Abstract Heap Relations for a Compositional Shape Analysis

Hugo Illous

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure

Abstractions Relationnelles de la Mémoire pour une Analyse Compositionnelle de Structures de Données

Abstract Heap Relations for a Compositional Shape Analysis

Soutenue par
Hugo ILLOUS
Le 16 Avril 2019

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Jean-Christophe FILLIÂTRE Directeur de recherche, CNRS	*Rapporteur*
Isabella MASTROENI Professeur, Università degli Studi di Verona	*Rapportrice*
Cezara DRĂGOI Chargée de recherche, ENS & INRIA Paris	*Examinatrice*
Bertrand JEANNET Directeur technique, Argosim	*Examinateur*
Jules VILLARD Ingénieur logiciel, Facebook	*Examinateur*
Francesco ZAPPA NARDELLI Directeur de recherche, ENS & INRIA Paris	*Examinateur*
Xavier RIVAL Directeur de recherche, ENS & INRIA Paris	*Directeur de thèse*
Matthieu LEMERRE Ingénieur-chercheur, CEA LIST	*Co-Directeur de thèse*

1794
ENS | PSL

Résumé

Les analyses statiques ont pour but d'inférer des propriétés sémantiques de programmes. Nous distinguons deux importantes classes d'analyses statiques : les analyses d'états et les analyses relationnelles. Alors que les analyses d'états calculent une sur-approximation de l'ensemble des états atteignables d'un programme, les analyses relationnelles calculent des propriétés fonctionnelles entre les états d'entrée et les états de sortie d'un programme. Les analyses relationnelles offrent plusieurs avantages, comme leur capacité à inférer des propriétés sémantiques plus expressives par rapport aux analyses d'états. De plus, elles offrent également la possibilité de rendre l'analyse compositionnelle, en utilisant les relations entrée-sortie comme des résumés de procédures, ce qui est un avantage pour le passage à l'échelle. Dans le cas des programmes numériques, plusieurs analyses ont été proposées qui utilisent des domaines abstraits numériques relationnels, pour décrire des relations. D'un autre côté, modéliser des abstractions de relations entre les états mémoires entrée-sortie tout en prenant en compte les structures de données est difficile. Dans cette Thèse, nous proposons un ensemble de nouveaux connecteurs logiques, reposant sur la logique de séparation, pour décrire de telles relations. Ces connecteurs peuvent exprimer qu'une certaine partie de la mémoire est inchangée, fraîchement allouée, ou désallouée, ou que seulement une seule partie de la mémoire a est modifiée (et de quelle manière). En utilisant ces connecteurs, nous construisons un domaine abstrait relationnel et nous concevons une analyse statique compositionnelle par interprétation abstraite qui sur-approxime des relations entre des états mémoires contenant des structures de données inductives. Nous avons implémenté ces contributions sous la forme d'un plug-in de l'analyseur Frama-C. Nous en avons évalué l'impact sur l'analyse de petits programmes écrits en C manipulant des listes chaînées et des arbres binaires, mais également sur l'analyse d'un programme plus conséquent qui consiste en une partie du code source d'Emacs. Nos résultats expérimentaux montrent que notre approche permet d'inférer des propriétés sémantiques plus expressives d'un point de vue logique que des analyses d'états. Elle se révèle aussi beaucoup plus rapide sur des programmes avec un nombre conséquent d'appels de fonctions sans pour autant perdre en précision.

Abstract

Static analyses aim at inferring semantic properties of programs. We distinguish two important classes of static analyses: state analyses and relational analyses. While state analyses aim at computing an over-approximation of reachable states of programs, relational analyses aim at computing functional properties over the input-output states of programs. Relational analyses offer several advantages, such as their ability to infer semantics properties more expressive compared to state analyses. Moreover, they offer the ability to make the analysis compositional, using input-output relations as summaries for procedures, which is an advantage for scalability. In the case of numeric programs, several analyses have been proposed that utilize relational numerical abstract domains to describe relations. On the other hand, designing abstractions for relations over input-output memory states and taking shapes into account is challenging. In this Thesis, we propose a set of novel logical connectives to describe such relations, which rely on separation logic. This logic can express that certain memory areas are unchanged, freshly allocated, or freed, or that only part of the memory is modified (and how). Using these connectives, we build an abstract domain and design a compositional static analysis by abstract interpretation that over-approximates relations over memory states containing inductive structures. We implement this approach as a plug-in of the Frama-C analyzer. We evaluate it on small programs written in C that manipulate singly linked lists and binary trees, but also on a bigger program that consists of a part of Emacs. The experimental results show that our approach allows to infer more expressive semantic properties than states analyses, from a logical point of view. It is also much faster on programs with an important number of function calls without losing precision.

Remerciements

Cette section sera rédigée ultérieurement.

Table of Contents

Chapter 1

Introduction

This chapter presents the context and the motivations for this Thesis. First, Section 1.1 makes a global presentation of program analysis. Then, Section 1.2 presents several verification techniques, describing their advantages and drawbacks. The different classes of semantic properties that aim at proving program analyses, specifically traces, relational and states properties, are introduced in Section 1.3. These classes of properties are defined using corresponding semantics, and compared between them using a hierarchy. Section 1.4 focuses on state and relational properties in the case of numeric and memory analyses. In particular, it insists on the fact that relational shape properties are few. Finally, Section 1.5 describes the contributions of this Thesis and establishes its outline.

1.1 Program Analysis and Verification

1.1.1 Motivations

Nowadays, programs are ubiquitous and are increasely used in all branches of activity of our industry, such as aeronautics, automotive, medical or nuclear. Moreover, besides their number, the programs themselves can be very complex and huge. For instance, in 2011, the Linux kernel consisted in 15 million lines of code. Both this immensity and complexity are propitious to *bugs*. A bug is a design error of a program that leads to an unexpected behavior or that may produce an incorrect result. The severity of a bug differs according to what the program is designed for. If a text editor crashes, the unsaved data before the crash may be lost, but it should be possible to simply restart this program and to use it without any difficulty. However, if the program of a nuclear power plant contains a bug, the consequences can be disastrous economically, ecologically and humanly.

To detect bugs in a program, the most natural approach is to review the program.

However, inspecting manually each line of code of program is a hard, painful and costly task, and it cannot guarantee formally that the program acts exactly the way we expect. Indeed, nothing ensures that the person that checked the program found all the bugs it contained.

Program analysis automates the verification process of a program, by writing another program. One of its mains goals is to ensure that a program does what it is supposed to do, to avoid potential (sever) errors during its execution.

1.1.2 Decidability, Soundness and Completeness

Generally, program analysis aims at computing semantic properties of programs. However, Rice's theorem [Ric53] states that computing such properties is *undecidable*: its semantics is not computable. In other words, there is no program analysis that can ensure that every programs are correct. On the other hand, it is possible to compute semantic properties specific to some kinds of programs with *sound* or *complete* program analyses.

A sound program analysis ensures that if no error has been found by the analysis, then the program is definitely correct. However, if an error is found, it cannot ensure that this later is a true error, but only that it may be an error. Conversely, when a complete program analysis finds an error, this latter is a real error. However, it cannot ensure the absence of errors for the whole program whereas a sound program analysis can.

Thus, some program analyses are only sound like abstract interpretation [CC77, CC92], or only complete like runtime verification [LS09]. Also, there are program analyses that are both sound and complete like deductive verification [Flo67] or model checking [EC80, CES86], but the analyzed programs are limited. All these program verifications are briefly described in Section 1.2.

1.2 Main Verification Techniques

We can distinguish two main classes of program analysis: the *dynamic analyses* and the *static analyses*. While dynamic analyses are performed during the execution of the program, static analyses are performed without executing it. In this section, we describe a (non-exhaustive) list of verification techniques: runtime verification, as dynamic analysis, and deductive verification, model checking and abstract interpretation as static analyses.

1.2.1 Runtime Verification

Runtime verification [LS09] consists on checking formal properties during the execution of a program. As a dynamic analysis, it does not ensure that the program is correct for any input, but only for the inputs from which it has been executed. However, when

a property is not verified, this proves that the program is not correct. Moreover, it is automatic and easier to use than other verification techniques.

1.2.2 Deductive Verification

Deductive verification [Flo67] consists in taking a program with its specification, given as a pre and postcondition [Hoa69] and other annotations such as loop invariants, and to generate correction theorems of this programs. These theorems allow to formalize the expected behavior of the program. The proof of these theorems are discarded to interactive theorem provers, automatic theorem provers, or SMT solvers. While deductive verification is both sound and complete, it is limited to annotated programs. Annotating programs involves to understand exactly what the program is supposed to do, and expressing its specification is often a very hard and long task. Furthermore, this technique is too costly: to prove a program of many million lines, it requires to understand and annotate it completely.

1.2.3 Model Checking

Model Checking [EC80, CES86] verifies if the model of a system satisfies some properties. Typically, the model is represented as an automaton with states and transitions and the properties to verify are described in *temporal logic* [Pnu77]. The model checker tries to prove these properties by searching exhaustively and automatically all the possible reachable states of the execution of the system. If a property is not verified, a counter example can be generated. Model checkers are generally both sound and complete, but are limited to finite systems. Moreover, some model checkers can reason over infinite systems, but give up on soundness (e.g. bounded model checking) or completeness.

1.2.4 Abstract Interpretation

In 1977, Patrick Cousot and Radhia Cousot introduced the abstract interpretation frameworks [CC77, CC92]. This framework relies on defining a computable *abstract semantics* $[\![\mathtt{P}]\!]^\sharp$ that is a sound over-approximation of the *concrete semantics* $[\![\mathtt{P}]\!]$ of a program `P`. The relation between the concrete and the abstract semantics is established with a *Galois connection*.

Definition 1.1 (Galois connection). *Let $(\mathcal{A}, \leqslant_{\mathcal{A}})$ and $(\mathcal{B}, \leqslant_{\mathcal{B}})$ be two partially ordered sets. A Galois connection between these two sets consists of two monotone functions: an abstraction function $\alpha \in \mathcal{A} \to \mathcal{B}$ and a concretization function $\gamma \in \mathcal{B} \to \mathcal{A}$, such that $\forall a \in \mathcal{A}$ and $\forall b \in \mathcal{B}$, we have:*

$$a \leqslant_{\mathcal{A}} \gamma(b) \quad \Longleftrightarrow \quad \alpha(a) \leqslant_{\mathcal{B}} b$$

We observe that if an element a of $\mathcal{A}$ is lower than the concretization of an element b of $\mathcal{B}$, the abstraction of a will be also lower than b, and reciprocally. In the case of abstract interpretation, the concrete semantics of a program P returns the set of *all* the behaviors of P. Its binary relation is thus the set inclusion $\subseteq$. Regarding to the abstract semantics, the binary relation is noted $\sqsubseteq^\sharp$ and depends on how the abstract semantics is defined. Defining a Galois connection between the concrete semantics $(\llbracket \mathtt{P} \rrbracket, \subseteq)$ and the abstract semantics $(\llbracket \mathtt{P} \rrbracket^\sharp, \sqsubseteq^\sharp)$ allows to have the following property:

$$\llbracket \mathtt{P} \rrbracket \subseteq \gamma(\llbracket \mathtt{P} \rrbracket^\sharp) \quad \Longleftrightarrow \quad \alpha(\llbracket \mathtt{P} \rrbracket) \sqsubseteq^\sharp \llbracket \mathtt{P} \rrbracket^\sharp$$

This means that $\llbracket \mathtt{P} \rrbracket$ is an *over-approximation* of $\llbracket \mathtt{P} \rrbracket^\sharp$.

Generally, the goal of a static analysis is to prove semantic properties. A semantic property Prop describes the set of the *desired* behaviors for a program P. To prove that Prop is satisfied by P, we need to prove that the concrete semantics of P is included in Prop:

$$\llbracket \mathtt{P} \rrbracket \subseteq \mathsf{Prop}$$

However, $\llbracket \mathtt{P} \rrbracket$ is not computable and such an inclusion cannot be checked. The abstract interpretation framework, as for it, can compute the abstract semantics $\llbracket \mathtt{P} \rrbracket^\sharp$ of P to prove the semantic property. Thus, the semantic property can be proven using the concretization function of the Galois connection:

$$\gamma(\llbracket \mathtt{P} \rrbracket^\sharp) \subseteq \mathsf{Prop}$$

The property Prop is well proven because we have $\llbracket \mathtt{P} \rrbracket \subseteq \gamma(\llbracket \mathtt{P} \rrbracket^\sharp)$.

As the abstract interpretation reasons on an over-approximation of the concrete semantics, the choice of this later is crucial to prove specific semantics properties.

1.3 Semantic Properties

In this section, we provide a (non comprehensive) list of semantic properties and we define relevant concrete semantics that can express them. Before defining them, we define a model of program that is at the base of the following program semantics.

1.3.1 A Model of Programs: Transition System

A program can be characterized by a set of *states* and a *transition relation*. A state $s \in \mathbb{S}$ describes a program status at an instant of its execution. A transition relation $s_0 \rightarrow s_1$, such as $\rightarrow \subseteq \mathcal{P}(\mathbb{S} \times \mathbb{S})$, describes that we can move from the state s_0 to the state s_1. Note that both the set of states $\mathbb{S}$ and the set of transitions $\rightarrow$ may be infinite. The pair $\mathcal{S} = (\mathbb{S}, \rightarrow)$ defines a *transition system*. Figure 1.1 shows a program using this transition system with $\mathbb{S} = \{s_0, s_1, s_2\}$ and $\rightarrow = \{(s_0, s_1), (s_0, s_2), (s_1, s_2), (s_2, s_1)\}$.

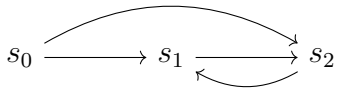


Figure 1.1: Some program for the transition system $\mathcal{S} = (\mathbb{S}, \rightarrow)$ with $\mathbb{S} = \{s_0, s_1, s_2\}$. We admit that the state s_0 is the initial state, and the s_0, s_1 and s_2 are all final states.

1.3.2 Classes of Semantics and Properties

Trace Semantics and Properties

A *trace* allows to represent an execution of a program as a *sequence* of program states. There are two kinds of traces. A *finite trace* is a finite sequence of program states $s_0, \ldots, s_n$ and is noted $\langle s_0, \ldots, s_n \rangle$. An *infinite trace* is an infinite sequence of states and is noted $\langle s_0, \ldots \rangle$. We write $\mathbb{S}^*$ for the set of finite traces, $\mathbb{S}^\omega$ for the set of infinite traces and $\mathbb{S}^\infty = \mathbb{S}^* \cup \mathbb{S}^\omega$ for the set of finite or infinite traces.

Both the finite and infinite traces define their own semantics. To define them, we consider the transition system $\mathcal{S} = (\mathbb{S}, \rightarrow)$.

Definition 1.2 (Semantics of finite traces). *The finite traces semantics* $[\![\mathcal{S}]\!]^*_{\mathcal{T}} \in \mathcal{P}(\mathbb{S}^*)$ *is defined by:*

$$[\![\mathcal{S}]\!]^*_{\mathcal{T}} = \{\langle s_0, \ldots, s_n \rangle \in \mathbb{S}^* \mid \forall i, s_i \rightarrow s_{i+1}\}$$

Definition 1.3 (Semantics of infinite traces). *The infinite traces semantics* $[\![\mathcal{S}]\!]^\omega_{\mathcal{T}} \in \mathcal{P}(\mathbb{S}^\omega)$ *is defined by:*

$$[\![\mathcal{S}]\!]^\omega_{\mathcal{T}} = \{\langle s_0, \ldots \rangle \in \mathbb{S}^\omega \mid \forall i, s_i \rightarrow s_{i+1}\}$$

The semantics of finite or infinite traces is simply the union of the semantics of finite and infinite traces.

Definition 1.4 (Semantics of traces). *The traces semantics* $[\![\mathcal{S}]\!]^\infty_{\mathcal{T}} \in \mathcal{P}(\mathbb{S}^\infty)$ *is defined by:*

$$[\![\mathcal{S}]\!]^\infty_{\mathcal{T}} = [\![\mathcal{S}]\!]^*_{\mathcal{T}} \cup [\![\mathcal{S}]\!]^\omega_{\mathcal{T}}$$

A trace property $\mathsf{Prop}_{\mathcal{T}}$ is a set of traces such that $\mathsf{Prop}_{\mathcal{T}} \subseteq \mathbb{S}^\infty$. It is verified if and only if all traces belong to $\mathsf{Prop}_{\mathcal{T}}$:

$$[\![\mathcal{S}]\!]^\infty_{\mathcal{T}} \subseteq \mathsf{Prop}_{\mathcal{T}}$$

The decomposition theorem [AS87] states that any trace property can be decomposed into the conjunction of a *safety property* and a *liveness property*. A safety property is a

property that specifies that some finite "bad" behavior will never happen. For instance, "the program does not reach the state s' after having visited the state s" is a safety property. A liveness property is a property that specifies that some "good" finitely observable behavior will eventually happen. For instance, "state s will eventually be reached by all executions" is a liveness property.

Relational Semantics and Properties

A *relational semantics* computes the set of the *input-output* states of a program [MT91]. We consider the transition system $\mathcal{S} = (\mathbb{S}, \rightarrow)$ to define it.

Definition 1.5 (Relational Semantics). *The relational semantics* $[\![\mathcal{S}]\!]_{\mathcal{R}} \in \mathcal{P}(\mathbb{S} \times \mathbb{S})$ *is defined by:*

$$[\![\mathcal{S}]\!]_{\mathcal{R}} = \{(s_0, s_n) \in \mathbb{S} \times \mathbb{S} \mid \langle s_0, \ldots, s_n \rangle \in [\![\mathcal{S}]\!]^*_{\mathcal{T}}\}$$

A relational property $\mathsf{Prop}_{\mathcal{R}}$ is a set of pairs such as $\mathsf{Prop}_{\mathcal{R}} \subseteq \mathbb{S} \times \mathbb{S}$. The relational property $\mathsf{Prop}_{\mathcal{R}}$ is satisfied if and only if all input and output states belong to $\mathsf{Prop}_{\mathcal{R}}$:

$$[\![\mathcal{S}]\!]_{\mathcal{R}} \subseteq \mathsf{Prop}_{\mathcal{R}}$$

Relational properties allow to describe *functional properties*, such as "the function `abs` inputs an integer `x` and outputs its absolute value". Also, this kind of properties can be expressed with class contract languages [BFM+08, LBR98], that let functions be specified by formulas that may refer both to the input and to the output states. We also observe that all relational properties are safety properties.

State Semantics and Properties

A state semantics computes all the *reachable states* of a program. As usual, we consider the transition system $\mathcal{S} = (\mathbb{S}, \rightarrow)$.

Definition 1.6 (State Semantics). *The state semantics* $[\![\mathcal{S}]\!]_{\mathcal{S}} \in \mathcal{P}(\mathbb{S})$ *is defined by:*

$$[\![\mathcal{S}]\!]_{\mathcal{S}} = \{s_n \in \mathbb{S} \mid \exists s_0 \in \mathbb{S}, \langle s_0, \ldots, s_n \rangle \in [\![\mathcal{S}]\!]^*_{\mathcal{T}}\}$$

A state property is a set of states $\mathsf{Prop}_{\mathcal{S}}$, such as $\mathsf{Prop}_{\mathcal{S}} \subseteq \mathbb{S}$. The state property $\mathsf{Prop}_{\mathcal{S}}$ is satisfied if and only if all reachable states belong to $\mathsf{Prop}_{\mathcal{S}}$:

$$[\![\mathcal{S}]\!]_{\mathcal{S}} \subseteq \mathsf{Prop}_{\mathcal{S}}$$

Absence of runtime errors is a state property, for instance when $\mathsf{Prop}_{\mathcal{S}} = \mathbb{S} \setminus \{\Omega\}$ and Ω is the error state. An other kind of state property is non termination, when $\mathsf{Prop}_{\mathcal{S}} = \{s \in \mathbb{S} \mid \exists s', s \rightarrow s'\}$. Also, like relational properties, all state properties are safety properties.

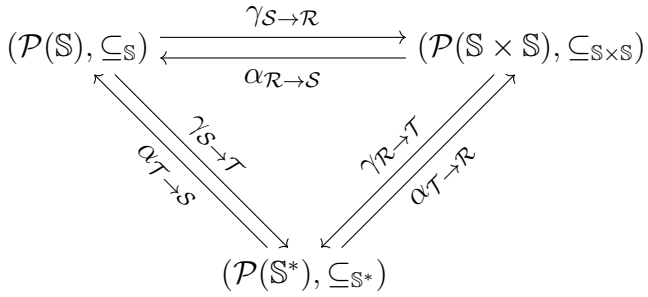


Figure 1.2: Galois connections between state, relational and finite trace semantics.

1.3.3 Hierarchy of Semantics

These different semantics of programs form a hierarchy [Cou97]. With a hierarchy between semantics, we can establish which kind of properties can be proven using which semantics and conversely, which kind of properties cannot be proven using which semantics. This means that we can prove different kind of properties using a single class of semantics. This indicates the importance of the choice of the semantics according to the properties we want to prove.

A hierarchy between semantics can be established with *Galois connections*. Figure 1.2 shows three Galois connections that link the state, relational and finite trace semantics. We consider only finite trace semantics and not infinite trace semantics because the state and relational semantics are built from finite trace semantics. Note that we equip each semantics with its own inclusion operator, $\subseteq_{\mathbb{S}}$ for the state semantics, $\subseteq_{\mathbb{S}\times\mathbb{S}}$ for the relational semantics and $\subseteq_{\mathbb{S}^*}$ for the finite trace semantics. The first Galois connection $(\alpha_{\mathcal{T}\to\mathcal{S}}, \gamma_{\mathcal{S}\to\mathcal{T}})$ abstracts finite traces into states with $\alpha_{\mathcal{T}\to\mathcal{S}}$ and concretizes states into finite traces with $\gamma_{\mathcal{S}\to\mathcal{T}}$. The second Galois connection $(\alpha_{\mathcal{R}\to\mathcal{S}}, \gamma_{\mathcal{S}\to\mathcal{R}})$ abstracts relations into states with $\alpha_{\mathcal{R}\to\mathcal{S}}$ and concretizes states into relations with $\gamma_{\mathcal{S}\to\mathcal{R}}$. Finally, the Galois connection $(\alpha_{\mathcal{T}\to\mathcal{R}}, \gamma_{\mathcal{R}\to\mathcal{T}})$ abstracts finite traces into relations with $\alpha_{\mathcal{T}\to\mathcal{R}}$ and concretizes relations into finite traces with $\gamma_{\mathcal{R}\to\mathcal{T}}$. These Galois connections are defined as follows:

Definition 1.7 (Galois connections between finite traces and states). *Let* $\mathbb{S}_i^* \subseteq \mathbb{S}^*$ *and* $\mathbb{S}_i \subseteq \mathbb{S}$. *The Galois connection* $(\alpha_{\mathcal{T}\to\mathcal{S}}, \gamma_{\mathcal{S}\to\mathcal{T}})$ *is defined such as:*

$$\alpha_{\mathcal{T}\to\mathcal{S}}(\mathbb{S}_i^*) = \{s_k \mid \langle s_0, \ldots, s_n\rangle \in \mathbb{S}_i^* \wedge 0 \leqslant k \leqslant n\}$$

$$\gamma_{\mathcal{S}\to\mathcal{T}}(\mathbb{S}_i) = \{\langle s_0, \ldots, s_n\rangle \mid \forall k, 0 \leqslant k \leqslant n \Rightarrow s_k \in \mathbb{S}_i\}$$

Definition 1.8 (Galois connections between states and relations). *Let* $\mathbb{S}_0 \times \mathbb{S}_n \subseteq \mathbb{S} \times \mathbb{S}$ *and* $\mathbb{S}_i \subseteq \mathbb{S}$. *The Galois connection* $(\alpha_{\mathcal{R}\to\mathcal{S}}, \gamma_{\mathcal{S}\to\mathcal{R}})$ *is defined as follows:*

$$\alpha_{\mathcal{R}\to\mathcal{S}}(\mathbb{S}_0 \times \mathbb{S}_n) = \mathbb{S}_0 \cup \mathbb{S}_n$$

$$\gamma_{\mathcal{S}\to\mathcal{R}}(\mathbb{S}_i) = \{(s_0, s_n) \mid s_0 \in \mathbb{S}_i \wedge s_n \in \mathbb{S}_i\}$$

Definition 1.9 (Galois connections between finite traces and relations). *Let* $\mathbb{S}_i^* \subseteq \mathbb{S}^*$ *and* $\mathbb{S}_0 \times \mathbb{S}_n \subseteq \mathbb{S} \times \mathbb{S}$. *The Galois connection* $(\alpha_{\mathcal{T}\to\mathcal{R}}, \gamma_{\mathcal{R}\to\mathcal{T}})$ *is defined as follows:*

$$\alpha_{\mathcal{T}\to\mathcal{R}}(\mathbb{S}_i^*) = \{(s_0, s_n) \mid \langle s_0, \ldots, s_n\rangle \in \mathbb{S}_i^*\}$$

$$\gamma_{\mathcal{R}\to\mathcal{T}}(\mathbb{S}_0 \times \mathbb{S}_n) = \{\langle s_0, \ldots, s_n\rangle \mid s_0 \in \mathbb{S}_0 \wedge s_n \in \mathbb{S}_n\}$$

Example 1.1 (Abstraction of traces into relations). We consider a set of finite traces for the program in Figure 1.1 and we abstract it into a set of relations:

$$\alpha_{\mathcal{T}\to\mathcal{R}}(\{\langle s_0\rangle, \langle s_0, s_1\rangle, \langle s_0, s_2\rangle, \langle s_0, s_1, s_2\rangle, \langle s_0, s_1, s_2, s_1\rangle\}) = \{(s_0, s_0), (s_0, s_1), (s_0, s_2)\}$$

The trace $\langle s_0\rangle$ is abstracted into the relation (s_0, s_0), the traces $\langle s_0, s_1\rangle$ and $\langle s_0, s_1, s_2, s_1\rangle$ are abstracted into the relation (s_0, s_1), and finally the traces $\langle s_0, s_2\rangle$ and $\langle s_0, s_1, s_2\rangle$ are abstracted into the relation (s_0, s_2).

Example 1.2 (Abstraction of relations into states). We give the abstraction into states of the relations computed in Example 1.1:

$$\alpha_{\mathcal{R}\to\mathcal{S}}((s_0, s_0), (s_0, s_1), (s_0, s_2)) = \{s_0, s_1, s_2\}$$

Example 1.3 (Concretization of states into relations). We now concretize the set of states obtained in Example 1.2 into relations.

$$\gamma_{\mathcal{S}\to\mathcal{R}}(\{s_0, s_1, s_2\}) = \{(s_i, s_j) \mid i, j \in \{0, 1, 2\}\}$$

A relational property $\mathsf{Prop}_{\mathcal{R}}$ can be verified with the finite trace semantics $[\![S]\!]^*_{\mathcal{T}}$, either (1) directly by abstracting the traces into relations, or (2) by concretizing $\mathsf{Prop}_{\mathcal{R}}$ into a trace property, More formally, $\mathsf{Prop}_{\mathcal{R}}$ can be verified by verifying that:

$$\begin{array}{lrcl} (1) & \alpha_{\mathcal{T}\to\mathcal{R}}([\![S]\!]^*_{\mathcal{T}}) & \subseteq_{\mathbb{S}\times\mathbb{S}} & \mathsf{Prop}_{\mathcal{R}} \\ \textit{or} & & & \\ (2) & [\![S]\!]^*_{\mathcal{T}} & \subseteq_{\mathbb{S}^*} & \gamma_{\mathcal{R}\to\mathcal{T}}(\mathsf{Prop}_{\mathcal{R}}) \end{array}$$

Similarly, every state property $\mathsf{Prop}_{\mathcal{S}}$ can be verified with the relational semantics $[\![\mathcal{S}]\!]_{\mathcal{R}}$, either (1') by abstracting the relations into states, or (2') by concretizing $\mathsf{Prop}_{\mathcal{S}}$ into a relational property. Thus, $\mathsf{Prop}_{\mathcal{S}}$ can be verified by verifying that:

$$\begin{array}{lllll} (1') & \alpha_{\mathcal{R}\to\mathcal{S}}([\![\mathcal{S}]\!]_{\mathcal{R}}) & \subseteq_{\mathbb{S}} & \mathsf{Prop}_{\mathcal{S}} \\ \textit{or} & & & \\ (2') & [\![\mathcal{S}]\!]_{\mathcal{R}} & \subseteq_{\mathbb{S}\times\mathbb{S}} & \gamma_{\mathcal{S}\to\mathcal{R}}(\mathsf{Prop}_{\mathcal{S}}) \end{array}$$

By transitivity, every state property can also be verified with the finite trace semantics. However, in general a relational property $\mathsf{Prop}_{\mathcal{R}}$ cannot be verified with the state semantics, since the abstraction function may abstract relations into states that can be concretized into relations that are not in $\mathsf{Prop}_{\mathcal{R}}$. For the same reason, a trace property cannot be verified neither by the relational semantics nor the state semantics.

This means that trace properties are more expressive than relational properties, that are intrinsically more expressive than states.

1.3.4 Compositionality

Until now, we considered the semantics of a program as a *global definition* of the whole system. Consequently, the semantic property inferred by the analysis is valid for the whole program.

Many programming languages allow to decompose *syntactically* programs into sub-programs. For instance, in common imperative programming languages, a program p can be decomposed into a sequence of two sub-programs $p_1; p_2$. If the analysis of a program is able to infer local properties for each of its sub-programs, an interesting feature is the *compositionality* of these properties.

Definition 1.10 (Composable property). *Let p be a program that is syntactically decomposed into two sub-programs p_1 and p_2, and $[\![p]\!]$ its semantics. Let $[\![p]\!] = \mathsf{Prop}$, $[\![p_1]\!] = \mathsf{Prop}_1$ and $[\![p_2]\!] = \mathsf{Prop}_2$, the property Prop is compositional if it exists an operator $\circ$ such as:*

$$\mathsf{Prop} = \mathsf{Prop}_1 \circ \mathsf{Prop}_2$$

Composable properties allow to make the analyses *modular* [CC02, PC06, JLRS10, CRL99, CDOY09] and *compositional*. Indeed, to analyze a sequence of two sub-programs, the analysis can simply analyze each sub-program separately, and compose the resulting properties. When sub-programs are functions, the analysis may analyze each function separately, and compute one summary per function, so that the analysis of a function call does not require re-analyzing the body of the function, which is an advantage for *scalabilty*.

Finite traces and relational properties are composable properties. However, state properties (thus for sets of states) are not. Consequently, to perform modular and compositional analyses, we must *at least* base our analysis on a relational semantics, that infers composable properties. We finally define the composition operator of the finite traces and relations.

Definition 1.11 (Composition of finite traces). *Let $\mathbb{S}_i^*, \mathbb{S}_j^* \subseteq \mathbb{S}^*$. The composition operator $\circ_{\mathcal{T}} \in \mathcal{P}(\mathbb{S}^*) \times \mathcal{P}(\mathbb{S}^*) \to \mathcal{P}(\mathbb{S}^*)$ of finite traces is defined as follows:*

$$(\mathbb{S}_i^*) \circ_{\mathcal{T}} (\mathbb{S}_j^*) = \{\langle s_0, \ldots, s_n\rangle \mid \exists s_k, \langle s_0, \ldots, s_k\rangle \in \mathbb{S}_i^* \wedge \langle s_k, \ldots, s_n\rangle \in \mathbb{S}_j^*\}$$

Definition 1.12 (Composition of relations). *Let $\mathbb{S}_0, \mathbb{S}_i, \mathbb{S}_j, \mathbb{S}_n \subseteq \mathbb{S}$, the composition operator $\circ_{\mathcal{R}} \in \mathcal{P}(\mathbb{S} \times \mathbb{S}) \times \mathcal{P}(\mathbb{S} \times \mathbb{S}) \to \mathcal{P}(\mathbb{S} \times \mathbb{S})$ of relations is defined as follows:*

$$(\mathbb{S}_0 \times \mathbb{S}_i) \circ_{\mathcal{R}} (\mathbb{S}_j \times \mathbb{S}_n) = \{(s_0, s_n) \mid \exists s_k, (s_0, s_k) \in (\mathbb{S}_0 \times \mathbb{S}_i) \wedge (s_k, s_n) \in (\mathbb{S}_j \times \mathbb{S}_n)\}$$

We remark that for both the finite traces and the relations, the properties of the two sub-programs must share states s_k in common, that are the output state of the first sub-program and the input state of the second sub-program.

1.4 Numeric and Memory Analyses

Previously, we saw different classes of semantic properties: trace, relational and state properties. This section focuses on the different ways to express state and relational properties for both numeric and memory analyses. A numeric analysis aims at computing properties about the different values of a program, for instance, integer or floating values. On the other hand, a memory analysis aims at computing properties about pointers and data structures. Regarding to memory, we will see that most state of the art memory analyses (more precisely shape analyses) compute only state properties.

1.4.1 Numeric Analysis

A *numeric analysis* is a program analysis that discovers and verifies properties about the values of a program. Let us consider the following program, that computes the sum of all the values from 0 to an integer `n`.

```
1 int sum = 0;
2 while(n >= 0) {
3   sum = sum + n;
4   n = n - 1;
5 }
```

At the end of this program, it is obvious that the value of variable `sum` is positive or null (we admit that the variable `n` has been well initialized). This property can be verified by a state analysis, using a description of all the possible values of `sum` at each program state such as intervals [CC76] or signs [CC77].

Remind that the sum from 0 to n is equal to $(n \times (n+1))/2$. Regarding to our program, this is a relational property. Indeed, it means that the output value of the variable `sum` is equal to the half of the product between the input value of variable `n` and the input value of variable $\mathtt{n}+1$. A common way to express numeric relations between input and output states consists in defining for each variable `x` a primed version $\mathtt{x}'$ that describes the value of `x` in the output state whereas the non-primed version denotes the value of `x` in the input state. Applied to our program, primed variables $\mathtt{sum}'$ and $\mathtt{n}'$ describe respectively the output values for variables `sum` and `n` .

In this context, state descriptions of values such as intervals or congruences [Gra89] cannot capture any interesting relation between input and output states. Conversely, relational numerical abstractions such as convex polyhedra [CH78], affine equalities [Kar76] or octagons [Min06] can effectively capture relations between input and output states, as shown in [PC06, ACI10]. Moreover, the relational numerical abstraction of [RCK07] can capture the relation $\mathtt{sum}' = (\mathtt{n} \times (\mathtt{n}+1))/2$, when applied to our example program.

1.4.2 Pointer and Alias Analyses

Pointer Analysis. A pointer analysis is a program analysis that attempts to determine which pointers can point to which memory locations. For the following program, a pointer analysis should infer that the pointer `p` points to either the address of `x` or the address of `y`.

```
1 int x;
2 int y;
3 int *p;
4 if(...) {p = &x;} else {p = &y;}
```

Two examples of pointer analyses are the Steensgaard's [Ste96] and Andersen's [And94] pointer analyses.

Alias Analysis. Pointer analyses are specific cases of *alias analyses*. Alias analyses differ from pointer analyses in the sense that they aim at specifying if two l-value expressions designate the same memory location. In the program below, the l-values $*\mathtt{p}$ and $*\mathtt{q}$ are aliased if `p` and `q` point to the same memory location. If they are aliased, the instruction $*\mathtt{q} = 2$; also assigns the value 2 to $*\mathtt{p}$, and thus, the value of `x` will be 5. If they are not aliased, the value of $*\mathtt{p}$ will not be modified and the value of `x` will be 4.

```
1 *p = 1;
2 *q = 2;
```

```
1  typedef struct list { struct list * next; int data; } list;
2
3  void insert_non_empty( list *l, int v ) {
4    list *c = l;
5    while( c->next != NULL && ... ){
6      c = c->next;
7    }
8    list *e = malloc( sizeof( list ) );
9    e->next = c->next; c->next = e; e->data = v;
10 }
```

Figure 1.3: A list insertion program

```
3 int x = *p + 3;
```

Most of the state of the art alias analyses such as [LH88, JM82, CBC93, CWZ90, HN90, HHN92, Deu92, LR92, Deu94] are state analyses. Indeed, alias are binary relations, but inside a single program state and not between an input state and an output state. In the last years, [DDAS11] proposed a relational alias analysis, that infers precise and compact procedure summaries describing all the possible configurations at the end of a procedure accordingly to the different alias configurations at the beginning of the procedure.

Generally, alias analyses are able to compute alias information for recursive and dynamically allocated data structures. However, they are not expressive enough to capture properties such as structural invariant or memory safety for such data structures. Therefore, more expressive analyses, named *shape analyses*, have been proposed to capture such properties.

1.4.3 Shape Analysis

A shape analysis aims at inferring complex structural invariants and proving functional properties for programs manipulating dynamically allocated data structures, like linked lists or trees. Let us consider the program of Figure 1.3, which implements the insertion of an element inside a non-empty singly linked list containing integer values. When applied to a pointer to an existing non-empty list `l` and an integer `v`, this function traverses it partially (based on a condition that is elided in the figure). It then allocates a new list element, insert it a the selected position, and assigns to its `data` field the value of the variable `v`.

State Shape Analysis. A state analysis for this program consists of capturing the fact that the input list must be a *non-empty well formed* linked list and that no null pointer or dangling pointer is ever dereferenced. Moreover, it should express that the output list is a well formed linked list of at least two consecutive elements.

Several shape analyses have been proposed like automata-based shape analyses [HHR$^+$11, HHL$^+$15] that use automata to represent memory states, graph-based shape analyses [GH96, LAIS06, MHKS08, BDES09] that describe memory states as graphs, and the three-valued logic based shape analysis [SRW99, LAS00, SRW02] (TVLA) that uses the Kleene's 3-valued logic [Kle52] to describe memory states. Also, separation logic [Rey02] provides an elegant description for memory states and is at the foundation of many analyses for heap properties [DOY06, CR08, BCO05, GVA07]. In particular, the separating conjunction connective $*$ expresses that two memory regions are disjoint and allows local reasoning. All of these shape analyses infer memory state properties, they cannot describe memory relations.

Relational Shape Analysis. For the program of Figure 1.3, a relational analysis could capture the fact that both the part of the list that is traversed and the tail of the list are not modified at all. More precisely, they are respectively *physically* equal (the same memory cells containing the same data) in the input and output states. Moreover, it could express that the new element has been freshly allocated by the function; that the modified node pointed to the tail of the list and now points to the allocated node.

To our knowledge, there is only one existing work that is able to compute such relational properties [JLRS04]. It consists of an extension of the TVLA framework that uses a doubled vocabulary that describes relations between the input and output states. On the other hand, we observe that there is no similar work that extends and takes benefits of the precise abstract states described by separation logic to abstract relations.

1.5 Contributions and Outline

In this Thesis, we propose a set of new logical connectives inspired by separation logic, that can describe relational shape properties. These connectives abstract relations (sets of pairs of states) instead of just sets of states, and seek for compositional static analysis algorithms. These connectives can express that a memory region has been left unmodified by a program fragment, or that memory states can be split into disjoint sub-regions that undergo different transformations. We build a relational shape abstract domain upon these connectives, and apply it to design a static analysis by abstract interpretation for programs manipulating dynamic data structures. This static analysis can compute expressive relational shape properties, and takes benefit of the inferred relations to make the analysis modular and compositional, without losing too much precision.

- Chapter 2 provides a global overview of this Thesis through a motivating example.

- Chapter 3 presents the target programming language of our static analysis. It consists of a C-like imperative programming language. We define its syntax, its memory states and its relational semantics.
- In Chapter 4, we first formalize a heap states abstraction based on separation logic. We then design a relational heap abstraction based on new relational logic connectives, the operands of which are abstract heaps. We finally define an abstraction for memory relations.
- In Chapter 5, we design a static analysis by abstract interpretation, that infers abstract memory relations. This relational analysis is intra-procedural: it does not handle function calls.
- In Chapter 6 we propose a generic extension of abstract heap relations defined in Chapter 4 that allows to capture more precise heap relations. We also integrate this extension in the relational intra-procedural analysis of Chapter 5.
- In Chapter 7, we formalize an operator, that over-approximates the composition of relations without loosing too much precision.
- In Chapter 8, we lift the relational intra-procedural analysis into a compositional inter-procedural analysis. This analysis still infers relational heap properties, but uses them as function summaries in order to compose them at call site instead of re-analyzing the function. We extend this analysis in Chapter 9 to handle recursive functions.
- Finally, in Chapter 10 we conclude this Thesis and we discuss perspectives for future directions.

Chapter 2

Overview

In this chapter, we present an overview of the Thesis. We first present a relational intra-procedural shape analysis, without function calls. This later introduces abstract heap relations, that describe relations between the input and output heap regions of a program. We then propose an extension of abstract heap relations, called abstract heap transformation predicates, that improves their precision. Finally, we present a compositional inter-procedural shape analysis, that composes functions when they are called instead of re-analyzing them. It mainly relies on a composition operator over abstract heap relations, that takes benefits from these relations to preserve information about the calling context.

2.1 Relational Intra-procedural Analysis

In this section, we present the relational shape analysis that computes relations between the input and output states of programs. This kind of analysis infers stronger properties than non-relational analyses.

To illustrate it, we consider the code shown in Figure 2.1 which implements the concatenation of two singly linked lists, `l1` and `l2`. When the list `l1` is empty, the function `concat` simply returns the second list `l2`. When the list `l1` is non-empty, the function traverses it until its last element, and makes its `next` field points to `l2`.

Our analysis is based on a relational shape abstract domain, inspired by separation logic [Rey02]. The abstract domain is formally defined in Chapter 4 and the relational analysis in Chapter 5.

2.1.1 Abstract Heaps and the Needs for Relations

We first present an abstraction of memory heaps based on separation logic [Rey02] and demonstrate why we require relations to describe more precisely the behaviour of pro-

```
1  list *concat(list *l1, list *l2) {
2    if (l1 == NULL) {
3      return l2;
4    }
5    list *t = l1;
6    while(t->next != NULL) {
7      t = t->next;
8    }
9    t->next = l2;
10   return l1;
11 }
```

Figure 2.1: Concatenation of two linked lists

grams.

Abstract Heaps

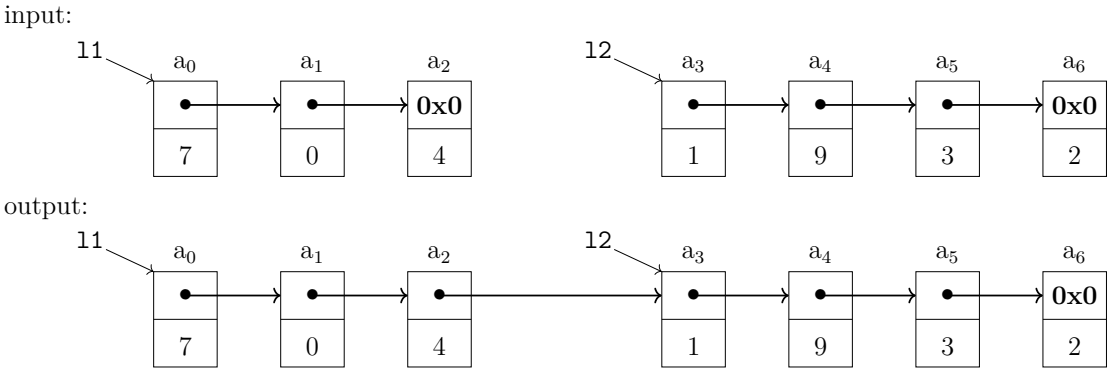
Separation logic[Rey02] relies on separating conjunction $*$ that expresses disjoint heap regions and allows local reasoning on complex data structures. If $\mathrm{h}_0^\sharp$ and $\mathrm{h}_1^\sharp$ describe two memory regions, $\mathrm{h}_0^\sharp * \mathrm{h}_1^\sharp$ ensures that $\mathrm{h}_0^\sharp$ and $\mathrm{h}_1^\sharp$ are disjoint.

We first discuss the Figure 2.2(a) that shows two possible concrete input lists for the function `concat`. The first one contains three list nodes whose addresses are a_0, a_1 and a_2. The second one contains four list nodes whose addresses are a_3, a_4, a_5 and a_6. These two lists can both be abstracted with *inductive predicates*, as used in [DOY06, CR08]. We assume that $\mathbf{list}(\alpha)$ describes heap regions that consist of a well-formed singly linked list starting at address α (α is a symbolic value used in the abstraction to denote a concrete address). This predicate is intuitievly defined by induction as follows:

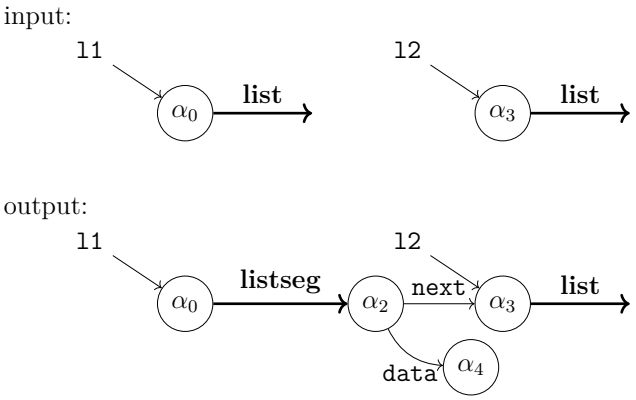
$$\begin{array}{rcl} \mathbf{list}(\alpha) & = & (\mathbf{emp}, \alpha = 0) \\ & \vee & (\alpha \cdot \texttt{data} \mapsto \delta * \alpha \cdot \texttt{next} \mapsto \beta * \mathbf{list}(\beta), \alpha \neq 0) \end{array}$$

It means either the region is empty (the predicate **emp** describes an empty heap region) and α is the null pointer, or the region is not empty and consists of a list element at address α with two fields, `data` and `next`. The `data` field contains a value described by the symbolic value δ, and the `next` field contains a value described by the symbolic value β, which is itself a region described by the predicate $\mathbf{list}(\beta)$. Thus, the two inputs lists can be abstracted with the abstract heap $\mathbf{list}(\alpha_0) * \mathbf{list}(\alpha_3)$, the graphical representation of which is shown in the top of Figure 2.2(b). The symbolic values α_0 and α_3 respectively denote the concrete addresses a_0 and a_3.

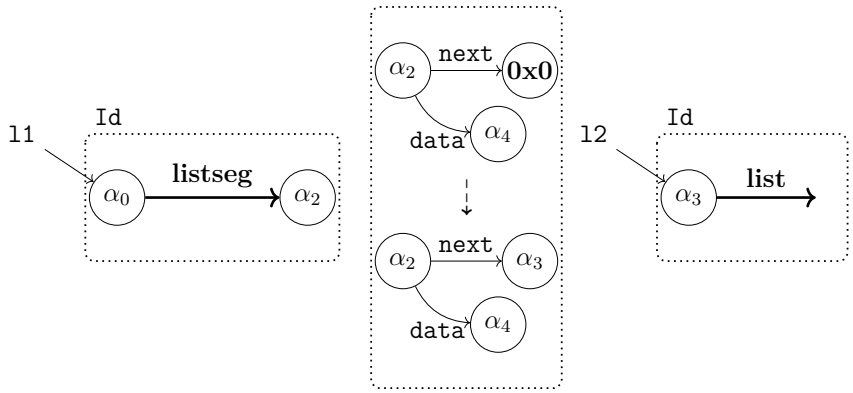
The bottom of Figure 2.2(a) shows the resulting output state of the concatenation function. We observe that the last node of the list `l1` points to the first node of the



(a) A pair of concrete input and output states



(b) A pair of abstract input and output states



(c) Abstract input-output relations

Figure 2.2: Abstract states and relations corresponding to the cases where `l1` is non-empty for the function `concat` of Figure 2.1

list `l2`. Thus, the analysis needs to describes a heap region starting at an address and ending at another. This requires a list segment predicate $\mathbf{listseg}(\alpha, \beta)$, that is defined in a similar way as for $\mathbf{list}(\alpha)$:

$$\begin{array}{rcl} \mathbf{listseg}(\alpha, \beta) & = & (\mathbf{emp}, \alpha = \beta) \\ & \vee & (\alpha \cdot \texttt{data} \mapsto \delta * \alpha \cdot \texttt{next} \mapsto \gamma * \mathbf{listseg}(\gamma, \beta), \beta \neq 0) \end{array}$$

It describes a region that stores a list starting at address α and whose the last element points to β. If the list is empty, we have $\alpha = \beta$. Using this predicate, we can use the separation logic formula $\mathbf{listseg}(\alpha_0, \alpha_2) * \alpha_2 \cdot \texttt{data} \mapsto \alpha_4 * \alpha_2 \cdot \texttt{next} \mapsto \alpha_3 * \mathbf{list}(\alpha_3)$ to abstract the output states of the concatenation function (in the case where `l1` is non-empty), as shown in the bottom of Figure 2.2(b). Indeed, it describes a list of any length whose last element points to α_2, that itself points to α_3.

The Needs for Relations

We also observe that this abstraction allows to express and verify that the function is memory safe, and returns a well-formed linked list. First, it captures the fact that no null or dangling pointer is ever dereferenced. Second, all states described by the abstract post-condition consist of a well-formed list, followed by an other well-formed linked list. On the other hand, it does not express anything about the locations of the lists in the output state with respect to the lists in the input state. More precisely, it cannot capture the fact that the elements stored at addresses a_3, a_4, a_5 and a_6 are left unmodified physically. Then, it does not express that the modified element (here at address a_2) belongs to the input list pointed by `l1`. Finally, in Figure 2.2(b), the input abstract state just describes two well-formed list, starting respectively at address α_0 and at address α_3, and the output abstract state just describes a well-formed list starting at address α_0 followed by a list node whose address is α_2, that points to an other list starting at address α_3, but nothing more. This is a consequence of the fact that each abstract state in Figure 2.2(b) independently describes a set of concrete states.

2.1.2 Abstract Heap Relations

To abstract relations, instead of sets of states, we now propose to define a new logical structure, shown in Figure 2.2(c), that is based on new predicates inspired by separation logic and that partially overlays the abstractions of input and output states. These new predicates are called *abstract heap relations*. They are presented Chapter 4.

First, we observe that the list pointed by `l2` is not modified at all, and that the list pointed by `l1` is also not modified until its last element. Thus, we describe the non-modification with a single predicate $\texttt{Id}(h^\sharp)$, that is named *identity relation*. It denotes pairs made of an input heap and an output heap, that are *physically identical* and that can both be abstracted by $h^\sharp$. For instance, the abstract heap relations $\texttt{Id}(\mathbf{list}(\alpha_3))$ and

$\texttt{Id}(\textbf{listseg}(\alpha_0, \alpha_2))$ ensure respectively that the list starting at address α_3 and the list starting at address α_0 and ending at address α_2 are both not modified by the concatenation function. An identity relation is graphically represented by a box, labeled by the word '`Id`' containing the unmodified abstract heap, as shown on the left and on the right of Figure 2.2(c).

Second, the concatenation function modified the last list node of the first list. In the input state, this node pointed to the null pointer, and in the output state, it points to the first node of `l2`. Thus, we need to describe relations between states where a region has been modified. To account for this, we need a new connective $[\mathrm{h}_0^\sharp \dashrightarrow \mathrm{h}_1^\sharp]$ which is applied to the two abstract heaps $\mathrm{h}_0^\sharp$ and $\mathrm{h}_1^\sharp$, both expressed by formulas in the usual separation logic with inductive predicates. The abstract relation $[\mathrm{h}_0^\sharp \dashrightarrow \mathrm{h}_1^\sharp]$ describes the local transformation of an input heap abstracted by $\mathrm{h}_0^\sharp$ into an output heap abstracted by $\mathrm{h}_1^\sharp$. This abstract relation is called a *transform-into relation*. This represented in the middle of Figure 2.2(c) by a box containing a dashed arrow that separates the input and output states.

Finally, we need to define a counterpart for separating conjunction at the relation level. Indeed, the effect of the concatenation function can be decomposed as its effect of the list pointed by `l1` (which only last node is modified) and the list pointed by `l2` (which is left unmodified). This *relational separating conjunction* is noted $*_{\mathtt{R}}$. If $\mathrm{r}_0^\sharp$ and $\mathrm{r}_1^\sharp$ are two abstract heap relations, then $\mathrm{r}_0^\sharp \ *_{\mathtt{R}} \ \mathrm{r}_1^\sharp$ ensures that $\mathrm{r}_0^\sharp$ and $\mathrm{r}_1^\sharp$ describe totally independent relations. To avoid confusion, from now on, we write $*_{\mathtt{S}}$ for the usual separating conjunction. Thus, a valid abstract heap relation for the concatenation function for the case where `l1` is non-empty is:

$$\begin{array}{l}\texttt{Id}(\textbf{listseg}(\alpha_0, \alpha_2)) \\ *_{\mathtt{R}} \ [(\alpha_2 \cdot \texttt{data} \mapsto \alpha_4 \ *_{\mathtt{S}} \ \alpha_2 \cdot \texttt{next} \mapsto \alpha_5) \dashrightarrow (\alpha_2 \cdot \texttt{data} \mapsto \alpha_4 \ *_{\mathtt{S}} \ \alpha_2 \cdot \texttt{next} \mapsto \alpha_3)] \\ *_{\mathtt{R}} \ \texttt{Id}(\textbf{list}(\alpha_3))\end{array}$$

To describe precisely the different cases of the function `concat` (where `l1` is empty and non-empty), we use a *disjunction* of abstract heap relations $\mathrm{r}_0^\sharp \vee \mathrm{r}_1^\sharp$, where its disjunct corresponds to one case. Remark that the case where `l1` is empty is simply described by $\texttt{Id}(\textbf{list}(\alpha_3))$.

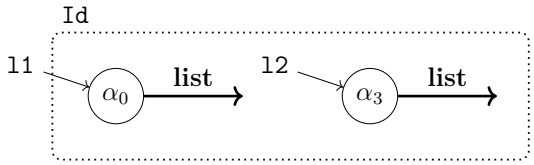
2.1.3 Analysis Algorithm

We now present informally how our analysis use these new connectives to compute a sound over-approximation of the input-output memory states relations. We also do not consider function calls.

Forward abstract interpretation. The analysis algorithm proceeds by forward abstract interpretation [CC77]. It computes for each program point an abstract relation between the input memory state and the memory state at this current program point.

Thus, when the analysis reaches the last program point, it has computed an abstract relation between the input and output memory states of the program.

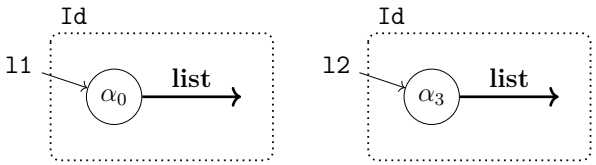
Initial abstract relation. The analysis starts with the identity abstract heap relation of a given pre-condition at function entry. Indeed, the first abstract relation of the analysis should describe an abstract relation between the input state and the input state itself, and this can be described by the identity relation. For instance, the analysis of the concatenation function starts with the abstract heap relation $\texttt{Id}(\mathbf{list}(\alpha_0) *_{\texttt{S}} \mathbf{list}(\alpha_3))$:



Reasoning about relational connectives. To infer new abstract heap relations, the analysis needs to reason about the relational connectives. Thus, the analysis mainly relies on three properties about abstract heap relations:

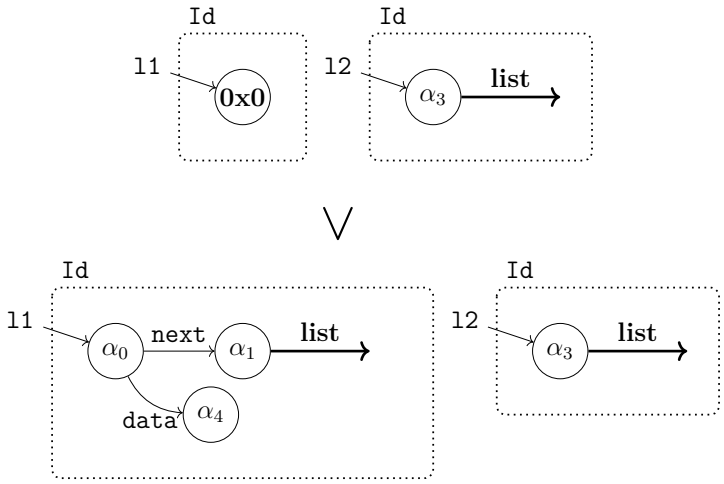
$$
\begin{array}{lrcl}
(1) & \texttt{Id}(\mathrm{h}_0^\sharp *_{\texttt{S}} \mathrm{h}_1^\sharp) & \Leftrightarrow & \texttt{Id}(\mathrm{h}_0^\sharp) *_{\texttt{R}} \texttt{Id}(\mathrm{h}_1^\sharp) \\
(2) & \texttt{Id}(\mathrm{h}^\sharp) & \Rightarrow & [\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp] \\
(3) & [\mathrm{h}_0^\sharp \dashrightarrow \mathrm{h}_1^\sharp] *_{\texttt{R}} [\mathrm{h}_2^\sharp \dashrightarrow \mathrm{h}_3^\sharp] & \Rightarrow & [(\mathrm{h}_0^\sharp *_{\texttt{S}} \mathrm{h}_2^\sharp) \dashrightarrow (\mathrm{h}_1^\sharp *_{\texttt{S}} \mathrm{h}_3^\sharp)]
\end{array}
$$

Property (1) allows to split and merge identity relations as need. Then, property (2) allows to forget the identity relation, weakening it into a transform-into relation. Finally, property (3) allows to forget that two transformations are independent by merging respectively their input and output abstract heaps. For instance, the initial abstract relation of the function `concat` is totally equivalent to the following abstract relation:



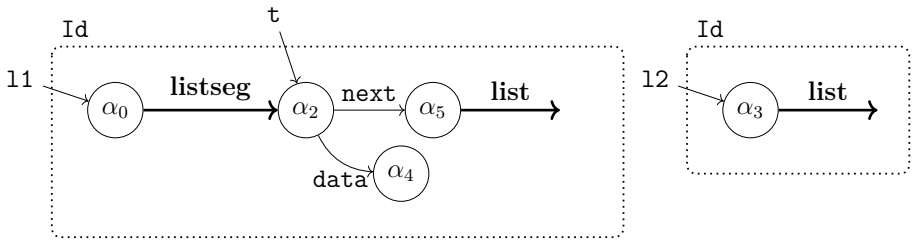
Unfolding operation. The analysis algorithm needs to *unfold* inductive predicates to materialize cells. For instance, after the condition test at line 2 of Figure 2.1, the analysis needs to take into account the effects of the test `l1 == NULL` in each of the branches. The analysis requires to unfold the predicate $\mathbf{list}(\alpha_0)$ using its definition. At this program point, this predicate is still under the $\texttt{Id}(.)$ connective (it has not been modified yet). Unfolding $\mathbf{list}(\alpha_0)$ under this connective produces a disjunction made of the cases of the

definition of **list**(α_0) under the identity relation:



In the branch where the test `l1 == NULL` holds, only the disjunct where `l1` points to the null pointer is kept. In the branch where the test does not hold, only the second disjunct is kept.

Folding operation. In order to analyze condition tests, loops, and to ensure its termination, the analysis also needs to *fold* inductive predicates. The analysis first proceeds to the folding operation at the abstract heap level, and then at the abstract heap relation level, accordingly. This step attempts to preserve identity relations whenever it is possible to do so, and weakens or merges the other abstract relations as needed, using properties (1), (2) and (3). For example, the loop invariant at line 6 for the concatenation function that is automatically computed by the folding operation is:

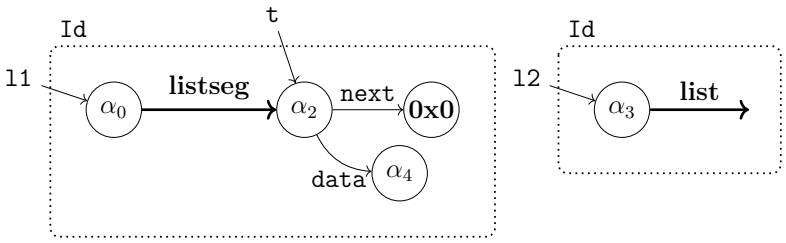


To express that the pointer `t` points somewhere in the list `l1`, the folding operation introduced a list segment between the addresses α_0 and α_2. The pointer `t` thus points to α_2 (that is not null) and `l1` still points to α_0. Both lists pointed by `l1` and `l2` are not modified at this program point, thus the identity relation is preserved.

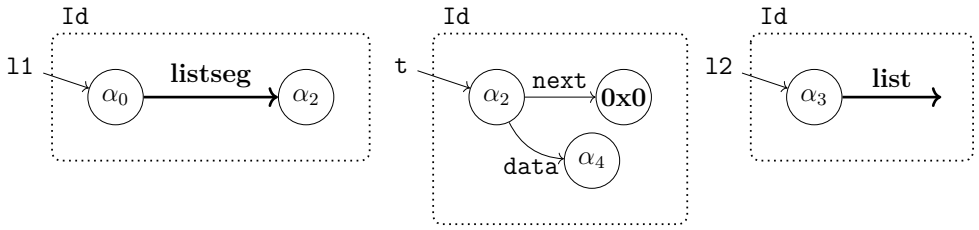
Transfer functions. At each program statement, the analysis performs the corresponding transfer function, such as assignment, allocation or deallocation. Each of these

operations relies on properties (1) and (2). This allows to preserve the identity relation over the heap regions that are not modified and to express the effect of the operation on the concerned heap region. It results in a new abstract heap relation, that expresses a relation between the input state and the state after analyzing the last visited statement.

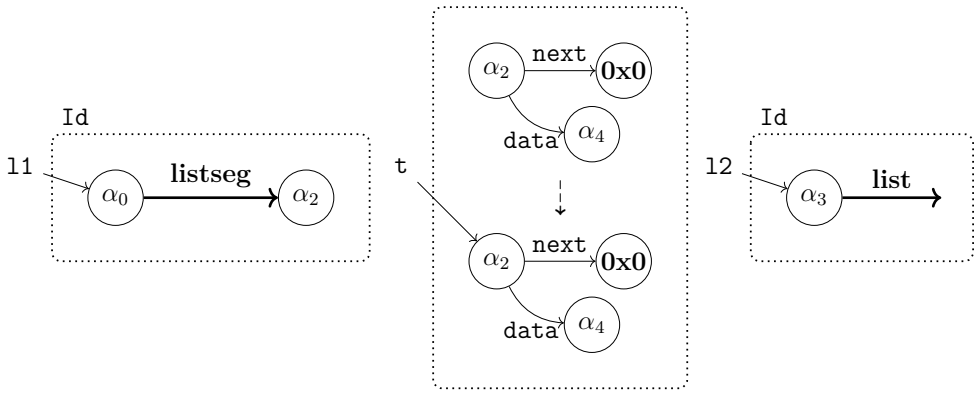
As an example, we discuss the case of the abstract assignement at line 9. After exiting the loop, we have that `t->next = NULL`, the current abstract relation of the analysis is thus:



To perform the assignement `t->next = l2` from the abstract relation, many steps are necessary. First, only the part related to α_2 is going to be modify. Thus, the analysis cannot keep the identity relation over this part. To keep the identity relation over the part that is not modified by the assignment, the analysis applies property (1):



Then, the analysis applies properties (2) to weaken the identity relation of the beeing modified memory part into a transform-into relation:



Actually, the analysis could also keep the `data` field of α_2 under the identity relation, but we do not show it for clarity. Finally, the analysis proceeds to the assignment and obtain the abstract relation of Figure 2.2(c).

```
1  list *sort(list *l) {
2    list *res = NULL;
3    while(l) {
4      list *p_max = l;
5      int v_max = l->data;
6      list *c = l;
7      while(c->next) {
8        if(c->next->data > v_max) {
9          v_max = c->next->data;
10         p_max = c;
11       }
12       c = c->next;
13     }
14     list *tmp;
15     if(p_max == l && v_max == l->data) {
16       //the maximum is the head
17       tmp = l;
18       l = l->next;
19     } else {
20       tmp = p_max->next;
21       p_max->next = p_max->next->next;
22     }
23      tmp->next = res;
24      res = tmp;
25   }
26   return res;
27 }
```

Figure 2.3: A list sort in place program

2.2 Abstract Heap Transformation Predicates

Previously, we demonstrated that abstract heap relations improve the way to describe the behaviour of programs. However, in some cases, abstract heap relations cannot express precisely how a heap region has been transformed into an other.

As illustration, we consider the code shown in Figure 2.3, that implements a sort in place of a linked list. This function, proceeds as follows: it first initializes the resulting list `res` to the null pointer. Then, it traverses in the input list until it does not point to the null pointer. At each iteration, it searches for the maximum element of the input list, deletes it, and adds it at the first position in the resulting list. We observe that the sort is in place: it works directly on the input list, without allocating or deleting cells.

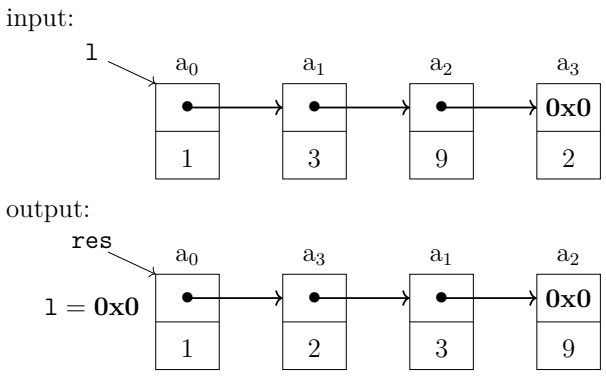


Figure 2.4: Example of pair of concrete input and output states for the sort program of Figure 2.3

Figure 2.4 shows a possible pair of input and output states for this function. We can see that exactly all the memory cells in the input list (at addresses a_0, a_1, a_2 and a_3) are physically present in the output list. We also notice that each value of the `data` field of each list element is unchanged (the `data` field at address a_0 is 1, at address a_1 is 3, ...), but that only the `next` field of some list elements has been modified. For instance, the `next` field of the list element at address a_3 pointed to the null pointer, now it points to the address a_1. However, the list element at address a_1 points to the address a_2 both in the input and the output state. We can say that the list is *partially modified*.

A valid abstract heap relation for all input and output states of this function is:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]$$

However, it only describes that the function inputs a well formed linked list $\mathbf{list}(\alpha)$ and outputs a well formed linked list $\mathbf{list}(\beta)$. It does not describe any information about how the output list is obtained. Abstract heap relations are expressive enough to describe precise relations for programs that strictly do not modify some heap regions, or only modify a finite number of memory cells. It reaches its limit of precision when a program modifies partially an unbounded number of memory cells. More generally, this loss of precision occurs when the [. $\dashrightarrow$.] connective is applied to inductive or segment predicates in its both sides.

To fix this kind of imprecision, we propose to extend abstract heap relations, and more particularly transform-into relations. Instead of creating a new connective for each specific property to describe, we parameterize the [. $\dashrightarrow$.] connectives with a generic set of predicates $\mathbb{T}^\sharp$ that express specific transformations. We named these predicates *abstract heap transformation predicates*. If $t^\sharp \in \mathbb{T}^\sharp$ is an abstract heap transformation predicate, then the abstract heap relation $[h_0^\sharp \dashrightarrow h_1^\sharp]_{t^\sharp}$ describes the transformation of the input heaps abstracted by $h_0^\sharp$ into the output heaps abstract by $h_1^\sharp$, respecting the

```
1 list *add_last(list *l, int v) {
2   list *node = malloc(sizeof(list));
3   node->next = NULL;
4   node->data = v;
5   return concat(l, node);
6 }
```

Figure 2.5: Creation of an element at the end of a list

conditions defined by $t^\sharp$.

Abstract heap transformation predicates are introduced in Chapter 6. We specifically propose an example of such predicates that allow to express that the function in Figure 2.3 returns a *permutation in place* of the input list. These example predicates have the advantage to be available for any data structure. We also integrate abstract heap transformation predicates in our analysis algorithm. The analysis does not depend on a specific set of such predicates $\mathbb{T}^\sharp$, it only requires that each set of predicates provides the same interface.

2.3 Compositional Inter-procedural Analysis

From now, we provided an overview of our relational shape analysis without taking into account function calls. We now lift this restriction and we present an inter-procedural analysis, that uses the abstract heap relations for composing functions, instead of always reanalyzing them at each call-site.

Generally, composing functions makes the shape analysis scalable but it may suffer of some loss of precision. We first propose a *composition operator*, that computes directly the effects of a function call using the abstract heap relation at the call-site and the abstract heap relation of the called function. This operator benefits from the expressiveness of abstract heap relations to avoid loss of precision.

We then present an algorithm that detects when it is possible to perform the composition, and when it is not the case, computes a new composable abstract relation for the called function. Formalizations of the composition operator and the analysis algorithm are presented in Chapter 8.

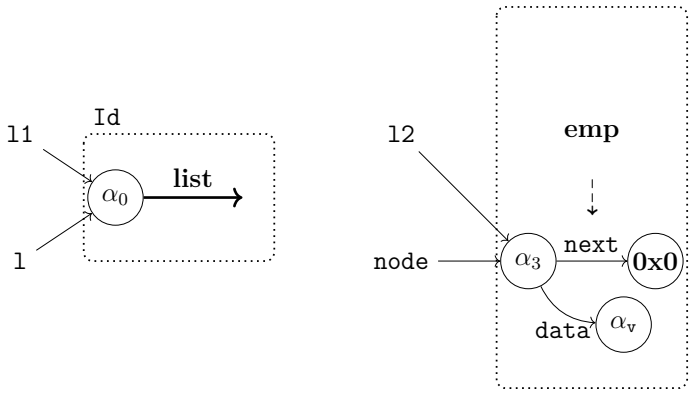
2.3.1 Composition of Abstract Heap Relations

In this section, we present the composition operator $\circ_{\mathbb{R}^\sharp}$ of abstract heap relations. It takes two abstract heap relations $r_0^\sharp$ and $r_1^\sharp$ and returns a new abstract heap relation $r^\sharp$, which represents the application of the relations described by $r_1^\sharp$ from the relations

described by $r_0^\sharp$.

The strength with abstract heap relations is that they describe local relations between input and output states of programs. This allows to compose independently each heap region, and thus to express precisely the effect of the called function on each region. To illustrate it, we discuss the effects of our composition operator, applied to the call of function `concat` in the function `add_last` of Figure 2.5. The function `add_last` inserts a new list node at the end of a list. This function inputs a list `l` and a value `v`, allocates a new list node, whose `next` field points to `NULL` and `data` field is assigned to `v`. The function `concat` is then called to add the new node at the end of the list `l`. For simplicity, we elide abstract heap transformation predicates and we do not consider the variable `v`.

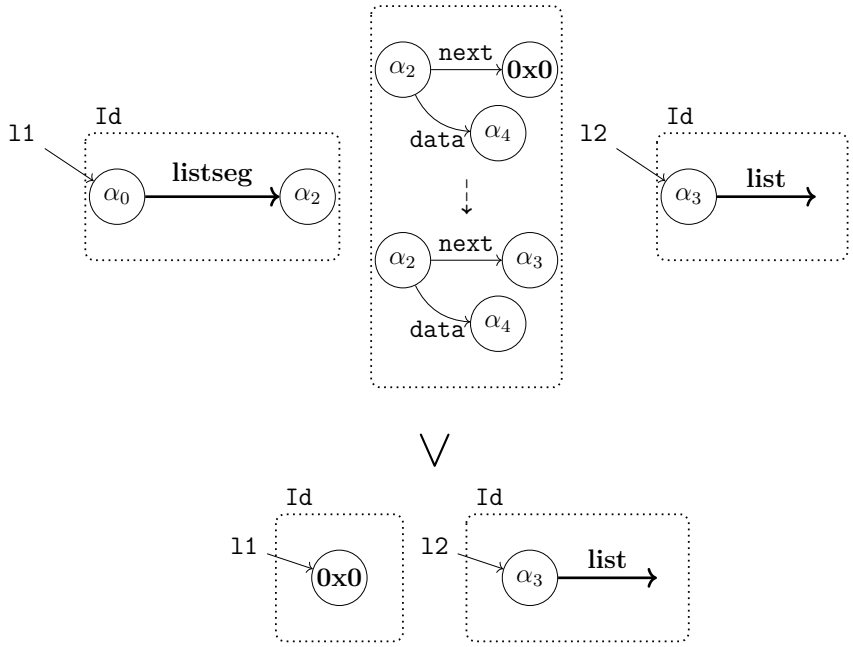
Just before the function `concat` is called, at line 5, we assume that the current abstract heap relation $r_0^\sharp$ for the function `add_last` is:



When the function `concat` is called, its parameters `l1` and `l2` are respectively assigned to `l` and `node`. If α_0 is the address pointed by `l` and α_3 the address pointed by `node`, this abstract heap relation describes a relation between the input state of `add_last` and the state before the call of `concat`. Indeed, at this program point, the list pointed by `l` is not modified and `node` points to a list element made of two fields `next` and `data`, and both of them have been freshly allocated. Note that the abstract heap relation $[\mathbf{emp} \dashrightarrow h^\sharp]$ means that the concrete heap region abstracted by $h^\sharp$ has been allocated, as an empty heap region has been transformed into another.

We also assume that the function `concat` is described by a disjunction of two abstract

heap relations $r_1^\sharp \vee r_2^\sharp$ that are the same that we presented in the previous section:



Note that the composition between $r_0^\sharp$ and this disjunction is defined as follows:

$$r_0^\sharp \circ_{\mathbb{R}^\sharp} (r_1^\sharp \vee r_2^\sharp) = (r_0^\sharp \circ_{\mathbb{R}^\sharp} r_1^\sharp) \vee (r_0^\sharp \circ_{\mathbb{R}^\sharp} r_2^\sharp)$$

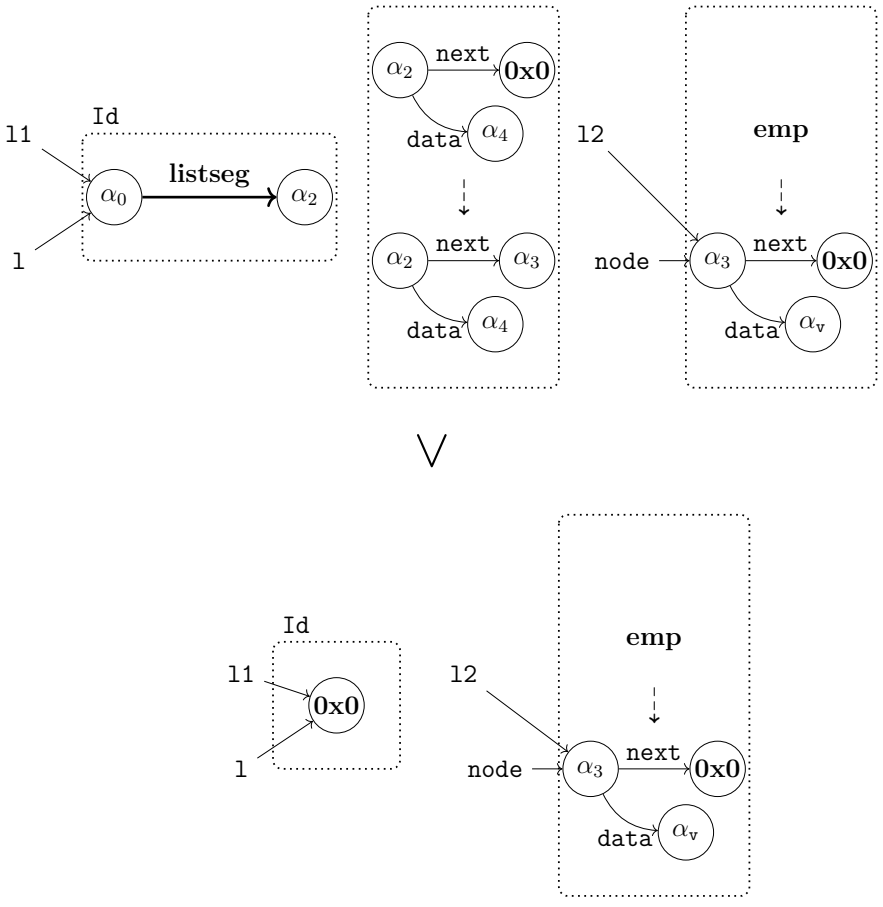
We first focus on $r_0^\sharp \circ_{\mathbb{R}^\sharp} r_1^\sharp$. Without relation between input and output states, it is not possible to express that the list pointed by `l2` is not modified by the concatenation function. Therefore, a sound composition of this function must loose information about this list and must return an other list of any length. For instance, after the call of `concat`, it would loose the information that the variable `node` points to a single list node. Thanks to the identity relation, this loss of precision is avoided. Using this relation, our composition operator is able to preserve the fact the variable `node` still points exactly to the same list node it pointed before the function call. Moreover, it preserves the information that this same node has been allocated in the function `add_last`. Regarding to the list pointed by `l1`, the composition operator infers that only its last element has been modified (and that this latter points to `l2`).

We now focus on $r_0^\sharp \circ_{\mathbb{R}^\sharp} r_2^\sharp$. The only difference with the previous case is about the list pointed to `l1`. As `l1` points to the null pointer in $r_2^\sharp$, the composition operator infers that `l1` also points to the null pointer in the resulting abstract relation.

```
1 list *create3(int x, int y, int z) {
2   list *h = add_last(NULL, x);
3   h = add_last(h, y);
4   h = add_last(h, z);
5   h = sort(h);
6   return h;
7 }
```

Figure 2.6: Creation of a sorted list of three elements

Finally, the result of $(r_0^\sharp \circ_{\mathbb{R}^\sharp} r_1^\sharp) \vee (r_0^\sharp \circ_{\mathbb{R}^\sharp} r_2^\sharp)$ is:



2.3.2 Analysis Algorithm

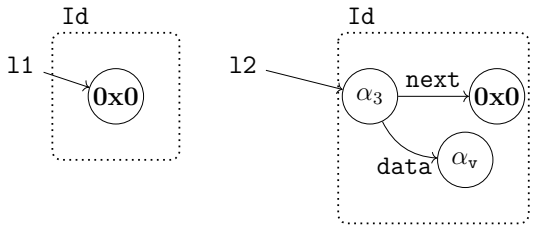
In the previous section, we presented how the composition operator was defined. We now present when and how it is applied by the analysis algorithm.

We consider the analysis of the function `create3` of Figure 2.6 that creates a list containing three values `x`, `y` and `z`, and sorts this list. This function calls `add_last` three times to create each list node and calls `sort` to perform the sort.

Computing a first abstract heap relation. We assume that the function `add_last` has never been analyzed before its first call at line 2 of Figure 2.6. Thus, during the first call of `add_last`, the algorithm needs to analyze it a first time in order to obtain an abstract heap relation that describes its behavior.

To do that, the algorithm assigns the expressions of the function call corresponding to the arguments. Here, it assigns the null pointer to the argument `l` to and the value of `x` to the argument `v`. The parts of memory that are not reachable from the arguments of the called function are discarded for its analysis, as the function will have not effect on them.

The analysis of `add_last` starts with the identity relation where `l` points to the null pointer and `v` contains the value of `x`. Similarly, when the analysis reaches the call of `concat`, there is no abstract heap relation to describe this function. The algorithm then proceeds to the analysis of `concat` from the identity relation where `l1` points to the null pointer and `l2` points to a list node. In this calling context, the returned abstract heap relation is simply the identity relation of a list node, such as `l1` is null and the function `concat` returns `l2`. Thus, the first abstract heap relation describing the function `concat` is:



After computing this abstract heap relation for the function `concat`, the algorithm enriches it with the identity relation of the part of the memory that is not reachable from the arguments of `concat`. This allows to compose the abstract heap relation at call site with the abstract heap relation describing the called function.

Reanalyzing the function with a more general pre-condition. For the second call of `add_last`, the algorithm cannot perform directly the composition. Indeed, the abstract heap relation previously computed for `add_last` describes the behavior of this function when it is called with the null pointer. Whereas in this calling context, it is called with the list node pointed by `h`. This is detected automatically by means of an *inclusion checking* operator. The algorithm thus requires an abstract heap relation whose pre-condition is valid both for the null pointer and for a list node.

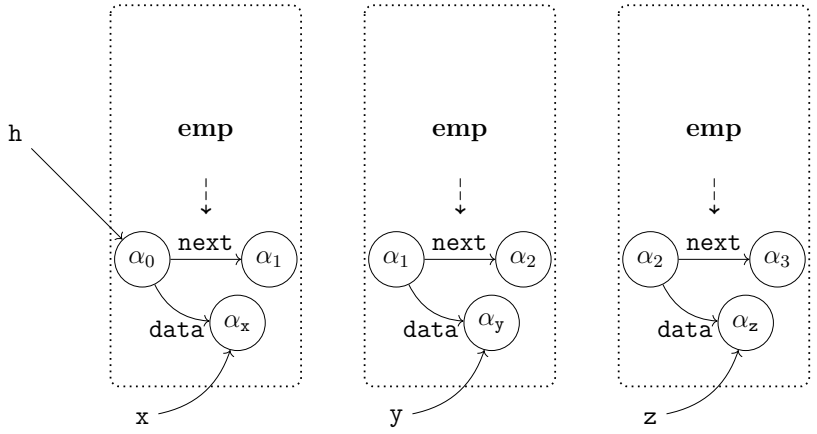
To obtain it, the algorithm uses a *join* operator that folds the null pointer and the list node into a list inductive predicate $\mathbf{list}(\alpha)$. The algorithm then reanalyzes the function

`add_last` from the identity relation of the joined abstract heap. In turn, the algorithm computes a new abstract heap relation for this function, whose pre-condition is valid for any linked list. The previous abstract heap relation computed from the null pointer is then forgotten by the algorithm and substituted with the new one. For similar reasons, the algorithm must reanalyze the function `concat` and compose it when it is called in `add_last`.

Finally, the algorithm can compose soundly the second call of `add_last` and continue the analysis.

Composing without reanalyzing the function. For the third call of `add_last`, the algorithm does not need to reanalyze this function. Indeed, the last computed abstract heap relation for `add_last` describes the behavior of this function when it is called with a list of any length. Thus, the algorithm can perform the composition directly, whatever the length of the list pointed by `h`.

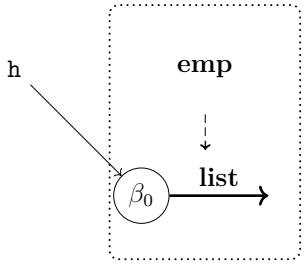
The algorithm always keeps the last computed abstract heap relation for a function. For instance, if `add_last` is called in an other function than `create3`, the algorithm will perform the composition without reanalyze it (if the calling context is included in the pre-condition of the abstract heap relation). We also extended this algorithm in Chapter 9 to analyze recursive functions. After the three calls to `add_last`, the analysis has inferred the abstract relation:



Observe that the analysis did not lose any precision. Indeed, the abstract heap relation above describes precisely a list where three nodes have been allocated, and that contains the values of `x`, `y` and `z` in the good order.

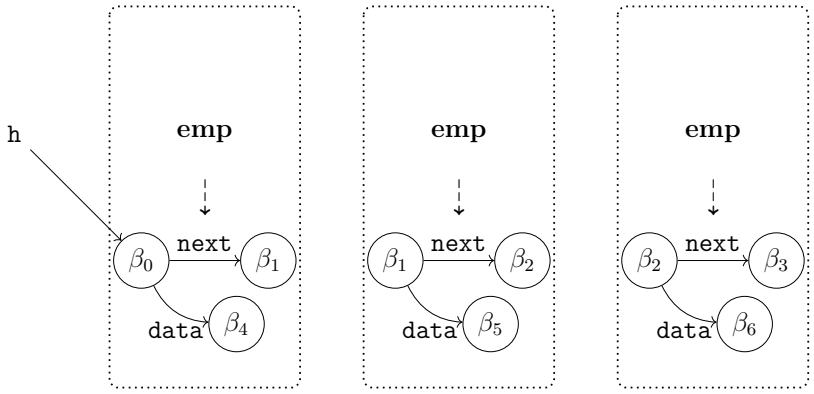
Benefits of abstract heap transformation predicates for the composition. We first assume that our compositional analysis does not consider abstract heap transformation predicates, and that the function `sort` is described by the abstract heap relation [**list**(α) `-->` **list**(β)]. During the call of `sort` in the function `create3`, the analysis

should perform the composition between the abstract heap relation of the last paragraph and [**list**(α) ⇢ **list**(β)] (that describes the behavior of `sort`). The composition should produce:



It does not describe that a list made of exactly three elements has been allocated, but only that a list of any length has been allocated. Indeed, the abstract heap relation attached to `sort` only expresses the transformation of a linked list into another. Consequently, the composition operator does not have enough information to infer more precise properties about the output list.

This loss of precision can be fixed with abstract heap transformation predicates, introduced in Section 2.2. The gain of precision depends on the precision of the predicates. For instance, with the abstract heap transformation predicates defined in Chapter 6 that allow to express a *permutation in place* (but not only) of any data structure, the composition is able to infer that the output list is made of exactly three elements:



However, it does not express precisely which node contains the value of `x`, `y` or `z`.

Chapter 3

Concrete Semantics

The C language is the target language for our program analysis. Thus, in this chapter, we introduce an imperative programming language, that is a fragment of the C language. This language handles neither arrays nor unions, but features pointers, recursive data structures and dynamic allocations. This allows to illustrate the family of programs our work aims to analyze. We also define a trace semantics and a relational semantics based on a transition system of this language.

3.1 C-Like Programming Language

The syntax of the language is defined in Figure 3.1. A program $p \in \mathcal{Prog}$ consists on a sequence of function definitions. Each function $f \in \mathcal{Fun}$ is defined by its arguments and its body, a command $c \in \mathcal{Cmd}$. For simplicity, all functions end by the returned command **ret** and do not return values. The command consist of assignments, memory allocations and deallocations, sequences of programs, conditionals, loops and function calls. The command **skip** denotes the command that does nothing. The sets $\mathcal{Lval}$ and $\mathcal{Expr}$ define the sets of l-values and expressions. An *l-value* $\ell \in \mathcal{Lval}$ denotes the address of a memory *cell* (also named memory *location*). It can be either a program variable $\mathtt{x} \in \mathbb{X}$, an l-value offset by a field ($\ell \cdot \mathtt{f}$), or the value of a pointer expression ($*exp$). We assume that all field names $\mathtt{f} \in \mathbb{F}$ are implicitly converted into numeric offsets, and that $\varnothing$ denotes the null offset. We also write $e\mathtt{->f}$ as a syntactic sugar for the location $(*e)\cdot\mathtt{f}$. An *expression* $e \in \mathcal{Expr}$ denotes a value. It can be the content at a memory location (ℓ), the address of a memory location ($\&\ell$), any value v, or a binary expression $e_1 \oplus e_2$. The operator $\oplus$ designs any standard binary operator, like addition, substraction, equality test, etc...

We admit for simplicity that this language does not make it possible to declare global variables, even if they could be integrated easily.

$$
\begin{array}{rcll}
\ell\ (\in \mathcal{Lval}) & ::= & \mathtt{x} & (\mathtt{x} \in \mathbb{X}) \\
& | & \ell_1 \cdot \mathtt{f} & (\ell_1 \in \mathcal{Lval}; \mathtt{f} \in \mathbb{F}) \\
& | & *e & (e \in \mathcal{Expr})
\end{array}
$$

$$
\begin{array}{rcll}
e\ (\in \mathcal{Expr}) & ::= & \ell & (\ell \in \mathcal{Lval}) \\
& | & \&\ell & (\ell \in \mathcal{Lval}) \\
& | & \mathrm{v} & (\mathrm{v} \in \mathbb{V}) \\
& | & e_1 \oplus e_2 & (e_1, e_2 \in \mathcal{Expr}) \\
\oplus & ::= & + | - | = | \ldots &
\end{array}
$$

$$
\begin{array}{rcll}
c\ (\in \mathcal{Cmd}) & ::= & \ell = e & (\ell \in \mathcal{Lval}; e \in \mathcal{Expr}) \\
& | & \ell = \mathbf{malloc}(\{\mathtt{f}_1, \ldots, \mathtt{f}_n\}) & (\ell \in \mathcal{Lval}; \mathtt{f}_i \in \mathbb{F}) \\
& | & \mathbf{free}(\ell) & (\ell \in \mathcal{Lval}) \\
& | & c_1\ ;\ c_2 & (c_1, c_2 \in \mathcal{Cmd}) \\
& | & \mathbf{if}\,(e)\ c_1\ \mathbf{else}\ c_2 & (e \in \mathcal{Expr}; c_1, c_2 \in \mathcal{Cmd}) \\
& | & \mathbf{while}\,(e)\ c_1 & (e \in \mathcal{Expr}; c_1 \in \mathcal{Cmd}) \\
& | & f(e_1, \ldots, e_n) & (\ell \in \mathcal{Lval}; f \in \mathcal{Fun}; e_i \in \mathcal{Expr}) \\
& | & \mathbf{ret} & \\
& | & \mathbf{skip} &
\end{array}
$$

$$
\begin{array}{rcll}
p\ (\in \mathcal{Prog}) & ::= & f(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{c; \mathbf{ret};\} & (f \in \mathcal{Fun}; \mathtt{x}_i \in \mathbb{X}; c \in \mathcal{Cmd}) \\
& | & p_1; p_2 & (p_1, p_2 \in \mathcal{Prog})
\end{array}
$$

Figure 3.1: Programming Language Syntax

3.2 Concrete Memory States

In the following, we define concrete states and provide a big-step operational semantics for this language.

Let $\mathbb{A}$ be the set of addresses and $\mathbb{V}$ the set of values, we assume that any address $a \in \mathbb{A}$ is also a value $v \in \mathbb{V}$, i.e. $\mathbb{A} \subseteq \mathbb{V}$. A *concrete heap* $h \in \mathbb{H} = \mathbb{A} \rightharpoonup \mathbb{V}$ is a partial function from addresses to values. We write $[a_1 \mapsto v_1; \ldots; a_n \mapsto v_n]$ for the concrete heap where each cell at address a_i contains the value v_i, with $1 \leq i \leq n$. We suppose that all cells have the same size. We also note $h[a \leftarrow v]$ the heap where we update the content of the cell at address a with value v in the heap h. We let the domain of h, noted $\mathbf{dom}(h)$, be the set of addresses at which it is defined. For example, the domain for the concrete heap $[a_1 \mapsto v_1; a_2 \mapsto v_2; a_3 \mapsto v_3]$ is $\{a_1, a_2, a_3\}$. Additionally, if h_0 and h_1 are two concrete heaps such that $\mathbf{dom}(h_0) \cap \mathbf{dom}(h_1) = \emptyset$, we let $h_0 \circledast h_1$ denote the concrete heap obtained by merging h_0 and h_1 (its domain is $\mathbf{dom}(h_0) \cup \mathbf{dom}(h_1)$). Moreover, $\mathbf{im}(h)$ denotes the set of values that are pointed by at least an address in h. Finally, if A is a set of addresses such as $A \subseteq \mathbb{A}$ and h a concrete heap, we note $h \ominus A$ the concete heap obtained by removing all the addresses of A in h. For instance $[a_1 \mapsto v_1; a_2 \mapsto v_2; a_3 \mapsto v_3] \ominus \{a_1, a_3\} = [a_2 \mapsto v_2]$.

Let $\mathbb{X}$ be the set of program variables. A *concrete environment* $e \in \mathbb{E} = \mathbb{X} \to \mathbb{A}$ binds each program variable $\mathtt{x} \in \mathbb{X}$ to its numeric address $a \in \mathbb{A}$. Thus, an environment indicates the address of a variable in the heap. We note $[|\mathtt{x}_1 = a_1, \ldots, \mathtt{x}_n = a_n|]$, the concrete environment where the address of each variable $\mathtt{x}_i$ is equal to a_i. We also note $e[|\mathtt{x}_1 = a_1, \ldots, \mathtt{x}_n = a_n|]$ the operation that adds in the environement e the *new variables* $\mathtt{x}_1, \ldots, \mathtt{x}_n$ with respectively the value of their address $a_1, \ldots, a_n$. In the same way as concrete heaps, we note $\mathbf{dom}(e)$ for the set of variables to which e is defined and $\mathbf{im}(e)$ the set of addresses that are associated to a variable in e.

A *concrete memory state* $m \in \mathbb{M} = \mathbb{E} \times \mathbb{H}$ is simply a pair made of a concrete environment and a concrete heap. The addresses of variables are also allocated in the heap. We assume that all the variables of a function, as well as the arguments and the local variables, are unique. Thus, two functions cannot have common variables.

Example 3.1 (A concrete memory state). We consider the following code:

```
1 int x = 13;
2 int y = 182;
3 int *p = &y;
```

At the end of this code, a possible concrete memory state is the pair (e, h) such as:

$$e = [|\mathtt{x} = 1, \mathtt{y} = 2, \mathtt{p} = 3|] \text{ and } h = [1 \mapsto 13, 2 \mapsto 182, 3 \mapsto 2]$$

Indeed, the environment e maps each variable to its address: `x` to the address 1, `y` to

$$\mathcal{L}[\![\mathtt{x}]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathrm{e}(\mathtt{x})$$
$$\mathcal{L}[\![\ell \cdot \mathtt{f}]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathcal{L}[\![\ell]\!](\mathrm{e},\mathrm{h}) + \mathtt{f}$$
$$\mathcal{L}[\![*e]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathcal{E}[\![e]\!](\mathrm{e},\mathrm{h})$$

$$\mathcal{E}[\![\ell]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathrm{h}(\mathcal{L}[\![\ell]\!](\mathrm{e},\mathrm{h}))$$
$$\mathcal{E}[\![\&\ell]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathcal{L}[\![\ell]\!](\mathrm{e},\mathrm{h})$$
$$\mathcal{E}[\![\mathrm{v}]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathrm{v}$$
$$\mathcal{E}[\![e_1 \oplus e_2]\!](\mathrm{e},\mathrm{h}) \stackrel{def}{=} \mathcal{E}[\![e_1]\!](\mathrm{e},\mathrm{h}) \oplus \mathcal{E}[\![e_2]\!](\mathrm{e},\mathrm{h})$$

Figure 3.2: Concrete semantics for l-values and expressions

the address 2 and `p` to the address 3. The concrete heap h describes each memory cell: the memory cell at the address of `x` contains the value 13, the memory cell at the address of `y` contains the value 182 and the memory cell at the address of `p` contains the address of `y` (here the value 2).

3.3 Concrete Trace Semantics

The semantics of locations and expressions are defined by two functions, $\mathcal{L}[\![loc]\!] \in \mathbb{M} \to \mathbb{A}$ and $\mathcal{E}[\![exp]\!] \in \mathbb{M} \to \mathbb{V}$, respectively from memory states into addresses and from memory states into values. They are mutually defined by induction on the structures of l-values and expressions, as shown in Figure 3.2. The environment provides the address of the variable `x` for $\mathcal{L}[\![\mathtt{x}]\!](\mathrm{e},\mathrm{h})$ whereas $\mathcal{E}[\![\ell]\!](\mathrm{e},\mathrm{h})$ first evaluates the address of the l-value ℓ, then returns the value contained in h at this address.

Figure 3.3 defines a transition system $\mathcal{S} = (\mathbb{S}, \longrightarrow)$ for commands. The set of states $\mathbb{S}$ is defined as follows. First, a *program configuration* is a pair of a command and a memory state noted $\langle c \mid \mathrm{m}\rangle$ (or $\langle c \mid (\mathrm{e},\mathrm{h})\rangle$). It associates the current memory state to a command. The program configuration $\langle \textbf{skip} \mid \mathrm{m}\rangle$ is the final configuration (if ever). To handle function calls (recursive or not), the transition system requires a stack to save the program configuration before executing a function call. Thus, the set of the transition system $\mathbb{S}$ is a stack $\sigma \in \Sigma$ of program configurations. We note ϵ for the empty stack. The stack $\langle c \mid \mathrm{m}\rangle :: \sigma$ is the stack of which the first element is $\langle c \mid \mathrm{m}\rangle$.

We now describe the transition system of commands for assignments, conditionals, loops, allocations, deallocations and function calls and returns. Note that only function calls and function returns modify the stack. Assignments are standard: they just consist of evaluating the lvalue and the expression, and to update the concrete heap accordingly. To allocate a new address in a heap h, the semantics takes an address that is not in $\textbf{dom}(\mathrm{h})$. Regarding to deallocations, the system evaluates the lvalue as an expression to a base address, and removes in the concrete heap all the memory block from this base

$$\frac{\mathcal{E}[\![e]\!](\mathrm{e},\mathrm{h}) = \mathrm{v} \quad \mathcal{L}[\![\ell]\!](\mathrm{e},\mathrm{h}) = \mathrm{a}}{\langle \ell = e \mid (\mathrm{e},\mathrm{h})\rangle :: \sigma \longrightarrow \langle \mathbf{skip} \mid (\mathrm{e}, h[\mathrm{a} \leftarrow \mathrm{v}])\rangle :: \sigma}$$

$$\frac{\langle c_1 \mid \mathrm{m}\rangle :: \sigma \longrightarrow \langle c_1' \mid \mathrm{m}'\rangle :: \sigma'}{\langle c_1 \ ; \ c_2 \mid \mathrm{m}\rangle :: \sigma \longrightarrow \langle c_1' \ ; \ c_2 \mid \mathrm{m}'\rangle :: \sigma'} \qquad \frac{\langle c_1 \mid \mathrm{m}\rangle :: \sigma \longrightarrow \langle \mathbf{skip} \mid \mathrm{m}'\rangle :: \sigma'}{\langle c_1 \ ; \ c_2 \mid \mathrm{m}\rangle :: \sigma \longrightarrow \langle c_2 \mid \mathrm{m}'\rangle :: \sigma'}$$

$$\frac{\begin{array}{c}\mathcal{L}[\![\ell]\!](\mathrm{e},\mathrm{h}) = \mathrm{a} \quad \exists \mathrm{a}' \in \mathbb{A}, \quad \mathrm{a}' \notin \mathbf{dom}(\mathrm{h}) \quad \mathrm{v}_1, \ldots, \mathrm{v}_n \in \mathbb{V} \\ \mathrm{h}' = [\mathrm{a}' + \mathtt{f}_1 \mapsto \mathrm{v}_1, \ldots, \mathrm{a}' + \mathtt{f}_n \mapsto \mathrm{v}_n]\end{array}}{\langle \ell = \mathbf{malloc}(\{\mathtt{f}_1, \ldots, \mathtt{f}_n\}) \mid (\mathrm{e},\mathrm{h})\rangle :: \sigma \longrightarrow \langle \mathbf{skip} \mid (\mathrm{e}, \mathrm{h}[\mathrm{a} \leftarrow \mathrm{a}'] \circledast \mathrm{h}')\rangle :: \sigma}$$

$$\frac{\mathcal{E}[\![\ell]\!](\mathrm{e},\mathrm{h}) = \mathrm{a} \quad \mathrm{h}' = \mathrm{h} \ominus \{\mathrm{a} + \mathtt{f} \mid \mathtt{f} \in \mathbb{F}\}}{\langle \mathbf{free}(\ell) \mid (\mathrm{e},\mathrm{h})\rangle :: \sigma \longrightarrow \langle \mathbf{skip} \mid (\mathrm{e},\mathrm{h}')\rangle :: \sigma}$$

$$\frac{\mathcal{E}[\![e]\!](\mathrm{m}) \neq 0}{\langle \mathbf{if}\,(e)\, c_1\, \mathbf{else}\, c_2 \mid \mathrm{m}\rangle :: \sigma \longrightarrow \langle c_1 \mid \mathrm{m}\rangle :: \sigma} \qquad \frac{\mathcal{E}[\![e]\!](\mathrm{m}) = 0}{\langle \mathbf{if}\,(e)\, c_1\, \mathbf{else}\, c_2 \mid \mathrm{m}\rangle :: \sigma \longrightarrow \langle c_2 \mid \mathrm{m}\rangle :: \sigma}$$

$$\frac{\mathcal{E}[\![e]\!](\mathrm{e},\mathrm{h}) \neq 0}{\langle \mathbf{while}\,(e)\, c \mid \mathrm{m}\rangle :: \sigma \longrightarrow \langle c\,;\, \mathbf{while}\,(e)\, c \mid \mathrm{m}\rangle :: \sigma}$$

$$\frac{\mathcal{E}[\![e]\!](\mathrm{e},\mathrm{h}) = 0}{\langle \mathbf{while}\,(e)\, c \mid \mathrm{m}\rangle :: \sigma \longrightarrow \langle \mathbf{skip} \mid (\mathrm{m})\rangle :: \sigma}$$

$$\frac{\begin{array}{c}f(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{c; \mathbf{ret};\} \in \mathcal{D}ef \\ \exists \mathrm{a}_1, \ldots, \mathrm{a}_n \in \mathbb{A}, \forall 1 \leqslant i \leqslant n, \mathrm{a}_i \notin \mathbf{dom}(\mathrm{h}) \\ \mathrm{e}' = [|\mathtt{x}_1 = \mathrm{a}_1, \ldots, \mathtt{x}_n = \mathrm{a}_n|] \\ \mathrm{h}' = [\mathrm{a}_1 \mapsto \mathcal{E}[\![e_1]\!](\mathrm{e},\mathrm{h}),\ \ldots,\ \mathrm{a}_n \mapsto \mathcal{E}[\![e_n]\!](\mathrm{e},\mathrm{h})] \circledast \mathrm{h}\end{array}}{\langle f(e_1, \ldots, e_n) \mid (\mathrm{e},\mathrm{h})\rangle :: \sigma \longrightarrow \langle c; \mathbf{ret}; \mid (\mathrm{e}',\mathrm{h}')\rangle :: \langle f(e_1, \ldots, e_n) \mid (\mathrm{e},\mathrm{h})\rangle :: \sigma}$$

$$\frac{\mathrm{h}'' = \mathrm{h} \ominus \mathbf{im}(\mathrm{e})}{\langle \mathbf{ret} \mid (\mathrm{e},\mathrm{h})\rangle :: \langle f(e_1, \ldots, e_n) \mid (\mathrm{e}',\mathrm{h}')\rangle :: \sigma \longrightarrow \langle \mathbf{skip} \mid (\mathrm{e}',\mathrm{h}'')\rangle :: \sigma}$$

Figure 3.3: Transition system

address. For conditionals and loops, the value 0 is evaluated as false, any other value is evaluated as true.

Regarding to function calls, the addresses of arguments are first allocated. These addresses are then associated to the arguments in a new environment e'. Then, each expression of the function call is evaluated and assigned to the address of the corresponding argument, and added in the concrete heap h. This results in the new concrete heap h'. A new program configuration is pushed on the stack: it is made of the body of the called function c; **ret**; and memory state $(\mathrm{e}', \mathrm{h}')$ previously created. Regarding to function returns, all the addresses in current environment are deleted in the current heap (this produces the heap h''). These addresses correspond to the argument of the function. The environment before the function called (e') is restored and the stack is popped.

Remark that functions can contain local variables. They are allocated at the beginning of the block they are declared and deallocated at the exit of this block exactly as for the arguments of a function. However, for concision we do not explicit it in the concrete semantics.

Definition 3.1 (Trace semantics of commands). *We define the finite trace, infinite trace and trace semantics of commands. We start by defining the finite trace semantics $[\![c]\!]^*_{\mathcal{T}} \in \mathcal{P}(\Sigma^*)$ of commands:*

$$[\![c]\!]^*_{\mathcal{T}} = \{\langle\langle c \mid \mathrm{m}_0\rangle :: \epsilon, \ldots, \langle \textbf{skip} \mid \mathrm{m}_n\rangle :: \epsilon\rangle \mid \forall i, 0 \leqslant i < n, \sigma_i \longrightarrow \sigma_{i+1}\}$$

Then, we define the infinite trace semantics $[\![c]\!]^\omega_{\mathcal{T}} \in \mathcal{P}(\Sigma^\omega)$ of commands:

$$[\![c]\!]^\omega_{\mathcal{T}} = \{\langle\langle c \mid \mathrm{m}_0\rangle :: \epsilon, \ldots\rangle \mid \forall i, 0 \leqslant i, \sigma_i \longrightarrow \sigma_{i+1}\}$$

Finally we define the trace semantics $[\![c]\!]^\infty_{\mathcal{T}} \in \mathcal{P}(\Sigma^\infty)$ of commands:

$$[\![c]\!]^\infty_{\mathcal{T}} = [\![c]\!]^*_{\mathcal{T}} \cup [\![c]\!]^\omega_{\mathcal{T}}$$

The initial state of a trace (finite or infinite) of a command is a stack with a single program configuration, that is made of the command itself and input memory state. The final state of a finite trace is a stack with a single program configuration: the final configuration $\langle \textbf{skip} \mid \mathrm{m}_n\rangle$ where m_n is the output memory state.

Example 3.2 (Function call and return). We consider the following function `incr`

```
1 void incr(int *p) {
2   *p = *p + 1;
3 }
```

We describe the effects of a call of this function `incr(&x)` applied from a memory

state composed of two variables `x` and `y`, that contain respectively the values 17 and 41. More formally, we are reducing $\langle \texttt{incr(\&x)} \mid (\mathrm{e}, \mathrm{h}) \rangle :: \sigma$, with:

$$\mathrm{e} = [|\texttt{x} = 1, \texttt{y} = 2|] \text{ and } \mathrm{h} = [1 \mapsto 17, 2 \mapsto 41]$$

A new address that is not already in h (thus neither 1 nor 2) is generated (in this example the address 3). This produces a new environment $\mathrm{e}' = [|\texttt{p} = 3|]$. Then, the expression `&x` is evaluated to 1, and the heap $[3 \mapsto 1]$ is merged with the original heap h. This results in the heap $\mathrm{h}' = [1 \mapsto 17, 2 \mapsto 41, 3 \mapsto 1]$.

The body of `incr` c is then reduced from the stack:

$$\langle c \mid (\mathrm{e}', \mathrm{h}') \rangle :: \langle \texttt{incr(\&x)} \mid (\mathrm{e}, \mathrm{h}) \rangle :: \sigma$$

After the reduction of c, we obtain the heap $\mathrm{h}'' = [1 \mapsto 18, 2 \mapsto 41, 3 \mapsto 1]$ where the value of `x` has been incremented. The return of the function `incr` deallocates the cell corresponding to the argument `p` with $\mathrm{h}'' \ominus \{3\}$, restores the original environment e, and pops the stack. We thus obtain:

$$\langle \textbf{skip}, (\mathrm{e}, [1 \mapsto 18, 2 \mapsto 41]) \rangle :: \sigma$$

An important feature with our program semantics is that we must consider that the set of addresses $\mathbb{A}$ is *infinite*, and that every allocation generates fresh addresses. In a real imperative programming language like C, the same address can be deallocated, then allocated randomly (if it is available) during the execution of a program. This feature does not allow to express that some parts of memory have been freshly allocated or deallocated between two program points whereas these are exactly the kind of properties that our work attempts to prove. Therefore, we admit for sake of simplicity that when a program needs to allocate a memory region, fresh addresses and values are generated. Note: the actual behavior of C programs could be modeled by considering addresses as pairs of (numeric address, number of allocations in the program). As this makes the formalization heavier, we have chosen to simplify it by fresh allocations.

Table 3.1 summarizes the notations for the concrete trace semantics of this language.

3.4 Concrete Relational Semantics

Given a command $c \in \mathcal{Cmd}$, we define its *relational semantics* $[\![c]\!]_{\mathcal{R}} \in \mathcal{P}(\mathbb{M} \times \mathbb{M})$.

Definition 3.2 (Relational semantics of commands). *The relational semantics of commands* $[\![c]\!]_{\mathcal{R}} \in \mathcal{P}(\mathbb{M} \times \mathbb{M})$ *is defined by:*

$$[\![c]\!]_{\mathcal{R}} = \{(\mathrm{m}_{\mathrm{in}}, \mathrm{m}_{\mathrm{out}}) \mid \langle \langle c \mid \mathrm{m}_{\mathrm{in}} \rangle :: \epsilon, \ldots, \langle \textbf{skip} \mid \mathrm{m}_{\mathrm{out}} \rangle :: \epsilon \rangle \in [\![c]\!]^*_{\mathcal{T}}\}$$

Program Syntax

Name	Set	Element	Evaluation
Field	$\mathbb{F}$	$\mathtt{f}, \varnothing$	
Funcion	$\mathcal{Fun}$	f	
L-value	$\mathcal{Lval}$	l	$\mathcal{L}[\![l]\!] \in \mathbb{M} \to \mathbb{A}$
Expression	$\mathcal{Expr}$	e	$\mathcal{E}[\![e]\!] \in \mathbb{M} \to \mathbb{V}$
Command	$\mathcal{Cmd}$	c	$[\![c]\!]^*_{\mathcal{T}} \in \mathcal{P}(\Sigma^*)$

Memory State

Name	Set	Element	Property
Value	$\mathbb{V}$	v	
Address	$\mathbb{A}$	a	$\mathbb{A} \subseteq \mathbb{V}$
Variable	$\mathbb{X}$	$\mathtt{x}$	
Heap	$\mathbb{H}$	h	$\mathbb{H} = \mathbb{A} \rightharpoonup \mathbb{V}$
Environment	$\mathbb{E}$	e	$\mathbb{E} = \mathbb{X} \to \mathbb{A}$
Memory	$\mathbb{M}$	m	$\mathbb{M} = \mathbb{E} \times \mathbb{H}$

Transition System $\mathcal{S} = (\mathbb{S}, \longrightarrow)$

Name	Set	Element	Property
Program Configuration	$\mathcal{Cmd} \times \mathbb{M}$	$\langle c \mid \mathrm{m} \rangle$	
Stack	Σ	σ, ϵ	$\mathbb{S} = \Sigma = (\mathcal{Cmd} \times \mathbb{M})$ stack

Table 3.1: Notations for the concrete trace semantics

We observe that $[\![c]\!]_{\mathcal{R}}$ describes the set of pairs made of an input memory state m_{in} and an output memory state $\mathrm{m}_{\mathrm{out}}$, where $\mathrm{m}_{\mathrm{out}}$ is obtained by executing the sequences of commands c from m_{in}. Thus, it can describe a relation between the input and output memory states of a function f, if c is its command body.

Note that this relational semantics only returns pairs of memory states, without associating them with a command. Indeed, in this case it is not necessary to explicit it, as the command associated to the input memory state is c and the command associated to the output memory state is just **skip**.

In this Thesis, we define an analysis to compute an over-approximation of $[\![c]\!]_{\mathcal{R}}$.

Chapter 4

Abstraction

In this chapter, we design a relational shape abstract domain that over-approximates input-output relations over memory states. This abstract domain relies on new relational connectives, that extends separation logic. Thus, we define in a first time an abstraction of memory heaps based on separation logic that supports inductive data structures. Then, we introduce our new relational connectives and formalize an abstract domain that describes input-output heaps relations and that is built upon the heaps abstraction. We finally design an abstraction for complete memory relations.

4.1 Abstract Heaps

Before defining abstraction relations over memory states, we need to define an abstraction of memory states, that will be at the base of our abstract relations. In this section, we introduce *abstract heaps* directly derived from [CRN07, CR08, CR13], that represent a set of memory heaps and are based on separation logic [Rey02]. In a first time, we consider finite abstract heaps, without unbounded data structures, that enumerate memory cells. In a second time, we extend abstract heaps to support such data structures.

4.1.1 Exact Abstract Heaps based on Separation Logic

We first consider an *exact heap abstraction* without unbounded dynamic data structures, that is, an abstraction of a finite number of memory cells. We assume a countable set $\mathbb{V}^\sharp = \{\alpha, \beta, \delta, \ldots\}$ of *symbolic values* that abstract concrete addresses and values. An *abstract heap* $\mathrm{h}^\sharp \in \mathbb{H}^\sharp$ is a separating conjunction of region predicates that abstract *separated* memory regions [Rey02] (as mentioned in Chapter 2, separating conjunction is denoted by $*_\mathtt{s}$). Thus we write $\mathrm{h}_1^\sharp *_\mathtt{s} \mathrm{h}_2^\sharp$ for the abstract heap that can be split into the two independent sub-abstract heaps $\mathrm{h}_1^\sharp$ and $\mathrm{h}_2^\sharp$. An individual cell is abstracted by an exact *points-to* predicate $\alpha \cdot \mathtt{f} \mapsto \beta$ where the memory cell at the address abstracted

by α with the offset $\mathtt{f}$ contains the value abstracted by β. We note $\alpha \mapsto \beta$ as syntactic sugar of $\alpha \cdot \varnothing \mapsto \beta$ ($\varnothing$ is the null offset). We also use **emp** to describe an empty memory region.

Definition 4.1 (Syntax of exact abstract heaps). *Exact abstract heaps are defined by the following syntax:*

$$\begin{array}{rcll} \mathrm{h}^\sharp\ (\in \mathbb{H}^\sharp) & ::= & \mathbf{emp} & \\ & | & \alpha \cdot \mathtt{f} \mapsto \beta & (\alpha, \beta \in \mathbb{V}^\sharp; \mathtt{f} \in \mathbb{F}) \\ & | & \mathrm{h}_1^\sharp *_\mathtt{s} \mathrm{h}_2^\sharp & (\mathrm{h}_1^\sharp, \mathrm{h}_2^\sharp \in \mathbb{H}^\sharp) \end{array}$$

We now define the meaning of exact abstract heaps using a *concretization function* [CC77], that associates abstract elements to the set of concrete elements that they describe. To concretize an abstract heap, we first need a *valuation* $\nu \in \mathbb{V}^\sharp \to \mathbb{V}$, a function that defines how symbolic values are bound to concrete values and addresses. We recall that $\mathrm{h}_1 \circledast \mathrm{h}_2$ denotes the concrete heap obtained by merging h_1 and h_2 when $\mathbf{dom}(\mathrm{h}_1) \cap \mathbf{dom}(\mathrm{h}_2) = \emptyset$ (see Section 3.2).

Definition 4.2 (Concretization of exact abstract heaps). *The concretization function $\gamma_{\mathbb{H}^\sharp} \in \mathbb{H}^\sharp \to \mathcal{P}(\mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$ maps an abstract heap into a set of pairs made of a concrete heap and a valuation. It is defined by induction on the structure of abstract heaps as follows:*

$$\begin{array}{rcl} \gamma_{\mathbb{H}^\sharp}(\mathbf{emp}) & = & \{([], \nu) \mid \nu \in \mathbb{V}^\sharp \to \mathbb{V}\} \\ \gamma_{\mathbb{H}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \beta) & = & \{([\nu(\alpha) + \mathtt{f} \mapsto \nu(\beta)], \nu) \mid \nu \in \mathbb{V}^\sharp \to \mathbb{V}\} \\ \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp *_\mathtt{s} \mathrm{h}_2^\sharp) & = & \{(\mathrm{h}_1 \circledast \mathrm{h}_2, \nu) \mid (\mathrm{h}_1, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp) \wedge (\mathrm{h}_2, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_2^\sharp)\} \end{array}$$

Example 4.1 (Exact abstract heap). The following exact abstract heap

$$\alpha_0 \mapsto \alpha_1 *_\mathtt{s} \alpha_1 \cdot \mathtt{data} \mapsto \alpha_2 *_\mathtt{s} \alpha_1 \cdot \mathtt{next} \mapsto \alpha_3 *_\mathtt{s} \beta_0 \mapsto \beta_1$$

describes a possible input heap for the `add_last` function of Figure 2.5 (Page 25), where the address of `l` is α_0 and `l` points to a list of one element, and the address of `v` is β_0.

4.1.2 Abstract Heaps with Summarization

This exact abstraction of memory cells does not allow to describe all the states of unbounded dynamic data structures, such as singly linked list or trees. Thus, we must extend abstract heaps with *inductive predicates* to *summarize* memory regions of unbounded size [DOY06, CR08]. For instance, the inductive predicate $\mathbf{list}(\alpha)$ describes a

singly linked list of any length starting at the address α, (where the list is empty, α is the null pointer). The set $\mathbb{I}^\sharp$ describes the set of all inductive predicates (for example we have $\mathbf{list}(\alpha) \in \mathbb{I}^\sharp$), and we note **ind** for any inductive predicate.

Definition 4.3 (Extended syntax of abstract heaps). *We extend the syntax of abstract heaps to support inductive predicates.*

$$\begin{array}{rcll} \mathrm{h}^\sharp\ (\in \mathbb{H}^\sharp) & ::= & \mathbf{emp} & \\ & | & \alpha \cdot \mathtt{f} \mapsto \beta & (\alpha, \beta \in \mathbb{V}^\sharp; \mathtt{f} \in \mathbb{F}) \\ & | & \mathrm{h}_1^\sharp *_\mathrm{s} \mathrm{h}_2^\sharp & (\mathrm{h}_1^\sharp, \mathrm{h}_2^\sharp \in \mathbb{H}^\sharp) \\ & | & \mathbf{ind} & (\mathbf{ind} \in \mathbb{I}^\sharp) \end{array}$$

Now, abstract heaps are *parameterized* by the set of inductive predicates $\mathbb{I}^\sharp$.

More generally, an inductive predicate $\mathbf{ind} \in \mathbb{I}^\sharp$ is defined by a finite set of *rules*. Each rule is a pair made of an abstract heap $\mathrm{h}^\sharp$ and a *pure formula* $\mathrm{p}^\sharp \in \mathbb{P}^\sharp$ that describes numerical constraints over symbolic values.

Definition 4.4 (Syntax of pure formulas). *We give the syntax for pure formulas over symbolic values:*

$$\begin{array}{rcll} \mathrm{p}^\sharp\ (\in \mathbb{P}^\sharp) & ::= & \alpha & (\alpha \in \mathbb{V}^\sharp) \\ & | & \mathrm{v} & (\mathrm{v} \in \mathbb{V}) \\ & | & \mathrm{p}_1^\sharp \oplus \mathrm{p}_2^\sharp & (\mathrm{p}_1^\sharp, \mathrm{p}_2^\sharp \in \mathbb{P}^\sharp) \\ & | & \mathbf{true} & \\ & | & \mathbf{false} & \\ \oplus & ::= & +\,|\,-\,|\,=\,|\,\ldots & \end{array}$$

The $\oplus$ operators of pure formulas are exactly the same as than Figure 3.1 (Page 34).

Example 4.2 (List inductive predicate). The list predicate $\mathbf{list}(\alpha)$ describes the structure of a singly linked list and is defined by induction as follows:

$$\begin{array}{rcl} \mathbf{list}(\alpha) & ::= & \{(\mathbf{emp}, \alpha = 0), \\ & & (\alpha \cdot \mathtt{data} \mapsto \delta *_\mathrm{s} \alpha \cdot \mathtt{next} \mapsto \beta *_\mathrm{s} \mathbf{list}(\beta), \alpha \neq 0)\} \end{array}$$

The first rule of that definition corresponds to the empty list, so that α is the null pointer. The second rule describes the case where the list contains one element and points to an other list (α cannot be the null pointer).

Example 4.3 (Binary tree inductive predicate). We can define similarly the **tree** predicate to describe the structure of a binary tree. We also can enrich the numerical constraint of the second rule to specify that all the elements of the tree are strictly positive:

$$\begin{aligned}\mathbf{tree}(\alpha) \quad ::= \quad & \{(\mathbf{emp},\ \alpha = 0),\\ & (\alpha \cdot \mathtt{e} \mapsto \delta \ *_{\mathtt{s}}\ \alpha \cdot \mathtt{l} \mapsto \beta_{\mathtt{l}} \ *_{\mathtt{s}}\ \alpha \cdot \mathtt{r} \mapsto \beta_{\mathtt{r}}\\ & \quad *_{\mathtt{s}}\ \mathbf{tree}(\beta_{\mathtt{l}}) \ *_{\mathtt{s}}\ \mathbf{tree}(\beta_{\mathtt{r}}),\ \alpha \neq 0 \wedge \delta > 0)\}\end{aligned}$$

Generally, inductive predicates such as $\mathbf{list}(\alpha)$ or $\mathbf{tree}(\alpha)$ summarize a memory region from a specific address. However, we often need to describe properties on different parts of a whole summarized memory region and inductive predicates are not precise enough to do that. For instance, in Figure 2.1 (Page 16), we need to express that the pointer `t` points to the last node of the list, that itself needs to be summarized. Such a property requires a *segment predicate* to be expressed. Segment predicates summarize a memory region between two addresses and allow to split a whole summarized region into many contiguous summarized sub-regions. For the sake of simplicity, we consider segment predicates as inductive predicates with an additional parameter that specifies the ending address of the summarized region. For example, the list segment predicate $\mathbf{listseg}(\alpha, \beta)$ is defined by induction as follows:

$$\begin{aligned}\mathbf{listseg}(\alpha, \beta) \quad ::= \quad & \{(\mathbf{emp}, \alpha = \beta),\\ & (\alpha \cdot \mathtt{data} \mapsto \delta \ *_{\mathtt{s}}\ \alpha \cdot \mathtt{next} \mapsto \gamma \ *_{\mathtt{s}}\ \mathbf{listseg}(\gamma, \beta), \alpha \neq 0)\}\end{aligned}$$

Thus, the **listseg** predicate denotes a list of any length starting at α and ending at β (when the list is empty, we have $\alpha = \beta$).

The concretization of inductive predicates is based on a function $\Delta \in \mathbb{I}^\sharp \to \mathcal{P}_{\text{fin}}(\mathbb{H}^\sharp \times \mathbb{P}^\sharp)$ that maps an inductive predicate into the finite set of rules of its definition.

Definition 4.5 (Concretization of inductive predicates). *The concretization function* $\gamma_{\mathbb{P}^\sharp} \in \mathbb{P}^\sharp \to \mathcal{P}(\mathbb{V} \times (\mathbb{V}^\sharp \to \mathbb{V}))$ *maps a pure formula into a set of pairs made of a concrete value and a valuation whereas* $\gamma_\Sigma \in \mathbb{H}^\sharp \times \mathbb{P}^\sharp \to \mathcal{P}(\mathbb{H}^\sharp \times (\mathbb{V}^\sharp \to \mathbb{V}))$ *concretizes pairs of abstract heap and pure formula. The concretization of inductive predicates extends the concretization of abstract heaps* $\gamma_{\mathbb{H}^\sharp}$*:*

$$
\begin{array}{rcl}
\gamma_{\mathbb{P}^\sharp}(\alpha) & = & \{(\nu(\alpha), \nu) \mid \nu \in \mathbb{V}^\sharp \to \mathbb{V}\} \\
\gamma_{\mathbb{P}^\sharp}(\mathrm{v}) & = & \{(\mathrm{v}, \nu) \mid \nu \in \mathbb{V}^\sharp \to \mathbb{V}\} \\
\gamma_{\mathbb{P}^\sharp}(\mathrm{p}_1^\sharp \oplus \mathrm{p}_2^\sharp) & = & \{(\mathrm{v}_1 \oplus \mathrm{v}_2, \nu) | (\mathrm{v}_1, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}_1^\sharp) \wedge (\mathrm{v}_2, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}_2^\sharp)\} \\
\gamma_{\mathbb{P}^\sharp}(\mathbf{true}) & = & \{(\mathrm{v}, \nu) \mid \mathrm{v} \neq 0 \wedge \nu \in \mathbb{V}^\sharp \to \mathbb{V}\} \\
\gamma_{\mathbb{P}^\sharp}(\mathbf{false}) & = & \{(0, \nu) \mid \nu \in \mathbb{V}^\sharp \to \mathbb{V}\} \\
\\
\gamma_\Sigma(\mathrm{h}^\sharp, \mathrm{p}^\sharp) & = & \{(\mathrm{h}, \nu) \mid (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp) \wedge \exists \mathrm{v}, (\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp) \wedge \mathrm{v} \neq 0\} \\
\\
\gamma_{\mathbb{H}^\sharp}(\mathbf{ind}) & = & \bigcup_{(\mathrm{h}^\sharp, \mathrm{p}^\sharp) \in \Delta(\mathbf{ind})} \gamma_\Sigma(\mathrm{h}^\sharp, \mathrm{p}^\sharp)
\end{array}
$$

Remark 1 (Value in the concretization of pure formulas)**.** *Let* $\mathrm{p}^\sharp \in \mathbb{P}^\sharp$ *a pure formula and* $(\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp)$*. If* $\mathrm{v} = 0$*, that means that* $\mathrm{p}^\sharp$ *is not a satisfiable pure formula (*0 *means false in the concrete semantic). As an example, consider that* $\mathrm{p}^\sharp = (\alpha = 1) \wedge (\alpha = 2)$*. It is obvious that* $\mathrm{p}^\sharp$ *cannot be satisfied, its concretization is thus* $\{(0, \nu) \mid \nu \in \mathbb{V}^\sharp \to \mathbb{V}\}$*. Therefore, we can express that a pure formula* $\mathrm{p}^\sharp$ *is satisfiable using the constraint* $\mathrm{v} \neq 0$ *if* $(\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp)$*, like we did in the definition of* γ_Σ *in Definition 4.5.*

Example 4.4 (Abstract heap with a summarized region). The following abstract heap

$$
\begin{array}{l}
\alpha_0 \mapsto \alpha_1 *_\mathsf{s} \alpha_4 \mapsto \alpha_3 \\
*_\mathsf{s}\ \mathbf{listseg}(\alpha_1, \alpha_2) \\
*_\mathsf{s}\ \alpha_2 \cdot \mathtt{next} \mapsto \alpha_3 *_\mathsf{s} \alpha_2 \cdot \mathtt{data} \mapsto \delta \\
*_\mathsf{s}\ \mathbf{list}(\alpha_3)
\end{array}
$$

describes all the possible output heaps for function `concat` of Figure 2.1 (Page 16) for the case where the first list is non-empty, where the addresses of `l1` and `l2` are respectively α_0 and α_4.

4.2 Abstract Heap Relations

We now define *abstract heap relations*, that describes input-output relations over memory heaps. They consist of new relational connectives, that rely on abstract heaps (i.e. separation logic formulas).

4.2.1 Relational Connectives

As explained in Chapter 2, two abstract heaps at different program points cannot describe relations (they describe any relation made of a pair of states that satisfy the state constraints). Thus, we propose *abstract heap relations* that describe a set of pairs made of an *input* heap $\mathrm{h_i}$ and an *output* heap $\mathrm{h_o}$. Abstract heap relations are defined by new logical connectives over abstract heaps as follows:

- the *identity relation* $\mathtt{Id}(\mathrm{h}^\sharp)$ describes pairs of heaps that are equal and both abstracted by $\mathrm{h}^\sharp$.
- the *transform-into relation* $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]$ describes pairs corresponding to the transformation of a heap abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ into a heap abstracted by $\mathrm{h}_\mathrm{o}^\sharp$.
- the *relational separating conjunction* $\mathrm{r}_1^\sharp *_\mathtt{R} \mathrm{r}_2^\sharp$ of two abstract heap relations $\mathrm{r}_1^\sharp$ and $\mathrm{r}_2^\sharp$ denotes a relation that can be described by combining independently the relations described by $\mathrm{r}_1^\sharp$ and $\mathrm{r}_2^\sharp$ on disjoint memory regions.

Definition 4.6 (Syntax of abstract heap relations). *Abstract heap relations are defined by the following syntax:*

$$\begin{array}{rcll} \mathrm{r}^\sharp\ (\in \mathbb{R}^\sharp) & ::= & \mathtt{Id}(\mathrm{h}^\sharp) & (\mathrm{h}^\sharp \in \mathbb{H}^\sharp) \\ & | & [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp] & (\mathrm{h}_\mathrm{i}^\sharp, \mathrm{h}_\mathrm{o}^\sharp \in \mathbb{H}^\sharp) \\ & | & \mathrm{r}_1^\sharp *_\mathtt{R} \mathrm{r}_2^\sharp & (\mathrm{r}_1^\sharp, \mathrm{r}_2^\sharp \in \mathbb{R}^\sharp) \end{array}$$

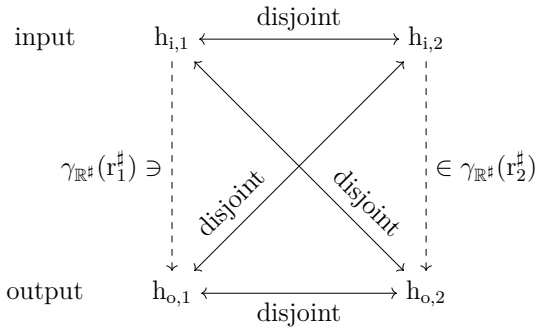
The concretization of abstract heap relations also requires using valuations as it also needs to define the concrete values that symbolic values denote.

Definition 4.7 (Concretization of abstract heap relations). *The concretization function $\gamma_{\mathbb{R}^\sharp} \in \mathbb{R}^\sharp \to \mathcal{P}(\mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$ maps an abstract heap relation into a set of triples made of an input heap, an output heap and a valuation. It is defined by induction on the structure of $\mathrm{r}^\sharp$:*

$$\begin{array}{rcl} \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) & = & \{(\mathrm{h},\ \mathrm{h},\ \nu) \mid (\mathrm{h},\ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\} \\ \gamma_{\mathbb{R}^\sharp}([\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]) & = & \{(\mathrm{h_i},\ \mathrm{h_o},\ \nu) \mid (\mathrm{h_i},\ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_\mathrm{i}^\sharp) \wedge (\mathrm{h_o},\ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_\mathrm{o}^\sharp)\} \\ \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp *_\mathtt{R} \mathrm{r}_2^\sharp) & = & \{(\mathrm{h}_{\mathrm{i},1} \circledast \mathrm{h}_{\mathrm{i},2},\ \mathrm{h}_{\mathrm{o},1} \circledast \mathrm{h}_{\mathrm{o},2},\ \nu) \mid \\ & & \qquad (\mathrm{h}_{\mathrm{i},1},\ \mathrm{h}_{\mathrm{o},1},\ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp) \wedge \mathbf{dom}(\mathrm{h}_{\mathrm{i},1}) \cap \mathbf{dom}(\mathrm{h}_{\mathrm{o},2}) = \emptyset \\ & & \quad \wedge (\mathrm{h}_{\mathrm{i},2},\ \mathrm{h}_{\mathrm{o},2},\ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_2^\sharp) \wedge \mathbf{dom}(\mathrm{h}_{\mathrm{i},2}) \cap \mathbf{dom}(\mathrm{h}_{\mathrm{o},1}) = \emptyset\} \end{array}$$

In the case of identity relations, we observe that the two concrete heaps are physically and totally identical: the same memory cells containing the same data. In the case of transform-into relations, we have no information about how the input concrete heap has been transformed into the output concrete heap. They may share common memory cells, or not, or they may even be identical. In the case of the relational separating conjunction,

we observe two important points. The first point is that the concrete input (respectively output) concrete heap of the first abstract relation is disjoint from the input (respectively output) concrete heap of the second abstract relation. The second point is that the *input* concrete heap of the first abstract relation is also disjoint from the *output* concrete heap of the second one (and vice versa). These two points make abstract heap relations totally independent: an abstract heap relation describes a *local* transformation. The figure below gives a graphical representation of the concretization of $\mathrm{r}_1^\sharp \mathbin{*_{\mathtt{R}}} \mathrm{r}_2^\sharp$.



We remark that $*_{\mathtt{R}}$ is commutative and associative. We can also define the neutral element $\mathbf{emp}_{\mathtt{R}}$ for $*_{\mathtt{R}}$ that is syntactic sugar for $\mathtt{Id}(\mathbf{emp})$ and $[\mathbf{emp} \dashrightarrow \mathbf{emp}]$: we have $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathbf{emp})) = \gamma_{\mathbb{R}^\sharp}([\mathbf{emp} \dashrightarrow \mathbf{emp}])$.

Example 4.5. The following abstract heap relation

$$\begin{aligned}&\mathtt{Id}(\alpha_0 \mapsto \alpha_1 *_{\mathtt{S}} \alpha_4 \mapsto \alpha_3)\\ &*_{\mathtt{R}} \mathtt{Id}(\mathbf{listseg}(\alpha_1, \alpha_2))\\ &*_{\mathtt{R}} [(\alpha_2 \cdot \mathtt{next} \mapsto \alpha_5) \dashrightarrow (\alpha_2 \cdot \mathtt{next} \mapsto \alpha_3)] *_{\mathtt{R}} \mathtt{Id}(\alpha_2 \cdot \mathtt{data} \mapsto \delta)\\ &*_{\mathtt{R}} \mathtt{Id}(\mathbf{list}(\alpha_3))\end{aligned}$$

describes the inferred abstract heap relation for function `concat` of Figure 2.5 (Page 16) for the case where the first list is non-empty, where the addresses of `l1` and `l2` are respectively α_0 and α_3.

4.2.2 Properties

We observe that abstract heap relations satisfy the following properties:

Theorem 4.1 (Equivalence and weakening properties of abstract heap relations). *Let* $\mathrm{h}^\sharp, \mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp, \mathrm{h}_{\mathrm{i},0}^\sharp, \mathrm{h}_{\mathrm{i},1}^\sharp, \mathrm{h}_{\mathrm{o},0}^\sharp, \mathrm{h}_{\mathrm{o},1}^\sharp$ *be abstract heaps. Then, we have the following properties:*

1. $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp))$
2. $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp])$ *(the opposite inclusion may not hold, as observed in Example 4.6);*
3. $\gamma_{\mathbb{R}^\sharp}([\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp] *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]) \subseteq \gamma_{\mathbb{R}^\sharp}([(\mathrm{h}_{\mathrm{i},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{i},1}^\sharp) \dashrightarrow (\mathrm{h}_{\mathrm{o},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{o},1}^\sharp)])$ *(the opposite inclusion may not hold, as observed in Example 4.7).*

Property 1 allows to split and merge identity relations. Property 2 allows to forget the identity relation on a heap, weakening it into a transform-into relation whereas Property 3 allows to forget that two relations are independent by merging respectively their inputs and output states. The relational static analysis described in Chapter 5 is mainly based on these properties.

Example 4.6 (Expressiveness 1). Let $\mathrm{r}_1^\sharp = \mathtt{Id}(\mathbf{list}(\alpha))$ and $\mathrm{r}_2^\sharp = [\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\alpha)]$. We observe that $\mathrm{r}_1^\sharp$ describes the identity relation applied to a well-formed linked list starting from the address α, whereas $\mathrm{r}_2^\sharp$ describes any transformation that inputs such a list and outputs such a list, but may modify its content, add or remove elements, or may modify the order of list elements (except for the first one which remains at address α). This means that $\gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp) \subset \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_2^\sharp)$.

Example 4.7 (Expressiveness 2). Let $\mathrm{r}_1^\sharp = [\mathbf{list}(\alpha) \dashrightarrow \mathbf{emp}] *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \mathbf{list}(\beta)]$ and $\mathrm{r}_2^\sharp = [\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]$. The abstract heap relation $\mathrm{r}_1^\sharp$ describes two distinct transformations: the first one expresses the deallocation of a list starting from the address α and the second one expresses the allocation of a list starting from the address β. The abstract heap relation $\mathrm{r}_2^\sharp$ describes simply the transformation of a list starting from α into a list starting from β. In $\mathrm{r}_1^\sharp$, we know that the two lists are physically different but in $\mathrm{r}_2^\sharp$, we have no information about the output list is obtained from the input list. This means that the two lists may be either physically different, may share some memory cells, or may even be totally equal. Actually, we have $\gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp) \subset \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_2^\sharp)$.

Proof of Theorem 4.1. We are now going to prove Theorem 4.1. For each property, we just reduce both concretizations using their definition to show that the equality (for Property 1) or the inclusion (for Property 2 and Property 3) holds:

1. Proof of $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp))$:

$$\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)) \\
&\quad = \{(\mathrm{h}, \mathrm{h}, \nu) \mid (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)\} \\
&\quad = \{(\mathrm{h}_0 \circledast \mathrm{h}_1, \mathrm{h}_0 \circledast \mathrm{h}_1, \nu) \mid (\mathrm{h}_0, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp) \wedge (\mathrm{h}_1, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp)\}
\end{aligned}$$

$$\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp)) \\
&\quad = \{(\mathrm{h}_0 \circledast \mathrm{h}_1, \mathrm{h}_0' \circledast \mathrm{h}_1', \nu) \mid \\
&\qquad (\mathrm{h}_0, \mathrm{h}_0', \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp)) \wedge (\mathrm{h}_1, \mathrm{h}_1', \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_1^\sharp)) \wedge \\
&\qquad \mathbf{dom}(\mathrm{h}_0) \cap \mathbf{dom}(\mathrm{h}_1') = \emptyset \wedge \mathbf{dom}(\mathrm{h}_1) \cap \mathbf{dom}(\mathrm{h}_0') = \emptyset\} \\
&\quad = \{(\mathrm{h}_0 \circledast \mathrm{h}_1, \mathrm{h}_0 \circledast \mathrm{h}_1, \nu) \mid (\mathrm{h}_0, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp) \wedge (\mathrm{h}_1, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp)\}\}
\end{aligned}$$

   So $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp))$

2. Proof of $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp])$:

$$\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \\
&\quad = \{(\mathrm{h}, \mathrm{h}, \nu) \mid (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\}
\end{aligned}$$

$$\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]) \\
&\quad = \{(\mathrm{h}_0, \mathrm{h}_1, \nu) \mid (\mathrm{h}_0, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp) \wedge (\mathrm{h}_1, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\}
\end{aligned}$$

   We clearly have: $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp])$

3. Proof of

   $\gamma_{\mathbb{R}^\sharp}([\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp] *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]) \subseteq \gamma_{\mathbb{R}^\sharp}([(\mathrm{h}_{\mathrm{i},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{i},1}^\sharp) \dashrightarrow (\mathrm{h}_{\mathrm{o},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{o},1}^\sharp)])$:

$$
\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}] *_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}]) \\
&\quad = \{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid \\
&\qquad (\mathrm{h}_{\mathrm{i},0}, \mathrm{h}_{\mathrm{o},0}, \nu) \in \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}]) \wedge \\
&\qquad (\mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}]) \wedge \\
&\qquad \mathbf{dom}(\mathrm{h}_{\mathrm{i},0}) \cap \mathbf{dom}(\mathrm{h}_{\mathrm{o},1}) = \emptyset \wedge \mathbf{dom}(\mathrm{h}_{\mathrm{i},1}) \cap \mathbf{dom}(\mathrm{h}_{\mathrm{o},0}) = \emptyset\} \\
&\quad = \{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid \\
&\qquad (\mathrm{h}_{\mathrm{i},0}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{i},0}) \wedge (\mathrm{h}_{\mathrm{o},0}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{o},0}) \wedge \\
&\qquad (\mathrm{h}_{\mathrm{i},1}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{i},1}) \wedge (\mathrm{h}_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{o},1}) \wedge \\
&\qquad \mathbf{dom}(\mathrm{h}_{\mathrm{i},0}) \cap \mathbf{dom}(\mathrm{h}_{\mathrm{o},1}) = \emptyset \wedge \mathbf{dom}(\mathrm{h}_{\mathrm{i},1}) \cap \mathbf{dom}(\mathrm{h}_{\mathrm{o},0}) = \emptyset\}
\end{aligned}
$$

$$
\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}([(\mathrm{h}^\sharp_{\mathrm{i},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{i},1}) \dashrightarrow (\mathrm{h}^\sharp_{\mathrm{o},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{o},1})]) \\
&\quad = \{(\mathrm{h}_{\mathrm{i}}, \mathrm{h}_{\mathrm{o}}, \nu) \mid \\
&\qquad (\mathrm{h}_{\mathrm{i}}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{i},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{i},1}) \wedge (\mathrm{h}_{\mathrm{o}}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{o},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{o},1})\} \\
&\quad = \{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid \\
&\qquad (\mathrm{h}_{\mathrm{i},0}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{i},0}) \wedge (\mathrm{h}_{\mathrm{o},0}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{o},0}) \wedge \\
&\qquad (\mathrm{h}_{\mathrm{i},1}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{i},1}) \wedge (\mathrm{h}_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{o},1})\}
\end{aligned}
$$

So finally:

$$\gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}] *_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}]) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{o},1}])$$

□

4.3 Abstract Memory Relations and Disjunctive Abstract Domain

From now, we have defined abstractions for relations between two heaps. In this section, we give an abstraction for relations between two memory states. We also define an abstraction for a disjunction of memory relations.

4.3.1 Numerical Abstract Domains

In our heap abstraction, we simply name addresses and values with symbolic values. We have no information about their concrete values. However, it is crucial in our analysis to remember whether a pointer is null or not. Moreover, it is also necessary to be able to capture numerical invariants to increase the precision of the analysis. Such information can be captured with *numerical abstract domains* such as intervals [CC77] or convex polyhedra [CH78]. Thus, we enrich our abstraction with a numerical abstract domain $\mathbb{N}^\sharp$. We do not fix a particular numerical abstract domain, as it does not change our analysis. It just requires to implement all the functions for numerical abstract domains

given all along Chapter 5 and to satisfy their soundness conditions.

A *numerical abstract value* $\mathrm{n}^\sharp \in \mathbb{N}^\sharp$ abstracts the numerical value of the symbolic values that appear in an abstract heap relation.

The concretization of abstract numerical domains $\gamma_{\mathbb{N}^\sharp}$ gives a concrete value to each symbolic value. Thus, it just binds a numerical abstract value into a set of valuations:

$$\gamma_{\mathbb{N}^\sharp} \in \mathbb{N}^\sharp \to \mathcal{P}(\mathbb{V}^\sharp \to \mathbb{V})$$

4.3.2 Abstract Memory Relations

To have a complete abstraction of memory relations, we have to abstract environments. An *abstract environment* $\mathrm{e}^\sharp \in \mathbb{E}^\sharp$ is simply a function that maps program variables to symbolic values that correspond to their concrete addresses (thus $\mathbb{E}^\sharp = \mathbb{X} \to \mathbb{V}^\sharp$).

Finally, an *abstract memory relation* $\mathrm{m}^\sharp_\mathcal{R} \in \mathbb{M}^\sharp_\mathcal{R} = \mathbb{E}^\sharp \times \mathbb{R}^\sharp \times \mathbb{N}^\sharp$ is a triple made of an abstract environment $\mathrm{e}^\sharp$, that binds program variables into their symbolic values in the abstract heap relation $\mathrm{r}^\sharp$, and a numerical abstract value $\mathrm{n}^\sharp$ that abstracts the value of symbolic values of $\mathrm{r}^\sharp$.

Definition 4.8 (Concretization of abstract memory relations). *The concretization of abstract memory relations* $\gamma_{\mathbb{M}^\sharp_\mathcal{R}} \in \mathbb{M}^\sharp_\mathcal{R} \to \mathcal{P}(\mathbb{M} \times \mathbb{M})$ *maps an abstract memory relation* $\mathrm{m}^\sharp_\mathcal{R} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$ *into the set of the pairs made of its concrete input and output memories.*

$$\begin{aligned}\gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp) \quad = \quad & \{((\nu \circ \mathrm{e}^\sharp,\ \mathrm{h_i}),\ (\nu \circ \mathrm{e}^\sharp,\ \mathrm{h_o})) \mid \\ & \qquad (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \wedge \nu \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}^\sharp)\}\end{aligned}$$

We observe that the concrete environment is always the same in the input and in the output memory as the address at which a variable is stored never changes between two program points.

Example 4.8 (Abstract memory relation). Let $\mathrm{m}^\sharp_\mathcal{R} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$ be an abstract memory relation. if $\mathrm{e}^\sharp(\texttt{l1}) = \alpha_0$ and $\mathrm{e}^\sharp(\texttt{l2}) = \alpha_4$ and $\mathrm{r}^\sharp$ is:

$$\begin{aligned}&\texttt{Id}(\alpha_0 \mapsto \alpha_1 *_\texttt{S} \alpha_4 \mapsto \alpha_3) \\ &*_\texttt{R} \texttt{Id}(\mathbf{listseg}(\alpha_1, \alpha_2)) \\ &*_\texttt{R} [(\alpha_2 \cdot \texttt{next} \mapsto \alpha_5) \dashrightarrow (\alpha_2 \cdot \texttt{next} \mapsto \alpha_3)] *_\texttt{R} \texttt{Id}(\alpha_2 \cdot \texttt{data} \mapsto \delta) \\ &*_\texttt{R} \texttt{Id}(\mathbf{list}(\alpha_3))\end{aligned}$$

describes the inferred abstract heap relation for function `concat` of Figure 2.5 (Page 16) for the case where the first list is non-empty. The abstract numerical value $\mathrm{n}^\sharp$ should express that $\alpha_1 \neq 0$ and that $\alpha_5 = 0$.

4.3.3 Abstract Memory States

Although our analysis mainly manipulates abstract memory relations, it requires to manipulate *abstract memory states*, specifically in Chapter 8. An abstract memory state $\mathrm{m}^\sharp \in \mathbb{M}^\sharp$ abstracts a set of concrete memory states and consists of a triple made of an abstract environement, an abstract heap and an abstract numerical value.

Definition 4.9 (Concretization of abstract memory states). *The concretization of abstract memory states $\gamma_{\mathbb{M}} \in \mathbb{M}^\sharp \to \mathcal{P}(\mathbb{M})$ maps an abstract memory state $\mathrm{m}^\sharp = (\mathrm{e}^\sharp, \mathrm{h}^\sharp, \mathrm{n}^\sharp)$ into the set of its concrete memory states.*

$$\gamma_{\mathbb{M}}(\mathrm{e}^\sharp, \mathrm{h}^\sharp, \mathrm{n}^\sharp) \quad = \quad \{(\nu \circ \mathrm{e}^\sharp,\ \mathrm{h}) \mid (\mathrm{h}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{h}^\sharp) \wedge \nu \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}^\sharp)\}$$

4.3.4 Abstract Disjunctions

The analysis algorithm may require to *unfold* inductive predicates (see Section 5.3). Unfolding such predicates may generate a *finite set* of abstract memory relations. Consequently, the analysis needs an abstraction layer that reasons over it. We thus let $\mathbb{R}^\vee = \mathcal{P}_{\text{fin}}(\mathbb{M}^\sharp_\mathcal{R})$ be the set of disjunctions of abstract memory relations. A disjunction of abstract memory relations $\mathbb{r}^\vee \in \mathbb{R}^\vee$ simply represents a finite set of abstract memory relations.

Definition 4.10 (Concretization of the disjunction of abstract memory relations). *The concretization function $\gamma_{\mathbb{R}^\vee} \in \mathbb{R}^\vee \to \mathcal{P}(\mathbb{M} \times \mathbb{M})$ maps a disjunction of abstract memory relations $\mathbb{r}^\vee$ into the set of the pairs made of its concrete input and output memories.*

$$\gamma_{\mathbb{R}^\vee}(\mathbb{r}^\vee) = \bigcup_{\mathrm{m}^\sharp_\mathcal{R} \in \mathbb{r}^\vee} \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_\mathcal{R})$$

Similarly, we define $\mathbb{M}^\vee = \mathcal{P}_{\text{fin}}(\mathbb{M}^\sharp)$, the set of disjunctions of abstract memory states. A disjunction of abstract memory states $\mathbb{m}^\vee \in \mathbb{M}^\vee$ simply represents a finite set of abstract memory relations.

Definition 4.11 (Concretization of the disjunction of abstract memory states). *The concretization function $\gamma_{\mathbb{M}^\vee} \in \mathbb{M}^\vee \to \mathcal{P}(\mathbb{M})$ maps a disjunction of abstract memory states $\mathbb{m}^\vee$ into the sets of its concrete memory states.*

$$\gamma_{\mathbb{M}^\vee}(\mathbb{m}^\vee) = \bigcup_{\mathrm{m}^\sharp \in \mathbb{m}^\vee} \gamma_{\mathbb{M}}(\mathrm{m}^\sharp)$$

Name	Set	Element	Abstracts
Symbolic Values	$\mathbb{V}^\sharp$	$\alpha, \beta, \delta, \ldots$	$\mathbb{V}$
Pure Formulas	$\mathbb{P}^\sharp$	$\mathrm{p}^\sharp$	$\mathcal{P}(\mathbb{V} \times (\mathbb{V}^\sharp \to \mathbb{V}))$
Inductive Predicates	$\mathbb{I}^\sharp$	$\mathbf{list}(\alpha), \ldots$	$\mathcal{P}(\mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$
Abstract Heaps	$\mathbb{H}^\sharp$	$\mathrm{h}^\sharp$	$\mathcal{P}(\mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$
Abstract Heap Relations	$\mathbb{R}^\sharp$	$\mathrm{r}^\sharp$	$\mathcal{P}(\mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$
Numerical Abstract Domains	$\mathbb{N}^\sharp$	$\mathrm{n}^\sharp$	$\mathcal{P}(\mathbb{V}^\sharp \to \mathbb{V})$
Abstract Environments	$\mathbb{E}^\sharp$	$\mathrm{e}^\sharp$	$\mathcal{P}(\mathbb{E})$
Abstract Memory States	$\mathbb{M}^\sharp$	$\mathrm{m}^\sharp$	$\mathcal{P}(\mathbb{M})$
Abstract Memory Relations	$\mathbb{M}^\sharp_\mathcal{R}$	$\mathrm{m}^\sharp_\mathcal{R}$	$\mathcal{P}(\mathbb{M} \times \mathbb{M})$
Disjunction of Abs. Memory States	$\mathbb{M}^\vee$	$\mathbb{m}^\vee$	$\mathcal{P}(\mathbb{M})$
Disjunction of Abs. Memory Relations	$\mathbb{R}^\vee$	$\mathbb{r}^\vee$	$\mathcal{P}(\mathbb{M} \times \mathbb{M})$

Table 4.1: Notations for the abstract domains and meta-variables used to denote an element of the corresponding domain.

Example 4.9 (Disjunction of abstract memory relations). In this example, we provide the disjunction of abstract memory relations inferred for function `concat` of Figure 2.5 (Page 16). It is made of two disjuncts, whose the first one is already provided in Example 4.8. The second disjunct $\mathrm{m}^\sharp_\mathcal{R} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$ corresponds to the case where `l1` is empty, where $\mathrm{e}^\sharp(\texttt{l1}) = \alpha_0$, $\mathrm{e}^\sharp(\texttt{l2}) = \alpha_4$ and $\mathrm{r}^\sharp$ is:

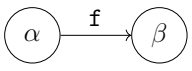
$$\texttt{Id}(\alpha_0 \mapsto \alpha_1 *_\texttt{S} \alpha_4 \mapsto \alpha_3) *_\texttt{R} \texttt{Id}(\mathbf{list}(\alpha_3))$$

The abstract numerical value $\mathrm{n}^\sharp$ should express that $\alpha_1 = 0$.

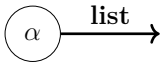
4.4 Graphical Representation

Table 4.1 summarizes all the notations for the abstract domains defined in this chapter. In this section, we give the graphical representations of abstract heaps, abstract heap relations and abstract memory states and relations.

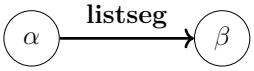
Abstract heaps. Abstract heaps are graphically represented by *shape graphs*. A symbolic value corresponds to a node. A points-to predicate $\alpha \cdot \texttt{f} \mapsto \beta$ is represented by an edge labeled by the field `f`, from the node α to the node β.



An inductive predicate (such as $\mathbf{list}(\alpha)$) is represented by a thick edge labeled by the name of the predicate, without destination node.



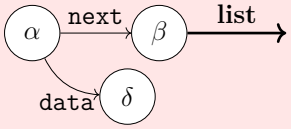
A segment predicate is represented similarly to inductive predicates, except that the destination node is represented. For instance for the segment $\mathbf{list}(\alpha, \beta)$:



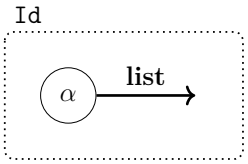
Example 4.10 (Graphical representation of an abstract heap). The following abstract heap

$$\alpha \cdot \mathtt{next} \mapsto \beta \ast_{\mathtt{s}} \alpha \cdot \mathtt{data} \mapsto \delta \ast_{\mathtt{s}} \mathbf{list}(\beta)$$

is graphically represented by:

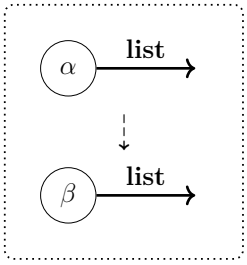


Abstract heap relations. An abstract heap relation $\mathrm{r}^\sharp$ is graphically represented by a box. There are two kinds of boxes, one for identity relations and the other for transform-into relations. The graphical representation of two independent abstract heap relations $\mathrm{r}_1^\sharp \ast_{\mathtt{R}} \mathrm{r}_2^\sharp$ is simply two boxes: a box for $\mathrm{r}_1^\sharp$ and a box for $\mathrm{r}_2^\sharp$. The box corresponding to $\mathtt{Id}(\mathrm{h}^\sharp)$ contains the shape graph corresponding to $\mathrm{h}^\sharp$. Its left top corner is labeled by the word '`Id`'. For instance, the box corresponding to $\mathtt{Id}(\mathbf{list}(\alpha))$ is:



The box corresponding to $[\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}}^\sharp]$ is divided into two parts: the first one (on the top) contains the shape graph corresponding to $\mathrm{h}_{\mathrm{i}}^\sharp$ and the second one (on the bottom) the shape graph corresponding to $\mathrm{h}_{\mathrm{o}}^\sharp$. The box contains a vertical dashed arrow (that looks like '$\dashrightarrow$') to distinguish these two parts. For instance, the box that corresponds

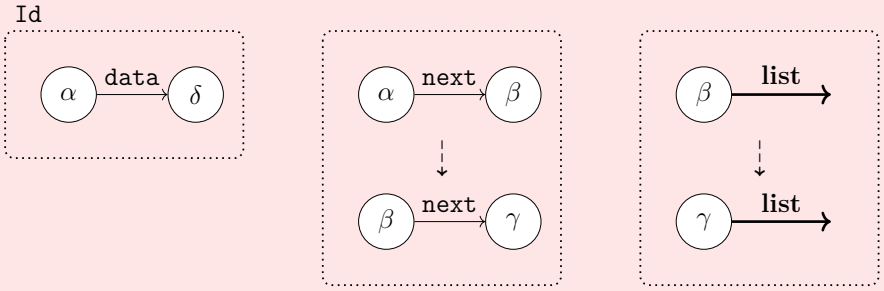
to $[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]$ is:



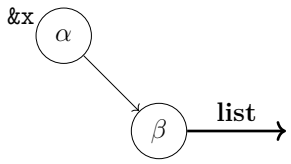
Example 4.11 (Graphical representation of an abstract heap relation). The following abstract heap relation

$$\mathtt{Id}(\alpha \cdot \mathtt{data} \mapsto \delta) *_{\mathtt{R}} [\alpha \cdot \mathtt{next} \mapsto \beta \dashrightarrow \alpha \cdot \mathtt{next} \mapsto \gamma] *_{\mathtt{R}} [\mathbf{list}(\beta) \dashrightarrow \mathbf{list}(\gamma)]$$

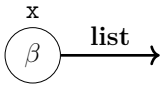
is graphically represented by:



Abstract memory states and relations. The graphical representation of abstract memory states and relations is similar to the one of abstract heaps and abstract heap relations, except that we make the program variables of the abstract environment explicit. For all variables $\mathtt{x} \in \mathbb{X}$, if $\mathrm{e}^\sharp(\mathtt{x}) = \alpha$, we write '$\mathtt{\&x}$' near the node corresponding to α. For instance, for the abstract heap $\alpha \mapsto \beta *_{\mathtt{S}} \mathbf{list}(\beta)$, we have:



To simplify our figures, we sometimes do not explicit graphically the node corresponding to the address of a variable $\mathtt{x}$, we only write '$\mathtt{x}$' near the node pointed by $\mathtt{x}$. For the same abstract heap above, we have:



4.5 Related Works

4.5.1 State Shape Abstractions with Separation

Our separation logic based shape abstraction is directly derived from [CRN07, CR08, CR13]. It is very similar to other shape abstractions based on separation logic, such as [DOY06, MNCL06] that use a list segment abstraction, whereas ours is parametrized by inductive definitions. An other shape abstraction based on separation logic is [BCC+07], that proposes generic high-order inductive predicates. This kind of inductive predicates allows to abstract different kinds of linked list, such as cyclic doubly-linked list of acyclic singly-linked lists, or singly-linked lists of cyclic doubly linked lists with back-pointers to head nodes. All these states abstractions describe memory cells. However, there are several works that use an other approach like [HR05, Kas06, ABB06, BNR08], that partition the heap using region inference.

An other nature of separated memories has been proposed in [HHR+11, HHL+15], that makes use of tree or forest of automata, where each automota describes a separated memory region.

4.5.2 Extension of Separation Logic

To our knowledge, our work is the first to propose relational logical connectives based on separation logic that support inductive predicates and describe input-output heaps relations. However, several works have already enriched separation logic in order to perform specific analyses.

For instance Charguéraud et al. [CP17] introduced the temporary read-only permission through a new connective for separation logic. This connective offers read-only access to any heap fragment described with a separation logic formula. Like our relational connectives, this read-only connective can ensure that some part of a heap is left unchanged, as it can only be read. An important extension of separation logic is the concurrent separation logic [O'H07], which allows independent reasoning about threads that access separate storage. The new connective $\|$ allows to evaluate two terms of separation logic in parallel. The main difference of these extensions and our connectives is that they modify directly the original separation logic; whereas in our extension, the relational connectives encompass the terms of separation logic, without modifying them. Moreover, We care for relations, and not just states (concretization). The fact that we encompass separation logic is a consequence of that, as we sometimes also talk about states with regular separation logic.

Fu et al. [FLF+10] introduced extension of separation logic to specify historical execution traces of heaps in the context of concurrent programs. They also have a new separating conjunction connective but for disjoint traces, whereas our relational separating conjunction expresses independent transformations. Desynchronized separation [CCR15]

also introduces a notion of overlaid state in separation logic, but does not support inductive predicates as our abstraction does. Instead, it allows to reason on abstractions of JavaScript open objects seen as dictionaries.

4.5.3 Other Kinds of Relational Properties

Our relational abstraction describes input-output states relations. However, many other works describe other kinds of relations. For instance, [FFJ12] and [BDES12] describe relations between the structures and their content, and [CR08] describes relations over disjoint regions in the same memory state. As our input-output states abstract relations are an extension of [CR08], they directly take benefits of its expressiveness. Also, [Rug04] describes the balance invariant in AVL trees , [CR07] the previous-next relationship in doubly-linked lists, and [MBCC07] maintains a relation between the shape of a linked list and its length. Related works that present input-output relations are discussed in Section 5.11.

Chapter 5

Relational Intra-procedural Shape Analysis

In this chapter, we define the abstract relational semantics, that is an over-approximation of the concrete relational semantics of our programming language. It is computed by a forward abstract interpretation, that starts from the identity relation of a given pre-condition, and infers abstract memory relations between the input and output states of the program. This analysis does not handle function calls.

5.1 Introduction

In this chapter, we propose a static analysis to compute the abstract memory relations that we defined in Section 4.3. It proceeds by forward abstract interpretation [CC77], starting from the abstract heap relation $\mathtt{Id}(\mathrm{h}^\sharp)$ where $\mathrm{h}^\sharp$ is a pre-condition supplied by the user. Indeed, before executing a program, any transformation has been performed on its initial state, so the initial relation of the analysis is the identity relation.

More generally, the analysis of a command $c \in \mathcal{Cmd}$ is a function $[\![c]\!]^\sharp_{\mathcal{R}}$ that over-approximates the concrete relational semantics $[\![c]\!]_{\mathcal{R}}$ defined in Section 3.4. It inputs an abstract disjunction describing a previous transformation $\mathcal{T}$ done on the input *before* running c and returns an other abstract disjunction describing that transformation $\mathcal{T}$ followed by the execution of c. Thus, $[\![c]\!]^\sharp_{\mathcal{R}}$ should meet the following soundness condition:

$$\begin{array}{l}\forall \mathtt{r}^\vee \in \mathbb{R}^\vee, \forall (\mathrm{m}_0, \mathrm{m}_1) \in \gamma_{\mathbb{R}^\vee}(\mathtt{r}^\vee), \forall \mathrm{m}_2 \in \mathbb{M}, \\ \qquad (\mathrm{m}_1, \mathrm{m}_2) \in [\![c]\!]_{\mathcal{R}} \Longrightarrow (\mathrm{m}_0, \mathrm{m}_2) \in \gamma_{\mathbb{R}^\vee}([\![c]\!]^\sharp_{\mathcal{R}}(\mathtt{r}^\vee))\end{array}$$

The abstract relational semantics $[\![c]\!]^\sharp_{\mathcal{R}}$ is defined by induction over the syntax of commands, similarly to $[\![c]\!]_{\mathcal{R}}$. To design it, it thus requires to define abstract evaluations of

$$\frac{\mathrm{e}^\sharp(\mathtt{x}) = \alpha}{\mathbf{eval}_{\mathrm{L}}(\mathtt{x}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \varnothing)}$$

$$\frac{\mathbf{eval}_{\mathrm{L}}(\ell, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f})}{\mathbf{eval}_{\mathrm{L}}(\ell \cdot \mathtt{g}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f} + \mathtt{g})}$$

$$\frac{\mathbf{eval}_{\mathrm{E}}(e, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \alpha)}{\mathbf{eval}_{\mathrm{L}}(*e, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \varnothing)}$$

$$\frac{\mathbf{eval}_{\mathrm{L}}(\ell, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f}) \quad \mathrm{r}^\sharp = \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)}{\mathbf{eval}_{\mathrm{E}}(\ell, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta, \beta)}$$

$$\frac{\mathbf{eval}_{\mathrm{L}}(\ell, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f}) \quad \mathrm{r}^\sharp = \mathrm{r}_0^\sharp *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow (\mathrm{h}_{\mathrm{o}}^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)]}{\mathbf{eval}_{\mathrm{E}}(\ell, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta, \beta)}$$

$$\frac{\mathbf{eval}_{\mathrm{L}}(\ell, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \varnothing)}{\mathbf{eval}_{\mathrm{E}}(\&\ell, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \alpha)}$$

$$\frac{\alpha \text{ fresh}}{\mathbf{eval}_{\mathrm{E}}(\mathrm{v}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathrm{v})}$$

$$\frac{\mathbf{eval}_{\mathrm{E}}(e_1, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha_1, \mathrm{p}_1^\sharp) \quad \mathbf{eval}_{\mathrm{E}}(e_2, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha_2, \mathrm{p}_2^\sharp) \quad \alpha_3 \text{ fresh}}{\mathbf{eval}_{\mathrm{E}}(e_1 \oplus e_2, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha_3, \mathrm{p}_1^\sharp \oplus \mathrm{p}_2^\sharp)}$$

Figure 5.1: Abstract evaluation of locations $\mathbf{eval}_{\mathrm{L}} \in \mathcal{Lval} \times \mathbb{E}^\sharp \times \mathbb{R}^\sharp \to \mathbb{V}^\sharp \times \mathbb{F}$ and expressions $\mathbf{eval}_{\mathrm{E}} \in \mathcal{Expr} \times \mathbb{E}^\sharp \times \mathbb{R}^\sharp \to \mathbb{V}^\sharp \times \mathbb{P}^\sharp$. We remind that $\varnothing$ designs the null offset.

l-values and expressions (Section 5.2), abstract assignments (Section 5.4), abstract allocations and deallocations (Section 5.5), and abstract condition tests (Section 5.6). This abstract relational semantics also defines an unfolding operator over inductive predicates to materialize memory cells (Section 5.3), and standard lattice operations to handle deal with control flow joins and loops (Section 5.7 and Section 5.8).

5.2 Abstract Evaluation of L-values and Expressions

In this section, we define how l-values and expressions are evaluated in our static analysis from abstract memory relations. We first consider only the abstract heap relations that do not contain inductive predicates.

We start by defining the abstract evaluation of l-values $\mathbf{eval}_{\mathrm{L}} \in \mathcal{Lval} \times \mathbb{E}^\sharp \times \mathbb{R}^\sharp \to \mathbb{V}^\sharp \times \mathbb{F}$ and expressions $\mathbf{eval}_{\mathrm{E}} \in \mathcal{Expr} \times \mathbb{E}^\sharp \times \mathbb{R}^\sharp \to \mathbb{V}^\sharp \times \mathbb{P}^\sharp$ from an abstract environment $\mathrm{e}^\sharp$ and an abstract heap relation $\mathrm{r}^\sharp$. They follow the same principles as $\mathcal{L}[\![loc]\!]$ and $\mathcal{E}[\![exp]\!]$ defined

in 3.4 and their definition rules are given in Figure 5.1.

The function $\mathbf{eval}_{\mathtt{L}}$ evaluates a l-value into a pair of an abstract value and an offset that corresponds to the address of the l-value in $\mathrm{r}^\sharp$.

The abstract evaluation $\mathbf{eval}_{\mathtt{E}}$ should return the symbolic value $\alpha \in \mathbb{V}^\sharp$ in $\mathrm{r}^\sharp$ corresponding to the result of the concrete evaluation of an expression *exp*. However, if *exp* is of the form $exp_1 \oplus exp_2$ or v, then there is no symbolic value α in $\mathrm{r}^\sharp$ that can result from the abstract evaluation of *exp*. To deal with this, the function $\mathbf{eval}_{\mathtt{E}}$ returns a pair of a symbolic value $\alpha \in \mathbb{V}^\sharp$ and a pure formula $\mathrm{p}^\sharp \in \mathbb{P}^\sharp$. The pure formula $\mathrm{p}^\sharp$ is simply a translation of the expression, and α is a (potentially fresh) name for $\mathrm{p}^\sharp$. The concretization of this pair is such that if $(\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp)$, then $\nu(\alpha) = \mathrm{v}$.

When the evaluation needs to read the content of a cell (when we apply $\mathbf{eval}_{\mathtt{E}}$ on a l-value *loc*), it needs to look at the value stored in the cell. For instance, if $\mathbf{eval}_{\mathtt{L}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f})$, in the case where $\mathrm{r}^\sharp = \mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \beta)$, it is obvious that the cell at address $\alpha \cdot \mathtt{f}$ contains β. In the case where $\mathrm{r}^\sharp = [(\alpha \cdot \mathtt{f} \mapsto \delta) \dashrightarrow (\alpha \cdot \mathtt{f} \mapsto \beta)]$, we remark that δ was the value in the cell at address $\alpha \cdot \mathtt{f}$ in the heap at the beginning of the analysis whereas β corresponds to its last value. Thus, $\mathbf{eval}_{\mathtt{E}}(loc, \mathrm{e}^\sharp, \mathrm{r}^\sharp)$ returns (β, β).

Theorem 5.1 (Soundness of $\mathbf{eval}_{\mathtt{L}}$ and $\mathbf{eval}_{\mathtt{E}}$). *The functions* $\mathbf{eval}_L$ *and* $\mathbf{eval}_E$ *are sound: Let* $\mathrm{e}^\sharp \in \mathbb{E}^\sharp$, $\mathrm{r}^\sharp \in \mathbb{R}^\sharp$, $(\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp)$, *exp* $\in \mathcal{E}xpr$ *and loc* $\in \mathcal{L}val$ *then:*

$$\begin{array}{l}\mathbf{eval}_L(\ell, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathtt{f}) \\ \quad \Rightarrow \mathcal{L}[\![\ell]\!](\nu \circ \mathrm{e}^\sharp,\ \mathrm{h_o}) = \nu(\alpha) + \mathtt{f}\end{array}$$

$$\begin{array}{l}\mathbf{eval}_E(e, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathrm{p}^\sharp) \\ \quad \Rightarrow \mathcal{E}[\![e]\!](\nu \circ \mathrm{e}^\sharp,\ \mathrm{h_o}) = \nu(\alpha) \wedge (\nu(\alpha),\ \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp)\end{array}$$

Example 5.1 (Abstract evaluation of the expression $((*\mathtt{x})\cdot\mathtt{f})+2$). We assume that $\mathrm{e}^\sharp(\mathtt{x}) = \alpha_0$ and $\mathrm{r}^\sharp = \mathtt{Id}(\alpha_0 \mapsto \alpha_1) *_{\mathtt{R}} [(\alpha_1 \cdot \mathtt{f} \mapsto \alpha_2) \dashrightarrow (\alpha_1 \cdot \mathtt{f} \mapsto \alpha_3)]_{\mathrm{t}^\sharp}$.

We remark that $\mathbf{eval}_{\mathtt{L}}(*\mathtt{x}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha_1, \varnothing)$, that $\mathbf{eval}_{\mathtt{E}}((*\mathtt{x}) \cdot \mathtt{f}, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta_1, \alpha_3)$ and that $\mathbf{eval}_{\mathtt{E}}(2, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta_2, 2)$ where β_1 and β_2 are fresh symbolic values.

Finally, we obtain that $\mathbf{eval}_{\mathtt{E}}(((*\mathtt{x}) \cdot \mathtt{f}) + 2, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\beta_3, \alpha_3 + 2)$, where β_3 is a fresh symbolic value.

5.3 Unfolding

In the previous section, we considered only abstract memory relations without inductive predicates. We now lift this restriction.

Unfolding Abstract Memory Relations

The abstract evaluation of a l-value may require to *unfold* inductive predicates. Indeed, when a l-value is summarized by an inductive predicate, there is no points-to predicate corresponding to this l-value and so, its abstract evaluation cannot be done. The analysis first proceeds to the *unfolding* [CR08] of the inductive predicate in order to materialize it into points-to predicates. This is performed by the function $\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathcal{R}}}$. This step generates a *finite disjunction* of abstract memory relations, where the inductive predicate has been syntactically substituted by the rules of its definition, as defined in Section 4.1 (one disjunct per rule of the inductive predicate). Also, the pure formula of each rule is evaluated and added in each disjunct of abstract memory relations. The irrelevant disjuncts with the l-value are discarded (for example the case where the summarized cell is null). The abstract evaluation of the l-value is then performed for each valid disjunct. This process is known in shape analysis as *materialization* [SRW02, DOY06, CR08]. The abstract memory relations unfolding operator $\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ builds upon the abstract heap relations unfolding operator $\mathbf{unfold}_{\mathbb{R}^\sharp}$.

The definition of $\mathbf{unfold}_{\mathbb{R}^\sharp}$ itself builds upon the (provided) abstract heap unfolding operator $\mathbf{unfold}_{\mathbb{H}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{H}^\sharp \to \mathcal{P}_{\text{fin}}(\mathbb{H}^\sharp \times \mathbb{P}^\sharp)$. It takes a symbolic value α which is the origin of an inductive predicate and an abstract heap $\mathrm{h}^\sharp$ that contains this inductive predicate. It returns the set of pairs $\{(\mathrm{h}^\sharp_{\mathrm{u}}, \mathrm{p}^\sharp)\}$ where each $\mathrm{h}^\sharp_{\mathrm{u}}$ refines $\mathrm{h}^\sharp$ following each definition rule of the inductive predicate and $\mathrm{p}^\sharp$ is the pure formula of the corresponding rule. For instance, $\mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, \mathbf{list}(\alpha))$ is $\{(\mathbf{emp}, \alpha = 0), (\alpha \cdot \texttt{next} \mapsto \alpha_n *_{\mathtt{S}} \alpha \cdot \texttt{data} \mapsto \alpha_d *_{\mathtt{S}} \mathbf{list}(\alpha_n), \alpha \neq 0)\}$. If there is no inductive predicate attached to α in $\mathrm{h}^\sharp$, we let $\mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, \mathrm{h}^\sharp) = \{(\mathrm{h}^\sharp, \mathbf{true})\}$. This operator is sound in the sense that, $\gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)$ is included in $\cup\{\gamma_\Sigma(\mathrm{h}^\sharp_{\mathrm{u}}, \mathrm{p}^\sharp) \mid (\mathrm{h}^\sharp_{\mathrm{u}}, \mathrm{p}^\sharp) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, \mathrm{h}^\sharp)\}$.

The unfolding operator over abstract heap relations $\mathbf{unfold}_{\mathbb{R}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{R}^\sharp \to \mathcal{P}_{\text{fin}}(\mathbb{R}^\sharp \times \mathbb{P}^\sharp)$ uses $\mathbf{unfold}_{\mathbb{H}^\sharp}$ to unfold the inductive predicate at abstract heap relations level.

Definition 5.1 (Abstract Heap Relation Unfolding). *Let $\mathrm{r}^\sharp \in \mathbb{R}^\sharp$, we define the unfolding operator for abstract heap relations* $\mathbf{unfold}_{\mathbb{R}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{R}^\sharp \to \mathcal{P}_{\text{fin}}(\mathbb{R}^\sharp \times \mathbb{P}^\sharp)$*:*

- $\mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, \texttt{Id}(\mathrm{h}^\sharp)) = \{(\texttt{Id}(\mathrm{h}^\sharp_{\mathrm{u}}), \mathrm{p}^\sharp) \mid (\mathrm{h}^\sharp, \mathrm{p}^\sharp) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, \mathrm{h}^\sharp)\}$
- $\mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, [\mathrm{h}^\sharp_{\mathrm{i}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o}}]_{\mathrm{t}^\sharp}) = \{([\mathrm{h}^\sharp_{\mathrm{i,u}} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o,u}}]_{\mathrm{t}^\sharp}, \mathrm{p}^\sharp_{\mathrm{i,u}} \wedge \mathrm{p}^\sharp_{\mathrm{o,u}}) \mid (\mathrm{h}^\sharp_{\mathrm{i,u}}, \mathrm{p}^\sharp_{\mathrm{i,u}}) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, \mathrm{h}^\sharp_{\mathrm{i}}) \wedge (\mathrm{h}^\sharp_{\mathrm{o,u}}, \mathrm{p}^\sharp_{\mathrm{o,u}}) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, \mathrm{h}^\sharp_{\mathrm{o}})\}$
- $\mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, \mathrm{r}^\sharp_0 *_{\mathtt{R}} \mathrm{r}^\sharp_1) = \{(\mathrm{r}^\sharp_{0,\mathrm{u}} *_{\mathtt{R}} \mathrm{r}^\sharp_1, \mathrm{p}^\sharp) \mid (\mathrm{r}^\sharp_{0,\mathrm{u}}, \mathrm{p}^\sharp) \in \mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, \mathrm{r}^\sharp_0)\}$*, when α carries an inductive predicate in $\mathrm{r}^\sharp_0$.*

Unfolding an inductive predicate under an identity relation only consists on unfolding the abstract heap and conserving the identity relation over the unfolded abstract

heap. Unfolding under a transform-into relation requires to unfold *independently* both the input and the output abstract heaps of the relation. The resulting pure formula is the conjunction of the pure formulas of each unfolded abstract heaps. Finally, unfolding an inductive under a relational separating conjunction consists on unfolding locally the abstract heap relation where the inductive predicate appears.

Definition 5.2 (Abstract Memory Relation Unfolding). *Let* $\mathrm{m}^\sharp_{\mathcal{R}} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp) \in \mathbb{M}^\sharp_{\mathcal{R}}$*, we define the unfolding operator for abstract memory relations* $\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathcal{R}}} \in \mathbb{V}^\sharp \times \mathbb{M}^\sharp_{\mathcal{R}} \to \mathcal{P}_{\text{fin}}(\mathbb{M}^\sharp_{\mathcal{R}})$*:*

$$\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathcal{R}}}(\alpha, \mathrm{m}^\sharp_{\mathcal{R}}) = \{(\mathrm{e}^\sharp, \mathrm{r}^\sharp_{\mathrm{u}}, \mathbf{guard}_{\mathbb{N}^\sharp}(\mathrm{p}^\sharp, \mathrm{n}^\sharp)) \mid (\mathrm{r}^\sharp_{\mathrm{u}}, \mathrm{p}^\sharp) \in \mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, \mathrm{r}^\sharp)\}$$

The function $\mathbf{unfold}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ applies the numerical conditions returned by $\mathbf{unfold}_{\mathbb{R}^\sharp}$ using $\mathbf{guard}_{\mathbb{N}^\sharp}$. This relies on the numerical guard $\mathbf{guard}_{\mathbb{N}^\sharp} \in \mathbb{P}^\sharp \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$ that updates an abstract numerical value $\mathrm{n}^\sharp$ taking into account the effects of a pure formula $\mathrm{p}^\sharp$. This numerical guard should satisfy the following condition:

Assumption 5.1 (Soundness of $\mathbf{guard}_{\mathbb{N}^\sharp}$). *Let* $\mathrm{p}^\sharp \in \mathbb{P}^\sharp, \mathrm{n}^\sharp \in \mathbb{N}^\sharp$*. The function* $\mathbf{guard}_{\mathbb{N}^\sharp}$ *is sound if:*

$$\nu \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}^\sharp) \wedge (\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp) \wedge \mathrm{v} \neq 0 \implies \nu \in \gamma_{\mathbb{N}^\sharp}(\mathbf{guard}_{\mathbb{N}^\sharp}(\mathrm{p}^\sharp, \mathrm{n}^\sharp))$$

Theorem 5.2 (Soundness of unfolding operators). *Let* $\gamma_\Pi \in \mathbb{R}^\sharp \times \mathbb{P}^\sharp \to \mathcal{P}(\mathbb{H} \times \mathbb{H} \times \mathbb{V}^\sharp \to \mathbb{V})$ *be the concretization function of pairs of abstract heap relation and pure formula defined as follow:*

$$\gamma_\Pi(\mathrm{r}^\sharp, \mathrm{p}^\sharp) = \{(\mathrm{h_i}, \mathrm{h_o}, \nu) \mid (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \wedge (\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp) \wedge \mathrm{v} \neq 0\}$$

Let $\mathrm{r}^\sharp \in \mathbb{R}^\sharp, \mathrm{m}^\sharp_{\mathcal{R}} \in \mathbb{M}^\sharp_{\mathcal{R}}$ *and* $\alpha \in \mathbb{V}^\sharp$*. Then, the unfolding operators are sound, in the sense that:*

$$\gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \subseteq \bigcup\{\gamma_\Pi(\mathrm{r}^\sharp_{\mathrm{u}}, \mathrm{p}^\sharp) \mid (\mathrm{r}^\sharp_{\mathrm{u}}, \mathrm{p}^\sharp) \in \mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, \mathrm{r}^\sharp)\}$$

$$\gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}}) \subseteq \bigcup\{\gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^{\mathrm{u}}}) \mid (\mathrm{m}^\sharp_{\mathcal{R}^{\mathrm{u}}}) \in \mathbf{unfold}_{\mathbb{M}^\sharp_{\mathcal{R}}}(\alpha, \mathrm{m}^\sharp_{\mathcal{R}})\}$$

Example 5.2 (Unfolding a transform-into relation). Consider the following abstract heap relation $\mathrm{r}^\sharp = [\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\alpha)]_{\mathrm{t}^\sharp}$. Unfolding $\mathrm{r}^\sharp$ at symbolic value α should generate four pairs of abstract heap relations and pure formulas, included two cases where the pure formula is $\alpha = 0 \wedge \alpha \neq 0$. These cases should be discarded by $\mathbf{guard}_{\mathbb{N}^\sharp}$ at the abstract memory relation level. The other remaining cases are thus: $([\mathbf{emp} \dashrightarrow \mathbf{emp}]_{\mathrm{t}^\sharp}, \alpha = 0 \wedge \alpha = 0)$ and $([(\alpha \cdot \mathtt{data} \mapsto \delta_\mathrm{i} *_\mathrm{s} \alpha \cdot \mathtt{next} \mapsto \beta_\mathrm{i} *_\mathrm{s} \mathbf{list}(\beta_\mathrm{i})) \dashrightarrow (\alpha \cdot \mathtt{data} \mapsto \delta_\mathrm{o} *_\mathrm{s} \alpha \cdot \mathtt{next} \mapsto \beta_\mathrm{o} *_\mathrm{s} \mathbf{list}(\beta_\mathrm{o}))]_{\mathrm{t}^\sharp}, \alpha \neq 0 \wedge \alpha \neq 0)$. Remark

that in the latter case we obtained different symbolic values for the `data` and `next` fields of α because we have unfolded independently each abstract heap.

Example 5.3 (Abstract memory relation unfolding). Let us consider the analysis of the insertion function of Figure 1.3 (Page 12). This function should be applied to states where `l` is a non null list pointer (the list should have at least one element). The analysis should start from the abstract memory relation $(e_0^\sharp, r_0^\sharp, n_0^\sharp)$ where:

$$r_0^\sharp = \mathtt{Id}(\alpha_0 \mapsto \alpha_1 *_{\mathtt{S}} \mathbf{list}(\alpha_1)) \text{ and } e_0^\sharp(\mathtt{l}) = \alpha_0$$

In this example, we omit `v` for the sake of concision. Note that we did not specify that α_1 is not null. Before the loop entry, the analysis computes the abstract memory relation $(e_1^\sharp, r_1^\sharp, n_1^\sharp)$ where:

$$r_1^\sharp = \mathtt{Id}(\alpha_0 \mapsto \alpha_1 *_{\mathtt{S}} \mathbf{list}(\alpha_1)) *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \alpha_2 \mapsto \alpha_1] \text{ and } e_1^\sharp(\mathtt{l}) = \alpha_0 \text{ and } e_1^\sharp(\mathtt{c}) = \alpha_2$$

To deal with the test `c->next != NULL`, the analysis should materialize α_1. This unfolding is performed under the `Id` connective, and produces the following abstract heap relation:

$$\begin{array}{l}(\mathtt{Id}(\alpha_0 \mapsto \alpha_1 *_{\mathtt{S}} \alpha_1 \cdot \mathtt{next} \mapsto \alpha_3 *_{\mathtt{S}} \alpha_1 \cdot \mathtt{data} \mapsto \beta_0 *_{\mathtt{S}} \mathbf{list}(\alpha_3)) \\ \qquad *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\alpha_2 \mapsto \alpha_1))], \alpha_1 \neq 0)\end{array}$$

$$\text{with } e_1^\sharp(\mathtt{l}) = \alpha_0 \text{ and } e_1^\sharp(\mathtt{c}) = \alpha_2$$

Then, the condition $\alpha_1 \neq 0$ is kept in the numerical abstract value. We observe that we have automatically inferred that the input of the function must be non-empty, as this condition also applies to the abstract input memory. In turn, the effect of the condition test and of the assignment in the loop body can be precisely analyzed from this abstract memory relation.

Remark 2. *In the next of this Chapter, we assume that unfolding is already performed before reading a l-value or an expression. So that we do not make the steps when unfolding occurs explicit.*

5.4 Assignment

In this section, we define the transfer function for assignments $\mathbf{assign}_{\mathbb{M}_{\mathcal{R}}^\sharp} \in \mathcal{Lval} \times \mathcal{Expr} \times \mathbb{M}_{\mathcal{R}}^\sharp \to \mathbb{M}_{\mathcal{R}}^\sharp$. Recall that the concrete assignment over a pair of memory states (m_i, m_o) results in an other pair of memory states (m_i, m_o') where only the output state m_o has been modified. Likewise the abstract assignment will keep the abstract input state unmodified.

It will also preserve as many relations between the abstract input and output states as possible.

The main part of the algorithm consists in the function $\mathbf{assign}_{\mathbb{R}^\sharp}$ that inputs a symbolic value α and a field $\mathtt{f}$ (the address), a symbolic value β (the value) and an abstract heap relation $\mathrm{r}^\sharp$. It performs the following steps:

1. Decompose inductively $\mathrm{r}^\sharp$ when it is of the form $\mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp$ until we find the points-to predicate whose address is $\alpha \cdot \mathtt{f}$. The remainder of the abstract relation is integrally preserved.
2. If the abstract heap relation that was found is of the form $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]$, replace it by $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}'}^\sharp]$ where $\mathrm{h}_{\mathrm{o}'}^\sharp$ reflects the assignment in the abstract heap $\mathrm{h}_\mathrm{o}^\sharp$.
3. If step 1 leads to an abstract heap relation of the form $\mathtt{Id}(\mathrm{h}^\sharp)$ to be found, decompose it into $\mathtt{Id}(\mathrm{h}_1^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_2^\sharp)$ where $\mathrm{h}_2^\sharp$ only contains the points-to predicate whose address is $\alpha \cdot \mathtt{f}$. Then weaken $\mathtt{Id}(\mathrm{h}_2^\sharp)$ into $[\mathrm{h}_2^\sharp \dashrightarrow \mathrm{h}_2^\sharp]$. Finally, proceed to the assignment in $[\mathrm{h}_2^\sharp \dashrightarrow \mathrm{h}_2^\sharp]$ and let $\mathtt{Id}(\mathrm{h}_1^\sharp)$ unchanged.

For example:

$$\begin{aligned}&\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta,\ \mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \delta))\\ &\qquad = \mathtt{Id}(\mathrm{h}^\sharp) *_{\mathtt{R}} [(\alpha \cdot \mathtt{f} \mapsto \delta) \dashrightarrow (\alpha \cdot \mathtt{f} \mapsto \beta)]\end{aligned}$$

On top of this algorithm, $\mathbf{assign}_{\mathbb{M}^\sharp_\mathcal{R}}$ evaluates the assignment $\ell = e$ over the abstract memory relation $\mathrm{m}^\sharp_\mathcal{R} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$ using the following steps:

1. Evaluate ℓ and e with respectively $\mathbf{eval}_{\mathtt{L}}$ and $\mathbf{eval}_{\mathtt{E}}$. Note that this may require prior unfoldings.
2. Update the abstract numerical value $\mathrm{n}^\sharp$ with the result of the evaluation of e, using the function $\mathbf{assign}_{\mathbb{N}^\sharp}$.
3. Update $\mathrm{r}^\sharp$ with $\mathbf{assign}_{\mathbb{R}^\sharp}$.

Definition 5.3 (Assignments for abstract heap relations). *We define the assignment function for abstract heap relations* $\mathbf{assign}_{\mathbb{R}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \times \mathbb{R}^\sharp \to \mathbb{R}^\sharp$*:*

- $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp) = \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_2^\sharp$, *if* $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{r}_1^\sharp) = \mathrm{r}_2^\sharp$
- $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp *_{\mathtt{S}} (\alpha \cdot \mathtt{f} \mapsto \gamma)]) = [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp *_{\mathtt{S}} (\alpha \cdot \mathtt{f} \mapsto \beta)]$
- $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \delta)) = \mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} [\alpha \cdot \mathtt{f} \mapsto \delta \dashrightarrow \alpha \cdot \mathtt{f} \mapsto \beta]$

Case of a Relational Separating Conjunction. We now assume that $\mathrm{r}^\sharp = \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp$. The points-to predicate at address $\alpha \cdot \mathtt{f}$ can only appear in one of $\mathrm{r}_0^\sharp$ or $\mathrm{r}_1^\sharp$. If it appears in $\mathrm{r}_0^\sharp$, the assignment should have no effect on $\mathrm{r}_1^\sharp$ (and vice versa).

The function $\mathbf{assign}_{\mathbb{R}^\sharp}$ can thus be applied recursively on the sub-abstract heap relation where the points-to predicate appears. This relies on the same principle as the Frame rule [Rey02] for separation logic, but for abstract heap relations. More generally, if $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{r}_1^\sharp) = \mathrm{r}_2^\sharp$, then $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp) = \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_2^\sharp$.

Case of a Transform-into Relation. In the case where $\mathrm{r}^\sharp = [\mathrm{h}_0^\sharp \dashrightarrow \mathrm{h}_1^\sharp]$, $\mathbf{assign}_{\mathbb{R}^\sharp}$ applies on $\mathrm{h}_1^\sharp$ the Frame rule [Rey02] of separation logic (separating the points-to predicate at address $\alpha \cdot \mathtt{f}$ from the rest). Then it updates this points-to predicate to point to β, producing a new abstract heap $\mathrm{h}_2^\sharp$. So a valid definition for this assignment is $[\mathrm{h}_0^\sharp \dashrightarrow \mathrm{h}_2^\sharp]$.

Case of an Identity Relation. We now assume that $\mathrm{r}^\sharp = \mathtt{Id}(\mathrm{h}^\sharp)$. If $\mathrm{h}^\sharp = \alpha \cdot \mathtt{f} \mapsto \delta *_{\mathtt{S}} \mathrm{h}_0^\sharp$, two preliminary steps are necessary before updating $\alpha \cdot \mathtt{f}$. Indeed, two important points need to be considered: first the points-to predicate $\alpha \cdot \mathtt{f} \mapsto \delta$ cannot be substituted by $\alpha \cdot \mathtt{f} \mapsto \beta$ under the $\mathtt{Id}$ connective, because the assignment breaks the identity relation. Second, the assignment should only modify the points-to predicate $\alpha \cdot \mathtt{f} \mapsto \delta$, and should preserve the identity relation over $\mathrm{h}_0^\sharp$, to improve precision.

The first step consists in splitting the identity relation into two identity relations. As observed in Theorem 4.1, $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp *_{\mathtt{S}} \mathrm{h}_1^\sharp)) = \gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp))$. In our case, $\mathrm{r}^\sharp$ is split into $\mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \delta) *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_0^\sharp)$. The first identity relation contains only the points-to predicate that is going to be modified and the second identity relation contains the other parts of $\mathrm{h}^\sharp$ that may only be read (but not written) the assignment.

The second step consists on weakening $\mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \delta)$ into a transform-into relation. In Theorem 4.1, we also have $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp])$. This property allows to weaken $\mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \delta)$ into $[(\alpha \cdot \mathtt{f} \mapsto \delta) \dashrightarrow (\alpha \cdot \mathtt{f} \mapsto \delta)]$.

After these steps, the analysis can perform the assignment in the obtained transform-into relation and left unchanged the split identity relation. This relies on the combination of the assignment in the cases of relational separating conjunction relations and transform-into relations. Thus, a valid definition for this assignment is:

$$\mathtt{Id}(\mathrm{h}_0^\sharp) *_{\mathtt{R}} [\alpha \cdot \mathtt{f} \mapsto \gamma \dashrightarrow \alpha \cdot \mathtt{f} \mapsto \beta]$$

Recall that $\mathrm{h}[\mathrm{a} \leftarrow \mathrm{v}]$ is the concrete heap where we update the content of the cell at address a with the value v in the concrete heap h. We now give the soundness theorem for the functions $\mathbf{assign}_{\mathbb{R}^\sharp}$.

Theorem 5.3 (Soundness of $\mathbf{assign}_{\mathbb{R}^\sharp}$). *Let $\alpha, \beta \in \mathbb{V}^\sharp, \mathtt{f} \in \mathbb{F}$ and $\mathrm{r}_0^\sharp \in \mathbb{R}^\sharp$. The function $\mathbf{assign}_{\mathbb{R}^\sharp}$ is sound:*

$$\begin{aligned}&\forall (\mathrm{h}_0, \mathrm{h}_1, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_0^\sharp)\\ &\quad \Longrightarrow (\mathrm{h}_0, \mathrm{h}_1[\nu(\alpha) + \mathtt{f} \leftarrow \nu(\beta)], \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{r}_0^\sharp))\end{aligned}$$

Definition 5.4 (Assignment in abstract memory relations). *Let $\mathrm{m}_{\mathcal{R}}^\sharp = (\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)$ be an abstract memory relation. If $\mathbf{eval}_{\mathcal{L}}(\ell, \mathrm{e}^\sharp, \mathrm{r}_0^\sharp) = (\alpha, \mathtt{f})$ and $\mathbf{eval}_{\mathcal{E}}(e, \mathrm{e}^\sharp, \mathrm{r}_0^\sharp) = (\beta, \mathrm{p}^\sharp)$, and if $\mathbf{assign}_{\mathbb{N}^\sharp}(\beta, \mathrm{p}^\sharp, \mathrm{n}_0^\sharp) = \mathrm{n}_1^\sharp$ and $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{r}_0^\sharp) = \mathrm{r}_1^\sharp$, then:*

$$\mathbf{assign}_{\mathbb{M}_{\mathcal{R}}^\sharp}(\ell, e, (\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)) = (\mathrm{e}^\sharp, \mathrm{r}_1^\sharp, \mathrm{n}_1^\sharp)$$

The function $\mathbf{assign}_{\mathbb{M}^\sharp_\mathcal{R}}$ applies respectively $\mathbf{eval}_\mathtt{L}$ and $\mathbf{eval}_\mathtt{E}$ on l and e. Remark that this step may unfold inductive predicates, as explained in the previous section. It then applies $\mathbf{assign}_{\mathbb{R}^\sharp}$ on the address returned by $\mathbf{eval}_\mathtt{L}$ and the value returned by $\mathbf{eval}_\mathtt{E}$. Remind that $\mathbf{eval}_\mathtt{E}$ returns a pair $(\beta, \mathrm{p}^\sharp)$, such as $\beta \in \mathbb{V}^\sharp$ and $\mathrm{p}^\sharp \in \mathbb{P}^\sharp$. To keep track of the numerical constraint $\beta = \mathrm{p}^\sharp$, the abstract assignment requires a function that over-approximates this constraint and put it in the abstract numerical value $\mathrm{n}^\sharp$. This is performed by the function $\mathbf{assign}_{\mathbb{N}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{P}^\sharp \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$, whose soundness assumption is:

Assumption 5.2 (Soundness of $\mathbf{assign}_{\mathbb{N}^\sharp}$). *Let $\beta \in \mathbb{V}^\sharp, \mathrm{p}^\sharp \in \mathbb{P}^\sharp$ and $\mathrm{n}^\sharp \in \mathbb{N}^\sharp$, then $\mathbf{assign}_{\mathbb{N}^\sharp}$ is sound if:*

$$\forall (\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\beta = \mathrm{p}^\sharp) \wedge \nu \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}^\sharp) \wedge \mathrm{v} \neq 0 \quad \Longrightarrow \quad \nu \in \gamma_{\mathbb{N}^\sharp}(\mathbf{assign}_{\mathbb{N}^\sharp}(\beta, \mathrm{p}^\sharp, \mathrm{n}^\sharp))$$

Theorem 5.4 (Soundness of $\mathbf{assign}_{\mathbb{M}^\sharp_\mathcal{R}}$). *Let $\mathit{l} \in \mathcal{Lval}, e \in \mathcal{Expr}$ and $\mathrm{m}^\sharp_\mathcal{R} \in \mathbb{M}^\sharp_\mathcal{R}$. Let $(\mathrm{m}_0, \mathrm{m}_1) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_\mathcal{R})$ such that $\mathrm{m}_1 = (\mathrm{e}_1, \mathrm{h}_1)$, then:*

$$(\mathrm{m}_0,\ (\mathrm{e}_1, \mathrm{h}_1[\mathcal{L}[\![\mathit{l}]\!](\mathrm{m}_1) \leftarrow \mathcal{E}[\![e]\!](\mathrm{m}_1)])) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathbf{assign}_{\mathbb{M}^\sharp_\mathcal{R}}(\mathit{l}, e, \mathrm{m}^\sharp_\mathcal{R}))$$

Example 5.4 (Assignment in transform-into relation). In this example, we consider the effect of $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha_1, \mathtt{f}, \beta_2, [\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_1])$, with

$$\mathrm{h}^\sharp_1 = \alpha_1 \cdot \mathtt{f} \mapsto \beta_1 \ast_\mathtt{s} \alpha_2 \cdot \mathtt{g} \mapsto \beta_2$$

Performing the assignment in $\mathrm{h}^\sharp_1$ produces the following abstract heap:

$$\mathrm{h}^\sharp_2 = \alpha_1 \cdot \mathtt{f} \mapsto \beta_2 \ast_\mathtt{s} \alpha_2 \cdot \mathtt{g} \mapsto \beta_2$$

Finally, $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha_1, \mathtt{f}, \beta_2, [\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_1]) = [\mathrm{h}^\sharp_0 \dashrightarrow \mathrm{h}^\sharp_2]$.

Example 5.5 (Assignment in an identity relation). In this example, we consider the effect of $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha_0, \mathtt{f}, \beta_1, \mathrm{r}^\sharp)$ with

$$\mathrm{r}^\sharp = \mathtt{Id}(\alpha_0 \cdot \mathtt{f} \mapsto \beta_0 \ast_\mathtt{s} \mathrm{h}^\sharp_1)$$

After splitting $\mathrm{r}^\sharp$ into $\mathrm{r}^\sharp_0 \ast_\mathtt{R} \mathrm{r}^\sharp_1$ such as $\mathrm{r}^\sharp_0 = \mathtt{Id}(\alpha_0 \cdot \mathtt{f} \mapsto \beta_0)$ and $\mathrm{r}^\sharp_1 = \mathtt{Id}(\mathrm{h}^\sharp_1)$, the analysis needs to weaken $\mathrm{r}^\sharp_0$. Then, the weakening of $\mathrm{r}^\sharp_0$ is $[(\alpha_0 \cdot \mathtt{f} \mapsto \beta_0) \dashrightarrow (\alpha_0 \cdot \mathtt{f} \mapsto \beta_0)]$. In turn, the analysis performs the assignment for abstract heap relations on the form

$\mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp$ and results:

$$[(\alpha_0 \cdot \mathtt{f} \mapsto \beta_0) \dashrightarrow (\alpha_0 \cdot \mathtt{f} \mapsto \beta_1)] *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_1^\sharp)$$

5.5 Allocation and Deallocation

In this section, we define the transfer functions for allocations $\mathbf{alloc}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ and deallocations $\mathbf{free}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ when reasoning abstract memory relations. We first comment on allocations.

Allocations.

The function $\mathbf{alloc}_{\mathbb{M}^\sharp_{\mathcal{R}}}(\ell, \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}, \mathrm{m}^\sharp_{\mathcal{R}})$ returns an abstract memory relation that over-approximates the effect of allocation $\ell = \mathbf{malloc}(\{\mathtt{f}_1, \ldots, \mathtt{f}_n\})$ in $\mathrm{m}^\sharp_{\mathcal{R}}$. It represents the creation of a new memory block of n cells (one cell per field of $\{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$), and assigns the address of this new block to the l-value ℓ. The resulting abstract memory relation should express that the new block has been freshly allocated, and the rest of the memory was left untouched.

The major part of the algorithm consists in the creation of a *single memory cell*. This is performed by the function $\mathbf{alloc}_{\mathbb{R}^\sharp}$ that inputs an abstract value β, a field $\mathtt{f}$ and an abstract heap relation $\mathrm{r}^\sharp$. It creates the cell $\beta \cdot \mathtt{f} \mapsto \delta$ (where δ is a fresh abstract value) and returns an abstract heap relation that expresses the allocation of this cell.

Definition 5.5 (Allocation for abstract heap relations). *We define the allocation function for abstract heap relations* $\mathbf{alloc}_{\mathbb{R}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{R}^\sharp \to \mathbb{R}^\sharp$*:*

$$\mathbf{alloc}_{\mathbb{R}^\sharp}(\beta, \mathtt{f}, \mathrm{r}^\sharp) = \mathrm{r}^\sharp *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{f} \mapsto \delta)],$$

where δ is a fresh symbolic value

The definition of $\mathbf{alloc}_{\mathbb{R}^\sharp}$ ensures that the new cell has been freshly allocated thanks to the transform-into relation $[\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{f} \mapsto \delta)]$. It also ensures that the input abstract heap $\mathrm{r}^\sharp$ is not affected by the allocation, by separating it from the latter transform-into relation with the relational separating conjunction $*_{\mathtt{R}}$. This definition is correct because of the assumption that the program never reallocates the same memory cell.

Theorem 5.5 (Soundness of $\mathbf{alloc}_{\mathbb{R}^\sharp}$). *Let $\beta \in \mathbb{V}^\sharp, \mathtt{f} \in \mathbb{F}$ and $\mathrm{r}^\sharp \in \mathbb{R}^\sharp$. The function $\mathbf{alloc}_{\mathbb{R}^\sharp}$ is sound if:*

$$\forall (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \Longrightarrow \\ \exists \mathrm{v} \in \mathbb{V}, (\mathrm{h_i}, \mathrm{h_o} \circledast [\nu(\beta) + \mathtt{f} \mapsto \mathrm{v}], \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathbf{alloc}_{\mathbb{R}^\sharp}(\beta, \mathtt{f}, \mathrm{r}^\sharp))$$

Definition 5.6 (Allocation in abstract memory relations). *Let* $(\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)$ *be an abstract memory relation and* $n \geqslant 1$ *the number of cells to allocate. We assume that* $\mathbf{eval}_{\mathrm{L}}(\ell, \mathrm{e}^\sharp, \mathrm{r}_0^\sharp) = (\alpha, \mathtt{g})$ *and that* β *is a fresh symbolic value. The abstract heap relations* $\mathrm{r}_1^\sharp, \ldots, \mathrm{r}_n^\sharp$ *are defined as follow:*

$\forall i \; s.t. \; 1 \leqslant i \leqslant n$, $\mathrm{r}_i^\sharp = \mathbf{alloc}_{\mathbb{R}^\sharp}(\beta, \mathtt{f}_i, \mathrm{r}_{i-1}^\sharp)$.

Finally, if $\mathrm{n}_1^\sharp = \mathbf{guard}_{\mathbb{N}^\sharp}((\beta \neq \mathbf{0x0}), \mathrm{n}_0^\sharp)$ *and* $\mathrm{r}_{n+1}^\sharp = \mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{g}, \beta, \mathrm{r}_n^\sharp)$*, then:*

$$\mathbf{alloc}_{\mathbb{M}_{\mathcal{R}}^\sharp}(\ell,\ \{\mathtt{f}_1, \ldots, \mathtt{f}_n\},\ (\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)) = (\mathrm{e}^\sharp, \mathrm{r}_{n+1}^\sharp, \mathrm{n}_1^\sharp)$$

Like abstract assignments, the function $\mathbf{alloc}_{\mathbb{M}_{\mathcal{R}}^\sharp}$ evaluates the l-value ℓ with $\mathbf{eval}_{\mathrm{L}}$ into a pair $(\alpha, \mathtt{g})$ (this step may also unfold inductive predicates). It then generates a fresh symbolic value β, that is the base address of the block being created. For each field $\mathtt{f}_i \in \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$, it applies $\mathbf{assign}_{\mathbb{R}^\sharp}(\beta, \mathtt{f}_i, \mathrm{r}_{i-1}^\sharp)$, where $\mathrm{r}_{i-1}^\sharp$ has accumulated the allocations of the previous cells. Once the full block is allocated, it keeps track of the fact that the address of the new block is not null with $\mathbf{guard}_{\mathbb{N}^\sharp}((\beta \neq \mathbf{0x0}), \mathrm{n}_0^\sharp)$ and assigns this address to the given l-value with $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{g}, \beta, \mathrm{r}_n^\sharp)$.

Theorem 5.6 (Soundness of $\mathbf{alloc}_{\mathbb{M}_{\mathcal{R}}^\sharp}$). *Let* $\ell \in \mathcal{L}val, \mathtt{f}_1, \ldots, \mathtt{f}_n \in \mathbb{F}$ *and* $\mathrm{m}_{\mathcal{R}}^\sharp \in \mathbb{M}_{\mathcal{R}}^\sharp$*. If* $(\mathrm{m}_0, (\mathrm{e}, \mathrm{h})) \in \gamma_{\mathbb{M}_{\mathcal{R}}^\sharp}(\mathrm{m}_{\mathcal{R}}^\sharp)$ *then:*

$$\exists \mathrm{a}' \in \mathbb{A}, \mathrm{v}_1, \ldots, \mathrm{v}_n \in \mathbb{V},$$

$$(\mathrm{m}_0,\ \ (\mathrm{e},\ \mathrm{h}[\mathcal{L}[\![\ell]\!](\mathrm{e}, \mathrm{h}) \leftarrow \mathrm{a}'] \circledast [\mathrm{a}' + \mathtt{f}_1 \mapsto \mathrm{v}_1, \ldots, \mathrm{a}' + \mathtt{f}_n \mapsto \mathrm{v}_n]))$$
$$\in$$
$$\gamma_{\mathbb{M}_{\mathcal{R}}^\sharp}(\mathbf{alloc}_{\mathbb{M}_{\mathcal{R}}^\sharp}(\ell, \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}, \mathrm{m}_{\mathcal{R}}^\sharp)$$

Example 5.6 (Allocation of a list element). We consider the analysis of the following statement, from the abstract memory relation $(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$, with $\mathrm{e}^\sharp(\mathtt{p}) = \alpha_0$ and $\mathrm{r}^\sharp = \mathtt{Id}(\alpha_0 \mapsto \alpha_1)$:

$$\mathtt{p} = \mathbf{malloc}(\{\mathtt{next}; \mathtt{data}\})$$

The abstract heap relation computed in this case is:

$$[(\alpha_0 \mapsto \alpha_1) \dashrightarrow (\alpha_0 \mapsto \beta)] *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{next} \mapsto \beta_1)]$$
$$*_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{data} \mapsto \beta_2)]$$

Deallocations.

The function $\mathbf{free}_{\mathbb{M}_{\mathcal{R}}^\sharp}(\ell, (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp))$ returns an abstract memory relation that over-approximates the effect of the statement $\mathbf{free}(\ell)$. It represents the deletion of the memory block pointed

by ℓ. The resulting abstract memory relation should express the absence of cells that were present in its input state. Similarly to abstract allocation, the abstract deallocation of the memory block can be done cell per cell.

The function $\mathbf{free}_{\mathbb{R}^\sharp}$ inputs a symbolic value α, a field $\mathtt{f}$ and an abstract heap relation $\mathrm{r}^\sharp$. It returns an abstract heap relation where the points-to predicate whose address is $\alpha \cdot \mathtt{f}$ has been deleted in the output abstract heap of $\mathrm{r}^\sharp$.

Definition 5.7 (Deallocation for abstract heap relations). *We define the deallocation function for abstract heap relations* $\mathbf{free}_{\mathbb{R}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{R}^\sharp \to \mathbb{R}^\sharp$*:*

- $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp) = \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_2^\sharp$, *if* $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}_1^\sharp) = \mathrm{r}_2^\sharp$
- $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp *_{\mathtt{S}} (\alpha \cdot \mathtt{f} \mapsto \beta)]) = [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]$
- $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)) = \mathtt{Id}(\mathrm{h}^\sharp) *_{\mathtt{R}} [\alpha \cdot \mathtt{f} \mapsto \beta \dashrightarrow \mathbf{emp}]$

To perform the deallocation in an abstract heap relation $\mathrm{r}^\sharp$, the function $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}^\sharp)$ should express the absence of the memory cell at address $\alpha \cdot \mathtt{f}$ in the concrete output heaps of $\mathrm{r}^\sharp$. If $(\mathrm{h_i},\ \mathrm{h_o} \circledast [\nu(\alpha) + \mathtt{f} \mapsto \mathrm{v}],\ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp)$, then $(\mathrm{h_i},\ \mathrm{h_o},\ \nu)$ must be in $\gamma_{\mathbb{R}^\sharp}(\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}^\sharp))$. The definition of $\mathbf{free}_{\mathbb{R}^\sharp}$ follows the same principles as $\mathbf{assign}_{\mathbb{R}^\sharp}$ in Section 5.4:

- When $\mathrm{r}^\sharp = \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp$, if the cell to delete is in $\mathrm{r}_0^\sharp$, $\mathbf{free}_{\mathbb{R}^\sharp}$ is called recursively on $\mathrm{r}_0^\sharp$ (we apply the Frame rule [Rey02] but for abstract heap relations). Similarly to abstract assignment, if $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}_1^\sharp) = \mathrm{r}_2^\sharp$, then $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_1^\sharp) = \mathrm{r}_0^\sharp *_{\mathtt{R}} \mathrm{r}_2^\sharp$.
- When $\mathrm{r}^\sharp = [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow (\mathrm{h}_\mathrm{o}^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)]$, the function $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}^\sharp)$ should return the abstract heap relation $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]$.
- When $\mathrm{r}^\sharp = \mathtt{Id}(\mathrm{h}^\sharp *_{\mathtt{S}} \alpha \cdot \mathtt{f} \mapsto \beta)$, the abstract deallocation proceeds exactly like the abstract assignment. It first splits $\mathrm{r}^\sharp$ into $\mathtt{Id}(\mathrm{h}^\sharp) *_{\mathtt{R}} \mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \beta)$, then weakens the right hand identity relation into $[(\alpha \cdot \mathtt{f} \mapsto \beta) \dashrightarrow (\alpha \cdot \mathtt{f} \mapsto \beta)]$, and performs the deallocation in the obtained transform into relation. It finally produces $\mathtt{Id}(\mathrm{h}^\sharp) *_{\mathtt{R}} [\alpha \cdot \mathtt{f} \mapsto \beta \dashrightarrow \mathbf{emp}]$.

Theorem 5.7 (Soundness of $\mathbf{free}_{\mathbb{R}^\sharp}$). *Let* $\alpha \in \mathbb{V}^\sharp, \mathtt{f} \in \mathbb{F}$ *and* $\mathrm{r}^\sharp \in \mathbb{R}^\sharp$*. Then the function* $\mathbf{free}_{\mathbb{R}^\sharp}$ *is sound:*

$$\exists \mathrm{v} \in \mathbb{V},\ \forall (\mathrm{h_i}, \mathrm{h_o} \circledast [\nu(\alpha) + \mathtt{f} \mapsto \mathrm{v}], \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \Longrightarrow (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}^\sharp))$$

The function $\mathbf{free}_{\mathbb{M}^\sharp_\mathcal{R}}$ mainly builds upon $\mathbf{free}_{\mathbb{R}^\sharp}$. It also requires to evaluate the base address of the block to delete and to obtain the set of fields of the block. For simplicity,

we assume that those fields are provided by the function **get_fields**, that we do not define. It inputs an abstract value α (that corresponds to the base address of the block), an abstract heap relation $\mathrm{r}^\sharp$ and returns the set of fields attached to α in $\mathrm{r}^\sharp$.

Definition 5.8 (Deallocation in abstract memory relations). *Let* $(\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}^\sharp)$ *be an abstract memory relation and* $n \geqslant 1$ *the number of cells to deallocate. We assume that* $\mathbf{eval}_E(\ell, \mathrm{e}^\sharp, \mathrm{r}_0^\sharp) = (\alpha, \alpha)$ *and* $\{\mathtt{f}_1, \ldots, \mathtt{f}_n\} = \mathbf{get\_fields}(\alpha, \mathrm{r}_0^\sharp)$*. The abstract heap relations* $\mathrm{r}_1^\sharp, \ldots, \mathrm{r}_n^\sharp$ *are defined as follow:*

$\forall i, 1 \leqslant i \leqslant n, \mathrm{r}_i^\sharp = \mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}_i, \mathrm{r}_{i-1}^\sharp)$.

Finally we have:

$$\mathbf{free}_{\mathbb{M}_\mathcal{R}^\sharp}(\ell, (\mathrm{e}^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}^\sharp)) = (\mathrm{e}^\sharp, \mathrm{r}_n^\sharp, \mathrm{n}^\sharp)$$

The function $\mathbf{free}_{\mathbb{M}_\mathcal{R}^\sharp}$ evaluates the base address α of the block pointed by ℓ with $\mathbf{eval}_\mathtt{E}$. It obtains the fields of the cells to delete $\{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$ with $\mathbf{get\_fields}(\alpha, \mathrm{r}_0^\sharp)$. For each field $\mathtt{f}_i \in \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}$, it applies $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}_i, \mathrm{r}_{i-1}^\sharp)$ where $\mathrm{r}_{i-1}^\sharp$ has accumulated the deletion of the previous cells. Finally, $\mathrm{r}_n^\sharp$ describes the deallocation of the entire block. Note that before performing $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}_i, \mathrm{r}_{i-1}^\sharp)$, the analysis may perform unfolding on α in order to materialize all its fields, if an inductive predicate is attached to α. For simplicity, we do not explicit this step and assume that α is already unfolded.

Theorem 5.8 (Soundness of $\mathbf{free}_{\mathbb{M}_\mathcal{R}^\sharp}$). *Let* $\ell \in \mathcal{L}val$ *and* $\mathrm{m}_\mathcal{R}^\sharp \in \mathrm{m}_\mathcal{R}^\sharp$.

If $(\mathrm{m}_0, (\mathrm{e}, \mathrm{h}_1 \circledast \mathrm{h}_1')) \in \gamma_{\mathbb{M}_\mathcal{R}^\sharp}(\mathrm{m}_\mathcal{R}^\sharp)$ *such that:*

$$\mathbf{dom}(\mathrm{h}_1') = \{\mathcal{E}[\![\ell]\!](\mathrm{e}, \mathrm{h}_1) + \mathtt{f} \mid \mathtt{f} \in \mathbb{F}\}$$

then:

$$(\mathrm{m}_0, (\mathrm{e}, \mathrm{h}_1)) \in \gamma_{\mathbb{M}_\mathcal{R}^\sharp}(\mathbf{free}_{\mathbb{M}_\mathcal{R}^\sharp}(\ell, \mathrm{m}_\mathcal{R}^\sharp))$$

Example 5.7 (Deallocation of a list element). We show the effect of the analysis of the deallocation of a list element, from the abstract memory relation $(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$, with $\mathrm{e}^\sharp(\mathtt{p}) = \alpha_0$ and

$$\mathrm{r}^\sharp = \mathtt{Id}(\alpha_0 \mapsto \alpha_1 *_\mathtt{S} \alpha_1 \cdot \mathtt{next} \mapsto \alpha_2) *_\mathtt{R} [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow (\mathrm{h}_\mathrm{o}^\sharp *_\mathtt{S} \alpha_1 \cdot \mathtt{data} \mapsto \alpha_3)]$$

Thus, the abstract heap relation computed for the instruction $\mathbf{free}(\mathtt{p})$ is:

$$\mathtt{Id}(\alpha_0 \mapsto \alpha_1) *_\mathtt{R} [(\alpha_1 \cdot \mathtt{next} \mapsto \alpha_2) \dashrightarrow \mathbf{emp}] *_\mathtt{R} [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]$$

Local Variables Initialization and Deletion.

The declaration or the initialization of a local variable to a program block is also considered as an allocation. This is justified by the fact that we do not distinguish the heap and the stack, and that a new program variable does not belong to the initial input memory state. Similarly, when we exit a program block, we deallocate all the addresses of local variables to this block.

5.6 Condition Test

In this section, we define the transfer function $\mathbf{guard}_{\mathbb{M}^\sharp_\mathcal{R}} \in \mathcal{E}\mathit{xpr} \times \mathbb{M}^\sharp_\mathcal{R} \to \mathbb{M}^\sharp_\mathcal{R}$ for condition tests. It evaluates the boolean expression of a condition test and returns an abstract memory relation that takes into account the effects of the expression. It first translates the expression into a pure formula with $\mathbf{eval}_\mathtt{E}$ (note that this step may perform materialization). Finally, it updates the abstract numerical value interpreting the pure formula with the function $\mathbf{guard}_{\mathbb{N}^\sharp}$, introduced in Section 5.3.

Definition 5.9 (Condition test in abstract memory relations). *Let* $(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp_0)$ *be an abstract memory relation and* e *an expression.*

If $\mathbf{eval}_E(e, \mathrm{e}^\sharp, \mathrm{r}^\sharp) = (\alpha, \mathrm{p}^\sharp)$ *and* $\mathrm{n}^\sharp_1 = \mathbf{guard}_{\mathbb{N}^\sharp}(\mathrm{p}^\sharp, \mathrm{n}^\sharp_0)$*, then:*

$$\mathbf{guard}_{\mathbb{M}^\sharp_\mathcal{R}}(e, (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp_0)) = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp_1)$$

Theorem 5.9 (Soundness of $\mathbf{guard}_{\mathbb{M}^\sharp_\mathcal{R}}$). *Let* $e \in \mathcal{E}\mathit{xpr}$ *and* $\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1} \in \mathbb{M}^\sharp_\mathcal{R}$.

If $\mathrm{m}^\sharp_{\mathcal{R}^1} = \mathbf{guard}_{\mathbb{M}^\sharp_\mathcal{R}}(e, \mathrm{m}^\sharp_{\mathcal{R}^0})$*, then:*

$$(\mathrm{m_i}, \mathrm{m_o}) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^0}) \wedge \mathcal{E}[\![e]\!](\mathrm{m_o}) \neq 0 \implies (\mathrm{m_i}, \mathrm{m_o}) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^1})$$

Example 5.8 (Condition test). Consider the following condition test:

$$\mathbf{if}(\mathtt{l \texttt{->} next} == \mathbf{0x0})$$

applied to the abstract memory relation $\mathrm{m}^\sharp_\mathcal{R} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$ where

$$\mathrm{r}^\sharp = \mathtt{Id}(\alpha_0 \mapsto \alpha_1 *_\mathtt{s} \mathbf{list}(\alpha_1)) \text{ and } \mathrm{e}^\sharp(\mathtt{l}) = \alpha_0$$

The analysis first unfolds the inductive predicate $\mathbf{list}(\alpha_1)$ into a points-to predicate

$$\frac{}{\mathbf{emp} \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{emp}}\ (\sqsubseteq_{\mathbf{emp}}) \qquad \frac{\Psi(\alpha_1) = \alpha_0 \qquad \mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathrm{h}_1^\sharp \qquad \underline{\Psi(\beta_1) = \beta_0}}{\alpha_0 \cdot \mathtt{f} \mapsto \beta_0 *_{\mathtt{s}} \mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \alpha_1 \cdot \mathtt{f} \mapsto \beta_1 *_{\mathtt{s}} \mathrm{h}_1^\sharp}\ (\sqsubseteq_{\mathrm{pt}})$$

$$\frac{\Psi(\alpha_1) = \alpha_0 \qquad \mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathrm{h}_1^\sharp}{\mathbf{list}(\alpha_0) *_{\mathtt{s}} \mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{list}(\alpha_1) *_{\mathtt{s}} \mathrm{h}_1^\sharp}\ (\sqsubseteq_{\mathrm{ind}})$$

$$\frac{\Psi(\alpha_1) = \alpha_0 \quad \mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{list}(\beta_1) *_{\mathtt{s}} \mathrm{h}_1^\sharp \quad \underline{\Psi(\beta_1) = \beta_0} \quad (\beta_1 \text{ fresh})}{\mathbf{listseg}(\alpha_0, \beta_0) *_{\mathtt{s}} \mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{list}(\alpha_1) *_{\mathtt{s}} \mathrm{h}_1^\sharp}\ (\sqsubseteq_{\mathrm{indseg}})$$

$$\frac{\Psi(\alpha_1) = \alpha_0 \quad \mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{listseg}(\alpha_1', \beta_1) *_{\mathtt{s}} \mathrm{h}_1^\sharp \quad \underline{\Psi(\alpha_1') = \alpha_0'} \quad (\alpha_1' \text{ fresh})}{\mathbf{listseg}(\alpha_0, \alpha_0') *_{\mathtt{s}} \mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{listseg}(\alpha_1, \beta_1) *_{\mathtt{s}} \mathrm{h}_1^\sharp}\ (\sqsubseteq_{\mathrm{seg}})$$

Figure 5.2: Inclusion checking rules for abstract heaps, directly derived from [CR08]

and obtains the following abstract heap relation:

$$\mathtt{Id}(\alpha_0 \mapsto \alpha_1 *_{\mathtt{s}} \alpha_1 \cdot \mathtt{data} \mapsto \delta *_{\mathtt{s}} \alpha_1 \cdot \mathtt{next} \mapsto \alpha_2 *_{\mathtt{s}} \mathbf{list}(\alpha_2))$$

It then stores the constraint $\alpha_2 = \mathbf{0x0}$ in the abstract numerical value $\mathrm{n}^\sharp$.

5.7 Inclusion

Like classical shape analyses [DOY06, CR08], our analysis needs to *fold* inductive predicates so as to (conservatively) decide inclusion and join abstract states. We first present the inclusion checking algorithm in this section.

Inclusion checking is used to verify logical entailment, to check the convergence of loop iterates, and to support the join / widening algorithm. It consists of a conservative function $\mathbf{isle}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ that inputs two abstract memory relations $\mathrm{m}^\sharp_{\mathcal{R}^0} = (\mathrm{e}_0^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)$ and $\mathrm{m}^\sharp_{\mathcal{R}^1} = (\mathrm{e}_1^\sharp, \mathrm{r}_1^\sharp, \mathrm{n}_1^\sharp)$, that either returns **true** (meaning that the inclusion of concretizations holds) or **false** (meaning that the analysis cannot decide whether inclusion holds).

An important feature of inclusion checking is that the underlying abstract heaps may have *distinct* sets of symbolic values. Yet, to compare abstract heaps, the algorithm requires to compare symbolic values. That is, the inclusion checking algorithm requires a *renaming function* $\Psi \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ that maps symbolic values of $\mathrm{m}^\sharp_{\mathcal{R}^1}$ into symbolic values of $\mathrm{m}^\sharp_{\mathcal{R}^0}$ in order to maintain a notion of equivalence between symbolic values.

The definition of inclusion checking consists in three steps: the *initialization of the renaming function* that creates the initial renaming function of inclusion checking from the abstract environments of the two abstract memory relations, the *inclusion checking in abstract heap relations* that performs inclusion checking at abstract heap relations level,

$$\frac{\alpha \text{ carries an inductive predicate in } \mathrm{r}_1^\sharp \quad (\mathrm{r}_\mathrm{u}^\sharp, \mathrm{p}^\sharp) \in \mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, \mathrm{r}_1^\sharp) \quad \mathrm{r}_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathrm{r}_\mathrm{u}^\sharp}{\mathrm{r}_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp} \ (\sqsubseteq_{\text{unfold}})$$

$$\frac{\mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathrm{h}_1^\sharp}{\mathtt{Id}(\mathrm{h}_0^\sharp) \sqsubseteq_{\mathbb{R}^\sharp} \mathtt{Id}(\mathrm{h}_1^\sharp)} \ (\sqsubseteq_{\mathtt{Id}}) \qquad \frac{\mathrm{h}_{\mathrm{i},0}^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathrm{h}_{\mathrm{i},1}^\sharp \qquad \mathrm{h}_{\mathrm{o},0}^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathrm{h}_{\mathrm{o},1}^\sharp}{[\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp] \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]} \ (\sqsubseteq_{\dashrightarrow})$$

$$\frac{\mathrm{r}_{0,0}^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathrm{r}_{1,0}^\sharp \quad \mathrm{r}_{0,1}^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathrm{r}_{1,1}^\sharp}{\mathrm{r}_{0,0}^\sharp *_\mathtt{R} \mathrm{r}_{0,1}^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathrm{r}_{1,0}^\sharp *_\mathtt{R} \mathrm{r}_{1,1}^\sharp} \ (\sqsubseteq_{*_\mathtt{R}})$$

$$\frac{\mathtt{Id}(\mathrm{h}_0^\sharp) *_\mathtt{R} \mathtt{Id}(\mathrm{h}_1^\sharp) *_\mathtt{R} \mathrm{r}^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp}{\mathtt{Id}(\mathrm{h}_0^\sharp *_\mathtt{S} \mathrm{h}_1^\sharp) *_\mathtt{R} \mathrm{r}^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp} \ (\sqsubseteq_{\mathtt{Id}-\mathrm{split}_\mathrm{L}}) \qquad \frac{\mathrm{r}_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathtt{Id}(\mathrm{h}_0^\sharp) *_\mathtt{R} \mathtt{Id}(\mathrm{h}_1^\sharp) *_\mathtt{R} \mathrm{r}^\sharp}{\mathrm{r}_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathtt{Id}(\mathrm{h}_0^\sharp *_\mathtt{S} \mathrm{h}_1^\sharp) *_\mathtt{R} \mathrm{r}^\sharp} \ (\sqsubseteq_{\mathtt{Id}-\mathrm{split}_\mathrm{R}})$$

$$\frac{\mathrm{r}^\sharp *_\mathtt{R} [\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp] \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]}{\mathrm{r}^\sharp *_\mathtt{R} \mathtt{Id}(\mathrm{h}^\sharp) \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]} \ (\sqsubseteq_{\mathtt{Id}-\mathrm{weak}})$$

$$\frac{\mathrm{r}^\sharp *_\mathtt{R} [\mathrm{h}_{\mathrm{i},0}^\sharp *_\mathtt{S} \mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp *_\mathtt{S} \mathrm{h}_{\mathrm{o},1}^\sharp] \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]}{\mathrm{r}^\sharp *_\mathtt{R} [\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp] *_\mathtt{R} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp] \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]} \ (\sqsubseteq_{\dashrightarrow-\mathrm{weak}})$$

Figure 5.3: Inclusion checking rules for abstract heap relations

and the *inclusion checking in the numerical abstract domain* that performs inclusion checking at abstract numerical domains level.

Initialization of the renaming function. First, the abstract environment domain generates an initial renaming function Ψ_{init}. It is clear that each program variable must be mapped to the same address, thus the initial renaming function is defined as follows: $\forall \mathtt{x} \in \mathbb{X}, \Psi_{\text{init}}(\mathrm{e}_1^\sharp(\mathtt{x})) = \mathrm{e}_0^\sharp(\mathtt{x})$.

Inclusion checking in abstract heap relations. Then, the inclusion checking algorithm proceeds to the inclusion checking of two abstract heap relations. It consists of a function $\mathbf{isle}_{\mathbb{R}^\sharp}(\Psi, \mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp)$ over the abstract heap relations $\mathrm{r}_0^\sharp$ and $\mathrm{r}_1^\sharp$ where Ψ is an initial renaming function. The inclusion holds if it returns $(\Psi', \mathbf{true})$ where Ψ' is the final renaming function. It requires a function $\mathbf{isle}_{\mathbb{H}^\sharp}(\Psi, \mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp)$ that returns $(\Psi', \mathbf{true})$ if the inclusion of abstract heaps $\mathrm{h}_0^\sharp$ and $\mathrm{h}_1^\sharp$ holds and extends Ψ into Ψ'.

The definition of $\mathbf{isle}_{\mathbb{H}^\sharp}$ relies on a conservative algorithm, that implements a proof search, based on the rules shown in Figure 5.2. This system rule is directly derived from [CR08, Figure 6] (for simplicity we only consider the list inductive predicate). In this system rule, we assume that the "final" renaming function Ψ is given. However,

the underlying constraints such that $\Psi(\beta_1) = \beta_0$ indicates informally how Ψ is extended. This system rule is based on the operator $\sqsubseteq_{\mathbb{H}^\sharp}$, that satisfies the following property:

Theorem 5.10 (Soundness of $\sqsubseteq_{\mathbb{H}^\sharp}$). *Let* $\mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp \in \mathbb{H}^\sharp$. *Then:*

$$\mathrm{h}_0^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathrm{h}_1^\sharp \implies \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp) \subseteq \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp)$$

The implementation of $\mathbf{isle}_{\mathbb{H}^\sharp}$ detects which rule to apply thanks to the renaming function Ψ. When two symbolic values α_0 and α_1 match (i.e. when $\Psi(\alpha_1) = \alpha_0$), $\mathbf{isle}_{\mathbb{H}^\sharp}$ tries to apply the corresponding rule. If the rule cannot be applied, $\mathbf{isle}_{\mathbb{H}^\sharp}$ returns **false**, otherwise it continues. We do not detail the rules of this system, as our contributions are only related to abstract heap relations.

The definition of $\mathbf{isle}_{\mathbb{R}^\sharp}$ also relies on a conservative algorithm, that implements a proof search, based on the rules shown in Figure 5.3 (for clarity, we omit the pure formulas inclusion checking). This system rule is based on two operators, $\sqsubseteq_{\mathbb{H}^\sharp}$ (from Figure 5.2), and $\sqsubseteq_{\mathbb{R}^\sharp}$ that reasons over abstract heap relations. It satisfies the following properties:

Theorem 5.11 (Soundness of $\sqsubseteq_{\mathbb{R}^\sharp}$). *Let* $\mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp \in \mathbb{R}^\sharp$. *Then:*

$$\mathrm{r}_0^\sharp \sqsubseteq_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp \implies \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_0^\sharp) \subseteq \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp)$$

We observe that the renaming function Ψ does not appear in this system rule. However, it is also used in the definition of $\mathbf{isle}_{\mathbb{R}^\sharp}$. Indeed, if we have $\Psi(\alpha_1) = \alpha_0$, $\mathbf{isle}_{\mathbb{R}^\sharp}$ first detects which abstract heap relation is attached respectively to α_0 and α_1, and tries to apply accordingly a rule from Figure 5.3.

The rule $(\sqsubseteq_{\mathrm{unfold}})$ unfolds the right-hand side abstract heap relation and tries to match the left-hand side with one of the disjuncts. The rules $(\sqsubseteq_{\mathtt{Id}})$ and $(\sqsubseteq_{\dashrightarrow})$ are the canonical rules for abstract heap relations. They apply $\sqsubseteq_{\mathbb{H}^\sharp}$ on the abstract heaps contained in the abstract heap relations. The rule $(\sqsubseteq_{*_{\mathtt{R}}})$ makes inclusion checking as local. Finally, the rules $(\sqsubseteq_{\mathtt{Id}-\mathrm{split}_{\mathrm{L}}})$, $(\sqsubseteq_{\mathtt{Id}-\mathrm{split}_{\mathrm{R}}})$, $(\sqsubseteq_{\mathtt{Id}-\mathrm{weak}})$, and $(\sqsubseteq_{\dashrightarrow-\mathrm{weak}})$ allow to derive inclusion over abstract heap relations, and implement the properties observed in Theorem 4.1 (page 48). Thus, their correctness derives from these properties.

Inclusion checking in the numerical abstract domain. Finally, the analysis proceeds to the inclusion checking between the two abstract numerical values $\mathrm{n}_0^\sharp$ and $\mathrm{n}_1^\sharp$, modulo the renaming function Ψ' that results from $\mathbf{isle}_{\mathbb{R}^\sharp}$. Thus the numerical abstract domain should provide a function $\mathbf{isle}_{\mathbb{N}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp) \times \mathbb{N}^\sharp \times \mathbb{N}^\sharp \times \{\mathbf{true}, \mathbf{false}\}$.

Assumption 5.3 (Soundness of $\mathbf{isle}_{\mathbb{N}^\sharp}$). *Let* $\Psi \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)$ *and* $\mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp \in \mathbb{N}^\sharp$. *Then:*

$$\mathbf{isle}_{\mathbb{N}^\sharp}(\Psi, \mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp) = \mathbf{true} \quad \Rightarrow \quad \forall \nu \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_0^\sharp), \quad \Psi \circ \nu \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_1^\sharp)$$

Definition 5.10 (Inclusion checking for abstract memory relations). *Let* $\mathrm{m}^\sharp_{\mathcal{R}^0} = (\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0) \in \mathbb{M}^\sharp_{\mathcal{R}}$ *and* $\mathrm{m}^\sharp_{\mathcal{R}^1} = (\mathrm{e}^\sharp_1, \mathrm{r}^\sharp_1, \mathrm{n}^\sharp_1) \in \mathbb{M}^\sharp_{\mathcal{R}}$.

$$\forall \mathtt{x} \in \mathbb{X}, \Psi_{\text{init}}(\mathrm{e}^\sharp_1(\mathtt{x})) = \mathrm{e}^\sharp_0(\mathtt{x}).$$

$$\textit{If } \mathbf{isle}_{\mathbb{R}^\sharp}(\Psi_{\text{init}}, \mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1) = (\Psi', \mathbf{true}) \textit{ and } \mathbf{isle}_{\mathbb{N}^\sharp}(\Psi', \mathrm{n}^\sharp_0, \mathrm{n}^\sharp_1) = \mathbf{true}\textit{, then:}$$
$$\mathbf{isle}_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1}) = \mathbf{true}$$

That is, the function $\mathbf{isle}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ calls $\mathbf{isle}_{\mathbb{R}^\sharp}$ with the initial renaming function Ψ_{init}. If the inclusion holds for the two abstract heap relations $\mathrm{r}^\sharp_0$ and $\mathrm{r}^\sharp_1$, it tests the inclusion of the abstract numerical values $\mathrm{n}^\sharp_0$ and $\mathrm{n}^\sharp_1$ with the resulting renaming function of $\mathbf{isle}_{\mathbb{R}^\sharp}$.

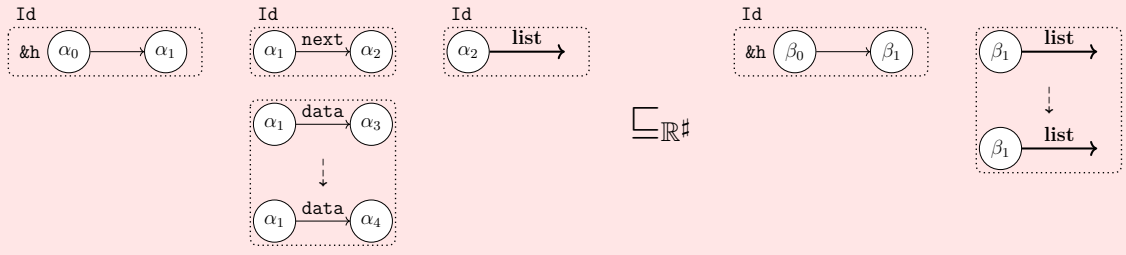
Theorem 5.12 (Soundness of inclusion checking). *If* $\mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1 \in \mathbb{H}^\sharp$, $\mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1 \in \mathbb{R}^\sharp$, $\Psi, \Psi' \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1} \in \mathbb{M}^\sharp_{\mathcal{R}}$ *then:*

$$\begin{aligned}
&\mathbf{isle}_{\mathbb{H}^\sharp}(\Psi, \mathrm{h}^\sharp_0, \mathrm{h}^\sharp_1) = (\Psi', \mathbf{true}) \\
&\quad \Longrightarrow \forall (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_0), (\mathrm{h}, \Psi' \circ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_1) \\
&\qquad \wedge \forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi(\alpha) = \beta \Rightarrow \Psi'(\alpha) = \beta
\end{aligned}$$

$$\begin{aligned}
&\mathbf{isle}_{\mathbb{R}^\sharp}(\Psi, \mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1) = (\Psi', \mathbf{true}) \\
&\quad \Longrightarrow \forall (\mathrm{h}_\mathrm{i}, \mathrm{h}_\mathrm{o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp_0), (\mathrm{h}_\mathrm{i}, \mathrm{h}_\mathrm{o}, \Psi' \circ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp_1) \\
&\qquad \wedge \forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi(\alpha) = \beta \Rightarrow \Psi'(\alpha) = \beta
\end{aligned}$$

$$\begin{aligned}
&\mathbf{isle}_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1}) = \mathbf{true} \\
&\quad \Longrightarrow \gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^0}) \subseteq \gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^1})
\end{aligned}$$

Example 5.9 (Inclusion checking). Let us discuss the computation of the inclusion checking of the two following abstract memory relations:



We distinguish only one variable: `h`, thus the initial renaming function Φ maps β_0 to α_0 ($\Psi(\beta_0) = \alpha_0$). Symbolic values α_0 and β_0 are both attached to a points-to

$$\frac{\Phi(\alpha) = (\alpha_0, \alpha_1) \quad \underline{\Phi(\beta) = (\beta_0, \beta_1)}}{\alpha_0 \cdot \mathtt{f} \mapsto \beta_0 \sqcup_{\mathbb{H}^\sharp} \alpha_1 \cdot \mathtt{f} \mapsto \beta_1 \rightsquigarrow \alpha \cdot \mathtt{f} \mapsto \beta} \ (\sqcup_{\text{pt}}) \qquad \frac{\Phi(\alpha) = (\alpha_0, \alpha_1)}{\mathbf{list}(\alpha_0) \sqcup_{\mathbb{H}^\sharp} \mathbf{list}(\alpha_1) \rightsquigarrow \mathbf{list}(\alpha)} \ (\sqcup_{\text{ind}})$$

$$\frac{\Phi(\alpha) = (\alpha_0, \alpha_1) \quad \mathrm{h}_1^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{list}(\alpha)}{\mathbf{list}(\alpha_0) \sqcup_{\mathbb{H}^\sharp} \mathrm{h}_1^\sharp \rightsquigarrow \mathbf{list}(\alpha)} \ (\sqcup_{\text{wind}})$$

$$\frac{\Phi(\alpha) = (\alpha_0, \alpha_1) \quad \Phi(\beta) = (\beta_0, \beta_1) \quad \mathrm{h}_1^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{listseg}(\alpha, \beta)}{\mathbf{listseg}(\alpha_0, \beta_0) \ \sqcup_{\mathbb{H}^\sharp} \ \mathrm{h}_1^\sharp \rightsquigarrow \mathbf{listseg}(\alpha, \beta)} \ (\sqcup_{\text{wseg}})$$

$$\frac{\Phi(\alpha) = (\alpha_0, \alpha_1) \quad \Phi(\beta) = (\alpha_0, \beta_1) \quad \mathrm{h}_1^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathbf{listseg}(\alpha, \beta)}{\mathbf{emp} \ \sqcup_{\mathbb{H}^\sharp} \ \mathrm{h}_1^\sharp \rightsquigarrow \mathbf{listseg}(\alpha, \beta)} \ (\sqcup_{\text{seg}-\text{intro}})$$

Figure 5.4: Join rewriting rules for abstract heaps, directly derived from [CR08]

predicate under the identity relation. Thus, rules ($\sqsubseteq_{\mathtt{Id}}$) and ($\sqsubseteq_{\text{pt}}$) are applied and Ψ is extended with $\Psi(\beta_1) = \alpha_1$. The inclusion checking now searches a rule α_1 and β_1. In the left-hand side, we observe that α_1 is attached to two abstract heap relations: the identity relation for its `next` field and a transform-into relation for its `data` field. Consequently, the inclusion checking applies rule ($\sqsubseteq_{\mathtt{Id}-\text{weak}}$), then rule ($\sqsubseteq_{\dashrightarrow-\text{weak}}$), in order to make α_1 is only attached to a single transform-into relation. In turns, the inclusion checking follows from rule ($\sqsubseteq_{\text{unfold}}$) to unfold β_1 in the right-hand side.

After applying many other rules, the inclusion checking should return **true**.

5.8 Join and Widening

5.8.1 Join

In the following, we define the abstract operator $\mathbf{join}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ that computes an over-approximation of the union of abstract memory relations. Like for the inclusion checking, $\mathbf{join}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ inputs two abstract memory relations $\mathrm{m}^\sharp_{\mathcal{R}^0} = (\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0)$ and $\mathrm{m}^\sharp_{\mathcal{R}^1} = (\mathrm{e}^\sharp_1, \mathrm{r}^\sharp_1, \mathrm{n}^\sharp_1)$, but instead of a boolean, outputs a new abstract memory relation $\mathrm{m}^\sharp_{\mathcal{R}} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$. It is defined such that the union of the concretizations of $\mathrm{m}^\sharp_{\mathcal{R}^0}$ and $\mathrm{m}^\sharp_{\mathcal{R}^1}$ is included in the concretization of $\mathrm{m}^\sharp_{\mathcal{R}}$.

The creation of a new abstract memory relation implies the creation of new symbolic values. Thus, the join operator needs to maintain a relation between the symbolic values of its two arguments and the resulting symbolic values. Slightly differently than the inclusion checking, the join requires a pair of renaming functions $\Phi = (\Psi_0, \Psi_1)$ that map each output symbolic value to the pair of the two corresponding input symbolic

$$\frac{h_0^\sharp \sqcup_{\mathbb{H}^\sharp} h_1^\sharp \rightsquigarrow h^\sharp}{\mathtt{Id}(h_0^\sharp) \ \sqcup_{\mathbb{R}^\sharp} \ \mathtt{Id}(h_1^\sharp) \rightsquigarrow \mathtt{Id}(h^\sharp)} \ (\sqcup_{\mathtt{Id}})$$

$$\frac{h_{i,0}^\sharp \sqcup_{\mathbb{H}^\sharp} h_{i,1}^\sharp \rightsquigarrow h_i^\sharp \qquad h_{o,0}^\sharp \sqcup_{\mathbb{H}^\sharp} h_{o,1}^\sharp \rightsquigarrow h_o^\sharp}{[h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp] \ \sqcup_{\mathbb{R}^\sharp} \ [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp] \rightsquigarrow [h_i^\sharp \dashrightarrow h_o^\sharp]} \ (\sqcup_{\dashrightarrow})$$

$$\frac{r_{0,0}^\sharp \ \sqcup_{\mathbb{R}^\sharp} \ r_{1,0}^\sharp \rightsquigarrow r_0^\sharp \qquad r_{0,1}^\sharp \ \sqcup_{\mathbb{R}^\sharp} \ r_{1,1}^\sharp \rightsquigarrow r_1^\sharp}{r_{0,0}^\sharp *_{\mathtt{R}} r_{0,1}^\sharp \ \sqcup_{\mathbb{R}^\sharp} \ r_{1,0}^\sharp *_{\mathtt{R}} r_{1,1}^\sharp \rightsquigarrow r_0^\sharp *_{\mathtt{R}} r_1^\sharp} \ (\sqcup_{*_{\mathtt{R}}})$$

$$\frac{[h_0^\sharp \dashrightarrow h_0^\sharp] \ \sqcup_{\mathbb{R}^\sharp} \ [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp] \rightsquigarrow r^\sharp}{\mathtt{Id}(h_0^\sharp) \ \sqcup_{\mathbb{R}^\sharp} \ [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp] \rightsquigarrow r^\sharp} \ (\sqcup_{\mathtt{Id}-\mathrm{weak}})$$

$$\frac{\mathtt{Id}(h_0^\sharp *_{\mathtt{S}} h_1^\sharp) *_{\mathtt{R}} r_0^\sharp \ \sqcup_{\mathbb{R}^\sharp} \ r_1^\sharp \rightsquigarrow r^\sharp}{\mathtt{Id}(h_0^\sharp) *_{\mathtt{R}} \mathtt{Id}(h_1^\sharp) *_{\mathtt{R}} r_0^\sharp \ \sqcup_{\mathbb{R}^\sharp} \ r_1^\sharp \rightsquigarrow r^\sharp} \ (\sqcup_{\mathtt{Id}-\mathrm{merge}})$$

$$\frac{[h_{i,0}^\sharp *_{\mathtt{S}} h_{i,1}^\sharp \dashrightarrow h_{o,0}^\sharp *_{\mathtt{S}} h_{o,1}^\sharp] *_{\mathtt{R}} r_0^\sharp \ \sqcup_{\mathbb{R}^\sharp} \ r_1^\sharp \rightsquigarrow r^\sharp}{[h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp] *_{\mathtt{R}} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp] *_{\mathtt{R}} r_0^\sharp \ \sqcup_{\mathbb{R}^\sharp} \ r_1^\sharp \rightsquigarrow r^\sharp} \ (\sqcup_{\dashrightarrow-\mathrm{weak}})$$

$$\frac{[h_0^\sharp \dashrightarrow h_0^\sharp] *_{\mathtt{R}} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp] *_{\mathtt{R}} r_0^\sharp \ \sqcup_{\mathbb{R}^\sharp} \ r_1^\sharp \rightsquigarrow r^\sharp}{\mathtt{Id}(h_0^\sharp) *_{\mathtt{R}} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp] *_{\mathtt{R}} r_0^\sharp \ \sqcup_{\mathbb{R}^\sharp} \ r_1^\sharp \rightsquigarrow r^\sharp} \ (\sqcup_{\dashrightarrow-\mathrm{intro}})$$

Figure 5.5: Join rewriting rules

values. For instance, if α is a resulting symbolic value of the join operator, we note $\Phi(\alpha) = (\alpha_0, \alpha_1)$ if $\Psi_0(\alpha) = \alpha_0$ and $\Psi_1(\alpha) = \alpha_1$. The join algorithm proceeds in the same way as the inclusion checking, following the three same main steps: *initialization* that creates the initial pair of renaming functions, *join of abstract heap relations* that joins two abstract heap relations and *join in the numerical abstract domain* that joins the two abstract numerical values.

Initialization. The join operation starts with the initialization of the pair of renaming functions and the generation of the resulting abstract environment $e^\sharp$ as follows: $\forall \mathtt{x} \in \mathbb{X}, \Phi_{\text{init}}(\alpha) = (e_0^\sharp(\mathtt{x}), e_1^\sharp(\mathtt{x}))$ and $e^\sharp(\mathtt{x}) = \alpha$.

Join of abstract heap relations. The join operation then proceeds to the computation of the new abstract heap relation $r^\sharp$. This is done by the functions $\mathbf{join}_{\mathbb{R}^\sharp}$ and $\mathbf{join}_{\mathbb{H}^\sharp}$ that operate respectively on abstract heap relations and abstract heaps. They also input and extend the pair of renaming functions.

The algorithm to compute these functions follows the same principle than inclusion

checking. The definition of $\mathbf{join}_{\mathbb{H}^\sharp}$ implements rewriting rules of Figure 5.4, based on the operator $\sqcup_{\mathbb{H}^\sharp}$, directly derived from [CR08, Figure 7]. It satisfies the following property:

Theorem 5.13 (Soundness of $\sqcup_{\mathbb{H}^\sharp}$). *Let* $\mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp \in \mathbb{H}^\sharp$. *Then:*

$$\mathrm{h}_0^\sharp \sqcup_{\mathbb{H}^\sharp} \mathrm{h}_1^\sharp \rightsquigarrow \mathrm{h}^\sharp \quad \Longrightarrow \quad \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp) \cup \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp) \subseteq \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)$$

In this system rule, we also assume that the "final" pair of renaming functions is given (the underlying constraints show informally how it is extended). Similarly to inclusion checking, the pair of renaming functions detects which rule to apply. For instance, if $\Phi(\alpha) = (\alpha_0, \alpha_1)$, $\mathbf{join}_{\mathbb{H}^\sharp}$ will search for a corresponding rule. We do not detail the rules of Figure 5.4, as the issue is orthogonal to the reasoning over abstract heap relations.

The definition of $\mathbf{join}_{\mathbb{R}^\sharp}$ implements the rewriting rules in Figure 5.5. This system rule is based on two operators: $\sqcup_{\mathbb{H}^\sharp}$ (from Figure 5.4) and $\sqcup_{\mathbb{R}^\sharp}$ that reasons over abstract heap relations. It satisfies the following property:

Theorem 5.14 (Soundness of $\sqcup_{\mathbb{R}^\sharp}$). *Let* $\mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp \in \mathbb{R}^\sharp$. *Then:*

$$\mathrm{r}_0^\sharp \sqcup_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp \rightsquigarrow \mathrm{r}^\sharp \quad \Longrightarrow \quad \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_0^\sharp) \cup \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp) \subseteq \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp)$$

The implementation of $\mathbf{join}_{\mathbb{R}^\sharp}$ uses Φ to detect in which abstract heap relations are attached to the two mapped symbolic values, in order to detect which rule to apply.

The rules $(\sqcup_{\mathtt{Id}})$ and $(\sqcup_{\dashrightarrow})$ are applied on two abstract heap relations that consist of the same relational connective (respectively $\mathtt{Id}(.)$ and $[. \dashrightarrow .]_.$). They simply apply $\sqcup_{\mathbb{H}^\sharp}$ on the abstract heaps they contain and conserve the relational connective. The rule $(\sqcup_{*_{\mathtt{R}}})$ is based on the separation principle and allows to apply the other rules independently. The next rules can all be applied symmetrically and follow the principles of Theorem 4.1 (page 48). When applied to an identity relation and a transform-into relation, the rule $(\sqcup_{\mathtt{Id}-\mathrm{weak}})$ first weakens the identity relation into a transform-into relation and applies recursively $\sqcup_{\mathbb{R}^\sharp}$. When the left operand of $\sqcup_{\mathbb{R}^\sharp}$ contains two identity relations, the rule $(\sqcup_{\mathtt{Id}-\mathrm{merge}})$ merges them. When the left operand contains two transform-into relations, the rule $(\sqcup_{\dashrightarrow-\mathrm{weak}})$ weakens them into one transform-into relation. Finally, when the left operand contains an identity relation and a transform-into relation, the rule $(\sqcup_{\dashrightarrow-\mathrm{intro}})$ weakens the identity relation into a transform-into relation and applies recursively $\sqcup_{\mathbb{R}^\sharp}$.

Join in the numerical abstract domain. Finally, the join operation proceeds to the join in the numerical abstract domain. Like for the inclusion checking, the numerical abstract values have to take into account the renaming performed by the abstract heap relations. Thus, the join in the numerical abstract domain is performed by the function $\mathbf{join}_{\mathbb{N}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{N}^\sharp \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$, which must satisfy the following assumption:

Assumption 5.4 (Soundness of $\mathbf{join}_{\mathbb{N}^\sharp}$). *Let* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp \in \mathbb{N}^\sharp$, *then:*

$$(\Psi_0 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_0^\sharp) \vee (\Psi_1 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_1^\sharp) \implies \nu \in \gamma_{\mathbb{N}^\sharp}(\mathbf{join}_{\mathbb{N}^\sharp}((\Psi_0, \Psi_1), \mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp))$$

Definition 5.11 (Join of abstract memory relations). *Let* $\mathrm{m}^\sharp_{\mathcal{R}^0} = (\mathrm{e}_0^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)$ *and* $\mathrm{m}^\sharp_{\mathcal{R}^1} = (\mathrm{e}_1^\sharp, \mathrm{r}_1^\sharp, \mathrm{n}_1^\sharp)$ *be two abstract memory relations.*

$$\forall \mathtt{x} \in \mathbb{X}, \Phi_{\text{init}}(\alpha) = (\mathrm{e}_0^\sharp(\mathtt{x}), \mathrm{e}_1^\sharp(\mathtt{x})) \text{ and } \mathrm{e}^\sharp(\mathtt{x}) = \alpha.$$

$$\textit{If } \mathbf{join}_{\mathbb{R}^\sharp}(\Phi_{\text{init}}, \mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp) = (\Phi', \mathrm{r}^\sharp) \textit{ and } \mathbf{join}_{\mathbb{N}^\sharp}(\Phi', \mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp) = \mathrm{n}^\sharp \textit{ then:}$$
$$\mathbf{join}_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1}) = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$$

Similarly as inclusion checking, $\mathbf{join}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ first initializes the pair of renaming functions Φ_{init} and creates the joined environment $\mathrm{e}^\sharp$. It then proceeds to the join of the abstract heap relations and to the join with Φ_{init}. Finally, it joins the abstract numerical values with the final pair of renaming functions returned by $\mathbf{join}_{\mathbb{R}^\sharp}$.

Theorem 5.15 (Soundness of $\mathbf{join}_{\mathbb{H}^\sharp}$). *Let* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp \in \mathbb{H}^\sharp$, *if* $\mathbf{join}_{\mathbb{H}^\sharp}((\Psi_0, \Psi_1), \mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp) = ((\Psi_0', \Psi_1'), \mathrm{h}^\sharp)$, *then:*

$$\begin{array}{lll}
\forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi_0(\alpha) = \beta & \implies & \Psi_0'(\alpha) = \beta \\
\forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi_1(\alpha) = \beta & \implies & \Psi_1'(\alpha) = \beta \\
(\mathrm{h}, \Psi_0' \circ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp) \vee (\mathrm{h}, \Psi_1' \circ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp) & \implies & (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)
\end{array}$$

Theorem 5.16 (Soundness of $\mathbf{join}_{\mathbb{R}^\sharp}$). *Let* $\mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp \in \mathbb{R}^\sharp$ *and* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$. *If* $\mathbf{join}_{\mathbb{R}^\sharp}((\Psi_0, \Psi_1), \mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp) = ((\Psi_0', \Psi_1'), \mathrm{r}^\sharp)$ *then:*

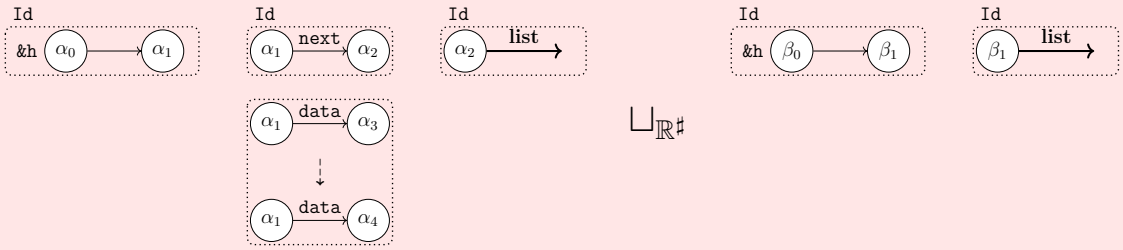
$$\begin{array}{lll}
\forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi_0(\alpha) = \beta & \implies & \Psi_0'(\alpha) = \beta \\
\forall \alpha, \beta \in \mathbb{V}^\sharp, \Psi_1(\alpha) = \beta & \implies & \Psi_1'(\alpha) = \beta \\
(\mathrm{h_i}, \mathrm{h_o}, \Psi_0' \circ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_0^\sharp) \vee (\mathrm{h_i}, \mathrm{h_o}, \Psi_1' \circ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp) & \implies & (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp)
\end{array}$$

Theorem 5.17 (Soundness of $\mathbf{join}_{\mathbb{M}^\sharp_{\mathcal{R}}}$). *Let* $\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1} \in \mathbb{M}^\sharp_{\mathcal{R}}$.

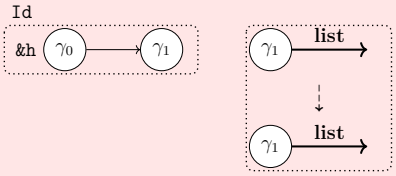
If $\mathbf{join}_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1}) = \mathrm{m}^\sharp_{\mathcal{R}}$ *then:*

$$\gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^0}) \cup \gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^1}) \subseteq \gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}})$$

Example 5.10 (Join with weakening). We discuss the computation of the join between these two abstract memory relations:



The initial pair of renaming functions is obtained by mapping the unique variable `h` of the two abstract memory relations (we have $\Phi(\gamma_0) = (\alpha_0, \beta_0)$). Then, the join operator applies rule ($\sqcup_{\texttt{Id}}$), that performs ($\sqcup_{\text{pt}}$) on α_0 and β_0. This extends Φ with $\Phi(\gamma_1) = (\alpha_1, \beta_1)$. We observe that the `data` field of α_1 is attached to a transform-into relation. Consequently, the join operator requires to apply rule ($\sqcup_{\dashrightarrow-\text{intro}}$), then rule ($\sqcup_{\dashrightarrow-\text{weak}}$) on the left-hand side to merge the predicates attached to α_1. Moreover, it requires to apply rule ($\sqcup_{\texttt{Id}-\text{weak}}$) on the right-hand side to weaken $\texttt{Id}(\textbf{list}(\beta_1))$ into $[\textbf{list}(\beta_1) \dashrightarrow \textbf{list}(\beta_1)]$. Finally, the join operator tries to apply rule ($\sqcup_{\text{wind}}$) between α_1 and β_0. However, a last weakening at abstract heap relation level is required. Indeed, the `next` field of α_1 points to α_2. Thus, the predicate $\texttt{Id}(\textbf{list}(\alpha_2))$ should also be merged with the abstract heap relation attached to α_1 (this is done by rule ($\sqcup_{\dashrightarrow-\text{intro}}$) then rule ($\sqcup_{\dashrightarrow-\text{weak}}$)). The resulting abstract memory relation is:



We observe that just because the `data` field of the left-hand side was modified, the join operator has lost the information that the other parts of the list was unmodified. We fix this loss of precision by extending abstract heap relations in Chapter 6.

5.8.2 Widening

During the analysis of loops and recursive programs, the number of iterations of the static analysis has to be *finite*. To always terminate in a finite number of steps, the analysis requires a widening operator $\textbf{wid}_{\mathbb{M}^\sharp_{\mathcal{R}}}$ that joins abstract memory relations and provides a convergence acceleration for the iteration process.

The widening operator $\textbf{wid}_{\mathbb{R}^\sharp}$ for abstract heap relations can be implemented using

the same rules system of Figure 5.5. Indeed, each rule strictly decreases the number of abstract heap relations, which ensures termination in a finite number of steps. Moreover, $\sqcup_{\mathbb{H}^\sharp}$ is already a widening operator, as it converges in a finite number of steps, as explained in [CR08].

However, $\mathbf{join}_{\mathbb{N}^\sharp}$ cannot be used as a widening operator. Indeed, some numerical abstract domains such as intervals [CC76] or convex polyhedra [CH78] require a specific widening operation. Thus, the abstract numerical domain should implement its own widening operator $\mathbf{wid}_{\mathbb{N}^\sharp}$.

Assumption 5.5 (Soundness of $\mathbf{wid}_{\mathbb{N}^\sharp}$). *Let* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp \in \mathbb{N}^\sharp$, *then:*

$$(\Psi_0 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_0^\sharp) \vee (\Psi_1 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_1^\sharp) \quad \Longrightarrow \quad \nu \in \gamma_{\mathbb{N}^\sharp}(\mathbf{wid}_{\mathbb{N}^\sharp}((\Psi_0, \Psi_1), \mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp))$$

The function $\mathbf{wid}_{\mathbb{N}^\sharp}$ *also enforces termination.*

The widening of abstract memory relation $\mathbf{wid}_{\mathbb{M}^\sharp_\mathcal{R}}$ can be defined like $\mathbf{join}_{\mathbb{M}^\sharp_\mathcal{R}}$ by substituting in its definition $\mathbf{join}_{\mathbb{N}^\sharp}$ by $\mathbf{wid}_{\mathbb{R}^\sharp}$ and by substituting $\mathbf{join}_{\mathbb{R}^\sharp}$ by $\mathbf{wid}_{\mathbb{N}^\sharp}$.

Definition 5.12 (Widening for abstract memory relations). *Let* $\mathrm{m}^\sharp_{\mathcal{R}^0} = (\mathrm{e}_0^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)$ *and* $\mathrm{m}^\sharp_{\mathcal{R}^1} = (\mathrm{e}_1^\sharp, \mathrm{r}_1^\sharp, \mathrm{n}_1^\sharp)$ *be two abstract memory relations.*

$$\forall \mathtt{x} \in \mathbb{X}, \Phi_{\text{init}}(\alpha) = (\mathrm{e}_0^\sharp(\mathtt{x}), \mathrm{e}_1^\sharp(\mathtt{x})) \text{ and } \mathrm{e}^\sharp(\mathtt{x}) = \alpha.$$

If $\mathbf{wid}_{\mathbb{R}^\sharp}(\Phi_{\text{init}}, \mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp) = (\Phi', \mathrm{r}^\sharp)$ *and* $\mathbf{wid}_{\mathbb{N}^\sharp}(\Phi', \mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp) = \mathrm{n}^\sharp$ *then:*

$$\mathbf{wid}_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1}) = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)$$

Theorem 5.18 (Soundness of $\mathbf{wid}_{\mathbb{M}^\sharp_\mathcal{R}}$). *Let* $\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1} \in \mathbb{M}^\sharp_\mathcal{R}$.

If $\mathbf{wid}_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1}) = \mathrm{m}^\sharp_\mathcal{R}$ *then:*

$$\gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^0}) \cup \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^1}) \subseteq \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_\mathcal{R})$$

The function $\mathbf{wid}_{\mathbb{M}^\sharp_\mathcal{R}}$ *also enforces termination*

5.9 Analysis Algorithm

Manipulating Disjunctions in the Analysis. Because of unfolding operations Section 5.3, the analysis must deal with disjunctions. This is exactly the role of the disjunctions of abstract memory relations defined in Section 4.3.4. They are built on top

abstract memory relations domain and allow to perform standard abstract operations (assignment, allocation, ...) over finite sets of abstract memory relations. Their interface is given below:

$$\begin{array}{rcll}
\mathbf{assign}_{\mathbb{R}^\vee} & \in & \mathcal{L}val \times \mathcal{E}xpr \times \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee \\
\mathbf{alloc}_{\mathbb{R}^\vee} & \in & \mathcal{L}val \times \mathcal{P}_{\text{fin}}(\mathbb{F}) \times \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee \\
\mathbf{free}_{\mathbb{R}^\vee} & \in & \mathcal{L}val \times \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee \\
\mathbf{guard}_{\mathbb{R}^\vee} & \in & \mathcal{E}xpr \times \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee \\
\mathbf{isle}_{\mathbb{R}^\vee} & \in & \mathbb{R}^\vee \times \mathbb{R}^\vee & \to \quad \{\mathbf{true}, \mathbf{false}\} \\
\mathbf{join}_{\mathbb{R}^\vee} & \in & \mathbb{R}^\vee \times \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee \\
\mathbf{wid}_{\mathbb{R}^\vee} & \in & \mathbb{R}^\vee \times \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee \\
\mathbf{partition}_{\mathbb{R}^\vee} & \in & \mathcal{P}_{\text{fin}}(\mathbb{R}^\vee) & \to \quad \mathbb{R}^\vee \\
\mathbf{collapse}_{\mathbb{R}^\vee} & \in & \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee
\end{array}$$

Each function of this interface applies the function of the same name, but at the abstract memory relation level. The functions $\mathbf{partition}_{\mathbb{R}^\vee}$ and $\mathbf{collapse}_{\mathbb{R}^\vee}$. respectively create and collapse disjunctions. The function $\mathbf{collapse}_{\mathbb{R}^\vee}$ may also be used to limit the number of disjunctions and to ensure the termination of the analysis. They should satisfy the following condition.

Theorem 5.19 (Soundness of $\mathbf{partition}_{\mathbb{R}^\vee}$ and $\mathbf{collapse}_{\mathbb{R}^\vee}$). *Let* $\mathrm{R}^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{M}^\sharp_{\mathcal{R}})$ *and* $\mathbb{r}^\vee \in \mathbb{R}^\vee$.

$$\begin{array}{rcl}
\bigcup\{\gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}}) \mid \mathrm{m}^\sharp_{\mathcal{R}} \in \mathrm{R}^\sharp\} & \subseteq & \gamma_{\mathbb{R}^\vee}(\mathbf{partition}_{\mathbb{R}^\vee}(\mathbb{r}^\vee)) \\
\gamma_{\mathbb{R}^\vee}(\mathbb{r}^\vee) & \subseteq & \mathbf{collapse}_{\mathbb{R}^\vee}(\mathbb{r}^\vee)
\end{array}$$

The soundness of the other functions of this interface and more details are discussed in [CR13].

Abstract Relational Semantics. The abstract semantics $[\![.]\!]^\sharp_{\mathcal{R}}$ relies on the abstract operations defined in Sections 5.4, 5.5 and 5.6, on the unfolding of Section 5.3 to analyze basic statements, and on the folding operations defined in Sections 5.7 and 5.8 to cope with control flow joins and loop invariants computation. It is defined by induction over the syntax of the programming language defined in Section 3.4 and operates over abstract disjunctions, as shown in Figure 5.6.

Soundness of $[\![.]\!]^\sharp_{\mathcal{R}}$ follows from the soundness of the basic operations.

Theorem 5.20 (Soundness). *The analysis is sound in the sense that, for all command* $\mathtt{c}$ *and for all disjunction of abstract memory relations* $\mathbb{r}^\vee$*:*

$$\begin{array}{l}
\forall (\mathrm{m}_0, \mathrm{m}_1) \in \gamma_{\mathbb{R}^\vee}(\mathbb{r}^\vee),\ \forall \mathrm{m}_2 \in \mathbb{M}, \\
\qquad (\mathrm{m}_1, \mathrm{m}_2) \in [\![\mathtt{c}]\!]_{\mathcal{R}} \quad \Longrightarrow \quad (\mathrm{m}_0, \mathrm{m}_2) \in \gamma_{\mathbb{R}^\vee}([\![\mathtt{c}]\!]^\sharp_{\mathcal{R}}(\mathbb{r}^\vee))
\end{array}$$

$$[\![\ell = e]\!]^\sharp_{\mathcal{R}}(\mathbb{r}^\vee) = \mathbf{assign}_{\mathbb{R}^\vee}(\ell, e, \mathbb{r}^\vee)$$

$$[\![\ell = \mathbf{malloc}(\{\mathtt{f}_1, \ldots, \mathtt{f}_n\})]\!]^\sharp_{\mathcal{R}}(\mathbb{r}^\vee) = \mathbf{alloc}_{\mathbb{R}^\vee}(\ell, \{\mathtt{f}_1, \ldots, \mathtt{f}_n\}, \mathbb{r}^\vee)$$

$$[\![\mathbf{free}(\ell)]\!]^\sharp_{\mathcal{R}}(\mathbb{r}^\vee) = \mathbf{free}_{\mathbb{R}^\vee}(\ell, \mathbb{r}^\vee)$$

$$[\![c_1; c_2]\!]^\sharp_{\mathcal{R}}(\mathbb{r}^\vee) = [\![c_2]\!]^\sharp_{\mathcal{R}}([\![c_1]\!]^\sharp_{\mathcal{R}}(\mathbb{r}^\vee))$$

$$[\![\mathbf{if}\,(e)\,c_1\,\mathbf{else}\,c_2]\!]^\sharp_{\mathcal{R}}(\mathbb{r}^\vee) = \mathbf{join}_{\mathbb{R}^\vee}([\![c_1]\!]^\sharp_{\mathcal{R}}(\mathbf{guard}_{\mathbb{R}^\vee}(e, \mathbb{r}^\vee)), [\![c_2]\!]^\sharp_{\mathcal{R}}(\mathbf{guard}_{\mathbb{R}^\vee}(\neg e, \mathbb{r}^\vee)))$$

$$\begin{aligned}[\![\mathbf{while}(e)\,c]\!]^\sharp_{\mathcal{R}}(\mathbb{r}_0^\vee) = \;& \text{If } \mathbf{isle}_{\mathbb{R}^\vee}(\mathbb{r}_1^\vee, \mathbb{r}_0^\vee) = \mathbf{true} \\ & \text{Then } \mathbf{guard}_{\mathbb{R}^\vee}(\neg e,\ \mathbf{join}_{\mathbb{R}^\vee}(\mathbb{r}_0^\vee, \mathbb{r}_1^\vee)) \\ & \text{Else } [\![\mathbf{while}(e)\,c]\!]^\sharp_{\mathcal{R}}(\mathbf{wid}_{\mathbb{R}^\vee}(\mathbf{join}_{\mathbb{R}^\vee}(\mathbb{r}_0^\vee, \mathbb{r}_1^\vee),\ \mathbb{r}_0^\vee)) \\ & \text{where } \mathbb{r}_1^\vee = [\![c]\!]^\sharp_{\mathcal{R}}(\mathbf{guard}_{\mathbb{R}^\vee}(e, \mathbb{r}_0^\vee))\end{aligned}$$

Figure 5.6: Abstract semantics for the programing language defined in Section 3.4, except for function calls and returns. The expression $\neg e$ is the negation of e.

5.10 Implementation and Experimental Evaluation

In this section, we report on the implementation of our analysis and try to evaluate:

1. whether it can infer precise and useful relational properties,
2. how it compares with a state shape analysis that does not compute relations.

Our implementation supports built-in inductive predicates to describe singly linked lists and binary trees. It provides both the analysis described in this paper, and a basic state shape analysis in the style of [CR08], and supporting the same inductive predicates. It was implemented as a Frama-C [KKP+15] plug-in consisting of roughly 18000 lines of OCaml.

We have ran both the state shape analysis and the relational shape analysis on series of small programs manipulating lists and trees listed in Table 5.1. This allows us to not only assess the results of the analysis computing abstract relations, but also to compare them with an analysis that infers abstract states.

The results obtained are listed in Table 5.1. The word 'State' means the result of the state analysis. The word 'Rel.' means the inferred abstract heap relation with the relational analysis. The analysis runtimes are averaged over 1000 runs of the analysis. The last column compares the logical strength of the inferred abstract states and abstract relations. This is indicated in the cells by comparison symbols ($<$, $>$ or $=$). For instance, if we find '$<$' in a cell of the State vs Rel. column, that means that relational analysis

Structure	Function	Time (in s)		Logical Strength
		State	Rel.	State vs Rel.
singly linked list	`allocation`	0.56	0.77	$<$
singly linked list	`deallocation`	0.46	0.80	$<$
singly linked list	`traversal`	0.58	0.79	$<$
singly linked list	`head_insertion`	0.43	0.43	$<$
singly linked list	`insert` (Figure 1.3)	1.11	1.92	$<$
singly linked list	`reverse`	0.60	1.01	$=$
singly linked list	`map`	0.59	0.92	$=$
singly linked list	`tail`	0.42	0.55	$<$
singly linked list	`nth`	0.70	1.17	$<$
singly linked list	`partition`	2.18	4.85	$=$
singly linked list	`append`	0.88	1.60	$<$
singly linked list	`contains`	0.82	1.22	$<$
singly linked list	`deep_copy`	1.15	2.08	$<$
singly linked list	`sort` (Figure 2.3)	4.09	21.95	$=$
singly linked list	`filter`	1.21	2.70	$=$
binary search tree	`allocation`	0.71	1.11	$<$
binary search tree	`search`	0.97	1.63	$<$
binary search tree	`insert`	2.25	6.10	$<$

Table 5.1: Experiment results. Time in seconds over 1000 runs on a laptop with Intel Core i5 running at 2.4 GHz, with 4 Gb RAM, for the state and relational analyses; the last column compares the logical strength of the inferred result of each analysis.

inferred a stronger property than the state analysis for the given function.

Logical Strength Comparison

We first discuss the logical strength. We observe the inferred abstract relations are never less strong than the inferred abstract states inferred by state analysis (there is no '>' in the State vs Rel. column). In most cases, the relational analyses has inferred stronger properties than the state analysis. We discuss some of these cases in the next paragraphs.

When the relational analysis inferred stronger properties. We consider the function `head_insertion` for which the relational analysis inferred the following abstract heap relation:

$$[\alpha_0 \mapsto \alpha_1 \dashrightarrow \alpha_0 \mapsto \beta] *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \beta \cdot \mathtt{next} \mapsto \alpha_1] \\ *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \beta \cdot \mathtt{data} \mapsto \delta] *_{\mathtt{R}} \mathtt{Id}(\mathbf{list}(\alpha_1))$$

It describes exactly the effects of the insertion of a new allocated element at the head of a given list. Indeed, it expresses explicitly that the input list (abstracted by the inductive predicate $\mathbf{list}(\alpha_1)$) has not been modified by the function. Moreover, it expresses the allocation of a new list element at address β whose `next` field points to the input list. This abstract relation is clearly more expressive than the result of the state analysis, that cannot capture the relational properties described above.

Another case where the relational analysis inferred stronger property is the function `deep_copy`. The inferred abstract heap relation for this function is:

$$\mathtt{Id}(\mathbf{list}(\alpha)) *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \mathbf{list}(\beta)]$$

It expresses that the input list $\mathbf{list}(\alpha)$ has not been modified and that a new list $\mathbf{list}(\beta)$ has been freshly allocated. These properties cannot be inferred by the state analysis, that simply expresses that the function inputs the list $\mathbf{list}(\alpha)$ and outputs two lists $\mathbf{list}(\alpha)$ and $\mathbf{list}(\beta)$.

When the relational analysis did not infer stronger properties. We now discuss some cases where the abstract heap relation inferred by the relational analysis is not stronger than the result of the state analysis. For example, the inferred abstract heap relation for the function `map`, that traverses a list and increments each `data` field, is:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\alpha)]$$

It just indicates that the input and output lists start at the same address. We have no more information compared to the state analysis.

Moreover, for the functions `reverse`, `sort` and `filter`, the relational analysis inferred the same abstract heap relation for these three functions:

$$[\mathbf{listseg}(\alpha_0, \alpha_1) \dashrightarrow \mathbf{list}(\alpha_2)]$$

The functions `reverse` and `sort` respectively reverse and sort a list in place. On the other hand, the function `filter` deallocates all the negative elements of the input list. The inferred abstract heap relation does not express any interesting properties compared to the result of the state analysis (only the transformation of a list into another). In Chapter 6, we propose an extension of abstract heap relations to improve their logical strength.

Runtime Comparison

We now compare the runtime of the relational analysis and of the state analysis. We observe in most cases that the relational analysis is slower than the state analysis, although the slow down factor is reasonable. Indeed, the time of relational analysis rarely exceeds the double of the state analysis (this is the case for the functions `partition`, `filter` and tree `insert`). An exception is the list `sort`, which is approximately 5 times slower. This is explained by the fact that this function contains a condition test in a nested loop and another condition test in the main loop. This implies to perform an important number of abstract joins and widenings. Conversely, the function `head_insertion`, that does not perform either abstract join or widening, incurs no slowdown. Moreover, we believe that we can optimize the implementation of these operators in our prototype analyzer, using a better strategy to detect the rules to apply.

While these test cases are not large, these results show that the relational analyses have a reasonable overhead and that they bring additional information compared to a classical state analysis. The difference time between the analyses is due to the fact that the relational analysis manipulates pairs of states whereas the state analysis manipulates only one state. In general, the relational analysis infers stronger properties.

5.11 Related Works

Our analysis computes an abstraction of the relational semantics of programs so as to capture the effect of programs using an element of some specifically designed abstract domain.

This technique has been applied to other abstractions in the past, and often applied to design *modular* static analyses [CC02], where program components can be analyzed once and separately. For numerical domains, it simply requires to duplicate each variable into two instances respectively describing the old and the new value, and to use a relational domain to the inputs and outputs. For instance, [PC06] implements this idea using

convex polyhedra and so as to infer abstract state relations for numerical programs. It has also been applied to shape analyses based on Three Valued Logic [SRW02] in [JLRS04, JLRS10]. This work is probably the closest to ours, but it relies on a very different abstraction using TVLA whereas we use a set of relational predicates based on separation logic. It uses the same variable duplication technique as mentioned above. Our analysis also has a notion of overlaid old / new predicates, but these are described heap regions, inside separation logic formulas.

Several other techniques have been used to specify memory properties. For instance, [YRSW03] uses temporal logic to specify temporal properties of heap evolutions and [TJ07] checks structural properties of codes using a specification language.

In the context of concurrent programs, [ARR$^+$07] verify linearizability of concurrent objects (unbounded linked list) maintaining isomorphism between two instances of memory layout. Also, [DKR04] uses an extension of temporal linear temporal logic and a tableau-based model-checking algorithm to specify the dynamic evolution of pointer structures. This latter technique has been applied in [DKR05] to prove the correctness of concurrent programs manipulating linked lists.

In the context of functional languages, [KJ14] allows to write down relations between function inputs and outputs, and relies on a solver to verify that constraints hold and [ZPJ16] computes shape specification by learning.

Regarding to shape analyses based on separation logic, [BDE$^+$10] infers combined list-data relations and has been extended in [BDES11] for inter-procedural analysis. They can infer precise relations between the data contained in a list, such as the sum of all data in a list is inferior to the length of this list. They also use a multi-set to represent the data of a list. For example, to prove a sort function, they compare the multi-set of the input list with the multi-set of the output list. If these multi-sets are the same (this means a permutation), and if the output list is sorted, then the sort function is proven. However, they do not have this notion of physical equality between the different memory cells that our relational domain can express. Consequently, they cannot express the *physical identity* relation of a program. Moreover, our relational properties do not focus on a specific data structure, but aim at being generic for any data structure.

Modular analyses that compute invariants by separate analysis of program components [CRL99, DDAS11, CNR$^+$15] use various sorts of abstractions for the behavior of program components. A common pattern uses tables of pairs made of an abstract pre-condition and a corresponding abstract post-condition, effectively defining a sort of cardinal power abstraction [CC79]. This technique has been used in several shape analyses based on separation logic [CDOY09, GCRN09, LGQC14, CDOY07]. We believe this tabular approach could benefit from abstractions of relations such as ours to infer stronger properties, and more concise summaries.

Chapter 6

Abstract Heap Transformation Predicates

In this chapter, we propose to extend abstract heap relations in order to capture stronger relational properties. This extension is generic: abstract heap relations are parameterized by abstract domains that describe specific relations. We propose three examples of such abstract domains, that are all data structure agnostic. We also integrate this extension in our relational intra-procedural analysis.

6.1 Motivations

Until now, we have two relational connectives that describe heap transformations: the identity relation and the transform-into relation. While the identity relation is very strong (it ensures that the heap is left unmodified), the transform-into relation is quite weak. Indeed, $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]$ just indicates that the input heap abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ has been transformed into the output heap abstracted by $\mathrm{h}_\mathrm{o}^\sharp$, but without describing specifically how the transformation occurred. For instance in Figure Figure 2.3 (page 23), the function `sort` performs a list sort in place, modifying only the order of its elements. A reasonable abstraction of the effects of this function is that the output list is a *permutation in place* of the input list.

Remark 3 (Properties to ensure)**.** *We remark that a permutation in place of a list can be expressed ensuring these two properties:*

(a) the permutation is in-place, so that the output list is obtained by manipulating directly the input list (the footprint of the two lists is the same).

(b) the data in the input and output lists are exactly the same (but may appear in a different order).

If we look at the abstract heap relation computed for the function `sort`:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]$$

We observe that this abstraction is too weak to ensure the points (a) and (b), as it describes only a transformation of a well formed linked list into a well formed linked list. More generally, to capture stronger relations, abstract heap relations should be extended.

6.2 Abstraction

To fix this imprecision, we could enrich abstract heap relations with a new connective that would express specifically these two points. The problem with this approach is that it would be certainly useless to describe the behavior of other functions using other data structures. Moreover, creating a relational connective in abstract heap relations to express a specific property when needed is too costly. Indeed, this requires to update all the functions related to abstract heap relations for each new relational connective.

An elegant and efficient approach to describe any binary relational properties between memory heaps without adding new connectives is to annotate transform-into relations by *abstract heap transformation predicates*. We henceforth write $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}$ for the transform-into relation annotated by the abstract heap transformation predicate $\mathrm{t}^\sharp$. It describes the transformation abstracted by $\mathrm{t}^\sharp$ of the heaps abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ into the heaps abstracted by $\mathrm{h}_\mathrm{o}^\sharp$.

We name an *abstract heap transformation predicates domain* a set $\mathbb{T}^\sharp$ of abstract heap transformation predicates. To be as generic as possible, we do not fix a specific abstract heap transformation predicates domain $\mathbb{T}^\sharp$. Instead, we parametrize the analysis with an interface of such $\mathbb{T}^\sharp$.

The concretization function $\gamma_{\mathbb{T}^\sharp}$ of an abstract heap transformation predicates domain $\mathbb{T}^\sharp$ should have the signature:

Condition 6.1 (Concretization). *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domains. Its concretization should have the following signature:*

$$\gamma_{\mathbb{T}^\sharp} : \mathbb{T}^\sharp \to \mathcal{P}(\mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$$

It simply maps an abstract heap transformation predicate into a set of triples made of an input concrete heap, an output concrete heap and a valuation function.

We can now update the concretization function $\gamma_{\mathbb{R}^\sharp}$ defined in Section 4.2 (page 46) of abstract heap relations to take into account transform-into relations annotated by the abstract heap transformation predicates.

Definition 6.2 (Concretization of annotated transform-into relations). *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain and $\mathrm{t}^\sharp \in \mathbb{T}^\sharp$, $\mathrm{h}_\mathrm{i}^\sharp, \mathrm{h}_\mathrm{o}^\sharp \in \mathbb{H}^\sharp$. Then, the concretization of transform-into relations annotated by abstract heap transformation predicates is defined as follow:*

$$\gamma_{\mathbb{R}^\sharp}([\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}) = \{(\mathrm{h}_\mathrm{i}, \mathrm{h}_\mathrm{o}, \nu) \mid (\mathrm{h}_\mathrm{i}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_\mathrm{i}^\sharp) \wedge (\mathrm{h}_\mathrm{o}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_\mathrm{o}^\sharp) \wedge (\mathrm{h}_\mathrm{i}, \mathrm{h}_\mathrm{o}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)\}$$

Like abstract heap relations, abstract heap transformation predicates can express identity (resp. independent transformations).

To that end, each abstract heap transformation predicates domain $\mathbb{T}^\sharp$ should define the identity $\mathbf{id}_{\mathbb{T}^\sharp} \in \mathbb{H}^\sharp \to \mathbb{T}^\sharp$ (resp. the separating conjunction operator $*_\mathtt{T} \in \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$), for abstract heap transformation predicates. These operators should satisfy the following assumptions:

Assumption 6.1 (Soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$). *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain, $\mathrm{t}^\sharp \in \mathbb{T}^\sharp$ and $\mathrm{h}^\sharp \in \mathbb{H}^\sharp$. Then $\mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp)$ is sound if:*

$$\{(\mathrm{h}, \mathrm{h}, \nu) \mid (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\} \subseteq \gamma_{\mathbb{T}^\sharp}(\mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp))$$

Assumption 6.2 (Soundness of $*_\mathtt{T}$). *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain and $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp \in \mathbb{T}^\sharp$. Then $\mathrm{t}_0^\sharp *_\mathtt{T} \mathrm{t}_1^\sharp$ is sound if:*

$$\gamma_{\mathbb{T}^\sharp}(\mathrm{t}_0^\sharp *_\mathtt{T} \mathrm{t}_1^\sharp) \supseteq \{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid (\mathrm{h}_{\mathrm{i},0}, \mathrm{h}_{\mathrm{o},0}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_0^\sharp) \wedge (\mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_1^\sharp)\}$$

Using these two operators, we can define new properties, similar to Theorem 4.1 (page 48), on abstract heap relations taking into account abstract heap transformation predicates.

Theorem 6.1 (Properties on abstract heap relations with abstract heap transformation predicates). *Let $\mathrm{h}^\sharp$, $\mathrm{h}_{\mathrm{i},0}^\sharp$, $\mathrm{h}_{\mathrm{i},1}^\sharp$, $\mathrm{h}_{\mathrm{o},0}^\sharp$, $\mathrm{h}_{\mathrm{o},1}^\sharp$ be abstract heaps, $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain and $\mathrm{t}^\sharp, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp \in \mathbb{T}^\sharp$. Then, we have the following properties:*

1. $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}^\sharp})$ *with* $\mathrm{t}^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp)$
2. $\gamma_{\mathbb{R}^\sharp}([\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp]_{\mathrm{t}_0^\sharp} *_\mathtt{R} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]_{\mathrm{t}_1^\sharp}) \subseteq \gamma_{\mathbb{R}^\sharp}([(\mathrm{h}_{\mathrm{i},0}^\sharp *_\mathtt{S} \mathrm{h}_{\mathrm{i},1}^\sharp) \dashrightarrow (\mathrm{h}_{\mathrm{o},0}^\sharp *_\mathtt{S} \mathrm{h}_{\mathrm{o},1}^\sharp)]_{\mathrm{t}^\sharp})$ *with* $\mathrm{t}^\sharp = \mathrm{t}_0^\sharp *_\mathtt{T} \mathrm{t}_1^\sharp$

We observe with the property *1.* that the identity of abstract heap transformation predicates may be less precise than the identity relation of abstract heap relations.

Proof of Theorem 6.1. To prove Theorem 6.1, we use the proof of the properties 2 and 3 of Theorem 4.1.

1. Proof of $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}^\sharp})$ with $\mathrm{t}^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp)$:

   We can prove this property exactly like we proved the property 2 of Theorem 4.1.

$$\begin{aligned}&\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp))\\ &\quad = \{(\mathrm{h},\mathrm{h},\nu) \mid (\mathrm{h},\nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\}\end{aligned}$$

$$\begin{aligned}&\gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}^\sharp})\\ &\quad = \{(\mathrm{h}_0,\mathrm{h}_1,\nu) \mid (\mathrm{h}_0,\nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp) \wedge (\mathrm{h}_1,\nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\\ &\qquad\qquad \wedge (\mathrm{h}_0,\mathrm{h}_1,\nu) \in \gamma_{\mathbb{T}^\sharp}(\mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp))\}\end{aligned}$$

   By soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$, it is obvious that:

$$\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}^\sharp}),\ \text{with } \mathrm{t}^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp)$$

2. Proof of

   $\gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}]_{\mathrm{t}^\sharp_0} *_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}]_{\mathrm{t}^\sharp_1}) \subseteq \gamma_{\mathbb{R}^\sharp}([(\mathrm{h}^\sharp_{\mathrm{i},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{i},1}) \dashrightarrow (\mathrm{h}^\sharp_{\mathrm{o},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{o},1})]_{\mathrm{t}^\sharp})$ with $\mathrm{t}^\sharp = \mathrm{t}^\sharp_0 \ *_{\mathtt{T}}\ \mathrm{t}^\sharp_1$:

   A major part of this property has been proven in the proof of Theorem 4.1 (property 3), the rest only relates to $\mathrm{t}^\sharp_0$, $\mathrm{t}^\sharp_1$ and $\mathrm{t}^\sharp$. So for the sake of clarity, we write "..." for the parts of the proof that already are in the proof of Theorem 4.1, property 3.

$$\begin{aligned}&\gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}]_{\mathrm{t}^\sharp_0} *_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}]_{\mathrm{t}^\sharp_1})\\ &\quad = \{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid\\ &\qquad (\mathrm{h}^\sharp_{\mathrm{i},0}, \mathrm{h}^\sharp_{\mathrm{o},0}, \nu) \in \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}]_{\mathrm{t}^\sharp_0}) \wedge\\ &\qquad (\mathrm{h}^\sharp_{\mathrm{i},1}, \mathrm{h}^\sharp_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}]_{\mathrm{t}^\sharp_1}) \wedge\\ &\qquad \ldots\}\\ &\quad = \{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid\\ &\qquad (\mathrm{h}^\sharp_{\mathrm{i},0}, \mathrm{h}^\sharp_{\mathrm{o},0}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp_0) \wedge\\ &\qquad (\mathrm{h}^\sharp_{\mathrm{i},1}, \mathrm{h}^\sharp_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp_1) \wedge\\ &\qquad \ldots\}\end{aligned}$$

$$
\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}([(\mathrm{h}^\sharp_{\mathrm{i},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{i},1}) \dashrightarrow (\mathrm{h}^\sharp_{\mathrm{o},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{o},1})]_{\mathrm{t}^\sharp}) \\
&\quad = \{(\mathrm{h_i}, \mathrm{h_o}, \nu) \mid \\
&\qquad (\mathrm{h_i}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{i},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{i},1}) \wedge \\
&\qquad (\mathrm{h_o}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{o},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{o},1}) \wedge \\
&\qquad (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)\} \\
&\quad = \{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid \\
&\qquad (\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp) \\
&\qquad \wedge \ldots\} \\
&\quad = \{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid \\
&\qquad (\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp_0 *_{\mathtt{T}} \mathrm{t}^\sharp_1) \\
&\qquad \wedge \ldots\}
\end{aligned}
$$

By the soundness of $*_{\mathtt{T}}$, we observe that:

$$
\begin{aligned}
&\{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid \\
&\qquad (\mathrm{h}^\sharp_{\mathrm{i},0}, \mathrm{h}^\sharp_{\mathrm{o},0}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp_0) \wedge (\mathrm{h}^\sharp_{\mathrm{i},1}, \mathrm{h}^\sharp_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp_1)\} \\
&\subseteq \\
&\{(\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \mid \\
&\qquad (\mathrm{h}_{\mathrm{i},0} \circledast \mathrm{h}_{\mathrm{i},1}, \mathrm{h}_{\mathrm{o},0} \circledast \mathrm{h}_{\mathrm{o},1}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp_0 *_{\mathtt{T}} \mathrm{t}^\sharp_1)\}
\end{aligned}
$$

So finally:

$$
\begin{aligned}
&\gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_{\mathrm{i},0} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},0}]_{\mathrm{t}^\sharp_0} *_{\mathtt{R}} [\mathrm{h}^\sharp_{\mathrm{i},1} \dashrightarrow \mathrm{h}^\sharp_{\mathrm{o},1}]_{\mathrm{t}^\sharp_1}) \\
&\subseteq \\
&\gamma_{\mathbb{R}^\sharp}([(\mathrm{h}^\sharp_{\mathrm{i},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{i},1}) \dashrightarrow (\mathrm{h}^\sharp_{\mathrm{o},0} *_{\mathtt{S}} \mathrm{h}^\sharp_{\mathrm{o},1})]_{\mathrm{t}^\sharp})
\end{aligned}
$$

□

In the following, we give two examples of abstract heap transformation predicates that, when combined together, are expressive enough to ensure the properties (a) and (b) of Remark 3. The first one describes relations between the sets of addresses that define the input and output heaps. The second one describes the set of fields for which the value of all cells may have been modified. Finally we define the combination of two abstract heap transformation predicates domains.

6.3 The Footprint Predicates Domain

Recall the property (a) of Remark 3. To ensure that the sorting algorithm operates in-place, we need to express that the set of addresses of the input and the output lists are strictly equal. To do that, we can design an abstract heap transformation predicates domain that compares the set of addresses of the input and output heaps. We name *the*

footprint predicates domain the abstract heap transformation predicates domain where $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$. Each element of $\mathbb{T}^\sharp$ compares the set of addresses that define the input heap with the set of addresses that define the output heap. They do not provide relations between the content of the memory cells.

Let $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}$ be a transform-into relation annotated by $\mathrm{t}^\sharp$. If $\mathrm{t}^\sharp$ is $=^\sharp$, then the input heaps abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ and the output heaps abstracted $\mathrm{h}_\mathrm{o}^\sharp$ are defined by the same set of addresses. It indicates that any allocation or deallocation may occur. When $\mathrm{t}^\sharp$ is $\subseteq^\sharp$ (respectively $\supseteq^\sharp$), the set of addresses of the heaps abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ is included in (respectively includes) or is equal to the set of addresses of the heaps abstracted by $\mathrm{h}_\mathrm{o}^\sharp$. This indicates that only allocations (respectively deallocations) may have taken place, as the output heap is bigger than (respectively smaller than) or is equal to the input heap. Finally, if $\mathrm{t}^\sharp = \top$, then it indicates no specific relations between the input and the output heap.

Definition 6.3 (Concretization of the footprint predicates domain). *We give a formal meaning to the set footprint predicates domain by defining its concretization function* $\gamma_{\mathbb{T}^\sharp} \in \mathbb{T}^\sharp \to \mathcal{P}(\mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$*:*

$$\begin{array}{rcl}
\gamma_{\mathbb{T}^\sharp}(=^\sharp) & = & \{(\mathrm{h}_\mathrm{i}, \mathrm{h}_\mathrm{o}, \nu) \in \mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}) \mid \mathbf{dom}(\mathrm{h}_\mathrm{i}) = \mathbf{dom}(\mathrm{h}_\mathrm{o})\} \\
\gamma_{\mathbb{T}^\sharp}(\subseteq^\sharp) & = & \{(\mathrm{h}_\mathrm{i}, \mathrm{h}_\mathrm{o}, \nu) \in \mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}) \mid \mathbf{dom}(\mathrm{h}_\mathrm{i}) \subseteq \mathbf{dom}(\mathrm{h}_\mathrm{o})\} \\
\gamma_{\mathbb{T}^\sharp}(\supseteq^\sharp) & = & \{(\mathrm{h}_\mathrm{i}, \mathrm{h}_\mathrm{o}, \nu) \in \mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}) \mid \mathbf{dom}(\mathrm{h}_\mathrm{i}) \supseteq \mathbf{dom}(\mathrm{h}_\mathrm{o})\} \\
\gamma_{\mathbb{T}^\sharp}(\top) & = & \mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V})
\end{array}$$

Example 6.1 (Expressiveness). In this example, we discuss the expressiveness of the footprint predicates domain, for each value of $\mathrm{t}^\sharp$ in the following transform-into relation:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]_{\mathrm{t}^\sharp}$$

If $\mathrm{t}^\sharp$ is $=^\sharp$, then we know any allocation or deallocation occurred: the two lists have the same size and the same physical memory cells. However, we have no information about the order or the value of each memory cell. If $\mathrm{t}^\sharp = \subseteq^\sharp$ (respectively $\mathrm{t}^\sharp = \supseteq^\sharp$), then the output list may contain more (respectively fewer) elements than the input list. Here too, any information about the order or the value of the elements. Finally, if $\mathrm{t}^\sharp = \top$, we have any interesting information about the transformation of the input list into the output list.

More generally, we have the following properties:

Theorem 6.2 (Properties of the footprint predicates domain). *We observe the following properties about the footprint predicates domain:*

1. $\gamma_{\mathbb{T}^\sharp}(=^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\subseteq^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\top)$
2. $\gamma_{\mathbb{T}^\sharp}(=^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\supseteq^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\top)$
3. $\gamma_{\mathbb{T}^\sharp}(\subseteq^\sharp) \cup \gamma_{\mathbb{T}^\sharp}(\supseteq^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\top)$

Proof of Theorem 6.2. The proof of these three properties is trivial using the definition of the concretization function. Thus we do not detail it. □

We can deduce from these properties that $\top$ is the less precise element of $\mathbb{T}^\sharp$ (it provides any relation). On the contrary, $=^\sharp$ is the most precise: the set of addresses of the input and output heaps are the same but not necessarily the content of each memory cell. As well, it is relevant to express that the input heap has been manipulated in place (point (a)).

Definition 6.4 (Function $\mathbf{id}_{\mathbb{T}^\sharp}$ of the footprint predicates domain).

$$\textit{Let } \mathrm{h}^\sharp \in \mathbb{H}^\sharp, \textit{ then: } \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp) = \quad =^\sharp$$

Definition 6.5 (Operator $*_{\mathtt{T}}$ of the footprint predicates domain). *For simplicity, we define this operator with a table as follows:*

$*_{\mathtt{T}}$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$=^\sharp$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$\subseteq^\sharp$	$\subseteq^\sharp$	$\subseteq^\sharp$	$\top$	$\top$
$\supseteq^\sharp$	$\supseteq^\sharp$	$\top$	$\supseteq^\sharp$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Theorem 6.3 (Soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_{\mathtt{T}}$). *The functions $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_{\mathtt{T}}$ of the footprint predicates domain satisfy respectively Assumption 6.1 and Assumption 6.2.*

Proof of Theorem 6.3. The proof of this theorem is trivial. For $\mathbf{id}_{\mathbb{T}^\sharp}$, we could choose any other predicate of the domain, but the chosen one is the most precise, as expressed by Theorem 6.2. □

6.4 The Fields Predicates Domain

When manipulating data structures, it is common that a function modifies *partially* the memory. For instance in Figure 2.3, the sort function may modify the order of the input list manipulating the `next` fields whereas the values of the `data` fields do not change. However, as data structures like linked lists are abstracted by inductive predicates that summarize memory heaps, we cannot capture the partial modification property. In our sort function, proving that only the values of the `next` fields may have been modified would prove that the values of `data` fields have not been modified. More specifically, it would ensure that all the data that are both in the input and output lists are the same.

We can express the set of structure fields whose value *may have changed* designing an abstract heap transformation predicates domain such as $\mathbb{T}^\sharp = \mathcal{P}(\mathbb{F})$. We name this

domain *the fields predicates domain*. Let $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_F$ be the transform-into relation annotated by $F \in \mathcal{P}(\mathbb{F})$. If $\mathtt{f}$ is a structure field that *is not* in F, then each field $\mathtt{f}$ of elements of the structure that is both in the input and output heaps abstracted respectively by $\mathrm{h}_\mathrm{i}^\sharp$ and $\mathrm{h}_\mathrm{o}^\sharp$ has the same value.

Definition 6.6 (Concretization of the fields predicates domain). *Let $F \in \mathcal{P}(\mathbb{F})$. We define the concretization of the fields predicates domain $\gamma_{\mathbb{T}^\sharp} \in \mathcal{P}(\mathbb{F}) \to \mathcal{P}(\mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$ as follows:*

$$\begin{aligned}\gamma_{\mathbb{T}^\sharp}(F) \quad = \quad & \{(\mathrm{h_i}, \mathrm{h_o}, \nu) \in \mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}) \mid \\ & \forall \mathtt{f} \notin F, \forall \mathrm{a} \in \mathbb{A}, (\mathrm{a} + \mathtt{f}) \in \mathbf{dom}(\mathrm{h_i}) \wedge (\mathrm{a} + \mathtt{f}) \in \mathbf{dom}(\mathrm{h_o}) \\ & \Rightarrow \mathrm{h_i}(\mathrm{a} + \mathtt{f}) = \mathrm{h_o}(\mathrm{a} + \mathtt{f})\}\end{aligned}$$

Example 6.2 (Expressiveness). We know discuss the expressiveness of the fields predicates domain for the following transformation-into relation:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]_{\mathrm{t}^\sharp}$$

If $\mathrm{t}^\sharp = \{\mathtt{next}\}$, then only the `next` fields of the elements of the two lists may have been modified. Consequently, all the `data` fields of the lists elements are unchanged. We observe that the lists may have a different number of elements. For instance, the output list can be the input where we deallocated randomly some elements, without modifying the value of the `data` fields.

Theorem 6.4 (Property of the fields predicates domain). *We observe the following property about the fields predicates domain:*

$$\forall F_1, F_2 \in \mathcal{P}(\mathbb{F}), F_1 \subseteq F_2 \Rightarrow \gamma_{\mathbb{T}^\sharp}(F_1) \subseteq \gamma_{\mathbb{T}^\sharp}(F_2)$$

This property means that the bigger the set of fields is, the less precise the predicate is. Indeed if a field is not in the set, then we know that all its values have not changed. On the contrary for all the fields in the set, we do not know if their values have changed or not.

Proof of Theorem 6.4. This proof is trivial using the definition of the concretization function. □

We made the choice to use the set of fields that may have changed instead of the set of fields that have not changed for the sake of simplicity. Indeed, if we would, we would have the following property: $\forall F_1, F_2 \in \mathcal{P}(\mathbb{F}), F_1 \supseteq F_2 \Rightarrow \gamma_{\mathbb{T}^\sharp}(F_1) \subseteq \gamma_{\mathbb{T}^\sharp}(F_2)$ that is not relevant and easy to manipulate.

Definition 6.7 (Function $\mathbf{id}_{\mathbb{T}^\sharp}$ of the fields predicates domain).

$$\textit{Let } \mathrm{h}^\sharp \in \mathbb{H}^\sharp\textit{, then: } \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp) = \{\}$$

Definition 6.8 (Operator $*_{\mathtt{T}}$ of the fields predicates domain).

$$\textit{Let } F_1, F_2 \in \mathcal{P}(\mathbb{F})\textit{, then: } F_1 *_{\mathtt{T}} F_2 = F_1 \cup F_2$$

Theorem 6.5 (Soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_{\mathtt{T}}$). *The functions $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_{\mathtt{T}}$ of the fields predicates domain satisfy respectively Assumption 6.1 and Assumption 6.2.*

Proof of Theorem 6.5. The proof of this theorem is trivial. For $\mathbf{id}_{\mathbb{T}^\sharp}$, we can see from Theorem 6.4 that the empty set is the most precise element of $\mathbb{T}^\sharp$. This is why we use it to define this function. □

6.5 The Combined Predicates Domain

The footprint predicates domain expresses relations between the set of addresses of the input and the output heaps but does not express information about the content of memory cells. On the other side, the fields predicates domain provides information about the content of memory cells but not about addresses. If we can combine the informations given by both of these abstract heap predicates domains, we can ensure the properties (a) and (b) of Remark 3.

A reduced product [CC79] between the footprint and the fields predicates domains is sufficient but not modular enough. Indeed, to analyze other programs, we have to combine other abstract heap transformation predicates domains. To be the most generic as possible, we define *the combined predicates domain*, the abstract heap transformation predicates domain $\mathbb{T}^\sharp = \mathbb{T}_1^\sharp \times \mathbb{T}_2^\sharp$ as the reduced product of any abstract heap transformation predicates domains $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$.

Let $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}$ be a transform-into relation annotated by $\mathrm{t}^\sharp$. If $\mathrm{t}^\sharp$ is the product between two abstract heap transformation predicates $\mathrm{t}_1^\sharp \in \mathbb{T}_1^\sharp$ and $\mathrm{t}_2^\sharp \in \mathbb{T}_2^\sharp$, then it describes the transformation of the heap abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ into the heap abstracted by $\mathrm{h}_\mathrm{o}^\sharp$ by combining the transformations described both by $\mathrm{t}_1^\sharp$ and $\mathrm{t}_2^\sharp$.

Definition 6.9 (Concretization of the combined predicates domain). *Let $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$ be two abstract heap transformation predicates domains and $\mathrm{t}_1^\sharp \in \mathbb{T}_1^\sharp$ and $\mathrm{t}_2^\sharp \in \mathbb{T}_2^\sharp$. Let $\gamma_{\mathbb{T}_1^\sharp}$ and $\gamma_{\mathbb{T}_2^\sharp}$ be the concretization function of respectively $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$, we define the concretization function of the product between $\mathrm{t}_1^\sharp$ and $\mathrm{t}_2^\sharp$, $\gamma_{\mathbb{T}^\sharp} \in \mathbb{T}_1^\sharp \times \mathbb{T}_2^\sharp \to \mathcal{P}(\mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$ as follows:*

$$\gamma_{\mathbb{T}^\sharp}(\mathrm{t}_1^\sharp, \mathrm{t}_2^\sharp) \quad = \quad \gamma_{\mathbb{T}_1^\sharp}(\mathrm{t}_1^\sharp) \cap \gamma_{\mathbb{T}_2^\sharp}(\mathrm{t}_2^\sharp)$$

Theorem 6.6 (Property of the combined predicates domain). *Let $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$ be two abstract heap transformation predicates domains and $\mathbb{T}^\sharp$ their product. Let $\gamma_{\mathbb{T}_1^\sharp}$, $\gamma_{\mathbb{T}_2^\sharp}$ and $\gamma_{\mathbb{T}^\sharp}$ be respectively their concretization functions. Then:*

$$\forall t_{1,a}^\sharp, t_{1,b}^\sharp \in \mathbb{T}_1^\sharp, \quad \forall t_{2,a}^\sharp, t_{2,b}^\sharp \in \mathbb{T}_2^\sharp,$$

$$\gamma_{\mathbb{T}_1^\sharp}(t_{1,a}^\sharp) \subseteq \gamma_{\mathbb{T}_1^\sharp}(t_{1,b}^\sharp) \wedge \gamma_{\mathbb{T}_2^\sharp}(t_{2,a}^\sharp) \subseteq \gamma_{\mathbb{T}_2^\sharp}(t_{2,b}^\sharp) \;\Rightarrow\; \gamma_{\mathbb{T}^\sharp}(t_{1,a}^\sharp, t_{2,a}^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(t_{1,b}^\sharp, t_{2,b}^\sharp)$$

Proof of Theorem 6.6. To prove this property we simply substitute $\gamma_{\mathbb{T}^\sharp}$ by its definition and we obtain:

$$\forall t_{1,a}^\sharp, t_{1,b}^\sharp \in \mathbb{T}_1^\sharp, \quad \forall t_{2,a}^\sharp, t_{2,b}^\sharp \in \mathbb{T}_2^\sharp,$$

$$\begin{gathered}\gamma_{\mathbb{T}_1^\sharp}(t_{1,a}^\sharp) \subseteq \gamma_{\mathbb{T}_1^\sharp}(t_{1,b}^\sharp) \wedge \gamma_{\mathbb{T}_2^\sharp}(t_{2,a}^\sharp) \subseteq \gamma_{\mathbb{T}_2^\sharp}(t_{2,b}^\sharp) \\ \Rightarrow \\ \gamma_{\mathbb{T}_1^\sharp}(t_{1,a}^\sharp) \cap \gamma_{\mathbb{T}_2^\sharp}(t_{2,a}^\sharp) \subseteq \gamma_{\mathbb{T}_1^\sharp}(t_{1,b}^\sharp) \cap \gamma_{\mathbb{T}_2^\sharp}(t_{2,b}^\sharp)\end{gathered}$$

□

Definition 6.10 (Function $\mathbf{id}_{\mathbb{T}^\sharp}$ of the combined predicates domain). *Let $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$ be two abstract heap transformation predicates domains and $\mathbb{T}^\sharp$ their product. Let $h^\sharp \in \mathbb{H}^\sharp$, then :*

$$\mathbf{id}_{\mathbb{T}^\sharp}(h^\sharp) = (\mathbf{id}_{\mathbb{T}_1^\sharp}(h^\sharp), \mathbf{id}_{\mathbb{T}_2^\sharp}(h^\sharp))$$

Definition 6.11 (Operator $*_\mathtt{T}$ of the combined predicates domain). *Let $\mathbb{T}_1^\sharp$ and $\mathbb{T}_2^\sharp$ be two abstract heap transformation predicates domains and $\mathbb{T}^\sharp$ their product. If $t_{1,a}^\sharp, t_{1,b}^\sharp \in \mathbb{T}_1^\sharp$ and $t_{2,a}^\sharp, t_{2,b}^\sharp \in \mathbb{T}_2^\sharp$ then:*

$$(t_{1,a}^\sharp, t_{2,a}^\sharp) *_\mathtt{T} (t_{1,b}^\sharp, t_{2,b}^\sharp) = (t_{1,a}^\sharp *_{\mathtt{T}_1} t_{1,b}^\sharp \;,\; t_{2,a}^\sharp *_{\mathtt{T}_2} t_{2,b}^\sharp)$$

Theorem 6.7 (Soundness of $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_\mathtt{T}$). *The functions $\mathbf{id}_{\mathbb{T}^\sharp}$ and $*_\mathtt{T}$ of the combined predicates domain satisfy respectively Assumption 6.1 and Assumption 6.2.*

Proof of Theorem 6.7. The function $\mathbf{id}_{\mathbb{T}^\sharp}$ is simply the product of the functions $\mathbf{id}_{\mathbb{T}_1^\sharp}$ and $\mathbf{id}_{\mathbb{T}_2^\sharp}$ of the sub-domains. It is easy to prove its soundness if we suppose that $\mathbf{id}_{\mathbb{T}_1^\sharp}$ and $\mathbf{id}_{\mathbb{T}_2^\sharp}$ are both sound. Similarly, we can prove that $*_\mathtt{T}$ is sound supposing that $*_{\mathtt{T}_1}$ and $*_{\mathtt{T}_2}$ are sound. □

$$
\begin{array}{rcll}
\mathbf{id}_{\mathbb{T}^\sharp} & \in & \mathbb{H}^\sharp & \to \quad \mathbb{T}^\sharp \\
*_{\mathrm{T}} & \in & \mathbb{T}^\sharp \times \mathbb{T}^\sharp & \to \quad \mathbb{T}^\sharp \\
\mathbf{assign}_{\mathbb{T}^\sharp} & \in & \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \times \mathbb{T}^\sharp & \to \quad \mathbb{T}^\sharp \\
\mathbf{unfold}_{\mathbb{T}^\sharp} & \in & \mathbb{V}^\sharp \times \mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathbb{T}^\sharp & \to \quad \mathcal{P}_{\mathrm{fin}}(\mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathbb{P}^\sharp) \\
\mathbf{alloc}_{\mathbb{T}^\sharp} & \in & \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp & \to \quad \mathbb{T}^\sharp \\
\mathbf{free}_{\mathbb{T}^\sharp} & \in & \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{T}^\sharp & \to \quad \mathbb{T}^\sharp \\
\mathbf{isle}_{\mathbb{T}^\sharp} & \in & (\mathbb{V}^\sharp \to \mathbb{V}^\sharp) \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp & \to \quad \{\mathbf{true}, \mathbf{false}\} \\
\mathbf{join}_{\mathbb{T}^\sharp} & \in & (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp & \to \quad \mathbb{T}^\sharp \\
\mathbf{wid}_{\mathbb{T}^\sharp} & \in & (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp & \to \quad \mathbb{T}^\sharp
\end{array}
$$

Figure 6.1: Interfaces for abstract heap transformation predicates domains.

Example 6.3 (Computed transform-into relation for the list sort). Using the product of the fields and the footprints predicates domains, we obtain the following transform-into relation for the list sort program in Figure 2.3:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta)]_{\mathrm{t}^\sharp}, \text{ with } \mathrm{t}^\sharp = (\{\texttt{next}\}, =^\sharp)$$

This abstract heap relation describes exactly the points (a) and (b) of Remark 3: the function works in place and the input and output lists have the same length, as they are defined by the same set of addresses (a). Furthermore, only the `next` fields of the list may have been modified, so this implies that the lists have exactly the same data (b).

The footprint and the fields predicates domains offer the advantages to be complementary and totally independent from data structures (they are not specific to linked lists). They can also express other properties like for example, if we traverse a binary tree and increment its data elements, the product of these domains is able to prove that only data may have been modified. For more specific relational properties, designing new abstract heap transformation predicates domains is possible.

6.6 Integration in the Analysis

Abstract heap transformation predicates extend abstract heap relations. Thus, to integrate them in the relational analysis of Chapter 5, it needs to update the functions related to abstract heap relations, that is $\mathbf{unfold}_{\mathbb{R}^\sharp}$, $\mathbf{assign}_{\mathbb{R}^\sharp}$, $\mathbf{alloc}_{\mathbb{R}^\sharp}$, $\mathbf{free}_{\mathbb{R}^\sharp}$, $\mathbf{isle}_{\mathbb{R}^\sharp}$, $\mathbf{join}_{\mathbb{R}^\sharp}$ and $\mathbf{wid}_{\mathbb{R}^\sharp}$. All of these functions are extended using their corresponding function at abstract heap transformation predicates level, whose signatures are provided in Figure 6.1.

6.6.1 Refined Unfolding

Abstract heap transformation predicates can also help to gain more precision. As an example, we consider the following abstract heap relation: $[\mathbf{list}(\alpha_0) \dashrightarrow \mathbf{list}(\alpha_0)]_{\mathrm{t}^\sharp}$. Unfolding α_0 will generate the disjuncts $([\mathbf{emp} \dashrightarrow \mathbf{emp}]_{\mathrm{t}^\sharp}, \alpha_0 = 0)$ and $([(\alpha_0 \cdot \mathtt{data} \mapsto \beta_1 *_\mathtt{s} \alpha_0 \cdot \mathtt{next} \mapsto \beta_2 *_\mathtt{s} \mathbf{list}(\beta_2)) \dashrightarrow (\alpha_0 \cdot \mathtt{data} \mapsto \delta_1 *_\mathtt{s} \alpha_0 \cdot \mathtt{next} \mapsto \delta_2 *_\mathtt{s} \mathbf{list}(\delta_2))]_{\mathrm{t}^\sharp}, \alpha_0 \neq 0)$.

In the second disjunct, we observe that we have no information that says whether the values of the fields `next` and `data` of α_0 are respectively the same in both sides of the $[. \dashrightarrow .]$ relation. However, if we consider that we use the fields predicates domain defined in Section 6.4 and that $\mathrm{t}^\sharp = \{\mathtt{data}\}$, we know that all `next` fields in this abstract relation are left unmodified. Thus, unfolding α_0 taking in account this information will generate as second disjunct $([(\alpha_0 \cdot \mathtt{data} \mapsto \beta_1 *_\mathtt{s} \alpha_0 \cdot \mathtt{next} \mapsto \alpha_2 *_\mathtt{s} \mathbf{list}(\alpha_2)) \dashrightarrow (\alpha_0 \cdot \mathtt{data} \mapsto \delta_1 *_\mathtt{s} \alpha_0 \cdot \mathtt{next} \mapsto \alpha_2 *_\mathtt{s} \mathbf{list}(\alpha_2))]_{\mathrm{t}^\sharp}, \alpha_0 \neq 0)$ (as only the value of `data` fields may have changed, we can use the same variable α_2 in both sides of the relation).

The refinement of materialization is performed by the function $\mathbf{unfold}_{\mathbb{T}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathbb{T}^\sharp \to \mathcal{P}_{\mathrm{fin}}(\mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathbb{P}^\sharp)$ that refines the unfolded input and output abstract heaps at the given address, taking in account the information provided by the transformation predicate. This function should satisfy the following assumption:

Assumption 6.3 (Soundness of $\mathbf{unfold}_{\mathbb{T}^\sharp}$). *Before giving the soundness condition of* $\mathbf{unfold}_{\mathbb{T}^\sharp}$*, we define the concretization function* $\gamma_\Pi \in \mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathbb{P}^\sharp \to \mathcal{P}(\mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$ *of a triplet* $(\mathrm{h}^\sharp_{\mathrm{i,u}}, \mathrm{h}^\sharp_{\mathrm{o,u}}, \mathrm{p}^\sharp) \in \mathbf{unfold}_{\mathbb{T}^\sharp}(\alpha, \mathrm{h}^\sharp_\mathrm{i}, \mathrm{h}^\sharp_\mathrm{o}, \mathrm{t}^\sharp)$*:*

$$\gamma_\Pi(\mathrm{h}^\sharp_{\mathrm{i,u}}, \mathrm{h}^\sharp_{\mathrm{o,u}}, \mathrm{p}^\sharp) = \{(\mathrm{h_i}, \mathrm{h_o}, \nu) \mid (\mathrm{h_i}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{i,u}}) \wedge (\mathrm{h_o}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{o,u}}) \\ \wedge \exists \mathrm{v}, (\mathrm{v}, \nu) \in \gamma_{\mathbb{P}^\sharp}(\mathrm{p}^\sharp) \wedge \mathrm{v} \neq 0\}$$

Let $\mathbb{T}^\sharp$ *be an abstract heap transformation predicates domain,* $\mathrm{t}^\sharp \in \mathbb{T}^\sharp, \mathrm{h}^\sharp_\mathrm{i}, \mathrm{h}^\sharp_\mathrm{o} \in \mathbb{H}^\sharp$ *and* $\alpha \in \mathbb{V}^\sharp$*. Then* $\mathbf{unfold}_{\mathbb{T}^\sharp}$ *is sound if:*

$$\{(\mathrm{h_i}, \mathrm{h_o}, \nu) \mid (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp) \wedge (\mathrm{h_i}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_\mathrm{i}) \wedge (\mathrm{h_o}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_\mathrm{o})\} \\ \subseteq \\ \bigcup\{\gamma_\Pi(\mathrm{h}^\sharp_{\mathrm{i,u}}, \mathrm{h}^\sharp_{\mathrm{o,u}}, \mathrm{p}^\sharp) \mid (\mathrm{h}^\sharp_{\mathrm{i,u}}, \mathrm{h}^\sharp_{\mathrm{o,u}}, \mathrm{p}^\sharp) \in \mathbf{unfold}_{\mathbb{T}^\sharp}(\alpha, \mathrm{h}^\sharp_\mathrm{i}, \mathrm{h}^\sharp_\mathrm{o}, \mathrm{t}^\sharp)\}\}$$

Intuitively, we can see that $\mathbf{unfold}_{\mathbb{T}^\sharp}(\alpha, \mathrm{h}^\sharp_\mathrm{i}, \mathrm{h}^\sharp_\mathrm{o}, \mathrm{t}^\sharp)$ propagates the information provided by $\mathrm{t}^\sharp$ into $\mathrm{h}^\sharp_\mathrm{i}$ and $\mathrm{h}^\sharp_\mathrm{o}$. The result of this propagation should not introduce more information than $\mathrm{t}^\sharp$.

Definition 6.12 (Extended definition of $\mathbf{unfold}_{\mathbb{R}^\sharp}$). *We now extend the definition of* $\mathbf{unfold}_{\mathbb{R}^\sharp}$ *(page 62) taking into account abstract heap transformations predicates:*

$$\begin{aligned}\mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha, [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}) = \{([\mathrm{h}_{\mathrm{i,t}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o,t}}^\sharp]_{\mathrm{t}^\sharp}, \mathrm{p}_\mathrm{t}^\sharp \wedge \mathrm{p}_{\mathrm{i,u}}^\sharp \wedge \mathrm{p}_{\mathrm{o,u}}^\sharp) \mid \\ (\mathrm{h}_{\mathrm{i,u}}^\sharp, \mathrm{p}_{\mathrm{i,u}}^\sharp) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, \mathrm{h}_\mathrm{i}^\sharp) \wedge (\mathrm{h}_{\mathrm{o,u}}^\sharp, \mathrm{p}_{\mathrm{o,u}}^\sharp) \in \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha, \mathrm{h}_\mathrm{o}^\sharp) \\ \wedge (\mathrm{h}_{\mathrm{i,t}}^\sharp, \mathrm{h}_{\mathrm{o,t}}^\sharp, \mathrm{p}_\mathrm{t}^\sharp) \in \mathbf{unfold}_{\mathbb{T}^\sharp}(\alpha, \mathrm{h}_{\mathrm{i,u}}^\sharp, \mathrm{h}_{\mathrm{o,u}}^\sharp, \mathrm{t}^\sharp)\}\end{aligned}$$

6.6.2 Assignment

The assignment computed on abstract heap transformation predicates level is defined by the function $\mathbf{assign}_{\mathbb{T}^\sharp}$. It inputs the same arguments as $\mathbf{assign}_{\mathbb{R}^\sharp}$: a symbolic value α and a field $\mathtt{f}$, such as $\alpha \cdot \mathtt{f}$ is the address of the points-to predicate being modified, another symbolic value β which denotes the value being assigned, and an abstract heap transformation predicate $\mathrm{t}^\sharp$. It returns a new abstract heap transformation predicate that expresses the assignment from $\mathrm{t}^\sharp$. It should satisfy the following assumption:

Assumption 6.4 (Soundness of $\mathbf{assign}_{\mathbb{T}^\sharp}$). *Let* $\mathbb{T}^\sharp$ *be an abstract heap transformation predicates domain and* $\mathrm{t}^\sharp \in \mathbb{T}^\sharp$*. Let* $\alpha, \beta \in \mathbb{V}^\sharp$ *and* $\mathtt{f} \in \mathbb{F}$*, then* $\mathbf{assign}_{\mathbb{T}^\sharp}$ *is sound if:*

$$\begin{aligned}\{(\mathrm{h_i},\ \mathrm{h_o}[\nu(\alpha) + \mathtt{f} \leftarrow \nu(\beta)],\ \nu) \mid (\mathrm{h_i},\ \mathrm{h_o}\ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)\} \\ \subseteq \gamma_{\mathbb{T}^\sharp}(\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{t}^\sharp))\end{aligned}$$

Definition 6.13 (Extended definition of $\mathbf{assign}_{\mathbb{R}^\sharp}$). *We extend the definition of* $\mathbf{assign}_{\mathbb{R}^\sharp}$ *(page 65) to take into account abstract heap transformation predicates:*

- $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp *_\mathtt{S} (\alpha \cdot \mathtt{f} \mapsto \gamma)]_{\mathrm{t}_1^\sharp}) = [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp *_\mathtt{S} (\alpha \cdot \mathtt{f} \mapsto \beta)]_{\mathrm{t}_2^\sharp}$ *with* $\mathrm{t}_2^\sharp = \mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{t}_1^\sharp)$
- $\mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathtt{Id}(\mathrm{h}_0^\sharp *_\mathtt{S} \alpha \cdot \mathtt{f} \mapsto \delta)) = \mathtt{Id}(\mathrm{h}_0^\sharp) *_\mathtt{R} [\alpha \cdot \mathtt{f} \mapsto \delta \dashrightarrow \alpha \cdot \mathtt{f} \mapsto \beta]_{\mathrm{t}^\sharp}$ *with* $\mathrm{t}^\sharp = \mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, \mathbf{id}_{\mathbb{T}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \delta))$

Case of a Transform-into Relation. This case is obvious. The assignment is expressed on abstract heap transformation predicates level by applying $\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{t}_1^\sharp)$, when α, $\mathtt{f}$, β are arguments of $\mathbf{assign}_{\mathbb{R}^\sharp}$ and $\mathrm{t}_1^\sharp$ the abstract heap transformation predicate attached the transform-into relation being modified. The resulting abstract heap transformation predicate $\mathrm{t}_2^\sharp$ is attached to the resulting transform-into relation.

Case of an Identity Relation. After splitting $\mathtt{Id}(\mathrm{h}_0^\sharp *_\mathtt{S} (\alpha \cdot \mathtt{f} \mapsto \delta))$ into $\mathtt{Id}(\mathrm{h}_0^\sharp) *_\mathtt{R} \mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \delta)$ as explained in Section 5.4, the analysis needs to weaken $\mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \delta)$

into a transform-into relation. In the first version of $\mathbf{assign}_{\mathbb{R}^\sharp}$, we used Theorem 4.1 (page 48) to perform this weakening. However, we now should take into account abstract heap transformation predicates. As observed in Theorem 6.1, we have $\gamma_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) \subseteq \gamma_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}^\sharp})$ with $\mathrm{t}^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp)$. Thus, we can weaken $\mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \delta)$ into $[(\alpha \cdot \mathtt{f} \mapsto \delta) \dashrightarrow (\alpha \cdot \mathtt{f} \mapsto \delta)]_{\mathrm{t}^\sharp_0}$ with $\mathrm{t}^\sharp_0 = \mathbf{id}_{\mathbb{T}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \delta)$. Finally, the analysis performs the assignment in the weakened transform-into relation, similarly as in the previous paragraph.

Definition 6.14 (Assignment for abstract heap transformation predicates domains). *We define the assignment function* $\mathbf{assign}_{\mathbb{T}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ *for each abstract heap transformation predicates domain defined respectively in Section 6.3, Section 6.4 and Section 6.5.*

1. *The footprint predicates domain,* $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$*:*

$$\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{t}^\sharp) = \mathrm{t}^\sharp$$

2. *The fields predicates domain,* $\mathbb{T}^\sharp = \mathcal{P}(\mathbb{F})$*:*

$$\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{t}^\sharp) = \mathrm{t}^\sharp \cup \{\mathtt{f}\}$$

3. *The combined predicates domain,* $\mathbb{T}^\sharp = \mathbb{T}^\sharp_1 \times \mathbb{T}^\sharp_2$*:*

$$\mathbf{assign}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \beta, (\mathrm{t}^\sharp_1, \mathrm{t}^\sharp_2)) = (\mathbf{assign}_{\mathbb{T}^\sharp_1}(\alpha, \mathtt{f}, \beta, \mathrm{t}^\sharp_1),\ \mathbf{assign}_{\mathbb{T}^\sharp_2}(\alpha, \mathtt{f}, \beta, \mathrm{t}^\sharp_2))$$

Theorem 6.8 (Soundness of Definition 6.14). *The operators from Definition 6.14 are sound in the sense of Assumption 6.4.*

For the footprint predicates domain, an assignment does not modify the set of addresses of a heap, that is why $\mathbf{assign}_{\mathbb{T}^\sharp}$ returns the input abstract heap transformation predicate. For the fields predicate domain, the assignment modifies the content of cell at address $\alpha \cdot \mathtt{f}$, consequently we have to add $\mathtt{f}$ to the set of the possible unpreserved fields. Finally, for the combined predicates domain, $\mathbf{assign}_{\mathbb{T}^\sharp}$ simply applies recursively this function for its sub-predicates domains.

6.6.3 Allocation and Deallocation

Allocation

The allocation at abstract heap transformation predicates level is performed by the function $\mathbf{alloc}_{\mathbb{T}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \to \mathbb{T}^\sharp$, such as $\mathbf{alloc}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \delta)$ returns an abstract heap trans-

formation predicate that over-approximates the allocation of the abstract heap $\alpha \cdot \mathtt{f} \mapsto \delta$. It should satisfy the following condition.

Assumption 6.5 (Soundness of $\mathbf{alloc}_{\mathbb{T}^\sharp}$). *Let $\mathbb{T}^\sharp$ be an abstract heap predicates domain. Let $\beta, \delta \in \mathbb{V}^\sharp$ and $\mathtt{f} \in \mathbb{F}$. The function $\mathbf{alloc}_{\mathbb{T}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \to \mathbb{T}^\sharp$ is sound if and only if:*

$$\{([],\ [\nu(\beta) + \mathtt{f} \mapsto \nu(\delta)],\ \nu) \in \mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V})\} \subseteq \gamma_{\mathbb{T}^\sharp}(\mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta))$$

Definition 6.15 (Extended definition of $\mathbf{alloc}_{\mathbb{R}^\sharp}$). *We extend the definition of $\mathbf{alloc}_{\mathbb{R}^\sharp}$ (page 68) to take into account abstract heap transformation predicates:*

$$\mathbf{alloc}_{\mathbb{R}^\sharp}(\beta, \mathtt{f}, \mathrm{r}^\sharp) = \mathrm{r}^\sharp *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow (\beta \cdot \mathtt{f} \mapsto \delta)]_{\mathrm{t}^\sharp},$$

$$\textit{where } \delta \textit{ is fresh and } \mathrm{t}^\sharp = \mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta)$$

This definition is simple. It just creates the abstract heap transformation predicate $\mathrm{t}^\sharp$ using $\mathbf{alloc}_{\mathbb{T}^\sharp}$ with the address given in $\mathbf{alloc}_{\mathbb{R}^\sharp}$ and with the initial value δ freshly generated. It then attaches $\mathrm{t}^\sharp$ to the resulting transform-into relation.

Definition 6.16 (Allocation for abstract heap transformation predicates domains). *We define the allocation function $\mathbf{alloc}_{\mathbb{T}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{V}^\sharp \to \mathbb{T}^\sharp$ for each abstract heap transformation predicates domain defined respectively in Section 6.3, Section 6.4 and Section 6.5.*

1. *The footprint predicates domain, $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$:*

$$\mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta) = \subseteq^\sharp$$

2. *The fields predicates domain, $\mathbb{T}^\sharp = \mathcal{P}(\mathbb{F})$:*

$$\mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta) = \{\}$$

3. *The combined predicates domain, $\mathbb{T}^\sharp = \mathbb{T}_1^\sharp \times \mathbb{T}_2^\sharp$:*

$$\mathbf{alloc}_{\mathbb{T}^\sharp}(\beta, \mathtt{f}, \delta) = (\mathbf{alloc}_{\mathbb{T}_1^\sharp}(\beta, \mathtt{f}, \delta),\ \mathbf{alloc}_{\mathbb{T}_2^\sharp}(\beta, \mathtt{f}, \delta))$$

Theorem 6.9 (Soundness of Definition 6.16). *The operators from Definition 6.16 are sound in the sense of Assumption 6.5.*

Regarding to the footprint predicates domain, there are two predicates that satisfy Assumption 6.5: $\subseteq^\sharp$ and $\top$. The predicate $\subseteq^\sharp$ is the most precise, and it expresses that some memory cells have been allocated, thus it is the most relevant predicate for this operation. Regarding to the fields predicates domain, an allocation does not modify a memory cell of the input heap, thus no field may be modified. That is why the function returns the empty set. Finally, for the combined predicates domain, $\mathbf{alloc}_{\mathbb{T}^\sharp}$ simply applies recursively this function for its sub-predicates domains.

Deallocation

The deallocation at abstract heap transformation predicates level is performed by the function $\mathbf{free}_{\mathbb{T}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$, such as $\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \mathrm{t}^\sharp)$ returns an abstract heap transformation predicate that over-approximates the deallocation of the cell at address $\alpha \cdot \mathtt{f}$ from $\mathrm{t}^\sharp$. Its soundness assumption is given below.

Assumption 6.6 (Soundness of $\mathbf{free}_{\mathbb{T}^\sharp}$). *Let $\mathbb{T}^\sharp$ be an abstract heap transformation predicates domain and $\mathrm{t}^\sharp \in \mathbb{T}^\sharp$. Let $\alpha \in \mathbb{V}^\sharp$ and $\mathtt{f} \in \mathbb{F}$. The function $\mathbf{free}_{\mathbb{T}^\sharp}$ is sound if and only if:*

$$\{(\mathrm{h_i},\ \mathrm{h_o},\ \nu) \mid \exists \mathrm{v} \in \mathbb{V}, (\mathrm{h_i},\ \mathrm{h_o} \circledast [\nu(\alpha) + \mathtt{f} \mapsto \mathrm{v}],\ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)\}$$
$$\subseteq$$
$$\gamma_{\mathbb{T}^\sharp}(\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \mathrm{t}^\sharp))$$

Definition 6.17 (Extended definition of $\mathbf{free}_{\mathbb{R}^\sharp}$). *We extend the definition of $\mathbf{free}_{\mathbb{R}^\sharp}$ (page 70) to take into account abstract heap transformation predicates:*

- $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp *_\mathtt{S} (\alpha \cdot \mathtt{f} \mapsto \beta)]_{\mathrm{t}_0^\sharp}) = [\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}_1^\sharp}$ *with* $\mathrm{t}_1^\sharp = \mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \mathrm{t}_0^\sharp)$
- $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathtt{Id}(\mathrm{h}^\sharp *_\mathtt{S} \alpha \cdot \mathtt{f} \mapsto \beta)) = \mathtt{Id}(\mathrm{h}^\sharp) *_\mathtt{R} [\alpha \cdot \mathtt{f} \mapsto \beta \dashrightarrow \mathbf{emp}]_{\mathrm{t}^\sharp}$
  with $\mathrm{t}^\sharp = \mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \mathbf{id}_{\mathbb{T}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \beta))$

When applied to a transform-into relation $\mathrm{r}^\sharp$ of the form $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow (\mathrm{h}_\mathrm{o}^\sharp *_\mathtt{S} \alpha \cdot \mathtt{f} \mapsto \beta)]_{\mathrm{t}_0^\sharp}$, $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}^\sharp)$ returns the transform-into relation $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}_1^\sharp}$, where $\mathrm{t}_1^\sharp = \mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \mathrm{t}_0^\sharp)$. When applied to an identity relation $\mathrm{r}^\sharp$ of the form $\mathtt{Id}(\mathrm{h}^\sharp *_\mathtt{S} \alpha \cdot \mathtt{f} \mapsto \beta)$, $\mathbf{free}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}^\sharp)$ proceeds like abstract assignment. It first splits $\mathrm{r}^\sharp$ into $\mathtt{Id}(\mathrm{h}^\sharp) *_\mathtt{R} \mathtt{Id}(\alpha \cdot \mathtt{f} \mapsto \beta)$ and weakens the new transform into relation into $[\alpha \cdot \mathtt{f} \mapsto \beta \dashrightarrow \alpha \cdot \mathtt{f} \mapsto \beta]_{\mathrm{t}_0^\sharp}$, with $\mathrm{t}_0^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \beta)$. It finally proceeds to the deallocation in the latter transform-into relation.

Definition 6.18 (Deallocation for abstract heap transformation predicates domains). *We define the deallocation function* $\mathbf{free}_{\mathbb{T}^\sharp} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ *for each abstract heap transformation predicates domain defined respectively in Section 6.3, Section 6.4 and Section 6.5.*

1. *The footprint predicates domain,* $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$*:*

$$\begin{aligned}\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, =^\sharp) &= \supseteq^\sharp \\ \mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \supseteq^\sharp) &= \supseteq^\sharp \\ \mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \subseteq^\sharp) &= \top \\ \mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \top) &= \top\end{aligned}$$

2. *The fields predicates domain,* $\mathbb{T}^\sharp = \mathcal{P}(\mathbb{F})$*:*

$$\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, \mathrm{t}^\sharp) = \mathrm{t}^\sharp$$

3. *The combined predicates domain,* $\mathbb{T}^\sharp = \mathbb{T}_1^\sharp \times \mathbb{T}_2^\sharp$*:*

$$\mathbf{free}_{\mathbb{T}^\sharp}(\alpha, \mathtt{f}, (\mathrm{t}_1^\sharp, \mathrm{t}_2^\sharp)) = (\mathbf{free}_{\mathbb{T}_1^\sharp}(\alpha, \mathtt{f}, \mathrm{t}_1^\sharp),\ \mathbf{free}_{\mathbb{T}_2^\sharp}(\alpha, \mathtt{f}, \mathrm{t}_2^\sharp))$$

Theorem 6.10 (Soundness of Definition 6.18). *The operators from Definition 6.18 are sound in the sense of Assumption 6.6.*

The deallocation for the footprint predicates domain is more complex than the other operations. When applied to the predicate $=^\sharp$, it returns the predicate $\supseteq^\sharp$, as we have to express that some memory cells have been deallocated from the predicate that ensures that neither deallocations nor allocations occured. When applied to the predicate $\supseteq^\sharp$, it also returns $\supseteq^\sharp$ because some memory cells still may have been deallocated. Finally, when applied to the predicate $\subseteq^\sharp$ or $\top$, we cannot capture any information about if some allocations, deallocations, or even if nothing occured. That is why the function returns $\top$.

Regarding to the fields predicates domain, deleting a cell does not modify its content, thus the deallocation returns the same set of fields.

Finally, as usually for the combined predicates domain, $\mathbf{free}_{\mathbb{T}^\sharp}$ simply applies recursively this function for its sub-predicates domains.

6.6.4 Inclusion

Some inclusion checking rules of Figure 5.3 (page 74) should be updated and take into account abstract heap transformation predicates. This requires to add a new operator

$$\frac{\mathrm{t}_0^\sharp \sqsubseteq_{\mathbb{T}^\sharp} \mathrm{t}_1^\sharp \qquad \mathrm{h}_{\mathrm{i},0}^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathrm{h}_{\mathrm{i},1}^\sharp \qquad \mathrm{h}_{\mathrm{o},0}^\sharp \sqsubseteq_{\mathbb{H}^\sharp} \mathrm{h}_{\mathrm{o},1}^\sharp}{[\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp]_{\mathrm{t}_0^\sharp} \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]_{\mathrm{t}_1^\sharp}} \; (\sqsubseteq_{\dashrightarrow})$$

$$\frac{\mathrm{t}_0^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}^\sharp) \quad \mathrm{r}^\sharp *_{\mathtt{R}} [\mathrm{h}^\sharp \dashrightarrow \mathrm{h}^\sharp]_{\mathrm{t}_0^\sharp} \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}}^\sharp]_{\mathrm{t}_1^\sharp}}{\mathrm{r}^\sharp *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}^\sharp) \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}}^\sharp]_{\mathrm{t}_1^\sharp}} \; (\sqsubseteq_{\mathtt{Id}-\mathrm{weak}})$$

$$\frac{\mathrm{t}_2^\sharp = \mathrm{t}_0^\sharp \; *_{\mathtt{T}} \; \mathrm{t}_1^\sharp \qquad \mathrm{r}^\sharp *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp *_{\mathtt{S}} \mathrm{h}_{\mathrm{o},1}^\sharp]_{\mathrm{t}_2^\sharp} \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}}^\sharp]_{\mathrm{t}^\sharp}}{\mathrm{r}^\sharp *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp]_{\mathrm{t}_0^\sharp} *_{\mathtt{R}} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]_{\mathrm{t}_1^\sharp} \sqsubseteq_{\mathbb{R}^\sharp} [\mathrm{h}_{\mathrm{i}}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o}}^\sharp]_{\mathrm{t}^\sharp}} \; (\sqsubseteq_{\dashrightarrow-\mathrm{weak}})$$

Figure 6.2: Update of some inclusion checking rules from Figure 5.3 (page 74)

$\sqsubseteq_{\mathbb{T}^\sharp}$ at the rules system, in order to reason about abstract heap transformation predicates. This operator is sound, in the sense that it should satisfy the following property:

Theorem 6.11 (Soundness of $\sqsubseteq_{\mathbb{T}^\sharp}$). *Let* $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp \in \mathbb{T}^\sharp$. *Then:*

$$\mathrm{t}_0^\sharp \sqsubseteq_{\mathbb{T}^\sharp} \mathrm{t}_1^\sharp \Longrightarrow \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_0^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_1^\sharp)$$

The updated rules are shown in Figure 6.2. In its original version, rule $(\sqsubseteq_{\dashrightarrow})$ only checked the inclusion of the input abstract heaps and the output abstract heaps accordingly. In its updated version, rule $(\sqsubseteq_{\dashrightarrow})$ also checks the inclusion of the abstract heap transformation predicates of the two transform-into relations. Finally, rules $(\sqsubseteq_{\mathtt{Id}-\mathrm{weak}})$ and $(\sqsubseteq_{\dashrightarrow-\mathrm{weak}})$ are directly derived from Theorem 6.1.

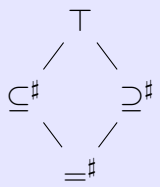
The function $\mathbf{isle}_{\mathbb{T}^\sharp}(\Psi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp)$ implements the operator $\sqsubseteq_{\mathbb{T}^\sharp}$, such as $\mathbf{isle}_{\mathbb{T}^\sharp}(\Psi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp)$ returns **true** if the set of relations that describes $\mathrm{t}_0^\sharp$ is included in the set of relations described by $\mathrm{t}_1^\sharp$ modulo the renaming function Ψ. It should satisfy the following assumption.

Assumption 6.7 (Soundness of $\mathbf{isle}_{\mathbb{T}^\sharp}$). *Let* $\Psi \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp \in \mathbb{T}^\sharp$, *if* $\mathbf{isle}_{\mathbb{T}^\sharp}(\Psi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp) = \mathbf{true}$ *then:*

$$(\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_0^\sharp) \Rightarrow (\mathrm{h_i}, \mathrm{h_o}, \Psi \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_1^\sharp)$$

Definition 6.19 (Inclusion for abstract heap transformation predicates domains). *We define the inclusion function* $\mathbf{isle}_{\mathbb{T}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp) \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \{\mathbf{true}, \mathbf{false}\}$ *for each abstract heap transformation predicates domain defined respectively in Section 6.3, Section 6.4 and Section 6.5.*

1. *The footprint predicates domain,* $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$*: the function* $\mathbf{isle}_{\mathbb{T}^\sharp}$ *is defined using the Hasse diagram of* $\mathbb{T}^\sharp$*:*



2. *The fields predicates domain,* $\mathbb{T}^\sharp = \mathcal{P}(\mathbb{F})$*:*

$$\mathbf{isle}_{\mathbb{T}^\sharp}(\Psi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp) = \mathrm{t}_0^\sharp \subseteq \mathrm{t}_1^\sharp$$

3. *The combined predicates domain,* $\mathbb{T}^\sharp = \mathbb{T}_a^\sharp \times \mathbb{T}_b^\sharp$*:*

$$\mathbf{isle}_{\mathbb{T}^\sharp}(\Psi, (\mathrm{t}_{a,0}^\sharp, \mathrm{t}_{b,0}^\sharp), (\mathrm{t}_{a,1}^\sharp, \mathrm{t}_{b,1}^\sharp)) = \mathbf{isle}_{\mathbb{T}_a^\sharp}(\Psi, \mathrm{t}_{a,0}^\sharp, \mathrm{t}_{a,1}^\sharp) \wedge \mathbf{isle}_{\mathbb{T}_b^\sharp}(\Psi, \mathrm{t}_{b,0}^\sharp, \mathrm{t}_{b,1}^\sharp)$$

Theorem 6.12 (Soundness of Definition 6.19). *The operators from Definition 6.19 are sound in the sense of Assumption 6.7.*

The soundness of this definition is proven using respectively Theorem 6.2 (page 94), Theorem 6.4 (page 96) and Theorem 6.6 (page 98).

6.6.5 Join and Widening

Join

Like inclusion checking, the join should take into account abstract heap transformation predicates. Consequenlty, some rules of the system defined in Figure 5.5 should be updated, by adding a new operator $\sqcup_{\mathbb{T}^\sharp}$ that reasons about abstract heap transformation predicates. This operator is sound if it satisfies the following property:

Theorem 6.13 (Soundness of $\sqcup_{\mathbb{T}^\sharp}$). *Let* $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp \in \mathbb{T}^\sharp$*. Then:*

$$\mathrm{t}_0^\sharp \sqcup_{\mathbb{T}^\sharp} \mathrm{t}_1^\sharp \rightsquigarrow \mathrm{t}^\sharp \quad \Longrightarrow \quad \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_0^\sharp) \cup \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_1^\sharp) \subseteq \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)$$

$$\frac{t_0^\sharp \sqcup_{\mathbb{T}^\sharp} t_1^\sharp \rightsquigarrow t^\sharp \qquad h_{i,0}^\sharp \sqcup_{\mathbb{H}^\sharp} h_{i,1}^\sharp \rightsquigarrow h_i^\sharp \qquad h_{o,0}^\sharp \sqcup_{\mathbb{H}^\sharp} h_{o,1}^\sharp \rightsquigarrow h_o^\sharp}{[h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp]_{t_0^\sharp} \sqcup_{\mathbb{R}^\sharp} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]_{t_1^\sharp} \rightsquigarrow [h_i^\sharp \dashrightarrow h_o^\sharp]_{t^\sharp}} (\sqcup_{\dashrightarrow})$$

$$\frac{t_0^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(h_0^\sharp) \qquad [h_0^\sharp \dashrightarrow h_0^\sharp]_{t_0^\sharp} \sqcup_{\mathbb{R}^\sharp} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]_{t_1^\sharp} \rightsquigarrow r^\sharp}{\mathtt{Id}(h_0^\sharp) \sqcup_{\mathbb{R}^\sharp} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]_{t_1^\sharp} \rightsquigarrow r^\sharp} (\sqcup_{\mathtt{Id}-\mathrm{weak}})$$

$$\frac{t^\sharp = t_0^\sharp *_{\mathtt{T}} t_1^\sharp \qquad [h_{i,0}^\sharp *_{\mathtt{S}} h_{i,1}^\sharp \dashrightarrow h_{o,0}^\sharp *_{\mathtt{S}} h_{o,1}^\sharp]_{t^\sharp} *_{\mathtt{R}} r_0^\sharp \sqcup_{\mathbb{R}^\sharp} r_1^\sharp \rightsquigarrow r^\sharp}{[h_{i,0}^\sharp \dashrightarrow h_{o,0}^\sharp]_{t_0^\sharp} *_{\mathtt{R}} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]_{t_1^\sharp} *_{\mathtt{R}} r_0^\sharp \sqcup_{\mathbb{R}^\sharp} r_1^\sharp \rightsquigarrow r^\sharp} (\sqcup_{\dashrightarrow-\mathrm{weak}})$$

$$\frac{t_0^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(h_0^\sharp) \qquad [h_0^\sharp \dashrightarrow h_0^\sharp]_{t_0^\sharp} *_{\mathtt{R}} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]_{t_1^\sharp} *_{\mathtt{R}} r_0^\sharp \sqcup_{\mathbb{R}^\sharp} r_1^\sharp \rightsquigarrow r^\sharp}{\mathtt{Id}(h_0^\sharp) *_{\mathtt{R}} [h_{i,1}^\sharp \dashrightarrow h_{o,1}^\sharp]_{t_1^\sharp} *_{\mathtt{R}} r_0^\sharp \sqcup_{\mathbb{R}^\sharp} r_1^\sharp \rightsquigarrow r^\sharp} (\sqcup_{\dashrightarrow-\mathrm{intro}})$$

Figure 6.3: Extension of join rewriting rules from Figure 5.5 (page 78)

Figure 6.3 presents the modified rules. Rule $(\sqcup_{\dashrightarrow})$ applies $\sqcup_{\mathbb{T}^\sharp}$ to the abstract heap transformation predicates of the two transform-into relations and attaches the rewrited abstract heap transformation to the rewrited transform-into relation. Rules $(\sqcup_{\mathtt{Id}-\mathrm{weak}})$, $(\sqcup_{\dashrightarrow-\mathrm{weak}})$ and $(\sqcup_{\dashrightarrow-\mathrm{intro}})$ are all directly inspired from Theorem 6.1

Any abstract heap transformation predicates domain should provide a function $\mathbf{join}_{\mathbb{T}^\sharp_\in}(\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ that implements the operator $\sqcup_{\mathbb{T}^\sharp}$. It should satisfy the following assumption.

Assumption 6.8 (Soundness of $\mathbf{join}_{\mathbb{T}^\sharp}$). *Let* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $t_0^\sharp, t_1^\sharp \in \mathbb{T}^\sharp$*, if* $\mathbf{join}_{\mathbb{T}^\sharp}((\Psi_0, \Psi_1), t_0^\sharp, t_1^\sharp) = t^\sharp$*, then:*

$$(h_i, h_o, \Psi_0 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(t_0^\sharp) \vee (h_i, h_o, \Psi_1 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(t_1^\sharp) \quad \Longrightarrow \quad (h_i, h_o, \nu) \in \gamma_{\mathbb{T}^\sharp}(t^\sharp)$$

Definition 6.20 (Join for abstract heap transformation predicates domains). *We define the join function* $\mathbf{join}_{\mathbb{T}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ *for each abstract heap transformation predicates domain defined respectively in Section 6.3, Section 6.4 and Section 6.5.*

1. *The footprint predicates domain,* $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$*: the definition of* $\mathbf{join}_{\mathbb{T}^\sharp}(\Phi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp) = \mathrm{t}^\sharp$ *is given by the following table (each line for* $\mathrm{t}_0^\sharp$ *and each column for* $\mathrm{t}_1^\sharp$*).*

$\mathrm{t}^\sharp$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$=^\sharp$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$\subseteq^\sharp$	$\subseteq^\sharp$	$\subseteq^\sharp$	$\top$	$\top$
$\supseteq^\sharp$	$\supseteq^\sharp$	$\top$	$\supseteq^\sharp$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

2. *The fields predicates domain,* $\mathbb{T}^\sharp = \mathcal{P}(\mathbb{F})$*:*

$$\mathbf{join}_{\mathbb{T}^\sharp}(\Phi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp) = \mathrm{t}_0^\sharp \cup \mathrm{t}_1^\sharp$$

3. *The combined predicates domain,* $\mathbb{T}^\sharp = \mathbb{T}_a^\sharp \times \mathbb{T}_b^\sharp$*:*

$$\mathbf{join}_{\mathbb{T}^\sharp}(\Phi, (\mathrm{t}_{a,0}^\sharp, \mathrm{t}_{b,0}^\sharp), (\mathrm{t}_{a,1}^\sharp, \mathrm{t}_{b,1}^\sharp)) = (\mathbf{join}_{\mathbb{T}_a^\sharp}(\Phi, \mathrm{t}_{a,0}^\sharp, \mathrm{t}_{a,1}^\sharp), \mathbf{join}_{\mathbb{T}_b^\sharp}(\Phi, \mathrm{t}_{b,0}^\sharp, \mathrm{t}_{b,1}^\sharp))$$

Theorem 6.14 (Soundness of Definition 6.20). *The operators from Definition 6.20 are sound in the sense of Assumption 6.8.*

Widening

In Section 5.8, we saw that the operators $\sqcup_{\mathbb{H}^\sharp}$ and $\sqcup_{\mathbb{R}^\sharp}$ can be used as widening operators. However, the operator $\sqcup_{\mathbb{T}^\sharp}$ cannot be used as a widening operator: it may not converge in a finite number of steps. Indeed, the used abstract heap transformation predicates domain $\mathbb{T}^\sharp$ may denote an infinite set. The solution is to define a converging widening operator $\triangledown_{\mathbb{T}^\sharp}$ for abstract heap transformation predicates. This operator is then implemented by a function $\mathbf{wid}_{\mathbb{T}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ that is assumed to ensure termination. Its soundness property is the same that the function $\mathbf{join}_{\mathbb{T}^\sharp}$, except that it also enforces termination.

Assumption 6.9 (Soundness of $\mathbf{wid}_{\mathbb{T}^\sharp}$). *Let* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp \in \mathbb{T}^\sharp$*, if* $\mathbf{wid}_{\mathbb{T}^\sharp}((\Psi_0, \Psi_1), \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp) = \mathrm{t}^\sharp$*, then:*

$$(\mathrm{h_i}, \mathrm{h_o}, \Psi_0 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_0^\sharp) \vee (\mathrm{h_i}, \mathrm{h_o}, \Psi_1 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_1^\sharp) \implies (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)$$

The function $\mathbf{wid}_{\mathbb{T}^\sharp}$ *also enforces termination.*

Definition 6.21 (Widening for abstract heap transformation predicates domains). *We define the widening function* $\mathbf{wid}_{\mathbb{T}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ *for each abstract heap transformation predicates domain defined respectively in Section 6.3, Section 6.4 and Section 6.5.*

1. *The footprint predicates domain,* $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$*:*

$$\mathbf{wid}_{\mathbb{T}^\sharp}(\Phi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp) = \mathbf{join}_{\mathbb{T}^\sharp}(\Phi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp)$$

2. *The fields predicates domain,* $\mathbb{T}^\sharp = \mathcal{P}(\mathbb{F})$*:*

$$\mathbf{wid}_{\mathbb{T}^\sharp}(\Phi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp) = \mathbf{join}_{\mathbb{T}^\sharp}(\Phi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp)$$

3. *The combined predicates domain,* $\mathbb{T}^\sharp = \mathbb{T}_a^\sharp \times \mathbb{T}_b^\sharp$*:*

$$\mathbf{wid}_{\mathbb{T}^\sharp}(\Phi, (\mathrm{t}_{a,0}^\sharp, \mathrm{t}_{b,0}^\sharp), (\mathrm{t}_{a,1}^\sharp, \mathrm{t}_{b,1}^\sharp)) = (\mathbf{wid}_{\mathbb{T}_a^\sharp}(\Phi, \mathrm{t}_{a,0}^\sharp, \mathrm{t}_{a,1}^\sharp), \mathbf{wid}_{\mathbb{T}_b^\sharp}(\Phi, \mathrm{t}_{b,0}^\sharp, \mathrm{t}_{b,1}^\sharp))$$

Theorem 6.15 (Soundness of Definition 6.21). *The operators from Definition 6.21 are sound in the sense of Assumption 6.9.*

Regarding to the footprint and the fields predicates domains, the function $\mathbf{wid}_{\mathbb{T}^\sharp}$ is defined similary as the function $\mathbf{join}_{\mathbb{T}^\sharp}$. These definitions are valid because the footprint and the fields predicate domains are both finite. On the other hand, the combined predicates domain applies recursively the widening of its two sub-predicates domains.

6.7 Implementation and Experimental Evaluation

In this section, we evaluate whether abstract heap transformation predicates improve the logical strength of the relational analysis for the functions analyzed in Section 5.10. Then, we verify if our enriched relational analysis is able to express similar properties than other approach of program verification: we study the case of the analysis of a list module of the operating system Contiki [DGV04].

Structure	Function	Time (in s)		Logical Strength
		Rel.	Rel.+	Rel. vs Rel.+
singly linked list	`allocation`	0.77	0.78	=
singly linked list	`deallocation`	0.80	0.79	=
singly linked list	`traversal`	0.79	0.77	=
singly linked list	`head_insertion`	0.43	0.44	=
singly linked list	`insert` (Figure 1.3)	1.92	1.93	=
singly linked list	`reverse`	1.01	1.06	<
singly linked list	`map`	0.92	0.91	<
singly linked list	`tail`	0.55	0.54	=
singly linked list	`nth`	1.17	1.15	=
singly linked list	`partition`	4.85	4.93	<
singly linked list	`append`	1.60	1.59	=
singly linked list	`contains`	1.22	1.24	=
singly linked list	`deep_copy`	2.08	2.16	=
singly linked list	`sort` (Figure 2.3)	21.95	22.16	<
singly linked list	`filter`	2.70	2.79	<
binary search tree	`allocation`	1.11	1.45	=
singly linked list	`search`	1.63	1.67	=
singly linked list	`insert`	6.10	6.22	=

Table 6.1: Experiment results (sll: singly linked lists; bst: binary search trees; time in seconds over 1000 runs on a laptop with Intel Core i5 running at 2.4 GHz, with 4 Gb RAM, for basic and enriched relational analyses; the last column compares the expressiveness of the inferred result of each analysis). These functions are the same than in Table 5.1, the times for the basic relational analysis are also the same than in this table.

6.7.1 A Standard Library of Lists and Trees

In this section, we re-analyze the functions in Section 5.10 with abstract heap transformation predicates. The goal of this analysis is to evaluate whether if abstract heap transformation predicates improve in practice the logical strength of the relational analysis. In this section, we name the relational analysis without abstract heap transformation predicates the *basic relational analysis* and the relational analysis with these predicates the *enriched relational analysis*. The results of the evaluation are given in Table 6.1. The word 'Rel' still denotes the basic relational analysis whereas 'Rel+' denotes the enriched relational analysis.

The used abstract heap transformation predicates domain is the combined predicate domain (Section 6.5), that combines the footprint (Section 6.3) and the fields (Section 6.4) domains.

We observe that for all the functions for which the basic relational analysis did not infer stronger properties than the state analysis (`reverse`, `map`, `partition`, `sort` and `filter`), the enriched relational analysis inferred stronger properties than the basic relational analysis (and thus also the state analysis).

For instance, for the `map` function, the inferred abstract heap relation by the basic relational analysis was:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\alpha)]$$

The enriched relational analysis added to this abstract heap relation the abstract transformation predicate $(=^\sharp, \{\texttt{data}\})$, that expresses that the footprint of the input and output lists are the same and that only the `data` fields may have been modified. Regarding to the functions `reverse`, `sort` and `filter`, for which the basic relational analysis inferred the same abstract heap relation:

$$[\mathbf{listseg}(\alpha_0, \alpha_1) \dashrightarrow \mathbf{list}(\alpha_2)]$$

the enriched relational analysis inferred the same abstract heap transformation predicates for `reverse` and `sort`, the predicate $(=^\sharp, \{\texttt{next}\})$ that expresses that the output list is a permutation in place of the input list. For the function `filter`, the enriched relational analysis inferred the abstract heap transformation predicate $(\supseteq^\sharp, \{\texttt{next}\})$, which means that some deallocations may occur and that no `data` field has been modified. Finally, for the function `partition`, the basic relational analysis inferred the following abstract heap relation:

$$[\mathbf{list}(\alpha) \dashrightarrow \mathbf{list}(\beta_1) *_{\mathrm{s}} \mathbf{list}(\beta_2)]$$

that only indicates the presence of two well formed linked lists in the output state. The enriched relational analysis inferred for this abstract heap relation the predicate $(=^\sharp, \{\texttt{next}\})$ that indicates that these two output lists are composed by the elements of the input list, and that the `data` fields of the latter have not been modified.

For the other cases, the enriched relational analysis often does not infer more information. The main reason is that the inferred relation already describes a very precise relation.

6.7.2 The List Module of The Operating System Contiki

In this section, we evaluate the ability of our relational analysis to infer similar properties than a less automatic but more precise approach of program verification. We compare our approach with the one of Blanchard et al. [BKL18] for the verification of the linked list module of the operating system Contiki [DGV04]. Their work performs a deductive verification of this list module and is based on a parallel view of a linked list via a companion ghost array. This approach requires the user to specify both the pre and post conditions of each function, to annotate the source code with many invariants (mostly loop invariants), to write ghost functions and to prove different lemmas using SMT solvers

Function	Similar properties	Num. of pre-conditions	times (in ms)
`list_add`	yes	2	211 + 207
`list_chop`	yes	1	204
`list_copy`	yes	1	207
`list_head`	yes	1	201
`list_init`	yes	1	203
`list_insert`	yes	3	208 + 203 + 204
`list_item_next`	yes	1	202
`list_length`	yes	1	203
`list_pop`	yes	1	200
`list_push`	yes	2	208 + 204
`list_remove`	yes	2	201 + 202
`list_tail`	yes	1	201

Table 6.2: Experiment results for the linked list module of the operating system Contiki. It Indicates if our relational analysis proved similar properties than Blanchard et al.'s one. The third column indicates with how many different pre-conditions we run the analysis of the function. The last column indicates the execution times (in ms) from each pre-condition of the function

or the Coq proof assistant. In total, for about 176 lines of C code in the list module, they wrote 46 lines of for ghost functions and about 1400 lines of annotations. Their verification has generated 798 goals to prove. Among these goals, 770 have been proven automatically by SMT solvers, 4 interactively and 24 proven using the Coq proof assistant.

While this approach is less automatic than ours, it has the advantage to prove formally programs. Thus, we aimed to check if our relational abstract domain was able to express similar properties but *more automatically*. We have analyzed all the functions given in their paper, using the source codes that they provide. These functions are listed in Table 6.2. It shows that for all of the functions, our analysis proved similar properties as Blanchard et al. We did not analyze the files that manipulate arrays such as `array_pop.c`, as our relational abstract domain does not require ghost array companions.

An important feature in this list module is that each element of a list has to be unique. So if a function adds an element into a list, and if this element is already in this list, the element first has to be removed from the list and then added at the desired position. This is why all the functions that add an element into a list (`list_push`, `list_add`) call the function `list_remove`. This latter removes the given list element from the input list if it is inside, or leaves the list unchanged otherwise. For these functions we run our analysis twice with these 2 different preconditions: when the input list contains the given element and when it does not contain it. We could have to run the analysis once with only one pre-condition consisting of a disjunction of these to previous pre-conditions, but

these disjuncts may have been joined during a widening. This would have begotten a too hight loss of precision.

The function `list_insert` inserts a given list element into a list at a given position. We analyzed this function with three different pre-conditions: when the element to insert is not in the list, when the element is in the list but before the position it is supposed to be inserted, and when the element is in the list but after the desired position insertion. This function actually contains a bug. Indeed, if the element is already in the list (no matter before or after the position of insertion), the function adds directly the element at the given position without removing it from the list. This breaks the structure of the list. Like Blanchard et al., we found this bug with our analysis.

For the function `list_length`, our analysis did not prove that the returned integer is indeed the length of the input list, whereas the work of Blanchard et al. does. Our relational abstract domain does not express this kind of properties.

Chapter 7

Abstract Composition

Abstract memory relations are binary relations over input and output memory states: they are thus composable. Composing two abstract memory relations produces a new abstract memory relation that describes a relation between the input memory states of the first one and the output memory states of the second one. This brings the advantage to make a compositional analysis, where subprograms are analyzed independently and then composed, without loosing too much precision. In this chapter, we define an operator that performs such composition for abstract memory relations. This operator mainly relies on the intersection of abstract memory states.

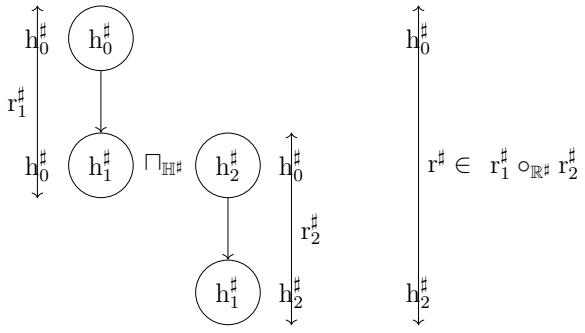


Figure 7.1: Graphical representation of the abstract composition

7.1 Introduction

In mathematics, if $R_1 \subseteq X \times Y$ and $R_2 \subseteq Y \times Z$ are two binary relations, the composition of R_1 and R_2 is defined by:

$$R_1 \circ R_2 = \{(x, z) \in X \times Z \mid \exists y \in Y : (x, y) \in R_1 \wedge (y, z) \in R_2\}$$

We observe that $R_1 \circ R_2$ is defined by the rule that says $(x, z) \in R_1 \circ R_2$ if and only if there is an element $y \in Y$ such that $(x, y) \in R_1$ and $(y, z) \in R_2$.

Abstract memory relations describe binary relations over concrete memory states, they are consequently composable. Concretely, if $\mathrm{m}^\sharp_{\mathcal{R}^1}$ and $\mathrm{m}^\sharp_{\mathcal{R}^2}$ are two abstract memory relations, the composition of the concrete relations described by $\mathrm{m}^\sharp_{\mathcal{R}^1}$ and the concrete relations described by $\mathrm{m}^\sharp_{\mathcal{R}^2}$ corresponds to $\gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^1}) \circ \gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^2})$ and is defined as follows:

$$\{(\mathrm{m}_1, \mathrm{m}_2) \mid \exists \mathrm{m} \in \mathbb{M} : (\mathrm{m}_1, \mathrm{m}) \in \gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^1}) \wedge (\mathrm{m}, \mathrm{m}_2) \in \gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^2})\}$$

In this chapter, we define an over-approximation of $\gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^1}) \circ \gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^2})$. This brings the advantage to make a *compositional analysis*. When abstract memory relations represent program fragments, these fragments are composed instead of re-analyzing them, which is an advantage for scalability. To define an over-approximation of composition, it requires to compute an over-approximation of the set of the common concrete memory states m of the two abstract memory relations. Naturally, this implies to define the *intersection* of abstract memory states.

Figure 7.1 represents graphically the abstract composition of two abstract heap relations $\mathrm{r}^\sharp_1$ and $\mathrm{r}^\sharp_2$. The abstract heaps $\mathrm{h}^\sharp_1$ and $\mathrm{h}^\sharp_2$ are respectively the output abstract heap of $\mathrm{r}^\sharp_1$ and the input abstract heap of $\mathrm{r}^\sharp_2$. First, the abstract composition $\mathrm{r}^\sharp_1 \circ_{\mathbb{R}^\sharp} \mathrm{r}^\sharp_2$ performs the abstract intersection $\sqcap_{\mathbb{H}^\sharp}$ between $\mathrm{h}^\sharp_1$ and $\mathrm{h}^\sharp_2$ to compute an over-approximation of common heaps between $\mathrm{r}^\sharp_1$ and $\mathrm{r}^\sharp_2$, on the form of a *finite set* of abstract heaps. Then, for all these common abstract heaps, it deduces a finite set of abstract heap relations describing relations between the input abstract heaps of $\mathrm{r}^\sharp_1$ and the output abstract heaps of $\mathrm{r}^\sharp_2$.

This chapter is organized as follows: Section 7.2 defines the intersection of abstract memory states, Section 7.3 presents the composition of abstract heap transformation predicates, that are necessary to define the composition of abstract memory relations in Section 7.4. Finally, Section 7.5 discusses related works.

7.2 Intersection of Abstract Memory States

We define the abstract operator $\mathbf{inter}_{\mathbb{M}^\sharp}$ that computes an over-approximation of the intersection of two abstract memory states. This operation follows the same principles that the abstract join operator, except that it outputs a *finite set of abstract memory*

$$\frac{}{\mathbf{emp} \sqcap_{\mathbb{H}^\sharp} \mathbf{emp} \rightsquigarrow \{\mathbf{emp}\}} \ (\sqcap_{\mathbf{emp}})$$

$$\frac{\mathrm{h}^\sharp_{l,0} \sqcap_{\mathbb{H}^\sharp} \mathrm{h}^\sharp_{r,0} \rightsquigarrow \mathrm{H}^\sharp_0 \quad \mathrm{h}^\sharp_{l,1} \sqcap_{\mathbb{H}^\sharp} \mathrm{h}^\sharp_{r,1} \rightsquigarrow \mathrm{H}^\sharp_1}{\mathrm{h}^\sharp_{l,0} *_{\mathtt{s}} \mathrm{h}^\sharp_{l,1} \sqcap_{\mathbb{H}^\sharp} \mathrm{h}^\sharp_{r,0} *_{\mathtt{s}} \mathrm{h}^\sharp_{r,1} \rightsquigarrow \{\mathrm{h}^\sharp_0 *_{\mathtt{s}} \mathrm{h}^\sharp_1 \mid \mathrm{h}^\sharp_0 \in \mathrm{H}^\sharp_0 \wedge \mathrm{h}^\sharp_1 \in \mathrm{H}^\sharp_1\}} \ (\sqcap_{*_{\mathtt{s}}})$$

$$\frac{\Phi(\alpha_l, \alpha_r) = \alpha \qquad \underline{\Phi(\beta_l, \beta_r) = \beta}}{\alpha_l \cdot \mathtt{f} \mapsto \beta_l \sqcap_{\mathbb{H}^\sharp} \alpha_r \cdot \mathtt{f} \mapsto \beta_r \rightsquigarrow \{\alpha \cdot \mathtt{f} \mapsto \beta\}} \ (\sqcap_{\mathrm{pt}})$$

$$\frac{\Phi(\alpha_l, \alpha_r) = \alpha}{\mathbf{list}(\alpha_l) \sqcap_{\mathbb{H}^\sharp} \mathbf{list}(\alpha_r) \rightsquigarrow \{\mathbf{list}(\alpha)\}} \ (\sqcap_{\mathrm{ind}})$$

$$\frac{\Phi(\alpha_l, \alpha_r) = \alpha \qquad \mathbf{list}(\beta_l) \sqcap_{\mathbb{H}^\sharp} \mathrm{h}^\sharp_r \rightsquigarrow \mathrm{H}^\sharp \qquad \underline{\Phi(\beta_l, \beta_r) = \beta} \qquad (\beta_l \text{ fresh})}{\mathbf{list}(\alpha_l) \sqcap_{\mathbb{H}^\sharp} \mathbf{listseg}(\alpha_r, \beta_r) *_{\mathtt{s}} \mathrm{h}^\sharp_r \rightsquigarrow \{\mathbf{listseg}(\alpha, \beta) *_{\mathtt{s}} \mathrm{h}^\sharp \mid \mathrm{h}^\sharp \in \mathrm{H}^\sharp\}} \ (\sqcap_{\mathrm{indseg}})$$

$$\frac{\Phi(\alpha_l, \alpha_r) = \alpha \quad \Phi(\beta_l, \beta_r) = \beta}{\mathbf{listseg}(\alpha_l, \beta_l) \sqcap_{\mathbb{H}^\sharp} \mathbf{listseg}(\alpha_r, \beta_r) \rightsquigarrow \{\mathbf{listseg}(\alpha, \beta)\}} \ (\sqcap_{\mathrm{segseg}})$$

$$\frac{\Phi(\alpha_l, \alpha_r) = \alpha \quad \mathbf{listseg}(\alpha'_l, \beta_l) \sqcap_{\mathbb{H}^\sharp} \mathrm{h}^\sharp_r \rightsquigarrow \mathrm{H}^\sharp \quad (\alpha'_l \text{ fresh}) \quad \underline{\Phi(\alpha'_l, \beta_r) = \alpha'}}{\mathbf{listseg}(\alpha_l, \beta_l) \sqcap_{\mathbb{H}^\sharp} \mathbf{listseg}(\alpha_r, \beta_r) *_{\mathtt{s}} \mathrm{h}^\sharp_r \rightsquigarrow \{\mathbf{listseg}(\alpha, \alpha') *_{\mathtt{s}} \mathrm{h}^\sharp \mid \mathrm{h}^\sharp \in \mathrm{H}^\sharp\}} \ (\sqcap_{\mathrm{seg}})$$

$$\frac{\begin{array}{c}\Phi(\alpha_l, \alpha_r) = \alpha \\ \text{there is no inductive or segment predicate attached to } \alpha_r \text{ in } \mathrm{h}^\sharp_r \\ \mathrm{H}^\sharp_u = \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha_l, \mathbf{list}(\alpha_l) *_{\mathtt{s}} \mathrm{h}^\sharp_l) \\ \mathrm{H}^\sharp = \{\mathrm{h}^\sharp \mid \forall \mathrm{h}^\sharp_u \in \mathrm{H}^\sharp_u,\ \mathrm{h}^\sharp_u \sqcap_{\mathbb{H}^\sharp} \mathrm{h}^\sharp_r \rightsquigarrow \mathrm{H}^\sharp_0 \wedge \mathrm{h}^\sharp \in \mathrm{H}^\sharp_0\}\end{array}}{\mathbf{list}(\alpha_l) *_{\mathtt{s}} \mathrm{h}^\sharp_l \sqcap_{\mathbb{H}^\sharp} \mathrm{h}^\sharp_r \rightsquigarrow \mathrm{H}^\sharp} \ (\sqcap_{\mathrm{u-ind}})$$

$$\frac{\begin{array}{c}\Phi(\alpha_l, \alpha_r) = \alpha \\ \text{there is no inductive or segment predicate attached to } \alpha_r \text{ in } \mathrm{h}^\sharp_r \\ \mathrm{H}^\sharp_u = \mathbf{unfold}_{\mathbb{H}^\sharp}(\alpha_l, \mathbf{listseg}(\alpha_l, \beta_l) *_{\mathtt{s}} \mathrm{h}^\sharp_l) \\ \mathrm{H}^\sharp = \{\mathrm{h}^\sharp \mid \forall \mathrm{h}^\sharp_u \in \mathrm{H}^\sharp_u,\ \mathrm{h}^\sharp_u \sqcap_{\mathbb{H}^\sharp} \mathrm{h}^\sharp_r \rightsquigarrow \mathrm{H}^\sharp_0 \wedge \mathrm{h}^\sharp \in \mathrm{H}^\sharp_0\}\end{array}}{\mathbf{listseg}(\alpha_l, \beta_l) *_{\mathtt{s}} \mathrm{h}^\sharp_l \sqcap_{\mathbb{H}^\sharp} \mathrm{h}^\sharp_r \rightsquigarrow \mathrm{H}^\sharp} \ (\sqcap_{\mathrm{u-seg}})$$

Figure 7.2: Intersection rewriting rules

states. It returns a finite set because it may require to unfold inductive or segment predicates, that also returns a finite set. The intersection of the concretizations of the two intersected abstract memory states should be included in the union of the concretizations of the resulting abstract memory states.

Similarly to abstract join, abstract intersection creates new abstract memory relations. This implies to create new symbolic values. Consequently, the abstract intersection also requires a pair of renaming functions $\Phi = (\Psi_0, \Psi_1)$, that map each output symbolic value with the pair of its two corresponding input symbolic values. If α is a resulting symbolic value of the intersection operator, we note $\Phi(\alpha_0, \alpha_1) = \alpha$ for $\Psi_0(\alpha) = \alpha_0$ and $\Psi_1(\alpha) = \alpha_1$. Concretely, the mapping $\Phi(\alpha_0, \alpha_1) = \alpha$ means that α_0 and α_1 correspond to the same concrete value, that is abstracted by α.

The abstract intersection algorithm follows the same three steps as the abstract join and the abstract inclusion: *initialization* that creates the initial pair of renaming functions, *abstract intersection of abstract heaps* that intersects two abstract heaps, and *abstract intersection* in the numerical abstract domain that intersects the two abstract numerical values.

Initialization. The abstract intersection operation starts with the initialization of the pair of renaming functions and the generation of the resulting abstract environment $\mathrm{e}^\sharp$ as follows: $\forall \mathtt{x} \in \mathbb{X}, \Phi_{\text{init}}(\mathrm{e}_0^\sharp(\mathtt{x}), \mathrm{e}_1^\sharp(\mathtt{x})) = \alpha$ and $\mathrm{e}^\sharp(\mathtt{x}) = \alpha$.

Abstract intersection of abstract heaps. The intersection of abstract heaps is performed by the function $\mathbf{inter}_{\mathbb{H}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{H}^\sharp \times \mathbb{H}^\sharp \to \mathcal{P}_{\text{fin}}((\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{H}^\sharp)$, that inputs a pair of renaming functions Φ and two abstract heaps, and returns a *finite set* of pairs of renaming functions (that are an extension of Φ) and of abstract heaps. This function implements the rewriting rules of Figure 7.2. This rules system is based on the operator $\sqcap_{\mathbb{H}^\sharp}$ that statisfies the property:

Theorem 7.1 (Soundness of $\sqcap_{\mathbb{H}^\sharp}$). *Let* $\mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp \in \mathbb{H}^\sharp$, $\mathrm{H}^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{H}^\sharp)$. *Then:*

$$\mathrm{h}_0^\sharp \sqcap_{\mathbb{H}^\sharp} \mathrm{h}_1^\sharp \rightsquigarrow \mathrm{H}^\sharp \quad \Longrightarrow \quad \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp) \cap \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp) \subseteq \bigcup \{\gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp) \mid \mathrm{h}^\sharp \in \mathrm{H}^\sharp\}$$

In this rules system, we indicate informally how the pairs of renaming function is extended, this is shown by underlined constraints on Φ such as $\underline{\Phi(\beta_l, \beta_r) = \beta}$. We also note $\mathrm{H}^\sharp$ for a finite set of abstract heaps ($\mathrm{H}^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{H}^\sharp)$). Rule $(\sqcap_{\text{pt}})$ is specific to points-to predicates. It is obvious that the intersection of two points-to predicates is a points-to predicate, modulo the renaming. Rules $(\sqcap_{\text{ind}})$ and $(\sqcap_{\text{segseg}})$ follow the same principle, but for respectively inductive and segment predicates. Rule $(\sqcap_{*_{\text{s}}})$ allows to intersect abstract heaps independently. Rules $(\sqcap_{\text{indseg}})$ and $(\sqcap_{\text{seg}})$ are directly inspired by respectively the inclusion rules $(\sqsubseteq_{\text{indseg}})$ and $(\sqsubseteq_{\text{seg}})$ of Figure 5.3 (page 74), except that they produce a

segment predicate. Rules ($\sqcap_{\text{u-seg}}$) and ($\sqcap_{\text{u-ind}}$) unfolds respectively segment and inductive predicates. These are the rules that may generate a set made of several abstract memory states in the result of the abstract intersection (one element per unfold path).

If one of these rules cannot be satisfied, the algorithm returns the empty set. For instance, for the intersection of a points predicate $\alpha_0 \cdot \texttt{data} \mapsto \alpha_1 *_{\texttt{s}} \alpha \cdot \texttt{next} \mapsto \alpha_2$ with a list inductive predicate $\mathbf{list}(\beta_0)$, the algorithm applies rule ($\sqcap_{\text{u-ind}}$) to unfold the inductive predicate into **emp** and $\beta_0 \cdot \texttt{data} \mapsto \beta_1 *_{\texttt{s}} \beta \cdot \texttt{next} \mapsto \beta_2 *_{\texttt{s}} \mathbf{list}(\beta_2)$. It then continues the intersection recursively with the two cases of the unfolding. The first case, that tries to intersect $\alpha_0 \cdot \texttt{data} \mapsto \alpha_1 *_{\texttt{s}} \alpha \cdot \texttt{next} \mapsto \alpha_2$ with **emp** fails, it thus returns the empty set. The second case succeeds: it applies rule ($\sqcap_{\text{pt}}$) for the `next` fields and the `data` fields, then applies ($\sqcap_{\text{u-ind}}$) on $\mathbf{list}(\beta_2)$ (this time, only the case where the predicate is unfolded to **emp** succeeds). Finally, the algorithm returns a singleton set of the form $\gamma_0 \cdot \texttt{data} \mapsto \gamma_1 *_{\texttt{s}} \gamma \cdot \texttt{next} \mapsto \gamma_2$.

Theorem 7.2 (Soundness of $\mathbf{inter}_{\mathbb{H}^\sharp}$). *Let* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp \in \mathbb{H}^\sharp$. $\forall (\mathrm{h}, \nu) \in (\mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$, $\forall \Psi_0', \Psi_1' \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$, *we have:*

$$\begin{aligned}&(\mathrm{h}, \Psi_0' \circ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_0^\sharp) \wedge (\mathrm{h}, \Psi_1' \circ \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}_1^\sharp)\\ &\quad \implies \forall \mathrm{h}^\sharp \text{ such that } ((\Psi_0', \Psi_1'), \mathrm{h}^\sharp) \in \mathbf{inter}_{\mathbb{H}^\sharp}((\Psi_0, \Psi_1), \mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp) : \quad (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)\end{aligned}$$

There is a very important property about the pair of renaming functions Φ. It allows to establish equality constraints between the symbolic values of the resulting abstract heap, and thus, to simplify this later abstract heap.

Theorem 7.3 (Property). *Let* $\alpha, \beta_1, \beta_2 \in \mathbb{V}^\sharp$, *we have:*

$$\Phi(\alpha, \beta_1) = \gamma_1 \wedge \Phi(\alpha, \beta_2) = \gamma_2 \quad \implies \quad \gamma_1 = \gamma_2$$

Remark that this property holds true for both sides (i.e. $\Phi(\alpha_1, \beta) = \gamma_1 \wedge \Phi(\alpha_2, \beta) = \gamma_2 \implies \gamma_1 = \gamma_2$). This allows to rename all occurrences of γ_1 by γ_2 (or vice versa) in the resulting abstract heap. Thus, after applying each rule, the $\mathbf{inter}_{\mathbb{H}^\sharp}$ proceeds to a *symbolic values substitution* phase using the equality constraints generated by Ψ.

Abstract intersection in the numerical abstract domain. Finally, the abstract intersection proceeds to the intersection in the numerical domain, using the renaming functions generated by $\mathbf{inter}_{\mathbb{H}^\sharp}$. This is performed by the function $\mathbf{inter}_{\mathbb{N}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{N}^\sharp \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$. It should satisfy the following soundness assumption.

Assumption 7.1 (Soundness of $\mathbf{inter}_{\mathbb{N}^\sharp}$). *Let* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp \in \mathbb{N}^\sharp$, *then,* $\forall \nu \in \mathbb{V}^\sharp \to \mathbb{V}$*:*

$$(\Psi_0 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_0^\sharp) \wedge (\Psi_1 \circ \nu) \in \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_1^\sharp) \quad \implies \quad \nu \in \gamma_{\mathbb{N}^\sharp}(\mathbf{inter}_{\mathbb{N}^\sharp}((\Psi_0, \Psi_1), \mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp))$$

Definition 7.1 (Intersection of abstract memory states). *Let* $\mathrm{m}_0^\sharp = (\mathrm{e}_0^\sharp, \mathrm{h}_0^\sharp, \mathrm{n}_0^\sharp)$ *and* $\mathrm{m}_1^\sharp = (\mathrm{e}_1^\sharp, \mathrm{h}_1^\sharp, \mathrm{n}_1^\sharp)$ *be two abstract memory states.*

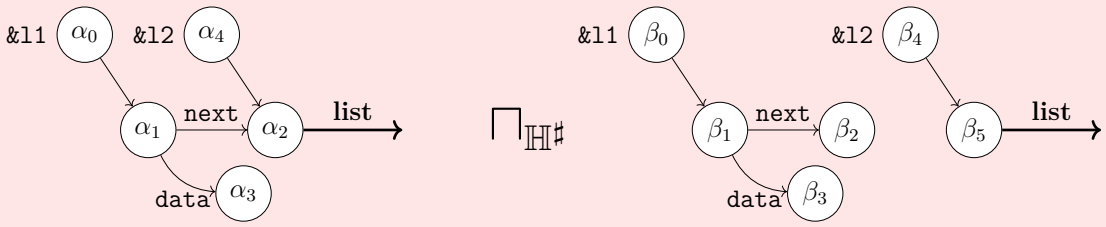
$$\forall \mathtt{x} \in \mathbb{X}, \Phi_{\text{init}}(\mathrm{e}_0^\sharp(\mathtt{x}), \mathrm{e}_1^\sharp(\mathtt{x})) = \alpha \text{ and } \mathrm{e}^\sharp(\mathtt{x}) = \alpha.$$

$$\mathbf{inter}_{\mathbb{M}^\sharp}(\mathrm{m}_0^\sharp, \mathrm{m}_1^\sharp) = \{(\mathrm{e}^\sharp, \mathrm{h}^\sharp, \mathbf{inter}_{\mathbb{N}^\sharp}(\Phi', \mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp)) \mid (\Phi', \mathrm{h}^\sharp) \in \mathbf{inter}_{\mathbb{H}^\sharp}(\Phi_{\text{init}}, \mathrm{h}_0^\sharp, \mathrm{h}_1^\sharp)\}$$

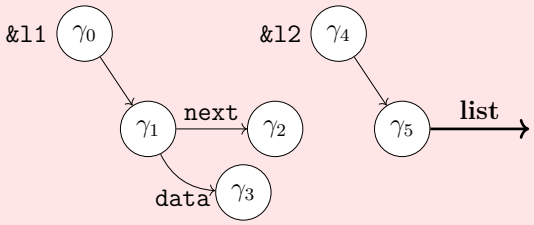
Theorem 7.4 (Soundness of $\mathbf{inter}_{\mathbb{M}^\sharp}$). *Let* $\mathrm{m}_1^\sharp, \mathrm{m}_2^\sharp \in \mathbb{M}^\sharp$*, then:*

$$\gamma_{\mathbb{M}}(\mathrm{m}_1^\sharp) \cap \gamma_{\mathbb{M}}(\mathrm{m}_2^\sharp) \subseteq \bigcup \{\gamma_{\mathbb{M}}(\mathrm{m}^\sharp) \mid \mathrm{m}^\sharp \in \mathbf{inter}_{\mathbb{M}^\sharp}(\mathrm{m}_1^\sharp, \mathrm{m}_2^\sharp))\}$$

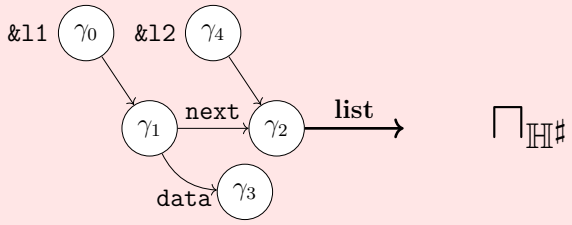
Example 7.1 (Intersection with symbolic value substitution). We discuss the result of the intersection of the following two abstract memory states.



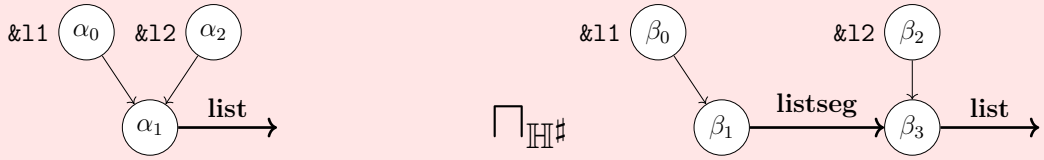
The algorithm starts by initializing the renaming functions, according to the variables `l1` and `l2`, $\Phi(\alpha_0, \beta_0) = \gamma_0$ and $\Phi(\alpha_4, \beta_4) = \gamma_4$. Then, the algorithm applies the rule $(\sqcap_{\text{pt}})$ respectively for $\Phi(\alpha_0, \beta_0) = \gamma_0$ and $\Phi(\alpha_4, \beta_4) = \gamma_4$. This extends the renaming function with $\Phi(\alpha_1, \beta_1) = \gamma_1$ and $\Phi(\alpha_2, \beta_5) = \gamma_5$, and produces the points to predicates $\gamma_0 \mapsto \gamma_1$ and $\gamma_4 \mapsto \gamma_5$. Finally, the algorithm applies respectively the rule $(\sqcap_{\text{pt}})$ for $\Phi(\alpha_1, \beta_1) = \gamma_1$ and the rule $(\sqcap_{\text{ind}})$ for $\Phi(\alpha_2, \beta_5) = \gamma_5$. This produces the following abstract memory state:



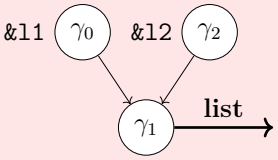
In the final renaming functions, we have $\Phi(\alpha_2, \beta_2) = \gamma_2 \wedge \Phi(\alpha_2, \beta_5) = \gamma_5$. According to Theorem 7.3, this means that $\gamma_2 = \gamma_5$. We can thus rename all occurrences of γ_5 by γ_2 and obtain:



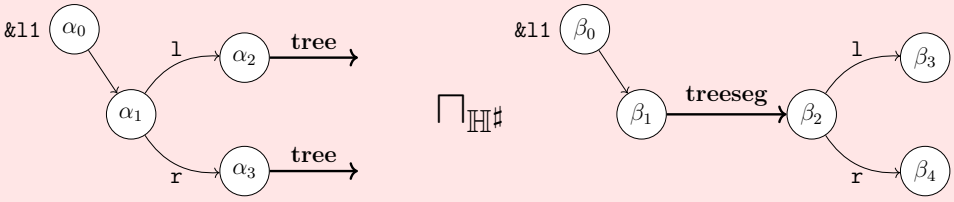
Example 7.2 (Intersection with unfolding). We consider the abstract intersection of the two abstract memory states.



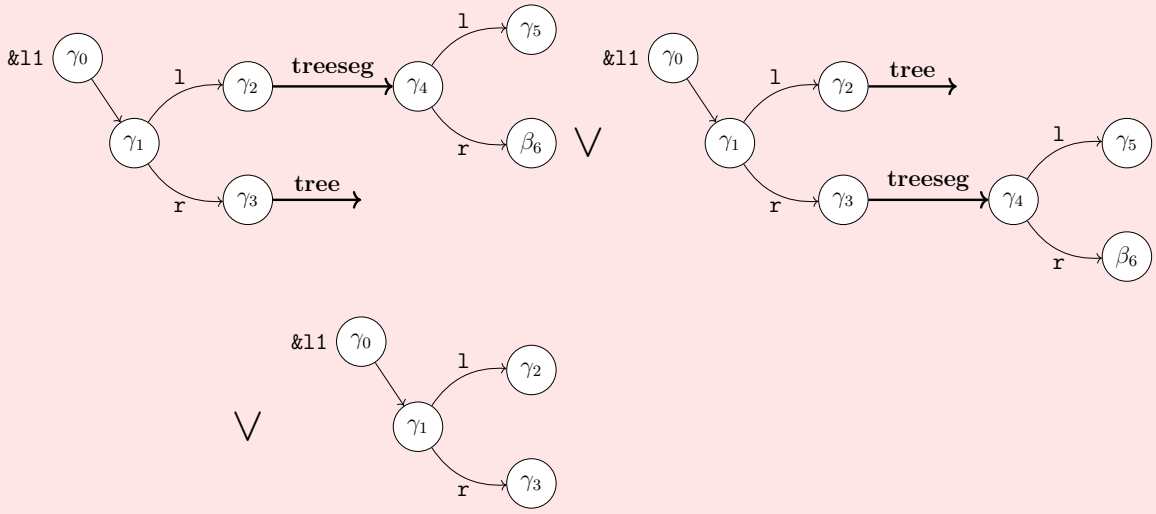
First, the algorithm applies rule $(\sqcap_{\text{pt}})$ for respectively $\Phi(\alpha_0, \beta_0) = \gamma_0$ and $\Phi(\alpha_2, \beta_2) = \gamma_2$. This extends Φ with $\Phi(\alpha_1, \beta_1) = \gamma_1$ and $\Phi(\alpha_1, \beta_3) = \gamma_3$. Then, rule $(\sqcap_{\text{ind}})$ is applied with the two inductive predicates and produces $\mathbf{list}(\gamma_3)$. As the inductive predicate $\mathbf{list}(\alpha_1)$ has been consumed by rule $(\sqcap_{\text{ind}})$, the algorithm applies rule $(\sqcap_{\text{u-seg}})$ between $\mathbf{emp}$ and $\mathbf{listseg}(\beta_1, \beta_3)$. This is clear the segment must be unfolded to the empty memory, and this means that $\beta_1 = \beta_3$, and by Theorem 7.3, that $\gamma_1 = \gamma_3$. Finally, the abstract intersection produces the following abstract memory state:



Example 7.3 (Intersection that produces a set of several abstract states). Until now, we saw examples of abstract intersection that produce only one result. We now show an example that produces a disjunction of abstract memory states. We thus consider the abstract of the two abstract memory states that denote both a tree of at least one node.



The abstract intersection of these two abstract memory states produces a disjunction of tree abstract memory states:



These three disjuncts are direct consequences of the definition of a tree segment predicate:

$$\begin{aligned}\mathbf{treeseg}(\alpha_0, \alpha_1) \quad &:= (\mathbf{emp}, \alpha_0 = \alpha_1) \\ &\vee (\alpha_0 \cdot \mathtt{r} \mapsto \alpha_2 *_\mathtt{s} \mathbf{treeseg}(\alpha_2, \alpha_1) *_\mathtt{s} \alpha_0 \cdot \mathtt{l} \mapsto \alpha_3 *_\mathtt{s} \mathbf{tree}(\alpha_3), \alpha \neq 0) \\ &\vee (\alpha_0 \cdot \mathtt{r} \mapsto \alpha_2 *_\mathtt{s} \mathbf{tree}(\alpha_2) *_\mathtt{s} \alpha_0 \cdot \mathtt{l} \mapsto \alpha_3 *_\mathtt{s} \mathbf{treeseg}(\alpha_3, \alpha_1), \alpha \neq 0)\end{aligned}$$

7.3 Composition of Abstract Heap Transformation Predicates

Abstract heap transformation predicates describe specific relations over the two abstract heaps of transform-into relations. It is necessary to compose them in order to compose abstract heap relations. We thus define the function $\mathbf{comp}_{\mathbb{T}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ that performs the composition of two abstract heap transformation predicates. Similarly to abstract intersection, this function inputs a pair of renaming functions Φ that maps the symbolic values that should correspond to the same concrete value. The function $\mathbf{comp}_{\mathbb{T}^\sharp}$ should satisfy the following assumption.

Assumption 7.2 (Soundness of $\mathbf{comp}_{\mathbb{T}^\sharp}$). *Let* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp \in \mathbb{T}^\sharp$, $\mathbf{comp}_{\mathbb{T}^\sharp}((\Psi_0, \Psi_1), \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp) = \mathrm{t}^\sharp$, *we have:*

$$\{(\mathrm{h}_0, \mathrm{h}_1, \nu) \mid \exists \mathrm{h}, (\mathrm{h}_0, \mathrm{h}, \Psi_0 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_0^\sharp) \wedge (\mathrm{h}, \mathrm{h}_1, \Psi_1 \circ \nu) \in \gamma_{\mathbb{T}^\sharp}(\mathrm{t}_1^\sharp)\} \subseteq \gamma_{\mathbb{T}^\sharp}(\mathrm{t}^\sharp)$$

Compositing abstract heap transformation predicates can infer interesting information. For instance, for a predicate `rev` that can express the reversal of a linked list, the composition of `rev` with `rev` can infer a predicate `id` that expresses the identity relation (this means that reversing a list twice restores the original list).

We now give the definition of $\mathbf{comp}_{\mathbb{T}^\sharp}$ for the abstract heap transformation predicates domains of Chapter 6.

Definition 7.2 (Composition for abstract heap transformation predicates domains). *We define the function* $\mathbf{comp}_{\mathbb{T}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{T}^\sharp \times \mathbb{T}^\sharp \to \mathbb{T}^\sharp$ *for each abstract heap transformation predicates domain defined respectively in Section 6.3, Section 6.4 and Section 6.5.*

1. *The footprint predicates domain,* $\mathbb{T}^\sharp = \{=^\sharp, \subseteq^\sharp, \supseteq^\sharp, \top\}$*: the definition of* $\mathbf{comp}_{\mathbb{T}^\sharp}(\Psi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp)$ *is given by the following table (each line for* $\mathrm{t}_0^\sharp$ *and each column for* $\mathrm{t}_1^\sharp$*).*

$\mathrm{t}^\sharp$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$=^\sharp$	$=^\sharp$	$\subseteq^\sharp$	$\supseteq^\sharp$	$\top$
$\subseteq^\sharp$	$\subseteq^\sharp$	$\subseteq^\sharp$	$\top$	$\top$
$\supseteq^\sharp$	$\supseteq^\sharp$	$\top$	$\supseteq^\sharp$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

2. *The fields predicates domain,* $\mathbb{T}^\sharp = \mathcal{P}(\mathbb{F})$*:*

$$\mathbf{comp}_{\mathbb{T}^\sharp}(\Phi, \mathrm{t}_0^\sharp, \mathrm{t}_1^\sharp) = \mathrm{t}_0^\sharp \cup \mathrm{t}_1^\sharp$$

3. *The combined predicates domain,* $\mathbb{T}^\sharp = \mathbb{T}_a^\sharp \times \mathbb{T}_b^\sharp$*:*

$$\mathbf{comp}_{\mathbb{T}^\sharp}(\Phi, (\mathrm{t}_{a,0}^\sharp, \mathrm{t}_{b,0}^\sharp), (\mathrm{t}_{a,1}^\sharp, \mathrm{t}_{b,1}^\sharp)) = (\mathbf{comp}_{\mathbb{T}_\mathrm{a}^\sharp}(\Phi, \mathrm{t}_{a,0}^\sharp, \mathrm{t}_{a,1}^\sharp), \mathbf{comp}_{\mathbb{T}_\mathrm{b}^\sharp}(\Phi, \mathrm{t}_{b,0}^\sharp, \mathrm{t}_{b,1}^\sharp))$$

Regarding to the footprint and the fields predicates domains, the definition of $\mathbf{comp}_{\mathbb{T}^\sharp}$ is totally similar to the function $\mathbf{join}_{\mathbb{T}^\sharp}$, the over-approximation of the union of two abstract heap transformation predicates (page 109). Indeed, in these cases, join is the most precise operation that satisfies Assumption 7.2.

Theorem 7.5 (Soundness of Definition 7.2). *The operators from Definition 7.2 are sound in the sense of Assumption 7.2.*

To define the composition of abstract heap relations, the abstract composition implements a rules system, in the style of the one in Figure 6.3 (page 108). To design such a

rules system for the composition of abstract heap relations, it also requires a composition operator $\circ_{\mathbb{T}^\sharp}$ related abstract heap transformation predicates. It should satisfy the following soundness property:

Theorem 7.6 (Soundness of $\circ_{\mathbb{T}^\sharp}$). *Let* $t_0^\sharp, t_1^\sharp, t^\sharp \in \mathbb{T}^\sharp$. *Then:*

$$t_0^\sharp \circ_{\mathbb{T}^\sharp} t_1^\sharp \rightsquigarrow t^\sharp \implies \{(h_0, h_1, \nu) \mid \exists h, (h_0, h, \nu) \in \gamma_{\mathbb{T}^\sharp}(t_0^\sharp) \wedge (h, h_1, \nu) \in \gamma_{\mathbb{T}^\sharp}(t_1^\sharp)\} \subseteq \gamma_{\mathbb{T}^\sharp}(t^\sharp)$$

The function $\mathbf{comp}_{\mathbb{T}^\sharp}$ can be considered as an implementation of the operator $\circ_{\mathbb{T}^\sharp}$.

7.4 Composition of Abstract Memory Relations

We now define the composition of abstract memory relations $\mathbf{comp}_{\mathbb{M}^\sharp_{\mathcal{R}}} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{M}^\sharp_{\mathcal{R}} \times \mathbb{M}^\sharp_{\mathcal{R}} \to \mathcal{P}_{\text{fin}}(\mathbb{M}^\sharp_{\mathcal{R}})$ that is built upon the abstract intersection. Since the abstract intersection returns a finite set, the abstract composition also returns a set. Per each common abstract memory states of the two abstract memory relations, the abstract composition returns a corresponding abstract memory relations. The composition of the concretizations of the input abstract memory relations should be included in the union of the concretizations of the resulting abstract memory relations.

Like the abstract intersection, the abstract composition also manipulates pairs of renaming functions Φ, that map the symbolic values that should correspond to the same concrete value (note that this is the same meaning that for abstract intersection).

The abstract composition algorithm follows three main steps: *extension* that extends the initial pair of renaming functions Φ_{init} and creates the resulting abstract environment, *abstract composition of abstract heap relations* that composes two abstract heap relations, and *abstract composition of abstract numerical abstract domains* that intersects the two abstract numerical values and projects away the result.

7.4.1 Extension of the initial pair of renaming functions

While standard lattice operations such as join or inclusion proceed to an initialization step, abstract composition inputs an *initial* pair of renaming functions Φ_{init} and extends it according to the abstract environments. This initial pair is necessary, as some symbolic values of the given abstract memory relations may be *unreachable* from the variables of the abstract environments. This specificity is made fully explicit in Chapter 8 and is out of scope of this current Chapter.

The extension step simply extends Φ_{init} into an other pair of renaming functions Φ_{ext}. This first consists on creating a temporary pair of renaming functions Φ_{tmp}, and creating the resulting abstract environment $e^\sharp$, such that $\forall \mathtt{x} \in \mathbb{X}$, $\Phi_{\text{tmp}}(e_0^\sharp(\mathtt{x}), e_1^\sharp(\mathtt{x})) = \alpha$ and $e^\sharp(\mathtt{x}) = \alpha$. The extended pair of renaming functions Φ_{ext} is then obtained by merging Φ_{init} and Φ_{tmp} (we have $\Phi_{\text{ext}} = \Phi_{\text{init}} \uplus \Phi_{\text{tmp}}$).

7.4.2 Abstract composition of abstract heap relations

The composition of abstract heap relations is perfomed by the function $\mathbf{comp}_{\mathbb{R}^\sharp} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{R}^\sharp \times \mathbb{R}^\sharp \to \mathcal{P}_{\text{fin}}((\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2 \times \mathbb{R}^\sharp)$, that is built upon the intersection of abstract heap $\mathbf{inter}_{\mathbb{H}^\sharp}$ and the composition of abstract heap transformation predicates (both defined previously).

The Rules System. The function $\mathbf{comp}_{\mathbb{R}^\sharp}$ implements the rules system of Figure 7.3, that is based on a new operator $\circ_{\mathbb{R}^\sharp}$ that reason over abstract heap relations, and on the operators $\sqcap_{\mathbb{H}^\sharp}$ and $\circ_{\mathbb{T}^\sharp}$. The operator $\circ_{\mathbb{R}^\sharp}$ satisfies the property:

Theorem 7.7 (Soundness of $\circ_{\mathbb{R}^\sharp}$). *Let* $\mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp \in \mathbb{R}^\sharp$ *and* $\mathrm{R}^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{R}^\sharp)$*. Then:*

$$\mathrm{r}_0^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp \rightsquigarrow \mathrm{R}^\sharp \implies \{(\mathrm{h}_0, \mathrm{h}_1, \nu) \mid \exists \mathrm{h}, (\mathrm{h}_0, \mathrm{h}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_0^\sharp) \wedge (\mathrm{h}, \mathrm{h}_1, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp)\} \subseteq \bigcup\{\gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \mid \mathrm{r}^\sharp \in \mathrm{R}^\sharp\}$$

In this rules system, we note $\mathrm{H}^\sharp$ a finite set of abstract heaps and $\mathrm{R}^\sharp$ a finite set of abstract heap relations (we have $\mathrm{H}^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{H}^\sharp)$ and $\mathrm{R}^\sharp \in \mathcal{P}_{\text{fin}}(\mathbb{R}^\sharp)$). Rules $(\circ_{\mathtt{Id}})$, $(\circ_{\dashrightarrow})$, $(\circ_{\dashrightarrow \mathtt{-Id}})$ and $(\circ_{\mathtt{Id} \dashrightarrow})$ all follow the same principle: they intersect the output abstract heap of the left abstract heap relation with the input abstract heap of the right abstract heap relation. In the case of rule $(\circ_{\mathtt{Id}})$, the composition of two identity relations is the identity relation of the intersection between their respective abstract heaps. Rules $(\circ_{\dashrightarrow \mathtt{-Id}})$ and $(\circ_{\mathtt{Id} \dashrightarrow})$ are both a kind of mix between rule $(\circ_{\mathtt{Id}})$ and rule $(\circ_{\dashrightarrow})$. Rule $(\circ_{*_{\mathtt{R}}})$ allows to compose independently abstract heap relations. Rules $(\circ_{\text{intro}_{\text{L}}})$ and $(\circ_{\text{weak}_{\text{L}}})$ both rely on Theorem 6.1 (page 91). Finally, rule $(\circ_{\text{unfold}_{\text{L}}})$ is inspired by rules $(\sqcap_{\text{u-ind}})$ and $(\sqcap_{\text{u-seg}})$ of Figure 7.2 except that the unfolding is performed at abstract heap relations level instead of abstract heaps level, in order to propagate the information provided by the unfolding to the whole abstract heap relations. A consequence of this rule is that $(\sqcap_{\text{u-ind}})$ and $(\sqcap_{\text{u-seg}})$ should never be applied during the abstract composition. This also means that the operator $\sqcap_{\mathbb{H}^\sharp}$ in Figure 7.3 should only produce singleton or empty sets.

Implementation of the Rules System. We now detail the implementation of the function $\mathbf{comp}_{\mathbb{R}^\sharp}$. Similarly to inclusion checking or join, the function $\mathbf{comp}_{\mathbb{R}^\sharp}$ uses the pair of renaming functions Φ to detect which rule to applied. If $\Phi(\alpha_0, \alpha_1) = \alpha$, the algorithm extracts the abstract heap relations attached respectively to α_0 and α_1, and tries to apply the corresponding rule. Note that rules $(\circ_{\text{unfold}_{\text{L}}})$ and $(\circ_{\text{unfold}_{\text{R}}})$ should be applied in priority compared to the other rules, as they have constraints on the form of the abstract heaps. If the abstract intersection of two abstract heaps fails (returns the empty set), the abstract composition also fails (and also returns the empty set). Thus, the returned finite set contains only the abstract heap relations for which the abstract composition is valid. The pair of renaming functions Φ is extended by the abstract

$$\frac{\mathrm{h}_0^\sharp \sqcap_{\mathbb{H}^\sharp} \mathrm{h}_1^\sharp \rightsquigarrow \mathrm{H}^\sharp}{\mathtt{Id}(\mathrm{h}_0^\sharp) \circ_{\mathbb{R}^\sharp} \mathtt{Id}(\mathrm{h}_1^\sharp) \rightsquigarrow \{\mathtt{Id}(\mathrm{h}^\sharp) \mid \mathrm{h}^\sharp \in \mathrm{H}^\sharp\}} \ (\circ_{\mathtt{Id}})$$

$$\frac{\mathrm{t}_0^\sharp \circ_{\mathbb{T}^\sharp} \mathrm{t}_1^\sharp \rightsquigarrow \mathrm{t}^\sharp \qquad \mathrm{h}_{\mathrm{o},0}^\sharp \sqcap_{\mathbb{H}^\sharp} \mathrm{h}_{\mathrm{i},1}^\sharp \rightsquigarrow \mathrm{H}^\sharp \qquad \mathrm{H}^\sharp \neq \{\}}{[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp]_{\mathrm{t}_0^\sharp} \circ_{\mathbb{R}^\sharp} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}_1^\sharp} \rightsquigarrow \{[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp}\}} \ (\circ_{\dashrightarrow})$$

$$\frac{\mathrm{r}_{0,0}^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_{1,0}^\sharp \rightsquigarrow \mathrm{R}_0^\sharp \qquad \mathrm{r}_{0,1}^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_{1,1}^\sharp \rightsquigarrow \mathrm{R}_1^\sharp}{\mathrm{r}_{0,0}^\sharp *_\mathtt{R} \mathrm{r}_{0,1}^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_{1,0}^\sharp *_\mathtt{R} \mathrm{r}_{1,1}^\sharp \rightsquigarrow \{\mathrm{r}_0^\sharp *_\mathtt{R} \mathrm{r}_1^\sharp \mid \mathrm{r}_0^\sharp \in \mathrm{R}_0^\sharp \wedge \mathrm{r}_1^\sharp \in \mathrm{R}_1^\sharp\}} \ (\circ_{*_\mathtt{R}})$$

$$\frac{\mathrm{t}_1^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}_1^\sharp) \qquad \mathrm{t}_0^\sharp \circ_{\mathbb{T}^\sharp} \mathrm{t}_1^\sharp \rightsquigarrow \mathrm{t}^\sharp \qquad \mathrm{h}_{\mathrm{o},0}^\sharp \sqcap_{\mathbb{H}^\sharp} \mathrm{h}_1^\sharp \rightsquigarrow \mathrm{H}_\mathrm{o}^\sharp}{[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp]_{\mathrm{t}_0^\sharp} \circ_{\mathbb{R}^\sharp} \mathtt{Id}(\mathrm{h}_1^\sharp) \rightsquigarrow \{[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp} \mid \mathrm{h}_\mathrm{o}^\sharp \in \mathrm{H}_\mathrm{o}^\sharp\}} \ (\circ_{\dashrightarrow - \mathtt{Id}})$$

$$\frac{\mathrm{t}_0^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}_0^\sharp) \qquad \mathrm{t}_0^\sharp \circ_{\mathbb{T}^\sharp} \mathrm{t}_1^\sharp \rightsquigarrow \mathrm{t}^\sharp \qquad \mathrm{h}_0^\sharp \sqcap_{\mathbb{H}^\sharp} \mathrm{h}_{\mathrm{i},1}^\sharp \rightsquigarrow \mathrm{H}_\mathrm{i}^\sharp}{\mathtt{Id}(\mathrm{h}_0^\sharp) \circ_{\mathbb{R}^\sharp} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}_1^\sharp} \rightsquigarrow \{[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]_{\mathrm{t}^\sharp} \mid \mathrm{h}_\mathrm{i}^\sharp \in \mathrm{H}_\mathrm{i}^\sharp\}} \ (\circ_{\mathtt{Id} - \dashrightarrow})$$

$$\frac{\mathrm{t}_0^\sharp = \mathbf{id}_{\mathbb{T}^\sharp}(\mathrm{h}_0^\sharp) \qquad [\mathrm{h}_0^\sharp \dashrightarrow \mathrm{h}_0^\sharp]_{\mathrm{t}_0^\sharp} *_\mathtt{R} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]_{\mathrm{t}_1^\sharp} *_\mathtt{R} \mathrm{r}_0^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp \rightsquigarrow \mathrm{R}^\sharp}{\mathtt{Id}(\mathrm{h}_0^\sharp) *_\mathtt{R} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]_{\mathrm{t}_1^\sharp} *_\mathtt{R} \mathrm{r}_0^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp \rightsquigarrow \mathrm{R}^\sharp} \ (\circ_{\mathrm{intro_L}})$$

$$\frac{\mathrm{t}_2^\sharp = \mathrm{h}_0^\sharp *_\mathtt{T} \mathrm{h}_1^\sharp \qquad [\mathrm{h}_{\mathrm{i},0}^\sharp *_\mathtt{S} \mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp *_\mathtt{S} \mathrm{h}_{\mathrm{o},1}^\sharp]_{\mathrm{t}_2^\sharp} *_\mathtt{R} \mathrm{r}_0^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp \rightsquigarrow \mathrm{R}^\sharp}{[\mathrm{h}_{\mathrm{i},0}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},0}^\sharp]_{\mathrm{t}_0^\sharp} *_\mathtt{R} [\mathrm{h}_{\mathrm{i},1}^\sharp \dashrightarrow \mathrm{h}_{\mathrm{o},1}^\sharp]_{\mathrm{t}_1^\sharp} *_\mathtt{R} \mathrm{r}_0^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp \rightsquigarrow \mathrm{R}^\sharp} \ (\circ_{\mathrm{weak_L}})$$

$$\frac{\begin{array}{c}\Phi(\alpha_0, \alpha_1) = \alpha \quad \alpha_0 \text{ carries an inductive or a segment predicate in } \mathrm{r}_0^\sharp \\ \text{there is no inductive or segment segment predicate attached to } \alpha_1 \text{ in } \mathrm{r}_1^\sharp \\ \mathrm{R}_u^\sharp = \mathbf{unfold}_{\mathbb{R}^\sharp}(\alpha_0, \mathrm{r}_0^\sharp) \\ \mathrm{R}^\sharp = \{\mathrm{r}^\sharp \mid \forall \mathrm{r}_u^\sharp \in \mathrm{R}_u^\sharp, \mathrm{r}_u^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp \rightsquigarrow \mathrm{R}_0^\sharp \wedge \mathrm{r}^\sharp \in \mathrm{R}_0^\sharp\}\end{array}}{\mathrm{r}_0^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp \rightsquigarrow \mathrm{R}^\sharp} \ (\circ_{\mathrm{unfold_L}})$$

Figure 7.3: Composition rewriting rules. The rules ($\circ_{\mathrm{intro_R}}$), ($\circ_{\mathrm{weak_R}}$) and ($\circ_{\mathrm{unfold_R}}$) are not given, as they are respectively symmetric to the rules ($\circ_{\mathrm{intro_L}}$), ($\circ_{\mathrm{weak_L}}$) and ($\circ_{\mathrm{unfold_L}}$).

intersection. All symbolic values that appear in the "final" Φ should be renamed in the resulting abstract heap relations. To illustrate it, we consider the following abstract composition with $\Phi(\alpha_0, \beta_0) = \gamma_0$:

$$[\alpha_0 \cdot \mathtt{f} \mapsto \alpha_1 \dashrightarrow \alpha_0 \cdot \mathtt{f} \mapsto \alpha_2]_{\mathrm{t}_0^\sharp} \circ_{\mathbb{R}^\sharp} [\beta_0 \cdot \mathtt{f} \mapsto \beta_1 \dashrightarrow \beta_0 \cdot \mathtt{f} \mapsto \beta_2]_{\mathrm{t}_1^\sharp}$$

This is clear that rule ($\circ_{\dashrightarrow}$) should be applied. In turn, the algorithm should perform the abstract intersection:

$$\alpha_0 \cdot \mathtt{f} \mapsto \alpha_2 \sqcap_{\mathbb{H}^\sharp} \beta_0 \cdot \mathtt{f} \mapsto \beta_1$$

This abstract intersection should extend Φ with $\Phi(\alpha_2, \beta_1) = \gamma_1$. In a first time, rule ($\circ_{\dashrightarrow}$) should produce the resulting singleton:

$$\{[\alpha_0 \cdot \mathtt{f} \mapsto \alpha_1 \dashrightarrow \beta_0 \cdot \mathtt{f} \mapsto \beta_2]_{\mathrm{t}^\sharp}\}$$

In the final Φ, we thus should have $\Phi(\alpha_0, \beta_0) = \gamma_0$ and $\Phi(\alpha_2, \beta_1) = \gamma_1$. Thus, all occurrences of α_0 and β_0 should be renamed by γ_0 and all occurrences of α_2 and β_1 should be renamed by γ_1 in the resulting abstract heap relation. As there is no occurrence of α_2 and β_1 in it, we should obtain:

$$\{[\gamma_0 \cdot \mathtt{f} \mapsto \alpha_1 \dashrightarrow \gamma_0 \cdot \mathtt{f} \mapsto \beta_2]_{\mathrm{t}^\sharp}\}$$

Theorem 7.8 (Soundness of $\mathbf{comp}_{\mathbb{R}^\sharp}$). *Let* $\Psi_0, \Psi_1 \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$ *and* $\mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp \in \mathbb{R}^\sharp$. $\forall (\mathrm{h}_0, \mathrm{h}_1, \nu) \in (\mathbb{H} \times \mathbb{H} \times (\mathbb{V}^\sharp \to \mathbb{V}))$, $\forall \Psi_0', \Psi_1' \in \mathbb{V}^\sharp \to \mathbb{V}^\sharp$, *we have:*

$$\begin{array}{l} \exists \mathrm{h},\ (\mathrm{h}_0, \mathrm{h}, \Psi_0' \circ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_0^\sharp) \wedge (\mathrm{h}, \mathrm{h}_1, \Psi_1' \circ \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}_1^\sharp) \\ \Longrightarrow \forall \mathrm{r}^\sharp \textit{ such that } (\Psi_0', \Psi_1', \mathrm{r}^\sharp) \in \mathbf{comp}_{\mathbb{R}^\sharp}((\Psi_0, \Psi_1), \mathrm{r}_0^\sharp, \mathrm{r}_1^\sharp) : \quad (\mathrm{h}_0, \mathrm{h}_1, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \end{array}$$

Remark that Theorem 7.3 is also valid for the abstract composition. This means that some simplifications can be performed in the resulting abstract heap relations.

7.4.3 Abstract Composition in the Numerical Abstract Domain

The abstract composition in the numerical abstract domains consists of intersecting the two numerical abstract values and projecting away the result.

Abstract intersection. The first step of the numerical abstract composition simply consists of the abstract intersection in the numerical abstract domain $\mathbf{inter}_{\mathbb{N}^\sharp}$ introduced previously. Indeed, there is one abstract numerical value per abstract memory relation to compose. A resulting abstract memory relation should thus contain an abstract numerical value common to the abstract numerical values of the two composed abstract memory relations.

Projection. The intersected abstract numerical value may contain information about some symbolic values that are not in the composed abstract heap relation. These symbolic values should be then deleted. To do that, the abstract numerical domain requires a function $\mathbf{proj}_{\mathbb{N}^\sharp} \in \mathcal{P}_{\text{fin}}(\mathbb{V}^\sharp) \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$. This function takes a finite set of symbolic values $\mathscr{E}$, an abstract numerical value $\mathrm{n}^\sharp$, and returns a new abstract numerical value that is $\mathrm{n}^\sharp$ without the symbolic values that *are not* in $\mathscr{E}$.

Assumption 7.3 (Soundness of $\mathbf{proj}_{\mathbb{N}^\sharp}$). *Let $\mathscr{E} \in \mathcal{P}_{\text{fin}}(\mathbb{V}^\sharp)$ and $\mathrm{n}^\sharp \in \mathbb{N}^\sharp$. Then the function $\mathbf{proj}_{\mathbb{N}^\sharp} \in \mathcal{P}_{\text{fin}}(\mathbb{V}^\sharp) \times \mathbb{N}^\sharp \to \mathbb{N}^\sharp$ is sound if and only if:*

$$\gamma_{\mathbb{N}^\sharp}(\mathrm{n}^\sharp) \subseteq \gamma_{\mathbb{N}^\sharp}(\mathbf{proj}_{\mathbb{N}^\sharp}(\mathscr{E}, \mathrm{n}^\sharp))$$

The set of symbolic values can be computed by a function $\mathbf{get\_sv}_{\mathbb{R}^\sharp} \in \mathbb{R}^\sharp \to \mathcal{P}_{\text{fin}}(\mathbb{V}^\sharp)$, that takes an abstract heap relation $\mathrm{r}^\sharp$ and returns a set containing all the symbolic values that appear in $\mathrm{r}^\sharp$. As this operation is obvious, we do not detail it.

7.4.4 Abstract Composition of Abstract Memory Relations

The steps described previously allow to define the abstract composition of abstract memory relations as follows:

Definition 7.3 (Composition of abstract memory relations). *Let $\mathrm{m}^\sharp_{\mathcal{R}^0} = (\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0)$, $\mathrm{m}^\sharp_{\mathcal{R}^1} = (\mathrm{e}^\sharp_1, \mathrm{r}^\sharp_1, \mathrm{n}^\sharp_1)$ be two abstract memory relations, and $\Psi_{\text{init}} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2$ be a pair of renaming functions.*

$$\forall \mathtt{x} \in \mathbb{X}, \Phi_{\text{tmp}}(\mathrm{e}^\sharp_0(\mathtt{x}), \mathrm{e}^\sharp_1(\mathtt{x})) = \alpha \text{ and } \mathrm{e}^\sharp(\mathtt{x}) = \alpha \text{ and } \Phi_{\text{ext}} = \Phi_{\text{init}} \uplus \Phi_{\text{tmp}}$$

$$\begin{aligned}\mathbf{comp}_{\mathbb{M}^\sharp_{\mathcal{R}}}(\Phi_{\text{init}}, (\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0), (\mathrm{e}^\sharp_1, \mathrm{r}^\sharp_1, \mathrm{n}^\sharp_1)) = \\ \{(\mathrm{e}^\sharp, \mathrm{r}^\sharp, (\mathbf{proj}_{\mathbb{N}^\sharp}(\mathbf{get\_sv}_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp), \mathrm{n}^\sharp)) \mid (\Phi', \mathrm{r}^\sharp) \in \mathbf{comp}_{\mathbb{R}^\sharp}(\Phi_{\text{ext}}, \mathrm{r}^\sharp_0, \mathrm{r}^\sharp_1) \wedge \\ \mathrm{n}^\sharp = \mathbf{inter}_{\mathbb{N}^\sharp}(\Phi', \mathrm{n}^\sharp_0, \mathrm{n}^\sharp_1)\}\end{aligned}$$

Theorem 7.9 (Soundness of $\mathbf{comp}_{\mathbb{M}^\sharp_{\mathcal{R}}}$). *Let $\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1} \in \mathbb{M}^\sharp_{\mathcal{R}}$ and $\Phi_{\text{init}} \in (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2$, then:*

$$\gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^0}) \circ \gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}^1}) \subseteq \bigcup\{\gamma_{\mathbb{M}^\sharp_{\mathcal{R}}}(\mathrm{m}^\sharp_{\mathcal{R}}) \mid \mathrm{m}^\sharp_{\mathcal{R}} \in \mathbf{comp}_{\mathbb{M}^\sharp_{\mathcal{R}}}(\Phi_{\text{init}}, \mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1})\}$$

We now discuss two examples of the abstract composition.

Example 7.4 (Composition). In this example, we perform the composition of two abstract heap relations containing linked lists. For clarity, we omit the `data`

fields of all the list nodes in this example. We assume that we have $\Phi(\alpha_0, \beta_0) = \gamma_0$.

$$[\mathbf{emp} \dashrightarrow \alpha_0 \cdot \mathtt{next} \mapsto \alpha_1]^{\sharp}_{\mathtt{t}_0} \circ_{\mathbb{R}^\sharp} [\mathbf{emp} \dashrightarrow \beta_1 \cdot \mathtt{next} \mapsto \beta_0]^{\sharp}_{\mathtt{t}_1} *_{\mathtt{R}} \mathtt{Id}(\mathbf{list}(\beta_0))$$

Rule $(\circ_{*_{\mathtt{R}}})$ allows to perform $[\mathbf{emp} \dashrightarrow \alpha_0 \cdot \mathtt{next} \mapsto \alpha_1]^{\sharp}_{\mathtt{t}_0} \circ_{\mathbb{R}^\sharp} \mathtt{Id}(\mathbf{list}(\beta_0))$ independently. The inductive predicate $\mathtt{Id}(\mathbf{list}(\beta_0))$ is unfolded into $\mathtt{Id}(\mathbf{emp})$ and $\mathtt{Id}(\beta_0 \cdot \mathtt{next} \mapsto \beta_2 *_{\mathtt{S}} \mathbf{list}(\beta_2))$ with rule $(\circ_{\mathrm{unfold_R}})$.

Then the algorithm tries to compute an abstract heap relation both for

$$[\mathbf{emp} \dashrightarrow \alpha_0 \cdot \mathtt{next} \mapsto \alpha_1]^{\sharp}_{\mathtt{t}_0} \circ_{\mathbb{R}^\sharp} \mathtt{Id}(\mathbf{emp})$$

and

$$[\mathbf{emp} \dashrightarrow \alpha_0 \cdot \mathtt{next} \mapsto \alpha_1]^{\sharp}_{\mathtt{t}_0} \circ_{\mathbb{R}^\sharp} \mathtt{Id}(\beta_0 \cdot \mathtt{next} \mapsto \beta_2 *_{\mathtt{S}} \mathbf{list}(\beta_2))$$

The first case obviously fails. However, the second case succeeds by applying successively rules $(\circ_{\dashrightarrow \_ \mathtt{Id}})$ (for the points-to predicates) and $(\circ_{\mathrm{unfold_R}})$ (to unfold $\mathbf{list}(\beta_2)$ into $\mathtt{Id}(\mathbf{emp})$). This extends Φ with $\Phi(\alpha_1, \beta_2) = \gamma_2$ and produces:

$$[\mathbf{emp} \dashrightarrow \gamma_0 \cdot \mathtt{next} \mapsto \gamma_2]^{\sharp}_{\mathtt{t}_2}$$

The next step of the algorithm is to compose $[\mathbf{emp} \dashrightarrow \beta_1 \cdot \mathtt{next} \mapsto \beta_0]^{\sharp}_{\mathtt{t}_1}$. As β_1 is not mapped with a symbolic value, the algorithm tries to compose it with the neutral element of abstract heap relations:

$$[\mathbf{emp} \dashrightarrow \mathbf{emp}]_{\mathbf{id}_{\mathbb{T}^\sharp}(\mathbf{emp})} \circ_{\mathbb{R}^\sharp} [\mathbf{emp} \dashrightarrow \beta_1 \cdot \mathtt{next} \mapsto \beta_0]^{\sharp}_{\mathtt{t}_1}$$

This composition succeeds with rule $(\circ_{\dashrightarrow})$. Finally, the algorithm also renames β_0 by γ_0 and the resulting abstract heap relation is:

$$[\mathbf{emp} \dashrightarrow \beta_1 \cdot \mathtt{next} \mapsto \gamma_0]^{\sharp}_{\mathtt{t}_3} *_{\mathtt{R}} [\mathbf{emp} \dashrightarrow \gamma_0 \cdot \mathtt{next} \mapsto \gamma_2]^{\sharp}_{\mathtt{t}_2}$$

Example 7.5 (Composition with weakening). We consider the composition of the following abstract heap relations with $\Phi(\alpha_0, \beta_0) = \gamma_0$ and $\Phi(\alpha_1, \beta_1) = \gamma_1$:

$$[\mathrm{h}^{\sharp}_0 \dashrightarrow \mathbf{list}(\alpha_0)]^{\sharp}_{\mathtt{t}_0} *_{\mathtt{R}} [\mathrm{h}^{\sharp}_1 \dashrightarrow \mathbf{list}(\alpha_1)]^{\sharp}_{\mathtt{t}_1} \circ_{\mathbb{R}^\sharp} [\mathbf{list}(\beta_0) *_{\mathtt{S}} \mathbf{list}(\beta_1) \dashrightarrow \mathrm{h}^{\sharp}_2]^{\sharp}_{\mathtt{t}_3}$$

In the right abstract heap relation, we observe that $\mathbf{list}(\beta_0)$ and $\mathbf{list}(\beta_1)$ are in the same transform-into relation. Consequently, we cannot apply rule $(\circ_{*_{\mathtt{R}}})$. The rule to applied is $(\circ_{\mathrm{weak_L}})$ that merges the two transform-into relations of the left abstract heap relation. In turn, the algorithm performs rule $(\circ_{\dashrightarrow})$.

7.5 Related Works

In this chapter, we used abstract relations to define an over-approximation of their composition. Other static analyses do not perform such an abstract composition, but perform the application of a relation. This is notably the case for typing systems, like [DM82]. Given a function f, its type is represented by a relation over two types $t \to t'$, that means that f inputs an element of type t and outputs and element of type t'. The typing rule of a function call '$f\ e$' states that if the type of e is t, and if the type of f is $t \to t'$, then the type of '$f\ e$' is t'. We can use our abstract composition to perform such an application. Indeed, if $\mathrm{r}^\sharp$ describes a relation for a function f and $\mathrm{h}^\sharp$ the calling state, we can perform $\mathtt{Id}(\mathrm{h}^\sharp) \circ_{\mathbb{R}^\sharp} \mathrm{r}^\sharp$ and reconstruct the output state to get the result of the application of $\mathrm{r}^\sharp$ by $\mathrm{h}^\sharp$. Conversely, typing systems cannot use the application to perform a composition of two relations.

In the context of works that define a composition of abstract relations, the work in [JLRS10] is probably the closest to ours. They also design a composition operator for input-output relations over memory states containing data structures, that is also based of the abstract intersection. The main difference is that their relations are described by a two-vocabulary of TVLA [SRW02] formulas, whereas we use relational connectives over separation logic [Rey02] formulas. Their composition operator is defined as follows: they promote the two two-vocabularies structures into two three-vocabularies structures, perform the intersection of the three-vocabularies and project away the middle vocabulary. Our approach is different in the sense that we implement a system rules over our relational connectives that performs the intersection over the common states, and unfolds inductive predicates. The work in [BDES11] is also very close to ours. They use a mechanism based on unfolding-fold/folding operations that allows to define an abstract intersection operator that can be used at function calls and returns, in order to minimize the loss of precision. The difference with our work is that these operations are over first order formulas and multiset constraints.

Chapter 8

Compositional Inter-procedural Shape Analysis

In this chapter, we present a compositional inter-procedural shape analysis that takes benefit of the abstract relations computed in the previous chapters, to lose a minimum of precision during the composition step. The analysis proceeds as follows: it maintains a mapping that associates for each function an abstract relation that is composed with the contextual abstract relation when the function is called. We demonstrate this process makes the shape analysis scalable without loosing too much precision.

8.1 Introduction

In this chapter, we complete the relational intra-procedural shape analysis that has been defined in Chapter 5 and extended in Chapter 6. We now allow to $[\![.]\!]^\sharp_{\mathcal{R}}$ to handle function calls and function returns. If $f(e_1, \ldots, e_n)$ is a call to the function f, the abstract relational semantics $[\![.]\!]^\sharp_{\mathcal{R}}$ computes an over-approximations of $[\![f(e_1, \ldots, e_n);]\!]_{\mathcal{R}}$. Thus, $[\![f(e_1, \ldots, e_n);]\!]^\sharp_{\mathcal{R}}$ should meet the following soundness condition:

$$\begin{array}{l}\forall \mathbb{r}^\vee \in \mathbb{R}^\vee, \quad \forall (\mathrm{m}_0, \mathrm{m}_1) \in \gamma_{\mathbb{R}^\vee}(\mathbb{r}^\vee), \quad \forall \mathrm{m}_2 \in \mathbb{M}, \\ \quad (\mathrm{m}_1, \mathrm{m}_2) \in [\![f(e_1, \ldots, e_n);]\!]_{\mathcal{R}} \quad \Longrightarrow \\ \quad (\mathrm{m}_0, \mathrm{m}_2) \in \gamma_{\mathbb{R}^\vee}([\![f(e_1, \ldots, e_n);]\!]^\sharp_{\mathcal{R}}(\mathbb{r}^\vee))\end{array}$$

This analysis is a top-down analysis, that goes from a root function to leaf functions. It is a natural extension of our intra-procedural analysis, it only requires to provide a pre-condition for the root function. Abstract relations describe *function summaries*, and are stored in a table $\tau^\sharp$ that associates a function to its summary (Section 8.2). A function summary is a pair made of a pre-condition (an abstract state) and an abstract relation.

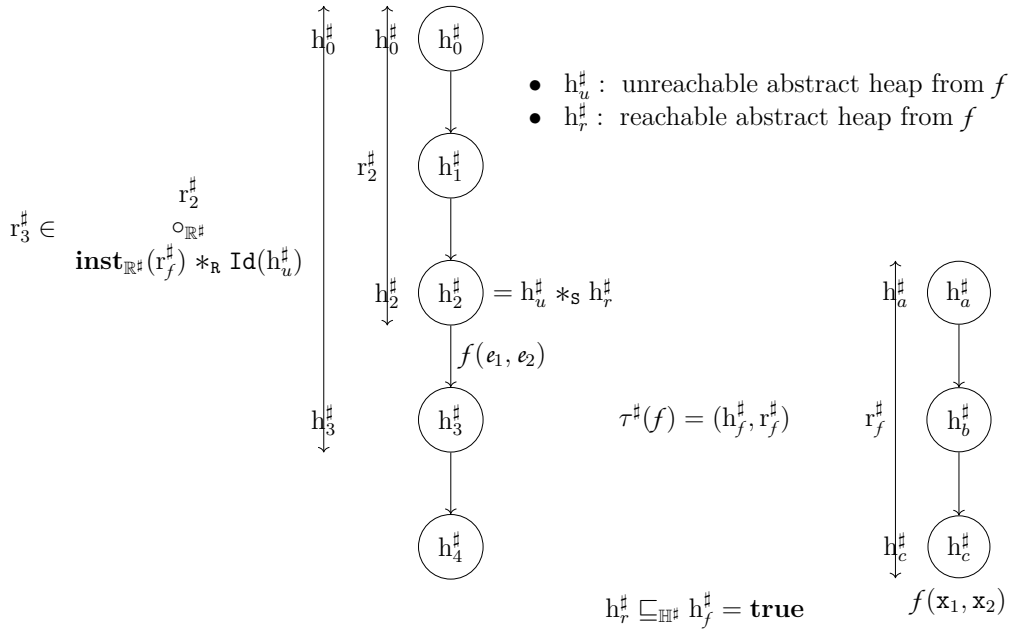


Figure 8.1: Graphical representation of the main steps of the inter-procedural compositional analysis (first case)

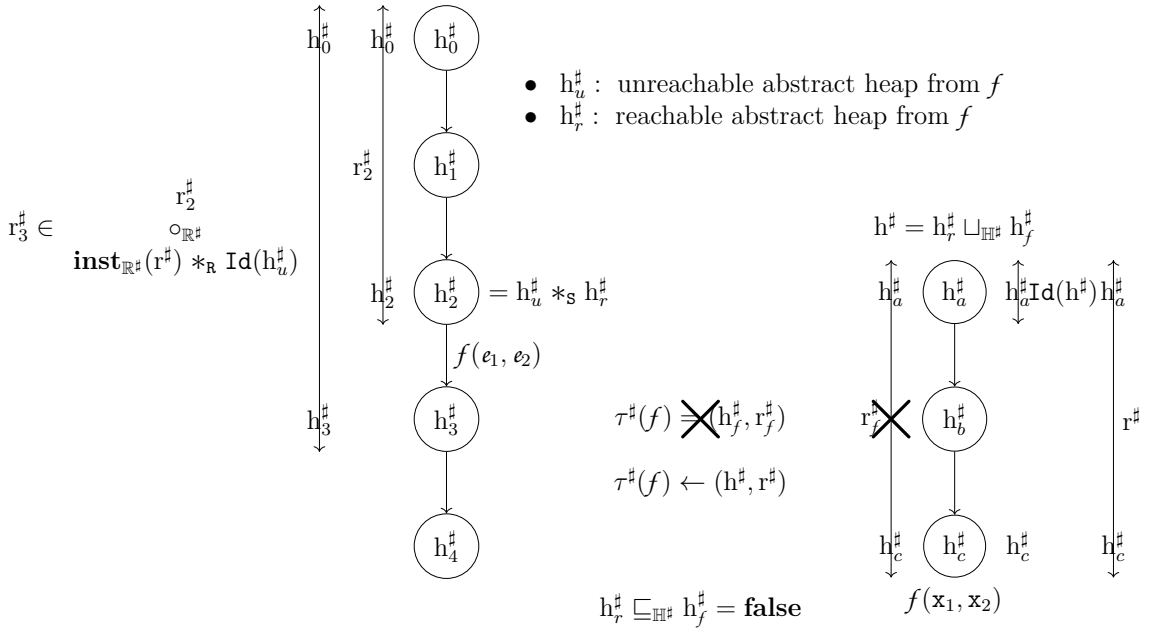


Figure 8.2: Graphical representation of the main steps of the inter-procedural compositional analysis (second case)

All function summaries are initialized to the bottom element (of which the concretization is the empty set), and composed with the abstract composition of Chapter 7 at each call.

There are two cases during the abstract evaluation of function calls: when a function summary is not updated and when it is. These two cases are represented respectively in Figure 8.1 and Figure 8.2. In these figures, nodes correspond to abstract states and edges to commands. Abstract relations are thus related to nodes.

We first comment on Figure 8.1. On the left, we observe a call to the function f, whose the abstract relation $\mathrm{r}_f^\sharp$ is represented on the right. To evaluate this function call, the analysis splits the calling abstract state $\mathrm{h}_2^\sharp$ of the current abstract relation $\mathrm{r}_2^\sharp$, into two parts: $\mathrm{h}_u^\sharp$ that represents the *unreachable abstract heap* from the actual parameters of f and $\mathrm{h}_r^\sharp$ that represents the *reachable abstract heap*. Then, the analysis tests if $\mathrm{h}_r^\sharp$ is included in the pre-condition $\mathrm{h}_f^\sharp$ of f, in order to check if the function summary is valid according to the calling context. In Figure 8.1, the inclusion test returns **true**. Thus, the analysis can directly compose the current abstract relation $\mathrm{r}_2^\sharp$ with an *instantiated* version of $\mathrm{r}_f^\sharp$ (where fresh symbolic values have been generated) and the identity relation of $\mathrm{h}_u^\sharp$ (by the compositional frame rule). All steps related to the abstract evaluation of function calls (included the instantiation and the compositional frame rule) are presented in Section 8.3.

We now comment on Figure 8.2, where $\mathrm{h}_r^\sharp$ is not included in $\mathrm{h}_f^\sharp$. In this case, the function summary of f is not valid for the calling context, the analysis then computes a *new function summary* for f, more general than the previous one. To do that, the analysis first computes a new pre-condition $\mathrm{h}^\sharp$ for f, by joining $\mathrm{h}_r^\sharp$ with $\mathrm{h}_f^\sharp$, and reanalyzes the body of f from the identity relation of $\mathrm{h}^\sharp$. The previous function summary of f is substituted by the new one.

Section 8.4 defines both the abstract evaluation of function returns and the abstract relational semantics. Section 8.5 discusses how our analysis can track cutpoints. In Section 8.6, we evaluate the analysis on a real benchmark, and we compare our approach with related works in Section 8.7.

8.2 Abstract Function Summaries

This section formalizes how our abstract relations are used to represent function summaries. In Section 8.2.1, we first explicit the meaning of function summaries in the concrete semantics from Chapter 3. Concrete function summaries are traces of function events, where function events is function calls and function returns. In Section 8.2.2, we define an abstraction of these traces of function events, that consists of a table that associates a function to a pair made of an abstract pre-condition and an abstract relation.

8.2.1 Traces of Function Events

An important feature of the compositional inter-procedural analysis is to compute for each function a *summary*, and compose this summary when the function is called instead of reanalyzing it.

Informally, a function summary abstracts the relations between the states *at the entry* of the function (before executing its first instruction) and the states *at the exit* of the function (after executing its last instruction). Before defining how these summaries are formally represented, it is important to give them a meaning in the concrete semantics. In the concrete semantics, this is not obvious how to associate each function call to its input and output states. Thus, the idea is to simplify the trace semantics defined in Chapter 3 by discarding all the elements of the trace except those corresponding to function calls and function returns, and to abstract them into *function events*. A function event is simply a triplet that associates a function to an event (`In` for input or `Out` for output) and a memory state. For instance, the function event $(f, \texttt{In}, \mathrm{m})$ indicates that the function f has been input with the memory state m.

To abstract the trace semantics into a set of traces of function events, we first need a function $\textbf{to\_event} \in \Sigma^\infty \to (\mathcal{F}un \times \{\texttt{In}, \texttt{Out}\} \times \mathbb{M})^\infty$ that converts a trace of stacks into a trace of function events. We remind that a stack $\sigma \in \Sigma$ contains pairs $\langle c \mid \mathrm{m} \rangle$ where c is a command and m is a concrete memory state.

Remark 4 (Notation for traces)**.** *We use the symbol τ to denote any trace, finite or infinite. We also define the trace $\tau_0 \bullet \tau_1$ as the concatenation of the traces τ_0 and τ_1, if τ_0 is a finite trace. Finally, we let the empty trace $\langle\rangle$ be the neutral element for $\bullet$.*

Definition 8.1 (Conversion of traces of stacks into traces of function events). *We define by induction over traces of stacks the conversion function* $\textbf{to\_event} \in \Sigma^\infty \to (\mathcal{F}un \times \{\texttt{In}, \texttt{Out}\} \times \mathbb{M})^\infty$ *from traces of stacks into traces of function events:*

$$\textbf{to\_event}(\langle\, \langle f(e_1, \ldots, e_n)\,;\, c \mid \mathrm{m}\rangle :: \sigma, \quad \langle c' \mid \mathrm{m}'\rangle :: \langle f(e_1, \ldots, e_n)\,;\, c \mid \mathrm{m}\rangle :: \sigma \,\rangle \bullet \tau) = \langle (f, \texttt{In}, \mathrm{m}')\rangle \bullet \textbf{to\_event}(\tau)$$

$$\textbf{to\_event}(\langle\, \langle \textbf{ret} \mid \mathrm{m}\rangle :: \langle f(e_1, \ldots, e_n)\,;\, c \mid \mathrm{m}'\rangle :: \sigma \,\rangle \bullet \tau) \langle (f, \texttt{Out}, \mathrm{m})\rangle \bullet \textbf{to\_event}(\tau)$$

$$\textbf{to\_event}(\langle\sigma\rangle \bullet \tau) = \textbf{to\_event}(\tau)$$

$$\textbf{to\_event}(\langle\rangle) = \langle\rangle$$

We remark that the first two cases of **to_event** correspond to the two last rules in Figure 3.3 (page 37): function calls and function returns. Regarding to function calls,

the memory state of the function event is the one from which we enter in the function but not the one from which we call the function. The third case discards all the other instructions of the trace. Also, **to_event** is defined both for finite and infinite traces.

Definition 8.2 (Abstraction of the trace semantics). *We define the abstraction function of the trace semantics* $\alpha_{\mathcal{T}} \in \mathcal{P}(\Sigma^\infty) \to \mathcal{P}((\mathcal{F}un \times \{\texttt{In}, \texttt{Out}\} \times \mathbb{M})^\infty)$ *that maps a set of traces of stacks into a set of traces of function events:*

$$\alpha_{\mathcal{T}}(\Sigma^\infty) = \{\textbf{to\_event}(\tau) \mid \tau \in \Sigma^\infty\}$$

Example 8.1 (Finite trace of function events). We consider the following function `f` that calls a function `g` if its integer argument `n` is not null. We assume that there is no function call neither in the **else** of `f` nor in the body of `g`.

```
1 void f(int n) {
2   if (n != 0) g(n);
3   else {
4     ...
5   }
6 }
```

We now consider the two consecutive calls to the function `f`:

```
1 f(1);
2 f(0);
```

A possible trace of function events for this sequence is the following trace:

$$\langle(\texttt{f}, \texttt{In}, \mathrm{m}_0), (\texttt{g}, \texttt{In}, \mathrm{m}_1), (\texttt{g}, \texttt{Out}, \mathrm{m}_2), (\texttt{f}, \texttt{Out}, \mathrm{m}_3), (\texttt{f}, \texttt{In}, \mathrm{m}_4), (\texttt{f}, \texttt{Out}, \mathrm{m}_5)\rangle$$

Example 8.2 (Infinite trace of function events). We consider the following recursive function `f` that does not terminate:

```
1 void f(int n) {
2   f(n+1);
3 }
```

The only possible trace of function events for the execution of this function is the infinite trace of the form:

$$\langle(\texttt{f}, \texttt{In}, \mathrm{m}_0), (\texttt{f}, \texttt{In}, \mathrm{m}_1), \ldots\rangle$$

8.2.2 Function Summaries Table

In this section, we define an abstraction of traces of function events, named a *function summary table*. A first and intuitive approach to define such abstraction is to associate to each function a disjunction of abstract memory relations. However, abstract memory relations describe relations between input states and output states. Here, the important point is that this is not true that for all input states there exists an output state. More concretely, a function can terminate from some input states but can also loop indefinitely or crash from some other input states. Thus, using only abstract memory relations is not enough to define a sound abstraction of function events.

The solution is to associate at each function a pair made of a disjunction of abstract memory states $\mathbb{m}^\vee$ and a disjunction of abstract memory relations $\mathbb{r}^\vee$, where $\mathbb{m}^\vee$ describes all the input states of the function, even those from which the function does not terminate; and where $\mathbb{r}^\vee$ describes relations between the input and output states of the function. Thus, a function summary table $\tau^\sharp \in \mathcal{T}^\sharp = \mathcal{F}un \to \mathbb{M}^\vee \times \mathbb{R}^\vee$ is a function that associates a function name to a disjunction of abstract memory states and a disjunction of abstract memory relations. We name the disjunction of abstract memory states the *pre-condition* of the function summary and the disjunction of abstract memory relations the *relation* of the function summary.

The concretization function $\gamma_{\tau^\sharp} \in \mathcal{T}^\sharp \to \mathcal{P}((\mathcal{F}un \times \{\mathtt{In}, \mathtt{Out}\} \times \mathbb{M})^\infty)$ of a function summary table simply maps it into a set of traces of function events. For clarity, we split this concretization function into two concretization functions: $\gamma_{[\tau^\sharp, \mathbb{H}^\sharp]}$ and $\gamma_{[\tau^\sharp, \mathbb{R}^\sharp]}$, that respectively concretize the pre-condition of all the functions and concretize the relation of all the functions. We first define $\gamma_{[\tau^\sharp, \mathbb{H}^\sharp]}$.

Definition 8.3 (Concretization of the pre-conditions $\mathcal{T}^\sharp$). *The concretization function $\gamma_{[\tau^\sharp, \mathbb{H}^\sharp]} \in \mathcal{T}^\sharp \to \mathcal{P}((\mathcal{F}un \times \{\mathtt{In}, \mathtt{Out}\} \times \mathbb{M})^\infty)$ of a function summary table $\tau^\sharp$ is defined as follows:*

$$\gamma_{[\tau^\sharp, \mathbb{H}^\sharp]}(\tau^\sharp) = \{\tau_0 \bullet \langle (f, \mathtt{In}, (\mathrm{e}, \mathrm{h})) \rangle \bullet \tau_1 \mid \forall \mathbb{m}^\vee \in \mathbb{M}^\vee, \mathbb{r}^\vee \in \mathbb{R}^\vee : \\ \tau^\sharp(f) = (\mathbb{m}^\vee, \mathbb{r}^\vee) \Longrightarrow \exists \mathrm{h}', \mathrm{h}'' \in \mathbb{H} : \ \mathrm{h} = \mathrm{h}' \circledast \mathrm{h}'' \ \wedge \ (\mathrm{e}, \mathrm{h}') \in \gamma_{\mathbb{M}^\vee}(\mathbb{m}^\vee)\}$$

We notice that function summary tables are concretized only in traces that relate the input states of functions. Moreover, the concrete heap h of the function event must be splittable into two concrete heaps h' and h'', where h' is a concrete heap in the concretization of $\mathbb{m}^\vee$. This allows $\mathbb{m}^\vee$ to describe a specific part of the input heap that is relevant only for the function called. We operate this property to make $\mathbb{m}^\vee$ *modular*: it can be valid for many calling contexts.

We now focus on the definition of $\gamma_{[\tau^\sharp, \mathbb{R}^\sharp]}$. A prerequisite of this definition is a way to associate corresponding pairs of input and output states in a trace of function events. Indeed, the same function may be called many times, and thus the trace of function events may contain many function events that related both input or output states for

this function. It is not automatic to associate an output state to its corresponding input state. To illustrate this, we consider the following trace:

$$\langle (f, \texttt{In}, \mathrm{m}_1), (f, \texttt{Out}, \mathrm{m}_2), (f, \texttt{In}, \mathrm{m}_3), (f, \texttt{In}, \mathrm{m}_4), (f, \texttt{Out}, \mathrm{m}_5), (f, \texttt{Out}, \mathrm{m}_6) \rangle$$

In this trace, we have many input states (m_1, m_3 and m_4) and many output states (m_2, m_5 and m_6) for the function f. To associate the corresponding pairs of input-outputs states (($\mathrm{m}_1, \mathrm{m}_2$), ($\mathrm{m}_3, \mathrm{m}_6$) and ($\mathrm{m}_4, \mathrm{m}_5$)) of this trace, we need a notion of *well parenthesized* trace. If a trace τ starts by a function event of the form $\langle (f, \texttt{In}, \mathrm{m}), \ldots, (f, \texttt{Out}, \mathrm{m}') \rangle$ and if τ is well parenthesized, this means that m' is the corresponding output state of the input state m. This requires to define a predicate **well_parenthesized** over traces of function events.

Definition 8.4 (The well parenthesized predicate). *The following inference rules define the predicate* **well_parenthesized** *by induction on the syntax of traces of function events:*

$$\frac{}{\textbf{well\_parenthesized}(\langle\rangle)}$$

$$\frac{\textbf{well\_parenthesized}(\tau_1) \quad \textbf{well\_parenthesized}(\tau_2)}{\textbf{well\_parenthesized}(\tau_1 \bullet \tau_2)}$$

$$\frac{\textbf{well\_parenthesized}(\tau)}{\textbf{well\_parenthesized}(\langle (f, \texttt{In}, \mathrm{m}_0) \rangle \bullet \tau \bullet \langle (f, \texttt{Out}, \mathrm{m}_1) \rangle)}$$

Note that the empty trace is well parenthesized and a non empty well parenthesized trace necessarily starts by a function event associating a function to an input state and ends by a function event associating the same function to an output state. We can now define $\gamma_{[\tau^\sharp, \mathbb{R}^\sharp]}$:

Definition 8.5 (Concretization of the relations of $\mathcal{T}^\sharp$). *The concretization function $\gamma_{[\tau^\sharp, \mathbb{R}^\sharp]} \in \mathcal{T}^\sharp \to \mathcal{P}((\mathcal{F}un \times \{\texttt{In}, \texttt{Out}\} \times \mathbb{M})^\infty)$ of a function summary table $\tau^\sharp$ is definied as follows:*

$$\begin{aligned}\gamma_{[\tau^\sharp, \mathbb{R}^\sharp]}(\tau^\sharp) = \{ & \tau_0 \bullet \langle (f, \texttt{In}, (\mathrm{e}, \mathrm{h_i})) \rangle \bullet \tau_1 \bullet \langle (f, \texttt{Out}, (\mathrm{e}, \mathrm{h_o})) \rangle \bullet \tau_2 \mid \forall \mathsf{m}^\vee \in \mathbb{M}^\vee, \mathsf{r}^\vee \in \mathbb{R}^\vee : \\ & \tau^\sharp(f) = (\mathsf{m}^\vee, \mathsf{r}^\vee) \wedge \textbf{well\_parenthesized}(\tau_1) \\ & \Longrightarrow \exists \mathrm{h}'_\mathrm{i}, \mathrm{h}'_\mathrm{o}, \mathrm{h}' \in \mathbb{H} : \\ & \quad \mathrm{h_i} = \mathrm{h}'_\mathrm{i} \circledast \mathrm{h}' \ \wedge \mathrm{h_o} = \mathrm{h}'_\mathrm{o} \circledast \mathrm{h}' \ \wedge ((\mathrm{e}, \mathrm{h}'_\mathrm{i}), \ (\mathrm{e}, \mathrm{h}'_\mathrm{o})) \in \gamma_{\mathbb{R}^\vee}(\mathsf{r}^\vee)\}\end{aligned}$$

This definition is similar to Definition 8.3. The abstract memory relations of $\tau^\sharp$ form well parenthesized traces of function events. We also remark that all abstract memory

relations are *modular*: they abstract a specific part of the heaps of the input and output states of the function events.

Using Definition 8.3 and Definition 8.5, we can give a definition for the concretization of function map tables.

Definition 8.6 (Concretization of function map tables). *The concretization function* $\gamma_{\tau^\sharp} \in \mathcal{T}^\sharp \to \mathcal{P}((\mathcal{F}un \times \{\texttt{In}, \texttt{Out}\} \times \mathbb{M})^\infty)$ *of function summary tables is defined as follows:*

$$\gamma_{\tau^\sharp}(\tau^\sharp) = \gamma_{[\tau^\sharp, \mathbb{H}^\sharp]}(\tau^\sharp) \cap \gamma_{[\tau^\sharp, \mathbb{R}^\sharp]}(\tau^\sharp)$$

Example 8.3. In this example, we consider the following function `tail` that inputs a list `l` and return its tail.

```
1 list *tail(list *l) {
2   return l->next;
3 }
```

Let $\tau^\sharp \in \mathcal{T}^\sharp$. We assume that:

$$\tau^\sharp(\texttt{tail}) = (\{(\mathrm{e}^\sharp, \mathrm{h}^\sharp, \mathrm{n}^\sharp)\},\ \{(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp)\})$$

where

$$\mathrm{h}^\sharp = \mathbf{list}(\alpha) \text{ and } \mathrm{r}^\sharp = \texttt{Id}(\alpha \cdot \texttt{data} \mapsto \delta \ast_{\mathrm{s}} \alpha \cdot \texttt{next} \mapsto \beta \ast_{\mathrm{s}} \mathbf{list}(\beta))$$

This is an example of abstraction of the semantics of the functions `tail`. Indeed, a valid pre-condition of this function is $\mathrm{h}^\sharp = \mathbf{list}(\alpha)$, and the relational analysis should consequently start with $\texttt{Id}(\mathbf{list}(\alpha))$. To evaluate the expression `l->next`, the analysis unfolds the predicate $\mathbf{list}(\alpha)$, that generates the abstract heap relation $\mathrm{r}^\sharp$ above.

8.3 Abstract Function Calls

This section formalizes the different steps of the abstract evaluation of function calls. This section is organized as follows: while Section 8.3.1 and Section 8.3.2 are related to the steps that manipulate the calling abstract relation, Section 8.3.3 and Section 8.3.4 are related to the steps that reason over the abstract relation of the called function. Finally, Section 8.3.5 describes how all these steps are combined to perform the abstract function calls.

8.3.1 Initialization of Function Calls

In this section, we define the *binding* operation on the calling abstract memory relation $\mathrm{m}^\sharp_{\mathcal{R}}$. This operation corresponds to the initialization of function calls. Suppose that the

function call to analyze is $f(e_1, \ldots, e_n)$ and that the arguments of f are the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$. The goal of the binding is to perform from $\mathrm{m}^\sharp_\mathcal{R}$ the following assignments $\mathtt{x}_1 = e_1, \ldots, \mathtt{x}_n = e_n$.

The binding proceeds as follows: it first creates a new abstract environment, that contains only the arguments of the called function. Then, for each argument, it requires to allocate a new memory cell, and to assign it the evaluation of its corresponding expression.

The creation of the new abstract environment is performed by the function $\mathbf{create}_{\mathbb{E}^\sharp} \in \mathcal{P}_{\text{fin}}(\mathbb{X}) \to \mathbb{E}^\sharp$ that takes a set of variables and returns an abstract environment that maps each variable to a fresh symbolic value.

Definition 8.7 (Creation of the abstract environment). *The function* $\mathbf{create}_{\mathbb{E}^\sharp} \in \mathcal{P}_{\text{fin}}(\mathbb{X}) \to \mathbb{E}^\sharp$ *that creates a new abstract environment from a set of variables is defined as follows:*

$$\mathbf{create}_{\mathbb{E}^\sharp}(\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}) = \mathrm{e}^\sharp, \quad \begin{array}{l}\textit{such that } \forall i, 1 \leqslant i \leqslant n, \mathrm{e}^\sharp(\mathtt{x}_i) = \alpha_i \textit{ (with } \alpha_i \textit{ fresh)} \\ \textit{and } \mathrm{e}^\sharp \textit{ is defined only for these variables}\end{array}$$

The function $\mathbf{bind\_cell}_{\mathbb{M}^\sharp_\mathcal{R}} \in \mathbb{V}^\sharp \times \mathbb{F} \times \mathcal{E}\mathit{xpr} \times \mathbb{M}^\sharp_\mathcal{R} \to \mathbb{M}^\sharp_\mathcal{R}$ proceeds to the binding of one memory cell. This means that $\mathbf{bind\_cell}_{\mathbb{M}^\sharp_\mathcal{R}}(\alpha, \mathtt{f}, e, \mathrm{m}^\sharp_\mathcal{R})$ binds the cell at address $(\alpha, \mathtt{f})$ to the evaluation of e in $\mathrm{m}^\sharp_\mathcal{R}$. It uses the functions defined in Chapter 5 $\mathbf{eval}_\mathtt{E}$ (Section 5.2) to evaluate the expression, $\mathbf{alloc}_{\mathbb{R}^\sharp}$ (Section 5.5) to allocate the new cell, and $\mathbf{assign}_{\mathbb{N}^\sharp}$ and $\mathbf{assign}_{\mathbb{R}^\sharp}$ (Section 5.4) to perform the assignment of the memory cell.

Definition 8.8 (Binding of a single cell). *Let* $\alpha \in \mathbb{V}^\sharp, \mathtt{f} \in \mathbb{F}, e \in \mathcal{E}\mathit{xpr}$ *and let* $\mathrm{m}^\sharp_\mathcal{R} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0)$ *be an abstract memory relation. Then:*

$$\begin{array}{l}\mathbf{bind\_cell}_{\mathbb{M}^\sharp_\mathcal{R}}(\alpha, \mathtt{f}, e, \mathrm{m}^\sharp_\mathcal{R}) = \\ \quad \textit{Let } (\beta, \mathrm{p}^\sharp) = \mathbf{eval}_E(e, \mathrm{e}^\sharp, \mathrm{r}^\sharp_0) \textit{ in} \\ \quad \textit{Let } \mathrm{n}^\sharp_1 = \mathbf{assign}_{\mathbb{N}^\sharp}(\beta, \mathrm{p}^\sharp, \mathrm{n}^\sharp_0) \textit{ in} \\ \quad \textit{Let } \mathrm{r}^\sharp_1 = \mathbf{alloc}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \mathrm{r}^\sharp_0) \textit{ in} \\ \quad \textit{Let } \mathrm{r}^\sharp_2 = \mathbf{assign}_{\mathbb{R}^\sharp}(\alpha, \mathtt{f}, \beta, \mathrm{r}^\sharp_1) \textit{ in} \\ \quad (\mathrm{e}^\sharp, \mathrm{r}^\sharp_2, \mathrm{n}^\sharp_1)\end{array}$$

Theorem 8.1 (Soundness of $\mathbf{bind\_cell}_{\mathbb{M}^\sharp_\mathcal{R}}$). *Let* $\alpha \in \mathbb{V}^\sharp$, $\mathtt{f} \in \mathbb{F}$, $e \in \mathcal{E}\mathit{xpr}$ *and* $\mathrm{m}^\sharp_\mathcal{R} \in \mathbb{M}^\sharp_\mathcal{R}$.

$$\forall (\mathrm{m_i}, (\mathrm{e}, \mathrm{h_o})) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_\mathcal{R}) \Longrightarrow$$

$$\exists \nu \in \mathbb{V}^\sharp \to \mathbb{V}, \quad (\mathrm{m_i}, (\mathrm{e}, \mathrm{h_o} \circledast [\nu(\alpha) + \mathtt{f} \mapsto \mathcal{E}[\![e]\!](\mathrm{e}, \mathrm{h_o})])) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathbf{bind\_cell}_{\mathbb{M}^\sharp_\mathcal{R}}(\alpha, \mathtt{f}, e, \mathrm{m}^\sharp_\mathcal{R}))$$

Finally, the function $\mathbf{bind}_{\mathbb{M}^\sharp_\mathcal{R}} \in \mathcal{P}_{\text{fin}}((\mathbb{X} \times \mathit{Expr})) \times \mathbb{M}^\sharp_\mathcal{R} \to \mathbb{M}^\sharp_\mathcal{R}$ proceeds at the binding for all the arguments of the called function. It inputs a set of pairs that associate each function argument to an expression, the calling abstract memory relation and outputs the initialized abstract memory relation. This latter should contain the abstract environment containing the addresses of the function arguments.

Definition 8.9 (Binding). *Let* $\mathrm{m}^\sharp_{\mathcal{R}^0} = (\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_0, \mathrm{n}^\sharp_0)$ *be an abstract memory relation. Let* $\mathrm{e}^\sharp_1 = \mathbf{create}_{\mathbb{E}^\sharp}(\{\mathtt{x}_1, \ldots, \mathtt{x}_n\})$*. The abstract memory relations* $\mathrm{m}^\sharp_{\mathcal{R}^1}, \ldots, \mathrm{m}^\sharp_{\mathcal{R}^n}$ *are defined as follows:*

$$\forall i, 1 \leqslant i \leqslant n, \ \textit{if}\ \mathbf{eval}_E(\mathtt{x}_i, \mathrm{e}^\sharp_1, \mathrm{r}^\sharp_0) = (\alpha_i, \mathtt{f}_i)$$

$$\textit{then}\ \mathrm{m}^\sharp_{\mathcal{R}^i} = \mathbf{bind\_cell}_{\mathbb{M}^\sharp_\mathcal{R}}(\alpha_i, \mathtt{f}_i, e_i, \mathrm{m}^\sharp_{\mathcal{R}^{i-1}})$$

If $\mathrm{m}^\sharp_{\mathcal{R}^n} = (\mathrm{e}^\sharp_0, \mathrm{r}^\sharp_n, \mathrm{n}^\sharp_n)$*:*

$$\mathbf{bind}_{\mathbb{M}^\sharp_\mathcal{R}}(\{(\mathtt{x}_1, e_1), \ldots, (\mathtt{x}_n, e_n)\}, \mathrm{m}^\sharp_{\mathcal{R}^0}) = (\mathrm{e}^\sharp_1, \mathrm{r}^\sharp_n, \mathrm{n}^\sharp_n)$$

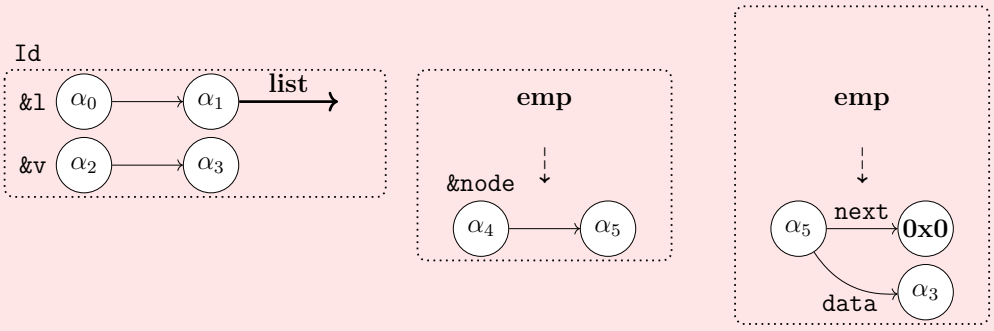
Theorem 8.2 (Soundness of $\mathbf{bind}_{\mathbb{M}^\sharp_\mathcal{R}}$). *Let* $\mathtt{x}_1, \ldots, \mathtt{x}_n \in \mathbb{X}$*,* $e_1, \ldots, e_n \in \mathit{Expr}$ *and* $\mathrm{m}^\sharp_{\mathcal{R}^0} \in \mathbb{M}^\sharp_\mathcal{R}$*. Let* $\mathrm{m}^\sharp_{\mathcal{R}^1} = \mathbf{bind}_{\mathbb{M}^\sharp_\mathcal{R}}(\{(\mathtt{x}_1, e_1), \ldots, (\mathtt{x}_n, e_n)\}, \mathrm{m}^\sharp_{\mathcal{R}^0})$*.*

$$\forall((\mathrm{e}_0, \mathrm{h}_\mathrm{i}),\ (\mathrm{e}_0, \mathrm{h}_\mathrm{o})) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^0})$$

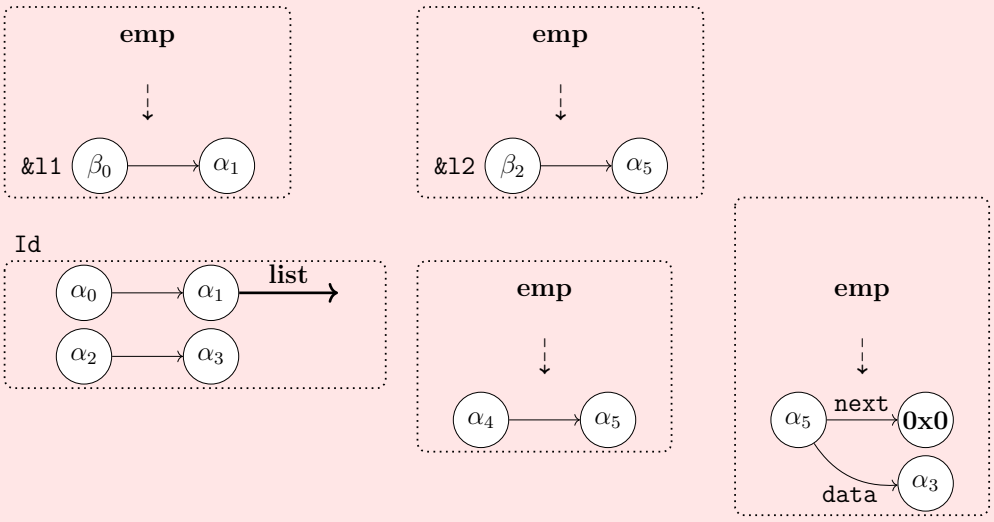
$$\Longrightarrow$$

$$\exists \mathrm{e}_1 \in \mathbb{E}, \quad ((\mathrm{e}_1, \mathrm{h}_\mathrm{i}),\ (\mathrm{e}_1, \mathrm{h}_\mathrm{o} \circledast [\mathcal{L}[\![\mathtt{x}_{1\ldots n}]\!](\mathrm{e}_1, \mathrm{h}_0) \mapsto \mathcal{E}[\![e_{1\ldots n}]\!](\mathrm{e}_0, \mathrm{h}_\mathrm{o})])) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^1})$$

Example 8.4 (Binding). In this example, we consider the binding step for the call to `concat` in the function `add_last` of Figure 2.5 (page 25). The binding is performed from the abstract memory relation below:



The binding creates two new variables `l1` and `l2`, that are parameters of the called function `concat`, that point to the same addresses than `l` and `node`. The resulting abstract environment only contains the variables `l1` and `l2`.



8.3.2 Cut

After the initialization step, the abstract environment of the calling abstract memory relation is only related to the arguments of the called function. Consequently, some parts of the abstract heap relation may be *unreachable* from these arguments. Indeed, it discards the abstract environment of the calling abstract memory relation, and substitutes it by a new one. We name the *reachable abstract heap* the part of the calling abstract heap that is reachable from the arguments of the calling function, after the initialization step. We also name the *unreachable abstract heap* the part of the calling abstract heap that is provably not reachable from the arguments of the calling function.

The goal of splitting the calling abstract heap is multiple. First, it allows to identify which part of the memory cannot be modified, or cannot be even read by the called function. During the composition step, the identity relation of the unreachable abstract heap will be added to the abstract heap relation of the function summary of the called function. Second, the reachable abstract heap is used by the analysis to perform the inclusion checking with the pre-condition of the function summary. If the inclusion holds, this means that the function summary is valid from this calling context. Conversely, if the inclusion does not hold, this means that the function summary does not express a valid relation from this calling context. In this case, the analysis joins the reachable abstract state with the pre-condition of the function summary, and reanalyzes completely the called function from the identity relation of the joined abstract state in order to obtain a new function summary, more general than the previous one.

In this section, we define the function $\mathbf{cut}_{\mathbb{M}^\sharp_{\mathcal{R}}} \in \mathbb{M}^\sharp_{\mathcal{R}} \to \mathbb{M}^\sharp \times \mathbb{H}^\sharp$ that takes an abstract memory relation and returns a pair $(\mathrm{m}^\sharp, \mathrm{h}^\sharp)$, where the abstract memory state $\mathrm{m}^\sharp$ denotes the reachable abstract memory state the input abstract memory relation, and $\mathrm{h}^\sharp$ its unreachable abstract heap.

Getting the Calling Abstract Heap

To be able to get the unreachable and the reachable abstract heaps, it requires to get the *calling abstract heap* of the current abstract heap relation. Let $\mathrm{r}^\sharp$ be an abstract heap relation. For all the triplets $(\mathrm{h_i}, \mathrm{h_o}, \nu)$ that are in the concretization of $\mathrm{r}^\sharp$, the calling abstract heap of $\mathrm{r}^\sharp$ is simply an abstract heap that abstracts $\mathrm{h_o}$. The function $\mathbf{get\_out}_{\mathbb{R}^\sharp} \in \mathbb{R}^\sharp \to \mathbb{H}^\sharp$ returns such abstract heaps.

Definition 8.10 (Definition of $\mathbf{get\_out}_{\mathbb{R}^\sharp}$). *The function* $\mathbf{get\_out}_{\mathbb{R}^\sharp} \in \mathbb{R}^\sharp \to \mathbb{H}^\sharp$ *is defined by induction on abstract heap relations as follows:*

$$\begin{aligned}
\mathbf{get_out}_{\mathbb{R}^\sharp}(\mathtt{Id}(\mathrm{h}^\sharp)) &= \mathrm{h}^\sharp \\
\mathbf{get_out}_{\mathbb{R}^\sharp}([\mathrm{h}^\sharp_\mathrm{i} \dashrightarrow \mathrm{h}^\sharp_\mathrm{o}]_{\mathrm{t}^\sharp}) &= \mathrm{h}^\sharp_\mathrm{o} \\
\mathbf{get_out}_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp_0 *_\mathtt{R} \mathrm{r}^\sharp_1) &= \mathbf{get_out}_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp_0) *_\mathtt{S} \mathbf{get_out}_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp_1)
\end{aligned}$$

This definition is trivial for each of these cases using the definition of $\gamma_{\mathbb{R}^\sharp}$ (page 46).

Theorem 8.3 (Soundness of $\mathbf{get\_out}_{\mathbb{R}^\sharp}$). *Let* $\mathrm{r}^\sharp \in \mathbb{R}^\sharp$ *and* $\mathrm{h}^\sharp \in \mathbb{H}^\sharp$.

If $\mathbf{get\_out}_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) = \mathrm{h}^\sharp$, *then:*

$$\forall (\mathrm{h_i}, \mathrm{h_o}, \nu) \in \gamma_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) \implies (\mathrm{h_o}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp)$$

Cutting the Calling Abstract Heap

This step consists in a depth-first search of an abstract heap $\mathrm{h}^\sharp$ from a set of symbolic values. All the regions of $\mathrm{h}^\sharp$ visited during this search describe the *reachable* abstract heap and the non-visited regions describe the *unreachable* abstract heap of the called function. The depth-first search is performed by the function $\mathbf{cut}_{\mathbb{H}^\sharp} \in \mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathcal{P}_{\mathrm{fin}}(\mathbb{V}^\sharp) \times \mathcal{P}_{\mathrm{fin}}(\mathbb{V}^\sharp) \to \mathbb{H}^\sharp \times \mathbb{H}^\sharp \times \mathcal{P}_{\mathrm{fin}}(\mathbb{V}^\sharp)$. At any point in the search, $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_\mathrm{u}, \mathrm{h}^\sharp_\mathrm{r}, \mathscr{E}, \mathscr{F})$ means that $\mathrm{h}^\sharp_\mathrm{u}$ is the unreachable abstract heap, $\mathrm{h}^\sharp_\mathrm{r}$ is the reachable abstract heap, $\mathscr{E}$ is the set of symbolic values to visit, and $\mathscr{F}$ is the set of the visited symbolic values. The depth-first search initially starts with $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp, \mathbf{emp}, \mathbf{im}(\mathrm{e}^\sharp), \{\})$, where $\mathrm{h}^\sharp$ is the abstract heap to cut, and $\mathbf{im}(\mathrm{e}^\sharp)$ is the set of symbolic values corresponding to the addresses of the variable in the abstract environment $\mathrm{e}^\sharp$.

Definition 8.11 (Cut of abstract heaps). *We define by induction over the abstract heap* $\mathrm{h}^\sharp_\mathrm{u}$ *and over the set of symbolic values* $\mathscr{E}$*, the cut function* $\mathbf{cut}_{\mathbb{H}^\sharp}$ *of abstract heaps:*

- $\mathbf{cut}_{\mathbb{H}^\sharp}(\alpha \cdot \mathtt{f} \mapsto \beta *_\mathtt{s} \mathrm{h}^\sharp_\mathrm{u},\ \mathrm{h}^\sharp_\mathrm{r},\ \{\alpha\} \uplus \mathscr{E},\ \mathscr{F}) =$
  $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_\mathrm{u},\ \alpha \cdot \mathtt{f} \mapsto \beta *_\mathtt{s} \mathrm{h}^\sharp_\mathrm{r},\ \{\beta\} \cup (\{\alpha\} \uplus \mathscr{E}),\ \mathscr{F})$
- $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathbf{listseg}(\alpha, \beta) *_\mathtt{s} \mathrm{h}^\sharp_\mathrm{u},\ \mathrm{h}^\sharp_\mathrm{r},\ \{\alpha\} \uplus \mathscr{E},\ \mathscr{F}) =$
  $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_\mathrm{u},\ \mathbf{listseg}(\alpha, \beta) *_\mathtt{s} \mathrm{h}^\sharp_\mathrm{r},\ \{\beta\} \cup \mathscr{E},\ \{\alpha\} \cup \mathscr{F})$
- $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathbf{list}(\alpha) *_\mathtt{s} \mathrm{h}^\sharp_\mathrm{u},\ \mathrm{h}^\sharp_\mathrm{r},\ \{\alpha\} \uplus \mathscr{E},\ \mathscr{F}) =$
  $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_\mathrm{u},\ \mathbf{list}(\alpha) *_\mathtt{s} \mathrm{h}^\sharp_\mathrm{r},\ \mathscr{E},\ \{\alpha\} \cup \mathscr{F})$
- $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_\mathrm{u},\ \mathrm{h}^\sharp_\mathrm{r},\ \{\alpha\} \uplus \mathscr{E},\ \mathscr{F}) =$ *if there is no predicate attached to* α *in* $\mathrm{h}^\sharp_\mathrm{u}$
  $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_\mathrm{u},\ \mathrm{h}^\sharp_\mathrm{r},\ \mathscr{E},\ \{\alpha\} \cup \mathscr{F})$
- $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_\mathrm{u},\ \mathrm{h}^\sharp_\mathrm{r},\ \{\},\ \mathscr{F}) = (\mathrm{h}^\sharp_\mathrm{u},\ \mathrm{h}^\sharp_\mathrm{r},\ \mathscr{F})$

All along the depth-first search, some parts of $\mathrm{h}^\sharp$ will be consumed and will enrich the reachable abstract heap. The depth-first search ends when $\mathscr{E}$ is empty and returns the unreachable and the reachable abstract heaps with the set of all visited symbolic values.

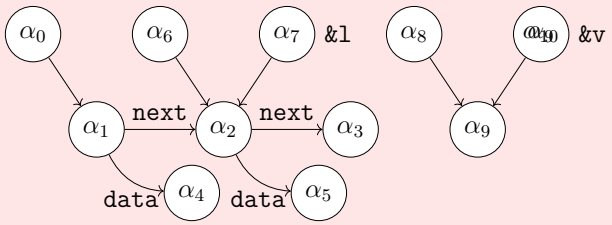
Theorem 8.4 (Soundness of $\mathbf{cut}_{\mathbb{H}^\sharp}$). *Let* $\mathrm{h}^\sharp_{\mathrm{u},0}, \mathrm{h}^\sharp_{\mathrm{u},1}, \mathrm{h}^\sharp_{\mathrm{r},0}\mathrm{h}^\sharp_{\mathrm{r},1} \in \mathbb{H}^\sharp$ *and* $\mathscr{E}_0, \mathscr{F}_0, \mathscr{F}_1 \in \mathcal{P}_\mathrm{fin}(\mathbb{V}^\sharp)$*. If* $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{u},0}, \mathrm{h}^\sharp_{\mathrm{r},0}, \mathscr{E}_0, \mathscr{F}_0) = (\mathrm{h}^\sharp_{\mathrm{u},1}, \mathrm{h}^\sharp_{\mathrm{r},1}, \mathscr{F}_1)$ *then:*

$$\forall \mathrm{h}_{\mathrm{u},0}, \mathrm{h}_{\mathrm{u},1}, \mathrm{h}_{\mathrm{r},0}\mathrm{h}_{\mathrm{r},1} \in \mathbb{H}, \exists \nu \in \mathbb{V}^\sharp \to \mathbb{V} \text{ such that:}$$

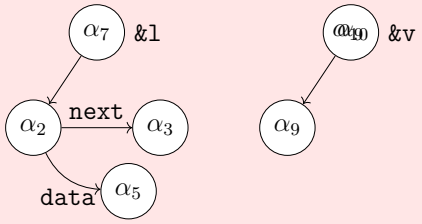
$$\begin{array}{l}
(\mathrm{h}_{\mathrm{u},0}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{u},0}) \wedge (\mathrm{h}_{\mathrm{u},1}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{u},1}) \wedge \\
(\mathrm{h}_{\mathrm{r},0}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{r},0}) \wedge (\mathrm{h}_{\mathrm{r},1}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{r},1}) \\
\implies (\mathrm{h}_{\mathrm{u},0} \circledast \mathrm{h}_{\mathrm{r},0}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{u},0} *_\mathtt{s} \mathrm{h}^\sharp_{\mathrm{r},0}) \wedge \\
\quad (\mathrm{h}_{\mathrm{u},1} \circledast \mathrm{h}_{\mathrm{r},1}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{u},1} *_\mathtt{s} \mathrm{h}^\sharp_{\mathrm{r},1}) \wedge \\
\quad \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{u},0} *_\mathtt{s} \mathrm{h}^\sharp_{\mathrm{r},0}) = \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp_{\mathrm{u},1} *_\mathtt{s} \mathrm{h}^\sharp_{\mathrm{r},1}) \wedge \\
\quad \{\nu(\alpha) \mid \alpha \in \mathscr{F}_1\} \subseteq \mathbf{dom}(\mathrm{h}_{\mathrm{r},1}) \cup \mathbf{im}(\mathrm{h}_{\mathrm{r},1})
\end{array}$$

This soundness theorem means that the reachable and unreachable abstract heaps must describe distinct regions, that $\mathbf{cut}_{\mathbb{H}^\sharp}$ moves some regions of the unreachable abstract heap into the reachable abstract heap, and that all the concrete values of $\mathscr{F}_1$ must be in the concrete heaps of $\mathrm{h}^\sharp_{\mathrm{r},1}$

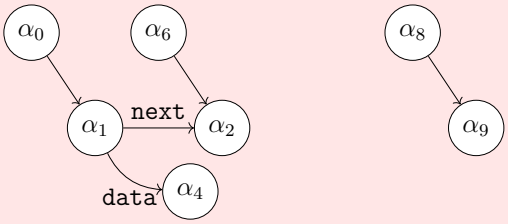
Example 8.5 (Cut an abstract heap). In this example, we consider the function call at line 4 of Figure 2.6 (page 28). We consider the initialization step already performed, and that $\mathbf{get\_out}_{\mathbb{R}^\sharp}$ returns the abstract heap $\mathrm{h}^\sharp$ graphically represented below. The symbolic values α_0, α_6 and α_8 respectively denote the addresses of variables `l1`, `l2` and `z`, that where in the abstract environment before the initialization step. For clarity, we ommit the symbolic values denoting the addresses of variables `x` and `y`. We provide the result for $\mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp, \mathbf{emp}, \{\alpha_7, \alpha_{10}\}, \{\})$.



On this abstract heap, $\mathbf{cut}_{\mathbb{H}^\sharp}$ will produce the following reachable abstract heap:



and the following unreachable abstract heap:



The resulting set of visited symbolic values is:

$$\{\alpha_7, \alpha_2, \alpha_3, \alpha_5, \alpha_{10}, \alpha_9\}$$

Remark 5. *The function* $\mathbf{cut}_{\mathbb{H}^\sharp}$ *removes elements that are not provably reachable. It may also remove some elements that are reachable in some concrete states. For instance in Example 8.5, the concretization of the heap may be such that* α_9 *and* α_1 *are actually equal. However, this does not make our analysis unsound. Indeed, if the analysis of the function body depends on this equality, it will fail anyway (as the equality is absent from the abstract state). Thus, it loses no further information. This is exactly the same*

phenomenon explained in [GBC06].

Then, the analysis requires a function $\mathbf{cut}_{\mathbb{N}^\sharp} \in \mathbb{N}^\sharp \times \mathcal{P}_{\text{fin}}(\mathbb{V}^\sharp) \to \mathbb{N}^\sharp$ to perform the cut in the numerical abstract domain. The function $\mathbf{cut}_{\mathbb{N}^\sharp}(\mathrm{n}_0^\sharp, \mathscr{F})$ should simply discard all occurences of the symbolic values from $\mathrm{n}_0^\sharp$ that are not in $\mathscr{F}$.

Assumption 8.1 (Soundness of $\mathbf{cut}_{\mathbb{N}^\sharp}$). *The soundness assumption of the function* $\mathbf{cut}_{\mathbb{N}^\sharp} \in \mathbb{N}^\sharp \times \mathcal{P}_{\text{fin}}(\mathbb{V}^\sharp) \to \mathbb{N}^\sharp$ *is:*

$$\mathbf{cut}_{\mathbb{N}^\sharp}(\mathrm{n}_0^\sharp, \mathscr{F}) = \mathrm{n}_1^\sharp \quad \Longrightarrow \quad \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_0^\sharp) \subseteq \gamma_{\mathbb{N}^\sharp}(\mathrm{n}_1^\sharp)$$

That is, $\mathbf{cut}_{\mathbb{N}^\sharp}$ looses information about some symbolic values, for instance those that are not in $\mathscr{F}$.

We can now define the cut function $\mathbf{cut}_{\mathbb{M}^\sharp_\mathcal{R}} \in \mathbb{M}^\sharp_\mathcal{R} \to \mathbb{M}^\sharp \times \mathbb{H}^\sharp$ of abstract memory relations.

Definition 8.12. *Let* $\mathrm{m}^\sharp_\mathcal{R} = (\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}_0^\sharp) \in \mathbb{M}^\sharp_\mathcal{R}$.
if:

$$\mathbf{get\_out}_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp) = \mathrm{h}^\sharp \text{ and } \mathbf{cut}_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp, \mathbf{emp}, \mathbf{im}(\mathrm{e}^\sharp), \{\}) = (\mathrm{h}_\mathrm{u}^\sharp, \mathrm{h}_\mathrm{r}^\sharp, \mathscr{F})$$

$$\text{and } \mathbf{cut}_{\mathbb{N}^\sharp}(\mathrm{n}_0^\sharp, \mathscr{F}) = \mathrm{n}_1^\sharp$$

then:

$$\mathbf{cut}_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}_0^\sharp) = ((\mathrm{e}^\sharp, \mathrm{h}_\mathrm{r}^\sharp, \mathrm{n}_1^\sharp), \mathrm{h}_\mathrm{u}^\sharp)$$

Theorem 8.5 (Soundness of cut). *Let* $\mathrm{e}^\sharp \in \mathbb{E}^\sharp$, $\mathrm{r}^\sharp \in \mathbb{R}^\sharp$, $\mathrm{n}_0^\sharp, \mathrm{n}_1^\sharp \in \mathbb{N}^\sharp$ *and* $\mathrm{h}_\mathrm{r}^\sharp, \mathrm{h}_\mathrm{u}^\sharp \in \mathbb{H}^\sharp$.

$$\text{if } \mathbf{cut}_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}_0^\sharp) = ((\mathrm{e}^\sharp, \mathrm{h}_\mathrm{r}^\sharp, \mathrm{n}_1^\sharp), \mathrm{h}_\mathrm{u}^\sharp)$$

$$\text{then } \gamma_\mathbb{M}(\mathrm{e}^\sharp, \mathbf{get\_out}_{\mathbb{R}^\sharp}(\mathrm{r}^\sharp), \mathrm{n}_0^\sharp) \subseteq \gamma_\mathbb{M}(\mathrm{e}^\sharp, \mathrm{h}_\mathrm{r}^\sharp *_\mathtt{s} \mathrm{h}_\mathrm{u}^\sharp, \mathrm{n}_1^\sharp)$$

Discussion for shared data structures

In the definition of $\mathbf{cut}_{\mathbb{H}^\sharp}$, we only used the list inductive ($\mathbf{list}(\alpha)$) and the segment ($\mathbf{listseg}(\alpha, \beta)$) predicates. As singly linked lists are *unshared data structures* (they do not contain an element that can be pointed by several other elements), this definition of $\mathbf{cut}_{\mathbb{H}^\sharp}$ is sound. Indeed, the visited part of the depth-first search is well the unreachable abstract heap. However, the definition of $\mathbf{cut}_{\mathbb{H}^\sharp}$ could be extended to take into account *shared data structures*, that may contain an element that can be pointed by other elements.

For instance, data structures with backward pointers, like double linked lists, are shared data structures. Such shared data structures have *structured sharing*, as they have

exactly one sharing. They are often abstracted with inductive and segment predicates that contain additional arguments that refer to the previous element of the data structure, that is abstracted by an other predicate. For example, the inductive predicate (this is not a segment predicate) of doubly linked lists is defined as follows:

$$\begin{aligned}&\mathbf{dll}(\alpha,\gamma) ::= \\ &\quad (\mathbf{emp},\ \alpha = \mathbf{0x0}) \\ &\vee\quad (\alpha \cdot \mathtt{prev} \mapsto \gamma *_{\mathtt{s}} \alpha \cdot \mathtt{next} \mapsto \beta *_{\mathtt{s}} \mathbf{dll}(\beta,\alpha),\ \alpha \neq \mathbf{0x0})\end{aligned}$$

In this definition, we can see that the `prev` field of the list points-to a symbolic value γ that is not folded in the inductive definition In turn, during the depth-first such, it is crucial to visit this element, as it belongs to the reachable abstract heap.

Also, [LRC15] defines an elegant abstraction based on separation logic for data structures with *unstructured sharing* (that have an unbounded number of sharing), such as acyclic graphs. In this abstraction, inductive definitions are parameterized by sets of symbolic values, that denote outer elements that are reachable from the folded region. Thus, the depth-first search could take into account the sets constraints of this kind of abstraction.

8.3.3 Instantiation of the Function Summary

Before performing abstract composition on the abstract relation of a function summary, many steps are necessary, included the *instantiation* step that we define in this section.

To support function calls that may generate fresh allocation, the analysis cannot use the same abstract relation for the abstract composition several times. Indeed, we saw in Chapter 5 that the abstract evaluation of memory allocation generates *fresh symbolic values*. If an abstract relation corresponding to a function performing memory allocation is composed many times, then the abstract composition produces false abstract relations. Indeed, using the same abstract relation does not generate fresh symbolic values, but duplicates existing ones.

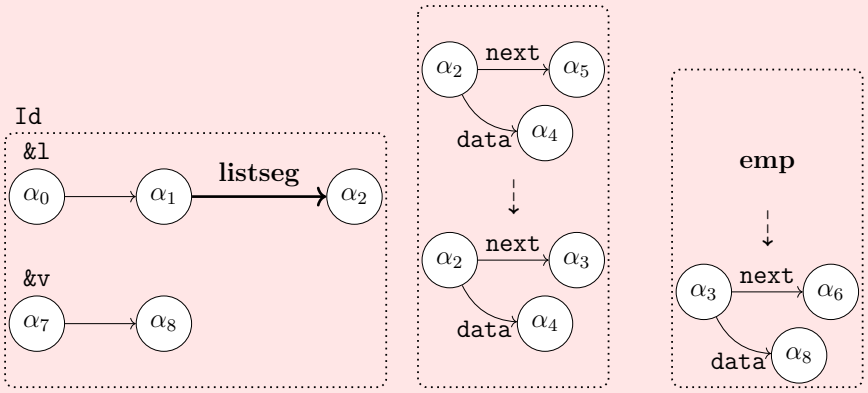
In order to be able to compose functions that may allocate memory cells, the analyse should use a *different* abstract relation at each abstract composition. To obtain a new abstract relation for a function, the analysis generates a fresh symbolic for each symbolic value of the abstract relation of the called function. We name this operation *instantiation*.

The function $\mathbf{inst}_{\mathbb{M}^\sharp_\mathcal{R}} \in \mathbb{M}^\sharp_\mathcal{R} \to \mathbb{M}^\sharp_\mathcal{R}$ instantiates an abstract memory relation $\mathrm{m}^\sharp_\mathcal{R}$. This function simply generates a new abstract memory relation, where each symbolic value in $\mathrm{m}^\sharp_\mathcal{R}$ has been replaced by a fresh symbolic value. As this operation is totally obvious, we just provide its soundness theorem and an example.

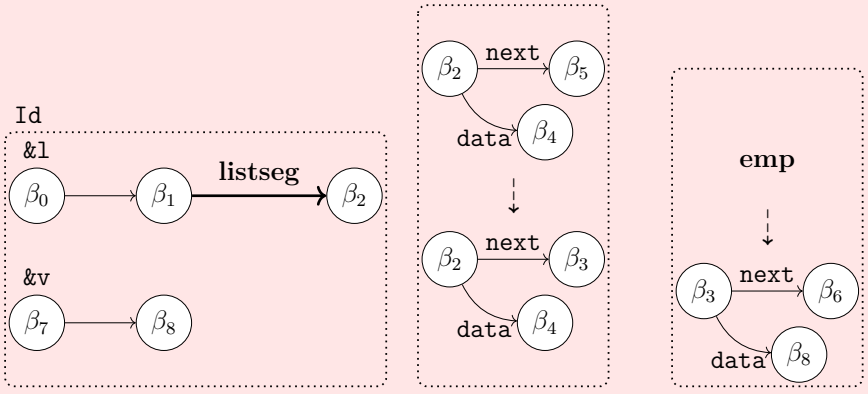
Theorem 8.6 (Soundness of $\mathbf{inst}_{\mathbb{M}^\sharp_\mathcal{R}}$). *Let* $\mathrm{m}^\sharp_{\mathcal{R}^0}, \mathrm{m}^\sharp_{\mathcal{R}^1} \in \mathbb{M}^\sharp_\mathcal{R}$. *The function* $\mathbf{inst}_{\mathbb{M}^\sharp_\mathcal{R}}$ *is sound if and only if:*

$$\gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^0}) \subseteq \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathbf{inst}_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_{\mathcal{R}^0}))$$

Example 8.6 (Instantiation of an abstract memory relation). In this example, we give the instantation of the abstract memory relation corresponding to the case where the list `l` is not empty in the function `add_last` of Figure 2.5 (page 25).



From the abstract memory relation above, the instantation step will produce:



8.3.4 Compositional Frame Rule

In this section, we define the principle of the *compositional frame rule*. When a function is called, it can have only effects on a specific part of the calling memory. This specific part is the reachable heap from the actual arguments of the called function. Consequently, the unreachable heap cannot be modified, or even read by the called function.

The compositional frame rule is an application of this property, and allows to preserve identically the unreachable heap during the abstract composition.

We now discuss how to apply it. Let $\mathrm{r}_0^\sharp$ and $\mathrm{r}_1^\sharp$ be two abstract heap relations, where $\mathrm{r}_0^\sharp$ is the caller and $\mathrm{r}_1^\sharp$ the called. A first intuition is to split $\mathrm{r}_0^\sharp$ into two abstract heap relations $\mathrm{r}_2^\sharp$ and $\mathrm{r}_3^\sharp$, such that $\mathrm{r}_0^\sharp = \mathrm{r}_2^\sharp *_{\mathtt{R}} \mathrm{r}_3^\sharp$, where $\mathrm{r}_2^\sharp$ is only related to the reachable heap and $\mathrm{r}_3^\sharp$ is related to the unreachable heap. Then, to compute an abstract heap relation that describes the composition of $\mathrm{r}_0^\sharp$ by $\mathrm{r}_1^\sharp$, we could use $(\mathrm{r}_2^\sharp \circ_{\mathbb{R}^\sharp} \mathrm{r}_1^\sharp) *_{\mathtt{R}} \mathrm{r}_3^\sharp$. However, this solution cannot be applied. Indeed, Theorem 6.1 (page 91) allows to merge transform-into relations but does not allow to split them: we cannot split $\mathrm{r}_0^\sharp$ into $\mathrm{r}_2^\sharp *_{\mathtt{R}} \mathrm{r}_3^\sharp$ if $\mathrm{r}_0^\sharp$ is a transform-into relation.

Another solution, that is less local but that can be applied in any cases, is to add the identity relation of the unreachable abstract heap of $\mathrm{r}_0^\sharp$ to $\mathrm{r}_1^\sharp$. In Section 8.3.2, we defined a function that extracts both the reachable and the unreachable abstract heaps. The analysis uses this function to get the unreachable abstract heap $\mathrm{h}_u^\sharp$ of $\mathrm{r}_0^\sharp$, and applies $\mathrm{r}_0^\sharp \circ_{\mathbb{R}^\sharp} (\mathrm{r}_1^\sharp *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}_u^\sharp))$. This operation is implemented by the function $\mathbf{enrich}_{\mathbb{M}_\mathcal{R}^\sharp} \in \mathbb{M}_\mathcal{R}^\sharp \times \mathbb{H}^\sharp \to \mathbb{M}_\mathcal{R}^\sharp$, that takes an abstract memory relation and an abstract heap, and that returns a new abstract memory relation where we added the identity relation of the given abstract heap.

Definition 8.13 (Enrich an abstract memory relation). *The function* $\mathbf{enrich}_{\mathbb{M}_\mathcal{R}^\sharp}$ *is defined as follows:*

$$\mathbf{enrich}_{\mathbb{M}_\mathcal{R}^\sharp}((\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp), \mathrm{h}^\sharp) = (\mathrm{e}^\sharp, \mathrm{r}^\sharp *_{\mathtt{R}} \mathtt{Id}(\mathrm{h}^\sharp), \mathrm{n}^\sharp)$$

Theorem 8.7 (Soundness of enrich). *Let* $\mathrm{m}_{\mathcal{R}^1}^\sharp \in \mathbb{M}_\mathcal{R}^\sharp$ *and* $\mathrm{h}^\sharp \in \mathbb{H}^\sharp$. *If* $\mathbf{enrich}_{\mathbb{M}_\mathcal{R}^\sharp}(\mathrm{m}_{\mathcal{R}^1}^\sharp, \mathrm{h}^\sharp) = \mathrm{m}_{\mathcal{R}^2}^\sharp$, *then:*

$$((\mathrm{e}, \mathrm{h_i}),\ (\mathrm{e}, \mathrm{h_o})) \in \gamma_{\mathbb{M}_\mathcal{R}^\sharp}(\mathrm{m}_{\mathcal{R}^1}^\sharp) \wedge (\mathrm{h}, \nu) \in \gamma_{\mathbb{H}^\sharp}(\mathrm{h}^\sharp) \implies ((\mathrm{e}, \mathrm{h_i} \circledast \mathrm{h}), (\mathrm{e}, \mathrm{h_o} \circledast \mathrm{h})) \in \gamma_{\mathbb{M}_\mathcal{R}^\sharp}(\mathrm{m}_{\mathcal{R}^2}^\sharp)$$

Before composing $\mathrm{m}_{\mathcal{R}^0}^\sharp$ with $\mathrm{m}_{\mathcal{R}^1}^\sharp$, the analysis adds the identity relation of the unreachable abstract heap of $\mathrm{m}_{\mathcal{R}^0}^\sharp$ to $\mathrm{m}_{\mathcal{R}^1}^\sharp$.

8.3.5 Abstract Call

The analysis of abstract function calls performs all the operations introduced in this section.

Informal Definition

We start with an informal definition of the analysis of abstract function calls, where we detail each step.

Preparation of the calling relation to the composition. It first proceeds to the initialization of the function call (Section 8.3.1) and then cuts (Section 8.3.2) the calling abstract heap to extract the reachable and the unreachable abstract heaps from the actual arguments of the called function.

Checking process of the function summary. The next step is probably the most crucial of the algorithm. As shown in Figure 8.1 and Figure 8.2, it tests if the reachable abstract heap is included in the pre-condition of the function summaries table $\tau^\sharp$. If the inclusion does not hold, this means that $\tau^\sharp$ does not express a valid relation from the actual calling context.

Computation of a new abstract relation. If the reachable abstract heap is included in the pre-condition, this step is not performed, as this means that the function summary already describes a relation valid from the calling context. Otherwise, the algorithm computes a new summary for the called function and puts this new summary instead of the previous one in $\tau^\sharp$. To do this, the algorithm joins the pre-condition with the reachable abstract heap, in order to obtain a new abstract memory that over approximates all concrete memory states described by both of them. This joined abstract memory is the new pre-condition of the function summary. To obtain the new relation, the algorithm reanalyzes the body of the function, from the *identity relation* of the new pre-condition.

Preparation of the function summary to the composition. At this step, the function summaries table contains a valid relation for the calling context, the analysis can thus prepare the relation of the called function to the abstract composition. It instantiates (Section 8.3.3) it, and enriches (Section 8.3.4) it with the unreachable abstract heap of the calling context.

Initialization of the abstract composition. This is the most technical step of the abstract function call. Remind in Chapter 7, we saw that the abstract composition requires an initial pair of renaming functions. We now explain why it is required and how it is initialized. In standard lattice operations such as join or inclusion checking, we initialize the renaming functions directly from the abstract environments, in order to know which symbolic values to map together. Regarding to the abstract composition, the principle is the same, except that the initialization of the function call (Section 8.3.1) deletes the abstract environment of calling abstract relation, and creates a new one only related to the parameters of the called function. All the symbolic values corresponding to the addresses of the deleted variables belong to the unreachable abstract heap of the calling context, that is itself added to the abstract relation of the called function, by the compositional frame rule (Section 8.3.4). The abstract composition cannot know how to map these symbolic values, it thus requires to input an initial pair of renaming functions.

To create such a pair, the algorithm first extracts the abstract environment from an abstract memory relation with the function $\mathbf{get\_env}_{\mathbb{M}^\sharp_\mathcal{R}} \in \mathbb{M}^\sharp_\mathcal{R} \to \mathbb{E}^\sharp$, such that $\mathbf{get\_env}_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{e}^\sharp, \mathrm{r}^\sharp, \mathrm{n}^\sharp) = \mathrm{e}^\sharp$. Second, it uses the function $\mathbf{init}_{\mathbb{E}^\sharp} \in \mathbb{E}^\sharp \to (\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2$ to create a pair of renaming functions from an abstract environment. This function is defined as follows: $\mathbf{init}_{\mathbb{E}^\sharp}(\mathrm{e}^\sharp) = \Phi$, such that $\forall \mathtt{x} \in \mathbb{X}, \Phi(\mathrm{e}^\sharp(\mathtt{x}), \mathrm{e}^\sharp(\mathtt{x})) = \mathrm{e}^\sharp(\mathtt{x})$. The algorithm creates the initial pair of the abstract composition by performing $\mathbf{init}_{\mathbb{E}^\sharp}(\mathbf{get\_env}_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_\mathcal{R}))$, where $\mathrm{m}^\sharp_\mathcal{R}$ is the "original" calling abstract memory relation (the one before the initialization step).

Abstract composition. After creating such a pair of renaming functions, the algorithm can effectively perform the abstract composition between the calling abstract relation and the abstract relation of the function summary.

Formal Definition

We now give the formal definition of the analysis of abstract function calls. It is performed directly at the disjunction of abstract memory relations level. Consequently, it is built upon the implementation of the functions defined in this chapter but at the disjunction level ($\mathbf{bind}_{\mathbb{R}^\vee}$, $\mathbf{cut}_{\mathbb{R}^\vee}$, $\mathbf{inst}_{\mathbb{R}^\vee}$, $\mathbf{enrich}_{\mathbb{R}^\vee}$, $\mathbf{init}_{\mathbb{R}^\vee}$, $\mathbf{get\_env}_{\mathbb{R}^\vee}$, $\mathbf{comp}_{\mathbb{R}^\vee}$). Their soundness theorem is similar to the respective soundness theorem at abstract memory relations level. Their interface is given below:

$$
\begin{array}{rcll}
\mathbf{comp}_{\mathbb{R}^\vee} & \in & \mathcal{P}_{\mathrm{fin}}((\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2) \times \mathbb{R}^\vee \times \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee \\
\mathbf{bind}_{\mathbb{R}^\vee} & \in & \mathcal{P}_{\mathrm{fin}}(\mathbb{X} \times \mathcal{E}\mathcal{X}\mathcal{P}r) \times \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee \\
\mathbf{cut}_{\mathbb{R}^\vee} & \in & \mathbb{R}^\vee & \to \quad \mathbb{M}^\vee \times \mathcal{P}_{\mathrm{fin}}(\mathbb{H}^\sharp) \\
\mathbf{inst}_{\mathbb{R}^\vee} & \in & \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee \\
\mathbf{enrich}_{\mathbb{R}^\vee} & \in & \mathbb{R}^\vee \times \mathcal{P}_{\mathrm{fin}}(\mathbb{H}^\sharp) & \to \quad \mathbb{R}^\vee \\
\mathbf{get\_env}_{\mathbb{R}^\vee} & \in & \mathbb{R}^\vee & \to \quad \mathcal{P}_{\mathrm{fin}}(\mathbb{E}^\sharp) \\
\mathbf{init}_{\mathbb{R}^\vee} & \in & \mathcal{P}_{\mathrm{fin}}(\mathbb{E}^\sharp) & \to \quad \mathcal{P}_{\mathrm{fin}}((\mathbb{V}^\sharp \to \mathbb{V}^\sharp)^2) \\
\mathbf{return}_{\mathbb{R}^\vee} & \in & \mathbb{R}^\vee \times \mathcal{P}_{\mathrm{fin}}(\mathbb{E}^\sharp) & \to \quad \mathbb{R}^\vee \\
\mathbf{call}_{\mathbb{R}^\vee} & \in & \mathcal{F}un \times \mathcal{P}_{\mathrm{fin}}(\mathcal{E}\mathcal{X}\mathcal{P}r) \times \mathbb{R}^\vee & \to \quad \mathbb{R}^\vee
\end{array}
$$

Abstract function calls, requires inclusion checking $\mathbf{isle}_{\mathbb{M}^\vee} \in \mathbb{M}^\vee \times \mathbb{M}^\vee \to \{\mathbf{true}, \mathbf{false}\}$ and join operator $\mathbf{join}_{\mathbb{M}^\vee} \in \mathbb{M}^\vee \times \mathbb{M}^\vee \to \mathbb{M}^\vee$ for disjunction of abstract memory states, in order to compare and weaken pre-conditions.

It also requires the function $\mathbf{id}_{\mathbb{M}^\vee} \in \mathbb{M}^\vee \to \mathbb{R}^\vee$, that lifts a disjunction of abstract memory states into a disjunction of abstract memory relations (the obtained relations are simply the identity relations of the given memory states):

$$\mathbf{id}_{\mathbb{M}^\vee}(\mathtt{m}^\vee) = \{(\mathrm{e}^\sharp, \mathtt{Id}(\mathrm{h}^\sharp), \mathrm{n}^\sharp) \mid (\mathrm{e}^\sharp, \mathrm{h}^\sharp, \mathrm{n}^\sharp) \in \mathtt{m}^\vee\}$$

Definition 8.14 (Abstract function calls). *Let* $e_1, \ldots, e_n \in \mathcal{E}xpr$, $\mathbb{r}_0^\vee \in \mathbb{R}^\vee$, $\tau^\sharp \in \mathcal{T}^\sharp$, *and* $f \in \mathcal{F}un$ *such that:*

$$f(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{c; \mathbf{ret}\} \in \mathcal{C}md$$

The function $\mathbf{call}_{\mathbb{R}^\vee}$ *is defined by the following algorithm:*

$$\begin{array}{l}
\mathbf{call}_{\mathbb{R}^\vee}(f, \{e_1, \ldots, e_n\}, \mathbb{r}_0^\vee) = \\
\quad \mathit{Let}\ (\mathbb{m}_f^\vee, \mathbb{r}_f^\vee) = \tau^\sharp(f)\ \mathit{in} \\
\quad \mathit{Let}\ \mathbb{r}_1^\vee = \mathbf{bind}_{\mathbb{R}^\vee}(\{(\mathtt{x}_1, e_1), \ldots, (\mathtt{x}_n, e_n)\}, \mathbb{r}_0^\vee)\ \mathit{in} \\
\quad \mathit{Let}\ (\mathbb{m}^\vee, \mathrm{H}^\sharp) = \mathbf{cut}_{\mathbb{R}^\vee}(\mathbb{r}_1^\vee)\ \mathit{in} \\
\quad \mathit{If}\ \mathbf{isle}_{\mathbb{M}^\vee}(\mathbb{m}^\vee, \mathbb{m}_f^\vee) = \mathbf{false} \\
\quad \mathit{Then}\ (\\
\qquad \mathit{Let}\ \mathbb{m}_f^\vee = \mathbf{join}_{\mathbb{M}^\vee}(\mathbb{m}^\vee, \mathbb{m}_f^\vee)\ \mathit{in} \\
\qquad \mathit{Let}\ \mathbb{r}_f^\vee = [\![c]\!]_{\mathcal{R}}^\sharp(\mathbf{id}_{\mathbb{M}^\vee}(\mathbb{m}_f^\vee))\ \mathit{in} \\
\qquad \tau^\sharp(f) \leftarrow (\mathbb{m}_f^\vee, \mathbb{r}_f^\vee) \\
\quad) \\
\quad \mathit{Let}\ (\mathbb{m}_f^\vee, \mathbb{r}_f^\vee) = \tau^\sharp(f)\ \mathit{in} \\
\quad \mathit{Let}\ \mathbb{r}_f^\vee = \mathbf{inst}_{\mathbb{R}^\vee}(\mathbb{r}_f^\vee)\ \mathit{in} \\
\quad \mathit{Let}\ \mathbb{r}_f^\vee = \mathbf{enrich}_{\mathbb{R}^\vee}(\mathbb{r}_f^\vee, \mathrm{H}^\sharp)\ \mathit{in} \\
\quad \mathbf{comp}_{\mathbb{R}^\vee}(\mathbf{init}_{\mathbb{R}^\vee}(\mathbf{get_env}_{\mathbb{R}^\vee}(\mathbb{r}_0^\vee)),\ \mathbb{r}_1^\vee,\ \mathbb{r}_f^\vee)
\end{array}$$

Theorem 8.8 (Soundness of $\mathbf{call}_{\mathbb{R}^\vee}$). *Let $f \in \mathcal{Fun}$, $e_1, \ldots, e_n \in \mathcal{Expr}$, $\mathbb{r}^\vee \in \mathbb{R}^\vee$, $\mathbb{m}_0^\vee \in \mathbb{M}^\vee$, $\mathbb{r}_0^\vee \in \mathbb{R}^\vee$, and $f \in \mathcal{T}^\sharp$ such that:*

$$f(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{c; \mathbf{ret}\} \in \mathcal{Cmd} \text{ and } \tau^\sharp(f) = (\mathbb{m}_0^\vee, \mathbb{r}_0^\vee)$$

Then:

1. *Soundness of the result*

$$\begin{array}{l}\forall((\mathrm{e}, \mathrm{h}_0), (\mathrm{e}, \mathrm{h}_1)) \in \gamma_{\mathbb{R}^\vee}(\mathbb{r}^\vee),\ \exists \mathrm{e}' \in \mathbb{E},\ \mathrm{h}', \mathrm{h}_2 \in \mathbb{H},\ \textit{such that:}\\ \mathbf{im}(\mathrm{e}') = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}\\ \wedge \mathrm{h}' = [\mathrm{e}'(\mathtt{x}_1) \mapsto \mathcal{E}[\![e_1]\!](\mathrm{e}, \mathrm{h}_1), \ldots, \mathrm{e}'(\mathtt{x}_n) \mapsto \mathcal{E}[\![e_n]\!](\mathrm{e}, \mathrm{h}_1)]\\ \wedge((\mathrm{e}', \mathrm{h}_1 \circledast \mathrm{h}'), (\mathrm{e}', \mathrm{h}_2 \circledast \mathrm{h}')) \in [\![c]\!]_{\mathcal{R}}\\ \\ \Longrightarrow ((\mathrm{e}', \mathrm{h}_0), (\mathrm{e}', \mathrm{h}_2 \circledast \mathrm{h}')) \in \gamma_{\mathbb{R}^\vee}(\mathbf{call}_{\mathbb{R}^\vee}(f, \{e_1, \ldots, e_n\}, \mathbb{r}^\vee))\end{array}$$

2. *Soundness of the table*

$$\begin{array}{l}\textit{If } \mathbf{call}_{\mathbb{R}^\vee}(f, \{e_1, \ldots, e_n\}, \mathbb{r}^\vee) \quad \Longrightarrow \quad \tau^\sharp(f) = (\mathbb{m}_1^\vee, \mathbb{r}_1^\vee)\\ \textit{Then } \gamma_{\mathbb{M}^\vee}(\mathbb{m}_0^\vee) \subseteq \gamma_{\mathbb{M}^\vee}(\mathbb{m}_1^\vee) \wedge \gamma_{\mathbb{R}^\vee}(\mathbb{r}_0^\vee) \subseteq \gamma_{\mathbb{R}^\vee}(\mathbb{r}_1^\vee)\end{array}$$

We observe that $\mathbf{call}_{\mathbb{R}^\vee}$ is an over-approximation of the analysis of the body c of the function f. The concrete environment e' contains only the arguments of the called function and the concrete heap h' contains the allocated cells for these arguments. The concrete h_2 is the resulting heap of the evaluation of the body of the function. The resulting disjunction of abstract memory relations still contains the arguments of the function. Moreover, $\mathbf{call}_{\mathbb{R}^\vee}$ can update the function summary table, with a more general function summary for the called function.

8.4 Abstract Function Returns and Abstract Relational Semantics

In this section, we define the abstract evaluation of function returns and we finally formalize the semantics of our compositional inter-procedural analysis.

8.4.1 Abstract Function Returns

The analysis of abstract function returns should deallocate the arguments of the called function, and restore the abstract environment of the calling abstract relation. This is implemented by the function $\mathbf{return}_{\mathbb{M}^\sharp_{\mathcal{R}}} \in \mathbb{M}^\sharp_{\mathcal{R}} \times \mathbb{E}^\sharp \to \mathbb{M}^\sharp_{\mathcal{R}}$. It deallocates with the

function $\mathbf{free}_{\mathbb{M}^\sharp_\mathcal{R}}$ (Section 5.5) all variables contained in its abstract environment, and substitutes its abstract environment by the other given abstract environment.

Definition 8.15 (Abstract function returns). *Let* $(\mathrm{e}_0^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp)$ *be an abstract memory relation and* $\mathrm{e}^\sharp$ *an abstract environment. We let* $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\} = \mathbf{dom}(\mathrm{e}_0^\sharp)$ *be the set of variables in* $\mathrm{e}_0^\sharp$*. The abstract memory relations* $(\mathrm{e}_1^\sharp, \mathrm{r}_1^\sharp, \mathrm{n}_1^\sharp), \ldots, (\mathrm{e}_n^\sharp, \mathrm{r}_n^\sharp, \mathrm{n}_n^\sharp)$ *are defined as follows:*

$$\forall i, 1 \leqslant i \leqslant n, \quad (\mathrm{e}_i^\sharp, \mathrm{r}_i^\sharp, \mathrm{n}_i^\sharp) = \mathbf{free}_{\mathbb{M}^\sharp_\mathcal{R}}(\&\mathtt{x}_i, \mathrm{m}^\sharp_{\mathcal{R}^{i-1}})$$

Then, we have:

$$\mathbf{return}_{\mathbb{M}^\sharp_\mathcal{R}}((\mathrm{e}_0^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp), \mathrm{e}^\sharp) = (\mathrm{e}^\sharp, \mathrm{r}_n^\sharp, \mathrm{n}_n^\sharp)$$

Theorem 8.9 (Soundness of $\mathbf{return}_{\mathbb{M}^\sharp_\mathcal{R}}$). *Let* $\mathrm{m}^\sharp_\mathcal{R} \in \mathbb{M}^\sharp_\mathcal{R}$ *and* $\mathrm{e}^\sharp \in \mathbb{E}^\sharp$*. Then:*

$$\forall((\mathrm{e}_0, \mathrm{h}_0), (\mathrm{e}_0, \mathrm{h}_1)) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathrm{m}^\sharp_\mathcal{R})$$
$$\Longrightarrow$$
$$\forall \nu \in \mathbb{V}^\sharp \to \mathbb{V}, ((\nu \circ \mathrm{e}^\sharp, \mathrm{h}_0), (\nu \circ \mathrm{e}^\sharp, \mathrm{h}_1 \ominus \mathbf{im}(\mathrm{e}_0))) \in \gamma_{\mathbb{M}^\sharp_\mathcal{R}}(\mathbf{return}_{\mathbb{M}^\sharp_\mathcal{R}}((\mathrm{e}_0^\sharp, \mathrm{r}_0^\sharp, \mathrm{n}_0^\sharp), \mathrm{e}^\sharp))$$

8.4.2 Abstract Relational Semantics

We now define the abstract relational semantics $[\![f(e_1, \ldots, e_n)]\!]^\sharp_\mathcal{R}$ that over-approximates both a call to a function and its return.

$$[\![f(e_1, \ldots, e_n)]\!]^\sharp_\mathcal{R}(\mathbb{r}_0^\vee) = \mathbf{return}_{\mathbb{R}^\vee}(\mathbf{call}_{\mathbb{R}^\vee}(f, \{e_1, \ldots, e_n\}, \mathbb{r}_0^\vee),\ \mathbf{get\_env}_{\mathbb{R}^\vee}(\mathbb{r}_0^\vee))$$

Theorem 8.10 (Soundness of the analysis). *Let* $f \in \mathcal{F}un$*,* $e_1, \ldots, e_n \in \mathcal{E}xpr$*,* $\mathbb{r}^\vee \in \mathbb{R}^\vee$*,* $\mathbb{m}_0^\vee \in \mathbb{M}^\vee$*,* $\mathbb{r}_0^\vee \in \mathbb{R}^\vee$*, and* $\tau^\sharp \in \mathcal{T}^\sharp$ *such that* $\tau^\sharp(f) = (\mathbb{m}_0^\vee, \mathbb{r}_0^\vee)$*.*
Then:

1. *Soundness of the result*

$$\begin{array}{l}\forall(\mathrm{m}_0, \mathrm{m}_1) \in \gamma_{\mathbb{R}^\vee}(\mathbb{r}^\vee), \quad \forall \mathrm{m}_2 \in \mathbb{M}, \\ \quad (\mathrm{m}_1, \mathrm{m}_2) \in [\![f(e_1, \ldots, e_n);]\!]_\mathcal{R} \\ \quad \Longrightarrow (\mathrm{m}_0, \mathrm{m}_2) \in \gamma_{\mathbb{R}^\vee}([\![f(e_1, \ldots, e_n);]\!]^\sharp_\mathcal{R}(\mathbb{r}^\vee))\end{array}$$

2. *Soundness of the table*

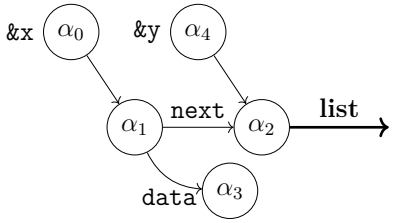
$$\begin{array}{l}\textit{If } [\![f(e_1, \ldots, e_n);]\!]^\sharp_\mathcal{R}(\mathbb{r}^\vee) \quad \Longrightarrow \quad \tau^\sharp(f) = (\mathbb{m}_1^\vee, \mathbb{r}_1^\vee) \\ \textit{Then } \gamma_{\mathbb{M}^\vee}(\mathbb{m}_0^\vee) \subseteq \gamma_{\mathbb{M}^\vee}(\mathbb{m}_1^\vee) \wedge \gamma_{\mathbb{R}^\vee}(\mathbb{r}_0^\vee) \subseteq \gamma_{\mathbb{R}^\vee}(\mathbb{r}_1^\vee)\end{array}$$

8.5 Discussion for Cutpoints

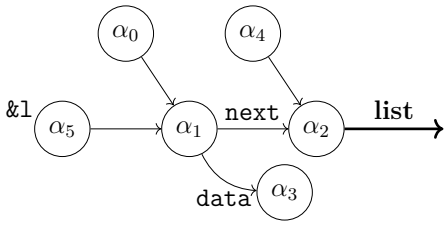
In this section, we discuss how our inter-procedural analysis deals with *cutpoints*. Informally, cutpoints are points in the heap accessible from both variables of the caller and the parameters of the called. To restore after the function call the path from the variables of the caller to the cutpoints, it is important to track cutpoints. This makes the analysis hard, as the function summaries give information only related to the parameters of the function called. Contrary to other inter-procedural analyses, our analysis does not need anything special for cutpoints, like in [RBR+05, RSY05, GBC06, RPHR+07, KRR+13], but may need precise enough relations. We start with a short introduction to cutpoints. We then discuss how our analysis deals with cutpoints according to the different kinds of our abstract relations (identity and transform-into relations).

8.5.1 Introduction to Cutpoints.

Let us consider the following abstract memory state.



We now admit that we have a function `f`, that has only a linked list `l` as argument. If this function is called with the local variable `x` (i.e. `f(x)`) from the above abstract memory state, the binding step (Section 8.3.1) will produce the following abstract memory state.



We observe that the abstract heap $\alpha_0 \mapsto \alpha_1 *_{\mathtt{s}} \alpha_4 \mapsto \alpha_2$ is the unreachable abstract heap obtained by the cut step (Section 8.3.2), and the rest is the reachable abstract heap. The common symbolic values shared by the unreachable and the reachable abstract heaps are named *cutpoints*. In this case, the cutpoints are thus α_1 and α_2. It is important to track cutpoints, as they represent locations reachable from some variables of the caller state. The challenge is that modular function summaries do not explicit them, and so on, it is not obvious what effects can have the called function on them, and as the relations with these are needed to produce precise post-states.

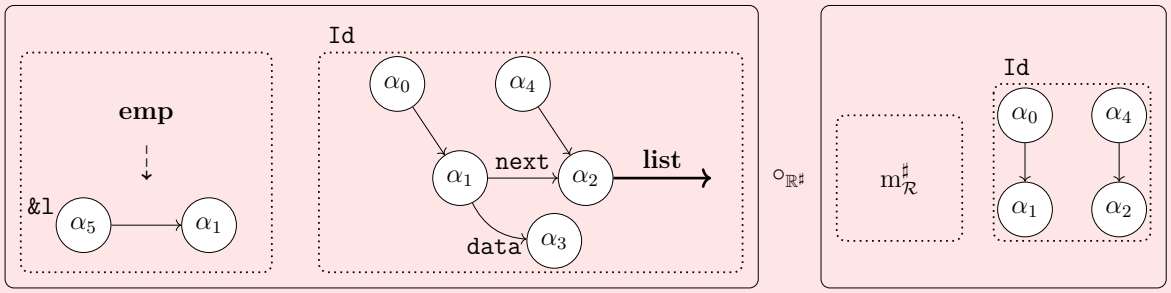
In our inter-procedural analysis, function summaries are represented by abstract memory relations. When a function summary is composed, some cutpoints could be lost: it depends on the precision of its abstract relation. Consequently, in the two following sections, we discuss how cutpoints are tracked when the function summary is an identity relation and a transform-into relation.

8.5.2 Tracking Cutpoints with Identity Relations

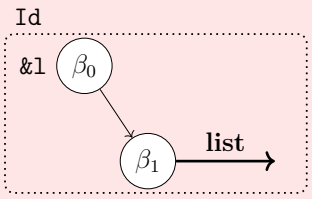
When the function summary is an identity relation, all cutpoints are tracked, without any loss of precision. Indeed, this relation is the most precise: it ensures that the function left the heap physically unmodified. Thus, the abstract composition can restore identically the calling heap, included all cutpoints.

We illustrate how cutpoints are tracked with an identity relation through an example.

Example 8.7 (Case of an identity relation). In this example, we consider the following abstract composition, that occurs during some function call:



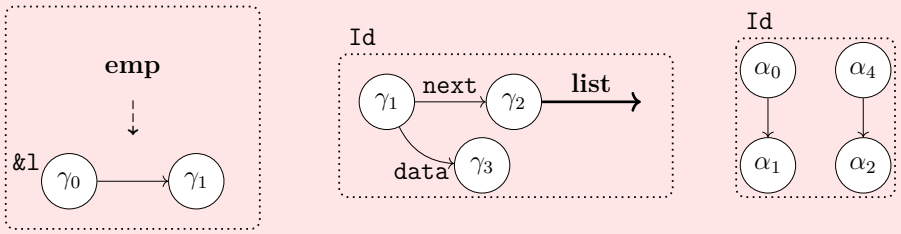
Here, $m^\sharp_{\mathcal{R}}$ is the abstract memory relation of the called function. We assume that $m^\sharp_{\mathcal{R}}$ is of the form:



The unreachable heap $\alpha_0 \mapsto \alpha_1 *_{\mathtt{s}} \alpha_4 \mapsto \alpha_2$ has been added to $m^\sharp_{\mathcal{R}}$ under the identity relation, by the compositional frame rule (Section 8.3.4). We observe two cutpoints: α_1 and α_2. Let us detail how these cupoints are tracked.

In a first time the abstract composition computes the following abstract memory

relation:



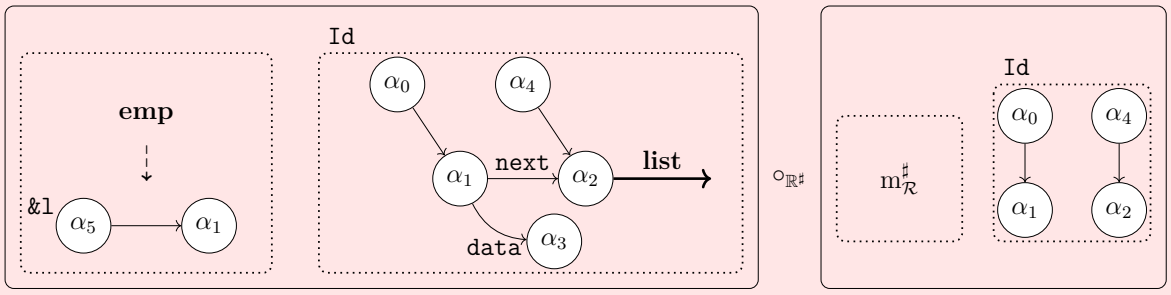
As explained in Section 8.3.5, the abstract composition starts with the initial pair of renaming values Φ where $\Phi(\alpha_0, \alpha_0) = \alpha_0$ and $\Phi(\alpha_4, \alpha_4) = \alpha_4$. The abstract composition has extended Φ with $\Phi(\alpha_5, \beta_0) = \gamma_0$, $\Phi(\alpha_1, \beta_1) = \gamma_1$, $\Phi(\alpha_1, \alpha_1) = \alpha_1$, $\Phi(\alpha_2, \beta_2) = \gamma_2$ and $\Phi(\alpha_2, \alpha_2) = \alpha_2$. This is at this step that Theorem 7.3 (page 119) is important. Indeed, it allows to establish that in the resulting abstract memory relation, we have $\gamma_1 = \alpha_1$ and $\gamma_2 = \alpha_2$, and so on to rename the resulting abstract memory relation accordingly, like in Example 7.1 (page 120).

Actually, adding the identity relation of the unreachable abstract heap to the summary of the function called is not really a limitation, as it allows to track simply all cutpoints.

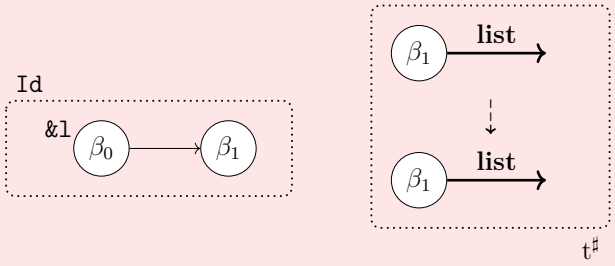
8.5.3 Tracking Cutpoints with Transform-into Relations

When the function summary is a transform-into relation, cutpoints tracking mainly depends on the precision of its attached abstract heap transformation predicate. The more its predicate describes precisely the behaviour of the function, the more cutpoints can be tracked. The next example illustrates this point.

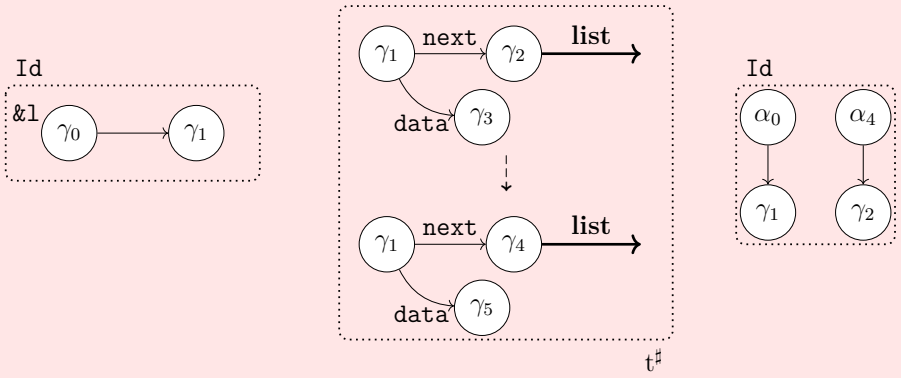
Example 8.8 (Case of a transform-into relation). In this example, we consider the same abstract composition as in Example 8.7:



However, in this case, we assume that $m^\sharp_{\mathcal{R}}$ is of the form:



The track of the cutpoints α_1 and α_2 mainly depends on what can express the abstract heap transformation predicate $t^\sharp$. If it does not express any specific transformation, the abstract composition should produce:



Thanks to Theorem 7.3 (page 119), the abstract composition has inferred that α_0 points to γ_1 and α_4 points to γ_2. However, while γ_1 is present both in the input and output abstract heaps, γ_2 is only present in the input abstract heap. This means that the abstract composition has lost where α_4 points to after the function call. This problem can be partially fixed thanks to more precise abstract heap transformation predicates. For instance, if $t^\sharp = (\{=^\sharp, \texttt{data}\})$, this means that no allocation or deallocation occurred, and that only some `data` fields may have been modified. This allows to infer that $\gamma_2 = \gamma_4$.

8.6 Experimental Evaluations

In this section, we report on the evaluation of our compositional inter-procedural shape analysis. The goal is to evaluate the efficiency and the precision of our analysis compared to a classical state shape analysis. Indeed, our analysis should be more efficient than a state analysis but does not suffer loss of precision. We analyzed two different programs that both manipulate linked lists: a small program, of approximately 300 lines of C codes,

that implements a Battle (the card game), and a larger program, of approximately 2000 lines of C codes, that consists of a part of Emacs (the text editor).

8.6.1 Battle Game

This program was originally written in Java, and used the `LinkedList` class of the standard library to represent desks. We completely translated this program in C, using the usual structure definition for linked lists. In addition, we wrote functions to deallocate the memory when necessary, as the original Java program is garbage-collected. This benchmark is important because it allows to understand when the composition boosts the speed-up of the analysis. Also, it is relevant with our relational shape analysis, as it mainly manipulates linked lists, traversing them, or adding or deleting elements. These are the kind of relational properties that our analysis can infer.

The code excerpt below defines the data structures used in this program.

```
1  typedef struct list {
2    int data;
3    struct list *next;
4  } list;
5
6  typedef struct deck {
7    list *pack_of_cards;
8  } deck;
9
10 typedef struct battle {
11   int nbVals;
12   deck *trick;
13   deck *player1;
14   deck *player2;
15 } battle;
```

That is, a deck is simply a linked list (where `data` fields represent the value of a card), and a battle is composed of two players and a trick, all represented by a deck.

We give in Figure 8.3 a static call graph of this program. It does not indicate how many times a function is called by another one, but which function can call another. It allows to interpret the result of the experimentation evaluation given in Table 8.1. The first two columns give the execution times of both analyses in seconds. These times in seconds correspond to the sum of 1000 runs of each function. The last column indicates if the compositional analysis produced a loss of precision compared to the state analysis, except for leaf functions, that obviously do not perform function calls. We mean by loss of precision if the abstract heaps of the compositional analysis are not included in the abstract heaps of the state analysis.

We observe that the deepest functions in the call graph (like `initDeck` or `pick`),

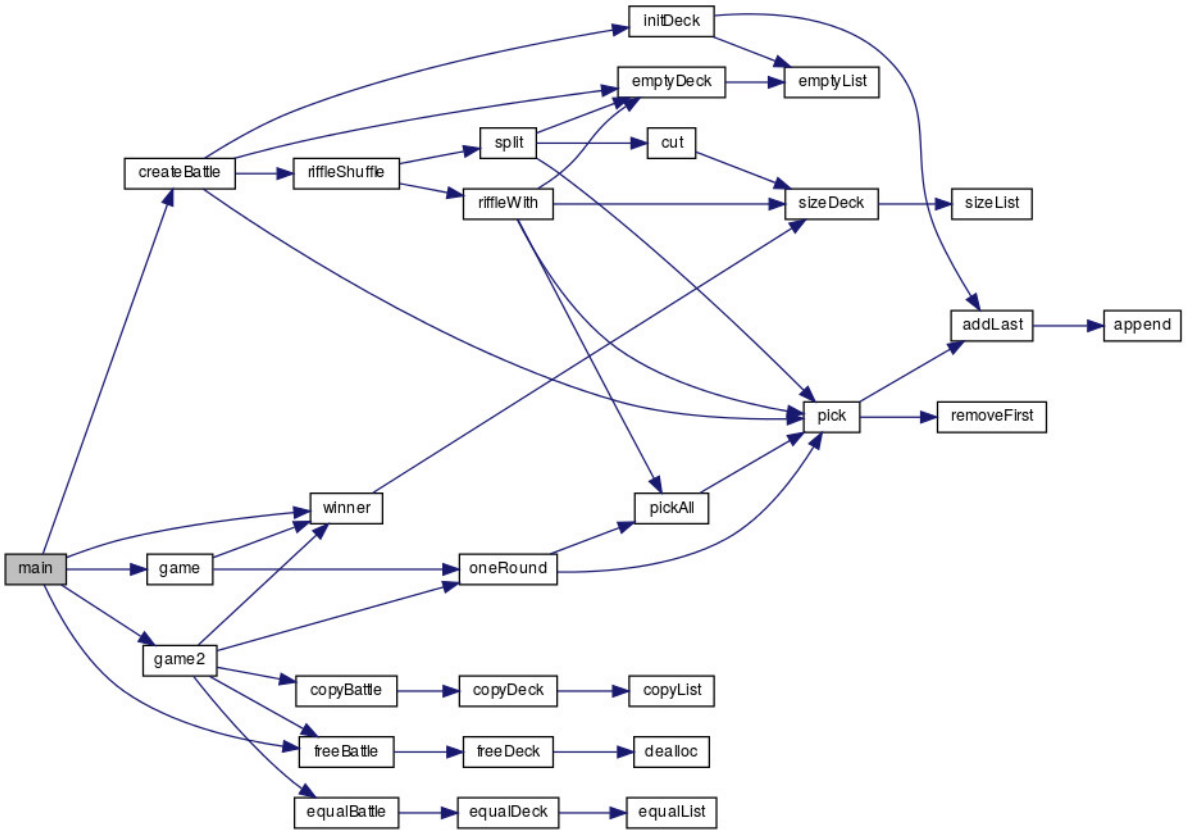


Figure 8.3: Static call graph of the Battle

Structure	Function	State	Composition	Precision	Depth
List	`empty_list`	0.236	0.252	-	5
List	`removeFirst`	0.248	0.268	-	6
List	`append`	0.608	1.124	-	7
List	`addLast`	0.872	1.584	no loss	6
List	`dealloc`	0.312	0.540	-	4
List	`sizeList`	0.456	0.648	-	5
List	`copyList`	0.808	1.520	-	4
List	`equalList`	1.096	2.364	-	4
Deck	`emptyDeck`	0.240	0.280	no loss	4
Deck	`initDeck`	3.888	4.124	no loss	2
Deck	`sizeDeck`	0.572	0.808	no loss	5
Deck	`pick`	1.364	2.348	no loss	5
Deck	`pickAll`	1.616	3.604	no loss	4
Deck	`isDeck`	1.052	1.812	no loss	0
Deck	`freeDeck`	0.420	0.6280	no loss	3
Deck	`cut`	1.700	1.844	no loss	4
Deck	`split`	4.932	6.972	no loss	3
Deck	`riffleWith`	15.064	14.548	no loss	3
Deck	`riffleShuffle`	53.024	22.992	no loss	2
Deck	`equalDeck`	1.592	2.544	no loss	3
Deck	`copyDeck`	1.072	1.696	no loss	3
Battle	`createBattle`	78.816	30.132	no loss	1
Battle	`freeBattle`	1.188	1.224	no loss	2
Battle	`oneRound`	28.816	22.736	no loss	2
Battle	`winner`	3.088	2.412	no loss	2
Battle	`copyBattle`	4.676	2.852	no loss	2
Battle	`equalBattle`	13.756	8.736	no loss	2
Battle	`game`	39.280	26.876	no loss	1
Battle	`game2`	185.252	51.396	no loss	1
Battle	`main` (with `game`)	188.588	78.962	no loss	0
Battle	`main` (with `game2`)	5319.648	864.282	no loss	0

Table 8.1: Measured time (in second) corresponding to the sum of 1000 iterations of each function. The column 'Precision' indicates if the compositional analysis loses precision compared to the state analysis. We node '-' for leaf functions in the control flow graph, as no composition operation is required for their analysis. The column 'depth' gives the depth of each function in the control flow graph. Performed on a laptop with Intel Core i7 running at 2.3 GHz, with 16 Gb RAM.

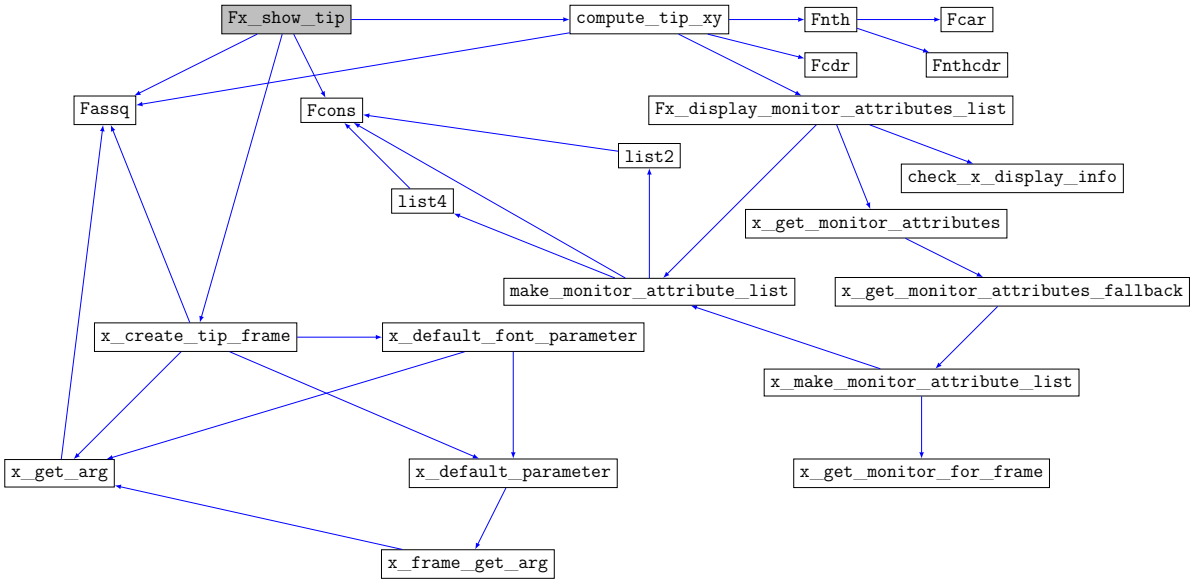


Figure 8.4: Static call graph of the analyzed part of Emacs.

the compositional analysis is reasonably slower (never more than the double, except for `pickAll` and `equalList`). However, for the shallower functions in the call graph the compositional analysis we observe a significant raise of the execution time (more than twice faster for some functions like `game2` or `createBattle`, more than six times for `main` (with `game2`). We also observe that the compositional analysis never produces a loss of precision compared to the state analysis.

8.6.2 Function of Emacs Manipulating Cons Lists

We now report on the evaluation of a larger program, a part of Emacs 25.3. The analyzed part is the function `Fx_show_tip`, from file src/xfns.c This program, written is C, mainly manipulates *Lisp Objects* that denote a set of primitive types of Lisp such as Strings, Integers, Vectors or Cons pairs. The primitive type we are interested in is the Cons pairs, as they are used to implement linked lists.

A cons pair is composed of two elements (x, y), provided by two operations: `car` and `cdr`. The operation `car`(x, y) returns the first element of the pair, so x, and the operation `cdr`(x, y) returns the second element of the pair, y. It should be noted that the elements x and y are both pointers to Lisp Objects. Consequently, we can implement linked lists on top of cons pairs, where `car` allows to access to the first element of the list and `cdr` allows to access the rest of the list. For instance, the following Lisp Object represents a linked list of three integers that ends by the special Lisp Object `Nil` that denotes the

Function	Time (in seconds)		Precision	Depth
	State	Comp.		
`Fcons`	0.329	0.337	-	8
`list2`	0.322	0.324	no loss	7
`list4`	0.334	0.324	no loss	7
`Fassq`	2.163	6.463	-	6
`Fcar`	0.318	0.332	-	3
`Fcdr`	0.330	0.332	-	2
`Fnthcdr`	0.332	0.345	-	3
`Fnth`	0.340	0.341	no loss	2
`make_monitor_attribute_list`	0.330	0.741	no loss	6
`x_get_arg`	11.817	8.760	no loss	5
`x_default_parameter`	23.001	8.908	no loss	3
`check_x_display_info`	0.327	0.330	-	3
`Fx_display_monitor_attributes_list`	0.412	0.856	no loss	2
`Fx_show_tip`	877.12	14.201	no loss	0
`x_frame_get_arg`	21.747	8.870	no loss	4
`x_get_monitor_for_frame`	0.395	0.369	-	6
`x_make_monitor_attribute_list`	0.475	0.867	no loss	5
`x_get_monitor_attributes_fallback`	0.350	1.014	no loss	4
`x_get_monitor_attributes`	0.348	1.106	no loss	3
`compute_tip_xy`	38.239	16.803	no loss	1
`x_default_font_parameter`	39.062	7.172	no loss	2
`x_create_tip_frame`	321.774	6.962	no loss	1

Table 8.2: Measured times in seconds for each the analysis of each function. The column 'Precision' indicates if the abstract states contained in the abstract relation inferred by the compositional analysis were less precise than the abstract states inferred by the state analysis. We note '-' for functions that do not perform function calls (leaf functions). The column 'Depth' indicates the maximum depth of each function in the control flow graph. Performed on a laptop with Intel Core i7 running at 2.3 GHz, with 16 Gb RAM.

empty list:

$$(13, (10, (92, \texttt{Nil})))$$

To abstract such Lisp Objects, we defined the following inductive and segment predicates:

$$\begin{array}{rcl}
\textbf{lisp_object}(\alpha) & := & (\textbf{emp},\ \alpha = \texttt{Nil}) \\
& \vee & (\alpha \cdot \texttt{car} \mapsto \beta *_{\texttt{s}} \textbf{lisp_object}(\beta) *_{\texttt{s}} \\
& & \ \alpha \cdot \texttt{cdr} \mapsto \gamma *_{\texttt{s}} \textbf{lisp_object}(\gamma),\ \alpha \neq \texttt{Nil}) \\
\\
\textbf{lisp_objectseg}(\alpha, \gamma) & := & (\textbf{emp},\ \alpha = \gamma) \\
& \vee & (\alpha \cdot \texttt{car} \mapsto \delta *_{\texttt{s}} \textbf{lisp_object}(\delta) *_{\texttt{s}} \\
& & \ \alpha \cdot \texttt{cdr} \mapsto \beta *_{\texttt{s}} \textbf{lisp_objectseg}(\beta, \gamma),\ \alpha \neq \texttt{Nil})
\end{array}$$

Observe that these predicates look like tree predicates. Indeed, both fields of a List Object point to another Lisp Object. The difference with trees is that the field `car` represents the data of the list, that can be any Lisp Object. For instance, the analyzed program mainly manipulates *association lists*, where the elements of the list are pairs of Lisp Object, for instance:

$$((13, \texttt{'d}'),\ ((10, \texttt{'m}'),\ ((92, \texttt{'y}'), \texttt{Nil})))$$

This kind of lists can be abstracted by the definitions above, thanks to the field `car` that is also a Lisp Object. Remark that we added other cases in these definitions, for instance to handle the case where a Lisp Object is a *frame*.

We remark that these definitions do not allow sharing between Lisp Objects whereas their implementation allows this. In the part of Emacs that we analyse, only one command introduces shared data structures. It is the `parms` = `Fcopy_alist`(`parms`) assignment, at the beginning of function `x_create_tip_frame`. Function `Fcopy_alist` copies partially an association list. The output list does not share the structure of the input list but shares its elements. As our implementation does not support abstraction for shared data structures and the original list `parms` is not used anymore in the rest of the program, we decided to delete the call to `Fcopy_alist`.

Our analysis also ignores functions that manipulate types not supported by our analysis, such as strings or vectors. It simply raises a warning when such a function is called. This makes our analysis sound only with respect to some behaviors (when the program manipulates data types supported by our inductive definitions), and does not produce meaningful behaviors to other kind of Lips Objects (like the vectors). However, the goal of this evaluation is only to evaluate how the precision and scalability of our compositional analysis compare with a more classical state shape analysis.

Figure 8.4 presents the static call graph of analyzed functions, and Table 8.2 the measured times (in seconds) for the analysis of each of them, both for the state and the relational analyses. For the leaf functions, such as `Fcons` or `Fcar`, the measured

times are of course negligible (approximately 0.3 seconds). An exception occurred for function `Fassq`. This function performs many conditional tests in the body of a loop, that generates many disjunctions and a combinatorial explosion when the analysis compares and widens them. We observe other functions for which the measured time of the state analysis is negligible but quite slower for the relational analysis, for instance `make_monitor_attribute_list` or `x_get_monitor_attributes_fallback`. This is because these functions perform few function calls, and consequently the relational analysis does not gain to perform composition. However, for the shallower functions in the call graph, the gain of the compositional analysis is in practice exponential, for instance for the functions `x_create_tip_frame` and `x_default_font_parameter`. Especially, for the root function `Fx_show_tip`, we observe that the compositional analysis is more than 60 times faster than the state analysis, and all of this without losing any precision.

In Table 8.3, we explicit how many times a function has been called during the analysis of the root function `Fx_show_tip`. For the state analysis, every function is analysed each time it is called, we thus indicate the number of calls in the column 'State'. For the compositional analysis, when a function is called, it is either composed directly, or re-analyzed if the composition is not possible. The column 'analyzed' indicates how many times a function has been analyzed, the column 'composed' indicates how many times the function has been *directly* composed (this means composed without reanalyzing the function). The column 'total', the sum of these two previous columns, indicates the total number of calls for a function. Thanks to this Table, we can see that the deepest functions are called considerably less often in the compositional analysis compared to the state analysis. Also, all functions except `Fcons` are analyzed only once, and composed directly the other times. This improves strongly the execution time of the compositional analysis.

8.7 Related Works

To our knowledge, our inter-procedural analysis is the first to use abstraction relations that rely on separation logic. However, some analyses [RSY05, GBC06, MHKS08, BDES11] proceed similarly to ours by extracting the reachable abstract heap from the actual arguments of a function. This technique is probably the most intuitive when considering function calls for an inter-procedural analysis. We believe that the use of our abstract relations makes this kind of analysis much more precise. Indeed, none of these analyses (except [BDES11]) defines a composition operator between abstract relations describing the shape of the heap.

Another important approach to compositional inter-procedural shape analysis is the analyses using the bi-abduction inference [CDOY09, GCRN09, LGQC14, CDOY07]. They aim at finding and using smaller specifications, that describe only what can be read (and thus written) by the function. While this method is more local than ours,

Function	State	Compositional		
		analyzed	composed	total
`Fcons`	296	3	44	47
`list2`	12	1	1	2
`list4`	24	1	3	4
`Fassq`	64	1	15	16
`Fcar`	24	1	0	1
`Fcdr`	12	1	3	4
`Fnthcdr`	24	1	0	1
`Fnth`	24	1	7	8
`make_monitor_attribute_list`	6	1	1	2
`x_get_arg`	19	1	4	5
`x_default_parameter`	15	1	14	15
`check_x_display_info`	3	1	0	1
`Fx_display_monitor_attributes_list`	3	1	0	1
`Fx_show_tip`	1	1	0	1
`x_frame_get_arg`	15	1	0	1
`x_get_monitor_for_frame`	6	1	1	2
`x_make_monitor_attribute_list`	3	1	0	1
`x_get_monitor_attributes_fallback`	3	1	0	1
`x_get_monitor_attributes`	3	1	0	1
`compute_tip_xy`	3	1	2	3
`x_default_font_parameter`	1	1	0	1
`x_create_tip_frame`	1	1	0	1

Table 8.3: Number of calls for each function during the analysis of `Fx_show_tip`. For the compositional analysis, the column 'analyzed' indicates how many times a function has been fully analyzed, the column 'composed' indicates how many times the function has been *directly* composed. The column 'total', indicates the total number of calls for a function.

their function summaries are tabulations of pre and post conditions of abstract memory states. We believe that our abstract relations are more compact and more expressive. Also, their method does not require to specify the pre-condition of any functions, whereas ours requires to specify only the pre-condition of the root function. This makes it more modular than ours but may generate specifications that are never used in the analysis. Moreover, it would be very interesting to combine the bi-abduction inference with our abstract relations, instead of using pre-post conditions. We could obtain more precise and compact results than these analyses and more efficient than ours.

Several other works rely one computing very precise and compact summaries for functions and composing them. For instance, [YYC08] computes precise and concise summaries that can encode IFDS [RHS95] and IDE [SRH96] problems. The composition of a summary does not generate any loss of precision. However, they are specialized in typestates properties and do not support inductive data structures. Also, the function summaries of [DDAS11] describe all the possible configurations of the heaps, according to the input alias relations in a very precise and compact way. However, this work does not address to compute summaries about shapes of data structures.

Chapter 9

Extension: Analysis of Recursive Functions

Until now, the compositional inter-procedural analysis we defined does not allow to analyse recursive functions. Indeed, the algorithm will loop indefinitely for the analysis of such functions. In this chapter, we thus extend the analysis to handle recursive functions.

9.1 Overview

Before formalizing the extension, we provide an overview on a running example. Consider the function `dealloc_tree` of Figure 9.1, that performs recursively the deallocation of a binary tree. The first version of the analysis of this function will loop indefinitely: at line 5 for the first recursive call, the analysis algorithm tests if the calling state is included in the pre-condition of the function summary. As this pre-condition is initialized to $\bot_{\mathbb{H}^\sharp}$, the test returns **false** and the algorithm re-analyzes the body of the recursive function. This process is repeated indefinitely.

The extended algorithm proceeds thus as follows:

1. At each recursive call, it widens the calling state with the pre-condition, and directly updates the function summary with the widened pre-condition, then reanalyzes the function. It repeats this operation until the inclusion of the calling state in the pre-condition holds.
2. When the inclusion holds, the algorithm, like in the first version, proceeds to the abstract composition. However until now, the analysis only updated the pre-condition of the function summary, but not its relation that is still to $\bot_{\mathbb{R}^\sharp}$. Consequently, the abstract composition returns $\bot_{\mathbb{R}^\sharp}$. The element $\bot_{\mathbb{R}^\sharp}$ is propagated until the end of the conditional branch, and joined with the inferred abstract relation of the other branch. This allows to finish the analysis of the recursive function for the first time. At this step, the algorithm obtains a relation corresponding to the *terminal case* of

```
1 void dealloc_tree(tree *t) {
2   if(t != NULL){
3      tree *t_l = t->l;
4      tree *t_r = t->r;
5      dealloc_tree(t_l);
6      dealloc_tree(t_r);
7      free(t);
8   }
9 }
```

Figure 9.1: Recursive deallocation of a binary tree.

the function.

3. The algorithm should compute a relation describing all cases of the function, not only the terminal case. In turn, the algorithm widens the obtained abstract relation with the one contained in the function summary, stores it in the function summary, and reanalyzes the recursive functions using the widened abstract relations during the abstract composition. The algorithm proceeds to these operations until the inferred abstract relation is included in the relation of the function summary.

Figure 9.2 shows the analysis of function `dealloc_tree` after reaching the fixpoint of the pre-condition. The algorithm can perform the abstract composition at the first recursive call. However, the relation of the current function summary is $\bot_{\mathbb{R}^\sharp}$. Consequently, $\bot_{\mathbb{R}^\sharp}$ is propagated until the end of the conditional branch, and joined with the inferred abstract relation of the other branch. The algorithm thus inferred the abstract relation for the terminal case. As this later is not included in $\bot_{\mathbb{R}^\sharp}$, the algorithm widens them (the result of the widening is obviously the abstract relation of the terminal case), and updates the function summary. The algorithm must reanalyse the function.

Figure 9.3 represents the next iteration of the analysis. For the two recursive calls, the algorithm uses the relation describing the terminal case for the abstract composition. As this relation describes the identity relation of the null pointer, the abstract composition of the two recursive calls infers that `t_l` and `t_r` are both the null pointer. After the recursive calls, the algorithm proceeds to the deallocation of `t`, and finishes the iteration by joining the two branches. As the inferred abstract relation that describes the deallocation of a tree is not included in the current relation of the function summary, the algorithm widen them, and updates the function summary. A third iteration is thus required.

Figure 9.4 shows the third iteration, at which fixpoint is reached. For the recursive calls, the algorithm uses the relation of the current function summary that describes the deallocation of a tree. That is, after the abstract composition of the two recursive calls, the algorithm has inferred that both the left and right children (respectively `t_l`

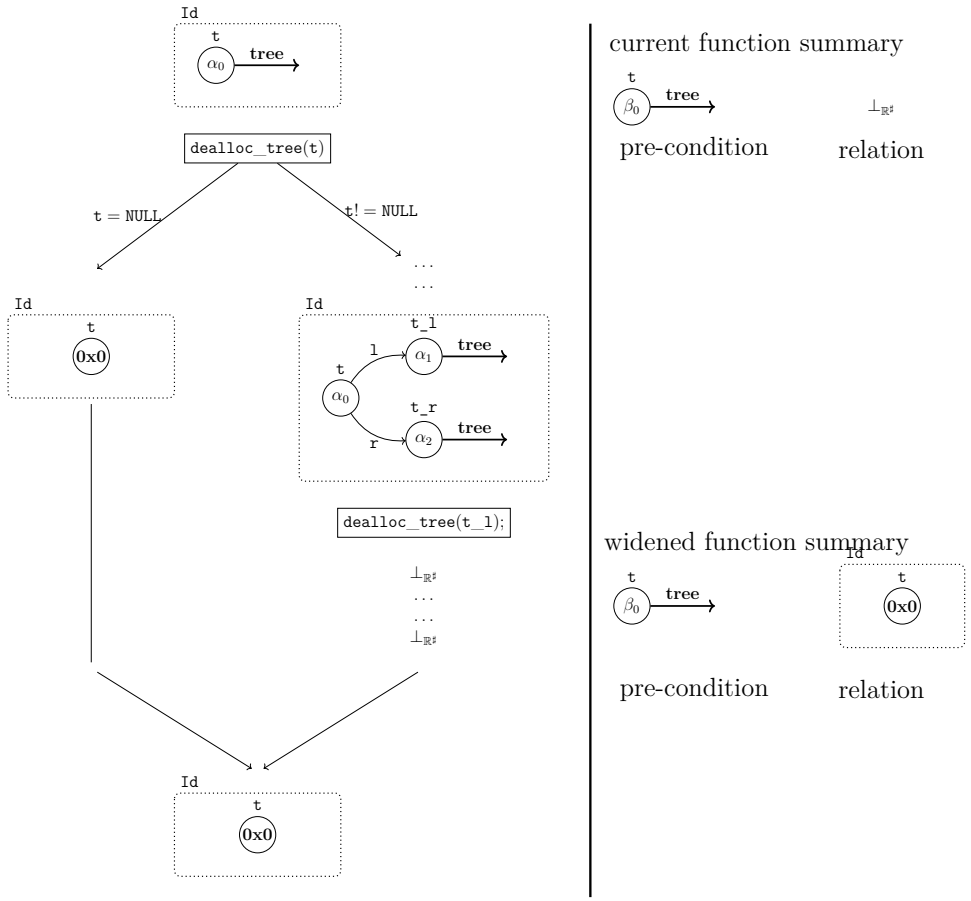


Figure 9.2: First iteration. This iteration generates the abstract relation corresponding to the terminal case of the function. During the abstract composition, the algorithm uses the relation of the current function summary $\bot_{\mathbb{R}^\sharp}$. The widened relation is obtained by widening the current relation on the right top with the inferred relation on the bottom.

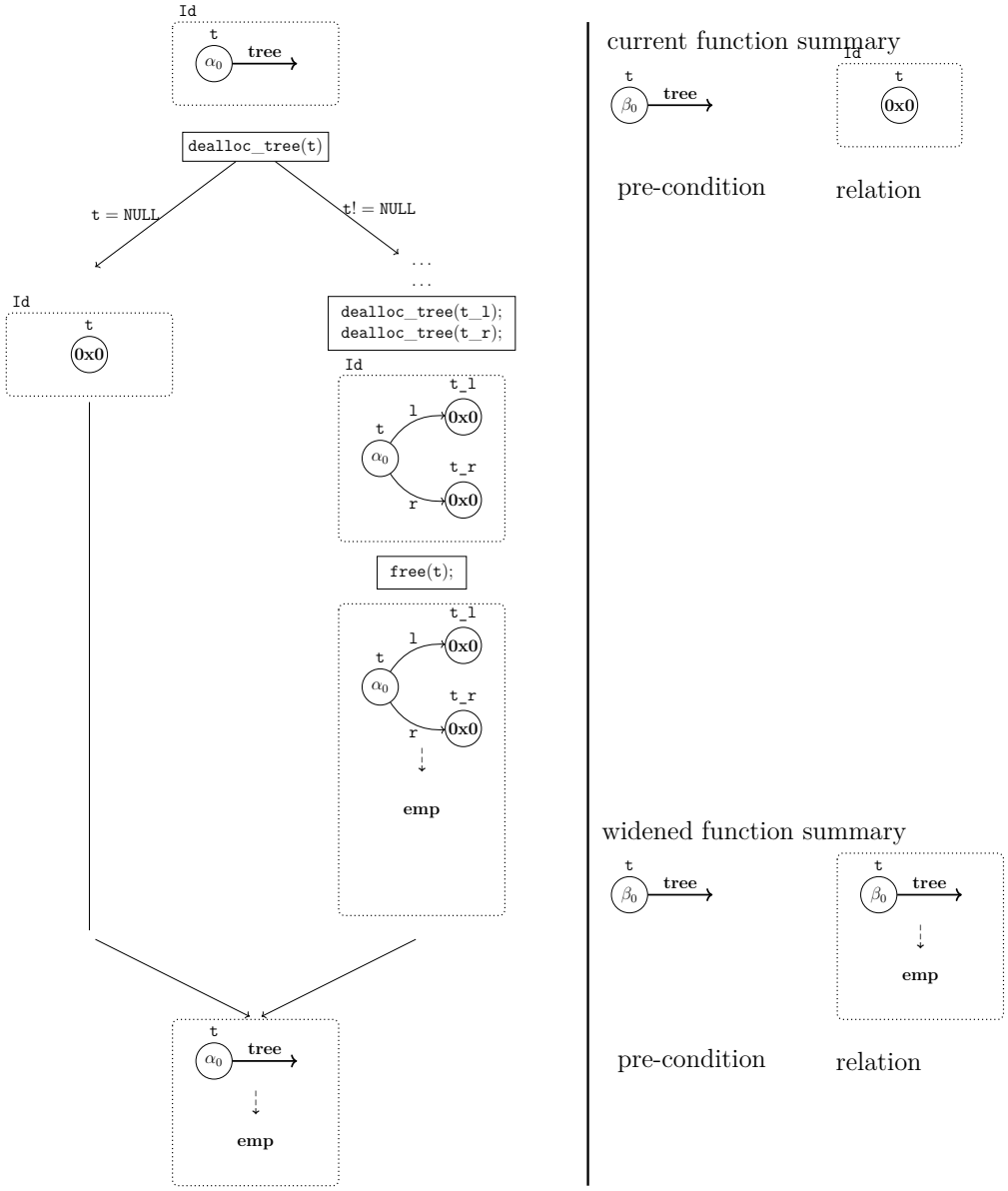


Figure 9.3: Second iteration. During this iteration, the algorithm uses the relation of the current function summary, that corresponds to the relation for the terminal case of the function generated in Figure 9.2.

and `t_r`) have been fully deallocated. This is also the case for the pointer `t`, after the command `free(t)`. Thus, the abstract relation inferred for this branch describes the full deallocation of the given tree. The inferred abstract relation, obtained by joining the two branches also describes the full deallocation of the tree. This abstract relation is included in the relation of the function summary, the fixpoint is thus reached.

9.2 Algorithm

This extension of the compositional analysis is very similar to the original version, except that it first computes a fixpoint for the pre-condition of the function summary, and then computes a fixpoint for the relation.

To compute a fixpoint for the relation of a function summary, the algorithm relies on a function **compute_fp**(f) that analyses the body of function f, widens the resulting abstract relation with the relation contained in the function summary for f, and continues until reaching a fixpoint.

Definition 9.1. *Let $f \in \mathcal{Fun}$ such that:*

$$f(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{c; \mathbf{ret};\} \in \mathcal{Prog}$$

The function **compute_fp** *is defined as follows:*

$$\begin{array}{l}
\mathbf{compute_fp}(f) = \\
\quad \mathit{Let}\ (\mathbb{m}_f^\vee, \mathbb{r}_f^\vee) = \tau^\sharp(f)\ \mathit{in} \\
\quad \mathit{Let}\ \mathbb{r}_1^\vee = [\![c]\!]^\sharp_{\mathcal{R}}(\mathbf{id}_{\mathbb{M}^\vee}(\mathbb{m}_f^\vee))\ \mathit{in} \\
\quad \mathit{If}\ \mathbf{isle}_{\mathbb{R}^\vee}(\mathbb{r}_1^\vee, \mathbb{r}_f^\vee) = \mathbf{false} \\
\quad \mathit{Then}\ (\\
\qquad \mathit{Let}\ \mathbb{r}_f^\vee = \mathbf{wid}_{\mathbb{R}^\vee}(\mathbb{r}_1^\vee, \mathbb{r}_f^\vee)\ \mathit{in} \\
\qquad \tau^\sharp(f) := (\mathbb{m}_f^\vee, \mathbb{r}_f^\vee); \\
\qquad \mathbf{compute_fp}(f) \\
\quad)
\end{array}$$

Theorem 9.1 (Soundness of function compute_fp). *Let $f \in \mathcal{Fun}$, $\mathbb{m}_0^\vee \in \mathbb{M}^\vee$, $\mathbb{r}_0^\vee \in \mathbb{R}^\vee$, and $\tau^\sharp \in \mathcal{T}^\sharp$ such that $\tau^\sharp(f) = (\mathbb{m}_0^\vee, \mathbb{r}_0^\vee)$.*

$$\begin{array}{l}
\mathit{If}\ \mathbf{compute_fp}(f) \implies \tau^\sharp(f) = (\mathbb{m}_1^\vee, \mathbb{r}_1^\vee) \\
\mathit{Then}\ \gamma_{\mathbb{M}^\vee}(\mathbb{m}_0^\vee) \subseteq \gamma_{\mathbb{M}^\vee}(\mathbb{m}_1^\vee) \wedge \gamma_{\mathbb{R}^\vee}(\mathbb{r}_0^\vee) \subseteq \gamma_{\mathbb{R}^\vee}(\mathbb{r}_1^\vee)
\end{array}$$

The algorithm also defines the function $\mathbf{rec\_call}_{\mathbb{R}^\vee} \in \mathcal{Fun} \times \mathcal{P}_{\text{fin}}(\mathcal{E}\mathit{xpr}) \times \mathbb{R}^\vee \to \mathbb{R}^\vee$ that is applied for calls of recursive functions. It is similar than $\mathbf{call}_{\mathbb{R}^\vee}$, except that it

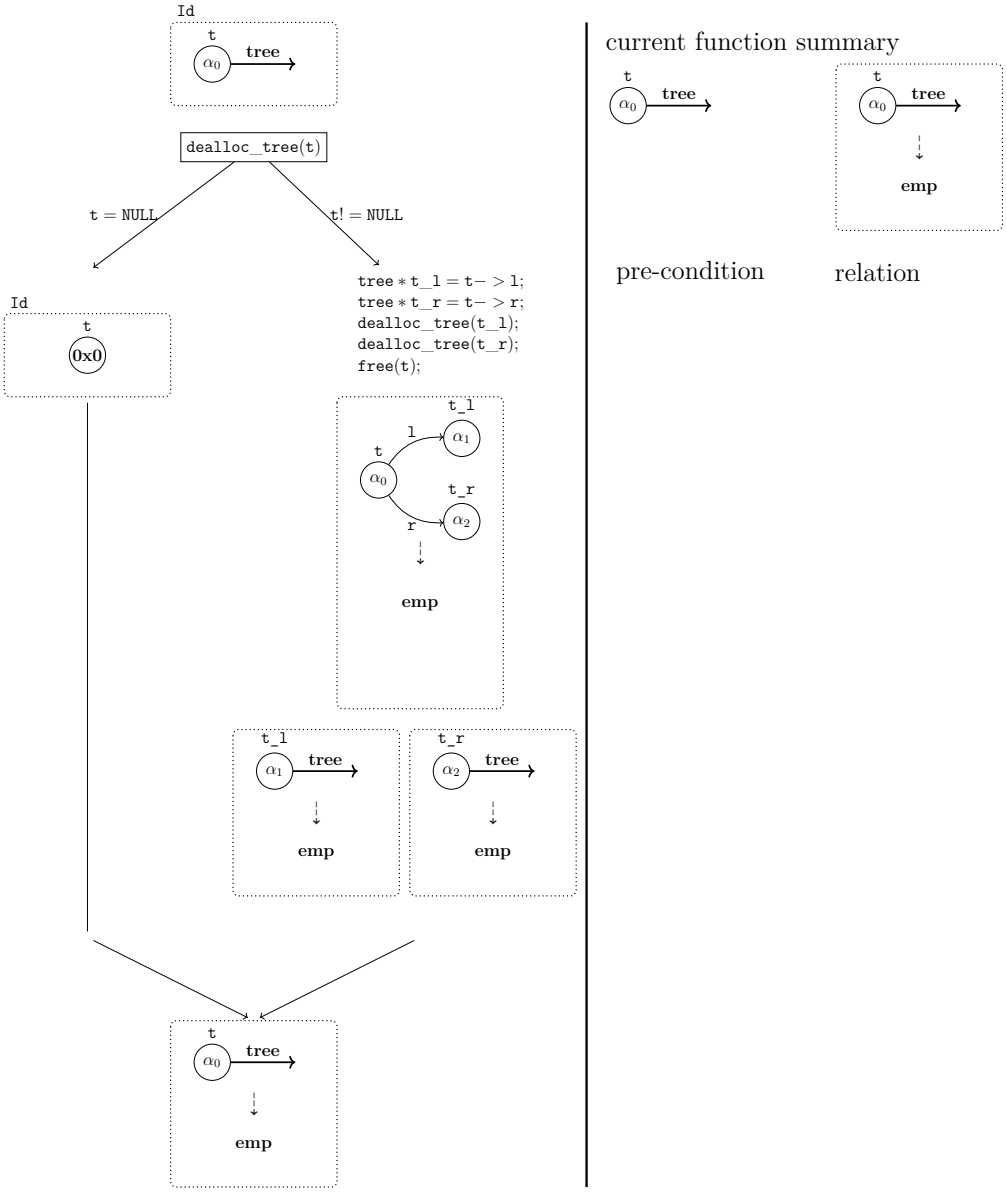


Figure 9.4: Third iteration at which fixpoint is reached. The fixpoint it reached because the current relation is included in the inferred relation.

widens (with the function $\mathbf{wid}_{\mathbb{M}^\vee} \in \mathbb{M}^\vee \times \mathbb{M}^\vee \to \mathbb{M}^\vee$) the calling state with the precondition of the function summary instead of joining them when the inclusion does not hold, and uses the function **compute_fp** to widen the abstract relations.

Definition 9.2. *Let* $e_1, \ldots, e_n \in \mathcal{E}xpr$, $\mathbb{r}_0^\vee \in \mathbb{R}^\vee$ *and* $f \in \mathcal{F}un$ *such that:*

$$f(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{c; \mathbf{ret};\} \in \mathcal{P}rog$$

The function $\mathbf{rec\_call}_{\mathbb{R}^\vee}$ *is defined as follows:*

$$\begin{array}{l}
\mathbf{rec_call}_{\mathbb{R}^\vee}(f, \{e_1, \ldots, e_n\}, \mathbb{r}_0^\vee) = \\
\quad \mathit{Let}\ (\mathbb{m}_f^\vee, \mathbb{r}_f^\vee) = \tau^\sharp(f)\ \mathit{in} \\
\quad \mathit{Let}\ \mathbb{r}_1^\vee = \mathbf{bind}_{\mathbb{R}^\vee}(\{(\mathtt{x}_1, e_1), \ldots, (\mathtt{x}_n, e_n)\}, \mathbb{r}_0^\vee)\ \mathit{in} \\
\quad \mathit{Let}\ (\mathbb{m}^\vee, \mathrm{H}^\sharp) = \mathbf{cut}_{\mathbb{R}^\vee}(\mathbb{r}_1^\vee)\ \mathit{in} \\
\quad \mathit{If}\ \mathbf{isle}_{\mathbb{M}^\vee}(\mathbb{m}^\vee, \mathbb{m}_f^\vee) = \mathbf{false} \\
\quad \mathit{Then}\ (\\
\qquad \mathit{Let}\ \mathbb{m}_f^\vee = \mathbf{wid}_{\mathbb{M}^\vee}(\mathbb{m}^\vee, \mathbb{m}_f^\vee)\ \mathit{in} \\
\qquad \tau^\sharp(f) \leftarrow (\mathbb{m}_f^\vee, \mathbb{r}_f^\vee); \\
\qquad \mathbf{compute_fp}(f) \\
\quad) \\
\quad \mathit{Let}\ (\mathbb{m}_f^\vee, \mathbb{r}_f^\vee) = \tau^\sharp(f)\ \mathit{in} \\
\quad \mathit{Let}\ \mathbb{r}_f^\vee = \mathbf{inst}_{\mathbb{R}^\vee}(\mathbb{r}_f^\vee)\ \mathit{in} \\
\quad \mathit{Let}\ \mathbb{r}_f^\vee = \mathbf{enrich}_{\mathbb{R}^\vee}(\mathbb{r}_f^\vee, \mathrm{H}^\sharp)\ \mathit{in} \\
\quad \mathbf{comp}_{\mathbb{R}^\vee}(\mathbf{init}_{\mathbb{R}^\vee}(\mathbf{get_env}_{\mathbb{R}^\vee}(\mathbb{r}_0^\vee)),\ \mathbb{r}_1^\vee,\ \mathbb{r}_f^\vee)
\end{array}$$

The soundness theorem of $\mathbf{rec\_call}_{\mathbb{R}^\vee}$ is the same as the soundness theorem of $\mathbf{call}_{\mathbb{R}^\vee}$ (Theorem 8.8) (page 152), we thus do not explicit it.

The particularity with our analysis is that it does not require explicit stack abstractions, as in [RS01] or [RC11]. Indeed, our approach is the *functional approach* [SP81] that is valid even in the presence of recursion. Our analysis simply uses intermediate function summaries that it composes to treat recursive function calls, and widens these intermediates summaries until reaching a fixpoint.

9.3 Experimental Evaluation

We report on the implementation of this compositional analysis. The goals of this evaluation is to check whether:

1. the analysis of the recursive version of some functions inferred similar relations as the analysis of the iterative version.
2. the analysis works for functions that is not obvious to implement iteratively, for example for functions manipulating trees.

Structure	Function	Time (in s)	Similar to iterative?
List	`length`	1.256	yes
List	`get_n`	2.179	yes
List	`alloc`	1.139	yes
List	`dealloc`	0.842	yes
List	`concat`	1.833	no
List	`map`	0.904	yes
List	`deep_copy`	1.540	yes
List	`filter`	3.357	yes
Tree	`visit`	1.078	-
Tree	`size`	1.951	-
Tree	`search`	3.818	yes
Tree	`dealloc`	1.391	-
Tree	`insert`	5.083	no
Tree	`deep_copy`	2.603	-

Table 9.1: Measured time (in second) over the sum of 1000 runs of each function. Last column indicates whether the inferred relation is similar to the iterative version. We note '-' for the functions we could not implement iteratively. Performed on a laptop with Intel Core i7 running at 2.3 GHz, with 16 Gb RAM.

Consequently, we selected some functions manipulating linked lists that are natural to implement recursively, and some functions manipulating binary trees, including some that are not obvious to implement iteratively (as they require an explicit stack). These functions are listed in Table 9.1 In this table, each timing measurement is the sum over 1000 runs of a function. The last column indicates if the analysis of the recursive version inferred a similar relation as the iterative one.

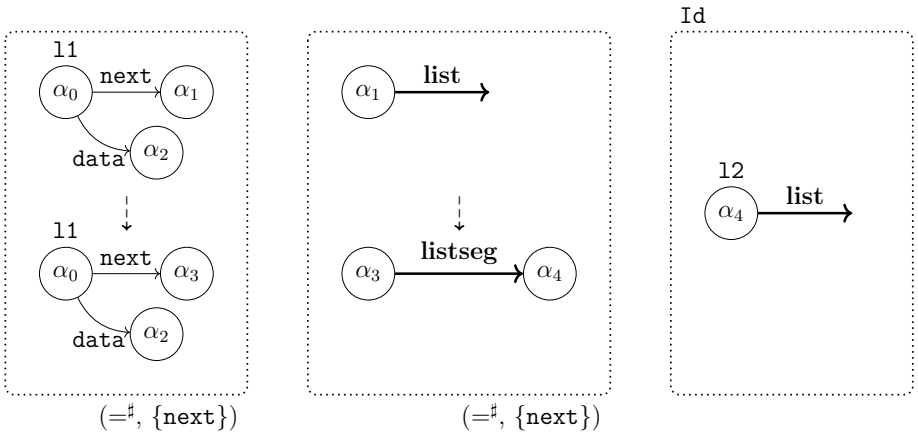
We observe that all of the functions has been analyzed in a reasonably time (remind that the given times are over the sum of 1000 runs). For the majority of the functions, the analysis of the recursive version inferred a similar relational property as the iterative version.

For instance, the analysis of the recursive version of `length` inferred $\texttt{Id}(\mathbf{list}(\alpha))$, whereas the analysis of the iterative version inferred $\texttt{Id}(\mathbf{listseg}(\alpha, \mathbf{0x0}))$. Even if they are not the same, they are very similar. This difference is explained by the fact that in the iterative version, the function uses a temporary cursor to traverse the list that makes the analysis introduce a segment definition, whereas the recursive version does not. We observe this phenomenon for all the functions.

We now discuss the case of function `concat`, for which we obtained a non-similar relational property compared to the iterative version. The iterative version of this function is given in Figure 2.1 (page 16) and the recursive version just below.

```
1 list *concat(list *l1, list *l2) {
2   if(l1 == NULL) {
3     return l2;
4   }
5   l1->next = concat(l1->next, l2);
6   return l1;
7 }
```

For the iterative version, we obtained a disjunction of two abstract relations: one for the case where the list `l1` is empty and the other where it is not empty. For the recursive version, we also obtained a disjunction of two abstract relations for the same cases. While the cases for where `l1` is empty are identical, the cases where it is not empty are quite different. The abstract relation below represents the inferred disjunct for the case where `l1` is not null.



We observe that in the input state, the `next` field of `l1` pointed to a list that ends by the null pointer and that in the output state, it points to another list that is connected to `l2`. Between these two lists, we know that they describe exactly the same addresses and that only `next` fields may have been modified. We can explain this difference with the iterative version by the fact that they do not concat the lists in the same way. Indeed, the command '`l1->next = concat(l1->next, l2);`' may modify at each execution of the function the `next` field of `l1`, whereas in the iterative version, it does it only once after traversing the list until its last element.

9.4 Related Works

According to [SP81], most of the inter-procedural analyses are derived from two approaches: the *call-strings* approach and the *functional* approach. While the call-strings approach consists of considering a whole program as a single flow-graph, the functional approach considers a program as a collection of blocks and aims at inferring input-output

relations for these blocks. Both of these approaches have drawbacks and advantages. The functional approaches require to abstract relations, whereas the call-strings approaches only require to abstract states (that is simpler). In the context of inter-procedural analyses, functional approaches compose their input-output relations to analyze function calls without reanalyzing the function. This technique is also valid in present of recursion and has been applied in several works ([JLRS04], [BDES11]). Our work is obviously a derivation of the functional approach. The call-strings based analyses inline function calls. As this operation is simpler than the composition of abstract relations, it cannot be applied in presence of recursion. To fix this limitation, several solutions have been proposed, and most of them make the analysis much more difficult. The first one is to consider the recursive function into an iterative function with a loop. However, this technique is limited to tail-recursive functions. In the case of shape analyses, functions manipulating linked list can often be implemented in tail-recursive fashion, whereas much functions manipulating trees cannot, as they may require an implicit (or explicit) stack. To deal with non-tail recursive function, some techniques have been proposed like [RS01] and [RC11]. These techniques require to abstract the call stack. The work in [RS01] is specific to the analysis of recursive functions that manipulate linked list, and is based on three-valued logic [SRW02]. The work in [RC11] is based on separation logic [Rey02] and uses inductive definitions to abstract the call stack. However, abstracting the call stack is very hard.

Chapter 10

Conclusion and Future Directions

In this chapter, we conclude this Thesis and we discuss the future directions.

10.1 Conclusion

While relational properties are harder to abstract than state properties, they are intrinsically more expressive and they offer the ability to make the analysis modular and compositional. In the context of data structures, shape analyses based on separation logic rely on the separating conjunction ($*$) that ensures that two memory regions are disjoint, and on inductive predicates that describe precise structural invariant over complex dynamic data structures. However, separation logic formulas describe a set of states, they cannot describe relations.

The stakes of this Thesis were multiple:

1. to describe expressive input-output relations taking into account inductive data structures.
2. to infer automatically such relations.
3. to take benefits from these relations to perform a compositional analysis, that improves the scalability of the analysis without loosing too much precision.

The following paragraphs sum up how we tackled each of these stakes.

Abstract relations. In Chapter 4, we have introduced a new set of logical connectives that rely on separation logic, and that can express input-output relations over memory states. In particular, the identity relation $\texttt{Id}(\mathrm{h}^\sharp)$ ensures that the heaps abstracted by $\mathrm{h}^\sharp$ are totally identical in the input and the output states. The transform-into relation $[\mathrm{h}_\mathrm{i}^\sharp \dashrightarrow \mathrm{h}_\mathrm{o}^\sharp]$ expresses that the heaps abstracted by $\mathrm{h}_\mathrm{i}^\sharp$ have been transformed into the heaps abstracted by $\mathrm{h}_\mathrm{o}^\sharp$. Moreover, the relational separating conjunction $*_\mathtt{R}$ describes independent relations and allows local reasoning on them. With these new connectives, we have built an abstract domain that supports inductive data structures and that over-approximates input-output relations over memory states.

However, transform-into relations do not describe any specific relation, but only the transformation of some part of the memory into another. Thus, in Chapter 6, we have proposed a generic extension of our relational connectives to improve their precision. More precisely, the transform-into predicates are now parameterized by sets of relational predicates, called abstract heap transformation predicates domains. The abstract relation $[h_i^\sharp \dashrightarrow h_o^\sharp]_{t^\sharp}$ signifies that the heaps abstracted by $h_i^\sharp$ has been transformed into the heaps abstracted by $h_o^\sharp$ according to the relation described by the predicate $t^\sharp$. We have formalized three abstract domains of such predicates, that are all data structures agnostic.

Automatic inference of abstract relations. In Chapter 5, we have designed a static analysis by abstract interpretation that infers automatically abstract relations over memory states. This analysis starts from the identity relation of a given pre-condition, and applies a corresponding transfer function to each program command (assignment, allocation, deallocation) to compute new abstract relations. This analysis needs also to unfold inductive predicates in order to materialize summarized memory cells. Moreover, it defines standard operations over lattices such as inclusion test, join and widening to deal with condition tests, loops, and to ensure termination. This analysis is intra-procedural: it does not handle function calls. We have implemented a prototype static analyzer as a Frama-C plug-in that supports both a classic state shape analysis and our relational shape analysis. We have obtained positive results by running both analyses on short functions manipulating singly linked lists and binary trees. In most of the cases, our relational analysis inferred stronger properties than the state analysis in a reasonable time.

Moreover, we have designed an extension of our relational intra-procedural shape analysis, that can take into account generically any abstract heap transformation predicates domains in Chapter 6. We have experimented this extended relational analysis and improved the results obtained for the first version without the relational predicates, and obtained similar results to an approach by verification deductive for the verification of a list module of the Contiki operating system.

Compositional analysis. In Chapter 8, we have lifted our relational intra-procedural shape analysis into a compositional inter-procedural shape analysis. Now, our analysis handles function calls, and takes benefits of the abstract relations. Indeed, they are used as function summaries that are composed with the current abstract relation when a function is called. It mainly relies on the composition operator over abstract relations that is itself built upon the abstract intersection of abstract memory states, both defined in Chapter 7. Composing abstract relations makes the relational analysis much more efficient: this allows to not reanalyze the body of functions every time they are called. Moreover, the expressiveness of our abstract relations allows a minimal loss of precision during the abstract composition, compared to an analysis that inlines functions. Our experimental evaluations show that our compositional analysis exponentially more efficient

than a state analysis that inlines functions but with no loss of precision for programs manipulating shapes with deep function calls. Finally, we have designed an extension of this compositional inter-procedural shape analysis that handles recursive functions in Chapter 9.

10.2 Future Directions

In this section, we discuss some possible future directions for our works. They are divided into three main axes: some that could improve the result of the analysis defined in this Thesis, some that could allow to express more relational properties, and some that could use our abstract relation for other kinds of analyses.

10.2.1 Improvement of our Analysis

Analysis of mutually recursive functions. We believe that our technique to analyze recursive functions can also be applied to mutually recursive functions. Indeed, it widens the function summaries of functions until reaching a fixpoint. This process should then work for mutually recursive functions.

Using silhouettes abstractions. The recent work in [LBCR17] introduces *silhouettes*, that abstract path relations between live program variables. Silhouettes are applied not only to clump relevant disjuncts of abstract memory states but also to guide the weakening operations over abstract memory states. We could adapt silhouettes to our abstract memory relations. In a first time, we could only use silhouettes for clumping and the weakening operations. As this improves the performance of state shape analyses, this could also improve the performance of our relational analysis. In a second time, we could use silhouettes to guide our abstract composition operator. Indeed, the current implementation of our abstract composition tries to compose all the disjuncts of the first disjunction with all the disjuncts of the second disjunction, the not composable abstract memory relations are automatically detected and discarded dynamically during the composition. We believe that silhouettes can help to detect before performing the abstract composition relevant abstract memory relations to compose.

10.2.2 Abstracting More Relational Properties

Designing other abstract heap transformation predicates domains. The abstract heap transformation predicates domains designed in this Thesis are totally independent from data structures. While they can prove interesting relational properties, they cannot prove properties specific to data structures, such as a list reversal or a list

sort. As we have designed an analysis parameterized by any abstract heap transformation predicates domain, we could formalize new ones to describe more specific relational properties.

Expressing non local abstract relations. An abstract heap transformation predicates describes a specific relation for its attached transform-into relation. This specific relation is *local*: it does not describe anything about the other abstract relations. Consequently, our abstract relations cannot express that two disjoint data structures have the same size, or contain the same data. We could draw inspiration from the work in [BDE+10] that can describe and infer such properties for linked lists.

10.2.3 Designing other Analyses

Combining state and relational analyses. In this Thesis, we have proposed an analysis that is fully relational, in contrast with a full state analysis. The inferred abstract relations are used to describe the behavior of functions. However, we can imagine that the abstract relations describe the behavior of more specific pieces of programs that are expensive to analyse, such as branches or loops. In turn, we could perform a state analysis, that is switched into a relational analysis for some pieces of programs. The inferred abstract relation could then be composed with the switched abstract state. This process could improve the performance of the state analysis, and make it faster than our compositional analysis.

Use the relational abstract domain with other kind of analyses. In this Thesis, we have used our relational abstract domain to designed a Top-down program analysis, that starts from a root function and that analyzes the program until its leafs. However, we could apply it for Bottum-up analyses, that begin from leaf functions and proceed from callees to callers, in the style of [CDOY09, GCRN09]. We believe that these analyses, based on the bi-abduction inference, could benefit of our abstract relations to obtain more compact and more precise summaries, instead of using tabulations of pre-post conditions. We also could inspire from [ZMNY14], that combines both Top-down and Bottom-up analyses in a manner that gains their benefits without suffering their drawbacks.

Bibliography

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *ACM SIGPLAN Notices*, volume 41, pages 91–102. ACM, 2006.

[ACI10] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 267(1):3–16, 2010.

[And94] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[ARR+07] Daphna Amit, Noam Rinetzky, Thomas Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *Conference on Computer Aided Verification (CAV)*, pages 477–490. Springer, 2007.

[AS87] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W O'hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Conference on Computer Aided Verification (CAV)*, pages 178–192. Springer, 2007.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W O'hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[BDE+10] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, Ahmed Rezine, and Mihaela Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *Conference on Computer Aided Verification (CAV)*, pages 72–88. Springer, 2010.

[BDES09] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. A logic-based framework for reasoning about composite data structures. In *International Conference on Concurrency Theory*, pages 178–195. Springer, 2009.

[BDES11] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 578–589. ACM, 2011.

[BDES12] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–22. Springer, 2012.

[BFM+08] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI C specification language, 2008.

[BKL18] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Ghosts for lists: A critical module of contiki verified in frama-c. In *Tenth NASA Formal Methods Symposium-NFM 2018*. Springer, 2018.

[BNR08] Anindya Banerjee, David A Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 387–411. Springer, 2008.

[CBC93] Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Symposium on Principles of Programming Languages (POPL)*, pages 232–245, 1993.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Conference on Compiler Construction*, pages 159–179. Springer, 2002.

[CCR15] Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. Desynchronized multi-state abstractions for open programs in dynamic languages. In *European Symposium on Programming (ESOP)*, pages 483–509. Springer, 2015.

[CDOY07] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Footprint analysis : A shape analysis that discovers preconditions. In *Static Analysis Symposium (SAS)*, pages 402–418. Springer, 2007.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Symposium on Principles of Programming Languages (POPL)*, pages 289–300. ACM, 2009.

[CES86] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages (POPL)*, pages 84–97. ACM, 1978.

[CNR+15] Ghila Castelnuovo, Mayur Naik, Noam Rinetzky, Mooly Sagiv, and Hongseok Yang. Modularity in lattices: A case study on the correspondence between top-down and bottom-up analysis. In *Static Analysis Symposium (SAS)*, pages 252–274. Springer, 2015.

[Cou97] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6:77–102, 1997.

[CP17] Arthur Charguéraud and François Pottier. Temporary read-only permissions for separation logic. In *European Symposium on Programming (ESOP)*, pages 260–286. Springer, 2017.

[CR07] Sigmund Cherem and Radu Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 234–250. Springer, 2007.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 247–260. ACM, 2008.

[CR13] Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In *Electronic Proceedings in Theoretical Computer Science*, pages 161–185. OPA, 2013.

[CRL99] Ramkrishna Chatterjee, Barbara G Ryder, and William A Landi. Relevant context inference. In *Symposium on Principles of Programming Languages (POPL)*, pages 133–146. ACM, 1999.

[CRN07] Bor-Yuh Evan Chang, Xavier Rival, and George C Necula. Shape analysis with structural invariant checkers. In *International Static Analysis Symposium*, pages 384–401. Springer, 2007.

[CWZ90] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 296–310, 1990.

[DDAS11] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 567–577. ACM, 2011.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proceedings of the 1992 international conference on computer languages*, pages 2–13. IEEE, 1992.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *ACM Sigplan Notices*, 29(6):230–241, 1994.

[DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[DKR04] Dino Distefano, Joost-Pieter Katoen, and Arend Rensik. Who is pointing when to whom? In *Foundations of Software Technology and Theoretical (FSTTCS)*, pages 250–262. Springer, 2004.

[DKR05] Dino Distefano, Joost-Pieter Katoen, and Arend Rensik. Safety and liveness in concurrent pointer programs. In *Formal Methods for Components and Objects (FMCO)*, pages 280–312. Springer, 2005.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages (POPL)*, pages 207–212. ACM, 1982.

[DOY06] Dino Distefano, Peter O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 287–302. Springer, 2006.

[EC80] E Emerson and Edmund Clarke. Characterizing correctness properties of parallel programs using fixpoints. *Automata, Languages and Programming*, pages 169–181, 1980.

[FFJ12] Pietro Ferrara, Raphael Fuchs, and Uri Juhasz. Tval+: Tvla and value analyses together. In *International Conference on Software Engineering and Formal Methods*, pages 63–77. Springer, 2012.

[FLF+10] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. Reasoning about optimistic concurrency using a program logic for history. In *International Conference on Concurrency Theory*, pages 388–402. Springer, 2010.

[Flo67] Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[GBC06] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium (SAS)*, pages 240–260. Springer, 2006.

[GCRN09] Bhargav S Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V Nori. Bottom-up shape analysis. In *Static Analysis Symposium (SAS)*, pages 188–204. Springer, 2009.

[GH96] Rakesh Ghiya and Laurie J Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15. ACM, 1996.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

[GVA07] Bolei Guo, Neil Vachharajani, and David I August. Shape analysis with inductive recursion synthesis. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 256–265. ACM, 2007.

[HHL+15] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forester: Shape analysis using tree automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 432–435. Springer, 2015.

[HHN92] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis of imperative programs. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, pages 249–260, 1992.

[HHR+11] Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. Forest automata for verification of heap manipulation. In *Conference on Computer Aided Verification (CAV)*, pages 424–440. Springer, 2011.

[HN90] Laurie J Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on parallel and distributed systems*, 1(1):35–47, 1990.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HR05] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Acm Sigplan Notices*, volume 40, pages 310–323. ACM, 2005.

[JLRS04] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symposium (SAS)*, pages 246–264. Springer, 2004.

[JLRS10] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):5, 2010.

[JM82] Neil D Jones and Steven S Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 66–74. ACM, 1982.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.

[Kas06] Ioannis T Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *International Symposium on Formal Methods*, pages 268–283. Springer, 2006.

[KJ14] Gowtham Kaki and Suresh Jagannathan. A relational framework for higher-order shape analysis. In *International Conference on Functional Programming (ICFP)*, pages 311–324. ACM, 2014.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[Kle52] Stephen C. Kleene. *Introduction to metamathematics*. Bibliotheca Mathematica. North-Holland, Amsterdam, 1952.

[KRR+13] Jörg Kreiker, Thomas Reps, Noam Rinetzky, Mooly Sagiv, Reinhard Wilhelm, and Eran Yahav. Interprocedural shape analysis for effectively cutpoint-free programs. In *Programming Logics*, pages 414–445. Springer, 2013.

[LAIS06] Tal Lev-Ami, Neil Immerman, and Mooly Sagiv. Abstraction for shape analysis with fast and precise transformers. In *Conference on Computer Aided Verification (CAV)*, pages 547–561. Springer, 2006.

[LAS00] Tal Lev-Ami and Mooly Sagiv. Tvla: A system for implementing static analyses. In *International Static Analysis Symposium*, pages 280–301. Springer, 2000.

[LBCR17] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. Semantic-directed clumping of disjunctive abstract states. In *Symposium on Principles of Programming Languages (POPL)*, volume 52, pages 32–45. ACM, 2017.

[LBR98] Gary T Leavens, Albert L Baker, and Clyde Ruby. Jml: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420, 1998.

[LGQC14] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Conference on Computer Aided Verification (CAV)*, pages 52–68. Springer, 2014.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 21–34, 1988.

[LR92] William Landi and Barbara G Ryder. A safe approximate algorithm for interprocedural aliasing. In *ACM SIGPLAN Notices*, volume 27, pages 235–248. ACM, 1992.

[LRC15] Huisong Li, Xavier Rival, and Bor-Yuh Evan Chang. Shape analysis for unstructured sharing. In *Static Analysis Symposium (SAS)*, pages 90–108. Springer, 2015.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[MBCC07] Stephen Magill, Josh Berdine, Edmund Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In *International Static Analysis Symposium*, pages 419–436. Springer, 2007.

[MHKS08] Mark Marron, Manuel Hermenegildo, Deepak Kapur, and Darko Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *International Conference on Compiler Construction*, pages 245–259. Springer, 2008.

[Min06] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[MNCL06] Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. Inferring invariants in separation logic for imperative list-processing programs. *SPACE*, 1(1):5–7, 2006.

[MT91] Robin Milner and Mads Torte. Co-induction in semantics. *Theoretical computer science*, 87:209–220, 1991.

[O'H07] Peter W O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[PC06] Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345. Springer, 2006.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[RBR+05] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *ACM SIGPLAN Notices*, volume 40, pages 296–309. ACM, 2005.

[RC11] Xavier Rival and Bor-Yuh Evan Chang. Calling context abstraction with shapes. In *Symposium on Principles of Programming Languages (POPL)*, volume 46, pages 173–186. ACM, 2011.

[RCK07] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007.

[Rey02] John Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logics In Computer Science (LICS)*, pages 55–74. IEEE, 2002.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):pp. 358–366, 1953.

[RPHR+07] Noam Rinetzky, Arnd Poetzsch-Heffter, Ganesan Ramalingam, Mooly Sagiv, and Eran Yahav. Modular shape analysis for dynamically encapsulated programs. In *European Symposium on Programming*, pages 220–236. Springer, 2007.

[RS01] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. In *International Conference on Compiler Construction*, pages 133–149. Springer, 2001.

[RSY05] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Static Analysis Symposium (SAS)*, pages 284–302. Springer, 2005.

[Rug04] Radu Rugina. Quantitative shape analysis. In *International Static Analysis Symposium*, pages 228–245. Springer, 2004.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications*, 1981.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[SRW99] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages (POPL)*, pages 105–118, 1999.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–41. ACM, 1996.

[TJ07] Mana Taghdiri and Daniel Jackson. Inferring specifications to detect errors in code. *Automated Software Engineering*, 14(1):87–121, 2007.

[YRSW03] Eran Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolution logic. In *European Symposium on Programming (ESOP)*, pages 204–222. Springer, 2003.

[YYC08] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *ACM SIGPLAN Notices*, volume 43, pages 221–234. ACM, 2008.

[ZMNY14] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid top-down and bottom-up interprocedural analysis. In *Conference on Programming Language Design and Implementation (PLDI)*, volume 49, pages 249–258. ACM, 2014.

[ZPJ16] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 491–507. ACM, 2016.

RÉSUMÉ

Les analyses statiques ont pour but d'inférer des propriétés sémantiques de programmes. Nous distinguons deux importantes classes d'analyses statiques : les analyses d'états et les analyses relationnelles. Alors que les analyses d'états calculent une sur-approximation de l'ensemble des états atteignables d'un programme, les analyses relationnelles calculent des propriétés fonctionnelles entre les états d'entrée et les états de sortie d'un programme. Les analyses relationnelles offrent plusieurs avantages, comme leur capacité à inférer des propriétés sémantiques plus expressives par rapport aux analyses d'états. De plus, elles offrent également la possibilité de rendre l'analyse compositionnelle, en utilisant les relations entrée-sortie comme des résumés de procédures, ce qui est un avantage pour le passage à l'échelle. Dans le cas des programmes numériques, plusieurs analyses ont été proposées qui utilisent des domaines abstraits numériques relationnels, pour décrire des relations. D'un autre côté, modéliser des abstractions de relations entre les états mémoires entrée-sortie tout en prenant en compte les structures de données est difficile. Dans cette Thèse, nous proposons un ensemble de nouveaux connecteurs logiques, reposant sur la logique de séparation, pour décrire de telles relations. Ces connecteurs peuvent exprimer qu'une certaine partie de la mémoire est inchangée, fraîchement allouée, ou désallouée, ou que seulement une seule partie de la mémoire a est modifiée (et de quelle manière). En utilisant ces connecteurs, nous construisons un domaine abstrait relationnel et nous concevons une analyse statique compositionnelle par interprétation abstraite qui sur-approxime des relations entre des états mémoires contenant des structures de données inductives. Nous avons implémenté ces contributions sous la forme d'un plug-in de l'analyseur FRAMA-C. Nous en avons évalué l'impact sur l'analyse de petits programmes écrits en C manipulant des listes chaînées et des arbres binaires, mais également sur l'analyse d'un programme plus conséquent qui consiste en une partie du code source d'Emacs. Nos résultats expérimentaux montrent que notre approche permet d'inférer des propriétés sémantiques plus expressives d'un point de vue logique que des analyses d'états. Elle se révèle aussi beaucoup plus rapide sur des programmes avec un nombre conséquent d'appels de fonctions sans pour autant perdre en précision.

MOTS CLÉS

analyse statique, analyse compositionnelle, propriétés relationnelles, structures de données dynamiques, logique de séparation, interprétation abstraite

ABSTRACT

Static analyses aim at inferring semantic properties of programs. We distinguish two important classes of static analyses: state analyses and relational analyses. While state analyses aim at computing an over-approximation of reachable states of programs, relational analyses aim at computing functional properties over the input-output states of programs. Relational analyses offer several advantages, such as their ability to infer semantics properties more expressive compared to state analyses. Moreover, they offer the ability to make the analysis compositional, using input-output relations as summaries for procedures, which is an advantage for scalability. In the case of numeric programs, several analyses have been proposed that utilize relational numerical abstract domains to describe relations. On the other hand, designing abstractions for relations over input-output memory states and taking shapes into account is challenging. In this Thesis, we propose a set of novel logical connectives to describe such relations, which rely on separation logic. This logic can express that certain memory areas are unchanged, freshly allocated, or freed, or that only part of the memory is modified (and how). Using these connectives, we build an abstract domain and design a compositional static analysis by abstract interpretation that over-approximates relations over memory states containing inductive structures. We implement this approach as a plug-in of the FRAMA-C analyzer. We evaluate it on small programs written in C that manipulate singly linked lists and binary trees, but also on a bigger program that consists of a part of Emacs. The experimental results show that our approach allows to infer more expressive semantic properties than states analyses, from a logical point of view. It is also much faster on programs with an important number of function calls without losing precision.

KEYWORDS

static analysis, compositional analysis, relational properties, dynamic data structures, separation logic, abstract interpretation