathcal{V}_s$, representing *symbolic sequence variables*, or more succinctly *sequence variables*. We note $S, S', S_i \in \mathcal{V}_s$ for such sequence variables.

Multiset variables Additionally, to express constraint on the content of a sequence, we pick a set of *symbolic multiset variables*, $\mathcal{V}_m$ that stand for multiset of values in $\mathbb{V}$. We write $\mathcal{M}(\mathbb{V})$ for the set of multisets of values in $\mathbb{V}$.

Attribute variables Given that our goal is to track information over the size and sortedness of sequences, we pick for each sequence variables $S \in \mathcal{V}_s$, three symbolic variables $\texttt{len}_S, \texttt{min}_S$, and $\texttt{max}_S$ to denote respectively the length, the minimum and maximum elements of S. Such variables are called *numeric attribute variables*.

Similarly, we select for each sequence variable a multiset symbolic variable $\texttt{mset}_S \in \mathcal{V}_m$, that denotes the multiset of its elements. This variable is referred as the *multiset attribute variable*.

3.1.2 Concrete states

A concrete state is formed by a tuple a valuation functions that map each type of symbolic variables (numerical, sequence, multiset) to elements of the corresponding type. However, given that there is an implicit relationship between sequence variables and their attribute variables, we restrict the set of concrete states to valid ones. A triple of valuations is valid, if for each sequence variables S, the valuations of its attribute variables is consistent with the valuation of S.

In the case S evaluates to the empty sequence, we follow the convention stating that its minimal and maximal elements corresponds respectively to the infimum and supremum of the empty set. That is to say $+\infty$ and $-\infty$. As a consequence, we extend the set of values to $\overline{\mathbb{V}} := \mathbb{V} \uplus \{-\infty, +\infty\}$.

Definition 3.1: Sequence concrete states

A *concrete sequence state* is a tuple $\sigma = (\sigma_n, \sigma_s, \sigma_m)$, of valuation functions $\sigma_n \in \mathcal{V} \to \overline{\mathbb{V}}$, $\sigma_s \in \mathcal{V}_s \to \mathbb{V}^*$, $\sigma_m \in \mathcal{V}_m \to \mathcal{M}(\mathbb{V})$, such that for any sequence variable S,

$$\sigma_s(S) = \varepsilon \Rightarrow \begin{cases} \texttt{len}_S = 0 \\ \texttt{min}_S = +\infty \\ \texttt{max}_S = -\infty \\ \texttt{mset}_S = \{\!\{ \}\!\} \end{cases} \qquad \sigma_s(S) = a_1 \dots a_n \Rightarrow \begin{cases} \texttt{len}_S = n \\ \texttt{min}_S = \min_{1 \leqslant i \leqslant n} a_i \\ \texttt{max}_S = \max_{1 \leqslant i \leqslant n} a_i \\ \texttt{mset}_S = \{\!\{ a_1, \dots, a_n \}\!\} \end{cases}$$

We write $\mathbb{D}_s$ for the set of concrete sequence states.

Example 3.1: Sequence concrete states

Figure 3.1 presents a simplified well-formed concrete state. In this state, we consider only three sequence variable (S, S_0, and S_1), their respective attribute variables, and two additional numerical symbolic variables α and β. Note that for each sequence variables, their corresponding attribute variables match the value of the expressed attribute. For instance, replacing the value

$$
\begin{aligned}
\mathbb{E}[\![\,\bullet\,]\!]_s &: \mathcal{E}_s \times (\mathcal{V} \to \mathbb{V}) \times (\mathcal{V}_s \to \mathbb{V}^*) \longrightarrow \mathbb{V}^* \\
\mathbb{E}[\![\texttt{[]}]\!]_s(\sigma_n, \sigma_s) &= \varepsilon \\
\mathbb{E}[\![\texttt{[}\alpha\texttt{]}]\!]_s(\sigma_n, \sigma_s) &= \sigma_n(\alpha) \\
\mathbb{E}[\![S]\!]_s(\sigma_n, \sigma_s) &= \sigma_s(S) \\
\mathbb{E}[\![E_1.E_2]\!]_s(\sigma_n, \sigma_s) &= \mathbb{E}[\![E_1]\!]_s(\sigma_n, \sigma_s).\mathbb{E}[\![E_2]\!]_s(\sigma_n, \sigma_s) \\
\mathbb{E}[\![\texttt{sort}(E)]\!]_s(\sigma_n, \sigma_s) &= c_{\pi(1)}c_{\pi(2)}\dots c_{\pi(n)} \\
\text{where } &\begin{cases} \mathbb{E}[\![E]\!]_s(\sigma_n, \sigma_s) = c_1 c_2 \dots c_n \\ \forall i \in [\![1, n-1]\!], c_{\pi(i)} \leqslant c_{\pi(i+1)} \\ \pi \in \mathfrak{S}_n \end{cases}
\end{aligned}
$$

Figure 3.2: Evaluation of sequence expressions

assigned to $\texttt{min}_S$ by 0 would lead to an inconsistent state as it does not reflect the minimum element of S.

3.1.3 Sequence expressions

We now present the set of sequence expressions $\mathcal{E}_s$ manipulated by the domain. An expression is either the empty sequence, written `[]`, a sequence of length one, called an *atom*, written $\texttt{[}\alpha\texttt{]}$, that consists of a single symbolic numerical variable $\alpha \in \mathcal{V}$, a sequence variable $S \in \mathcal{V}_s$, a concatenation of sequence expressions, or the sorting of a sequence expression. For the latter, we employ a symbolic function $\mathbf{sort} : \mathcal{E}_s \to \mathcal{E}_s$ that maps any sequence to it sorted permutation.

Definition 3.2: Sequence expressions

Sequence expressions are defined by the following grammar:

$$
\begin{aligned}
E(\in \mathcal{E}_s) ::= \ & \texttt{[]} \\
\mid \ & \texttt{[}\alpha\texttt{]} \qquad \alpha \in \mathcal{V} \\
\mid \ & S \qquad S \in \mathcal{V}_s \\
\mid \ & E.E \\
\mid \ & \mathbf{sort}(E)
\end{aligned}
$$

The semantics of a sequence expression $E \in \mathcal{E}_s$, is a function $\mathbb{E}[\![E]\!]_s : (\mathcal{V} \to \mathbb{V}) \times (\mathcal{V}_s \to \mathbb{V}^*) \to \mathbb{V}^*$ that takes as argument a numerical and a sequence valuation and evaluates the sequence expression according to these valuations. The definition of sequence expressions semantics is presented in Figure 3.2. The constant `[]` always evaluates to the empty sequence ε. The semantics of an atom or a sequence variable boils down to the value of the variable with the corresponding valuation function. Concatenation of expressions results in the concatenation of their evaluations. Lastly, when invoking the **sort** symbolic function, the sub-expression undergoes evaluation. Then, it is permuted to ensure the resulting sequence is sorted.

Remark 3.1: Implicit rewrites of sequence expressions

As a consequence of sequence expressions semantics, the concatenation of sequence expressions is associative. That is, for every sequence expressions E, E', and E'', and any concrete state σ, we have the following equality: $\mathbb{E}[\![(E.E').E'']\!]_s(\sigma) = \mathbb{E}[\![E.(E'.E'')]\!]_s(\sigma)$. Additionally, the empty sequence symbol `[]` can be removed from a sequence expression containing numerical or sequence variables without modifying its semantics. Finally, when an expression contains nested calls to the **sort** symbolic function, the innermost ones are superfluous. For instance, for any expressions E and E', and any concrete state, $\mathbb{E}[\![\mathbf{sort}(E.\mathbf{sort}(E'))]\!]_s(\sigma) = \mathbb{E}[\![\mathbf{sort}(E.E')]\!]_s$. Each of the principles listed above correspond to rewriting rules on sequence expressions that are performed implicitly.

Example 3.2: Evaluation of sequence expressions

Let us consider the concrete sequence state, $\sigma = (\sigma_n, \sigma_m, \sigma_s)$, depicted in Figure 3.1. In this state, the evaluation of the expression $S.\texttt{[}\alpha\texttt{]}$ corresponds to the sequence expressed by S appended with the value denoted by α. That is to say $\mathbb{E}[\![S.\texttt{[}\alpha\texttt{]}]\!]_s(\sigma_n, \sigma_s) = 1\,2\,5\,8\,3$.

For the expression $\mathbf{sort}(\texttt{[}\alpha\texttt{]}.\texttt{[}\beta\texttt{]}.S)$, the evaluation starts by the expression inside the **sort**

symbolic function. This expression represents the sequence denoted by S, preceded with the values of α and β. Its evaluation results in 3 9 1 2 5 8. Then, this sequence is rearranged to be sorted. We obtain: $\mathbb{E}[\![\mathbf{sort}([\alpha].[\beta].S)]\!]_s(\sigma_n, \sigma_s) = 1\,2\,3\,5\,8\,9$.

3.1.4 Sequence constraints

The abstract domain expresses two kinds of sequence constraints. This first kind are called *sequence definitions* and correspond to an equality constraint between a sequence variable and a sequence expression. The second one simply states that a sequence corresponding to symbolic sequence variable has no repeating elements.

Definition 3.3: Sequence constraints

The sequence constraints are defined with the following grammar:

$$\begin{aligned} C_s(\in \mathcal{C}_s) ::= & \; S = E && S \in \mathcal{V}_s \\ | \; & \mathbf{unique}(S) && S \in \mathcal{V}_s \end{aligned}$$

Their semantics is defined by the satisfiability judgement $\vDash_s$:

$$\begin{aligned} \sigma_n, \sigma_s \vDash_s S = E \quad & \text{iff } \sigma_s(S) = \mathbb{E}[\![E]\!]_s(\sigma_n, \sigma_s) \\ \sigma_n, \sigma_s \vDash_s \mathbf{unique}(S) \quad & \text{iff } \begin{array}{c} \sigma_s(S) = c_1 \dots c_n \\ \wedge \forall i \neq j, c_i \neq c_j \end{array} \end{aligned}$$

Remark 3.2: Sortedness as a function and not as a predicate

In our approach we model sortedness using a symbolic function $\mathbf{sort} : \mathcal{E}_s \to \mathcal{E}_s$ and not a predicate $\mathbf{sorted}(S)$ stating the sequence expressed by S is sorted. Indeed, it is possible to express the latter with the former and sequence definitions: $\mathbf{sorted}(S) :\Leftrightarrow S = \mathbf{sort}(S)$. Additionally, this function allows us to express constraints implying sorted constraints in a simple manner. For example, if a list contains a sequence S that is assumed sorted (*i.e.* $S = \mathbf{sort}(S)$), then the sequence, S', obtained after inserting an element α in this list to preserve sortedness is expressed as $S' = \mathbf{sort}([\alpha].S)$.

Remark 3.3: Definition is not restrictive

Restricting the left part of sequence definition may seem to limit the expressiveness of our approach. However, it simplifies the manipulation of constraints. Moreover, this is enough in our application. The only constraints generated by the analysis, to verify or assume, are all sequence definitions.

Example 3.3: Sequence constraints satisfaction

Let us consider once again the concrete sequence state, $\sigma = (\sigma_n, \sigma_m, \sigma_s)$, presented in Figure 3.1. This state satisfies the following constraints:

- $\sigma_n, \sigma_s \vDash_s S_1 = S.[\beta]$, since $\sigma_s(S_1) = 1\,2\,5\,8\,9 = \mathbb{E}[\![S.[\beta]]\!]_s(\sigma)$.
- $\sigma_n, \sigma_s \vDash_s S_0 = \mathbf{sort}(S.[\alpha])$, because $\sigma_s(S_0) = 1, 2\,3\,5\,8 = \mathbb{E}[\![\mathbf{sort}([\alpha]\,S)]\!]_s$.
- $\sigma_n, \sigma_s \vDash_s \mathbf{unique}(S_1)$ as the sequence corresponding to $\sigma_s(S_1)$, $1\,2\,5\,8\,9$, has no repeating elements.

3.2 Elements of the abstract domain

3.2.1 Underlying abstract domains

In order to handle reasoning on numerical constraints and content constraints, the sequence abstract domain leverages two underlying abstract domains. These domains are parameters of the sequence domain.

$$\begin{aligned}\mathscr{E}(\in \mathcal{E}_{ms}) ::= {} & \{\} \\ \mid {} & \{\alpha\} \qquad \alpha \in \mathcal{V} \\ \mid {} & \mathscr{M} \qquad \mathscr{M} \in \mathcal{V}_m \\ \mid {} & \mathscr{E} \uplus \mathscr{E}\end{aligned}$$

(a) Syntax of multiset expressions

$$\begin{aligned}C_{ms}(\in \mathcal{C}_{ms}) ::= {} & \alpha \in \mathscr{M} \\ \mid {} & \mathscr{E} = \mathscr{E}\end{aligned}$$

(b) Syntax of multiset constraints

$$\begin{aligned}\mathbb{E}[\![\bullet]\!]_{ms} &: \mathcal{E}_m \times (\mathcal{V} \to \mathbb{V}) \times (\mathcal{V}_m \to \mathcal{M}(\mathbb{V})) \longrightarrow \mathcal{M}(\mathbb{V}) \\ \mathbb{E}[\![\{\}]\!]_{ms}(\sigma_n, \sigma_m) &:= \varnothing \\ \mathbb{E}[\![\{\alpha\}]\!]_{ms}(\sigma_n, \sigma_m) &:= \{\!\{ \sigma_n(\alpha) \}\!\} \\ \mathbb{E}[\![\mathscr{M}]\!]_{ms}(\sigma_n, \sigma_m) &:= \sigma_m(\mathscr{M}) \\ \mathbb{E}[\![\mathscr{E} \uplus \mathscr{E}']\!]_{ms}(\sigma_n, \sigma_m) &:= \mathbb{E}[\![\mathscr{E}]\!]_{ms}(\sigma_n, \sigma_m) \uplus \mathbb{E}[\![\mathscr{E}']\!]_{ms}(\sigma_n, \sigma_m)\end{aligned}$$

(c) Semantics of multiset expressions

$$\begin{aligned}(\sigma_n, \sigma_m) \vDash_{ms} \alpha \in \mathscr{M} \qquad & \text{iff } \sigma_n(\alpha) \in \sigma_m(\mathscr{M}) \\ (\sigma_n, \sigma_m) \vDash_{ms} \mathscr{E} = \mathscr{E}' \qquad & \text{iff } \mathbb{E}[\![\mathscr{E}]\!]_{ms} = \mathbb{E}[\![\mathscr{E}']\!]_{ms}\end{aligned}$$

(d) Semantics of multiset constraints

Figure 3.3: Expressions and constraints in the multiset abstract domain

Numerical abstract domain The first parameter domain is the numerical abstract domain, $\mathbb{D}^\sharp_n$. For the purpose of the sequence domain, we assume that this domain implements the same signature as the one specified in the last chapter (page 36). Additionally, we require this domain to provide an abstract operator for comparison between symbolic variables $\mathbf{compare}^\sharp_n : \mathcal{V} \times \{=; <; \leqslant; >; \geqslant\} \times \mathbb{D}^\sharp_n \to \wp(\mathcal{V})$. This operator takes as argument a numerical symbolic variable $\alpha \in \mathcal{V}$, a comparison operator $\bowtie$, and a numerical abstract value $\sigma^\sharp_n$, and returns a set of symbolic variable such that the (in)-equality holds between α and all the symbolic variables in the returned set. That is to say:

$$\mathbf{compare}^\sharp_n(\alpha, \bowtie, \sigma^\sharp_n) \subseteq \left\{\beta \in \mathcal{V} \mid \forall \sigma_n \in \gamma_n(\sigma^\sharp_n), \sigma_n \vDash_n \alpha \bowtie \beta\right\}$$

Multiset abstract domain The second parameter abstract domain is used to perform reasoning on content of sequences regardless of their order of appearance. That is to say on the multiset variables $\mathtt{mset}_S$. This multiset abstract domain, written $\mathbb{D}^\sharp_{ms}$, provides a concretization function $\gamma_{ms} : \mathbb{D}^\sharp_{ms} \to \wp(\mathcal{V} \to \mathbb{V} \times \mathcal{V}_m \to \mathcal{M}(\mathbb{V}))$. This domain must be able to manipulate multiset expressions that describe the content of sequence expressions. Additionally, it must represent two kinds of constraints. The first ones are the multiset equality constraints, *i.e.* equality between two multisets expressions. The second kind of constraints are membership constraints. The syntax and semantics of multiset expressions and constraints are presented in Figure 3.3.

Furthermore, we assume that $\mathbb{D}^\sharp_{ms}$ implements some abstract operators. Since this domain only expresses constraints on multiset, it is not required to provide an assignment operator. Note that the $\mathbf{guard}^\sharp_{ms}$, and $\mathbf{sat}^\sharp_{ms}$ operators both input a multiset constraint. Moreover, given that this domain manipulates both numerical and multiset symbolic variables, the signatures of $\mathbf{supp}^\sharp_{ms}$ and $\mathbf{prune}^\sharp_{ms}$ are expended accordingly. The signature of the multiset abstract domain $\mathbb{D}^\sharp_{ms}$ is listed in Figure 3.4.

In the following examples, we will represent elements of parameters domains as a finite conjunction of numerical and multiset constraints, respectively.

3.2.2 Definition and concretization

We can now define the elements of the sequence abstract domain.

Definition 3.4: Sequence abstract domain

An *abstract sequence value* $\sigma^\sharp$ is either a bottom value, $\bot^\sharp_s$, denoting the empty set of valuations, or a tuple $(\sigma^\sharp_n, \sigma^\sharp_{ms}, \sigma^\sharp_s)$, where:

- $\sigma^\sharp_n, \sigma^\sharp_{ms}$ are non-bottom abstract values from the numerical and the multiset parameter

- $\mathbf{guard}^\sharp_{ms} : \mathcal{C}_{ms} \times \mathbb{D}^\sharp_{ms} \to \mathbb{D}^\sharp_{ms}$
- $\sqcup^\sharp_{ms} : \mathbb{D}^\sharp_{ms} \times \mathbb{D}^\sharp_{ms} \to \mathbb{D}^\sharp_{ms}$
- $\nabla^\sharp_{ms} : \mathbb{D}^\sharp_{ms} \times \mathbb{D}^\sharp_{ms} \to \mathbb{D}^\sharp_{ms}$
- $\sqsubseteq^\sharp_{ms} : \mathbb{D}^\sharp_{ms} \times \mathbb{D}^\sharp_{ms} \to \{\mathbf{true}, \mathbf{false}\}$
- $\top^\sharp_{ms} : \mathbb{D}^\sharp_{ms}$
- $\bot^\sharp_{ms} : \mathbb{D}^\sharp_{ms}$
- $\mathbf{sat}^\sharp_{ms} : \mathcal{C}_{ms} \times \mathbb{D}^\sharp_{ms} \to \{\mathbf{true}, \mathbf{false}\}$
- $\mathbf{supp}^\sharp_{ms} : \mathbb{D}^\sharp_{ms} \to \wp_{fin}(\mathcal{V} \uplus \mathcal{V}_m)$
- $\mathbf{prune}^\sharp_{ms} : \mathbb{D}^\sharp_{ms} \times (\mathcal{V} \uplus \mathcal{V}_m) \to \mathbb{D}^\sharp_{ms}$

Figure 3.4: Signature of the multiset abstract domain $\mathbb{D}^\sharp_{ms}$

$$\left(\begin{array}{l} \mathtt{max}_{S_1} = \beta \geqslant \mathtt{max}_S \\ \wedge\, \mathtt{len}_S \geqslant \mathtt{len}_{S_2} = 0 \\ \wedge\, \mathtt{len}_{S_0} = \mathtt{len}_{S_1} \\ \wedge\, \mathtt{len}_{S_0} = \mathtt{len}_S + 1 \\ \wedge\, \mathtt{min}_{S_0} \leqslant \mathtt{min}_S \\ \wedge\, \mathtt{max}_S \leqslant \mathtt{min}_{S_0} \end{array} , \begin{array}{l} \mathtt{mset}_{S_0} = \{\alpha\} \uplus \mathtt{mset}_S \\ \wedge\, \mathtt{mset}_{S_1} = \{\beta\} \uplus \mathtt{mset}_S \\ \wedge\, \mathtt{mset}_{S_2} = \{\} \end{array} , \begin{array}{l} S_2 = [\,] \\ \wedge\, S_0 = \mathbf{sort}([\alpha].S) \\ \wedge\, S_1 = S.[\beta] \\ \wedge\, S = \mathbf{sort}(S) \wedge S_0 = \mathbf{sort}(S_0) \\ \wedge\, S_1 = \mathbf{sort}(S_1) \wedge S_2 = \mathbf{sort}(S_2) \\ \wedge\, \mathbf{unique}(S_1) \\ \wedge\, \mathbf{unique}(S_2) \end{array}\right)$$

Figure 3.5: Example of sequence abstract value

domains, respectively,

- $\sigma^\sharp_s$ is a (possibly empty) finite conjunction of sequence constraints.

We write $\mathbb{D}^\sharp_s$ for the set of abstract sequence values.

By employing the concretization of the parameter domains and the satisfiability relation of sequence constraints, we can now introduce the concretization function of the sequence abstract domain.

Definition 3.5: Abstract sequence domain concretization

The concretization of the sequence abstract domain is:

$$\begin{array}{rcll} \gamma_s : & \mathbb{D}^\sharp_s & \longrightarrow & \wp(\mathbb{D}_s) \\ & \bot^\sharp_s & \longmapsto & \varnothing \\ & (\sigma^\sharp_n, \sigma^\sharp_{ms}, \bigwedge_i C_i) & \longmapsto & \left\{ (\sigma_n, \sigma_m, \sigma_s) \;\middle|\; \begin{array}{l} \sigma_n \in \gamma_n(\sigma^\sharp_n) \\ (\sigma_n, \sigma_m) \in \gamma_{ms}(\sigma^\sharp_{ms}) \\ \forall i, (\sigma_n, \sigma_s) \vDash_s C_i \end{array} \right\} \end{array}$$

Example 3.4: Abstract sequence value

Figure 3.5 presents an example of an abstract sequence value. The concrete state from Figure 3.1 is a member of the concretization of this sequence abstract state.

Example 3.5: Top sequence abstract state

Let us consider the abstract sequence state $(\top^\sharp_n, \top^\sharp_{ms}, \varnothing)$. It expresses no numerical, nor multiset, nor sequences constraints. Therefore, its concretization contains all possible concrete states. This element is written $\top^\sharp_s$.

3.2.3 Machine representation of sequence constraints

Though we defined two kinds of sequence constraints (uniqueness constraints and definitions), we may distinguish several sorts of definitions constraints. Indeed, uniqueness constraints $\mathbf{unique}(S)$ can already be efficiently represented using a finite set $\mathfrak{U}$. This set contains all sequence variables known to be free of repeating elements. Distinguishing further definitions constraints facilitates the representation of such constraints by the sequence abstract domain.

3.2.3.1 Equality constraints between sequence variables

In a set of sequence constraints, equalities between sequence variables (*i.e.* constraints of the representation $S = S'$) can be saturated in order to obtain an equivalence relation. Indeed, the satisfiability of these constraints boils down to the equality of the valuation of the sequence variables. Therefore, the set of constraints denoting equalities between sequence variables, can be represented using a union-find data structure. This allows the domain to compute the symmetric and transitive closure of the equality relation in an efficient manner.

3.2.3.2 Empty and sorted variables

Another example of sequence constraints that can be efficiently represented are emptiness (*i.e.* definitions of the representation $S = []$) as well as sortedness constraints (*i.e.* constraints of the representation $S = \mathbf{sort}(S)$). Such constraints can be represented by the finite set of sequence variables that meet them. In the sequence abstract domain, the set of sequences known to be empty is written $\mathfrak{E}$, and the set of sorted variables known to be sorted $\mathfrak{S}$.

3.2.3.3 Other definitions

This class of definition constraints contains all the definitions that do not fit in any of the class listed above. Such definitions are represented by a partial map assigning to each sequence variable a non-empty list of its known definitions. Such a map is written $\mathfrak{D}$.

Additionally, we enforce several invariants on the content of this map.

(I) The definition map $\mathfrak{D}$ must not contain emptiness or sortedness constraints, as well as equality between variables.

(II) The set $\mathfrak{R}$ represents an equivalence relation. In the map of definitions, only sequence variables that are class representative can occur.

(III) Sequence variables that are known to denote the empty sequence, *i.e.* elements of $\mathfrak{E}$, do not occur in any definition.

(IV) Definitions must be compacted. This means that if there exists a definition $S = E$ in $\mathfrak{D}$, then any occurrence of E in other definitions should be replaced by S.

(V) Definitions must not contain mutually cyclic constraints.

Each of these invariants corresponds to some operation that must be performed when a new definition is added. These operations will be presented in details in Section 3.4.1.

Sequence definitions expressing that a sequence variable is empty, sorted or equal to another sequence variable are called *simple definitions*. Definitions that do not fall into these categories are referred as *generic definitions*. Finally, uniqueness constraints or simple definitions are termed as *simple constraints*. Figure 3.6 presents the Euler diagram describing this categorization of sequence constraints.

Thanks to this classification, we can define a machine representation of sequence abstract values. This representation proves useful to define abstract operators in the subsequent sections.

Definition 3.6: Machine representation of an abstract state

The *machine representation* of a finite conjunction of sequence constraints is formed by a tuple $(\mathfrak{R}, \mathfrak{E}, \mathfrak{S}, \mathfrak{U}, \mathfrak{D})$, where:

- $\mathfrak{R} \subseteq \mathcal{V}_s^2$ is an equivalence relation denoting equality constraints between sequence variables.
- $\mathfrak{E}, \mathfrak{S}, \mathfrak{U} \subseteq \mathcal{V}$ are finite sets of sequence variables that are known to be, respectively, empty, sorted, and without repetitions.
- $\mathfrak{D} \in \mathcal{V}_s \rightharpoonup (\mathcal{E}_s)^+$ is a map assigning to each sequence variable its known definitions.

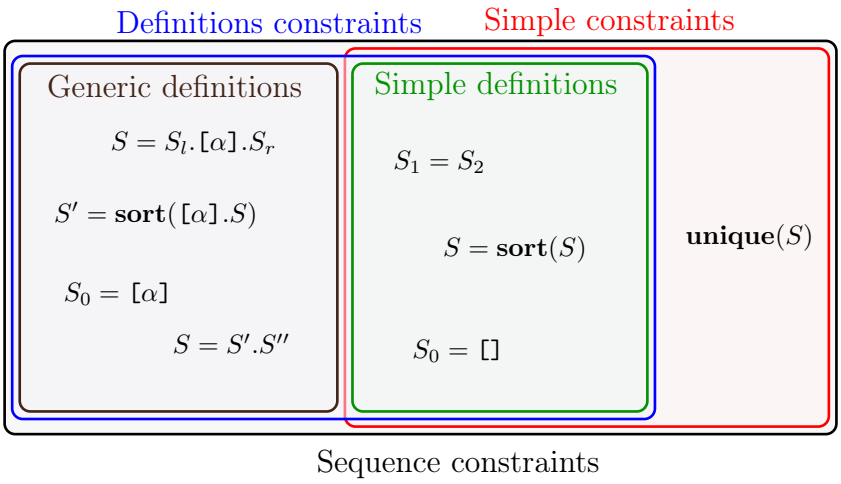


Figure 3.6: Classification of sequence constraints

Example 3.6: Machine representation of the abstract state from Figure 3.5

Let us reconsider the abstract state presented in Figure 3.5. The machine representation of its conjunction of sequence constraints is:

$$\left(\mathfrak{R} = \varnothing, \mathfrak{E} = \{S_2\}, \mathfrak{S} = \{S, S_0, S_1, S_2\}, \mathfrak{U} = \{S_1, S_2\}, \mathfrak{D} = \left\{ \begin{array}{l} S_1 \mapsto \mathbf{sort}([\alpha].S) \\ S_2 \mapsto S.[\beta] \end{array} \right\}\right)$$

To facilitate the reading of machine representation, we label elements of the tuple with their name. Additionally, we display only non-trivial equality classes of $\mathfrak{R}$, *i.e.* equality classes with at least two elements. So $\mathfrak{R} = \varnothing$, means that it contains only singleton classes.

3.3 Operators for the management of symbolic variables

This section presents abstract operators dealing with symbolic variables. Namely, the support operator computing an over-approximation of the symbolic variables constrained by a sequence abstract state, and the pruning operator that removes a symbolic variable from the support of the abstract value.

3.3.1 Support

A simple heuristic to compute the set of variables constrained in an abstract state boils down to the collection of all the symbolic variables appearing in sequence constraints stored in the abstract state.

Definition 3.7: Support of abstract sequence values

The support of a sequence expression is:

$$\begin{aligned} \mathbf{supp}^\sharp_s &: \mathcal{E}_s \to \wp(\mathcal{V} \uplus \mathcal{V}_s) \\ \mathbf{supp}^\sharp_s([]) &:= \varnothing \\ \mathbf{supp}^\sharp_s([\alpha]) &:= \{\alpha\} \\ \mathbf{supp}^\sharp_s(S) &:= \{S\} \\ \mathbf{supp}^\sharp_s(E_1.E_2) &:= \mathbf{supp}^\sharp_s(E_1) \cup \mathbf{supp}^\sharp_s(E_2) \\ \mathbf{supp}^\sharp_s(\mathbf{sort}(E)) &:= \mathbf{supp}^\sharp_s(E) \end{aligned}$$

The support of a sequence constraint is defined as :

$$\begin{aligned} \mathbf{supp}^\sharp_s &: \mathcal{C}_s \to \wp(\mathcal{V} \uplus \mathcal{V}_s) \\ \mathbf{supp}^\sharp_s(\mathbf{unique}(S)) &:= \{S\} \\ \mathbf{supp}^\sharp_s(S = E) &:= \{S\} \cup \mathbf{supp}^\sharp_s(E) \end{aligned}$$

The support of a sequence abstract state is:

$$\begin{aligned}\mathbf{supp}^\sharp_s &: \mathbb{D}^\sharp_s \to \wp(\mathcal{V} \uplus \mathcal{V}_m \uplus \mathcal{V}_s)\\ \mathbf{supp}^\sharp_s(\bot^\sharp_s) &:= \varnothing\\ \mathbf{supp}^\sharp_s(\sigma^\sharp_n, \sigma^\sharp_{ms}, \textstyle\bigwedge_i C_i) &:= \mathbf{supp}^\sharp_n(\sigma^\sharp_n) \cup \mathbf{supp}^\sharp_{ms}(\sigma^\sharp_{ms}) \cup \left(\textstyle\bigcup_i \mathbf{supp}^\sharp_s(C_i)\right)\end{aligned}$$

Note that when a sequence variable S is present in a constraint, the definition of the support does not add the corresponding attribute variables ($\texttt{len}_S, \texttt{min}_S, \texttt{max}_S$, and $\texttt{mset}_S$). Adding attribute variables is not mandatory to define a sound support operator, because of the consistency constraints over the concrete sequence domain $\mathbb{D}_s$. Indeed, if two concrete states map a sequence variable to the same sequence of values, then the attribute variables of this sequence variable have the same values in both concrete states.

Theorem 3.1: Soundness of $\mathbf{supp}^\sharp_s$

The $\mathbf{supp}^\sharp_s$ operator is sound. That is to say for any abstract value $\sigma^\sharp$, and any pair of concrete states $\sigma = (\sigma_n, \sigma_m, \sigma_s)$ and $\sigma' = (\sigma'_n, \sigma'_m, \sigma'_s)$, if both concrete states coincide on all variables in the support of $\sigma^\sharp$, *i.e.* their evaluation of symbolic variables in $\mathbf{supp}^\sharp_s(\sigma^\sharp)$ are equal, then either both should be in the concretization of $\sigma^\sharp$ or none should be. That is to say:

$$\left.\begin{array}{l}\forall \alpha \in \mathbf{supp}^\sharp_s(\sigma^\sharp) \cap \mathcal{V}, \sigma_n(\alpha) = \sigma'_n(\alpha)\\ \forall \mathscr{M} \in \mathbf{supp}^\sharp_s(\sigma^\sharp) \cap \mathcal{V}_m, \sigma_m(\mathscr{M}) = \sigma'_m(\mathscr{M})\\ \forall S \in \mathbf{supp}^\sharp_s(\sigma^\sharp) \cap \mathcal{V}_s, \sigma_s(S) = \sigma'_s(S)\end{array}\right\} \implies \left(\sigma \in \gamma_s(\sigma^\sharp) \Leftrightarrow \sigma' \in \gamma_s(\sigma^\sharp)\right)$$

Proof. Regarding the support operator for sequence expressions, we prove by structural induction over E that for any valuations (σ_n, σ_s) and (σ'_n, σ'_s), if their restrictions to $\mathbf{supp}^\sharp_s(E)$ are equal, then $\mathbb{E}[\![E]\!]_s(\sigma_n, \sigma_s) = \mathbb{E}[\![E]\!]_s(\sigma'_n, \sigma'_s)$.

The remaining of the proof is straightforward. □

Example 3.7: Support of sequence abstract state

To demonstrate the support of a sequence abstract state, let us consider the abstract state from Figure 3.5. The support of all sequence constraints in this state is $\{\alpha, \beta, S, S_0, S_1, S_2, \}$. Additionally, if we add the support of numerical and multiset abstract values, we obtain:

$$\left\{\begin{array}{l}\alpha, \beta, S, S_0, S_1, S_2,\\ \texttt{mset}_S, \texttt{mset}_{S_0}, \texttt{mset}_{S_1}, \texttt{mset}_{S_2},\\ \texttt{len}_S, \texttt{len}_{S_0}, \texttt{len}_{S_1}, \texttt{len}_{S_2}, \texttt{min}_S, \texttt{max}_S, \texttt{min}_{S_0}, \texttt{max}_S, \texttt{max}_{S_2}\end{array}\right\}$$

The first line corresponds to variables in the support of sequence constraints. The second and third lines correspond to variables added by respectively the support of the multiset and numerical parts of the abstract value.

3.3.2 Abstract state pruning

We now introduce the $\mathbf{prune}^\sharp_s$ operator which discards a symbolic variable from an abstract state. To simplify the presentation, we consider the three types of symbolic variables (numerical, multiset, sequence) independently. Note that the variable to remove must not be an attribute variable. In other words, if the variable is a numeric or a multiset symbolic variable, it should not represent the length, the minimum, the maximum or the content of a sequence variable.

3.3.2.1 Removing a numerical symbolic variable

In order to take a numerical symbolic variable α out of a sequence abstract value $(\sigma^\sharp_n, \sigma^\sharp_{ms}, \sigma^\sharp_s)$, the sequence abstract domain considers two cases.

- The first case corresponds to the one where there exists a symbolic variable β that is known to be equal to α in $\sigma^\sharp_n$. In this circumstance, all instances of α are replaced by β in every constraint.
- Otherwise, if no such replacement variable is identified, the domain simply eliminates all definition constraints where α appears.

In both cases, the numerical and multiset parts of the abstract are pruned as well. The definition of $\mathbf{prune}^\sharp_s$ in the case of the removal of numerical variable is presented in Figure 3.7a.

$$\begin{array}{rccl}\mathbf{prune}_s^\sharp : & \mathbb{D}_s^\sharp \times \mathcal{V} & \longrightarrow & \mathbb{D}_s^\sharp \\ & \bot_s^\sharp, \_ & \longmapsto & \bot_s^\sharp \\ (\sigma_n^\sharp, \sigma_{ms}^\sharp, \bigwedge_i C_{s,i}), \alpha & \longmapsto & \left\{ \begin{array}{ll} \left(\begin{array}{l} \mathbf{prune}_n^\sharp(\sigma_n^\sharp, \alpha) \\ \mathbf{prune}_{ms}^\sharp(\sigma_{ms}^\sharp, \alpha) \\ \bigwedge_i C_i[\beta/\alpha] \end{array} \right) & \text{where } \beta \in \mathbf{compare}_n^\sharp(\sigma_n^\sharp, \alpha, =) \setminus \{\alpha\} \\ \left(\begin{array}{l} \mathbf{prune}_n^\sharp(\sigma_n^\sharp, \alpha) \\ \mathbf{prune}_{ms}^\sharp(\sigma_{ms}^\sharp, \alpha) \\ \bigwedge_{i, \alpha \notin \mathbf{fv}(C_i)} C_i \end{array} \right) & \text{otherwise} \end{array} \right. \end{array}$$

(a) Numerical symbolic variable case

$$\begin{array}{rccl}\mathbf{prune}_s^\sharp : & \mathbb{D}_s^\sharp \times \mathcal{V}_m & \longrightarrow & \mathbb{D}_s^\sharp \\ & \bot_s^\sharp, \_ & \longmapsto & \bot_s^\sharp \\ & (\sigma_n^\sharp, \sigma_{ms}^\sharp, \sigma_s^\sharp), \mathscr{M} & \longmapsto & (\sigma_n^\sharp, \mathbf{prune}_{ms}^\sharp(\sigma_{ms}^\sharp, \mathscr{M}), \sigma_s^\sharp) \end{array}$$

(b) Multiset symbolic variable case

$$\begin{array}{rcl}\mathbf{prune}_s^\sharp : (\mathcal{C}_s)^* \times \mathcal{V}_s & \longrightarrow & (\mathcal{C}_s)^* \\ (\mathfrak{R}, \mathfrak{E}, \mathfrak{S}, \mathfrak{U}, \mathfrak{D}), S & \longmapsto & (\mathfrak{R} \setminus (\{S\} \times \mathcal{V}_s), \mathfrak{E} \setminus \{S\}, \mathfrak{S} \setminus \{S\}, \mathfrak{U} \setminus \{S\}, \mathfrak{D}') \end{array}$$

$$\text{where } \mathfrak{D}' := \left\{ \begin{array}{ll} \mathfrak{D}[S'/S] & \text{where } \begin{array}{l} S \neq S' \\ \wedge (S, S') \in \mathfrak{R} \end{array} \\ \bigcup_{\substack{S' \mapsto E' \in \mathfrak{D} \\ S' \neq S}} \bigcup_{S \mapsto E \in \mathfrak{D}} \{S' \mapsto E'[E/S]\} & \text{otherwise} \end{array} \right.$$

(c) Sequence symbolic variable case

Figure 3.7: Definition of $\mathbf{prune}_s^\sharp$

3.3.2.2 Removing a multiset symbolic variable

When the variable to be eliminated is a multiset variable, the $\mathbf{prune}_s^\sharp$ operator boils down to apply the corresponding operator in the multiset part of the abstract state. This corresponds to the definition provided in Figure 3.7b.

3.3.2.3 Removing a sequence variable

To remove a sequence variable S from a conjunction of constraints in machine representation the domain considers two cases to compute the resulting map of definitions.

- The first case arises when there exists another sequence variable S' that is known to be equal to S, *i.e.* $(S, S') \in \mathfrak{R}$. In this situation, all occurrences of S are replaced by S' in the definition part of the machine representation.
- When no such variable exists, the set of definitions is computed by replacing all occurrences of S in definitions of other sequences by all known definitions of S. This implies that if S has no definitions, then all definitions containing S are dropped.

In all cases, the equality relation $\mathfrak{R}$ and the sets of empty, sorted, and unique variables are updated by removing S.

Example 3.8: Removing sequence variables

To illustrate the removing of a sequence variable, let us consider the following conjunction of sequence constraints in machine representation.

$$\left(\mathfrak{R} = \{S \sim S_0\}, \mathfrak{S} = \{S, S_0, S_1, S_2, S_3, S_4\}, \mathfrak{E} = \varnothing, \mathfrak{U} = \varnothing, \mathfrak{D} = \left\{ \begin{array}{l} S \mapsto S_1.[\alpha].S_2;\ S_3.S_4 \\ S' \mapsto [\beta].S \\ S'' \mapsto S.[\delta];\ S_2.S_5 \end{array} \right\} \right)$$

We now consider the case of removing S from this machine representation. The equality class of S in $\mathfrak{R}$ contains another sequence variable S_0. Therefore, S is substituted by S_0 in $\mathfrak{D}$ and all

occurrences of S in other elements of the machine representation are discarded. The outcome of this suppression is:

$$\left(\mathfrak{R} = \varnothing, \mathfrak{S} = \{S_0, S_1, S_2, S_3, S_4\}, \mathfrak{E} = \varnothing, \mathfrak{U} = \varnothing, \mathfrak{D} = \left\{ \begin{array}{l} S_0 \mapsto S_1.[\alpha].S_2;\ S_3.S_4 \\ S' \mapsto [\beta].S_0 \\ S'' \mapsto S_0.[\delta];\ S_2.S_5 \end{array} \right\}\right)$$

We highlight in blue parts of $\mathfrak{D}$ that were modified by the suppression of S.

Now, let us continue this example by eliminating the sequence variable S_0. There is no other sequence variable in the equality class of S_0 in $\mathfrak{R}$. But since there exist known definitions of S_0 in $\mathfrak{D}$, all occurrences of S_0 are replaced by its possible definitions. Since the abstract state contains two possible definitions of S_0, each occurrence of S_0 creates two new definitions. We obtain the following machine representation:

$$\left(\mathfrak{R} = \varnothing, \mathfrak{S} = \{S_1, S_2, S_3, S_4\}, \mathfrak{E} = \varnothing, \mathfrak{U} = \varnothing, \mathfrak{D} = \left\{ \begin{array}{l} S' \mapsto [\beta].S_1.[\alpha].S_2;\ [\beta].S_3.S_4; \\ S'' \mapsto S_1.[\alpha].S_2.[\delta];\ S_3.S_4.[\delta];\ S_2.S_5 \end{array} \right\}\right)$$

Finally, we consider the case where we remove S_1. The equality class of S_1 contains no other sequence variable, and S_1 has no known definitions in $\mathfrak{D}$. Therefore, we discard all definitions containing S_1. This yields:

$$\left(\mathfrak{R} = \varnothing, \mathfrak{S} = \{S_2, S_3, S_4\}, \mathfrak{E} = \varnothing, \mathfrak{U} = \varnothing, \mathfrak{D} = \left\{ \begin{array}{l} S' \mapsto [\beta].S_3.S_4; \\ S'' \mapsto S_3.S_4.[\delta];\ S_2.S_5 \end{array} \right\}\right)$$

Theorem 3.2: Soundness of $\mathbf{prune}_s^\sharp$

For any abstract state $\sigma^\sharp \in \mathbb{D}_s^\sharp$, and any symbolic variable $v \in \mathcal{V} \uplus \mathcal{V}_m \uplus \mathcal{V}_s$, the following statements hold:

- $\gamma_s(\sigma^\sharp) \subseteq \gamma_s(\mathbf{prune}_s^\sharp(\sigma^\sharp, v))$
- $\mathbf{supp}_s^\sharp \circ \mathbf{prune}_s^\sharp(\sigma^\sharp, v) = \mathbf{supp}_s^\sharp(\sigma^\sharp) \setminus \{v\}$

Proof. The soundness of $\mathbf{prune}_s^\sharp$ stems from the soundness of the parameter domains operators and from the following *substitution lemma*.

Lemma 3.1: Substitution lemma

For any sequence concrete state $\sigma = (\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s$, and definition constraints $S = E$ and $S' = E'$, if $\sigma_n, \sigma_s \vDash_s S' = E'$, then $\mathbb{E}[\![E]\!]_s(\sigma_n, \sigma_s) = \mathbb{E}[\![E[E'/S']]\!]_s(\sigma_n, \sigma_s)$, and in particular $\sigma_n, \sigma_s \vDash_s S = E$ if and only if $\sigma_n, \sigma_s \vDash_s S = E[E'/S']$.

The proof of the second point comes from the fact that the variable is syntactically removed of the set of constraints. □

3.4 Operators to add and verify constraints

This section presents two abstract operators for abstract sequence states. Both operators take an abstract state and a constraint as input. The first operator, $\mathbf{guard}_s^\sharp$, refines the input abstract value with the provided constraint. The second operator, $\mathbf{sat}_s^\sharp$, determines whether the abstract state entails the constraint.

3.4.1 Adding a new sequence constraint

First, let us consider the *abstract sequence condition* operator $\mathbf{guard}_s^\sharp : \mathbb{D}_s^\sharp \times \mathcal{C}_s \to \mathbb{D}_s^\sharp$. This operator refines an abstract state thanks to a new sequence constraint. For this operator, the *bottom case* is immediate. An infeasible abstract state remains infeasible when a new constraint is added. As a consequence, we define for any constraint C, $\mathbf{guard}_s^\sharp(\bot_s^\sharp, C) := \bot_s^\sharp$. For the non-bottom case, a straightforward implementation might just add the new constraint to the conjunction of sequence

constraints. That is to say: $\mathbf{guard}_s^\sharp((\sigma_n^\sharp, \sigma_{ms}^\sharp, \bigwedge_i C_i), C) := (\sigma_n^\sharp, \sigma_{ms}^\sharp, C \wedge \bigwedge_i C_i)$. However, the latter would be too imprecise. Indeed, the conjunction $C \wedge \bigwedge_i C_i$ may be inconsistent. In that case, the operator should return $\bot_s^\sharp$. Moreover, the conjunction may entail other constraints that are more precise than any constraint in the conjunction. Finally, this may violate the invariants **(I)**, **(II)**, **(III)**, **(IV)**, and **(V)** defined in Section 3.2.3.

Our implementation leverages the machine representation $(\mathfrak{R}, \mathfrak{E}, \mathfrak{S}, \mathfrak{U}, \mathfrak{D})$ of a conjunction of constraints. At a high level, the $\mathbf{guard}_s^\sharp$ operator proceeds through the five following steps:

1. *Normalization and compaction* rewrites the new constraint to maintain consistency with the invariants satisfied by the abstract value.
2. *Insertion* adds the new definition in the relevant part of the machine representation. Cyclic constraints are removed if necessary.
3. *Parameter domains translation* adds in the numerical and multiset parts of the abstract value numerical and multiset constraints that are entailed by the new sequence constraint.
4. *Constraint propagation* extends emptiness, sortedness and uniqueness constraints to subsequences.
5. *Constraint saturation* tries to infer new constraints by using heuristics.

Example 3.9: Assuming a new sequence constraint

Before presenting in depth the steps followed by $\mathbf{guard}_s^\sharp$, let us outline the addition of the constraint $S_0 = [\beta].S_2.S_3$ in the following sequence abstract state:

$$\sigma^\sharp = \left(\sigma_n^\sharp, \sigma_{ms}^\sharp, \mathfrak{R} = \{S \sim S_0\}, \mathfrak{E} = \{S_3\}, \mathfrak{S} = \{S, S_0, S_3, S', S''\}, \mathfrak{D} = \left\{ \begin{array}{l} S \mapsto [\alpha].S'.S'' \\ S_1 \mapsto [\beta].S_2.S_4 \end{array} \right\}\right)$$

The first step, **normalization and compaction**, rewrites the new constraint to satisfy the invariants. Here, the sequence denoted by the variable S_3 is empty, it is removed from the definition. Additionally, the class representative of the equality class containing the variable S_0 is S. So, S_0 is substituted by S. After the first step, the resulting constraint is: $S = [\beta].S_2$.

The second step, **insertion** adds the resulting constraint in $\mathfrak{D}$. Observe that the new definition is a sub-expression of the definition of S_1. Therefore, the domain compacts the latter using the former. The resulting abstract state is:

$$\left(\sigma_n^\sharp, \sigma_{ms}^\sharp, \mathfrak{R} = \{S \sim S_0\}, \mathfrak{E} = \{S_3\}, \mathfrak{S} = \{S, S_0, S_3, S', S''\}, \mathfrak{D} = \left\{ \begin{array}{l} S \mapsto [\alpha].S'.S''; \ [\beta].S_2 \\ S_1 \mapsto S.S_4 \end{array} \right\}\right)$$

The third step, **parameter domain translation**, add multiset and numerical constraints implied by $S = [\beta].S_2$. Preservation of content entails the multiset constraint is $\mathtt{mset}_S = \{\beta\} \uplus \mathtt{mset}_{S_2}$. Similarly, conservation of length implies the constraint $\mathtt{len}_S = 1 + \mathtt{len}_{S_2}$. Finally, since S is sorted, and its first element is denoted by the symbolic variable β, the domain asserts that β is the minimal value of S, and that all values in S_2 are greater than β. That is to say, the domain guard the numerical constraint $\mathtt{min}_S = \beta \leqslant \mathtt{min}_{S_2}$. Finally, all elements in S_2 are also elements of S. This implies the following constraint $\mathtt{max}_{S_2} \leqslant \mathtt{max}_S$.

The fourth step, **constraint propagation**, extends sortedness, emptiness and uniqueness constraints to subsequences. Here, since S_2 is a subsequence of the sorted sequence denoted by S, it is sorted as well. Therefore, S_2 is added to the set $\mathfrak{S}$.

The final step, **constraint saturation** tries to infer new constraints. Here by matching the two possible definitions of S, the domain concludes that $\alpha = \beta$ and $S_2 = S'.S''$. The first constraint is added in the numerical part of the abstract value. The second one is added by calling $\mathbf{guard}_s^\sharp$ recursively.

This new constraint allows the domain to compact the definitions of S into a single one $S = [\beta].S_2$. Additionally, the parameter translation adds constraints between the attribute variables of S_2, S', and S''. To conclude, the outcome of $\mathbf{guard}_s^\sharp(\sigma^\sharp, S_0 = [\beta].S_2.S_3)$ is:

$$\left(\sigma_n^{\sharp\prime}, \sigma_{ms}^{\sharp\prime}, \mathfrak{R} = \{S \sim S_0\}, \mathfrak{E} = \{S_3\}, \mathfrak{S} = \{S, S_0, S_2, S_3, S', S''\}, \mathfrak{D} = \left\{\begin{array}{l} S \mapsto [\beta].S_2 \\ S_1 \mapsto S.S_4 \\ S_2 \mapsto S'.S'' \end{array}\right\}\right)$$

where:

$$\sigma_n^{\sharp\prime} := \mathbf{guard}_n^\sharp \left(\sigma_n^\sharp, \begin{array}{c} \mathtt{len}_S = 1 + \mathtt{len}_{S_2} \\ \wedge\, \mathtt{len}_{S_2} = \mathtt{len}_{S'} + \mathtt{len}_{S''} \\ \wedge\, \alpha = \beta = \mathtt{min}_S \leqslant \mathtt{min}_{S_2} \leqslant \mathtt{min}_{S'}, \mathtt{min}_{S'} \\ \wedge\, \mathtt{max}_{S'}, \mathtt{max}_{S''} \leqslant \mathtt{max}_{S_2} \leqslant \mathtt{max}_S \end{array}\right)$$

$$\sigma_{ms}^{\sharp\prime} := \mathbf{guard}_{ms}^\sharp \left(\sigma_{ms}^\sharp, \begin{array}{c} \mathtt{mset}_S = \{\beta\} \uplus \mathtt{mset}_{S_2} \\ \wedge\, \mathtt{mset}_{S_2} = \mathtt{mset}_{S'} \uplus \mathtt{mset}_{S''} \end{array}\right)$$

3.4.1.1 Normalization and compaction

Removing empty sequence variables To establish the invariant **(III)**, the normalization process starts by removing all sequence variables that are known to be empty. This corresponds to the following rewriting operation $S = E[[]/S']$ for all sequence variables $S' \in \mathfrak{E}$. Note that, even if the sequence variable S at the left-hand side of the equality is known to be empty, it is not substituted by $[]$.

Using class representative of $\mathfrak{R}$ This step rewrites the constraint to enforce invariant **(II)**, stating that each variable appearing in $\mathfrak{D}$ must be a class representative of $\mathfrak{R}$. Unlike the previous step, this rewriting takes place in both sides of the equality, in the case of a definition constraint.

Definition compaction Given a definition constraint $S = E$ and a conjunction of sequence definitions $\bigwedge S_i = E_i$, such that none of the expressions E_i is the empty sequence nor a single symbolic variable and the conjunction does not contain any cyclic constraints, then the compact representation of defined as $S = E[S_1/E_1] \ldots [S_k/E_k]$.

Since the compaction of one definition may make another definition appear, several compaction steps are required to enforce that no sub-expression on the right-hand side of the equality corresponds to a definition in the conjunction. This process ultimately terminates since the conjunction has no mutually cyclic constraints.

Since the normalization and compaction only replace expressions by sequence variables, their soundness is a direct consequence of the substitution lemma (Lemma 3.1).

Example 3.10: Normalization of a sequence definition

To illustrate the normalization process, let us consider the constraint $S_0 = S'.[\alpha].S''$ in the following conjunction of constraints in machine representation:

$$\left(\mathfrak{R} = \{S_2 \sim S''\}, \mathfrak{E} = \{S'\}, \mathfrak{S} = \{S', S_1\}, \mathfrak{U} = \{S_1\}, \mathfrak{D} = \left\{\begin{array}{l} S \mapsto S_1.S_2 \\ S_1 \mapsto [\alpha] \end{array}\right\}\right)$$

The first rewriting step removes empty sequences. In this example, since S' is known to be empty, the constraint is rewritten into $S_0 = [\alpha].S''$.

In the second step, each sequence variable is changed to its class representative in $\mathfrak{R}$. The class representative of S'' is S_2. It follows that the constraint becomes $S_0 = [\alpha].S_2$.

The final rewriting compacts the constraint according to known definitions in $\mathfrak{D}$. The expression $[\alpha]$ corresponds to a definition of S_1, so the constraint is rewritten into $S_0 = S_1.S_2$. The right-hand side of the equality matches the definition of S. So the definition is compacted.

To conclude, after the normalization process, the constraint boils down to $S_0 = S$.

If the constraint computed by the rewriting steps is already stored in the abstract value, or a trivial constraint such as $S = S$, then $\mathbf{guard}_s^\sharp$ returns the sequence abstract state unchanged.

3.4.1.2 Insertion of the constraint

The second stage of the $\mathbf{guard}_s^\sharp$ operator adds the normalized constraint in the machine representation. This addition may require some rewriting in the machine representation.

Sorted and unique cases In the case where the constraint takes the form of either $S = \mathbf{sort}(S)$ or $\mathbf{unique}(S)$, the insertion boils down to adding S, and its equality class in $\mathfrak{R}$, to either $\mathfrak{S}$ or $\mathfrak{U}$, respectively.

Empty case When the constraint is an emptiness constraint of the form $S = [\,]$, the insertion includes S, and its equivalence class according to $\mathfrak{R}$, in $\mathfrak{E}$, $\mathfrak{S}$, and $\mathfrak{U}$. Furthermore, if $\mathfrak{E}$ already contained some sequence variable S', then the equality classes of S and S' are merged together. Finally, the domain discards all instances of S in definitions stored in $\mathfrak{D}$.

Sequence equality case In the case of an equality between two sequence variables $S = S'$, the equivalence classes of S and S' are merged in $\mathfrak{R}$. Additionally, if either S or S' is a member of $\mathfrak{E}, \mathfrak{S}$ or $\mathfrak{U}$, then the other variable, and its equality class in $\mathfrak{R}$, are added to the corresponding set. Finally, the definitions of S and S' in $\mathfrak{D}$ are merged into a single list, and all occurrences of S and S' in $\mathfrak{D}$ are replaced by new class representative.

Generic definition When the constraint is a definition $S = E$ that does not match any of the cases mentioned above, it is inserted in $\mathfrak{D}$. Subsequently, following the detection and elimination of mutually cyclic constraints (see next paragraph), the definition map $\mathfrak{D}$ undergoes compression due to the addition of the new definition.

Detection of cyclic constraints To uphold invariant **(V)**, *i.e.* the absence of cyclic constraints, the domain performs a depth-first search traversal of the dependency graph represented by the conjunction of definitions.

Definition 3.8: Dependancy graph of a definition map

The *dependency graph* of a definition map $\mathfrak{D}$ is a directed graph (V, E), such that the set of vertices V contains all sequence variables occurring in $\mathfrak{D}$. The set of edges E contains a directed edge $S \to S'$ if and only if there exists a definition $S = E$ in $\mathfrak{D}$ such that S' occurs in E.

If the exploration finds a cycle in the graph, this entails the existence of n definitions $\bigwedge_{i \leqslant i \leqslant n} S_i = E_i$ where S_{i+1} appears in E_i and S_1 in E_n. After inlining all definitions, we obtain a constraint $S_1 = E$, where $E := E_1[E_2/S_2] \dots [E_n/S_n]$. The resulting expressions E contains the sequence variable S_1. At this stage, the domain considers two possible cases:

- If E contains an atom $[\alpha]$, then the constraint is infeasible. In this case, the sequence abstract value is reduced to $\bot_s^\sharp$.
- When E contains only sequence variables, then the constraint $S = E$ is satisfiable if and only if all variables in E, except S_1, are empty. Additionally, this implies that $S_1, \dots, S_n$ are equal.

It is worth noting that the empty and general cases, as well as the elimination of cyclic constraints, alter the definitions in $\mathfrak{D}$. These modifications may convert definitions into simple constraints that are not meant to be stored in $\mathfrak{D}$ but rather in a designated set in the machine representation. When such scenarios are encountered, the constraint is extracted from $\mathfrak{D}$ and reintegrated in the machine representation following the procedure described above. The insertion process stops since each extra iteration corresponds to a definition removed from $\mathfrak{D}$.

Example 3.11: Insertion of a new constraint

To illustrate the insertion step, let us examine the addition of constraint $S_1 = [\alpha].S_0$ in the following machine representation:

$$\left(\mathfrak{R} = \varnothing, \mathfrak{E} = \varnothing, \mathfrak{S} = \varnothing, \mathfrak{U} = \varnothing, \mathfrak{D} = \left\{ \begin{array}{l} S \mapsto S_3.S_1 \\ S_2 \mapsto S_3.[\alpha].S_0 \end{array} \right\}\right)$$

Since the inserted constraint is not a simple definition, the general case applies. The definition

is added in $\mathfrak{D}$.

$$\left(\mathfrak{R}=\varnothing,\mathfrak{E}=\varnothing,\mathfrak{S}=\varnothing,\mathfrak{U}=\varnothing,\mathfrak{D}=\left\{\begin{array}{c}S\mapsto S_3.S_1\\ S_1\mapsto [\alpha].S_0\\ S_2\mapsto S_3.[\alpha].S_0\end{array}\right\}\right)$$

Parts of the machine representation in blue correspond to elements that were modified during the insertion step. Thanks to the definition of S_1, the domain is able to compact the definition of S_2 into $S_2 \mapsto S_3.S_1$. Thanks to this new constraint, the definition of S can be compacted into S_2. At this stage, the abstract state is:

$$\left(\mathfrak{R}=\varnothing,\mathfrak{E}=\varnothing,\mathfrak{S}=\varnothing,\mathfrak{U}=\varnothing,\mathfrak{D}=\left\{\begin{array}{c}S\mapsto S_2\\ S_1\mapsto [\alpha].S_0\\ S_2\mapsto S_3.S_1\end{array}\right\}\right)$$

We notice that the definition of S is a simple equality constraint between sequence variables. Such a constraint must not be kept in $\mathfrak{D}$, but rather in $\mathfrak{R}$. Therefore, it is removed from $\mathfrak{D}$ and added back to $\mathfrak{R}$. This insertion merges the equality classes of S and S_2, and rewrites all occurrences of S_2 into S. To conclude, the result of the insertion is:

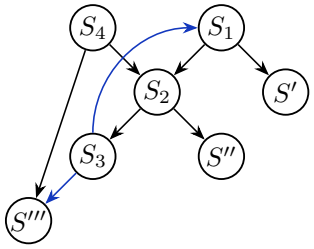
$$\left(\mathfrak{R}=\{S\sim S_2\},\mathfrak{E}=\varnothing,\mathfrak{S}=\varnothing,\mathfrak{U}=\varnothing,\mathfrak{D}=\left\{\begin{array}{c}S\mapsto S_3.S_1\\ S_1\mapsto [\alpha].S_0\end{array}\right\}\right)$$

Example 3.12: Detection and removal of cyclic constraints

To illustrate, the detection and handling of mutually cyclic constraints, let us consider the insertion of constraint $S_3 = S_1.S'''$ in the following machine representation:

$$\left(\mathfrak{R}=\varnothing,\mathfrak{E}=\varnothing,\mathfrak{S}=\varnothing,\mathfrak{U}=\varnothing,\mathfrak{D}=\left\{\begin{array}{c}S_1\mapsto S_2.S'\\ S_2\mapsto S''.S_3\\ S_4\mapsto S_2.S'''\end{array}\right\}\right)$$

After adding the mapping $S_3 \mapsto S_1.S'''$ in $\mathfrak{D}$ we obtain the dependency graph displayed below. Edges corresponding to the new definition are drawn in blue. A cycle containing the sequence variables S_1, S_2 and S_3 is detected.



Inlining the definitions of S_2 and S_3 in the definition of S_1 produces the following definition $S_1 = S''.S_1.S'''.S'$. This constraint entails that S', S'', S''' are empty and that S_1, S_2, and S_3 are equal. So, the definitions of S_1, S_2, and S_3 are removed. This yields the following machine representation:

$$(\mathfrak{R}=\varnothing,\mathfrak{E}=\varnothing,\mathfrak{S}=\varnothing,\mathfrak{U}=\varnothing,\mathfrak{D}=\{S_4\mapsto S_2.S'''\})$$

Then, the constraints $S_1 = S_2 = S_3$ and $S' = S'' = S''' = []$ are inserted.

$$\left(\mathfrak{R}=\left\{\begin{array}{c}S_1\sim S_2\sim S_3\\ S'\sim S''\sim S'''\end{array}\right\},\begin{array}{c}\mathfrak{E}=\{S',S'',S'''\},\\ \mathfrak{S}=\{S',S'',S'''\},\mathfrak{D}=\{S_4\mapsto S_1\}\\ \mathfrak{U}=\{S',S'',S'''\},\end{array}\right)$$

Adding the constraint $S_1 = S_2$ in the abstract state rewrites the definition of S_4 into $S_1.S'''$. Furthermore, adding the emptiness constraint of S''' removed this variable from the definition of S_4. Therefore, the definition of S_4 boils down to S_1. This constraint should not be stored in

$\mathfrak{D}$, but rather in $\mathfrak{R}$. As a consequence, the constraint $S_4 = S_1$ is removed from $\mathfrak{D}$, and inserted back in the abstract state. The outcome of this insertion is:

$$\left(\mathfrak{R} = \left\{\begin{matrix} S_1 \sim S_2 \sim S_3 \sim S_4 \\ S' \sim S'' \sim S''' \end{matrix}\right\}, \begin{matrix} \mathfrak{E} = \{S', S'', S'''\}, \\ \mathfrak{S} = \{S', S'', S'''\}, \mathfrak{D} = \varnothing \\ \mathfrak{U} = \{S', S'', S'''\}, \end{matrix}\right)$$

Example 3.13: Detection and removal of cyclic constraints

Now, let us consider the abstract state from the previous example where the definition of S_2 is $S_2 = [\alpha].S_3$, that is to say:

$$\left(\mathfrak{R} = \varnothing, \mathfrak{E} = \varnothing, \mathfrak{S} = \varnothing, \mathfrak{U} = \varnothing, \mathfrak{D} = \left\{\begin{matrix} S_1 \mapsto S_2.S' \\ S_2 \mapsto [\alpha].S_3 \\ S_3 \mapsto S_1.S''' \\ S_4 \mapsto S_2.S''' \end{matrix}\right\}\right)$$

Then, inlining the definitions of S_2 and S_3 computes $S_1 = [\alpha].S_1.S'''.S''$. This recursive constraint contains the atom $[\alpha]$ and is not feasible. Therefore, the whole sequence abstract state is replaced by $\bot_s^\sharp$.

3.4.1.3 Parameter domain translations

When a new definition $S = E$ is added to the conjunction of sequence constraints, the $\mathbf{guard}_s^\sharp$ operator translates this definition in terms of multiset constraints and numerical constraints. These constraints are then added in the respective parts of the abstract value using the $\mathbf{guard}_n^\sharp$ and $\mathbf{guard}_{ms}^\sharp$ operators. If the outcome of one of these additions is a bottom abstract value, then $\mathbf{guard}_s^\sharp$ automatically returns $\bot_s^\sharp$.

Content translation This translation transforms a sequence definition $S = E$ into a multiset equality constraint $\mathtt{mset}_S = \tau_{ms}(E)$. The function τ_{ms}, defined in Figure 3.8, translates a sequence expression into a multiset one. The outcome of this translation corresponds to a multiset expression denoting the content of E without considering the position of the elements. This translation is sound: any concrete sequence value satisfying the sequence definition satisfies its content translation.

Lemma 3.2: Soundness of τ_{ms}

For any concrete state $(\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s$ and any sequence definition $S = E$,

$$\sigma_n, \sigma_s \vDash_s S = E \Longrightarrow \sigma_n, \sigma_m \vDash_s \mathtt{mset}_S = \tau_{ms}(E)$$

Proof. The proof of this lemma is constructed by induction over E. □

Length translation Similarly, given a sequence definition $S = E$, the domain derives an equality constraint $\mathtt{len}_S = \tau_{\mathtt{len}}(S)$ from the equality of the length of both sides. The definition of $\tau_{\mathtt{len}}$ is presented in Figure 3.9.

Additionally, if the constraint introduces in the sequence abstract a new sequence variable S, the domain adds the constraint stating that its length attribute variable is positive.

This translation is also sound.

Lemma 3.3: Soundness of $\tau_{\mathtt{len}}$

For any concrete state $(\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s$ and any sequence definition $S = E$,

$$\begin{matrix} \forall S' \in \mathbf{fv}(E) \cup \{S\} \sigma_n \vDash_n \mathtt{len}_{S'} \geqslant 0 \\ \wedge\, \sigma_n, \sigma_s \vDash_s S = E \Longrightarrow \sigma_n \vDash_n \mathtt{len}_S = \tau_{\mathtt{len}}(E) \end{matrix}$$

Proof. The proof of this lemma is constructed by induction over E. □

Bounds translation To translate a sequence definition into numerical ones the domain considers several possible cases.

$$\begin{aligned}
\tau_{ms} &: \mathcal{E}_s \to \mathcal{E}_{ms} \\
\tau_{ms}(\texttt{[]}) &= \{\} \\
\tau_{ms}(\texttt{[}\alpha\texttt{]}) &= \{\alpha\} \\
\tau_{ms}(S) &= \texttt{mset}_S \\
\tau_{ms}(E.E') &= \tau_{ms}(E) \uplus \tau_{ms}(E') \\
\tau_{ms}(\mathbf{sort}(E)) &= \tau_{ms}(E)
\end{aligned}$$

Figure 3.8: Content translation

$$\begin{aligned}
\tau_{\texttt{len}} &: \mathcal{E}_s \to \mathcal{E}_n \\
\tau_{\texttt{len}}(\texttt{[]}) &= 0 \\
\tau_{\texttt{len}}(\texttt{[}\alpha\texttt{]}) &= 1 \\
\tau_{\texttt{len}}(S) &= \texttt{len}_S \\
\tau_{\texttt{len}}(E.E') &= \tau_{\texttt{len}}(E) + \tau_{\texttt{len}}(E') \\
\tau_{\texttt{len}}(\mathbf{sort}(E)) &= \tau_{\texttt{len}}(E)
\end{aligned}$$

Figure 3.9: Length translation

$$\frac{S = \mathbf{sort}(S) \qquad S = \dots S' \dots S'' \dots}{\texttt{max}_{S'} \leqslant \texttt{min}_{S''}}$$

$$\frac{S = \mathbf{sort}(S) \qquad S = \dots S' \dots \texttt{[}\alpha\texttt{]} \dots}{\texttt{max}_{S'} \leqslant \alpha} \qquad \frac{S = \mathbf{sort}(S) \qquad S = \dots \texttt{[}\alpha\texttt{]} \dots S' \dots}{\alpha \leqslant \texttt{min}_{S'}}$$

$$\frac{S = \mathbf{sort}(S) \qquad S = \texttt{[}\alpha\texttt{]} \dots}{\alpha = \texttt{min}_S} \qquad \frac{S = \mathbf{sort}(S) \qquad S = \dots \texttt{[}\alpha\texttt{]}}{\alpha = \texttt{max}_S}$$

Figure 3.10: Inference rules for bound translation (sorted case)

General case The general case simply states that when we add a definition $S = E$, then any atom $\texttt{[}\alpha\texttt{]}$ in E is comprised between $\texttt{min}_S$ and $\texttt{max}_S$. Additionally, any sequence variable S' in E has its bounds between the bounds of S.

Lemma 3.4: Soundness of bound translation (general case)

For any concrete state $(\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s$ and any sequence definition $S = E$,

$$\sigma_n, \sigma_s \vDash_s S = E \implies \begin{cases} \forall \alpha \in \mathbf{fv}(E) \cap \mathcal{V}, \sigma_n \vDash_n \texttt{min}_S \leqslant \alpha \leqslant \texttt{max}_S \\ \forall S' \in \mathbf{fv}(E) \cap \mathcal{V}_s, \sigma_n \vDash_n \texttt{min}_S \leqslant \texttt{min}_{S'} \wedge \texttt{max}_{S'} \leqslant \texttt{max}_S \end{cases}$$

Proof. The inequalities between the value of α, and the bound attributes variables of S are immediate.

If the sequence variable S' occurs in E, then we can consider two cases. Either $\sigma_s(S') = \varepsilon$, then the inequalities hold since the case implies that $\texttt{min}_{S'} = +\infty$ and $\texttt{max}_{S'} = -\infty$. Or $\sigma_s(S') \neq \varepsilon$, then the evaluation of E is also non-empty. In that case, the inequalities hold since all elements of S' are elements of S. □

Empty case When the definition states that a sequence variable S is empty, then the domain should the constraints stating that $\texttt{min}_S = +\infty$ and $\texttt{max}_S = -\infty$. However, the numerical domain is not able to express these constraints. To address this problem, we could translate these constraints by stating that any numerical symbolic value is between $\texttt{max}_S$ and $\texttt{min}_S$. But this is too costly. As a consequence, we do not add any numerical constraints at this stage.

Sorted case When the new definition concerns a sequence variable that is known to be sorted, the domain adds constraints between the elements of the definition. Put simply, the numerical constraints express that the maximum of an element in the sequence E must be less than the minimum of any other element to its right. For instance if E is $S_1.S_2.S_3.S_4$ then the constraints are:

$$\begin{array}{lll} \texttt{max}_{S_1} \leqslant \texttt{min}_{S_2} & \texttt{max}_{S_1} \leqslant \texttt{min}_{S_3} & \texttt{max}_{S_1} \leqslant \texttt{min}_{S_4} \\ & \texttt{max}_{S_2} \leqslant \texttt{min}_{S_3} & \texttt{max}_{S_2} \leqslant \texttt{min}_{S_4} \\ & & \texttt{max}_{S_3} \leqslant \texttt{min}_{S_4} \end{array}$$

We obtain a number of inequalities that is quadratic in the size of the expression. The rationale behind not exclusively asserting bound constraints between neighboring sequence variables (*i.e.* $\texttt{max}_{S_i} \leqslant \texttt{min}_{S_{i+1}}$) is explained below.

Moreover, if the leftmost element is an atom, then it is equal to the minimal element of S, and similarly, if the rightmost element is also an atom, it corresponds to its maximal value. The bound constraints added in the sorted case are presented as inference rules in Figure 3.10.

Lemma 3.5: Soundness of bound translation (sorted case)

The inference rules presented in the Figure 3.10 are sound. That is to say, for any inference rule $\frac{P_1 \quad \dots \quad P_i}{Q_1 \quad \dots \quad Q_j}$ and any concrete state $\sigma \in \mathbb{D}_s$, if σ satisfies all the premises $P_1, \dots, P_i$, then it also satisfies all the conclusion $Q_1, \dots, Q_j$.

Proof. The proof if this rule is established similarly as for the non-sorted case. Either S' (or S'') is empty, then the inequalities are trivial. Or, the values of these sequences are non-empty, then the bounds corresponds to some element of S' (or S''). The sortedness of S implies that the inequalities hold. □

For examples of parameter domain translations, we refer the reader to Example 3.9.

3.4.1.4 Constraint propagation

In essence, constraint propagation is a top-down exploration of the dependency graph. The top-down exploration asserts that any subsequence of an empty sequence is empty as well. This principle applies similarly to sorted sequences and to sequences without repetition. This exploration starts when a new sequence variable is asserted to be either empty, sorted, or without repetition, or when a variable in either $\mathfrak{E}, \mathfrak{S}$ or $\mathfrak{U}$ receives a new definition. Figure 3.11 presents the rules used for each case.

Empty case When the abstract domain determines that S is empty, it traverses all its known definitions. It asserts that all sequence variable appearing in any definition is also empty. Additionally, to enforce invariant **(III)** stating that empty sequence variable cannot appear in $\mathfrak{D}$, the definition is removed from $\mathfrak{D}$ after being examined.

When a definition contains an atom, then it is inconsistent: an empty sequence cannot contain one element. In this case the exploration immediately returns $\bot_s^\sharp$.

Unique case When a sequence has no repeating element, then so does any subsequences. Additionally, for two numerical symbolic variables α, β appearing in a definition, the domain asserts that the must be different. In the case where a single numerical symbolic variable α occurs twice in a definition, the constraint boils down to $\alpha \neq \alpha$. Therefore, the $\mathbf{guard}_s^\sharp$ operator returns $\bot_s^\sharp$.

Sorted case For the sorted case, any sequence variable S' in a definition of a sorted sequence must be sorted as well. It is important to note that $S = \dots S' \dots$ notation means that S' must not appear in a call to the **sort** symbolic function. The sorted case also considers the case when a new definition trivially states that a sequence is sorted since the right-hand side is a call to the **sort** symbolic function.

When a new constraint is inferred during constraint saturation, it is inserted in the sequence abstract state, and marked as a starting point for further exploration. Constraint propagation terminates since the only possible modification of the dependency graph involves removing a vertex, *i.e.* a sequence variable. After the graph stabilizes, the top-down exploration stops since the dependency graph is cycle-free.

Lemma 3.6: Soundness of constraint propagation

All the inference rules presented in the Figure 3.11 are sound.

3.4.1.5 Constraint saturation

Constraint saturation infers new constraints that are not explicitly present in the abstract state though they are entailed by it. This step works by examining all definitions of sequence variables that depend on part of the abstract state that was modified by a previous step. This corresponds to a bottom-up exploration of the dependency graph, starting from modified parts of the abstract state. Here, the expression "*all definitions*" means that the constraint saturation also consider definitions computed after inlining others in $\mathfrak{D}$. Exploring all definition ultimately terminates since there is no cyclic constraints.

Additionally, there exists two types of inference rules. *Unary rules* try to infer that a sequence is sorted or empty by examining one of its definition. *Binary rules* infer new definitions by comparing

$$\frac{S = [] \qquad S = E \qquad S' \in \mathbf{fv}(E)}{S' = []} \qquad\qquad \frac{S = [] \qquad S = E \qquad \delta \in \mathbf{fv}(E)}{\bot_s^\sharp}$$

(a) Empty case

$$\frac{\mathbf{unique}(S) \qquad S = E \qquad S' \in \mathbf{fv}(E)}{\mathbf{unique}(S')} \qquad\qquad \frac{\mathbf{unique}(S) \qquad S = E \qquad \alpha, \beta \in \mathbf{fv}(E)}{\alpha \neq \beta}$$

(b) Unique case

$$\frac{S = \mathbf{sort}(S) \qquad S = \ldots.S'.\ldots}{S' = \mathbf{sort}(S')} \qquad\qquad \frac{S = \mathbf{sort}(E)}{S = \mathbf{sort}(S)}$$

(c) Sorted case

Figure 3.11: Constraint propagation inference rules

two possible definitions of a same sequence variable. These rules may also use other information about this variable such that it sortedness or the fact that it contains no repetition.

Unary rules The first kind of rules concerns definitions of variables impacted by the new definition. That is to say equality constraints computed when the new definition has been inlined at some point in the computation. These indeed corresponds to new definitions. Figure 3.12 presents these unary rules.

Empty rules The first category of unary rules concerns emptiness constraints. A definition constraint $S = E$ implies that S is empty when all sequence variables in S are empty. Additionally, the domain might infer emptiness from attribute constraints. Indeed, the sequence domain can interrogate the numerical part, using the $\mathbf{sat}_n^\sharp$ operator to check whether the length of a sequence variable is null, or whether the values of supremum or infimum are inverted. If one of the two constraints are valid in the numerical part, then the sequence domain deduces that S is empty.

Sorted rule The second category of rules pertains to sortedness constraints. To check if a definition $S = S_1 \ldots S_n$ entails the sortedness of S, the domain must check that all sequence variable S_i are themselves sorted, and that for all pair of variables S_i and S_j with $i < j$, the bound inequality $\mathtt{max}_{S_i} \leqslant \mathtt{max}_{S_j}$ holds. This means that the numerical domain must check a number of constraints that is quadratic in the length of the definition. Example 3.14 explains why it is unsound to check only bound inequality on neighboring variables. This fact also explains why assuming the definition of a sorted sequence result in a quadratic amount of constraints at the bound parameter translation phase described earlier.

Example 3.14: On sortedness checking

This example highlights the importance of checking bounds inequality between all pairs of sequence variables, not just neighboring ones. Let us consider the following abstract state (for simplicity, we do not show the multiset part):

$$\sigma^\sharp := \begin{pmatrix} \mathtt{max}_{S_1} \leqslant \mathtt{min}_{S_2} & S = S_1.S_2.S_3 \\ \wedge\, \mathtt{max}_{S_2} \leqslant \mathtt{min}_{S_3}, & \wedge\, S_i = \mathbf{sort}(S_i) \quad i \in \{1,2,3\} \end{pmatrix}$$

This abstract state is not sufficient to deduce that S is sorted. That is to say $\gamma_s(\sigma^\sharp) \not\subseteq \{\sigma \mid \sigma \vDash_s S = \mathbf{sort}(S)\}$. The counter-example disproving the inclusion is:

$$\sigma_s = \{S \mapsto 5\,1;\ S_1 \mapsto 5; S_2 \mapsto \varepsilon; S_3 \mapsto 1;\}$$

The numeric valuation is inferred by consistency of sequence domain elements. The definitions in the sequence part of the abstract domain are satisfied by this valuation. Additionally, since $\sigma_s(S_2) = \varepsilon$, $\sigma_n(\mathtt{min}_{S_2}) = +\infty$ and $\sigma_n(\mathtt{max}_{S_2}) = -\infty$. As a consequence, the numerical constraints are satisfied as well. But we don't have $\sigma \vDash_s S = \mathbf{sort}(S)$ since $5\,1$ is not a sorted sequence.

In cases where the definition contains atoms $[\alpha]$, it is unnecessary to verify the bound inequality between sequence variables from each side of the atom. The rule outlined in Figure 3.12b utilizes

$$\frac{S = E \qquad \forall S' \in \mathbf{fv}(E), S' = [] \qquad \mathbf{fv}(E) \cap \mathcal{V} = \varnothing}{S = []} \qquad \frac{\mathtt{len}_S = 0}{S = []} \qquad \frac{\mathtt{min}_S > \mathtt{max}_S}{S = []}$$

(a) Empty case

$$\frac{\begin{array}{c}\forall i, j, S_{i,j} = \mathbf{sort}(S_{i,j}) \wedge \alpha_i \leqslant \mathtt{min}_{S_{i,j}} \wedge \mathtt{max}_{S_{i,j}} \leqslant \alpha_{i+1} \\ \forall i, j < j', \mathtt{max}_{S_{i,j}} \leqslant \mathtt{min}_{S_{i,j'}} \qquad \forall i, \alpha_i \leqslant \alpha_{i+1} \\ S = S_{0,1} \dots S_{0,l_1} [\alpha_1] S_{1,1} \dots [\alpha_i] S_{i,1} \dots S_{i,l_i} [\alpha_{i+1}] \dots [\alpha_k] S_{k+1,1} \dots S_{k+1,l_{k+1}}\end{array}}{S = \mathbf{sort}(S)}$$

(b) Sorted case

Figure 3.12: Constraint saturation unary rules

this fact to reduce the number of checks.

When unary rules infer emptiness or sortedness constraints, these constraints are inserted in the machine representation of the abstract state. These constraints undergo the parameter domains translation and the constraint propagation steps. The termination of the unary rule phases is established using the same argument as the termination of constraint propagation.

Binary rules Finally, the last phase of the $\mathbf{guard}^\sharp_s$ operator involves comparing definitions of variables sequences depending on modified parts of the sequence abstract value. To limit the exploration, we enforce that the left definition must involve a modified part of the abstract value.

General case When a sequence variable has two possible definitions, the domain tries to infer new constraints by matching the prefixes of the two definitions. The principle used here is the equality of same-length prefixes. If the matching is successful, then the comparison continues with the remaining of the definitions. This corresponds to the first line of inference rules presented in Figure 3.13a. For the sake of efficiency, we only try to match the first elements of each definition. We do not look further in the definition to check if the length of the first variable in the first definition can be matched by considering several elements in the right one. Naturally, the domain employs analogous rules (not depicted here) to match suffixes of equal lengths.

When one definition has been completely matched then this means that the remaining part of the second definition is equal to the empty sequence. These deductions correspond to the second row of inference rules.

Sorted case If a sequence variable is sorted, then the can use this information in order to split the two definitions in the middle. To do so, the domain looks for some numerical symbolic variable α that is an upper bound of elements in the left parts of the definitions, and that is strictly lower that any variables in the right-hand side of the definitions. This corresponds to the rule from Figure 3.13b. Note that we also have a similar rule obtained by inverting the strict and non-strict inequalities between α and bounds of elements in the definitions.

It is important to note that the sequence expressions denoted by E_1, E'_1... may correspond to the empty sequence symbol `[]`. To ensure that equalities inferred by binary rules involve at least one strict sub-expression of original definitions, the rule has premises stating that we cannot have matched expressions to be both empty on the same side of the cut.

Unique case Finally, if a sequence known to be free of repetitions has two possible definitions, then the domain computes numerical symbolic variable appearing in both definitions. If these variables appear in different orders, then $\mathbf{guard}^\sharp_s$ returns $\bot^\sharp_s$. This corresponds to the first rule of Figure 3.13c. Otherwise, the second comparison rule asserts that we can match sub-expressions from both sides of a common numerical symbolic variable.

When a binary rule successfully infers a new equality constraints between sub-expressions, the computed equalities can be used for other binary rules. This process stops since there is a finite number of definitions to be compared and all binary rules produces equalities between strict sub-expressions.

$$\frac{[\alpha].E_1 = [\beta].E_2}{\alpha = \beta \qquad E_1 = E_2} \qquad \frac{S.E_1 = [\beta].E_2 \qquad \mathtt{len}_S = 1}{S = [\beta] \qquad E_1 = E_2} \qquad \frac{S_1.E_1 = S_2.E_2 \qquad \mathtt{len}_{S_1} = \mathtt{len}_{S_2}}{S_1 = S_2 \qquad E_1 = E_2}$$

$$\frac{S.E = []}{S = [] \qquad E = []} \qquad \frac{\mathbf{sort}(E) = []}{E = []} \qquad \frac{[\alpha].E = []}{\top^\sharp_s}$$

(a) Generic comparison rules

$$\frac{E_1.E_2 = E_1'.E_2' \qquad \begin{array}{c}\forall S_1 \in \mathbf{fv}(E_1) \cup \mathbf{fv}(E_1'), \mathtt{max}_{S_1} \leqslant \alpha \\ \forall \beta_1 \in \mathbf{fv}(E_1) \cup \mathbf{fv}(E_1'), \beta_1 \leqslant \alpha \\ E_1 \neq [] \vee E_1' \neq []\end{array} \qquad \begin{array}{c}\forall S_2 \in \mathbf{fv}(E_2) \cup \mathbf{fv}(E_2'), \alpha < \mathtt{min}_{S_2} \\ \forall \beta_2 \in \mathbf{fv}(E_2) \cup \mathbf{fv}(E_2'), \alpha < \beta_2 \\ E_2 \neq [] \vee E_2' \neq []\end{array}}{E_1 = E_1' \qquad E_2 = E_2'}$$

(b) Sorted comparison rules

$$\frac{E_1.[\alpha].E_2.[\beta].E_3 = E_1'.[\beta].E_2'.[\alpha].E_3'}{\bot^\sharp_s} \qquad \frac{E_1.[\alpha].E_2 = E_1'.[\alpha].E_2'}{E_1 = E_1' \qquad E_2 = E_2'}$$

(c) Unique comparison rules

Figure 3.13: Comparison rules

Additionally, if one side of the equality is a sequence variable, then it corresponds to a newly inferred sequence definition. This definition is added by recursively calling $\mathbf{guard}^\sharp_s$. Note that we cannot infer an infinite number of emptiness, sortedness constraints as well as equality constraints between sequence variables because a sequence abstract state manipulates a finite amount of sequence variables. To guarantee the termination of $\mathbf{guard}^\sharp_s$, we impose that recursively added generic definitions must contain at most three symbolic variables (*i.e.* numerical and sequence variables) on the right-hand side of the equality. This ensures that, since an abstract state manipulates a finite number of variables, only a finite number of constraints may be recursively added.

Lemma 3.7: Soundness of constraint saturation

The inference rules from Figures 3.12 and 3.13 are sound.

Proof. **Constraint saturation** The first rule is proved by induction on E. Note that the constraint $\mathbf{fv}(E) \cap \mathcal{V} = \varnothing$ excludes the atom case $[\alpha]$. The proofs of the other two rules is immediate.

For the sorted case, let us consider two contiguous elements c_i and c_{i+1}, of $\sigma_s(S)$. The proof is constructed by case analysis on the origin of these elements in the definition of S. Either these elements correspond to the atoms α_i or some elements of the subsequence $S_{i,j}$.

Comparison rules Let $w_i := \mathbb{E}[\![E_i]\!]_s(\sigma_s, \sigma_n)$ for $i \in \{1, 2\}$. Similarly, let us define $w_i' := \mathbb{E}[\![E_i']\!]_s(\sigma_s, \sigma_n)$ for $i \in \{1, 2\}$. Without loss of generality, let us assume that $|w_1| \geqslant |w_1'|$. By hypothesis, $w_1.w_2$ and $w_1'.W_2'$ are equal and sorted. Levi's lemma implies that there exists a word z such that, $w_1 = w_1'.z$ and $w_2' = z.w_2$. By hypothesis all elements of z are smaller than $\sigma_n(\alpha)$ and greater or equal than $\sigma_n(\alpha)$. Such elements cannot exist. Consequently, we deduce that z is empty.

Uniqueness comparison rules are proved similarly. □

After going through all the steps, $\mathbf{guard}^\sharp_n$ computes an abstract value that also satisfy the invariants, and whose concretization contains all concrete states in the concretization of the input abstract value satisfying the input constraint.

Theorem 3.3: Correction and soundness of $\mathbf{guard}^\sharp_s$

For any abstract value $\sigma^\sharp \in \mathbb{D}^\sharp_s$ such that its machine representation satisfies **(I)**, **(II)**, **(III)**, **(IV)**, and **(V)**, and any constraint C, $\mathbf{guard}^\sharp_s(\sigma^\sharp, C)$ returns an output that also satisfies the same invariants and that is sound over-approximation of concrete states satisfying C and constraints in $\sigma^\sharp$, *i.e.*:

$$\gamma_s(\sigma^\sharp) \cap \{\sigma \in \mathbb{D}_s \mid \sigma \models C\} \subseteq \gamma_s\left(\mathbf{guard}^\sharp_s(\sigma^\sharp, C)\right)$$

Proof. The soundness of $\mathbf{guard}_s^\sharp$, is implied by Lemmas 3.2, 3.3, 3.4, 3.5, 3.6, and 3.7. The conservation of the invariants is ensured by the normalization and insertion steps of $\mathbf{guard}_s^\sharp$. □

3.4.2 Adding a numerical constraint

To assume a numerical constraint C_n in a sequence abstract state $(\sigma_n^\sharp, \sigma_{ms}^\sharp, \sigma_s^\sharp)$, one can simply add this constraint in the numerical part of the abstract value using $\mathbf{guard}_n^\sharp$. However, this new numerical constraint (alongside already known constraints in the sequence abstract state) may entail new sequence constraints. For instance, the constraint $\alpha \leqslant \beta$ implies that the sequence variable S, with a known definition being $[\alpha].[\beta]$, is sorted. To identify these cases, the sequence domain gathers all numerical variables within the constraint C_n. Using the $\mathbf{compare}_n^\sharp$ operator, it computes the set of all numerical symbolic variables associated with them. If this set contains attribute variables of a sequence variable S or a symbolic variable occurring in an atom $[\alpha]$ of a known definition of S, then the performs constraint saturation beginning from this sequence variable.

Obviously, if the input abstract state is $\bot_s^\sharp$, or if the outcome of $\mathbf{guard}_n^\sharp(\sigma_n^\sharp, C_n)$ is $\bot_n^\sharp$, then the operator returns $\bot_s^\sharp$.

The soundness of the addition of a numerical variable stems from the soundness of $\mathbf{guard}_n^\sharp$ and from the soundness of constraint saturation (Lemma 3.7)

Example 3.15: Adding a numerical constraint

To illustrate how the insertion of a numerical constraint impacts the sequence part of the abstract value, let us consider the addition of the inequality $\beta \geqslant \delta$ in the following abstract state:

$$\left(\begin{array}{l} \mathtt{max}_S \leqslant \alpha \leqslant \beta \leqslant \mathtt{min}_{S'} \\ \wedge\, \delta > \mathtt{max}_{S''}, \alpha \end{array}, \begin{array}{l} \mathfrak{R} = \varnothing, \mathfrak{E} = \varnothing, \mathfrak{U} = \varnothing, \\ \mathfrak{S} = \{S_0, S, S', S''\}, \\ \mathfrak{D} = \left\{ \begin{array}{c} S_0 \mapsto S.[\alpha].[\beta].S' \\ S''.[\delta] \end{array} \right\} \end{array}\right)$$

Thanks to the new numerical constraint, the two known definitions of S_0 are split using a binary comparison rule. Indeed, δ is a strict upper bound of S, α, and S'', as well as a lower bound of β and S'. This corresponds to the following split:

$$\begin{array}{ccc} & < \delta \leqslant & \\ S.[\alpha] & | & [\beta].S' \\ S'' & | & [\delta] \end{array}$$

The equality $S'' = S.[\alpha]$ is a definition constraint. It is added in the abstract value with the $\mathbf{guard}_s^\sharp$ operator. Additionally, matching the prefixes in the equality $[\beta].S' = [\delta]$ yields the constraints $\beta = \delta$ and $S' = []$. These constraints are also added in the abstract value. The outcome of the addition of the constraint $\beta \geqslant \delta$ is:

$$\left(\begin{array}{l} \mathtt{max}_S \leqslant \alpha < \beta = \delta \\ \wedge\, \delta > \mathtt{max}_{S''} \end{array}, \begin{array}{l} \mathfrak{R} = \varnothing, \mathfrak{E} = \{S'\}, \mathfrak{U} = \{S'\}, \\ \mathfrak{S} = \{S_0, S, S', S''\}, \\ \mathfrak{D} = \left\{ \begin{array}{l} S_0 \mapsto S''.[\beta] \\ S'' \mapsto S.[\alpha] \end{array} \right\} \end{array}\right)$$

3.4.3 Verifying a sequence constraint

Now let us present the $\mathbf{sat}_s^\sharp : \mathbb{D}_s^\sharp \times \mathcal{C}_s \to \{\mathbf{true}, \mathbf{false}\}$ operator that inputs a sequence abstract value $\sigma^\sharp$ and a sequence constraint $C_s \in \mathcal{C}_s$, and checks if $\sigma^\sharp$ implies the constraint. Similarly to $\mathbf{guard}_s^\sharp$, the bottom case is straightforward. The empty state entails all constraints. Therefore, we define for any sequence constraint $C_s \in \mathcal{C}_s$, $\mathbf{sat}_s^\sharp(\bot_s^\sharp, C_s) := \mathbf{true}$. In the following, we present the non-bottom case.

To check if a constraint C_s is valid in a sequence abstract state $(\sigma_n^\sharp, \sigma_{ms}^\sharp, (\mathfrak{R}, \mathfrak{E}, \mathfrak{S}, \mathfrak{U}, \mathfrak{D}))$, $\mathbf{sat}_s^\sharp$ starts by normalizing the constraint. This means that C_s undergoes the same process described in Section 3.4.1.1. If the outcome of the normalization is a simple constraint, then $\mathbf{sat}_s^\sharp$ boils down to checking if the constraint is known in the corresponding element of the machine representation. For a generic definition $S = E$, the constraint checking operator essentially inlines definitions of variables present in the constraint until both sides of the equality are syntactically equal. To limit the

$$\frac{E = E''.E' \qquad S \mapsto E'' \in \mathfrak{D}}{E = S.E'} \qquad \frac{E = E'}{S.E = S.E'} \qquad \frac{\alpha = \beta \qquad E = E'}{[\alpha].E = [\beta].E'}$$

$$\frac{\tau_{ms}(E) = \tau_{ms}(E') \qquad E \text{ is sorted}}{E = \mathbf{sort}(E')}$$

Figure 3.14: Inference rule for $\mathbf{sat}^\sharp_s$

exploration, the domain removes common prefixes. In the case where the prefixes are formed by two atoms $[\alpha]$ and $[\beta]$, then the equality constraint $\alpha = \beta$ must be verified in the numerical part of the abstract domain. This means that the operator only needs to inline the leftmost sequence variable to continue the exploration. Such limitation restricts the research space for the sake of efficiency. This corresponds to the first line of inference rules presented in Figure 3.14.

When the exploration reaches an equality of the form $E = \mathbf{sort}(E')$, the $\mathbf{sat}^\sharp_s$ operator concludes using a specific rule. Using the multiset parameter domain, it checks that the content of E and E' are equal. Additionally, it verifies that the sequence E is sorted. This sortedness checking is carried out using the same rule as the one inferring sortedness from a definition during constraint saturation presented in Figure 3.11c.

Example 3.16: Verifying a sequence constraint

This example presents the verification of the sequence constraint $S = S'_1.S'_0.S'_2$ in the following sequence abstract value:

$$\left(\sigma^\sharp_n, \sigma^\sharp_{ms}, \mathfrak{R} = \left\{\begin{matrix} S_1 \sim S'_1 \\ S_l \sim S'_0 \end{matrix}\right\}, \mathfrak{E} = \varnothing, \mathfrak{U} = \varnothing, \mathfrak{S} = \varnothing, \mathfrak{D} = \left\{\begin{matrix} S \mapsto S_1.S_0.S_2 \\ S_0 \mapsto S_l.[\alpha].S_r \\ S'_2 \mapsto [\beta].S_r.S_2 \end{matrix}\right\}\right)$$

The verification operator starts by rewriting the constraint to obtain its normal form. In our example, S'_0 and S'_1 are replaced by their respective class representative. Then the $\mathbf{sat}^\sharp_s$ operator computes the following derivation which concludes that the abstract value entails the constraint $S = S'_1.S'_0.S'_2$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{S_r.S_2 = S_r.S_2 \qquad \mathbf{sat}^\sharp_n(\sigma^\sharp_n, \alpha = \beta) = \mathbf{true}}{[\alpha].S_r.S_2 = [\beta].S_r.S_2} \qquad S'_2 \mapsto [\beta].S_r.S_2 \in \mathfrak{D}}{[\alpha].S_r.S_2 = S'_2}}{S_l.[\alpha].S_r.S_2 = S_l.S'_2} \qquad S_0 \mapsto S_1.S_0.S_2 \in \mathfrak{D}}{S_0.S_2 = S_l.S'_2}}{S_1.S_0.S_2 = S_1S_l.S'_2} \qquad S \mapsto S_1.S_0.S_2 \in \mathfrak{D}}{S = S_1.S_l.S'_2}$$

Example 3.17: Verifying a sequence constraint with sortedness checking

To illustrate the verification of a constraint requiring a sortedness checking let us consider the verification of the constraint $S' = S_1.[\alpha].[\delta].S_r.S_2$ in following abstract state:

$$\left(\begin{matrix} \mathtt{max}_{S_1} \leqslant \alpha < \delta \leqslant \mathtt{min}_{S_r} \\ \wedge\, \delta, \mathtt{max}_{S_r} \leqslant \mathtt{min}_{S_2} \end{matrix}, \begin{matrix} \mathtt{mset}_S = \mathtt{mset}_{S_0} \\ \uplus\, \mathtt{mset}_{S_1} \uplus\, \mathtt{mset}_{S_2} \\ \wedge\, \mathtt{mset}_{S_1} = \{\delta\} \uplus\, \mathtt{mset}_{S_r} \\ \wedge\, \mathtt{mset}_{S'} = \{\alpha\} \uplus\, \mathtt{mset}_{S'} \end{matrix}, \begin{matrix} \mathfrak{R} = \varnothing, \mathfrak{E} = \varnothing, \mathfrak{U} = \varnothing, \\ \mathfrak{S} = \{S, S_0, S_1, S_2, S_r\}, \\ \mathfrak{D} = \left\{\begin{matrix} S \mapsto S_0.S_1.S_2 \\ S_1 \mapsto [\delta].S_r \\ S' \mapsto \mathbf{sort}\,[\alpha].S \end{matrix}\right\} \end{matrix}\right)$$

After inlining the definition of S', the constraint to verify is $\mathbf{sort}([\alpha].S) = S_1.[\alpha].[\delta].S_r.S_2$. Therefore, using the $\mathbf{sat}^\sharp_{ms}$ operator, the sequence domain checks that the two sides of the equality have the same content. This corresponds to the multiset constraint:

$$\{\alpha\} \uplus \mathtt{mset}_S = \{\alpha, \delta\} \uplus \mathtt{mset}_{S_1} \uplus \mathtt{mset}_{S_2} \uplus \mathtt{mset}_{S_r}$$

This check succeeds. Then the domain proceeds with the proof of the sortedness of the

sequence expression at the right-hand side of the equal sign. It checks that all sequence variables S_1, S_r, S_2 in the expression are known to be sorted because they are all members of the $\mathfrak{S}$ set. Finally, using the numerical abstract part, $\mathbf{sat}^\sharp_s$ verifies that the bound constraints hold to conclude that the sequence constraint is entailed by the abstract value. Here, these constraints are:

$$\begin{array}{rlll} \mathtt{max}_{S_1} \leqslant \alpha & & \delta \leqslant \mathtt{min}_{S_r} & \mathtt{max}_{S_r} \leqslant \mathtt{min}_{S_2} \\ \alpha \leqslant \delta & & \delta \leqslant \mathtt{min}_{S_2} & \end{array}$$

Theorem 3.4: Soundness of $\mathbf{sat}^\sharp_s$

For any sequence constraint $C_s \in \mathcal{C}_s$ and any sequence abstract state $\sigma^\sharp \in \mathbb{D}^\sharp_s$,

$$\mathbf{sat}^\sharp_s(C_s, \sigma^\sharp) = \mathbf{true} \Longrightarrow \gamma_s(\sigma^\sharp) \subseteq \{(\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s \mid \sigma_n, \sigma_s \vDash_s C_s\}$$

Proof. The proof is done by induction over the derivation of the equality judgment. The soundness of the first rule is a consequence of the substitution lemma (Lemma 3.1). For the last rule, the soundness of $\mathbf{sat}^\sharp_{ms}$ implies that E and E' have the same content, *i.e.* E is obtained by permuting the elements of E'. Additionally, the soundness of the sortedness checking (Lemma 3.6) entails that the value of E is equal to the sorted permutation of E', *i.e.* $\mathbf{sort}E$. □

3.5 Lattice operators

3.5.1 Inclusion test

The sequence abstract domain inclusion operator $\bullet \sqsubseteq^\sharp_s \bullet : \mathbb{D}^\sharp_s \times \mathbb{D}^\sharp_s \to \{\mathbf{true}, \mathbf{false}\}$, takes two sequence abstract values $\sigma^\sharp_1$ and $\sigma^\sharp_2$ as inputs and returns a boolean value. If the operator concludes that $\sigma^\sharp_2$ is a sound over-approximation of $\sigma^\sharp_1$, then the outcome is **true**.

Similarly to the abstract operators introduced in the previous section, the cases involving one of the inputs being $\bot^\sharp_s$ are straightforwardly defined. Any sequence abstract value is a sound over-approximation of $\bot^\sharp_s$. Moreover, if the left input is not $\bot^\sharp_s$, this means that the sequence domain is not able to prove that it represents the empty set of sequence concrete states. Therefore, the inclusion test returns **false** when the right input is $\bot^\sharp_s$.

In order to check that $(\sigma^{\sharp\prime}_n, \sigma^{\sharp\prime}_{ms}, \sigma^{\sharp\prime}_s)$ is a sound over-approximation of $(\sigma^\sharp_n, \sigma^\sharp_{ms}, \sigma^\sharp_s)$, the sequence abstract domain checks if the inclusion holds between the numerical and multiset parts of the abstract values. To do so, the domain uses the $\sqsubseteq^\sharp_n$ and $\sqsubseteq^\sharp_{ms}$ operators. Finally, by repeated applications of $\mathbf{sat}^\sharp_s$, the inclusion operator verifies that all sequence constraints in the right input are entailed by the left one. Note that for simple constraints, these repeated applications boil down to a series of set inclusion checking.

Definition 3.9: Abstract inclusion operator $\sqsubseteq^\sharp_s$

The sequence abstract domain inclusion operator is defined by:

$$\begin{array}{rl} \bullet \sqsubseteq^\sharp_s \bullet \ : \ \mathbb{D}^\sharp_s \times \mathbb{D}^\sharp_s \longrightarrow \{\mathbf{true}, \mathbf{false}\} & \\ \bot^\sharp_s \sqsubseteq^\sharp_s \_ := \mathbf{true} & \\ (\sigma^\sharp_n, \sigma^\sharp_{ms}, \sigma^\sharp_s) \sqsubseteq^\sharp_s \bot^\sharp_s := \mathbf{false} & \\ (\sigma^\sharp_n, \sigma^\sharp_{ms}, \sigma^\sharp_s) \sqsubseteq^\sharp_s (\sigma^{\sharp\prime}_n, \sigma^{\sharp\prime}_{ms}, \bigwedge_j C'_j) := & \begin{array}{l} \sigma^\sharp_n \sqsubseteq^\sharp_n \sigma^{\sharp\prime}_n \\ \wedge\, \sigma^\sharp_{ms} \sqsubseteq^\sharp_{ms} \sigma^{\sharp\prime}_{ms} \\ \wedge\, \forall j, \mathbf{sat}^\sharp_s((\sigma^\sharp_n, \sigma^\sharp_{ms}, \sigma^\sharp_s), C'_j) = \mathbf{true} \end{array} \end{array}$$

Theorem 3.5: Soundness of $\sqsubseteq^\sharp_s$

The sequence abstract domain inclusion operator $\sqsubseteq^\sharp_s$ is sound. This means that for any pair of abstract states $\sigma^\sharp_1$, and $\sigma^\sharp_2$,

$$\sigma^\sharp_1 \sqsubseteq^\sharp_s \sigma^\sharp_2 = \mathbf{true} \Longrightarrow \gamma_s\left(\sigma^\sharp_1\right) \subseteq \gamma_s\left(\sigma^\sharp_2\right)$$

Proof. The soundness of $\sqsubseteq_s^\sharp$ is the direct consequence of the soundness of $\sqsubseteq_n^\sharp$ and $\sqsubseteq_{ms}^\sharp$, as well as the soundness of $\mathbf{sat}_s^\sharp$. □

3.5.2 Upper bound operators

We now present operators computing an upper bound of two sequence abstract states. The join operator $\bullet \sqcup_s^\sharp \bullet : \mathbb{D}_s^\sharp \times \mathbb{D}_s^\sharp \to \mathbb{D}_s^\sharp$ computes an abstract state corresponding to a sound over-approximation of its inputs. The widening operator $\bullet \nabla_s^\sharp \bullet : \mathbb{D}_s^\sharp \times \mathbb{D}_s^\sharp \to \mathbb{D}_s^\sharp$ satisfies the same soundness criteria. Additionally, it guarantees the termination property.

Here again, the bottom cases are immediate. For both operators, if one of the inputs is $\bot_s^\sharp$, then the other one is returned.

3.5.2.1 Join

We now detail the steps followed by the upper bound operator $\sqcup_s^\sharp$ to compute a sound over-approximation of $(\sigma_n^\sharp, \sigma_{ms}^\sharp, (\mathfrak{R}, \mathfrak{E}, \mathfrak{S}, \mathfrak{U}, \mathfrak{D}))$ and $(\sigma_n^{\sharp\prime}, \sigma_{ms}^{\sharp\prime}, (\mathfrak{R}', \mathfrak{E}', \mathfrak{S}', \mathfrak{U}', \mathfrak{D}'))$.

Parameter domain parts To compute the numerical part $\sigma_n^{\sharp\sqcup}$ and the multiset part $\sigma_{ms}^{\sharp\sqcup}$ of the output, the domain applies the join operator from the corresponding parametric domain.

However, an extra step is required for the numerical part. Recall that in the $\mathbf{guard}_s^\sharp$ operator, during the parameter translation of bounds in the case of an empty sequence S, we do not add inequality constraints stating that $\mathtt{min}_S = +\infty$ and $\mathtt{max}_S = -\infty$. Therefore, in the situation where there exists a sequence variable S that is known to be empty in $\sigma^\sharp$ but not in $\sigma^{\sharp\prime}$, $\sqcup_s^\sharp$ computes all upper bounds of $\mathtt{max}_S$, and lower bounds of $\mathtt{min}_S$ known in $\sigma^{\sharp\prime}$ using $\mathbf{compare}_n^\sharp$. The bounds are added in the numerical part of the output.

Simple constraints To determine the elements of the machine representation corresponding to simple constraints, the domain computes the intersection of these elements. For instance, the equality relation $\mathfrak{R}^\sqcup$ in the outcome is defined as $\mathfrak{R}^\sqcup := \mathfrak{R} \cap \mathfrak{R}'$.

Generic definitions

Enriching definition maps The first step to compute an over-approximation of maps expressing generic definitions is to enrich them. In essence, this step corresponds to the addition of simple definitions that are no longer expressed by $\mathfrak{R}$. Indeed, the class representatives of $\mathfrak{R}$ and $\mathfrak{R}'$ form a subset of the those from $\mathfrak{R}^\sqcup$. Therefore, we must adapt the definitions map to this new equality relation. Enriching a definition map $\mathfrak{D}$ consists in duplicating definitions for each class representative of $\mathfrak{R}^\sqcup$. Additionally, for each equality constraint between sequence variables $S = S'$ expressed in $\mathfrak{R}$ but not in $\mathfrak{R}^\sqcup$, mapping $S \mapsto S'$ and $S' \mapsto S$ are added in the definition map.

Unification of expressions To compute an over-approximation of the maps expressing generic definitions, the abstract domain proceeds by unification. Given two abstract states $\sigma^\sharp$, $\sigma^{\sharp\prime}$, and two expressions E and E', $\mathbf{unify}_s^\sharp(\sigma^\sharp, \sigma^{\sharp\prime}, E, E')$ tries to compute a set of expressions $\{E_i\}_i$, such that each E_i is equal to E in all concrete states synthesized by $\sigma^\sharp$ and equal to E' in all concrete states in the concretization of $\sigma^{\sharp\prime}$. The $\mathbf{unify}_s^\sharp$ function is presented in Figure 3.15. To simplify the presentation, we omit the abstract states, and we extend the concatenation of sequence expressions to sets of expressions. For example, $E.\mathbf{unify}_s^\sharp(E', E'')$ corresponds to the set $\{E.E_i \mid E_i \in \mathbf{unify}_s^\sharp(E', E'')\}$.

In essence, the unification tries to match the first elements of each expression. If the match is successful, the unification proceeds with the tails of the expressions. If an expression starts with a sequence variable that is known to be empty in the other abstract state, then it is implicitly matched with `[]`. A specific case arises when both expressions start with sequence variables that are known to be empty in the other sequence abstract state. In such situation, unification considers the two possible ordering of matching these variables.

If none of the cases described above apply, the unification process attempts to inline known definitions. Observe that this inlining step can only be performed a finite amount of times since there is no recursive definitions. This ensures that the unification precess eventually terminates since all other recursive calls happens on sub-expressions. Finally, if the unification process cannot apply any rule, it aborts and returns the empty set of expressions.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{unify}^\sharp_s \ : \ \mathcal{E}_s \times \mathcal{E}_s \to \mathcal{E}_s^*} \\
\mathbf{unify}^\sharp_s(E \quad , E \quad) := \{E\} & & \\
\mathbf{unify}^\sharp_s([\alpha].E, [\beta].E') := [\delta].\mathbf{unify}^\sharp_s(E, E') & \text{when} & \begin{array}{l} \delta \in \mathbf{compare}^\sharp_n(\sigma^\sharp_n, \alpha, =) \\ \wedge\, \delta \in \mathbf{compare}^\sharp_n(\sigma^{\sharp\prime}_n, \beta, =) \end{array} \\
\mathbf{unify}^\sharp_s(S.E \quad , S'.E' \quad) := S''.\mathbf{unify}^\sharp_s(E, E') & \text{when} & \begin{array}{l} S \sim S'' \in \mathfrak{R} \\ \wedge\, S' \sim S'' \in \mathfrak{R}' \end{array} \\
\mathbf{unify}^\sharp_s(S.E \quad , S'.E' \quad) := \begin{array}{l} S.S'.\mathbf{unify}^\sharp_s(E, E') \\ \cup\, S'.S.\mathbf{unify}^\sharp_s(E, E') \end{array} & \text{when} & \begin{array}{l} S \in \mathfrak{E}' \\ \wedge\, S' \in \mathfrak{E} \end{array} \quad \dagger \\
\mathbf{unify}^\sharp_s(S.E \quad , E' \quad) := S.\mathbf{unify}^\sharp_s(E, E') & \text{when} & S \in \mathfrak{E}' \\
\mathbf{unify}^\sharp_s(E \quad , S'.E' \quad) := S'.\mathbf{unify}^\sharp_s(E, E') & \text{when} & S' \in \mathfrak{E} \quad \dagger \\
\mathbf{unify}^\sharp_s(S.E \quad , E' \quad) := \bigcup_{S \mapsto E_S \in \mathfrak{D}} \mathbf{unify}^\sharp_s(E_S.E, E') & & \\
\mathbf{unify}^\sharp_s(E \quad , S'.E' \quad) := \bigcup_{S' \mapsto E_{S'} \in \mathfrak{D}} \mathbf{unify}^\sharp_s(E, E_{S'}.E') & & \\
\mathbf{unify}^\sharp_s(\_ \quad , \_ \quad) := \varnothing & \text{otherwise} &
\end{array}
$$

Figure 3.15: Sequence expression unification

If the unification process succeeds, the common definitions are inserted in the outcome of the union following the same insertion steps as described in Sections 3.4.1.1 and 3.4.1.2.

Lemma 3.8: Soundness of $\mathbf{unify}^\sharp_s$

For any pair of abstract states $\sigma^\sharp, \sigma^{\sharp\prime} \in \mathbb{D}^\sharp_s$, and for all sequence expressions $E, E', E'' \in \mathcal{E}_s$,

$$
E'' \in \mathbf{unify}^\sharp_s(\sigma^\sharp, \sigma^{\sharp\prime}, E, E') \Longrightarrow \begin{cases} \forall \sigma \in \gamma_s(\sigma^\sharp), \mathbb{E}[\![E]\!]_s(\sigma) = \mathbb{E}[\![E'']\!]_s(\sigma) \\ \forall \sigma \in \gamma_s(\sigma^{\sharp\prime}), \mathbb{E}[\![E']\!]_s(\sigma) = \mathbb{E}[\![E'']\!]_s(\sigma) \end{cases}
$$

Proof. The proof is constructed by induction over the computation steps used to obtain the result. Recall that this □

Example 3.18: Abstract union

To illustrate the union operator of the sequence abstract domain, let us consider the following sequence abstract values:

$$
\sigma^\sharp = \left(\begin{array}{l} \kappa \leqslant \mathtt{min}_{S_2} \\ \wedge\, \mathtt{min}_S \leqslant \mathtt{min}_{S_l}, \mathtt{min}_{S_2} \\ \wedge\, \mathtt{max}_{S_l}, \mathtt{max}_{S_2} \leqslant \mathtt{max}_S \\ \wedge\, \mathtt{max}_{S_l} = \mathtt{max}_{S_0} \wedge \mathtt{min}_{S_l} = \mathtt{min}_{S_0} \\ \wedge\, \mathtt{len}_{S_l} = \mathtt{len}_{S_0} \wedge \mathtt{len}_{S_1} = 0 \\ \wedge\, \mathtt{len}_S = \mathtt{len}_{S_l} + \mathtt{len}_{S_2} \end{array} , \begin{array}{l} \mathfrak{R} = \{S_l \sim S_0\}, \\ \mathfrak{E} = \{S_1\}, \\ \mathfrak{U} = \varnothing, \\ \mathfrak{S} = \{S, S_0, S_1, S_2, S_l, S_r\}, \\ \mathfrak{D} = \{S \mapsto S_l.S_2\} \end{array} \right)
$$

$$
\sigma^{\sharp\prime} = \left(\begin{array}{l} \mathtt{max}_{S_1} \leqslant \kappa \\ \wedge\, \mathtt{min}_S \leqslant \mathtt{min}_{S_r}, \mathtt{min}_{S_1} \\ \wedge\, \mathtt{max}_{S_r}, \mathtt{max}_{S_1} \leqslant \mathtt{max}_S \\ \wedge\, \mathtt{max}_{S_r} = \mathtt{max}_{S_0} \wedge \mathtt{min}_{S_r} = \mathtt{min}_{S_0} \\ \wedge\, \mathtt{len}_{S_r} = \mathtt{len}_{S_0} \wedge \mathtt{len}_{S_2} = 0 \\ \wedge\, \mathtt{len}_S = \mathtt{len}_{S_r} + \mathtt{len}_{S_1} \end{array} , \begin{array}{l} \mathfrak{R}' = \{S_r \sim S_0\}, \\ \mathfrak{E}' = \{S_2\}, \\ \mathfrak{U}' = \varnothing, \\ \mathfrak{S}' = \{S, S_0, S_1, S_2, S_l, S_r\}, \\ \mathfrak{D}' = \{S \mapsto S_1.S_r\} \end{array} \right),
$$

For the sake of simplicity, we do not consider the multiset parts of sequence abstract values, and we do not display numerical inequalities expressing that length attributes variables are positive.

Joining the two numerical parts of the abstract values yields the numerical abstract state presented below. Since S_1 is empty in $\sigma^\sharp$ but not in $\sigma^{\sharp\prime}$, the operator saturates the upper bounds of $\mathtt{max}_{S_1}$ and the lower bounds of $\mathtt{min}_{S_1}$ in the numerical part of $\sigma^{\sharp\prime}$ Similarly, given that S_2 is

empty only in $\sigma^{\sharp\prime}$, the bounds of the extreme values of S_2 are saturated in $\sigma^\sharp$. The outcome of the saturation is :

$$\sigma_n^{\sharp\sqcup} = \left(\begin{array}{l} \mathtt{max}_{S_1} \leqslant \kappa \leqslant \mathtt{min}_{S_2} \\ \wedge\, \mathtt{min}_S \leqslant \mathtt{min}_{S_l}, \mathtt{min}_{S_r}, \mathtt{min}_{S_0}, \mathtt{min}_{S_1}, \mathtt{min}_{S_2} \\ \wedge\, \mathtt{max}_{S_l}, \mathtt{max}_{S_r}, \mathtt{max}_{S_0}, \mathtt{max}_{S_1}, \mathtt{max}_{S_2} \leqslant \mathtt{max}_S \\ \wedge\, \mathtt{max}_{S_1} \leqslant \mathtt{min}_{S_0}, \mathtt{min}_{S_2} \\ \wedge\, \mathtt{max}_{S_0} \leqslant \mathtt{min}_{S_2} \\ \wedge\, \mathtt{len}_S = \mathtt{len}_{S_0} + \mathtt{len}_{S_1} + \mathtt{len}_{S_2} \end{array}\right)$$

Constraints in blue correspond to the infinite bounds saturation principle:

$$(\mathtt{min}_S = +\infty) \sqcup_n^\sharp (\alpha \leqslant \mathtt{min}_S) = (\alpha \leqslant \mathtt{min}_S)$$
$$(\mathtt{max}_S = -\infty) \sqcup_n^\sharp (\mathtt{max}_S \leqslant \alpha) = (\mathtt{max}_S \leqslant \alpha)$$

Now, the join computes the elements of the machine representation corresponding to simple constraints by intersection. All resulting elements are empty, excepted the set of sorted sequence variables that is equal to $\mathfrak{S}^\sqcup = \{S, S_0, S_1, S_2, S_l, S_r\}$.

Then, the definitions maps of the inputs are enriched to add equality constraints between variables that are stored in $\mathfrak{R}$ or $\mathfrak{R}'$ but not in $\mathfrak{R}^\sqcup$.

$$\mathfrak{D} = \left\{\begin{array}{l} S \mapsto S_l.S_2 \\ S_l \mapsto S_0 \\ S_0 \mapsto S_l \end{array}\right\} \qquad \mathfrak{D}' = \left\{\begin{array}{l} S \mapsto S_1.S_r \\ S_r \mapsto S_0 \\ S_0 \mapsto S_r \end{array}\right\}$$

The two maps have two common keys: S and S_0. Therefore, the join operator attempts to unify their definitions. The unification of the definitions of S_0 fails. However, the unification of the definitions of S successfully computes a common definition of S in both abstract states.

$$\begin{array}{lll} \mathbf{unify}_s^\sharp(S_l.S_2, S_1.S_r) = S_1.\mathbf{unify}_s^\sharp(S_l.S_2, S_r) & & \text{since } S_1 \in \mathfrak{E} \\ \quad = S_1.S_0.\mathbf{unify}_s^\sharp(S_2, \mathtt{[]}) & & \text{since } \begin{array}{r} S_0 \sim S_l \in \mathfrak{R} \\ \wedge\, S_0 \sim S_r \in \mathfrak{R}' \end{array} \\ \quad = S_1.S_0.S_2\mathbf{unify}_s^\sharp(\mathtt{[]}, \mathtt{[]}) & & \text{since } S_2 \in \mathfrak{E}' \\ \quad = S_1.S_0.S_2 & & \end{array}$$

To conclude, the outcome of $\sigma^\sharp \sqcup_s^\sharp \sigma^{\sharp\prime}$ is

$$\sigma^{\sharp\sqcup} = \left(\sigma_n^{\sharp\sqcup}, \mathfrak{R}^\sqcup = \varnothing, \mathfrak{E}^\sqcup = \varnothing, \mathfrak{S}^\sqcup = \{S, S_1, S_2, S_l, S_r\}, \mathfrak{U}^\sqcup = \varnothing, \mathfrak{D}^\sqcup = \{S \mapsto S_1.S_0.S_2\}\right)$$

Example 3.19: Enriching definition maps

To illustrate the importance of definition maps enrichment, let us consider the two machine representations:

$$\sigma_s^\sharp = \left(\mathfrak{R} = \varnothing, \mathfrak{E} = \varnothing, \mathfrak{S} = \varnothing, \mathfrak{U} = \varnothing, \mathfrak{D} = \left\{\begin{array}{l} S \mapsto S_1.S_2 \\ S_1 \mapsto S'.S'' \end{array}\right\}\right)$$

$$\sigma_s^{\sharp\prime} = (\mathfrak{R}' = \{S \sim S_2; S_1 \sim S' \sim S''\}, \mathfrak{E}' = \{S_1, S', S''\}, \mathfrak{S}' = \varnothing, \mathfrak{U}' = \varnothing, \mathfrak{D}' = \varnothing)$$

Here again, the elements of the machine representation expressing simple constraints have an empty intersection. Therefore, the map $\mathfrak{D}'$ must be enriched to capture sequence definitions that were not saved in $\mathfrak{R}^\sqcup$. We obtain the following map:

$$\mathfrak{D}' = \left\{\begin{array}{l} S \mapsto S_2 \\ S_2 \mapsto S \\ S_1 \mapsto S';\ S'' \\ S' \mapsto S_1;\ S'' \\ S'' \mapsto S_1;\ S' \end{array}\right\}$$

The maps $\mathfrak{D}$ and $\mathfrak{D}'$ have two common keys S and S_1. Therefore, the unification process is applied on all pairs of definitions from these variables.

$$\begin{aligned}
\mathbf{unify}_s^\sharp(S_1.S_2, S_2) &= S_1.\mathbf{unify}_s^\sharp(S_2, S_2) && \text{since } S_1 \in \mathfrak{E}' \\
&= S_1.S_2 \\
\mathbf{unify}_s^\sharp(S'.S'', S') &= S'.\mathbf{unify}_s^\sharp(S'', []) \\
&= S'.S''.\mathbf{unify}_s^\sharp([], []) && \text{since } S'' \in \mathfrak{E}' \\
&= S'.S'' \\
\mathbf{unify}_s^\sharp(S'.S'', S'') &= S'.\mathbf{unify}_s^\sharp(S'', S'') && \text{since } S' \in \mathfrak{E}' \\
&= S'.S''
\end{aligned}$$

The unification between the definition of S_1 in $\mathfrak{D}$ and the two definitions of S_1 in the enriched version of $\mathfrak{D}'$ compute the same result. Therefore, after adding the definition $S_1 = S'.S''$ obtained in the first unification, the second one definition is normalized into $S_1 = S_1$.

$$\sigma_s^{\sharp\sqcup} = \left(\mathfrak{R}^\sqcup = \varnothing, \mathfrak{E}^\sqcup = \varnothing, \mathfrak{S}^\sqcup = \varnothing, \mathfrak{U}^\sqcup = \varnothing, \mathfrak{D}^\sqcup = \left\{ \begin{array}{c} S \mapsto S_1.S_2 \\ S_1 \mapsto S'.S'' \end{array} \right\}\right)$$

In the case where the operator had not enriched the map $\mathfrak{D}'$, there would have been no common keys between the two maps. Consequently, the result of the union would have been the empty map.

Theorem 3.6: Soundness of the sequence union operator

The sequence abstract domain union operator $\bullet \sqcup_s^\sharp \bullet : \mathbb{D}_s^\sharp \times \mathbb{D}_s^\sharp \to \mathbb{D}_s^\sharp$ is sound. That is to say, for any sequence abstract values $\sigma^\sharp, \sigma^{\sharp\prime} \in \mathbb{D}_s^\sharp$,

$$\gamma_s\left(\sigma^\sharp\right) \cup \gamma_s\left(\sigma^{\sharp\prime}\right) \subseteq \gamma_s\left(\sigma^\sharp \sqcup_s^\sharp \sigma^{\sharp\prime}\right)$$

Proof. The soundness of the $\sqcup_s^\sharp$ operator is the consequence of the soundness of $\mathbf{unify}_s^\sharp$ as well as the soundness of $\mathbf{guard}_s^\sharp$ for the bound saturation as well as the addition of new constraints. □

3.5.2.2 Widening

The widening of $\sigma^\sharp$ with $\sigma^{\sharp\prime}$ is computed similarly than their union, with one exception: the numerical and multiset parts are computed using the widening operators from the corresponding parameter domain.

Theorem 3.7: Soundness and termination of the sequence widening operator

The sequence abstract domain widening operator $\bullet \nabla_s^\sharp \bullet : \mathbb{D}_s^\sharp \times \mathbb{D}_s^\sharp \to \mathbb{D}_s^\sharp$ is sound. That is to say, for any sequence abstract values $\sigma^\sharp, \sigma^{\sharp\prime} \in \mathbb{D}_s^\sharp$,

$$\gamma_s\left(\sigma^\sharp\right) \cup \gamma_s\left(\sigma^{\sharp\prime}\right) \subseteq \gamma_s\left(\sigma^\sharp \nabla_s^\sharp \sigma^{\sharp\prime}\right)$$

Additionally, for any sequence of abstract values $(\sigma_i^\sharp)_{i\geqslant 0}$, the sequence $(\sigma_i^{\sharp\nabla})_{i\geqslant 0}$ defined as $\sigma_0^{\sharp\nabla} := \sigma_0^\sharp$ and $\sigma_{k+1}^{\sharp\nabla} := \sigma_k^{\sharp\nabla} \nabla_s^\sharp \sigma_{k+1}^\sharp$ is ultimately stationary.

Proof. The only difficult point here is the proof of convergence of the definition map.

Note that in all unification cases presented in Figure 3.15, only the ones marked with a † introduce a variable that was not present in the left definition. Such rules can only be applied a finite number of times. Indeed, they apply if and only if $S' \in \mathfrak{E}$ and $S' \notin \mathfrak{E}'$. This means that the rules inserting a new sequence variable in a definition are applied only when the number of sequence variables known to be empty is strictly decreasing.

When the set of sequence variable known to be empty is stabilized, the sum of the lengths of all possible definitions is finite and cannot increase. The finiteness of this quantity stems from the fact that there is no cyclic definitions. Here, the expression *all definitions* take into account

definitions that can be obtained by:

- replacing symbolic numerical variables or sequence variables by other variables that are known to be equal in the numerical part of the abstract value or $\mathfrak{R}$, respectively;
- inlining definitions.

□

Chapter 4

A product of shape and sequence abstractions

This chapter describes the integration of the sequence reasoning presented in Chapter 3 in the separation logic-based shape analysis outlined in Chapter 2. This combination aims to compute constraints over the content of dynamic data structures. The integration necessitates extending inductive predicates definition with sequence parameters to describe the content summarized in these predicates. It also requires to complete the abstract operators that manipulate these predicates during the analysis, including the folding and unfolding of inductive predicates, as well as the lattice operators.

4.1 Adding sequence parameters to inductive predicates

The goal of this section is to present how to extend the inductive predicates presented in Section 2.3.2 with sequence parameters. This section describes the definition of sequence parametrized inductive predicates, their concretization, as well as the possible types of parameters. Each type of parameters allows the analysis to infer specific constraints such as emptiness or uniqueness constraints. It also presents in depth the process that derives the segment counterpart of a full inductive predicate.

4.1.1 Generic form of inductive predicates with sequence parameters

4.1.1.1 Definition

Figure 4.1 presents the syntax of inductive predicates extended with sequence parameters. An instance of an inductive predicate is parametrized by numerical symbolic variables $\vec{\kappa}$ as well as sequence variables $\vec{S}$. Similarly to the definition introduced earlier in Section 2.3.2, an inductive predicate is defined as a finite and non-empty disjunction of *rules*. Each rule is a formula existentially quantified by both numerical and sequence symbolic variables. For the sake of simplicity, we assume that these existentially quantified variables are disjoint from the parameter of the inductive predicates. The formula is formed by three parts.

Shape part The first part is the shape part that describes the memory layout specific to this rule. The syntax of the shape part is presented in Figure 4.1a.

Similarly to the presentation from Section 2.3.2, we restrict the kind of numerical symbolic variables that may be used in separation logic predicates. Only the main parameter α may be used as the source of points-to predicates. Recursive and nested instances of inductive predicates may only have an existentially quantified variable β as their main parameter. Note that the destinations of points-to predicates, as well as parameters of inductive predicates may correspond to the main parameter α, numerical parameters κ, or an existentially quantified variable β. This is noted $\delta \in \{\alpha, \vec{\beta}, \vec{\kappa}\}$ in the syntax.

Additionally, we impose that each sequence parameter for all nested and recursive instances of inductive predicates are existentially quantified sequence variables, and that these parameters are different. In the following, for a rule indexed by k, we will write the shape part as $m_k^\sharp * (\circledast_j \beta_{k,j}.\mathbf{pred}(\vec{S}_{k,j}))$ to dissociate the local part (*i.e.* the cell part and the nested part) from the recursive part. This means that $m_k^\sharp$ does not contain any recursive instances of inductive predicate. Using this notation, in the k^{th} rule, in the j^{th} recursive instance of **pred**, its i^{th} sequence parameter is denoted as $S_{k,j,i}$.

Pure part Similarly to the definition introduced in the Section 2.3.2, the pure part is a finite conjunction of numerical constraints. These constraints are comparison of expressions where variables are numerical symbolic variables. The syntaxes of these expressions and constraints are those presented in Figures 2.9b and 2.9c.

Sequence part The last part, called the *sequence part* is specific to inductive predicates extended with sequence parameters. This part is a finite conjunction of sequence definition. Naturally, the variables that can occur in these definition constraints must be either parameters of the predicate or existentially quantified variables. Observe that syntactically restricting the parameters of recursive instances to be distinct sequence variables does not limit the expressiveness of inductive predicates. Indeed, it remains feasible to ensure that two sequence variables are equal in the sequence part of the rule.

$$
\begin{array}{lll}
m_\iota^\sharp ::= \alpha.\texttt{f} \mapsto \delta & \delta \in \{\alpha, \vec{\beta}, \vec{\kappa}\} & \text{(cell part)} \\
\quad \mid \beta.\mathbf{pred}\left(\vec{\delta}, \vec{S'}\right) & \delta \in \{\alpha, \vec{\beta}, \vec{\kappa}\} & \text{(recursive/nested calls)} \\
\quad \mid \mathbf{emp} & & \\
\quad \mid m_\iota^\sharp * m_\iota^\sharp & &
\end{array}
$$

(a) Syntax of memory part of inductive predicates

$$\alpha.\mathbf{pred}(\vec{\kappa}, \vec{S}) = \bigvee \exists \vec{\beta}, \vec{S'}, \left(m_\iota^\sharp \wedge \varphi_\mathcal{V} \wedge \left(\bigwedge S = E\right)\right)$$

(b) Syntax of inductive predicates

Figure 4.1: Syntax of inductive predicates

Example 4.1: Singly linked list inductive predicate with sequence parameter

This example presents the singly linked predicate from Example 2.3.2 extended with a sequence parameter S representing the content stored in the list. That is to say, the sequence of `data` fields of the nodes in the list.

When the list is empty, then so is the sequence of its elements. Therefore, in the empty rule the sequence constraint on S boils down to $S = \texttt{[]}$.

When the list contains at least one node, then its sequence of elements is computed by considering the content of the `data` fields. Such content is denoted by the symbolic numerical variable β_d. Then, this variable is appended by the sequence of elements in the remaining of the list. This corresponds to the sequence parameter of the recursive instance, *i.e.* the existentially quantified variable S'. This yields the sequence constraint $S = \texttt{[}\beta_d\texttt{]}.S'$. In summary, the sequence parametrized inductive predicate representing a singly linked list is:

$$
\begin{array}{rl}
\alpha.\mathbf{list}(S) := & \mathbf{emp} \wedge \alpha = 0 \wedge S = \texttt{[]} \\
\mid & \exists \beta_d, \beta_n, S',\ \alpha.\texttt{next} \mapsto \beta_n * \alpha.\texttt{data} \mapsto \beta_d * \beta_n.\mathbf{list}(S') \\
& \qquad \wedge \alpha \neq 0 \wedge S = \texttt{[}\beta_d\texttt{]}.S'
\end{array}
$$

Example 4.2: List of singly linked lists inductive

This example presents an inductive predicate **listOfLists** representing a list of nested lists. All nested lists have the same content represented by the sequence parameter S. When the list is empty, there is no constraint on the parameter S. If the list contains at least one node, then the nested list pointed by this node has its sequence parameter S'' that is asserted to be equal to the sequence variable S. Additionally, the sequence parameter of the recursive instance S' is also equal to S.

$$
\begin{array}{rl}
\alpha.\mathbf{listOfLists}(S) := & \mathbf{emp} \wedge \alpha = 0 \\
\mid & \exists \beta_d, \beta_n, S', S'',\ \alpha.\texttt{next} \mapsto \beta_n * \alpha.\texttt{data} \mapsto \beta_d \\
& \qquad * \beta_n.\mathbf{listOfLists}(S') \\
& \qquad * \beta_d.\mathbf{list}(S'') \\
& \qquad \wedge \alpha \neq 0 \wedge S = S' \wedge S = S''
\end{array}
$$

Example 4.3: List with addresses and content sequences

The **list** predicate introduced in Example 4.1 only specifies the sequence of values stored by the list in the `data` field. However, it is also useful to reason over the sequence formed by the addresses of the nodes in the list. To perform such reasoning, we introduce a novel list predicate called **addrList**. The rules of this predicate are similar to the **list** ones: the memory and numerical parts are the same. But **addrList** has two sequence parameters.

The first one, called S_a corresponds to the sequence formed by the addresses of the nodes in the list. In the first rule, the list is empty. Therefore, the sequence parameter S_a is also empty. This corresponds to the constraint $S_a = \texttt{[]}$. Additionally, in the non-empty case presented in the second rule, the sequence of addresses is computed by prepending the address α of the head of the list to the sequence of addresses of nodes inside the tail of the list. This is the sequence constraint $S_a = \texttt{[}\alpha\texttt{]}.S'_a$.

$$\alpha.\mathbf{task}(\tau) := \exists \beta_w, \alpha.\texttt{wst} \mapsto \tau * \alpha.\texttt{weight} \mapsto \beta_w \wedge \alpha \neq \texttt{0x0}$$

$$\begin{array}{rll}
\alpha.\mathbf{tree}(\kappa_p, S) := & \mathbf{emp} \wedge \alpha = 0 \wedge S = [] & \\
\vee\ \exists \beta_t, \beta_l, \beta_r, \tau, S_l, S_r & \alpha.\texttt{task} \mapsto \beta_t * \alpha.\texttt{parent} \mapsto \kappa_p & \left.\begin{array}{l}\\\end{array}\right\} \text{cell} \\
& * \alpha.\texttt{left} \mapsto \beta_l * \alpha.\texttt{right} \mapsto \beta_r & \left.\begin{array}{l}\\\end{array}\right\} \text{part} \\
& * \beta_t.\mathbf{task}(\tau) & \left.\begin{array}{l}\\\end{array}\right\} \text{nested part} \\
& * \beta_l.\mathbf{tree}(\alpha, S_l) & \left.\begin{array}{l}\\\end{array}\right\} \text{recursive} \\
& * \beta_r.\mathbf{tree}(\alpha, S_r) & \left.\begin{array}{l}\\\end{array}\right\} \text{part} \\
& \wedge \alpha \neq 0 & \left.\begin{array}{l}\\\end{array}\right\} \text{numerical part} \\
& \wedge S = S_l.[\tau].S_r & \left.\begin{array}{l}\\\end{array}\right\} \text{sequence part}
\end{array}$$

Figure 4.2: Inductive predicates for WFS

The second parameter S_v denotes the sequence of values stored in the list. Its constraints are similar to the sequence constraints in Example 4.1. To sum up, the whole definition of **addrList** is:

$$\begin{array}{rl}
\alpha.\mathbf{addrList}(S_a, S_v) := & \mathbf{emp} \wedge \alpha = 0 \wedge S_a = [] \wedge S_v = [] \\
\mid & \exists \beta_n, \beta_d, S'_a, S'_v,\ \alpha.\texttt{next} \mapsto \beta_n * \alpha.\texttt{data} \mapsto \beta_d * \beta_n.\mathbf{addrList}(S'_a, S'_v) \\
& \wedge \alpha \neq 0 \wedge S_a = [\alpha].S'_a \wedge S_v = [\beta_d].S'_v
\end{array}$$

Example 4.4: Sorted list

In order to summarize a sorted list one can simply extend the singly linked predicate from Example 4.1 by inserting a constraint stating that S is sorted. In the empty rule, this constraint is superfluous since the emptiness of S implies its sortedness. Therefore, the constraint $S = \mathbf{sort}(S)$ is inserted only in the second rule. This yields the following inductive predicate:

$$\begin{array}{rl}
\alpha.\mathbf{sortedList}(S) := & \mathbf{emp} \wedge \alpha = 0 \wedge S = [] \\
\mid & \exists \beta_n, \beta_d, S',\ \alpha.\texttt{next} \mapsto \beta_n * \alpha.\texttt{data} \mapsto \beta_d * \beta_n.\mathbf{sortedList}(S') \\
& \wedge \alpha \neq 0 \wedge S = [\beta_d].S' \wedge S = \mathbf{sort}(S)
\end{array}$$

Example 4.5: Inductive predicate for WFS

Figure 4.2 presents the inductive predicates representing the data structures manipulated by the weighted fair scheduler presented in Section 1.4.1. The **tree** inductive predicate is parametrized by a sequence variable S. This variable denotes the sequence of weighted service time of the tasks stored in the tree, read in infix order, *i.e.* from left to right. The first rule corresponds to the empty case, thus the sequence is empty as well. In the second rule, the sequence is obtained by concatenating together the sequence of the left subtree, the weighted service time of the node, and the sequence of the right subtree. Therefore, the weighted service time of the task τ stored in the node must appear in the sequence constraint. However, this quantity is summarized in the **task** inductive predicate. To address this issue, the weighted service time τ is existentially quantified in the non-empty rule of **tree**, and it is set as an integer parameter of the **task** instance.

4.1.1.2 Satisfiability

In order to define the satisfiability relation $\vDash_m$ stating that a concrete heap m models an abstract memory $m^\sharp$, we need a sequence valuation $\sigma_s : \mathcal{V}_s \to \mathbb{V}^*$ to monitor the values taken by sequence parameters of inductive predicates. Therefore, the satisfiability judgement is extended with a sequence valuation. We enforce that these valuations satisfy the consistency requirement regarding sequence concrete states expressed in Definition 3.1.

Definition 4.1: Memory satisfiability relation

The memory satisfiability relation $\vDash_m$ is defined inductively on the syntax of abstract memory heaps as:

$$
\begin{aligned}
m, \sigma_n, \sigma_s &\vDash_m \mathbf{emp} && \text{iff } m = \varnothing \\
m, \sigma_n, \sigma_s &\vDash_m \alpha.\mathtt{f} \mapsto \beta && \text{iff } m = \{\sigma_n(\alpha) + \varphi_{\mathbb{F}} \mapsto \sigma_n(\beta)\} \\
m, \sigma_n, \sigma_s &\vDash_m m_1^\sharp * m_2^\sharp && \text{iff } \exists m_1, m_2, \begin{cases} \mathbf{supp}(m_1) \cap \mathbf{supp}(m_2) = \varnothing \\ m = m_1 \uplus m_2 \\ m_1, \sigma_n, \sigma_s \vDash_m m_1^\sharp \\ m_2, \sigma_n, \sigma_s \vDash_m m_2^\sharp \end{cases} \\
m, \sigma_n, \sigma_s &\vDash_m \alpha.\mathbf{pred}(\vec{\kappa}, \vec{S}) && \text{iff } \exists \vec{c}, \vec{w}, k \begin{cases} m, \sigma_n[\vec{\beta} \mapsto \vec{c}], \sigma_s[S' \mapsto \vec{w}] \vDash_m m_{\iota,k}^\sharp \\ \sigma_n[\vec{\beta} \mapsto \vec{c}] \vDash_n \varphi_{\mathcal{V},k} \\ \sigma_n[\vec{\beta} \mapsto \vec{c}], \sigma_s[\vec{S'} \mapsto \vec{w}] \vDash_s \varphi_{s,k} \end{cases} \\
& && \text{where } \alpha.\mathbf{pred}(\vec{\kappa}, \vec{S}) := \bigvee_k \exists \vec{\beta}, \vec{S}, m_{\iota,k}^\sharp \wedge \varphi_{\mathcal{V},k} \wedge \varphi_{s,k}
\end{aligned}
$$

Observe that since existentially quantified variables are assigned fresh values in the judgement defining the satisfiability of inductive predicates, only the values of symbolic variables that syntactically occur in the abstract heap are relevant to establish the satisfiablity judgement. This is formalized by the following lemma:

Lemma 4.1: Support of abstract memory states

For any concrete heap m, any abstract heap $m^\sharp$ and any pair of symbolic valuations (σ_n, σ_s) and (σ'_n, σ'_s), if:

- $\forall \alpha \in \mathbf{fv}(m^\sharp) \cap \mathcal{V}, \sigma_n(\alpha) = \sigma'_n(\alpha)$,
- $\forall S \in \mathbf{fv}(m^\sharp) \cap \mathcal{V}_s, \sigma_s(S) = \sigma'_s(S)$,

then, $m, \sigma_n, \sigma_s \vDash_m m^\sharp \implies m, \sigma'_n, \sigma'_s \vDash_m m^\sharp$.

Proof. The proof is carried out by induction over the derivation of the judgement $m, \sigma_n, \sigma_s \vDash_m m^\sharp$.

The base cases, *i.e.* the empty and points-to predicates, are straightforward. Indeed, they hold if and only if m is equal to a concrete heap that only depends on the value of variables that syntactically occur in $m^\sharp$.

The separating conjunction case is immediately proved by applying the induction hypothesis.

Let us consider the case where $m^\sharp = \alpha.\mathbf{pred}(\vec{\kappa}, \vec{S})$. This means for some rule index k, there exist some values $\vec{c}$ and $\vec{w}$ for existentially quantified variables $\vec{\delta}$ and $\vec{S}$ such that the updated valuations $\sigma_n[\vec{\beta} \mapsto \vec{c}]$ and $\sigma_s[\vec{S} \mapsto \vec{w}]$ satisfy the shape, numerical and sequence parts of the rule. Recall, that by definition all variables that appear in a rule are either parameters of the inductive predicate or existentially quantified variables. Therefore, if we consider $\sigma'_n[\vec{\beta} \mapsto \vec{c}]$ and $\sigma_s[\vec{S} \mapsto \vec{w}]$, any free variables in the memory part of the rule has the same image by the two updated valuations. By applying the induction hypothesis, we derive that the updated versions of σ'_n and σ'_s also satisfy the memory part of the rule. Additionally, we establish that $\sigma'_n[\vec{\beta} \mapsto \vec{c}]$ and $\sigma_s[\vec{S} \mapsto \vec{w}]$ satisfy the numerical and sequence parts of the rule. This is done by induction over these parts. Such induction is straightforward since it does not require dealing with existentially quantified variables. Therefore, by definition of $\vDash_m$, we conclude that $m, \sigma'_n, \sigma'_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}, \vec{S})$. □

Given the satisfiability relation, we can define the concretization function. It maps an abstract memory state $m^\sharp \in \mathbb{M}^\sharp$ to a set of pairs. The first element is the concrete memory state. The second one is a tuple of valuations. We enforce that these valuations are picked from the sequence concrete domain $\mathbb{D}_s$ to ensure consistency of values assigned to attribute variables.

Definition 4.2: Concretization of abstract memory states

For an abstract memory state $m^\sharp$, the corresponding set of memory states with numerical and sequence valuation functions is defined by:

$$
\begin{aligned}
\gamma_m : \mathbb{M}^\sharp &\longrightarrow \wp(\mathbb{S} \times \mathbb{D}_s) \\
m^\sharp &\longmapsto \left\{ ((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \;\middle|\; \begin{array}{l} m, \sigma_n, \sigma_s \vDash_m m^\sharp \\ \forall \mathtt{x} \in \mathbb{X}, \rho(\mathtt{x}) = \sigma_n(\underline{\mathtt{x}}) \end{array} \right\}
\end{aligned}
$$

4.1.1.3 Segment predicates

In order to define the segment counter-part of the list predicate, we could simply follow the approach followed by integer and pointer parameters. In this view, the segment predicate $\alpha.\mathbf{list}(S) \nRightarrow \alpha.\mathbf{list}(S')$ is parametrized by two sequence variables S and S'. The sequence denoted by S' corresponds to the missing sequence that must be appended at the tail of the segment in order to form a list containing the elements expressed by the sequence S.

However, if we append a list containing a different sequence $\alpha'.\mathbf{list}(S'')$, then we cannot concatenate the segment with this new predicate instance. Indeed, the right part of the segment and the new predicate do not match. Restricting the right part of the segment to be syntactically equal to the new predicate is overly restrictive for sequence parameters. In the case of list segment predicate, the sequence S can be decomposed into two parts: the one actually summarized by the segment predicate S_0, and the sequence needed to obtain S, *i.e.* S'. This means that the segment predicate is equivalent to $\forall S', \alpha.\mathbf{list}(S_0.S') \nRightarrow \alpha'.\mathbf{list}(S')$. This entails that only S_0 is relevant here in order to parametrize the segment since it is its actual content. Naturally, this only works for sequence parameters describing content of data structure. Such parameters form the first family of parameters called *additive parameters*.

4.1.2 Additive parameter

4.1.2.1 Definition

As expressed above, an additive sequence parameter essentially describes the content of the structure summarized by an inductive predicate. This means that for each rule of the predicate, the expression of an additive parameter can be derived from a single definition that connects it to the value of the parameters used in recursive calls.

Definition 4.3: Additive parameter

A sequence parameter S_i of an inductive predicate **pred** is *additive* if and only if, for each rule containing a recursive instance of **pred**,

- the numerical part does not contain any occurrence of attribute variables of sequence parameters,
- the parameter S_i as well as the parameters of the recursive calls $S_{k,j,i}$ occur only in a single definition constraint $S_i = E_i$,
- each parameter $S_{k,j,i}$ occurs exactly once in E_i, and this occurrence is not in an instance of a **sort** symbolic function.

Example 4.6: Additive parameters of inductive predicates

This example highlights the additive sequence parameters of the previously introduced inductive predicates.

The sequence parameter of **list** defined in Example 4.1 is additive. Indeed, in the empty rule, there is a single definition of S stating that S is empty. This definition is valid since there is no recursive call in the empty rule. In the non-empty case, the constraint $S = [\beta_d].S'$ serves as a valid definition of an additive parameter, since the parameter of the recursive call S' appears once. Similarly, we observe that in the **addrList** predicate defined in Example 4.3, both sequence parameters S_a and S_v are additive.

Example 4.2 defines an inductive predicate **listOfLists**. The sequence parameter of this predicate is not additive. Indeed, in the second rules this parameter has two definitions. However, if we replace the two constraints in the second rule by $S = S'$ or $S = S''.S'$, then the parameter becomes additive. In the latter case, we obtain an inductive predicate describing a list of nested lists where the sequence parameter describes the content of all nested lists according to their order of appearance in the main list:

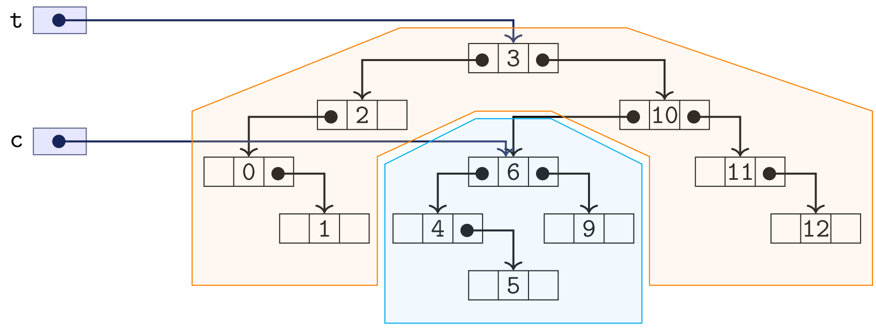


Figure 4.3: Synthesizing a binary tree with a full and a segment predicates

$$\begin{aligned}\alpha.\mathbf{nestedLists}(S) := \ & \mathbf{emp} \wedge \alpha = 0 \wedge S = [\,] \\ \mid \ & \exists \beta_d, \beta_n, S', S'', \ \alpha.\mathtt{next} \mapsto \beta_n * \alpha.\mathtt{data} \mapsto \beta_d \\ & * \beta_n.\mathbf{nestedLists}(S') \\ & * \beta_d.\mathbf{list}(S'') \\ & \wedge \alpha \neq 0 \wedge S = S''.S'\end{aligned}$$

The sequence parameter of the **sortedList** predicate presented in Example 4.4 is not additive because it does not violate two conditions of additive parameters. Firstly, the second rule contains two definitions of S. Second, in the constraint expressing the sortedness S appears as an argument of the **sort** symbolic function.

Finally, for the inductive predicates of WFS presented in Example 4.5, the sequence parameter of **tree** is additive as well. For the first rule, the argument is similar to the one presented for **list**. For the second rule, the constraint presenting the definition of S from S_l and S_r satisfies the criteria stated in Definition 4.3.

In the remaining of the section, we assume that all sequence parameters are additive.

4.1.2.2 Segment predicates

Thanks to the criteria of additive sequence parameters, we can now present the algorithm deriving segment predicates from full ones. But before doing so, let us motivate the usage of two sequence variables for each additive sequence parameter.

Using two sequence variables Consider the memory state presented in Figure 4.3. It represents a memory state during the exploration of a binary search tree by a cursor denoted by the pointer variable `c`. The part of the tree pointed by `c`, presented in blue, corresponds to a full binary tree. Therefore, it can be summarized as an instance of a full inductive predicate `c`.**tree**(S) where S corresponds to the numerical sequence 4 5 6 9. The remaining of the tree, highlighted in orange should be summarized by a segment predicate whose sequence parameter must denote the content of the partial tree. However, using a single sequence parameter to denote the content of the partial tree, *i.e.* 0 1 2 3 10 11 12, is too imprecise. Indeed, the segment predicate must also recall the position of the missing sequence. That is to say the point corresponding to the end of the segment. In order to remember the position of the missing sequence, let us use two sequence parameters S_l and S_r, one for each side of the insertion point. In the example of the memory state presented in Figure 4.3, S_l corresponds to the sequence 0 1 2 3, whereas S_r is equal to 10 11 12. Though, S_l and S_r are two sequence parameters of the segment predicate, they are denoted using the following notation $S_l \boxdot S_r$, to signify that each one denotes one side of a common sequence parameter from the full inductive predicate. The placeholder notation $\boxdot$ stands for the sequence in the missing part of the segment predicate. Finally, since $S_l \boxdot S_r$ represents the content of the segment predicate, it is depicted in the middle of the segment symbol in the following manner:

$$\mathtt{t}.\mathbf{tree} \divideontimes\!\!\{ S_l \boxdot S_r \}\!\!\models \mathtt{c}.\mathbf{tree}$$

This example show that is some cases, we need two sequence parameters to represent the content of a segment predicate. But are two parameters always enough? To answer this, let us recall that by definition of an additive parameter, the parameter expressing the content of a sub part of a data

structure occurs exactly once in the definition of the content of the whole data structure. This means that if the part is removed from the data structure, then this creates a single hole in the sequence describing the content of the partial data structure. Therefore, for additive parameters two parameters are enough for segment predicates.

Deriving segment predicates with additive parameters We now present how to define the algorithm deriving the segment counter-part of a full inductive predicate. To simplify the presentation, let us consider an inductive predicate $\alpha.\mathbf{pred}(\vec{\kappa}, S)$ with a single additive sequence parameter S. The rules of the predicate are indexed by a variable k. For each rule, we separate the local part into a single abstract memory $m_k^\sharp$ from the recursive instances. The recursive instances are explicitly written using the iterated version of the separating conjunction: $\circledast_j \beta_{k,j}.\mathbf{pred}(\vec{\delta}_{k,j}, S_{k,j})$. Each recursive instance is indexed by another variable j. The numerical part of the rule is represented by a symbolic formula $\varphi_{\mathcal{V},k}$. Finally, the sequence part is split as well. Since S is an additive parameter, we know there exists a definition $S = E_k$, such that each parameter of recursive instance occurs exactly once in E_k. All other sequence constraints are regrouped in $\varphi_{s,k}$. By definition of an additive parameter, we know that neither S nor any parameter of recursive instances $S_{k,i}$ occurs in $\varphi_{s,k}$. To sum up, the considered inductive predicate can be written as follows:

$$\alpha.\mathbf{pred}(\vec{\kappa}, S) := \bigvee_k \exists \vec{\beta}, \vec{S}, m_k^\sharp * \left(\circledast_j \beta_{k,j}.\mathbf{pred}(\vec{\delta}_{k,j}, S_{k,j})\right) \wedge \varphi_{\mathcal{V},k} \wedge \varphi_{s,k} \wedge S = E_k$$

Empty rule In order to define the segment predicate $\alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*}\{S_l \boxdot S_r\}\!\models \alpha'.\mathbf{pred}(\vec{\kappa}')$, let us first consider the empty case. This case arises when the extremities of the segment α and α' are equal. Therefore, there is no memory cell summarized by the segment: the memory part of the rule boils down to **emp**. The numerical part of the rule states that the two extremities of the segment as well as the numerical parameter are equal. Finally, the sequence part of the rule expresses that the two sequence parameters $S_l \boxdot S_r$ are empty. We obtain the following empty rule:

$$\mathbf{emp} \wedge \alpha = \alpha' \wedge \vec{\kappa} = \vec{\kappa}' \wedge S_l = [] \wedge S_r = []$$

Non-empty rules Now let us present the derivation of the non-empty rules. Each of these rules corresponds to a possible location of the end of the segment in all recursive instance of the inductive predicate. This means that for all rules of the full predicate indexed by k, and for any recursive instance indexed by i, $\beta_{k,i}.\mathbf{pred}(\vec{\delta}_{k,i}, S_{k,i})$, we add a corresponding rule in the segment predicate. This rule is existentially quantified by the same variables as the rule from the full predicate, except the recursive sequence parameter $S_{k,i}$ that is replaced by the parameters of the segment predicate, $S_{i,l}$ and Si, r.

The memory part of this rule is similar to the rule from the full predicate, with one exception: the recursive instance $\beta_{k,i}.\mathbf{pred}(\vec{\delta}_{k,i}, S_{k,i})$ is replaced by the recursive instance of the segment predicate $\beta_i.\mathbf{pred}(\vec{\delta}_{k,i}) \mathrel{-\!\!*}\{S_{i,l} \boxdot S_{i,r}\}\!\models \alpha'.\mathbf{pred}(\vec{\kappa}')$.

The numerical part of the rule $\varphi_{\mathcal{V},k}$ as well as the sequence part, $\varphi_{s,k}$ not containing the sequence definition of the additive parameter are left unchanged.

Finally, the definition of the parameter $S = E_k$ is used to derive the definitions of S_l and S_r. Recall that by definition of an additive sequence parameter, the parameter $S_{k,i}$ of the recursive instance occurs exactly once in E_k, and that this occurrence is not in an argument of the **sort** symbolic function. This means that we can rewrite E_k as $E'_k.S_{k,i}.E''_k$, for some sequence expressions E'_k and E''_k that do not contain any occurrence of $S_{k,i}$. If we replace S by $S_l \boxdot S_r$ and $S_{i,k}$ by $S_{i,l} \boxdot S_{i,r}$, we obtain the following equality: $S_l \boxdot S_r = E'_k.S_{i,l} \boxdot S_{i,r}.E''_k$. By matching the expressions on each side of the placeholder symbol $\boxdot$, we derive the definitions of the sequence parameters: $S_l = E'_k.S_{i,l}$ and $S_r = S_{i,r}.E''_k$. In the following, we use the notation $S_l \boxdot S_r = E'_k.S_{i,l} \boxdot S_{i,r}.E''_k$ to denote the conjunction of the two definitions obtained after matching each side of the symbol $\boxdot$.

To conclude, the segment counterpart of the full predicate $\alpha.\mathbf{pred}(\vec{\kappa}, S)$ is:

$$\begin{aligned}\alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*}\{S_l \boxdot S_r\}\!\models \alpha'.\mathbf{pred}(\vec{\kappa}') := \mathbf{emp} \wedge \alpha = \alpha' \wedge S_l = S_r = []\\ \bigvee_k \bigvee_i \exists \vec{\beta}, \vec{S_j}, S_{i,l}, S_{i,r}, m_k^\sharp * \left(\circledast_{j \neq i} \beta_j.\mathbf{pred}(\vec{\delta}_{k,j}, S_j)\right)\\ * \beta_i.\mathbf{pred}(\vec{\delta}_{k,i}) \mathrel{-\!\!*}\{S_{i,l} \boxdot S_{i,r}\}\!\models \alpha'.\mathbf{pred}(\vec{\kappa}')\\ \wedge \varphi_{\mathcal{V},k} \wedge \varphi_{s,k}\\ \wedge S_l \boxdot S_r = E_k[S_{i,l} \boxdot S_{i,r}/S_{k,i}]\end{aligned}$$

Example 4.7: Singly linked list segment

This example illustrates the derivation of the segment counterpart of the singly linked list inductive predicate $\alpha.\mathbf{list}(S)$ presented in Example 4.1. This segment predicate is written down $\alpha.\mathbf{list} \mathrel{-\!\!*\!\!\!\!=}\{S_l \boxdot S_r\}\models \alpha'.\mathbf{list}$.

The derivation of the empty rule is straightforward: $\mathbf{emp} \wedge \alpha = \alpha' \wedge S_l = [] \wedge S_r = []$.

Since there is only one recursive instance in all rules of **list**, the segment predicate contains only one non-empty rule. The memory part of this rule is obtained by replacing the recursive instance $\beta_n.\mathbf{list}(S')$, by the recursive instance of the segment predicate:. This corresponds to the following separation logic formula:

$$\alpha.\mathtt{next} \mapsto \beta_n * \alpha.\mathtt{data} \mapsto \beta_d * \beta_n.\mathbf{list} \mathrel{-\!\!*\!\!\!\!=}\{S'_l \boxdot S'_r\}\models \alpha'.\mathbf{list}$$

The numerical part of the rule, $\alpha \neq 0$, is added to the segment rule. Finally, replacing sequence parameters by their segment counterparts yields the following equality: $S_l \boxdot S_r = [\beta_d].S'_l \boxdot S'_r$. This equality corresponds to the definitions: $S_l = [\beta_d].S'_l$ and $S_r = S'_r$.

To conclude, the singly linked list segment predicate is:

$$\begin{aligned}\alpha.\mathbf{list} \mathrel{-\!\!*\!\!\!\!=}\{S_l \boxdot S_r\}\models \alpha'.\mathbf{list} := {} & \mathbf{emp} \wedge \alpha = \alpha' \wedge S_l = [] \wedge S_r = [] \\ \mid {} & \exists \beta_n, \beta_d, S'_l, S'_r, \alpha.\mathtt{next} \mapsto \beta_n * \alpha.\mathtt{data} \mapsto \beta_d \\ & * \beta_n.\mathbf{list} \mathrel{-\!\!*\!\!\!\!=}\{S'_l \boxdot S'_r\}\models \alpha'.\mathbf{list} \\ & \wedge \alpha \neq 0 \\ & \wedge S_l = [\beta_d].S'_l \wedge S_r = S'_r\end{aligned}$$

Similarly, we derive the segment counterpart of the **addrList** predicate introduced in Example 4.3. Since this predicate has two sequence parameters, S_a and S_v, the segment predicate has four, $S_{a,l}, S_{a,r}, S_{v,l}$, and $S_{v,r}$.

$$\begin{aligned}\alpha.\mathbf{addrList} \mathrel{-\!\!*\!\!\!\!=}\{S_{a,l} \boxdot S_{a,r}, S_{v,l} \boxdot S_{v,r}\}\models \alpha'.\mathbf{addrList} := & \\ \mathbf{emp} \wedge \alpha = \alpha' \wedge S_{a,l} = [] \wedge S_{a,r} = [] \wedge S_{v,l} = [] \wedge S_{v,r} = [] & \\ \mid \exists \beta_n, \beta_d, S'_{a,l}, S'_{a,r}, S'_{v,l}, S'_{v,r}, \alpha.\mathtt{next} \mapsto \beta_n * \alpha.\mathtt{data} \mapsto \beta_d & \\ * \beta_n.\mathbf{addrList} \mathrel{-\!\!*\!\!\!\!=}\{S'_{a,l} \boxdot S'_{a,r}, S'_{v,l} \boxdot S'_{v,r}\}\models \alpha'.\mathbf{addrList} & \\ \wedge \alpha \neq 0 & \\ \wedge S_{a,l} = [\alpha].S'_{a,l} \wedge S_{a,r} = S'_{a,r} & \\ \wedge S_{v,l} = [\beta_d].S'_{v,l} \wedge S_{v,r} = S'_{v,r} &\end{aligned}$$

Finally, the segment version of the nested list inductive predicate introduced in Example 4.6 is derived in a similar fashion. The outcome of this derivation is:

$$\begin{aligned}\alpha.\mathbf{nestedLists} \mathrel{-\!\!*\!\!\!\!=}\{S_l \boxdot S_r\}\models \alpha'.\mathbf{nestedLists} := & \\ \mathbf{emp} \wedge \alpha = \alpha' \wedge S_l = S_r = [] & \\ \mid \exists \beta_d, \beta_n, S', S'', \alpha.\mathtt{next} \mapsto \beta_n * \alpha.\mathtt{data} \mapsto \beta_d & \\ * \beta_n.\mathbf{nestedLists} \mathrel{-\!\!*\!\!\!\!=}\{S'_l \boxdot S'_r\}\models \alpha'.\mathbf{nestedLists} := & \\ * \beta_d.\mathbf{list}(S'') & \\ \wedge \alpha \neq 0 & \\ \wedge S_l = S''.S'_l \wedge S_r = S'_r &\end{aligned}$$

Example 4.8: Tree segment predicate

This example demonstrates how to derive the segment counterpart of the $\alpha.\mathbf{tree}(\kappa_p, S)$ inductive predicate used in WFS presented in Example 4.5. The segment predicate has the following form: $\alpha.\mathbf{tree}(\kappa_p) \mathrel{-\!\!*\!\!\!\!=}\{S_1 \boxdot S_2\}\models \alpha'.\mathbf{tree}(\kappa'_p)$.

Once again, the empty segment rule is derived immediately:

$$\mathbf{emp} \wedge \alpha = \alpha' \wedge \kappa_p = \kappa'_p \wedge S_1 = [] \wedge S_2 = []$$

The **tree** predicate has a total of two recursive instances. Therefore, the segment predicate has two non-empty rules: either the end of the segment is in the left subtree or in the right one.

In the case where the end of the segment is in the left subtree, the memory part is obtained by replacing $\beta_l.\mathbf{tree}(\alpha, S_l)$ by the recursive segment instance. This yields the following separation

$$
\begin{aligned}
\alpha.\mathbf{tree}(\kappa_p) \mathrel{-\!\!*}\{S_1 \boxdot S_2\}\models \alpha'.\mathbf{tree}(\kappa'_p) := {} & \mathbf{emp} \wedge \alpha = \alpha' \wedge \kappa_p = \kappa'_p \wedge S_1 = [] \wedge S_2 = [] \\
& \mid \exists \beta_t, \beta_l, \beta_r, \tau, S'_1, S'_2, S_r, \\
& \quad \alpha.\mathtt{task} \mapsto \beta_t * \alpha.\mathtt{parent} \mapsto \kappa_p \\
& \quad * \alpha.\mathtt{left} \mapsto \beta_l * \alpha.\mathtt{right} \mapsto \beta_r * \beta_t.\mathbf{task}(\tau) \\
& \quad * \beta_l.\mathbf{tree}(\alpha) \mathrel{-\!\!*}\{S'_1 \boxdot S'_2\}\models \alpha'.\mathbf{tree}(\kappa'_p) \\
& \quad * \beta_r.\mathbf{tree}(\alpha, S_r) \\
& \quad \wedge \alpha \neq 0 \\
& \quad \wedge S_1 = S'_1 \wedge S_2 = S'_2.[\tau].S_r \\
& \mid \exists \beta_t, \beta_l, \beta_r, \tau, S_l, S'_1, S'_2, \\
& \quad \alpha.\mathtt{task} \mapsto \beta_t * \alpha.\mathtt{parent} \mapsto \kappa_p \\
& \quad * \alpha.\mathtt{left} \mapsto \beta_l * \alpha.\mathtt{right} \mapsto \beta_r * \beta_t.\mathbf{task}(\tau) \\
& \quad * \beta_l.\mathbf{tree}(\alpha, S_l) \\
& \quad * \beta_r.\mathbf{tree}(\alpha) \mathrel{-\!\!*}\{S'_1 \boxdot S'_2\}\models \alpha'.\mathbf{tree}(\kappa'_p) \\
& \quad \wedge \alpha \neq 0 \\
& \quad \wedge S_1 = S_l.[\tau].S'_1 \wedge S_2 = S'_2
\end{aligned}
$$

Figure 4.4: Binary tree segment predicate

logic formula:

$$
\begin{aligned}
& \alpha.\mathtt{task} \mapsto \beta_t * \alpha.\mathtt{parent} \mapsto \kappa_p \\
* \, & \alpha.\mathtt{left} \mapsto \beta_l * \alpha.\mathtt{right} \mapsto \beta_r * \beta_t.\mathbf{task}(\tau) \\
* \, & \beta_l.\mathbf{tree}(\alpha) \mathrel{-\!\!*}\{S'_1 \boxdot S'_2\}\models \alpha'.\mathbf{tree}(\kappa'_p) \\
* \, & \beta_r.\mathbf{tree}(\alpha, S_r)
\end{aligned}
$$

The numerical part of the rule, $\alpha \neq 0$, forms the numerical part of the segment predicate rule. Finally, to obtain the sequence constraints, let us replace S by $S_1 \boxdot S_2$ and S_l by $S'_1 \boxdot S'_2$ in $S = S_l.[\tau].S_r$. The outcome of the substitution is $S_1 \boxdot S_2 = S'_1 \boxdot S'_2.[\tau].S_r$. After matching each side of the placeholder symbol, we deduce the sequence constraints: $S_1 = S'_1$ and $S_2 = S'_2.[\tau].S_r$.

The third rule, where the end of the segment is in the right subtree is inferred in a similar manner. The right instance of the **tree** predicate is replaced by the recursive instance of the segment predicate $\beta_r.\mathbf{tree}(\alpha) \mathrel{-\!\!*}\{S'_1 \boxdot s'_2\}\models \alpha'.\mathbf{tree}(\kappa'_p)$. The sequence constraints are obtained by replacing full predicate parameters by their segment counterparts and matching each side of the placeholder:

$$
\begin{aligned}
& S\,[S_1 \boxdot S_2/S] = (S_l.[\tau]\,S_r)\,[S'_1 \boxdot S'_2/S_r] \\
\equiv\ & S_1 \boxdot S_2 = S_l.[\tau].S'_1 \boxdot S'_2 \\
\equiv\ & \begin{cases} S_1 = S_l.[\tau].S'_1 \\ S_2 = S'_2 \end{cases}
\end{aligned}
$$

To conclude, the full definition of the binary tree segment predicate derived is presented in Figure 4.4.

Graphical representation As mentioned in Section 2.3.2, separation logic formulas are manipulated as graphs. Therefore, this graph representation is updated to take into account sequence parameters. For both instances of full predicates $\alpha.\mathbf{pred}(\vec{\kappa}, S)$ and instances of segment predicates, $\alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*}\{S_l \boxdot S_r\}\models \beta.\mathbf{pred}(\vec{\kappa})$, the sequence parameters are displayed at the center of the thick edge denoting an inductive predicate. The numerical parameters $\vec{\kappa}$ and $\vec{\kappa}'$ are still represented at either the source or the end of the predicate edge. Figure 4.5 presents the graph representation of full and segment inductive predicates.

4.1.2.3 Concatenating segment predicates with other predicates

Following this discussion, we can now study the concatenation of segment predicates with full and other segment predicates. If we concatenate an instance of a segment predicate $\alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*}\{S_l \boxdot S_r\}\models \alpha'.\mathbf{pred}(\vec{\kappa}')$ with a full instance $\alpha'.\mathbf{pred}(\vec{\kappa}', S_0)$, we obtain a full inductive predicate whose sequence parameter is obtained by replacing the placeholder symbol by the sequence in the full predicate. This

(a) Graphical representation of $\alpha.\mathbf{pred}(\vec{\kappa}, S)$

(b) Graphical representation of $\alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*\!\!\!\!\!\!\{S_l \boxdot S_r\}\!\!\models} \beta.\mathbf{pred}(\vec{\kappa})$

Figure 4.5: Graphical representation of inductive predicates

corresponds to the sequence expression $S_l \boxdot S_r[S_0/\boxdot] = S_l.S_0.S_r$. Therefore, the concatenation of a segment and a full predicate to form another full instance $\alpha.\mathbf{pred}(\vec{\kappa}, S)$, holds if S is equal to $S_l.S_0.S_r$.

Lemma 4.2: Concatenation lemma (segment/full case)

Let $\alpha.\mathbf{pred}(\overrightarrow{\kappa}, S)$ be an inductive predicate such that S is an additive predicate. For any concrete heap m, and any numerical and sequence valuations σ_n, σ_s, and any symbolic variables $\alpha, \alpha', \vec{\kappa}, \vec{\kappa}', S, S_l, S_r, S_0$, if:

- $\sigma_n, \sigma_s \vDash_s S = S_l.S_0.S_r$,
- $m, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*\!\!\!\!\!\!\{S_l \boxdot S_r\}\!\!\models} \alpha'.\mathbf{pred}(\vec{\kappa}') * \alpha'.\mathbf{pred}(\vec{\kappa}', S_0)$,

then $m, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}, S)$.

Proof. Let $m, \sigma_n, \sigma_s, \alpha, \alpha', \vec{\kappa}, \vec{\kappa}', S_l, S_r, S_0$ satisfying the hypotheses of the lemma. The second hypothesis entails the existence of two disjoint concrete heaps m' and m'' such that:

- $m', \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*\!\!\!\!\!\!\{S_l \boxdot S_r\}\!\!\models} \alpha'.\mathbf{pred}(\vec{\kappa}')$,
- $m'', \sigma_n, \sigma_s \vDash_m \alpha'.\mathbf{pred}(\vec{\kappa}', S_0)$.
- $m = m' \uplus m''$

The proof proceeds by induction over the derivation of the first judgement.

Basic case The basic case corresponds to the only rule that does not contain any recursive instance of segment predicate, the empty rule:

$$\mathbf{emp} \wedge \alpha = \alpha' \wedge \vec{\kappa} = \vec{\kappa}' \wedge S_l = S_r = [\,]$$

This implies that $m' = \varnothing$, leading to $m = m''$. Additionally, since $\sigma_s(S_l) = \sigma_s(S_r) = \varepsilon$, we deduce from the first hypothesis that $\sigma_s(S) = \sigma_s(S_0)$. Similarly, we observe that numerical constraints of the empty rule entail that $\sigma_n(\alpha) = \sigma_n(\alpha')$ and that $\sigma_n(\vec{\kappa}) = \sigma_n(\vec{\kappa}')$. Therefore, by substituting the variables in the satisfiability judgement over m'', we prove that: $m, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}, S)$.

Recursive case Schematically, the proof of the recursive case works as follows: the satisfiability judgement entails that the concrete heap and the valuations satisfy some non-empty rule of the segment. This rule contains a recursive instance of the segment. Therefore, by using the induction hypothesis, this recursive instance is concatenated with the full predicate. This means that the concrete heap and the valuations satisfy the segment rule where the instance of the segment is replaced by a full predicate. By construction of the segment predicate this abstract memory heap corresponds to a rule of the full predicate.

We consider the case where m', σ_n, σ_s satisfy a recursive rule of the inductive segment predicate $\alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*\!\!\!\!\!\!\{S_l \boxdot S_r\}\!\!\models} \alpha'.\mathbf{pred}(\vec{\kappa}')$:

$$\begin{aligned}\exists \vec{\beta}, \vec{S_j}, S_{i,l}, S_{i,r}, m_k^\sharp * \Big(\circledast_{j \neq i}\, \beta_j.\mathbf{pred}(\vec{\delta}_{k,j}, S_j)\Big) \\ * \beta_i.\mathbf{pred}(\vec{\delta}_{k,i}) \mathrel{-\!\!*\!\!\!\!\!\!\{S_{i,l} \boxdot S_{i,r}\}\!\!\models} \alpha'.\mathbf{pred}(\vec{\kappa}') \\ \wedge \varphi_{\mathcal{V},k} \wedge \varphi_{s,k} \\ \wedge S_l = E'_k.S_{i,l} \wedge S_r = S_{i,r}.E''_k\end{aligned}$$

This means that there exist values $\vec{c}$, $\vec{w_j}, w_{i,l}, w_{i,r}$ for existentially quantified variables, updated numerical σ'_n and sequence σ'_s, and two distinct concrete heaps m'_l and m'_r such that:

(a) $\sigma'_n = \sigma_n[\vec{\beta} \mapsto \vec{c}]$,

(b) $\sigma'_s = \sigma_s[\vec{S_j} \mapsto \vec{w_j}]$,

(c) $m'_l, \sigma'_n, \sigma'_s \vDash_m m_k^\sharp * \left(\circledast_{j \neq i} \beta_j.\mathbf{pred}(\vec{\delta}_{k,j}, S_j)\right)$,

(d) $m'_r, \sigma'_n, \sigma'_s \vDash_m \beta_i.\mathbf{pred}(\vec{\delta}_{k,i}) \mathrel{-\!\!*}\{S_{i,l} \boxdot S_{i,r}\}\models \alpha'.\mathbf{pred}(\vec{\kappa}')$,

(e) $\sigma'_n \vDash_n \varphi_{\mathcal{V},k}$,

(f) $\sigma'_n, \sigma'_s \vDash_s \varphi_{s,k}$,

(g) $\sigma'_n, \sigma'_s \vDash_s S_l = E'_k.S_{i,l}$ and $\sigma'_n, \sigma'_s \vDash_s S_r = S_{i,r}.E''_k$.

Let us pick a fresh sequence variable S_i and let us update further the sequence valuation function[1] $\sigma''_s := \sigma'_s[S_i \mapsto \sigma_s(S_{i,l}).\sigma_s(S_0).\sigma_s(S_{i,r})]$. Since σ'_n and σ''_s differ from σ_n and σ_s only $\vec{\beta}, \vec{S}$, and S_i, *i.e.* variables that are not free in $\alpha'.\mathbf{pred}(\vec{\kappa}', S_0)$, we deduce from Lemma 4.1 that: $m'', \sigma'_n, \sigma''_s \vDash_m \alpha'.\mathbf{pred}(\vec{\kappa}', S_0)$. Moreover, by construction of σ''_s, the following judgement holds: $\sigma'_n, \sigma''_s \vDash_s S_i = S_{i,l}.S_0.S_{i,r}$. This means that we can apply the induction hypothesis to deduce that:

$$m'_r \uplus m'', \sigma'_n, \sigma''_s \vDash_s \beta_i.\mathbf{pred}(\vec{\delta}_{k,i}, S_i)$$

Given that S_i do not appear in the rest of the segment rule and that σ'_s and σ''_s only differ over S_i, we can modify the judgement (c) according to Lemma 4.1 in the following manner:

$$m'_l, \sigma'_n, \sigma''_s \vDash_m m_k^\sharp * \left(\circledast_{j \neq i} \beta_j.\mathbf{pred}(\vec{\delta}_{k,j}, S_j)\right)$$

By combining the last two judgements, we deduce that:

$$\underbrace{m'_l \uplus m'_r \uplus m''}_{m}, \sigma'_n, \sigma''_s \vDash_m m_k^\sharp * \left(\circledast_{j} \beta_j.\mathbf{pred}(\vec{\delta}_{k,j}, S_j)\right)$$

Recall that by construction of segment predicate, this rule is computed using a rule from the full inductive predicate of the following form:

$$\exists \vec{\beta}, \vec{S}, m_k^\sharp * \left(\circledast_{j} \beta_j.\mathbf{pred}(\vec{\delta}_{k,j}, S_{k,j})\right) \wedge \varphi_{\mathcal{V},k} \wedge \varphi_{s,k} \wedge S = E'_k.S_i.E''_k$$

We established that the memory part is satisfied by m, σ'_n, σ''_s. Moreover, the numerical part as well as the sequence constraints $\varphi_{s,k}$ are also satisfied by σ'_n, σ''_s because S_i does not occur in $\varphi_{s,k}$. To conclude, by repeating applications of the substitution Lemma 3.1, we deduce that:

$$\begin{array}{ll} \sigma'_n, \sigma''_s \vDash_s S = S_l.S_0.S_r & \text{By hypothesis over } \sigma_n, \sigma_s \\ \Rightarrow \sigma'_n, \sigma''_s \vDash_s S = E'_k.S_{i,l}.S_0.S_{i,r}.E''_k & \text{By hypothesis over } \sigma'_s, \textit{ cf.} \text{ (g)} \\ \Rightarrow \sigma'_n, \sigma''_s \vDash_s S = E'_k.S_i.E''_k & \text{By definition of } \sigma''_s \end{array}$$

This means that m, σ'_n, σ''_s satisfy the rule of the full inductive predicate. Therefore, they satisfy the full predicate instance $\alpha.\mathbf{pred}(\vec{\kappa}, S)$. We conclude that m, σ_n, σ_s also satisfy the full predicate instance by using Lemma 4.1. □

It is also possible to concatenate a segment instance $\alpha.\mathbf{pred} \mathrel{-\!\!*}\{S'_l \boxdot S'_r\}\models \alpha'.\mathbf{pred}$ with another one $\alpha'.\mathbf{pred} \mathrel{-\!\!*}\{S''_l \boxdot S''_r\}\models \alpha''.\mathbf{pred}$. This combination forms a third segment $\alpha.\mathbf{pred} \mathrel{-\!\!*}\{S_l \boxdot S_r\}\models \alpha''.\mathbf{pred}$. Similarly to the segment/full concatenation case discussed above, the content of the resulting segment is obtained by replacing the placeholder symbol in the left segment by the content of the right one. That is to say: $S_l \boxdot S_r = (S'_l \boxdot S'_r)[S''_l \boxdot S''_r / \boxdot] = S'_l.S''_l \boxdot S''_r.S'_r$. This corresponds to the following constraints: $S_l = S'_l.S''_l$ and $S_r = S''_r.S'_r$.

[1] When defining σ''_s by setting a new value to S_i we omit to update the values of its attribute variables in the numerical and multiset valuations for the sake of readability. The full proof ensures that all valuations occurring at the left-hand side of any satisfiability judgement are consistent according to Definition 3.1. Updates of attribute variables, according to modification in the values of sequence variables will be omitted in the coming proofs.

Lemma 4.3: Concatenation lemma (segment/segment case)

Let $\alpha.\mathbf{pred}(\vec{\kappa}, S)$ be an inductive parameter such that S is an additive predicate. For any concrete heap m, and any numerical and sequence valuations σ_n, σ_s, and any symbolic variables $\alpha, \alpha', \alpha'', \vec{\kappa}, \vec{\kappa}', \vec{\kappa}'', S_l, S_l', S_l'', S_r, S_r', S_r''$, if:

- $\sigma_n, \sigma_s \vDash_s S_l = S_l'.S_l''$,
- $\sigma_n, \sigma_s \vDash_s S_r = S_r''.S_r'$,
- $m, \sigma_n, \sigma_s \vDash_m \begin{array}{l} \alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*}\{S_l' \boxdot S_r'\}\!\models \alpha'.\mathbf{pred}(\vec{\kappa}') \\ * \alpha'.\mathbf{pred}(\vec{\kappa}') \mathrel{-\!\!*}\{S_l'' \boxdot S_r''\}\!\models \alpha''.\mathbf{pred}(\vec{\kappa}'')' \end{array}$

then $m, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*}\{S_l \boxdot S_r\}\!\models \alpha''.\mathbf{pred}(\vec{\kappa}'')$.

Proof. The proof is similar to the proof of Lemma 4.2. □

4.1.3 Head parameter

The second kind of sequence parameters is a subclass of additive parameters. In essence, a sequence parameter is a *head parameter* if it denotes the sequence of addresses of the nodes described in the inductive predicate, excluding nodes summarized by nested predicates.

Definition 4.4: Head parameter

A sequence parameter S_i is a *head* parameter if and only if it is an additive parameter, and for each rule, the corresponding definition of this parameter contains only:

- parameters of recursive calls,
- the main parameter of the inductive instance α, when the cell part contains a points-to predicate $\alpha.\mathtt{f}_0 \mapsto \delta$ where the value of the field $\mathtt{f}_0$ is $\varphi_{\mathbb{F}}(\mathtt{f}_0) = 0$.

If we reason in terms of multiset constraints, the definition of a head parameter means that the content of S is formed by the content of the parameters of recursive instances S_i, and possibly α if the node is not empty. This boils down to the following multiset constraint:

$$\mathtt{mset}_S \subseteq \{\alpha\} \uplus_i \mathtt{mset}_{S_i}$$

Example 4.9: Head parameter

The parameter S_a from the **addrList** from Example 4.3 is an example of a head parameter.

Furthermore, since all these nodes are combined using the separating conjunction, their addresses are pairwise distinct. Therefore, a head sequence parameter does not contain any repeating element. This holds also in the case of a segment predicate.

Lemma 4.4: Uniqueness of head parameter

Let $\alpha.\mathbf{pred}(\vec{\kappa}, S)$ be an inductive predicate such that S is a head parameter. For any concrete heap m, any numerical and sequence valuation σ_n, σ_s, and any symbolic variables $\alpha, \vec{\kappa}, S$, if: $m, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}, S)$, then $\sigma_n, \sigma_s \vDash_s \mathbf{unique}(S)$,

Similarly, for any concrete heap m, symbolic valuations σ_n, σ_s, and symbolic variables α, α', $\vec{\kappa}$, $\vec{\kappa}'$, S_l, S_r, if $m, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{-\!\!*}\{S_l \boxdot S_r\}\!\models \alpha'.\mathbf{pred}(\vec{\kappa}')$, then $\sigma_n, \sigma_s \vDash_s \mathbf{unique}(S_l)$ and $\sigma_n, \sigma_s \vDash_s \mathbf{unique}(S_r)$.

Proof. By induction over the derivation of the satisfiability judgement $m, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}, S)$, we prove that $\sigma_n, \sigma_s \vDash_s \mathbf{unique}(S)$ and that any element in $\sigma_s(S)$ is in $\mathbf{supp}(m)$.

Base cases The base cases correspond to cases where the rule used to establish the satisfiability judgement does not contain any recursive instance of the predicate. Two definitions of S are possible here.

The first definition is $S = \mathtt{[]}$. This case is straightforward.

In the second definition, S boils down to $\mathtt{[}\alpha\mathtt{]}$. This case is feasible only if the memory part

contains an inductive predicate of the form $\alpha.\mathtt{f}_0 \mapsto \delta$. This means that $\sigma_n(\alpha) + \varphi(\mathtt{f}_0) = \sigma_n(\alpha) \in \mathbf{supp}(m)$.

Recursive cases The recursive cases arise when the rule used to establish the satisfiabilty judgement contains recursive instances of the inductive predicate $\mathbf{pred}(S_i)$. For the sake of brevity, let us focus on the situation where α occurs in the sequence definition of S. By the definition of a head parameter, this means that the memory part of the rule can be rewritten as follows:

$$\alpha.\mathtt{f}_0 \mapsto \delta * \left(\circledast_j \beta_j.\mathbf{pred}(\vec{\delta_j}, S_j)\right) * m^{\sharp\prime}$$

As a consequence, there exists disjoint concrete memory heaps m_α, m_j, and m' such that:

- $m_\alpha, \sigma_n, \sigma_s \vDash_m \alpha.\mathtt{f}_0 \mapsto \delta$
- $m_j, \sigma_n, \sigma_s \vDash_m \beta_j.\mathbf{pred}(\vec{\delta_j}, S_j)$
- $m', \sigma_n, \sigma_s \vDash_m m^{\sharp\prime}$

From the sequence definition of S we deduce that any element of $\sigma_s(S)$ is either the value of $\sigma_n(\alpha)$ which belongs to the support of m_α, or an element of $\sigma_s(S_j)$ which is contained in the support of m_j by induction hypothesis. Therefore, any element of $\sigma_s(S)$ belongs in the support of m.

Additionally, for each pair of elements c, c' occurring at different positions in $\sigma_s(S)$, either both of them are part of the same recursive sequence parameter S_j, in that case they are distinct by induction hypothesis, or they are not. The latter case implies that c and c' belongs to distinct sub-concrete heaps of m. As a consequence, they are distinct since these heaps are disjoint. □

4.1.4 Left-only and right-only parameters

The third type of sequence parameters concerns additive parameters where the local part of their sequence definitions, *i.e.* the elements that describe the content of the node, occurs only on the left of this definition.

Definition 4.5: Left-only parameter

A sequence parameter S_i of an inductive predicate **pred** is a *left-only* parameter if and only if it is an additive parameter and each rule contains at most one recursive call $\beta.\mathbf{pred}(\vec{\kappa}', S')$ where the definition of S is $S = E.S'$.

Example 4.10: Left-only parameters

Sequence parameters of list-like predicates such as **list**, **addrList**, and **nestedLists** are all instances of left-only parameters.

Detecting left-only parameters is useful, since it helps to reduce the number of sequence parameters. Indeed, if all the local content occurs only one the left on the recursive instance, then this means that the right part of the corresponding segment predicate is always empty.

Lemma 4.5: Sequence parameters of left-only parameters

Let $\alpha.\mathbf{pred}(\vec{\kappa}, S)$ be an inductive predicate such that S is a left-only parameter. For any concrete heap m, any numerical and sequence valuation σ_n, σ_s, and any symbolic variables $\alpha, , \alpha', \vec{\kappa}, \vec{\kappa}', S_l, S_r$, if: $m, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}) \twoheadrightarrow\!\!\!\!\!\!\{S_l \boxdot S_r\}\!\!\!\!\!\!\models \alpha'.\mathbf{pred}(\vec{\kappa}')$, then $\sigma_n, \sigma_s \vDash_s S_r = \mathtt{[]}$.

Proof. The proof is established by induction on the derivation of the satisfiabilty judgement $m, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{pred}(\vec{\kappa}) \twoheadrightarrow\!\!\!\!\!\!\{S_l \boxdot S_r\}\!\!\!\!\!\!\models \alpha'.\mathbf{pred}(\vec{\kappa}')$.

Empty case The empty case is straightforward since the empty segment rule implies that both sequence parameters are empty.

Recursive cases By definition of a left-only parameter, the sequence constraints in recursive segment rules are of the form: $S_l \boxdot S_r = (E.S')[S'_l \boxdot S'_r / S'] = E.S'_l \boxdot S'_r$. This is equivalent to

$S_l = E.S'_l$ and $S_r = S'_r$. Using the induction hypothesis, we deduce that $\sigma_n, \sigma_s \vDash_s S_r = \texttt{[]}$. □

As a consequence of Lemma 4.5, in the case of a segment predicate containing a left-only parameter, the right component of the parameter is superfluous since it is always empty. Therefore, this component is omitted, and the segment predicate is simply rewritten as $\alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{*}\!\!\!\rightarrow\!\!\!\{S_l \boxdot\}\!\!\!\models \alpha'.\mathbf{pred}(\vec{\kappa}')$

Similarly, we define *right-only* sequence parameters. And we establish the right-only counterpart of Lemma 4.5, stating that in the case of an inductive predicate with a right-only sequence parameter, for all instances of segment predicate $\alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{*}\!\!\!\rightarrow\!\!\!\{S_l \boxdot S_r\}\!\!\!\models \alpha'.\mathbf{pred}(\vec{\kappa}')$, the left part parameter S_l is always empty. This entails that the segment can be shortened as $\alpha.\mathbf{pred}(\vec{\kappa}) \mathrel{*}\!\!\!\rightarrow\!\!\!\{\boxdot S_r\}\!\!\!\models \alpha'.\mathbf{pred}(\vec{\kappa}')$.

4.2 The reduced product domain

This section presents the elements of the abstract domain resulting from the combination of the separation-logic based domain and the sequence domain. It defines their concretization. It also states a result expressing that adding a constraint on a fresh variable does not alter the concretization of the abstract state.

4.2.1 Definition and concretization

4.2.1.1 Elements of the reduced product

The combination of the shape domain extended using sequence parameter $\mathbb{M}^\sharp$ with the sequence abstract domain $\mathbb{D}_s^\sharp$ defined in Chapter 3, follows the same construction as the one outlined in Section 2.3.4. We use two symbols, $\top_{\mathbb{S}}^\sharp$ and $\bot_{\mathbb{S}}^\sharp$, to distinguish extreme elements. Other elements of the domain are pairs of abstract memory states $m^\sharp \in \mathbb{M}^\sharp$ and sequence abstract values $\sigma^\sharp \in \mathbb{D}_s^\sharp$. These elements are called *non-extreme* abstract values. The latter sequence abstract value cannot be the minimal value $\bot_s^\sharp$ to ensure the coalescence of the combination.

Definition 4.6: Combined abstract domain

The combined abstract domain is defined as $\mathbb{S}^\sharp := \left\{\top_{\mathbb{S}}^\sharp, \bot_{\mathbb{S}}^\sharp\right\} \uplus \left(\mathbb{M}^\sharp \times \mathbb{D}_s^\sharp \setminus \{\bot_s^\sharp\}\right)$.

4.2.1.2 Concretization

To define the concretization of the reduced product between the shape and sequence domains, we follow a two-step process.

Extended concretization In the first step, we define an *extended concretization*. For each concrete memory state, this concretization keeps track of the valuations that were used to establish the memory satisfiability judgment and that link the memory part of the abstract value to its sequence part. Such valuations are called *relevant valuations* of the abstract state.

Definition 4.7: Extended concretization

The extended concretization of the combined domain $\gamma_e : \mathbb{S}^\sharp \longrightarrow \wp\left(\mathbb{S}_\Omega \times \mathbb{D}_s\right)$ is defined as:

$$\begin{aligned}
\gamma_e\left(\top_{\mathbb{S}}^\sharp\right) &:= \mathbb{S}_\Omega \times \mathbb{D}_s \\
\gamma_e\left(\bot_{\mathbb{S}}^\sharp\right) &:= \varnothing \\
\gamma_e\left((m^\sharp, \sigma^\sharp)\right) &:= \left\{ ((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \;\middle|\; \begin{array}{l} ((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_m(m^\sharp) \\ \wedge\, (\sigma_n, \sigma_m, \sigma_s) \in \gamma_s(\sigma^\sharp) \end{array} \right\}
\end{aligned}$$

Full concretization With the extended concretization established, we can proceed to the second step. The *full concretization* maps each abstract state to the set of concrete states it represents. In this process, valuations are existentially quantified within the definition of the concretization and are not included in its outcome.

Definition 4.8: Full concretization of $\mathbb{S}^\sharp$

The full concretization $\gamma_{\mathbb{S}} : \mathbb{S}^\sharp \to \wp(\mathbb{S}_\Omega)$ of the combined abstract domain is defined as:

$$\gamma_{\mathbb{S}}\left(s^\sharp\right) := \left\{ s \mid \exists (\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s, (s, (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_e(s^\sharp) \right\}$$

Remark 4.1: Link between the extended concretization and the concretization

If we define the **forget** $: \wp(\mathbb{S}_\Omega \times \mathbb{D}_s) \to \wp(\mathbb{S}_\Omega)$ operator as $\textbf{forget}(X) := \{s \mid \exists \rho, (s, \rho) \in X\}$, then the definition of the full concretization boils down to $\gamma_{\mathbb{S}} = \textbf{forget} \circ \gamma_e$.

Furthermore, by defining the concretization function $\gamma_\times : \wp(\mathbb{S}_\Omega) \to \wp(\mathbb{S}_\Omega \times \mathbb{D}_s)$ as $\gamma_\times(X^\sharp) := X^\sharp \times \mathbb{D}_s$, we obtain the following Galois connection:

$$\left(\wp(\mathbb{S}_\Omega \times \mathbb{D}_s), \subseteq\right) \underset{\textbf{forget}}{\overset{\gamma_\times}{\leftrightarrows}} \left(\wp(\mathbb{S}_\Omega), \subseteq\right)$$

In essence, this means that the result of the extended concretization bears more information than the outcome of the full concretization. Consequently, reasoning with the extended concretization is too restrictive to establish some soundness results. Similarly, certain properties necessitate reasoning over relevant valuations, which requires the use of extended concretization.

4.2.2 Support and instantiation lemmas

With the concretization defined, we can now define the support of an abstract state. For extreme values $\top^\sharp_{\mathbb{S}}$ and $\bot^\sharp_{\mathbb{S}}$ the support boils down to the empty set. Indeed, their extended concretizations do not depend on valuations. The support of extreme values is defined as the union of the variables that appear in the memory part and the support of the sequence part.

Definition 4.9: Support in the combined domain

The support operator of the combined abstract domain $\textbf{supp}^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \to \wp(\mathcal{V}_n \uplus \mathcal{V}_s)$ is defined as:

$$\begin{aligned} \textbf{supp}^\sharp_{\mathbb{S}}(\bot^\sharp_{\mathbb{S}}) &:= \varnothing \\ \textbf{supp}^\sharp_{\mathbb{S}}(\top^\sharp_{\mathbb{S}}) &:= \varnothing \\ \textbf{supp}^\sharp_{\mathbb{S}}(m^\sharp, \sigma^\sharp) &:= \textbf{fv}(m^\sharp) \cup \textbf{supp}^\sharp_s(\sigma^\sharp) \end{aligned}$$

The support defined above is sound. If two triple of valuations are equal when we restrict them to the support of some abstract state, then both must be in the extended concretization of this state or neither.

Theorem 4.1: Soundness of $\textbf{supp}^\sharp_{\mathbb{S}}$

For any abstract value $s^\sharp \in \mathbb{S}^\sharp$, any concrete state $s \in \mathbb{S}_\Omega$, and any tuples of valuations $(\sigma_n, \sigma_m, \sigma_s)$ and $(\sigma'_n, \sigma'_m, \sigma'_s)$, if:

- $\forall \alpha \in \textbf{supp}^\sharp_{\mathbb{S}}(s^\sharp) \cap \mathcal{V}, \sigma_n(\alpha) = \sigma'_n(\alpha)$,
- $\forall \mathscr{M} \in \textbf{supp}^\sharp_{\mathbb{S}}(s^\sharp) \cap \mathcal{V}_m, \sigma_m(\mathscr{M}) = \sigma'_m(\mathscr{M})$,
- $\forall S \in \textbf{supp}^\sharp_{\mathbb{S}}(s^\sharp) \cap \mathcal{V}_s, \sigma_s(S) = \sigma'_s(S)$,

then $(s, (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_e(s^\sharp) \iff (s, (\sigma'_n, \sigma'_m, \sigma'_s)) \in \gamma_e(s^\sharp)$

Proof. By the definition of the extended concretization of extended elements, this equivalence always holds.

For non-extreme elements, the soundness of $\textbf{supp}^\sharp_{\mathbb{S}}$ is the consequence of the soundness of $\textbf{supp}^\sharp_s$ and of Lemma 4.1. □

Thanks to the definition of the $\textbf{supp}^\sharp_{\mathbb{S}}$ operator, we can now state the instantiation lemmas. In essence, these lemmas assert that if we add a fresh variable $\alpha^\dagger$ in an abstract value $s^\sharp$, *i.e.* a variable that is not a member of $\textbf{supp}^\sharp_{\mathbb{S}}(s^\sharp)$, and if we constrain this variable using the $\textbf{guard}^\sharp_{\mathbb{S}}$ operator, then the full concretization of the abstract value is not modified. That is to say, it is a sound approximation of the identity function with respect to the full concretization.

This holds if it is possible to assign a value $\alpha^\dagger$ so that the constraint is feasible. Here, we limit the instantiation scheme to definition constraints. That is to say equality constraints where the left-hand side of the equality is the fresh variable $\alpha^\dagger$, and this variable does not appear in the right-hand side. Additionally, we assert that for each relevant numerical valuation, *i.e.* valuation in the extended concretization of $s^\sharp$, the evaluation of right-hand side expression is well-defined. That is to say, it does not contain any division by zero.

Since we did not define the $\mathbf{guard}^\sharp_{\mathbb{S}}$ operator yet, we simply assume at this point that it satisfies the following soundness condition, that will be established in Theorem 4.2: for any abstract state $s^\sharp$, and any constraint C,

$$\gamma_e(s^\sharp) \cap (\mathbb{S}_\Omega \times \{(\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s \mid \sigma_n, \sigma_m, \sigma_s \models C\}) \subseteq \gamma_e \circ \mathbf{guard}^\sharp_{\mathbb{S}}(s^\sharp, C)$$

Lemma 4.6: Instantiation lemma (numerical case)

For any non-extreme abstract state $(m^\sharp, \sigma^\sharp)$, fresh symbolic variable $\alpha^\dagger$, and numerical symbolic expressions $e_{\mathcal{V}}$ that do not contain free occurrences of $\alpha^\dagger$, if for all relevant valuations $((\sigma_n, \sigma_m, \sigma_s), (\rho, m)) \in \gamma_e(m^\sharp, \sigma^\sharp)$ of the state, $\mathbb{E}[\![e_{\mathcal{V}}]\!]_{\mathcal{V}}(\sigma_n)$ is well-defined, then $\gamma_{\mathbb{S}}(m^\sharp, \sigma^\sharp) \subseteq \gamma_{\mathbb{S}} \circ \mathbf{guard}^\sharp_{\mathbb{S}}((m^\sharp, \sigma^\sharp), \alpha^\dagger = e_{\mathcal{V}})$.

Proof. To establish the inclusion, let us consider a concrete state $(\rho, m) \in \gamma_{\mathbb{S}}(m^\sharp, \sigma^\sharp)$. By definition, this means that there exists a triple of valuations $(\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s$, such that $((\sigma_n, \sigma_m, \sigma_s), (\rho, m)) \in \gamma_e(m^\sharp, \sigma^\sharp)$. Let us define $\sigma'_n := \sigma_n \left[\alpha^\dagger \mapsto \mathbb{E}[\![e_{\mathcal{V}}]\!]_{\mathcal{V}}(\sigma_n)\right]$. The hypothesis of the lemma ensures that this numerical valuation is well-defined.

Since $\alpha^\dagger$ is not in the support of $m^\sharp, \sigma^\sharp$, the restrictions of σ'_n and σ_n to $\mathbf{supp}^\sharp_{\mathbb{S}}(m^\sharp, \sigma^\sharp)$ are equal. By soundness of $\mathbf{supp}^\sharp_{\mathbb{S}}$, we deduce that $((\sigma'_n, \sigma_m, \sigma_s), (\rho, m)) \in \gamma_e(m^\sharp, \sigma^\sharp)$.

Additionally, by definition of σ'_n and since $\alpha^\dagger$, does not occur in $e_{\mathcal{V}}$, $\sigma'_n(\alpha^\dagger) = \mathbb{E}[\![e_{\mathcal{V}}]\!]_{\mathcal{V}}(\sigma'_n)$. This implies that $\sigma'_n \vDash_n \alpha^\dagger = e_{\mathcal{V}}$.

By soundness of $\mathbf{guard}^\sharp_{\mathbb{S}}$, we infer that the updated numerical valuation is in the extended concretization of the guarded abstract state: $((\sigma'_n, \sigma_m, \sigma_s), (\rho, m)) \in \gamma_e \circ \mathbf{guard}^\sharp_{\mathbb{S}}((m^\sharp, \sigma^\sharp), \alpha^\dagger = e_{\mathcal{V}})$. Finally, using the definition of the full concretization, we conclude that $(\rho, m) \in \gamma_{\mathbb{S}} \circ \mathbf{guard}^\sharp_{\mathbb{S}}((m^\sharp, \sigma^\sharp), \alpha^\dagger = e_{\mathcal{V}})$. □

Similarly, for a fresh sequence variable $S^\dagger$, we can assert that it is equal to a sequence expression E that does not contain $S^\dagger$. Given that the evaluation of a sequence expression is always defined, there is no assumption over the set of relevant valuations.

Lemma 4.7: Instantiation lemma (sequence case)

For any non-extreme abstract state $(m^\sharp, \sigma^\sharp)$, any fresh sequence variable $S^\dagger$, and sequence symbolic expressions E such that $S^\dagger \notin \mathbf{fv}(E)$, $\gamma_{\mathbb{S}}(m^\sharp, \sigma^\sharp) \subseteq \gamma_{\mathbb{S}}(m^\sharp, \mathbf{guard}^\sharp_s(S^\dagger = E))$.

Proof. The proof is similar to the proof of Lemma 4.6. □

Remark 4.2: Representation of unbounded environments

Following the classification proposed in [Lem24], our approach is *value-based*. That is to say, our analysis may manipulate an unbounded number of memory locations. In order to address this issue, our analysis adds fresh symbolic variables during unfolding (see below). To keep track of variables possibly used by the analysis, the implementation relies on a *key-allocator* that collects all symbolic variables manipulated by the current analysis state. This notion of explicit support can be formalized using a family of abstract domains $\mathbb{D}^\sharp_{\mathcal{S}}$. Each domain corresponds to a specific set $\mathcal{S}$ of symbolic variables that can be constrained by abstract values in the domain. Thus, the whole abstract domain can be constructed as a dependent sum $\sum_{\mathcal{S} \in \wp_{fin}(\mathcal{V})} . \mathbb{D}^\sharp_{\mathcal{S}}$. In order to define the concretization as well as binary operators on values over different domains, one can employ *cofibered domains* from Venet [Ven96].

$$
\begin{aligned}
\mathbf{guard}^\sharp_{\mathbb{S}} &: \mathbb{S}^\sharp \times (\mathcal{C}_n \uplus \mathcal{C}_s) \longrightarrow \mathbb{S}^\sharp \\
\mathbf{guard}^\sharp_{\mathbb{S}}((m^\sharp, \sigma^\sharp), \alpha = \beta) &:= \begin{cases} \bot^\sharp_{\mathbb{S}} & \text{if } m^\sharp = \alpha.\mathtt{f} \mapsto \delta * \beta.\mathtt{f} \mapsto \delta' * m^{\sharp\prime} \\ m^\sharp\,[\alpha/\beta]\,, \mathbf{guard}^\sharp_n(\sigma^\sharp, \alpha = \beta) & \text{otherwise} \end{cases} \\
\mathbf{guard}^\sharp_{\mathbb{S}}((m^\sharp, \sigma^\sharp), C_n) &:= \left(m^\sharp, \mathbf{guard}^\sharp_n(\sigma^\sharp, C_n)\right) \\
\mathbf{guard}^\sharp_{\mathbb{S}}((m^\sharp, \sigma^\sharp), C_s) &:= \left(m^\sharp, \mathbf{guard}^\sharp_s(\sigma^\sharp, C_s)\right)
\end{aligned}
$$

Figure 4.6: Definition of $\mathbf{guard}^\sharp_{\mathbb{S}}$

4.3 Abstract transfer function operators

This section presents the abstract operators for the abstract transfer functions of the combined abstract domain $\mathbb{S}^\sharp$. Similarly to the presentation of Section 2.4, we define operators only on non-extreme values of $\mathbb{S}^\sharp$. These definitions can be extended to extreme elements through strictness, as well as disjunctive values using pointwise lifting of these operators.

4.3.1 Symbolic guard

The first operator is the symbolic guard operator $\mathbf{guard}^\sharp_{\mathbb{S}}$ used in the instantiation lemmas. It inputs a symbolic constraint *i.e.* either a numerical constraint over symbolic variables or a sequence constraint and an abstract state. The outcome is an abstract state refined thanks to the new constraint. Its definition is presented in Figure 4.6.

When the new constraint is an equality between two symbolic variables, then the $\mathbf{guard}^\sharp_{\mathbb{S}}$ operator considers two cases.

- The first case arises when the memory part of the abstract value contains two points-to predicates at addresses $\alpha.\mathtt{f}$ and $\beta.\mathtt{f}$ respectively. In this situation, the new constraint implies that these two addresses are equal which violates the separating conjunction. Therefore, the whole abstract value is reduced to $\bot^\sharp_{\mathbb{S}}$.
- In the second case, the constraint is used to replace all occurrences of β by α in the memory part of the abstract value. Additionally, the new constraint is guarded in the sequence part of the abstract state using the $\mathbf{guard}^\sharp_n$ operator.

When the constraint is a generic numerical constraint or a sequence constraint, then $\mathbf{guard}^\sharp_{\mathbb{S}}$ boils down to applying the corresponding operator to the sequence part of the abstract value.

The resulting abstract operator is sound: its concretization contains all concrete states with their valuations that are abstracted by the input abstract value and that satisfy the new constraint.

Theorem 4.2: Soundness of $\mathbf{guard}^\sharp_{\mathbb{S}}$

For any abstract value $s^\sharp$, any numerical constraint over symbolic variables $C_n \in \mathcal{C}_n$,

$$\gamma_e(s^\sharp) \cap (\mathbb{S}_\Omega \times \{(\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s \mid \sigma_n \vDash_n C_n\}) \subseteq \gamma_e \circ \mathbf{guard}^\sharp_{\mathbb{S}}(s^\sharp, C_n)$$

Similarly, for any abstract value $s^\sharp$, any sequence constraint $C_s \in \mathcal{C}_s$,

$$\gamma_e(s^\sharp) \cap (\mathbb{S}_\Omega \times \{(\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s \mid \sigma_n, \sigma_s \vDash_s C_s\}) \subseteq \gamma_e \circ \mathbf{guard}^\sharp_{\mathbb{S}}(s^\sharp, C_s)$$

Proof. Let (ρ, m) be a concrete state and $(\sigma_n, \sigma_m, \sigma_s)$ a tuple of valuations. The proof works by case analysis over the case used in the definition of $\mathbf{guard}^\sharp_{\mathbb{S}}$.

Bottom case If the constraints is a numerical equality between symbolic variables $\alpha = \beta$, and there exists two points-to predicates in $m^\sharp$ from α and β and with the same offset $\mathtt{f}$, then the hypothesis implies that there exist disjoint concrete heaps m_α, m_β, and m' such that:

- $m_\alpha, \sigma_n, \sigma_s \vDash_m \alpha.\mathtt{f} \mapsto \delta$

- $m_\beta, \sigma_n, \sigma_s \vDash_m \beta.\mathtt{f} \mapsto \delta'$
- $m', \sigma_n, \sigma_s \vDash_m m^{\sharp\prime}$

From the definition of $\vDash_m$, we deduce that $m_\alpha = \{\sigma_n(\alpha) + \varphi_{\mathbb{F}}(\mathtt{f}) \mapsto \sigma_n(\delta)\}$ and that $m_\beta = \{\sigma_n(\beta) + \varphi_{\mathbb{F}}(\mathtt{f}) \mapsto \sigma_n(\delta')\}$. Additionally, since, $\sigma_n \vDash_n \alpha = \beta$, we infer that $\sigma_n(\alpha) = \sigma_n(\beta)$. This equality is absurd since m_α and m_β are disjoint concrete heaps. Therefore, the left hand-side of the inclusion corresponds to the empty set.

Non-bottom equality case In this case, the satisfiability of the new constraint also implies that $\sigma_n(\alpha) = \sigma_n(\beta)$. By syntactic induction over $m^\sharp$, we prove that $m, \sigma_n, \sigma_s \vDash_m m^\sharp[\alpha/\beta]$. The remaining of the proof is similar to the generic numerical constraint case presented below.

Generic numerical constraint

$$((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_e(m^\sharp, \sigma^\sharp) \cap (\mathbb{S}_\Omega \times \{(\sigma_n, \sigma_m, \sigma_s) \in \mathbb{D}_s \mid \sigma_n \vDash_n C_n\})$$

$$\Rightarrow \begin{cases} ((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_m(m^\sharp) \\ (\sigma_n, \sigma_m, \sigma_s) \in \gamma_s(\sigma^\sharp) \\ \sigma_n \vDash_n C_n \end{cases} \quad \text{by definition of } \gamma_m$$

$$\Rightarrow \begin{cases} ((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_m(m^\sharp) \\ (\sigma_n, \sigma_m, \sigma_s) \in \gamma_s \circ \mathbf{guard}^\sharp_n(\sigma^\sharp, C_n) \end{cases} \quad \text{by soundness of } \mathbf{guard}^\sharp_n$$

$$\Rightarrow ((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_e\left(m^\sharp, \mathbf{guard}^\sharp_n(\sigma^\sharp, C_n)\right) \quad \text{by definition of } \gamma_m$$

Sequence constraint The proof is similar to the generic numerical constraint one. □

4.3.2 Unfolding

The second type of operators used in the combined abstract domain are the predicate unfolding operators. These operators make explicit the memory cells summarized by an inductive predicate. Considering that there are several ways for the analysis to detect that some symbolic variable denotes the address of a cell summarized by a predicate instance, we define several kinds of unfolding.

4.3.2.1 Forward unfolding

The simplest form of predicate unfolding is called *forward unfolding*. It corresponds to the unfolding predicate described in Section 2.4. When the analysis must manipulate a memory cell at an address denoted by a symbolic variable α and when α is the main parameter of an instance of some inductive predicate $\mathbf{p}$, the $\mathbf{unfold}^\sharp_{\mathbb{S}}$ operator returns a disjunction of abstract states where the predicate instance is replaced by the memory parts of all rules of the definition of $\mathbf{p}$. The pure and sequence parts of the rules are assumed using the $\mathbf{guard}^\sharp_{\mathbb{S}}$ operator. To avoid variable names conflict, existentially quantified variables in the rules are replaced by fresh variables.

Definition 4.10: Forward unfolding

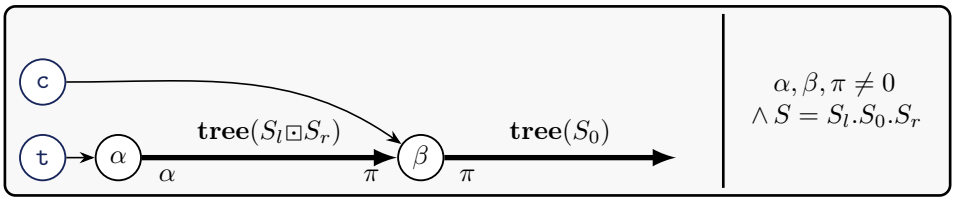
For an instance of inductive predicate $\alpha.\mathbf{p}(\vec{\kappa}, \vec{S})$, defined as $\alpha.\mathbf{p}(\vec{\kappa}, \vec{S}) := \bigvee_k \exists \vec{\beta}, \vec{S}', m^\sharp_k \wedge \varphi_{\mathcal{V},k} \wedge \varphi_{s,k}$, the unfolding predicate is defined as:

$$\mathbf{unfold}^\sharp_{\mathbb{S}}\left((\alpha.\mathbf{p}(\vec{\kappa}, \vec{S}) * m^{\sharp\prime}, \sigma^\sharp), \alpha\right) := \bigvee_k \mathbf{guard}^\sharp_{\mathbb{S}}\left(\left(m^{\sharp\flat}_k * m^{\sharp\prime}, \sigma^\sharp\right), \varphi^\flat_{\mathcal{V},k} \wedge \varphi^\flat_{s,k}\right)$$

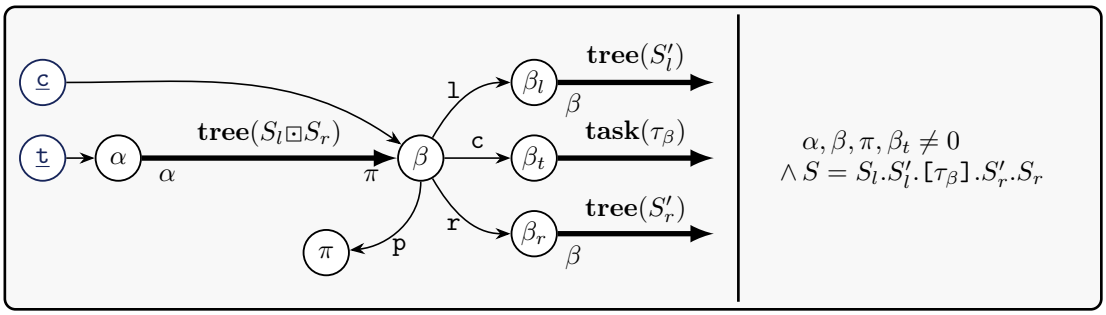
where the superscript $\bullet^\flat$ denotes the element $\bullet$ where all occurrences of existentially quantified variables $\vec{\beta}$ and $\vec{S}'$ have been replaced by fresh variables $\vec{\delta}^\dagger$ and $\vec{S}^\dagger$, *e.g.* $m^{\sharp\flat}_k := m^\sharp_k[\vec{\delta}^\dagger/\vec{\beta}][\vec{S}^\dagger/\vec{S}']$.

Example 4.11: Forward unfolding of tree

To illustrate the forward unfolding, let us consider the abstract state depicted in Figure 4.7a. This state corresponds to the exploration of a binary tree by a pointer `c`. Now let us examine the evaluation of expression `c->prev`. The variable `c` evaluates to β. Then the analysis attempts to



(a) Abstract state before the forward unfolding



(b) Abstract state resulting from the non-empty rules

Figure 4.7: Abstract state computed in the forward unfolding of the **tree** predicate

dereference this value. Given that β is the main parameter of the predicate instance $\beta.\mathbf{tree}(\pi, S_0)$, the analysis performs a forward unfolding on this instance.

The empty tree rule generates the constraint $\beta = 0$. Guarding this constraint reduces the sequence part of the abstract state to $\bot^\sharp_s$, since the constraint is inconsistent with the constraint stating that β is non-null. Consequently, the whole abstract state is reduced to $\bot^\sharp_{\mathbb{S}}$: the analysis discards it.

The non-empty rule introduces fresh variables β_l, β_r, β_t, τ_β, S'_l, and S'_r. The predicate instance is replaced by the memory part of the rule and the unfolding adds the constraints $\beta \neq 0$ and $S_0 = S'_l.[\tau_\beta].S'_r$. Finally, since the sequence variable S_0 no longer occurs in the memory part of the abstract state it is removed from the sequence part as well using the **prune**$^\sharp_s$. The outcome of the unfolding is presented in Figure 4.7b. In this abstract state, the expression `c->prev` evaluates to the symbolic variable π.

The unfolding operation is sound: the disjunction of all returned abstract states over-approximates the input abstract state.

Theorem 4.3: Soundness of unfold$^\sharp_{\mathbb{S}}$

For any abstract states $s^\sharp, s^\sharp_1, \ldots, s^\sharp_n$, and any symbolic variable α, if $\mathbf{unfold}^\sharp_{\mathbb{S}}(s^\sharp, \alpha) = \bigvee_k s^\sharp_k$, then $\gamma_{\mathbb{S}}\left(s^\sharp\right) \subseteq \bigcup_k \gamma_{\mathbb{S}}\left(s^\sharp_k\right)$.

Proof. Let (ρ, m) be a concrete state in $\gamma_{\mathbb{S}}(s^\sharp)$. By definition of the full concretization, this means that there exists a triple of valuations $(\sigma_n, \sigma_m, \sigma_s)$, and two disjunct concrete heaps $m_{\mathbf{p}}$ and m' such that:

- $m = m_{\mathbf{p}} \uplus m'$
- $m_{\mathbf{p}}, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{p}(\vec{\kappa}, \vec{S})$
- $m', \sigma_n, \sigma_s \vDash_m m^{\sharp\prime}$
- $\sigma_n, \sigma_m, \sigma_s \in \gamma_s(\sigma^\sharp)$.

The satisfiability judgment on $\alpha.\mathbf{p}(\vec{\kappa}, \vec{S})$ implies that there exists a rule index k, as well as updated valuations σ'_n, σ'_s such that:

- $m_{\mathbf{p}}, \sigma'_n, \sigma'_s \vDash_m m^\sharp_k$

- $\sigma'_n \vDash_n \varphi_{\mathcal{V},k}$
- $\sigma'_n, \sigma'_s \vDash_s \varphi_{s,k}$

Let us define the valuations, $\sigma^\flat_n, \sigma^\flat_m$ and $\sigma^\flat_s$, assigning to the fresh variables $\vec{\delta}^\dagger$ and $\vec{S}^\dagger$ the values assigned to existentially quantified variables in σ'_n and σ'_s. That is to say: $\sigma^\flat_n := \sigma_n[\delta^\dagger \mapsto \sigma'_n(\beta)]$ and $\sigma^\flat_s := \sigma_s[S^\dagger \mapsto \sigma'_s(S')]$.

By an induction over the formulas we establish that $\sigma^\flat_n \vDash_n \varphi^\flat_{\mathcal{V},k}$ and $\sigma^\flat_n, \sigma^\flat_s \vDash_s \varphi^\flat_{s,k}$. Likewise, we infer that $m_{\mathbf{p}}, \sigma^\flat_n, \sigma^\flat_s \vDash_m m^{\sharp\flat}_k$.

Additionally, since $\vec{\beta}^\dagger$ and $\vec{S}^\dagger$ are fresh variables, this means that $\sigma_n, \sigma_m, \sigma_s$ and $\sigma^\flat_n, \sigma^\flat_m, \sigma^\flat_s$ have equal reduction to the support of the abstract state $(m^\sharp, \sigma^\sharp)$. This implies that $m', \sigma^\flat_n, \sigma^\flat_s \vDash_m m^{\sharp\prime}$, as well as $\sigma^\flat_n, \sigma^\flat_m, \sigma^\flat_s \in \gamma_s(\sigma^\sharp)$. By definition of the partial concretization, we deduce that $\big((\rho, m), (\sigma^\flat_n, \sigma^\flat_m, \sigma^\flat_s)\big) \in \gamma_e(m^{\sharp\flat}_k * m^{\sharp\prime}, \sigma^\sharp)$.

Using the soundness of $\mathbf{guard}^\sharp_{\mathbb{S}}$, we conclude that:

$$\Big((\rho, m), (\sigma^\flat_n, \sigma^\flat_m, \sigma^\flat_s)\Big) \in \gamma_e \circ \underbrace{\mathbf{guard}^\sharp_{\mathbb{S}}\left(\left(m^{\sharp\flat}_k * m^{\sharp\prime}, \sigma^\sharp\right), \varphi^\flat_{\mathcal{V},k} \wedge \varphi^\flat_{s,k}\right)}_{s^\sharp_k}$$

This implies that $(\rho, m) \in \gamma_{\mathbb{S}}\left(s^\sharp_k\right)$. □

4.3.2.2 Backward unfolding

The second kind of predicate unfolding, called *backward unfolding* [CR08], materializes the end of a segment. For instance, let us consider the abstract state from Figures 4.7b. We examine the case where the analysis attempts to evaluate the task denoted by the expression `c->prev->content`. As explained in Example 4.11, the sub-expression `c->prev` evaluates to π. There is no points-to predicate nor any inductive predicate originating from π. However, π is the parameter denoting the backward pointer at the end of the segment between α and β. From here, two possible cases apply:

- Either the segment is empty. Then π, α, and β are all equal. In that case, the expression denotes the task in the node at address β.
- Or the segment contains at least one node. Therefore, the last node before the segment must be at address π. This means that we can safely materialize this last node.

Naturally, this is correct only if the parameter guiding this unfolding is an instance of a parameter denoting a backward pointer. Such parameters are called *backward parameters*.

Definition 4.11: Backward parameter

A numerical parameter π of an inductive predicate is a *backward parameter*, if in all recursive instances within its definition, this parameter is equal to the main parameter of the current instance.

Example 4.12: Backward pointer

In the **tree** inductive predicate, the numerical parameter π is a backward parameter. In both recursive instances in the definition of **tree**, $\beta_l.\mathbf{tree}(\alpha, S_l)$ and $\beta_r.\mathbf{tree}(\alpha, S_r)$, the numerical parameter is the main parameter, α, of the current instance of the **tree** predicate.

To describe how the backward unfolding operates, recall that the definition of a segment predicates respects the following pattern: it contains an empty rule, and for each rule in the definition of the full predicate indexed by a variable k, and for each recursive instance in this rule indexed by a variable i, the definition of the segment predicate contains a rule where this i^{th} instance is replaced by the recursive instance of the segment predicate. Remark that, by definition of a backward parameter, in this segment recursive instance, the backward parameter is equal to the main parameter α. To sum up, the definition of the segment predicate is as follows (for simplicity, the predicate has a single numerical parameter, the backward parameter, and only one pair of sequence parameters):

$$\textbf{b-unfold}^\sharp_{\mathbb{S}}\left(\left(m^\sharp * \alpha.\mathbf{p}(\pi) \mathrel{-\!\!\!*}\{S_l \boxdot S_r\}\models \alpha'.\mathbf{p}(\pi'), \sigma^\sharp\right), \alpha\right) :=$$

$$\textbf{guard}^\sharp_{\mathbb{S}}\left((m^\sharp, \sigma^\sharp), S_l = S_r = [] \wedge \alpha = \alpha' \wedge \pi = \pi'\right)$$

$$\bigvee_k \bigvee_i \textbf{guard}^\sharp_{\mathbb{S}} \left(m^\sharp * \alpha.\mathbf{p}(\pi) \mathrel{-\!\!\!*}\{S_l^\dagger \boxdot S_r^\dagger\}\models \pi.\mathbf{p}(\pi^\dagger) * m_k^{\sharp\flat}, \begin{array}{l} \beta_i = \alpha' \\ \wedge\, S_l = S_l^\dagger.E'^{\,\flat}_{k,i} \\ \wedge\, S_r = E''^{\,\flat}_{k,i}.S_r^\dagger \\ \wedge\, \varphi^\flat_{\mathcal{V},k} \\ \wedge\, \varphi^\flat_{s,k} \end{array} \right)$$

$$\text{where } \mathcal{X}^\flat := \mathcal{X}[\pi^\dagger/\pi'][\pi'/\alpha][\vec{\beta}^\dagger/\vec{\beta}][\vec{S}^\dagger/\vec{S}]$$

Figure 4.8: Definition of $\textbf{b-unfold}^\sharp_{\mathbb{S}}$

$$\begin{aligned} \alpha.\mathbf{p}(\pi) \mathrel{-\!\!\!*}\{S_l \boxdot S_r\}\models \alpha'.\mathbf{p}(\pi') := \mathbf{emp} \wedge \alpha = \alpha' \wedge S_l = S_r = [] \\ \bigvee_k \bigvee_i \exists \vec{\beta}, \vec{S_j}, S_{i,l}, S_{i,r}, m_k^\sharp * \beta_i.\mathbf{p}(\alpha) \mathrel{-\!\!\!*}\{S_{i,l} \boxdot S_{i,r}\}\models \alpha'.\mathbf{p}(\pi') \\ \wedge\, \varphi_{\mathcal{V},k} \wedge \varphi_{s,k} \\ \wedge\, S_l = E'_{k,i}.S_{i,l} \wedge S_r = S_{i,r}.E''_{k,i} \end{aligned}$$

The backward unfolding starts by applying the empty segment rule. It boils down to removing the segment predicate and guarding the emptiness constraints: $\alpha = \alpha'$, $\pi = \pi'$, and $S_l = S_r = []$.

Then, the backward unfolding proceeds with the non-empty cases. Given that the segment is non-empty, we know that π' is a node in the segment, the analysis can split the segment in the middle at the node π'. This yields the following memory state:

$$\alpha.\mathbf{p}(\pi) \mathrel{-\!\!\!*}\{S_l^\dagger \boxdot S_r^\dagger\}\models \pi'.\mathbf{p}(\pi^\dagger) * \pi'.\mathbf{p}(\pi^\dagger) \mathrel{-\!\!\!*}\{S_l'^\dagger \boxdot S_r'^\dagger\}\models \alpha'.\mathbf{p}(\pi')$$

The backward parameter at the split location as well as the sequence parameters in the two segments are set to be fresh variables $\pi^\dagger, S_l^\dagger, S_r^\dagger, S_l'^\dagger$, and $S_r'^\dagger$. The sequence variables in the obtained segments are linked to the parameters of the original segment using constraints similar to the ones from the segment/segment concatenation lemma (Lemma 4.3): $S_l = S_l^\dagger.S_l'^\dagger$ and $S_r = S_r'^\dagger.S_r^\dagger$.

Then, the backward unfolding performs a forward unfolding in the right-hand side segment. Given that the empty segment case has already been considered, the backward unfolding focuses on non-empty rules. It results in a disjunction of memory states:

$$\bigvee_k \bigvee_i \alpha.\mathbf{p}(\pi) \mathrel{-\!\!\!*}\{S_l^\dagger \boxdot S_r^\dagger\}\models \pi'.\mathbf{p}(\pi^\dagger) * m_k^{\sharp\flat} * \beta_i.\mathbf{p}(\pi') \mathrel{-\!\!\!*}\{S_l''^\dagger \boxdot S_r''^\dagger\}\models \alpha'.\mathbf{p}(\pi')$$

In each element of the disjunction, the part of the separation logic formula corresponding to the local part of the rule as well as the constraints are instantiated with fresh variables replacing existentially quantified variables. This unfolding generates some constraints over sequence parameters $S_l'^\dagger \boxdot S_r'^\dagger$: $S_l'^\dagger = E'^\flat_{k,i}.S_l''^\dagger$ and $S_r'^\dagger = S_r''^\dagger.E''^\flat_{k,i}$, where $E'^\flat_{k,i}$ and $E''^\flat_{k,i}$ denote the outcomes of the instantiation of the sequence expressions.

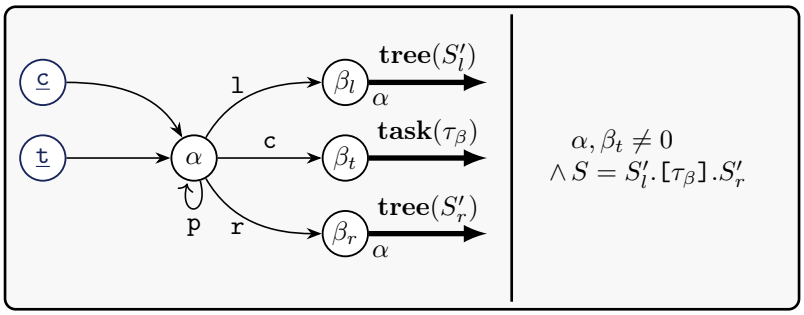
Finally, the backward unfolding removes the rightmost segment using the empty rule. This entails the following constraints: $\beta_i = \alpha'$ and $S_l''^\dagger = S_r''^\dagger = []$. After inlining the sequence constraints, we obtain constraints over S_l and S_r:

$$\begin{aligned} S_l = S_l^\dagger.S_l'^\dagger \left[E'^\flat_{k,i}.S_l''^\dagger / S_l'^\dagger\right]\left[[]/S_l''^\dagger\right] = S_l^\dagger.E'^\flat_{k,i} \\ S_r = S_r'^\dagger.S_r^\dagger \left[S_r''^\dagger.E''^\flat_{k,i} / S_r'^\dagger\right]\left[[]/S_r''^\dagger\right] = E''^\flat_{k,i}.S_r^\dagger \end{aligned}$$

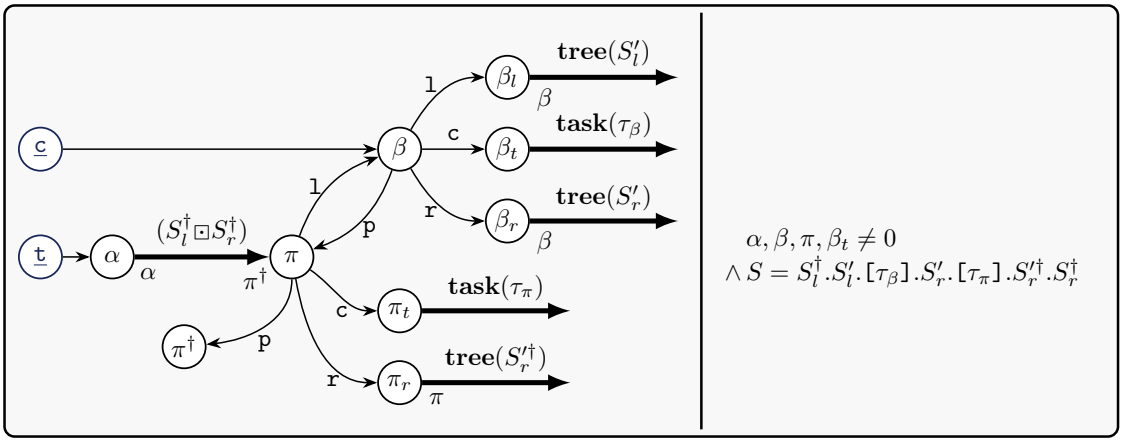
In conclusion, the definition of the backward unfolding operator $\textbf{b-unfold}^\sharp_{\mathbb{S}}$ is presented in Figure 4.8.

Example 4.13: Backward unfolding of tree

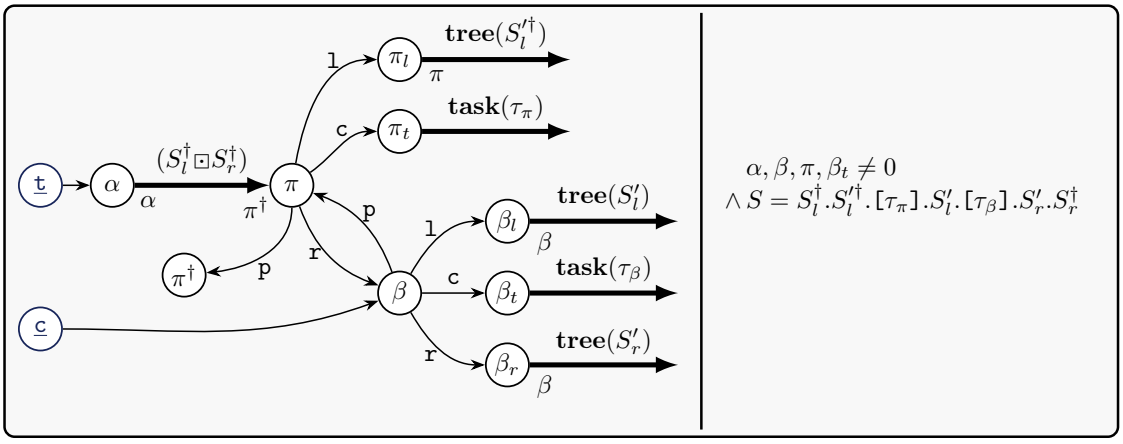
This example revisits the motivating example presented at the beginning of this section. After the forward unfolding of the predicate instance $\beta.\mathbf{tree}(\pi, S_0)$, the analysis yields the abstract state presented in Figure 4.7b.



(a) Empty case



(b) Left case



(c) Right case

Figure 4.9: Abstract state computed in the backward unfolding of the **tree** predicate

The empty case generates the following numerical constraints $\alpha = \beta$ and $\alpha = \pi$. Therefore, all occurrences of β and π are replaced by α in the memory part of the abstract state. Additionally, guarding the sequence constraints $S_l = S_r = \texttt{[]}$ simplifies the definition of S to $S = S'_l.\texttt{[}\tau_\beta\texttt{]}.S'_r$. Finally, given that the variables β, π, S_l, and S_r no longer appear in the shape part of the abstract value, they are discarded in the sequence part using $\mathbf{prune}^\sharp_s$. The result of the empty case is presented in Figure 4.9a.

Now, the backward unfolding considers the non-empty cases. It generates a segment between α and π with a fresh final backward parameter $\pi^\dagger$ and fresh sequence parameters $S_l^\dagger \boxdot S_r^\dagger$. There are two non-empty rules in the tree segment predicate. Let us consider the left one first. That is to say the rule where the segment is in the left subtree. The backward unfolding adds to the memory part of the abstract state the local part of the rule. This introduces variables π_t, π_r, τ_π, as well as predicates expressing the right subtree and the task corresponding to the node. The corresponding sequence constraints are $S_l = S_l^\dagger$ and $S_r = \texttt{[}\tau_\pi\texttt{]}.S_r'^\dagger.S_r^\dagger$. These constraints are added in the abstract state, and the variables S_l and S_r are removed from the sequence part. The abstract resulting from the left case is depicted in Figure 4.9b. For segment edges, we omit the name of the **tree** predicate since it is the only possible predicate for this example.

The backward unfolding derives the last case in a similar fashion. The result is the abstract state presented in Figure 4.9c.

To conclude, in the state resulting from the empty segment case, the result of the evaluation of expression `c->prev->content` is β_t. In the last two cases, the expression evaluates to π_t.

Theorem 4.4: Soundness of backward unfolding

For any abstract states $s^\sharp, s_1^\sharp, \ldots, s_n^\sharp$, and any symbolic variable α, if $\mathbf{b\text{-}unfold}^\sharp_{\mathbb{S}}(s^\sharp, \alpha) = \bigvee_k s_k^\sharp$, then $\gamma_{\mathbb{S}}\left(s^\sharp\right) \subseteq \bigcup_k \gamma_{\mathbb{S}}\left(s_k^\sharp\right)$.

Proof. Let (ρ, m) be a concrete state in $\gamma_{\mathbb{S}}(m^{\sharp\prime} * \alpha.\mathbf{p}(\pi) \mathrel{-\!\!\!*}\!\!\{S_l \boxdot S_r\}\!\models \alpha'._i p(\pi'), \sigma^\sharp)$. This implies that there exists disjoint concrete heaps m_s and m', as well as valuations $(\sigma_n, \sigma_m, \sigma_s)$ such that:

- $m = m_s \uplus m'$
- $m_s, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{p}(\pi) \mathrel{-\!\!\!*}\!\!\{S_l \boxdot S_r\}\!\models \alpha'.\mathbf{p}(\pi')$
- $m', \sigma_n, \sigma_s \vDash_m m^{\sharp\prime}$
- $\sigma_n, \sigma_m, \sigma_s \in \gamma_s(\sigma^\sharp)$.

For the sake of simplicity, this proof focuses solely on the separation logic formula as well as the sequence constraints on the segment parameters. It ignores the pure part and the remaining of the sequence part. The proof is done by case analysis.

Empty case The empty rule entails that $m_s = \varnothing$, $\sigma_n(\alpha) = \sigma_n(\alpha')$, $\sigma_n(\pi) = \sigma_n(\pi')$, and $\sigma_s(S_l) = \sigma_s(S_r) = \varepsilon$. Using the soundness condition of $\mathbf{guard}^\sharp_{\mathbb{S}}$, we deduce that:

$$((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_e \circ \mathbf{guard}^\sharp_{\mathbb{S}}\left((m^\sharp, \sigma^\sharp), S_l = S_r = \texttt{[]} \wedge \alpha = \alpha' \wedge \pi = \pi'\right)$$

Non-empty case For the non-empty rules, the proof proceeds by induction over the derivation of the satisfiability judgment. The induction hypothesis is: if $m_s \neq \varnothing$, then there exists a rule index k, variables $\pi^\dagger$, $S_l^\dagger$, $S_r^\dagger$, as well as updated valuations σ'_n and σ'_s such that:

$$\begin{cases} m_s, \sigma'_n, \sigma'_s \vDash_m \alpha.\mathbf{p}(\pi) \mathrel{-\!\!\!*}\!\!\{S_l^\dagger \boxdot S_r^\dagger\}\!\models \pi.\mathbf{p}(\pi^\dagger) * m_k^\sharp \\ \sigma'_n, \sigma'_s \vDash_s S_l = S_l^\dagger.E'_k \\ \sigma'_n, \sigma'_s \vDash_s S_r = E''_k.S_r^\dagger \end{cases}$$

Base case The base case, *i.e.* the empty segment rule, does not apply here since m_s must be non-empty.

Recursive case Let us assume that the satisfiability judgment comes from some non-empty rule indexed by a variable k': $m^\sharp_{k'} * \beta_i.\mathbf{p}(\alpha) \mathrel{-\!\!*}\!\{S'_l \boxdot S'_r\}\!\models \alpha'.\mathbf{p}(\pi') \wedge S_l = E'_k.S'_l \wedge S_r = S'_r.E''_k$. This implies that there exists values $\vec{c}$ and $\vec{w}$ for existentially quantified variables $\vec{\beta}$ and $\vec{S'}$, updated valuation $\sigma'_n = \sigma_n[\vec{\beta} \mapsto \vec{c}]$ and $\sigma'_s = \sigma_s[\vec{S'} \mapsto \vec{w}]$, and disjoint concrete heaps m'' and m'_s such that:

- $m_s = m' \uplus m'_s$
- $m'', \sigma'_n, \sigma'_s \vDash_m m^\sharp_{k'}$
- $m'_s, \sigma'_n, \sigma'_s \vDash_m \beta_i.\mathbf{p}(\alpha) \mathrel{-\!\!*}\!\{S'_l \boxdot S'_r\}\!\models \alpha'.\mathbf{p}(\pi')$
- $\sigma'_s, \sigma'_n \vDash_s S_l = E'_{k'}.S'_l$
- $\sigma'_s, \sigma'_n \vDash_s S_r = S'_r.E''_{k'}$

If m'_s is empty, then the only possible segment rule satisfying the judgment is the empty one. This implies that $m_s = m''$, $\sigma'_n(\alpha) = \sigma'_n(\pi)$, $\sigma'_n(\beta_i) = \sigma'_n(\alpha')$, and $\sigma_s(S'_l) = \sigma_s(S'_r) = \varepsilon$. From the definitions of σ'_n, we deduce that $\sigma_n(\alpha) = \sigma_n(\pi)$. To conclude this case, we simply prepend an empty segment to $m^\sharp_{k'}$. We define $\sigma''_n := \sigma'_n[\pi^\dagger \mapsto \pi]$ and $\sigma''_s := \sigma'_s[S'^\dagger_l, S'^\dagger_r \mapsto \varepsilon]$. This implies that:

$$\begin{cases} m_s, \sigma''_n, \sigma'_s \vDash_m \alpha.\mathbf{p}(\pi) \mathrel{-\!\!*}\!\{S^\dagger_l \boxdot S^\dagger_r\}\!\models \pi'.\mathbf{p}(\pi^\dagger) * m^\sharp_{k'} \\ \sigma''_n, \sigma''_s \vDash_s S_l = S^\dagger_l.E'_{k'} \\ \sigma''_n, \sigma''_s \vDash_s S_r = E''_{k'}.S^\dagger_r \end{cases}$$

If m'_s is non-empty, then the induction hypothesis applies. This means that there exists a rule index k, variables $\pi^\dagger$, $S'^\dagger_l$, $S'^\dagger_r$, as well as updated valuations σ''_n and σ''_s such that:

- $m'_s, \sigma''_n, \sigma''_s \vDash_m \beta_i.\mathbf{p}(\alpha) \mathrel{-\!\!*}\!\{S'^\dagger_l \boxdot S'^\dagger_r\}\!\models \pi.\mathbf{p}(\pi^\dagger) * m^\sharp_k$
- $\sigma'_n, \sigma'_s \vDash_s S'_l = S'^\dagger_l.E'_k$
- $\sigma'_n, \sigma'_s \vDash_s S'_r = E''_k.S'^\dagger_r$

This implies that we can find two disjunctive concrete heaps m''_s and m_k such that:

- $m'_s = m''_s \uplus m_k$
- $m''_s, \sigma''_n, \sigma''_s \vDash_m \beta_i.\mathbf{p}(\alpha) \mathrel{-\!\!*}\!\{S'^\dagger_l \boxdot S'^\dagger_r\}\!\models \pi'.\mathbf{p}(\pi^\dagger)$
- $m_k, \sigma''_n, \sigma''_s \vDash_m m^\sharp_k$

Let us define the sequence valuation:

$$\sigma'''_s := \sigma''_s \left[S^\dagger_l \mapsto \mathbb{E}[\![E'_{k'}.S'^\dagger_l]\!]_s(\sigma''_n, \sigma''_s); S^\dagger_r \mapsto \mathbb{E}[\![S'^\dagger_r.E''_{k'}]\!]_s(\sigma''_n, \sigma''_s) \right]$$

From this definition as well as the previous hypotheses, we deduce that:

$$\begin{cases} m_{k'} \uplus m''_s, \sigma''_n, \sigma'''_s \vDash_m m^\sharp_{k'} * \beta_i.\mathbf{p}(\alpha) \mathrel{-\!\!*}\!\{S'^\dagger_l \boxdot S'^\dagger_r\}\!\models \pi.\mathbf{p}(\pi^\dagger) \\ \sigma''_n, \sigma'''_s \vDash_s S^\dagger_l = E'_{k'}.S'^\dagger_l \\ \sigma''_n, \sigma'''_s \vDash_s S^\dagger_r = S'^\dagger_r.E''_{k'} \end{cases}$$

This matches some rule of the segment predicate. Therefore, we conclude that:

$$m_{k'} \uplus m''_s, \sigma''_n, \sigma'''_s \vDash_m \alpha.\mathbf{p}(\pi) \mathrel{-\!\!*}\!\{S^\dagger_l \boxdot S^\dagger_r\}\!\models \pi'.\mathbf{p}(\pi^\dagger)$$

Finally, by merging this satisfiability judgment with the one over m_k, we deduce that:

$$m_{k'} \uplus m''_s \uplus m_k, \sigma''_n, \sigma'''_s \vDash_m \alpha.\mathbf{p}(\pi) \mathrel{-\!\!*}\!\{S^\dagger_l \boxdot S^\dagger_r\}\!\models \pi'.\mathbf{p}(\pi^\dagger) * m^\sharp_k$$

Regarding the sequence constraints, we simply apply the substitution lemma (Lemma 3.1). For example for the constraint on S_l, we show that:

$$\begin{array}{rl} \sigma_n'', \sigma_s''' \vDash_s S_l = E'_{k'}.S'_l & \text{By unfolding constraint over } S_l \\ \Rightarrow \sigma_n'', \sigma_s''' \vDash_s S_l = E'_{k'}.S_l'^{\dagger}.E'_k & \text{By induction hypothesis} \\ \Rightarrow \sigma_n'', \sigma_s''' \vDash_s S_l = S_l^{\dagger}.E'_k & \text{By definition of } \sigma_s''' \end{array}$$

The remaining of the proof boils down to applying the soundness of $\mathbf{guard}^\sharp_{\mathbb{S}}$ as well as the definition of $\gamma_{\mathbb{S}}$. □

4.3.2.3 Non-local unfolding

The last kind of predicate unfolding, called *non-local unfolding* [LRC15], concerns the materialization of a cell at an arbitrary position in an inductive data structure. For instance, let us consider the predicate $\alpha.\mathbf{addrList}(S_a, S_v)$ such that S_a is constrained to be equal to $S'.[\beta].S''$. This means that β denotes the address of some node in the list. Therefore, it is correct to attempt to read the content of this memory cell. To do so, the analysis splits the predicate in the middle, at address β. It introduces a segment from α to β as well as a full predicate from β. The parameters of the predicates are set to fresh variables. This produces the following separation logic formula:

$$\alpha.\mathbf{addrList} \twoheadrightarrow\!\!\!\!\!\!\{S^\dagger_{a,l} \boxdot S^\dagger_{a,r}, S^\dagger_{v,l} \boxdot S^\dagger_{v,r}\}\!\!\models \beta.\mathbf{addrList} * \beta.\mathbf{addrList}(S^\dagger_{a,c}, S^\dagger_{v,c})$$

Additionally, the analysis asserts the fresh sequence variables satisfy the same constraints as the concatenation lemma: $S_a = S^\dagger_{a,l}.S^\dagger_{a,c}.S^\dagger_{a,r}$ and $S_v = S^\dagger_{v,l}.S^\dagger_{v,c}.S^\dagger_{v,r}$. Furthermore, the analysis guards the constraint $\beta \in \mathtt{mset}_{S_{a,c}}$ to ensure that the predicate $\beta.\mathbf{addrList}(S^\dagger_{a,c}, S^\dagger_{v,c})$ is not empty. Therefore, performing a forward unfolding on this predicate materializes the cell at address β.

Definition 4.12: Non-local unfolding (full predicate case)

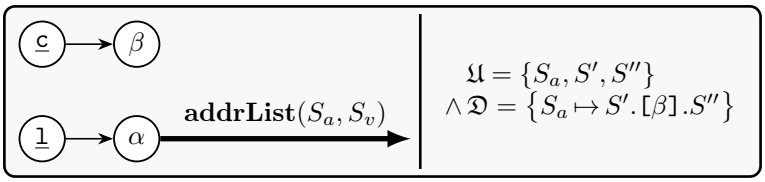
For an inductive predicate $\mathbf{p}(\vec{\kappa}, S)$ such that the sequence parameter S is a head parameter, and for all symbolic variables α and β, the non-local unfolding operator is defined as:

$$\begin{array}{l} \mathbf{nl\text{-}unfold}^\sharp_{\mathbb{S}}((\alpha.\mathbf{p}(\vec{\kappa}, S) * m^\sharp, \sigma^\sharp), \alpha, \beta) := \\ \qquad \left(\begin{array}{l} \alpha.\mathbf{p}(\vec{\kappa}) \twoheadrightarrow\!\!\!\!\!\!\{S^\dagger_l \boxdot S^\dagger_r\}\!\!\models \beta.\mathbf{p}(\vec{\kappa}^\dagger) \\ * \beta.\mathbf{p}(\vec{\kappa}^\dagger, S^\dagger_0) * m^\sharp \end{array}, \mathbf{guard}^\sharp_s \left(\begin{array}{l} \beta \in \mathtt{mset}_{S^\dagger_0} \\ \wedge S = S^\dagger_l.S^\dagger_0.S^\dagger_r \end{array}, \sigma^\sharp \right) \right) \\ \qquad \text{when } \mathbf{sat}^\sharp_s(\beta \in \mathtt{mset}_S) = \mathbf{true} \end{array}$$

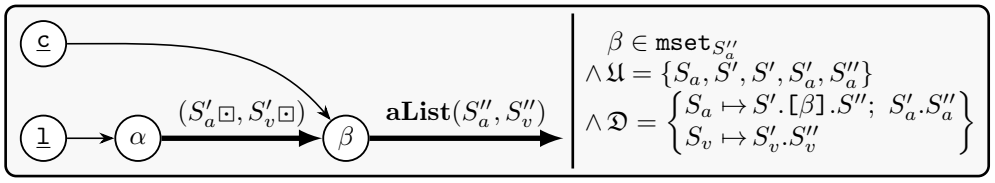
Example 4.14: Non-local unfolding

Let us investigate in detail the motivating example of this section. We consider the evaluation of the expression `c->data` in the abstract state presented in Figure 4.10a. Remark that since S_a is a head parameter, it is also without repeating elements. Hence, the sequence part of the abstract value contains the constraint $\mathbf{unique}(S_a)$. The sub-expression `c` evaluates to β. Since β is a member of $\mathtt{mset}_{S_a}$, the analysis performs a non-local unfolding on the full inductive predicate. This splits the predicate in two forming a segment predicate and another full predicate, and adds constraints between the sequence parameters of the original predicate and the result of the unfolding. Since both sequence parameters of the **addrList** predicate are left-only, we omit the right elements of the pairs in the parameters of the segment. Additionally, the non-local inserts the multiset constraint $\beta \in \mathtt{mset}_{S''_a}$. The outcome of the non-local unfolding is presented in Figure 4.10b.

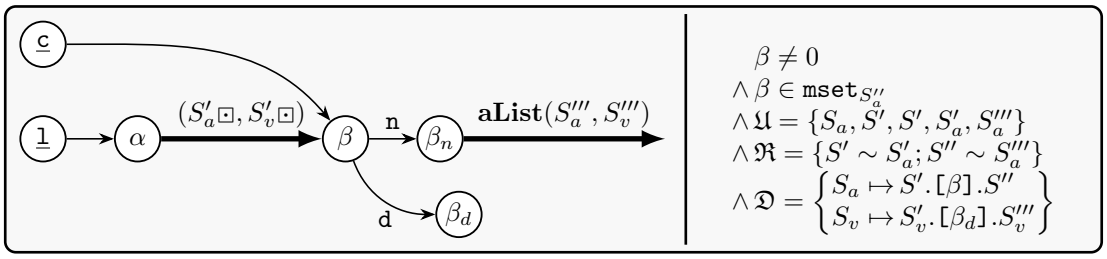
Thanks to the non-local unfolding, β is now the main parameter of an inductive predicate. The analysis unfolds it to materialize a memory cell at address β. The empty case is inconsistent. Indeed, the empty rule adds the sequence constraint $S''_a = [\,]$, which translates into the multiset constraint $\mathtt{mset}_{S''_a} = \{\}$. This constraint contradicts the one added by the non-local unfolding stating that β is a member of $\mathtt{mset}_{S''_a}$. The only feasible case is the non-empty rule. It inserts the numerical constraint $\beta \neq 0$ as well as definitions of S''_a and S''_v. That is to say: $S''_a = [\beta].S'''_a$ and $S''_v = [\beta_d].S'''_v$. After inserting the definition of S''_a, the sequence $\mathbf{guard}^\sharp_s$ operator notices that S_a has two possible definitions that share a common element, namely $S = S'.[\beta].S''$ and $S = S'_a.[\beta].S'''_a$. Consequently, the operator matches the two sides of the $[\beta]$ atom. This adds the sequence equalities $S' = S'_a$ and $S'' = S'''_a$. Finally, the unfolding operator removes sequence variables S''_a and S''_v from the sequence part of the abstract state since they are no longer in the memory part. To conclude, the result of the forward unfolding is presented in Figure 4.10c. In



(a) Initial state



(b) Abstract state after the non-local unfolding of α.**addrList**



(c) Abstract state after unfolding β.**addrList**

Figure 4.10: Abstract states computed during the evaluation of `c->data`

this state, the expression `c->data` evaluates to β_d.

Theorem 4.5: Soundness of non-local unfolding

For any abstract state $s^\sharp$ and $s^{\sharp\prime}$, and any symbolic variables α and β, if **nl-unfold**$^\sharp_{\mathbb{S}}(s^\sharp, \alpha, \beta) = s^{\sharp\prime}$, then $\gamma_{\mathbb{S}}\left(s^\sharp\right) \subseteq \gamma_{\mathbb{S}}\left(s^{\sharp\prime}\right)$.

Proof. Let $(\rho, m) \in \gamma_{\mathbb{S}}(\alpha.\mathbf{p}(S) * m^\sharp, \sigma^\sharp)$. This means that there exists some valuations $\sigma_n, \sigma_m, \sigma_s$ as well as two distinct concrete heaps m' and $m_{\mathbf{p}}$ such that:

- $m = m' \uplus m_{\mathbf{p}}$
- $m, \sigma_n, \sigma_s \vDash_m m^\sharp$
- $m_{\mathbf{p}}, \sigma_n, \sigma_s \vDash_m \alpha.\mathbf{p}(S)$
- $\sigma_n(\beta) \in \sigma_m(\texttt{mset}_S)$

The proof proceeds by induction over the derivation of the satisfiability judgment for the inductive predicate $\alpha.\mathbf{p}(S)$.

Base case The base corresponds to a rule containing no recursive instance of $\mathbf{p}$. Since S is a head parameter and since there is no recursive instance, then it has two possible definitions in this rule. The first one is $S = \texttt{[]}$. It implies that $\sigma_m(\texttt{mset}_S) = \varnothing$. This is inconsistent with the assumption $\sigma_n(\beta) \in \sigma_m(\texttt{mset}_S)$, therefore, this definition is ruled out. The only possible definition remaining is $S = \texttt{[}\alpha\texttt{]}$. It implies that $\sigma_m(\texttt{mset}_S) = \{\!\{ \sigma_n(\alpha) \}\!\}$. Given that $\sigma_n(\beta) \in \sigma_m(\texttt{mset}_S)$, we deduce that $\sigma_n(\alpha) = \sigma_n(\beta)$. Therefore, we let $\sigma'_s := \sigma_s[S^\dagger_l, S^\dagger_r \mapsto \varepsilon; S^\dagger_0 \mapsto \sigma_s(S)]$. From this definition we deduce that:

$$\begin{cases} \varnothing, \sigma'_n, \sigma'_s \vDash_m \alpha.\mathbf{p} \mathrel{-\!\!*}\!\{S_l^\dagger \boxdot S_r^\dagger\}\!\vDash \beta.\mathbf{p} \\ m_{\mathbf{p}}, \sigma'_n, \sigma'_s \vDash_m \beta.\mathbf{p}(, S_0^\dagger) \\ \sigma'_n, \sigma'_s \vDash_s S = S_l^\dagger.S_0^\dagger.S_r^\dagger \\ \sigma_n(\beta) \in \sigma'_m(\mathtt{mset}_{S_0^\dagger}) \end{cases}$$

Recursive case Let us assume that there exists some rule $\exists \vec{\beta}, \vec{S'}, m_k^\sharp * (\circledast_i \beta_i.\mathbf{p}(S_i)) \wedge S = E_k$ The recursive case has two subcases.

The first one occurs when $\sigma_n(\alpha) = \sigma_n(\beta)$. It is handled similarly to the empty case: the segment between α and β is empty.

In the second case $\sigma_n(\alpha) \neq \sigma_n(\beta)$. The satisfiability of the rule implies that there exist values $\vec{c}$ and $\vec{w}$ for existentially quantified variables $\vec{\beta}$ and $\vec{S'}$, updated valuations $(\sigma'_n, \sigma'_m, \sigma'_s)$ as well as disjoint concrete heaps m' and $m_1, \ldots, m_n$ such that:

- $m_{\mathbf{p}} = m' \uplus_i m_i$
- $m', \sigma'_n, \sigma'_s \vDash_m m_k^\sharp$
- $\forall i, m_i, \sigma'_n, \sigma'_s \vDash_m \beta_i.\mathbf{p}(\vec{\delta}_i, S_i)$
- $\sigma'_n, \sigma'_s \vDash_s S = E_k$.

Recall that by definition, a head sequence parameter satisfies the following multiset constraint: $\mathtt{mset}_S \subseteq \{\alpha\} \uplus_i \mathtt{mset}_{S_i}$. Hence, the inequality $\sigma_n(\alpha) \neq \sigma_n(\beta)$, implies that for some index i, $\sigma'_n(\beta) \in \sigma'_m(\mathtt{mset}_{S'_i})$. Therefore, we can apply the induction hypothesis.

This means that there exists variables S'_l, S'_r, and $S_0^\dagger$, disjoint concrete heaps m_s and m_f, as well as updated valuations σ''_n, σ''_m, and σ''_s such that:

- $m_i = m_s \uplus m_f$
- $m_s, \sigma''_n, \sigma''_s \vDash_m \beta_i.\mathbf{p}(\vec{\delta}) \mathrel{-\!\!*}\!\{S'_l \boxdot S'_r\}\!\vDash \beta.\mathbf{p}$
- $m_f, \sigma''_n, \sigma''_s \vDash_m \beta.\mathbf{p}(, S_0^\dagger)$
- $\sigma''_n, \sigma''_m \vDash_s S_i = S'_l.S_0^\dagger.S'_r$
- $\sigma_n(\beta) = \sigma''_m(S_0^\dagger)$

Additionally, recall that the sequence constraint $S = E_k$ can be rewritten as $S = E'_k.S_i.E''_k$, since S is an additive parameter. Now let us define

$$\sigma'''_s := \sigma''_s \left[S_l^\dagger \mapsto \mathbb{E}[\![E'_k.S'_l]\!]_s(\sigma''_n, \sigma''_s), S_r^\dagger \mapsto \mathbb{E}[\![S'_r.E''_k]\!]_s(\sigma''_n, \sigma''_s) \right]$$

This new valuation implies that $\sigma''_n, \sigma'''_s \vDash_s S_l^\dagger \boxdot S_r^\dagger = E'_k.S'_l \boxdot S'_r.S''_k$. Furthermore, by construction of segment predicates, we observe that $m_k^\sharp * (\circledast_{j \neq i} \beta_j.\pi(\vec{\delta}_j, S_j)) * \beta_i.\mathbf{p}(\vec{\delta}_i) \mathrel{-\!\!*}\!\{S'_l \boxdot S'_r\}\!\vDash \beta.\mathbf{p}$ matches some rule of the inductive segment predicate. This means that:

$$m' \uplus_{j \neq i} m_j \uplus m_s, \sigma''_n, \sigma'''_s \vDash_m \alpha.\mathbf{p} \mathrel{-\!\!*}\!\{S_l^\dagger \boxdot S_r^\dagger\}\!\vDash \beta.\mathbf{p}$$

Finally, using the substitution lemma, we prove that:

$$\begin{array}{ll} \sigma_n, \sigma'''_s \vDash_s S = E'_k.S_i.E''_k & \text{Rule constraint} \\ \Rightarrow \sigma_n, \sigma'''_s \vDash_s S = E'_k.S'_l.S_0^\dagger.S'_r.E''_k & \text{By induction hypothesis} \\ \Rightarrow \sigma_n, \sigma'''_s \vDash_s S = S_l^\dagger.S_0^\dagger.S_r^\dagger & \text{By definition of } S_l^\dagger \text{ and } S_r^\dagger \end{array}$$

The end of the proof boils down to apply the soundness of $\mathbf{guard}_{\mathbb{S}}^\sharp$ as well as the definition of $\gamma_{\mathbb{S}}$. □

Similarly, one could expect that if the abstract state contains a segment predicate $\alpha.\mathbf{p} \mathrel{-\!\!*}\!\{S_l \boxdot S_r\}\!\vDash \alpha'.\mathbf{p}$, where the sequence parameters $S_l \boxdot S_r$, corresponding to a head parameter, contain a variable β, then it is safe to cut the segment in two as follows:

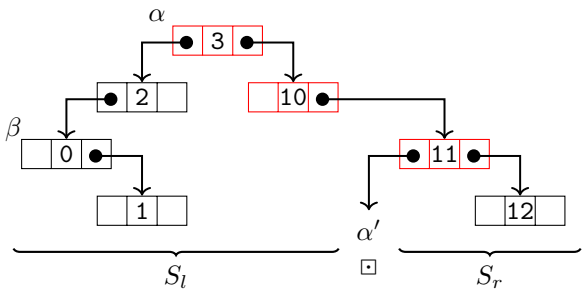


Figure 4.11: Possible concretization of $\alpha.\mathbf{p} \ast\!\!\{S_l \boxdot S_r\}\!\models \alpha'.\mathbf{p}$

$$\alpha.\mathbf{p} \ast\!\!\{S'_l \boxdot S'_r\}\!\models \beta.\mathbf{p} * \beta.\mathbf{p} \ast\!\!\{S''_l \boxdot S''_r\}\!\models \alpha'.\mathbf{p} \wedge S_l = S'_l.S''_l \wedge S_r = S''_r.S'_r$$

However, this is incorrect. To understand why, let us consider the concrete heap displayed in Figure 4.11. This heap presents a binary tree that satisfies all the required hypotheses. Nevertheless, it is impossible to construct a segment predicate from the node at address β to the end of the segment at address α'. The split makes sense if and only if the node denoted by β is a node in the path from α to α', *i.e.* one of the nodes marked in red.

To circumvent this issue, we limit the non-local unfolding to segment predicate where the head sequence parameter is also left or right-only. This implies that the corresponding full predicate has at most one recursive instance in each rule.

Definition 4.13: Non-local unfolding (segment predicate case)

For an inductive predicate $\mathbf{p}(\vec{\kappa}, S)$ such that the sequence parameter S is a head parameter, and for all symbolic variables α and β, the non-local unfolding operator is defined by:

Left case

$$\textbf{nl-unfold}^\sharp_{\mathbb{S}}((\alpha.\mathbf{p}(\vec{\kappa}) \ast\!\!\{S_l \boxdot\}\!\models \alpha'.\mathbf{p}(\vec{\kappa}') * m^\sharp, \sigma^\sharp), \alpha, \beta) := \begin{pmatrix} \alpha.\mathbf{p}(\vec{\kappa}) \ast\!\!\{S_l'^\dagger \boxdot\}\!\models \beta.\mathbf{p}(\vec{\kappa}^\dagger) \\ * \beta.\mathbf{p}(\vec{\kappa}^\dagger) \ast\!\!\{S_l''^\dagger \boxdot\}\!\models \alpha'.\mathbf{p}(\vec{\kappa}'), \textbf{guard}^\sharp_s \begin{pmatrix} \beta \in \mathtt{mset}_{S_l''^\dagger} \\ \wedge\, S_l = S_l'^\dagger.S_l''^\dagger, \sigma^\sharp \end{pmatrix} \\ * m^\sharp \end{pmatrix}$$
$$\text{when } \textbf{sat}^\sharp_s(\beta \in \mathtt{mset}_{S_l}) = \textbf{true}$$

Right case

$$\textbf{nl-unfold}^\sharp_{\mathbb{S}}((\alpha.\mathbf{p}(\vec{\kappa}) \ast\!\!\{\boxdot S_r\}\!\models \alpha'.\mathbf{p}(\vec{\kappa}') * m^\sharp, \sigma^\sharp), \alpha, \beta) := \begin{pmatrix} \alpha.\mathbf{p}(\vec{\kappa}) \ast\!\!\{\boxdot S_r'^\dagger\}\!\models \beta.\mathbf{p}(\vec{\kappa}^\dagger) \\ * \beta.\mathbf{p}(\vec{\kappa}^\dagger) \ast\!\!\{\boxdot S_r''^\dagger\}\!\models \alpha'.\mathbf{p}(\vec{\kappa}'), \textbf{guard}^\sharp_s \begin{pmatrix} \beta \in \mathtt{mset}_{S_r''^\dagger} \\ \wedge\, S_r = S_r''^\dagger.S_r'^\dagger, \sigma^\sharp \end{pmatrix} \\ * m^\sharp \end{pmatrix}$$
$$\text{when } \textbf{sat}^\sharp_s(\beta \in \mathtt{mset}_{S_r}) = \textbf{true}$$

Proof. The proof of the non-local unfolding for segment predicates is similar to the one for full predicates: by induction over the derivation of the satisfiability judgment of the segment. The induction hypothesis is guarded by the condition $\sigma_n(\beta) \in S_l$.

The base case corresponds to the empty segment and is contradictory with the guard of the induction hypothesis.

For the recursive cases, the case $\sigma_n(\alpha) = \sigma_n(\beta)$ is treated as the full predicate case: we introduce an empty segment. In the case $\sigma_n(\alpha) \neq \sigma_n(\beta)$, we leverage the fact that the sequence parameter is left-only or right-only to establish that there is no recursive instance of the full predicate. This implies that $\sigma_n(\beta)$ is a member of the parameter of the recursive segment instance. Therefore, the induction hypothesis applies. From this point, the proof is conducted as in the full predicate case. □

4.3.3 Operators for abstract evaluation

Finally, to take into account the backward and non-local unfolding we extend the definition of the $\textbf{read}^\sharp_{\mathbb{S}}$ and $\textbf{write}^\sharp_{\mathbb{S}}$ operators. Their definitions are presented in Figures 4.12 and 4.13 respec-

tively. Note that, when several cases are possible, the analysis applies the first one. For instance, $\mathbf{read}^\sharp_{\mathbb{S}}(\alpha, 0, (\alpha.\mathbf{p}(\pi) \mathrel{*\!\!\!\multimap} \{S_l \boxdot S_r\} \models \beta.\mathbf{p}(\alpha), \top^\sharp_s))$ will simply trigger a forward unfolding. Similarly, the analysis handles $\mathbf{read}^\sharp_{\mathbb{S}}(\pi, 0, (\alpha.\mathbf{p}(\pi') \mathrel{*\!\!\!\multimap} \{S_l \boxdot S_r\} \models \beta.\mathbf{p}(\pi), S_r = [\pi]))$ with a backward unfolding of the segment predicate and not with a non-local unfolding.

Finally, given that the abstract evaluation operators $\mathbb{E}[\![\bullet]\!]^\sharp_{\mathbb{S}}$ and $\mathbb{L}[\![\bullet]\!]^\sharp_{\mathbb{S}}$ (presented in Figure 2.14), as well as the abstract transfer functions $\mathbf{assign}^\sharp_{\mathbb{S}}$, defined in Figure 2.15b, and $\mathbf{malloc}^\sharp_{\mathbb{S}}$, introduced in Definition 2.15, only depend on the $\mathbf{guard}^\sharp_{\mathbb{S}}$, $\mathbf{read}^\sharp_{\mathbb{S}}$, and $\mathbf{write}^\sharp_{\mathbb{S}}$ and do not explicitly manipulate inductive predicates, their definitions are not modified.

4.4 Lattice operators

This section presents the lattice operators of the combined abstract domain $\mathbb{S}^\sharp$. That is to say the inclusion checking $\sqsubseteq^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \to \{\mathbf{true}, \mathbf{false}\}$, as well as the upper bound operators $\sqcup^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ and $\nabla^\sharp_{\mathbb{S}} : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp$. All these operators follow the same three-step approach. First they operate on the memory parts of their input abstract states. This step has two purposes: it establishes the result (*i.e.* either inclusion or upper bound) concerning the memory parts, and it constructs a mapping that relates symbolic variables between different inputs in order to guide the second step. The second step, called the *instantiation step*, makes the sequence parts of the inputs uniform using the mapping established in the memory step. That is to say, it updates the two sequence abstract values to ensure that they use comparable symbolic variables. Finally, the third step computes the sequence part of the result using the updated sequence abstract states.

4.4.1 Inclusion checking

4.4.1.1 Memory step

The memory step verifies that an abstract heap $m^\sharp_l$ implies another one $m^\sharp_r$ by establishing an *abstract heap entailment judgment* of the form $\sigma^\sharp, C_s, \Phi \vdash m^\sharp_l \sqsubseteq^\sharp_{\mathbb{M}} m^\sharp_r$, where:

- $\sigma^\sharp \in \mathbb{D}^\sharp_s$ is the sequence part of the left abstract state.
- $C_s \in \mathcal{C}^*_s$ is a finite conjunction of sequence constraints.
- $\Phi : \mathcal{V} \rightharpoonup \mathcal{V}$ is a partial mapping from symbolic variables in the right memory abstract state to their equivalent in the left one.
- $m^\sharp_l \in \mathbb{M}^\sharp$ and $m^\sharp_r \in \mathbb{M}^\sharp$ are respectively the left and right abstract heaps.

In essence, this judgment states that memory concrete states in the memory concretization of $m^\sharp_l$ which satisfy both the sequence abstract state $\sigma^\sharp$ and the sequence constraints in C_s are in the concretization of $m^\sharp_r$.

The inference rules are presented in Figure 4.14. For the sake of brevity, we consider inductive predicates with a single sequence parameter and no numerical parameter. Additionally, we do not discuss the construction of the mapping Φ here. We simply mention that this mapping is the identity function for root variables. That is to say the numerical symbolic variables $\underline{\mathtt{x}}$ that denote the addresses of `MemImp` variables, as well as sequence variables S that occur in the pre-condition.

The first two rules (**sep-sep** and **ind-ind**) boil down to matching similar predicate. In the second rule, the two full inductive predicate instances match if and only if their sequence parameters are equal. Consequently, the rule adds the constraint $S_l = S_r$ in C_s in order to be proven latter.

The third rule (**sep**) allows the shape inclusion to leverage the separating conjunction in order to reason locally.

The fourth rule matches an inductive segment in the left memory part with a part of a full inductive instance. In essence, this rule splits the full predicate instance in two at a fresh symbolic variable $\beta^\dagger_r$. This yields a segment and a full predicate. The new segment is matched with the segment in the left abstract memory. Then the inclusion is checked recursively between the remaining of the left abstract memory and the remaining full predicate instance. Finally, this rule asserts that this matching is correct if the sequence parameter S of the full instance is equal to the sequence parameters of the segment instance $S^1_l \boxdot S^2_l$ where the placeholder symbol $\boxdot$ is replaced by the sequence parameter of the new full instance $S^\dagger$.

$$\mathbf{read}^\sharp_{\mathbb{S}} : \mathcal{V} \times \mathbb{V} \times \mathbb{S}^\sharp \to \left(\mathbb{S}^\sharp \times \mathcal{V} \times \mathbb{V}\right) \uplus \{\top\}$$

$$\mathbf{read}^\sharp_{\mathbb{S}}(\alpha, c, (\alpha.\mathtt{f} \mapsto \beta * m^{\sharp\prime}, \sigma^\sharp)) := ((\alpha.\mathtt{f} \mapsto \beta * m^{\sharp\prime}, \sigma^\sharp), \beta, 0)$$

$$\mathbf{read}^\sharp_{\mathbb{S}}(\alpha, c, (\alpha.\mathbf{p}(\vec{\kappa}, \vec{S}) * m^{\sharp\prime}, \sigma^\sharp)) :=$$
$$\quad \mathbf{read}^\sharp_{\mathbb{S}}(\alpha, c, \mathbf{unfold}^\sharp_{\mathbb{S}}((\alpha.\mathbf{p}(\vec{\kappa}, \vec{S}) * m^{\sharp\prime}, \sigma^\sharp), \alpha))$$

$$\mathbf{read}^\sharp_{\mathbb{S}}(\pi', c, (\alpha.\mathbf{p}(\pi) \mathrel{*\!\!\!-\!\!\!\!\{S_l \boxdot S_r\}\!\!=} \alpha'.\mathbf{p}(\pi') * m^{\sharp\prime}, \sigma^\sharp)) :=$$
$$\quad \mathbf{read}^\sharp_{\mathbb{S}}(\pi', c, \mathbf{b\text{-}unfold}^\sharp_{\mathbb{S}}((\alpha.\mathbf{p}(\pi) \mathrel{*\!\!\!-\!\!\!\!\{S_l \boxdot S_r\}\!\!=} \alpha'.\mathbf{p}(\pi') * m^{\sharp\prime}, \sigma^\sharp), \alpha))$$
$$\quad \text{if } \pi \text{ is a backward parameter of } \mathbf{p}$$

$$\mathbf{read}^\sharp_{\mathbb{S}}(\beta, c, (\alpha.\mathbf{p}(\vec{\kappa}, S) * m^{\sharp\prime}, \sigma^\sharp)) :=$$
$$\quad \mathbf{read}^\sharp_{\mathbb{S}}(\beta, c, \mathbf{nl\text{-}unfold}^\sharp_{\mathbb{S}}((\alpha.\mathbf{p}(\vec{\kappa}, S) * m^{\sharp\prime}, \sigma^\sharp), \alpha, \beta))$$
$$\quad \text{if } S \text{ is a head parameter of } \mathbf{p} \text{ and } \mathbf{sat}^\sharp_s(\sigma^\sharp, \beta \in \mathtt{mset}_S) = \mathbf{true}$$

$$\mathbf{read}^\sharp_{\mathbb{S}}(\alpha, c, (m^\sharp, \sigma^\sharp)) := \top \qquad \text{otherwise}$$

Figure 4.12: Definition of $\mathbf{read}^\sharp_{\mathbb{S}}$

$$\mathbf{write}^\sharp_{\mathbb{S}} : \mathcal{V} \times \mathbb{V} \times \mathcal{V} \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp \uplus \{\top\}$$

$$\mathbf{write}^\sharp_{\mathbb{S}}(\alpha, c, \delta, (\alpha.\mathtt{f} \mapsto \beta * m^{\sharp\prime}, \sigma^\sharp)) := (\alpha.\mathtt{f} \mapsto \delta * m^{\sharp\prime}, \sigma^\sharp)$$

$$\mathbf{write}^\sharp_{\mathbb{S}}(\alpha, c, \delta, (\alpha.\mathbf{p}(\vec{\kappa}, \vec{S}) * m^{\sharp\prime}, \sigma^\sharp)) :=$$
$$\quad \mathbf{write}^\sharp_{\mathbb{S}}(\alpha, c, \delta, \mathbf{unfold}^\sharp_{\mathbb{S}}((\alpha.\mathbf{p}(\vec{\kappa}, \vec{S'}) * m^{\sharp\prime}, \sigma^\sharp), \alpha))$$

$$\mathbf{write}^\sharp_{\mathbb{S}}(\pi', c, \delta, (\alpha.\mathbf{p}(\pi) \mathrel{*\!\!\!-\!\!\!\!\{S_l \boxdot S_r\}\!\!=} \alpha'.\mathbf{p}(\pi') * m^{\sharp\prime}, \sigma^\sharp)) :=$$
$$\quad \mathbf{write}^\sharp_{\mathbb{S}}(\pi', c, \delta, \mathbf{b\text{-}unfold}^\sharp_{\mathbb{S}}((\alpha.\mathbf{p}(\pi) \mathrel{*\!\!\!-\!\!\!\!\{S_l \boxdot S_r\}\!\!=} \alpha'.\mathbf{p}(\pi') * m^{\sharp\prime}, \sigma^\sharp), \alpha))$$
$$\quad \text{if } \pi \text{ is a backward parameter of } \mathbf{p}$$

$$\mathbf{write}^\sharp_{\mathbb{S}}(\beta, c, \delta, (\alpha.\mathbf{p}(\vec{\kappa}, S) * m^{\sharp\prime}, \sigma^\sharp)) :=$$
$$\quad \mathbf{write}^\sharp_{\mathbb{S}}(\beta, c, \delta, \mathbf{nl\text{-}unfold}^\sharp_{\mathbb{S}}((\alpha.\mathbf{p}(\vec{\kappa}, S) * m^{\sharp\prime}, \sigma^\sharp), \alpha, \delta))$$
$$\quad \text{if } S \text{ is a head parameter of } \mathbf{p} \text{ and } \mathbf{sat}^\sharp_s(\sigma^\sharp, \beta \in \mathtt{mset}_S) = \mathbf{true}$$

$$\mathbf{write}^\sharp_{\mathbb{S}}(\alpha, c, \delta, (m^\sharp, \sigma^\sharp)) := \top \qquad \text{otherwise}$$

Figure 4.13: Definition of $\mathbf{write}^\sharp_{\mathbb{S}}$

$$\frac{\Phi(\alpha_r) = \alpha_l \qquad \Phi(\beta_r) = \beta_l}{\sigma^\sharp, C_s, \Phi \vdash \alpha_l.\mathtt{f} \mapsto \beta_l \sqsubseteq^\sharp_{\mathbb{M}} \alpha_r.\mathtt{f} \mapsto \beta_r}(\textbf{pt-pt})$$

$$\frac{\Phi(\alpha_r) = \alpha_l}{\sigma^\sharp, S_l = S_r, \Phi \vdash \alpha_l.\mathbf{p}(S_l) \sqsubseteq^\sharp_{\mathbb{M}} \alpha_r.\mathbf{p}(S_r)}(\textbf{ind-ind})$$

$$\frac{\sigma^\sharp, C_s, \Phi \vdash m_l^\sharp \sqsubseteq^\sharp_{\mathbb{M}} m_r^\sharp \qquad \sigma^\sharp, C'_s, \Phi \vdash m_l^{\sharp\prime} \sqsubseteq^\sharp_{\mathbb{M}} m_r^{\sharp\prime}}{\sigma^\sharp, C_s \wedge \mathcal{C}'_s, \Phi \vdash m_l^\sharp * m_l^{\sharp\prime} \sqsubseteq^\sharp_{\mathbb{M}} m_r^\sharp * m_r^{\sharp\prime}}(\textbf{sep})$$

$$\frac{\Phi(\alpha_r) = \alpha_l \qquad \Phi' := \Phi \uplus \{\beta_r^\dagger \mapsto \beta_l\} \qquad \sigma^\sharp, C_s, \Phi' \vdash m_l^\sharp \sqsubseteq^\sharp_{\mathbb{M}} \beta_r^\dagger.\mathbf{p}(S^\dagger)}{\sigma^\sharp, C_s \wedge S = S_l^1.S^\dagger.S_l^2, \Phi \vdash \alpha_l.\mathbf{p} \mathrel{*\!\!-\!\!\!\boxminus} \{S_l^1 \boxdot S_l^2\} \beta_l.\mathbf{p} * m_l^\sharp \sqsubseteq^\sharp_{\mathbb{M}} \alpha_r.\mathbf{p}(S)}(\textbf{seg-ind})$$

$$\frac{\Phi(\alpha_r) = \alpha_l \qquad \Phi' := \Phi \uplus \{\beta_r^\dagger \mapsto \beta_l\} \qquad C'_s := C_s \wedge S_r^1 = S_l^1.S^{1\dagger} \wedge S_r^2 = S^{2\dagger}.S_l^2 \qquad \sigma^\sharp, C_s, \Phi' \vdash m_l^\sharp \sqsubseteq^\sharp_{\mathbb{M}} \beta_r^\dagger.\mathbf{p} \mathrel{*\!\!-\!\!\!\boxminus} \{S^{1\dagger} \boxdot S^{2\dagger}\} \delta_r.\mathbf{p}}{\sigma^\sharp, C'_s, \Phi \vdash \alpha_l.\mathbf{p} \mathrel{*\!\!-\!\!\!\boxminus} \{S_l^1 \boxdot S_l^2\} \beta_l.\mathbf{p} * m_l^\sharp \sqsubseteq^\sharp_{\mathbb{M}} \alpha_r.\mathbf{p} \mathrel{*\!\!-\!\!\!\boxminus} \{S_r^1 \boxdot S_r^2\} \delta_r.\mathbf{p}}(\textbf{seg-seg})$$

$$\frac{\alpha.\mathbf{p}(S) := \bigvee \left(m_r^{\sharp\prime} \wedge \varphi_\mathcal{V} \wedge \varphi_s\right) \qquad \mathbf{sat}^\sharp_s(\sigma^\sharp, \Phi(\varphi_\mathcal{V})) = \mathbf{true} \qquad \sigma^\sharp, C_s, \Phi \vdash m_l^\sharp \sqsubseteq^\sharp_{\mathbb{M}} m_r^\sharp}{\sigma^\sharp, C_s \wedge \varphi_s, \Phi \vdash m_l^\sharp \sqsubseteq^\sharp_{\mathbb{M}} \alpha.\mathbf{p}(S)}(\textbf{fold})$$

Figure 4.14: Inference rules for memory inclusion checking

The fifth rule (**seg-seg**) works in a similar manner when the right abstract memory contains a segment instead of a full predicate instance.

The final rule (**fold**) corresponds to the folding principle. If the right abstract memory contains an inductive predicate, then the inclusion algorithm checks whether the left abstract memory matches one of its rules. Contrary to $\mathbf{unfold}^\sharp_\mathbb{S}$, the inclusion checking does not assert the constraints of the rule. It checks whether the numerical constraints translated according to the mapping Φ hold in the left abstract value $\sigma^\sharp$. The sequence constraints are added to the conjunction C_s in order to be checked later.

Lemma 4.8: Soundness of abstract heap inclusion checking

If $\sigma^\sharp, C_s, \Phi \vdash m_l^\sharp \sqsubseteq^\sharp_{\mathbb{M}} m_r^\sharp$, then

$$\forall((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_m(m_l^\sharp), \begin{cases} \sigma_n, \sigma_s \vDash_s C_s \\ \sigma_n, \sigma_m, \sigma_s \in \gamma_s(\sigma^\sharp) \end{cases} \Rightarrow ((\rho, m), (\sigma_n \circ \Phi, \sigma_m \circ \Phi, \sigma_s \circ \Phi)) \in \gamma_m(m_r^\sharp)$$

Proof. The proof is constructed by induction over the rules used to derive the heap entailment judgment.

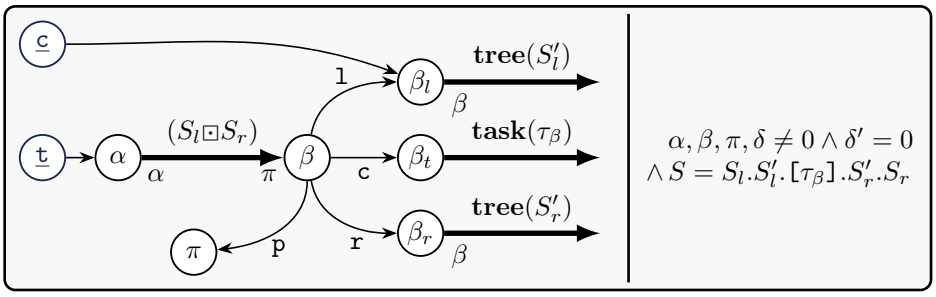
For rules **pt-pt**, **ind-ind** and **sep**, this is straightforward.

The correctness of rule **seg-ind** is a consequence of the segment-full concatenation lemma (Lemma 4.2). Similarly, the soundness of rule **seg-seg** is a consequence of the segment-segment lemma (Lemma 4.3).

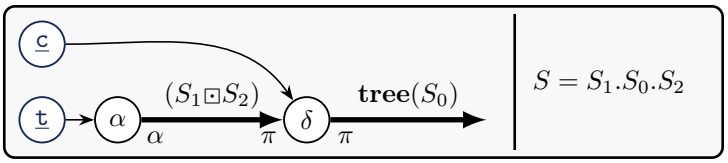
Finally, the rule **fold** is a consequence of the definition of the satisfiability judgment of an inductive predicate instance. □

Example 4.15: Inclusion checking between abstract memories

To illustrate the inclusion checking of the combined abstract domain $\mathbb{S}^\sharp$, let us consider the two abstract states presented in Figure 4.15 The derivation of the abstract heap inclusion judgment



(a) Left input



(b) Right input

Figure 4.15: Inputs of the inclusion checking example

is depicted in Figure 4.17. For each intermediate inference rule we display only the relevant bindings of Φ as well as the sequence constraints added by the rule. The inclusion checking starts by matching the points-to predicates from t and x. Then, the operator proceeds by matching the full predicate instances pointed by c. This adds the constraint $S_0 = S'_l$.

Next, the abstract heap inclusion checking establishes that the remaining of the left memory part is included in the segment between α and δ. The derivation of this judgment is presented in Figure 4.16. It starts by matching the segment from α to β in the left input with a part of the right one. This generates the constraints $S_1 = S_l.S'_1$ and $S_2 = S'_2.S_r$ where $S'_1 \boxdot S'_2$ are the parameters of the new segment in the right abstract memory. After this, the remaining segment in the right memory is unfolded. This unfolding generates the constraint $\Phi(\beta^\dagger) \neq 0$ that is checked in the sequence part of the left input since it is equivalent to $\beta \neq 0$. It also adds the sequence definition constraints $S'_1 = S''_1$ and $S'_2 = S''_2.[\tau^\dagger].S''_r$. Next, the various predicates are matched together. For the sake of concision, we omit the rules matching points-to predicates. The interesting step concerns the derivation of the inclusion between the empty memory predicate and the segment. This derivation boils down to unfold the segment using the empty rule. The numerical constraints generated by this rule are trivially checked since $\Phi(\delta) = \Phi(\beta_r^\dagger)$ and $\Phi(\pi) = \Phi(\beta_p^\dagger)$. The empty segment rule introduces the sequence constraints $S''_1 = []$ and $S''_2 = []$.

To conclude, the conjunction of sequence constraints to be verified in order to establish the inclusion is:

$$C_s = \begin{array}{l} S''_1 = [] \ \wedge S_0 = S'_l \\ \wedge S''_2 = [] \ \wedge S'_2 = S''_2.[\tau^\dagger].S''_r \\ \wedge S''_r = S'_r \ \wedge S_1 = S_l.S'_1 \\ \wedge S'_1 = S''_1 \ \wedge S_2 = S'_2.S_r \end{array}$$

4.4.1.2 Instantiation step

After establishing the inclusion between the shape parts of its inputs, the inclusion checking proceeds with the instantiation step. This step starts by handling the accumulated constraints in C_s. To do so, the inclusion operator rewrites the accumulated constraints using the Φ mapping. However, Φ does not map sequence variables but only numerical ones. This means that some sequence variables from the right input have no counter-part in the left one. For instance, in the sequence constraints accumulator C_s from Example 4.15, only the sequence variable S occurs in the left abstract state. All other sequence variables are not in the support of the left input.

To address this issue, the unknown sequence variables are instantiated. For each sequence variable S', the inclusion checking picks one sequence constraint that corresponds to some definition of S', and

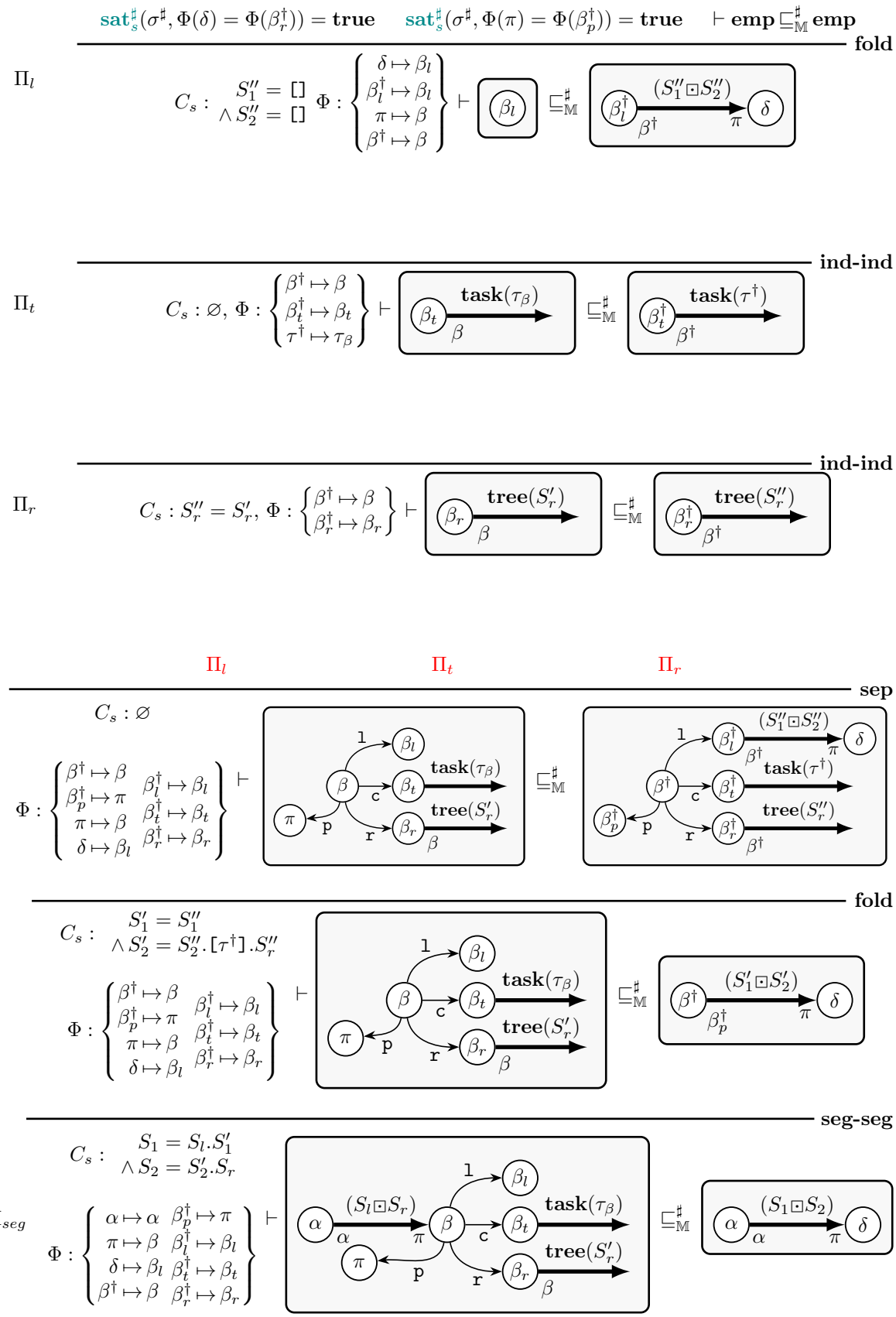


Figure 4.16: Example of abstract heap inclusion checking (continued)

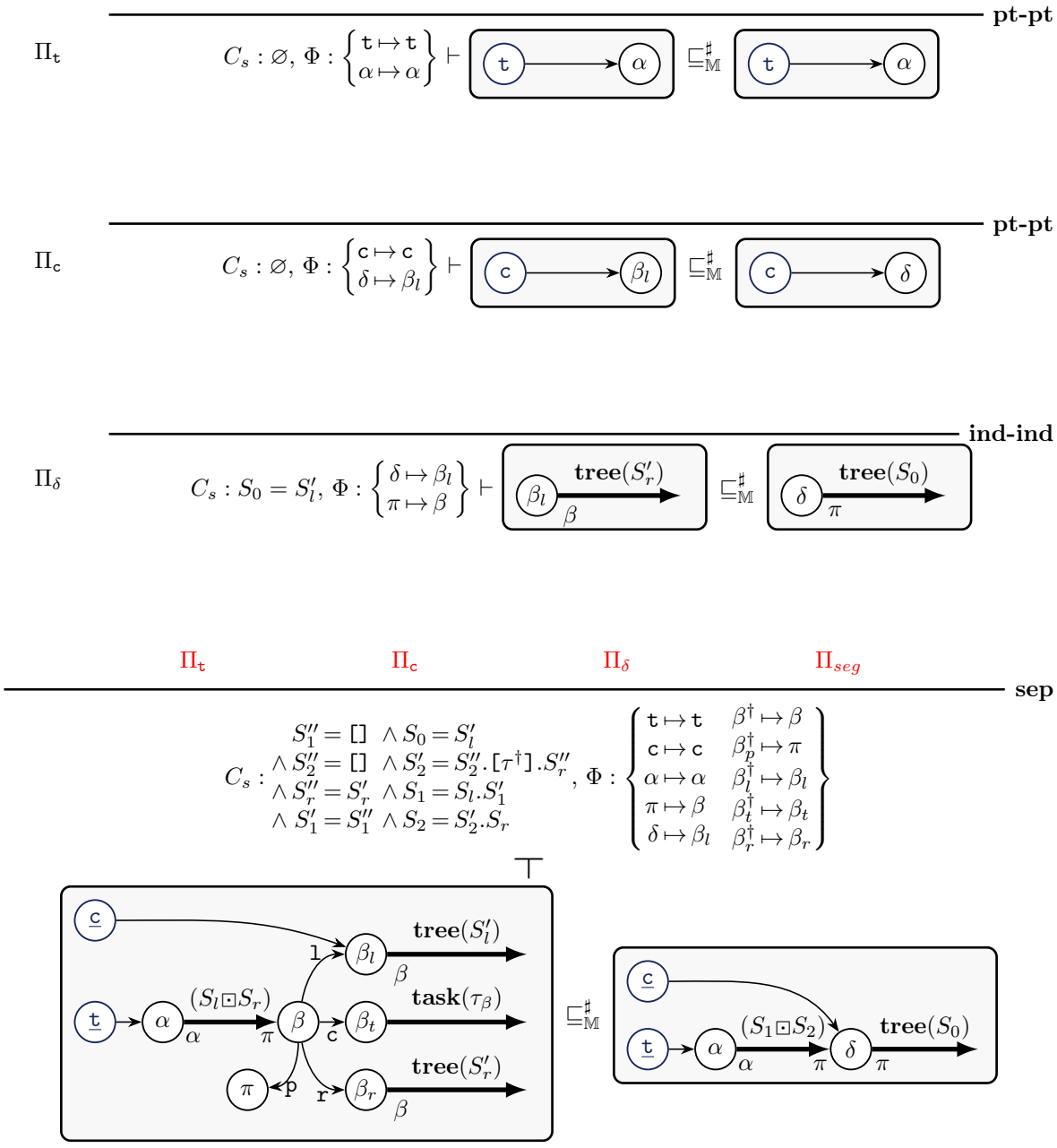


Figure 4.17: Example of abstract heap inclusion checking

Algorithm 1 Definition of $\mathbf{instantiate}^\sharp_{\mathbb{S}}(\sigma^\sharp, \Phi, C_s)$

while there exists $S = E$ in C_s such that $\mathbf{supp}^\sharp_s(E) \subseteq \mathbf{supp}(\Phi)$ and $S \notin \mathbf{supp}^\sharp_s(\sigma^\sharp) \cup E$ **do**
  remove $S = E$ from C_s
  $\sigma^\sharp := \mathbf{guard}^\sharp_s(\sigma^\sharp, S = \Phi(E))$
  $\Phi := \Phi \uplus \{S \mapsto S\}$
if there exists $S = E \in C_s$ such that $\mathbf{supp}^\sharp_s(E) \cup \{S\} \not\subseteq \mathbf{supp}(\Phi)$ **then**
  Fail
Return $(\sigma^\sharp, \Phi, C_s)$

guards this constraint in the sequence part of the left abstract state. That is to say, this constraint must be a definition constraint where S' appears on the left-hand side of the equality and not on the right-hand side. This instantiation step updates the mapping Φ such that all sequence variables in constraints in C_s are now mapped to some variable in the instantiated abstract value.

Algorithm 1 presents the definition of the $\mathbf{instantiate}^\sharp_{\mathbb{S}}$ operator. It inputs the left abstract state $\sigma^\sharp$, the variable mapping Φ, and the conjunction of sequence constraints C_s. This operator repeatedly attempts to instantiate unknown sequence variables using a suitable definition in C_s. A definition is suitable if it can be translated using the Φ mapping and if it does not contain S. In that case, the definition is guarded in the sequence abstract state, and the mapping is updated. If after these attempts, some sequence variable in C_s remains not translatable by the mapping Φ, then the instantiation operator fails. Otherwise, it returns the updated abstract sequence value, mapping, and constraint accumulator.

The sequence instantiation lemma (Lemma 4.7) ensures that this instantiation step is sound when it does not fail.

Lemma 4.9: Soundness of $\mathbf{instantiate}^\sharp_{\mathbb{S}}$

For any abstract state $(m^\sharp, \sigma^\sharp) \in \mathbb{S}^\sharp$, conjunction of constraints C_s and variable mapping Φ, if $\mathbf{instantiate}^\sharp_{\mathbb{S}}(\sigma^\sharp, \Phi, C_s) = (\sigma^{\sharp\prime}, \Phi', C'_s)$, then $\gamma_{\mathbb{S}}(m^\sharp, \sigma^\sharp) \subseteq \gamma_{\mathbb{S}}(m^\sharp, \sigma^{\sharp\prime})$.

4.4.1.3 Sequence step

After this instantiation step, the inclusion checking proceeds with the sequence step. First its checks that the remaining constraints are valid in the instantiated left sequence abstract state using the $\mathbf{sat}^\sharp_s$ operator. Finally, the right sequence part is renamed according to the mapping Φ and the inclusion checking is performed between the instantiated sequence part of the left input and the renamed sequence part of the right one.

Example 4.16: Inclusion checking (sequence step)

To illustrate the instantiation step, let us consider the inclusion example from Example 4.15. In the inferred sequence constraints accumulator C_s, sequence variables S''_1, S''_2, S''_r, S'_1, S'_2, S_1, S_2, and S_0 are not present in the left input. So the inclusion operator instantiates them. This means that the definition constraints of these variables in C_s are guarded. This yields the abstract state presented below. For the sake of brevity we focus solely on sequence constraints.

$$\sigma^{\sharp\prime}_l := \left(\mathfrak{E} = \{S'_1, S''_1, S''_2\}; \mathfrak{R} = \left\{ \begin{array}{l} S'_r \sim S''_r \\ S_l \sim S_1 \\ S'_l \sim S_0 \\ S'_1 \sim S''_1 \sim S''_2 \end{array} \right\}; \mathfrak{D} = \left\{ \begin{array}{r} S \mapsto S_l.S'_l.S_2 \\ S_2 \mapsto S'_2.S_r \\ S'_2 \mapsto [\tau].S'_r \end{array} \right\} \right)$$

The $\mathbf{guard}^\sharp_s$ operator compacts the definition of S when the new definition of S'_2 is assumed. Likewise, inserting the definition of S_2 compacts further the definition of S.

Finally, given that there is no constraint left in C_s after the instantiation, the sequence inclusion checking $\sigma^{\sharp\prime}_l \sqsubseteq^\sharp_s \sigma^\sharp_r$ is performed. Verifying the only definition in $\sigma^\sharp_r$, $S = S_1.S_0.S_2$, boils down to rename the sequence variables by their class representative according to $\mathfrak{R}$. This yields the definition of S in $\sigma^{\sharp\prime}_r$.

Consequently, the combined abstract domain inclusion checking operator concludes that the inclusion holds.

$$\sigma_l^\sharp, m_l^\sharp) \sqsubseteq_{\mathbb{S}}^\sharp (\sigma_r^\sharp, m_r^\sharp) := \begin{cases} \textbf{true} & \text{if } \begin{array}{l} \sigma_l^\sharp, C_s, \Phi \vdash m_l^\sharp \sqsubseteq_{\mathbb{M}}^\sharp m_r^\sharp \\ \wedge\, \textbf{sat}_s^\sharp(\sigma_l^{\sharp\prime}, \Phi'(C_s')) = \textbf{true} \\ \wedge\, \sigma_l^{\sharp\prime} \sqsubseteq_s^\sharp \Phi'(\sigma_r^\sharp) = \textbf{true} \\ \text{where } \sigma_l^{\sharp\prime}, \Phi', C_s' := \textbf{instantiate}_{\mathbb{S}}^\sharp(\sigma_l^\sharp, \Phi, C_s) \end{array} \\ \textbf{false} & \text{otherwise} \end{cases}$$

Figure 4.18: Abstract inclusion checking

To sum up, the definition of the abstract inclusion checking operator in the combined abstract domain is presented in Figure 4.18. The resulting inclusion checking is sound.

Theorem 4.6: Soundness of $\sqsubseteq_{\mathbb{S}}^\sharp$

For any abstract states $s_l^\sharp$ and $s_r^\sharp$, if $s_l^\sharp \sqsubseteq_{\mathbb{S}}^\sharp s_r^\sharp = \textbf{true}$, then $\gamma_{\mathbb{S}}(s_l^\sharp) \subseteq \gamma_{\mathbb{S}}(s_r^\sharp)$.

Proof. Let us consider two non-extreme abstract states $(m_l^\sharp, \sigma_l^\sharp) \in \mathbb{S}^\sharp$ and $(m_r^\sharp, \sigma_r^\sharp) \in \mathbb{S}^\sharp$, such that $(m_l^\sharp, \sigma_l^\sharp) \sqsubseteq_{\mathbb{S}}^\sharp (m_r^\sharp, \sigma_r^\sharp) = \textbf{true}$. This means that the inclusion checking operator successfully derived the abstract entailment judgment: $\sigma_l^\sharp, C_s, \Phi \vdash m_l^\sharp \sqsubseteq_{ms}^\sharp m_r^\sharp$.

According to Lemma 4.9, it is sufficient to prove that $\gamma_{\mathbb{S}}(m_l^\sharp, \sigma_l^{\sharp\prime}) \subseteq \gamma_{\mathbb{S}}(m_r^\sharp, \sigma_r^\sharp)$, where $\sigma_l^{\sharp\prime}$ is the instantiated counterpart of $\sigma_l^\sharp$. Let us consider $(\rho, m) \in \gamma_{\mathbb{S}}(m_l^\sharp, \sigma_l^{\sharp\prime})$. This means that there exists a triple of valuations $(\sigma_n, \sigma_m, \sigma_s)$, such that:

- $(\sigma_n, \sigma_m, \sigma_s) \in \gamma_s(\sigma_l^{\sharp\prime})$
- $((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_m(m_l^\sharp)$

Additionally, by definition of the $\textbf{instantiate}_{\mathbb{S}}^\sharp$ operator and since $\textbf{sat}_s^\sharp(\sigma_l^{\sharp\prime}, C_s') = \textbf{true}$, we know that $(\sigma_n, \sigma_s) \vDash_s C_s$. Let us define the triple $(\sigma_n', \sigma_m', \sigma_s') := (\sigma_n \circ \Phi, \sigma_m \circ \Phi, \sigma_s \circ \Phi)$. By definition of the abstract heap inclusion judgment, we infer that $((\rho, m), (\sigma_n', \sigma_m', \sigma_s')) \in \gamma_m(m_r^\sharp)$.

Finally, given that $\sigma_l^{\sharp\prime} \sqsubseteq_s^\sharp \Phi(\sigma_r^\sharp) = \textbf{true}$, we conclude by soundness of the sequence domain inclusion operator $\sqsubseteq_s^\sharp$ that $(\sigma_n', \sigma_m', \sigma_s') \in \gamma_s(\sigma_r^\sharp)$. This proves that $(\rho, m) \in \gamma_{\mathbb{S}}(m_r^\sharp, \sigma_r^\sharp)$. □

4.4.2 Upper bounds

We now present the upper bounds operators $\sqcup_{\mathbb{S}}^\sharp : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ and $\nabla_{\mathbb{S}}^\sharp : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \to \mathbb{S}^\sharp$. Since these operators are similar, we present the join operator first. Then, we outline the differences between the join and widening operators.

4.4.2.1 Memory step

The upper bound operators compute the memory parts of their outcome by establishing an *upper bound equality judgment* of the form $\sigma_l^\sharp, C_l, \sigma_r^\sharp, C_r \vdash_\Psi m_l^\sharp \sqcup_{\mathbb{M}}^\sharp m_r^\sharp = m_o^\sharp$ where:

- $\sigma_l^\sharp \in \mathbb{D}_s^\sharp$ and $\sigma_r^\sharp \in \mathbb{D}_s^\sharp$ are respectively the left and right sequence parts of the inputs.
- C_l and C_r are conjunctions of sequence constraints that are assumed to be satisfied in respectively the left and the right abstract state,
- $\Psi : \mathcal{V} \rightharpoonup \mathcal{V}^2$ is a partial mapping associating to each symbolic variable in the input abstract state its corresponding one in the left and right input states.
- $m_l^\sharp \in \mathbb{M}^\sharp$ and $m_r^\sharp \in \mathbb{M}^\sharp$ are respectively the left and right abstract memory heaps.
- $m_o^\sharp \in \mathbb{M}^\sharp$ is the result abstract memory heap.

The inference rules used to establish these judgments are presented in Figure 4.19. Once again, we focus solely on sequence parameters, and we omit details regarding numerical parameters of inductive predicates.

Rules **pt-pt** and **ind-ind** state that the upper bound of similar separation logic predicates is an instance of this predicate. In rule **ind-ind**, the upper bound operator ensures that the upper bound is a sound one as long as the sequence parameter in the outcome is equal to the parameters in the inputs.

Rule **ind-weak** attempts to weaken the right abstract memory by folding it into a full inductive predicate instance. To do so, the upper bound operators rely on the shape part of the inclusion checking operator in order to check that the memory entails a predicate instance. If the inclusion holds, then the sequence constraints generated by it are accumulated in the upper bound equality judgment.

Similarly, rule **seg-weak** weakens an abstract memory into a segment by performing an inclusion checking.

The **seg-intro** rule matches an empty memory with a memory that can be folded into a segment. This rule ensures that the symbolic variables corresponding to the extremities of the segment are equal in the left input using the $\mathbf{sat}_s^\sharp$ operator.

Note that these last three rules have a symmetric counterpart. For instance, there exists a rule **weak-ind** where the memory in the left input is matched with a predicate in the right input to obtain a predicate in the outcome.

Finally, the **sep** rule allows the upper bound operator to reason locally. In essence, it states that we can compose the upper bound using the separating conjunction. In this case, the sequence constraints are formed by the conjunction of the constraint derived by each premise.

Lemma 4.10: Soundness of abstract heap upper bound

If $\sigma_l^\sharp, C_l, \sigma_r^\sharp, C_r \vdash_\Psi m_l^\sharp \sqcup_{\mathbb{M}}^\sharp m_r^\sharp = m_o^\sharp$, then:

- $\forall((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_m(m_l^\sharp), \begin{cases} \sigma_n, \sigma_s \vDash_s C_l \\ \sigma_n, \sigma_m, \sigma_s \in \gamma_s(\sigma_l^\sharp) \end{cases}$
  $\implies ((\rho, m), (\sigma_n \circ \mathbf{fst} \circ \Psi, \sigma_m \circ \mathbf{fst} \circ \Psi, \sigma_s \circ \mathbf{fst} \circ \Psi)) \in \gamma_m(m_o^\sharp)$
- $\forall((\rho, m), (\sigma_n, \sigma_m, \sigma_s)) \in \gamma_m(m_r^\sharp), \begin{cases} \sigma_n, \sigma_s \vDash_s C_r \\ \sigma_n, \sigma_m, \sigma_s \in \gamma_s(\sigma_r^\sharp) \end{cases}$
  $\implies ((\rho, m), (\sigma_n \circ \mathbf{snd} \circ \Psi, \sigma_m \circ \mathbf{snd} \circ \Psi, \sigma_s \circ \mathbf{snd} \circ \Psi)) \in \gamma_m(m_o^\sharp)$

Proof. For rules **pt-pt**, **ind-ind**, and **sep**, the proof is straightforward. For weakening rules (*i.e.* **ind-weak**, **seg-weak**, and **seg-intro**), the validity of the rule is a consequence of the validity of abstract heap inclusion rules. □

Example 4.17: Abstract join (memory step)

To illustrate the join operator, let us consider the two abstract states in Figure 4.20. The derivation of the memory part of the result is presented in Figure 4.21. For the sake of brevity, we do not display the judgments matching points-to predicates.

The full predicate instances from the nodes pointed by c are matched together by the rule (**ind-ind**) to form another instance $\beta.\mathbf{tree}(S_0)$. This matching inserts the constraints $S_0 = S$ in C_l and $S_0 = S_l$ in C_r.

The part of the tree between the pointers t and c is matched by a segment predicate using the **seg-intro** rule. In the left abstract memory, the segment is empty. Consequently, the constraints $S_1 = []$ and $S_2 = []$ are inserted in C_l. Note that the verification of the numerical constraint asserting that the extremities of the segment are equal is verified since $\mathbf{fst} \circ \Psi(\alpha) = \mathbf{snd} \circ \Psi(\alpha) = \alpha$. Then, the join operator performs an abstract memory inclusion test between the remaining of the right abstract memory and the segment predicate. This inclusion test succeeds and generates the sequence constraints $S_1 = []$, $S_2 = [\beta_t].S_r'$, and $S_r' = S_r$.

4.4.2.2 Instantiation step

After computing the memory part $m_o^\sharp$ of its outcome, the join operator proceeds similarly to the inclusion checking operator. It uses the conjunction of sequence constraints C_l and C_r to instantiate

$$\frac{\Psi(\alpha_o) = (\alpha_l, \alpha_r) \qquad \Psi(\beta_o) = (\beta_l, \beta_r)}{\sigma_l^\sharp, \top, \sigma_r^\sharp, \top \vdash_\Psi \alpha_l.\mathtt{f} \mapsto \beta_l \sqcup_\mathbb{M}^\sharp \alpha_r.\mathtt{f} \mapsto \beta_r = \alpha_o.\mathtt{f} \mapsto \beta_o}(\textbf{pt-pt})$$

$$\frac{\Psi(\alpha_o) = (\alpha_l, \alpha_r)}{\sigma_l^\sharp, S_o = S_l, \sigma_r^\sharp, S_o = S_r \vdash_\Psi \alpha_l.\mathbf{p}(S_l) \sqcup_\mathbb{M}^\sharp \alpha_r.\mathbf{p}(S_r) = \alpha_o.\mathbf{p}(S_o)}(\textbf{ind-ind})$$

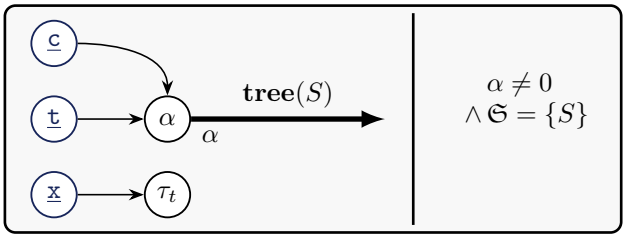
$$\frac{\Psi(\alpha_o) = (\alpha_l, \alpha_r) \qquad \sigma_r^\sharp, C_s, \mathbf{snd} \circ \Psi \vdash m_r^\sharp \sqsubseteq_\mathbb{M}^\sharp \alpha_o.\mathbf{p}.(S_o^\dagger)}{\sigma_l^\sharp, S_o^\dagger = S_l, \sigma_r^\sharp, C_s \vdash_\Psi \alpha_l.\mathbf{p}(S_l) \sqcup_\mathbb{M}^\sharp m_r^\sharp = \alpha_o.\mathbf{p}(S_o)}(\textbf{ind-weak})$$

$$\frac{\begin{array}{c}\Psi(\alpha_o) = (\alpha_l, \alpha_r) \\ \Psi(\beta_o) = (\beta_l, \beta_r)\end{array} \qquad \sigma_r^\sharp, C_s, \mathbf{snd} \circ \Psi \vdash m_r^\sharp \sqsubseteq_\mathbb{M}^\sharp \alpha_o.\mathbf{p} \mathrel{-\!\!*}\{S_o^1 \boxdot S_o^2\}\models \beta_o.\mathbf{p}}{\sigma_l^\sharp, S_o^1 = S_r^1 \wedge S_o^2 = S_r^2, \sigma_r^\sharp, C_s, \vdash_\Psi \alpha_l.\mathbf{p} \mathrel{-\!\!*}\{S_l^1 \boxdot S_l^2\}\models \beta_l.\mathbf{p} \sqcup_\mathbb{M}^\sharp m_r^\sharp = \alpha_o.\mathbf{p} \mathrel{-\!\!*}\{S_o^1 \boxdot S_o^2\}\models \beta_o.\mathbf{p}}(\textbf{seg-weak})$$

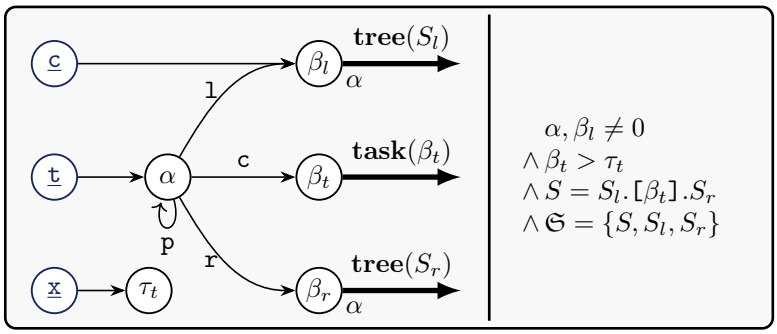
$$\frac{\begin{array}{c}\sigma_r^\sharp, C_s, \mathbf{snd} \circ \Psi \vdash m_r^\sharp \sqsubseteq_\mathbb{M}^\sharp \alpha_o.\mathbf{p} \mathrel{-\!\!*}\{S_o^1 \boxdot S_o^2\}\models \beta_o.\mathbf{p} \\ \Psi(\alpha_o) = (\alpha_l, \alpha_r) \qquad \Psi(\beta_o) = (\beta_l, \beta_r) \qquad \mathbf{sat}_s^\sharp(\sigma_l^\sharp, \alpha_l = \beta_l) = \mathbf{true}\end{array}}{\sigma_l^\sharp, S_o^1 = [] \wedge S_o^2 = [], \sigma_r^\sharp, C_r \vdash_\Psi \mathbf{emp} \sqcup_\mathbb{M}^\sharp m_r^\sharp = \alpha_o.\mathbf{p} \mathrel{-\!\!*}\{S_o^1 \boxdot S_o^2\}\models \beta_o.\mathbf{p}}(\textbf{seg-intro})$$

$$\frac{\sigma_l^\sharp, C_l, \sigma_r^\sharp, C_r \vdash_\Psi m_l^\sharp \sqcup_\mathbb{M}^\sharp m_r^\sharp = m_o^\sharp \qquad \sigma_l^\sharp, C_l', \sigma_r^\sharp, C_r' \vdash_\Psi m_l^{\sharp\prime} \sqcup_\mathbb{M}^\sharp m_r^{\sharp\prime} = m_o^{\sharp\prime}}{\sigma_l^\sharp, C_l \wedge C_l', \sigma_r^\sharp, C_r \wedge C_r' \vdash_\Psi m_l^\sharp * m_l^{\sharp\prime} \sqcup_\mathbb{M}^\sharp m_r^\sharp * m_r^{\sharp\prime} = m_o^\sharp * m_o^{\sharp\prime}}(\textbf{sep})$$

Figure 4.19: Inference rules for memory join



(a) Left input



(b) Right input

Figure 4.20: Inputs of the join example

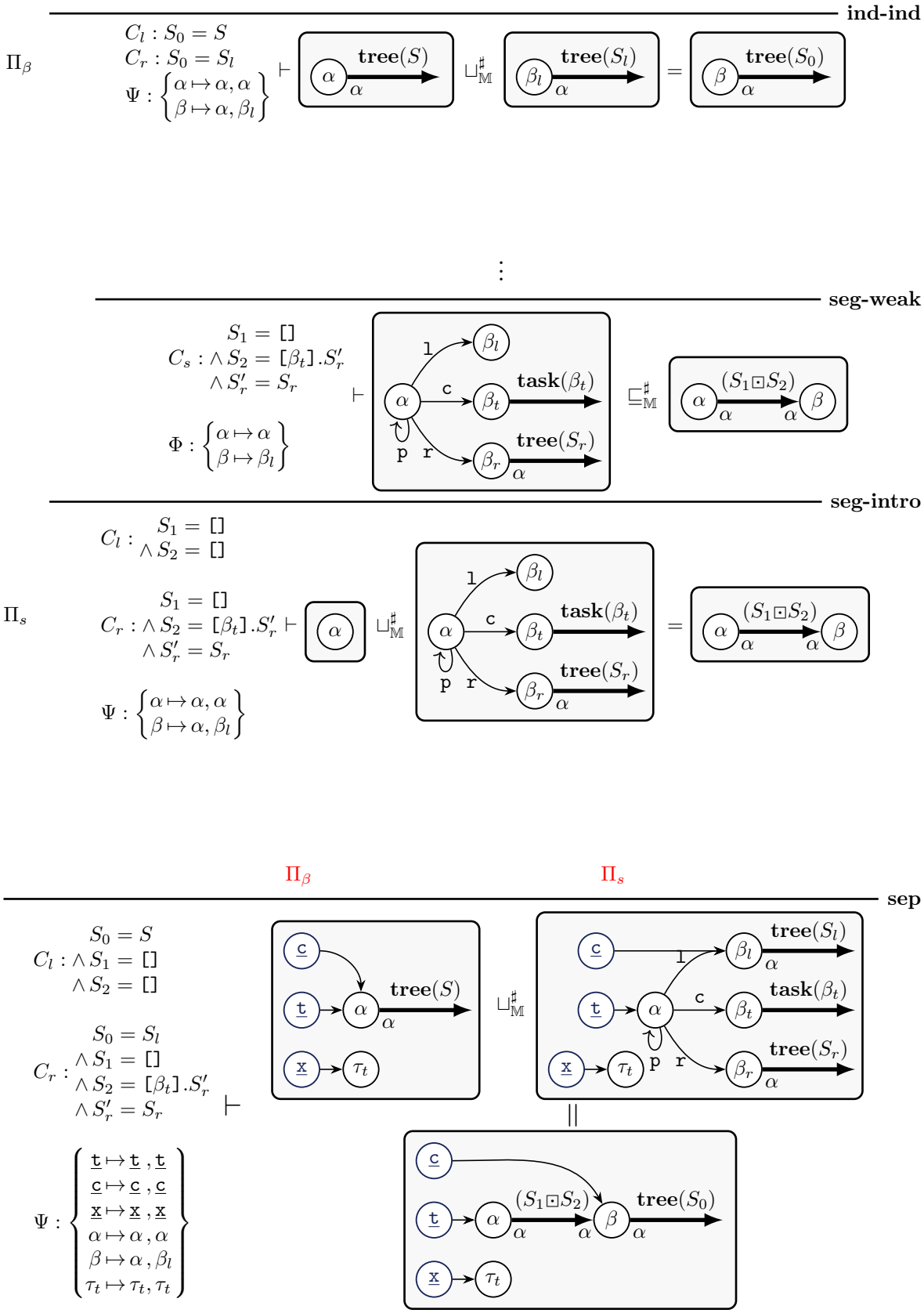


Figure 4.21: Example of abstract heap upper bound computation

sequence variables that occur in $m_o^\sharp$ but not in the left or right input, thanks to the $\mathbf{instantiate}_{\mathbb{S}}^\sharp$ operator. This instantiation also completes the mapping from the output abstract memory value $m_o^\sharp$ and the inputs, *i.e.* $\mathbf{fst} \circ \Psi$ and $\mathbf{snd} \circ \Psi$. This yields the following instantiated sequence abstract values, symbolic variables mappings, and translated conjunction of constraints:

- $(\sigma_l^{\sharp\prime}, \Phi_l, C_l') := \mathbf{instantiate}_{\mathbb{S}}^\sharp(\sigma_l^\sharp, \mathbf{fst} \circ \Psi, C_l)$
- $(\sigma_r^{\sharp\prime}, \Phi_r, C_r') := \mathbf{instantiate}_{\mathbb{S}}^\sharp(\sigma_r^\sharp, \mathbf{snd} \circ \Psi, C_r)$

4.4.2.3 Sequence step

If after the instantiation, the conjunctions of constraints, C_l' and C_r' are not empty, then the remaining constraints are checked in the instantiated sequence abstract values, $\sigma_l^{\sharp\prime}$ and $\sigma_r^{\sharp\prime}$, using the $\mathbf{sat}_s^\sharp$ operator.

Next, the join operator computes the sequence part $\sigma_o^\sharp$ of its outcome. This boils down to call the sequence domain join operator to the instantiated sequence abstract states. That is to say,

$$\sigma_o^\sharp := \Phi_l^{-1}(\sigma_l^{\sharp\prime}) \sqcup_s^\sharp \Phi_r^{-1}(\sigma_r^{\sharp\prime})$$

Example 4.18: Abstract join (instanciation and sequence steps)

Let us extend Example 4.17 further to illustrate the sequence part of the join operator.

The instantiation of the left abstract value is straightforward and yields:

$$\sigma_l^{\sharp\prime} : \left(\begin{array}{c} \alpha = \beta \neq 0; \\ \end{array} \begin{array}{c} \mathfrak{E} = \{S_1, S_2\} \\ \mathfrak{R} = \{S_1 \sim S_2; S \sim S_0\} \\ \mathfrak{S} = \{S, S_0, S_1, S_2\} \end{array} \right)$$

In the instantiation of the right abstract value, the new definition $S_2 = [\beta_t].S_r'$ compacts the definition of S and introduces the numerical equality constraints $\beta_t = \mathtt{min}_{S_2}$ since S_2 is sorted. The result of the instantiation is:

$$\sigma_r^{\sharp\prime} : \left(\begin{array}{c} \alpha, \beta \neq 0 \\ \wedge\, \tau < \beta_t = \mathtt{min}_{S_2} \\ \end{array} ; \begin{array}{c} \mathfrak{E} = \{S_1\} \\ \mathfrak{R} = \{S_l \sim S_0; S_r \sim S_r'\} \\ \mathfrak{S} = \{S, S_0, S_1, S_2, S_l, S_r, S_r'\} \\ \mathfrak{D} = \left\{ \begin{array}{c} S \mapsto S_l.S_2 \\ S_2 \mapsto [\beta_t].S_r \end{array} \right\} \end{array} \right)$$

Finally, all accumulated sequence constraints are used in the instantiation step and the join operator computes the sequence part of its result $\sigma_o^\sharp = \sigma_l^{\sharp\prime} \sqcup_s^\sharp \sigma_r^{\sharp\prime}$. The definitions of S, $S = S_0$ and $S = S_l.S_2$ are unified into $S = S_0.S_2$. Additionally, since S_2 is empty in the left input, the bound constraint from the right input $\tau < \mathtt{min}_{S_2}$ is added to the result. We obtain the following sequence abstract state:

$$\sigma_o^\sharp : \left(\begin{array}{c} \alpha, \beta \neq 0 \\ \wedge\, \tau < \mathtt{min}_{S_2} \\ \end{array} ; \begin{array}{c} \mathfrak{E} = \{S_1\} \\ \mathfrak{S} = \{S, S_0, S_1, S_2\} \\ \mathfrak{D} = \{S \mapsto S_0.S_2\} \end{array} \right)$$

To sum up, the full definition of the combined abstract domain join operator is presented in Figure 4.22. If the abstract heap upper bound operator fails then the combined abstract domain operator returns $\top_{\mathbb{S}}^\sharp$. This happens when the analysis attempts to compute the upper bound of abstract states that do not share common pattern.

Theorem 4.7: Soundness of $\sqcup_{\mathbb{S}}^\sharp$

For any abstract states $s_l^\sharp$ and $s_r^\sharp$, $\gamma_{\mathbb{S}}(s_l^\sharp) \cup \gamma_{\mathbb{S}}(s_r^\sharp) \subseteq \gamma_{\mathbb{S}}(s_l^\sharp \sqcup_{\mathbb{S}}^\sharp s_r^\sharp)$

Proof. The argument is similar to the proof of Theorem 4.6. That is to say, we prove that the full concretizations of the instantiated inputs are included in the full concretization of the outcome. □

$$(\sigma_l^\sharp, m_l^\sharp) \sqcup_{\mathbb{S}}^\sharp (\sigma_r^\sharp, m_r^\sharp) := \begin{cases} (\sigma_o^\sharp, m_o^\sharp) & \text{if} \quad \begin{array}{r} \sigma_l^\sharp, C_l, \sigma_r^\sharp, C_r, \Psi \vdash m_l^\sharp \sqcup_{\mathbb{M}}^\sharp m_r^\sharp = m_o^\sharp \\ \wedge\, \mathbf{sat}_s^\sharp(\sigma_l^{\sharp\prime}, \Phi_l' C_l') = \mathbf{true} \\ \wedge\, \mathbf{sat}_s^\sharp(\sigma_r^{\sharp\prime}, \Phi_r' C_r') = \mathbf{true} \\ \sigma_l^{\sharp\prime}, \Phi_l, C_l' := \mathbf{instantiate}_{\mathbb{S}}^\sharp(\sigma_l^\sharp, \mathbf{fst} \circ \Psi, C_l) \\ \text{where } \sigma_r^{\sharp\prime}, \Phi_r, C_r' := \mathbf{instantiate}_{\mathbb{S}}^\sharp(\sigma_r^\sharp, \mathbf{snd} \circ \Psi, C_r) \\ \sigma_o^\sharp := \Phi_l^{-1}(\sigma_l^{\sharp\prime}) \sqcup_s^\sharp \Phi_r^{-1}(\sigma_r^{\sharp\prime}) \end{array} \\ \top_{\mathbb{S}}^\sharp & \text{otherwise} \end{cases}$$

Figure 4.22: Abstract join

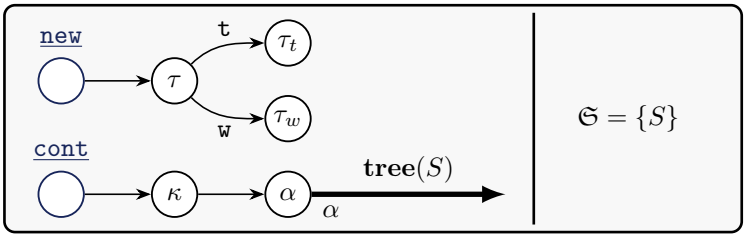


Figure 4.23: Pre-Condition

4.4.2.4 Widening

The widening operator of the combined abstract domain $\nabla_{\mathbb{S}}^\sharp$ is obtained by modifying the join operator in order to ensure the convergence of iterations.

In the memory step, we forbid the usage of rule **seg-intro**, in the case it inserts a segment when the left abstract memory is empty. Indeed, it is the only rule that creates a separation logic predicate out of thin air. This ensures that the function mapping to any abstract memory graph the pair formed by its number of points-to edges and the number of inductive edges is a decreasing measure.

The instantiation step is not modified in the widening.

The sequence step of the widening is similar to the one from the join except the last step. To compute the sequence part of the outcome, the widening of the combined abstract domain uses its sequence domain counterpart instead of the sequence domain join.

Theorem 4.8: Soundness and termination of $\nabla_{\mathbb{S}}^\sharp$

For any abstract states $s_l^\sharp$ and $s_r^\sharp$, $\gamma_{\mathbb{S}}(s_l^\sharp) \cup \gamma_{\mathbb{S}}(s_r^\sharp) \subseteq \gamma_{\mathbb{S}}(s_l^\sharp \nabla_{\mathbb{S}}^\sharp s_r^\sharp)$.

Additionally, for any sequence $(s_n^\sharp)_{n\in\mathbb{N}}$ of abstract states, the sequence $(s_n^{\sharp(\nabla)})_{n\in\mathbb{N}}$ defined as $s_0^{\sharp(\nabla)} := s_0^\sharp$ and $\forall n \in \mathbb{N}, s_{n+1}^{\sharp(\nabla)} := s_n^{\sharp(\nabla)} \nabla_{\mathbb{S}}^\sharp s_{n+1}^\sharp$ is ultimately stationary.

Proof. The soundness part is similar to the proof of Theorem 4.7.

Since the termination is independent of the sequence parametrization of inductive predicates, we refer the reader to [CR08]. The termination argument boils down to observing that the number of edges in the graph does not increase. This entails that after some iteration the memory part of the abstract state is stationary. Consequently, the sequence domain widening converges, then the widening of the reduced product converges as well. □

4.5 A final example

In this section, we revisit the analysis of the `insert` function from Listing 1.2 presented in Section 2.4.4 thanks to the analysis based on the combined abstract domain. The pre-condition of this analysis is depicted in Figure 4.23. In the pre-condition, the container is now summarized by a **tree** inductive predicate with a sequence parameter S. The sequence part of the precondition simply asserts that this sequence is sorted.

Listing 4.1: Insertion function from Listing 1.2

```
1  void insert(task* new, task_container** container){
2    node* node = malloc(sizeof(node));
3    node->task = new;
4    node->left = node->right = null;
5    if( *container ){ // Non-Empty Case
6      struct node* c = *container;
7      while(c->content->wst <= new->wst && c->left ||
8            c->content->wst >  new->wst && c->right )
9        c = c->content->wst <= new->wst ? c->left : c->right;
10     node->parent = c;
11     if( c->content->wst <= new->wst ){
12       c->left = node;
13     } else {
14       c->right = node;
15     }
16   } else { // Empty Case
17     *container = node->parent = node;
18   }
19 }
```

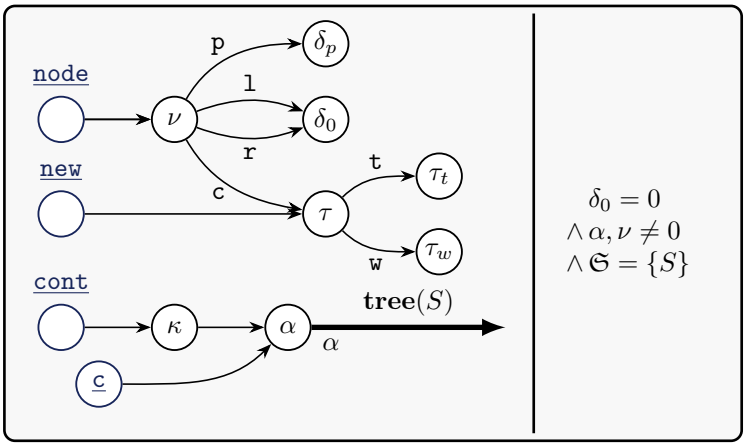


Figure 4.24: Abstract state at the end of line 6

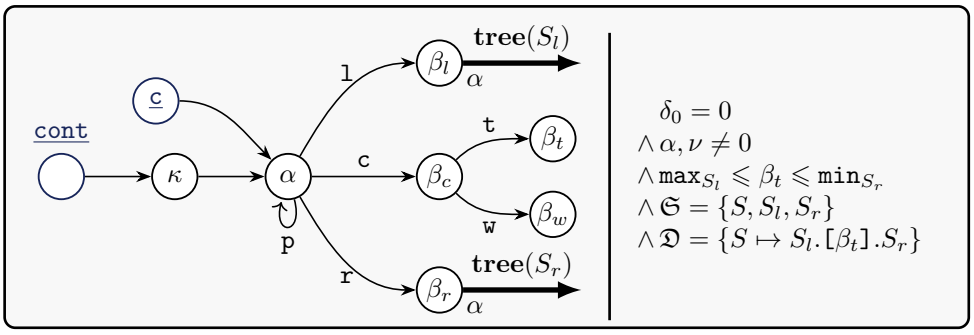
4.5.1 Initialization

The analysis of the initialization of `insert` (*i.e.* lines 2 to 4) is similar to the one presented earlier. A new memory cell corresponding to a tree node is allocated at some non-null address ν. Next, the fields of this cell are assigned. This adds points-to predicates $\nu.\texttt{task} \mapsto \tau$, $\nu.\texttt{left} \mapsto \delta_0$, and $\nu.\texttt{right} \mapsto \delta_0$, where δ_0 is a symbolic variable that is equal to the null value in the sequence part of the abstract state. Since that `prev` field of this node is not assigned, it points to some unconstrained symbolic variable δ_p.

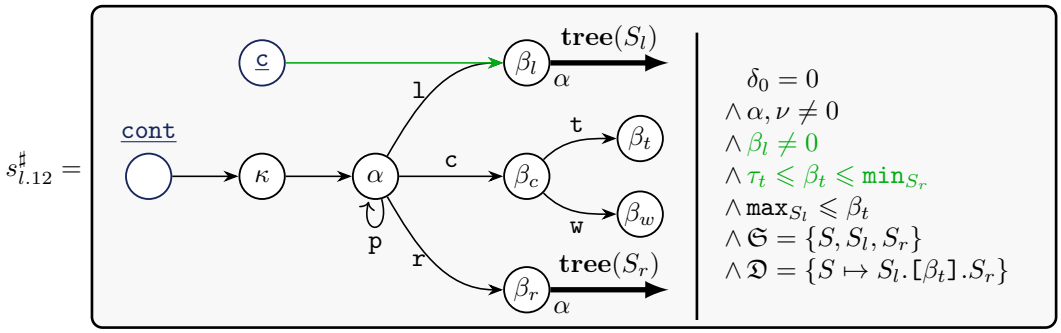
Like the baseline analysis presented in Section 2.4.4, we focus here on the non-empty case. The constraint of the conditional statement in line 5, corresponds to $\alpha \neq 0$. It is added in the numerical part of the abstract state. Additionally, after performing the assignment of line 6, the analysis obtains the abstract state, written $s^\sharp_{l.6}$, presented in Figure 4.24.

4.5.2 Analysis of the loop

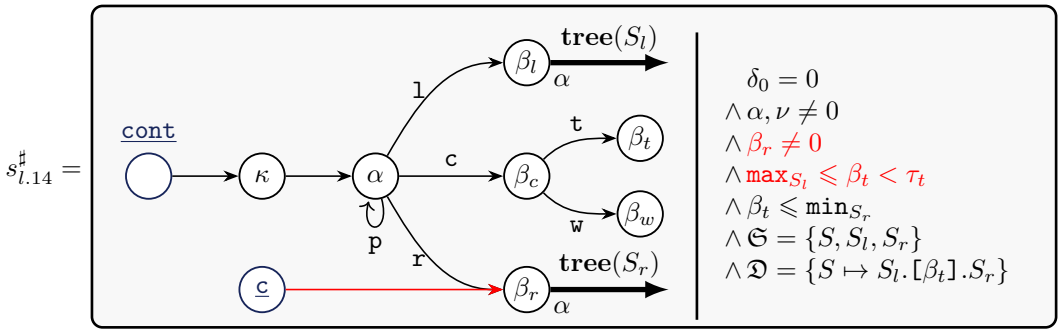
First iteration Next, the analysis proceeds with the loop. Given that the loop condition comprises a disjunction of two cases, each case is analyzed separately. In the first one, the analysis needs to evaluate `c->content->wst`. To do so, the analysis unfolds the inductive predicate instance $\alpha.\textbf{tree}(S)$. The empty rule is not feasible since it is inconsistent with the numerical constraint $\alpha \neq 0$. The non-empty rule introduces the sequence constraint $S = S_l.[\beta_t].S_r$ in the sequence part of the abstract state. The sortedness of S implies that S_l and S_r are sorted and that $\texttt{max}_{S_l} \leqslant \beta_t \leqslant \texttt{min}_{S_r}$. For the sake of brevity, we omit numerical inequalities that are not relevant for the analysis as well as multiset constraints since they are derived by translation of sequence constraints. Then, the analysis



(a) Abstract state obtained after unfolding α.**tree**(S) and β_c.**task**(β_t)



(b) Abstract state computed at the end of the first iteration (left case)



(c) Abstract state computed at the end of the first iteration (right case)

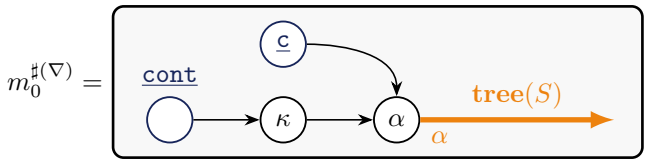
Figure 4.25: Abstract states computed during the first iteration

unfolds the **task** predicate contained in the node. After these two unfolding, the analysis computes the abstract state presented in Figure 4.25a.

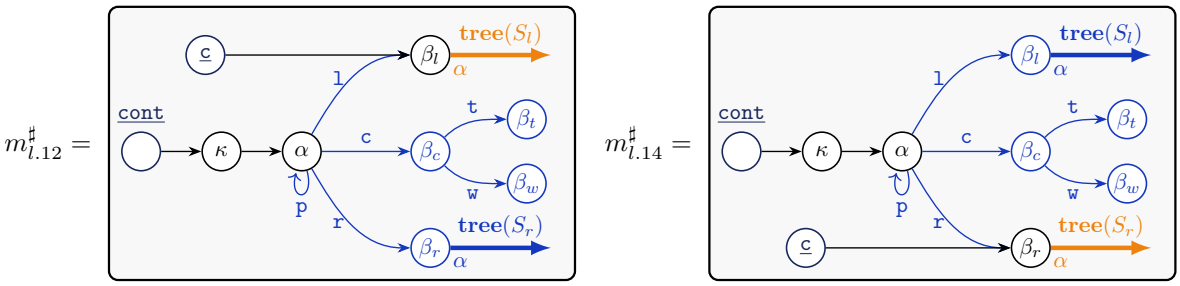
In this abstract state, the two constraints in the first disjunction boil down to $\beta_t \leqslant \tau_t$ and $\beta_l \neq 0$. They are added in the numerical part of the abstract state. Then, the analysis interprets the body of the loop. In the current abstract state, the assignment is performed by writing β_l as the destination of the points-to predicate from c. After the first iteration through the loop, the analysis generates the abstract state presented in Figure 4.25b.

Analyzing the second loop condition is carried out similarly, resulting in the state shown in Figure 4.25c.

First widening After the first iteration, the analysis joins the state at the head of the loop (presented in Figure 4.24) with the result of the analysis of the body of the loop (presented in Figures 4.25b and 4.25c). This join resembles the one discussed in Examples 4.17 and 4.18. In all states, c points to a full inductive predicate. Consequently, the abstract output memory contains an inductive predicate β.**tree**(S_0), pointed by c. The remaining of the tree, *i.e.* the part between the nodes pointed by cont and c is summarized by a segment predicate. Figure 4.26 presents for each input what part of the memory states are summarized in an inductive predicate in the outcome. It also displays, next to



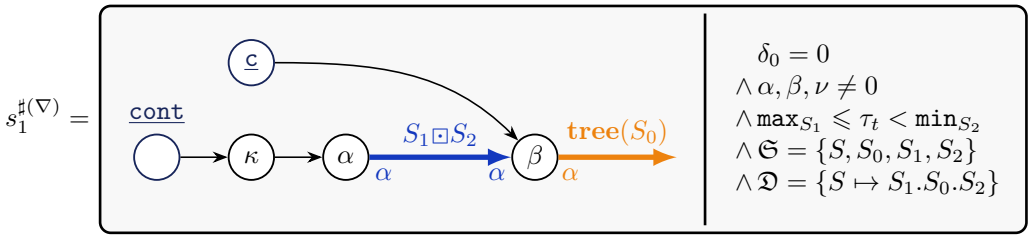
(a) Left input



(b) Right inputs

Left input	Right inputs	
	$m_{l.14}^{\sharp}$	$m_{l.16}^{\sharp}$
$S_0 = S$	$S_0 = S_l$	$S_0 = S_r$
$\wedge\, S_1 = []$	$\wedge\, S_1 = []$	$\wedge\, S_1 = S_l.[\beta_t]$
$\wedge\, S_2 = []$	$\wedge\, S_2 = [\beta_t].S_r$	$\wedge\, S_2 = []$

(c) Accumulated sequence constraints



(d) Output

Figure 4.26: First widening

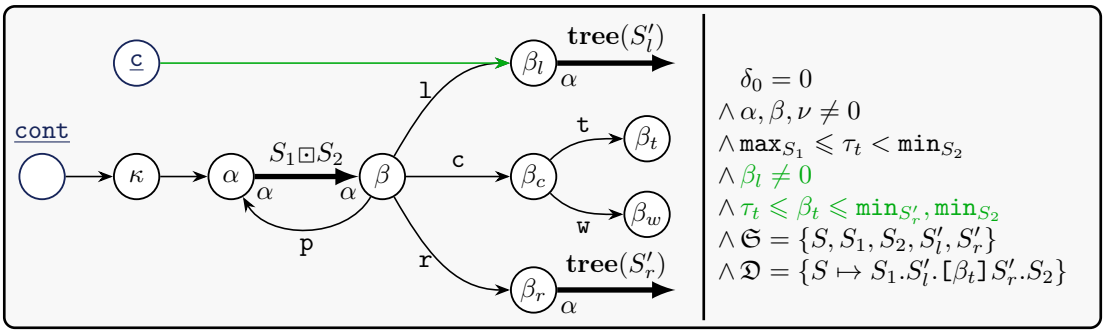
each input, the sequence constraints inferred during the memory join.

Following the memory part, the sequence parts of the inputs are instantiated. Since all sequence constraints are definitions of fresh sequence variables they are all guarded. The **guard**$_s^{\sharp}$ operator infers that S_0, S_1, and S_2 are sorted in all states. Following the instantiation, the sequence abstract values are joined. The join operator successfully unifies all definitions of S into $S = S_1.S_0.S_2$. This constraint expresses the conservation of the content of the tree during the exploration. Additionally, using the infinite bounds saturation principle for empty sequences, the operator infers the numerical inequalities $\mathtt{max}_{S_1} \leqslant \tau_t < \mathtt{min}_{S_2}$. In essence, this constraint states that the point of insertion of the new task is located somewhere between sequences S_1 and S_2.

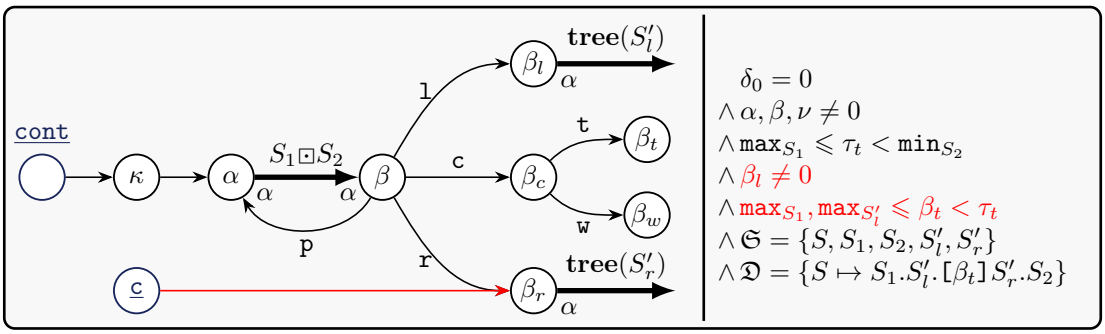
This first widening yields the abstract state $\sigma_1^{\sharp(\nabla)}$ presented at the bottom of Figure 4.26.

Second iteration Since $s_{l.6}^{\sharp}$ differs from $s^{\sharp(1)}$, the loop invariant is not stable. Therefore, the analysis continues with another iteration. The second iteration follows the same principle as the first one, given that c corresponds to the non-null address of a well-formed binary tree. It is unfolded, as well as the task stored at the corresponding node. Once again, we obtain a disjunction consisting of two elements presented in Figure 4.27.

Second widening After the second iteration, the analysis performs a widening between the previous widened abstract state, and the result of the iteration. The widening follows the same pattern



(a) Abstract state computed at the end of the second iteration (left case)



(b) Abstract state computed at the end of the second iteration (right case)

Figure 4.27: Abstract states computed during the second iteration

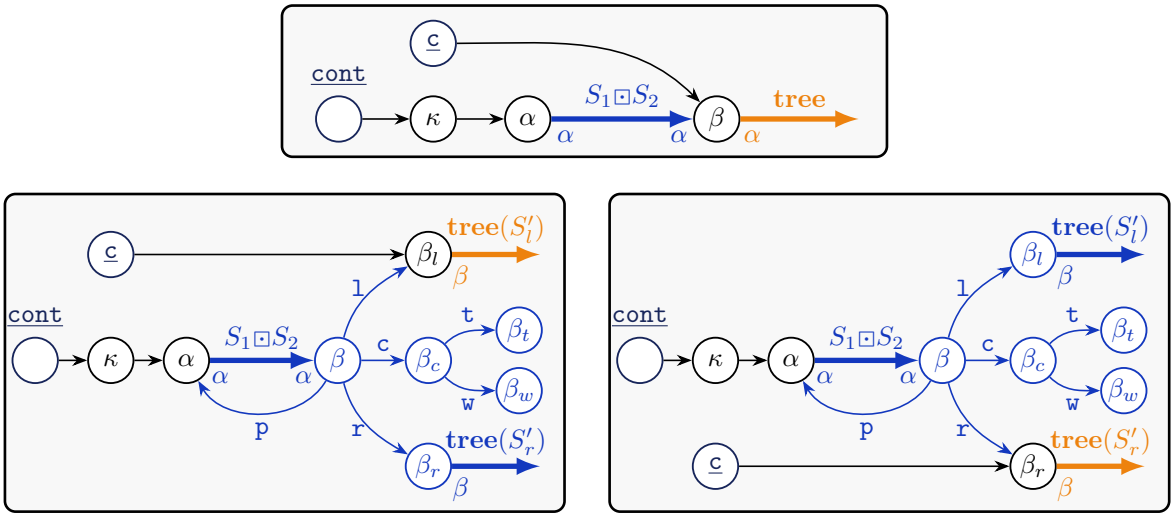
as the previous one: `c` points to a full inductive predicate and the part of tree between `container` and `c` is summarized in a segment predicate. Figure 4.28 displays the parts of the abstract states that are matched together, as well as the result of the second widening $s_2^{\sharp(\nabla)}$. The sole difference between $s_1^{\sharp(\nabla)}$ and $s_2^{\sharp(\nabla)}$ lies in the fact that the address α pointed by `container` is no longer the backward pointer of the tree pointed by `c`. Instead, it is some non-null address π.

Third iteration The invariant remains unstable. Therefore, the analysis performs a third iteration. Following the widening of $s_2^{\sharp(\nabla)}$ with the result of the third iteration, we observe that $s_2^{\sharp(\nabla)}$ is stable. This means that $s_2^{\sharp(\nabla)}$ is the invariant of the loop inferred by the analysis.

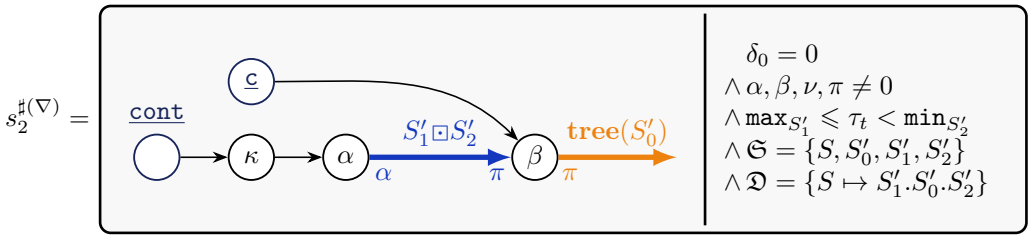
4.5.3 Insertion

Once the invariant computed, the analysis proceeds with the insertion of the new task in the tree. There are two possible cases for program executions to leave the loop. Either, the value in the node pointed by `c` is lower or equal than τ_t and the left child of the node is null, or the value is greater and the right subtree is null. Let us consider the first case. To evaluate the expressions, the analysis unfolds the subtree pointed by `c`. Once again, the empty rule is inconsistent so the analysis discards it. And, the non-empty rule guards the sequence constraint $S = S_l.[\beta_t].S_r$. Then, the analysis unfolds the **task** predicate in the node, and it guards the constraints $\tau_t \leqslant \beta_t$ and $\beta_l = 0$. The last constraint implies the left subtree is empty. Consequently, the analysis infers the sequence definition $S_l = []$. At the exit of the loop, the analysis computes the abstract state presented in Figure 4.29a.

Next, the analysis handles the assignments in lines 10 and 12. This boils down to changing the destinations of points-to predicates to obtain $\beta.\texttt{left} \mapsto \nu$ and $\nu.\texttt{prev} \mapsto \beta$. To sum up, the final abstract state is depicted in Figure 4.29b.

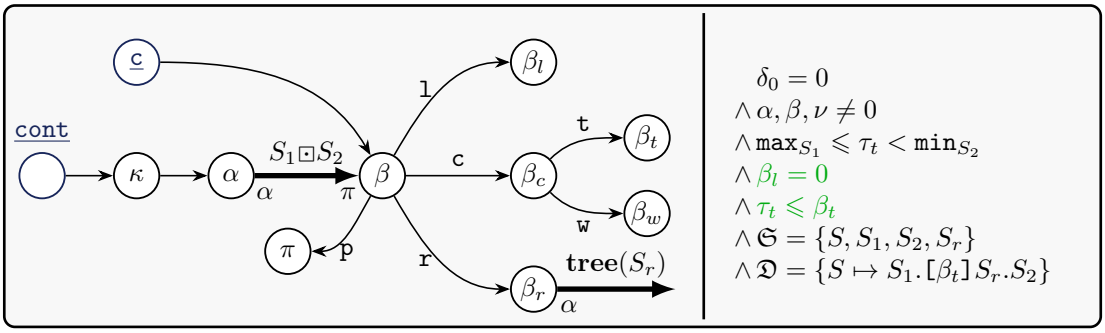


(a) Inputs (memory parts)

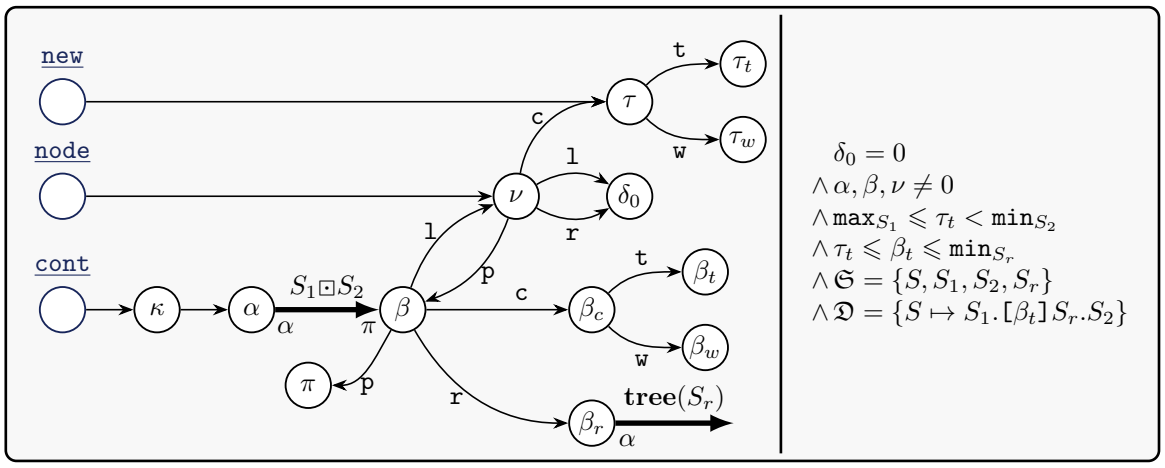


(b) Output

Figure 4.28: Second widening



(a) Abstract state computed at the exit of the loop (left case)



(b) Final state (left case)

Figure 4.29: Abstract states computed during the insertion (left exit case)

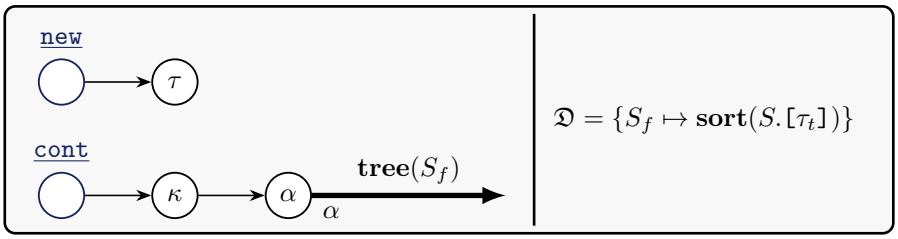


Figure 4.30: Post-Condition

4.5.4 Verifying the post-condition

Once the analysis computed the final states, it verifies that they imply the post-condition by performing an inclusion checking. The post condition of the `insert` function is resented in Figure 4.30. The post-condition expresses that `cont` points to a pointer to a full binary tree containing a sequence of elements S_f. The sequence part of the post-condition constraints this sequence variable to be equal to the sorted counterpart of S appended by the element denoted by τ_t. In essence this definition states that S_f is the sequence obtained after inserting τ_t in S while maintaining sortedness. Therefore, this post-condition is expressive enough for the partial functional correctness of the insertion in a binary search tree.

For instance, let us consider the inclusion checking between the final state presented in figure 4.29b and the post-condition. The inclusion between the memory parts succeeds and produces the following sequence constraints:

$$\begin{aligned} &\wedge S'' = [\tau_t] \\ &\wedge\ S' = S''.[\beta_t].S_r \\ &\wedge S_f = S_1.S'.S_2 \end{aligned}$$

These constraints are added in the sequence part of the final state during the instantiation phase. Then, the inclusion operator checks that the only constraint in the post-condition is verified in the instantiated final state. Given that this is the case, the analysis concludes that the `insert` function is a correct implementation (modulo the termination) of the insertion in a binary search tree.

4.6 Implementation and evaluation

In this section, we report on the implementation and evaluation of the product shape and sequence analysis. We consider the following research questions:

- **(RQ1)** Is the combined analysis precise enough to prove functional properties on programs implementing classical algorithms over dynamic data structures (like lists, sorted lists, and binary search trees), and does it improve the baseline analysis to verify that structural invariants are preserved?
- **(RQ2)** Can this analysis successfully verify real-world C libraries?
- **(RQ3)** How significant is the overhead of the combined analysis compared to the baseline?

Implementation. We have implemented the sequence abstract domain and the product with the shape abstraction of the MemCAD static analyzer [LBCR17, GRR23a]. The analysis inputs C programs and user-supplied inductive predicates describing data structures together with pre- and post-conditions and attempts to verify them, as well as absence of runtime errors. We set convex polyhedra [CH78] implemented in the Apron library [JM09] as numerical abstraction and an extension of [CCLR15] as multiset abstraction.

Experiments. We consider two sets of experiments. The first one (Table 4.1) consists of custom implementations of classical algorithms over lists, sorted lists, and binary search trees and includes sorting, insertion and deletion algorithms. The second (Table 4.2) collects list data structure implementations taken from the Linux [Tor22] and FreeRTOS [Inc22] operating systems as well as the Generic data structure library (GDSL) [Dar04], which all involve specificities like back pointers or

Example	without seq			with seq parameters					
	time (ms)	#iter	**PrSafe**	time (ms)				#iter	**PrSafe**
	all		verified	all	num	seq	shape		+ **Fc** verified
Singly linked list									
Push	4.0		✓	4.8	0.5	0.5	0.9		✓
Pop	5.1		✓	5.4	0.9	1.4	0.8		✓
Pop (empty)	4.9		✓	4.7	0.8	0.5	1.4		✓
concat	6.5	2	✓	15.7	3.4	3.3	2.7	2	✓
deep copy	12.1	2	✓	20.4	3.7	2.9	5.5	2	✓
length	9.5	3	✓	45.0	22.5	5.0	8.1	3	✓
insert at position	19.0	3	✓	101.9	61.3	7.9	12.2	3	✓
remove at position	17.2	3	✓	92.5	55.5	6.5	12.5	3	✓
inserting in a sorted list	13.5	3	✓	82.5	39.0	10.0	9.2	3	✓
minimum	11.8	3	✓	92.3	42.4	11.1	16.8	3	✓
maximum	11.8	3	✓	93.2	42.9	11.2	17.0	3	✓
insertion sort	24.6	2, 2	✓	714.6	328.6	90.0	126.3	4, 3	✓
bubble sort	40.6	2;2,3	✓(†)	776.3	399.5	89.2	141.5	3;3,3	✓(†)
merge	36.8	4	✓	352.2	180.9	41.0	54.9	4	✓
Binary trees									
Delete leftmost	11.2	3	✓	80.5	38.2	9.4	12.0	3	✓
Delete rightmost	11.5	2	✓	58.1	27.5	6.8	7.6	2	✓
Binary search trees									
Insertion	25.2	2	✓	150.4	58.0	17.2	15.5	2	✓
Delete max	22.9	2	✗	141.2	68.6	15.2	17.2	2	✓
Delete min	22.0	3	✗	177.9	87.9	19.2	22.8	3	✓
Search (present)	26.6	2	✓	107.2	48.6	15.7	14.4	2	✓
Search (absent)	24.0	3	✓	76.7	29.4	11.4	11.7	3	✓
BST to list (heap sort)	23.8	3	✓	76.5	29.2	11.4	11.7	3	✓
list to BST (heap sort)	34.2	2,2	✓	408.0	188.0	56.5	68.4	3,2	✓

Table 4.1: Experimental results on custom examples

sentinel nodes. For each data structure, we provide an inductive definition written in the DSL of MemCAD. This amounts to a single definition a few lines long shared across all tests of a given series. For each test, we also specify the pre- and post-condition of procedures. When a procedure may behave differently depending on the shape of its input, we provide two pre-/post-condition pairs. This occurs for the "Pop" function, which does nothing when applied to the empty list. Two target properties are studied:

- **PrSafe**: absence of memory errors and structural preservation (with respect to list or tree invariants but without checking anything about their contents);
- **Fc**: partial functional correctness (including sortedness and the preservation of the elements stored in data structures).

We ran the experiments on a machine with an i7-8700 processor with 32 GB of RAM running Ubuntu 18.04. For each test case, we run the analysis *without* and then *with* sequence abstraction to compare runtimes and check whether the analyses prove the expected property. When using the analysis without sequence abstraction, only **PrSafe** is considered (this abstraction cannot express **Fc**), whereas the analysis of sequences attempts to discharge both **PrSafe** *and* **Fc**. Table 4.1 displays raw results for the first series of tests. Table 4.2 shows the results of the tests in the second series. These tables present the time, in milliseconds, averaged over 100 runs, spent by different elements of the analysis. We also present the number of loop iterations required by the analysis. For loop iterations, disjoint loops are separated by a semicolon, nested loops by a comma, and the first number corresponds to the outer loop. For inner loops, we take the maximum number of iterations needed to stabilize it.

Verification of complex properties. As shown in Table 4.1, the analysis with sequences fully verifies both memory safety and functional correctness (**PrSafe** and **Fc**) for all target codes including

Example	without seq			with seq parameters					
	time (ms)	#iter	property	time (ms)				#iter	property
	all		verified	all	num	seq	shape		verified
Linux lists									
Init	1.1		✓	2.6	0.2	0.3	1.1		✓
Input	13.6		✓	21.4	2.7	2.4	8.2		✓
Output	22.7		✓	31.5	4.8	4.8	10.5		✓
Output (empty)	33.8		✓	9.3	1.4	1.0	2.5		✓
FreeRTOS lists									
vListInit	4.3		✓	6.1	1.3	0.4	0.6		✓
vListInsertEnd	23.8		✓	40.3	10.8	1.8	5.3		✓
vListInsert	87.4	4	✓	370.5	202.4	27.2	37.9	4	✓
vListRemove	47.5		✓	163.4	82.6	9.2	20.0		✓
GDSL (lists)									
Alloc	12.0		✓	14.8	2.2	1.4	3.0		✓
Flush	24.3	2	✓	59.4	18.4	5.4	16.1	2	✓
Free	35.3	2	✓(†)	79.9	25.1	7.4	24.0	2	✓(†)
Get size	3.6		✓	6.1	2.5	0.3	0.9		✓
Is empty (non-empty)	8.1		✓(†)	14.0	4.6	1.2	3.7		✓(†)
Is empty (empty)	8.1		✓(†)	24.3	12.1	1.8	3.9		✓(†)
Get head (empty)	9.5		✓	14.7	4.5	1.1	3.6		✓
Get head (non-empty)	9.4		✓	29.8	14.5	1.8	4.3		✓
Get tail (empty)	11.1		✓(†)	26.6	14.4	1.6	4.7		✓(†)
Get tail (non-empty)	11.0		✓(†)	58.5	35.5	2.8	6.6		✓(†)
Insert head	23.2		✓	42.9	13.3	2.9	10.5		✓
Insert tail	25.0		✓(†)	54.3	20.5	3.5	11.8		✓(†)
Remove head (empty)	34.1		✓	111.9	50.9	6.5	25.4		✓
Remove head (non-empty)	34.0		✓	16.3	5.7	1.1	3.7		✓
Remove tail (empty)	49.5		✓	284.8	165.0	13.6	39.3		✓
Remove tail (non-empty)	49.5		✓	16.2	5.7	1.1	3.6		✓
Search max	69.7	5	✓	708.4	429.7	43.1	145.7	5	✓(†)
Search min	69.4	5	✓	634.0	380.3	35.4	131.2	5	✓(†)
Search by position	104.5	3;2	✗(†)	1182.8	796.3	40.7	108.2	3;3	✓(†)

Table 4.2: Experimental results on real-world libraries

three different list sorting programs, operations on binary search trees as well as heap sort (elements of a list are all inserted in an empty binary search tree and collected in a left to right order back into a list). The majority of these examples requires the inference of fairly involved invariants. The analysis without sequences can in theory only verify at most **PrSafe**, yet it fails to do so in several examples, where the use of sequences actually also lets the analysis verify **PrSafe** (in addition to **Fc**). This result is somewhat surprising, as we would not expect sequence information be required to establish basic safety. One caveat is that one example (bubble sort) required the manual insertion of directives to MemCAD in order to avoid folding (see Remark 4.3). All other analyses are fully automatic. We conclude the product with sequences not only allows to prove **Fc** even in challenging cases, but may also help with **PrSafe**.

Remark 4.3: Unfolding directive in the bubble sort

In the bubble sort program presented in Listing 4.2, the list to sort is traversed be a pointer called `bubble`. The invariant we try to establish for this list traversal is "the list cell pointed by `bubble` contains the maximum value encountered so far". However, since the node pointed by `bubble` is not materialized in the state computed by the analysis before entering the loop, this cell is folded in the invariant computed by the analysis (depicted in Figure 4.31a). This means that the value stored in this cell is not explicitly denoted by a symbolic variable in the shape part of the invariant. Consequently, the sequence part of the invariant cannot express the expected constraint since it is not able to express any constraint on the content of the cell pointed by `bubble`.

Listing 4.2: Code of bubble sort

```
1  void bubble_sort(list l){
2    if( l != null ){
3      list bubble = l; // unfold♯_𝕊(bubble)
4      while( bubble->next != null ){
5        int v1 = bubble->data;
6        int v2 = bubble->next->data;
7        if (v2 < v1){
8          bubble->data = v2;
9          bubble->next->data= v1;
10       }
11       bubble = bubble->next;
12     }
13     list end = bubble
14     while( end != l ){
15       bubble = l; // unfold♯_𝕊(bubble)
16       while( bubble->next != end){
17         int v1 = bubble->data;
18         int v2 = bubble->next->data;
19         if (v2 < v1){
20           bubble->data = v2;
21           bubble->next->data= v1;
22         }
23         bubble = bubble->next;
24       }
25       end = bubble;
26     }
27   }
28 }
```

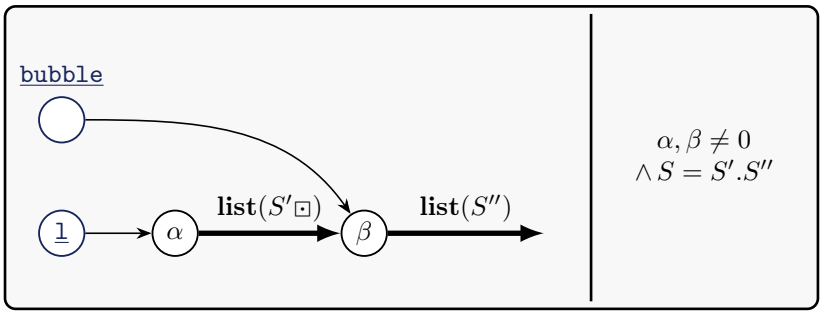
To address this issue, we introduce a directive at the beginning of each list traversal. This directive tells the analysis to perform a forward unfolding on the **list** predicate pointed by `bubble`. This ensures that in the invariant computed by iterated applications of $\nabla^\sharp_{\mathbb{S}}$, presented in Figure 4.31b, the data stored in the pointed cell is expressed by a symbolic variable δ. Thanks to this directive, the sequence part of the invariant expresses the expected constraint between the data stored in the cell pointed by `bubble` and the sequence of values already traversed: $\texttt{max}_{S'} \leqslant \delta$.

Verification of real-world libraries. We now consider Table 4.2. These examples involve lists with invariants that are considerably more sophisticated than **list**, as they are all doubly-linked lists with headers. While GDSL lists contain a pointer to stored value blocks, both Linux and FreeRTOS lists are intrusive lists in the sense of the Linux kernel terminology: the `C` struct containing the `next` and `prev` fields is a substructure of the list node, which implies structure accesses require more complex pointer operations. FreeRTOS lists explicitly store a pointer from substructures to owners, whereas Linux lists rely on pointer arithmetic to access containing blocks. Finally, both FreeRTOS and GDSL lists have a header that stores the number of elements in the lists. FreeRTOS list nodes store a pointer to this header.

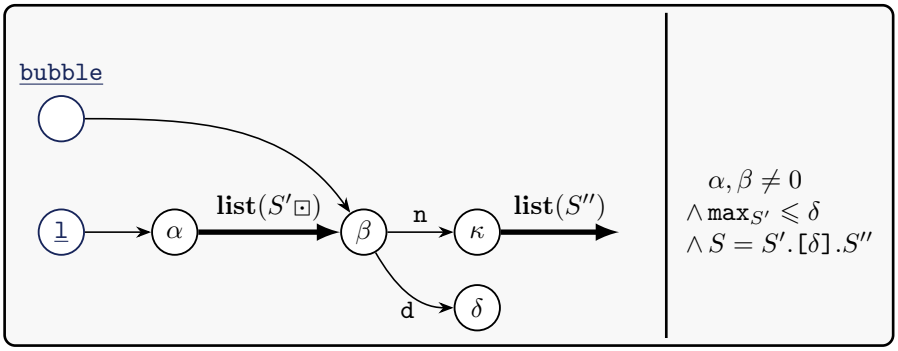
Remark 4.4: Specification of FreeRTOS lists

Listing 4.3 presents the definition of FreeRTOS lists, and Figure 4.32a shows an instance of such a list. To represent circular doubly-linked list, we employ a segment predicate. The corresponding full inductive predicate, **FreeRTOSNode** is defined in Figure 4.32b. In addition to the main parameter α, this predicate has two numerical parameters. The first one, π is a backward parameter for the value of the `pxPrev` field. The second one, κ denotes the address of the leader of the list. Similarly to predicate **addrList**, the FreeRTOS list item predicate has two sequence parameters: one for the sequence of nodes addresses S_a, the second one, S_v, for their values. Since the analysis only manipulates the segment counter-part of **FreeRTOSNode**, there is no need to define a base case. Consequently, its definition contains a single rule: the recursive one.

To express a complete list, the header is represented explicitly using points-to predicates. Next, the items in the list are summarized by a segment predicate from the node pointed by the `pxPrev` filed of the header back to the header. The remaining invariants of the list are expressed in the sequence part of the abstract value. The sequence of values is sorted, and the



(a) Invariant computed without the unfolding directive



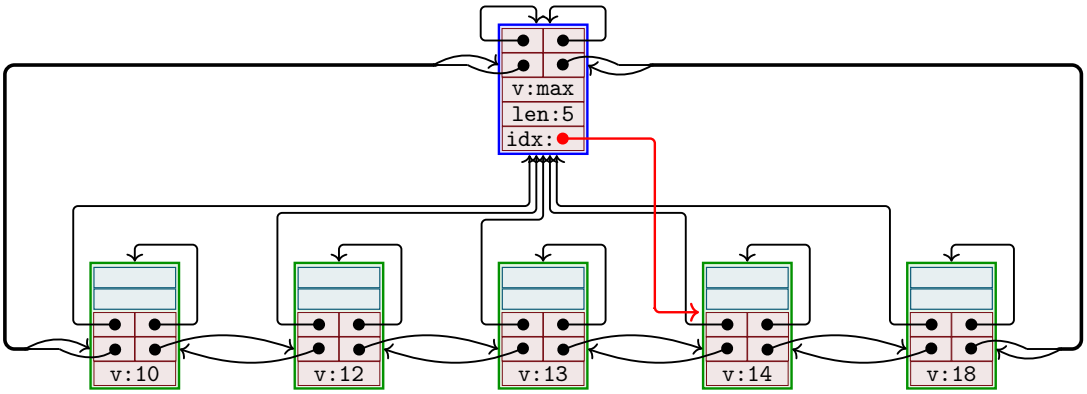
(b) Invariant computed with the unfolding directive

Figure 4.31: Invariants of the first list traversal in the bubble sort

value contained in the header is an upper-bound of these values. Additionally, the value of the `pxIndex` field in the header is an element of the sequence of addresses. To sum up, Figure 4.32c presents an abstract state that corresponds to a well-formed FreeRTOS list, where the header is denoted by the node η.

The analysis with sequences proves both **PrSafe** and **Fc** for all Linux and FreeRTOS primitives. It was also able to fully verify almost all the GDSL list library, although seven cases required a manual directive to prevent aggressive folding both with and without sequences (as for bubble sort in Table 4.1) (they are marked with (†) in the tables). All tests are successfully verified. In the case of GDSL, we observed that the baseline analysis was not able to prove the absence of run-time error in the Search by position example. Indeed, this function checks that the searched position is correct, by comparing it with the length value stored in the header. If this check succeeds, then the function traverse the list to reach the desired location. During this traversal, no check is performed to ensure that it does not reach the end of the list. The baseline analysis is not able to use the sanity check performed at the beginning of the function since it cannot link the length value stored in the header with the content of the list. In the contrary, the content aware analysis manages to establish that the sanity check ensures that the list traversal is free of run-time error. We conclude the analysis can handle real-world programs.

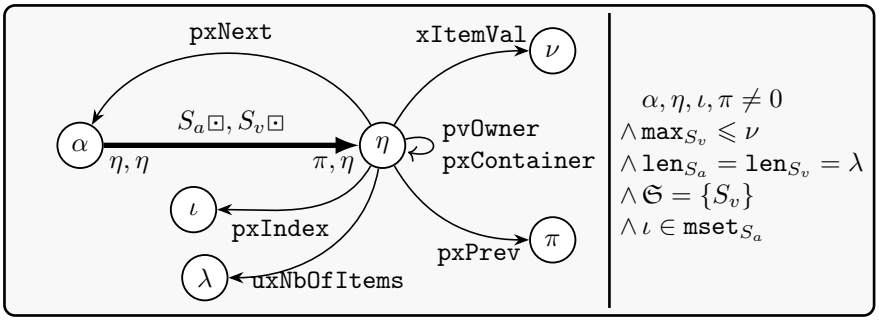
Overhead. We now compare performance between the analyses with/without sequences in Tables 4.1 and 4.2. While the overhead is modest for the smaller programs, it becomes higher for the more challenging cases, up to roughly 10x-20x. While significant, this cost should be considered in comparison to the much stronger properties proved (*i.e.*, not only **PrSafe** but also partial correctness **Fc** in addition to **PrSafe**). We found two reasons for this increase. First, as shown in the tables, most of the increase is accounted for by the numerical abstract domain partly due to the larger number of symbolic variables that stand for sequence bounds. We believe this overhead could be much reduced with a finer-grained numerical domain packing [BCC+03, SPV17]. By contrast, the time spent in memory and sequence domains remains reasonable. For instance, the time taken by the sequence domain is no more than half of the time spent in the baseline analysis. Second, the analysis with sequences requires greater numbers of abstract iterates to stabilize loop invariants, as shown in the



(a) FreeRTOS list

$$\begin{aligned}\alpha.\mathbf{FreeRTOSNode}(\pi,\kappa,S_a,S_v) := \exists \beta_n,\beta_v,S'_a,S'_v,\ & \alpha.\texttt{xItemValue} \mapsto \beta_v \\ & * \alpha.\texttt{pxNext} \mapsto \beta_n * \alpha.\texttt{pxPrev} \mapsto \pi \\ & * \alpha.\texttt{pvOwner} \mapsto \alpha * \alpha.\texttt{pxContainer} \mapsto \kappa \\ & * \beta_n.\mathbf{freeRTOSNode}(\alpha,\kappa,S'_a,S'_v) \\ & \wedge \alpha \neq 0 \wedge S_a = [\alpha].S'_a \wedge S_v = [\beta_v].S'_v\end{aligned}$$

(b) Inductive predicate for FreeRTOS lists item



(c) Abstract state denoting a FreeRTOS list

Figure 4.32: Formalization of a FreeRTOS list

Listing 4.3: FreeRTOS list type definitions

```
/** Definition of the only type of object that a list can contain. */
struct xLIST_ITEM
{
  configLIST_VOLATILE TickType_t xItemValue;
  /**< The value being listed.  In most cases this is used to sort the list in ascending order. */
  struct xLIST_ITEM * configLIST_VOLATILE pxNext;
  /**< Pointer to the next ListItem_t in the list. */
  struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
  /**< Pointer to the previous ListItem_t in the list. */
  void * pvOwner;
  /**< Pointer to the object (normally a TCB) that contains the list item. */
  struct xLIST * configLIST_VOLATILE pxContainer;
  /**< Pointer to the list in which this list item is placed (if any). */
};
typedef struct xLIST_ITEM ListItem_t;
typedef struct xLIST_ITEM MiniListItem_t;

/** Definition of the type of queue used by the scheduler. */
typedef struct xLIST
{
  volatile UBaseType_t uxNumberOfItems;
  MiniListItem_t xListEnd;
  /**< List item that contains the maximum possible item value. */
  ListItem_t * configLIST_VOLATILE pxIndex;
  /**< Used to walk through the list. */
} List_t;
```

tables, which explains an important slowdown. This is to be expected due to the more complex value constraints (including polyhedra) used in the analysis with sequences.

4.7 Related work

In this section, we discuss previous work on the abstractions of sequences stored in data structures.

4.7.1 Linear and contiguous structures (arrays and strings)

Arrays Several previous works have tried to tie properties of container data structures with properties of their contents. In particular, [GRS05, GMT08] have extended array abstractions with basic contents properties. Subsequently, Halbwachs *et al.* [HP08] introduced array segmentations and Cousot *et al.* [CCL11] made the computation of the array segmentations dynamic during the analysis. The latter two can express that an array is sorted and verify that a function produces sorted arrays. However, they do so with specific predicates rather than an abstraction for sequences. Thus, they cannot express that the set of elements in an array is preserved, which is required to prove a sorting function correct. By contrast, our sequence abstraction handles both sortedness and contents preservation.

Strings and regular expressions Strings and buffers also motivated many research works, as operations on them may incur a security risk. In particular, improper handling of zero terminated strings make opens the door to buffer overrun attacks. Therefore, works such as CSSV [DRS03] abstract the presence or absence of zeroes in strings and their positions in order to verify buffer operations. Besides zeroes, these works do not keep any contents' information.

As noted earlier, several recent works applied concepts such as regular expressions and automata in order to build string abstract domains, that convey precise contents information [MNN16, AM19, NAFC21]. These works are typically aimed at inferring precise information on strings that denote pieces of programs meant to be computed and evaluated at runtime as in the case of JavaScript's `eval` construction. Automata and regular expressions are most adequate for such target properties. More recently, Arceri *et al.* [AOCF22] extended these works with length and element position constraints. These abstractions are not aimed at numerical sequences, and fail to express sortedness. By contrast, our sequence abstraction relies on length, extreme elements and sortedness constraints and fails to express regular expressions-based properties as these would not be useful for our intended application.

4.7.2 Shape analyses for dynamic data structures

Shape analysis Many abstractions for dynamic data structures have been proposed. Sagiv *et al.* introduced a shape analysis based on three value logic in [SRW98], that was later extended to handle more complex data structures such as tree [LRS06]. The seminal work by Reynolds [Rey02], introduced separation logic, that many analyses including ours rely upon. Separation logic has been used in order to reason over not only sequential programs [CDOY07] but also concurrent programs [O'H04, VP07] and to prove properties like linearizability of concurrent data structures [Vaf09]. It serves as a basis for structure abstraction in several static analyzers like Smallfoot [CDOY07], Facebook Infer [CDOY11] (which also performs bi-abduction to synthesize pre- and post-condition pairs), Forester [HLR+13] (which uses automata to represent abstract states), and MemCAD [LBCR17] (which features a modularized abstract domain). Bi-abduction methods have also been extended to infer inductive predicates on a per-function basis [LGQC14] or to infer pre- and post-conditions for programs manipulating lists and using bit-level memory accesses and pointer arithmetic [HPR+22]. All the shape abstractions mentioned so far can only keep track of very limited contents properties.

Indeed, inferring precise information about the contents of dynamic data structures is notoriously difficult, since the memory abstraction layout changes depending on the program point which makes abstraction complex.

Shape analysis with numerical constraint on the content A first approach to this issue consists in splitting the analysis in two phases, where the first analysis infers only structural invariants and translates the initial program into a purely numerical program, that is taken as input by the second analysis, that discovers numerical invariants. This technique has been applied by [MTLT10, FHR+18] in order to infer complexity bounds and verify termination of programs based on information on the size of the data structures. A second approach [CR08] consists of a reduced product between a memory abstract domain and a numerical abstract domain. While harder to implement, it ensures information can be communicated in both directions between the memory and the value abstract domains, whereas the staged analysis approach only lets the value abstract domain benefit from memory layout information. More recently, Li *et al.* [LRC15] combines shape and set abstractions with a reduced product which allows verifying programs on graphs. As it only considers set constraints, it does not capture any order information.

Analysis with sequence content abstraction The tools CINV [BDE+10] and CELIA [BDES12a] (extended with interprocedural analysis support in [BDES11]) are the most closely related to our approach. These static analyzers handle list manipulating programs and are parameterized by an abstract domain called a *data-word domain* to reason on the structure and contents of lists by attaching size or set constraints to them, or constraints quantified over the position of elements, which allows expressing sortedness. Although the heap abstraction does not make explicit use of separation logic the list abstraction follows a similar structure.

A first important difference with our work is that CINV and CELIA only handle singly linked lists, whereas our analysis supports a large range of inductive definitions included doubly linked-lists, trees, binary search trees with and without parent pointers. Indeed, our approach integrates sequence reasoning into a shape analysis that can be parameterized by a wide variety of inductive predicates. This more general scope requires extensions to the analysis algorithms, such as the automatic inference of concatenation lemmas (Lemma 4.2 and 4.3) and the use of abstract operators based on them.

A second difference comes from the sequence domain and the interaction with it. The data-word domain to handle sortedness relies on a decidable fragment of first order array theory based on constraints of the form $\forall \mathbf{y}, P(\mathbf{y}) \Rightarrow U(\mathbf{y}, Q_1, \dots)$, where the guard constraint $P(\mathbf{y})$ belongs to a predefined, user-provided set of *guard-patterns* constraining the index variables y_j, and U is a conjunction of linear constraints on y_j and $Q_i[y_j]$. This domain does not manipulate symbolic sequence expressions but rather follows a structural approach. For example, the concatenation constraint $S = S_1.S_2$ is expressed as $\forall y_1, y_2, y_1 < \texttt{len}_{S_1} \wedge y_2 < \texttt{len}_{S_2} \Rightarrow S_1[y_1] = S[y_1] \wedge S_2[y_2] = S[y_2 + \texttt{len}_{S_1}]$. Therefore, it requires the user to specify prior to the analysis the appropriate guard pattern. Our sequence abstraction requires no such parameterization.

4.7.3 Provers for memory and contents properties

Separation logic has also been used as foundation for verification tools based on entailment checking procedures, some of which also consider contents properties. Mnacho *et al.* [EP23] established that

entailment checking in separation logic with inductive predicates is undecidable even with simple numerical theories such as first-order arithmetic with only the successor function and the inequality predicate. Songbird [TLKC16] uses a sequent-based approach to attempt deciding implication in a fragment of separation logic enriched with pure predicates. The procedure presented in [IRV14] relies on tree automata to decide implications that involve inductive predicates. CSL [BDES09] and SLAD [BDES12b] decide entailment on a logic for singly linked lists and the data stored in them. It handles order constraints on linear structures like lists and arrays.

HIP/SLEEK [CDNQ07] combines a symbolic execution tool that performs predicate unfolding with an entailment checking procedure based on predicate folding. This solver is not able to derive segment predicates from full predicates, nor is it able to discriminate between full and segment predicates. However, this tool accepts user-provided lemmas to guide it. In that case, it attempts to prove the correctness of these lemmas. The inductive predicates can be parameterized by integer (to express its size or extreme values stored in the structure) and by multiset variables (to express its content). Consequently, it has to derive constraints for each of these parameters, whereas our approach only requires the shape operators to derive the sequence constraints. Bounds and size constraints are derived by translation from the sequence ones.

In [ESW15], Enea *et al.* propose an inference mechanism for lemma required by separation logic solvers. One kind of lemmas generated, called *completion lemmas* is similar to the concatenation lemmas from Section 4.1. However, it is worth mentioning that, in these lemmas, the segments have two multiset parameters $\alpha.\mathbf{tree}(E) \mathrel{-\!\!*\!\!\!\!=} \beta.\mathbf{tree}(E')$. The second parameter E' corresponds to the missing content, and the first one corresponds to the content obtained in the full tree if we add a predicate $\beta.\mathbf{tree}(E')$. The actual content of the segment is $E \setminus E'$. Therefore, the completion lemmas require that the content parameters match exactly. It is not able to perform reasoning such as $\alpha.\mathbf{tree}(E) \mathrel{-\!\!*\!\!\!\!=} \beta.\mathbf{tree}(E') * \beta.\mathbf{tree}(E'') \implies \alpha.\mathbf{tree}(E \setminus E' \uplus E'')$.

More recently, [CL20] used bi-abduction to reason about ordered data by explicitly storing bounds on elements in the inductive predicate. This work only considers full structure predicates and does not handle segment predicates. All these tools can be used to discharge implication proof obligations and can be used in verification tools where invariants are either manually written or inferred by some other means.

Additionally, separation logic is also heavily used in approaches based on proof assistants [Cha11, JKJ+18, Cha20]. In that case, contents properties are naturally expressed in the proof assistant language.

4.7.4 Solvers for sequence properties

Finally, we remark that our language of sequence constraints based on concatenation of atoms has some similarity with the string logic that can be found in some decision procedures. Though the logic of word equations with at least two atoms is known to be undecidable [Qui46], its quantifier free fragment has a PSPACE complete decision procedure [Mak77]. Following the work of [Ama21] that classifies the field of string constraints solving in three main branches, the automata based approach, using finite state automata to represent the set of constraints [LRT+14], the word based approach, that decomposes constraints using algebraic results such as Levi's lemma [BGZ17], and the unfolding based approach, which expresses each string variable as a bounded sequence of variables such as bit vectors [KGA+13], our abstraction can be categorized as mostly word-based. To the best of our knowledge, no SMT solver is able to reason on the sortedness of word expressions. We refer the reader to [Ama21] for a comprehensive survey on string constraint solving. By comparison with these works, we provide an abstract domain interface on top of the sequence operation, which allows its use in a static analysis tool, following an instance of reduced product [CC79].

5 Chapter

Analyzing an instance of FreeRTOS

In this chapter, we present the verification effort of the task scheduler picked from an instance of the FreeRTOS real-time operating system. First, we explain the general principles of the FreeRTOS scheduler. Then, we describe the specification of the global invariants of the scheduler used by the instance and the functions of the instance, and we report the results of the verification of these function by the analysis.

FreeRTOS is a small real-time operating system aimed at embedded applications supporting multitasking and implementing preemptive scheduling policies. The core of FreeRTOS, *i.e.* the `FreeRTOS-Kernel` repository, consists of four elements, each one in a specific source file.

Listing 5.1: Definition of tasks in FreeRTOS (simplified)

```
typedef struct tskTaskControlBlock
{
  volatile StackType_t * pxTopOfStack;
  /**< Points to the location of the last item placed on the tasks stack. */
  ListItem_t xStateListItem;
  /**< The list that the state list item of a task is reference from
       denotes the state of that task (Ready, Blocked, Suspended). */
  ListItem_t xEventListItem;
  /**< Used to reference a task from an event list. */
  UBaseType_t uxPriority;
  /**< The priority of the task. */
  StackType_t * pxStack;
  /**< Points to the start of the stack. */
  char pcTaskName[ configMAX_TASK_NAME_LEN ];
  /**< Descriptive name given to the task when created.  Facilitates debugging only. */
} tskTCB;
```

- `tasks.c` contains the FreeRTOS scheduler.
- `queue.c` provides functions for communication and synchronization between tasks.
- `stream_buffer.c` provides utilities functions for communication between two tasks.
- `timers.c` allows the execution of functions periodically.

Additionally, FreeRTOS provides a list library (analyzed in Section 4.6), memory management libraries for dynamic memory allocation, and architecture specific low-level functions. Overall, the core of FreeRTOS consists in approximately 10k LoC, half of which corresponds to the scheduler.

An interesting feature of FreeRTOS lies in its customizability. Each instance of FreeRTOS is configured with up to 196 parameters. These parameters are declared as `C` macros in a separate header file. They enable or disable features such as timers and queues. They also declare global parameters specific to the instance such as the size of the stack for each task.

Remark 5.1: Support for multi-processing

In its original design, FreeRTOS did not support tasks running in parallel. In 2017, the FreeRTOS development team introduced *asymmetric multiprocessing* (AMP). This allows the user to run one instance of FreeRTOS on each core of the device. Tasks running on different cores can still communicate using stream buffers. However, this extension has no impact on the scheduler, since from its point of view, it only has to pick a single running task. In December 2023, FreeRTOS gained support for *symmetric multiprocessing* (SMP). In this extension, one instance of FreeRTOS manages several cores. Consequently, the scheduler may pick several tasks to be running for work conservation. This feature also introduced new parameters. For instance, SMP adds to each task an *affinity* parameter to forbids execution of this task on some core. This extension is out of scope of our work. In the following, we assume that the instance runs without any form of multiprocessing.

5.1 Overview of the FreeRTOS scheduler

This section gives a brief description of the FreeRTOS scheduler as well as the actual instance we seek to verify. We refer the reader to [BT24] for more information on the scheduler.

The best way to understand the behavior of the FreeRTOS scheduler is to consider the attributes of tasks manipulated in the implementation. Listing 5.1 shows a simplified definition used in FreeRTOS for Task Control Blocks (TCB). This type possesses fields that are necessary for tasks context switch such as stack related parameters. These are not relevant for the scheduling policies.

5.1.1 Tasks states

The first field of the TCB used by the scheduler is `xStateListItem`. It is used to denote the state of the task. That is to say, the state of the task is expressed by a list membership. At a high-level, tasks in a FreeRTOS instance can be in four different states:

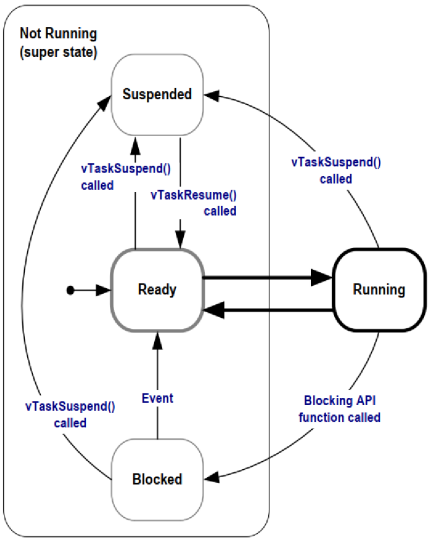


Figure 5.1: FreeRTOS tasks state transition diagram from [BT24]

- A task is in the *ready* state if the scheduler can pick it to be the next task to be run.
- When a task is not available to the scheduler, it is in the *suspended state*. Transitions from and to the delayed state are performed manually by calling corresponding functions.
- A task is in the *blocked* state when it is waiting either for some time to pass or for some event to happen. In the latter case, it may or may not specify a maximum waiting time. Additionally, the TCB is inserted in an event list using the `xEventListItem` field. A task with a maximum blocking time is called *delayed*. In every case, when the waiting time is elapsed or if the event happened the scheduler automatically sets the task back to the ready state.
- A task is in the *running* state if it is the one selected by the scheduler to be executed. The running task is not extracted from the ready list it belongs. It is simply marked using the `pxIndex` pointer in the header of the list.

Figure 5.1 presents the task state machine of the FreeRTOS scheduler.

5.1.2 Priority

The second task attribute relevant to the scheduler is its priority. The priority level of a task is a number between `0` and `configMAX_PRIORITIES-1`, stored in the `uxPriority` field of the TCB. To each priority level, corresponds a list that stores all ready tasks from this priority level. The various headers of these lists are stored in an array of headers, where the array index is equal to the level priority of the tasks stored in the list.

5.1.3 Scheduling policies

The scheduling policies of FreeRTOS depend on two parameters of the instance. These parameters impact the decision of the scheduler to select a new running task to execute.

Selecting the next running task The selection of the next running task is independent of the scheduling policy. The scheduler always selects the next task with the highest priority among all ready tasks. This task may be the running task if it is the only one with the highest priority to be in the ready state. The selection follows a *Round-Robin* scheme among the list of ready tasks corresponding to the highest priority.

Preemptive scheduling The first parameter that impacts the moment a context switch happens is `configUSE_PREEMPTION`. When this parameter is used, the scheduler runs in preemptive mode. It may interrupt the execution of the running task to select another one. Not using this parameter sets

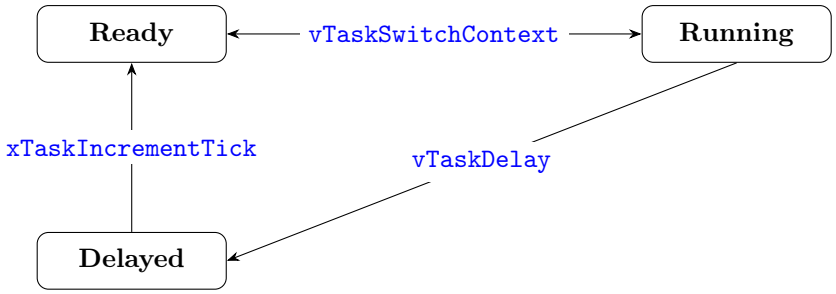


Figure 5.2: State transition diagram of tasks in the analyzed instance

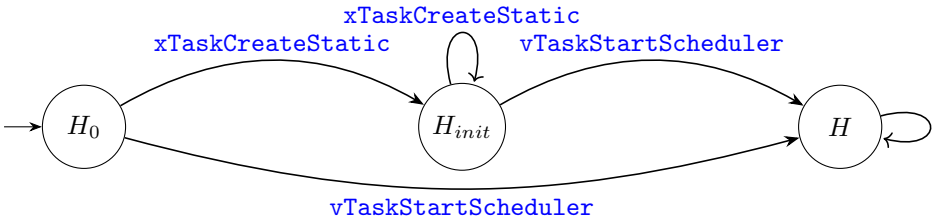


Figure 5.3: State diagram of the initialization of FreeRTOS scheduler

the scheduler in cooperative mode. No context switch is performed until the running task yields, even if there exists another task in the ready state with a priority higher than the priority of the running task.

Time slicing The second parameter is the `configUSE_TIME_SLICING` flag. Using this parameter tells the scheduler to perform a context switch when there is another ready task with the same priority level than the ready task. Otherwise, a context switch happens only if the scheduler detects a ready task with a higher priority.

5.1.4 Overview of the instance

To conclude this section, let us describe the actual FreeRTOS instance we seek to verify. Our goal here, is to establish real-time constraints that is to say, constraints of the following form:

> If the scheduler is running, then no delayed task should exceed its waiting time.

As a result, we do not account for the suspended state. We also exclude events, as this instance of FreeRTOS does not handle any data structures involving events such as queues. Consequently, the blocked state is limited to the delayed sub-state, meaning we do not consider tasks that are indefinitely blocked since they have no way to exit this state. To simplify the ready state, we assume a single priority level

Additionally, to avoid dealing with concurrency issues, we assume all interruptions are suspended except for the tick interrupt, which is managed by a function within the scheduler. Specifically, the tick interrupt service routine is `xTaskIncrementTick`.

Lastly, the selected scheduling policy is preemptive with time-slicing.

We discuss in Section 5.7 the cost of adding new features to this instance of the scheduler. Figure 5.2 presents the simplified state transition diagram of the instance.

5.2 Method overview

We now describe how the analysis presented in the previous chapter is used to verify an instance of the FreeRTOS scheduler.

Our verification aims to verify that no well-formed call to functions of the scheduler causes a run-time error, or violate the assertions in the code of FreeRTOS. We also seek to establish that all well-formed calls preserve the scheduler invariant, and have the specified effect on the state of the scheduler.

5.2.1 Model of the application

Our method follows the same principle as the life-cycle of an application summarized in Figure 5.3. It starts from the state after the boot of the program: all static allocations are assumed to be performed, but their content is not initialized. This corresponds to the state H_0 in the Figure 5.3. Then, the `main` function is executed. This function is provided by the user to declare the tasks using the function `xTaskCreateStatic` as well as other elements. Adding one task partially initializes the data structures used by the scheduler. Therefore, after one call to the `xTaskCreateStatic` function, the scheduler is in the H_{init} state. All other calls to `xTaskCreateStatic` let the scheduler in the H_{init} state. When the setup is ready, the user concludes this function by calling the function `vTaskStartScheduler` to give control over the FreeRTOS kernel. This function creates the *idle task* to ensure that there is always a task in the running state and starts the scheduler by running the architecture dependent instruction. From this point onward, the scheduler is in the H state. All calls to other functions are loops on this state.

Note that the states presented in Figure 5.3 denote a set of memory states defined by some parameters. For instance, H denote all the possible states that satisfy the invariants of the running scheduler. These states will be presented in the next section.

5.2.2 Application agnostic

Though our verification effort targets a specific instance of FreeRTOS, *i.e.* a given set of parameters of the scheduler, our method is not bound to a specific application. This means that we make no assumption on the number of tasks declared by the user, as well as their actual code except that all arguments of scheduler function calls are well-formed *i.e.* they satisfy the specified precondition of the function.

5.2.3 Intraprocedural analysis

When the analysis encounters a function call, then the body of the function is analyzed using the current abstract state. We do not leverage the possible contracts of the functions to compute directly the abstract state with the post-condition as after the function returns. Indeed, this would require writing new contracts for each function call to ensure that the final state is strong enough.

Another reason justifying this choice is that, in some cases, the abstract state does not satisfy the pre-condition since the call may temporarily break the scheduler invariant. For example, in `vTaskDelay`, when the function `xTaskResumeAll` is called, the pointer `pxCurrentTCB` does not point to a task in the list of ready tasks but to a task in the list of delayed tasks, since it is corresponds to the running task before the system call to `vTaskDelay`.

5.3 Specification of the states of the scheduler

The section presents the specification of the internal states of the scheduler. First, we introduce the main state of the scheduler, H, *i.e.* the state describing a *running* scheduler. This presentation includes the variables and the data structures manipulated by the scheduler as well as the invariants between these components. Then, we describe the other states of the scheduler, *i.e.* the states H_0 and H_{init}.

Although specifications of FreeRTOS have already been proposed in [FHQ12, CWD15, Haw17], our specification is based on [dH21]. We adapted it to the verified instance since some elements of the scheduler are not used in this instance. Additionally, the specification language of MemCAD [RBLL24] is designed to represent abstract values. Therefore, it is not fit to express complex set-based constraints such as set comprehension equalities.

5.3.1 Lists of tasks

As stated above, all tasks are inserted in a list according to their state. Consequently, the scheduler manipulates a single type of data structure: lists of tasks. The specification of a list of tasks follows the same approach as the one presented in Remark 4.4. That is to say, we use an inductive segment predicate pointing to the header of the list in order to summarize the nodes of the list. Figure 5.4 presents the definition of the **task** inductive predicate. In essence, this predicate is a modified version of the predicate **FreeRTOSNode** introduced in Figure 4.32b to include other fields related to TCB.

$$
\begin{aligned}
\alpha.\mathbf{task}(\pi, \kappa, S_a, S_v) \;:=\; & \exists \beta_t, \ldots, S'_a, S'_v, \alpha.\texttt{pxTopOfStack} \mapsto \beta_t \\
& * \alpha.\texttt{xStateListItem.xItemValue} \mapsto \beta_v \\
& * \alpha.\texttt{xStateListItem.pxNext} \mapsto \beta_n.\texttt{xStateListItem} \\
& * \alpha.\texttt{xStateListItem.pxPrevious} \mapsto \pi.\texttt{xStateListItem} \\
& * \alpha.\texttt{xStateListItem.pvOwner} \mapsto \alpha \\
& * \alpha.\texttt{xStateListItem.pxContainer} \mapsto \kappa \\
& * \alpha.\texttt{xEventListItem.xItemValue} \mapsto \beta_0 \\
& * \alpha.\texttt{xEventListItem.pxNext} \mapsto \beta_0 \\
& * \alpha.\texttt{xEventListItem.pxPrevious} \mapsto \beta_0 \\
& * \alpha.\texttt{xEventListItem.pvOwner} \mapsto \alpha \\
& * \alpha.\texttt{xEventListItem.pxContainer} \mapsto \beta_0 \\
& * \alpha.\texttt{uxPriority} \mapsto \beta_p \\
& * \alpha.\texttt{pxStack} \mapsto \beta_s \\
& * \alpha.\texttt{pcTaskName} \mapsto \beta_{name} \\
& * \beta_n.\mathbf{task}(\alpha, \kappa, S'_a, S'_v) \\
& \wedge \alpha \neq 0 \\
& \wedge \beta_p = \beta_0 = 0 \\
& \wedge S_a = [\alpha].S'_a \\
& \wedge S_v = [\beta_v].S'_v
\end{aligned}
$$

Figure 5.4: Definition of the **task** inductive predicate

The tasks state list field `xStateListItem` is used for the list structure. For instance, the field `xItemValue` forms the local part of the sequence of values S_v, and the `pxNext` is the source of the recursive instance. Since `xStateListItem` is not the first field of the Task Control Block (TCB) structure, we assert that the previous and next pointers point to the list nodes at the offset corresponding to the position of `xStateListItem` in the TCB. The other list field used for events is simply not used. All related fields are set to 0 or α for the `pvOwner` field. Additionally, the value of the `uxPriority` field is set to be equal to 0 since it is the only priority level available in the instance. Finally, the stack pointers as well as the `pcTaskName` fields are set to existentially quantified values since they are not relevant to the scheduler.

5.3.2 Ready part

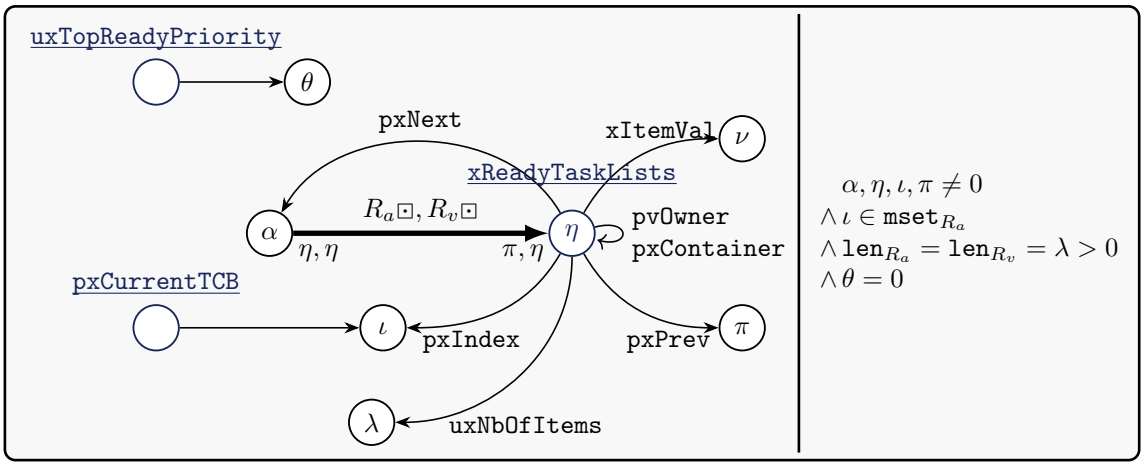
The ready part of the scheduler invariant expresses the list of ready tasks (there is only one since there is a single priority level), as well as the value of variables whose values depend on this list. Figure 5.5a presents this part of the scheduler.

The variable `xReadyTaskLists` stores the header of the list of ready tasks. This list contains all tasks in the ready state, one of which is pointed by the `pxIndex` field of the header as well as the `pxCurrentTCB` variable. This task corresponds to the running task. Consequently, the list of ready tasks is not empty. Note that the sequence of values stored in the list, R_v, is not constrained except the constraint stating that it has the same length as R_a. For instance, the ready part does not assert that R_v is sorted nor that the value in the header is an upper bound of R_v. Finally, the last part of the ready part is the global variable `uxTopReadyPriority` with value 0.

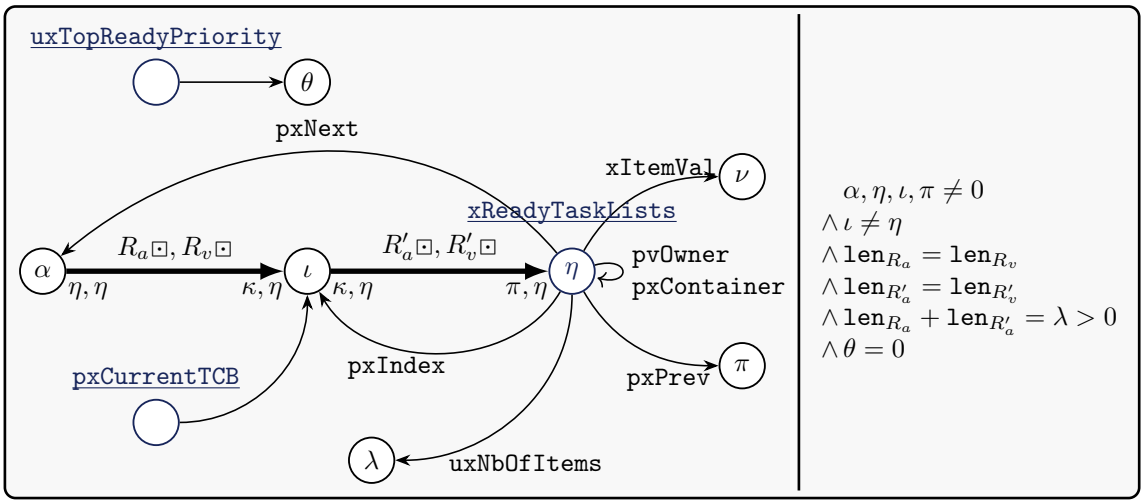
To sum up, the ready part of the scheduler can be expressed as a formula[1] with three parameters $\mathbf{Ready}(R_a, R_v, \iota)$.

Extended ready part For some preconditions, we employ a different formula presented in Figure 5.5b. In this formula, we make explicit the currently running task by splitting the segment of tasks in two at the position of the task pointed by `pxCurrentTCB`. Consequently, the content of the list of tasks in the ready state is described by four sequence variables. The variables R_a and R_v correspond to the content of the list between the header and the running task (excluded), while R'_a and R'_v denote the content after the running task. Additionally, to ensure that the segment from ι

[1] This predicate does not correspond to an inductive predicate as expressed in the last chapter but rather to a formula in the specification language of MemCAD (see [RBLL24] for more information on this language). Variable appearing in the formula that are not a parameter of the formula are implicitly existentially quantified.



(a) Standard specification



(b) Strong specification

Figure 5.5: Ready part of the FreeRTOS scheduler

to the header is not empty, we add the numerical disequality $\eta \neq \iota$, and we modify the constraints between λ and the lengths of the sequence parameters. This extended specification of the ready part is summarized in a formula with five parameters $\mathbf{Ready}_e(R_a, R_v, R'_a, R'_v, \iota)$.

The reason why we need to employ an extended specification for some functions is that the non-local unfolding is not sufficient to reason about the sequence of values. As shown in Example 4.14, head parameters are exactly matched in the non-local unfolding *i.e.* if the abstract state has a constraint $R_a = R'_a.[\iota].R''_a$, then the analysis is able to infer that the parameters of the segments to and from ι are equal to R'_a and R''_a, respectively. However, it is impossible to use a similar argument to split the sequence of values. As a consequence, we use this extended version of the ready part for functions calls that modify the sequence of values stored in the list of ready tasks.

The extended specification is equivalent to the normal one, in the sense that if we use the correct sequence parameters, both formulas express the same set of memory states. However, our analysis can only prove the following inclusion:

$$\begin{array}{c}\mathbf{Ready}_e(R'_a, R''_a, R'_v, R''_v, \iota) \\ \wedge\, R_a = R'_a.R''_a \wedge R_v = R'_v.R''_v\end{array} \sqsubseteq^{\sharp}_{\mathbb{S}} \mathbf{Ready}(R_a, R_a, \iota)$$

5.3.3 Delayed part

The second part of the scheduler is the delayed part. It contains a list whose header is pointed by variable `pxDelayedTaskList`. The list of delayed tasks is sorted according to the value of field

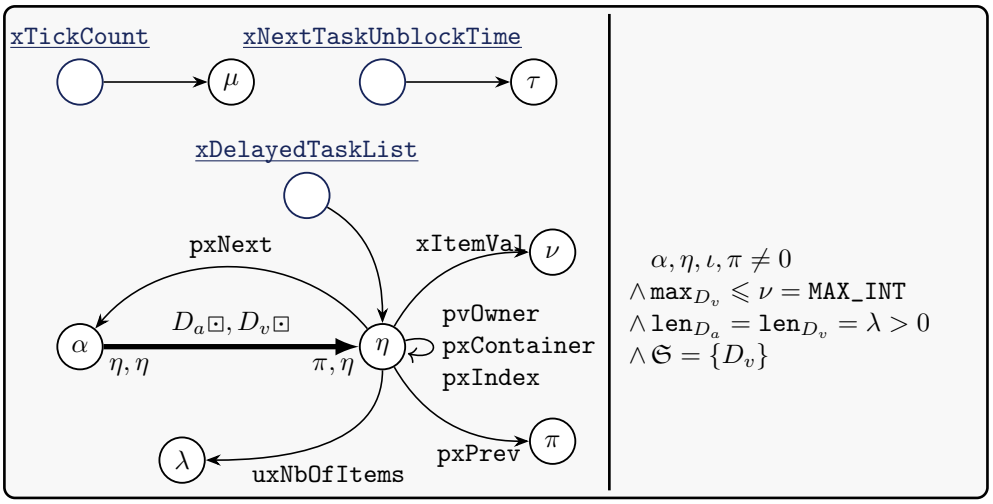


Figure 5.6: Delayed part of the FreeRTOS scheduler

`xStateListItem.xItemValue`. This field corresponds to the moment the task must be unblocked by the scheduler and put back to the list of ready tasks. To avoid repeated list lookups to check if a task exceeded its delay, this value of the first node is stored inside a global variable `xNextTaskUnblockTime`. When the list of delayed tasks is empty, this variable is set to the value stored in the header which corresponds to `MAX_INT`. Such a disjunction cannot be expressed in the specification language. Therefore, we do not express it here. We leave it to the function goals (see more below). However, there is a constraint between the value of `xNextTaskUnblockTime` and the value of the variable storing the value of tick `xTickCount`. The latter must always be smaller than the former. Together with the constraint between the minimum element of D_v and `xNextTaskUnblockTime`, the delayed part expresses that the value of the current tick cannot exceed the unblocking time of tasks in the delayed part. Note that since the delayed list never uses its `pxIndex` field we express no constraint on its value. Figure 5.6 presents the delayed part of the scheduler.

The variable `xTickCount` that stores the value of the tick is an unsigned integer. It can overflow if the instance is run long enough[2]. To address such tick overflow, the delayed part uses two lists, `xDelayedTaskList` already presented, as well as `xOverflowDelayedTaskList` to store tasks that should be unblocked after an overflow of `xTickCount`. Since the numerical domains used in the MemCAD analyzer do not support machine integers (but simply unbounded mathematical integers), we do not analyze the part of the scheduler that handles integer overflow of the tick counter, and we omit in this specification the `xOverflowDelayedTaskList` list.

To conclude, the delayed part of the scheduler can be summarized in a formula with four parameters **Delayed**(D_a, D_v, μ, τ).

5.3.4 Merging the parts

The global state of the scheduler can be expressed as a combination of the ready and delayed parts with additional constraints on other elements of the scheduler that do no fit in either parts. These additional elements are obtained by the following variables:

- `uxSchedulerSuspended` is an integer variable expressing when the scheduler is running. Since the scheduler allows nested suspensions, this variable is incremented when the scheduler is suspended and decremented when resumed. Some features of the scheduler are available only when the scheduler is fully resumed, *i.e.* when `uxSchedulerSuspended` is equal to 0. Its value is expressed by the symbolic variable γ.

- `xPendedTicks` is a variable that accumulates tick counts that possibly occurred when the scheduler was suspended. Its value is expressed by the symbolic variable μ_γ.

- `xYieldPending` is the boolean flag used by the scheduler to tell when a context switch should

[2]For example, with a tick interrupt each millisecond (the default value in all FreeRTOS example applications), this overflow occurs after approximately 50 days on a 32-bit architecture.

Part	LoS
H	108
task predicate	28
Ready part	14
Delayed part	24
Environment	32
Others	10
H_s	32
$\mathbf{Ready}_s$	21
Others	11
H_0	29
H_{init}	22
Total	191

Table 5.1: Size of the specification of the internal states of FreeRTOS

have taken place, but could not since the scheduler is suspended. It is denoted by the symbolic variable τ_γ

- `uxCurrentNumberOfTasks` stores the total amount of tasks handled by the scheduler. Its value is denoted by the symbolic variable θ. By definition, there is a relation between its value and the number of tasks in the ready and delayed parts. This corresponds to the constraint: $\theta = \texttt{len}_{R_a} + \texttt{len}_{D_a}$.

To conclude, the main state of the scheduler can be fully specified by a formula with only ten parameters $H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma, \tau_\gamma)$, and the state obtained by using the extended version of the ready part can be fully specified with twelve parameters $H_e(R_a, R_v, R'_a, R'_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma, \tau_\gamma)$.

5.3.5 Initialization states

We now present the states of the scheduler during its initialization.

Uninitialized state As stated earlier, the uninitialized state H_0 correspond to the state of the scheduler at the boot of the application. The values of all global variables as well as the fields of the list headers are set to some unconstrained symbolic variables.

Partially initialized state The second state, noted H_{init}, corresponds to the state of the scheduler H where the list headers have been initialized. The only differences between H and H_{init} are the following:

- The field `pxIndex` of the header of the ready list still points to itself and not to some task in the list.
- Since all new tasks are set to the ready state, the delayed list is empty.
- The variables `xTickCount` and `xNextTaskUnblockTime` are not yet initialized.
- The variable `xTaskSchedulerRunning` is set to 0.

To conclude this section, let us discuss the cost of specifying the scheduler states. Table 5.1 presents the number of lines of specification (LoS) required to write the different parts of the specification of the scheduler states. Since our specification language allows us to share formulas (similarly to the ACSL [BCF+24] and Z [2202] specification languages), we only count for other scheduler states, the number of lines that were not already counted in H. For instance, the extended precondition H_e reuses all components of the H, except the ready part. Writing the strong version of this part requires 21 lines, and writing other constraints specific to this formula amounts to 11 other lines. Consequently, the strong precondition formula amounts to 32 lines of specification.

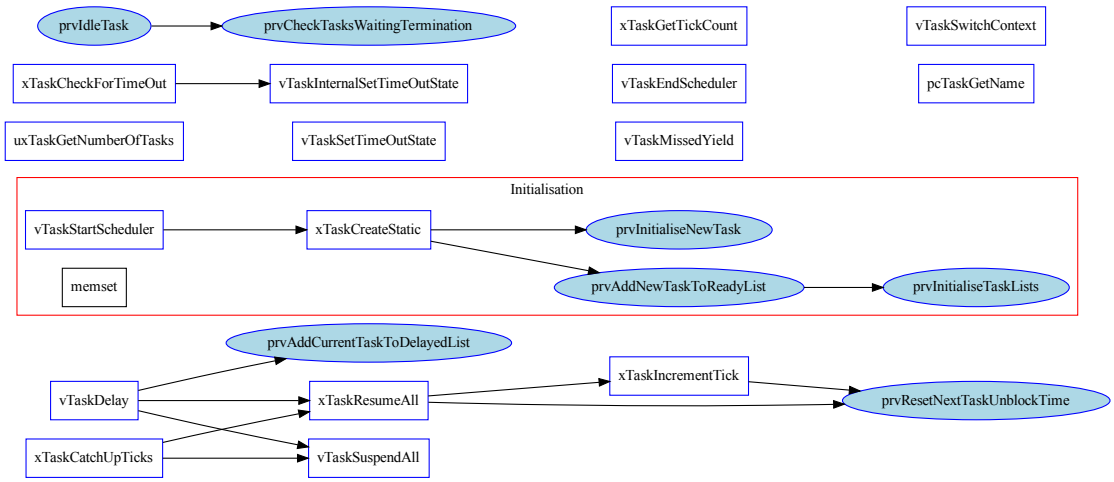


Figure 5.7: Call graph of the FreeRTOS instance

5.4 Specification of the functions

This section presents the specification of the verified instance scheduler. Figure 5.7 shows the call graph of the FreeRTOS scheduler. Functions in oval boxes are private. These functions are meant for internal use by the scheduler and should not be called from outside the scheduler. Indeed, they may break temporary break invariants of the scheduler and this is up to the caller to restore these invariants.

5.4.1 General form of goals

Both pre- and post-conditions of functions are written as the combination of some scheduler state H, H_{init}, or H_0 instantiated with specific parameters to denote the current state of the scheduler, and a goal specific formula to express constraints on function parameters and returned value. For instance, the pre- and post-conditions of `uxTaskGetNumberOfTasks` are expressed by:

pre-condition	post-condition
$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma)$	$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma)$ $\wedge\, \texttt{res} = \texttt{len}_{R_a} + \texttt{len}_{D_a}$

In the post-condition, the variable `res` in the numerical constraints stands for the value returned by the function. When the function has a parameter or a returned value that is a simple numerical value, then we omit it in the shape part of the abstract value, and we employ the name of the variable directly in the numerical constraint.

Using the $H_\bullet$ formula ensures that the functions maintain the invariants of the schedulers. This includes the structural invariants of the lists as well as constraints between distinct parts of the scheduler.

5.4.2 Using several goals

Each function may have several goals since the function has behaviors that cannot be fully encompassed in single goal due to a lack of expressiveness of the specification language and behind it the abstract domain. For instance let us consider the specification of `pcTaskGetName`. This function inputs a pointer to a TCB and returns a pointer to the `pcTaskName` field of the TCB pointed by the input. When the input is equal to the null pointer, then the function returns the address of the name field of the TCB pointed by `pxCurrentTCB`. It is impossible to represent this disjunction of behaviors in a single goal since the link between the input and output cannot be expressed in the abstract domain of the analysis. Consequently, the specification of `pcTaskGetName` is formed by the disjunction of the two following goals:

Pre-condition	Post-condition
$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma)$ $\wedge$ `xTaskHandle` $= 0$	$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma)$ $\wedge$ `res` $= \iota + \varphi_{\mathbb{F}}$(`pcTaskName`)
$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma)$ $\wedge$ `xTaskHandle` $\neq 0$	$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma)$ $\wedge$ `res` $=$ `xTaskHandle` $+ \varphi_{\mathbb{F}}$(`pcTaskName`)

5.4.3 Specification of `xTaskIncrementTick`

To illustrate how we specified the functions of FreeRTOS, let us present the specification of the function that forms the core of the scheduler: `xTaskIncrementTick`. Table 5.2 presents the six goals of this function.

This first one corresponds to the case where the tick interruption happens when the scheduler is suspended. This suspension is expressed by the constraint $\gamma > 0$. In this case, the function boils down to incrementing the value of `xPendedTicks` (denoted by the symbolic variable μ_γ) and returning 0. In all remaining cases, the scheduler is assumed to be running. So they all contain the constraints $\gamma = 0$ and $\mu_\gamma = 0$.

The second, third, and fourth case corresponds to the possibility where there is no delayed task that expire. This is expressed by the constraint stating that the value of `xNextTaskUnblockTime` is greater than the value of the incremented tick: $\tau > \mu + 1$. The three cases differ by their result. In the goal b, there is a single task in the ready state. Therefore, no context switch is required and the function returns 0. In the other goals, there is more than two tasks in the ready state or the scheduler detected that a context switch could not take place before since the scheduler was suspended. So the function returns 1 in order to express that a context switch should happen.

The last two goals correspond to the cases where the waiting times of tasks in the delayed state expired. In goal d, the waiting time of all delayed tasks expired. Therefore, in the post-condition, the sequence of delayed tasks is empty and the value of `xNextTaskUnblockTime` is equal to `MAX_INT`. Finally, in the last goal, the sequence of values in the list of delayed task can be split in two. There exists a non-empty suffix of this sequence, written D_v^{exp}, whose values of elements are lower than the incremented tick. Additionally, the remaining of the sequence, denoted by the sequence variable D_v^{rem}, is non-empty and starts with some element ν. In the post-condition, the tasks whose waiting time have expired are put back in the ready state, and the delayed list contains the sequence of tasks that have not yet expired. Additionally, the value of `xNextTaskUnblockTime` is set to the first element in the sequence of value remaining in the list of delayed task: ν. The two goals use the extended pre-condition H_e since the sequences of values stored in the list of ready tasks are modified in these goals. The sequence of tasks stored by the ready list in the post-condition is obtained by inserting the sequence of expired tasks between the two sequences of ready tasks in the pre-condition.

5.4.4 Cost of the specification of functions

To conclude this section, let us discuss the cost of the specification of the function. The size of the specification of each goal is presented in Table 5.3.

For about one-third of the goals, the specification requires fewer than 10 lines. These goals typically involve two instantiations of the schedulers and the declaration of function arguments, along with some constraints between these variables and the parameters of the scheduler instances. An example of such a goal is the one used to verify `vTaskSuspendAll`. This goal boils down to specifying that the variable `uxSchedulerSuspended` is incremented. Another third of the goals span between 10 and 22 lines of specifications. These goals either require a more complex memory layout for function arguments than a simple numerical variable or need to express a larger number of numerical or sequence constraints. Goals a, b, c, and d of the specification of `xTaskIncrementTick` belong to this group of goals. The final third involves goals that necessitate a substantial number of constraints, including sequence definitions and complex constraints involving sequence attributes. Goals e and f of the specification of `xTaskIncrementTick` fall into this category.

In total, the size of all specified goals amounts to 696 Lines of Specifications. If we add the lines of specifications used in the definition of scheduler states, the total specification of the scheduler takes 887 LoS. Considering that the scheduler take up to 824 Lines of Code, the ratio between the specification effort and the verified code is less than 1.1.

code	Pre-condition	Post-condition
a	$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \gamma > 0$	$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu'_\gamma, \tau_\gamma)$ $\wedge\, \mu'_\gamma = \mu_\gamma + 1$ $\wedge\, \texttt{res} = 0$
b	$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \gamma = 0 \wedge \mu_\gamma = 0$ $\wedge\, \texttt{len}_{R_a} = 1$ $\wedge\, \tau > \mu + 1$	$H(R_a, R_v, D_a, D_v, \iota, \mu', \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \mu' = \mu + 1$ $\wedge\, \texttt{res} = 0$
c	$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \gamma = 0 \wedge \mu_\gamma = 0$ $\wedge\, \texttt{len}_{R_a} > 1$ $\wedge\, \tau_\gamma > 0$ $\wedge\, \tau > \mu + 1$	$H(R_a, R_v, D_a, D_v, \iota, \mu', \tau, \gamma, \mu_\gamma, \tau'_\gamma)$ $\wedge\, \mu' = \mu + 1$ $\wedge\, \tau'_\gamma = 0$ $\wedge\, \texttt{res} = 1$
d	$H(R_a, R_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \gamma = 0 \wedge \mu_\gamma = 0$ $\wedge\, \texttt{len}_{R_a} > 1$ $\wedge\, \tau > \mu + 1$	$H(R_a, R_v, D_a, D_v, \iota, \mu', \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \mu' = \mu + 1$ $\wedge\, \texttt{res} = 1$
e	$H_e(R_a, R_v, R'_a, R'_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \gamma = 0 \wedge \mu_\gamma = 0$ $\wedge\, \tau = \texttt{min}_{D_a} = \texttt{max}_{D_v} = \mu + 1$ $\wedge\, \texttt{len}_{D_a} = \texttt{len}_{D_v} > 1$	$H(R^f_a, R^f_v, D^f_a, D^f_v, \iota, \mu', \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \mu' = \mu + 1$ $\wedge\, \texttt{res} = 1$ $\wedge\, D^f_a = D^f_v = \texttt{[]}$ $\wedge\, R^f_a = R_a.D_a.R'_a$ $\wedge\, R^f_v = R_v.D_v.R'_v$ $\wedge\, \tau = \texttt{MAX\_INT}$
f	$H_e(R_a, R_v, R'_a, R'_v, D_a, D_v, \iota, \mu, \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \gamma = 0 \wedge \mu_\gamma = 0$ $\wedge\, D_a = D^{exp}_a.D^{rem}_a$ $\wedge\, D_v = D^{exp}_v.D^{rem}_v \wedge D^{rem}_v = [\nu].D'_v$ $\wedge\, \texttt{len}_{D^{exp}_a} = \texttt{len}_{D^{exp}_v} > 1$ $\wedge\, \texttt{max}_{D^{exp}_a} \leqslant \tau = \mu + 1 < \texttt{min}_{D^{rem}_a}$	$H(R^f_a, R^f_v, D^{rem}_a, D^{rem}_v, \iota, \mu', \tau, \gamma, \mu_\gamma, \tau_\gamma)$ $\wedge\, \mu' = \mu + 1$ $\wedge\, \texttt{res} = 1$ $\wedge\, R^f_a = R_a.D^{exp}_a.R'_a$ $\wedge\, R^f_v = R_v.D^{exp}_v.R'_v$ $\wedge\, \tau = \nu$

Table 5.2: Specification of `xTaskIncrementTick`

Listing 5.2: Original code of `xTaskResumeAll` (simplified version)

```
1  if( xPendedTick > ( TickType_t ) 0U )
2  {
3    do
4    {
5      if( xTaskIncrementTick() != pdFALSE ) // increment xTickCount
6        xYieldPending = pdTRUE;
7      --xPendedTick ;
8    } while( xPendedTick > ( TickType_t ) 0U );
9  }
```

Listing 5.3: Modified code of `xTaskResumeAll` (simplified version)

```
1  if( xPendedTick > ( TickType_t ) 0U )
2  {
3    TickType_t xTotalTick = xTickCount + xPendedCounts;
4    do
5    {
6      if( xTaskIncrementTick() != pdFALSE ) // increment xTickCount
7        xYieldPending = pdTRUE;
8      --xPendedTick ;
9    } while( xTickCount != xTotalTick );
10 }
```

5.5 Analysis of the functions

This section discusses the analysis of the functions of the scheduler. First we present the work required on the source code of the instantiated scheduler to make the analysis work, then we show the result of our verification effort. Finally, we discuss the time performance of our analysis.

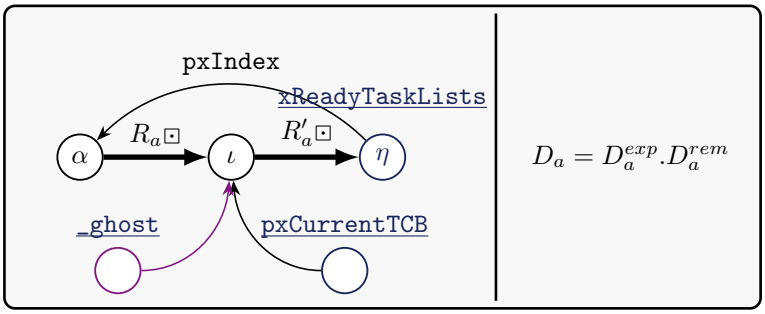
5.5.1 Modification of the source code

In order to prove the specified goals, we modified the analyzed program to guide the analysis. There are two kinds of modifications that were required to make the analysis succeed. The first modification aims to guide the reduction heuristic of the sequence domain. The other modification corresponds to *ghost code*. That is to say instructions that are added to guide the widening to obtain a loop invariant that is expressive enough to prove the post-condition.

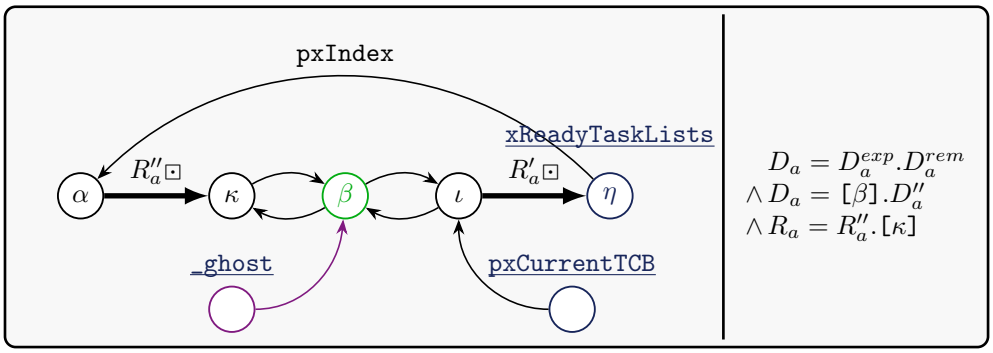
Logical transformation The only logical transformation (*i.e.* a modification that actually modifies the verified code) concerns the function `xTaskResumeAll`. As expressed in the previous section, when a tick interruption occurs while the scheduler is suspended (*i.e.* `uxschedulerSuspended` > 0), then the tick is accumulated in the variable `xPendedTicks` in order to be treated later, *i.e.* when the scheduler is resumed by `xTaskResumeAll`. In the original version of the code, presented in Listing 5.2, these ticks are treated in a loop that calls `xTaskIncrementTick` to increment `xTickCount` and that decrements `xPendedTick`. This loop is performed until `xPendedTick` become null. This means that the exit condition of the loop boils down to `xPendedTick` $= 0$. Such a constraint involves no symbolic variable that are related to bounds of sequence variables. Consequently, while treating this constraint, the sequence domain does not attempt to perform any reduction heuristic in the sequence part of the abstract value.

To address this issue, we modify the code as follows: we add a variable `xTotalTick` that is equal to the value expected at the end of iteration, and we change the loop condition as `xTotalTick` $\neq$ `xTickCount`. This constraint is able to trigger the reduction heuristic performed by the $\mathbf{guard}^\sharp_n$ operator. Listing 5.3 presents the loop with the modified condition. This modification does not alter the correctness of the program.

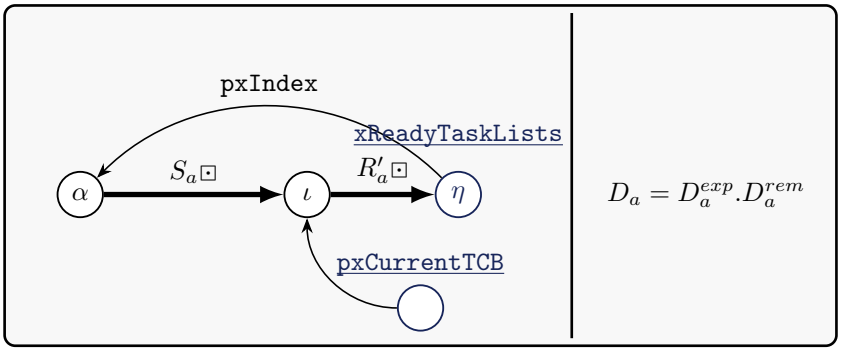
Ghost code The second kind of code modification is intended to guide the shape part of the abstract value to compute a loop invariant that is suitable to prove the goal. The most noticeable example of this technique concerns the `xTaskIncrementTick` function. In this function, tasks whose delays have expired are inserted back in the ready state by a loop. This insertion is performed before the task pointed by the `pxIndex` field of the ready tasks list. Figures 5.8a and 5.8b presents



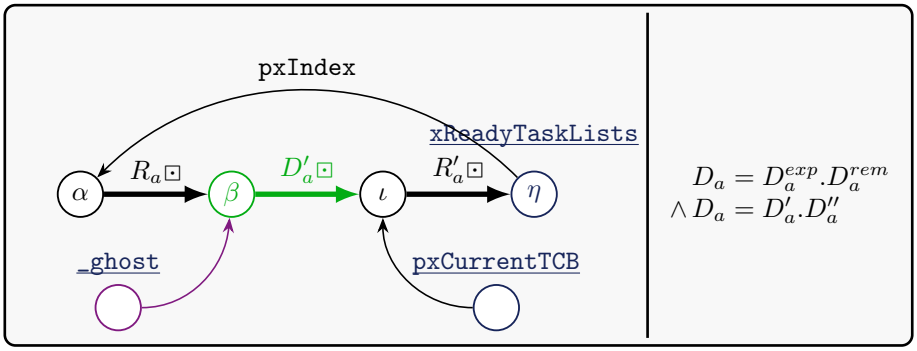
(a) Abstract state before the insertion of the first element



(b) Abstract state after the insert of the first expired task



(c) Loop invariant computed without ghost pointers



(d) Loop invariant computed with ghost pointers

Figure 5.8: Example of ghost pointer usage

the abstract states computed by the analysis, respectively before and after the insertion of the first expired task. For simplicity, we display only the ready part of the abstract state, and we focus solely on sequences of addresses. After performing the first widening, the analysis computes the abstract state shown in Figure 5.8c. In this state, the newly inserted task is merged with the segment of tasks before ι into a single segment whose sequence of address is denoted by S_a. With this shape part, it is impossible for the sequence domain to express that S_a is the concatenation of R_a with the sequence of tasks that were inserted in the ready list. Indeed, the latter is not denoted by a sequence variable occurring in the shape part of the abstract value.

To address this issue, we insert in the code a ghost pointer, that initially points to ι, and that is assigned to points to the first expired task. This ghost pointer guides the upper bound operator to insert a segment from itself to the currently running task. The resulting abstract state is depicted in Figure 5.8d. Thanks to this hint, the sequence part is now able to express that the sequence of addresses stored in this segment D'_a appended with the sequence of addresses still in the delayed list D''_a forms the whole sequence of addresses that were originally in delayed list.

Overall, we inserted 41 new pieces of code in order to guide the analysis.

5.5.2 Verification results

Table 5.3 presents the result of the verification of the scheduler functions as well as time and memory consumption of the goals. All goals, except two, are successfully proved by the analysis. We explain below what our analysis lacks in order to prove them. Nevertheless, the functions involved in the initialization of the scheduler (`vTaskStartScheduler` and `xTaskCreateStatic`) are fully verified as well as two out of three functions presented in the task state transition diagram (`xTaskIncrementTick` and `vTaskSwitchContext`).

Though our verification effort may not seem successful, since we did not fully verify all functions, the results obtained so far are encouraging. Indeed, this experiments suggest that our approach is able to fully prove the partial functional correctness of complex functions from a real-world task scheduler. For instance, the invariants inferred by our analysis for the goal b of the `xTaskCatchUpTicks` involve subtle numerical constraints between the various bound variables and the `xTickCount` variable, and the reductions performed let the analysis derive precise sequence constraints.

The missing lines Function `vTaskDelay` put the running task in the delayed state. This insertion in a sorted list requires to scan the list to determine the point of insertion of the running task. Since this may take an unbounded amount of time, the scheduler is suspended during the insertion. Once the insertion is performed, the scheduler resumes. Consequently, the scheduler must apply the ticks that accumulated during the insertion by repeatedly calling `xTaskIncrementTick`.

In the goals a and b, no task in the delayed state has had its delay expire. Therefore, the behavior of the function boils down to performing an insertion in a sorted list. Goals c and d address the situation where at least one task in the delayed state has had its waiting time expired. These goals differ in whether the running task has also reached the end of its waiting time.

In goal c, we assume that the running task did not expire its waiting time. To prove this goal, the analysis must detect that the prefix of expired tasks is put back in the ready state. However, due to a loss of precision in the numerical part of the analysis, the goal cannot be proved. Nevertheless, it successfully established that the function is free of run-time error and preserves the structural invariant of the scheduler.

Goal d pertains to the situation where the delay of the running task expires during its insertion. The loop invariant is a disjunction of two states: either there have not been enough ticks for the delay of the running task to expire, or the resuming procedure has applied enough ticks. In the first state, the task remains in the delayed list, while in the second state, it was moved to the ready list. To verify this, the widening operator must increase the number of disjunctions during the repeated iterations to compute the invariant.

5.5.3 Performance of the analysis

Regarding the time and memory consumption, we observe two behaviors. Simple goals, *i.e.* goals that do not require loop analysis, are proved in less than a second and the memory consumption is limited. Goals that involve the analysis of loops and the computation of invariant using widening require a longer amount of time to perform the analysis. In order to understand this explosion, we conducted

Function name	Code	LoS	Property verified	time (s) all	time (s) num	memory usage (MB)
`xTaskCreateStatic`	a	24	✓	0.65	0.01	29.77
	b	26	✓	0.64	0.01	30.19
`vTaskStartScheduler`	a	19	✓	0.66	0.01	30.03
	b	19	✓	0.67	0.01	30.23
`vTaskSuspendAll`	a	5	✓	0.62	0.00	28.10
`vTaskMissedYield`	a	5	✓	0.62	0.00	28.13
`xTaskGetTickCount`	a	7	✓	0.61	0.00	28.36
`uxTaskGetNumberOfTasks`	a	7	✓	0.63	0.00	28.19
`vTaskSetTimeOutState`	a	12	✓	0.62	0.00	28.19
`vTaskInternalSetTimeOutState`	a	12	✓	0.82	0.00	28.29
`pcTaskGetName`	a	11	✓	0.62	0.00	28.62
	b	11	✓	0.63	0.00	28.59
`xTaskCheckForTimeOut`	a	24	✓	0.63	0.01	28.36
	b	22	✓	0.65	0.01	28.59
	c	22	✓	0.63	0.00	28.29
`xTaskIncrementTick`	a	10	✓	0.82	0.03	29.55
	b	17	✓	0.75	0.06	30.10
	c	14	✓	0.73	0.04	30.42
	d	14	✓	0.74	0.04	30.06
	e	26	✓	36.48	33.39	68.10
	f	24	✓	21.80	19.69	42.35
`vTaskSwitchContext`	a	6	✓	0.63	0.00	28.11
	b	14	✓	0.76	0.08	29.37
	c	15	✓	0.79	0.09	29.73
	d	9	✓	0.72	0.05	29.28
`xTaskResumeAll`	a	36	✓	178.05	163.72	203.81
	b	34	✓	316.83	284.04	298.86
	c	9	✓	0.69	0.01	31.93
	d	25	✓	2.36	1.26	34.39
	e	26	✓	1.85	0.91	36.09
`xTaskCatchUpTicks`	a	26	✓	214.09	197.00	204.54
	b	28	✓	463.48	410.65	384.84
	c	17	✓	1.55	0.73	36.45
	d	18	✓	1.62	0.78	36.29
`vTaskDelay`	a	31	✓	14.51	12.94	35.48
	b	31	✓	21.31	19.44	37.96
	c	40	**PrSafe Fc**	1363.15	1235.54	835.74
	d	42	✗			

Table 5.3: Verification results of FreeRTOS functions

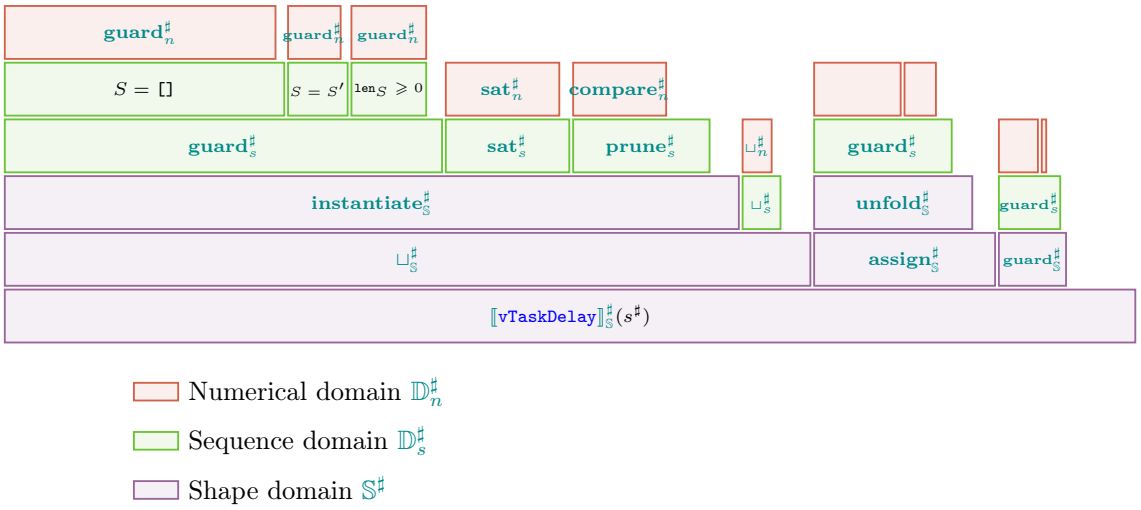


Figure 5.9: Time consumption of the operators in goal c of `vTaskDelay`

Listing 5.4: Code snippet used to merge disjunctions

```
1  if( cond ){ /* merge (⋁_i s_i^♯) */ } else { /* merge (⋁_j s_j^♯) */ }
```

some profiling investigation. Figure 5.9 depicts the result of the time spent by each of the operators during the analysis of goal c of `vTaskDelay`. Stacked layers represent nested calls between operators, and the lowest rectangle $[\![\texttt{vTaskDelay}]\!]^\sharp_{\mathbb{S}}(s^\sharp)$ represents the whole analysis. The width of each rectangle is proportional to the time taken by the operator. For the sake of readability, we displayed only the major operators, and we merged the join with the widening since they have a similar working. We found several factors that explain such explosion in the time used by the analysis.

Cost of the disjunction As explained in Section 2.3.4, our analysis relies on disjunctions to gain precision. Recall that the analysis manipulates a disjunctive abstract state that is formed by several elements of the shape abstract domain. Disjunctions are introduced by either the conditional statements or by the unfolding of inductive predicates. Once a disjunction is introduced it is kept until the analysis reaches a widening point or a directive that forces the merge of the sub-states in the disjunctions. Since the operations on doubly linked list, such as deletion or addition, involve the modification of both neighbors, they require one forward unfolding and one backward unfolding. Each of the these two unfoldings generally have two feasible cases, since the segment may be empty or not. As a consequence, the number of disjunctions quadruples. For example, the analysis of goal c of `vTaskDelay` manipulates a disjunctive abstract state with 34 sub-states from the shape domain. We also observed that the number of disjunctions impacts the memory consumption of our analysis.

To mitigate this issue, we inserted merging directives in the code. Indeed, once the insertion or removal of a list element is performed, we can regroup the sub-states in the disjunction. However, in order to keep some information that is expressed by the disjunction, we put the merging directive into conditional statements as shown in Listing 5.4. Merging the disjunctions also has its cost since it done by the $\sqcup^\sharp_{\mathbb{S}}$ operator (see more below). Therefore, it is a work of trial and error, that managed to find the correction positions of these merge directives, so the overall analysis time reduces while keeping sufficient precision.

Cost of the polyhedra domain As shown in the Table 5.3, most of the time of the analysis, between 85 and 92 %, is spent in the numerical domain. Even in the list algorithms analyzed in the previous chapter (see Tables 4.1 and 4.2), this ratio did not exceed 60 %. To understand why the FreeRTOS goals exert such pressure on the numerical domains, we need to delve into its internal workings. The numerical domain used by the analysis is a reduced product between the polyhedra domain [CH78] and lighter domains for equalities, disequalities and inequalities between symbolic variables. Since these domains are quicker, they are normally used by the sequence domain for reduction heuristics that rely on the **compare**$^\sharp_n$ operator. Recall that this operator inputs a symbolic variable α, a numerical

abstract state, a comparison operator ($=, \neq, \leqslant, \dots$) and returns a set of symbolic variables such that the comparisons between α and the variables in the set are valid in the abstract state.

However, one major issue of these lighter domains is that they cannot maintain these constraints if the values are assigned. For instance, if we have a known equality constraint $\alpha = \beta$ and the analysis performs the assignments $\alpha' := \alpha + 1$ and $\beta' := \beta + 1$, then the outcome does not express the equality between α' and β'. This implies that the lighter domains are particularly fit to prove functions that do not alter the data stored in the data structures such as sorting algorithm. However, they fall short when the analysis involves reasoning about incremented values. Since this is the case for `xTaskIncrementTick` (and all functions calling it), the sequence domain has to rely on the polyhedra domain, which has an exponential cost, in order to detect new bound inequalities and length equalities. This constitutes a scalability issue for our analysis, since the abstract states may involve up to 40 sequence variables and 150 numerical symbolic variables. For instance, the $\mathbf{compare}_n^\sharp$ alone accounts for 36 % of the time spent by the sequence domain, and the $\mathbf{guard}_n^\sharp$ operator accounts for 25 %.

Some of these performance issues were mitigated thanks to our profiling work. For instance, we discovered superfluous attempts to perform some reduction heuristics. Additionally, we memoized the $\mathbf{compare}_n^\sharp$ operator so that repeated calls to this operator have a fixed cost. This latter optimization alone divided the time spent by analysis by two on goals e and f of the analysis of `xTaskIncrmentTick`.

Cost of the instantiation Due to the large number of disjunctions, the analysis has to compute many upper bounds of abstract states with the $\sqcup_{\mathbb{S}}^\sharp$ and $\nabla_{\mathbb{S}}^\sharp$ operators. For instance, in the goal c of `vTaskDelay`, the analysis performs 311 joins or widening between states of the shape domain. By profiling, we observed that these operators are costly. In this example, the upper bound operators account for 71 % of the overall time spent by the analysis. In particular, the instantiation step is expensive. It accounts for 86 % of the time spent by the upper bounds operators. Indeed, the $\mathbf{instantiate}_{\mathbb{S}}^\sharp$ operator performs repeated calls to $\mathbf{guard}_s^\sharp$, that is the most complex operator of the sequence domain since it attempts to apply many reduction heuristics. The instantiation step uses this operator even for simple equality constraints. For example, the emptiness constraints accounts for 25 % of the total amount of time spent by the analysis, and the step that guards that all length attribute variables are positive accounts for 7 %.

However, we would like to point out that we performed some improvements on this subject as well. We successfully reduced the time spent by the instantiation step when it guards equality constraints between two sequence variables. Recall that guarding the constraint $S = S'$ involves adding the constraints between the attributes variables: $\mathtt{len}_S = \mathtt{len}_{S'}, \mathtt{max}_S = \mathtt{max}_{S'}$, and $\mathtt{min}_S = \mathtt{min}_{S'}$. To reduce the cost of these constraints, we add the last two only in the simple numerical domains (for inequalities, ...), and not in the polyhedra domain. Indeed, these domains are sufficient to prove sortedness. We observed that complex constraints, such as those involving an incremented quantity, are generated by the analysis since they correspond to loop conditions. In these cases, the (upper or lower) bound of the sequences is made explicit by predicate unfolding. The unfolding generates a constraint stating that the unfolded element is equal to the bound of the sequence. This is the equality that is added to the polyhedra domain. Such a heuristic managed to divide the cost of adding equality constraints between sequence variables by a factor 1.5.

It is difficult to quantify exactly what is the actual cost of the factors listed above. Indeed, it seems that they have a cumulative effect on the time consumption. The control flow of the function `xTaskCatchUpTicks` is formed by two nested loops. So, the control flow of this function is comparable to the control flow of the sorting algorithms analyzed in Section 4.6. However, the complexity of the data structures involved, the nature of the operations performed, and the scale of our problem lead to an explosion in the time spent by our analysis, that we did not observe earlier with the experiments conducted in Section 4.6.

5.6 Lessons learned

To conclude the presentation of our verification of an instance of the FreeRTOS scheduler, let us discuss some important lessons learned along the way.

5.6.1 Not all specifications are equal

The specification presented in Sections 5.3 and 5.4 was not written in a single attempt. Instead, it is the outcome of a process consisting of several corrections and improvements. Some adjustments addressed errors in the specification, while others rectified poor choices. We already mentioned in Remark 4.4 the importance of representing circular list as segments. Here, we would like to give additional examples of the trial and error involved in developing the specification.

Using two tasks predicates In our initial specification attempt, we used two tasks predicates: one for each possible state of the tasks. For the list of tasks in the ready state, the sequence of values did not matter since this list is not sorted. We believed that omitting this sequence in the abstract state would simplify the analysis. However, in the analysis of `xTaskIncrementTick`, moving a task from the delayed state to the ready state led to the introduction of a segment of delayed tasks instead of a segment of ready tasks. This occurred because of the heuristics used by the analysis to determine which predicate to apply when using the rule **seg-intro**: if a memory cell comes from an unfolding of a predicate, the analysis uses that predicate.

Treating `MAX_VALUE` as a constant In the code of FreeRTOS, `MAX_VALUE` is a macro that is set to the greater unsigned integer representable by the architecture. In the specification, this macro was set to the corresponding constant in a 32-bit architecture: 4 294 967 295. Adding such a constant in the polyhedra domain resulted in an explosion of the coefficients used by the polyhedra domain. The analysis ended up with the constant 18 446 744 065 119 617 025 in a constraint after a widening and causing it to get stuck. This issue was solved by expressing `MAX_VALUE` as a variable instead.

The more goals the better Though it may seem preferable to group as many behaviors of a function into a single goal to ease the burden of the specification, the opposite is true. Indeed, the effort needed to merge behaviors in a single goal and to have the analysis succeeds on the goal outweighs the effort required to write several simpler goals.

5.6.2 One aspect of static analysis by abstract interpretation

We would like to discuss one aspect of static analysis by abstract interpretation compared to deductive methods. If the analysis fails, one can still investigate why this failure occurred by looking at the computed abstract states. Deductive methods rely either on theorems provers, in which case the user has to write proof himself, or on external provers such as SMT solvers to discharge proof obligations. However, it happens that the solver does not manage to prove this obligation nor is it able to construct a counter example. This could be caused by a lack of time from the solver or by the fact that the solver entered an infinite loop of quantifier instantiation. Though there exist some tools to debug the exploration of a solver [BMS19], they are not available for all solvers and require some insight on the triggers used by the solver.

In contrast, our approach allow us to pinpoint where the analysis fails to compute the expected invariant to prove the goal. When this occurs, one can determine whether this failure is due to a false goal (because of an error in the specification), or the analysis lacks precision. For example, aggressive folding by the widening operator led to a loss of information in the loop invariant. When this happened, we quickly identified the point where precision was lost and resolved the issue by adding analysis directives or ghost code.

However, when the loss of precision is caused by a bound constraint that is not correctly propagated or a reduction heuristic that is not triggered by our analysis, as in the case analyzing goal c of `vTaskDelay`, then inspecting the abstract state becomes a tedious exercise. One has to figure out what are the symbolic variables involved by looking at the memory part of the abstract value, and inspect the numerical part of the abstract state to figure out what is missing. Given that the analysis of FreeRTOS produces abstract state with a numerical part that requires up to 300 lines of log (we show an example of such abstract state in Figure 5.10, the numerical state corresponds to the green box on the left), manually inspecting these lines of log become wearying.

5.6.3 Improving the analysis

While attempting to verify the functions of FreeRTOS, we made several improvements to the analysis. We already mentioned earlier the modification of the analysis aiming at improving its performance.

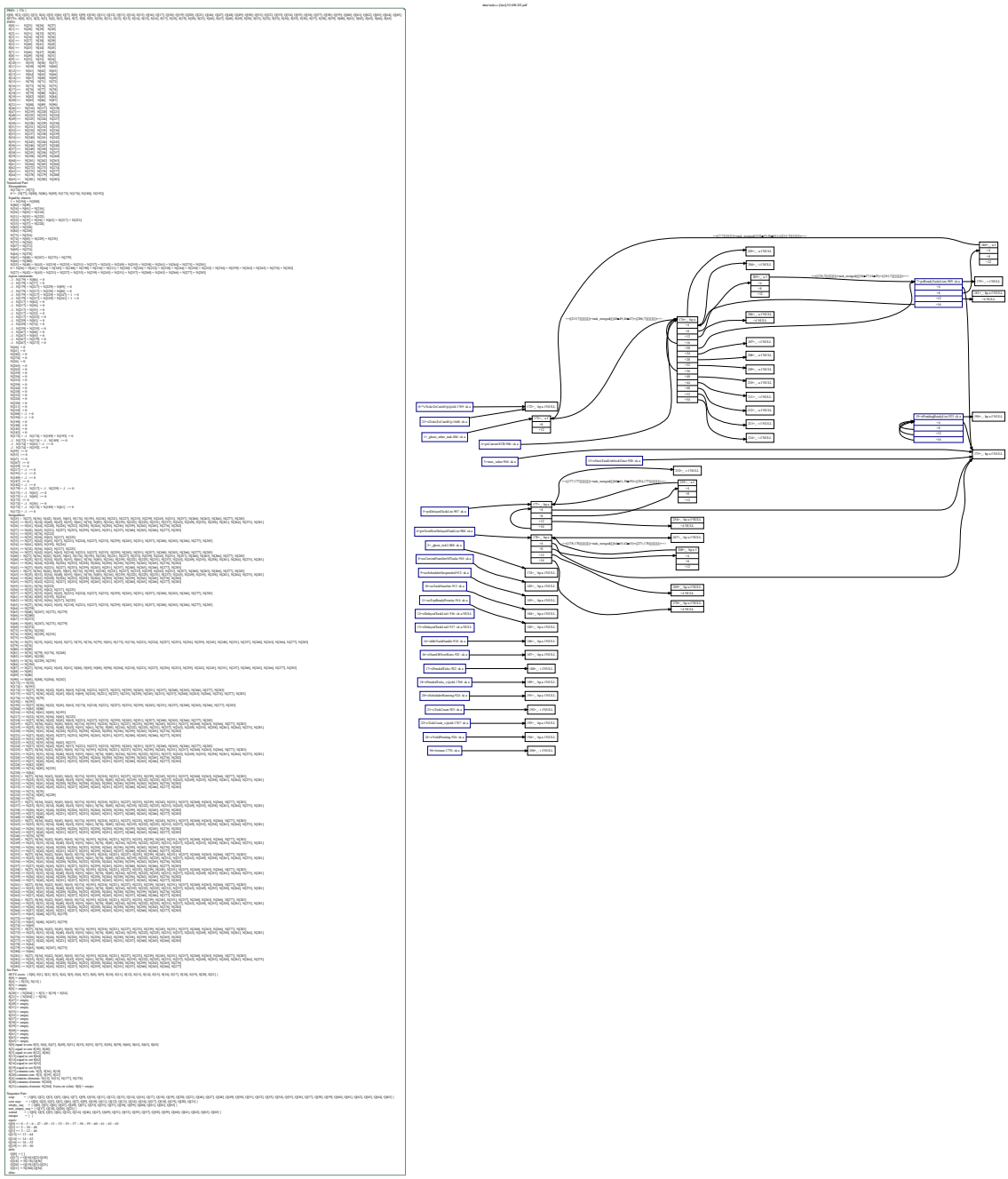


Figure 5.10: Final state computed by the analysis of goal *b* of `xTaskCatchUpTicks`

Additionally, we extended our specification language to support elements required by the specification of FreeRTOS.

Shape part Regarding the memory part of our analysis, the improvements mostly consist of improving the rules used by the lattice operators. For instance, rule **seg-weak** attempts to fold a part of the abstract memory into a segment. To do so, it must infer what is the node corresponding to the end of this segment. In the analysis of `xTaskIncrmentTick`, the analysis failed to infer this end point because several choices were possible. Our heuristic consists of choosing the candidate that is the minimal element according to the topological order induced by the memory graph.

Sequence part The analysis of the FreeRTOS scheduler required a single modification on the sequence domain, specifically to the $\mathbf{guard}_s^\sharp$ operator. We did not add a new inference rule, but we rather changed the mechanism used to trigger them. Recall that the $\mathbf{guard}_s^\sharp$ operator applies inference rule when it detects new information, such as a novel definition of a sequence variable. For the analysis of FreeRTOS, we added another trigger: when the $\mathbf{guard}_s^\sharp$ operator infers that two sequence variables S and S' are equal, it looks for other sequence variables that are known to have the same length as these sequences, and it marks them as modified. This lets the operator infer new constraints on these variables.

Such a modification was useful for the analysis of `xTaskIncrementTick`. The loop invariant states that the sequence of values D_v^r and the sequence of addresses D_a^r of the tasks that were put back in the ready state have the same length. When the analysis leaves the loop, it infers that the sequence D_v^r is equal to the sequence of values of the tasks whose waiting time expired, noted D_v^{exp} in Table 5.2. Then, our new trigger tells the $\mathbf{guard}_s^\sharp$ operator to also attempts to perform reduction on sequence variables that have the same length as D_v^{exp} and D_v^r, namely D_a^{exp} and D_a^r. Since these two variables are prefixes of the same sequence, the operator infers that they are equal.

Although these improvements to the precision of the analysis were necessary for our results, they involved only minor modifications, consisting of a few lines of code, and did not constitute the core of our verification effort. This indicates that the operators were already precise enough to handle most of the computations required by the analysis.

5.6.4 The verification effort and its distribution

To conclude this section, let us discuss the effort required to verify the functions of this instance. Overall, the verification of the functions of FreeRTOS took 9 months. We estimate the distribution of work as follows: about 25 % of the time was spent on specification work, almost 60 % on inspecting abstract values to identify why the analysis did not work, and the remaining 15 % on modifications to the MemCAD tool and enhancements to the specification language.

However, this distribution is not uniform across all verified functions. A full month was required for the initial specification work to describe the default state of the scheduler, H, presented in Section 5.3. This preliminary work was completed before any verification attempts were made for individual functions. The analysis of `xTaskIncrementTick` alone required 8 weeks of works, since its complex invariant required the addition of ghost code and some improvements of the analysis. Most of these two months was spent chasing loss of precision by inspecting abstract states. However, this effort is compensated by the analysis of `xTaskResumeAll` and `xTaskCatchUpTicks`. Proving these functions was significantly faster, taking only 3 weeks for both functions. Similarly, the attempts to analyze `vTaskDelay` took 2 months, most of which was spent trying to identify the reason our analysis does not manage to prove this function.

5.7 Extending the verified instance

This section outlines the potential costs associated with incorporating previously omitted features into the verified instance.

5.7.1 Adding other states

The first extension to the verified instance would be to consider other task states. That is to say the *blocked* or *suspended* states. Each state corresponds to a list that stores all the lists in this state. These

features introduce functions that remove a task from the running state to the blocked or suspended states and put back a task from these states to the ready state. We argue that verifying these functions is straightforward using a precondition that explicitly splits the list of blocked or suspended tasks to materialize the task of interest similarly to the strong version of the ready part.

5.7.2 Multiple levels of priorities

Another feature that we ignored in this instance is the possibility to use several levels of priorities. However, analyzing such features requires an abstract domain to reason about the content stored in data structure that is more expressive than the sequence domain presented here. To understand why, let us consider an instance with two levels of priorities, 0 and 1. Each level corresponds to a list. Consequently, to analyze the function `xTaskIncrementTick`, specifying the content of these lists requires expressing "the subsequence of expired tasks whose priority level is equal to 0". This kind of *sequence by comprehension* constraint is out of the scope of the sequence domain proposed here.

5.7.3 Events

From a scheduler point of view, the synchronization between tasks in FreeRTOS relies on events. To each event, *e.g.* a new element is added in a queue, corresponds a list of tasks. As explained before, the tasks in the list are linked using the `xEventListItem`. Such a list is sorted in decreasing order according to the priority level of the tasks. In addition, the tasks waiting for an event to happen are in the blocked state. Consequently, it must be in either the list of delayed or blocked tasks. This means that a waiting task can be accessed from two lists. Expressing such sharing requires some care to ensure that the separating conjunction holds. Li *et al.* [LRC15] introduced an analysis that support this kind of sharing. One of the two lists is explicitly expressed as a standard inductive predicate, while the second one is implicitly specified by content constraints. This means that for a task belonging to both lists, the values of the `pxNext` and `pxPrev` fields of the implicit list are constrained to be in the set of addresses in the explicit lists. However, this extension is not yet able to assert that both lists are sorted, nor is it able to analyze an insertion in the implicit list.

5.7.4 Support for interruptions

Our analysis assumes that the scheduler runs without interruptions, except the tick interruption. This hypothesis allows the analysis to consider only sequential code. Indeed, interruptions introduces concurrency between the scheduler code and the interruption service routines (ISR) (that may also be part of the scheduler). For instance, Andronick *et al.* [ALM+16] models interruption using parallelism: all ISR are specified by an infinite loop running in parallel of the function to verify. This means that verifying a FreeRTOS instance with interruptions requires an analysis that can take into account all interleaving of ISR with the verified functions.

5.8 Related works

This section discusses the other attempts to apply formal methods in order to verify some properties on the FreeRTOS scheduler.

Specification In [DGaM09], Déharbe *et al.* present a formalization of a subset of the FreeRTOS API, including scheduler related functions in B. Lin [Lin10] later extended this specification in Z. This specification was later used by Mühlberg *et al.* [MF11] to perform bounded model checking on the FreeRTOS scheduler and the Queue model.

Additionally, Cheng *et al.* [CWD15] leveraged an unpublished formal model of FreeRTOS written in B by Jean Raymond Abrial to check the consistency of the task model using the Z/Eves tool [Hut99]. This work is complementary to ours. It is not able to check that the code of the scheduler satisfies the specification. However, this model proves to be useful to reason about fixed FreeRTOS applications in order to show that higher priority tasks are delayed often enough to avoid starvation of lower priority tasks.

Deductive methods The verification of FreeRTOS using deductive methods have been studied. For example Ferreira *et al.* [FHQ12] applied the Hip/Sleek prover on several functions of the FreeRTOS scheduler. It is worth noting that this tool, in addition to user-provided specification and inductive predicates, relies on user-provided lemmas similar to our concatenation lemmas. Additionally, this work focuses solely on loop-free functions. For example, it does not verify the insertion in a sorted list function `vListInsert`.

To date, the most advanced verification effort of the FreeRTOS scheduler is the one proposed by Divikaran *et al.* [DDK+15]. Using a refinement approach with the VCC tool [CDH+09], this work managed to find actual bugs in the model of FreeRTOS. It is worth noting that this work required an extensive work of code annotation. For instance, in order to prove the 17 functions of the scheduler, 361 LoC in total, the verification effort required 2347 lines of annotations in the source code, as well as 2005 LoC corresponding to higher level models. Therefore, the obtained specification/code ratio is equal to 12. This annotation effort is more intensive than ours, as it requires specifying each of the 4 layers used in the refinement proof. Notably, they separate the reasoning about the layout of the doubly-linked list data structure from the reasoning about its content by using an Abstract Data-Type in the higher-level specification of the scheduler. In contrast, our works requires only a single level of specification. Additionally, refinement proofs necessitate extra information such as the loop invariants and important intermediary states to guide the solvers. Our approach does not need these since it is designed to infer them automatically.

To our knowledge, there exists no work based on automatic static analysis to verify the FreeRTOS scheduler.

6 Chapter

Conclusion & Future works

In this thesis we examined the verification of a critical OS component, namely the task scheduler, using an automatic and sound approach: abstract interpretation-based static analysis. By verification, we mean not only proving that the program is free of runtime errors, but also ensuring the preservation of the invariants of the scheduler and the partial functional correctness of its constituent functions. To achieve this goal, we designed an analysis that is able to express sophisticated constraints on the content of inductive data structures.

The first step in the design of this analysis was to construct an abstract domain that can reason over sequence of values. This domain expresses relational constraints over expressions built with sequence and numerical symbolic variables and composed either by concatenating or sorting them. Additionally, our sequence domain employs two auxiliary domains that parametrize it. A numerical domain is used to express constraints over extreme elements of sequences and their length. To reason over the content of the sequences regardless of the position of their elements, the sequence domain uses a multiset domain.

In a second step, we extended a shape analysis based on separation logic using a reduced product of this analysis with the sequence domain. This extension works by adding to inductive predicates a novel type of parameter, a sequence parameter that expresses the content of the data structure summarized by the inductive predicate. We proposed a classification of these arguments to detect which one can be used in segment predicates and to automatically infer properties over these parameters. The abstract transfer functions of the shape domain were also extended to take into account the sequence parameters. We implemented this analysis, and it turned out to be expressive enough to prove the partial functional correctness of complex programs such as implementations of sorting algorithms using either lists and binary search trees. This analysis also successfully analyzed list libraries taken from real-world applications.

Finally, we applied this analysis to an instance of an industrial operating system, FreeRTOS. We wrote down a specification of this instance to express both the different states of the scheduler and the pre- and post-conditions of functions in the instance. Since our analysis does not require providing loop invariant, the overall specification effort was modest compared to other approaches. Then we attempted to verify the instance. The analysis required to perform a few modifications on the verified source code in order to guide it. Ultimately, we successfully verified the partial functional correctness of all functions of the scheduler instance except for one.

We do not ignore that our results are obtained thanks to our knowledge of its workings. For instance, we acknowledge that the modifications of the code could not be reproduced by an inexperienced user. Additionally, our analysis shows poor performance regarding time consumption of the analysis of FreeRTOS functions, due to the complexity of the numerical constraints it must express. Nonetheless, we still regard these results as encouraging, regarding our initial goal to automatically prove the partial functional correctness of a task scheduler. Indeed, most of the reasoning, especially the complex reasoning regarding the content of the data structures, is performed automatically by our analysis.

This work opens up multiple opportunities for further extensions. Some involve addressing weaknesses in our analysis that were detected during the analysis of FreeRTOS and that should be remedied to improve it, others are possible improvements to enhance our analysis, and some are specific to our effort in verifying task schedulers.

Improving performances We have acknowledged that our analysis faces a performance issue: it does not scale well. The main reason for the issue is the use of the polyhedra domain, which has a worst-case complexity that is exponential in the number of variables. This has posed a significant challenge for the analysis of FreeRTOS. We argue that this issue could be solved by using two different abstract domains. Since length constraints boil down to linear equalities, they can be represented by the Karr domain [Kar76] in a efficient manner. Regarding the inequalities of extreme values of sequence, can be stored in the octagon [Min01] domain. Both domains have a polynomial worst-case complexity. Additionally, they can be further optimized using variable packing [BCC+03].

Strengthening the reduced product Another caveat of our reduced product is that some abstract transfer functions perform operation on the shape part of the abstract value without taking the sequence part into account. This is especially true for the lattice operators. They compute the shape part of their outcome without using information available in the sequence part. Though these limitations can be circumvented using either analysis directive or ghost code, these require some insight into the rules used by the operators. Therefore, we find worthy to improve the reduced product operators in order to let the sequence part of the abstract value give some hint to its shape part. This could allow us to get rid of these *ad hoc* solution to guide the analysis. It is our opinion that this could also solve the unproved goal in the analysis of the FreeRTOS scheduler.

Instantiation The instantiation mechanism used by the lattice operators of the reduced product is a major bottleneck for the performance of our implementation. This stems from the fact that we systematically used the $\mathbf{guard}_s^\sharp$ operator even though most of the constraints are equalities between sequence variables. As a consequence, we find interesting to investigate further the instantiation steps of the lattice operators.

Using other sequence domains Since the combined shape analysis with sequence constraints is a reduced product, it can leverage any sequence domain expressing concatenation based constraints. Many domains have been proposed that express constraints over sequences of values, that are complementary to the constraints expressed by our domain. For instance, the domain proposed by Bouajjani *et al.* [BDES09] expresses constraints such as "the elements of S are equal to the elements of S' incremented".

Proving termination As stated in the introduction termination is a liveness property. This implies that our approach that computes an over-approximation of the set of reachable states for all points in the program is not sufficient to establish it. Indeed, proving termination typically works by inferring a variant for each loop in the program. In all our examples, the length of some sequence in the abstract state is a loop variant. Therefore, we find interesting to investigate a combination of our abstract domain together with a domain that is able to infer loop variants such as the one proposed by Urban *et al.* [UM14]. This could enable our analysis to establish the functional correctness of the analyzed functions.

Applying the analysis on different versions of the analyzed instance While the minimal effort required for our specification supports the automatic nature of our approach, we propose another method to validate it. Maintaining verified software programs also requires to maintain its proof. Lawall *et al.* [LNLed] evaluated the cost of reusing the verification of a function in Linux Completely Fair Scheduler proved using the WP plugin of Frama-C that leverages deductive methods. Regarding the possibility to reuse a verification effort through iterated modifications of the code, they conclude that:

> “ In the cases where the core algorithm has not changed, the proof has gone through with only minor tweaks to the specifications, many of which we were able to perform in a matter of minutes. In the cases where the core algorithm does change, substantially more effort was required, with several months of work required for the changes in the most recent Linux versions.

We find interesting to attempt replicating the same protocol on our analyzed instance. In the event our proof effort is reusable as is, this would provide additional support for the automaticity of our approach and help offset the cost of the initial proof effort.

Analyzing other instances Another way to test the automaticity of our approach is to attempt to analyze different instances of FreeRTOS. Though we noted in Section 5.7, that some features of FreeRTOS, such as events, are out of the scope of what our analysis can express, there is still plenty of opportunities to analyze different instances. We could try to analyze different scheduling policies, for example by disabling time slicing. This instance does not require to modify the specification of the states of the scheduler presented in Section 5.3. Additionally, we could attempt to verify new features that can already be expressed by the analysis such as the blocked and suspended states.

Bibliography

[2202] ISO/IEC JTC 1/SC 22. Z formal specification notation — Syntax, type system and semantics. Standard, International Organization for Standardization, Geneva, CH, July 2002.

[2218] ISO/IEC JTC 1/SC 22. Programming languages — C. Standard, International Organization for Standardization, Geneva, CH, March 2018.

[AHC+16] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 175–188, New York, NY, USA, 2016. Association for Computing Machinery.

[ALM+16] June Andronick, Corey Lewis, Daniel Matichuk, Carroll Morgan, and Christine Rizkallah. Proof of os scheduling behavior in the presence of interrupt-induced concurrency. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 52–68, Cham, 2016. Springer International Publishing.

[AM19] Vincenzo Arceri and Isabella Mastroeni. An automata-based abstract semantics for string manipulation languages. In Alexei Lisitsa and Andrei P. Nemytykh, editors, *Proceedings Seventh International Workshop on Verification and Program Transformation, VPT@Programming 2019, Genova, Italy, 2nd April 2019*, volume 299 of *EPTCS*, pages 19–33, 2019.

[Ama21] Roberto Amadini. A survey on string constraint solving. *ACM Comput. Surv.*, 55(1), nov 2021.

[AOCF22] Vincenzo Arceri, Martina Olliaro, Agostino Cortesi, and Pietro Ferrara. Relational string abstract domains. In Bernd Finkbeiner and Thomas Wies, editors, *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*, volume 13182 of *Lecture Notes in Computer Science*, pages 20–42. Springer, 2022.

[BA20] Kevin Boos and Chen Ang. *Theseus: Rethinking Operating Systems Structure and State Management*. PhD thesis, Rice University, USA, 2020. AAI28735941.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Conference on Programming Languages Design and Implementation (PLDI)*, pages 196–207, 2003.

[BCF+24] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Ansi/iso c specification language. Technical report, CEA-List, Inria, 2024.

[BDE+10] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, Ahmed Rezine, and Mihaela Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 72–88, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BDES09] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. A logic-based framework for reasoning about composite data structures. In *Proceedings of the 20th International Conference on Concurrency Theory*, CONCUR 2009, page 178–195, Berlin, Heidelberg, 2009. Springer-Verlag.

[BDES11] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 578–589, New York, NY, USA, 2011. Association for Computing Machinery.

[BDES12a] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 1–22, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BDES12b] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis*, pages 167–182, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BGGP99] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revising hull and box consistency. In *Proceedings of the 1999 International Conference on Logic Programming*, page 230–244, USA, 1999. Massachusetts Institute of Technology.

[BGZ17] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: a string solver with theory-aware heuristics. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, FMCAD '17, page 55–59, Austin, Texas, 2017. FMCAD Inc.

[BKA+21] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 84–99, New York, NY, USA, 2021. Association for Computing Machinery.

[BKL18] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Ghosts for lists: A critical module of contiki verified in frama-c. In Aaron Dutle, César Muñoz, and Anthony Narkawicz, editors, *NASA Formal Methods*, pages 37–53, Cham, 2018. Springer International Publishing.

[BMS19] Nils Becker, Peter Müller, and Alexander J. Summers. The axiom profiler: Understanding and debugging smt quantifier instantiations. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–116, Cham, 2019. Springer International Publishing.

[BT24] Richard Barry and The FreeRTOS Team. *Mastering the FreeRTOS Real Time Kernel - A Hands On Tutorial Guide*, 2024.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, page 269–282, New York, NY, USA, 1979. Association for Computing Machinery.

[CCK+] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. *SIGPLAN Not.*, 46(1):105–118, January 2011.

[CCLR15] Arlen Cox, Bor-Yuh Evan Chang, Huisong Li, and Xavier Rival. Abstract domains and solvers for sets reasoning. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 356–371, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 23–42, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[CDNQ07] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 307–320, 2007.

[CDOY07] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Footprint analysis: a shape analysis that discovers preconditions. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, page 402–418, Berlin, Heidelberg, 2007. Springer-Verlag.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), dec 2011.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, page 84–96, New York, NY, USA, 1978. Association for Computing Machinery.

[Cha11] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 418–430, New York, NY, USA, 2011. Association for Computing Machinery.

[Cha20] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020.

[Cho20] Nathan Chong. Ensuring the memory safety of freertos part 1. https://www.freertos.org/2020/02/ensuring-the-memory-safety-of-freertos-part-1.html, 2020. Accessed: 2024-25-01.

[CJ21] Nathan Chong and Bart Jacobs. Formally verifying freertos' interprocess communication mechanism. In *Embedded World Exhibition & Conference 2021*, 2021.

[CL20] Christopher Curry and Quang Loc Le. Bi-abduction for shapes with ordered data, 2020.

[Con05] Software Freedom Conservancy. The git version control system. https://git-scm.com/, 2005.

[Cor23] The MITRE Corporation. 2023 cwe top 10 kev weaknesses list insights. https://cwe.mitre.org/top25/archive/2023/2023_kev_insights.html, 2023. Accessed: 2024-25-01.

[Cou02] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1):47–103, 2002. Static Analysis.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 247–260, New York, NY, USA, 2008. Association for Computing Machinery.

[CWD15] Shu Cheng, Jim Woodcock, and Deepak D'Souza. Using formal reasoning on a model of tasks for freertos. *Form. Asp. Comput.*, 27(1):167–192, jan 2015.

[Dar04] Nicolas Darnis. The generic data-structure library, 2004. https://directory.fsf.org/wiki/GDSL.

[DDK+15] Sumesh Divakaran, Deepak D'Souza, Anirudh Kushwah, Prahladavaradan Sampath, Nigamanth Sridhar, and Jim Woodcock. Refinement-based verification of the freertos scheduler in vcc. In Michael Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering*, pages 170–186, Cham, 2015. Springer International Publishing.

[Dev15] Redox Developers. Redox, a unix-like operating system written in rust. https://www.redox-os.org, 2015. Accessed: 2023-19-12.

[Dev16] FreeBSD Developers. Kyua, a testing framework for infrastructure software. https://github.com/jmmv/kyua/, 2016. Accessed: 2023-19-12.

[dev23] The Linux Kernel devellopers. Coding guidelines. https://docs.kernel.org/rust/coding-guidelines.html, 2023.

[DGaM09] David Déharbe, Stephenson Galvão, and Anamaria Martins Moreira. *Formalizing FreeRTOS: First Steps*, page 101–117. Springer-Verlag, Berlin, Heidelberg, 2009.

[dH21] Charles de Haro. Spécifications des tâches de freertos. Technical report, École normale supérieure, 2021.

[DHK21] Adel Djoudi, Martin Hána, and Nikolai Kosmatov. Formal verification of a javacard virtual machine with frama-c. In *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings*, page 427–444, Berlin, Heidelberg, 2021. Springer-Verlag.

[Dij70] Edsger Wybe Dijkstra. *Notes On Structured Programming*. 1970.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[dMP13] Leonardo de Moura and Grant Olney Passmore. The strategy challenge in smt solving. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, pages 15–44, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 155–167, New York, NY, USA, 2003. Association for Computing Machinery.

[DS07] David Delmas and Jean Souyris. Astrée: From research to industry. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, pages 437–451, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[EP23] Mnacho Echenim and Nicolas Peltier. An undecidability result for separation logic with theory reasoning. *Inf. Process. Lett.*, 182(C), aug 2023.

[ESW15] Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. On automated lemma generation for separation logic with inductive definitions. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *Automated Technology for Verification and Analysis*, pages 80–96, Cham, 2015. Springer International Publishing.

[FHQ12] Joao F. Ferreira, Guanhua He, and Shengchao Qin. Automated verification of the freertos scheduler in hip/sleek. In *Proceedings of the 2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*, TASE '12, page 51–58, USA, 2012. IEEE Computer Society.

[FHR+18] Tomáš Fiedor, Lukáš Holík, Adam Rogalewicz, Moritz Sinn, Tomáš Vojnar, and Florian Zuleger. From shapes to amortized complexity. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 205–225, Cham, 2018. Springer International Publishing.

[FP23] Jean-Christophe Filliâtre and Andrei Paskevich. L'arithmétique de séparation. In Timothy Bourke and Delphine Demange, editors, *JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs*, pages 274–283, Praz-sur-Arly, France, January 2023.

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 235–246, New York, NY, USA, 2008. Association for Computing Machinery.

[GN23] Vincent Giraud and David Naccache. Power analysis pushed too far: Breaking android-based isolation with fuel gauges. In Junji Shikata and Hiroki Kuzuno, editors, *Advances in Information and Computer Security*, pages 3–15, Cham, 2023. Springer Nature Switzerland.

[GRR23a] Josselin Giet, Felix Ridoux, and Xavier Rival. Artifact for "A Product of Shape and Sequence Abstractions", July 2023.

[GRR23b] Josselin Giet, Félix Ridoux, and Xavier Rival. A product of shape and sequence abstractions. In Manuel V. Hermenegildo and José F. Morales, editors, *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings*, volume 14284 of *Lecture Notes in Computer Science*, pages 310–342. Springer, 2023.

[GRS05] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 338–350, New York, NY, USA, 2005. Association for Computing Machinery.

[GSC+16a] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[GSC+16b] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, November 2016. USENIX Association.

[GSL+17] Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, and Nicolas Palix. Ipanema : un langage dédié pour le développement d'ordonnanceurs multi-coeur sûrs. In *Compas 2017: Conférence d'informatique en Parallélisme, Architecture et Système*, Sophia Antipolis, France, June 2017.

[Haw17] Jasper Hawinkel. Verification of the freertos scheduler with verifast, a case study in software verification. Master's thesis, KU Leuven, 2017.

[HLR+13] Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří ŠimáăźEk, and Tomáš Vojnar. Fully automated shape analysis based on forest automata. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, page 740–755, Berlin, Heidelberg, 2013. Springer-Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 339–348, New York, NY, USA, 2008. Association for Computing Machinery.

[HPR+22] Lukáš Holík, Petr Peringer, Adam Rogalewicz, Veronika Šoková, Tomáš Vojnar, and Florian Zuleger. Low-Level Bi-Abduction. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:30, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Hut99] Z/eves version 1.5: An overview. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Applied Formal Methods — FM-Trends 98*, pages 367–376, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Inc22] Amazon Inc. The freertos kernel, 2022. https://github.com/FreeRTOS.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. *SIGPLAN Not.*, 36(3):14–26, jan 2001.

[IRV14] Radu Iosif, Adam Rogalewicz, and Tomáš Vojnar. Deciding entailments in inductive separation logic with tree automata. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis*, pages 201–218, Cham, 2014. Springer International Publishing.

[JKJ+18] RALF JUNG, ROBBERT KREBBERS, JACQUES-HENRI JOURDAN, ALEŠ BIZJAK, LARS BIRKEDAL, and DEREK DREYER. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 661–667, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[JSP10] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In Kazunori Ueda, editor, *Programming Languages and Systems*, pages 304–311, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6(2):133–151, jun 1976.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[KGA+13] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4), feb 2013.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[KNC15] Steve Klabnik, Carol Nichols, and The Rust Community. The rust programming language. https://doc.rust-lang.org/book/, 2015. Accessed: 2023-11-22.

[KS19] Krono-Safe. The asterios real-time kernel. https://www.krono-safe.com/asterios-rtk/, 2019.

[KT14] Daniel Kroening and Michael Tautschnig. CBMC – C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *LNCS*, pages 389–391. Springer, 2014.

[LABS12] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012.

[LAC+15] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, page 21–26, New York, NY, USA, 2015. Association for Computing Machinery.

[LBCR17] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. Semantic-directed clumping of disjunctive abstract states. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 32–45, New York, NY, USA, 2017. Association for Computing Machinery.

[Lem20] Matthieu Lemerre. EducRTOS. https://github.com/EducRTOS/EducRTOS, 2020.

[Lem24] Matthieu Lemerre. Ssa translation is an abstract interpretation. Presentation at Atntique seminar, 2024.

[LGC+20] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. Provable multicore schedulers with ipanema: Application to work conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[LGQC14] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 52–68, Cham, 2014. Springer International Publishing.

[Lin10] Yuhui Lin. Formal analysis of freertos. Master's thesis, University of York, 2010.

[LLC19] Google LLC. Kunit, a lightweight unit testing framework for the linux kernel. https://doc.rust-lang.org/book/, 2019. Accessed: 2023-11-22.

[LMU05] Julia L. Lawall, Gilles Muller, and Richard Urunuela. Tarantula: Killing driver bugs before they hatch. In *The 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 13–18, Chicago, IL, March 2005.

[LNLed] Julia Lawall, Keisuke Nishimura, and Jean-Pierre Lozi. Should we balance? towards formal verification of the linux kernel scheduler. Under review, to be published.

[LR17] Jiangchao Liu and Xavier Rival. An array content static analysis based on non-contiguous partitions. *Computer Languages, Systems & Structures*, 47:104–129, 2017. Special issue on the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2015).

[LRC15] Huisong Li, Xavier Rival, and Bor-Yuh Evan Chang. Shape analysis for unstructured sharing. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis*, pages 90–108, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[LRS06] Alexey Loginov, Thomas Reps, and Mooly Sagiv. Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In Kwangkeun Yi, editor, *Static Analysis*, pages 261–279, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[LRT+14] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A dpll(t) theory solver for a theory of strings and regular expressions. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 646–662, Cham, 2014. Springer International Publishing.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[Mak77] Gennady S. Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of the USSR—Sbornik*, 32(4):129–198, 1977.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[MDK16] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. Formal Verification of a Memory Allocation Module of Contiki with Frama-C: a Case Study. In *CRiSIS 2016 - 11th International Conference on Risks and Security of Internet and Systems*, Roscoff, France, September 2016.

[MF11] Jan Tobias Mühlberg and Leo Freitas. Verifying freertos: from requirements to binary code. In Alexander Romanovsky, Cliff Jones, Jens Bendiposto, and Michael Leuschel, editors, *Proceedings of the 11th international workshop on automated verification of critical systems (AVoCS 2011)*. Electronic communications of the EASST, 2011.

[Mil19] Matt Miller. Trends, challenges, and shifts in software vulnerability mitigation. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf, 2019. Accessed: 2024-25-01.

[Min01] Antoine Miné. The octagon abstract domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, page 310, USA, 2001. IEEE Computer Society.

[Min17] Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Found. Trends Program. Lang.*, 4(3–4):120–372, dec 2017.

[MNN16] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. A parametric abstract domain for lattice-valued regular expressions. In Xavier Rival, editor, *Static Analysis*, pages 338–360, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[MTLT10] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 211–222, New York, NY, USA, 2010. Association for Computing Machinery.

[NAFC21] Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. Twinning automata and regular expressions for string static analysis. In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 267–290, Cham, 2021. Springer International Publishing.

[NBG+19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

[NLBR21] Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival. No crash, no exploit: Automated verification of embedded kernels. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 27–39, 2021.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[O'H04] Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[OMLB16] Abdelraouf Ouadjaout, Antoine Miné, Noureddine Lasla, and Nadjib Badache. Static analysis by abstract interpretation of functional properties of device drivers in tinyos. *Journal of Systems and Software*, 120:114–132, 2016.

[Pér18] Thibault Pérami. Sel4: Fixing long-standing issue. https://www.cl.cam.ac.uk/~tp496/pages/internships.html#internships, 2018. Accessed: 2023-19-12.

[Qui46] W. V. Quine. Concatenation as a basis for arithmetic. *Journal of Symbolic Logic*, 11(4):105–114, 1946.

[RBLL24] Xavier Rival, Francois Berenger, Jiangchao Liu, and Huisong Li. *MemCAD - user manual*, 2024.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74, 1953.

[RV99] Alexandre Riazanov and Andrei Voronkov. Vampire. In *Automated Deduction — CADE-16*, pages 292–296, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., USA, 1989.

[SPV17] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Fast polyhedra abstract domain. In *Symposium on Principles of Programming Languages (POPL)*, pages 46–59. ACM, 2017.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, jan 1998.

[Stu98] Michael Stutz. Sunk by windows nt. https://www.wired.com/1998/07/sunk-by-windows-nt/, 1998. Accessed: 2024-31-01.

[Stu08] Henrik Stuart. Hunting bugs with coccinelle. Master thesis, University of Copenhagen, 2008.

[Tea15] The Rust Team. The rustonomicon. https://doc.rust-lang.org/nomicon/, 2015. Accessed: 2023-11-22.

[Tea23a] The CertikOS Development Team. The mc2 verified concurrent operating system. https://flint.cs.yale.edu/certikos/mc2.html, 2023.

[Tea23b] The CertikOS Development Team. The mcertikos verified hypervisor. https://flint.cs.yale.edu/certikos/mcertikos.html, 2023.

[Tea23c] The Coq Development Team. The Coq proof assistant. https://coq.inria.fr, 1999-2023.

[TLKC16] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. Automated mutual explicit induction proof in separation logic. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, pages 659–676, Cham, 2016. Springer International Publishing.

[Tor22] Linus Torvalds. The linux kernel, 2022. https://git.kernel.org.

[UM14] Caterina Urban and Antoine Miné. An abstract domain to infer ordinal-valued ranking functions. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 412–431. Springer, 2014.

[Vaf09] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 335–348, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Ven96] Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer, 1996.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 256–271, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

Index

R

S

T

U

V

Index of notations

Symbols

A

Abstraction

B

Bottom

C

Concretization

D

Domain

E

F

G

H

I

Inclusion

J

Join

L

M

N

O

P

R

S

T

U

V

W

X

List of Theorems

List of Definitions

List of Figures

List of Lemmas

List of Tables

List of Listing

List of Remarks

RÉSUMÉ

Le but de cette thèse est la vérification d'ordonnanceurs de tâches de systèmes d'exploitation par analyse statique basée sur l'interprétation abstraite. Les systèmes d'exploitation sont des ensembles de logiciels présents sur presque tout ordinateur. Leur but est de permettre aux autres programmes de s'exécuter sans avoir à gérer des spécificités bas niveau comme la mémoire. En conséquence de ce rôle central, les systèmes sont devenus des composants critiques des infrastructures informatiques : toute erreur au niveau du système d'exploitation peut avoir des conséquences sur les autres programmes allant jusqu'au plantage de l'ordinateur.
Un des composants au cœur d'un système d'exploitation est l'ordonnanceur de tâches. Ce dernier est chargé de déterminer quelle tâche peut s'exécuter à quel moment, en suivant une politique préétablie. Les ordonnanceurs de tâches utilisent des structures de données dynamiques non bornées afin de stocker les éléments nécessaires à leur fonctionnement. Ces structures de données permettent de déplacer facilement les éléments d'une structure vers l'autre. Par conséquent, la vérification d'un ordonnanceur de tâche nécessite de concevoir une analyse capable de représenter correctement ces structures de données et leur contenu.
La première partie de cette thèse décrit un langage impératif jouet semblable au C manipulant explicitement la mémoire. On donne ensuite la sémantique concret de ce langage, puis on présente une analyse statique numérique afin de déterminer la plage de valeur des variables ainsi qu'une analyse de forme capable de raisonner sur des structures de données inductives non bornées.
La seconde partie est consacrée à la présentation d'un domaine abstrait relationnel capable de raisonner sur des séquences symboliques. Ce domaine exprime des contraintes sur le contenu de ces séquences comme leurs longueurs, leurs valeurs extrémales et leurs caractères triés.
La troisième partie présente la combinaison de l'analyse de forme présentée dans la première partie avec le domaine de séquences. Cette combinaison augmente l'expressivité de l'analyse. Cette dernière est maintenant capable de prouver la correction fonctionnelle partielle d'algorithmes complexes comme des algorithmes de tris sur les listes ou les arbres binaires, ainsi que sur des bibliothèques de listes provenant d'applications réelles.
La dernière partie de cette thèse présente le travail d'application de l'analyse sur une instance de l'ordonnanceur de tâches de FreeRTOS. La première étape de la vérification est la formalisation des propriétés que nous cherchons à établir sur les fonctions de l'ordonnanceur. Cela inclut les invariants globaux de l'ordonnanceur. La seconde étape concerne le travail de validation pour montrer que ces propriétés spécifiées sont vérifiées par les fonctions de l'instance au moyen de l'analyse.

MOTS CLÉS

Interpretation abstraite, Systèmes d'exploitation, Ordonnanceurs, Logique de séparation

ABSTRACT

The aim of this thesis is the verification of task schedulers for operating systems through static analysis based on abstract interpretation. Operating systems are collections of software present on almost every computer. Their purpose is to allow other programs to run without having to manage low-level problems such as memory. Due to this central role, operating systems have become critical components of IT infrastructures: any error in the operating system can have consequences on other programs, potentially causing the entire computer to crash.
One component at the core of an operating system is the task scheduler. This component is responsible for determining, according to a predefined policy, which task can execute at what time. These components use unbounded dynamic data structures to store the necessary elements for their operation. These data structures allow elements to be easily moved between them. Verifying a task scheduler requires designing an analysis capable of accurately representing these data structures and their contents.
The first part of this thesis describes a toy imperative language that explicitly manipulates memory. We then provide the concrete semantics of this language, followed by a presentation of a numerical static analysis to determine the range of numerical variables and a shape analysis capable of reasoning about unbounded inductive data structures.
The second part is devoted to presenting a relational abstract domain capable of reasoning about symbolic sequences. This domain expresses constraints on the contents of these sequences, such as their lengths, extreme values, and sorted characteristics.
The third part presents the combination of the shape analysis described in the first part with the sequence domain. This combination enhances the expressiveness of the analysis. It is now capable of proving the partial functional correctness of complex algorithms, such as sorting algorithms on lists or binary trees, as well as list libraries drawn from real applications.
The final part of this thesis presents the application of the analysis to an instance of the FreeRTOS task scheduler. The first step in the verification process is formalizing the properties we seek to establish on the scheduler's functions. The second step aims to show that the specified properties are verified by the instance's functions using the analysis.

KEYWORDS

Abstract interpretation, Operating system, scheduler, separation logic