Habilitation à Diriger des Recherches

ÉCOLE NORMALE SUPÉRIEURE

Xavier Rival

2011

Domaines Abstraits pour l'Analyse Statique de Programmes Manipulant des Structures de Données Complexes

Abstract Domains for the Static Analysis of Programs Manipulating Complex Data Structures

---

Résumé

Dans ce mémoire, je présente les travaux de recherche que j'ai menés au cours des cinq dernières années sur l'analyse statique des programmes manipulant des structures de données dynamiques. Mes travaux reposent sur le cadre de travail de l'interprétation abstraite, qui permet de calculer de manière automatique, et en temps fini des invariants de programmes, exprimés dans un domaine abstrait, c'est-à-dire une algèbre de prédicats munie d'opérateurs d'analyse. Mon objectif à long terme est de concevoir un domaine abstrait généraliste qui soit adapté à la représentation de propriétés des états mémoire de programmes manipulant des structures de données complexes. Les travaux présentés dans ce mémoire posent les fondations d'un tel domaine abstrait et visent à permettre son implantation sous la forme d'une librairie indépendante, pouvant être integrée à un analyseur statique, sans modification importante.

Ce domaine abstrait peut être praramétré de plusieurs manières. Tout d'abord, il dépend du choix d'un domaine abstrait numériques qui permet de représenter les contraintes entre valeurs de types de base (entiers, flottants, booléens...). Par ailleurs, il est également paramétré par le choix d'un ensemble de définitions décrivant les structures de données devant être considérées au cours de l'analyse. Pour cela, nous utilisons un langage de définitions inductives, qui permet d'exprimer une vaste famille de structures de données, incluant par exemple de nombreux types de listes ou d'arbres. Le chapître 2 décrit ce domaine abstrait et son instantiation avec divers paramètres classiques.

Ensuite, le chapître 3 présente les fonctions abstraites de ce domaine abstrait, permettant ainsi de l'utiliser pour l'analyse de programmes impératifs dans un langage tel qu'un sous-ensemble minimal de C ou Java. En particulier, le "dépliage" permet de rendre plus concrête la description d'une partie de la mémoire, et ainsi d'utiliser des fonctions de transfert abstraites standard pour les affectations ou les tests. Par ailleurs, les opérations de comparaison et d'union abstraite reposent sur la possibilité de "replier" une description concrête d'une partie de la mémoire.

De plus, il est important de prendre en compte les aspects spécifiques de certains langages de programmation. Ainsi, le langage C, qui est très utilisé pour le développement de logiciels embarqués ou de systèmes d'exploitations, permet au programmeur d'effectuer des manipulations de bas niveau. Nous proposons dans le chapître 4 une adaptation de notre domaine pour traiter de tels langages.

Par ailleurs, le domaine abstrait présenté dans ce mémoire est adapté à la représentation de structures de données complexes, et permet en particulier une abstraction précise et élégante de la pile d'appels de programmes contenant des fonctions récursives. Une telle abstraction est très importante pour l'analyse de programmes fonctionnels. Cette application est présentée dans le chapître 5.

Enfin, nous présentons dans le chapître 6 quelques unes des nombreuses extensions et applications possibles du domaine abstrait introduit dans ce mémoire, qui correspondent pour certaines à des travaux d'ores-et-déjà engagés.

Abstract

In this manuscript, I present the research work I have been carrying out over the last five years on the topic of static analysis of programs which manipulate complex dynamic data structures. This work is based on the the abstract interpretation framework, which allows to compute program invariants automatically and using a finite amount of resources (memory, time). A static analysis is based on an abstract domain, that is a predicate algebra which provides analysis operators. My long term goal is to design a general abstract domain for expressing and inferring properties of programs that manipulate complex data structures. The results which are described in this manuscript lay out the basis for such an abstract domain, and take part to a larger effort to implement it as a standalone library that could be integrated in various static analysis tools, without requiring radical changes.

This abstract domain can be parameterized in several ways. First, it depends on the choice of a numerical abstract domain, in order to represent constraints among values of base types (integers, floating point, booleans...). Furthermore, it is parameterized by the choice of a set of definitions that describe the data structures that should be considered by the analysis. In order to do that, we use a language of inductive definitions, which allows to describe a large family of common data structures including many kinds of lists and trees. Chapter 2 describes this abstract domain and some classical instantiations.

Then, chapter 3 presents the abstract transfer functions of this abstract domain, which allow to use it for the static analysis of imperative programs written in a language such as a minimal subset of C or Java. In particular, the "unfolding" operation allows to make the representation of a memory region more concrete, so that standard transfer functions can be used for assignments or conditions. Moreover, the abstract comparison and union operations require to fold back a concrete description of a fragment of the memory.

In practice, static analysis tools should support specific features of programming languages. For instance the C language supports low level operations and is very commonly used in embedded systems or operating systems. In chapter 4, we adapt our abstract domain so as to handle such programming languages features.

Besides, our abstract domain for expressing properties of memory states can represent very complex data structures, and can capture a precise and elegant abstraction of the call stack of programs with recursive procedures. Such an abstraction is important for the analysis of functional programs. This application is described in chapter 5.

Last, we present a few extensions and applications of our abstract domain in chapter 6, some of which have already been the topic of investigations.

Contents

Chapter 1

Introduction

In this chapter, we overview the state of the art and the current challenges in program verification with a particular emphasis on safety critical embedded systems. Moreover, we introduce the issue of the verification of programs manipulating complex data structures, and the need for abstract domains adapted to their analysis.

1.1 The program verification challenge

Today, many complex and critical systems rely on large softwares. For instance, a modern aircraft relies on many computer controlled systems, including fly-by-wire command (controlling ailerons, elevators, flaps and other flight surfaces), autopilot, FADEC (Full Authority Digital Engine Control, taking full control of all engine parameters), flight warning systems, communication and air traffic control control softwares, etc. Each of these computers runs a very large and complex software comprising typically more than one million lines of C code. A bug in such a program may have severe consequences, ranging from more or less serious inconvenience (e.g., when a monitoring program reports a false issue with the aircraft systems, causing a flight diversion) to an immediate loss of control, and a major accident.

The literature provides many examples of high profile embedded system failures the root cause of which can be tracked down to a software bug. The loss of the Ariane 501 flight in 1996, with the destruction of payload satellites is one of the most cited such occurrences: the first flight of the Ariane 5 satellite launcher failed due to an arithmetic error (an integer overflow arose when attempting to convert a 64-bits floating point number into a 16-bits integer), which caused the launcher trajectory control computers to crash. The inquiry board did conclude the disaster was the consequence of multiple design issues [74, 1]; in particular, both the main and backup systems were running the same software; the assumptions that were used to validate it were not compatible with Ariane 5 but with Ariane 4 (from which code was reused) and the computation that led to the launcher loss was useless after takeoff. The estimated cost of this failure adds up to more than $ 300.000.000, due to the loss of satellites and the one year down time of Ariane 5, following flight 501 failure.

The Ariane 501 flight failure is not the only example of disaster caused by software errors. In 1991, the Patriot missile failure at Dahran [114] was caused by accumulated imprecisions in fixed point computations, leading to the failure to confirm the trajectory of a foe missile (28 fatalities). In 1992, control systems of US navy ship Yorktown went down during over three hours due to a division by zero [115] bug.

Such failures are obviously unacceptable in modern critical systems. Domain specific regulations set standards for software development, so as to prevent catastrophic errors. For instance, avionic software systems regulation DO 178 [45] requires designers to carefully express what consequences a fault may have, for each component. The most critical components such as flight-by-wire systems are assigned level A (which means a failure would put the aircraft into a very imminent and severe risk) are required to be certified with respect to stringent requirements whereas non critical components (e.g., level E, like in-flight entertainment systems) do not need anywhere near the same amount of certification work. Components such as navigation or flight warning management systems have a high criticality level (typically B or C), as a failure of these systems would likely cause a serious flight disruption (yet it would not prevent the safe control of the airplane as opposed to a flight-by-wire system failure).

As a consequence, the development cost of embedded softwares is very high and typically represents over 90 % of the overall cost of the systems. Standard solutions used in industry range from manual code review to testing, which are both expensive and not satisfactory in the sense that they do not prove the correctness of the software, even

though important efforts are made to ensure errors are very likely to be found in the process (for instance, it is common to separate development and certification teams, so as to prevent both groups to make the same mistake). Indeed, human inspection is not reliable, and testing is incomplete when it is impossible to test *all* executions, which is the case of all large scale systems. Moreover, such approaches make software maintenance very costly: for instance, when a bug is found late in the development process and needs to be fixed at that point, testing campaigns should usually be completely redone, which is long and costly.

Therefore, *automatic* tools are a necessity, in order to both improve the level of trust in the results and to reduce the cost of certification. Ideally, verification tools should be *correct* (i.e., able to catch all issues of a certain kind), *complete* (i.e., never reject correct programs) and *automatic* (i.e., able to run without any manual intervention). Since most relevant program properties (like the absence of runtime errors) are *not computable* a common approach consists in dropping completeness: this means analysis tools may reject some correct programs and raise alarms (i.e., indication that the property of interest could not be proved). The important point is that no incorrect code will pass unnoticed, which means the level of trust in the final code is very high. Ideally static analysis tools should be *precise*, that is reject few correct programs (at least among some families of relevant applications) and *efficient* so as to cope with large, industrial size applications.

1.2 Program analysis in safety critical embedded systems

In the last decade we have observed a dramatic progress in software static analysis tools.

Several approaches have been pursued. A first approach consists in designing static analyses targeted at all programs written in a given language. As it is very difficult to treat in a very precise manner such large sets of programs, analyzers following this approach usually sacrifice precision.

In the other hand, specializing analysis tools to some carefully selected families of programs and of properties makes it easier to achieve fully automatic proofs of correctness even though the analysis is actually incomplete. The idea is that static analyzers which follow this approach should treat well interesting classes of programs, even though there exists many programs that they would not be able to analyze precisely.

In particular, many static analyses were proposed for softwares such as fly-by-wire control systems: these programs are highly critical (level *A* in civil aeronautics) and mostly perform complex numerical computations. These characteristics can be exploited for the specialization of analysis tools [11]. A wide range of properties have been successfully verified on such applications, including:

- **Bounds on worst case execution time:** fly-by-wire softwares follow a synchronous design [19] thus it is important to verify each task will not take more than the time allocated to it; worst-case execution time analyses [3, 118] compute precise and safe over-approximations for worst case execution times, taking into account complex cache and pipeline behaviors.
- **Absence of runtime errors and undetermined behaviors:** runtime errors such as the arithmetic error that caused the failure of Ariane 501 [1] should be avoided at all cost; the ASTRÉE analyzer [10, 11, 34, 8] is able to prove the absence of runtime errors by fully automatic static analysis, based on abstract interpretation [29]; it was applied not only to avionic softwares [41] but also to spaceship control programs [7].
- **Precision of floating point computations:** precision and stability issues have been the source of major problems (including the Patriot missile failure); hence, static analyzers such as FLUCTUAT [52, 40] have been designed so as to bound the imprecision errors in floating point computations.

These static analyses are all based on the *abstract interpretation* frameworks [29, 30, 31]. The principle of abstraction is to rely on an *abstract domain*, that is a family of predicates, which can express not only the property of interest but also all intermediate invariants required in the proof. Then, abstract interpretation performs a *conservative* analysis of programs using the abstract domain, where each concrete computation step is over-approximated so as to account for *all* concrete behaviors: this soundness property follows from the use of conservative analysis operators, and from a step-by-step over-approximation of the programs concrete semantics. Furthermore, widening operators ensure the termi-

nation of static analyses even when in presence of infinite concrete executions and using infinite abstract domains.

I took part to the design and implementation of the ASTRÉE analyzer since the beginning of the project in 2001, together with (in alphabetical order) Bruno BLANCHET, Patrick COUSOT, Radhia COUSOT, Jérôme FERET, Laurent MAUBORGNE, Antoine MINÉ and David MONNIAUX. As we started this project before the end of my PhD, I do not discuss it in detail in this manuscript. As part of this project, I designed a trace partitioning abstract domain [83, 108] together with Laurent MAUBORGNE; this abstract domain relies on history properties of execution traces as a criterion for creating and collapsing disjunctions of invariants, improving both precision and efficiency.

In parallel, I studied techniques to verify either target code using source invariants [103, 102] or to verify compilation using translation validation approach [104], so as to ensure that target code could be verified correct, as requested by the DO 178 [45] regulations. I also investigated techniques for the investigation of the origin of alarms raised by static analyzers such as ASTRÉE, using backward analysis [106] and abstract dependencies [105].

Since 2009, the ASTRÉE analyzer has been licensed to AbsInt Angewandte Informatik for industrialization [65], and is now offered for sale, to companies dealing with embedded softwares written in C. The ASTRÉE project also contributed to improve the theory of programs static analysis, e.g. allowing to compare various sorts static analysis approaches [37], understanding the issue of scalability [36] and the ways of combining several abstract domains into powerful analysis tools [35].

ASTRÉE has been successful not only for the analysis of fly-by-wire applications, but also when applied to other families of programs. This strong result could be achieved thanks to a modular design of the abstract domain, which combines [30, 35] a wide collection of simpler abstract domains, that take care of basic programs features. Some layers abstract executions traces [108] and memory states [88] whereas many numerical abstract domains [87, 89, 48, 47] can deal with various computations performed in programs to analyze floating point computations (with rounding errors that need be abstracted), digital filters, boolean control... This architecture also allows to incrementally extend the analyzer so as to cope with new kinds of programs, by adding new abstract domains to the modular domain.

1.3 Need for analyses targeted at memory properties

Despite these very important progresses, many embedded programs verification problems remain open. Mainly, the current state of the art does not allow to verify some other interesting classes of properties and of programs.

Outstanding challenges: While many families of safety properties can now be treated reasonably well, we cannot say the same for *functional properties*, specifying that a program will produce the expected results when in presence of some given inputs.

Another outstanding challenge consists in verifying the whole system including software, sensors and the environment in a *closed loop*, which requires modeling the environment.

In the other hand, other families of programs than those mentioned in section 1.2 should be analyzed properly. Indeed, the ASTRÉE analyzer [11] focuses on synchronous applications which perform mostly numerical computations. As of today, the two main challenges in terms of classes of applications are related to the presence of *asynchronous behaviors* (which means static analyzers should abstract parallel executions) and to the use of *complex memory structures* (which means static analyzers should abstract intricate sets of memory states). My interest for symbolic abstractions led me to consider the reasoning on programs manipulating complex data structures during the last five years.

Programs manipulating complex memory states: Many critical embedded codes deal with non trivial data structures, including navigation softwares, flight warning systems or air traffic control programs. These systems are typically level B or C following to the DO 178 regulation terminology, thus, they need to comply with rather stringent qualification requirements. Some level A codes like device drivers for fly-by-wire system interfaces [90] also manipulate such complex structures.

Dynamic memory allocation is usually not allowed in these programs, yet this does not necessar-

ily make the problem any simpler, as they may work on large data structures, which are allocated statically, but where pointers are dynamic, and where complex memory invariants should be preserved. When these structures are large, static analyzers have to carry out some kind of abstraction, in order to be efficient enough: for instance, while a modified version of ASTRÉE can be applied to the verification of an embedded USB driver [90], this analysis turns out less efficient by an order of magnitude than for purely numeric codes, due to the lack of a more effective memory abstraction among other reasons. Therefore, analyzing such programs is just as hard as if they were relying on dynamic memory allocation.

Similar needs for precise and efficient abstraction for complex memory states also arise when considering other common families of programs including operating systems components or device drivers [6]. Analyzing multi-threaded programs also requires precise heap abstraction [94]. In the same way as very diverse numerical algorithms required many well chosen numerical abstract domains to be developed, we should not expect a simple universal memory abstract domain to cope with all interesting cases.

Properties of interest: As programs manipulating complex data structures may cause other kinds of errors, the task of analyses for such classes of programs is actually more ambitious than that of analyses for fly-by-wire codes.

Indeed, these analyses should not only capture a precise enough view of the memory states to allow properties be proved about the numerical computations, but also allow to prove specific properties including:

- the **safety** of the program, that is the absence of runtime errors (null or dangling pointer dereference);
- the **preservation of structural invariants**, which means that the data structures manipulated by the program cannot be corrupted; in many cases, proving preservation of structural invariant is a prerequisite for proving safety, since pointer dereferences are correct only under the assumption that the structures are well formed;
- the **functional correctness** of programs which are supposed to produce outputs that verify some known structural invariants.

As an example, we consider a program operating on binary trees. Safety states that the program should not perform any illegal pointer operation while traversing the tree or while dereferencing pointers. When the program performs destructive updates to the tree structure (e.g., in order to add an element), then the analysis should prove that the structure is still a well formed tree in the end. Safety may indeed rely on that preservation property: if the tree structure is read again after being modified, the read operation will cause no runtime error only under the assumption that the tree is still well formed (otherwise, reading a branch pointer may cause an invalid or dangling pointer dereference). Functional correctness may be particularly relevant when a program builds a structure which is supposed to be manipulated by other functions afterwards, as is the case of, e.g., an initialization routine, which should create a *balanced* tree from a set of unlinked static nodes.

Existing analyses for programs manipulating complex data structures: Early works on pointer analyses were targeted at the inference of points-to or aliasing relations among pointers and memory locations [22, 70, 24]. Later, analyses based on store-less models encoded more complex aliasing properties using, e.g. numerical constraints to capture unbounded numbers of aliasing relations as in [42] or more recently in [116]. In the other hand, many very fast pointer analyses were developed, that can handle very large codes [58].

However, such analyses cannot express precise data structure invariants or prove the preservation of structural invariants. Very evolved pointer analyses did attempt to capture shape properties using subtle encodings as [43]. By contrast, shape analyses for analyzing programs that perform destructive update on complex data structures were developed using large families of logical formulas, such as graphs [110], three valued logic formulas [111, 73] or separation logic formulas [60, 97] like SPACEINVADER [44], or on predicate abstraction [79]. These analyses are much more complex than regular points-to analyses, but also more expressive. For instance, TVLA [111] is an extensible analysis framework were users may define predicates to capture the data structures they are interested in.

Initially, these analyses were viewed as expensive analyses, yet dramatic progresses have been

accomplished in terms of efficiency, using improved algorithms [78, 13] and finely tuned abstract domains [6, 123].

Meanwhile, a wide body of works were done for the verification of programs manipulating complex structures. This solution requires programmers to annotate programs with invariants for at least some control points (typically, loops or function boundaries). Among others, [121, 59, 124] followed this approach. HAVOC [23] allows to express properties of arrays and singly-linked list, for verification. Formalisms such as separation logic were also used to (manually) formalize proofs of complex problems such as the Deutsch-Shorr-Waite algorithms [122].

Last, we remark that many analyses were developed for specific data structures, such as strings [113, 2], buffers [46] and arrays [50, 57, 38]. Such analyses usually make abstractions which are very specific and cannot be generalized, but which handle complex properties of buffers, strings, arrays; in particular the invariants of these structures typically involve non trivial numeric properties that other shape analyses cannot track. Similarly, few analyses were designed for inferring both shape properties and properties on values [76, 84].

Context: This work was started as part of a collaboration with Bor-Yuh Evan Chang during my post doctorate internship at UC Berkeley in the group of George Necula, and while Bor-Yuh Evan Chang was preparing his PhD Thesis in George Necula's group. We did actively pursue this collaboration since then, after I moved to INRIA Paris-Rocquencourt as a Junior Researcher in the Abstraction Project-team (at École Normale Supérieure) and Evan moved to the University of Colorado at Boulder as an Assistant Professor. As part of this collaboration, Evan had the opportunity to visit ENS and I had the chance to visit University of Colorado.

Moreover, several students took part to the project. In particular, two of them contributed to some parts of the works presented in this manuscript:

- Vincent Laviron did do his MPRI Master Internship on the analysis of low level programs (chapter 4);
- Suzanne Renard (from École des Mines de Paris) did do a Master Internship on the analysis of shared structures (section 6.2.4).

Furthermore, at the time of the writing of this manuscript, Antoine Toubhans just started working on this topic as part of his MPRI Master Internship.

Research directions: My main long term goals is to improve the design of abstract domains for programs manipulating complex data structures, so as to integrate such an abstraction into a static analyzer like Astrée [11], for verifying safety, structural invariant preservation and functional correctness properties on wide families of programs, using *hybrid* invariants, that is, properties about non only the data structures but also about their content.

As a consequence, the abstraction for data structures should be combined with other numerical and symbolic abstract domains. Thus, the analysis operations for the shape domain should integrate well into the structure of an abstract interpretation based static analyzer. A first direction I have been pursuing consists in setting up modular algorithms where memory abstractions can be combined and exchange information with numerical abstractions.

Moreover, it should be general enough to apply to several kinds of data structures, and not only one kind of structures. Indeed, many shape analyses are specialized to one kind of data structures, and would need be reworked deeply in order to handle other kinds of structures. In particular, I think that it would be ideal to integrate specific analyses for buffers, strings and arrays, when such structures are used. As the memory abstraction is supposed to manage the splitting of memory states into regions and the abstractions of those regions, it sounds reasonable to expect the memory abstract domain be open on specific domains for sub-regions. Therefore, I have been searching for parameterized abstractions for memory states, and paid special attention to their use in other contexts (programming languages, data structures, properties) than the one I intend them for.

Last, I think that the implementation of such a domain is one of the most important goals of such a work. Therefore, I devoted a significant part of my time implementing memory abstract domains. So far, this implementation work mainly consisted in the design of Xisa (eXtensible Inductive Shape Analyzer), a prototype tool for the analysis of C programs, which is parameterized by user supplied inductive definitions, and by the choice of a numerical domain. As part of this implementation work, I

improved my understanding of the data structures and algorithms required for the analysis, and I am currently starting the implementation of a new version of the abstract domain, which should be more independent from the language, and its parameter domains.

1.4 Outline

This manuscript introduces our parametric abstract domain at a high level. While we provide formal definitions, we refer the reader to the published articles for details.

Chapter 2 and chapter 3 set up the foundations of our abstract domain, and respectively describe the abstraction and the main transfer functions for the analysis of programs.

Then, the following chapters present extensions and applications of our domain. Chapter 4 shows how it can be adapted to analyze C programs and take into account low level pointer manipulations, that are commonly used in embedded codes or device drivers. Chapter 5 describes an interprocedural analysis which relies on our abstract domain, and allows to treat procedures in a very context sensitive manner.

Last, chapter 6 summarizes and assesses the most salient choices made in the design of this abstract domain and presents perspectives for the extension and the implementation of this abstract domain as a standalone abstract domain.

Chapter 2

Abstraction of memory states

In this chapter, we describe a parametric abstract domain for the static analysis of programs manipulating complex data structures. First, we discuss in section 2.1 the general principles for the abstraction of properties about complex data structures and overview the features of our abstract domain. Then, we formalize the basis of this domain in section 2.2, that is, the definition of its abstract element and of its concretization. Last, we demonstrate how this domain can be parameterized, and we show how it can be used in order to express non trivial hybrid properties in section 2.3.

2.1 Principles for the abstraction of complex data structures

2.1.1 Complex data structures

Before we discuss the foundation of our abstraction, we review a few common data structures, and the difficulties that should be solved in order to perform static analysis on programs using them.

Structures and arrays: Aggregates (including structures and arrays) allow to store data in a *contiguous* region, the size of which is defined either when its data type is defined or when declaring a variable. The fact the size of these structures may not change after the declaration is a limitation for programmers, but it simplifies the static analysis, as it is possible to use "flat abstractions" including *field sensitive* approaches which distinguish all fields and *smashing* [11] approaches which abstract the values of all fields with one piece of abstract information [11]. Composite data types such as structures have a well-defined set of fields and resolving an update to a field is usually not hard, even when a smashing abstraction is used and weak updates should be performed. In the other hand, arrays offer a lot of opportunities for index arithmetics, which can make program analysis challenging.

Linked dynamic structures: In many situations, using a fixed size structure is not an option, as the amount of data a program may manipulate is not known at compile time or even at the point where structures are declared. *Dynamic memory allocation* allows to acquire or release memory cells when needed, which can be appended to linked structures (lists, trees...). In the static analysis point of view, this situation brings up several challenges:

- the memory space used by a program is *unbounded*, so that unbounded size regions should be abstracted in a compact manner;
- non trivial topological properties of the program memory space also have to be tracked; for instance, a failure to maintain *reachability* of all allocated cells may translate into a memory leak, damaging the whole system safety, whereas a failure to track *pointer* links will prevent the verification of pointer dereferences and memory accesses.

Such structures are much harder to analyze precisely. *Pointer* analyses [22, 43] cannot capture precise invariants on such structures. *Shape* analyses typically use shape graphs [110] or powerful families of formulas [111, 44].

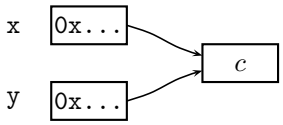
Relational linked dynamic structures: In practice, the implementation of efficient algorithms requires a very careful use of dynamic structures. For instance, maintaining additional pointers typically enables faster traversal algorithms as in *doubly-linked lists* or *skip lists*. Balanced trees (like *red black trees* or *AVL trees*) ensure better amortized complexity at the expense of additional fields at each node, the values of which are tightly related to the topology of the trees. Relations among pointers or between pointers and numeric fields bring an

additional challenge to static analysis, as these relations also need be tracked despite abstraction [20]. Structures with a lot of sharing (such as graphs or directed acyclic graphs) involve even *global* relations.

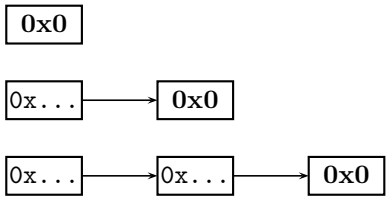
2.1.2 Static analysis related issues

To perform static analysis of programs manipulating a wide spectrum of data structures, several important issues need to be overcome:

- **Expressive abstractions** have to be found so as to describe memory states containing arrays and linked structures, with relations among pointers and numeric values.
- **Destructive updates** should be analyzed precisely [111, 44]; for instance, in the situation below, both `x` and `y` point to the same cell and if an assignment of the form $\star$`x` $= \ldots$ is performed, then the modification can be observed from `y`:



- **Termination** of static analyses should be ensured, although we have seen programs may generate unbounded structures; for instance, a function allocating a list of length passed as argument may generate lists of unbounded length, thus enforcing the termination of the abstract computation is not trivial; the first iterations would produce the results below and termination should be enforced using some sort of *widening operator* [29]:



2.1.3 Foundations of an abstraction for data structures

In this manuscript, we intend to propose a generic abstraction for a wide range of memory structures. Of course, this abstraction may not be able to capture all possible data structures, since many very different kinds of structures can be found in real programs. Instead, we aim at expressing a large set of common data structures, and attempt to do so in a rather *generic* way, that is without overspecializing our abstract domain.

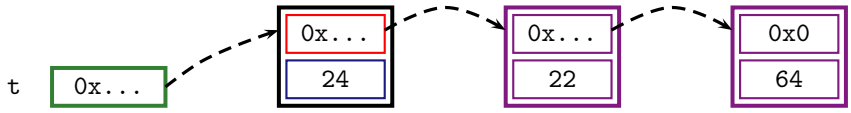
At this stage, we can introduce the main principles that we plan to exploit, and overview the main constructions of our shape abstract domain.

This domain relies on a *graph* representation where:

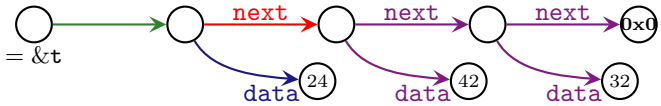
- *nodes* denote *values* stored in the memory or used in the analysis;
- *edges* capture information about *memory regions*.

In the following we illustrate this concept with a very simple example shown in figure 2.1: figure 2.1(a) shows an excerpt of a concrete memory state, with only one variable (t) and a heap allocated singly-linked list with three elements, each of them storing an integer value, whereas figure 2.1(b) and figure 2.1(c) depict two abstract versions (respectively, without and with summarization of memory regions).

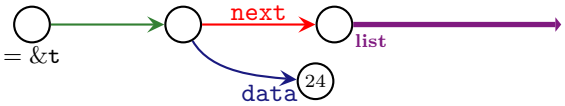
Disjointness of regions: First of all, we make the convention that the regions denoted by any pair of edges are *always disjoint*, which makes *local reasoning* [60] possible, following the principles of *separation logic* [97]. Several formalisms allow to reason on regions disjointness. Separation logic is based on the very clear convention that memory regions paired with separating conjunction connector $*$ are always separate. Other logics, such as region logics [5] accept more flexible statements, where region sharing, partial sharing and disjointness can be expressed. Typically, regions logic requires additional work to express whenever two regions are separate whereas separation logic requires a special treatment for shared regions (using non separating conjunction). We opted in favor of separation logic, as we found that most of the time, static analyzers reason about disjoint regions while cases where non separating conjunctions is necessary are rare and local (section 4.3). As a consequence, whenever a program statement may modify the value of a memory cell, the analysis should determine which fragment of the graph may be modified. When that fragment can be narrowed down to one edge, it follows that the rest of the graph will *not* be altered; in other words, the modification is *local*. More generally, when a graph G comprises the edges of two sub-graphs G_0 and G_1, G actually denotes the sep-



(a) Concrete memory state



(b) Towards abstraction: graph representation without summarization



(c) Towards abstraction: graph representation, with summarization

Figure 2.1: Memory abstraction

arating conjunction of the denotations (in the sense of the $*$ operator of separation logic) of G_0 and G_1. In general reasoning on destructive updates is easier when relying on separation [44, 21, 77]. In turn, this poses severe constraints on the analysis, as it should always maintain the invariant that distinct edges denote *disjoint* memory regions at all times, and avoid losing precision.

For instance, the concrete memory state of figure 2.1(a) is divided into several memory regions, corresponding to colors (green, blue, red and purple). The graph representations shown in figure 2.1(b) and figure 2.1(c) over-approximate that concrete memory state in a per-region manner: each edge stands for an abstraction of a region of the concrete state. For instance, the green edges in both graphs stand for an abstraction of the green area of the concrete state, and the same for the other colors. Local reasoning means that any operation that would impact the red region in the concrete would only need be analyzed with respect to the red edges in the graphs.

Abstraction of contiguous regions: Contiguous regions, such as structures or arrays can be abstracted precisely, and form the atomic elements of our abstract domain; these can be described by sets of basic *points-to* edges. Roughly speaking, a graph which consists in a single points-to edge $\alpha \overset{s}{\mapsto} \beta$ describes stores which are made of just one memory cell:

- of size s,
- the address of which is denoted by α,
- and which contains the value denoted by β.

This representation can be used to describe any contiguous memory region, since β may abstract any sequence of bits and s may take any value. However, when a structure data type has several fields, it is usually better to represent each field separately by a points-to edge. Indeed, blurring them together would capture the same information, yet would make it harder to analyze an assignment to one of the fields.

All edges of the graph of figure 2.1(b) are points-to edges, which means each of them corresponds to a contiguous region in the concrete. It would be possible not to split an element of the list into a pair of fields and to use only one points-to edge for that element (i.e., that edge would describe both fields), yet that would not be very effective, since it would mean a sequence of bits containing a pointer and an integer value would be abstracted by *one* node, which would not be very convenient (e.g., a pointer dereference could not be analyzed at this level of abstraction).

Summarization: Another kind of edges represents unbounded memory regions, that consist in regular patterns, and that can be described by *inductive definitions*. We could have chosen more gen-

eral sets of definitions (as discussed in section 6.2.5), but inductive definitions turn out to capture a large and interesting set of data structures including simple linked structures such as lists and trees, but also linked structures with relations among pointers, and between pointers and values, such as skip lists or balanced trees. Besides, inductive definitions offer good opportunities for efficient algorithms, as will be shown in chapter 3: indeed, abstract interpretation based static analysis is precisely about inferring inductive invariants, thus it can be expected to work pretty well on properties defined by induction. A rather hard issue is to chose proper inductive definitions. Depending on the context, we shall assume they are either supplied by the user (section 2.3.2), or derived by the static analyzer itself (section 5.3.2 and section 6.2.1).

In figure 2.1, we consider a concrete singly-linked list data structure; the data type of singly-linked lists can be defined by induction, thus it makes sense to rely on an inductive definition for those (we do not formalize it here, as this will be done in the next section). Figure 2.1(c) shows the result of summarizing the purple region using a more abstract information, that it contains a complete list structure. During that step, the information that the list in the purple region has two elements is lost, and the same for the concrete values stored in those list elements, yet the graph of figure 2.1(c) does convey the property that the list pointed to by `t` has at least one element, since the first element is not summarized. If the red region and the blue region were joined together with the purple region, abstracting that region into one list predicate would lose even more information as it would not rule out the possibility that `t` be the null pointer (empty list, stored in an empty region).

Combination of abstract domains: Since one of our goals is to express composite invariants, both about the shape of memory structures and about their contents, we also need to get numerical information to fit into our abstractions. Instead of embedding such information in the graphs, we propose to rely on existing value abstract domains and to build a composite abstraction. In practice, a numerical abstract domain element will be used in order to express numerical constraints about the values represented by graph nodes. This way, we can take advantage of existing, efficient static analysis algorithms for values while relying on our graph representation for manipulating shape properties.

For instance, the numerical values stored in the `data` fields of the concrete store of figure 2.1(a) are still shown in the nodes of the graphs in the two abstract versions, yet these equalities would actually be stored in a numerical abstract domain. Furthermore, since the concrete list is also sorted, we may wish not to abstract that information away while summarizing the purple region. To do that, we would need to replace the **list** inductive predicate with a more expressive one, also capturing sortedness.

2.2 A shape abstract domain

We now formalize the abstraction of the abstract domain which was sketched in section 2.1.3.

2.2.1 Concrete states and notations

In the following, we let $\mathbb{X}$ denote a fixed set of *variables* and $\mathbb{V}$ denote a set of *values*. Values include machine integers ($\mathbb{V}_{\text{int}}$) and addresses ($\mathbb{V}_{\text{addr}}$); thus, $\mathbb{V}_{\text{int}} \cup \mathbb{V}_{\text{addr}} \subseteq \mathbb{V}$ ($\mathbb{V}_{\text{int}}$ and $\mathbb{V}_{\text{addr}}$ may be disjoint or not, depending on the target language). Intuitively, a *store* (or *memory state*) σ is a *partial* function which maps addresses into values ($\mathbf{read}(\sigma, x) = v$ or, for short $\sigma(x) = v$, means that value v is stored at address x). However, this intuition does not account for the fact a memory read should also take a *size* s (or, more generally a *type* t) as a parameter, and read exactly s bytes (or $\mathbf{sizeof}(t)$ bytes). In this view, $\mathbf{read}(\sigma, x, 4)$ should be consistent with $\mathbf{read}(\sigma, x, 2)$ and with *endianness* assumptions (and the same for the semantics of the **write** operation which writes a value into a memory cell). We will not make these constraints explicit, and simply let concrete memory states be defined following this model. We let $\mathbb{M}$ denote the set of memory states, and assume the following primitives are defined over $\mathbb{M}$:

- $\mathbf{read} : \mathbb{M} \times \mathbb{V}_{\text{addr}} \times \mathbb{N} \longrightarrow \mathbb{V}$;
- $\mathbf{write} : \mathbb{M} \times \mathbb{V}_{\text{addr}} \times \mathbb{N} \times \mathbb{V} \longrightarrow \mathbb{M}$;
- $\mathbf{dom} : \mathbb{M} \longrightarrow \mathcal{P}(\mathbb{V}_{\text{addr}})$ returns the set of addresses at which at least one byte can be read.

Furthermore, we assume those functions satisfy the usual properties ([97]), such as $\mathbf{read}(\mathbf{write}(\sigma, x, s, v), x, s) = v$.

An *environment* $\hat{\epsilon} \in \mathbb{E}$ is a partial function from variable names into addresses. A *state* is a tuple

made of an environment $\hat{\epsilon} \in \mathbb{E}$ and a memory state $\sigma \in \mathbb{M}$. We let $\mathbb{S} = \mathbb{E} \times \mathbb{M}$ denote the set of states. For instance, in figure 2.1(a), $\hat{\epsilon}$ contains only variable `t`.

2.2.2 Decomposition of the abstraction

Our abstract domain aims at over-approximating sets of states: an abstract value should describe a set of states. Before we formalize this abstraction, we discuss the structure of abstract elements and of the concretization relation.

Structure of the abstract domain: As a concrete state comprises two components (an environment and a memory state), the abstraction of a set of states follows a similar structure. The shape information will be expressed in graphs. Abstract values should also include an abstraction for the environments; this abstraction should bind variable names to abstractions of their addresses. As mentioned in section 2.1.3, nodes abstract values (whereas edges abstract memory regions). We let $\mathbb{V}^\sharp$ denote a set of *node symbolic names*, which will usually be denoted by Greek letters $\alpha, \beta, \gamma, \delta \ldots$. An abstract environment $\hat{\epsilon}^\sharp : \mathbb{X} \longrightarrow \mathbb{V}^\sharp$ maps each variable name into a node that denotes its address.

The relation between a concrete state and an abstract value cannot be described directly, as we need to specify how the nodes and edges of the abstract values describe components of the concrete value. This relation between nodes in the abstract element and concrete values will be described by a *valuation*, that is a function $\nu \in \mathbb{V}\text{al} = \mathbb{V}^\sharp \longrightarrow \mathbb{V}$ from nodes into values. A concrete memory state will only be related to an abstract element of the shape domain $\mathbb{D}^\sharp_{\mathrm{S}}$ *together* with some such valuation ν. In the other hand, the correspondence between memory regions and edges in the graphs will appear in the concretization.

Moreover, numerical relations that cannot be expressed inside the shape abstract domain may be expressed in some other abstract domain. For instance, the numerical relation $\alpha = \beta + 2$ that binds symbolic names α and β (for example, it may denote a constraint on pointer values, arising from pointer arithmetics) may be described using linear equalities [66]. In the other hand, it cannot be expressed as memory structure information, as this relation is purely independent from the fact α and β correspond to actual values in the memory state. A very wide range of numerical abstract domains can be used to capture such constraints, as will be shown in section 2.3.1. Thus, an element of a numerical abstract domain $\mathbb{D}^\sharp_{\mathrm{N}}$ will be used in order to express purely numerical constraints among graph nodes. It will constrain the valuations, that can be considered when expressing the concretization.

Last, precise static analysis often requires disjunctions of abstract values to give more precise results, thus it will use a disjunction domain such as trace partitioning [83].

Formalization of the domain structure: We now summarize the structure of abstract domains. The table below summarizes their definitions:

$\mathbb{D}^\sharp_{\mathrm{S}}$			abstract graphs
$\mathbb{D}^\sharp_{\mathrm{N}}$			numerical constraints
$\mathbb{M}^\sharp$	$=$	$\mathbb{D}^\sharp_{\mathrm{S}} \times \mathbb{D}^\sharp_{\mathrm{N}}$	store abstraction
$\mathbb{E}^\sharp$	$=$	$\mathbb{X} \longrightarrow \mathbb{V}^\sharp$	abstract environments
$\mathbb{S}^\sharp$	$=$	$\mathbb{E}^\sharp \times \mathbb{M}^\sharp$	abstract states

The concretization functions of these domains are defined as follows:

- **abstract graphs** concretize into sets of pairs made of a memory state and a valuation:

$$\gamma_{\mathrm{S}} : \mathbb{D}^\sharp_{\mathrm{S}} \longrightarrow \mathcal{P}(\mathbb{M} \times \mathbb{V}\text{al})$$

- **abstract numerical values** concretize into sets of valuations:

$$\gamma_{\mathrm{N}} : \mathbb{D}^\sharp_{\mathrm{N}} \longrightarrow \mathcal{P}(\mathbb{V}\text{al})$$

- elements of the **product abstraction** concretize into the meet of the concretizations:

$$\begin{array}{r} \gamma_{\mathbb{M}} : \mathbb{M}^\sharp \longrightarrow \mathcal{P}(\mathbb{M} \times (\mathbb{V}\text{al})) \\ \forall (S, N) \in \mathbb{M}^\sharp, \\ \gamma_{\mathbb{M}}(S, N) = \{(\sigma, \nu) \mid \nu \in \gamma_{\mathrm{N}}(N) \\ \wedge\ (\sigma, \nu) \in \gamma_{\mathrm{S}}(S)\} \end{array}$$

- **abstract states** concretize into sets of concrete states, such that there exists a valuation, which is coherent with the abstract environment and the abstract graph:

$$\begin{array}{r} \gamma_{\mathbb{S}} : \mathbb{S}^\sharp \longrightarrow \mathcal{P}(\mathbb{S}) \\ \forall (\hat{\epsilon}^\sharp, (S, N)) \in \mathbb{S}^\sharp, \\ \gamma_{\mathbb{S}}(\hat{\epsilon}^\sharp, (S, N)) = \{(\hat{\epsilon}, \sigma) \mid \exists \nu \in \gamma_{\mathrm{N}}(N), \\ (\sigma, \nu) \in \gamma_{\mathbb{M}}(S) \\ \wedge\ \forall x \in \mathbb{X},\ \hat{\epsilon}(x) = \nu(\hat{\epsilon}^\sharp(x))\} \end{array}$$

In the following subsections, we will define the shape domain itself and its concretization.

2.2.3 Abstraction without summarization

We first consider abstract values that do not include information about *summarized* regions. These will be discussed in the next section.

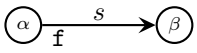
Points-to edges concretization: An abstract value of $\mathbb{D}^\sharp_{\mathrm{S}}$ consists in a graph that is a set of nodes (which denote concrete values) and a set of edges (which denote constraints over memory regions). In particular, we have seen in section 2.1.3 that a *points-to edge* denotes a memory cell or a set of contiguous memory cells.

When considering a structure type, it is often useful to decompose its contents into the contents of its fields; thus, it should rather be represented by a collection of points-to edges corresponding to each field. The addresses of these cells are correlated, since they all correspond to the sum of the *base address* of the structure and of an *offset*. Thus, it makes more sense to avoid using several distinct nodes in order to represent the base address of each of these cells. Instead, we add the offset as a label on the edge itself.

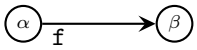
Thus, the general format of a points-to edge is:

$$\alpha \cdot \mathtt{f} \overset{s}{\mapsto} \beta$$

We use the graphical representation below to denote such an edge in a graph:



When the size does not need be tracked explicitly, we will always use the simplified version $\alpha \cdot \mathtt{f} \mapsto \beta$ (`f` will also be dropped when depicting a cell at base address α, with no offset, i.e. that is not part of an aggregate):



The denotation $\gamma_{\mathrm{S}}(\alpha \cdot \mathtt{f} \overset{s}{\mapsto} \beta)$ of this edge is the set of pairs (σ, ν) such that:

- memory state σ is a memory state with *only one cell* of address $x = \nu(\alpha) + \mathbf{offset}(\mathtt{f})$ (where $\mathbf{offset}(\mathtt{f})$ denotes the numerical offset of field `f`), and of size s, which boils down to:

$$\mathbf{dom}(\sigma) = \{x, x+1, x+2, \ldots, x+s-1\}$$

- the content of this cell is $v = \nu(\beta)$:

$$\mathbf{read}(\sigma, x, s) = v = \nu(\beta)$$

For instance, a next field of a list element could be described by $\alpha \cdot \mathtt{next} \overset{4}{\mapsto} \beta$, assuming a 32-bits architecture (i.e., that pointers are four bytes long).

Separation concretization: As mentioned in section 2.1.3, we rely on separating conjunction in order to make *local reasoning* possible.

In the concrete level, we let the *separating conjunction* binary operator $\circledast$ be defined as follows: if $\mathbf{dom}(\sigma_0) \cap \mathbf{dom}(\sigma_1) = \emptyset$, then $\sigma_0 \circledast \sigma_1$ is the concrete store defined by

$$\mathbf{dom}(\sigma_0 \circledast \sigma_1) = \mathbf{dom}(\sigma_0) \cup \mathbf{dom}(\sigma_1)$$
$$\text{if } \mathbf{read}(\sigma_i, x, s) = v \text{ then } \mathbf{read}(\sigma_0 \circledast \sigma_1, x, s) = v$$

Then, the concretization of a graph $S \in \mathbb{D}^\sharp_{\mathrm{S}}$ such that S is made of the set of edges $\{e_0, \ldots, e_n\}$, then

$$\gamma_{\mathrm{S}}(S) \quad = \quad \{(\sigma_0 \circledast \sigma_1 \circledast \ldots \circledast \sigma_n, \nu) \mid \forall i,\ (\sigma_i, \nu) \in \gamma_{\mathrm{S}}(e_i)\}$$

The empty graph (i.e., abstract element with no edge) **emp** is the neutral element for $\circledast$.

For the sake of concision, we do not introduce explicitly a "$\top$" abstract value, standing for an undetermined region (equivalent to the **true**) separation logic value. Instead, we will simply assume nothing is known about heap regions not described in the graphs.

2.2.4 Summarization with inductive definitions

In section 2.1.3, we observed that inductive structures offer a good basis for summarizing complex and unbounded memory regions using compact formulas. We now define a language of inductive definitions and show how they can be relied on in order to describe memory states.

A language of inductive definitions: Many data structures used by programmers follow a *recursive pattern*. For instance, a list pointer is either a null pointer or a non null pointer, which points to a structure with several fields including a "next" field, which contains another list pointer. We observe there are two cases (the null pointer and the non null pointer), thus an inductive definition should consist in a conjunction of cases (or *rules*). Each rule describes a memory region (using contiguous regions or other instances of the inductive

structure, or both) and numerical predicates (e.g., that a pointer is not equal to null). Furthermore, in the case of a singly-linked list, elements should be pairwise distinct, which can be expressed with the separating conjunction operator.

We observe that in this description the *argument* of the list inductive predicate is the address of its first element. This is very convenient as nodes stand for values, i.e., including addresses; that way, we do not have to explicitly relate the "inductive list" predicate to the region it describes. In the following, we will always assume the existence of such a main argument or parameter. Yet, in some cases, we may need to add *additional* (or, *auxiliary*) parameters. For instance, in the case of a doubly-linked list, we also need to propagate information about the address the "prev" field should point to (section 2.3.2).

At this point, we can formalize our language of inductive definitions. The definition of inductive ι has the form $\alpha \cdot \iota(\beta_0, \ldots, \beta_{n-1}) ::= r_0 \vee r_1 \vee \ldots \vee r_k$, where α is the main parameter, $\beta_0, \ldots, \beta_{n-1}$ are the auxiliary parameters and $r_0, r_1, \ldots, r_k$ are the rules. Each rule consists in the conjunction of a shape formula and a pure formula (or numerical formula), where the shape part is the separating conjunction of a set of terms, which are either points-to edges or "recursive calls", that is application of an inductive definition to a set of parameters. These definitions are summarized in the grammar below:

$$
\begin{array}{rcll}
r & ::= & \text{inductive rule} & \\
 & | & (F_{\text{Shape}}, F_{\text{Num}}) & \\
F_{\text{Shape}} & ::= & \text{shape formula} & \\
 & | & F_{\text{Shape}} * F_{\text{Shape}} & \\
 & | & \mathbf{emp} \quad \text{(empty region)} & \\
 & | & \alpha \cdot \mathtt{f} \overset{s}{\mapsto} \beta & \\
 & | & \alpha \cdot \iota(\beta_0, \ldots, \beta_{n-1}) & \\
F_{\text{Num}} & ::= & \text{pure formula} & \\
 & | & \alpha = c & (\text{where } c \in \mathbb{V}) \\
 & | & \alpha \neq c & (\text{where } c \in \mathbb{V}) \\
 & | & \ldots & \\
\alpha, \beta, \gamma, \delta & \in & \mathbb{V}^\sharp & \\
\iota & : & \text{inductive name} & \\
\mathtt{f}, \mathtt{f}' & : & \text{structure fields} &
\end{array}
$$

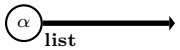
For instance, the **list** inductive definition can be written as follows:

$$
\begin{array}{rl}
\alpha \cdot \mathbf{list} ::= & \\
 & (\mathbf{emp}, \alpha = 0) \\
\vee & (\alpha \cdot \mathtt{next} \mapsto \beta * \alpha \cdot \mathtt{data} \mapsto \gamma \\
 & \quad * \beta \cdot \mathbf{list}, \alpha \neq 0)
\end{array}
$$

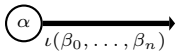
The first rule corresponds to the empty list case (it describes an empty memory region and a null pointer). The second rule corresponds to the case of a non empty list, where α is a non-null pointer, which points to a structure with two fields called `next` and `data`; β and γ denote the respective values of these two fields, and β points to another, disjoint singly-linked list. Observe that the two fields of the structure also refer to disjoint memory regions, as shows the use of separating conjunction.

There are other ways to look at such inductive definitions: we can consider them *constructors*, that assemble non trivial structures in a recursive manner, from basic elements, but we can also view them as *data structures checkers* [21]. Indeed, these definitions are rather close to checking functions, which can be used in order to check that a memory structure satisfies some global invariants. Such functions are often used by programmers in order to check that the data structures they implement satisfy structural invariants; in debug mode, it is common practice to call such checking functions before and after performing destructive updates to the structures, so as to spot where the invariant is first violated, when testing a program does not seem to behave as intended. Though, there exists an noticeable difference with such function, as our inductive definitions rely on separating logic, which is not usually done when implementing checking functions (this would be extremely expensive to check, as it would require to store sets of elements that were visited).

Inductive edges: We now need to exploit inductive definitions in order to abstract sets of memory regions in graphs. Let us consider the case of a singly-linked list inductive definition first. If α is the address of a singly-linked list, then we simply write that $\alpha \cdot \mathbf{list}$ holds, and we will use the graphical notation:



In general, if we consider inductive definition ι with n additional parameters, then we will write $\alpha \cdot \iota(\beta_0, \ldots, \beta_{n-1})$ to state that ι holds when applied to the main parameter α and to the auxiliary parameters $\beta_0, \ldots, \beta_n$; then, we will use the graphical notation:



Unfolding inductive definitions: Let us consider property $\alpha \cdot \iota$, where ι is the disjunction of rules $r_0, \ldots, r_n$. Intuitively, it means that the current state satisfies the property expressed by one of these rules. Let us assume the property expressed by rule r_i holds. Therefore, we can rewrite property $\alpha \cdot \iota$ using the definition of rule r_i. However, the shape part and the pure part of rule r_i rely on the formal parameters of ι, that is those used in its definition. This is no major difficulty, as standard substitution of actual parameters to formal parameters allows to rewrite r_i into a pair of formulas that contain only the variables of the original edge (i.e., α), and possibly, some *fresh* nodes created to represent new values, of fields that get exposed during this step. This operation is called *syntactic unfolding a rule*. Moreover, when it results in the pair of formulas $(F_{\text{shape}}, F_{\text{num}})$, we write:

$$\alpha \cdot \iota \xrightarrow{\mathcal{U}}_{\iota, r_i} (F_{\text{shape}}, F_{\text{num}})$$

Furthermore, when there exists a rule r_i in ι such that the above statement holds, we also write:

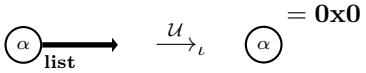
$$\alpha \cdot \iota \xrightarrow{\mathcal{U}}_{\iota} (F_{\text{shape}}, F_{\text{num}})$$

This defines the *syntactic unfolding of an inductive definition*.

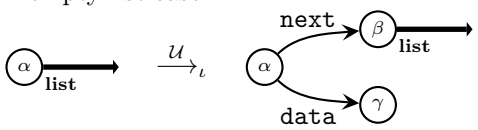
This unfolding operation is not exactly an internal domain operator: the shape part of the result is a valid graph (i.e., shape domain abstract element), however the pure part is an arbitrary logical formula, which $\mathbb{D}^\sharp_{\text{N}}$ may not be able to represent exactly. We will address this issue when designing the unfolding domain operation in section 3.3 (this operation will actually be largely based upon the syntactic unfolding).

Syntactic unfolding can also be applied to graphical representations, in a straightforward manner. For instance, the two pictures below describe both possible syntactic unfoldings of a **list** inductive edge:

- empty list case:

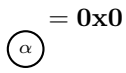


- non-empty list case:

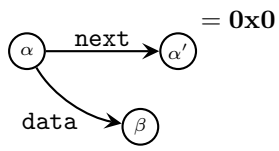


Concretization of inductive edges: The syntactic unfolding operation defined previously gives the intuitive meaning of inductive predicates; therefore, it is actually the foundation of the concretization of such predicates. This concretization is defined naturally by induction over the inductive predicates. Intuitively, if $\alpha \cdot \iota \xrightarrow{\mathcal{U}} (S, N)$, $(\sigma, \nu) \in \gamma(S)$, and ν satisfies pure formula N (which we write $\nu \vdash N$), then $(\sigma, \nu) \in \gamma_{\text{S}}(\alpha \cdot \iota)$. Unsurprisingly, this definition of the concretization corresponds to the least fixpoint of a continuous operator over sets of pairs (σ, ν). For instance, in the case of list, we obtain the following cases:

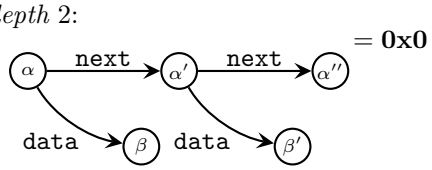
- at *depth* 0:



- at *depth* 1:



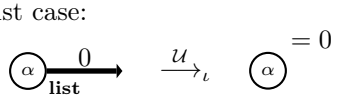
- at *depth* 2:



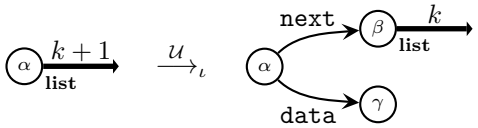
Reasoning on the length of data structures: It is sometimes useful to reason over the length of a list, or the depth of a tree. This length information can be made explicit on our definitions. In this paragraph, we label inductive edges with an integer standing for the induction depth and we will write $\alpha \cdot \iota^k$ for inductive edge $\alpha \cdot \iota$, with induction depth k (where $k \geq 0$). The syntactic unfolding of such an edge should generate a set of inductive predicates such that the maximal element of their depths is $k - 1$. Similarly, we may label edges with an *upper bound* on the length of derivation.

For instance, if we consider the **list** definition, we get the following syntactic unfolding configurations (length labels are placed on top of edges), using exact length indices (i.e., not upper bounds):

- empty list case:



- non-empty list case:



We can prove the following property:

$$\gamma_S(\alpha \cdot \iota) = \bigcup_{k \in \mathbb{N}} \gamma_S(\alpha \cdot \iota^k)$$

This provides an alternate definition of the concretization of any inductive predicate, since it is possible to define $\gamma_S(\alpha \cdot \iota^{k+1})$ from $\gamma_S(\alpha \cdot \iota^k)$, using the unfolding rule.

2.2.5 Segment edges

Let us consider the list definition in more details. In practice, it is common to describe a list structure l, such that several pointers point to some elements in l. For instance, this situation arises when implementing a (very classical) list traversal algorithm: indeed, in this case, there is a pointer to the first element of the list, and a "cursor", which points to the current element. In this case, it is usually more convenient *not* to summarize the whole list with one predicate of the form $\alpha \cdot \mathbf{list}$, and to use *two* predicates, describing respectively the part of the list that has already been visited and the part of the list that is left to be visited. Thus, we should also describe *list segments*.

We can observe that a list segment can also be described using an inductive definition, using an additional parameter to denote the "end-point":

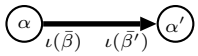
$$\begin{array}{ll} & \alpha \cdot \mathbf{list\_endp}(\pi) ::= \\ & \quad (\mathbf{emp}, \alpha = \pi) \\ \vee & \quad (\alpha \cdot \texttt{next} \mapsto \beta * \alpha \cdot \texttt{data} \mapsto \gamma \\ & \qquad * \beta \cdot \mathbf{list\_endp}(\pi), \alpha \neq 0) \end{array}$$

We observe that this definition is actually quite similar to that of a regular singly-linked list, except for the following two small differences:

- the inductive predicate now expects one additional parameter π, which should denote the end-point of the list segment;
- in the base case, the value constraint on pointer α requires it be equal to end-point π.

In fact, this definition is not only close to that of a regular list segment, but it can also be derived from **list** in a systematic way, and it could generalize to other structures as well. Indeed, **list_endp** describes a list with a "hole", that is a list where a sub-list is missing. The same could be done just as easily with a tree structure.

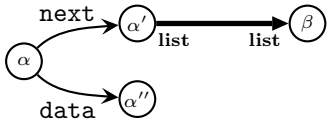
Therefore, we opt to include the notion of segment once and for all in our abstract values. In this purpose, we add another kind of edges, which we call *segment edges*; a segment edge is of the form $\alpha \cdot \iota(\bar{\beta}) \mathrel{*\!\!=\!\!=} \alpha' \cdot \iota(\bar{\beta}')$, and we use the graphical representation below:



In particular, we observe the following two graphs convey the same information, i.e., there exists a list segment between node α and node β:

Concretization: The essence of segments is that they unfold slightly differently compared to regular inductive edges. Let us consider segment $\alpha \cdot \iota(\bar{\beta}) \mathrel{*\!\!=\!\!=} \alpha' \cdot \iota(\bar{\beta}')$, and discuss how it can be unfolded:

- an *empty segment* (or segment of length 0) stands for an *empty region* and is such that both ends are equal, i.e., $\alpha = \alpha'$ and $\bar{\beta} = \bar{\beta}'$;
- a *non empty segment* unfolds as an inductive edge, except that *exactly one* recursive predicate of the unfolding result is a segment; for instance, in the case of lists, unfolding a non empty list segment results in:



Using this definition of unfolding, the concretization of a segment edge is defined in a very similar way as that of an inductive edge (section 2.2.4): if $\alpha \cdot \iota(\bar{\beta}) \mathrel{*\!\!=\!\!=} \alpha' \cdot \iota(\bar{\beta}') \xrightarrow{u} (S, N)$, $(\sigma, \nu) \in \gamma_S(S)$ and $\nu \vdash N$, then $(\sigma, \nu) \in \gamma_S(\alpha \cdot \iota(\bar{\beta}) \mathrel{*\!\!=\!\!=} \alpha' \cdot \iota(\bar{\beta}'))$.

We can observe that this notion of segment is "stronger" than the *separating implication* (also called "*magic wand*"):

$$\gamma_S(\alpha \cdot \iota \mathrel{*\!\!=\!\!=} \alpha' \cdot \iota) \subseteq \gamma_S(\alpha \cdot \iota \mathrel{*\!\!-} \alpha' \cdot \iota)$$

This property can be proved by induction, using the concretization relation. It means that magic wand also models segments, and we successfully used it for static analysis [21], yet it also describe stores that are *not* segments, which makes it impossible to prove certain strong properties and we found that using inductively defined segments turns out more powerful [20].

Generalizations: We can add explicit length information on segments, in the same way as we did for regular inductive predicates in section 2.2.4. Such segment indices represent the number of unfolding steps from the origin to the destination of the segment. Such a predicate is denoted $\alpha \cdot \iota \overset{k}{\ast\!\!=} \alpha' \cdot \iota$.

It is also feasible to set up a notion of segments of arity equal greater than 2, or multi-segments of arbitrary arity [56]. Such edges would come as a straightforward generalization of segments, where there are k "holes" instead of 1. A list structure may have only one such hole, so multi segments are not relevant in that case, however, in the case of trees, a 2-segment would correspond to a tree fragment that would be a complete tree if two sub-trees were added [49]. However, we found that standard segments (i.e., segments of arity 1) have a reasonably high level of expressivity, and that segments of higher arity are more cumbersome to manipulate in automatic static analysis, thus we did not integrate them formally into our abstraction.

2.2.6 Shape domain

At this stage, we can summarize the definition of our shape domain. Our domain is parametric in the sense that it is *parameterized* by the data of a set of inductive definitions. Inductive definitions may either be supplied by the user (section 2.3.2), or derived by some automatic analysis itself (section 5.3.2 and section 6.2.1).

Abstract values are described by the grammar of figure 2.2(a), graphical representation of edges in figure 2.2(b) and the concretization rules are shown in figure 2.2(c).

2.3 Expressiveness and parameterization of the abstract domain

Our abstract domain can be parameterized in several ways:

- it relies on a numerical domain in order to express constraints on basic values (i.e., integer, floating point and boolean values);
- it can also be parameterized by the data of a finite set of inductive definitions, to be used for summarization.

In this section, we discuss difficulties inherent in this parameterization and show examples of structures that fall in the scope of the parametric abstract domain.

2.3.1 Combination with a numerical domain

The abstraction of concrete states described in rule ($\mathbb{S}$) appears like a flavor of a product abstraction [30], yet is slightly more complex.

An asymmetric product abstraction: Indeed, the product of the shape domain with the numerical domain is not symmetric, since the set of symbolic nodes used as variables in a numerical abstract value corresponds to the set of nodes that appear in the shape abstract value. This means that a valid abstract element (S, N) should be such that N belongs to the numerical abstract domain lattice defined by the data of the nodes in S. When computing transfer functions or joins, that set of nodes may change and the numerical invariants should be updated accordingly. Furthermore, this also imposes some consequences on widening algorithms, as will be shown in section 3.4. This construction is actually an instance of the cofibered abstract domain [119].

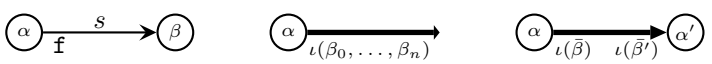
Unlike [76], we opted for a product analysis instead than a splitting into two separate analyses, as this usually proves a better choice in the precision point of view.

Choice of a numerical domain: In practice, any numerical abstract domain can be used, including intervals [29], octagons [89], linear equalities [66], convex polyhedra [39] and others. Combinations of domains (e.g., by reduced product [30]) can also be used. Our framework only requires the existence of a concretization function $\gamma_{\mathrm{N}} : \mathbb{D}^{\sharp}_{\mathrm{N}} \longrightarrow \mathcal{P}(\mathbb{V}\mathbb{a}\mathbb{l})$. However, we should note that the "variables" which are accounted for in that numerical domain are *not* program variables nor concrete memory cells but graph nodes, representing sets of concrete values.

The property that a symbolic node represents the null pointer can be expressed by the domains of constants, intervals... Addresses inequalities can be described with linear equalities or octagons, or

$$
\begin{array}{rcl}
S & ::= & \text{graph} \\
& | & e * \ldots * e \\
e & ::= & \text{edge} \\
& | & \alpha \cdot \mathtt{f} \overset{s}{\mapsto} \beta \\
& | & \alpha \cdot \iota(\bar{\beta}) \\
& | & \alpha \cdot \iota(\bar{\beta}) \mathrel{*\!\!=} \alpha' \cdot \iota(\bar{\beta}')
\end{array}
$$

(a) Domain definition



(b) Edges representation

$$
\frac{x = \nu(\alpha) + \mathbf{offset}(\mathtt{f}) \quad v = \nu(\beta) \quad \mathbf{dom}(\sigma) = \{x, \ldots, x+s-1\} \quad \mathbf{read}(\sigma, x, s) = v}{(\sigma, \nu) \in \gamma_{\mathrm{S}}(\alpha \cdot \mathtt{f} \overset{s}{\mapsto} \beta)} \mapsto
$$

$$
\frac{\forall i, j,\ i \neq j \Longrightarrow \mathbf{dom}(\sigma_i) \cap \mathbf{dom}(\sigma_j) = \emptyset \quad \forall i,\ (\sigma_i, \nu) \in \gamma_{\mathrm{S}}(e_i)}{(\sigma_0 \circledast \ldots \circledast \sigma_n, \nu) \in \gamma_{\mathrm{S}}(e_0 * \ldots * e_n)} *
$$

$$
\frac{e \text{ is an inductive or a segment} \qquad e \xrightarrow{\mathcal{U}} (S, N) \qquad (\sigma, \nu) \in \gamma_{\mathrm{S}}(S) \qquad \nu \vdash N}{(\sigma, \nu) \in \gamma_{\mathrm{S}}(e)} \mathcal{U}
$$

$$
\frac{\exists \nu \in \gamma_{\mathrm{N}}(N),\ (\sigma, \nu) \in \gamma_{\mathbb{M}}(S) \ \wedge\ \forall x \in \mathbb{X},\ \hat{\epsilon}(x) = \nu(\hat{\epsilon}^\sharp(x))}{(\hat{\epsilon}, \sigma) \in \gamma_{\mathrm{S}}(\hat{\epsilon}^\sharp, (S, N))} \mathbb{S}
$$

(c) Concretization rules

Figure 2.2: Shape abstract values

using a union find based abstraction of equivalence relations.

For implementation, using a library such as Apron [62] is the best solution, as it makes it possible to switch from one numerical abstract domain to another a trivial change in the analyzer code.

2.3.2 Recursive data structures

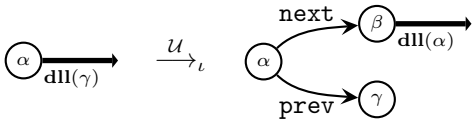
In section 2.2, we mentioned a singly-linked lists inductive definition, yet many other useful data structures have inductive definitions that can be used as parameters for our domain.

List structures: Circular lists are singly-linked lists such that the `next` field of the last element points to the first element; in other words, they can be defined using the list segment predicate, that is, either with an **list** segment edge or with the list with end-point inductive definition of section 2.2.5.

Doubly-linked lists are more complex, as each elements has *two* pointer fields, which are closely related to each other. Indeed, the `prev` field of the element pointed to by a `next` field should correspond to the current element. As observed in section 2.2.4, such a structure can be defined by induction, using an auxiliary parameter, so as to store where the `prev` field should point to. Then, we obtain the definition below:

$$
\begin{array}{rl}
\alpha \cdot \mathbf{dll}(\gamma) ::= & \\
& (\mathbf{emp}, \alpha = 0) \\
\vee & (\alpha \cdot \mathtt{next} \mapsto \beta * \alpha \cdot \mathtt{prev} \mapsto \gamma \\
& \quad * \beta \cdot \mathbf{dll}(\alpha), \alpha \neq 0)
\end{array}
$$

The second rule of this definition corresponds to the following graphical syntactic unfolding:

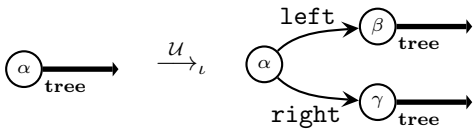


Other much more involved forms of lists can be described in our framework. In particular, we provided a two levels skip-list inductive definition in [21], where each element has a next field, pointing to the next element and a "skip" field pointing to the next "level 2" element in the list. In that case, an auxiliary parameter stands for the next level 2 element.

Tree structures: Inductive definition of binary trees is a straightforward extension to that of lists; the only difference is that there are now two recursive calls:

$$\begin{aligned}&\alpha \cdot \mathbf{tree} ::= \\ &\quad (\mathbf{emp}, \alpha = 0) \\ &\vee \quad (\alpha \cdot \texttt{left} \mapsto \beta * \alpha \cdot \texttt{right} \mapsto \gamma \\ &\qquad * \beta \cdot \mathbf{tree} * \gamma \cdot \mathbf{tree}, \alpha \neq 0)\end{aligned}$$

The second rule of this definition corresponds to the following graphical syntactic unfolding:



Trees with parent pointers combine features of trees (two sub-structures) and of doubly-linked lists (pointer to the previous element), thus can be described just as easily.

2.3.3 Relations between shape and numerics

Our language of inductive definitions also captures relations between shape properties and numerical properties.

Length: Inductive definition below expresses that its main argument points to a list the length of which is represented by its auxiliary parameter:

$$\begin{aligned}&\alpha \cdot \mathbf{list\_len}(\delta) ::= \\ &\quad (\mathbf{emp}, \alpha = 0 \wedge \delta = 0) \\ &\vee \quad (\alpha \cdot \texttt{next} \mapsto \beta * \alpha \cdot \texttt{prev} \mapsto \gamma \\ &\qquad * \beta \cdot \mathbf{list\_len}(\delta), \alpha \neq 0 \wedge \alpha = \delta + 1)\end{aligned}$$

We note that the same notion can also be expressed using a primitive notion of length (section 2.2.4 and section 2.2.5).

Sortedness: A sorted list inductive definition can be obtained by adding an additional parameter which denotes a lower bound for the first value in the list; in turn, this first value will be passed as an additional parameter to the recursive call, to serve as a lower bound for the next element. To express that a list is sorted (with no assumption on the range of the first element), we simply need to start with $-\infty$ as a lower bound for the very first element:

$$\begin{aligned}&\alpha \cdot \mathbf{list\_sort} ::= \alpha \cdot \mathbf{aux}(-\infty) \\ &\alpha \cdot \mathbf{aux}(\delta) ::= \\ &\quad (\mathbf{emp}, \alpha = 0) \\ &\vee \quad (\alpha \cdot \texttt{next} \mapsto \beta * \alpha \cdot \texttt{prev} \mapsto \gamma \\ &\qquad * \beta \cdot \mathbf{aux}(\beta), \alpha \neq 0 \wedge \delta \leq \beta)\end{aligned}$$

Balanced trees: Adding branch relation between length of branches and tree shape yields balanced tree data structures. Such structural invariants can be expressed using our language of inductive definitions; for instance, in the case of *red-black trees*:

- correct alternance of black nodes and red nodes can be expressed in a similar manner as for the length in **list_len**;
- binary search tree structure can be expressed in a similar manner as the sortedness in **list_sort**;
- optionally, parent pointers can be expressed in a similar way as in **dll**.

A full definition of inductive predicate **red_black** was presented in [20]. Other balanced tree structures such as AA trees, or AVL trees can also be expressed.

Chapter 3

Shape analysis algorithms

In this chapter, we set up an abstract interpretation based static analysis [29] which uses the abstract domain introduced in chapter 2. This static analysis has also been described in [21] and in [20], and was implemented in Xisa (eXtensible Inductive Shape Analyzer). Section 3.1 reviews the principle of the analysis including some specific operations which consist in *unfolding* and *folding* inductive predicates. Then, section 3.2 reviews common transfer functions such as assignment and tests in a more thorough manner. Section 3.3 provides an in-depth discussion of unfolding. Section 3.4 describes the folding algorithms. Finally, section 3.5 and section 3.6 summarize the signature of domains and section 3.6 and the Xisa implementation.

3.1 Analysis principles

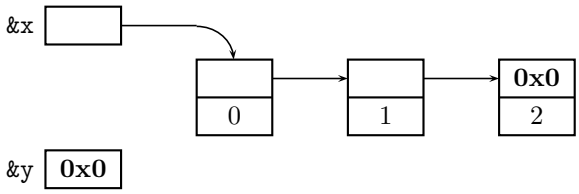
Before describing the analysis operations formally, we illustrate it on a simple example.

A list reversal example: In this section, we consider the program below, which is a classical example of a list reversal algorithm:

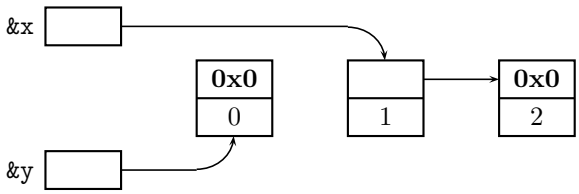
```
list ⋆ l assumed to point to a list
list ⋆ r = NULL;
list ⋆ t;
while(l != NULL){
    t = l -> next;
    l -> next = r;
    r = l;
    l = t;
}
```

This program performs an "in-place" reversal of a singly-linked list, which means that it does not allocate or deallocate any element; instead, it simply switches pointers, so that in the end the whole list (that was initially pointed to by variable `l`) is reversed (the final result is pointed to by variable `r`). For instance, when applied to a list of length 3, we obtain the following run (we do not display intermediate values of variable `t`, which serves only as a temporary):

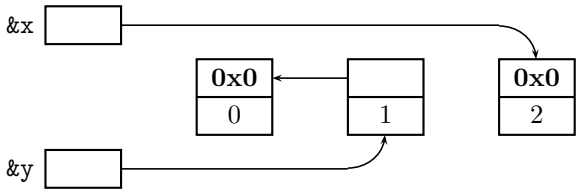
- initial configuration:



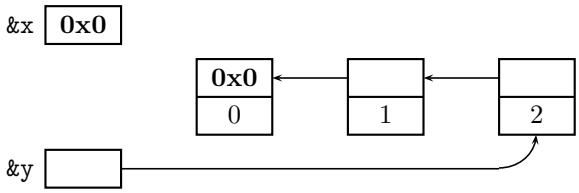
- after one iteration:



- after two iterations:



- final configuration, after three iterations:

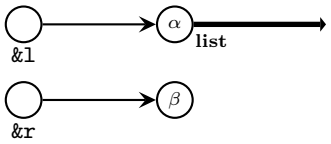


A static analysis of this program should rely on the assumption that `l` points to a well-formed singly-linked list, and use that assumption in order to:

1. prove memory safety (absence of dereferences of a null or dangling pointer, absence of memory leaks...);
2. prove that the structure obtained in the end is coherent with the intuition of this algorithm, that is, that in the end, `r` points to a well-formed singly-linked list.

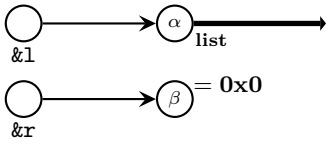
We may expect the static analysis to prove more than this, e.g., that in the result, the elements of the list appear in the reverse order compared to the initial structure, yet we do not consider such a goal in this section.

Analysis sketch: As this program manipulates singly-linked lists, we are going to assume the domain is parameterized with only one inductive definition, that is the **list** inductive definition shown in section 2.2.4. The assumption that `l` points to a well formed singly-linked list translates to the invariant shown in the graphical representation below:

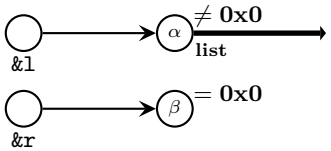


We note that no information is known about the value of `r`, so there is no constraint on β (in particular, this node has no outgoing edge). Also, we drop the temporary variable `t` for the sake of clarity.

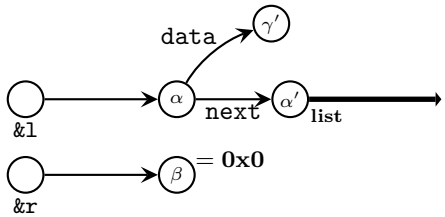
The analysis is based on a standard abstract interpretation of the program [29, 28]. The assignment of the null pointer to `r` results in a constraint that the value stored into `r` be null. Such a fact should be represented inside the numerical domain $\mathbb{D}^\sharp_{\mathrm{N}}$, yet, we let node labels depict such constraints in the following, for the sake of clarity:



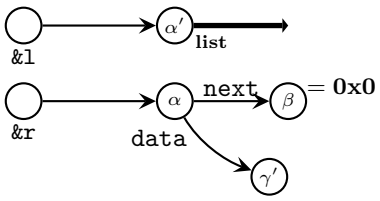
The analysis of a loop should rely on a fixpoint computation, where the analysis of the loop body is iterated until a fixpoint is reached, that over-approximates the set of all states at the loop head. At the loop entry, the analysis should take into account the fact that the test succeeds, thus the value stored in variable `l` is not equal to the null pointer, which should also be taken care of inside $\mathbb{D}^\sharp_{\mathrm{N}}$.



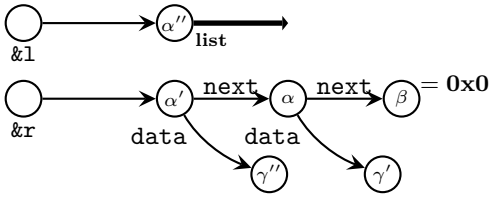
At this, stage, a series of four assignments need be analyzed inside the body of the loop. However, we note that these require to read `l->next` even though that memory cell does not appear explicitly in the graph; it is actually *summarized* as part of the inductive predicate $\alpha \cdot \mathbf{list}$. In order to make the analysis of these statements possible, we need to "*unfold*" the inductive predicate, so as to make the first element of the list explicit. That unfolding into a list element and a tail is sound, since the value of `l` is known to be non null, which means the list is not empty. This unfolding operation is the topic of section 3.3. Thus, we can rewrite the above abstract value into the following:



Once this unfolding is performed, assignments can be safely analyzed by flipping points-to edges so as to reflect each operation, so that, at the end of the analysis of the first iteration of the loop, we get:

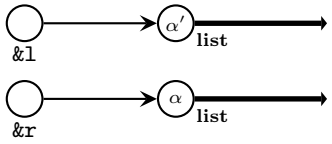


If we run the analysis of the loop one more time, we get the graph below as a result:



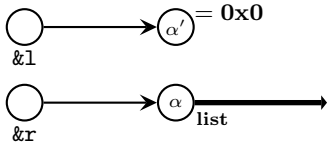
Repeating this same series of analysis steps a fixed number of times would be feasible, and yield sound over-approximations of the concrete states that can be encountered after some given number of iterations. However, iterating this process until stabilization would *not terminate*, as it would generate more and more points-to edges by unfolding, and never abstract any region.

In the other hand, we notice that the structure pointed to by `r` in these two graphs indeed corresponds to a well formed singly-linked list, that could be summarized using the **list** predicate. We would then obtain the graph below:



We remark that this abstract value conveys the information that both list summaries correspond to *disjoint* memory regions, which is exactly what we observed when looking at a concrete run of the analysis. It can be shown that this graph is indeed an invariant for the loop. It over-approximates both graphs that were produced after zero, one and two iterations of the analysis of the loop. As usual, the difficult step is to have the analysis *infer* that loop invariant, with some sort of a *widening* operator. In this case, we can see that widening should perform the converse operation of unfolding, i.e., it should *fold* inductive predicates. Folding is the topic of section 3.4

Furthermore, analyzing the loop exit condition from that graph gives the following, which is the expected analysis result (i.e., `l` stores the null pointer whereas `r` points to a well formed singly-linked list):



Analysis main operations: This overview shows that conventional analysis operations such as assignment, condition test, and widening are needed. However, we note that transfer functions may require a preliminary *unfolding* of inductive predicates. This unfolding operation can actually be found in all shape analyses that perform summarization [109, 44]. In the other hand, the widening operator is also based on a new operation, that is not common to, e.g., static analyses targeted at the inference of numerical invariants. Indeed, widening operators can often be viewed as operators discarding unstable constraints. However, in our case, widening relies on the *folding* of inductive predicates, which amounts to abstracting local properties with more global ones.

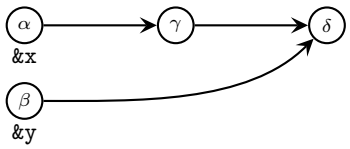
3.2 Transfer functions

We first consider standard transfer functions, such as assignments and condition tests. Such transfer functions need to evaluate expressions in the abstract level, and update abstract values both in $\mathbb{D}^\sharp_S$ and in $\mathbb{D}^\sharp_N$ accordingly. We assume here the memory regions they operate on are fully described by points-to edges; the case where inductive predicates need be unfolded will be dealt with in section 3.3.

Evaluation of expressions and l-values: A points-to edge describes precisely a contiguous memory region, and binds a node representing its address into a node representing its contents, as we observed in section 2.1.3. Therefore:

- an expression of scalar type should evaluate into a node (a node abstracts a set of values);
- an l-value should evaluate into a points-to edge $\alpha \cdot \mathtt{f} \mapsto \beta$ representing the memory cell it evaluates to; then the address of that l-value is the origin of the edge, that is $\alpha \cdot \mathtt{f}$.

For instance, let us consider the abstract value described in the graphical representation below:



Then, l-value `x` evaluates into the edge $\alpha \mapsto \gamma$, i.e. α abstracts the result of expression `&x`, and γ abstracts the result of expression `x`. Similarly, l-value `y` evaluates into the edge $\beta \mapsto \delta$. Last, since the value of expression `x` is γ, l-value $\star$`x` is abstracted by the edge $\gamma \mapsto \delta$, which means the value of expression $\star$`x` is abstracted by node δ [71].

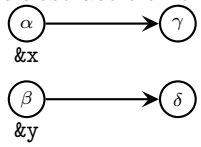
Assignments: The transfer function for analyzing assignments should over-approximate the effect of an assignment instruction. Following the above discussion on evaluation of expressions, we remark that the analysis of an assignment $l := e$ (where $l \in \mathrm{Lvals}_{\mathbb{X}}$ is an l-value with variables in $\mathbb{X}$ and $e \in \mathrm{Exprs}_{\mathbb{X}}$ an expression with variables in $\mathbb{X}$) can be split into three steps:

1. evaluating l into an edge $\alpha \cdot \mathtt{f} \mapsto \beta$, which represents the cell that is modified by the assignment;

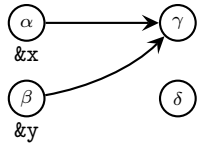
2. evaluating e into a node γ, over-approximating the result of the evaluation of the expression in the concrete;
3. replacing edge $\alpha \cdot \mathtt{f} \mapsto \beta$ with edge $\alpha \cdot \mathtt{f} \mapsto \gamma$; then, node β can be discarded unless it appears somewhere else in the graph.

There are actually two main cases, depending on how e evaluates.

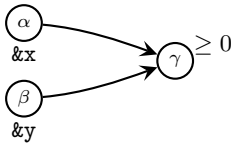
- When e is the dereference of an l-value, there already exists a node in the graph, which over-approximates the result of e. For instance, let us consider the abstract element below:



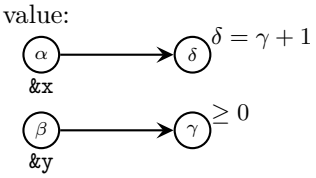
  Then, assignment $\mathtt{y} := \mathtt{x}$ boils down to the flipping of the edge departing from the node representing `&y` to the node representing the value stored in variable `x`:



  Then, node δ can be dropped, since there is no constraint left about the value it represents.
- When e is more complex (e.g., involves numeric computation), there exists no such node, thus a new node should be created, and the assignment should be performed in the underlying domain $\mathbb{D}^\sharp_{\mathrm{N}}$. For instance, let us assume that both `x` and `y` have integer type, and analyze assignment $\mathtt{x} := \mathtt{x} + 1$, from the abstract pre-condition below, which expresses the fact that `x` and `y` store the same value, which is known to be positive:



  Then, no node in the graph represents the new value, hence a fresh node δ should be added to the graph, and the assignment $\delta = \gamma + 1$ should be analyzed in the numerical domain (we note that this assignment is entirely expressed in terms of the symbolic nodes of the graph); we then obtain the abstract value below, which encloses additional numeric invariants expressing the relation between the old value (which is still stored in variable `y`) and the new value:



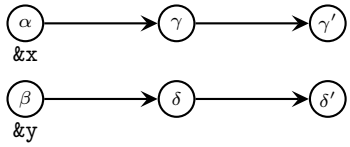
To summarize, the $\mathtt{assign}_{\mathbb{S}^\sharp} : \mathbb{S}^\sharp \times \mathrm{Lvals}_{\mathbb{X}} \times \mathrm{Exprs}_{\mathbb{X}} \longrightarrow \mathbb{S}^\sharp$ transfer function for analyzing an assignment relies on the creation of fresh nodes, and on a sound $\mathtt{assign}_{\mathrm{N}} : \mathbb{D}^\sharp_{\mathrm{N}} \times \mathrm{Lvals}_{\mathbb{V}^\sharp} \times \mathrm{Exprs}_{\mathbb{V}^\sharp} \longrightarrow \mathbb{D}^\sharp_{\mathrm{N}}$ transfer function in the underlying numerical domain, which over-approximates the effect of assignments over functions in $\mathbb{V} \mathrel{\square} \mathbb{I}$. The soundness condition of the underlying operator $\mathtt{assign}_{\mathrm{N}}$, for assignments involving only symbolic variables writes down:

$$\forall l \in \mathrm{Lvals}_{\mathbb{V}^\sharp}, \forall e \in \mathrm{Exprs}_{\mathbb{V}^\sharp}, \forall \nu \in \gamma(N),$$
$$\nu[[\![l]\!](\nu) \leftarrow [\![e]\!](\nu)] \in \gamma_{\mathrm{N}}(\mathtt{assign}_{\mathrm{N}}(N, l, e))$$

where $[\![e]\!] : \mathbb{V} \mathrel{\square} \mathbb{I} \longrightarrow \mathbb{V}$ (resp., $[\![l]\!] : \mathbb{V} \mathrel{\square} \mathbb{I} \longrightarrow \mathbb{V}_{\mathrm{addr}}$) describe the concrete semantics of expressions (resp., l-values). Then, $\mathtt{assign}_{\mathbb{S}^\sharp}$ is also sound, which writes down in a very similar way, except that it operates on expression using regular variables (i.e., as found in the program).

Condition tests: The analysis of condition tests follows a very similar process as that of assignments. Evaluation of conditions works essentially the same way as for any other expression, as conditions are simply expressions with boolean type. Then, the analysis of the condition can be done either using the information that is present in the graph or in the numerical domain.

- In some cases, a condition can be asserted in the graph. This is the case for many pointer tests. For instance, a node that is at the origin of a points-to edge is known to be non null, since the presence of an outgoing edge means the node denotes the address of a regular memory cell, hence it cannot be the null pointer. Similarly, testing the equality of pointers can often be done in the graph, in an exact manner. For example, the abstract value represented in the graph below contains the information that pointers `x` and `y` are not equal, as separation entails the cell of address γ is disjoint from that of address δ (in the other hand, it says nothing about the values of δ' and γ'):



- When the condition being tested involves non trivial numerical conditions, the expression involving program variables should be translated into an equivalent expression involving symbolic nodes, and a $\mathfrak{guard}_{\mathrm{N}} : \mathbb{D}^{\sharp}_{\mathrm{N}} \times \mathrm{Exprs}_{\mathbb{V}^{\sharp}} \times \mathbb{B} \longrightarrow \mathbb{D}^{\sharp}_{\mathrm{N}}$ operator should be applied to that condition, where $\mathfrak{guard}_{\mathrm{N}}$ is assumed to be sound in the sense that:

$$\forall c \in \mathrm{Lvals}_{\mathbb{V}^{\sharp}}, \forall b \in \{\mathbf{true}, \mathbf{false}\}, \forall \nu \in \gamma(N), \\ [\![c]\!](\nu) = b \Longrightarrow \nu \in \gamma_{\mathrm{N}}(\mathfrak{guard}_{\mathrm{N}}(N, c, b))$$

This defines an operator $\mathfrak{guard}_{\mathbb{S}^{\sharp}} : \mathbb{S}^{\sharp} \times \mathrm{Exprs}_{\mathbb{X}} \times \mathbb{B} \longrightarrow \mathbb{S}^{\sharp}$, which satisfies a soundness condition very similar to that of $\mathfrak{guard}_{\mathrm{N}}$.

As mentioned in the beginning of this section, these operators cannot be applied when part of the l-values or expressions that are considered involve memory cells which are not exposed as points-to edges. In such cases, it is necessary to unfold some inductive predicates first, and then to apply the operators discussed above when the condition that all memory cells involved be exposed is satisfied.

3.3 Unfolding inductive edges

The unfolding operation actually comprises two steps:

1. determining which inductive predicate needs to be unfolded;
2. performing the unfolding of that predicate.

We first assume the inductive predicate to unfold is known and show how it can be unfolded in section 3.3.1 and 3.3.2; then, we discuss the determination of the predicate to unfold in section 3.3.3.

3.3.1 Unfolding of an inductive edge

Syntactic unfolding of an inductive predicate was introduced first in section 2.2.4. The principle of syntactic unfolding is to rewrite an inductive predicate $\alpha \cdot \iota$ into the definition of one of the rules of ι, which generates a pair made of a shape abstract value and of a pure formula, constraining nodes of that shape abstract value.

To turn this principle into an abstract domain operation, we have to address two issues:

- for the sake of soundness, we need unfolding to account for *all* rules of ι;
- $\mathbb{D}^{\sharp}_{\mathrm{N}}$ cannot deal with arbitrary pure formulae, thus we need to over-approximate this part.

Unfolding algorithm: To solve the first issue, we let the analysis manipulate *disjunctions* of elements of $\mathbb{S}^{\sharp}$, as observed in section 2.2.2: the analysis computes a finite set of elements of $\mathbb{S}^{\sharp}$ at each control state, instead of just one abstract element. This way, whenever an inductive predicate is unfolded, we can simply collect the set of all elements that arise from syntactical unfolding of all the rules of the inductive definition.

The second issue will actually be taken care of by applying the $\mathfrak{guard}_{\mathrm{N}}$ operator to the pure part of the rule, which may cause some loss of precision, but will always yield a sound result.

Thus, the $\mathfrak{unfold} : \mathbb{S}^{\sharp} \longrightarrow \bigvee \mathbb{S}^{\sharp}$ *unfolding domain operator* can be defined as follows:

Definition 1. Unfolding.

Let us assume inductive definition ι is composed of n rules $r_0, \ldots, r_{n-1}$.
Then,

$$\mathfrak{unfold}(\hat{\epsilon}^{\sharp}, S * \alpha \cdot \iota, N) = \bigvee_{0 \leq i < n} (\hat{\epsilon}^{\sharp}, S_i, N_i)$$
$$\text{where } \begin{cases} \alpha \cdot \iota \xrightarrow{\mathcal{U}}_{\iota, r_i} (S_i, F_i) \\ N_i = \mathfrak{guard}_{\mathrm{N}}(F_i, \mathbf{true}, N) \end{cases}$$

This algorithm produces sound results:

Theorem 1. Soundness of unfolding.

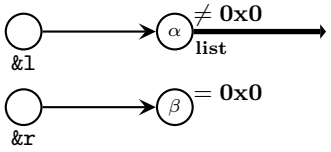
We use the same notations as in definition 1. Then:

$$\gamma_{\mathbb{S}}(\hat{\epsilon}^{\sharp}, S, N) \subseteq \bigcup_{0 \leq i < n} \gamma_{\mathbb{S}}(\hat{\epsilon}^{\sharp}, S_i, N_i)$$

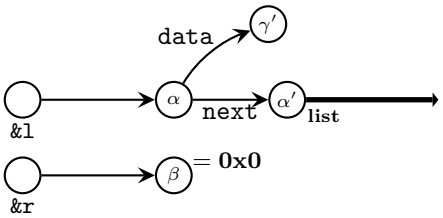
This result follows from the definition of the concretization of inductive predicates, and from the soundness of $\mathfrak{guard}_{\mathrm{N}}$. This operation may lead to a loss of precision, when $\mathfrak{guard}_{\mathrm{N}}$ itself loses precision.

This unfolding operator can be related to the *focus* operation of TVLA [109], and should be applied in order to expose fields which are summarized: this use matches well with the "partial concretization" role of the focus operator, for the computation of precise transfer functions [96, 125].

Examples: Let us first consider the case of the **list** inductive predicate, that arose in section 3.1. When attempting to analyze the series of assignments in the loop body, we came across the following abstract element, where it was necessary to materialize `l -> next`, by unfolding the **list** predicate:



Unfolding this predicate naturally yields two disjuncts, one corresponding to the empty list, and the other to the non empty list. In the empty list case, the $\alpha \cdot \mathbf{list}$ edge disappears (it is replaced by an empty region) and the fact that α is the null pointer. However, as the predicate $\alpha \neq \mathbf{0x0}$ is carried out in $\mathbb{D}^\sharp_{\mathrm{N}}$, asserting that $\alpha = \mathbf{0x0}$ results in the $\bot$ abstract value in $\mathbb{D}^\sharp_{\mathrm{N}}$ (meaning that no state satisfies both properties). In the other hand, the non empty list case corresponds to the abstract element below, where the first element of the list is completely exposed:



Thus, in this case, the unfolding actually does not generate an additional disjunct.

3.3.2 Unfolding segments

The concretization of segments is also based on syntactic unfolding, as shown in section 2.2.5, thus segments can also be unfolded. However, we also observed the base case was different, which we expect the unfolding to reflect.

Moreover, segments may need be unfolded from *either end*. Indeed, it may make sense to refine the abstract information about the tail of a segment, e.g., in the case of a backward traversal of a doubly-linked list.

Thus, we start with a preliminary result, which will be useful in order to set up the unfolding operator for segments. This result states that a segment of length $k = i + j$ (segment length was introduced in section 2.2.5) can be split into a segment of length i and a segment of length j, which are both labeled with the same inductive in the middle (however, that inductive may not be of the same kind as those at both ends of the initial segment):

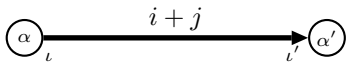
Theorem 2. Segment splitting.

Let $(\sigma, \nu) \in \gamma_{\mathrm{S}}(\alpha \cdot \iota \overset{i+j}{\ast\!\!\Longleftarrow} \alpha' \cdot \iota')$. Then, there exist an inductive ι'', a fresh node $\alpha'' \notin \mathbf{dom}(\nu)$, and a concrete value v such that:

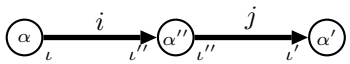
$$(\sigma, \nu[\alpha'' \leftarrow v]) \quad \in \quad \gamma_{\mathrm{S}}(\alpha \cdot \iota \overset{i}{\ast\!\!\Longleftarrow} \alpha'' \cdot \iota'' \ast \alpha'' \cdot \iota'' \overset{j}{\ast\!\!\Longleftarrow} \alpha' \cdot \iota')$$

This result can be proved by induction on i. The new inductive definition ι'' can actually be known more precisely, by determining what other inductive definitions may arise, when unfolding ι (e.g., if ι is **list**, then ι'' may only be **list**).

Using graphical representations, this theorem states that whenever a concrete state can be described by



then, it can also be described by some element of the form below, for a certain ι'':



In the following, we consider the two practically interesting cases:

- if $i = 1$, this corresponds to an unfolding at the head of the structure;
- if $j = 1$, this corresponds to an unfolding at the tail of the structure.

Furthermore, when $\iota = \iota'$, the empty segment will also need to be considered: it unfolds into an empty region, and generates the constraints that its destination and origin be equal, and the same for the arguments at the destination and at the origin, following segment concretization (section 2.2.5).

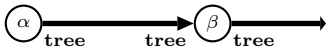
Forward segment unfolding: Let us consider an abstract element containing the segment $\alpha \cdot \iota \ast\!\!\Longleftarrow \alpha' \cdot \iota'$. The forward unfolding algorithm carries out the following steps:

- generate a disjunct for the empty segment;

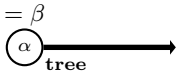
- apply the splitting result with $i = 1$ and generate one disjunct per choice of an inductive predicate in a rule of ι, and per intermediate inductive ι'' (e.g., when unfolding an **tree** segment, two cases would appear, where the segment follows the left and right subtrees); in each case, a new set of edges should generated as in the case of inductive unfolding, and pure constraints should be over-approximated in the numerical domain using $\mathfrak{guard}_{\mathrm{N}}$.

This algorithm is sound in the same sense as in theorem 1.

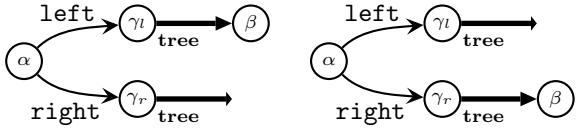
Forward segment unfolding is very useful when analyzing multi-cursor traversal algorithms. For instance, let us consider the case of a tree structure (described by **tree**), and the analysis of a procedure that inputs a tree, uses a first cursor to look down into the tree, and then attempts to visit the structure with another cursor; then we get the situation below, with a tree structure, which is summarized into two parts (a segment and a sub-tree):



The empty segment case corresponds to the case where $\alpha = \beta$; in practice, the unfolding should them rename β into α everywhere in the graph, and produce:



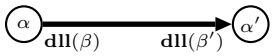
The empty **tree** rule generate no disjunct here, as it does not generate any new inductive predicate. However, the other **tree** rule does generate *two* disjuncts since it generates two inductive predicates corresponding to the left and right sub-trees.



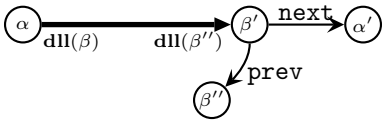
Intuitively, these two disjuncts correspond to the cases where the target of the initial segment is in the left or in the right sub-tree.

Backward segment unfolding: The principle of backward segment unfolding is the same as for the forward unfolding, except that it operates on the target node. The soundness result is also similar.

Backward segment unfolding is useful when analyzing algorithms that attempt to traverse data structures backwards, which can be done, e.g., with a doubly-linked list. Let us consider the segment below:



The backward unfolding of this segment produces *two* disjuncts, including the empty segment, which comes with the constraints that $\alpha = \alpha'$ and that $\beta = \beta'$ and the element below:

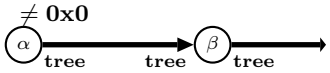


3.3.3 Unfolding control

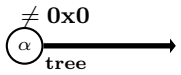
In the last two subsections, we have shown how to unfold a specific edge. However, in practice it is not always clear what edge should be unfolded, and determining which unfolding to perform is non trivial in general. Before we formalize rules triggering unfolding, we consider a few basic cases.

Unfolding triggered from a main argument: When the analysis needs to look up for an edge of the form $\alpha \cdot \mathtt{f} \mapsto \beta$ (where α and `f` are given) while no such edge exists, and if α is the origin of an inductive predicate $\alpha \cdot \iota$, it is natural to unfold that edge. Indeed, the unfolding an inductive predicate associated to α tends to expose fields from α. This is how we treated the **list** definition in the list reversal example, in section 3.1. It actually appears that this unfolding did succeed, since the non empty list rule exposes fields `next` and `data`, starting from α and since α is known to be non null so that the empty list rule does not apply.

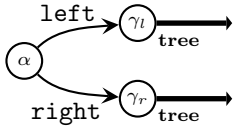
Repeated unfolding: The case where α is the origin of a segment edge can be treated in a similar way, yet it may generate some complications, when the segment may unfold into the empty segment. Let us consider the concrete example below where we try to expose a `left` field starting from the root of a tree, which is known to be non empty and which is decomposed into a segment and a sub-tree, as in section 3.3.2:



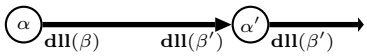
Then, the disjunct corresponding to the empty segment is:



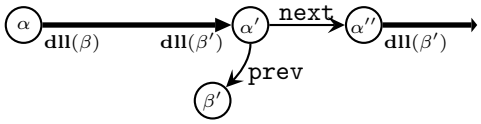
We can see that the first unfolding *failed* to expose the `left` field, yet a second unfolding step will achieve this:



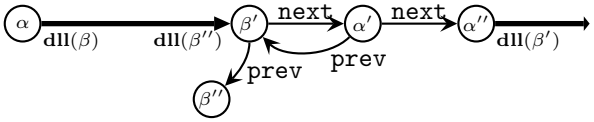
Indirect unfolding: Some cases are even trickier. Let us consider the doubly-linked list case below, assuming that we are interested in exposing two successive `prev` fields from α' (we also assume all necessary non required null-ness conditions are guaranteed):



The first one is summarized by the inductive edge which starts from α', since $\alpha' \cdot \mathbf{dll}(\beta')$ expresses α' is the address of a doubly-linked list element. Unfolding that predicate results in the following:



The `prev` field from α' is exposed and corresponds to node β'; however, unfolding another `prev` field from β' is *not trivial*, since β' is not the origin of any inductive edge. Yet, we remark that it is also the parameter of the inductive predicate at the endpoint of the segment. This means that β' is actually the address of a doubly-linked list element that is summarized as part of the segment, which suggests we should unfold the segment from its end. Applying the backward segment unfolding algorithm of section 3.3.2, we obtain the following:

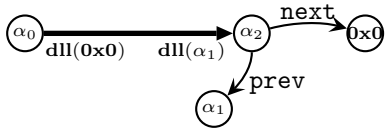


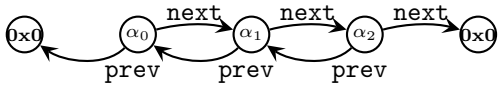
Thus, we note that in this case, exposing all fields needed to evaluate an expression of the form `x -> next -> next` (where α denotes the content of `x`) requires unfolding *two distinct* summarized regions (first, the inductive edge, and the segment, backwards).

The unfolding problem: As we could see in examples, deciding which inductive edge to unfold is not trivial; it is actually a non computable issue due to the expressiveness of inductive definitions. In practice, a static analyzer should rely on strategies which may fail to determine the right unfolding. When the strategies fail, the analysis will typically suffer a serious loss of precision.

The strategy implemented in Xisa [20] relies on the following principles:

- it relies on a notion of *types* of inductive parameters, which relates fields to ranks, corresponding to the elements they point to, in sequences of unfoldings; for instance, in a doubly-linked list, field `next` points to the next element whereas field `prev` points to the *previous* one; for instance, in the figure below, the first graph is "more abstract" than the second one; intuitively, dereferencing the `prev` field from α_1 in the first graph requires unfolding the segment, which is consistent with the fact α_1 appears as a parameter of the segment:





- it iterates a bounded number of repeated unfoldings (so as to ensure termination), and fails if the points-to edge required by the analysis does not unfold after k iterations.

3.4 Folding

We now study operations that aim at *folding* sets of atomic predicates into more abstract properties, involving inductive predicates.

3.4.1 Folding principles

In section 3.1, we noticed that some sort of widening operator was required in order to infer shape invariants, and we remarked that this widening operator should basically do the opposite of unfolding.

There are actually many flavors of folding in the literature. We now set up the foundations for this class of domain operations, before we provide an in-depth discussion of the two which are actually needed in our analysis.

Folding applications: *Widening* is a first application for folding, since repeated unfolding of inductive predicates is a potential source of non termination. However, folding is required in other operators as well:

- **Inclusion checking** takes two abstract elements $X, Y \in \mathit{amems}$ and attempts to establish that the concretization of X is included in that of Y (this check is required to establish that a series of iterations to compute an abstract post-fixpoint converged). One way to achieve this is to refine Y (by unfolding some inductive predicates) and to try to specialize it by successive unfolding into X, however this process is not obvious, as each step should *under-approximate* Y (otherwise, the process would not be sound) and the unfolding operator of section 3.3 produces *over-approximations* (theorem 1), and not under-approximations. Therefore, it seems much easier to coarsen X by repeated folding until we reach Y; then, it means we obtain a proof that $\gamma(X) \subseteq \gamma(Y)$.
- **Unary abstraction** takes an abstract element X and discards information that does not sound immediately useful, so as to produce a simpler element Y. This may be desirable even when termination is not a concern, as it may reduce the cost of subsequent analysis steps, so that the cost of computing a more abstract element is worthwhile. A typical way to achieve this is to replace some precise points-to predicates with some more abstract inductive predicates. In practice, this amounts to forgetting the number of elements of a list (resp., the overall structure of a tree), keeping only the fact that the whole structure is a list (resp., a tree).

These problems are all closely related, as they all require to collapse parts of the graphs into coarser properties. The algorithms to solve these problems are actually also related, and face similar difficulties, which we discuss in the following.

Per region folding: Section 3.3 shows that unfolding a given inductive or segment edge is rather straightforward, yet determining which edge to unfold is hard. Finding out how to fold a set of edges that has yet to be determined is even harder, as there exist a very high number of possible choices (exponential in the number of edges in the graph). However, when a set of edges corresponding to an instance of a more abstract inductive predicate is known, checking that this folding is sound is easier.

Let $(\hat{\epsilon}^\sharp, S, N)$ be an abstract value, such that graph S can be split into two sets of edges S_0 and S_1 such that $S = S_0 \uplus S_1$. We also assume that $\gamma_{\mathrm{S}}(S_0) \subseteq \gamma_{\mathrm{S}}(\alpha \cdot \iota)$. In other words, it is sound to fold S_0 into $\alpha \cdot \iota$. Then, the concretization rules of figure 2.2(c) allow to prove that:

$$\gamma_{\mathrm{S}}(S) = \gamma_{\mathrm{S}}(S_0 \uplus S_1) \subseteq \gamma_{\mathrm{S}}(\{\alpha \cdot \iota\} \uplus S_1)$$

This property is actually a direct consequence of separation. This means that graphs can be weakened region by region: at each step, we simply need to select a set of edges to weaken, and iterate this process.

Therefore, folding algorithms should carry out two important steps:

- **determination of disjoint regions**, that are candidate to folding;
- **search for folded inductive predicates** over-approximating each region.

The second step can be done in a completely independent way for each separate regions.

The naming issue: When attempting to check inclusion of two abstract elements or to compute an over-approximation of their join, the per-region weakening discussed in the previous paragraph actually needs be performed in both graphs in the same time. For instance, inclusion checking should compute *compatible* partitions of both inputs, to that inclusion checking can succeed. Otherwise, inclusion checking will not succeed.

This imposes a constraint upon the inclusion checking algorithm, but it also brings the opportunity for one graph to help partition the other one and vice versa. Therefore the two steps mentioned

above will actually be performed on both graphs *in the same time*.

However, symbolic node names may not be consistent in both graphs. This is actually impossible to avoid since graphs may not have the same numbers of nodes, e.g., when one has a very "concrete" representation for a memory region and the other simply contains an inductive predicate for that region. Similarly, join and widening operators need to produce a new abstract element, with a set of symbolic nodes that may not be in a one-to-one correspondence with those of the arguments.

Therefore, folding algorithms should maintain consistent *naming relations* between the abstract elements they are applied to or that they generate. These node mappings will also indirectly relate memory regions: for instance, in the case of a join, if node α (resp., β) of the first argument is bound to node α' (resp., β') of the second argument, then a list segment between α and β should naturally be related to a structure between α' and β'; if that structure can also be folded into a list segments, then both regions can be abstracted by a list segment.

Dealing with numerical invariants: Folding should also take care of numerical abstract values; there are actually two separate issues related to the numerical part:

- since syntactic unfolding also generates "pure formulas" that constrain the values denoted by graph symbolic nodes, doing the reverse folding operation requires to check that these constraints are actually satisfied;
- after applying folding, a consistent and weakened numeric abstract element needs be computed.

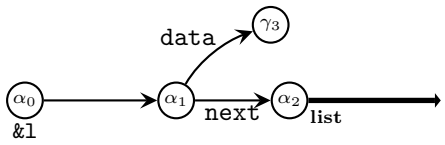
However, since numerical abstract values express constraints on symbolic variables, the naming issue that we mentioned above also needs be taken care of when considering these two points. To do that, we will also rely on the node mappings that relate nodes of the arguments and results of folding algorithms.

3.4.2 Inclusion checking

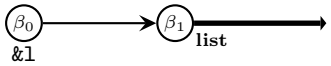
We are now going to discuss the two main forms of folding that we use in our analysis, that is inclusion checking and widening. Since inclusion checking is used in the definition of join, we consider it first.

A simple example: Let us consider an example inspired by some of the abstract elements found in section 3.1, and try to check the inclusion of X_l into X_r, where:

- X_l is the abstract element



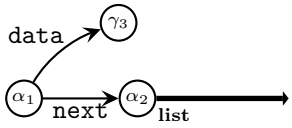
- X_r is the abstract element



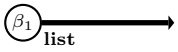
Note that we assume the symbolic nodes of both graphs belong to disjoint name spaces to clear any confusion. As a first step, we note that both α_0 and β_0 denote the value `&l`, thus it makes sense to bind them together. Then, we note that both graphs contain an edge from these nodes, that correspond to the null offset; these edges respectively point to α_1 and β_1, which suggests that we should:

- bind α_1 and β_1;
- conclude that the "abstract region" $\alpha_0 \mapsto \alpha_1$ is included into $\beta_0 \mapsto \beta_1$ (they are actually equal).

At this stage, to finish checking that X_l is included in X_r, we simply need to check the inclusion of



into



To do that, we can note that unfolding the inductive predicate in the right hand side will produce a graph that is equal to the left hand side up to node renaming, and with the constraint that β_1 (mapped to α_1) be non null clearly α_1 cannot be the null pointer, as it is the origin of several points-to edges.

Inclusion checking algorithm: Following the principles outlined in section 3.4.1, the inclusion checking algorithm should maintain node mappings of the form $\Phi : \mathbb{V}^\sharp \longrightarrow \mathbb{V}^\sharp$, mapping nodes of the right argument into nodes of the left argument. Each step should refine that mapping and/or prove the inclusion of sub-graphs (which then get discarded). In the example shown above, we would

use node mapping Φ defined by:

$$\Phi : \begin{array}{lcl} \beta_0 & \mapsto & \alpha_0 \\ \beta_1 & \mapsto & \alpha_1 \end{array}$$

In the example, we noticed that the mapping may get extended when matching regions, thus the inclusion checking algorithm should infer both mappings in the same time as it also weakens sets of edges into inductive predicates in the left argument.

This process can be seen as a rewriting process over configurations of the form $(S_\mathrm{l}, S_\mathrm{r}, \Phi)$, where:

- S_l (resp., S_r) is the current leftover of the shape part of the left (resp., right) argument;
- Φ is the current mapping.

Furthermore, the left and right numerical invariants N_l and N_r are supposed fixed. We let $\overset{\sqsubseteq}{\rightsquigarrow}$ denote the rewriting relation used to go from one configuration to another (we give its definition in the next paragraph). The foundation of the proof of correctness of the algorithm is the following property that should be preserved by all rewriting rules:

$$\begin{array}{l}(S_\mathrm{l}, S_\mathrm{r}, \Phi) \overset{\sqsubseteq}{\rightsquigarrow} (S'_\mathrm{l}, S'_\mathrm{r}, \Phi') \\ \quad \Longrightarrow \forall (\sigma, \nu) \in \gamma_{\mathbb{M}}(S_\mathrm{l}, N_\mathrm{l}),\ (\sigma, \nu \circ \Phi') \in \gamma_{\mathbb{S}}(S'_\mathrm{r})\end{array} \tag{3.1}$$

When applied to $(\hat{\epsilon}^\sharp_\mathrm{l}, S_\mathrm{l}, N_\mathrm{l}), (\hat{\epsilon}^\sharp_\mathrm{r}, S_\mathrm{r}, N_\mathrm{r}) \in \mathbb{S}^\sharp$, the inclusion checking algorithm proceeds as follows:

1. first, it derives an initial mapping Φ such that $\Phi(\alpha_{sider}) = \alpha_\mathrm{l}$ if and only if $\exists x \in \mathbb{X},\ \hat{\epsilon}^\sharp_\mathrm{l}(x) = \alpha_\mathrm{l} \wedge \hat{\epsilon}^\sharp_\mathrm{r}(x) = \alpha_\mathrm{r}$;
2. then, it applies progression rules repeatedly, rewriting one configuration into another by applying $\overset{\sqsubseteq}{\rightsquigarrow}$ from initial state $(S_\mathrm{l}, S_\mathrm{r}, \Phi)$, until both shape arguments get empty or until no rule applies.

The algorithm concludes inclusion holds (which we write $(S_\mathrm{l}, N_\mathrm{l}) \sqsubseteq^{\Phi'}_{\mathbf{S}} (S_\mathrm{r}, N_\mathrm{r})$) if a configuration of the form $(\mathbf{emp}, \mathbf{emp}, \Phi')$ is eventually reached, and if sound inclusion checking operator $\sqsubseteq_{\mathbf{N}}$ of $\mathbb{D}^\sharp_\mathrm{N}$ establishes that $N_\mathrm{l} \sqsubseteq_{\mathbf{N}} N_\mathrm{r} \circ \Phi'$ holds; otherwise, it does not conclude (it is incomplete).

Shape inclusion checking rules: Each rewriting step builds a fragment of a proof tree, showing that the inclusion holds. The most significant rewriting rules are shown in figure 3.1. Rule $(\mathbf{i} - \mathbf{sep})$ makes inclusion checking steps *local*; rule $(\mathbf{i} - \mathbf{pt})$ and rule $(\mathbf{i} - \mathbf{ind})$ allow to match regions described by the same edges up-to renaming (a similar rule $(\mathbf{i} - \mathbf{seg})$ exists for segments and is not shown); rule $(\mathbf{i} - \mathbf{segind})$ and rule $(\mathbf{i} - \mathbf{segseg})$ split segments or inductive predicates in the right hand side to over-approximate segments in the left hand side; last, rule $(\mathbf{i}\text{–}\mathcal{U})$ unfolds an inductive predicate in the right hand size so as to try to prove the left hand side can be folded into that predicate. Some rules, like rule $(\mathbf{i} - \mathbf{pt})$ enrich mapping Φ. We note that rule $(\mathbf{i}\text{–}\mathcal{U})$ requires a sound numerical domain operator $\mathfrak{prove}_\mathrm{N}$, to discharge proof obligations that arise in unfolding definitions in the right hand side; it should meet the following soundness condition:

$$\forall \nu \in N,\ \mathfrak{prove}_\mathrm{N}(N, F) = \mathbf{true} \Longrightarrow \nu \vdash F$$

A more exhaustive version of these rules is shown in [20]; rewriting relation $\overset{\sqsubseteq}{\rightsquigarrow}$ follows these.

Soundness: Each rule can be proved sound using the concretization rules of figure 2.2(c). Furthermore, equation 3.1 can be proved by induction on the length of sequences of rewritings using $\overset{\sqsubseteq}{\rightsquigarrow}$. The soundness of the inclusion checking follows:

Theorem 3. Soundness of inclusion checking. *If* $(\hat{\epsilon}^\sharp_\mathrm{l}, S_\mathrm{l}, N_\mathrm{l}) \sqsubseteq^{\Phi'}_{\mathbf{S}} (\hat{\epsilon}^\sharp_\mathrm{r}, S_\mathrm{r}, N_\mathrm{r})$*, then:*

$$\gamma_{\mathbb{S}}(\hat{\epsilon}^\sharp_\mathrm{l}, S_\mathrm{l}, N_\mathrm{l}) \subseteq \gamma_{\mathbb{S}}(\hat{\epsilon}^\sharp_\mathrm{r}, S_\mathrm{r}, N_\mathrm{r})$$

3.4.3 Join and widening

Join and *widening* operators should return an over-approximation of their arguments (we only consider *binary* operators here). In addition to that, widening operators enforce termination, that is, any sequence of iterations of this operator will eventually reach a limit. In this section, we introduce both join and widening operators. The principle of the join algorithm is to select carefully pairs of regions in both arguments, and to guess an over-approximation for each pair of regions:

- when those regions are equal up-to renaming, computing an over-approximation for both of them is trivial;
- otherwise, the algorithm will guess an inductive predicate and try to prove that both regions can be proved included in this predicate using $\sqsubseteq_{\mathbf{S}}$.

$$\frac{(S_\mathrm{l}, S_\mathrm{r}, \Phi) \overset{\sqsubseteq}{\rightsquigarrow} (S'_\mathrm{l}, S'_\mathrm{r}, \Phi')}{(S_\mathrm{l} * \mathring{s}_\mathrm{l}, S_\mathrm{r} * \mathring{s}_\mathrm{r}, \Phi) \overset{\sqsubseteq}{\rightsquigarrow} (S'_\mathrm{l} * \mathring{s}_\mathrm{l}, S'_\mathrm{r} * \mathring{s}_\mathrm{r}, \Phi')}\ \mathbf{i-sep}$$

$$\frac{\Phi(\alpha_\mathrm{r}) = \alpha_\mathrm{l} \qquad \Phi' = \Phi \uplus \{\beta_\mathrm{r} \mapsto \beta_\mathrm{l}\}}{(\alpha_\mathrm{l} \cdot \mathtt{f} \overset{s}{\mapsto} \beta_\mathrm{l}, \alpha_\mathrm{r} \cdot \mathtt{f} \overset{s}{\mapsto} \beta_\mathrm{r}, \Phi) \overset{\sqsubseteq}{\rightsquigarrow} (\mathbf{emp}, \mathbf{emp}, \Phi')}\ \mathbf{i-pt}$$

$$\frac{\Phi(\alpha_\mathrm{r}) = \alpha_\mathrm{l} \qquad \Phi(\bar{\beta}_\mathrm{r}) = \bar{\beta}_\mathrm{l}}{(\alpha_\mathrm{l} \cdot \iota(\bar{\beta}_\mathrm{l}), \alpha_\mathrm{r} \cdot \iota(\bar{\beta}_\mathrm{r}), \Phi) \overset{\sqsubseteq}{\rightsquigarrow} (\mathbf{emp}, \mathbf{emp}, \Phi)}\ \mathbf{i-ind}$$

$$\frac{\beta_\mathrm{r} \text{ is fresh} \qquad \Phi(\alpha_\mathrm{r}) = \alpha_\mathrm{l} \qquad \Phi' = \Phi \uplus \{\beta_\mathrm{r} \mapsto \beta_\mathrm{l}\}}{(\alpha_\mathrm{l} \cdot \iota \mathrel{*\!\!=} \beta_\mathrm{l} \cdot \iota * S_\mathrm{l}, \alpha_\mathrm{r} \cdot \iota, \Phi) \overset{\sqsubseteq}{\rightsquigarrow} (S_\mathrm{l}, \beta_\mathrm{r} \cdot \iota, \Phi)}\ \mathbf{i-segind}$$

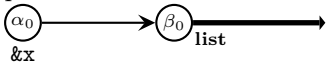
$$\frac{\beta_\mathrm{r} \text{ is fresh} \qquad \Phi(\alpha_\mathrm{r}) = \alpha_\mathrm{l} \qquad \Phi' = \Phi \uplus \{\beta_\mathrm{r} \mapsto \beta_\mathrm{l}\}}{(\alpha_\mathrm{l} \cdot \iota \mathrel{*\!\!=} \beta_\mathrm{l} \cdot \iota * S_\mathrm{l}, \alpha_\mathrm{r} \cdot \iota \mathrel{*\!\!=} \gamma_\mathrm{r} \cdot \iota, \Phi) \overset{\sqsubseteq}{\rightsquigarrow} (S_\mathrm{l}, \beta_\mathrm{r} \cdot \iota \mathrel{*\!\!=} \gamma_\mathrm{r} \cdot \iota, \Phi')}\ \mathbf{i-segseg}$$

$$\frac{e \text{ is an inductive or a segment} \qquad e \xrightarrow{\mathcal{U}} (S_\mathrm{r}, F) \qquad \mathfrak{prove}_\mathrm{N}(N_\mathrm{l}, F \circ \Phi) = \mathbf{true}}{(S_\mathrm{l}, e, \Phi) \overset{\sqsubseteq}{\rightsquigarrow} (S_\mathrm{l}, S_\mathrm{r}, \Phi)}\ \mathbf{i}\text{–}\mathcal{U}$$

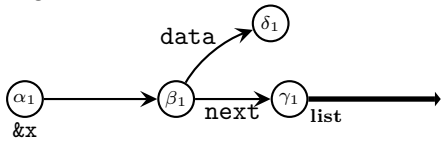
Figure 3.1: Rules for checking inclusion

A simple example: We consider a simple example of a program that adds a random number of elements to a list structure.

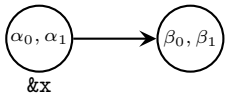
- we assume that the program starts with the abstract pre-condition:



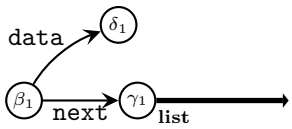
- then, after one iteration, we get the same structure with one element added at the beginning:



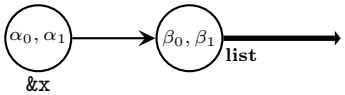
As for the inclusion checking, we need to maintain a pairing between nodes, but we actually have to relate each such mapping to a node in the result abstract value that is being built; for the sake of clarity we will simply let pairs denote names of nodes in the result instead. Since α_0 (resp., α_1) represents `&x`, it makes sense to pair these two nodes together. We remark that both graphs contain a points-to edge starting from these nodes, so it is a reasonable choice to pair these two edges, and their destinations as well. At this stage, we get the (fragment of) join result below:



At this stage, the input graphs differ. Yet, the first argument contains an inductive edge from (β_0, β_1); moreover the inclusion checking algorithm can prove that $\beta_1 \cdot \mathbf{list}$ over-approximates the remaining part of the second argument:



This means that we can over-approximate both regions with inductive predicate $(\beta_0, \beta_1) \cdot \mathbf{list}$, and we get the join result:



That result is equal to the first argument (up-to renaming), which means that this is actually a fixpoint.

Shape join algorithm and rules: The steps of the join algorithm are very similar to those of the inclusion checking (section 3.4.2), thus we only give the main definitions and refer the reader to [20] for details.

After initializing an initial node mapping using the abstract environments, the join algorithm will perform a series of rewriting steps using configurations of the form $(S_\mathrm{l}, S_\mathrm{r}, S_\mathrm{out}, \Psi)$, where:

- S_l (resp., S_r) is the leftover of the left (resp., right) argument that needs to be over-approximated;
- S_out over-approximates the parts of both inputs that were treated already;
- $\Psi : \mathbb{V}^\sharp \longrightarrow \mathbb{V}^\sharp \times \mathbb{V}^\sharp$ maps nodes of S_out into pairs of nodes of both inputs that the over-approximate (in the following, we let $\Psi_\mathrm{l}, \Psi_\mathrm{r} : \mathbb{V}^\sharp \longrightarrow \mathbb{V}^\sharp$ be defined by $\Psi_\mathrm{l}(\alpha) = \alpha_\mathrm{l}$ and $\Psi_\mathrm{r}(\alpha) = \alpha_\mathrm{r}$ if $\Psi(\alpha) = (\alpha_\mathrm{l}, \alpha_\mathrm{r})$).

Numerical parts $N_\mathrm{l}, N_\mathrm{r}$ do not get modified in this process (thus they are omitted in the configurations). The algorithm proceeds by a sequence of rewriting steps on such configurations, using rewrite relation $\overset{\sqcup}{\rightsquigarrow}$, which is defined by a symmetric set of rules. The most salient rewriting rules of $\overset{\sqcup}{\rightsquigarrow}$ are shown in figure 3.2 (each rule is presented only in one side).

Rule ($\sqcup - *$) (based on the separation principle) expresses that local rewriting is sound, and allows to define the other rules *locally*. Rule ($\sqcup$−**pt**) over-approximates a pair of matching points-to edges and extends the node mapping accordingly. Rule ($\sqcup$−**ind**) (resp., rule ($\sqcup$−**ext**)) over-approximates an inductive predicate (resp., segment) and a set of edges which can be proved included in a predicate of the same form by inclusion checking (section 3.4.2), with another predicate of the same form. Rule ($\sqcup$−**ext**) allows to *introduce* a segment, corresponding to an *empty* segment on one side after checking the inclusion of a region in a segment predicate in the other side (section 3.4.2). The soundness of each step of the join algorithm is based on the property below:

$$\begin{array}{l}(S_\mathrm{l}, S_\mathrm{r}, S_\mathrm{out}, \Psi) \overset{\sqcup}{\rightsquigarrow} (S'_\mathrm{l}, S'_\mathrm{r}, S_\mathrm{out}, \Psi') \\ \quad\Longrightarrow \left\{ \begin{array}{l} \forall \mathrm{i} \in \{\mathrm{l}, \mathrm{r}\}, \\ \quad \gamma_{\mathbb{M}}(S_\mathrm{i} * \Psi_\mathrm{i}(S_\mathrm{out}), N_\mathrm{i}) \\ \qquad \subseteq \gamma_{\mathbb{M}}(S'_\mathrm{i} * \Psi'_\mathrm{i}(S'_\mathrm{out}), N_\mathrm{i}) \end{array} \right. \end{array} \tag{3.2}$$

This means that each rewriting step will only weaken both arguments. The join algorithm starts from a configuration of the form $(S_\mathrm{l}, S_\mathrm{r}, \mathbf{emp}, \Psi)$, where Ψ is computed from the abstract environments. It succeeds when it reaches a configuration of the form $(\mathbf{emp}, \mathbf{emp}, S_\mathrm{out}, \Psi')$, that is, when all edges of both arguments were removed and over-approximated in the the result S_out using a rewriting rule. In case of success, the shape join is defined by $S_\mathrm{l} \sqcup_\mathbf{S} S_\mathrm{r} = S_\mathrm{out}$.

After reaching such a configuration, numerical abstract values $N_\mathrm{l}, N_\mathrm{r}$ can be over-approximated by applying a sound abstract join operator $\sqcup_\mathbf{N}$, after renaming the symbolic nodes using the final mapping Ψ'. Then, $(S_\mathrm{l}, N_\mathrm{l}) \sqcup_{\mathbb{M}} (S_\mathrm{r}, N_\mathrm{r}) = (S_\mathrm{out}, N_\mathrm{out})$, and $\sqcup_{\mathbb{S}^\sharp} : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \longrightarrow \mathbb{S}^\sharp$ is defined similarly (by adding the abstract environment).

Soundness: The proof of soundness of each rule follows from the definition of the concretization of figure 2.2(c). By combining the property of equation 3.2, we can prove that the join algorithm is sound, in the sense that it produces a result which over-approximates both inputs:

Theorem 4. Soundness of the shape join.

Let us assume that the shape join algorithm succeeds. Then:

$$\left\{ \begin{array}{l} \gamma_{\mathbb{M}}(S_\mathrm{l}, N_\mathrm{l}) \subseteq \gamma_{\mathbb{M}}(S_\mathrm{out}, N_\mathrm{out}) \\ \gamma_{\mathbb{M}}(S_\mathrm{r}, N_\mathrm{r}) \subseteq \gamma_{\mathbb{M}}(S_\mathrm{out}, N_\mathrm{out}) \end{array} \right.$$

Widening: So far, we focused on the soundness of the join operator and did not discuss how to enforce termination of sequences of iterates.

A first important result is that sequences of joins over shape abstract values terminate:

Theorem 5. Termination.

Operator $\sqcup_\mathbf{S}$ is a widening on $\mathbb{D}^\sharp_\mathrm{S}$.

The proof is based on the following arguments:

- the number of points-to edges is decreasing, and after finitely many iterations points-to edges are stable; moreover, after that point the set of nodes in the graph is also stable;
- once the points-to edges are stable, only rule ($\sqcup$−**intro**) may introduce new edges, however this rule may not be applied more than a fixed number of times (equal to the number of node pairs in the graph);
- when rule ($\sqcup$−**intro**) does not apply anymore, the number of edges may only decrease.

$$\frac{(S_\mathrm{l}, S_\mathrm{r}, S_\mathrm{out}, \Psi) \overset{\sqcup}{\rightsquigarrow} (S'_\mathrm{l}, S'_\mathrm{r}, S'_\mathrm{out}, \Psi)}{(S_\mathrm{l} * \mathring{s}_\mathrm{l}, S_\mathrm{r} * \mathring{s}_\mathrm{r}, S_\mathrm{out}, \Psi) \overset{\sqcup}{\rightsquigarrow} (S'_\mathrm{l} * \mathring{s}_\mathrm{l}, S'_\mathrm{r} * \mathring{s}_\mathrm{r}, S'_\mathrm{out}, \Psi)} \sqcup\!-\!*$$

$$\frac{\Psi(\alpha_\mathrm{out}) = (\alpha_\mathrm{l}, \alpha_\mathrm{r}) \qquad \Psi' = \Psi \uplus \{\beta_\mathrm{out} \mapsto (\beta_\mathrm{l}, \beta_\mathrm{r})\}}{(\alpha_\mathrm{l} \cdot \mathtt{f} \overset{s}{\mapsto} \beta_\mathrm{l}, \alpha_\mathrm{r} \cdot \mathtt{f} \overset{s}{\mapsto} \beta_\mathrm{r}, \mathbf{emp}, \Psi) \overset{\sqcup}{\rightsquigarrow} (\mathbf{emp}, \mathbf{emp}, \alpha_\mathrm{out} \cdot \mathtt{f} \overset{s}{\mapsto} \beta_\mathrm{out}, \Psi')} \sqcup\!-\!\mathbf{pt}$$

$$\frac{\Psi(\alpha_\mathrm{out}) = (\alpha_\mathrm{l}, \alpha_\mathrm{r}) \qquad (S_\mathrm{r}, N_\mathrm{r}) \sqsubseteq_\Phi \alpha_\mathrm{r} \cdot \iota \qquad \text{where } \Phi = \{\alpha_\mathrm{r} \mapsto \alpha_\mathrm{r}\}}{(\alpha_\mathrm{l} \cdot \iota, S_\mathrm{r}, \mathbf{emp}, \Psi) \overset{\sqcup}{\rightsquigarrow} (\mathbf{emp}, \mathbf{emp}, \alpha_\mathrm{out} \cdot \iota, \Psi)} \sqcup\!-\!\mathbf{ind}$$

$$\frac{\Psi(\alpha_\mathrm{out}) = (\alpha_\mathrm{l}, \alpha_\mathrm{r}) \quad \Psi(\beta_\mathrm{out}) = (\beta_\mathrm{l}, \beta_\mathrm{r}) \quad (S_\mathrm{r}, N_1) \sqsubseteq_\Phi \alpha_\mathrm{r} \cdot \iota \mathrel{*\!\!=} \beta_\mathrm{r} \cdot \iota \quad \text{with } \Phi = \{\alpha_\mathrm{r} \mapsto \alpha_\mathrm{r}, \beta_\mathrm{r} \mapsto \beta_\mathrm{r}\}}{(\mathbf{emp}, S_\mathrm{r}, \mathbf{emp}, \Psi) \overset{\sqcup}{\rightsquigarrow} (\mathbf{emp}, \mathbf{emp}, \alpha_\mathrm{out} \cdot \iota \mathrel{*\!\!=} \beta_\mathrm{out} \cdot \iota, \Psi)} \sqcup\!-\!\mathbf{intro}$$

$$\frac{\Psi(\alpha_\mathrm{out}) = (\alpha_\mathrm{l}, \beta_\mathrm{l}) \quad \Psi(\beta_\mathrm{out}) = (\beta_\mathrm{l}, \beta_\mathrm{r}) \quad (S_\mathrm{r}, N_\mathrm{r}) \sqsubseteq_\Phi \alpha_\mathrm{r} \cdot \iota \mathrel{*\!\!=} \beta_\mathrm{r} \cdot \iota \quad \text{with } \Phi = \{\alpha_\mathrm{r} \mapsto \alpha_\mathrm{r}, \beta_\mathrm{r} \mapsto \beta_\mathrm{r}\}}{(\alpha_\mathrm{l} \cdot \iota \mathrel{*\!\!=} \beta_\mathrm{l} \cdot \iota, S_\mathrm{r}, \mathbf{emp}, \Psi) \overset{\sqcup}{\rightsquigarrow} (\mathbf{emp}, \mathbf{emp}, \alpha_\mathrm{out} \cdot \iota \mathrel{*\!\!=} \beta_\mathrm{out} \cdot \iota, \Psi)} \sqcup\!-\!\mathbf{ext}$$

Figure 3.2: Rules for the computation of a shape join

A consequence of this result is that, if we apply a widening operator $\triangledown_\mathbf{N}$ inside $\mathbb{D}^\sharp_\mathrm{N}$ instead of $\sqcup_\mathbf{N}$, we obtain a widening $\triangledown$ operator over $\mathbb{S}^\sharp$:

Theorem 6. Widening.

Operator $\triangledown_{\mathbb{S}^\sharp}$ obtained by combining $\sqcup_\mathbf{S}$ and $\triangledown_\mathbf{N}$ is a widening operator on $\mathbb{S}^\sharp$.

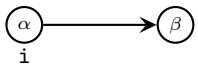
Intuitively, we can prove the convergence of any sequence of iterates $(\hat{\epsilon}^\sharp_n, S_n, N_n)_{n\in\mathbb{N}}$ computed using $\triangledown_{\mathbb{S}^\sharp}$ as follows:

- since $\triangledown_\mathbf{S}$ is a widening operator (theorem 5), the iteration sequence on graphs $(S_n)_{n\in\mathbb{N}}$ eventually stabilizes; let us call n_0 the rank at which it becomes stable;
- then, for all n greater than n_0, $\hat{\epsilon}^\sharp_n = \hat{\epsilon}^\sharp_{n_0}$ and $S_n = S_{n_0}$ (up to renaming), which means only the numerical abstract value may continue to evolve; however, since $\triangledown_\mathbf{N}$ is a widening operator over $\mathbb{D}^\sharp_\mathrm{N}$, $(N_n)_{n\in\mathbb{N},\, n\geq n_0}$ eventually stabilizes at some rank n_1.

Thus, sequence $(\hat{\epsilon}^\sharp_n, S_n, N_n)_{n\in\mathbb{N}}$ is stable from rank n_1. This proof corresponds to an instance of the widening of the cofibered domain [119]: the iteration sequence over elements of $\mathbb{S}^\sharp$ first stabilizes in the lattice of graphs, and then all further iterates will use the same numerical lattice, so it will converge there too.

Examples: In the example of section 3.1, the sequence of widening iterates converges after two iterations, and compute the invariant shown at the end of that section

Let us consider a numeric example, where one integer variable `i` starts at 0 and gets incremented indefinitely. The shape part is stable, and corresponds to the graph below:



Thus, for all iterates, numerical abstract values consist in constraints over symbolic nodes α, β. Assuming $\mathbb{D}^\sharp_\mathrm{N}$ is the domain of intervals, at iteration 0, $N_0(\beta) = [0, 0]$, and at iteration 1, $N_1 = N_0 \triangledown_\mathbf{N} [1, 1] = [0, +\infty[$. Thus, we observe that the analysis of a program which performs only numerical computations is similar to what would be obtained with a standard static analysis such as Astrée [11].

Several complex examples mixing shape and numerical information (list of length n, binary search tree, red-black trees) are shown in [20].

3.4.4 Other global abstraction mechanisms

Other flavors of global abstraction operators for widening can be found in the literature.

First, we note that our folding rules are derived from separation logic and make fast inclusion checking and join algorithms possible (despite the cost of

graph traversal algorithms) compared to abstract domains made of unstructured, quantified logical formulas [54].

Canonicalization: In particular, a canonicalization [109, 44] is an upper closure operator $\mathfrak{can} : \mathbb{D}^\sharp_S \longrightarrow \mathbb{D}^\sharp_S$, such that $\mathfrak{can}(\mathbb{D}^\sharp_S)$ is a lattice of finite height. TVLA relies on a canonicalization operator [109] which "blurs" logical structures, according to some pre-defined sets of predicates. Other canonicalization (as in SpaceInvader [44]) operators use sets of rewriting rules, the proof of which is based on the separation principle, in the same way as for the inclusion checking rules of figure 3.1 of the join rules of figure 3.2. Applying such an operator to the power-set completion of $\mathbb{D}^\sharp_S$ also allows to enforce termination of shape analyses. However, it is harder to extend to product domains, e.g., with a numerical domain with infinite height (in that case, a combination with a widening is required, yet is harder to design and implement).

Comparing widening and canonicalization approaches: In theory, widening operators [32] are more powerful than analyses based on finite height abstraction, mainly due to the fact the widening iterates are not fixed in advance, and the analysis may generate very different widening chains depending on the program, as they are history guided. Furthermore, widening operators fit particularly well with standard static analyzers design, and lots of results have been found on how to apply them efficiently [31, 11].

Yet, in practice, canonicalization operators are also relevant. Indeed, when too many unfolding operations are performed, abstract values that consist in very large disjunctions may arise, which may prevent the analysis from scaling to very large blocks of code with no widening point. In the precision point of view, both rewriting rule canonicalization operators and our widening operators are determined by the set of folding rules.

As of now, Xisa [20] relies solely on widening and this turned out well for the analysis of medium sized examples, yet we consider the addition of a canonicalization operator is necessary in the long term (section 6.3).

Comparing combined analyses and separate analyses: Our analysis infers both shape and numerical properties in a *single* analysis phase. In the other hand, other authors proposed [76] to split the process in two analyses. However, it is well known that combining both shape and numerical analyses into one single abstract interpretation using reduced product [29] or cofibered domains [119] usually yields more precise results as one domain may benefit from information in the other and vice versa. We indeed observed this when considering data structure which induce relations between shape and numerical predicates, in both directions [20]. This effect can be seen in folding rules that rely on the checking of numerical conditions as in rule ($\mathbf{i}$–$\mathcal{U}$) or indirectly in rule ($\sqcup$–**ext**).

3.5 A domain signature

Based on the abstraction of chapter 2, we have built an abstract domain for the abstraction of sets of memory states, with transfer functions, inclusion test, join and widening, that is now suitable for the static analysis of imperative programs.

It appears that this domain takes a numerical domain $\mathbb{D}^\sharp_N$ as a parameter and its transfer functions and folding operators all refer to those of $\mathbb{D}^\sharp_N$. Therefore, our abstract domain appears as a *functor*, taking a numerical domain as a parameter, and returning a domain for the abstraction of sets of memory states. Indeed, figure 3.3 collects the definitions of the elements and functions that $\mathbb{D}^\sharp_N$ was required to provide (each of them should meet usual assumptions of soundness and termination for widening). Given such a numerical abstract domain, we have provided transfer functions for abstract domain $\mathbb{S}^\sharp$, the signature of which is shown in figure 3.4 which also enjoy usual soundness and termination properties (for the sake of concision, we do not recall these properties here, as they are either classic or were given in previous sections).

We can remark that the signatures of $\mathbb{D}^\sharp_N$ and $\mathbb{S}^\sharp$ are not similar. In particular, $\mathbb{D}^\sharp_N$ was required to provide a function $\mathfrak{prove}_N$, to attempt to prove that a numerical abstract value entails some numeric predicate is true. Furthermore, $\mathbb{D}^\sharp_N$ handles expressions of symbolic nodes whereas $\mathbb{S}^\sharp$ handles program expressions (for assignment and condition test transfer functions). Besides, $\mathbb{S}^\sharp$ also provides an $\mathfrak{unfold}$ operation which takes an abstract value and an l-value to materialize (section 3.3). In the other hand, they both provide inclusion check, join

and widening operators with very similar signatures, even though these are based on radically different algorithms.

least element
$\bot \in \mathbb{D}^\sharp_N$

assignment operator
$\mathfrak{assign}_N : \mathbb{D}^\sharp_N \times \text{Lvals}_{\mathbb{V}^\sharp} \times \text{Exprs}_{\mathbb{V}^\sharp} \longrightarrow \mathbb{D}^\sharp_N$

condition test transfer function
$\mathfrak{guard}_N : \mathbb{D}^\sharp_N \times \text{Exprs}_{\mathbb{V}^\sharp} \times \mathbb{B} \longrightarrow \mathbb{D}^\sharp_N$

conservative condition verification
$\mathfrak{prove}_N : \mathbb{D}^\sharp_N \times \text{Exprs}_{\mathbb{V}^\sharp} \longrightarrow \mathbb{B}$

inclusion test
$\sqsubseteq_{\mathbf{N}} : \mathbb{D}^\sharp_N \times \mathbb{D}^\sharp_N \longrightarrow \mathbb{B}$

conservative abstract join
$\sqcup_{\mathbf{N}} : \mathbb{D}^\sharp_N \times \mathbb{D}^\sharp_N \longrightarrow \mathbb{D}^\sharp_N$

widening operator
$\nabla_{\mathbf{N}} : \mathbb{D}^\sharp_N \times \mathbb{D}^\sharp_N \longrightarrow \mathbb{D}^\sharp_N$

Figure 3.3: Numerical domain

assignment operator
$\mathfrak{assign}_{\mathbb{S}^\sharp} : \mathbb{S}^\sharp \times \text{Lvals}_{\mathbb{X}} \times \text{Exprs}_{\mathbb{X}} \longrightarrow \mathbb{S}^\sharp$

condition test transfer function
$\mathfrak{guard}_{\mathbb{S}^\sharp} : \mathbb{S}^\sharp \times \text{Exprs}_{\mathbb{X}} \times \mathbb{B} \longrightarrow \mathbb{S}^\sharp$

unfolding operator
$\mathfrak{unfold} : \mathbb{S}^\sharp \times \text{Lvals}_{\mathbb{X}} \longrightarrow \bigvee \mathbb{S}^\sharp$

inclusion test
$\sqsubseteq_{\mathbb{S}^\sharp} : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \longrightarrow \mathbb{B}$

conservative abstract join
$\sqcup_{\mathbb{S}^\sharp} : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \longrightarrow \mathbb{S}^\sharp$

widening operator
$\nabla_{\mathbb{S}^\sharp} : \mathbb{S}^\sharp \times \mathbb{S}^\sharp \longrightarrow \mathbb{S}^\sharp$

Figure 3.4: Shape domain

3.6 Implementation

We implemented the Xisa shape analyzer, an academic prototype, based on the abstract domain of chapter 2 and the abstract interpretation based analysis described in this chapter. The table below displays an excerpt of the results [21, 20] obtained when applying this analyzer to micro-benchmarks and medium size codes, including examples of classical structures (singly-linked lists, binary search trees), more complex structures (doubly-linked list, binary search trees with parent pointers) and a simple device driver example. Columns respectively list numbers of lines (in LOCs), analysis runtime (in milliseconds), peak number of disjuncts and total numbers of iterations (including iterations in nested loops, if applicable).

These experiments shows the analysis generates rather low number of disjuncts and converge in relatively low numbers of iterates. The higher number of iterates observed for the device driver example takes into account several nested loops, so in fact, abstract iterations for each loop converge fast.

Example	size	time	peak	iters
list reverse	19	7	1	3
list remove element	27	16	4	6
list insertion sort	56	21	4	7
dll copy	50	53	2	3
dll insert	40	38	2	4
binary search tree find	23	10	2	4
binary search tree insert	150	83	5	5
scull driver	894	9 710	4	16

Chapter 4

Analysis of low-level C programs

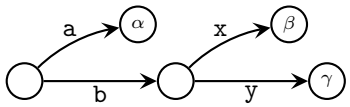
So far, we did not consider a full featured programming language and all the examples considered in chapter 2 and chapter 3 were either written in an unspecified imperative language, which could be seen as a subset of Java or C. However, in practice, we intend to analyze a real programming language and handle most of its features, including low level memory manipulations. Many authors proposed very precise encodings of such low level features so as to run symbolic model checking [25, 120] or verification [23, 26] tools. Yet, few abstractions were proposed for such programs. Antoine Miné [88] designed a very powerful analysis yet which does not handle unbounded structures. Moreover, [17] proposed an analysis for lists with elements of non fixed length.

In this chapter, we show that the abstract domain that was presented in chapter 2 can be adapted to the analysis of ANSI C 99 [4] programs, and that the static analysis algorithms of chapter 3 need only minimal adaptations. The specialization of our shape analysis framework which are presented in this chapter were designed during the Master Internship of Vincent Laviron (March 2009 till September 2009) and were published in [71]; in particular, this work focuses on the analysis of a program that evaluates and simplifies arithmetic expressions represented by structures with nested unions and structures. Furthermore, we proposed a classification for pointer models in [117].

Section 4.1 overviews the main issues that arise when analyzing C programs, including dependencies on the implementation. Section 4.2 extends the handling of contiguous regions. Section 4.3 shows how multiple views on regions can be treated in our framework. Last, section 4.4 focuses on issues related to memory management and section 4.5 assesses implementation results.

4.1 Overview of specific issues related to C programs

In the previous chapters, points-to edges always represented memory cells corresponding to structure fields, and storing either base type values, or pointers to the base address of other structures. For instance, the graph below describes a structure with an integer field and a pointer field, which points to another structure, with two integer fields (in this graph, only nodes corresponding to a value of a base type are labeled):



Such a structure corresponds to an instance of the type definition:

```
typedef struct {
    int a;
    tt_1 * b;
} tt_0
typedef struct {
    int x;
    int y;
} tt_1;
```

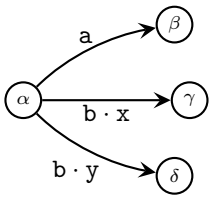
Nested structures: However, when **tt**_1 is never used out independently, it is also common to define a *nested structure*, where fields `x` and `y` are inside object **tt**_0:

```
typedef struct {
    int a;
    struct {
        int x;
        int y;
    } b;
} tt
```

The picture below shows an excerpt of a concrete state containing an instance of such a structure:

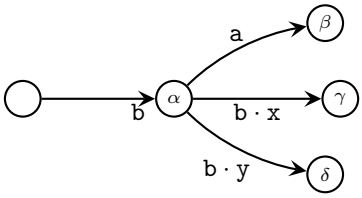
a	64
b · x	112
b · y	24

The graph corresponding to the instance of **tt**_0 does not describe **tt**; instead, all points-to edges should start from the same base address, and should correspond respectively to the offsets of fields a, b·x and b · y (i.e., β denotes 64, γ denotes 112 and δ denotes 24):



In this example, we considered a case where all fields have type **int**, i.e. have the same size, and will be aligned on a 4 bytes boundary when using a 32-bits machine. However, when the types are not equal, *alignment* rules apply and may require padding bytes be added between fields, which are not captured in the graph abstract elements we have been using up to now.

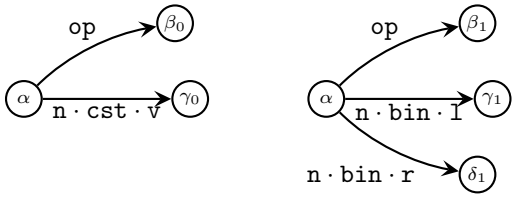
Pointers to fields: It is also common practice to use pointers not only to heap allocated objects, but also to *fields* of objects. For instance, if x points to a **tt** heap allocated object, &(x->b) is a pointer to the inner structure. Hence, the value of such a pointer cannot be denoted by node α; instead, it should be described by α plus some offset corresponding to field b. This means we should also augment points-to edges with offsets at the destination site:



Multiple views: The C language allows to maintain several views on a memory region and to switch from one to another. One way of doing so is pointer casts; another way is unions. The C standard [4] does not provide a strong guarantee about the accesses using another view than the one used to create and fill the fields of a structure, but many implementations do, so that it is possible to assume that a write using an alternate view will not erase all the properties known about the contents of the structure. Let us consider the type definition below, which corresponds to a typical abstract syntax tree, with several cases:

```
typedef struct arith {
    char op;
    union {
        struct{double v;}cst
        struct{
            struct arith * l;
            struct arith * r;
        } bin;
    } n;
} arith;
```

The two views induced by this definition can be respectively abstracted by the abstract elements below, assuming a 32-bits architecture is used (note that most implementations would require some padding bytes be added due to alignment constraints; we ignore those here):



However, a limitation of our current domain is that it cannot maintain both views in the same time. In particular, these two graphs do not make sense in the same time, due to separation. Indeed, separation requires all points-to edges to denote *disjoint* memory regions, yet, in this case, $\alpha \cdot \mathtt{n} \cdot \mathtt{bin} \cdot \mathtt{l} \mapsto \gamma_0$ and $\alpha \cdot \mathtt{n} \cdot \mathtt{cst} \cdot \mathtt{v} \mapsto \gamma_1$ correspond to the same physical memory cells.

Field level arithmetics: Pointer casts allow to view a structure like an array, and address its fields using integer indexes. Again, the behavior of programs using such techniques is usually implementation dependent (with exact field sizes and alignment rules defined in *Application Binary Interfaces*, or ABI). Analyzing such programs requires going beyond the symbolic names used to denote fields, and precisely representing the integer values they correspond to, according to a precisely known ABI. Therefore, points-to edges should be labeled using either symbolic offsets or integer offsets, depending on the intended application:

- *portable code* can usually be analyzed using symbolic field names;
- *non portable code* may require the use of numeric offsets in order to annotate points-to edges.

Memory management: A last important feature of C is manual memory management. This means the programmers are responsible for the allocation (using **malloc**) and the deallocation (using **free**) of all the cells programs manipulate. Memory management is actually the source of many bugs and errors:

- the failure to deallocate a region before all pointers to it are discarded would result in a *memory leak*;
- the attempt to free a pointer that does not correspond to the base address of a valid allocated region would cause an *immediate crash*;
- the use (for memory read or write operations) pointers into a region that was freed would result in a *dangling pointer dereference* (and thus an abrupt crash).

In the static analysis point of view, it means that the allocation and deallocation of memory regions should be tracked precisely. In particular, when a region is deallocated, the analysis should precisely remove all points-to edges corresponding to that region.

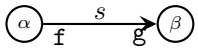
4.2 Abstraction of contiguous regions

As we remarked in section 4.1, refining the handling of contiguous memory regions (i.e., points-to edges) brings a solution for several C specific issues.

4.2.1 Fields, offsets and pointers

Points-to edges were introduced in section 2.2.3, and were annotated with a starting node, a field name (corresponding to an offset) and an optional type or size annotation.

Pointers to fields: Points-to edges with a target node representing the sum of a base address and of an offset may represent pointers to fields, yet this representation would not be very efficient as it would require numerical relations among nodes to express, e.g., that several edges point to fields of the same structure. Instead, it makes more sense to refer to the node which represents the base address. Thus we simply extend the syntax of points-to edges with another label at the edge destination, which denotes the destination field; for instance if a pointer value contained in a memory cell can be decomposed into the sum of a base address represented by β and an offset corresponding to field `g`, we would get:



Such an edge would be noted:

$$\alpha \cdot \mathtt{f} \mapsto \beta \cdot \mathtt{g}$$

Moreover, the concretization of this edge is defined in a very similar way as that of regular points-to edges. With the same notations as in section 2.2.3, $\gamma_S(\alpha \cdot \mathtt{f} \overset{s}{\mapsto} \beta \cdot \mathtt{g})$ is the set of pairs (σ, ν) such that σ is a memory state with just one cell, of address $x = \nu(\alpha) + \mathbf{offset}(\mathtt{f})$, of size s, and of content $v = \nu(\beta) + \mathbf{offset}(\mathtt{g})$.

Revised offsets: In section 4.1, we also remarked that field names are not enough to express all offsets the analysis may need to handle, especially in the case of nested fields. Therefore, we replace fields with a grammar of symbolic offsets:

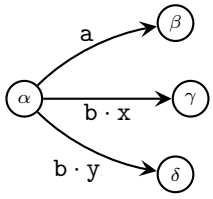
$$\begin{array}{rcll} \mathtt{o} & ::= & \epsilon & \text{null offset} \\ & | & \mathtt{o} \cdot \mathtt{f} & \text{field dereference} \end{array}$$

Function **offset** can be extended to this set of symbolic offsets in a straightforward way, by induction over the syntax:

$$\begin{array}{rcl} \mathbf{offset}(\epsilon) & = & 0 \\ \mathbf{offset}(\mathtt{o} \cdot \mathtt{f}) & = & \mathbf{offset}(\mathtt{o}) + \mathbf{offset}(\mathtt{f}) \end{array}$$

Thus, extending the syntax of points-to edges with this more general notion of offsets preserves the notion of concretization as above. Hence, the static analysis algorithms (chapter 3) also remain unchanged.

In this model, the abstract element below takes the intuitive meaning described in section 4.1 and describes set of fields of a nested structure:



4.2.2 Pointer arithmetics

Some programmers rely on machine dependent pointer arithmetics, where the integer values of offsets need be taken into account. To handle such programs in our framework, we simply need to turn symbolic offsets into numerical values, and to rely on function **offset** in order to perform all operation on points-to edges using integer values. Then, the analysis should treat statements of the form:

$$\alpha \cdot 4 \overset{4}{\mapsto} \beta \cdot 12$$

The translation of all symbolic offsets into numeric values is based on alignments and size assumptions typically defined in the ABI.

Analysis Algorithms: When using either extended symbolic offsets or numerical offsets, static analysis algorithms remain similar to those defined in chapter 3. In practice, it is best to use numeric offsets to check a folding rule apply, while preserving symbolic field names, that may help finding out which inductive definition to fold (i.e., to decide which folding rule to try to apply).

Pointer models: In [117], we formalized and compared pointer models used in shape analysis. In our classification, we found four models which are derived by making two independent choices:

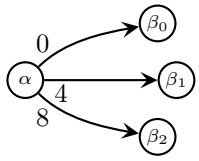
- whether pointers to structure fields are allowed or not;
- whether offsets are considered numeric values, or symbolic names.

Our abstract domain may handle any of these four models. Abstract domain of chapter 2 relies on symbolic offsets and forbids pointers to fields. The extension proposed in section 4.2.1 allows pointers to fields, whereas that proposed above allows to treat numeric offsets. In all cases, the algorithms remain similar.

Shape analyses can be classified along these categories. For instance, Kreiker [69] proposed several encodings to extend TVLA with pointer to fields, whereas initial TVLA do not handle them [111, 73].

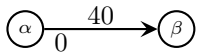
4.2.3 Contiguous regions and arrays

A common programming languages formalization trick consists in viewing arrays as particular instances of structures where fields are indexes. Our abstract domain is compatible with this view. Indeed, we can describe an array of fixed known length (3 in the example below) with a base address and one edge per cell:



Note that the indexes used in this graph do not correspond to indexes of array cells, but to offsets in the array regions, which are obtained by multiplying the array indexes by the size of an element of the array. This representation is equivalent to the fully expanded array abstraction (featured, e.g., in Astrée [11]).

As we saw in section 2.2.3, points-to edges may represent any contiguous sequence of bits in memory. In particular, a points-to edge can be used in order to describe an array, using a single block. Let us consider an array of integers, of size 10. Assuming a 32 bits architecture, the representation of an element takes 4 bytes, thus the whole array is a contiguous block of 40 bytes:

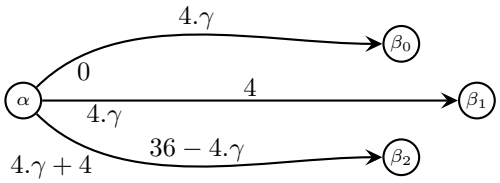


In this view, node β represents a sequence of 40 bytes, but it does not say anything precise about the values contained in the array. However, $\mathbb{D}^\sharp_N$ views it as one cell of 40 bytes, and may be used in order to express various properties about it:

- it may abstract the values contained in each cell of the array with one set of integer values and over-approximate that set, using the smashing approach used for large arrays in Astrée [11];
- it may rely on a fixed partitioning of the array region, which may coincide with the cells of the array or not; for instance, it may use 5 abstract cells, such that abstract cell k over-approximates cells k and $5 + k$ in the initial array of size 10;
- $\mathbb{D}^\sharp_N$ may include a more general abstraction of arrays, based e.g., on array partitioning so as to abstract array regions with various predicates, using either static partitions [57] or dynamic partitions [38].

In the smashing approach, the shape domain can still refine the 40 bytes thick points-to edge. For

instance, if the value γ of variable `i` is known to be in range $[0, 9]$, we may access the cell of index `i`, and it is even possible to materialize it, by *splitting* the 40 bytes thick points-to edge into:



This edge splitting is sound, and allows to turn an update of the cell of index `i` into a strong update even though the array cells are all abstracted into one. This principle is actually at the core of the static analyses based on array partitions [57, 53, 38]. Analyses such as [38] can dynamically modify or refine the splitting or array regions.

4.3 Abstraction of multiple views

Section 4.1 brought up the need to maintain *several views* of the same region, in the same time, which is equivalent to using *non separating conjunctions*.

4.3.1 A local conjunction operator

The obvious solution would consist in extending abstract values with non separating conjunctions of sets of edges. However, such an extension would break most of the static analysis algorithms of chapter 3, as it would break separation, which is the basis of folding and of transfer functions for assignments, among others. Therefore, we intend to keep the effect of the conjunction operator as local as possible, i.e., to contiguous regions; this will require us to fix the static analysis algorithms presented in chapter 3, but their overall structure should remain the same. Similarly, [88] also keeps non separating conjunctions local to blocks.

A local conjunction operator: In practice, we should expect several views be needed only for contiguous regions, since unions and pointer casts modify the interpretation of a single region. Therefore, we introduce conjunction edges as an extension of points-to edges. More precisely, a *conjunction multi-edge* is of the form

$$\bigwedge_{0 \leq i < n}^{\alpha \cdot [s_{\text{beg}}, s_{\text{end}}[} \left(\alpha \cdot \mathtt{o}_i \overset{s_i}{\mapsto} \beta_i \right)$$

where points-to edges $\alpha \cdot \mathtt{o}_0 \overset{s_0}{\mapsto} \beta_0, \ldots, \alpha \cdot \mathtt{o}_{n-1} \overset{s_{n-1}}{\mapsto} \beta_{n-1}$

- may overlap;
- all describe memory cells included in the region of base address α plus s_{beg} and of size $s_{\text{end}} - s_{\text{beg}}$.

In other words, the concretization of this conjunction multi-edge is the set of pairs (σ, ν) such that:

$$\begin{array}{l} \mathbf{dom}(\sigma) = \{\nu(\alpha) + s_{\text{beg}}, \nu(\alpha) + s_{\text{beg}} + 1, \ldots, \\ \qquad \nu(\alpha) + s_{\text{beg}} - s_{\text{end}} - 1\} \\ \forall i \in \{0, 1, \ldots, n-1\} \\ \quad \left\{ \begin{array}{ll} \mathbf{read}(\sigma, x_i, s_i) = v_i & \\ \text{where} & x_i = \nu(\alpha_i) + \mathbf{offset}(\mathtt{o}_i) \\ \text{and} & v_i = \nu(\beta_i) \end{array} \right. \end{array}$$

A conjunction multi-edge abstracts a fixed contiguous memory region, using a set of usual points-to constraints over sub-regions. An important note is that a multi-edge is still viewed as a *single* edge in the graph, which means that the graph meaning is still the separating conjunction of the concretization of the edges it is made of (in other words non separating conjunction is purely local).

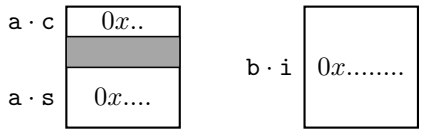
When $n = 0$, the conjunction corresponds to a fixed memory region of size $s_{\text{end}} - s_{\text{beg}}$, which is constrained by no points-to edge; in other words it fixes no constraint about the region. It can be used e.g., to describe un-initialized cells, or freshly allocated memory cells (section 4.4).

When $n = 1$, $\mathtt{o}_0 = s_{\text{beg}}$ and $s_0 = s_{\text{end}} - s_{\text{beg}}$, we get a standard points-to edge.

Unions: We consider the union type below, and rely on a 32-bits implementation which satisfies the assumption that writing through one view will not invalidate the other view:

```
typedef union u {
    struct {
        char c;
        short s;
    } a;
    struct {
        int i;
    } b;
} u;
```

Typical concrete memory states of type **u** would correspond to the two cases below (padding areas are grayed out):



It would make sense to maintain both views if analyzing a code which relies on, e.g., the `a` selector in order to write data in the field and on the `b` selector in order to read sub-sequences of bits included in the whole structure (this is typical in low-level system code and device drivers).

If α represents the base offset of such a structure, an abstract value that would keep both views exposed would be of the form (using numeric offsets):

$$\bigwedge^{\alpha\cdot[0,4[} \begin{cases} \alpha \cdot 0 \overset{1}{\mapsto} \beta_{\mathtt{c}} \\ \alpha \cdot 2 \overset{2}{\mapsto} \beta_{\mathtt{s}} \\ \alpha \cdot 0 \overset{4}{\mapsto} \beta_{\mathtt{i}} \end{cases}$$

Of course, as these edges all refer to the same concrete states, the right hand sides are bound by implicit relations which express that all views be consistent with respect to the concrete sequences of bits. In this case, assuming the architecture is little endian (like Intel architectures), the properties below hold:

$$\begin{cases} \beta_{\mathtt{i}} / 2^{16} = \beta_{\mathtt{s}} \\ \beta_{\mathtt{i}} \bmod 2^8 = \beta_{\mathtt{c}} \end{cases}$$

The need to maintain several views on a memory region may also arise in other cases, such as pointer casts [88]. Other formalisms such as fractional permissions [16, 14] have been proposed so as to express which edges may share a region.

4.3.2 Analysis with local conjunctions

We now extend the analysis operations so as to handle multi-edges.

Extension of inductive definitions: Conjunctions multi-edges generalize points-to edges. As points-to edges were used as part of the syntax of inductive definitions, it appears reasonable to also allow inductive definitions to include such multi-edges. Though, in practice each view corresponds to a specific rule, so that we found that, in practice multi-edges are not of much use there.

Extension of static analysis algorithms: As a graph is still a separating conjunction of edges, the overall structure of the analysis algorithms of chapter 3 remains unchanged. However, of course, new cases need be defined for multi-edges.

The unfolding algorithm does not need to be modified.

The folding algorithms have to be extended with new rules. We consider the case of the inclusion checking (section 3.4.2) as the extension of e.g., the join algorithm (section 3.4.3) would be similar.

Let us consider the pair of multi-edges below:

$$\begin{aligned} e^{\mathrm{l}} &= \textstyle\bigwedge_{0\leq i<n^{\mathrm{l}}}^{\alpha\cdot[s_{\mathrm{beg}},s_{\mathrm{end}}[} \left(\alpha \cdot \mathtt{o}^{\mathrm{l}}_i \overset{s^{\mathrm{l}}_i}{\mapsto} \beta^{\mathrm{l}}_i\right) \\ e^{\mathrm{r}} &= \textstyle\bigwedge_{0\leq i<n^{\mathrm{r}}}^{\alpha\cdot[s_{\mathrm{beg}},s_{\mathrm{end}}[} \left(\alpha \cdot \mathtt{o}^{\mathrm{r}}_i \overset{s^{\mathrm{r}}_i}{\mapsto} \beta^{\mathrm{r}}_i\right) \end{aligned}$$

Note that we partly factor out the naming problem that needed be solved as part of the folding algorithms of section 3.4, by assuming both multi-edges have the same origin α. However, we also assume that both edges have the same physical size (parameters s_{beg}, s_{end} are the same in both edges), which is critical for both edges to describe the same region.

Then, we can prove that $\gamma_{\mathrm{S}}(e^{\mathrm{l}}) \subseteq \gamma_{\mathrm{S}}(e^{\mathrm{r}})$ when the condition below is satisfied:

$$\begin{gathered} \forall i \in \{0,1,\ldots,n^{\mathrm{l}}-1\},\ \exists j \in \{0,1,\ldots,n^{\mathrm{r}}-1\}, \\ \gamma_{\mathrm{S}}(\alpha \cdot \mathtt{o}^{\mathrm{l}}_i \overset{s^{\mathrm{l}}_i}{\mapsto} \beta^{\mathrm{l}}_i) \subseteq \gamma_{\mathrm{S}}(\alpha \cdot \mathtt{o}^{\mathrm{r}}_j \overset{s^{\mathrm{r}}_j}{\mapsto} \beta^{\mathrm{r}}_j) \end{gathered}$$

Thus, the inclusion checking algorithm should do the following in presence of such a pair of edges:

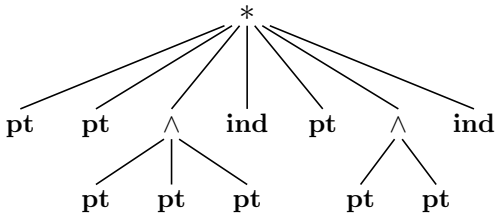
1. check that the above property holds;
2. add to the node pairing relation Φ (section 3.4.2) all pairs of the form $(\beta^{\mathrm{r}}_j, \beta^{\mathrm{l}}_i)$, such that the above property holds.

Local conjunctions: We can remark that this multi-edge rule that we outlined is *quadratic*, since the number of possible pairs is proportional to $n.m$.

In general dealing with multi-edges always incurs such a quadratic cost, however it is *local* to the multi-edge, that is to the fields of the union structure, which makes this quadratic factor a non issue: first, the size of union structures is usually negligible compared to the size of the programs, and it is also usually possible to cut down that quadratic cost by using tries in order to store offsets from a node.

In the other hand, a quadratic complexity factor would be a major problem if it applied to the whole memory abstraction. This is the reason why we chose to keep the non separating conjunction effect local, as a means to keep the cost of analysis operations down.

To summarize, the predicates represented by abstract elements are restricted to a very small subset of a separation logic formulas illustrated in the figure below, where only points-to edges may appear under non-separating conjunction and where separating conjunction always appears at the top of formulas. This restriction turns out to offer an interesting tradeoff in terms of analysis cost and expressivity.



4.4 Memory management

Section 4.1 also brought up the issue of the tracking of memory allocated cells.

Stack cells and heap allocated cells: A node representing the base address of a region allocated by **malloc** should be annotated so that it would be possible to check that a **free** of that node is valid. Furthermore, we should also ensure that the analysis of **free** is conservative, i.e., that it removes *all* edges corresponding to (part of) the deallocated region. In particular, nodes that correspond to variable addresses (i.e., to cells which are inside the stack) should never be deallocated (**free** should fail, if applied to such an address). Last, inductive definitions should enclose information about the allocated cells they summarize, so as to allow the checking that a **free** of cells that were summarized will not crash.

Therefore, we should add a label on each node, so as to express:

- whether it corresponds to a valid address or an uncategorized value;
- when it corresponds to a valid address, whether that address is part of the stack or heap allocated;
- when it is heap allocated, whether it is the base address of an allocated region and of what size that allocated region is.

Thus, we propose to use the type system below:

$$\begin{array}{rcll} \tau & ::= & \textbf{any} & \text{denotes } \top \\ & | & \textbf{stack} & \text{stack cell} \\ & | & \textbf{heap}\langle n\rangle & \text{heap region, of size } n \in \mathbb{N} \end{array}$$

In this type discipline, we implicitly assume that the sizes of all heap regions are known statically; to deal with heap regions of arbitrary size, we would simply need to replace parameter n in $\textbf{heap}\langle n\rangle$ by a symbolic node, that would represent that size.

We remarked that inductive definitions should reflect the fact that some of the nodes they summarize correspond to heap allocated cells. For instance, the **list** inductive definition we gave in section 2.2.4 should now write as follows (we assume a 32-bits architecture is used):

$$\begin{array}{rl} \alpha \cdot \textbf{list} ::= & \\ & (\textbf{emp}, \alpha = 0) \\ \vee & (\alpha \cdot \texttt{next} \mapsto \beta * \alpha \cdot \texttt{data} \mapsto \gamma * \beta \cdot \textbf{list}, \\ & \quad \alpha \neq 0 \wedge \textbf{heap}\langle 8\rangle) \end{array}$$

Partitioning of the heap into chunks and concretization: The concretization γ_{S} of $\mathbb{D}^\sharp_{\mathrm{S}}$ should reflect the meaning of the node types. Intuitively, that denotation can be expressed in terms of separation logic. Indeed, it expresses a two stage partitioning of the memory:

- first, the memory is partitioned into allocated regions such as stack cells or heap regions of a given size,
- secondly, each of these regions is partitioned into edges.

For the sake of clarity, we assume that all node type constraints are represented in an abstract domain $\mathbb{D}^\sharp_{\mathbb{A}}$; thus, we simply need to give the concretization $\gamma_{\mathbb{A}}$ of $\mathbb{D}^\sharp_{\mathbb{A}}$, and the constraints between abstract values in $\mathbb{D}^\sharp_{\mathrm{S}}$ and $\mathbb{D}^\sharp_{\mathbb{A}}$. First an element of $\mathbb{D}^\sharp_{\mathbb{A}}$ is a separating conjunction of node types:

$$\mathbb{D}^\sharp_{\mathbb{A}} = \{\alpha_0 : \tau_0 * \ldots * \alpha_n : \tau_n \mid \alpha_0, \ldots, \alpha_n \in \mathbb{V}^\sharp\}$$

The domain for approximating sets of states now becomes $\mathbb{S}^\sharp = \mathbb{E}^\sharp \times \mathbb{D}^\sharp_{\mathrm{S}} \times \mathbb{D}^\sharp_{\mathbb{A}} \times \mathbb{D}^\sharp_{\mathrm{N}}$. As usual, the concretization relies on the use of valuations to capture the relations between addresses, values, and memory states. The concretization $\gamma_{\mathbb{A}}$ is defined by:

- $\gamma_{\mathbb{A}}(\alpha_0 : \tau_0 * \ldots * \alpha_n : \tau_n) = \{(\sigma_0 \circledast \ldots \circledast \sigma_1, \nu) \mid \forall i,\ (\sigma_i, \nu) \in \gamma_{\mathbb{A}}(\alpha_i : \tau_i)\}$;
- $\gamma_{\mathbb{A}}(\alpha : \mathbf{any}) = \{(\emptyset, \nu) \mid \nu \in \mathbb{V}\mathfrak{al}\}$;
- $\gamma_{\mathbb{A}}(\alpha : \mathbf{heap}\langle n \rangle)$ is the set of pairs (σ, ν) such that σ consists in a heap block of base address $\nu(\alpha)$ and of size n;
- $\gamma_{\mathbb{A}}(\alpha : \mathbf{stack})$ is the set of pairs (σ, ν) such that σ consists in a stack block of base address $\nu(\alpha)$.

An abstract value $(\hat{\epsilon}^\sharp, S, \mathcal{A}, N) \in \mathbb{S}^\sharp$ is valid if each points-to edge in S is included in one type region of N (inductive and segment edges may summarize regions both in $\mathbb{D}^\sharp_{\mathrm{S}}$ and in $\mathbb{D}^\sharp_{\mathbb{A}}$). The concretization in the product domain follows similar rules.

Main transfer functions: Transfer functions and analysis algorithms are mostly left unchanged except that we need to set up transfer functions for **malloc** and **free**. The only new change regarding to the algorithms of chapter 3 is that the unfolding should properly propagate the allocation predicates into $\mathbb{D}^\sharp_{\mathbb{A}}$ (in the same way as it already does for $\mathbb{D}^\sharp_{\mathrm{N}}$). We now consider the memory management operations.

Allocation: the transfer function for analyzing **malloc**(sz) should add a fresh node α to the graph, and add constraint $\mathbf{heap}\langle n \rangle$ in domain $\mathbb{D}^\sharp_{\mathbb{A}}$; furthermore it should add some edge(s) from α, in order to represent the memory cells that were allocated (and their content is unknown):

- either by adding one points-to edge $\alpha \cdot \epsilon \overset{n}{\mapsto} \beta$, where β is a fresh node;
- or by adding several points-to edges accounting for the (uninitialized) fields of a structure, when those can be guessed, e.g., from a cast of the **malloc** output (most C programmers always cast the pointers returned by **malloc** into the proper type);
- or simply by adding an empty conjunction multi-edge $\bigwedge_{\emptyset}^{\alpha \cdot [0,n[}$ of size n.

Deallocation: the transfer function for analyzing **free**(p) should evaluate p into a symbolic node α, check that this node satisfies a constraint of the form $\alpha : \mathbf{heap}\langle n \rangle$ inside $\mathbb{D}^\sharp_{\mathbb{A}}$ (otherwise, report that the deallocation may fail, as it may be applied to a pointer that is not the base address of an allocated region), materialize points-to edges describing the full range $\alpha \cdot [0, n[$ and remove them. If that range cannot be fully materialized into a set of points-to edges, the analysis should raise an alarm that the deallocation cannot be proved safe (i.e., may crash or may discard edges part of other summaries, hence corrupting the abstract value).

4.5 Assessment

We extended Xisa so as to support all features mentioned in this chapter except arrays. Basically, offsets may denote either symbolic or numerical values. Moreover, the local conjunctions for unions (section 4.3) and the abstraction of memory management (section 4.4) were also implemented.

The results obtained by applying this extended version to the analysis of a set of functions on arithmetic expressions described by structures containing unions of structures and constructor tags [71] were in line with those found with the initial version of Xisa (chapter 3). For instance, the analysis of an evaluation procedure took 60 milliseconds whereas the analysis of an iterative application of distribution arithmetic rule took 144 milliseconds. This was to be expected as the fundamental static analysis algorithms were not modified and only very local operations (such as offsets comparison, or multi-edges operations) were modified.

Chapter 5

Application to the interprocedural analysis

Procedures play a key role in programming, and need be analyzed properly. In this chapter, we apply our abstract domain to the abstraction of call stacks, so as to design precise, very context sensitive static analyses, while retaining the analysis algorithms of analyzers such as Astrée [11]. This analysis was presented in [107].

First, we discuss various approaches to interprocedural analysis in section 5.1. Then, in section 5.2, we formalize concrete call stacks and set up an abstraction based on inductive summaries. Section 5.3 describes the interprocedural analysis. An important feature of that analysis is the inductive definition inference algorithm introduced in section 5.3.2. Last, section 5.4 assesses our approach.

5.1 Approaches to interprocedural analysis

Many techniques for analyzing procedures [33] have been proposed with different strengths and weaknesses. The first approach computes procedure summaries (e.g., as in [61, 55, 126, 18]). These summaries are then used to modularly interpret function calls (i.e., derivatives of the *functional approach* [112]). This approach is common, as modularity is important, if not a prerequisite, for scalability. Unfortunately, computing effective summaries is not easy for all families of properties. Intuitively, it is more complex to abstract relations between pairs of states than to abstract sets of states. The former is the essence of what needs to be done to compute procedure summaries, while the latter is what is more typical in program analysis. In shape analysis, computing precise procedure summaries necessitates some reasoning on the boundary between the procedure frame and the footprint of procedures, so as to express which part of the heap a procedure call may modify. This problem is called *frame inference* and many algorithms were developed [51, 80, 18] to solve it, yet some issues such as *cutpoints* [98] make frame inference a very hard problem.

The second approach is to perform *whole program analyses* that, intuitively inlines function calls, virtually ignoring procedure boundaries. Such analyses compute abstract iterates on interprocedural control flow graphs [95] or abstract syntax trees [11]. Moreover, they only need to abstract sets of states (instead of relations on pairs of states), which is simpler. In the other hand, this approach faces several significant challenges, especially for attaching precise context information to the analysis of each call, while still achieving decent scalability.

A challenge common to all interprocedural analysis is to compute precise enough *context* information, to make sure that the analysis will take into account specific situations arising from distinct call sites. Call-string [112] and state partitioning [15] or trace partitioning techniques [108] achieve a good level of context sensitivity by avoiding to merge the information about call sites that should not be abstracted together in order to avoid a loss of precision. Control flow analyses [64, 67, 86] also infer precise context information. However, the properties inferred by these analyses does not provide precise and compact information on the *contents* of the call stack, which is usually abstracted [63].

However, call stacks can be viewed as regular data structures living in the memory states, and could thus be abstracted using shape analysis techniques. This idea was investigated by Noam Rinetzky [99], using the TVLA framework. In this chapter, we propose an interprocedural analysis which also summarizes the call stack, but using the shape abstract domaine which we introduced in chapter 2, which features inductive predicates based summarization, and can thus exploit the inductive structure of the call stack to provide more concise summaries. This analysis not only achieves a high level of context sensitivity, but also handles value ab-

stractions in a very standard manner based on abstractions of sets of states [11].

Interestingly, other recent works highlighted the relation between shape analysis, call stack abstraction and context sensitivity. In particular, Matthew MIGHT did show control flow analyses can be viewed as particular kinds of shape analyses [85]. Moreover, Pascal SOTIN [116] proposed a pointer analysis relying on a numerical domain to express pointer relations in the call stack.

5.2 Call stack abstraction

Before the interprocedural analysis can be formalized, we set up a concrete model for programs with procedures and an abstraction for this concrete model.

5.2.1 Concrete call stacks

Our model makes the call stack explicit: the advantage of this approach is that the context will be viewed as part of the memory state. In this section, we consider the example below, of a list reverse function called in function main:

```
list ⋆ rev(list ⋆ l, list ⋆ acc){
    if(l == NULL){
        return acc;
    } else {
        list ⋆ m = l -> next;
        l -> next = acc;
        return rev(m, l);
    }
}
list ⋆ t;
void main(){
    ...
    //t assumed to point to a singly-linked list
    list ⋆ u = rev(t);
}
```

Dynamic environments: When a program calls a procedure, the environment gets modified, due to the addition of local variables and procedure parameters. Therefore, the set of variables to be mapped to an address should depend on the context (in a structured programming language such as C, this set of variables is fully determined by the execution path, including the calling context). For instance, in the example, after three recursive calls to function `rev`, the environment should contain:

- global variable `t`;
- local variable `u` of function `main`;
- three instances of parameters `l`, `acc` and local variable `m`, corresponding to each recursive call to function `rev`.

This *dynamic* environment structure is not ideal, as it means that when analyzing recursive functions, the analysis will have to summarize not only the content of the memory, but also the environment itself, as it may grow unbounded.

A more concrete model of call stacks: However, we can avoid dealing with an infinite environment in the analysis, in the same way as compilers do, by using a very *concrete* model of the execution environment of programs. The vast majority of compilers implement procedure calls using a call stack, which we can simply view as an inductively defined data structure (section 2.3.2). Then, at any point of the execution (or, at any point during the analysis), the environment should contain only the global variables, and the currently active local variables, that is the local variables such that the current control point belongs to their scope.

If we take an even more concrete point of view, the local variables of function `f` are actually fields of its *activation records*, that is of the region allocated on the stack to store the local state of function `f`. In that view, the environment only needs to contain global variables, and the address of the topmost activation record (i.e., of the activation record corresponding to the current function), which can be considered a variable $\bar{\mathbf{ar}}$. In a real implementation, temporary space would also be reserved inside each activation record in order to store, e.g. return values. We do not represent it unless absolutely necessary.

For instance, at the entry point of the third recursive call to function `rev`, the structure of concrete states is shown in the diagram in figure 5.1 (local variables `m`, and global variable `t` are not represented for the sake of clarity). This diagram shows the *call stack* (the left part), and the *heap* (right part). We can actually remark the first elements of the list are reversed, whereas those that remain to visit are yet to reverse. In this concrete state, we notice four activation records (one for `main` and one for each recursive call to `rev`). Additionally *frame pointers* link from one activation record to the pre-

vious one. In a real implementation, frame pointers are just a field of activation records.

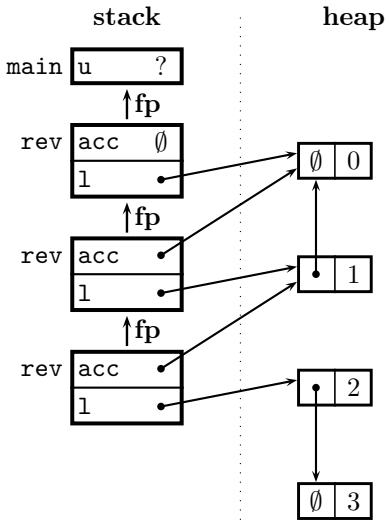


Figure 5.1: Concrete state at the beginning of a recursive call

5.2.2 Abstract call stacks

We now propose to bring the model of call stacks of section 5.2.1 to the abstract level. The main highlights of our extended abstraction are:

- the abstraction of call stack regions together with heap regions;
- the use of abstractions guided by inductive definitions as in chapter 2.

Local variables and abstract environments: Section 5.2.1 did set up a *fixed* notion of concrete environment, even for programs with recursive procedures. As a consequence, abstract environments can be defined in a straightforward manner, in the same way as in section 2.2.2. In other words, abstract environment $\hat{\epsilon}^\sharp$ should map to nodes representing the addresses of:

- the *global* variables;
- the *current activation record*, denoted by $\bar{\mathbf{ar}}$.

Then, local variables of the current call appear as fields of the node representing the address of $\bar{\mathbf{ar}}$.

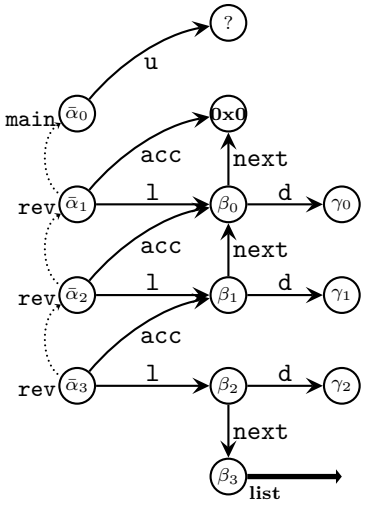
For instance, in the example of section 5.2.1, the environment contains $\bar{\mathbf{ar}}$ and global variable `t`; it is the same at all points during the analysis.

Activation records and frame pointers: The other activation records, corresponding to *caller* procedures should also appear in the abstract (as they are still present on the stack during the execution of the *callee*). They are bound to the current activation record by the *frame pointer* links.

Obviously, each activation record should be characterized by a node representing its base address; local variables of the function it represents are its fields. Activation record address nodes should be annotated with type **stack** (section 4.4).

Abstract, non summarized stacks: The figure below shows an abstract element, which summarizes the concrete state presented in section 5.2.1; in this figure, with the following conventions:

- nodes representing a stack activation record address have a dotted contour, and are annotated with the name of the function they correspond to;
- frame pointer edges are dotted.



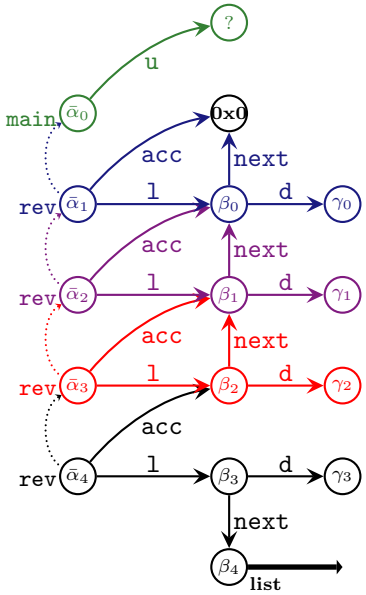
Furthermore, we leave the tail of the list (which remains to be traversed) summarized by an inductive edge. As in section 5.2.1, we omit the `m` fields and global variable `t`.

5.2.3 Summarizing the call stack

In section 5.2.1 and section 5.2.2, we pushed most of the environment into the call stack so that the environment is now fixed, and we also formalized the stack space as we used to describe any other memory area using the abstract domain of chapter 2 and

chapter 4. Thanks to these modifications, we can now apply summarization based on inductive definitions (section 2.2.4) to the call stack.

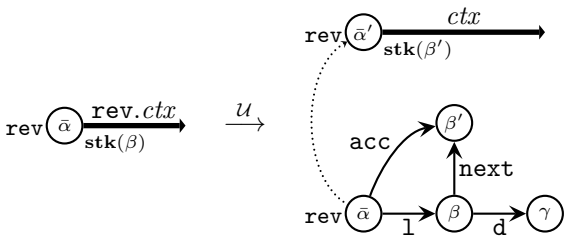
Before we do that, let us observe the shape of a concrete state after one more recursive call. The figure below shows an abstraction of those states, that reach the fourth recursive call to `rev`, and it highlights a repeating pattern, using colors:



The presence of this repeating pattern means that this structure can be summarized using an appropriate inductive definition **stk** such that:

- inductive predicate $\bar{\alpha}\cdot$**stk** denotes a folded activation record together with heap elements it directly points to; thus, the right hand side of the rule required to summarize *one* activation record should look like the red colored fragment of the above picture;
- when folding successive activation records, the relations between the corresponding stack and heap regions should be preserved by the abstraction:
  - the chaining of stack regions shall be taken care of by the frame pointer edges (the backbone of inductive definition **stk** should look like a list, even though it also summarizes many more edges);
  - the relation in the heap should also be captured; this is achieved using auxiliary parameters (section 2.2.4).

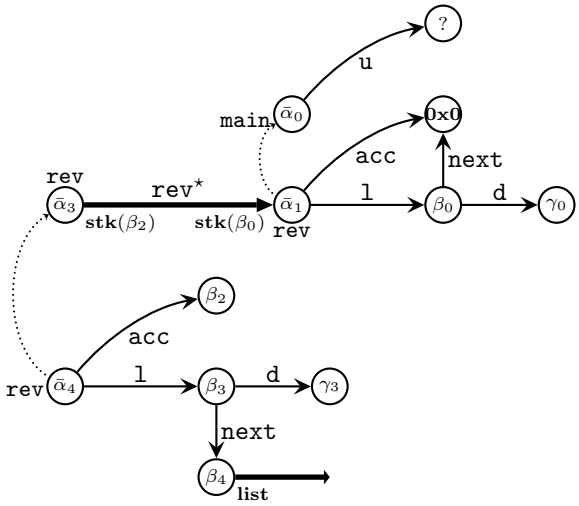
Therefore, we obtain the rule depicted in the graphical representation below:



Note that we annotated the inductive predicates with the contexts they correspond to:

- *ctx* stands for an arbitrary context;
- `rev`.*ctx* stands for context *ctx* plus an ongoing call to `rev`.

Using that rule, we can summarize the abstract states after three or four successive recursive calls, or more, using a stack segment:



We observe this abstract element summarizes any state reached after any number of recursive calls greater than 2, i.e., where the segment stands for any number of calls to `rev`:

- the empty segment corresponds to the case where $\bar{\alpha}_3 = \bar{\alpha}_1$ and $\beta_2 = \beta_0$, i.e., when induction depth is equal to 2;
- if we consider the non summarized state with four recursive calls to `rev`, activation records of addresses $\bar{\alpha}_3$ and $\bar{\alpha}_1$ are summarized here, together with the list elements they point to.

Other examples are provided in [107], for recursive functions such as list construction, list traversal or in-place modification of a doubly-linked list.

5.3 Interprocedural analysis

We now extend the analysis of chapter 3 into an interprocedural analysis, using the abstraction of section 5.2.

5.3.1 Overview of the analysis

To perform the interprocedural analysis, we have to consider the analysis of procedure calls and procedure returns in addition to all the abstract operations introduced in chapter 3 and chapter 4.

Common transfer functions: Transfer functions such as assignments, condition tests or memory management operations do not need be modified to work with the modified abstract domain of section 5.2. Folding operations (section 3.4) such as join or inclusion checking of abstract elements corresponding to the same calling context also extend straightforwardly as well. For all these operations, the only noticeable difference is that the notion of abstract environment was modified, and encloses only global variables, and the address of the current activation record (section 5.2).

Function calls: At a function call site, a new activation record is pushed onto the call stack, and corresponds to the new active function. The abstract transfer function for function calls should reflect that and the effect of parameter passing as well. Parameter passing boils down to a series of assignments to the parameters (i.e., some fields of the new topmost activation record). Moreover, when a function is recursive, the analysis should also *fold* a part of the call stack, using inductive definitions capturing the structure of the stack, like the one shown in section 5.2.3. Since such definitions are not straightforward to write thus we should not expect users to supply them and we will propose an inference algorithm. Besides, the analysis should also carry out an abstract post-fixpoint computation on recursive call sites, as the number of recursive calls is unbounded. Function calls will be discussed in more details in section 5.3.2 and section 5.3.3.

Function returns: Conversely, a function return should remove the topmost activation record after taking care of a possible return value (which boils down to an assignment to a field of the caller activation record).

We remarked that folding should be performed at recursive call sites. Similarly, stack summaries need to be *unfolded* when analyzing returns from recursive functions. Indeed, the topmost activation record should be exposed at all time, in order to guarantee local variables can be accessed; thus, when the analysis encounters a function return while the second activation record is summarized into a segment (as in the figure shown in the end of section 5.2.3), the analysis needs to unfold that segment before removing the current topmost activation record. Besides, the analysis should also carry out an abstract post-fixpoint computation on recursive return sites, since these points are on cycles in the interprocedural control flow graph.

Function return will be discussed in more details in section 5.3.4.

5.3.2 Subtraction algorithm

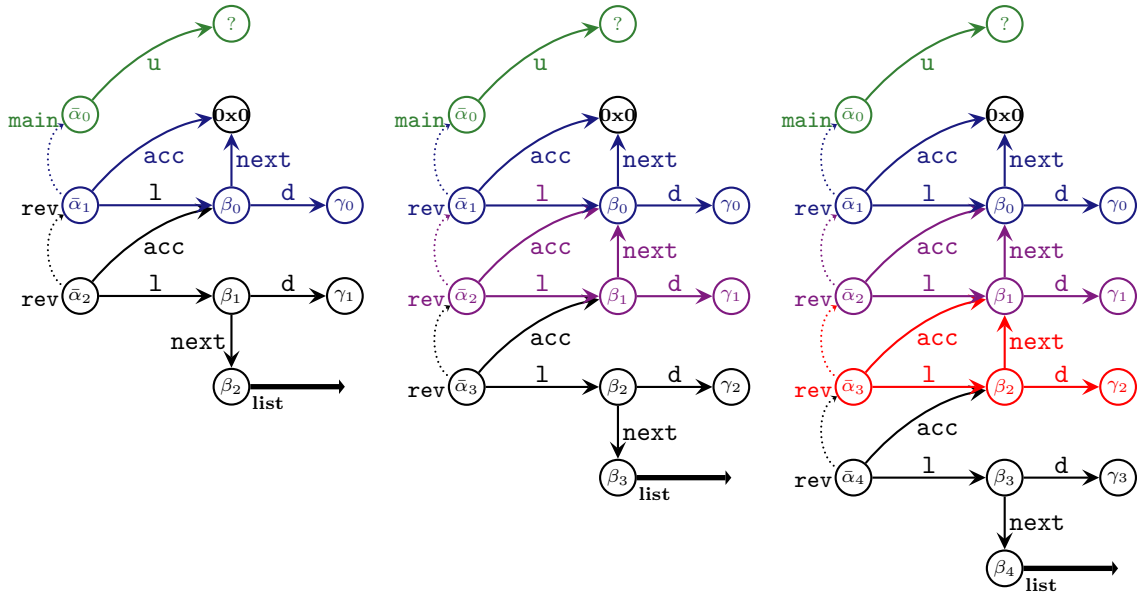
Even though the reverse function of section 5.2.1 is trivial, the inductive definition that was shown to be able to summarize its call stack is not simple. Therefore, such definitions should be inferred automatically, as part of the interprocedural analysis. More precisely, we seek for an algorithm which:

- inputs two consecutive abstract iterates;
- outputs a candidate inductive rule for **stk**.

First iterates: We first consider the first recursive calls to function `rev`, without trying to summarize stack or heap regions. The abstract elements obtained after 2, 3 and 4 recursive calls are shown in figure 5.2, and we use colors to highlight the "contribution" of each iterate to the abstract state, including the activation record of the corresponding function call and the heap fragment that it can be associated to. These colored fragments form the repeating pattern, that was used in order to guess an inductive definition rule in section 5.2.3.

We can also observe that:

- the "difference" between an abstract iterate and the next one corresponds exactly to the colored fragments: for instance, the difference between the second and the third iterates corresponds to the purple fragment in figure 5.2(b); similarly, the difference between the third and the fourth iterates corresponds to the red fragment in figure 5.2(c);
- the auxiliary parameters which were used in order to capture the relation between the val-



(a) After two recursive calls (b) After three recursive calls (c) After four recursive calls

Figure 5.2: Abstract states at the first recursive calls to `rev`

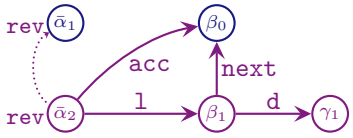
ues of the fields of a stack activation record and those of the previous ones correspond to the nodes which are pointed to by edges of two colors.

These observations set up the basis for the rule inference algorithm which we describe in more details in the next few paragraphs.

Graph subtraction: The graph subtraction algorithm [107] extracts the difference between two shape graphs, and outputs the core of the candidate rule. It achieves this using a double graph traversal, similar to that of folding algorithms (section 3.4): thus, it also relies on a pairing function (so as to bind pairs of nodes that the analysis should abstract together) and applies a set of local erasure rules (removing one edge in both arguments) until one of the arguments is empty; then, it returns the non-empty argument. The initial pairing should bind pairs of addresses of global variables, and pairs of addresses of the topmost activation records, as well as addresses of activation records which are present in both graphs, starting from `main`. For instance, when computing the subtraction of the graph of figure 5.2(b) and of the graph of figure 5.2(a), $\bar{\alpha}_2$ in figure 5.2(b) is not bound and the initial mapping contains the following pairs:

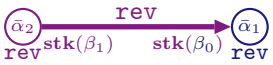
in figure 5.2(a)		in figure 5.2(b)
$\bar{\alpha}_0$	$\leftrightarrow$	$\bar{\alpha}_0$
$\bar{\alpha}_1$	$\leftrightarrow$	$\bar{\alpha}_1$
$\bar{\alpha}_2$	$\leftrightarrow$	$\bar{\alpha}_3$

As in the other algorithms traversing two graphs, the edge removal process based on edge pairing may fail to return the expected result (i.e., remove all edges in one arguments, leaving the remains of the other argument as the subtraction result), and terminate with a pair of non empty graphs, with some edges left in both of them. This is a consequence of the non confluence of the rewriting system induced by edge removal rules. Therefore, an adequate strategy should be used. Fortunately, in our experience, a breadth-first style strategy was found sufficient in all cases.



From this result, inferring additional parameters and pure predicates boils down to the computation of intersections of sets of nodes and to the slicing of the numerical abstract values. In the example, only

β_0 is shared with the previous activation record, hence only one auxiliary parameter is required, to summarize the result of subtraction with a stack segment of length 1:



In the example, the graph subtraction algorithm infers the inductive rule of section 5.2.3. Frame inference [18] is a related issue, where an analysis attempts to identify the footprint and the frame of a function call. However, we are looking here for an exact match as opposed to an entailment between two configurations.

5.3.3 Analysis of calls and recursive calls

The abstract post-fixpoint: When the analyzer detects that a function call is recursive (i.e., one instance of the callee is already present in the context), it should infer an inductive rule, using the algorithm shown in section 5.3.2, and then use that rule for folding fragments of the call stack:

- after applying the subtraction algorithm, the graph fragment extracted by subtraction is replaced with a segment of length 1;
- at the next iteration, the analyzer attempts to reuse the same inductive rule for widening, in order to extend that segment.

The computation of abstract iterates terminates and produces a sound over-approximation for the set of concrete states that can be observed at the entrance into the function being analyzed [107].

In our example, figure 5.3(a) shows the abstract element obtained after the subtraction, and after replacing the extracted fragment with a stack segment edge. Figure 5.3(b) shows the next iterate, before widening is applied. The widening of these two abstract elements is isomorphic to the graph of figure 5.3(a), up-to renaming, thus it is the abstract post-fixpoint.

The auto-parameterizing domain: The extended interprocedural static analysis infers additional inductive rules for the definition of **stk**. This brings up another issue: how should this inductive rules generation process be controlled so that the analysis terminates, and produces precise results.

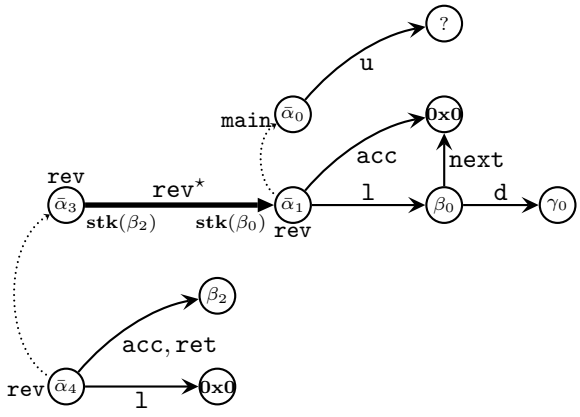
We have shown in [107] that this iteration scheme can be viewed as a static analysis using a cofibered domain [119], where the abstract domain is built upon the shape abstract domain of chapter 2, parameterized by the data of a set of rules for **stk**. Therefore, we may let the analyzer generate additional rules for **stk**, provided that the process of adding new rules terminates, i.e., a widening on the powerset of rules lattice should be used. The proof of termination is twofold: first, the set of rules for **stk** stabilizes; once it is stable, the analysis works exactly as in chapter 3.

We also remarked in [107] that this process can also *weaken* some **stk** rules, which is sound, as it only makes all abstract values that are based on **stk** weaker.

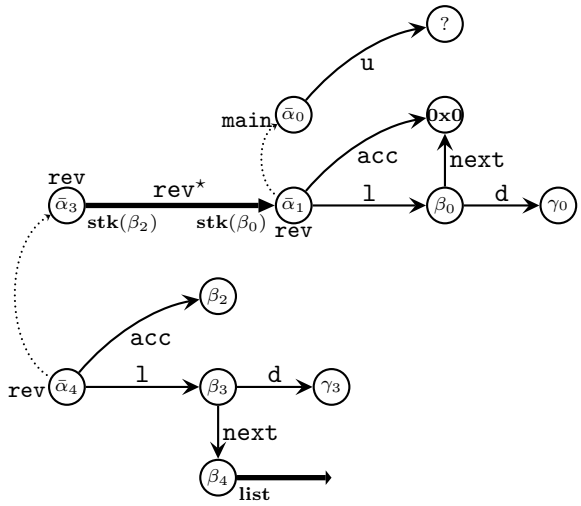
5.3.4 Analysis of function returns

During the execution of a series of recursive calls, the length of the chain of returns is equal to that of the chain of calls, thus it is also unbounded, and the analysis needs to compute an abstract post-fixpoint at return sites.

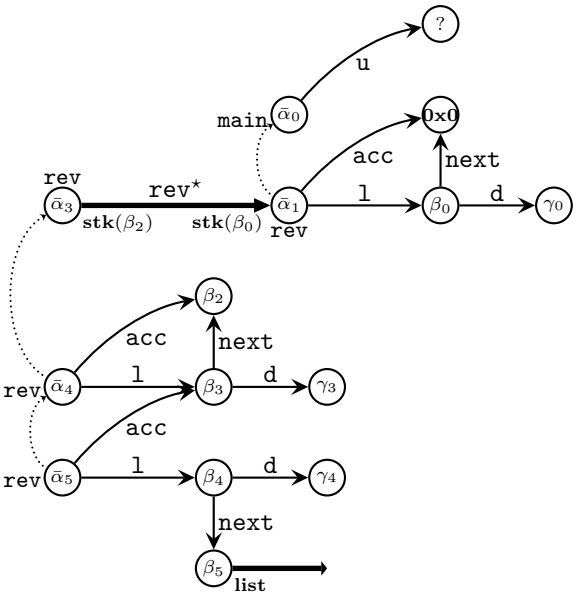
At a return site, the topmost activation record should be removed, and the analyzer should go back to the analysis of the caller. However, it should maintain the new topmost activation record exposed at all times, thus the activation record of the caller should be materialized after the return. For instance, the analysis of the `rev` example gives the abstract value below (where we also represent the `ret` temporary return variable of the returning activation record), at the **return** in the **true** branch of the **if** statement:



After unfolding the stack segment and right before discarding the activation record of the callee, we get the abstract value below:

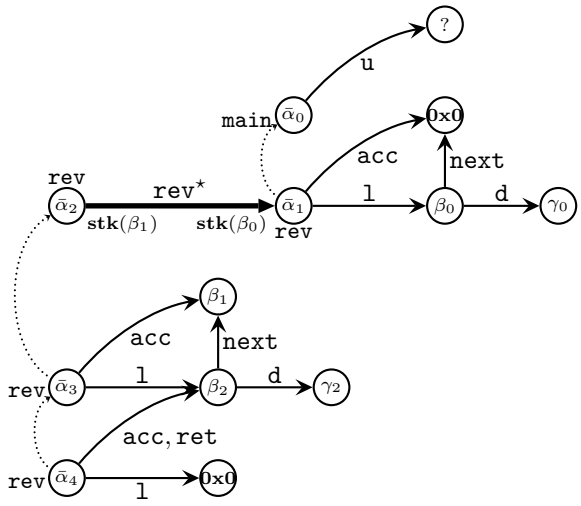


(a) First widened iterate; abstract post-fixpoint



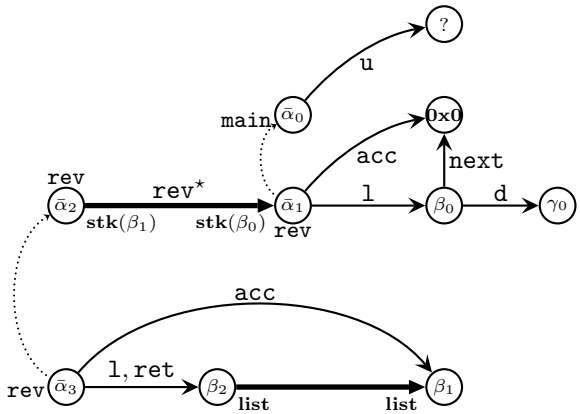
(b) Next abstract iterate, before widening

Figure 5.3: Abstract iterates with widening, at the call site

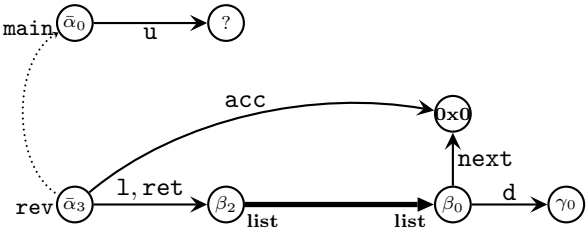


We note that the **false** branch of the condition cannot be executed in the deepest recursive call (as it generates itself another recursive call). Therefore, the analysis should then go back to the next return site (which is actually in the **false** branch of the caller), following [95]. From that point, each return from a recursive call should branch to that point.

After one more iteration and a widening, the analysis outputs the invariant below, which shows a list segment starting pointed to by `ret`:

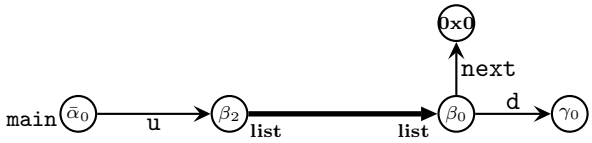


Once this fixpoint is reached, we obtain an abstract post-condition for the control point right after the call to `rev` in `main` after unfolding the stack segment into an empty segment. This results in the invariant below:



After return to `main`, we obtain the property that

the procedure returns a well-formed list:



Experimental assessment: In [107], we modified the Xisa analyzer in order to abstract call stacks as shown in section 5.2 and infer call stack summaries using the subtraction algorithm. Except for the obvious parts (the new model of environments, the subtraction algorithm and the interprocedural iteration engine), this new analysis did not require very significant modifications of the analyzer.

The analysis was tested on a series of micro-benchmarks, and it appears that the analysis of a recursive version of simple programs (implementing, e.g., usual list algorithms) is slower than the iterative version by a factor of 3 to 5, which is coherent with the fact the analysis of the recursive program actually computes *two* fixpoints instead of one (one fixpoint at the call site and one fixpoint at the return site), and that it also needs to perform subtraction. More ambitious experimentation was left as future work.

5.4 Discussion

We now discuss various features of our analysis, and compare it with the modular approach.

5.4.1 The cutpoint problem

Modular shape analyses such as [61, 98, 51, 18] typically infer an abstraction of the *effect* of a procedure on a fragment of the heap; computing that fragment is called *frame inference*. However, such an analysis must also abstract *cutpoints* [98] carefully to obtain precise results. Cutpoints are the locations at the border of the callee's reachable heap, that is, any node that is reachable from the callee's parameters (though not directly pointed-to by them) and reachable from a pending activation or a global (without following links inside the callee's reachable heap). It is important to track cutpoints precisely in a modular analysis, as they are used to reflect the effect of the callee in the caller's state on function return. Doing so can be challenging, as an unbounded number of cutpoints may arise either due to unbounded recursion or due to the traversal of unbounded heap structures. While there is no general solution for cutpoints today, several partial solutions have been proposed. For example, they focus on isolating their effect by proving cutpoint-freeness [100] or by proving that cutpoints are not live [68], or they reason up to a bounded number of them [98].

Our analysis based on call stack summarization also needs to cope with cutpoints so that the widening iteration reaches a precise fixed point. However, the principle of call-stack summarization is to infer a (possibly complex) predicate, for abstracting the call stack, including any cutpoint it may contain. While this may seem a general solution in theory, one should note that cutpoints will make inductive stack definitions more complex, so this approach obviously has a limit. Yet, we notice that our approach allows to summarize unbounded numbers of cutpoints as we have shown in [107], where a procedure modifies a doubly-linked list and generates an unbounded number of cutpoints; this unbounded set of cutpoints is abstracted by an inductive definition with auxiliary parameters, thanks to the *regular* structure of the cutpoints. We also pointed out that very complex cutpoint problems may be due to orthogonal issues like sharing (which we discuss in section 6.2.4.

5.4.2 Combining modular and stack summarization

We observed in section 5.1 that modular and whole program analyses have very different sets of advantages and drawbacks. Intuitively, modular analyses are especially adapted to programs where each procedure can be reasoned about in a very independent manner, as for, e.g., a library. In the other hand, sometimes procedures with very few call sites are designed with very specific and complex assumptions in mind, which are related to those call sites, and should better be analyzed in their context. Besides, some softwares cannot be reasoned about precisely in a per procedure manner like the specific families of programs analyzed by [11].

Thus, the design of a static analysis framework which would apply either technique depending on the situation, or combine them efficiently would be an obvious and very useful follow up to this work. For instance, when analyzing a library,

such a framework would compute relations over-approximating the semantics of "public" functions whereas complex private functions (possibly called by a public function) should be analyzed in their context, using call stack summarization in a more local way. A general framework should also provide a mor elegant solution to the issue of cutpoints.

Chapter 6

Conclusion and perspectives

In this chapter, we make a series of concluding remarks, regarding to the most important principles of our domain (section 6.1) and show directions for further extensions (section 6.2), including some ongoing and unpublished works. Last, section 6.3 discusses the

6.1 Foundation for memory abstract domains

In section 2.1, we did set up the foundations to our abstract domain, in order to attack a wide variety of static analysis problems, thus we now assess how these principles contribute to the design of the abstraction and of the analysis algorithms.

From the concrete: A first important note is that the structure of our abstraction remains very close to the definition of concrete states, and that it is based on a very concrete representation, using explicit abstractions for addresses, values and memory cells. At first, this may not look the most intuitive approach, yet it turns out a rewarding decision, since it actually makes the extension of the analysis to deal with various programming languages features more natural. Indeed, since the abstract values closely match the structure of concrete state, slight modification in the concrete level are easier to match in the abstract level.

In particular, we found this made the adaptation of the analysis to low-level codes more intuitive, as our abstract shape graphs are already designed so as to describe precisely memory cells and their addresses, as we could see in chapter 4. In the other hand, our abstract values are still compact, and when concrete features are irrelevant, they can often be dropped (for instance, numeric offsets may be abstracted by symbolic names when programs do not rely on pointer arithmetic, as shown in section 4.2.1).

Similarly, we found in chapter 5 that this abstraction also allows a nice encoding of the call stack, which is actually fairly close to the call stack of the compiled program. In turn, this representation makes the summarization of the call stack rather natural.

Finally, *separation* [60, 97] serves as a basis for all our static analysis algorithms, which all rely on local reasoning (chapter 3). This notion of separation is also based on a very concrete idea, to abstract memory states region by region.

Overall, we believe that this very concrete root is a strong asset of the abstraction, and expect it to make further extensions easier, e.g., in order to analyze more radically different programming languages.

The expressive power of inductive definitions: Secondly, inductive definitions which are the basis of summarization turn out very powerful in practice, and we observed that many data structures have an inductive structure, such as many kinds of lists and trees (section 2.2.4), and even call stacks (chapter 5). Inductive formulas have long been known to be a powerful means to express complex mathematical properties [27], and are at the root of, e.g., the Coq proof assistant [92].

The nature of induction is also independent from the basic atoms of our abstractions, which may account for low level details (such as precise numerical information about offsets, cells sizes...).

Inductive definitions are not adapted to all cases, though:

- *arrays* also need summarization and have no inductive structures; our abstraction views them as contiguous regions, which allows to apply array abstractions locally though (section 4.2.3).
- *graphs* and similar structures may have no clear inductive definition, and would be more naturally represented as unstructured sets of memory regions.

However, we will show in section 6.2.5 that even in such cases, the experience gained using inductive definition should still offer some important benefit.

Induction in the heap abstraction, and in the analysis: The notion of induction is at the core of abstract interpretation, as one may view an abstract post-fixpoint an inductive program proof. In fact, our abstraction does also rely on induction for the summarization of heap regions. Therefore, our analysis relies on two notions of induction (on the structure of programs, and on the structure of the heap). This gives a very intuitive view of our algorithms:

- *unfolding* (section 3.3) performs case analysis on inductive structures, like logical "destructor" (or "elimination") rules, and allows to use an inductively defined predicate for reasoning locally on the heap [60];
- in the other hand, *folding* algorithms act like "constructor" (or "introduction") rules, and allow to build inductive predicates, which are easier to propagate along during analyses;
- as usual, *widening* (section 3.4.3) accelerates convergence of abstract iterates towards a post-fixpoint, which should stand for an inductive proof; it relies on *folding* in order to build inductive summaries in heap abstractions.

We remark that the interaction between operations that affect fixpoints in the heap and the induction in the analysis is very natural, since pieces of code that require inductive reasoning often create, treat or dispose of inductively defined heap structures, like a list allocation (or deallocation) loop, or like a recursive program (building an inductively defined all stack at function call sites, and destroying it at return sites).

6.2 Perspectives for further developments

In this section, we present some directions for extensions to our framework. We have already started investigating some of these, whereas others are longer term goals.

6.2.1 Inference of inductive definitions

Inductive definitions describing user-specific structures are not trivial to write, and may also be rather large. Therefore, we should try to make this task as automatic as possible. In the following, we propose a few ways of synthesizing inductive definitions for the parameterization of our abstract domain.

Syntactic interpretation of type definitions: Low level programs such as device drivers typically contain series of definitions for structures, with only a small proportion of them having a complex layout: typically, type definitions comprise a large number of nested structures and only a few of them point to lists or trees of structures. Therefore, in many cases, appropriate definitions may be computed by assuming all structures are chained in a simple manner, with no sharing. This works well when there is no sharing, as in the case of lists and trees. This approach was followed in [6] in order to treat data structures that consist in hierarchical data types, with lists of lists. However, this syntactic approach will not discover any relation among pointer fields as in, e.g., doubly-linked lists.

Inference of inductive definitions: In section 5.3.2, we observed that inductive definition for summarizing the call stack can be inferred automatically, using a subtraction algorithm, which proved able to infer relations among pointer fields e.g., when summarizing *cutpoints*. However, that algorithm did rely on two important assumptions:

1. a sequence of calls *builds* a call stack, each call adding one activation record;
2. the *head* of the structure is also known, and corresponds to the address of the topmost activation record.

To generalize this algorithm to the inference of other inductive definitions than for call stack summarization, we need to isolate cases where the same assumptions would hold. In general, this happens whenever a piece of code is dedicated to the construction of a data structure. In particular, in object oriented languages, constructors allocate objects and initialize them, so that they could be used in order to derive some rules for a candidate definition for the summarization of objects of their class. Definitions derived that way may not be completely

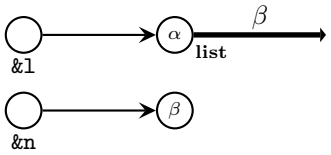
adequate though, as the initialized objects may satisfy much stronger invariants than the class invariants [75]; thus, invariants derived from the constructors may have to be significantly weakened at a later stage.

Further research on generalizations of the subtraction algorithm (section 5.3.2) will make the abstract domain easier to adapt to the analysis of larger applications.

6.2.2 Strengthening inductive predicates

We have shown our language of inductive definitions is quite expressive. It is actually possible to make inductive predicates even more expressive using additional annotations that constrain derivation trees proving that a concrete element be part of the concretization of an abstract one (section 2.2.6), further refining the meaning of abstract elements. In fact, we actually relied on such annotations in order to summarize call stacks while preserving information about call strings, in section 5.2.3. This approach can be greatly generalized.

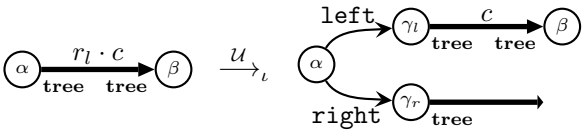
Using length of derivations: Length of structures may be expressed using auxiliary parameters, as in the case of **list_len** (section 2.3.3), yet it is also possible to use a primitive notion of length of inductive predicates (section 2.2.4 and section 2.2.5). That primitive notion of length was actually used in order to define forward and backward segment unfolding (section 3.3.2). For instance, the abstract element below over-approximates states with two variables `l` and `n`, such that `n` contains the length of the list pointed to by `l`:



String abstraction: In section 5.2.3, we relied on regular expressions constraints on the shape of inductive derivations in order to express that a stack segment corresponds to a set of call strings. Regular expressions provide a natural abstraction for sets of words over a given alphabet, and can thus be used in order to constrain linear sequences of unfoldings (as opposed to general derivations which are trees).

For instance, a **tree** segment may unfold either to an empty segment, or to a tree node the left (resp., right) sub-tree of which points to another segment. Thus, a concrete store that can be abstracted by a segment of the form $\alpha{\cdot}\mathbf{tree} \Rightarrow\!\!* \beta{\cdot}\mathbf{tree}$ can be characterized by a path from α to β, that is a sequence of left or right branches, ending in β. Therefore, it can be characterized by a word over alphabet $\{r_l, r_r\}$, where r_l (resp., r_r) stands for a left (resp., right) step. When a precise information about the shape of the path from α to β is known, we may annotate a segment with e.g., a regular expression corresponding to that information.

Annotations constrain unfolding. For example, annotation $r_l \cdot c$ stands that the segment has length greater than 1 (hence, is not empty) and may start only with a left edge, so only one possible unfolding:



Moreover, annotation $r_l^\star$ means that β is located on a left-most path, starting from α.

Abstraction of paths and derivations: Length or path information may be of great interest in static analysis, as it allows to greatly refine the information expressed in inductive predicates, without requiring more complex predicates. More generally, concretization derivations can be viewed like trees, and could be refined using abstractions for trees [81] and sets of trees [82]. Similarly, graphs derived by unfolding should be filtered according to the derivation abstraction.

The advantage of this construction is twofold. First, it increases the expressivity of inductive definitions without requiring more complex definitions be used as parameters for the abstract domain. Second, it makes use of existing abstractions for length information (i.e., integers), words over an alphabet corresponding to the rules of the inductive definition or derivation trees.

6.2.3 Internal reduction operator

We observed inductive definition are very expressive. However, there is a cost to pay for this expressiveness: in particular, some heap properties

may be expressed in many different ways. For instance, doubly-linked lists may be described either forward or backward:

- *forward* inductive definition:

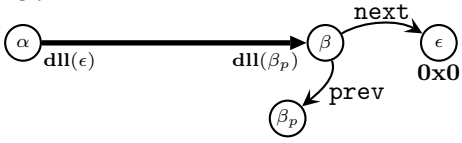
$$\begin{array}{ll} & \alpha \cdot \mathbf{dll}(\gamma) ::= \\ & \quad (\mathbf{emp}, \alpha = 0) \\ \vee & \quad (\alpha \cdot \mathtt{next} \mapsto \beta * \alpha \cdot \mathtt{prev} \mapsto \gamma \\ & \qquad * \beta \cdot \mathbf{dll}(\alpha), \alpha \neq 0) \end{array}$$

- *backward* inductive definition:

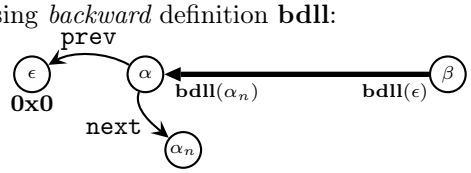
$$\begin{array}{ll} & \alpha \cdot \mathbf{bdll}(\gamma) ::= \\ & \quad (\mathbf{emp}, \alpha = 0) \\ \vee & \quad (\alpha \cdot \mathtt{next} \mapsto \beta * \alpha \cdot \mathtt{prev} \mapsto \gamma \\ & \qquad * \gamma \cdot \mathbf{bdll}(\alpha), \alpha \neq 0) \end{array}$$

Thus, a doubly-linked list the first (resp., last) element of which has address α (resp., β) may be abstracted by either of the following graphs:

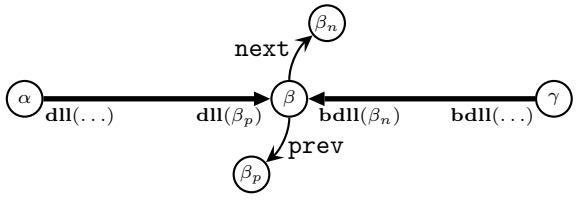
- using *forward* definition **dll**:



- using *backward* definition **bdll**:



In the static analysis point of view, this may actually be the source of major imprecisions. Indeed, folding rules (section 3.4) cannot take such an equivalence into account, and may get fail when trying to fold a graph containing both flavors of doubly-linked lists. For instance, the graph below could be abstracted by a doubly-linked list segment, even though none of the folding rules of section 3.4 will allow this weakening:



Before any of the folding rules of section 3.4 can be applied, the analysis should perform a rewriting step based on the equivalence between forward and backward definitions of doubly-linked lists. Such an operator is actually an *reduction* in the sense of [30].

In practice, designing and implementing such an operator is usually very hard, and in our case, no optimal reduction can be designed. A *conservative* reduction operator should solve the following three issues:

1. *infer what* rewriting is useful to the analysis;
2. *check the soundness* of that rewriting;
3. *decide when* to apply the rewriting.

In the last few years, we proposed a solution to the second point [101], which reduces the checking that an entailment between sub-graphs containing inductive definitions to a static analysis. Indeed, the concretization of an inductive predicate boils down to a least fixpoint, and that our static analysis algorithms (chapter 3) aim at over-approximating fixpoints. Therefore this analysis can be used in order to prove that all memory states in the concretization of an inductive predicate ι are also in the concretization of inductive predicate ι' (the analysis checking the entailment will "run" ι and use ι' to abstract the sets of stores it generates [101]). For instance, the checking that the backward doubly-linked list segment can also be approximated by a forward one was reduced to an abstract interpretation of the backward segment predicate, using our abstract domain parameterized with the forward doubly-linked list predicate.

In the other hand, the other two points remain unsolved. Except for simple cases, choosing the right rewriting relations seem to require a rather deep understanding of the structure of abstract elements. However, we observed that such rewriting are mainly helpful when *folding*, thus a static analyzer relying on a search strategy may be able to make appropriate decisions on what reduction to perform, e.g., when an attempt to compute a join fails in a situation where no rule applies because inductive edges do not match, as in the above example.

6.2.4 Reasoning on sharing

Many complex data structures, such as directed acyclic graphs, graphs, shared binary trees... involve a high degree of *sharing*, where some cells are pointed to by many links. As our abstract domain is based on separation logic, an abstract element cannot describe a single concrete cell several times, and shared cells are no exception. This makes the encoding of *shared structures* tricky.

In the following, we overview the solution that

was investigated during the internship of Suzanne RENARD from École des Mines de Paris (September 2010–March 2011).

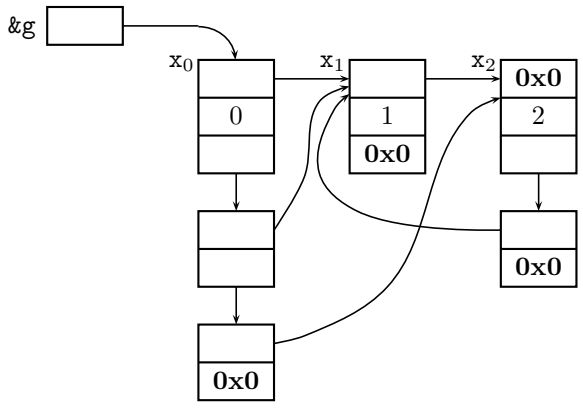
A graph example: In the following, we consider a graph structure, described by a list of nodes, where each node encloses a name and an adjacency list of outgoing edges (graphs may not be defined using an inductive problem —we discuss this in section 6.2.5—, but we focus on sharing which can be observed on this inductive data type just as well as on a non inductive data type). Each outgoing edge may be represented by a pointer to its destination. This data structure can be described by the type definition below:

```
typedef struct adjList{
    struct adjList ⋆ next;
    struct nodeList ⋆ dest;
}adjList;

typedef struct nodeList{
    struct nodeList ⋆ N;
    int name;
    adjList ⋆ edges;
}nodeList;

typedef nodeList graph;
```

For instance, a graph with three nodes labeled $\{0, 1, 2\}$, and with transitions $0 \rightarrow 1$, $0 \rightarrow 2$ and $2 \rightarrow 1$ may be described by the concrete store below:



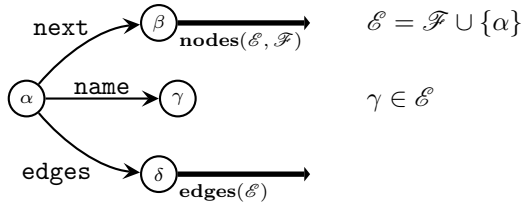
While this graph representation is mostly based on lists, which can be dealt with in our framework, there is a significant new difficulty: the destination node of each edge may be any node in the graph.

A graph inductive definition: In order to express precisely all the structural invariants of this structure, we need to augment our inductive definitions with additional parameters representing sets of nodes. This way, we can specify that each destination field should point to a node in the list. We also need to specify that the set of nodes in the list corresponds exactly to this set. Therefore, two set parameters should be used. Hence, we obtain the set of inductive definitions below:

$$
\begin{array}{ll}
\alpha \cdot \mathbf{graph}(\mathscr{E}) ::= & \\
\quad (\alpha \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{E}), \mathbf{true}) & \\
\alpha \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F}) ::= & \\
\quad (\mathbf{emp}, \alpha = 0 \wedge \mathscr{F} = \emptyset) & \\
\vee \quad (\alpha \cdot \mathtt{next} \mapsto \beta * \alpha \cdot \mathtt{name} \mapsto \gamma & \\
\quad * \alpha \cdot \mathtt{edges} \mapsto \delta * \beta \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{F} \setminus \{\alpha\}) & \\
\quad * \delta \cdot \mathbf{edges}(\mathscr{E}), & \\
\quad \alpha \neq 0 \wedge \alpha \in \mathscr{E}) & \\
\alpha \cdot \mathbf{edges}(\mathscr{E}) ::= & \\
\quad (\mathbf{emp}, \alpha = 0) & \\
\vee \quad (\alpha \cdot \mathtt{dest} \mapsto \beta * \alpha \cdot \mathtt{N} \mapsto \gamma * \gamma \cdot \mathbf{edges}(\mathscr{E}), & \\
\quad \alpha \neq 0 \wedge \beta \in \mathscr{E}) &
\end{array}
$$

In all these definitions, $\mathscr{E}$ denotes the set of nodes of the whole graph; **edges** checks all edges in an adjacency list point to some node in the graph. Moreover, auxiliary parameter $\mathscr{F}$ of **nodes** tracks nodes that appear in the list of nodes, so as to check each node appears exactly once in the structure.

As an example, the diagram below displays a partially unfolded $\alpha \cdot \mathbf{graph}(\mathscr{E})$ (basically, only the first element is unfolded):



Product with a set constraints domain: As the above example shows, unfolding a **graph** predicate generates additional constraints among set variables, that our abstract domain should also faithfully represent.

Therefore, we use a domain of sets constraints $\mathbb{D}^{\sharp}_{\text{set}}$, to represent constraints on *set variables* $\mathscr{E}, \mathscr{F}, \ldots \in \mathbb{V}^{\sharp}_{\text{set}}$ and node symbolic names ([93] proposes such an abstract domain). The concretization γ_{set} of this domain maps a set of constraints into a set of pairs $(\eta, \nu) \in \mathbb{Val}_{\text{set}} \times \mathbb{Val}$, where

$\mathbb{Val}_{\text{set}} = \mathbb{V}^\sharp_{\text{set}} \longrightarrow \mathcal{P}(\mathbb{V})$, since abstract values represent constraints not only on sets, but also on elements (e.g., stating that $\alpha \in \mathscr{E}$ for some node symbolic name α and some set name $\mathscr{E}$).

Product abstraction for memory states should now include three arguments:

$$\mathbb{M}^\sharp = \mathbb{D}^\sharp_{\text{S}} \times \mathbb{D}^\sharp_{\text{N}} \times \mathbb{D}^\sharp_{\text{set}}$$

Concretization of elements of that domain relies on the existence of a compatible concrete set mapping, and on a conjunction:

$$\begin{array}{l}(\sigma,\nu) \in \gamma_{\mathbb{M}}(S,N,C) \\ \Leftrightarrow \exists \eta \in \mathbb{Val}_{\text{set}} \left\{ \begin{array}{lll} & (\sigma,\nu) \in \gamma_{\text{S}}(S) \\ \wedge & \nu \in \gamma_{\text{N}}(N) \\ \wedge & (\eta,\nu) \in \gamma_{\text{set}}(C) \end{array} \right.\end{array}$$

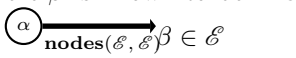
This defines a valid instance of the cofibered abstract domain of [119], extending that of section 2.2.

The concrete graph shown in the beginning of the section is in the concretization of the abstract element shown above, with the following concrete mappings of set and symbolic variables (omitting variable $\mathtt{g}$):

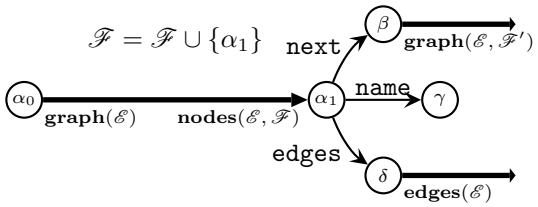
$$\begin{array}{lcllcl} \alpha & \mapsto & \mathtt{x}_0 & \mathscr{E} & \mapsto & \{\mathtt{x}_0, \mathtt{x}_1, \mathtt{x}_2\} \\ \beta & \mapsto & \mathtt{x}_1 & \mathscr{F} & \mapsto & \{\mathtt{x}_1, \mathtt{x}_2\} \\ \gamma & \mapsto & 0 & & & \end{array}$$

Static analysis: The unfolding and folding algorithms remain mostly unchanged, except for the fact that they should also track constraints on sets in addition to numerical constraints. However, those algorithms are not sufficient to handle the analysis of many common graph algorithms, as the reduction issue raised in section 6.2.3 needs to be solved, as the following examples show:

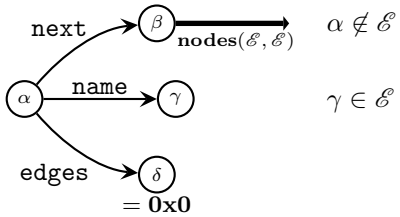
- **Non localized unfolding:** We consider the abstract element below, where α is known to be the address of the first element of a graph the set of node addresses of which corresponds to $\mathscr{E}$, and where β is known to be in $\mathscr{E}$:

  Under this assumption, it would be perfectly valid to materialize the node of address β. Such a situation would actually arise when considering, e.g., a traversal algorithm and when β is the address of the next node to be reached in the traversal. It would also occur in an algorithm attempting to recognize a word in an automaton (an automaton data structure would be rather similar to the graph considered here).
  The above abstract element can be proved equivalent to the abstract element below, where node of address β is materialized:

  Furthermore, the inclusion of the initial $\alpha \cdot \mathbf{graph}(\mathscr{E})$ into that abstract element can be proved by induction over **graph**, using a similar algorithm as that of [101].
- **Folding with different set parameters:** Let us consider the graph below, where one node was added into a graph, and was put in the first position (this operation may be part of a graph construction loop, adding new nodes to an initially empty graph):

  The whole structure would fold into a completely summarized **graph** structure, yet the set parameters would have to be updated so as to reflect the fact the set of nodes for the whole graph is $\{\alpha\} \cup \mathscr{E}$. However, the folding rules of section 3.4 will not fold the above abstract element into $\alpha \cdot \mathbf{graph}(\{\alpha\} \cup \mathscr{E})$, since the first auxiliary parameter of the recursive inductive predicate $\beta \cdot \mathbf{nodes}(\mathscr{E}, \mathscr{E})$ does not match with $\{\alpha\} \cup \mathscr{E}$: indeed, this inductive predicate describes a graph fragment such that all edges point into set $\mathscr{E}$ whereas we would need a predicate describing a graph fragment such that all edges point into set $\mathscr{E} \cup \{\alpha\}$. The implication of these two properties is very intuitive, yet proving it requires reasoning by induction over the structure of the predicate. We can actually notice that **edges** is monotone over its predicate, and that **nodes** is monotone over its second pa-

rameter:

$$\begin{array}{l}\mathscr{E} \subseteq \mathscr{E}' \wedge (\sigma,\nu,\eta) \in \gamma_{\mathrm{S}}(\alpha \cdot \mathbf{nodes}(\mathscr{E},\mathscr{F})) \\ \quad \Longrightarrow (\sigma,\nu,\eta) \in \gamma_{\mathrm{S}}(\alpha \cdot \mathbf{nodes}(\mathscr{E}',\mathscr{F})) \\ \mathscr{E} \subseteq \mathscr{E}' \wedge (\sigma,\nu,\eta) \in \gamma_{\mathrm{S}}(\alpha \cdot \mathbf{edges}(\mathscr{E})) \\ \quad \Longrightarrow (\sigma,\nu,\eta) \in \gamma_{\mathrm{S}}(\alpha \cdot \mathbf{edges}(\mathscr{E}'))\end{array}$$

Such reasoning could be done using a monotonicity analysis such as [91].

6.2.5 Abstract domains based on packing and unpacking operations

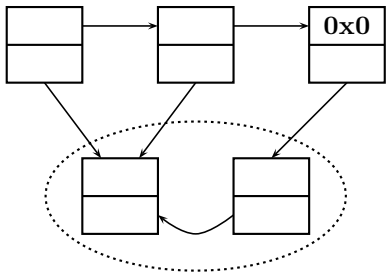
Abstract domain operations are based on two main operations for "packing" memory predicates into more abstract summaries un "unpacking" summaries into more concrete memory predicates. In our domain, these *pack* and *unpack* operations are tied to the notion of *induction*, like our notion of summaries. Other shape analyses such as TVLA rely on very different formalisms, yet also feature a notion of *local concretization* (or *focus*) and *global abstraction* (or *canonicalization*).

However, other forms of summarization can be considered, even though they are not based on induction.

Array analyses: Array analyses exploit partitions of arrays into a finite number of sub-sets, and apply abstractions to each element of the partition, thus each element of such a partition defines a *summary*. Furthermore, some array analyses like [38] use dynamic summaries, which means they may change during the analysis (hence, fewer summaries may be used, and provide a more accurate description of the concrete behaviors at a cheaper cost).

Operation on summaries such as pack and unpack can also be defined, which split and merge partitions.

Collecting sets of cells: Some structures cannot be described in an elegant way using induction. For instance, a graph comprises a set of nodes, which may not be ordered in any way. The example of section 6.2.4 was using a list of nodes, in order to avoid this problem. However, we may imagine a graph structure where nodes are not list elements; instead, another list may be used as an index (which may not be one to one):



Furthermore, it would be possible to define a structure where not all nodes are pointed to by an index element; *reference counting* would ensure that a node will be deleted when the last reference to it is discarded.

Such examples are hard to describe with our inductive definitions. While the index has list structure and can easily be described by induction, the rest of the structure should be described as a set of pairwise disjoint sub-regions, which satisfy some local properties (i.e., that their outgoing edges remain in that set). Thus this set should be described by a summary, and the abstraction would also feature pack/unpack operations, where operation pack merges two sets of elements with the same property together and unpack singles out one element of the set. Such an abstraction shows some resemblance with TVLA [111] yet relies on separated heap regions.

Generalization: While very different abstraction may need be developed for more complex kinds of data structures, we remark that the basic operations on them are not fundamentally different. Therefore, it seems possible to extend our static analysis frameworks so as to integrate other notions of summaries, under the constraint that operations to pack and unpack them can be defined.

6.3 Towards a standalone abstract domain

The abstraction proposed in chapter 2 proved flexible and easy to extend, so as to deal with non trivial features of programming languages, such as low-level memory manipulations (chapter 4), call-stack summarization (chapter 5), sharing (section 6.2.4) but it also turned out rather hard to implement. So far, we worked only on prototype implementations, as it was not possible to analyze very large programs without a solid handling for a wide range of

program features (including, e.g., arrays and strings which are not supported in Xisa as of today).

Therefore, an important task I am planning to pursue is to revise the current implementation and turn it into a standalone abstraction. In the longer term, this implementation effort should also allow to integrate a shape domain into general purpose or domain specific static analyzers (like Astrée [11]).

The following paragraphs list a few main areas for further research motivated by the goal of improving implementation.

Combination of abstract domains: First, the splitting of the memory (including the call stack) in regions and the abstraction of values with an underlying domain (section 2.3.1) which could be either a numerical domain or a more complex one (e.g., for abstracting arrays) make our abstraction compatible with a wide range of structures. In particular, we have seen in section 4.2.3 that an array domain could be used as a value domain, in order to capture constraints over the contents of an array region.

To take full advantage of this possibility, we need to extend the interfaces which were used in chapter 3; indeed, a domain for abstracting strings has a different signature than a domain for abstracting base values.

Besides, this scheme also has some limitations. For instance, let us consider an array of pointers to structures in located somewhere else in the memory (as in the example shown in section 6.2.5); then, abstract values should not only describe the structures, but also the relations between them. Therefore, much works remains to be done carefully choosing abstract domain interfaces, so as to capture a wide set of structure while not impeding performance.

Scalability and disjunction control: The analysis implemented in Xisa is complex, and needs a lot of tuning to compute optimal results. This tuning is usually the result of strategies (e.g., for deciding which edge to unfold) and/or hard-coded choices (e.g., the definition of the iteration strategies), which always need more improvements.

As shown in section 3.3, unfolding generate disjunctions of abstract elements. However, keeping too many disjuncts incurs a non negligible cost, so an ideal analysis would determine when some disjuncts can be folded together. The Xisa analyzer follows a rather simplistic strategy and currently does not merge disjuncts except at widening points.

Early experiments show that trace partitioning [108] techniques do not infer the optimal folding decisions, as the shape of disjuncts cannot be related solely on executions paths. Therefore, finding criteria based on the shape of abstract values like [78], which would trigger accurate and compact sets of disjuncts requires some additional investigation. Ideally, folding should be decided based on semantic criteria, and identifying easily computable such criteria is an important future task.

Abstract domain verification: Our abstract domain relies on a complex abstraction and on complex algorithm, thus proving a mechanized formalization of this domain would increase the trust in the results produced by analyses relying on it. The experience also shows that such formalization and proving efforts often lead to the design of simpler, more elegant algorithms.

Therefore, I formalized a limited version of the abstract domain using the Coq proof assistant, including the abstraction provided in chapter 2 (with an axiomatized numerical domain), and part of the unfolding and folding operators of chapter 3, yet this represents only a very small part of the domain. In the longer term it should be feasible to prove the abstract domain transfer functions formally, with respect to a formal semantics of C statements [12], which is used in the CompCert compiler [72]. Such a formalization would open the way to a fully verified static analyzer as verified numerical domains already exist [9] for a large subset of C, including complex data structures, which would be of a great interest to the critical embedded systems community.

We remark that the approach proposed in this manuscript in order to decompose abstract domain into simpler ones, to be combined according to carefully designed interfaces should help significantly in making this formalization and proof effort tractable.

Bibliography

[1] Inquiry board traces ariane 5 failure to overflow error. *SIAM News*, 29(8):12, Oct. 1996.

[2] X. Allamigeon, W. Godard, and C. Hymans. Static analysis of string manipulations in critical embedded c programs. In K. Yi, editor, *Static Analysis Symposium (SAS)*, volume 4134 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2006.

[3] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In R. Cousot and D. A. Schmidt, editors, *Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996.

[4] ANSI ISO/IEC. *International Standard – Programming Languages – C*, 1999.

[5] A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In J. Vitek, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer, 2008.

[6] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In W. Damm and H. Hermanns, editors, *Conference on Computer-Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2007.

[7] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *American Institute of Aeronautics and Astronautics (AIAA 2010), Atlanta, Georgia, USA*, Apr. 2010.

[8] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. In *In Third IEEE International workshop UML and Formal Methods, 2010, Shanghai, China*. IEEE, Nov. 2010.

[9] F. Besson, D. Cachera, T. P. Jensen, and D. Pichardie. Certified static analysis by abstract interpretation. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 223–257. Springer, 2009.

[10] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer, 2002.

[11] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, 2003.

[12] S. Blazy and X. Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[13] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping tvla: Making parametric shape analysis competitive. In W. Damm and H. Hermanns, editors, *Conference on Computer-Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 221–225. Springer, 2007.

[14] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 259–270. ACM, 2005.

[15] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programing*, 2(4):407–423, 1992.

[16] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium (SAS)*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003.

[17] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In K. Yi, editor, *Static Analysis Symposium (SAS)*, volume 4134 of *Lecture Notes in Computer Science*, pages 182–203. Springer, 2006.

[18] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In Z. Shao and B. C. Pierce, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 289–300. ACM, 2009.

[19] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages (POPL)*, pages 178–188. ACM, 1987.

[20] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In G. C. Necula and P. Wadler, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 247–260. ACM, 2008.

[21] B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In H. R. Nielson and G. Filé, editors, *Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 384–401. Springer, 2007.

[22] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 296–310. ACM, 1990.

[23] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In O. Grumberg and M. Huth, editors, *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2007.

[24] J.-D. Choi, M. G. Burke, and P. 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. ACM, 1993.

[25] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In K. Jensen and A. Podelski, editors, *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[26] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In Z. Shao and B. C. Pierce, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 302–314. ACM, 2009.

[27] T. Coquand and C. Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988.

[28] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[29] P. Cousot and R. 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)*, pages 238–252, 1977.

[30] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[31] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[32] P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer, 1992.

[33] P. Cousot and R. Cousot. Modular static program analysis. In R. N. Horspool, editor, *Conference on Compiler Construction (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002.

[34] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In S. Sagiv, editor, *European Symposium on Programming (ESOP)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[35] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the astrée static analyzer. In M. Okada and I. Satoh, editors, *Advances in Computer Science - ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.

[36] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design (FMSD)*, 35(3):229–264, 2009.

[37] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with astree. In *Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 3–20. IEEE Computer Society, 2007.

[38] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In T. Ball and M. Sagiv, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 105–118. ACM, 2011.

[39] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages (POPL)*, pages 84–96, 1978.

[40] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In M. Alpuente, B. Cook, and C. Joubert, editors, *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[41] D. Delmas and J. Souyris. Astrée: From research to industry. In H. R. Nielson and G. Filé, editors, *Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.

[42] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond -limiting. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 230–241. ACM, 1994.

[43] A. Deutsch. An overview of semantic models and static analysis techniques for inductive data structures and pointers. In *Symposium on Partial Evaluation and Program Manipulation (PEPM)*, pages 226–229. ACM, 1995.

[44] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In H. Hermanns and J. Palsberg, editors, *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[45] DO-178B: Software considerations in airborne systems and equipment certification. Technical report, Radio Technical Commission on Aviation, 1999.

[46] N. Dor, M. Rodeh, and S. Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 155–167. ACM, 2003.

[47] J. Feret. Static analysis of digital filters. In D. A. Schmidt, editor, *European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2004.

[48] J. Feret. The arithmetic-geometric progression abstract domain. In R. Cousot, editor, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2005.

[49] P. Gardner and M. J. Wheelhouse. Small specifications for tree update. In C. Laneve and J. Su, editors, *WS-FM*, volume 6194 of *Lecture Notes in Computer Science*, pages 178–195. Springer, 2009.

[50] D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In J. Palsberg and M. Abadi, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 338–350. ACM, 2005.

[51] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In K. Yi, editor, *Static Analysis Symposium (SAS)*, volume 4134 of *Lecture Notes in Computer Science*, pages 240–260. Springer, 2006.

[52] E. Goubault. Static analyses of the precision of floating-point operations. In P. Cousot, editor, *Static Analysis Symposium (SAS)*, volume 2126 of *Lecture Notes in Computer Science*, pages 234–259. Springer, 2001.

[53] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In G. C. Necula and P. Wadler, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 235–246. ACM, 2008.

[54] S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In W. Damm and H. Hermanns, editors, *Conference on Computer-Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 379–392. Springer, 2007.

[55] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In R. D. Nicola, editor, *European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2007.

[56] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In J. Ferrante and K. S. McKinley, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 256–265. ACM, 2007.

[57] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In R. Gupta and S. P. Amarasinghe, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 339–348. ACM, 2008.

[58] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 254–263. ACM, 2001.

[59] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. Verification via structure simulation. In R. Alur and D. Peled, editors, *Conference on Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 281–294. Springer, 2004.

[60] S. S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In H. R. Nielson, editor, *Symposium on Principles of Programming Languages (POPL)*, pages 14–26. ACM, 2001.

[61] B. Jeannet, A. Loginov, T. W. Reps, and S. Sagiv. A relational approach to interprocedural shape analysis. In R. Giacobazzi, editor, *Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*, pages 246–264. Springer, 2004.

[62] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *Conference on Computer-Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[63] B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST)*,

volume 3116 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2004.

[64] N. D. Jones. Flow analysis of lambda expressions. In S. Even and O. Kariv, editors, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 1981.

[65] D. Kaestner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the absence of runtime errors. In *In Embedded Real Time Software and Systems (ERTS 2010)*, May 2010.

[66] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976. Springer.

[67] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In U. Kastens and P. Pfahler, editors, *Conference on Compiler Construction (CC)*, volume 641 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 1992.

[68] J. Kreiker, T. Reps, N. Rinetzky, M. Sagiv, R. Wilhelm, and E. Yahav. Interprocedural shape analysis for effectively cutpoint-free programs. Technical report (lncs special issue commemorating harald ganzinger), Queen Mary University of London, 2010.

[69] J. Kreiker, H. Seidl, and V. Vojdani. Shape analysis of low-level c with overlapping structures. In G. Barthe and M. V. Hermenegildo, editors, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5944 of *Lecture Notes in Computer Science*, pages 214–230. Springer, 2010.

[70] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In M. V. Deusen and B. Lang, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 93–103. ACM, 1991.

[71] V. Laviron, B.-Y. E. Chang, and X. Rival. Separating shape graphs. In A. D. Gordon, editor, *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 387–406. Springer, 2010.

[72] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 42–54. ACM, 2006.

[73] T. Lev-Ami and S. Sagiv. Tvla: A system for implementing static analyses. In J. Palsberg, editor, *Static Analysis Symposium (SAS)*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2000.

[74] J.-L. Lions et al. ARIANE 5, flight 501 failure, report by the inquiry board. Technical Report, European Space Agency and Centre National d'Études Spatiales, 1996.

[75] F. Logozzo. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems & Structures*, 35(2):100–142, 2009.

[76] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In H. R. Nielson and G. Filé, editors, *Static Analysis Symposium (SAS)*, volume 4634 of *Lecture Notes in Computer Science*, pages 419–436. Springer, 2007.

[77] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In O. Grumberg and M. Huth, editors, *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2007.

[78] R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In R. Giacobazzi, editor, *Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2004.

[79] R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In R. Cousot, editor, *Conference on Verification, Model Checking, and Abstract Interpretation*

(VMCAI), volume 3385 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2005.

[80] M. Marron, M. V. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In L. J. Hendren, editor, *Conference on Compiler Construction (CC)*, volume 4959 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2008.

[81] L. Mauborgne. Binary decision graphs. In A. Cortesi and G. Filé, editors, *Static Analysis Symposium (SAS)*, volume 1694 of *Lecture Notes in Computer Science*, pages 101–116. Springer, 1999.

[82] L. Mauborgne. Improving the representation of infinite trees to deal with sets of trees. In G. Smolka, editor, *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2000.

[83] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In S. Sagiv, editor, *European Symposium on Programming (ESOP)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005.

[84] B. McCloskey, T. W. Reps, and M. Sagiv. Statically inferring complex heap, array, and numeric invariants. In R. Cousot and M. Martel, editors, *Static Analysis Symposium (SAS)*, volume 6337 of *Lecture Notes in Computer Science*, pages 71–99. Springer, 2010.

[85] M. Might. Shape analysis in the absence of pointers and structure. In G. Barthe and M. V. Hermenegildo, editors, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5944 of *Lecture Notes in Computer Science*, pages 263–278. Springer, 2010.

[86] M. Might and O. Shivers. Analyzing the environment structure of higher-order languages using frame strings. *Theoretical Computer Science (TCS)*, 375(1-3):137–168, 2007.

[87] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. A. Schmidt, editor, *European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004.

[88] A. Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In M. J. Irwin and K. D. Bosschere, editors, *Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 54–63. ACM, 2006.

[89] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[90] D. Monniaux. Verification of device drivers and intelligent controllers: a case study. In C. M. Kirsch and R. Wilhelm, editors, *International Conference on Embedded Software (EMSOFT)*, pages 30–36. ACM, 2007.

[91] A. S. Murawski and K. Yi. Static monotonicity analysis for lambda-definable functions over lattices. In A. Cortesi, editor, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2294 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2002.

[92] C. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications (TLCA)*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[93] V. Perrelle and N. Halbwachs. An analysis of permutations in arrays. In G. Barthe and M. V. Hermenegildo, editors, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5944 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2010.

[94] D. Prountzos, R. Manevich, K. Pingali, and K. S. McKinley. A shape analysis for optimizing parallel graph programs. In T. Ball and M. Sagiv, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 159–172. ACM, 2011.

[95] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In P. Lee, editor, *Symposium on Principles of Programming Languages (POPL)*, pages 49–61. ACM, 1995.

[96] T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In B. Steffen and G. Levi, editors, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2004.

[97] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.

[98] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In J. Palsberg and M. Abadi, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 296–309. ACM, 2005.

[99] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In R. Wilhelm, editor, *Conference on Compiler Construction (CC)*, volume 2027 of *Lecture Notes in Computer Science*, pages 133–149. Springer, 2001.

[100] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 284–302. Springer, 2005.

[101] X. Rival. Parametric abstract domains for shape analysis. The Theory Workshop of the Verified Software (VSTTE'08).

[102] X. Rival. Abstract interpretation-based certification of assembly code. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2575 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2003.

[103] X. Rival. Certification of compiled assembly code by invariant translation. *STTT*, 6(1):15–37, 2004.

[104] X. Rival. Symbolic transfer function-based approaches to certified compilation. In N. D. Jones and X. Leroy, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 1–13. ACM, 2004.

[105] X. Rival. Abstract dependences for alarm diagnosis. volume 3780 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2005.

[106] X. Rival. Understanding the origin of alarms in astrée. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 303–319. Springer, 2005.

[107] X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In T. Ball and M. Sagiv, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 173–186. ACM, 2011.

[108] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *Transactions on Programming Languages And Systems (TOPLAS)*, 29(5), 2007.

[109] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Transactions on Programming Languages And Systems (TOPLAS)*, 24(3):217–298, 2002.

[110] S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Symposium on Principles of Programming Languages (POPL)*, pages 16–31, 1996.

[111] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symposium on Principles of Programming Languages (POPL)*, pages 105–118, 1999.

[112] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[113] A. Simon and A. King. Analyzing string buffers in c. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology (AMAST)*, volume 2422 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2002.

[114] R. Skeel. Roundoff error cripples patriot missile. *SIAM News*, 25(4):11, July 1992.

[115] G. Slabodkin. Software glitches leave navy smart ship dead in the water. *Government Computer News (GCN)*, July 1998.

[116] P. Sotin and B. Jeannet. Precise interprocedural analysis in the presence of pointers to the stack. In G. Barthe, editor, *European Symposium on Programming (ESOP)*, volume 6602 of *Lecture Notes in Computer Science*, pages 459–479. Springer, 2011.

[117] P. Sotin, B. Jeannet, and X. Rival. Concrete memory models for shape analysis. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 267(1):139–150, 2010.

[118] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *DSN*, pages 625–630. IEEE Computer Society, 2003.

[119] A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In R. Cousot and D. A. Schmidt, editors, *Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer, 1996.

[120] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In J. Palsberg and M. Abadi, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 351–363. ACM, 2005.

[121] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In W. Pugh and C. Chambers, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 25–34. ACM, 2004.

[122] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

[123] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In A. Gupta and S. Malik, editors, *Conference on Computer-Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.

[124] G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In L. Aceto and A. Ingólfsdóttir, editors, *Conference on Foundations Of Software Science and Computation Structures (FoSSaCS)*, volume 3921 of *Lecture Notes in Computer Science*, pages 94–110. Springer, 2006.

[125] G. Yorsh, T. W. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In K. Jensen and A. Podelski, editors, *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 530–545. Springer, 2004.

[126] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In G. C. Necula and P. Wadler, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 221–234. ACM, 2008.