Automated verification of heap-manipulating programs with infinite data

Thèse dirigée par: Ahmed Bouajjani et Mihaela Sighireanu

Soutenue le: 8 décembre 2011

Jury
M. Ahmed Bouajjani co-directeur
M. Roberto Di Cosmo président du jury
M. Jean Goubault-Larrecq rapporteur
M. Nicolas Halbwachs examinateur
M. Viktor Kuncak examinateur
M. Andreas Podelski rapporteur
M. Shmuel Sagiv rapporteur
Mme. Mihaela Sighireanu co-directrice
Abstract

In this thesis, we focus on the verification of safety properties for sequential programs manipulating dynamic data structures carrying unbounded data. We develop a logic-based framework where program specifications are given by formulas. First, we address the issue of automatizing pre/post-condition reasoning. We define a logic, called CSL, for the specification of linked structures or arrays, as well as compositions of these structures. The formulas in CSL can describe reachability relations between cells in the heap following some pointer fields, the size of the heap, and the scalar data. We prove that the satisfiability problem of CSL is decidable and that CSL is closed under the computation of the strongest post-condition. Second, we address the issue of automatic synthesis of assertions for programs with singly-linked lists. We define an abstract interpretation based framework which combines a specific finite-range abstraction on the shape of the heap with an abstract domain on sequences of data. Different abstractions on sequences are considered allowing to reason about their sizes, the multisets of their elements, or relations on their data at different positions. We define an interprocedural analysis that computes the effect of each procedure in a local manner, by considering only the part of the heap reachable from its actual parameters. We have implemented our techniques in a tool which shows that our approach is powerful enough for automatic generation of non-trivial procedure summaries and pre/post-condition reasoning.
Résumé

Le développement de techniques rigoureuses et automatiques pour la vérification des systèmes logiciels est une tâche importante. Cette thèse porte sur la vérification des propriétés de sûreté pour des programmes avec mémoire dynamique et données infinies. Elle développe un cadre basé sur la logique où les spécifications des programmes sont données par des formules. Premièrement, nous considérons l’automatisation du raisonnement prédé/préséquent. Nous définissons une logique, appelée CSL, pour la spécification des structures chaînées ou des tableaux, ainsi que des compositions de ces structures. Les formules CSL décrivent des relations d’accessibilité entre les cellules de mémoire, la taille du tas et les données scalaires. Nous prouvons que le problème de la satisfiabilité pour CSL est décidable et que CSL est fermée par le calcul de la post-condition. Ensuite, nous considérons la synthèse automatique d’assertions pour des programmes avec des listes simplement chaînées. Un cadre basé sur l’interprétation abstraite est défini qui combine une abstraction finie sur la forme du tas avec une abstraction sur des séquences de données. Les abstractions sur les séquences permettent de raisonner sur leurs tailles, les multi-ensembles de leurs éléments, ou les relations entre leurs données à des différentes positions. Nous définissons une analyse inter-procédurale qui calcule l’effet de chaque procédure de façon locale sur la partie du tas accessible à partir de ses paramètres. Ces techniques sont implantées dans un outil qui montre que notre approche est assez puissante pour la génération automatique de résumés de procédure non-triviales et le raisonnement prédé/post-condition.
Acknowledgements

First of all I would deeply like to thank my supervisors Mihaela SIGHIREANU and Ahmed BOUAJJANI for their guidance and assistance during my phd. Together with them I’ve walked on the solid ground of decidability and through the vast world of abstractions. It was an enriching experience and a pleasure to work with you!

I would like to thank the reviews for their time and for their comments that helped me improve this manuscript. Also, I would like to thank all the members of the jury for accepting to participate to the defense of my phd thesis.

During my phd in LIAFA I was part of the Verification team where I met Tayssir Touilli, Peter Habermehl, Eugène Asarin, Florian Horn and Arnaud Sangnier, amazing people together with whom I’ve attended seminars, project meetings, and lunch breaks. Noëlle Degado and Nathalie Rousseau took care of all the administrative problems that arose during my phd. I would like to thank them for their availability and efficiency.

I would like to thank all the phd students with whom I’ve shared, during these years, an office, a coffee break or a Friday cake, and especially to the ones that were next by me during the writing of this manuscript Adeline, Irene, Robin, Antoine, Denis, Hervé, Medhi, Luc, Thach, Stéphane. Also, Claire, Marie, Mathilde, and Faouzi offered me a very warm welcome during my first days in Paris and helped me accommodate to what today is a familiar environment.

Along these years I have gained scientific experience but also a few friends Linda, Michael, and Roland. Our common interest in discovering Paris and its surroundings brought me closer to them in the beginning. From there to daily activities and lots of shared stories, it was just the logical sequence of facts. Despite the distance, my old friends, Mihaela, Ionut, and Robert, stood really close to me during these years. To all of them I would like to thank. Last but far from being the least, I would like to thank Costin, with whom I’ve shared all the happy or sad, and funny or scary, moments of my phd.

Finally, I would like to thank my parents Mihaela and Cezar for their patience, care, and understanding. Also, I would like to thank my grandmother, to whom I dedicate this manuscript.
# Contents

1 Introduction ........................................ 9
   1.1 Motivation ........................................ 9
      1.1.1 Pre/post-condition reasoning ................. 11
      1.1.2 Synthesis of program assertions ............... 11
   1.2 Contributions .................................... 12
      1.2.1 Pre/post-condition reasoning ................. 13
      1.2.1.1 Logic gCSL ................................ 13
      1.2.1.2 Logic CSL ................................ 14
      1.2.2 Synthesis of program assertions ............... 15
      1.2.2.1 Abstract domains for programs manipulating singly-linked lists over integers .......... 16
      1.2.2.2 Compositional inter-procedural analysis for programs manipulating singly-linked lists over integers .......... 19
   1.3 Thesis organization ................................ 25

2 Related work ........................................ 27
   2.1 Decidable logics for program verification ............ 27
   2.2 Program analysis techniques ....................... 31

3 Preliminaries ....................................... 37
   3.1 Transition Systems ................................ 37
   3.2 Verification of transition systems .................. 39
   3.3 Analysis of transition systems ..................... 39
      3.3.1 Fixed point computations ....................... 40
         3.3.1.1 Fixed point approximations ............... 41
         3.3.1.2 Chaotic iteration .......................... 44
         3.3.1.3 Combinations of abstract domains .......... 46
   3.4 Numerical abstract domains ....................... 48

4 Programs with dynamic data structures ............ 51
   4.1 Data types and program type system ............... 52
   4.2 Programs without procedure calls .................. 54
      4.2.1 Program syntax ................................ 54
      4.2.2 Examples ...................................... 56
      4.2.3 Program semantics .............................. 59
         4.2.3.1 Intra-procedural control flow graph ........ 59
         4.2.3.2 Representation of program configurations .... 60
         4.2.3.3 Concrete transformers for program statements .... 63
         4.2.3.4 Reachability collecting semantics .......... 66
   4.3 Extending programs with procedure calls .......... 68
      4.3.1 Syntax ......................................... 68
      4.3.2 Examples ...................................... 70
      4.3.3 Program semantics .............................. 73
1.1 Motivation

Software systems are ubiquitous and their development and their spread is crucial to many essential activities in our modern society (communication, transportation, energy, health, business, finance, etc.). It is therefore of the highest importance to have rigorous methods and automated techniques for their design and verification, allowing to ensure a high degree of reliability and of confidence in their behaviors.

However, programs can have extremely complex behaviors, making their design and verification very difficult and highly challenging tasks, both from the theoretical and from the practical point of view. This complexity is due to various aspects, either related to the control or to the data. Concerning control, programs can have (potentially recursive) procedure calls, concurrent threads, etc. As for the data aspects, programs may use variables over unbounded domains (e.g., integers or reals) and manipulate unbounded (dynamic) data structures such as singly/doubly-linked lists, arrays, as well as composite data structures, (e.g., lists of lists or arrays of lists) that are commonly used in software systems like operating systems, data bases, etc.

Due to these aspects, the structure of the program configurations can be quite complex, of unbounded size and shape. This makes reasoning about the potentially infinite set of reachable configurations very hard in general. From the theoretical point of view, this means that in general verification problems are undecidable, and from the practical point of view, even in the decidable cases, these problems can be of a high computational complexity. Therefore, the challenge is to find efficient and powerful verification techniques that are applicable to significant classes of programs and properties.

Program properties can be divided as usual into termination/liveness properties and safety properties establishing the partial correctness of the program. In the case of programs manipulating dynamic data structures, one can distinguish intrinsic correctness properties such as absence of null dereference that do not need to be specified explicitly, from program-specific properties that correspond to the properties of their configurations at different points in their execution, and especially the relation between their inputs and outputs. In order to check the program-specific properties, we need to express them as assertions on the program configurations (e.g., invariants), using typically a logic-based specification language. This language must be rich enough to allow expressing constraints both on the shape of the data structures (for instance the fact that the structure is a linked list, doubly-linked list, a list of lists, etc.) as well as on the data that are stored at different locations in these structures.

In this thesis, we focus on the problem of verifying safety properties of sequential programs manipulating dynamic linked data structures carrying infinite data. Our aim is to develop a framework for formal specification and automatic verification of such
CHAPTER 1. INTRODUCTION

programs. We assume that programs can be annotated by assertions expressing their specifications, and our aim is to provide techniques that allow to verify automatically that these specifications are satisfied.

Let us see an example. Consider the procedure given in Figure 1.1 which initializes a singly-linked list with the first even numbers. The specification states that whenever the procedure receives as input a (potentially empty) list it returns a list of the same length such that the values of the data fields form an ascending sorted sequence containing the first even numbers. The properties satisfied by the input of the procedure are given after the keyword requires while the description of the output is given after the keyword ensures. We are interested in proving that every execution of this procedure satisfies this specification.

A logic which is expressive enough for this specification should be able to state (1) properties on the pointer variables, like aliasing \( p = q \), where \( p, q \) are two pointer variables, or reachability relations between pointers \( p \rightarrow q \), where \( \rightarrow \) is the pointer field in the list type, (2) properties on the values of data fields, which come from an unbounded domain such as the set of integers, and (3) for objects of record type, relations between the values of their data fields and their offset. For instance, the following first order formula describes the output of the procedure initEven as specified at line 9:

\[
\Bigl((head = \text{null}) \lor \bigl(dt(head) = 0 \land (head \rightarrow_{\text{next}} \text{null}) \land \forall y \ \forall l. (head \rightarrow_{\text{next}} l = \text{null}) \implies dt(y) = 2 \times l\bigr)\Bigl),
\]

where \( head \rightarrow_{\text{next}} \text{null} \) states that \( head \) points to a memory cell that reaches \( \text{null} \) following the pointer field \( \text{next} \). \( y \) is a variable denoting a list element, \( l \) is an integer.
variable that denotes the number of allocated list elements between head and y, and dt(y) represents the value of the data field dt of the list element denoted by y.

1.1.1 Pre/post-condition reasoning

A first issue we address is automatizing pre/post-condition reasoning, i.e., given an annotated program, checking the validity of the assertions, assuming that the user/programmer provides loop invariants, and procedure specifications. Already this task is not obvious. Indeed, based on the inference rules from the Hoare logic [Hoare 1969], this reduces to checking the validity of Hoare triples of the form

$$\{\varphi_{pre}\} \text{St} \{\varphi_{post}\},$$

where St is a basic program statement (assignments, boolean conditions) and $\varphi_{pre}, \varphi_{post}$ are two formulas in the specification logic, describing sets of program configurations. The validity of such triples is equivalent to the validity of $\text{post}(\varphi_{pre}, \text{St}) \implies \varphi_{post}$, where post is the strongest post-condition, i.e. the set of program configurations reachable from the ones described by $\varphi_{pre}$ by executing the statement St. To automatically check the validity of Hoare triples one has to (1) automatize the computation of the strongest post-condition and to (2) define a decision procedure to test the entailment between two formulas in the specification logic.

Usually, data structures are implemented using user-defined record types which contain a set of data fields, representing scalar data such as integers, reals, etc., and a set of pointer fields, representing references to other objects of record type. In this thesis, programs are modeled by transition systems where states are graphs that describe the allocated memory. The configurations of the allocated memory are typically described by graphs where vertices represent objects of record type and edges represent the values of the pointer fields. Moreover, the vertices are labeled by the values of the data fields.

To reason about the reachability relation between vertices in the graphs denoting program configurations, one would ideally need the first order logic with transitive closure. This logic is known to be undecidable [Börger 1997]. Even if we consider programs that manipulate only linear data structures, similar undecidability results were proven in [Bradley 2006, Bouajjani 2007, Habermehl 2008]. Moreover, we want our techniques to scale to large programs therefore the complexity of the validity test is an important issue. This leads to the challenging problem of finding a good compromise between expressiveness, decidability, and complexity.

1.1.2 Synthesis of program assertions

An even more challenging issue is the automatic synthesis of program assertions such as loop invariants and procedure summaries. This would allow to improve the automation of the framework by providing a tool for discovering relevant properties of the program configurations and relations between the variables at different execution points. The challenge is to design powerful and efficient techniques allowing to synthesize invariants and summaries that are strong enough to establish the desired properties.

Then, an important issue is the fact that in many cases, the useful invariants and program assertions must be expressed in languages that are beyond the known decidable (fragments of) logics. In general, program specifications combine various kinds of constraints on data structures such as order relations between scalar values, constraints on
Chapter 1. Introduction

Multisets of data, constraints on the sizes of the sub-structures, etc. For example, the specification of a sorting procedure for singly-linked lists contains the sortedness property but also the fact that the procedure preserves the multiset of values from the input list. In such cases, it is important to develop sound approximate decision procedures. When applied to formulas coming from decidable fragments, an approximate decision procedure might sacrifice completeness for efficiency. Usually, this is done by avoiding the exploration of the corner cases, that might not even show up in practice.

Therefore, we need to manipulate constraints in rich specification languages. In order to do that, a natural idea is to investigate approximate but sound procedures based on abstraction in order to analyze programs and synthesize automatically their invariants in such languages. Here again the challenge is to design abstraction techniques that are accurate enough to handle all typical programs used in practice.

We study techniques based on abstract-interpretation [Cousot 1977a, Cousot 1979], which allows designing a static analysis sound by construction, based on a sound approximation of the program semantics. The framework of abstract interpretation offers a way of automatizing the synthesis of invariants and program assertions. Indeed, these assertions can be seen as abstractions of the sets of possible reachable configurations at the considered point in the program, and these abstractions can be computed automatically using fixed point evaluations in appropriate abstract domains. Other methods to generate invariants differ in the assumed user guidance. For instance, in predicate abstraction based techniques, the user has to supply the predicates and the tool searches for invariants which are boolean formulas over the considered predicates. In constraint solving based techniques, the user provides a template (e.g., a linear constraint with unknown coefficients) and the tool discovers invariants which are instances of this template (e.g., it discovers the values of the template coefficients). From a slight different perspective, interpolation based methods strongly rely on decision procedures. In this case, the generated assertions are constrained to belong to a decidable logic.

1.2 Contributions

The contributions of the thesis can be summarized as follows:

- A logic-based framework for the specification and the verification of sequential programs with composite multi-linked lists. We propose a logic for expressing properties of such programs concerning the shape of the structures as well as their sizes and the data values they contain. We show that this logic is decidable and suitable for pre/post-condition reasoning.

- A framework based on abstract interpretation for the analysis of sequential programs manipulating lists with data. The framework offers abstract domains allowing to reason about various constraints such as, ordering constraints expressed using universally quantified first-order formulas, multiset constraints, size constraints, etc.

- Inter-procedural analysis techniques allowing to synthesize in a modular way procedure summaries and invariants for sequential programs manipulating lists with data. The developed techniques are accurate and allow to generate automatically complex program assertions such as sortedness, preservation of elements, equality
1.2. CONTRIBUTIONS

between structures, etc., but also to detect null pointer dereferences and dangling pointers.

- A tool called Celia based on the framework above allowing to verify C programs. The tool is a plug-in of the Frama-C platform.

1.2.1 Pre/post-condition reasoning

We introduce a framework for deductive verification where the program configurations are represented by graphs and the annotations and the specification are given in a first-order logic over graphs. Concerning the expressiveness of the logic we define, it contains:

- reachability predicates (or ordering predicates) between variables interpreted as vertices in the graph,
- constraints on the length of the paths in the graph, and
- constraints on the scalar data labeling the vertices in the graph (which may come from an unbounded domain) stored in the data structures.

The validity of $\text{post}(\varphi_{\text{pre}}, St) \implies \varphi_{\text{post}}$, where $St$ is a basic program statement, $\varphi_{\text{pre}}, \varphi_{\text{post}}$ are two formulas in the considered logic, and post is the strongest post-condition (which is equivalent to the validity of the Hoare triple $\{ \varphi_{\text{pre}} \} St \{ \varphi_{\text{post}} \}$) is reduced to the unsatisfiability of $\text{post}(\varphi_{\text{pre}}, St) \land \neg \varphi_{\text{post}}$. We provide a procedure that computes automatically the strongest post-condition and a procedure for deciding the satisfiability of a formula in this logic.

1.2.1.1 Logic $gCSL$

We define a multi-sorted first order logic on graphs, called General Composite Structures Logic ($gCSL$ for short), to reason about configurations of programs manipulating dynamically allocated composite data structures. For simplicity, we assume that scalar data manipulated by the program comes from a (possibly infinite) domain $D$. To express constraints on the scalar data, $gCSL$ is parameterized by a first-order logic $\text{FO}(D, O, P)$ over $D$, where $O$ is the set of operation symbols and $P$ is the set of predicate symbols.

The models of the logic are graphs where each vertex is typed. The vertex types correspond to the record data types defined in the program. Pointer fields define successor relations in the structure (they correspond to edges in the graph) and values of data fields are represented by vertex labels.

Reachability predicates: Location variables are used to refer to vertices in the graphs. They can be existentially or universally quantified. Formulas in $gCSL$ can express that two vertices are related by a path following some set of pointer fields. For instance, the formula $x[{\text{next}, \text{prev}}] \rightarrow z$ says that the vertices $x$ and $z$ are related by a doubly linked path, where next (resp. prev) is the forward (resp. backward) successor field. Terms of the form $a[i]$ denote array elements and are interpreted into vertices such that (1) $a$ is an array variable interpreted into the vertex corresponding to the first element of the array and (2) $i$ is an index variable representing the length of the path (following a distinguished array field) between the vertex representing $a$ and the vertex representing $a[i]$. Array and index variables can be existentially or universally quantified. For example, the following
CHAPTER 1. INTRODUCTION

formula describes an array of doubly-linked lists like in Figure 1.2, where \( \text{dll} \) is the pointer field of any array element that points to an element of some doubly-linked list:

\[
\exists a \, \forall i \, \exists x, z. a[i] \xrightarrow{\text{dll}} x \wedge (x \xrightarrow{\text{next, prev}} z \wedge z \xrightarrow{\text{next, 1}} \text{null}) \vee (x \xrightarrow{\text{next, 1}} z \wedge z = \text{null}).
\]

Figure 1.2: An array of doubly-linked lists.

**Length constraints:** To reason about distances between elements in the arrays, arithmetical constraints on index variables are used. This can be generalized to linked structures using formulas of the form \( x \xrightarrow{[\text{next, prev}], l} z \), where \( l \) is an index variable representing the length of the path between \( x \) and \( z \). For example, if we consider the array of doubly-linked lists given in Figure 1.2, the following formula states that the list pointed to by the first element of the array \( a \) has strictly more elements than the list pointed to by the second element of the array \( a \):

\[
\exists a \, \exists x, z. a[0] \xrightarrow{\text{dll}} x \wedge a[1] \xrightarrow{\text{dll}} z \wedge x \xrightarrow{\text{next, 1}} \text{null} \wedge z \xrightarrow{\text{next, 1}} \text{null} \wedge l_x < l_z.
\]

**Data constraints:** The constraints on the values of the data fields and the values of the program scalar variables are formulas in the underlying (first-order) data logic. They are built over data terms of the form \( \text{data}(x) \), where \( x \) is a location variable and \( \text{data} \) is a data field associated with the record type of \( x \), and program scalar variables. For example, the following formula states that a doubly-linked list of integers starting in the location denoted by \( x \) is strictly sorted:

\[
(x \xrightarrow{\text{next, prev}} z \wedge z \xrightarrow{\text{next, prev}} \text{null} \wedge x \xrightarrow{\text{prev, 1}} \text{null}) \wedge \\
\forall y, y'. (x \xrightarrow{\text{next}} y \xrightarrow{\text{next}} y' \wedge y \neq y') \implies \text{data}(x) < \text{data}(y) < \text{data}(y'),
\]

where the underlying logic is Presburger arithmetics, \( \text{data} \) is an integer data field, and \( < \) is the order relation over integers. For more details on the syntax and semantics of \( g\text{CSL} \) please see Section 4.4.

1.2.1.2 Logic CSL

We define a fragment of \( g\text{CSL} \), called Composite Structures Logic (CSL for short) [Bouajjani 2009a], and we prove that the satisfiability problem of CSL is decidable whenever the underlying data logic is decidable. Moreover, we prove that CSL is closed under the computation of the strongest post-condition in the considered class of programs and that the post-condition is effectively computable.

To obtain the decidability result, the quantification over positions (i.e., location, array, and index variables) in CSL formulas is restricted according to an ordered partition on
1.2. CONTRIBUTIONS

the vertex types. This partition decomposes the heap graph into classes and assigns
to each vertex in the graph a level. (Notice, that we do not require that the induced
graph by this partition be a tree.) Then, roughly speaking, by associating a level to each
variable, the quantification part of CSL formulas has the form $\exists^*_k \forall^*_k \exists^*_k-1 \forall^*_k-1 \ldots \exists^*_1 \forall^*_1$. Here it is important to notice that we allow some form of alternation between universal
and existential quantifiers on positions, but this alternation is restricted according to
the considered ordered partition of the structure. Allowing such quantifier alternation is
important for the definition of (1) initial conditions, and of (2) precise invariants.

For example, consider again the array of doubly-linked lists from Figure 1.2. Choosing
the partition that associates level 1 to the vertices belonging to the doubly-linked lists
and level 2 to the vertices representing elements of the arrays, we obtain that the following
formula belongs to CSL:

$$\exists a \forall i \exists x, z. a[i] \xrightarrow{\text{dll}} x \land x \{\text{next, prev}\} \rightarrow z \land \text{flag}(x) + \text{flag}(z) \geq 2.$$  

There are some other restrictions on the use of reachability predicates and length con-
straints in CSL formulas which, roughly, forbid properties that relate any two successive
vertices in the heap graph (for details please see Section 5.2). For example, in CSL, the
values of the universally quantified index variables can be related only by order constraints
(we can compare two universal variables or we can compare a universal variable to an
expression built using existential variables). On existentially quantified index variables,
CSL allows any form of linear constraint. For example,

$$\exists x, x', z, z'. \exists l, l'. x \{\text{next, prev}\}, l \rightarrow z \land x' \{\text{next, prev}\}, l' \rightarrow z' \land ((l = l') \lor (l + l' \leq 9)).$$

**Satisfiability:** The satisfiability problem of CSL is reduced to the satisfiability prob-
lem of its underlying data logic. The reduction is essentially based on computing a
finite number of minimal-size models for the given formula, and on interpreting universal
quantifiers on the positions of these models. Defining such minimal models is nontrivial
in presence of shape constraints (reachability predicates), of list and array length con-
straints, as well as of nesting of data structures. The proof is given in Section 5.2.3 and
Section 5.2.5.

**A fragment of CSL closed under negation:** CSL is not closed under negation,
therefore, we identify ICSL a fragment of CSL for which the invariant checking problem
and the validity of Hoare triples are decidable. This fragment is useful to prove partial cor-
rectness of programs using the deductive approach. Even if ICSL is more restrictive than
CSL, it preserves its main expressibility features, that is, it allows quantifier alternation,
reachability predicates, size and data constraints (it is formally defined in Section 5.2.2).

1.2.2 Synthesis of program assertions

We investigate analysis techniques to automatically discover program invariants and pro-
cedure summaries which describe properties of the shape of the allocated memory but
also on the data stored in each memory cell. Mixing these two aspects in an automatic
synthesis procedure is a challenging task. We restrict the class of program such that they
manipulate only singly-liked lists with integer data. These programs are complex enough
to show the main difficulties that one has to face when reasoning about this combination.
CHAPTER 1. INTRODUCTION

1.2.2.1 Abstract domains for programs manipulating singly-linked lists over integers

To compute the set of reachable states of the program, we use algorithmic approaches based on fixed points computations, and to prove the specification automatically, without the intervention of the user.

We consider in this thesis program static analysis, based on abstract interpretation \cite{Cousot1977}, which is a compile-time algorithmic method for computing sound over-approximations of the set of values or behaviors occurring at run-time when the program is executed. An approximate symbolic execution of the program is performed until a property that remains unchanged by further executions of the program is reached. The discovered properties are inductive invariants by construction. In order to guarantee termination, the method uses an extrapolation operator called widening.

To instantiate the abstract interpretation framework, one has to define an abstract domain, that is a class of properties that form a lattice, together with transformers that manipulate them, corresponding to the program statements, which define a sound approximation of the program semantics. If the lattice does not have finite height, then the abstract domain must be equipped with a widening operator.

To this, our main challenge is to define abstract domains which (1) capture properties of programs with dynamically allocated data structures and (2) are equipped with widening operators that ensure the termination of the fixed point computation. Each abstract domain corresponds to a particular approximation of the program semantics. Therefore, the choice of the abstract domain depends on the properties that need to be inferred.

**Abstraction principle:** We define a generic abstract domain $\mathcal{A}_{\text{HS}}(\mathcal{A}_W)$, whose elements are called the abstract domain of heap sets, for reasoning about dynamic singly-linked lists with unbounded data \cite{Bouajjani2010}. The elements of this abstract domain represent (relations between) programs configurations. They are sets of abstract heaps, which are pairs composed of a heap backbone and an abstract data constraint.

The heap backbone is an abstraction of the heap graph (the graph representing the allocated memory) where only a bounded number of nodes are kept, including all sharing nodes and all the nodes pointed to by program variables. An edge in the backbone represents a path (without sharing nodes) relating the source and target nodes in the original heap. Figure 1.3(b) shows the heap backbone associated to the memory configuration given in Figure 1.3(a) (which is a memory configuration reachable during the execution of the procedure initEven from Figure 1.1) and contains a list pointed to by the pointer variable head whose 4th element is pointed to by headi; $\sharp$ is a distinguished node that denotes the constant null). This heap backbone contains only the nodes pointed to by program variables. Also, for any node $n$, the sequence associated to $n$ is the sequence of values on the path represented by the edge that starts in $n$.

The data constraint is given as an element of some abstract domain $\mathcal{A}_W$ and allows to specify properties of the data sequences represented by the edges of the heap backbone. The abstract domain $\mathcal{A}_W$ is intended to abstract the sequences of data in the lists by capturing relevant aspects such as their size, the sum or the multiset of their elements, or some class of constraints on their data at different (linearly ordered or successive) positions. In the following, a sequence of data values is also called a word. An abstract domain $\mathcal{A}_W$ whose elements represent sequences of data, is called a data words abstract domain, $\mathbb{D}W$-domain for short. We identify an interface for $\mathbb{D}W$-domains, such that any
1.2. CONTRIBUTIONS

(a) Memory configuration for initEven

(b) A heap backbone

(c) An abstract heap

Figure 1.3: The abstraction of a memory configuration.

abstract domain implementing this interface can be combined with the heap backbone in order to obtain an abstract domain of heap sets. For example, Figure 1.3(c) contains an abstract heap representing the memory configuration in Figure 1.3(a). It uses a universally-quantified formula $\varphi$ to describe the list segments abstracted by the edges of the heap backbone. This formula expresses the fact that any list element $y$ between head and headi contains an integer which is twice the length of the path between head and $y$.

**Data words abstract domains:** We introduce several $\mathcal{D}W$-domains corresponding to different ways of abstracting the content of the lists:

1. $\mathcal{A}_U$ whose elements are formulas in an universally quantified fragment of the first-order logic over data words,
2. $\mathcal{A}_M$ whose elements are conjunctions of equality constraints between unions of multisets of data in the words,
3. $\mathcal{A}_S$ whose elements are conjunctions of equality constraints between the sum of data of different words.

**$\mathcal{D}W$-domain $\mathcal{A}_U$:** The elements of $\mathcal{A}_U$ are conjunctions between a quantifier free part, denoted $E$, and a conjunction of universally quantified formulas in the form of implication:

$$E \land \bigwedge \forall y. \ (P \Rightarrow U).$$

The quantifier-free formula $E$ contains constraints on the data program variables or on the values of the data fields stored in the locations pointed to by pointer variables. Then, $y$ is a vector of variables interpreted to positions in the words and $P$ is a constraint on the positions (seen as integers) associated with the $y$’s. Finally, $U$ is a constraint on the data values at the positions denoted by $y$ and possibly other data variables like the ones constrained in $E$.

We assume that $\mathcal{A}_U$ is parametrized by:

- a fixed set of templates $\mathcal{P}$ (such as order constraints or difference constraints) such that any constraint $P$ from the left-hand side of an universally quantified implication is obtained from a template in $\mathcal{P}$ by applying a variable substitution and
• some numerical abstract data domain \( \mathcal{A}_Z \) such that \( E \) and \( U \) are represented as objects in \( \mathcal{A}_Z \) (examples of numerical abstract data domains are the Octagons abstract domain [Miné 2006], the Polyhedra abstract domain [Cousot 1978], etc.).

The formulas that we consider belong to a logic that has an undecidable satisfiability problem (also the validity of the implication between two such formulas is in general undecidable). Our approach is to define a sound procedure for checking the entailment, based on the syntax of formulas, which uses the entailment in \( \mathcal{A}_Z \). This procedure checks the entailment between the quantifier free parts and the entailment of the right hand side of the implications with the same guard (for more details please see Section 6.4).

An analysis using the abstract domain \( \mathcal{A}_{\text{HS}}(\mathcal{A}_W) \) is able to capture typical properties of singly linked lists such as sortedness:

\[
\forall y_1, y_2. \text{head} \xrightarrow{\text{next}} y_1 \xrightarrow{\text{next}} y_2 \implies \text{data(head)} \leq \text{data}(y_1) \leq \text{data}(y_2), \quad (1.2.1)
\]

lists equality \( \forall y_1, y_2. \ (\text{head} \xrightarrow{\text{next}} y_1 \land \text{head} \xrightarrow{\text{next}} y_2 \land y_1 = y_2) \implies \text{data}(y_1) = \text{data}(y_2), \)

\[
\land \text{data(head)}_1 = \text{data(head)}_2 \land \text{head} \xrightarrow{\text{next,1}} \text{null} \land \text{head} \xrightarrow{\text{next,1}} \text{null} \land l_1 = l_2 \text{ or even more complex relations between data such as the Fibonacci property:}
\]

\[
\forall y_1, y_2, y_3. \text{head} \xrightarrow{\text{next}} y_1 \xrightarrow{\text{next,1}} y_2 \xrightarrow{\text{next,1}} y_3 \implies \text{data}(y_1) = \text{data}(y_2) + \text{data}(y_1). \quad (1.2.2)
\]

\( \mathcal{D}_W \)-domain \( \mathcal{A}_S \): Formulas in \( \mathcal{A}_S \) are linear constraints over program data variables, data terms denoting the values of the data fields and terms representing the sum of integers of a list segment. For example, an interesting property of the Fibonacci sequence \( \{f_i\}_{i \geq 1} \) is that \( \sum_{i=1}^{n} f_i = 2f_n + f_{n-1} - 1 \). This is not expressible in the \( \mathcal{A}_U \) domain, but an analysis that uses \( \mathcal{A}_S \) as data word domain is able to capture it. It can be expressed by the following constraint

\[
\text{data(head)} + \sum_{y \xrightarrow{\text{next,1}}} \text{data}(y) = 2 \times m_2 + m_1 - 1, \quad (1.2.3)
\]

where \( m_2 \) and \( m_1 \) are data variables denoting the values of the last two elements of the sequence (for more details please see Section 6.6).

\( \mathcal{D}_W \)-domain \( \mathcal{A}_{\text{ms}} \): Using the multiset abstract domain the analysis captures multiset equalities, like

\[
\text{ms(head)}_i = \text{ms(head)}_o,
\]

which for example is a part of the specification of a sorting procedure where \( \text{head} \) denotes the input list, \( \text{head}_o \) the output sorted one, and \( \text{ms(head)}_i \) denotes the multiset of integers of the list pointed to by \( \text{head}_i \). Other multiset constraints express relations between the multiset of different lists such as

\[
\text{ms(head)} = \text{ms(gr)} \cup \text{ms(sm)} \quad (1.2.4)
\]

where \( \text{ms(head)} \) represents the multiset of the elements of the list pointed to by \( \text{head} \), which equals the union between the multiset of the list pointed by gr and the multiset of the list pointed to by sm (for more details please see Section 6.5).

Logic for \( \mathcal{A}_{\text{HS}}(\mathcal{A}_W) \): We show that the elements of \( \mathcal{A}_{\text{HS}}(\mathcal{A}_W) \) correspond to a fragment of the first-order logic with reachability predicates, denoted SL3W, i.e., for any
1.2. CONTRIBUTIONS

There exists a formula in $\text{SL}^3 W$ whose models are exactly the graphs abstract by $\overline{H S}$. $\text{SL}^3 W$ is parametrized by a logic on sequences, corresponding to the logic of elements in $A_W$. The logic $\text{SL}^3 U$, defined for $A_W = A_U$, is a fragment of $g\text{CSL}$. When $A_W = A_M$, the logic $\text{SL}^3 W$ is a fragment of an extension of $g\text{CSL}$ with reachability predicates of the form $x^{\text{next}, ms} \rightarrow z$, where $x, z$ are location variables, $\text{next}$ is the pointer field defining the successor in the list, and $ms$ is a variable interpreted as the multiset of data in the list segment between $x$ and $z$. Similarly, one could extend $g\text{CSL}$ to include formulas in $\text{SL}^3 S$.

Moreover, we identify a fragment of $\text{SL}^3$ parametrized by $A_U$ formulas, that belongs to $\text{ICSL}$, when the latter is parametrized by a logic over integers. Roughly, this fragment is obtained by restricting the set of patterns to order relations such as $\leq$ and $<$ and no arithmetical constraints. Also, relations between position variables are eliminated from the right hand side of the implications.

1.2.2.2 Compositional inter-procedural analysis for programs manipulating singly-linked lists over integers

We introduce a compositional inter-procedural analysis for cut-point free programs manipulating singly-linked lists [Bouajjani 2010, Bouajjani 2011]. Roughly, cut-point free programs have the property that any location from the memory visible to a procedure at call time is reachable from a pointer variable in the context only by paths passing through actual parameters.

Given a program as a collection of potentially recursive procedures, we develop an analysis over the domain of abstract heap sets $A_{\text{HS}}(A_W)$, which considers each procedure independently. It computes procedure summaries which are reused at every invocation of the procedure.

1.2.2.2.1 Generating universally-quantified formulas

The analysis that we introduce is based on (a) unfolding the structures in order to reveal the properties of some internal nodes in the lists, which makes necessary to introduce in the structures some nodes, called simple nodes, others than the sharing nodes or those pointed to by variables, and then (b) folding the structures, to keep the graphs finite, by eliminating the simple nodes and in the same time collecting the informations on these nodes using a formula that speaks about sequences of data. The analysis is iterated several times, which may lead to additional unfoldings and foldings. Then, widening techniques on numerical domains are used in order to force termination.

We define sound abstract transformers for the statements in the class of programs we consider. The folding step corresponds to a quantifier elimination and it is actually quite delicate and special care has to be taken in order to keep preciseness. In particular, this is the crucial step that allows to generate universally quantified properties from a number of relations between a finite (bounded) number of nodes.

We illustrate the unfold/fold mechanism on the procedure $\text{initEven}$ from Figure 1.1. We analyze this program using the abstract domain of heap sets, $A_{\text{HS}}(A_U)$, where $A_U$ is parametrized by (1) a pattern (for the guards of the universally quantified implications) of the form $\text{hd}^{\text{next}} \Rightarrow y$, which states that $y$ denotes a node in the list segment starting in $\text{head}$ and (2) the Polyhedra numerical abstract domain. The analysis begins to unroll the loop of the procedure starting from the abstract heap given in Figure 1.4. This abstract
heap is an abstraction for the set of all heap graphs (graphs that model the allocated memory) that consist in a path between a vertex labeled by head, headi and the special node ♯ which denotes the constant null. Therefore, the nodes of the abstract heap denote memory locations while edges denote paths in the heap graph. Every execution

Figure 1.4: Initial configuration for the procedure initEven

of the statement headi=head->next in the loop creates two heap graphs: the first one corresponds to the case when headi points to null (the list traversal ends) and the second one unfolds the structure, i.e., introduces a new node which is pointed to by headi. Figure 1.5 shows the abstract heaps resulting after unrolling once the loop and Figure 1.6 shows the abstract heaps obtained after unrolling three times the loop (edges labeled by 1 denote paths of length 1). The formulas in $A_U$ associated to these abstract heaps do

Figure 1.5: Abstract heaps obtained after the first unrolling of the loop of initEven.

Figure 1.6: Abstract heaps obtained after the third unrolling of the loop of initEven.

not contain universally-quantified formulas; they consist in a quantifier-free formula that constrains only the values of the data fields associated to the nodes of the abstract heap. For simplicity, we omit them and we represent the values of the data fields as labels on the nodes.

The size of the list pointed to by head is potentially unbounded, so the size of the heap backbones grows at each unrolling. One way to ensure the termination of the fixed point computation is to design an analysis that manipulates only heap backbones such that every node is either labeled by a program variable or it is a sharing node (i.e., it has two predecessors). With this restriction and the fact that we assume a garbage collector, the size of the heap backbones is bounded by the number of pointer variables used in the program. However, this restriction can be relaxed by allowing heap backbones that contain a number of nodes which are neither pointed to by program variables nor sharing nodes bounded by a fixed constant $k$. In fact, for this example we consider $k = 1$. Notice that after the third unrolling of the loop, the heap backbones contain two nodes,
which are neither pointed to by program variables nor sharing nodes. To keep the size of the abstract heaps bounded, the analysis eliminates these nodes but, before that, it collects the information that the unfolding of the loop revealed about them. This step is called folding the structure. It generates a universally quantified formula that describes the data properties of the nodes that are eliminated. Because it is parametrized by the pattern $\text{hd} \xrightarrow{\text{next}} y$, the analysis generates a universally quantified formula of the form $\forall y. \text{hd} \xrightarrow{\text{next}} y \Rightarrow U$, where $U$ is some numerical constraint on the value of the data field $\text{dt}(y)$. To this, it searches for all possible instantiations of the variable $y$ that satisfy the pattern, in this case the nodes labeled by 2 and 4, and it applies the join in the numerical abstract domain between the constraints on these nodes, i.e., $\text{dt}(y) = 2$ and $\text{dt}(y) = 4$.

The resulting abstract heaps are given in Figure 1.7.

![Abstract heaps](image)

Figure 1.7: Folding abstract heaps generated for `initEven`

The unfold and fold steps are repeated until the analysis reaches a fixed point. To ensure the convergence of the fixed point computation, apart from bounding the size of the graphs, we use widening operators over the numerical abstract domain. In the considered example, widening makes the length constraints converge to the fact that the list pointed to by `head` is greater than or equal to one. Consequently, the universally quantified formula from Figure 1.7 is generalized to the entire list.

### 1.2.2.2 Combining analyses over different abstract domains

For programs with procedure calls, we define a compositional analysis such that the summary of a procedure is computed only once and then reused when the procedure is called. To this, we follow the approach of local heap semantics introduced in [Rinetzky 2005a], where at each procedure call, the callee has access only to the part of the heap that is reachable from its actual parameters, called the local heap. The use of local heaps is delicate due to the fact that there are relations between elements from the local heap of the callee, and elements from the heaps of the procedures that are in the call stack. If these relations are lost during the analysis, the results can be unsound in some cases, or very imprecise in others. However, it is not feasible to maintain explicitly these relations during the analysis. Let us examine this crucial problem.

This problem has been addressed in [Rinetzky 2005a] in a framework where data are not considered. In this case, the relations inter-local heaps are due to reachability: nodes in the local heap of the callee can be reachable from the local heaps of the other procedures through paths that do not contain nodes pointed to by the parameters of the procedure (the entry points of the local heap). If during the call these nodes become locally unreachable and deleted, the analysis becomes unsound. The solution to this proposed in [Rinetzky 2005a] consists in maintaining along the calls the points, called cutpoints, where these (inter-local heap) paths enter local heaps. This is a tricky problem since in general
Figure 1.8: Relation between caller and callee local heaps.

the number of cutpoints may be unbounded. However, there is a significant class of programs for which cutpoints are never generated during the analysis. The class of such cutpoint free programs includes programs such as sorting algorithms, traversal of lists, insertion, deletion, etc. In this thesis, we consider cutpoint free programs and we focus on the problems induced by data manipulation.

Combining the analyses with $A_{\mathbb{HS}}(\mathcal{A}_U)$ and $A_{\mathbb{HS}}(\mathcal{A}_M)$: Indeed, even for cutpoint free programs, the problem above persists when data constraints are considered as we do in our framework. The reason is that elements in the local heap of a procedure can be related to elements in the rest of the heap with data constraints such as equality, ordering, etc. This situation is depicted in Figure 1.8. Elements in the local heap of the callee are linked at the call point to external elements by some data relation, $\varphi$, and the analysis generates a summary $\psi_{\text{sum}}$ of the procedure relating the input heap with the output heap. Then, the problem is whether there is a link $\varphi'$ between the elements in the callee output heap and the external elements in the caller heap. This problem depends on the accuracy of the used summarization technique.

Consider for instance the algorithm quicksort in Figure 1.9 that sorts the input list pointed to by the variable $a$ (and where the call $\text{split}(\text{start}, d, &\text{left}, &\text{right})$ copies all the cells of the list pointed to by $\text{start}$ which have data larger than $d$ in the list $\text{right}$, and all the other ones in the list $\text{left}$). The specification of quicksort includes (1) the sortedness of the output list pointed to by $\text{res}$, expressed by the formula given in (1.2.1), where $\text{head}$ is substituted by $\text{res}$, and (2) the preservation property saying that the multiset of data of the input list $a$ is equal to the multiset of data of the output list $\text{res}$. This property is expressed by

$$ms(a^0) = ms(\text{res}) \quad (1.2.5)$$

where $ms(a^0)$ (resp. $ms(\text{res})$) denotes the multiset of integers stored in the list pointed to by $a$ at the beginning of the procedure (resp. $\text{res}$ at the end of the procedure).

The quicksort procedure takes the first element $d$ of the input list $a$ as the pivot, splits the tail of the list $a$ into two lists $\text{left}$ and $\text{right}$, where all the elements of $\text{left}$ resp. $\text{right}$ are smaller resp. greater than $d$, and then performs two recursive calls on the lists $\text{left}$ and $\text{right}$, before composing the results, together with $d$, into a sorted list. After the recursive call at line 16 on $\text{left}$, the information we obtain from the analysis with the domain of first-order formulas is that the output list $\text{left}'$ is sorted. Since we had already the information that the elements of the input list $\text{left}$ were all less than $d$, we must infer after the return from that call that the elements of $\text{left}'$ are also less than
1.2. CONTRIBUTIONS

```c
1 typedef struct list {
2     int data;
3     struct list *next;
4 } list;
5 list* quicksort(list *a){
6     list *left,*right,*pivot,*res,*start;
7     int d;
8     if (a == NULL || a->next == NULL)
9         res = clone(a);
10     else {
11         d = a->data;
12         pivot = create(1); // list of length 1
13         pivot->data = d;
14         start = a->next;
15         split(start,d,&left,&right);
16         left = quicksort(left);
17         right = quicksort(right);
18         res = concat(left,pivot,right); }
19     return res; }
```

Figure 1.9: The quicksort algorithm on singly-linked lists.

d. But, since the link between d and the elements of left has not been passed to the recursive call, this information cannot be computed. This is because the used abstract domain cannot express the fact that a list is a permutation of another list (which requires formulas beyond the universally quantified fragment). Again, maintaining all the relations between the elements of the local heaps and the external elements is not feasible.

Our solution to this problem is based on strengthening the analysis in the domain of first-order formulas with the analysis in the domain of multiset constraints. Indeed, for the quicksort example, knowing that left and left’ have the same multisets of elements should allow to infer from the fact that all elements of left are less than d, that the same fact also holds about the elements of left’.

The combination of analysis is based on a partial reduction operator [Cousot 1979] between abstract heap sets over $\mathcal{A}_U(P)$ formulas and abstract heap sets with multiset constraints, denoted by $\text{strengthen}^{HS}_{HM}$ (for more details, see Section 6.7).

Combining the analyses with universally quantified formulas over different sets of patterns: Another problem that we address for the design of a compositional analysis is due to the use of patterns P for the guards of the universally quantified implications. Indeed, the analysis of different procedures may need the use of different sets of patterns, and therefore it is important to be able to localize the choice of these patterns to each procedure. Otherwise, it would be necessary to use a set of patterns that includes the union of all the sets that are used during the whole analysis. This would obviously make the analysis inefficient.

Consequently, during the analysis, at procedure calls and returns, we need to switch from an abstract domain of formulas parametrized by some set of patterns, say $\mathcal{P}$, to an abstract domain parametrized by another set of patterns $\mathcal{P}_1$ or $\mathcal{P}_2$ as shown in Figure 1.10 ($\mathcal{A}_U(P)$ denotes the domain of first-order formulas parametrized by the set of patterns $\mathcal{P}$). This transformation is defined using partial reduction operators that are defined over $\mathcal{A}_U(P) \times \mathcal{A}_U(P_1)$, respectively $\mathcal{A}_U(P) \times \mathcal{A}_U(P_2)$ (for more details, see Section 6.7).

The two problems exposed above show that in order to define a compositional and accurate inter-procedural analysis, we need to define an operation for computing
CHAPTER 1. INTRODUCTION

over-approximations of the intersection between elements that belong to different abstract domains (e.g., first-order formulas with multiset constraints, or first-order formulas parametrized by different sets of patterns). We propose a mechanism based on unfolding/folding operations, which allows to compute these approximations and which can be used at procedure calls and returns to (1) compute an over-approximation of the intersection between a first-order formula and a multiset constraint and (2) convert universal formulas defined over a set of patterns \( P_1 \) to formulas defined over a set of patterns \( P_2 \).

**Sound approximate decision procedures:** Beyond compositional summary computation, the partial reduction operators that we define allow to increase the precision of the entailment test. First, it improves the entailment operation in the domain of universally quantified formulas by, allowing to compare formulas over different guards, obtained from different sets of patterns denoted \( P_1 \) and \( P_2 \). Again, the idea is to use the partial reduction operator over \( A_{HS}(A_U(P_1)) \times A_{HS}(A_U(P_2)) \) in order to convert a formula defined over \( P_1 \) into a formula over \( P_2 \) and then, to compare them using the entailment operator from \( A_{HS}(A_U(P_2)) \).

Moreover, the partial reduction operator \( strengthen_{HS}^{M} \) over \( A_{HS}(A_U(A_M)) \) allows us to check the entailment between combined constraints of the form \( \phi \land \psi \Rightarrow \phi' \), where \( \phi \) and \( \phi' \) are two universal first-order formulas (potentially over different sets of patterns) and \( \psi \) is a multiset constraint. This provides a lightweight, sound (but not complete) decision procedure for such kind of formulas. It is useful to increase the precision of assertion checking. For instance, to check that two sorting procedures \( S_1 \) and \( S_2 \) are equivalent, it is possible to call each of them on two identical input lists \( I_1 \) and \( I_2 \) and then assert that the two outputs \( O_1 \) and \( O_2 \) are equal. Performing a combined analysis, that uses the partial reduction operator \( strengthen_{HS}^{M} \), we were able to prove the assertion statement that states the equality between the outputs of the two procedures (this example is detailed in Section 6.8.2).

**1.2.3 Celia: a tool for Static Analysis of C Programs with Dynamic Lists**

We have implemented the inter-procedural analysis techniques in a tool for the static analysis of C programs manipulating dynamic (singly linked) lists. It takes as input C programs and it computes for each control point of a C program the assertions which are true (i.e., invariant) at this control point. These invariants are relations between the input (of the procedure) and the reachable set of states at the considered control
1.3. THESIS ORGANIZATION

point. CELIA is a plug-in of the Frama-C platform [CEA] and it uses the heap abstract domains provided by the CINV library and the fixpoint computation engine FIXPOINT [Jeannet]. CINV implements the domain of abstract heap sets parametrized by (1) the universally quantified formulas over a particular set of patterns, (2) the abstract domain of multiset constraints and (3) the abstract domain of constraints over the sum of elements of a list. The implementation of the $\mathbb{D}\mathbb{W}$-domain uses the numerical domains provided by APRON [Jeannet 2009].

1.3 Thesis organization

The remainder of this thesis is organized as follows.

Chapter 3 describes in terms of transition systems the reachability problem and the verification of safety properties. It briefly recalls the framework of abstract interpretation and its application to the design of static analyses for transition systems.

Chapter 4 introduces the class of programs that we consider. These are annotated C-like programs with procedures that manipulate dynamically allocated data structures. From the point of view of memory representation, we consider a storeless semantics. Moreover, for procedure calls, we consider the local semantics given in [Rinetzky 2005a].

Chapter 5 introduces the logics CSL and ICSL. Section 5.2.3 contains the decidability proof for CSL and Section 5.2.6 contains the definition of the strongest post-condition. The chapter ends with Section 5.2.7 that shows examples of specifications that were checked using ICSL.

Chapter 6 introduces analyses based on abstract interpretation for programs manipulating singly-linked lists. It defines a family of abstract domains that symbolically represent relations between program configurations. The abstraction principle is defined in Section 6.3. Section 6.4 presents the abstract domain of universally quantified formulas $\mathcal{A}_U$. Then, Section 6.5 and Section 6.6 present the abstract domain of multisets $\mathcal{A}_M$ and the abstract domain of sums $\mathcal{A}_S$. Section 6.7 introduces the partially reduction operators that are defined to increase the precision of the inter-procedural analysis. Finally, Section 6.8 describes an implementation of the techniques described in this chapter.
2.1 Decidable logics for program verification

In this section we discuss the relation between CSL and other logics for program verification introduced in the literature.

Logics on numerical data types: We discuss first logics that can describe configurations of programs which manipulate only numerical data types. Probably the oldest mathematical theory is the arithmetic with addition and multiplication over natural numbers, called Peano arithmetic. Deciding the satisfiability or the validity of formulas in this logic is undecidable. By eliminating the multiplication operator, we obtain Presburger arithmetic \cite{Presburger1929} which is decidable. The logic introduced by Presburger was extend to integers and multiplication with a constant (in fact, these formulas can be reduced to the initial Presburger arithmetic). In \cite{Fischer1974}, a double exponential nondeterministic time lower bound was given and Oppen \cite{Oppen1978} showed that Cooper’s quantifier elimination algorithm \cite{Cooper1972} has triple exponential deterministic time worst-case complexity. If we restrict ourselves to quantifier free formulas then the decision procedure becomes NP-complete.

Almost as old as the arithmetic of integers is the arithmetic on rational numbers and later on real numbers. Tarski showed that the theory of reals (inequalities of polynomial expressions) is decidable \cite{Tarski1951}. The decision procedure uses an algorithm for quantifier elimination which has non-elementary complexity. Therefore, a lot of works define complex decision procedures for this theory \cite{Basu1998,Tiwari2005,Monniaux2008,Monniaux2010}. A full description of theories on scalar data types can be found in \cite{Bradley2007}.

In \cite{Nelson1983}, a logic with ternary reachability predicates of the form $x \xrightarrow{f} z \rightarrow y$, similar to the reachability predicates in gCSL, is introduced ($x \xrightarrow{f} z \rightarrow y$ holds if $y$ is reachable from $x$ without passing through $z$). The paper defines the weakest precondition for formulas in this logic but, there is no decidability result for the satisfiability of this logic. Since then, various approaches have been developed for the verification of programs with dynamic data structures including works based on abstraction techniques, e.g., \cite{Balaban2005,Balaban2007,Berdine2007,Sagiv2002a,Bingham2006,Rakamaric2007}, on logics for reasoning about graph structures, e.g., \cite{Benedikt1999,Bradley2006,Yorsh2007,Lahiri2008,Madhusudan2011a,Reynolds2002,Zhou2010}, and on automata-theoretic techniques, e.g., \cite{Bouajjani2006,Habermehl2008,Moller2001}. 
First-order logics with reachability: The works based on predicate abstraction assume that the user provides a set of predicates and that the properties to be checked are expressible using predicates in this set. The logic introduced in [Balaban 2005] expresses properties of heaps with singly-liked lists using a reachability predicate of the form $x \rightarrow y$ (which holds if the memory location $y$ is reachable from $x$). It does not permit the universal quantification over variables appearing in reachability predicates. Concrete properties of programs that manipulate singly-linked lists are abstracted into formulas over a fixed set of predicates using a decision procedure based on a small model argument. It is assumed that the scalar data comes from a finite domain. This work is extended to handle tree structures with fixed arity in [Balaban 2007]. In [Rakamaric 2007], the authors introduce a system of inference rules for a logic with reachability that can describe heaps with multi-liked lists. Like [Balaban 2005], the logic does not support quantifiers. To prove the completeness of the system of inference rules they use a decision procedure similar to the one in [Balaban 2005].

Separation logic: Separation logic [Reynolds 2002] is used to reason about programs that manipulate dynamic data structures (using a store-based semantics). The logic uses the operator $\star$, called separating conjunction, to express the fact that the heap can be split into two disjoint parts where its two arguments hold, respectively. Even if the logic is undecidable, various fragments have been defined for which there exist (sound) procedures to check the entailment between formulas, e.g. [Bansal 2009, Berdine 2004, Calcagno 2001, Cook 2011, Bozga 2008]. These procedures have been successfully applied to reason about shape properties in a tool called Space Invade [Calcagno]. The first decidable fragment (for the model checking problem) of separation logic is defined in [Calcagno 2001]. It consists in quantifier free formulas that use the separating conjunction, equalities between variables that represent memory locations, and the points-to relation (predicates of the form $x \rightarrow y_1, y_2$ which hold if $x$ points to a binary heap cell containing $y_i$ in the $i$-th component). Later, in [Berdine 2004], this fragment is extended with list segment predicates $ls(x, y)$ which hold if there exists a list segment from $x$ to $y$. It is proven that the validity of implications between separation logic formulas in this fragment is decidable. The decision procedure is based on a system of inference rules and it belongs to NP. Using a different decision procedure (that reduces the validity of the implication to the graph homomorphism problem), in [Cook 2011], the complexity of checking the validity of the implication is proven to be polynomial. The $\varphi_{\text{SLL}}$ formulas that we consider in Section 6.2 belong (up to some small modifications) to the fragment studied in [Berdine 2004, Cook 2011]. The logic CSL introduced in this thesis is incomparable to this fragment because it is not able to fully describe the separating conjunction but it is able to describe properties of composite data structures using not only reachability constraints but also length and data constraints. In general, to express the separating conjunction $\star$ in $gCSL$ one needs length constraints over universally quantified location variables (see Section 6.2) which is not allowed in CSL. However, CSL can express disjointness of two singly-linked list, by adding ghost variables to label the head of the segments in the case of cycles. The corresponding CSL formulas are given in Section 6.2.

Two extensions of the fragment in [Berdine 2004], for which the validity of the implication between formulas is decidable, are considered in [Bozga 2008, Bansal 2009]. The logic in [Bozga 2008] uses a list segment predicate of the form $1ls^k(x, y)$, which holds if
2.1. DECIDABLE LOGICS FOR PROGRAM VERIFICATION

there exists a list segment from \( x \) to \( y \) of length \( k \), in order to constrain the lengths of the lists in the heap. On the other hand, the logic in [Bansal 2009] uses a predicate of the form \( \text{val}(x) \leq \text{val}(y) \), that asserts that the value stored at the location \( x \) is smaller than the one stored at \( y \), in order to constrain the data in the lists. The logic CSL is incomparable to these fragments. It can handle more complicate data structures but it can not fully describe the separating conjunction.

The approach in [Berdine 2007] introduces an abstract domain composed of separation logic formulas in order to reason about programs that manipulate composite data structures. These formulas use a class of higher-order predicates which define recursively the composite data structure. The focus in that work is on the heap shape properties, assuming that data have been abstracted away. In contrast, our approach allows precise invariant checking to reason about the same type of structures, taking into account constraints on data and sizes: CSL allows (1) to specify (a finite number of) shared locations, (2) to reason about the lengths of lists and arrays, and (3) to model explicitly data over infinite domains.

First-order logics with reachability and data: Decidable first-order logics to reason about shape and data were introduced in [McPeak 2005, Lahiri 2008, Ranise 2006]. In [McPeak 2005], the authors introduce an axiomatization and a decision procedure for a logic that is able to express properties about the shape and the scalar data in the heap. Their decidable fragment does not allow disequalities between terms that represent memory locations. Also, singly-linked lists are encoded into doubly-linked lists using ghost fields (because their logic does not support reachability predicates with transitive closure as introduced by Nelson in [Nelson 1983]). In [Ranise 2006], the authors introduce a decidable theory for singly-linked lists with reachability predicates and arithmetical constraints. They are able to capture interesting properties about pointers, like list disjointness or contiguous memory address among the list cells. They use a combination of theories, reachability plus a theory for data or pointer arithmetics. This combination can not be instantiated in order to obtain the decidability of the logic CSL with length constraints that we consider, mainly because not every solution of the length constraints leads to a model of the CSL formula. Therefore, from this point of view, the two logics are incomparable. Also, the decidable logic in [Ranise 2006] does not allow universal quantification.

A classification of decidable many-sorted first-order logics with quantifier alternation was introduced in [Abadi 2010]. The authors identify three decidable fragments based on a partitioning of sorts into levels. None of the fragments includes reachability predicates. Probably the work most closely related to ours is the Logic of Interpreted Sets and Bounded Quantification (LISBQ) introduced by [Lahiri 2008]. This logic is able to express properties on the shape and data of composite data structures. The definition of LISBQ is based on a partial order on the types (similar to the one in [Abadi 2010]) of the program which forbids having links going from a smaller type to a greater one, whereas in CSL, although we use an ordered partition of the types, we do not impose constraints on the fields of the linked structures (we allow having edges going from one class to another one, and other edges going back). Formulas in LISBQ may contain only universal quantifiers and the restrictions on LISBQ formulas do not allow to reason completely about doubly-linked lists as it is possible in CSL (or even in CSL$_1$). Moreover, LISBQ does not handle any constraints on the size of the structures, which is possible in CSL (or
CHAPTER 2. RELATED WORK

even in CSL\(^1\). For example, \texttt{doubly-ll(a)} or \texttt{dll-len2(a)} given in Section 4.4.3 are not expressible in LISBQ. Moreover, CSL allows to specify multi-linked lists and arrays, to express constraints on the lengths of the lists, and formulas in CSL may contain alternations of universal and existential quantifiers. (In fact, LISBQ can be seen as a strict fragment of CSL\(^1\).)

Another example of a formula not expressible in LISBQ is the formula \texttt{data-doubly-ll}, which describes an array of acyclic doubly-linked lists (each array element contains an integer field \(dt'\) and a field \(dll\) that points to a doubly-linked list) such that each doubly-linked list contains an integer (in the field \(dt\)) whose value is twice the value of the integer from the corresponding array element:

\[
data-doubly-ll(a) \equiv \texttt{doubly-ll}(a) \land \\
\forall i. \exists dl_i, dl. \left( a[i] \xrightarrow{\{dll\}} dl_i \land \\
\left( dl_i \xrightarrow{\{\text{next, prev}\}} dl \lor dl = dl_i \right) \land dt(dl) = 2 \times dt'(a[i]) \right).
\]

\texttt{doubly-ll(a)} is the formula in Section 4.4.3 that describes an array of doubly-linked lists, and \texttt{next} and \texttt{prev} are the pointer fields used in the doubly-linked lists. Due to the quantifier alternation, this formula does not belong to LISBQ but it belongs to CSL. In LISBQ, the underlying data logic must be Presburger arithmetics or equality with uninterpreted functions whereas, in CSL, it is possible to use any decidable data logic.

Monadic second-order logic over graphs: There are several works that consider second order logics to reason about program configurations [Yorsh 2007, Møller 2001, Madhusudan 2011a, Madhusudan 2011b]. The pointer assertion logic engine (PALE) ([Møller 2001]) uses monadic second-order logic (MSO) to express properties involving reachability. It can handle structures that are definable over tree-skeletons. Although the logic can express more complex shape properties than the ones allowed by our logic, it has a decision procedure with high complexity (non-elementary) and also, data is abstracted by a fixed set of predicates. The logic LRP [Yorsh 2007] is also incomparable to CSL. LPR imposes constraints on the neighborhood of every node that is reachable via a regular expression over pointer fields from a designated node. It can express properties of complex data structures like trees, but it assumes a finite data domain and it can not express constraints on the size of the heap. The decidability of the satisfiability (validity) problem is reduced to weak monadic second order logic over trees.

The logic STRAND introduced in [Madhusudan 2011a] allows to express properties on the shape of the heap but also, properties on data from a potentially unbounded domain. Like PALE, it can handle structures definable over tree-skeletons. To obtain the decidability of the satisfiability problem, the authors impose semantical restrictions to STRAND. The decision procedure works by combining a decision procedure for MSO on trees with a decision procedure for the data theory. In a different paper [Madhusudan 2011b], the authors present a decision procedure for a syntactic fragment of STRAND. The shape constraints in the decidable fragments of STRAND are more expressive than the ones in CSL. On the other hand, these fragments can not describe the size of the heap (this constraint makes even CSL\(^1\) incomparable to STRAND) and they do not allow \(\forall^* \exists^*\) quantifier alternation over variables appearing in data terms (terms that represent values of data fields). For example, the formula given in 2.1.1 does not belong to the decidable fragments of STRAND due to the quantifier alternation but, it belongs to CSL.
2.2. PROGRAM ANALYSIS TECHNIQUES

First-order logics on arrays: Recently, several works have addressed the issue of reasoning about programs manipulating linear structures with unbounded data, like words, e.g. [Bouajjani 2007] or arrays, e.g. [Bradley 2006, Habermehl 2008, Zhou 2010]. The logic in [Bouajjani 2007] allows to reason about words but it is not closed under pointer manipulation operations. In the following, we compare CSL$_1$ restricted to arrays with the approaches from [Bradley 2006, Habermehl 2008, Zhou 2010] which allow to reason about unidimensional arrays.

The Array Property Fragment (APF) in [Bradley 2006] allows formulas of the form $\forall j. G(j) \rightarrow U(j)$, where $j$ is a set of variables interpreted into array indices. CSL$_1$ is strictly more expressive than APF. For instance, it is possible to express strict order properties between the array elements:

$$\forall j, j'. \ j < j' \Rightarrow dt(a[j]) < dt(a[j'])$$

(2.1.2)

(a is an array variable, $dt$ is a data field, and $j, j'$ are index variables). Moreover, CSL allows to quantification over array variables, which is not the case for AFP, so in (2.1.2) one could quantify $a$ existentially (or universally). In general, CSL is incomparable with APF because formulas in APF are interpreted over unbounded arrays (the domain of indices is arbitrarily large and, consequently, there is always some index different from the values of all index terms). The arrays of bounded size used in programs are represented by unbounded arrays in the logic APF. Consequently, a model of an APF formula which describes the specification of a program declaring an array $a$ may use array positions outside of the bounds of $a$. Therefore, we consider that CSL which interprets array indices inside the array bounds is more suited for program verification.

The Logic on Integer Arrays (LIA) introduced in [Habermehl 2008] is incomparable with CSL$_1$. LIA allows modulo constraints on array indices and constraints of the form $j - j' = 1$ ($j, j'$ are universally quantified index variables) which are not permitted in CSL$_1$. On the other hand, the right part of the implications is restricted to conjunctions of constraints on integers of the form $d - d' \leq n$, where $d, d'$ are integer variables and $n$ is a constant. The strict sortedness on arrays of integers from (2.1.2) is expressible in LIA. The following formula states a property of an array containing the Fibonacci sequence:

$$\exists a \ \forall j_1, j_2, j_3. \ j_1 < j_2 < j_3 \Rightarrow dt(a[j_1]) + dt(a[j_2]) < dt(a[j_3]),$$

(2.1.3)

which is neither in APF nor in LIA but, it is a CSL$_1$ formula.

The array logic given in [Zhou 2010] considers that the array values come from a bounded data domain. With this restriction, the logic is more expressive than CSL$_1$, AFP, and LIA. For example, it allows $\forall^* \exists^*$ quantifier alternation over array indices.

2.2 Program analysis techniques


31
Shape analysis: The abstract interpretation based analysis in [Sagiv 2002a], called shape analysis, is able to synthesize complex invariants that describe the structure of the allocated memory. The invariants are expressed in a three-valued first-order logic. Differently from the work presented in this thesis, they suppose that data is abstracted away or that it comes from a finite domain. The approach in [Rinetzky 2005b] can synthesize procedure summaries that describe data if the instrumentation predicates which guide the abstraction speak about data. Providing patterns is simpler than providing instrumentation predicates on data because patterns contain only constraints between (universally-quantified) positions (in the left-hand-side of the implication) and no constraints on data. For example, in [Rinetzky 2005b] the predicate \texttt{dle(v, u)} allows to synthesize the summary for a procedure that sorts in ascending order, but cannot be used for a procedure that sorts in descending order. However, using the pattern \(y1 \leq y2\) allows with our approach to synthesize the summaries for both kind of procedures. The same pattern may also allow to discover other properties than sortedness. Actually, patterns are in many cases simple (ordering/equality constraints) and can be discovered using natural heuristics based on the program syntax or proposed/guessed by the user, whereas constraints on data can be more complex. Our approach allows to discover (maybe unpredictable) data constraints for given guard patterns. To establish the fact that a procedure preserves the data values in the input list, the method used in [Rinetzky 2005b] is based on reachability, that is, every cell in the input list remains reachable in the output list. This method can be applied only for programs that never modify/permute the contents of data fields. In our approach, using the multiset domain, we can handle programs that can permute positions of cells in the list or modify/permute the contents of their data fields.

The analysis in [Gulwani 2009] combines a numerical abstract domain with a shape analysis. It is not restricted by the class of data structures but it considers only properties related to the shape and to the size of the memory, assuming that data have been abstracted away. Our approach is less general concerning shape properties but it is more expressive concerning properties on data.

The combination of abstract domains in [McCloskey 2010] can also be used to synthesize invariants in the form of universally-quantified first-order formulas which combine shape and data constraints. It allows any kind of abstract domain to describe the shape. In this work, we combine a fixed abstract domain for singly-linked lists with various abstract domains describing sequences of data. Our approach requires less user guidance. In [McCloskey 2010] the user must provide all the heap and data predicates which are shared by the abstract domains while in our approach the user must provide only a set of guard patterns (they correspond to heap predicates in [McCloskey 2010]). The rest of the formula is automatically synthesized by our analysis. Also, the work in [McCloskey 2010] does not consider multiset constraints and it handles procedure calls by inlining.

Boolean heap abstraction [Podelski 2005, Podelski 2010] is a symbolic shape analysis based on the underlying concepts in three-valued shape analysis [Sagiv 2002b]. The abstraction of the heap is defined starting from a set of predicates on the heap objects. Then, the abstract semantics is defined by a program over boolean heaps (sets of bitvectors). Computing the most precise abstract program can be very complex due to the exponential number of calls to the decision procedure. A more efficient approach based on counter-example guided refinement is proposed in [Podelski 2010]. The idea is to start from an abstract program defined using the Cartesian abstraction [Ball 2003] and then to refine it using spurious counter-examples. The refinement can add more predicates to
the definition of the abstract domain or it can refine the abstract program. The latter corresponds to fine-tuning a focus operator (from [Sagiv 2002b]) that adapts the effect of the abstract transformers. As in the case of three-valued shape analysis, our approach has the advantage of being more automatic. The user must provide only (some of) the guard patterns, the rest of the formula being synthesized by the analysis. Although the CEGAR approach in [Podelski 2010] can discover some new predicates on the heap objects, it can not synthesize complex arithmetical constraints as we do in our framework. Our approach is more generic because it can handle more than universally-quantified first-order formulas, e.g. multiset constraints. Nevertheless, the approach in [Podelski 2005, Podelski 2010] can be applied for a wider class of data structures.

**Synthesizing assertions in separation logic:** The abstract interpretation based techniques introduced in [Chang 2008, Distefano 2006, Rival 2011, Vafeiadis 2009] synthesize assertions expressed in separation logic. As in the previous case, they focus on shape properties; either they suppose that the data has been abstracted away or they require a set of predicates on data which are more difficult to be guessed than the patterns from our approach. Based on separation logic, in [Calcagno 2009], the authors introduce a method for generating pre/post conditions for procedures without using any context of the call. Their technique is based on an analysis that, for each procedure, starts with an empty heap and generates candidates for the precondition such that every pointer dereference and procedure call are safely performed. Then, a second analysis is performed to discover the post-condition corresponding to the computed precondition. Their technique is modular and it can handle nested singly-linked lists but, no data constraints can be inferred (for example, sortedness properties or even more simple specifications about data are beyond the language of assertions that they support).

**Predicate abstraction:** The work in [Flanagan 2002] introduces a technique based on predicate abstraction for generating universally quantified invariants for programs with arrays and lists. This technique consists in introducing some fresh variables in the program and requires that the user provides a set of predicates over these variables and the usual program variables. Since the additional variables represent some values not constrained by the program they can be soundly universally quantified in the invariant. The abstract state space is built as usual in predicate abstraction using a decision procedure.

With respect to the works presented above, our approach requires less user guidance. The analysis over the domain of abstract heaps sets with universally quantified formulas, $\mathcal{A}_{HS}(\mathcal{P})$, is able to automatically synthesize complex universally quantified invariants and procedure summaries based only on a set of guard patterns $\mathcal{P}$.

**Abstract interpretation for synthesizing assertions about shape and data:** The approach in [Deutsch 1990] considers abstract domains where the elements are pairs formed of a graph and a constraint on data. The inter-procedural analysis based on these domains can not synthesize constraints in form of universally-quantified formulas as our analysis can do. In [Gulwani 2008b], a technique for the synthesis of universally quantified invariants is presented. The approach in this thesis differs from this one by the type of user guiding information. Indeed, the quantified formulas considered in [Gulwani 2008b] are of the form $E \land \bigwedge_i \forall y. G_i \implies F_i$ but, the right-hand side of the implications, i.e., $F_i$,
is defined from a template (an atomic formula) given by the user. Moreover, to define
the join operator of the considered abstract domain, an under-approximation operator for
the guards is used. More precisely, for implications whose right-hand sides correspond to
instances of the same template, e.g. $E_1 \land \forall y_1. (G_1 \Rightarrow F)$ and $E_2 \land \forall y_2. (G_2 \Rightarrow F)$, the
join of the two formulas is obtained by under-approximating the union of the guards, i.e
$G_1 \lor G_2$. The idea is to start with the union of $E_1 \land G_1$ and $E_2 \land G_2$ and then, to add more
atomic formulas implied by $E_1 \land G_1$ and $E_2 \land G_2$ (the conjunction with the environments $E_1$, respectively $E_2$, is important in order to obtain a guard that is satisfiable for some
values of the universally quantified variables). In contrast, our approach is guided by
the form of the formulas in left hand side of the implications and synthesizes the right
hand side. Also, it does not need an under-approximation operator which improves the
scalability. Therefore, the two approaches are in principle incomparable.

**Abstract interpretation for programs with arrays:** The techniques in
\cite{Gopan2005, Halbwachs2008} are applicable to programs with arrays. The class of in-
variants they can generate is included in the one handled by our approach using $\mathcal{A}_{\mathcal{HS}}(\mathcal{A}_U)$.
These techniques are based on an automatically generated finite partitioning of the array
indices. We consider a larger class of programs for which these techniques can not be ap-
plicated. The analysis introduced in \cite{Perrelle2010} for programs with arrays can synthesize
invariants on multisets of the elements in array fragments. This technique can generate
invariants which are more complex than the ones expressible in $\mathcal{A}_{\mathcal{HS}}(\mathcal{A}_M)$ but it can not
be applied directly to programs with dynamic lists.

**Synthesizing assertions based on constraint solving:** The works based
on constraint solving from \cite{Beyer2007, Gupta2009, Sankaranarayanan2004b, Sankaranarayanan2004a} are
*template based*, i.e., they require some parameterized form
of the invariant which is used by a constraint-based analysis in order to generate rel-
ationships between the parameters. Every instantiation of the parameters satisfying
these relationships leads to an inductive invariant. For example, in the case of numerical
programs, the approach in \cite{Sankaranarayanan2004b} is based on templates of the form
$a_0 + a_1 \cdot x_1 + \ldots + a_n \cdot x_n \geq 0$, where $x_1, \ldots, x_n$ are program variables and $a_0, \ldots, a_n$ are
some integer variables that represent the coefficients of the linear constraint. Given such a
template for each loop in the program (the template fixes the form of the inductive invari-
ant associated to the loop), a system of (existentially quantified) non-linear constraints
is built which represents the transition relation of the program. The inductive invariants
are obtained by solving this set of constraints using the Farkas lemma. This approach is
extended to templates in polynomial arithmetic in \cite{Sankaranarayanan2004a}. The work
from \cite{Beyer2007} uses constraint solving in order to synthesize invariants expressed in the
combined theory of linear arithmetic and uninterpreted function symbols. The key idea
is to reduce constraint solving in the combined theory to constraint solving in the theory
of linear arithmetic. These techniques can be used to generate invariants for programs
that manipulate arrays and set data structures. In \cite{Gupta2009}, in order to improve
the performance of constraint solving, the system of constraints is simplified using other
techniques like abstract interpretation or dynamic testing.

**Interpolation based synthesis:** To improve the search for predicates, techniques
based on interpolation were introduced in \cite{Jhala2007} for programs with arrays. The
2.2. PROGRAM ANALYSIS TECHNIQUES

interpolating prover, introduced in [McMillan 2008], for a theory of arrays with transitive
closure is used to generate universally quantified invariants for programs with arrays and
heap data structures. Invariants are generated from the interpolants obtained by refuting
unwindings of the program of increasing length. Usually, this sequence of interpolants di-
verges (for example, they exhibit increasing numeric constants) and the solution proposed
in this paper is to bound the language of the interpolating prover.

Automata-based assertion synthesis: In [Bouajjani 2006], the authors define accu-
rate abstractions of programs that manipulate singly-linked lists using counter automata.
Invariants that describe the structure of the allocated memory or its size can be synthe-
sized by an analysis of the counter automaton that abstracts the program. This technique
can not be used to synthesize universally quantified invariants as we do in this thesis.

Trace partitioning abstract domains: In [Rival 2007], the authors introduce trace
partitioning abstract domains which start from a partition of the set of traces and com-
pute an invariant for each class. The partitioning can be static (usually based on the
control structure of the program) or dynamic. From this point of view, the approach in
[Rival 2007] considers mainly statically-defined partitions. The abstract domains in this
thesis, based on the unfolding/folding operations, can be seen as an instance of a trace
partitioning abstract domain with a dynamic partitioning. The corresponding partition-
ing puts in the same class all the traces for which the number of dereferences of the next
pointer field is the same modulo some fixed constant $k$ (which is a parameter of the anal-
ysis). The approach in [Rival 2007] considers mainly numerical abstract domains and it is
not faced to the difficulties raised by a compositional analysis on programs manipulating
dynamic data structures.
In this chapter, we introduce two commonly used frameworks for the verification of safety properties on transition systems.

First, we describe *pre/post condition reasoning* where the main artifact is the notion of inductive invariant. To prove that a transition systems $TS$ satisfies a safety property given by some set of states $Safe$, we search for a set of states $I$, called inductive invariant, such that the initial states of $TS$ is included in $I$, $I$ is included in the set of safe states $Safe$, and the immediate successors of the states in $I$ are included in $I$.

Second, we introduce the *abstract interpretation framework* [Cousot 1977a] where we compute an over-approximation of the set of reachable states of the transition system. The set of reachable states can be defined as the least fixed point of an operator $post$ that gives for any set of states, the set of immediate successors (according to the transition relation). To compute an over-approximation of this least fixed point we can use some lattice $A$, called *abstract domain*, such that (1) there exists a Galois connection (or a concretization function) between $A$ and the lattice $C$ whose elements are sets of states and (2) the lattice $A$ is finite or there exists a *widening operator* for $A$.

### 3.1 Transition Systems

**Definition 3.1.1** (Transition system). A transition system is a tuple $TS = (S, Act, \rightarrow, I)$ where

- $S$ is a set of states,
- $Act$ is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation over $S$, and
- $I \subseteq S$ is a set of initial states.

$TS$ is called finite if $S$ and $Act$ are finite.

For convenience, we write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \rightarrow$. Intuitively, the transition system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\rightarrow$. That is, if $s$ is the current state, then a nondeterministically chosen transition $s \xrightarrow{a} s'$ from $s$ is taken, and $s'$ becomes the current state of the system. This selection procedure is repeated in state $s'$ and finishes once a state is encountered that has no outgoing transitions.
CHAPTER 3. PRELIMINARIES

Definition 3.1.2 (Direct Successors). Let $TS = (S, Act, \rightarrow, I)$ be a transition system. For $s \in S$ and $a \in Act$, the set of direct $a$-successors of $s$, denoted $\text{post}(s, a)$, is defined as:

$$\text{post}(s, a) = \{ s' \in S \mid s \xrightarrow{a} s' \}.$$

Then, the set of all direct successors of $s$, denoted $\text{post}(s)$, is defined as:

$$\text{post}(s) = \bigcup_{a \in Act} \text{post}(s, a).$$

The notations for sets of direct successors extend to sets of states in an obvious way: given $S' \subseteq S$,

$$\text{post}(S', a) = \bigcup_{s \in S'} \text{post}(s, a) \quad \text{and} \quad \text{post}(S') = \bigcup_{s \in S'} \text{post}(s). \quad (3.1.1)$$

The terminal states of a transition system are the states without any outgoing transitions.

Definition 3.1.3 (Terminal state). A state $s$ of a transition system $TS$ is called terminal if $\text{post}(s) = \emptyset$.

Once the transition system reaches a terminal state, the complete system halts.

Definition 3.1.4 (Execution fragment). Let $TS = (S, Act, \rightarrow, I)$ be a transition system. A finite execution fragment of $TS$ is a sequence of alternating states and actions ending with a state:

$$\rho = s_0a_1s_1a_2s_2a_3\ldots a_n s_n$$

such that $(s_i, a_{i+1}, s_{i+1}) \in \rightarrow$ for every $0 \leq i \leq n - 1$. An execution fragment is denoted by:

$$\rho = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots \xrightarrow{a_n} s_n.$$

We refer to $n$ as the length of the execution fragment $\rho$.

An infinite execution fragment is an infinite alternating sequence of states and actions:

$$\rho = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots.$$ 

Definition 3.1.5 (Execution). An execution of a transition system is either a finite execution fragment that ends with a terminal state or an infinite execution fragment.

Definition 3.1.6 (Reachable states). A state $s$ is called reachable if there exists some execution fragment starting from some initial state and ending in $s$.

For a transition system $TS = (S, Act, \rightarrow, I)$ and some $s_0 \in I$ we define the set $\text{post}^k(s_0)$ to be the set of states reachable in $k$-execution steps, with $k > 0$:

$$\text{post}^0(s_0) = \{ s_0 \} \quad \text{and} \quad \text{post}^k(s_0) = \{ s \in S \mid \text{there exists } s_1, \ldots, s_{k-1} \in S \text{ and } a_1, \ldots, a_k \in Act \text{ such that } s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_{k-1}} s_{k-1} \xrightarrow{a_k} s \text{ is an execution fragment} \}$$

The set $\text{post}^*(s_0) = \bigcup_{k \geq 0} \text{post}^k(s_0)$ denotes the set of all reachable states. We extend as usual the definition of $\text{post}^*$ to sets of states:

$$\text{post}^*(S') = \bigcup_{s \in S'} \text{post}^*(s).$$
3.2 Verification of Transition Systems

The transition system induces a reachability relation between its states which is denoted by \( \text{Reach} \subseteq S \times S \). In the following, for any \( s, s' \in S \), \( \text{Reach}(s, s') \) denotes the fact that \( (s, s') \in \text{Reach} \). Formally, it is defined inductively by

1. \( \text{Reach}(s, s) \) for all \( s \in I \),
2. \( \text{Reach}(s, s') \) for all \( a \in \text{Act} \) such that \( s \xrightarrow{a} s' \) and
3. \( \text{Reach}(s_1, s_2) = \exists s. \text{Reach}(s_1, s) \land \text{Reach}(s, s_2) \)

Moreover, \( \text{Reach}_I \) denotes the binary relation obtained by intersecting \( \text{Reach} \) with \( I \times S \). Notice that \( \text{Reach}_I = \{(s, s') \mid s \in I, s' \in \text{post}^*(s)\} \).

3.2 Verification of transition systems

In this section, we introduce the verification problems that we consider in this thesis.

Definition 3.2.1 (Pre/post condition reasoning). Given two sets of states, \( \text{Pre} \subseteq S \) and \( \text{Post} \subseteq S \), the pre/post condition reasoning consists in proving that \( \text{post}(\text{Pre}) \subseteq \text{Post} \).

We are interested in the verification of safety properties. That is, let \( \text{Safe} \subseteq S \) be a set of safe states of some transition systems, \( T S \). To check that \( T S \) is safe we need to check that every reachable state of \( T S \) belongs to \( \text{Safe} \).

To prove this, we use an inductive property called inductive invariant.

Definition 3.2.2 (Inductive invariant). Given a transition system \( T S \), a set of states \( S_I \subseteq S \) is called inductive invariant if (1) any initial state \( s_0 \in I \) belongs to \( S_I \), i.e., \( I \subseteq S_I \) and (2) the direct successors of any state in \( S_I \) belong to \( S_I \), i.e., \( \text{post}(S_I) \subseteq S_I \).

Therefore, to prove that all the reachable states of a transition system \( T S \) belong to \( \text{Safe} \), one could find a set of states \( S_I \), that is an inductive invariant for \( T S \), such that \( S_I \subseteq \text{Safe} \).

In this context, the inductive invariants are provided (by the user for example) and the verification problem consists in checking (1) that they are indeed inductive invariants and (2) that they are precise enough to prove the safety condition (the specification). To increase the degree of automatism an interesting approach is to automatically synthesize the invariants, starting from the transition system. One method for synthesizing them is to define an analyses based on abstract interpretation, that computes the set of reachable states of the transition system.

3.3 Analysis of transition systems

In this thesis, we are interesting in analyzing transition systems, to discover their behavior in terms of reachable sets of states (or an approximation of it). To this, we use abstract interpretation, that is analysis method which is not guided by some objective, like a safety property.

Given a transition system \( T S = (S, \text{Act}, \rightarrow, I) \) the problem of computing the set of reachable states can be reduced to computing the least fixed point of a monotone function (that is, the least fixed point \( F \) is the smallest set of states \( X \) such that \( F(X) = X \)).
CHAPTER 3. PRELIMINARIES

Let $\text{post} : \mathcal{P}(S) \to \mathcal{P}(S)$, be the function defined by the transition relation $\rightarrow$ as in (3.1.1). Notice that $\text{post}$ is a monotone function. Then,

$$\text{post}^*(I) = R,$$

where

1. $I \subseteq R$, and
2. for any $R' \subseteq S$ if $I \subseteq R'$ and $\text{post}(R') = R'$ then $R \subseteq R'$.

Computing this least fixed point of the function $\text{post}$ is a difficult problem because the state space $S$ is potentially infinite. In the following, we introduce the main concepts and results that are used to compute this fixed point (when possible). Then, we present the essential ideas in the abstract interpretation framework \cite{Cousot1977}. We use abstract interpretation to compute (an over-approximation of) $\text{post}^*$.

### 3.3.1 Fixed point computations

First, we introduce some basic notions in the computation of fixed points.

**Definition 3.3.1** (Partial order set). A partially ordered set (or poset) $(L, \sqsubseteq)$ is a non-empty set $L$ together with a partial order relation $\sqsubseteq \subseteq L \times L$ which is reflexive, antisymmetric, and transitive.

**Definition 3.3.2** (Direct set). A directed set $(Y, \sqsubseteq)$ is a non-empty set $Y$ together with a preorder order relation $\sqsubseteq \subseteq Y \times Y$ which is reflexive and transitive, such that every pair of elements $a, b \in Y$ has an upper bound in $Y$, i.e., there exists $c \in Y$ such that $a \sqsubseteq c$ and $b \sqsubseteq c$.

**Definition 3.3.3** (Complete partial order set). A complete partial order (cpo) is a poset that is complete, if every direct subset $Y \subseteq L$ has a least upper bound: (1) $\forall l' \in Y, l' \sqsubseteq l$ and (2) for any $l_0$ such that $\forall l' \in Y, l' \sqsubseteq l_0$ then $l \sqsubseteq l_0$.

For any poset $L$, the greatest lower bound (or glb) and the least upper bound (or lub) of a set $Y \subseteq L$ are unique when they exist and are denoted by $\sqcup Y$, respectively $\sqcap Y$. Sometimes $\sqcup$ is called the join operator and $\sqcap$ is called the meet operator and we shall write $a \sqcup b$ instead of $\sqcup \{a, b\}$, respectively $a \sqcap b$ instead of $\sqcap \{a, b\}$, for any $a, b \in L$. We denote by $\perp$, resp. $\top$, the least element and the greatest element of $L$, if they exist.

**Definition 3.3.4** (Lattice). A lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ is a poset where each pair of elements $a, b \in L$ has a least upper bound, denoted by $a \sqcup b$, and a greatest lower bound, denoted by $a \sqcap b$. A lattice is said to be complete if any set $Y \subseteq L$ has a least upper bound (denoted $\sqcup Y$) and a greater lower bound (denoted $\sqcap Y$). In this case, it is denoted by $(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$.

An important example of a complete lattice is the power-set lattice of some set $S$, i.e., $(\mathcal{P}(S), \subseteq, \cup, \cap, \emptyset, S)$. It is the lattice that we will use for the analysis of a transition system, where $S$ is the set of states. We recall that a fixed point of an operator $F$ is an element $X$ such that $F(X) = X$. In \cite{Tarski1955} the author showed that:

**Theorem 3.3.1** (Tarski). Given a complete lattice $(L, \sqsubseteq, \cup, \cap, \perp, \top)$ and a monotone function $F : L \to L$, the set of all fixed points of $F$ form a complete lattice with:

$$\text{lfp}(F) = \bigsqcup\{X \in L \mid F(X) \sqsubseteq X\} \text{ and } \text{gfp}(F) = \bigsqcap\{X \in L \mid X \sqsubseteq F(X)\}.$$
We denote by \( \text{lfp}_Z(F) \) the least fixed point of \( F \) that is larger than some \( Z \in L \). Formally, \( \text{lfp}_Z(F) = \bigcup \{ X \in L, Z \subseteq X \mid F(X) \subseteq X \} \).

Theorem 3.3.1 ensures the existence of the (least/greatest) fixed point of \( F \) but does not provide a procedure to compute it. Therefore, it is customary to use the characterization in terms of sequences of iterations given by Kleene. We recall that a operator \( F : L \rightarrow L \) is continuous if \( F(\bigcup_{n \geq 0} Y^n) = \bigcup_{n \geq 0} F(Y_n) \), for all \( Y_n \in L \) with \( Y_n \subseteq Y_{n+1} \).

**Theorem 3.3.2 (Kleene).** Given \( (L, \sqsubseteq) \) a complete partial order set with a least element \( \bot \) and a monotone and continuous operator \( F : L \rightarrow L \), then,

\[
\text{lfp}(F) = \bigcup_{n \geq 0} F^n(\bot).
\]

Moreover, for any \( X \in L \) such that \( X \subseteq F(X) \), \( \text{lfp}_X(F) = \bigcup_{n \geq 0} F^n(X) \).

This theorem gives a method to compute a fixed point, but the problem is far from being trivial because one must compute the limit of a possibly infinite sequence. Moreover, in general, imposing a continuity requirement for \( F \) might be too strong. However, without this requirement the sequence is not guaranteed to converge towards a fixed point.

**Definition 3.3.5 (Chains of elements).** A subset \( Y \subseteq L \) of a poset \( (L, \sqsubseteq) \) is a chain if

\[
\forall a, b \in Y. a \sqsubseteq b \ or \ b \sqsubseteq a.
\]

If \( Y \) is finite then the chain is finite. A sequence \( (a_n)_{n \in \mathbb{N}} \) with \( a_n \in L \) for all \( n \in \mathbb{N} \) is called ascending chain iff \( \forall n, m \in \mathbb{N} \ n \leq m \implies a_n \sqsubseteq a_m \). A sequence \( (a_n)_{n \in \mathbb{N}} \) with \( a_n \in L \) for all \( n \in \mathbb{N} \) is called descending chain iff \( \forall n, m \in \mathbb{N} \ n \leq m \implies a_m \sqsubseteq a_n \)

We say that \( L \) has no infinite ascending (descending) chains iff for any ascending (descending) chain \( (a_n)_{n \in \mathbb{N}} \) there exists \( n_0 \in \mathbb{N} \) such that \( \forall n. n \geq n_0 \) then \( a_n = a_{n_0} \) (\( \forall n. n \leq n_0 \) then \( a_n = a_{n_0} \)).

If \( L \) does not have any infinite ascending chains then there exists \( n \) such that \( F^n(\bot) = F^{n+1}(\bot) \) and hence \( \text{lfp}(F) = F^n(\bot) \) (any monotone operator \( F \) over a partially ordered set \( L \) that does not have any infinite ascending chains is also continuous). Similarly, if \( L \) has no infinite descending chains then there exists \( n \) such that \( F^n(\top) = F^{n+1}(\top) \) and hence \( \text{gfp}(F) = F^n(\top) \). The longest path through the partial order from greatest to least element (from \( \top \) to \( \bot \)) defines the height of a lattice.

**3.3.1 Remark.** If \( (L, \sqsubseteq, \sqcup, \sqcap, \top, \bot) \) is a complete lattice of finite height then the sequence defined by Theorem 3.3.2 converges in a finite amount of steps, because \( L \) has neither infinite ascending chains nor infinite descending chains.

**3.3.1.1 Fixed point approximations**

Since we cannot always assume that the sequence \( (F^n(\bot))_{n \geq 0} \) eventually stabilizes, or that it stabilizes after a reasonable number of iterations, we must consider a way of approximating \( \text{lfp}(F) \).

We recall that the operator \( F \) whose fixed point we want to compute defines the behavior of a transition system. The idea in abstract interpretation is to over-approximate the behavior of the transition system. More precisely, to define an abstraction, denoted \( F^a \), of the operator \( F \) whose fixed point we want to compute. The operator \( F^a \) is defined
over a set of states $S^I$. The fixed point computation of $F^I$ is done using the lattice $\mathcal{P}(S^I)$, that is called abstract domain. The domain of $F$ (i.e., $\mathcal{P}(S)$) is called concrete domain.

There are several formalisms allowing to establish a connection between the fixed point of a concrete operator and the corresponding abstract one. In this theses we have considered the most classical one, which is the Galois connection and one which has weaker requirements, the concretization based abstraction [Cousot 1992].

**Galois connection based abstraction** We introduce first the notion of abstract domain based on a Galois connection.

**Definition 3.3.6** (Galois connection). Let $(L, \sqsubseteq)$ and $(L^I, \sqsubseteq^I)$ be two partial order sets. A Galois connection between $L$ and $L^I$ is a pair of functions $(\alpha, \gamma)$ such that:

- $\alpha : L \rightarrow L^I$ is monotonic,
- $\gamma : L^I \rightarrow L$ is monotonic, and,
- $\forall X \in L$ and $\forall X^I \in L^I$
  \[ \alpha(X) \sqsubseteq^I X^I \iff X \sqsubseteq \gamma(X^I). \]

It is denoted by $(L, \alpha, \gamma, L^I)$. Usually, $L$ is called the concrete domain (typically, it represents exactly the states of the transition system), $\alpha$ is called abstraction function because $\alpha(X)$ defines a sound approximation for the set $X$. $L^I$ is called the abstract domain (its elements are an abstract representation for the system’s states), and $\gamma$ is a called the concretisation function.

With this definition a concrete element can be represented by different (incomparable) abstract elements. That is, $\gamma$ is not necessarily injective.

**Definition 3.3.7** (Galois insertion). A Galois connection $(\alpha, \gamma)$ is called Galois insertion if $\alpha \circ \gamma = Id.$

Let $(L, \alpha, \gamma, L^I)$ be a Galois connection, let $F : L \rightarrow L$ be a transformer on $L$, and $F^I : L^I \rightarrow L^I$ a transformer on $L^I$.

**Definition 3.3.8** (Abstract transformer). $F^I$ is an abstract transformer corresponding to $F$ w.r.t. $(L, \alpha, \gamma, L^I)$ iff $\forall X \in L. (\alpha \circ F \circ \gamma)(X) \sqsubseteq F^I(X)$.

Moreover, $F^I$ is a best abstract transformer w.r.t. $(L, \alpha, \gamma, L^I)$ corresponding to $F$ iff $F^I = \alpha \circ F \circ \gamma$. If $\gamma \circ F^I = F \circ \gamma$ then $F^I$ is called exact abstract transformer.

In general, the definition of best (exact) abstract transformers can be quite complicated. On one hand, they depend on the abstraction, on how well it can represent the concrete elements. Notice that the Galois connection ensures that $\alpha$ associates to a concrete element the best (the smallest w.r.t. $\sqsubseteq^I$) abstract element that is an approximation of the concrete one. Even if the abstraction is exact, the abstract transformer is not usually able to capture exactly the behavior of the concrete one.
3.3. ANALYSIS OF TRANSITION SYSTEMS

Concretization based abstraction  One can define an abstract domain using a lighter theory, which does not require a Galois connection between the concrete and the abstract domains but only either the abstraction or the concretization function. The abstract domains that we will consider in the following that do not support a Galois connection are related by a concretization function.

Definition 3.3.9 (Concretization based abstraction). Let \((L, \sqsubseteq)\) and \((L^\sharp, \sqsubseteq^\sharp)\) be two partial order sets. A concretization function is a monotonic function \(\gamma : L^\sharp \to L\). An element \(X^\sharp \in L^\sharp\) is called an abstraction of \(X \in L\) if \(X \sqsubseteq \gamma(X^\sharp)\).

Definition 3.3.10 (Abstract transformer). \(F^\sharp\) is an abstract transformer corresponding to \(F\) w.r.t. a concretization function \(\gamma : L^\sharp \to L\) with \(L\) and \(L^\sharp\) partial order sets if and only if \(\forall X^\sharp \in L^\sharp\) \((\gamma \circ F^\sharp)(X^\sharp) \sqsubseteq (\gamma \circ \gamma)(X^\sharp)\).

If \(\gamma \circ F^\sharp = F \circ \gamma\) then \(F^\sharp\) is called exact abstract transformer.

Because there is no abstraction function \(\alpha\) there is no notion of best abstraction of a concrete element, therefore, there is no notion of best abstract transformer.

Fixed point transfer  The point of introducing an abstract version of a transition system was to compute the least fixed point of the abstract transformer \(\text{post}^\sharp\) in the abstract domain and to relate it to the least fixed point of \(\text{post}\) in the concrete domain. The following theorem by Tarski \cite{Tarski55} defines the transfer between the two fixed points.

Theorem 3.3.3. Given \((L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)\) and \((L^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot^\sharp, \top^\sharp)\) two complete lattices, a monotonic concretization function \(\gamma : L^\sharp \to L\) with \(L\) and \(L^\sharp\) partial order sets if and monotonous transformer \(F : L \to L\), and the monotonic abstract transformer \(F^\sharp : L^\sharp \to L^\sharp\) corresponding to \(F\), then,

\[ lfp_\gamma(Z)(F) \subseteq \gamma(lfp_Z(F^\sharp)) \]

Moreover, if \(F^\sharp\) is exact then \(lfp_\gamma(Z)(F) = \gamma(lfp_Z(F^\sharp))\).

This theorem ensures that the least fixed point of the abstract transformer \(F^\sharp\) is an over-approximation (an abstraction) of the least fixed point of the corresponding concrete transformer \(F\). Notice that this theorem uses only the concretization function, therefore it can be applied when the abstract domain is connected to the concrete domain by a Galois connection or only by a monotonic concretization function.

If the abstract domain \(L^\sharp\) has no infinite ascending chains, then the chain defined in Kleene’s theorem \cite{Kleene55} converges after a finite number of iterations. The lattice of signs introduced in \cite{Cousot77b} is a lattice of finite height and so it has no infinite chains.

If \(L^\sharp\) has infinite ascending chains, or if one wants to speed up the convergence of the fixed point computation, a classical way to compute an over-approximation of the least fixed point is to use a widening operator.

Definition 3.3.11 (Widening operator). Let \((L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)\) be a complete lattice. An operator \(\triangledown : L \times L \to L\) totally defined over \(L \times L\) is a widening operator if and only if:

- \(\forall X, Y \in L, X \sqsubseteq X \triangledown Y\) and \(Y \sqsubseteq X \triangledown Y\),

43
for any ascending chain \((X_n)_{n \geq 0}\) from \(L\), the ascending chain defined by:

\[
X^0_V = X^0, \\
X^{n+1}_V = X^n_V \vee \overline{X^{n+1}},
\]

stabilizes in a finite number of iterations, that is, there exists \(m \geq 0\), such that \(X^m_V = X^{m+1}_V\).

The following theorem, given in [Cousot 1979], shows how the widening operator is used to force (or even to speed up) the convergence of the fixed point computation.

**Theorem 3.3.4.** Given two complete lattices \((L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)\) and \((L^2, \sqsubseteq^2, \sqcup^2, \sqcap^2, \perp^2, \top^2)\), a widening operator \(\triangledown : L^2 \times L^2 \rightarrow L^2\), \(F : L \rightarrow L\) a monotone function, \(F^\triangledown : L^2 \rightarrow L^2\) an abstract transformer corresponding to \(F\), and \(X^2 \in L^2\) such that \(\gamma(X^1) \sqsubseteq F(\gamma(X^2))\) then the sequence \((F^n_V)_{n \geq 0}\) defined by:

\[
F^n_V = \begin{cases} 
X^2, & \text{if } n = 0, \\
F^{n-1}_V, & \text{if } n > 0 \text{ and } F^\triangledown(F^{n-1}_V) \sqsubseteq F^{n-1}_V, \\
F^{n-1}_V \triangledown F^\triangledown(F^{n-1}_V), & \text{otherwise.}
\end{cases}
\]

(3.3.1) eventually stabilizes, that is, there exists \(m \geq 0\) such that \(F^m_V \sqsubseteq F^m\) and \(\text{lfp} \gamma(X^2) \sqsubseteq \gamma(F^m_V)\).

A simple way to improve the precision of the approximation of \(\text{lfp}(F)\) is to continue to compute elements of the sequence, that is \(F^{m+1}_V, F^{m+2}_V\) and so on. For every \(i \geq 1\), \(F^{m+i}_V \sqsubseteq F^{m+i-1}_V\) which means that with every iteration we obtain a potentially smaller element. This computation does not guarantee that we reach the least fixed point. Many times \(X^2\) is chosen such that \(\gamma(X^1) = \perp\), and \(\perp \sqsubseteq F(\perp)\). Notice that \(F^\triangledown\) is not necessarily a monotone operator.

Therefore, to compute an over-approximation for post\(^*(I)\) we must define (1) an abstract set of states \(S^\triangledown\) and the abstract domain \((\mathcal{P}(S^\triangledown), \sqsubseteq^\triangledown)\), (2) a concretization function \(\gamma : \mathcal{P}(S^\triangledown) \rightarrow \mathcal{P}(S)\), (3) an abstract transformer post\(^\triangledown\) corresponding to post, and (4) a widening operator (if the lattice \(\mathcal{P}(S^\triangledown)\) is finite, a widening operator is not necessary but it can be used to speed up the convergence) and then, compute an over-approximation of \(\text{lfp}_I(\text{post}^\triangledown)\), where \(I \sqsubseteq \gamma(I^\triangledown)\).

### 3.3.1.2 Chaotic iteration

Suppose that a state of the transition system is a total function in \([\mathcal{C} \rightarrow S]\), where \(\mathcal{C} = \{c_1, \ldots, c_p\}\) and \(S\) is a possibly infinite set. Then, a state \(s\) is a total state \(s : \mathcal{C} \rightarrow S\) associates to every \(c_i\), \(1 \leq i \leq p\), an element from \(S\) denoted \(s_i = s(c_i)\).

The transition relation is defined by \(\rightarrow \in (\mathcal{C} \rightarrow S) \times \text{Act} \times (\mathcal{C} \rightarrow S)\), such that the successor relation between states post : \((\mathcal{C} \rightarrow S) \rightarrow (\mathcal{C} \rightarrow S)\) can be decomposed in \(p\) components, denoted post\(_k : (\mathcal{C} \rightarrow S) \rightarrow S\) such that

\[
\text{post}((S_1, \ldots, S_p)) = (\text{post}_1((S_1, \ldots, S_p)), \ldots, \text{post}_p((S_1, \ldots, S_p))).
\]

In this case, for the efficiency of the computation, the fixed point equation \(\text{post}((S_1, \ldots, S_p)) = (S_1, \ldots, S_p)\) that we have to solve to compute post\(^*\) can be replaced with the system of equations:

\[
\bigwedge_{i=1}^{p} \text{post}_i((S_1, \ldots, S_p)) = S_i
\]
3.3. ANALYSIS OF TRANSITION SYSTEMS

Given a widening operator $\nabla : S \times S \rightarrow S$, a fixed point solution for this system of equations is computed using the chaotic iteration. It consist roughly in defining the sequences $(\text{post})^n$ for each $1 \leq i \leq p$.

Let $(L, \subseteq, \sqcup, \sqcap, \sqcup, \top)$ be a complete lattice, $C$ a finite set of labels and $(C \rightarrow L, \sqsubseteq, \sqcup, \sqcap, \sqcup, \top)$ be the lattice resulting by point-wise lifting $L$ w.r.t. $C$. Also, let $F_1, \ldots, F_p$ be a set of monotone operators $F_i : (C \rightarrow L) \rightarrow L$ for any $1 \leq i \leq p$ and consider the following system of fixed point equations:

$$\bigwedge_{i=1}^{p} F_i((X_1, \ldots, X_p)) = X_i$$

(3.3.2)

with $|C| = p$ and $1 \leq i \leq p$.

To compute an over-approximation for the least fixed-point of this system of equations, let us consider an abstract lattice $L^2$ and $F_i^2 : [C \rightarrow L^2] \rightarrow L^2$ a sound abstract transformers corresponding to $F_i : [C \rightarrow L] \rightarrow L$ for every $1 \leq i \leq p$.

**Definition 3.3.12 (Chaotic iteration).** Let $W \subseteq C$ be a set of widening points and $(C_q)_{q \in \mathbb{N}}$ an iteration policy, that is an unbounded sequence of sets $C_i \subseteq C$ s.t. every $c \in C$ appears infinitely often in the sequence $(C_q)_{q \in \mathbb{N}}$. Computing a fixed point for 3.3.2 w.r.t. $(C_q)_{q \in \mathbb{N}}$ consists in defining the sequences:

$$(X_i)^{n+1}_q = \begin{cases} 
(X_i)^n_q \quad &\text{if } i \notin C_n \\
(X_i)^n_q \sqcup F_i^q((X_1)^n_q, \ldots, (X_p)^n_q) &\text{if } i \in C_n \text{ and } i \in W \\
(X_i)^n_q \sqcap F_i^q((X_1)^n_q, \ldots, (X_p)^n_q) &\text{if } i \in C_n \text{ and } i \notin W.
\end{cases}
$$

The sequence $(X_j)^{n}_q$ is called iteration policy because at each iteration it defines the $(X_j)^{n}_q$ with $j \in C_n$, such that their computation does not depend on the values of the components $(X_i)^n_q$ with $l \in C_n \setminus \{j\}$, where $n$ is the current iteration. We say that the policy choses the computations that are parallelized.

Each $F_i^q$ depends on some $X_j$ with $j \in C$. These dependencies are represented by a directed graph, whose nodes are the labels in $C$ and an edge between $i$ and $j$ represents the fact that the definition of $F_i^q$ depends on $X_j$, for any $i, j \in C$. Cycles in the dependency graph imply mutual recurrences between the operators $(F_i^q)_{i \in C}$. To ensure the convergence of the fixed point computation, every cycle in the dependency graph must pass through at least one widening point. One could apply widening after each step, but this choice leads to a great loss of precision. Another aspect is the choice of the iteration policy, that could also improve the efficiency and precision of the computation. Iteration policies have been studied in [Bourdoncle 1993].

The following theorem from [Cousot 1979] states the conditions that the chaotic iterations converges in a finite number of iterations to a fixed point of the abstract system.

**Theorem 3.3.5 (Approximative solution of a system of equations using the chaotic iteration).** Let $\bigwedge_{i=1}^{p} X_i = F_i(X_1, \ldots, X_n)$ be a concrete system of fixed point equations $(X_i \in L$, for all $1 \leq i \leq p$). Then, the chaotic iteration 3.3.12 of the abstract system of fixed point equations:

$$\bigwedge_{i=1}^{p} X_i^q = F_i^q(X_1^q, \ldots, X_n^q),$$

w.r.t. a set of widening points $W \subseteq C$ such that every cycle of the dependency graph passes through a node in $W$, according to an iteration policy $(C_q)_{q \in \mathbb{N}}$ and having as initial values
3.3.1.3 Sequential composition

In this thesis we considered a method for the sequential composition of Galois connections and two methods for relational composition of Galois connections, which are defined in Cousot 1979.

3.3.1.3.1 Sequential composition

When developing an analysis sometimes is useful to do it in stages: starting from the concrete lattice \((L, \subseteq, \sqcup, \sqcap, \bot, \top)\), first we introduce an abstract lattice \((L_1^\sharp, \subseteq_1^\sharp, \sqcup_1^\sharp, \sqcap_1^\sharp, \bot_1^\sharp, \top_1^\sharp)\) such that \(L\) and \(L_1^\sharp\) are related by a Galois connection \((L, \alpha_1, \gamma_1, L_1^\sharp)\). Then, one might need to approximate even more the concrete lattice (for reasons like the convergence of the fixed point computation). So if \((L_2^\sharp, \subseteq_2^\sharp, \sqcup_2^\sharp, \sqcap_2^\sharp, \bot_2^\sharp, \top_2^\sharp)\) is an abstract lattice connected through a Galois connection to the lattice \(L_1^\sharp, (L_1^\sharp, \alpha_2, \gamma_2, L_2^\sharp)\). Then, \(L_2^\sharp\) is connected by a Galois connection to the concrete lattice using the “functional composition” of two Galois connections which is a Galois connection: \((L, \alpha_1 \circ \alpha_2, \gamma_2 \circ \gamma_1, L_2^\sharp)\).

Relational composition

Let \((L, \subseteq, \sqcup, \sqcap, \bot, \top)\) be a concrete lattice, \((L_1^\sharp, \subseteq_1^\sharp, \sqcup_1^\sharp, \sqcap_1^\sharp, \bot_1^\sharp, \top_1^\sharp)\) and \((L_2^\sharp, \subseteq_2^\sharp, \sqcup_2^\sharp, \sqcap_2^\sharp, \bot_2^\sharp, \top_2^\sharp)\) be two abstract complete lattices and \((L_1^\sharp, \alpha_1, \gamma_1, L), (L_2^\sharp, \alpha_2, \gamma_2, L)\), two Galois connections.

One way to combine the two abstract domains is to define a Galois connection between their direct product and the concrete lattice: \((L, \alpha, \gamma, (L_1^\sharp \times L_2^\sharp))\). We consider the complete lattice \((L_1^\sharp \times L_2^\sharp, \subseteq_{1\times2}, \sqcup_{1\times2}, \sqcap_{1\times2}, \bot_1^\sharp \times \bot_2^\sharp, \top_1^\sharp \times \top_2^\sharp)\) by lifting the lattice operators of \(L_1^\sharp\) and \(L_2^\sharp\) to the Cartesian product. Then,

\[
\alpha : L \rightarrow L_1^\sharp \times L_2^\sharp, \text{is defined by } \alpha(X) = (\alpha_1(X), \alpha_2(X)) \text{ and }
\gamma : L_1^\sharp \times L_2^\sharp \rightarrow L, \text{is defined by } \gamma(X_1^\sharp, X_2^\sharp) = \gamma_1(X_1^\sharp) \sqcap \gamma_2(X_2^\sharp).
\]

Let \(F : L \rightarrow L\) be a concrete transformer and \(F_1^\sharp : L_1^\sharp \rightarrow L, F_2^\sharp : L_2^\sharp \rightarrow L\), be sound abstract transformers on \(L_1^\sharp\), resp. \(L_2^\sharp\), corresponding to \(F\). A sound abstract transformer on the cartesian product lattice is

\[
F^\sharp : L_1^\sharp \times L_2^\sharp \rightarrow L_1^\sharp \times L_2^\sharp \text{ defined by } F^\sharp(X_1^\sharp, X_2^\sharp) = (F_1^\sharp(X_1^\sharp), F_2^\sharp(X_2^\sharp)).
\]

Computing a fixed point using the direct product increases the precision compared to computing over-approximations using each abstraction individually. The cartesian product of abstractions captures in one shot the information expressed separately by the two abstractions.

There is no precision gained by performing all computations simultaneously instead of performing them one after another and finally taking their conjunctions. In the following, we describe a method such that at each computation step one abstract element in the sequence can benefit from the information gathered by the corresponding element in the other abstract sequence.
3.3. ANALYSIS OF TRANSITION SYSTEMS

Definition 3.3.13 (Reduction operator). An operator \( \sigma : (L_1^2 \times L_2^2) \rightarrow (L_1^2 \times L_2^2) \) such that

\[
\sigma(X_1^2, X_2^2) = \bigcap \{ (Y_1^2, Y_2^2) \mid \gamma_1(Y_1^2) \land \gamma_2(Y_2^2) = \gamma_1(X_1^2) \land \gamma_2(X_2^2) \}
\]

is called a reduction operator.

Since we have considered that \( L \) is related to \( L_1^2 \), respectively \( L_2^2 \), through a Galois connection then \( \sigma = \alpha \circ \gamma \).

Theorem 3.3.6 (Reduced product). Let \( (L_1^1, \alpha_1, \gamma_1, L) \), \( (L_2^2, \alpha_2, \gamma_2, L) \) be two Galois connections and \( \sigma : (L_1^1 \times L_2^2) \rightarrow (L_1^2 \times L_2^2) \) be a reduction operator. Then \( (\sigma(L_1^1 \times L_2^2), \sqsubseteq_1^{\times 2}, \sqcup_1^{\times 2}, \sqcap_1^{\times 2}, \top_1^{\times 2}) \) is a complete lattice, called reduced product of \( L_1^1 \) and \( L_2^2 \), and \( (L, \sigma \circ \alpha, \gamma, \sigma(L_1^1 \times L_2^2)) \) is a Galois connection.

Over-approximating the fixed point of the concrete operator \( F \) using the reduced product is more precise then using the direct product. This happens because instead of over-approximating the fixed point of \( F^\sharp \) we compute the fixed point of \( G^\sharp = \sigma \circ F^\sharp \circ \sigma \).

The problem is that a reduction operator \( \sigma \) is not easy to define, or it might not even exist if the abstraction function is missing. In this case, we can define a less precise reduction after each abstract transformer application \( F^\sharp \). This operator transfers only partially the information between the two abstract domains. It is more precise than the direct product (except for the case when is the identity function).

Definition 3.3.14 (Partial reduction operator). Let \((L_1, \sqsubseteq, \sqcup, \sqcap, \perp, \top)\) be a concrete lattice and \((L_1^2, \sqsubseteq_1^2, \sqcup_1^2, \sqcap_1^2, \top_1^2)\), \((L_2^2, \sqsubseteq_2^2, \sqcup_2^2, \sqcap_2^2, \top_2^2)\) two complete abstract lattices. Also, let \( \gamma_1 : L_1^2 \rightarrow L \) and \( \gamma_2 : L_2^2 \rightarrow L \) be two monotone concretization functions.

An operator \( \sigma : (L_1^2 \times L_2^2) \rightarrow (L_1^2 \times L_2^2) \) defined by

\[
\sigma(X_1^2, X_2^2) = (Y_1^2, Y_2^2) \text{ with }
\]

\[
Y_1^2 \sqsubseteq_1^2 X_1^2, \ Y_2^2 \sqsubseteq_2^2 X_2^2 \text{ and } \gamma_1(Y_1^2) \land \gamma_2(Y_2^2) = \gamma_1(X_1^2) \land \gamma_2(X_2^2)
\]
is called partial reduction operator.

Definition 3.3.15 (Partially reduced product). Let \((L_1^1, \alpha_1, \gamma_1, L)\), \((L_2^2, \alpha_2, \gamma_2, L)\) be two Galois connections and \( \sigma : (L_1^1 \times L_2^2) \rightarrow (L_1^2 \times L_2^2) \) be a partial reduction operator. Then \((\sigma(L_1^1 \times L_2^2), \sqsubseteq_1^{\times 2}, \sqcup_1^{\times 2}, \sqcap_1^{\times 2}, \top_1^{\times 2})\) is a complete lattice, called partial reduced product of \( L_1^1 \) and \( L_2^2 \), and \((L, \sigma \circ \alpha, \gamma, \sigma(L_1^1 \times L_2^2))\) is a Galois connection.

An over-approximation of the least fixed point of the concrete operator \( F \) is computed by computing an over-approximation of the least fixed point of \( \sigma \circ F^\sharp \), using an widening operator, in the direct product lattice \((L_1^2 \times L_2^2, \sqsubseteq_1^{\times 2}, \sqcup_1^{\times 2}, \sqcap_1^{\times 2}, \top_1^{\times 2})\). This corresponds exactly to computing the least fixed point \( F \) in the partially reduced product induced by \( \sigma \).
3.4 Numerical abstract domains

In this section, we give several examples of abstract domains that are going to be used by the analysis introduced in this thesis. These are numerical abstract domains, that is, they define abstractions for scalar types like integers, rationals, or reals. The abstract domains introduced in the literature cover a much wider class of data domains; we provide a detailed comparison in Section 2.2.

Given a set of variables $V$ and a scalar domain denoted $D$ (e.g., $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$), an element of a numerical abstract domain represents sets of valuations from $[V \rightarrow D]$ using a particular language of constraints over $V$. Usually, these abstract domains are connected to the concrete lattice $(P([V \rightarrow D]), \subseteq, \cup, \cap, \emptyset, [V \rightarrow D])$ through a concretization function (some of them are connected through a Galois connection).

The numerical abstract domains can be classified as follows: (1) non-relational abstract domains such as the Signs abstract domain $\text{Cousot 1977a}$ or the Interval abstract domain $\text{Cousot 1977a}$, (2) weakly-relational abstract domains like the Zone abstract domain $\text{Minc 2001}$ or the Octagons abstract domain $\text{Minc 2006b}$, and (3) relational abstract domains which include various abstract domains for linear constraints, e.g., the Polyhedra abstract domain $\text{Cousot 1978}$. There is a trade-off between the precision of an abstract domain and the complexity of an analysis that uses it.

Non-relational abstract domains: These domains capture properties of individual variables and they can not be used to describe relations between variables. They have a poor precision but they are very fast in practice.

The lattice of signs is one of the first abstract domains ever introduced. It is a finite lattice formed of $\{ot, \top\}$ with the following order relation: $[c,c'] \subseteq [d,d']$ iff $d \leq c$ and $c' \leq d'$. The domain does not allow disjunctions therefore, $[c,c'] \cup [d,d'] = \{\min(c,d), \max(c',d')\}$ and $[c,c'] \cap [d,d'] = \{\max(c,d), \min(c',d')\}$. An widening operator is defined for this domain: $[c,c'] \uparrow [d,d'] = [a,b]$ where (1) $a = c$ if $c \leq c'$ and $a = \bot^D$, otherwise and (2) $b = d'$ if $d \leq d'$ and $b = \top^D$, otherwise. The notation $\bot^D$, resp. $\top^D$, stands for the minimal, resp. the maximal, element of $D$. The top element of the interval lattice is the set $D$ and it is defined as $[\bot^D, \top^D]$. The bottom element is the empty interval $[\top^D, \bot^D]$.

Non-relational domains based on congruence relations where introduced in $\text{Granger 1989}$ and $\text{Masdupuy 1993}$. The elements of the abstract domain introduced in $\text{Granger 1989}$ are conjunctions of constraints of the form $v_i \equiv c_i \mod d_i$, where $c_i \in \mathbb{N}^*$ and $d_i \in \mathbb{Z}$. This domain was extended later in $\text{Masdupuy 1993}$ to modulo constraints over intervals. The domain in $\text{Masdupuy 1993}$ associates to every variable $v_i \in V$ an abstract value $[c_i, c_i']$ mod $d_i$, which represents the set of values $(-\infty, c_i') \cup (c_i, +\infty)$ if $c_i > c_i'$ and $d_i = 0$ or otherwise, it is the set of values $\{x \mid x = x_0 \times k + d_i, \text{ for some } x_0 \in \mathbb{Z} \cap [c_i, c_i'] \text{ and } k \in \mathbb{Z}\}$. 

48
3.4. NUMERICAL ABSTRACT DOMAINS

Weakly-relational abstract domains These abstract domains capture relations between variables using linear arithmetical constraints which, in general, are restricted in two ways: either there exists a bound on the number of variables or there exists a bound on the values of the coefficients from the linear constrains. One of the first weakly-relational abstract domains is the abstract domain of Zones introduced in [Miné 2001]. The elements of this domain are conjunctions of constraints of the form \( v_i - v_j \leq c \) and \( \pm v_i \leq c \) where \( v_i, v_j \in V \) and \( c \) is an integer or a real. These constraints are represented by difference bound matrices, called DBM for short (which can be seen as adjacency matrices of graphs).

A difference bound matrix, denoted \( m \), representing a zone constraint, is a \( n \times n \) matrix where \( |V| = n - 1 \). For any \( 0 \leq i \leq n \), \( m[i][i] = 0 \). For any constraint of the form \( v_i - v_j \leq c \), the matrix \( m \) has the value \( m[j][i] = c \). For any constraint of the form \( v_i \leq c \), \( m[0][i] = c \) and for any constraint of the form \( v_i \geq c \), \( m[i][0] = c \).

The operators of the lattice of zones are defined as follows. For any three zones \( m_1, m_2, m_3 \):

- \( m_1 \sqsubseteq m_2 \iff \forall i, j. m_1[i][j] \leq m_2[i][j], \perp \subseteq m_2 \subseteq \top \)
- \( m_1 \sqcup m_2 = m_3 \) where \( \forall i, j. m_3[i][j] = \max(m_1[i][j], m_2[i][j]) \)
- \( m_1 \sqcap m_2 = m_3 \) where \( \forall i, j. m_3[i][j] = \min(m_1[i][j], m_2[i][j]) \)

\( \forall i, j. \top[i][j] = +\infty \) and \( \forall i, j. \perp[i][j] = -\infty \).

The zone abstract domain has infinite ascending chains therefore it requires an widening operator: \( m \sqcup \nabla n = p \) where \( \forall i, j. p[i][j] = m[i][j] \) if \( m[i][j] \geq n[i][j] \) and \( +\infty \) otherwise.

The Octagons abstract domain was introduced in [Miné 2002]. It uses similar constraints, only the signs of the coefficients are relaxed. That is, an element of this abstract domain is a conjunction of constraints of the form \( \pm v_i \pm v_j \leq c \) and \( \pm v_i \leq c \), where \( v_i, v_j \) are variables from \( V \) and \( c \) is an integer or a real. Octagons are represented by DBMs of size \( 2 \cdot n \times 2 \cdot n \), where \( n \) is the number of variables of \( V \).

Furthermore, in [Simon 2002], an abstract domain is introduced, which uses linear constraints over two variables with arbitrary coefficients. This domain is called the TVLPI abstract domain and its elements are conjunctions of constraints of the form \( c_i \cdot v_i + c_j \cdot v_j \leq c \), where \( c_i, c_j, c \) are values in \( \mathbb{D} \).

The Zones abstract domain was extended in [Miné 2002] to the Zone congruence abstract domain whose elements are conjunctions of constraints of the form \( v_i - v_j \equiv c \mod d \), where \( c, d \) are either integers or rationals.

The octahedron abstract domain was introduced in [Clarisó 2004]. Its elements are conjunctions of constraints of the form \( \sum_{v_i \in V} c_i \cdot v_i \leq d_i \), where \( c_i \in \{0, 1, -1\} \) and \( v_i > 0 \).

Relational abstract domains Very early, in [Karr 1976], the domain of linear equalities \( \langle \Sigma_{v_i \in V} c_i \cdot v_i = d \rangle \) was introduced. This domain has a good precision because the lattice of linear equalities (affine relations) has a finite height, so there is no need to have a widening operator. Moreover, all operations are polynomial in the number of variables.

More recently, in [Chen 2010], was introduced the interval linear equality domain, whose elements are conjunction of \( \Sigma_{v_i \in V} [c_i, d_i] \cdot v_i = d \). It enjoys similar complexity as the linear equalities domains, that is all operations are polynomial in the number of variables.

Probably one of the most expressive numerical abstract domains is the Polyhedra domain introduced in [Cousot 1978]. Its elements are conjunctions of linear constraints of the form \( \Sigma_{v_i \in V} c_i \cdot v_i \leq d \).
More recently, in [Chen 2009], was introduced the interval polyhedra domain, whose elements are conjunction of $\Sigma_{v_i \in V}[c_i, d_i] \cdot v_i \leq d$. 
Chapter 4

Programs with dynamic data structures

In this chapter, we introduce SimpleC, a simple imperative C-like language. SimpleC programs can use (1) basic data types like booleans, integers, or reals, (2) record data types containing references to other record types (called pointer fields) or basic data types (called data fields), and (3) arrays of a record type. Therefore, programs manipulate dynamically allocated data structures like doubly-linked lists, arrays, doubly-linked lists of singly-linked lists.

We first describe the syntax and the semantics of programs without procedure calls. We give a reachability collecting semantics [Cousot 1977a] which is storeless, i.e., a heap is represented as a graph where each vertex represents an object of a record type (it is labeled by values of data fields and by program pointer variables) and edges are labeled by sets of pointer fields. Two isomorphic graphs are indistinguishable which implies that addresses of objects in the memory are abstracted away. An object is identified by the unique set of paths that point to it.

For programs with procedure calls, we consider a local heap semantics [], i.e., a procedure is invoked on a part of the heap that contains only the objects reachable from the actual parameters. The relations between the local heap of the callee and the heap of the caller are difficult to model when the connection between the two parts of the heap is not done through actual parameters. Because the main objectives of this thesis are related to the verification of program specifications that describe the shape of the heap but also the size and the data stored in the heap, we have chosen to ignore the relations between the two parts of the heap which concern only the reachability relation between objects. Therefore, we consider cutpoint free programs [Rinetzky 2005b] where, at each procedure invocation, all the paths between objects in the heap of the caller and objects in the local heap pass through objects pointed to by actual parameters.

Assertions and program specifications are written in a multi-sorted first order logic on graphs, called Generalized Composite Structures Logic (gCSL, for short). Formulas of gCSL describe (1) the structure of the allocated memory, also called shape (e.g. the memory contains singly-linked lists, doubly-linked lists, arrays, etc.), (2) the size of the allocated memory, i.e. the number of allocated blocks of record type (the size of a block of record type is considered constant), and (3) the relations between the values of the data fields and between the values of the data fields and the basic type variables, called data relations in the following (e.g. equality between values of data fields and variables, or sortedness properties).

In general, we are interested in verifying a wide spectrum of programs including programs performing data structure traversal to search or to update data, programs with destructive updates (e.g., list dispatch or reversal, sorting algorithms w.r.t. some data
field, such as insertion sort), and programs computing complex arithmetical relations. In this chapter, we give several examples of such programs.

**Outline:** In Section 4.1 we describe the types allowed in SimpleC programs and the type system associated to a program. The syntax and the semantics of programs without procedure calls is defined in Section 4.2. Then, in Section 4.3 we extend this class of programs with procedure calls. Finally, Section 4.4 presents the logic gCSL used for assertions and program specifications.

### 4.1 Data types and program type system

The types used in SimpleC programs are basic data types, user defined data types called record data types, and arrays.

**Basic data types** The basic data types are booleans (denoted `bool`), integers (denoted `short`, `int`, `long`), reals (denoted `float`, `double`) and enumerative data types.

We generally denote by $\mathcal{DT}$ the set of basic data types manipulated by the program. For every basic data type $\mathcal{D}_t \in \mathcal{DT}$, we consider the set of operations $\mathcal{O}_{\mathcal{D}_t}$ and the set of predicates $\mathcal{P}_{\mathcal{D}_t}$ defined in the programming language C. For example, for the `int` data type $\mathcal{O}_{\text{int}} = \{+, -, \div, \%\} \cup \mathbb{Z}$ and $\mathcal{P}_{\text{int}} = \{<, \leq, >, \geq, ==, !=\}$. For every basic data type $\mathcal{D}_t$, the program manipulates expressions built using $\mathcal{O}_{\mathcal{D}_t}$ and boolean expressions built using $\mathcal{P}_{\mathcal{D}_t}$.

**Record types** A record type is a container consisting of members of various types. For simplicity, the programs manipulate only references to record types. The members of a record type are called pointer fields if they denote a reference to a record type or data fields if they denote an object of basic type. We denote by $\mathcal{PF}$ and $\mathcal{DF}$ the set of all pointer fields and respectively, the set of all data fields, used in the program’s type declaration.

The record types correspond to C structures and are defined using the `struct` keyword followed by an identifier name and a list of members declarations. We do not allow inner struct definitions, that is a record type definition contained in another record type declaration. Additionally, all programs have an implicit type called `void`. We denote by $\mathcal{RT}$ the set of all record types declared in the program.

**Arrays** Arrays represent structures of consecutive elements of the same type. For simplicity, we consider only arrays of objects of record types (we do not consider arrays of references). The considered class of programs does not support multi-dimensional arrays (a framework that handles programs with multi-dimensional arrays is given in [Bouajjani 2009b]). To manipulate arrays of basic type objects we assume an encapsulation of each basic type into a record type.

**Example 4.1.1.** The program in Figure 4.1 declares two record types and an array of doubly linked lists. The record `a_ty` (lines 2–6) stores an integer (field `id`) and a reference to a record of type `dll_ty` (field `dll`). The record `dll_ty` (lines 1 and 7–11) defines doubly linked lists storing a boolean (field `flag`) and a reference to an `a_ty` record (field `root`). The variable `a` is declared (line 12) as an array of `a_ty` records. Figure 4.1 also gives a possible heap configuration for this program.
4.1. DATA TYPES AND PROGRAM TYPE SYSTEM

```c
1: typedef struct _dll_ty dll_ty;
2: typedef struct _a_ty a_ty;
3: struct _a_ty {
4: int id;
5: dll_ty* dll;
6: }
7: struct _dll_ty {
8: a_ty* root;
9: bool flag;
10: dll_ty* next, prev;
11: }
12: a_ty a[10];
```

(a) Record types declaration

(b) A memory configuration

Figure 4.1: A declaration for an array of doubly linked lists `a` and a heap for `a`.

**Program type system** Based on the program types declaration, we introduce a **program type system** $\Sigma = (\mathcal{T}, \mathcal{F}, \mathcal{Var}, \tau)$ defined by:

- a set of types $\mathcal{T} = \mathcal{DT} \cup \mathcal{RT}$,
- the set of all (data/pointer) field symbols that are used in the declaration of the types in $\mathcal{T}$, including a set of **array fields** described hereafter, denoted $\mathcal{F} = \mathcal{PF} \cup \mathcal{DF}$ (the set of array fields is included in $\mathcal{PF}$),
- a set of program variables $\mathcal{Var} = \mathcal{DVar} \cup \mathcal{PVar}$, where $\mathcal{DVar}$ denotes the set of program variables of basic type (called **data variables**), and $\mathcal{PVar}$ denotes the set of variables of type reference to a record type or of array type (called **pointer variables**), and
- a typing function $\tau$ mapping each symbol into a (functional) type defined over $\mathcal{T}$.

The typing function $\tau$ associates with a data field $df$ of type $Dt \in \mathcal{DT}$ declared in some record type $Rt \in \mathcal{RT}$ the type $Rt \rightarrow Dt$ (i.e., $df$ represents a mapping from $Rt$ to $Dt$). Similar definitions are given for all pointer fields. A pointer field $f$ is called **recursive** if the domain and the co-domain of $\tau(f)$ are the same. A pointer field $h \in \mathcal{PF}$ is **non-recursive** if the domain and the co-domain of $\tau(h)$ are different.

For every record type $Rt$, we consider a distinct **array field** denoted $aRt$, whose typing is $\tau(aRt) : Rt \rightarrow Rt$. We denote by $\mathcal{AF}$ the set of all array fields. We assume that $\mathcal{AF}$ is a subset of $\mathcal{PF}$. Also, $\mathcal{PF}_r$ denotes the set of recursive pointer fields that are used in the definition of types in $\mathcal{T}$ together with the set of array fields $\mathcal{AF}$.

The typing function $\tau$ associates with each program variable its declaration type: (1) for any variable $p \in \mathcal{Var}$ of type reference to a record type $Rt$, $\tau(p) = Rt$, (2) for any array variable $a \in \mathcal{Var}$ of type array of objects of type $Rt$, $\tau(a) = Rt$, and (3) for any variable $d \in \mathcal{Var}$ of basic type $Dt$, $\tau(d) = Dt$. The program constant $null$ is of type $\text{void}$, i.e., $\tau(null) = \text{void}$. Since programs manipulate only references to record types, the only way to access the array elements is to use terms of the form $\&a[i]$ that denote the reference to the $i$th element of the array $a$. 

53
The components of $\Sigma$ are detailed in Table 4.1. The field symbols are classified (see column *Components*), depending on their typing given in the column *Typing*. Generic notations for representatives of user defined record types and pointer fields are given in the column *Elements*. The last column illustrates the typing system corresponding to Example 4.1.1.

### Table 4.1: Type system associated with programs.

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>Components</th>
<th>Elements</th>
<th>Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>$\mathcal{D}T$</td>
<td>basic types</td>
<td>$\mathcal{D}T \cap \mathcal{R}T = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{D}T$</td>
<td>$\mathcal{D}T$</td>
<td>$\mathcal{R}T$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{R}T$</td>
<td>record types</td>
<td>$\mathcal{R}T$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{D}T \cap \mathcal{R}T = \emptyset$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\mathcal{D}T$</td>
<td>$\mathcal{D}T$</td>
<td>$\mathcal{R}T$</td>
</tr>
<tr>
<td>Field</td>
<td>$\mathcal{D}F$</td>
<td>data fields</td>
<td>$\mathcal{D}F \cap \mathcal{P}F = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{P}F$</td>
<td>pointer fields</td>
<td>$\mathcal{D}F \cap \mathcal{P}F = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{P}F_r$</td>
<td>recursive fields</td>
<td>$\mathcal{P}F_r \subseteq \mathcal{P}F$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{A}F$</td>
<td>array fields</td>
<td>$\mathcal{A}F \subseteq \mathcal{P}F_r$</td>
</tr>
<tr>
<td>Variables</td>
<td>$DVar$</td>
<td>data variables</td>
<td>$\mathcal{D}T \cap \mathcal{R}T = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>$PVar$</td>
<td>pointer variables</td>
<td>$\mathcal{R}T \cap \mathcal{P}F = \emptyset$</td>
</tr>
<tr>
<td></td>
<td>and terms</td>
<td>$\mathcal{D}T \cap \mathcal{R}T = \emptyset$</td>
<td></td>
</tr>
</tbody>
</table>

### 4.2 Programs without procedure calls

We consider programs with no procedures whose variables are basic type variables, references to record types and array variables.

We first introduce the program syntax, which is roughly the syntax of sequential C programs without pointer arithmetic and without procedure calls. Then, we give several examples that are going to be consider along the thesis. Finally, we define a reachability collecting semantics for the programs based on transitions systems. We consider a storeless semantics, that is, we forget about the address of an allocated object. More precisely, the memory is represented by a typed graph, an object is represented by a vertex of the corresponding type and pointer fields are represented by edges. We do not use a representation based on access paths like in [Deutsch 1992], [Venet 1999], [Jonkers 1981].

#### 4.2.1 Program syntax

The program written in *SimpleC* manipulate data type variables, declared by $\mathcal{D}T \ d$, references to record types, declared by $\mathcal{R}T \ *p$, and array variables, declared by $\mathcal{R}T \ *a$ or $\mathcal{R}T \ a[ct]$, where $ct$ is a positive integer.
4.2. PROGRAMS WITHOUT PROCEDURE CALLS

Programs manipulate the heap using memory allocation/deallocation statements (\texttt{new record_type()}/\texttt{new record_type[...]/free}), record field updates (\texttt{x->f=...}), and variable assignments (\texttt{x=...}). The allocation of an object of type \texttt{Rt} is done using the statement \texttt{x = new Rt()} where the variable \texttt{x} is declared as a reference to an object of type \texttt{Rt}. Arrays declared by \texttt{Rt a[ct]} are allocated at declaration time. To dynamically allocate arrays, which are declared by \texttt{Rt * a}, the program uses the statements \texttt{a = new Rt[ct]} and \texttt{a = new Rt[n]}, where \texttt{ct} is positive integer and \texttt{n} is a variable of type \texttt{int}. The programs use different allocation statements for arrays because we consider only arrays of objects of record type and not arrays of references.

The only operations allowed on references are the field access (\texttt{x->f}) and the dereferencing of an array element (\texttt{&a[i]}); no arithmetics on references is allowed. To modify an object of record type, stored as an element of an array, e.g. \texttt{a[3]}, one needs to use an auxiliary reference variable such that \texttt{tmp = &a[3]} and then modify the pointer fields and data fields of \texttt{tmp}.

Boolean conditions on pointers are built using predicates testing for aliasing and definiteness of pointer variables (\texttt{p==q} and \texttt{p==null}). For every basic data type \texttt{Dt}, expressions of type \texttt{Dt} are built using operators in \texttt{O_{Dt}} over variables of type \texttt{Dt} or data fields of type \texttt{Dt}. Then, boolean conditions are defined using predicates in \texttt{P_{Dt}}.

We enrich the C syntax with \texttt{assert(\varphi)} and \texttt{assume(\varphi)} statements, where \varphi is a formula in a logic describing memory configurations which is defined in Chapter 4.4. The control is changed by sequential composition (\texttt{;}), conditionals (\texttt{if-then-else}), and \texttt{while} loops. A program is defined by a declaration of types and a sequence of statements. The variables used in the statements are of basic type, of type reference a record type previously defined, or of type array of objects of record type.

The program syntax is given in Figure 4.2. As usual, the notation \( \alpha^* (\alpha^+) \) denotes a (non-empty) sequence of \( \alpha \)’s.

**Assumptions on the program syntax** We consider that all programs are precompiled as follows:

- each reference assignment of the form \texttt{(&a[i])->f=q} and \texttt{(&a[i])->df=d} is transformed into \texttt{p=&a[i]; p->f=q} and respectively, \texttt{p=&a[i]; p->df=d}, by introducing a fresh pointer variable \texttt{p};
- each reference assignment of the form \texttt{p=new Rt()}, \texttt{p=new Rt[ct]}, \texttt{p=new Rt[n]}, \texttt{p=q}, \texttt{p=&a[i]}, or \texttt{p=q->f} is immediately preceded by an assignment of the form \texttt{p=null};
- a reference assignment of the form \texttt{p=p->f} is turned into \texttt{q=p, p=null, p=q->f}, by introducing a fresh variable \texttt{q};
- each reference assignment of the form \texttt{p->f=q} is immediately preceded by \texttt{p->f=null}.
- data variables at declaration time are initialized with some default constant value \texttt{c_{Dt}}, corresponding to the basic type of the variable \texttt{Dt}.

**Program annotations:** Besides \texttt{assert} and \texttt{assume} statements that are included in the syntax given above, program can be annotated with loop invariants. They are written
between comments (/*...*/ ) immediately after a while statement ( is a formula from the logic used in assert/assume statements):

```
while(...){
  /!*@ loop invariant *@
  ...
```

### 4.2.2 Examples

We present hereafter programs to exemplify the considered syntax. The programs are annotated with assertions (most of them assert statements) which express the properties that the program should have at the indicated control point.

**I. Program Insert** The program given in Figure 4.3 manipulates an array of doubly-linked lists declared as in Figure 4.1. It performs an insertion of an element into a doubly-linked list which is an element of an array. The associated type system is given in Figure 4.4 (it corresponds to the one given in Table 4.1 extended to the program variables).

The program declares `a` as an array of objects of type `a_ty`. Every such object contains a reference to a doubly-linked list. It contains an infinite loop which accesses in a random manner one of the array elements (`rand()` returns a random integer value). It inserts an object of type `dll_ty`, denoted by `node`, in the doubly-linked list whose first element is reachable from `a[k]`, i.e. `tmp1=&a[k]->dll`. It inserts `node` as the third element of this list.
4.2. PROGRAMS WITHOUT PROCEDURE CALLS

void main()
{
a_ty a[10];
assume(''doubly-ll and root-fld and dll-len2''');
int k=0;
while (true) {
    /*@ loop invariant (‘‘doubly-ll and root-fld and dll-len2’’) */
    k = rand() mod 10;
dll_ty p = &a[k];
dll_ty *tmp1 = p->dll;
dll_ty *tmp2 = tmp1->next;
dll_ty *node = new dll_ty();
    node->prev = tmp1;
    node->root = &a[k];
    tmp1->next = node;
    tmp2->prev = node;
    node->next = tmp2;
}
}

Figure 4.3: Insertion of an element in an array of doubly-liked lists.

<table>
<thead>
<tr>
<th>( \Sigma )</th>
<th>in Figure 4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( DT )</td>
<td>basic types</td>
</tr>
<tr>
<td>int, bool</td>
<td></td>
</tr>
<tr>
<td>( RT )</td>
<td>record types</td>
</tr>
<tr>
<td>a_ty, dll_ty</td>
<td></td>
</tr>
<tr>
<td>( DF )</td>
<td>data fields</td>
</tr>
<tr>
<td>( \tau(flag) = dll_ty \rightarrow int )</td>
<td></td>
</tr>
<tr>
<td>( \tau(id) = a_ty \rightarrow int )</td>
<td></td>
</tr>
<tr>
<td>( \tau(next) = dll_ty \rightarrow dll_ty, )</td>
<td></td>
</tr>
<tr>
<td>( \tau(prev) = dll_ty \rightarrow dll_ty, )</td>
<td></td>
</tr>
<tr>
<td>( \tau(root) = dll_ty \rightarrow a_ty, )</td>
<td></td>
</tr>
<tr>
<td>( \tau(aa_ty) = a_ty \rightarrow a_ty, )</td>
<td></td>
</tr>
<tr>
<td>( PF )</td>
<td>pointer fields</td>
</tr>
<tr>
<td>( \tau(stemp1) = dll_ty )</td>
<td></td>
</tr>
<tr>
<td>( \tau(stemp2) = dll_ty )</td>
<td></td>
</tr>
<tr>
<td>( \tau(node) = dll_ty )</td>
<td></td>
</tr>
<tr>
<td>( \tau(a) = a_ty )</td>
<td></td>
</tr>
<tr>
<td>( AF )</td>
<td>array fields</td>
</tr>
<tr>
<td>( DVar )</td>
<td>data variables</td>
</tr>
<tr>
<td>( \tau(k) = int )</td>
<td></td>
</tr>
<tr>
<td>( PVar )</td>
<td>pointer variables</td>
</tr>
</tbody>
</table>

Figure 4.4: The type system induced by the program Insert in Figure 4.3.

Assertions: The assume statement at line 15 fixes the structure of the array \( a \) that the program manipulates. It is an array of doubly-linked lists. The loop invariant states that \( a \) is an array of doubly-liked lists at the beginning of the loop and that this structure is preserved after each iteration of the loop. In order to illustrate the class of annotations we are interested in, we consider some examples of properties that can be asserted at the end of the program and that refer to several aspects such as (1) the shape of the heap, (2) the size of the heap, and (3) the data attached to the elements of the heap:

**Shape:**

\[ \text{doubly-ll} := \text{“the array } a \text{ contains in each cell a reference to an acyclic doubly linked list”,} \]

\[ \text{root-fld} := \text{“each cell in the doubly linked lists stores in the field root a reference to the entry of the array referencing the list”} \]

\[ \text{doubly-cll} := \text{“the doubly linked lists are cyclic”}. \]
typedef struct _list list;

struct _list {
    list* next;
    int dt;
};

void main()
{
    list *a;
    assume( "a-singly-ll");
    list *ai=a;
    int v;
    list* sm, *gr, *tmp;
    sm = tmp = NULL;
    gr = NULL;

    while (ai != NULL)
    {
        tmp = ai->next;
        if (ai->dt <= v)
        {
            ai->next = NULL;
            ai->next = sm;
            sm = NULL;
            sm = ai;
        }
        else{
            ai->next = NULL;
            ai->next = gr;
            gr = NULL;
            gr = ai;
        }
        ai = NULL;
        ai = tmp;
    }
    assert( "gr-singly-ll and gr-greater-v" );
}

Figure 4.5: A program that dispatches the elements of a according to the value of v.

Sizes:

dll-len2 ::= "each doubly-linked list has at least two elements"

dll-len ::= "the array a is sorted in decreasing order of lengths of lists stored"

Data:

sorted-id ::= "the array a is sorted w.r.t. the values of the field id"

flag-1 ::= "there exists a list with all fields flag set to 1"

II. Program Dispatch The program given in Figure 4.5 fits into the category of programs with destructive updates that change the shape of the allocated memory. The corresponding type system is given in Figure 4.6. It uses the record type list where an object of list type is an element of a singly-linked list with one integer data field. The program dispatches the elements of the singly-linked list a to the singly-linked lists pointed to by sm and gr by comparing the value of their data field, dt, with the value of the variable v. Notice that all elements of a are transferred to sm or gr through pointer updates.

Assertions: The assert statement at line 34 states that the program created correctly the list pointed to by gr. In the following, we list the properties from the assert/assume statements in Figure 4.5.
4.2. Programs without procedure calls

<table>
<thead>
<tr>
<th></th>
<th>$\Sigma$</th>
<th>in Figure 4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DT$</td>
<td>basic types</td>
<td>int</td>
</tr>
<tr>
<td>$RT$</td>
<td>record types</td>
<td>next</td>
</tr>
<tr>
<td>$DF$</td>
<td>data fields</td>
<td>$\tau(dt) = \text{list} \rightarrow \text{int}$</td>
</tr>
<tr>
<td>$TF$</td>
<td>pointer fields</td>
<td>$\tau(next) = \text{list} \rightarrow \text{list}$</td>
</tr>
<tr>
<td>$DVar$</td>
<td>data variables</td>
<td>$\tau(v) = \text{int}$</td>
</tr>
<tr>
<td>$PVar$</td>
<td>pointer variables</td>
<td>$\tau(a) = \text{list}, \tau(ai) = \text{list}$</td>
</tr>
</tbody>
</table>

Figure 4.6: The type system induced by the program Dispatch in Figure 4.5.

Shape:

\[\text{a-singly-ll} ::= \text{“a points to an acyclic singly-linked list or it points to null”}\]

\[\text{gr-singly-ll} ::= \text{“gr points to an acyclic singly-linked list or it points to null”}\]

Data:

\[\text{gr-greater-v} ::= \text{“all elements in the list pointed to by gr have the values of the field dt greater than v”}\]

4.2.3 Program semantics

We consider a reachability collecting semantics which defines the set of all memory states reached at each program control point starting from some initial configuration. We consider that programs are represented by intra-procedural control flow graphs. We first define a transition system that describes the behavior of the program. Then, the semantics of the program is defined as the set of reachable states of this transition system. The reachability collecting semantics corresponds to a partition of the set of reachable states with respect to the program control points.

This semantics was introduced in Cousot 1977a. We have chosen the reachability collecting semantics because, in this thesis, we are interested in reasoning about safety properties. Other kinds of program semantics have been designed in the literature, some of them being more precise than the collecting semantics proposed here. For example, to reason about liveness properties one should consider a trace semantics.

For the simplicity of the formalization, in the following we assume that the program manipulates only one basic type, except for the type int used for variables that represent indexes of the arrays, denoted by ($\mathbb{D}, \mathcal{O}, \mathcal{P}$), where $\mathbb{D}$ is the set of values, $\mathcal{O}$ is a set of operations and $\mathcal{P}$ is a set of predicates, except for the index variables that are used in array terms which are of type $\mathbb{Z}$. We distinguish a constant $c_{\mathbb{D}}$ of basic type, that is called the default value of $\mathbb{D}$.

4.2.3.1 Intra-procedural control flow graph

Given a program $P_{\Pi}$, the corresponding intra-procedural control flow graph (CFG, for short) is a directed graph $(N_{\Pi}, E_{\Pi})$ such that $N_{\Pi}$ contains a distinct node for each
program control point in $P$ (which is not associated with variable or type declaration) and the edges in $E_P$ represent the flow of control (they are labeled by program basic statements, boolean conditions from if-then-else or while statements and assume/assert statements). The control points corresponding to tests of boolean conditions (in if-then-else or while statements) are represented by nodes with two successors: one corresponding to the control point where the control is passed if the boolean condition holds (the edge between them is labeled by $\text{cond}$) and one where the control is passed when the boolean condition does not hold (the edge between them is labeled by $\neg\text{cond}$). Let $St_P$ denote the set of labels of the edges in $E_P$.

Any CFG has a node $s_P$ that denotes the unique entry point of the program and a node $e_P$ that denotes the unique exit point (i.e. $s_P$ has no predecessor nodes in $N_P$ and $e_P$ has no successor nodes in $N_P$). All the other nodes have at least one successor and at least one predecessor.

A program configuration is defined by a control point and a memory configuration. To define a transition system over program configurations, we use a representation of memory configurations based on graphs, that we introduce in the following.

4.2.3.2 Representation of program configurations

A memory configuration is defined by the state of the heap (i.e. the dynamically allocated memory) and by the valuation of the program variables, either to values of basic types or to addresses in the heap.

We start by describing the representation that we consider for the heap. We model the heap by a labeled oriented graph called heap graph. Vertices in this graph represent values/objects of record types defined in the program (from $RT$). The graphs are defined over the type system associated with the program $\Sigma = (T,F,Var,\tau)$. We consider two distinguished vertices: $\sharp$ and $\sharp'$ which are the only vertices of void type. $\sharp$ represents the constant null and $\sharp'$ is a node that has no successors and represents all the dangling pointers.

The edges are labeled with pointer fields and array fields from $F$ and represent relations between record objects defined by the values of these pointer/array fields: an edge from $v$ to $v'$ labeled by $f$ represents the fact that the pointer filed $f$ of the record object denoted by $v$ points to the object denoted by $v'$. The values of the data fields are represented by labels on the vertices. An array of record objects of type $Rt$ is represented by an acyclic path, whose edges are labeled by the array field $aRt$ corresponding to $Rt$, and every edge on this path relates two successive cells of the array.

For every pointer field $f \in PF \setminus AF$ that appears in the program (in the definition of some record type), we introduce a new symbol, the “inverse pointer” $\overline{f}$ such that if $\tau(f) : Rt \to Rt'$ then $\tau(\overline{f}) : Rt' \to Rt$ and $\overline{f} = f$. We extend the “inverse” notation to sets of fields, i.e., $\overline{F} = \{ \overline{f} | f \in F \}$ for any set $F \subseteq PF \setminus AF$. We denote by $PF^{\overline{F}} = PF \setminus \overline{AF}$.

We extend the program type system $\Sigma$ over the vertices of the graph. Each vertex is typed by the type of the object it represents.

**Definition 4.2.1** (Heap graph). A heap graph over the type system $\Sigma = (T,F,Var,\tau)$ is an oriented labeled graph $G = (V,E,L,L_D)$, where

- $V$ is a finite set of vertices with $\sharp,\sharp' \in V$,
- $E \subseteq V \times V$ is a finite set of directed edges such that for all $v \in V$, $(\sharp',v) \notin E$ and if $v \neq \sharp$ then $(\sharp,v) \notin E$.
4.2. PROGRAMS WITHOUT PROCEDURE CALLS

- $L : PVar \cup \{null\} \rightarrow V$ is a labeling function for vertices with sets of program variables such that $\sharp = L(null)$,
- $L_E : E \rightarrow 2^{PF^*}$ is a labeling function for edges with sets of pointer fields
- $L_D : V \rightarrow [DF \rightarrow D]$ is a labeling function for vertices with a data field valuation, and
- for every $e = (v, v') \in E$ and every pointer/array field $f$ in $L_E(e)$, if $\tau(v) = Rt$ and $\tau(v') = Rt'$ then $\tau(f) = Rt \rightarrow Rt'$.

Definition 4.2.2 (Well formed heap graph). A heap graph $G$ over $\Sigma = (T, F, Var, \tau)$ is well formed if it satisfies the following constraints:

- determinism: any vertex has at most one successor w.r.t. any pointer/array field in $PF^*$. Formally, for all $v_1, v_2, v_3 \in V$, we have that
  1. if $\{(v_1, v_2), (v_1, v_3)\} \subseteq E$ and $L_E(v_1, v_2) \cap L_E(v_1, v_3) \cap PF \neq \emptyset$ then $v_2 = v_3$,
  2. if $\{(v_1, v_2), (v_3, v_1)\} \subseteq E$ and $L_E(v_1, v_2) \cap L_E(v_3, v_1) \cap AF \cap PF \neq \emptyset$ then $v_2 = v_3$,
  3. if $\{(v_2, v_1), (v_3, v_1)\} \subseteq E$ and $L_E(v_2, v_1) \cap L_E(v_3, v_1) \cap PF \cap AF \neq \emptyset$ then $v_2 = v_3$;
- array well-formedness: array fields create acyclic distinct paths in $G$. Moreover, two arrays never share elements, i.e. for every $aRt$ and $aRt'$ two array fields, there is no edge labeled by $aRt$ and $aRt'$.

Remark that an array may share vertices with a path labeled by some pointer field which is not an array field. We may omit the representation of the dangling node $\sharp'$ if it is not labeled by any program variable.

![Figure 4.7: A heap graph for the program Insert given in Figure 4.3.](image)

4.2.1 Remark. From now on, we consider only well formed heap graphs.

Definition 4.2.3 (Memory configuration/Heap). A memory configuration over a type system $\Sigma = (T, F, Var, \tau)$ is a pair $(G, \delta)$ where $G$ is a well formed heap graph over $\Sigma$...
and $\delta$ is an evaluation for the set of variables $DVar$, i.e., $\delta : DVar \rightarrow D \cup Z$ ($DVar$ contains variables of type $D$ and $Z$, which are interpreted according to their type). Sometimes, $(G, \delta)$ is called a heap and it is denoted by $H$. Let $\mathcal{H}(\Sigma)$ be the set of all heaps over $\Sigma$.

**Example 4.2.1.** The heap graph given in Figure 4.7 represents the configuration of the allocated memory pictured in Figure 4.1(b). It is defined over the type system given in Figure 4.4 and over the pointer variables $PVar = \{a, \text{tmp1}, \text{tmp2}, \text{node}, \text{null}\}$. We use different shapes for the vertices in order to point out their different types: circles for vertices representing $\texttt{a}$.ty records and boxes for vertices representing $\texttt{dll}$.ty records. The vertices are labeled by the values of the data fields, and some of them by labels in $PVar$. The edges labeled by the array field $\texttt{aa}$.ty represent the relation between successive cells in the array $a$. The labels $\texttt{prev}$ and $\texttt{root}$ represent the inverse of the relations defined by $\texttt{prev}$ and $\texttt{root}$ respectively. There are no dangling pointers therefore $\sharp'$ is not labeled.

This heap graph represents a configuration of the program Insert given in Figure 4.3.

**Example 4.2.2.** The heap graph in Figure 4.8 represents a state of the allocated memory in the program Dispatch given in Figure 4.5. It is a heap graph built over the type system given in Figure 4.6 that contains only one record type $\texttt{list}$. An element of type $\texttt{list}$ is represented by an ellipse. There are no dangling pointers therefore $\sharp'$ is not labeled.

Notice that, using inverse pointer fields we represent heaps by oriented labeled graphs (i.e., it contains at most one edge between any two vertices) but this representation is not unique. That is, an edge between two vertices $v$ and $v'$ can be labeled by $v^{\{\text{next,prev}\}} \rightarrow v'$ or equivalently, by changing the orientation of the edge, it becomes $v^{\{\text{next,prev}\}} \Rightarrow v$. These representations correspond to a unique graph where we allow multiple edges between vertices, each of them being labeled by a unique pointer field (e.g. two edges between $v$ and $v'$: $v^{\text{next}} \rightarrow v'$ and $v^{\text{prev}} \Rightarrow v$).

**Definition 4.2.4** ($F$-path). Given a well-formed heap graph $G$ and two vertices $v$ and $v'$ in $G$, we call $F$-path between $v$ and $v'$ in $G$, the only path in $G$ between $v$ and $v'$ such that every edge in this path is labeled with the set of pointer fields $F$.

**Definition 4.2.5** (Reachability relation on heaps). A vertex $v'$ is reachable from another vertex $v$ in a heap $(G, \delta)$ if there exists an $F$-path in $G$ from $v$ to $v'$, for some set of pointer fields $F$.

**Definition 4.2.6** (Error heap). An error in a program is signaled using a distinguished heap denoted by $(G_{err}, \delta_{err})$. $(G_{err}, \delta_{err})$ belongs to $\mathcal{H}(\Sigma)$ for any type system $\Sigma$. 

62
The state of the allocated memory is garbage-free if all the allocated memory cells are reachable from the program variables. A heap graph is called garbage-free if it represents a garbage-free state of the allocated memory.

**Definition 4.2.7 (Garbage-free heap graph).** A heap graph \( G = (V, E, L, L_E, L_D) \) over the type system \( \Sigma \) is called garbage free if for any vertex \( v \in V \setminus \{*, *'\} \), \( v \) is either labeled by a program variable in \( P\text{Var} \) or there exists a path in \( G \) from a vertex labeled by a program variable in \( P\text{Var} \) to \( v \).

**Example 4.2.3.** The heap graphs shown in Figures 4.7 and 4.8 are garbage-free heap graphs.

### 4.2.3.3 Concrete transformers for program statements

Let \( P_H \) be a program, and let \( \Sigma = (\mathcal{F}, \mathcal{V}, \text{Var}, \tau) \) be its type system. Also, let \((N_{P_H}, E_{P_H})\) be the control flow graph associated with \( P_H \).

A program state is a pair \((pc, (G, \delta))\) where \( pc \in N_{P_H} \) is a node of the control flow graph and \((G, \delta)\) is a memory configuration in \( \mathcal{H}(\Sigma) \). The program defines a transition system \((S, Act, \rightarrow, I)\) over the state space \( S = N_{P_H} \times \mathcal{H}(\Sigma) \), the set of actions \( Act \) containing the labels of the edges in the control flow graph and some set of initial states \( I \).

The transition relation \( \rightarrow \subseteq S \times Act \times S \) is defined by:

\[
(pc, (G, \delta)) \xrightarrow{a} (pc', (G', \delta')) \quad \text{iff} \quad pc \xrightarrow{a} pc' \in E_{P_H}, \quad F[a] \text{ defined in } (G, \delta), \text{ and } (G', \delta') = F[a](G, \delta)
\]

where \( F[a] : \mathcal{H}(\Sigma) \rightarrow \mathcal{H}(\Sigma) \) is a partial function, called *concrete transformer*, defined hereafter.

We define the concrete transformer \( F[a] \) for any \( a \in Act \) which is a program assignment, a program boolean condition or an `assert/assume` statement.

The statement \( x = \textbf{new} \ Rt() \) allocates a new memory block pointed to by \( x \) representing an object of type \( \text{Rt} \). The corresponding transformer \( F[x := \textbf{new} \ Rt()] \) adds to the heap graph a new vertex \( v \) of type \( \text{Rt} \) labeled by \( x \) and by the data field valuation defined by \( L_D(v)(df) = c_D \), for every data field \( dt \) in \( \text{Rt} \). The statement \( a = \textbf{new} \ Rt[n] \) allocates an array of \( \delta(n) \) elements of type \( \text{Rt} \), so the corresponding transformer \( F[a := \textbf{new} \ Rt[n]] \) adds to the heap graph \( \delta(n) \) new vertices of type \( \text{Rt} \) connected through a path with edges labeled by \( \text{pRt} \) such that the first node is labeled by the program variable \( a \). The transformer \( F[a := \textbf{new} \ Rt[\text{ct}]] \), where \( \text{ct} \) is a positive integer, is defined in a similar manner. The static allocation of an array \( \text{Rt} a[\text{ct}] \), where \( \text{ct} \) is a positive integer, can be rewritten as \( \text{Rt} \ a[\text{ct}] := \textbf{new} \ Rt[\text{ct}] \). The transformer \( F[\textbf{free}(x)] \) removes the vertex labeled by \( x \) and labels \( x' \) with \( x \). The transformers corresponding to pointer assignments like \( x=y \) and \( x=*y \rightarrow f \) modify only the heap graph vertex labeling, while the ones corresponding to assignments like \( x->f=y \) modify the vertex labeling and the edges in the graph. The formal definition of the transformers corresponding to pointer manipulation is given in Figure 4.9, Figure 4.11 and Figure 4.10 where \((G, \delta)\) is a memory configuration in \( \mathcal{H}(\Sigma) \) and \( G = (V, E, L, L_E, L_D) \). It uses the assumptions on the program syntax defined in Section 4.2.1.
Figure 4.9: Concrete transformers for `new/free` on \((G = (V, E, L_E, L_D), \delta)\).

Figure 4.10: Concrete transformers for pointer assignments on \((G = (V, E, L_E, L_D), \delta)\).

The concrete transformers corresponding to data assignments are given in Figure 4.12 where `eval` is a function that evaluates a data expression in a memory configuration \((G = (V, E, L_E, L_D), \delta)\).

For any boolean condition `cond` labeling an edge of the control flow graph, the transformer \(F[\text{cond}](G, \delta)\) is undefined if `cond` does not hold in the configuration \((G, \delta)\) (the function `evalb` evaluates a boolean condition). If the boolean condition evaluates to true, the transformer leaves the configuration unchanged. Acceding dangling pointer is considered an error. A boolean condition where dangling pointers are accessed is evaluated to \(\bot\). Any boolean condition evaluates to \(\bot\) if one of the literals evaluates to \(\bot\). \(F[\text{cond}](G, \delta) = (G_{\text{err}}, \delta_{\text{err}})\) if the boolean condition `cond` is build with dangling pointer variables.

The language of assertions is more expressive than the boolean conditions used in the program. A formula \(\varphi\) from an `assert/assume` statement may express properties of the shape of the allocated memory but also properties of the values of the data fields (the syntax of these formulas is given in Section 4.4). We denote by \([\varphi]\) the set of models of \(\varphi\). Then, \(F[\text{assert } \varphi](G, \delta)\) is the error heap \((G_{\text{err}}, \delta_{\text{err}})\) if \((G, \delta)\) is not a model of \(\varphi\) and \(F[\text{assert } \varphi](G, \delta) = (G, \delta)\), otherwise. In the case of `assume` statements, \(F[\text{assume } \varphi](G, \delta) = (G_{\text{err}}, \delta_{\text{err}})\) if \(\varphi\) has no model otherwise \(F[\text{assume } \varphi](G, \delta) = (G', \delta')\), where \((G', \delta') \in [\varphi]\). The full definition of these transformers is given in Figure 4.13 and 4.14.

4.2.2 Remark. The concrete transformers defined so far, \(F[a]\) with \(a \in \text{Act}\), are defined
4.2. PROGRAMS WITHOUT PROCEDURE CALLS

\[ F[x\rightarrow f](G, \delta) = (G', \delta) \ \\
\text{where} \ \\
(G', \delta) = (G_{err}, \delta_{err}), \text{ if } L(y) = v \text{ or } L(y) = v'
\]

\[ G' = (V, E, L', L_E, L_D) \text{ with } L(y) = v_y, \ \\
L'|_{PVar \setminus \{x\}} = L, \text{ and:} \ \\
\begin{itemize}
  \item if \((v_y, v'_y) \in E \text{ and } f \in L_E(v_y, v'_y)\) then \ \\
    \(L'(x) = v'_y,\)
  \item if \((v'_y, v_y) \in E \text{ and } f \in L_E(v'_y, v_y)\) then \ \\
    \(L'(x) = v'_y,\) otherwise.
\end{itemize}

\[ F[x\rightarrow f](G, \delta) = (G', \delta) \ \\
\text{where} \ \\
(G', \delta) = (G_{err}, \delta_{err}), \text{ if } L(x) = v \text{ or } L(x) = v'
\]

\[ G' = (V, E', L', L'_E, L'_D) \text{ with } L(x) = v_x, L(y) = v_y \ \\
(v'_x, v_x) \in E \text{ such that } f \in L_E(v'_x, v_x), \text{ and:} \ \\
\begin{itemize}
  \item if \(L_E(v'_x, v_x) = \{f\} \text{ and } (v_x, v_y) \in E \text{ then} \ \\
    E' = E \setminus (v'_x, v_x) \text{ and } L'_E(v_x, v_y) = L_E(v_x, v_y) \cup \{f\}\)
  \item if \(L_E(v'_x, v_x) = \{f\} \text{ and } (v_x, v_y) \notin E \text{ then} \ \\
    E' = (E \setminus (v'_x, v_x)) \cup (v_x, v_y) \text{ and } L'_E(v_x, v_y) = \{f\}\)
  \item if \(L_E(v'_x, v_x) \supset \{f\} \text{ and } (v_x, v_y) \notin E \text{ then} \ \\
    E' = E \cup (v_x, v_y), \text{ and } L'_E(v_x, v_y) = L_E(v_x, v_y) \setminus \{f\}\)
  \item if \(L_E(v'_x, v_x) \supset \{f\} \text{ and } (v_x, v_y) \in E \text{ then} \ \\
    E' = E, \text{ and } L'_E(v_x, v_y) = L_E(v'_x, v_x) \setminus \{f\}\)
\end{itemize}

Figure 4.11: Concrete transformers for pointer assignments on \((G = (V, E, L, L_E, L_D), \delta)\).

over heaps which are not garbage free. Moreover, if the input heap is garbage free then vertices representing garbage may be introduced by pointer assignments of the form \(x = \text{null}\) and \(x\rightarrow f = \text{null}\).

To define a garbage free semantics, we compose the definition of each \(F[a]\) with the concrete transformer \(\text{elim\_gb}\), where \(\text{elim\_gb} : \mathcal{H}(\Sigma) \to \mathcal{H}(\Sigma)\) removes all the vertices in the heap graph that are not reachable from a vertex labeled by some program variable.

Definition 4.2.8 (Garbage collector). Let \(H = (G, \delta)\) and \(G = (V, E, L, L_E, L_D)\). We define \(\text{elim\_gb}(G, \delta) = (G', \delta)\), where \(G' = (V', E', L', L'_E, L'_D)\) such that:

- \(V' = V \setminus \{v \mid \text{for any } x \in \text{PVar and } F \in \mathcal{P}F, \text{ there is no } F\text{-path from } L(x) \text{ to } v\}\),
CHAPTER 4. PROGRAMS WITH DYNAMIC DATA STRUCTURES

\[
\begin{align*}
\text{eval}(G, \delta, d) &= \delta(d) \\
\text{eval}(G, \delta, x \to df) &= L_D(v)(dt), \text{ where } L(x) = v \\
\text{eval}(G, \delta, O(dt_1, \ldots, dt_n)) &= O(\text{eval}(G, \delta, dt_1), \ldots, \text{eval}(G, \delta, dt_n))
\end{align*}
\]

\[F[d = dt](G, \delta) = (G, \delta') \quad \text{where} \quad \delta' \downharpoonright \text{DVar}\{d\} = \delta \text{ and } \delta'(d) = \text{eval}(G, \delta, dt)\]

\[F[x \to df = dt](G, \delta) = (G', \delta') \quad \text{where} \quad G' = (V, E, L, L_E, L_D), L(x) = v \in V,\]

\[L_D \mid_{\forall(v)} = L_D, \text{ and } L_D'(v) = \text{eval}(G, \delta, dt)\]

Figure 4.12: Concrete transformers for data assignments on \((G = (V, E, L, L_E, L_D), \delta)\).

\[
\begin{align*}
\text{eval}_4((G, \delta), P(dt_1, \ldots, dt_n)) &= P(\text{eval}(G, \delta, dt_1), \ldots, \text{eval}(G, \delta, dt_n)) \\
\text{eval}_4((G, \delta), p == q) &= \begin{cases} 
\text{true, if there exists } v \in V \setminus \{\gamma'\} \text{ such that } L(p) = L(q) = v, \\
\perp, \text{ if } L(p) = \gamma' \text{ or } L(q) = \gamma' \\
\text{false, otherwise.}
\end{cases} \\
\text{eval}_4((G, \delta), p == \text{null}) &= \begin{cases} 
\perp, \text{ if } L(p) = \gamma' \\
\text{false, otherwise.}
\end{cases} \\
F[\text{cond}](G, \delta) &= \begin{cases} 
(G_{\text{err}}, \delta_{\text{err}}), \text{ if } \text{eval}_4((G, \delta), \text{cond}) == \perp, \\
(G, \delta), \text{ if } \text{eval}_4((G, \delta), \text{cond}) == \text{true, undefined, otherwise.}
\end{cases}
\end{align*}
\]

Figure 4.13: Concrete transformers for boolean conditions

\[
\begin{align*}
F[\text{assert } \varphi](G, \delta) &= \begin{cases} 
(G, \delta), \text{ if } (G, \delta) \in \llbracket \varphi \rrbracket, \\
(G_{\text{err}}, \delta_{\text{err}}), \text{ otherwise.}
\end{cases} \\
F[\text{assume } \varphi](G, \delta) &= \begin{cases} 
(G', \delta'), \text{ where } (G', \delta') \in \llbracket \varphi \rrbracket \neq \emptyset, \\
(G_{\text{err}}, \delta_{\text{err}}), \text{ if } \llbracket \varphi \rrbracket = \emptyset.
\end{cases}
\end{align*}
\]

Figure 4.14: Concrete transformers for assert and assume statements

- \(E' = E \setminus \{(v, v') \mid v \notin V' \text{ or } v' \notin V'\}\),
- \(L' = L |_{V'}, L'_E = L_E |_{E'}, \text{ and } L'_D = L_D |_{V'}\).

Notice that, by the definition of the transition system, when \(F[a](G, \delta)\) is undefined, there exists no transition labeled by \(a\) from \((G, \delta)\).

For any action \(a \in \text{Act}\), we extend \(F[a]\) to sets of program configurations as usual.

4.2.3.4 Reachability collecting semantics

For the transition system defined above, we consider the operator \(\text{post} : 2^S \to 2^S\), as defined in Chapter 3.

Given a set of initial program configurations \(I\), the reachability collecting semantics of a program \(P_I\) is defined as the set of reachable states of the transition system defined by \(P_I\), i.e. \(\text{post}^*(I)\) which is the least fixed point of \(\text{post}\) larger than \(I\), denoted by \(\text{lfp}_I(\text{post})\). The set of reachable states at a control point \(pc\) in the CFG of \(P_I\) is obtained by restricting \(\text{post}^*(I)\) to the states whose first component equals \(pc\).

We define an isomorphism relation between heaps over the same type system \(\Sigma\) based on the notion of graph isomorphism.

66
4.2. PROGRAMS WITHOUT PROCEDURE CALLS

Definition 4.2.9 (Heap isomorphism). Two heaps over $\Sigma$, $H = (G = (V, E, L, L_E, L_D), \delta)$ and $H' = (G' = (V', E', L', L'_E, L'_D), \delta')$ are isomorphic, denoted $H \sim H'$, iff

1. the underlying graphs $G$ and $G'$ are isomorphic, i.e. there is a bijection $h : V \rightarrow V'$ such that (1) any two vertices $v_1$ and $v_2$ from $V$ are adjacent in $G$ iff $h(v_1)$ and $h(v_2)$ are adjacent in $G'$, (2) for any two vertices $v_1$ and $v_2$ from $V$ adjacent in $G$, $L_E(v_1, v_2) = L'_E(h(v_1), h(v_2))$, and (3) for any $v \in V$, $L(x) = v$ iff $L'(x) = h(v)$, and $L_D(v) = L'_D(h(v))$,

2. for any $d \in DVar$, $\delta(d) = \delta'(d)$.

Notice that $\sim$ is an equivalence relation.

The concrete lattice of heaps for the program $P_{\Pi}$, denoted by $C(\Sigma)$, is defined by:

$$C(\Sigma) = (\mathcal{P}(\mathcal{H}(\Sigma)/\sim), \subseteq / \sim, \cup, \cap, \emptyset, \mathcal{H}(\Sigma)/\sim),$$

where $\mathcal{H}(\Sigma)/\sim$ is the quotient set of $\mathcal{H}(\Sigma)$ by the equivalence relation $\sim$ and the order relation $\subseteq / \sim$ is defined over equivalence classes. The concrete transformer $F[a] : \mathcal{H}(\Sigma) \rightarrow \mathcal{H}(\Sigma)$, define in Section 4.2.3.3 is compatible with the equivalence relation $\sim$, that is

$$\text{if } H \sim H' \text{ then } F[a](H) \sim F[a](H')$$

for every $H, H' \in \mathcal{H}(\Sigma)$ and $a \in Act$. This implies that post is also compatible with the relation $(=, \sim)$, where $=$ is the equality relation on control points.

The set of states $S$ can be seen as a total function from nodes in the CFG $(N_{P_{\Pi}}, E_{P_{\Pi}})$ to sets of memory configurations. Thus a set of states is denoted by a tuple $(X_{n_1}, \ldots, X_{n_q})$ where $N_{P_{\Pi}} = \{n_1, \ldots, n_q\}$ with $s_{P_{\Pi}} = n_1$ (the start node), $e_{P_{\Pi}} = n_q$ (the exit node), and $X_{n_i} \subseteq \mathcal{H}(\Sigma)$, for every $1 \leq i \leq q$. It corresponds to $\{(n_1, x_1) \mid x_1 \in X_{n_1}\} \cup \ldots \cup \{(n_q, x_q) \mid x_q \in X_{n_q}\}$.

The operator post can also be decomposed into $|N_{P_{\Pi}}|$ components:

$$\text{post}_i : 2^{\mathcal{H}(\Sigma)/\sim} \times \cdots \times 2^{\mathcal{H}(\Sigma)/\sim} \rightarrow 2^{\mathcal{H}(\Sigma)/\sim}$$

times q \text{ times}

for every $1 \leq i \leq q$.

We extend $C(\Sigma)$ to the lattice of tuples of sets of concrete memory configurations by:

$$C_{pe}(\Sigma) = \left(\mathcal{P}(\mathcal{H}(\Sigma)/\sim))^p, \subseteq, \cup, \cap, \emptyset, \mathcal{H}(\Sigma)/\sim\right).$$

Then, the semantics is equivalently defined (modulo the equivalence relation $\sim$) by the least fixed point of the following system of equations in the lattice $C_{pe}(\Sigma)$:

$$\bigwedge_{i=1}^{q=|N_{P_{\Pi}}|} \text{post}_i(X_{n_1}, \ldots, X_{n_q}) = X_i$$

where

$$\text{post}_i(X_{n_1}, \ldots, X_{n_q}) = \bigcup_{n_j, n_i \in N_{P_{\Pi}}, n_j \sim n_i \in E_{P_{\Pi}}} F[a](X_{n_j}), \text{ for all } 2 \leq i \leq q,$$ and
CHAPTER 4. PROGRAMS WITH DYNAMIC DATA STRUCTURES

\[
\text{post}_1(X_{n_1}, \ldots, X_{n_q}) = I.
\]

If the set of heap graphs associated to a control point contains the error heap \( H_{err} \) then, at that control point, null pointer dereference has occurred or a dangling pointer has been used or an assert statement has failed. It is just a matter of formalization to distinguish between these errors, but for the purpose of this exposition signaling only one type of errors is sufficient.

4.3 Extending programs with procedure calls

We extend the programs define in SimpleC (given in Section 4.2) with procedure calls. First, we modify the syntax given in Section 4.2.1 and then, we define the program semantics.

There are two semantics which are widely used to describe the behavior of programs with procedures: the operational semantics and the relational semantics [Sharir 1981]. The first one uses a stack of valuations for the program variables, that is, a configuration for a procedure is a tuple between the allocated memory and the call stack. The relational semantics [Sharir 1981, Cousot 1977b] is based on computing relations between input and output memory configurations for each procedure. In this thesis, we have chosen the second approach.

We consider a local semantics for procedural calls, as introduced in [Rinetzky 2005a]. This semantics has the advantage of characterizing the behavior of a procedure by ignoring the parts of the heap that are not relevant to the procedure. Furthermore, we use a storeless representation for relations between program configurations, based on graphs, which abstracts away the addresses in the memory. This abstraction is problematic in the context of procedure calls. As we have already said, a procedure receives as input only a part of the heap, called local heap. When returning from the call, due to destructive updates, the connections between the local heap and the rest of the heap can not in general be reestablished without storing some additional information at call time. We will consider a class of programs, called cutpoint free programs, which do not pose this problem.

4.3.1 Syntax

A program is defined by a set of procedures, with a special procedure called main, which is the procedure called when the execution of the program begins.

A procedure \( P_\pi \) is defined by a prototype and by a sequence of statements called the body of the procedure. The prototype of a procedure consists in the number, the type and the names of its input and output parameters and the type of the returned value, if any. Parameters may be of type reference to a record type or of basic data type.

The body of a procedure is a sequence of variable declarations and program statements as in Section 4.2.1 (assignments, if-then-else, while, and assume/assert statements), and procedure calls.

The syntax of the class of programs we consider is given in Figure 4.15. We forbid pointers to procedures. The vectors of program variables \( ai \) and \( ao \) denote the actual input parameters, respectively the actual output parameters corresponding to a call of the procedure \( proc\_name \). The declaration of a procedure \( proc\_name \) consists in defining
4.3. EXTENDING PROGRAMS WITH PROCEDURE CALLS

Procedure calls: \[ \text{procStmt} ::= q = P_{\pi}(a_i, a_o) \mid P_{\pi}(a_i, a_o) \]

Statements: \[ \text{stmt} ::= \text{asgnStmt} \mid \text{ifStmt} \mid \text{whileStmt} \mid \text{assert}(\varphi) \mid \text{assume}(\varphi) \mid \text{procStmt} \]

Type name: \[ \text{type} ::= \text{Rt}^* \mid \text{Dt} \]

Procedure: \[ \text{proc decl} ::= \text{type} \text{name proc name}((\text{type} \text{name} f_i)^*, ((\text{type} \text{name} & f_o)^*)) \]

Declarations: \[ \{	ext{var decl}^{*} \text{stmt}^{*} \text{return}[\text{ret}_{P\pi}];\} \]

Programs: \[ \text{program} ::= \text{type decl}^{*} \text{proc decl}^{*} \]

Figure 4.15: SimpleC syntax.

(1) the type of the return value, (2) the types and the names of the formal input and output parameters, denoted by \( f_i \) and respectively \( f_o \), (output parameters are passed by reference), (3) a sequence of variable declarations and program statements, and (4) the returned value if any. The syntax of \( \text{asgnStmt} \), \( \text{ifStmt} \), \( \text{whileStmt} \), \( \text{assert}(\varphi) \), and \( \text{assume}(\varphi) \) is given in Figure 4.2.

The \text{assert}/\text{assume} statements use formulas from a logic that we define in the next section, and describe relations between memory configurations (see Section 4.3.3 for more details).

We denote by \( fpi(fpo) \) the formal input (output) parameters in \( f_i(f_o) \) which are of type reference to a record type (array variables are also reference to a record type). Also, \( fdi(fdo) \) denotes the formal input (output) parameters which are of basic data type. Similarly, we consider \( ai = api \cup adi \) and \( ao = apo \cup ado \) where \( apo, api \) are variables of type reference to a record type and \( ado, adi \) are variables of basic data type. For a procedure \text{name proc name} we denote by \( \text{loc} \) the set of variables used by the procedure. The set \( \text{loc} \) includes the formal input and output parameters \( f_i \) and \( f_o \) and the variables declared in the body of the procedure. If the returned type of the procedure is not \text{void}, we consider a distinguished output parameter \( \text{ret}_{P\pi} \in f_o \), for the returned variable. If \( \text{ret}_{P\pi} \) is of basic type then \( \text{ret}_{P\pi} \in fdo \), otherwise \( \text{ret}_{P\pi} \in fpo \).

We denote by \( DVar \) and \( PVar \) the set of data variables, respectively pointer variables, used in the program.

Assumptions: We assume that each procedure uses a distinct set of variables, and that the output parameters are assigned to \text{null} before the call. Also, we consider that each procedure before it terminates (before the \text{return} statement) assigns \text{null} to all pointer variables that are not formal parameters. W.l.o.g. we assume that all input parameter are passed by value. This does not restrict the class of considered programs. Also, we consider that all local data variables are initialized with the default value corresponding to the type of the variable.

Program annotations: The formulas used in \text{assert}/\text{assume} statements do not describe memory configurations as in the intra-procedural case. An \text{assert}/\text{assume} statement at some control point \text{pc} belonging to a procedure \( P_{\pi} \) describes relations between input memory configurations for \( P_{\pi} \) and memory configurations reachable at \text{pc}.

For every procedure, its specification is given as an annotation (it is written between comments /*...*/ directly in the source files). It consists in a precondition and a postcondition: the precondition describes the input memory configurations and the postcondition
describes the (relation between the input configurations and ) the output memory configurations. These annotations are placed before the declaration of the procedure, using the keywords \texttt{requires} and \texttt{ensures}. More precisely, for every procedure $P_{\pi}$, the specification is given by \texttt{@requires} $\varphi_{\text{pre}}$ and \texttt{@ensures} $\varphi_{\text{post}}$, where $\varphi_{\text{pre}}$ is a formula in the logic of assertions which specifies the precondition of the procedure and $\varphi_{\text{post}}$ is a formula in the same logic specifying the postcondition of the procedure:

\begin{verbatim}
/*@requires \varphi_{\text{pre}}
@ensures \varphi_{\text{post}}*/
\end{verbatim}

4.3.2 Examples

As in the intra-procedural case, we consider programs that perform data structure traversal, programs with destructive updates that change the shape of the allocated memory (e.g., list dispatch or reversal, sorting algorithms), and programs computing complex arithmetical relations. However, the programs considered in this section have a more complex control structure.

III. Program List sharing

The program in Figure 4.16 creates two lists $x_i$ and $z_i$ using the procedure \texttt{alloc} (we assume a standard implementation for this procedure) and then, it calls the procedure \texttt{list\_share}. The latter receives as input two singly-linked lists and reorders their elements such that the values belonging to both of them are the last ones. Moreover, only one copy of the common values is stored. It uses the record type \texttt{list} defined in Figure 4.5.

IV. Program Fibonacci

An example that fits into the category of programs creating complex arithmetical relations is the one given in Figure 4.17. The program creates a list using the procedure \texttt{alloc} given in Figure 4.16 and then, it initializes the list using the procedure \texttt{Fibonacci}. This procedure takes a list as input (defined using the record type \texttt{list}) and initializes the data field of its elements with the first numbers from the Fibonacci sequence. After returning from the call to \texttt{alloc}, the \texttt{assert} statement at line 11 checks that $a$ has been correctly created. Also, the \texttt{assert} statement at line 14 checks that after returning from the call to \texttt{Fibonacci} the list is correctly initialized and its length is preserved.

Figure 4.18 describes the type system induced by the program given in Figure 4.17, where \texttt{var\_alloc} is a generic name that stands for any variable used in the body of the procedure \texttt{alloc}. We recall that a type system for a program assigns a type to all the variables used in the program, including the variables used by the different procedures declared in the program.

V. Program AddV

The program in Figure 4.19 creates a list of length 10 using the procedure \texttt{alloc}, calls the procedure \texttt{Fibonacci} on this list and then it calls the recursive procedure \texttt{addV} that increments by $v$ the value of each element in the list. At the end, it asserts that the obtained list is sorted. The list does not contain any longer a Fibonacci sequence, but its content is still sorted. The type system for this program is given in Figure 4.20.

VI. Program Quicksort

Figure 4.21 gives an implementation for the quicksort algorithm. The recursive procedure \texttt{quicksort} sorts the input list pointed to by the variable
void main()
{
    list *xi, *zi, *x, *z;
    int size = 6;
    alloc(xi, size);
    alloc(zi, size);
    x = NULL;
    z = NULL;
    list_share(xi, zi, x, z);
    xi = NULL;
    zi = NULL;
    z = NULL;
    return;
}

/*@requires true;
@ensures ‘z points to an acyclic singly-linked list of length size’;
*/
void alloc(list*& z, int size)
{
    list ai, *node;
    if (z == NULL) {
        while (size > 0) {
            node = NULL;
            node = new list();
            node->next = NULL;
            if (zi == NULL) {
                z = node;
                zi = node; }
            else zi->next = node;
            size--;
        }
    } return;
}

void concat(list* a, list* b)
{
    if (a != NULL) {
        ai = a; aj = a->next;
        while (aj != NULL) {
            if (aj->data == ai->data) {
                ai->next = aj->next;
                aj->next = c;
                c = ai;
                a = aj;
            } else {
                a = aj;
                aj = a->next;
                } if (foundp != found) {
                    foundp->next = found->next;
                } else if (ai != NULL) {
                    if (aip != ai) {
                        aip->next = ai->next;
                        ai->next = c;
                        c = ai;
                        a = aj;
                    } else {
                        a = aj;
                        aj = aj->next;
                    } if (foundp != found) {
                        foundp->next = found->next;
                    } else if (ai->data != aj->data) {
                        a = aj;
                        aj = aj->next;
                    } else ai->aip = aip--; }
        } return;
    } void list_share (list* ain, list* bin, list*& a, list*& b) {
        a = ain; b = bin;
        list ai=ain,*aip=a,*test;
        list bi,*bi1,*found,*foundp, *c;
        while (ai != NULL) {
            bi = b; found = NULL; foundp = bi;
            while (bi != NULL && found == NULL) {
                if (bi->data == ai->data) found = bi;
                else foundp = bi;
                bi = bi->next;
            }
            if (found != NULL) {
                if (aip != ai) {
                    aip->next = ai->next;
                    ai->next = c;
                    c = ai;
                    aip = aip->next;
                } else {
                    a = ai;
                    ai = ai->next;
                    } if (foundp != found) {
                        foundp->next = found->next;
                    } else b = bi->next;
            }
        }
    } /*@requires ‘ain points to an acyclic list or it points to NULL and
    bin points to an acyclic list or it points to NULL’;
    @ensures ‘a points to an acyclic list or it points to NULL,
    b points to an acyclic list or it points to NULL,
    any two cells from a and respectively b, which are not shared, have different
    values for the data field dt’*/
    list ai=ai,*aip=a,*test;
    list bi,*bi1,*found,*foundp, *c;
    while (ai != NULL) {
        bi = b; found = NULL; foundp = bi;
        while (bi != NULL && found == NULL) {
            if (bi->data == ai->data) found = bi;
            else foundp = bi;
            bi = bi->next;
        }
        if (found != NULL) {
            if (aip != ai) {
                aip->next = ai->next;
                ai->next = c;
                c = ai;
                aip = aip->next;
            } else {
                a = ai;
                ai = ai->next;
            } if (foundp != found) {
                foundp->next = found->next;
            } else if (ai != NULL) {
                if (aip != ai) {
                    aip->next = ai->next;
                    ai->next = c;
                    c = ai;
                    aip = aip->next;
                } else {
                    a = ai;
                    ai = ai->next;
                } if (foundp != found) {
                    foundp->next = found->next;
                } else b = bi->next;
            }
        }
    }
    return;
}

Figure 4.16: List sharing: Reordering the elements of two lists such that they share their common elements (we use statements of the form x=x->next which can be transformed in the required syntax as described in Section 4.2.1).

a. It first chooses the pivot to be the first element of the list. Then, it calls the procedure split which creates two lists left and right which contain the elements from a which are less than the pivot, respectively greater than the pivot. The procedure split can be implemented similarly to the program Dispatch, i.e. with destructive updates, or using auxiliary storage (that is, left and right contain copies of the elements of a). For the simplicity of the presentation, we consider the second implementation for split.
typedef struct _list list;

struct _list {
    list* next;
    int dt;
};

void main(){
    list *a = NULL;
    int n = 15;
    alloc(a,n);
    assert(''a points to a linked-list of length n'');
    Fibonacci(a);
    assert('' a points to a singly-linked list of length n whose elements form a Fibonacci sequence'');
    return;
}

/*@requires ''head points to an acyclic singly-linked list or head points to NULL'';
@ensures ''head points to an acyclic singly-linked list whose elements form a Fibonacci sequence or head points to NULL'';
*/
void Fibonacci(list*& head){
    list *x, *aux;
    x = head;
    int m1 = 1;
    int m2 = 0;
    while(x != NULL){
        x->dt = m1+m2;
        m1 = m2;
        m2 = x->dt;
        aux = x;
        x = NULL;
        x = aux->next;
        aux = NULL;
    }
    return;
}

Figure 4.17: Fibonacci: Initialization of a list with the Fibonacci sequence.

<table>
<thead>
<tr>
<th>$D$</th>
<th>basic types</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>record types</td>
<td>list</td>
</tr>
<tr>
<td>$F$</td>
<td>data fields</td>
<td>$\tau(dt) = list \rightarrow int$</td>
</tr>
<tr>
<td>$P$</td>
<td>pointer fields</td>
<td>$\tau(next) = list \rightarrow list$</td>
</tr>
<tr>
<td>$DVar$</td>
<td>data variables</td>
<td>$\tau(m1) = int$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\tau(m2) = int$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\tau(n) = int$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\tau(size) = int$</td>
</tr>
<tr>
<td>$PVar$</td>
<td>pointer variables</td>
<td>$\tau(x) = list$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\tau(head) = list$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\tau(a) = list$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\tau(var_alloc) = list$</td>
</tr>
</tbody>
</table>

Figure 4.18: The type system induced by the program in Figure 4.17.

Finally, the list segments pointed to by left, pivot and right are concatenated using the procedure concat and the result is the return value of the procedure quicksort. The procedure concat concatenates three singly-linked lists in the order they are given as parameters. For the moment we give only its prototype, the implementation details are not very important.

VII. Program Sorting procedure equivalence The main procedure given in Figure 4.22 creates a list h1 of length 10 and initialize its data fields with some integers. We may assume that the procedure init chooses randomly these values. Then, it creates a copy of h1, using the procedure copy. Therefore, at line 18 the assert statement states that when returning from copy, h2 points to a list equal to h1 (they have the same
4.3. EXTENDING PROGRAMS WITH PROCEDURE CALLS

```c
void main()
    list *a = NULL;
    int n = 10;
    int y = 3;
alloc(a,n);
Fibonacci(a);
addV(a,y);
assert('a points to a sorted singly-linked list of length n');
return;
} /* @requires 'head points to an acyclic singly-linked list or it points to NULL'; */
void addV(list*& head,int v){
    list *headi, *aux;
    headi = head;
    if(headi!=NULL){
        aux = headi;
        headi = NULL;
        headi = aux->next;
        aux->dt = aux->dt + v;
        aux = NULL;
        addV(headi,v);
        assert('the values of the data fields are incremented by v w.r.t. the input ones');
    }
    return;
}
```

Figure 4.19: AddV: Adding v to each element of a list.

<table>
<thead>
<tr>
<th>DT</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>JT</td>
<td>list</td>
</tr>
<tr>
<td>DF</td>
<td>τ(dt) = list → int</td>
</tr>
<tr>
<td>PF</td>
<td>τ(next) = list → list,</td>
</tr>
<tr>
<td>DVar</td>
<td>τ(y) = int</td>
</tr>
<tr>
<td></td>
<td>τ(v) = int</td>
</tr>
<tr>
<td></td>
<td>τ(n) = int</td>
</tr>
<tr>
<td>PVar</td>
<td>τ(a) = τ(x) = list</td>
</tr>
<tr>
<td></td>
<td>τ(head) = list</td>
</tr>
<tr>
<td></td>
<td>τ(headi) = list</td>
</tr>
<tr>
<td></td>
<td>τ(aux) = list</td>
</tr>
<tr>
<td></td>
<td>τ(var_alloc) = list</td>
</tr>
</tbody>
</table>

Figure 4.20: The type system induced by the program in Figure 4.19

length and they store the same values in the same order). A copy of each of these lists is sorted using a different sorting procedure; in the example, we have used quicksort and mergesort. After sorting the returned lists should be equal. This property is checked at line 29 by the assert statement.

4.3.3 Program semantics

In this thesis, we have chosen a relational semantics to describe the behavior of programs with procedures. More precisely, we consider a forward reachability collecting semantics that associates at each control location pc of a procedure \( P_\pi \), the set of all relations (mi, mc)
where $\text{mi}$ is an input memory configuration of $P_\pi$ and $\text{mc}$ is a memory configuration reachable at $pc$ from $\text{mi}$.

First, we represent a program with an *inter-procedural control flow graph*. Then, we introduce a representation for relations between memory configurations, based on the representation of a memory configuration introduced in Section 4.2. We define a transition system whose states are pairs between memory configurations ($\text{mi}, \text{mc}$) as above. The actions of the transition system (labeling the transitions) are the labels of the edges of the inter-procedural control flow graph.

The relational forward reachability collecting semantics is defined as the least fixed point of a system of equations which describes the set of reachable states of this transition system.

We assume a local semantics for procedure calls, as introduced in [Rinetzky 2005a], where the callee has only access to the part of the memory reachable from its input parameters, called local heap. If the local heap of the callee is related to the heap of the caller, these relations are difficult to model when the connection between the two parts of the heap is not done through input parameters. For example, if an object in the local heap of the callee is accessible from an object in the heap of the caller by a path that does not passes through actual parameters, due to the store-less semantics, the connection between

Figure 4.21: **Quicksort**: The *quicksort* algorithm on singly-linked lists.

```c
/* @requires '' a points to an
   acyclic singly-linked list
   or a points to NULL'';
@ensures '' res points to a sorted
   acyclic singly-linked list
   or res points to NULL'';
*/
list* quicksort(list *a) {
    list *left,*right,*pivot,*res,*start;
    int d;
    if (a == NULL || a->next == NULL)
        copy(a,res);
    else {
        d = a->dt;
        alloc(&pivot,1); // list of length 1
        pivot->dt = d;
        start = a->next;
        split(start,d,left,right);
        assert('' left points to a singly-linked list and
               the value of the data field of each
               list node reachable from left is less than d
               or left points to NULL'');
        left = quicksort(left);
        right = quicksort(right);
        res = concat(left,pivot,right);
        assert(''res points to a sorted
               acyclic singly-linked list'');
    }
    return res;
}  
```
4.3. EXTENDING PROGRAMS WITH PROCEDURE CALLS

```c
list* quicksort(list *a) {...}

/*@requires "' b points to an acyclic singly-linked list
or a points to NULL'';
@ensures "' res points to a sorted
acyclic singly-linked list
or res points to NULL'';
*/
list* mergesort(list *b); {...}

void main(){
    list *hi1, *hi2, *h1, *h2;
    alloc(h1,10);
    init(h1);
    copy(h1,h2);
    assert(''h1 and h2 point to
two equal lists'');
    copy(hi1,h1); copy(hi2,h2);
    assert(''hi1 and hi2 point to
two equal lists'');
    h1 = quicksort(h1);
    h2 = mergesort(h2);
    assert(''h1 and h2 point
to two equal lists'');
}
```

Figure 4.22: Sorting procedure equivalence: calling two sorting procedures on equal inputs (res denotes the return variable in both quicksort and mergesort).

the two objects cannot be reestablished when returning from the call (without keeping some additional information at call time). Such objects which are on the frontier of the local heap are called cutpoints. In this thesis, we have chosen to emphasize the problems that arise when verifying program specifications that describe the shape of the heap but also the data in the heap. We consider that handling cutpoints is a pure “shape problem”, orthogonal to the ones we are interested in. Therefore, in the following, we consider only cutpoint free programs [Rinetzky 2005b]. This class of programs is significantly large; it includes sorting algorithms, traversal of lists, insertion, deletion etc.

The local semantics for procedure calls introduced in [Rinetzky 2005a] keeps track of all cutpoints for each invocation of a procedure. This complicates a lot the definition of the transformation associated to the return-site node of the ICFG.

### 4.3.3.1 Inter-procedural control flow graph

We represent programs by inter-procedural control flow graphs (ICFG, for short). An ICFG is a directed graph consisting of a collection of control flow graphs, one for each procedure.

For each procedure $P_\pi$ we build a CFG $(N_{P_\pi}, E_{P_\pi})$ which contains a distinct node in $N_{P_\pi}$ for each program control point and the edges in $E_{P_\pi}$ represent the flow of control (they are labeled by program basic statements or boolean conditions). Moreover, any CFG has a node $s_{P_\pi}$ that denotes the unique entry point to the procedure, and a unique
CHAPTER 4. PROGRAMS WITH DYNAMIC DATA STRUCTURES

exit point $e_{P_\pi}$, such that $s_{P_\pi}$ has no predecessor nodes in $N_{P_\pi}$ and $e_{P_\pi}$ has no successor nodes in $N_{P_\pi}$.

A procedure call is defined by two nodes: a call node and a return-to-call node and two inter-procedural edges. The call node is connected through an inter-procedural edge to the start node of the called procedure, labeled by call $q=\pi(a_i,a_o)$, and the exit point of the called procedure is connected through an inter-procedural edge to the return-to-call node of the calling procedure, labeled by return $q=\pi(a_i,a_o)$. If the procedure has no return value then the call node and the entry node are connected by an inter-procedural edge labeled by call $\pi(a_i,a_o)$ and the edge between the exit node of $P_\pi$ and the return-to-call node is labeled by return $\pi(a_i,a_o)$.

Figure 4.23 shows the ICFG associated with the program in Figure 4.19. $\psi_1$ and $\psi_2$ represent the two assertions annotating this program ($\psi_2$ is the sorted-ness property from main line 10 and $\psi_1$ is the property from line 38 in the procedure addV).

Figure 4.23: The inter-procedural control flow graph associated with the program AddV in Figure 4.19.
4.3. EXTENDING PROGRAMS WITH PROCEDURE CALLS

4.3.3.2 Representing relations between program configurations

Let \( P \) be a program and let \( \Sigma = (T, F, \text{Var}, \tau) \) be the type system induced by \( P \). Let \( P_\pi = [\text{fi}, \text{fo}, \text{loc}, C] \) be a procedure whose formal input and output parameters are \( \text{fi} \) and respectively \( \text{fo}, \text{loc} \) denotes the set of variables used in the procedure \( (\text{fi} \cup \text{fo} \subseteq \text{loc}) \), and \( C \) is the CFG associated to \( P_\pi \). We recall that if the returned type of the procedure is not \text{void}, we distinguish \text{ret}_{P_\pi} \in \text{fo} \) as the returned formal output parameter.

A heap as in Definition 4.3.1 represents the memory state of a program without procedures. Therefore, it contains an evaluation for all program variables. For programs with procedures, we define a heap (memory configuration) for a procedure \( P_\pi \) that considers only the variables used in \( P_\pi \) (the formal parameters and the local variables).

**Definition 4.3.1** (Type system of a procedure). Let \( P \) be a program and let \( \Sigma = (T, F, \text{Var}, \tau) \) be the type system induced by \( P \). Given \( P_\pi = [\text{fi}, \text{fo}, \text{loc}, C] \) a procedure declared in \( P \), the type system associated with the procedure \( P_\pi \) is denoted \( \Sigma_\pi \) and it is obtained by restricting \( \Sigma \) to the variables of \( P_\pi \), that is \( \Sigma_\pi = (T, F, \text{Var} \cap \text{loc}, \tau) \).

**Definition 4.3.2** (Memory configuration/Heap for a procedure). A heap (memory configuration) for a procedure \( P_\pi = [\text{fi}, \text{fo}, \text{loc}, C] \) of a program \( P \) is a pair \((G, \delta)\) where \( G \) is a heap graph over the type system \( \Sigma_\pi \) associated to \( P_\pi \) and \( \delta \) is an evaluation of \( D\text{Var} \cap \text{loc} \), i.e. \( \delta : D\text{Var} \cap \text{loc} \to \mathbb{D} \).

A pair of memory configurations for a procedure \( P_\pi \) is denoted by \([H^1, H^2]\), where \( H^1 = (G^1, \delta^1) \) and \( H^2 = (G^2, \delta^2) \) are heaps over \( \Sigma_\pi \).

Using a double vocabulary for the variables of a procedure, a pair of heaps \([H^1, H^2]\) as above can be represented by a single heap, denoted by \( H^1 \oplus H^2 \). More precisely, a pair of heaps \([H^1, H^2]\) over a type system \( \Sigma_\pi = (T, F, \text{Var} \cap \text{loc}, \tau) \) can be represented by a heap over the type system

\[(T, F, ((D\text{Var} \cap \text{loc}) \cup \text{dloc}^0) \cup ((P\text{Var} \cap \text{loc}) \cup \text{ploc}^0), \tau')\],

where

- \( T \) is the set of data types and record types defined in the program,
- \( F \) is the set of all pointer fields, data fields and array fields,
- \( \text{dloc}^0 \) is a copy of the data local variables of \( P_\pi \) and it is defined by \( \text{dloc}^0 = \{d^0 | d \in \text{loc} \cap D\text{Var}\} \),
- \( \text{ploc}^0 \) is a copy of the pointer local variables of \( P_\pi \) and it is defined by \( \text{ploc}^0 = \{p^0 | p \in \text{loc} \cap P\text{Var}\} \),
- \( \tau' \) is an extension of \( \tau \) to \( \text{dloc}^0 \cup \text{ploc}^0 \) and it is defined by \( \tau(x^0) = \tau(x) \) for any \( x^0 \in \text{dloc}^0 \cup \text{ploc}^0 \) which is a copy of \( x \in \text{loc} \).

Intuitively, \( H^1 \oplus H^2 \) is defined by (1) renaming any variable \( x \in \text{loc} \) in \( H^1 \) to \( x^0 \in \text{dloc}^0 \cup \text{ploc}^0 \) and (2) taking the union of the heap obtained in the previous step and \( H^2 \) (the nodes \( x^0 \) and \( y^0 \) are shared by the two graphs). Let \( H^1 = (G^1 = (V^1, E^1, L^1, L^1_E, L^1_D), \delta^1) \) and \( H^2 = (G^2 = (V^2, E^2, L^2, L^2_E, L^2_D), \delta^2) \). Then,

\[H^1 \oplus H^2 = (G = (V, E, L, L_E, L_D), \delta), \quad (4.3.1)\]

where:

77
CHAPTER 4. PROGRAMS WITH DYNAMIC DATA STRUCTURES

- \( V = V_1 \cup V_2 \), \( E = E_1 \cup E_2 \),
- for any \( p^0 \in \text{ploc}^0 \), \( L(p^0) = L^1(p) \) and for any \( \text{ploc} \cap PVar, L(p) = L^2(p) \),
- for any \((v_1, v_2) \in E^i \), \( L_E(v_1, v_2) = L^i_E(v_1, v_2) \), for \( 1 \leq i \leq 2 \),
- for any \( v \in V^1 \), \( L_D(v) = L^i_D(v) \), for all \( 1 \leq i \leq 2 \),
- for any \( d^0 \in \text{dloc}^0 \), \( \delta(d^0) = \delta^1(d) \) and for any \( d \in \text{loc} \cap DVar, \delta(d) = \delta^2(d) \).

**Definition 4.3.3 (Pair of memory configurations).** A pair of memory configurations a procedure \( P_\pi \) is denoted by \( [H^1, H^2] \), where \( H^1 = (G^1, \delta^1) \) and \( H^2 = (G^2, \delta^2) \) are heaps over \( \Sigma_\pi \). The set of all these pairs of configurations for some procedure \( P_\pi \) is denoted by \( \mathcal{R}(\Sigma_\pi) \), where \( \Sigma_\pi \) is the type system of the procedure \( P_\pi \). The set of all program’s pairs of memory configurations is denoted by \( \mathcal{R}(\Sigma) \), where \( \Sigma \) is the program’s type system. A pair of memory configurations is equivalently represented by the heap graph denoted \( H^1 \oplus H^2 \).

### 4.3.3.3 Concrete transformers for program statements

In the following, we give the semantics of a program in terms of transition systems. We will denote by \((N, E)\) the ICFG associated with a program \( P_\Pi \) and by \((N_{P_\pi}, E_{P_\pi})\) the intra-procedural CFG corresponding to each procedure \( P_\pi \) declared in \( P_\Pi \).

A program relation is a pair \((n, [H^1, H^2])\) where \( n \in N_{P_\pi} \) is a node of the control flow graph of some procedure \( P_\pi \) and \([H^1, H^2]\) is a a pair of two heaps (memory configurations) over \( \Sigma_\pi \). \( H^1 \) represents an input configuration of \( P_\pi \) and \( H^2 \) is a configuration reachable from \( H^1 \) at the control point \( n \).

The program defines a transition system \((S, Act, \rightarrow, I)\), where (1) the set of states \( S = [N_{P_\Pi} \rightarrow \mathcal{H}(\Sigma) \times \mathcal{H}(\Sigma)] \) is the set of functions that associate to every node in the ICFG \((N_{P_\Pi} \) denotes the set of nodes of the ICFG) a pair of heaps from \( \mathcal{H}(\Sigma) \), (2) the set of actions \( Act \) contains the edge labels of the ICFG, and (3) \( I \) is some set of initial states. We consider an order on the set of nodes in the ICFG and in the following we represent the set states by a tuples:

\[
S = \left\{ (n_1, [H_1^1, H_1^2]), \ldots, (n_p, [H_p^1, H_p^2]) \right\} \quad \text{for all } 1 \leq j \leq p, n_j \in N_{P_\Pi},
\]

\[
[\Sigma_i, \text{the type system induced by } P_\pi],
\]

The transition relation \( \rightarrow \subseteq S \times Act \times S \) is defined as follows. For any action \( a \) except for procedure returns, we define a partial function \( U[a] : \mathcal{R}(\Sigma) \rightarrow \mathcal{R}(\Sigma) \) and, for any action \( a \) which is a procedure return, we define a partial function \( U[a] : \mathcal{R}(\Sigma) \times \mathcal{R}(\Sigma) \rightarrow \mathcal{R}(\Sigma) \). The functions \( U[a] \) and \( U[a] \) are called **concrete transformers**.

Let \( a \in Act \) and

\[
s = (n_1, [H_1^1, H_1^2]), \ldots, (n_p, [H_p^1, H_p^2]),
\]

\[
s' = (n_1, [H_1'^1, H_1'^2]), \ldots, (n_p, [H_p'^1, H_p'^2]).
\]

We have that:
4.3. EXTENDING PROGRAMS WITH PROCEDURE CALLS

1. if a is a statement different from procedure calls and returns, i.e., \( n_k \xrightarrow{a} n_l \in E_{P_n} \), for some \( n_k, n_l \in N_{P_n} \), then

   \[
   s \xrightarrow{a} s' \quad \text{iff} \quad U[a] \text{ is defined in } [H_k^1, H_k^2],
   \]
   \[
   [H_i^1, H_i^2] = U[a]([H_k^1, H_k^2]), \quad \text{and}
   \]
   \[
   [H_s^1, H_s^2] = [H_k^1, H_k^2], \text{ for all } 1 \leq s \neq l \leq p.
   \]

2. if \( a := \text{call } P_{\pi_j}(ai, ao) \) or \( a := \text{call } q = P_{\pi_j}(ai, ao) \) and \( n_k \xrightarrow{a} n_l \in E \), for some \( n_k \in N_{P_n}, n_l \in N_{P_{\pi_j}} \), then

   \[
   s \xrightarrow{a} s' \quad \text{iff} \quad U[a] \text{ is defined in } [H_k^1, H_k^2],
   \]
   \[
   [H_i^1, H_i^2] = U[a]([H_k^1, H_k^2]), \quad \text{and}
   \]
   \[
   [H_s^1, H_s^2] = [H_k^1, H_k^2], \text{ for all } 1 \leq s \neq l \leq p.
   \]

3. if \( a := \text{return } P_{\pi_j}(ai, ao) \) or \( a := \text{return } q = P_{\pi_j}(ai, ao) \), and \( n_k \xrightarrow{a} n_l \in E \), for some \( n_k \in N_{P_{\pi_j}}, n_l \in N_{P_n} \), then

   \[
   s \xrightarrow{a} s' \quad \text{iff} \quad n_c \in N_{P_{\pi_j}} \text{ is the call node associated with}
   \]
   \[
   \text{the return-to-call node } n_l,
   \]
   \[
   U[a] \text{ is defined in } ([H_k^1, H_k^2], [H_{\pi_j}^1, H_{\pi_j}^2]),
   \]
   \[
   [H_i^1, H_i^2] = U[a]([H_{\pi_j}^1, H_{\pi_j}^2], [H_k^1, H_k^2]),
   \]
   \[
   [H_s^1, H_s^2] = [H_k^1, H_k^2], \text{ for all } 1 \leq s \neq l \leq p,
   \]

Concrete transformers associated to the labels of an intra-procedural CFG

For any \( a \in \text{Act} \) which is an assignment, a boolean condition, or an \text{assert/assume} statement, the definition of the concrete transformers \( U[a] \) is based on the concrete transformers \( F[a] \) given in Section 4.2.3

The transformer \( U[a] \) modifies only the second component of the input pair of heaps, because program statements do not have access to the variables with superscript zero. This is due to the fact that the relational semantics we give considers pairs between the input heap of a procedure and the heap reachable from that input at some procedure control point. Therefore, as long as the control stays inside a procedure (no inter-procedural edges are taken) the input heap remains unchanged. Therefore, \( U[a] \) modifies the second component of the input pair of heaps by applying the transformer \( F[a] \). If we consider a semantics based on garbage collection then, we apply the transformer \( F[a] \) composed with \text{elim}_gb \ (the transformer that removes the memory cells not reachable from program pointer variables).

For any \text{assert/assume} statement, \( U[\text{assert } \varphi](H^1, H^2) \) and \( U[\text{assume } \varphi](H^1, H^2) \) are defined as in Figure 4.24.
U[assert ϕ](H₁, H₂) = \begin{cases} (H₁, H₂), & \text{if } H₁ ⊕ H₂ ∈ ϕ, \\ (G_{err}, δ_{err}), & \text{otherwise}. \end{cases}

U[assume ϕ](H₁, H₂) = \begin{cases} (H₁, H₂), & \text{if } H₁ ⊕ H₂ ∈ ϕ, \\ \text{undefined}, & \text{otherwise}. \end{cases}

Figure 4.24: Concrete transformers for assert and assume statements

4.3.3.4 Concrete transformers for inter-procedural edges of the ICFG

The local semantics for procedure calls is based on the idea that procedures should be analyzed by ignoring parts of the heap that are not relevant to the procedure. In this semantics, to a called procedure is passed only a part of the heap, the one reachable from the actual parameters of the call. It has been introduced in [Rinetzky 2005a, Rinetzky 2005b] in order to increase the modularity of the analysis therefore leading to better scalability results.

The downside of this approach is that, due to destructive updates, the values of the pointer fields corresponding to pointer variables not local to the call cannot always be defined in terms of the state prior to the call. We’ll explain it using an example:

Example 4.3.1. Let \( P_π \) be a procedure that deletes all the elements of the input list (\( P_π \) receives as input a pointer to an object of type list, defined as in Figure 4.17) whose data field equals 3. Suppose that \( H_c = [G_c, \emptyset] \) represent a context for a call to \( P_π(z) \), where \( G_c \) is the heap graph in Figure 4.25(a) and \( \emptyset \) denotes the empty function. Also, let \( [H_e, H_x] \) be the relation between the input configuration \( H_e \) of \( P_π \) (corresponding to \( H_c \)) and the memory configuration \( H_x \) reached from \( H_e \) at the return control point of \( P_π \). \( H_e = (G_e, \emptyset) \) and \( H_x = (G_x, \emptyset) \), where \( G_e \) and \( G_x \) are given in Figure 4.25(b). When the control returns to the caller procedure the value of the pointer field next of the record value pointed to by \( x \) cannot be defined because we use a store-less semantics where memory addresses are abstracted away. The context of the call states that \( x \rightarrow \text{next} \) points to the second element in the list pointed to by \( z \) which contains the integer data 5. When returning from the call \( P_π(z) \), it is unsound to define \( x \rightarrow \text{next} \) as the second or as the third element in the output list (see Figure 4.25(d)). This is because we have lost the relation between the addresses of the cells in the output list and the addresses of the cells in the input list. If we consider a store-based semantics (where each vertex in the heap graph carries an unique id denoting the address of the corresponding memory block) then, we preserve the equality between the addresses of the cells in the output list and the addresses of the cells in the input list. The memory configuration that we would obtain in this case is given in Figure 4.25(c).

The vertices of a heap graph, except for the ones labeled by actual parameters, which separate the part of the heap that an invoked procedure can access from the rest of the heap are called cutpoints.

Definition 4.3.4 (Cutpoints). A cutpoint for an invocation of a procedure \( P_π, \) in the heap \((G, δ)\) is a vertex of \( G \) which is (1) reachable from a vertex labeled by an actual parameter and (2) reachable from a vertex labeled by some pointer variable, which is not an actual parameter, through a path that does not contain any vertices labeled by actual parameters.
4.3. EXTENDING PROGRAMS WITH PROCEDURE CALLS

4.3.1 Remark. The number of cutpoints for programs with recursive procedures is potentially unbounded.

Definition 4.3.5 (Cutpoint free programs). A cutpoint free procedure invocation is a procedure invocation which does not have cutpoints. A program execution in which all invocations are cutpoint free is called a cutpoint free execution, and a program in which all executions are cutpoint free is called a cutpoint free program.

Verifying cutpoint freedom: Let $H^c = (G^c = (V^c, E^c, L^c, L_E^c, L_D^c), \delta^c)$ be a heap for a procedure $P_{\pi_i}$ and let $P_{\pi_j}(ai, ao)$ be a procedure call.

We define a predicate $cpf(\text{api}, G^c)$ which holds if the invocation of $P_{\pi_j}(ai, ao)$ (under the assumption that the output parameters point to null) in the context $G^c$ does not have cutpoints. Thus, let $V^{api}$ denote the set of vertices in $G^c$ labeled by actual input parameters, i.e., $V^{api} = \bigcup_{api \in \text{api}} L^{c}(api)$. Also, let $V^l$ denote the set of vertices reachable from $V^{api}$, i.e., $V^l = \{v \in V^c \mid$ there exists a path from some vertex $v_p \in V^{api}$ to $v\}$.

Then, $cpf(\text{api}, G^c) = true$ iff for any $v \in V^c \setminus (V^{api} \cup V^l \cup \{\# , \#'\})$ and $v' \in V^l \setminus V^{api}$, any path from $v$ to $v'$ passes through a node in $V^{api}$.

Procedure call: For any $a ::= \text{call } P_{\pi_j}(ai, ao)$ or $a ::= \text{call } q = P_{\pi_j}(ai, ao)$, $U[a]$ computes the pair of heaps associated with the entry node of the CFG corresponding to $P_{\pi_j}$. It is defined only if the procedure call does not have cutpoints. The output of $U[a]$ is a pair between two identical heaps that denote the input configuration of the procedure $P_{\pi_j}$.
Intuitively, for any pair of heaps \([H, H^c] = (G^c, \delta^c)\), we define

\[ U[a][[H, H^c]] = [H^e = (G^e, \delta^e), H^e] \]

if \(cpf(\text{api}, G^c)\) holds, where (1) \(G^e\) is the sub-graph of \(G^c\) that contains all vertices reachable from the actual input parameters of the call, where actual parameters are replaced by formal parameters, and (2) \(\delta^e\) is obtained from \(\delta^c\) by copying the values of the actual input data parameters.

The local heap graph corresponding to this call is computed by a function

\[
\text{local}(G^e, P_{\pi_j}(\text{ai}, \text{ao})) = (V^l, E^l, L^l, E^l_D), \tag{4.3.2}
\]

where \(E^l = E^c \cap (V^l \times V^l), L^l = L^c|_{V^l}, L^l_D = L^c_D|_{V^l}\), and \(L^l = L^c|_{V^l}\). Then, the heap graph \(G^e = (V^e, E^e, L^e, E^e_D)\) is defined by:

\[
\begin{align*}
& \bullet V^e = V^l, E^e = E^l, L^e = L^l, \quad E^e_D = L^l_D, \\
& \bullet L^e \text{ is obtained from } L^l \text{ such that for any actual input parameter } x \in \text{api} \text{ corresponding to a formal input parameter } y \in \text{fpi}, L^e(y) = L^c(x). \\
\end{align*}
\]

Moreover, for every \(z \in (\text{loc} \setminus \text{fpi}) \cap \text{PVar}, L^e(z) = \emptyset\).

Then, for any actual input data parameter \(d_x \in \text{adi} \) corresponding to a formal input parameter \(d_x \in \text{fdi}\), \(\delta^e(d_x) = \delta^c(d_x)\). We assume that all local data variables are initialized with some default value \(d\), corresponding to the type of the variable. Thus, \(\delta^e(l) = d\), for all \(l \in (\text{loc} \setminus \text{fdi}) \cap \text{DVar}\).

**Procedure return:** For any label \(a ::= \text{return } P_{\pi_j}(\text{ai}, \text{ao})\) or \(a ::= \text{return } q = P_{\pi_j}(\text{ai}, \text{ao})\) of some edge in the ICFG between two nodes \(n_k\) and \(n_l\) (\(n_l\) belongs to the CFG of some procedure \(P_{\pi_j}\)), \(\overline{U}[a]\) takes as input two pairs of heaps: \([H, H^c]\) which is associated with the call-node corresponding to \(n_l\) and \([H^e, H^f]\) which is a relation between some input configuration \(H^e\) of \(P_{\pi_j}\) and the corresponding output memory configuration \(H^f\). Then, it computes the pair of heaps \([H, H^c]\) associated with the ICFG return-to-call node \(n_l\).

\(\overline{U}[a]\) is defined only if the procedure call does not have cutpoints and if \(H^e\), the input configuration of the procedure \(P_{\pi_j}\), and \(H^f\), the context of the call, agree on the values of the actual input parameters.

For any pairs of heaps \([H, H^c] = (G^c, \delta^c)\) and \([H^e = (G^e, \delta^e), H^f = (G^f, \delta^f)]\), we define

\[ \overline{U}[a][[H, H^c], [H^e, H^f]] = [H, H^c] = (G^c, \delta^c) \text{ if } H^c \sim_{\text{ai}, \text{ao}} H^e \text{ and } cpf(\text{api}, G^c)\]

where \(H^c \sim_{\text{ai}, \text{ao}} H^e\) holds iff \(U[a'][[H, H^c]] = [H^e, H^c]\) when \(a' ::= \text{call } P_{\pi_j}(\text{ai}, \text{ao})\) and \(H^r\) is defined as follows:

1. we build \((G^{\text{out}}, \delta^{\text{out}})\) from \((G^e, \delta^e)\) by (1) restricting \(\delta^e\) to the variables of the procedure \(P_{\pi_j}\) which are input or output parameters \(\text{DVar} \cap (\text{fi} \cup \text{fo})\) and (2) taking the sub-graph of \(G^e\) that contains all the vertices labeled with variables in \(\text{fi} \cup \text{fo}\) together with all the vertices reachable from these labeled vertices; all vertices of \(G^{\text{out}}\) are labeled only by pointer variables in \(\text{fi} \cup \text{fo}\) (we remove all other pointer variables from the node labeling of \(G^{\text{out}}\));

2. we take the union, denoted \(G^{c,\text{out}}\), between the call graph \(G^c\) and the graph \(G^{\text{out}}\) obtained from the output configuration of \(P_{\pi_j}\). Let \(G^{c,\text{out}} = (V^{c,\text{out}}, E^{c,\text{out}}, L^{e,\text{out}}, L^{c,\text{out}}_E, L^{c,\text{out}}_D)\).
3. finally, $G^{\tau}$ is obtained from $G^{c\text{-}out}$ by applying the following transformations:

(a) we disconnect the vertices pointed to by actual parameters from their successors, that is, we remove every edge $e = (v_{ai}, v) \in E^{c\text{-}out}$ with $L^{c\text{-}out}(ai) = v_{ai}$, for some $ai \in \text{api}$ an actual input parameter;

(b) we connect the vertices pointed to by actual input parameters to the successors of the vertices pointed to by the corresponding formal input parameters. Formally, for every edge $e_z = (v_f, v_x) \in E^{c\text{-}out} \cap E^{out}$, such that $v_f$ is labeled by some formal input parameter $fi \in \text{fpi}$, we add the edge $e = (v, v_x)$ with $L^e(f) = L^{c\text{-}out}(e_z)$ where $v$ is the vertex in $G^{c\text{-}out} \cap G^c$ labeled by the actual input parameter $ai \in \text{api}$ which is bound to the formal input parameter $fi$;

(c) we move the labels representing actual output parameters from $\text{ao}$, to the vertices labeled by formal output parameters. Formally, for every $fo \in \text{fpo}$ let $v = L^{c\text{-}out}(fo)$ then, $L^e(\text{ao}) = v$ where $\text{ao}$ is the actual output parameter bound to $fo$;

(d) we redefine the data labeling of vertices pointed to by actual input parameters according to the memory configuration outputted by $P_{\pi_j}$. That is, for every $ai \in \text{api}$ let $v = L^{c\text{-}out}(ai)$, we redefine $L_D^e(v)$ by $L_D^e(\text{ao}) = L^{c\text{-}out}(\text{ao})$ where $L^{c\text{-}out}(\text{fi}) = v_f$ where $\text{fi}$ is the formal input parameter bounded to $ai$;

(e) if $a := \text{return } q = P_{\pi_j}(ai, \text{ao})$ then
   
   - if the return value is a reference to a record type then $\text{ret}_{P_{\pi_j}} \in \text{fpo}$ is the returned variable and $L^e(\text{ret}_{P_{\pi_j}}) = L^{c\text{-}out}(\text{ret}_{P_{\pi_j}})$.
   
   - otherwise, if the return value is of basic type, $\text{ret}_{P_{\pi_j}} \in \text{fdo}$, then $\delta^e(d) = \delta(\text{ret}_{P_{\pi_j}})$;

(f) finally we eliminate from $L^{c\text{-}out}$ all the vertex labels that are formal parameters and in case of a garbage free semantics we apply $\text{elim}_{\text{gb}}$ on the resulting graph;

4. $\delta^e$ equals $\delta^c$ on the actual input parameters, $\text{adi}$ (because we assumed that parameters are passed by value) and $\delta^e(\text{ao}) = \delta^e(\text{fo})$ where $\text{ao} \in \text{ado}$ is the actual parameter bounded to the formal parameter $fo \in \text{fdo}$.

4.3.3.4.1 Reachability collecting semantics For the transition system defined above, the operator $\text{post} : 2^S \to 2^S$, defined in Chapter 3 gives the set of immediate successors of a set of states.

Given a set of initial program configurations $I$, the reachability collecting semantics of a program $P_{\Pi}$ is defined as the set of reachable states of the transition system defined by $P_{\Pi}$, i.e. $\text{post}^*(I)$ which is the least fixed point of $\text{post}$ larger than $I$, denoted by $\text{lfp}_I(\text{post})$.

We say that two pairs of memory configurations are isomorphic if the corresponding heaps are isomorphic. This isomorphism relation is denoted also by $\sim$.

Definition 4.3.6 (Concrete lattice). The concrete lattice of pairs of memory configurations for the program $P_{\Pi}$, denoted by $C^e(\Sigma)$, is defined by:

$$C^e(\Sigma) = (\mathcal{P}(\mathcal{R}(\Sigma)/\sim), \subseteq / \sim, \cup, \cap, \emptyset, \mathcal{R}(\Sigma)/\sim)$$

where any subset of $\mathcal{R}(\Sigma)/\sim$ is the quotient set with respect to $\sim$, and $\subseteq$ is the order relation defined w.r.t. $\sim$. More precisely, for any $RS_1, RS_2 \in \mathcal{P}(\mathcal{R}(\Sigma)/\sim)$, $RS_1 \subseteq / \sim RS_2$ iff for every $[H_1, H_2]_1 \in RS_1$ there is $[H_1, H_2]_2 \in RS_2$ such that $[H_1, H_2]_1 \sim [H_1, H_2]_2$.
Let \( n_i \) be a node in the ICFG of \( P_\Pi \), \((N_{P\Pi}, E_{P\Pi})\), that belongs to a procedure \( P_\pi \). The set of all relations \((m_i, m_c)\), where \( m_i \) is an input memory configuration of \( P_\pi \) and \( m_c \) is a memory configuration reachable at \( n_i \) from \( m_i \), is obtained by extracting from each state \((n_1, [H^1_1, H^2_1]), \ldots, (n_p, [H^1_p, H^2_p])\) in \( \text{post}^*(I) \) the pair of heaps corresponding to \( n_i \), i.e. \([H^1_i, H^2_i]\). The computation of \( \text{post}^* \) is done using the chaotic iteration.

We extend \( C^r(\Sigma) \) to the concrete lattice of tuples of pairs of memory configurations by:

\[
C^r_{pc}(\Sigma) = (\mathcal{P}(\mathcal{R}(\Sigma)/\sim))^P, \subseteq / \sim, \cup, \cap, \emptyset, (\mathcal{R}(\Sigma)/\sim)^P).
\]

Consequently, the semantics is equivalently defined (modulo the equivalence relation \(~\)) by the least fixed point of the following system of equations in the lattice \( C^r_{pc}(\Sigma) \):

\[
igwedge_{i=1}^{p=|N|} \text{post}_i(X_{n_1}, \ldots, X_{n_p}) = X_i
\]

where

\[
\text{post}_1(X_{n_1}, \ldots, X_{n_p}) = I
\]

and for all \( 2 \leq i \leq p \),

\[
\text{post}_i(X_{n_1}, \ldots, X_{n_p}) = \bigcup_{n_j, n_i \in N_{P\Pi}} U[a](X_{n_j}), \quad \text{if}\ n_i \text{ is not a return-to-call node},
\]

\[
\text{post}_i(X_{n_1}, \ldots, X_{n_p}) = \bigcup_{n_j, n_i \in E_{P\Pi}} U[a](X_{n_j}) \quad \text{if}\ n_i \text{ is a return node and}\ n_c \text{ is the corresponding call node}
\]

### 4.4 A general logic \( \text{gCSL} \) for reasoning over program configurations

The language of program annotations (used to express the specification and the \text{assert/assume} statements) is a key aspect in reasoning about programs. We consider annotations which are first-order formulas, whose free variables include only the program variables of the procedure where the annotation occurs.

An annotation describes a set of (pairs of) memory configurations, which are pairs between a graph and a valuation for the program data variables. The language of program annotations we consider is a multi-sorted first order logic on graphs, called \textit{Generalized Composite Structures Logic} (\( \text{gCSL} \), for short) We define \( \text{gCSL} \) the language of program annotations. The considered programs manipulate dynamically allocated data structures, therefore \( \text{gCSL} \) describes (1) the \textit{structure} of the allocated memory, also called \textit{shape} (e.g. the memory contains singly-linked lists, doubly-linked lists, arrays, etc.); (2) the \textit{size} of the allocated memory, i.e. the number of allocated blocks of record type (the size of a block of record type is considered constant) (3) the relations between the values of the data fields and between the values of the data fields and the basic type variables, called \textit{data relations} in the following (e.g. equality between values of data fields and variables, or sortedness properties).

We first define the syntax and the semantics of \( \text{gCSL} \), and some shorthands useful for the readability of the formulas. Then, we give several examples to show the expressiveness of \( \text{gCSL} \).
4.4. A GENERAL LOGIC gCSL FOR REASONING OVER PROGRAM CONFIGURATIONS

4.4.1 Syntax of gCSL

gCSL is used to characterize memory configurations and, therefore, there are several symbols that are shared between the program syntax and the logic syntax, e.g. the names of the variables and the name of the pointer fields. Moreover, the type system induced by the program is inherited by the logic. More precisely, the logic is defined over

- a set of types or sorts \( \mathcal{T} = \mathbb{Z} \cup \mathbb{D} \cup \mathcal{RT} \), where \( \mathbb{D} \) denotes the basic data type and \( \mathcal{RT} = \{Rt_1, \ldots, Rt_n\} \) is a set of record types;
- a set of program variables, denoted \( \mathit{Var} = \mathit{PVar} \cup \mathit{DVar} \), where \( \mathit{PVar} \) are pointer variables and \( \mathit{DVar} \) are variables of basic type;
- a set of field symbols, denoted \( \mathcal{F} = \mathcal{AF} \cup \mathcal{DF} \cup \mathcal{PF} \), where \( \mathcal{PF} \) (\( \mathcal{DF} \)) is the set of pointer (data) fields used in the declaration of the data structures, and \( \mathcal{AF} \) is the set of array fields, one for each type in \( \mathcal{RT} \); recall that \( \mathcal{PF}^* \) is the set of all pointer fields, including the inverse pointer \( \overline{f} \), for every \( f \in \mathcal{PF} \setminus \mathcal{AF} \). For simplicity, we consider only data fields of type \( Rt \rightarrow \mathbb{D} \), for some \( Rt \in \mathcal{RT} \).
- a type system \( \Sigma = (\mathcal{T}, \mathcal{F}, \mathit{Var}, \tau) \).

Furthermore, to express the transformations (or the boolean conditions) performed by the program on the values of basic type we consider a first order logic over \( \mathbb{D} \), \( \mathit{FO}(\mathbb{D}, \mathit{O}, \mathit{P}) \), where \( \mathit{O} \) is a set of operations and \( \mathit{P} \) is a set of predicates over terms interpreted as values in \( \mathbb{D} \).

**Syntax** The logic gCSL is interpreted over heaps (over the type system \( \Sigma \)) which are defined as in Section 4.2.3. Roughly, to describe them, gCSL uses:

1. reachability predicates (or ordering predicates) between vertices in the heap graph following some pointer fields; these predicates allow to describe the shape of the allocated memory;
2. linear constraints on the lengths of the paths in the heap graph and the indexes of the arrays, in order to express the size constraints, and
3. predicates from the underlying logic \( \mathit{FO}(\mathbb{D}, \mathit{O}, \mathit{P}) \) on the values of the data fields and basic type variables, in order to represent the data relations.

The syntax of gCSL formulas is defined in Figure 4.26. Reachability predicates are built over location variables that denote vertices in the graph. The set of location variables is denoted by \( \mathit{Loc} \). The labeling of the vertices in the graph with pointer variables is constrained by predicates of the form \( \ell(x) \), where \( \ell \in \mathit{PVar} \), and \( x \) is a location variable. Index variables are used to express constraints on the size of the data structures through reachability predicates and in terms of the form \( a[\text{ind}] \), where \( a \) is an array variable, designating as usual the \( \text{ind} \)-th element of the array denoted by \( a \). Index variables can be used in linear arithmetical constraints, like \( \mathit{Exp} \). The set of all index variables is denoted by \( \mathit{Ind} \) (this set includes the set of program variables of type \( \mathbb{Z} \)). The predicates of the underlying logic \( \mathit{P} \) are used for the data constraints. These predicates are applied to data variables and data terms. The data terms, \( t_{dt} \), are obtained using the operation symbols from \( \mathit{O} \) over data variables in \( \mathit{Data} \) (that includes the program data variables in \( \mathit{DVar} \))
of type $\mathcal{D}$) and terms of the form $dt(x)$, that denote the value of the data field $dt$ in the object denoted by $x$.

As usual, conjunction ($\land$), implication ($\Rightarrow$), and universal quantification ($\forall$) can be defined in terms of $\lnot$, $\lor$, and $\exists$. The logic $\text{gCSL}$ allows quantification over all kinds of variables. We assume w.l.o.g. that each variable is quantified at most once. We denoted by $\sharp$ the vertex labeled by $\text{nu1}$.

$x, x' \in \text{Loc}$ location variables $a \in \text{Arr}$ array variable
d $\in \text{Data}$ data variable of type $\mathcal{D}$ $i \in \text{Ind}$ index variable
$\ell \in \text{PVar}$ label $ct \in \mathbb{N}$ integer constant
$p \in \mathcal{P}$ predicate over $\mathcal{D}$ $\text{ind} \in \text{Ind} \cup \mathbb{N}$
o $\in \mathcal{O}$ operator over $\mathcal{D}$ $dt \in \mathcal{DF}$ data field

Location terms: $t \ ::= \ x \mid a[\text{ind}]$

Data terms: $t_d \ ::= \ d \mid o(t_d, \ldots, t_d) \mid dt(x)$

Index expressions: $\text{Exp} \ ::= \ i \mid ct \mid \text{Exp} + \text{Exp} \mid \text{Exp} - \text{Exp}$

Formulas: $\phi \ ::= \ \ell(x) \mid x = t \mid
x^A, B, \text{ind} \rightarrow x' \mid x^A, B \rightarrow x' \mid x \rightarrow x'$
$p(t_d, \ldots, t_d) \mid \text{Exp} < ct \mid \text{Exp} = ct \mid$
$\exists x. \phi \mid \exists a. \phi \mid \exists d. \phi \mid \exists i. \phi \mid \lnot \phi \mid \phi \lor \phi$, where $A \neq \emptyset$, $A \subseteq (\text{PF}_r \setminus \mathcal{AF}) \cup \text{PF}_l \setminus \mathcal{AF}$, $B \subseteq \text{PVar}$, and $H \subseteq (\text{PF} \setminus \text{PF}_r) \cup \text{PF} \setminus \text{PF}_l$.

Figure 4.26: Syntax of $\text{gCSL}$ over $\Sigma = (T, F, \text{Var}, \tau)$.

We distinguish the array terms which are location term of the form $a[\text{ind}]$, where $a \in \text{Arr}$ is an array variable and $\text{ind} \in \text{Ind}$ is an index variable. Notice that these terms appear only in sub-formulas of the form $x = a[\text{ind}]$, where $x \in \text{Loc}$ is a location variable.

**Shorthands** We introduce several shorthands concerning arithmetical expressions:

- we allow terms of the form $a[\text{Exp}]$ to denote the element of the array $a$ on the position obtained by evaluating the expression $\text{Exp}$. A formula $\phi$ containing $a[\text{Exp}]$ is equivalent to the formula $\text{Exp} - i = 0 \land \phi(a[i]/a[\text{Exp}])$ where $i$ is a fresh existentially quantified index variable;

- for arrays of objects containing a data field $dt$, one can use $dt(a[\text{Exp}])$ to refer to the data value stored by $a[\text{Exp}]$ in the field $dt$. A formula $\phi$ containing $dt(a[\text{Exp}])$ is equivalent to the formula $a[\text{Exp}] = x \land \phi(dt(x)/dt(a[\text{Exp}]))$, where $x$ is a fresh existentially quantified location variable;

- formulas of the form $a[i]^A, B, \text{ind} \rightarrow c[i']$ are abbreviations of $x_1 = a[i] \land x_2 = c[i'] \land x_1^A, B, \text{ind} \rightarrow x_2$ for some fresh existentially quantified location variables $x_1$ and $x_2$;

- the comparison between two arithmetical expressions can be written as $\text{Exp} < \text{Exp}'$ ($\text{Exp} = \text{Exp}'$) instead of $\text{Exp} - \text{Exp}' < 0$ ($\text{Exp} - \text{Exp}' = 0$).
4.4. A GENERAL LOGIC GCSL FOR REASONING OVER PROGRAM CONFIGURATIONS

The type system of \textit{g}CSL The typing function \(\tau\) of \(\Sigma\) is extended to the variables of \textit{g}CSL formulas. We denote by \(\Sigma^L\) the type system of \textit{g}CSL, where \(\Sigma^L = (\mathcal{T}, \mathcal{PF}^*, \mathcal{Var}, \tau^L)\) and \(\tau^L : \mathcal{(DVar} \cup \mathcal{Data}) \cup \mathcal{Ind} \cup \mathcal{(PVar} \cup \mathcal{Loc}) \rightarrow \mathcal{T}\).

The type of a location variable in \(\text{Loc}\) is always a record type. The type of the index variables in \(\text{Ind}\) is \(\mathbb{Z}\), and the type of the data variables in \(\text{Data}\) is \(\mathbb{D}\). We require that formulas are type consistent. Thus,

- for every formula of the form \(x \xrightarrow{H} x'\), where \(H \subseteq \mathcal{PF} \setminus (\mathcal{PF}_r \cup \mathcal{PF}_f)\) is a set of non-recursive pointer fields, we must have \(\tau(x) = \mathcal{Rt}, \tau(x') = \mathcal{Rt}', \mathcal{Rt} \neq \mathcal{Rt}',\)
  \(\tau(h) = \mathcal{Rt} \rightarrow \mathcal{Rt}',\) for any \(h \in H \cap \mathcal{PF}\), and \(\tau(h) = \mathcal{Rt}' \rightarrow \mathcal{Rt},\) for any \(\mathcal{Rt} \in H\), and

- for every formula of the form \(x \xrightarrow{A,B,\text{ind}} x'\) (or \(x \xrightarrow{A,B} x'\)), with \(A \subseteq (\mathcal{PF}_r \setminus \mathcal{AF}) \cup \mathcal{PF}_f \setminus \mathcal{AF}\) we require that \(\tau(x) = \tau(x') = \tau(z)\) for every \(z \in B\), \(\tau(f) = \tau(x) \rightarrow \tau(x),\) for every \(f \in A\), and \(\tau(f) = \tau(x) \rightarrow \tau(x),\) for every \(f \in A\).

That is, \(x \xrightarrow{A,B,\text{ind}} x'\) is true if there are \textit{ind}-th allocated memory blocks of the same type as \(x\) in the \(A\)-path between the vertices denoted by \(x\) and \(x'\).

A data term \(dt(x)\) is well typed if \(\tau(x) = \mathcal{Rt}\) and \(\tau(dt) = \mathcal{Rt} \rightarrow \mathbb{D}\). The data constraints are formed using predicates from \(\mathcal{P}\) over well typed data terms.

4.4.2 Semantics of \textit{g}CSL A \textit{g}CSL formula over a type system \(\Sigma^L\) is interpreted over a heap \(H = (G, \delta)\), where \(G = (V, E, L, L_E, L_D)\), over the type system \(\Sigma\) w.r.t type preserving valuations of free variables.

Let \(\mu : \text{Loc} \rightarrow \mathbb{V}, \nu : \text{Ind} \rightarrow \mathbb{N},\) and \(\kappa : \text{Data} \rightarrow \mathbb{D}\) be a valuation of the location, index, and data variables, respectively.

Location variables are interpreted into vertices of the heaps graphs. Array variables are mapped to array vertices which have no incoming edge labeled by an array field, i.e., they are first elements of (possibly singleton) arrays in \(G\). For any array vertex \(w\), let \(\text{Dom}(w)\) (the domain of \(w\)) be the length of the maximal path starting in \(w\) where all the edges are labeled by some array field \(a \in \mathcal{AF}\) (this array field should be unique by the well-formedness of \(G\)). Let \(\mathcal{AV}\) denote the set of array vertices in \(V\) and \(\theta : \mathcal{Arr} \rightarrow \mathcal{AV}\) be the valuation of the array variables.

Location terms are interpreted either in a vertex in \(G\) or in \(\bot\) as follows:

- any location variable \(x\) is interpreted by \(\mu\) into a vertex of \(G\): \(\langle x \rangle_{H,\mu,\nu,\kappa} = \mu(x),\)
  and

- for any \(a \in \mathcal{Arr}\) and \(i \in \mathcal{Ind}\), if \(\nu(i) > \text{Dom}(\theta(a))\) then \(\langle a[i] \rangle_{H,\mu,\theta,\nu,\kappa} = \bot,\)
  otherwise, \(\langle a[i] \rangle_{H,\mu,\theta,\nu,\kappa}\) is the vertex \(v\) reachable from \(\theta(a)\) by a path of \(\nu(i)\) edges labeled by an array field.

Data terms are interpreted in \(\mathbb{D}\) as follows: (1) \(\langle d \rangle_{H,\mu,\kappa} = \kappa(d);\) (2) for any \(dt \in \mathcal{DF}\), \(\langle dt(x) \rangle_{H,\mu,\kappa} = L_D(\mu(x))(dt)\); (3) for any \(a \in \mathcal{O}\), \(\langle o(t_1, \ldots, t_n) \rangle_{H,\mu,\kappa} = o(\langle t_1 \rangle_{H,\mu,\kappa}, \ldots, \langle t_n \rangle_{H,\mu,\kappa}).\) The interpretation \(\langle \text{Exp} \rangle_{\nu}\) of an index expression \(\text{Exp}\) as an integer is defined as usual.

Then, the interpretation \(\llbracket \phi \rrbracket_{H,\mu,\theta,\nu,\kappa}\) of a \textit{g}CSL formula \(\phi\) is a value in \(\{0, 1, \bot\}\). Boolean operators are interpreted as usual on \(\{0, 1\}\), and the boolean composition of \(\bot\) with any value leads to \(\bot\). Then, for every formula \(\phi\),

87
CHAPTER 4. PROGRAMS WITH DYNAMIC DATA STRUCTURES

- if there exists an array term \( a[\text{ind}] \) of \( \varphi \) such that \( \llbracket a[\text{ind}] \rrbracket_{H,\mu,\theta,\nu,\kappa} = \bot \) then 
  \( \llbracket \varphi \rrbracket_{H,\mu,\theta,\nu,\kappa} = \bot \):
- otherwise, \( \llbracket \varphi \rrbracket_{H,\mu,\theta,\nu,\kappa} = 1 \) iff \( H \models_{\mu,\theta,\nu,\kappa} \varphi \) and \( \llbracket \varphi \rrbracket_{H,\mu,\theta,\nu,\kappa} = 0 \) iff \( H \not\models_{\mu,\theta,\nu,\kappa} \varphi \),
  where the relation \( \models \) is defined in Figure 4.27.

Given a gCSL formula in prenex normal form \( \varphi = Q. \phi \), a heap \( (G, \delta) \), and \( \mu, \theta, \nu, \kappa \) evaluations for free variables of gCSL, then the terms that evaluate in \( \bot \) are the array terms of the following form:

- \( a[\text{ind}] \) where \( a \in \text{Arr} \) and \( \text{ind} \in \text{Ind} \) are free variables, such that \( \mu \) interprets \( \text{ind} \) into a value that corresponds to a position out of the bounds of the array \( \theta(a) \), i.e. \( \mu(i) \notin \text{Dom}(\theta(a)) \),

- \( a[\text{ct}] \) where \( a \in \text{Arr} \) is a free variable, and \( \text{ct} \) is an integer constant which is outside the bounds of the array \( \theta(a) \) denoted by \( a \), i.e. \( \text{ct} \notin \text{Dom}(\theta(a)) \).

These are all the terms that can evaluate to \( \bot \) because according to the syntax of gCSL formulas the terms \( dt(a[\text{ind}]) \) are a shorthand for \( dt(x) \) where \( x \) is a location variable and \( x = a[\text{ind}] \).

We omit the subscripts of \( [\cdot] \) and \( \models \) when they are clear from the context.

An atomic formula \( x^{A,B,:\text{ind}} \rightarrow x' \) is a reachability predicate expressing the fact that there is a nonempty path in the heap graph relating the locations designated by \( x \) and \( x' \) such that (1) all its vertices are distinct, except maybe for the extremal points, (2) all its edges are labeled by a set which includes \( A \), (3) all its vertices are not labeled with program variables from \( B \), and (4) its length (i.e., the number of its edges) is \( \text{ind} \) which is either an index variable, or a constant. The formula \( x^{A,B} \rightarrow x' \) is similar but it does not allow to refer to the length of the path (the set \( B \) is omitted when it equals \( \emptyset \)). Moreover, the formula \( x^{H} \rightarrow x' \) with \( H \) a set of non-recursive pointer fields in \( \text{PF} \) denotes the fact that there exists an edge between \( x \) and \( x' \) whose label contains \( H \).

Let us discuss some subtleties of the semantics of the reachability predicates and formulas over arrays.

**Reachability:** The heap in Figure 4.28 satisfies the formula \( \exists z, x, x'. x^{\{f_1, f_2\}} \rightarrow x' \land x^{\{b\}} \rightarrow z \) (we omit the valuation of the data variables). The same heap satisfies the formula \( \exists x, x'. x^{\{f_1, f_2\}, \text{ind}} \rightarrow x' \land \text{ind} \geq 2 \). Even if the heap satisfies the conjunction \( \exists x, x'. x^{\{f_1, f_2\}, \text{ind}} \rightarrow x' \land x^{\{g\}, \text{ind}} \rightarrow x' \land \text{ind} \geq 2 \) it does not satisfy the formula \( \exists x, x'. x^{\{f_1, g\}, \text{ind}} \rightarrow x \land \text{ind} \geq 2 \). The only \( f_1 \)-path of length two does not contain the same vertices as the only \( g \)-path of length two.

**Arrays:** Recall that given a heap \( H \) and some valuations of the variables, the value of a term \( a[i] \) is well-defined (\( \neq \bot \)) if the value of \( i \) is in the domain of the array associated with \( a \). Then, the interpretation of a formula \( \exists i. \varphi \) (resp. \( \forall i. \varphi \)) is 1 if the interpretation of \( \varphi \) is 1 for some value (resp. for all values) of \( i \) in the domains of all arrays \( a \) such that \( a[i] \) occurs in \( \varphi \).

For example, let us consider the heap in Figure 4.29 (again, we omit the valuation of the data variables). It is built over a type system where \( \mathbb{D} = \mathbb{Z} \) (data are integers), it has only one record type denoted by \( \bigcirc \), \( af \) is the array field corresponding to arrays of integers and \( dt \) is an integer data field from the record type \( \bigcirc \).
4.4. A GENERAL LOGIC GCSL FOR REASONING OVER PROGRAM CONFIGURATIONS

For reasoning over program configurations, GCSL can be used:

\[ H \models_{\mu,\theta,\nu,\kappa} \ell(x) \quad \text{iff} \quad L(\ell) = \mu(x) \]

\[ H \models_{\mu,\theta,\nu,\kappa} x = t \quad \text{iff} \quad \mu(x) = \llangle t \rrangle_{H_{\mu,\theta,\nu}} \]

\[ H \models_{\mu,\theta,\nu,\kappa} x \xrightarrow{A, B, \text{ind}} x' \quad \text{iff} \quad \text{there exists a path } \mu(x) = w_0, w_1, \ldots, w_m = \mu(x') \text{ such that } \]

\[ m = \nu(\text{ind}) \geq 1, i/w_j \neq i/j' \text{ if } j \neq j' \text{ and } (j, j') \notin \{(0, m), (m, 0)\}, \]

for any \( \ell \in B \) and for any \( j \geq 1, L(\ell) \neq w_j, \]

if \((w_j, w_{j+1}) \in E \) then \( A \subseteq L_E(w_j, w_{j+1}) \), and

if \((w_{j+1}, w_j) \in E \) then \( A \subseteq L_E(w_{j+1}, w_j) \),

\[ H \models_{\mu,\theta,\nu,\kappa} x \xrightarrow{H} x' \quad \text{iff} \quad \text{there exists an edge from } \mu(x) \text{ to } \mu(x') \text{ with } \mu(x) \neq \mu(x') \text{ and } H \subseteq L_E(\mu(x), \mu(x')) \text{ or there exists an edge from } \mu(x') \text{ to } \mu(x) \text{ with } \mu(x) \neq \mu(x') \text{ and } \overline{H} \subseteq L_E(\mu(x'), \mu(x)), \]

\[ H \models_{\mu,\theta,\nu,\kappa} \text{Exp} < ct \quad \text{iff} \quad \llangle \text{Exp} \rrangle_{\nu} < ct, \]

\[ H \models_{\mu,\theta,\nu,\kappa} p(t_1, \ldots, t_n) \quad \text{iff} \quad p(\llangle t_1 \rrangle_{H_{\mu,\nu,\kappa}}, \ldots, \llangle t_n \rrangle_{H_{\mu,\nu,\kappa}}) \]

\[ H \models_{\mu,\theta,\nu,\kappa} \exists x. \varphi \quad \text{iff} \quad \text{there exists } w \in V \text{ with } \tau(w) = \tau(x) \]

such that \( H \models_{\mu[x\leftarrow w],\theta,\nu,\kappa} \varphi \)

\[ H \models_{\mu,\theta,\nu,\kappa} \exists a. \varphi \quad \text{iff} \quad \text{there exists } w \in AV \text{ with } \tau(w) = \tau(a) \]

such that \( H \models_{\mu,\theta[\alpha\leftarrow w],\nu,\kappa} \varphi \)

\[ H \models_{\mu,\theta,\nu,\kappa} \exists d. \varphi \quad \text{iff} \quad \text{there exists } \text{val} \in D \text{ such that } H \models_{\mu,\theta,\nu,\kappa[dt\leftarrow\text{val}]} \varphi \]

\[ H \models_{\mu,\theta,\nu,\kappa} \exists i. \varphi \quad \text{iff} \quad \text{there exists } m \in N \text{ such that } H \models_{\mu,\theta,\nu,\kappa[\text{ind}\leftarrow m],\kappa} \varphi. \]

Figure 4.27: The relation \( \models \) between heaps and gCSL formulas

The semantics of the existential quantifier implies that:

\[ [\exists i. (dt(a[i]) = 3 \land dt(c[i]) = 4)] = 1, \quad [\exists i. (dt(a[i]) = 3 \lor dt(c[i]) = 8)] = 0, \]

and \([\exists i. (dt(a[i]) = 3 \land dt(c[i]) = 8)] = 1\). Notice also that the formula

\[ [\exists i. (dt(a[i]) = 4 \lor dt(c[i]) = 8)] = 0 \text{ whereas } \llbracket [\exists i. dt(a[i]) = 4] \lor \llbracket [\exists i. dt(c[i]) = 8] \rrbracket = 1. \]

This shows that \( \lor \) (resp. \( \land \)) does not distribute, in general, w.r.t. \( \exists \) (resp. \( \forall \)). However, these distributivity properties hold in the fragment of gCSL without arrays.

The semantics of the universal quantifier implies that on the heap graph from Figure 1.29, \([\forall j. dt(a[j]) = 3] = 1 \text{ and } [\forall j. dt(c[j]) = 4] = 0 \text{ but } [\forall j. dt(a[j]) = 3 \land dt(c[j]) = 4] = 1\). The later holds because \([\exists i. (dt(a[i]) = 3 \lor dt(c[i]) = 4)] = 0 \text{ (only for } i = 3 \text{ the model satisfies } dt(c[i]) \neq 4 \text{ but in this case } [dt(a[3]) = 3] = 1\). Moreover, when universally quantified index variables are used in terms over universally quantified array variables then their evaluation depends on the evaluation of the universally quantified array variables. For example, on the heap graph from Figure 4.29, \([\forall b \forall j. dt(b[j]) \leq 4] = 0 \text{ because } [\exists b \exists j. dt(b[j]) > 4] = 1\).
### 4.4.3 Examples of program assertions in \textit{gCSL}

Let $P_{\Pi}$ be a program and let $\Sigma$ be the type system induced by the record type definition in $P_{\Pi}$. If the program $P_{\Pi}$ has no procedures, then all assertions are \textit{gCSL} formulas over the type system $\Sigma^L$ (the type system obtained from $\Sigma$ by adding the variables of the logic). If the program has procedures, denoted $P_\pi$, then the assertions of each procedure are \textit{gCSL} formulas over $\Sigma^L_\pi$ which is obtained by adding the variables of the logic to the type system of the procedure $P_\pi$, $\Sigma_\pi$.

**Program Insert:** Consider the program \textbf{Insert} given in Figure 4.3 over the data structure defined in Figure 4.1. All the properties given in Section 4.2.2 describing configurations of a program that manipulates this data structure are expressible in \textit{gCSL} over the type system $\Sigma$ given in Figure 4.4 extended to data and location variables (assuming Presburger arithmetics as data logic, for instance) as follows:

- $P_{\Pi}$
- $\Sigma$
- $\Sigma^L$
- $\Sigma^L_\pi$
- $P_\pi$
- $\Sigma_\pi$
4.4. A GENERAL LOGIC GCSL FOR REASONING OVER PROGRAM CONFIGURATIONS

Shape:

doubly-ll := “the array $a$ contains in each cell a reference to an acyclic doubly linked list”,

$$\forall i. \exists dl_i. (a[i] \xrightarrow{\text{dll}} dl_i \land (dl_i = \text{null} \lor (dl_i \xrightarrow{\text{next}, \emptyset, 1} \text{null} \land dl_i \xrightarrow{\text{prev}, \emptyset, 1} \text{null})))$$

$$\exists dl. (dl \xrightarrow{\text{next, prev}} dl \land dl \xrightarrow{\text{next}, \emptyset, 1} \text{null})$$

root-fld := “each cell in the doubly linked lists stores in the field root a reference to the entry of the array referencing the list”

$$\forall i. \forall dl_i, dl. (a[i] \xrightarrow{\text{dll}} dl_i \land dl_i \xrightarrow{\text{next}} dl \Rightarrow dl \xrightarrow{\text{root}} a[i])$$

doubly-cll := “the doubly linked lists are cyclic”.

$$\forall i. \forall dl_i. (a[i] \xrightarrow{\text{dll}} dl_i = \Rightarrow dl_i \xrightarrow{\text{next, prev}} dl_i))$$

Sizes:

dll-len2 := “each doubly-linked list has at least two elements”

$$\forall i. \exists dl_i, dl_i'. (a[i] \xrightarrow{\text{dll}} dl_i \land dl_i \neq dl_i' \land dl_i' \neq \text{null} \land dl_i \neq \text{null} \land dl_i \xrightarrow{\text{next, prev}} dl_i')$$

dll-len := “the array $a$ is sorted in decreasing order w.r.t. the lengths of the pointed lists”

$$\forall j, j'. (j < j' \Rightarrow \exists dl_j, dl_j', l, l'. (a[j] \xrightarrow{\text{dll}} dl_j \land a[j'] \xrightarrow{\text{dll}} dl_j' \land dl_j \neq \text{null} \land dl_j' \xrightarrow{\text{next}, \emptyset, l} \text{null} \land dl_j' \xrightarrow{\text{next}, \emptyset, l'} \text{null} \land l' \leq l))$$

Data:

sorted-id := “the array $a$ is sorted w.r.t. the values of the field id”

$$\forall i, j. (i < j \Rightarrow \text{id}(a[i]) < \text{id}(a[j]))$$

flag-1 := “there exists a list with all fields flag set to 1”

$$\exists i. \exists dl_i. (a[i] \xrightarrow{\text{dll}} dl_i \land \forall y. (dl_i \xrightarrow{\text{next}, \emptyset} y \Rightarrow \text{flag}(y) = 1))$$

Program Dispatch: Consider the program Dispatch given in Figure 4.5. Assuming Presburger arithmetics as data logic, all the properties given in Section 4.2.2 describing configurations of this program are expressible in gCSL over the type system $\Sigma$ given in Figure 4.6 extended to data and location variables as follows:
CHAPTER 4. PROGRAMS WITH DYNAMIC DATA STRUCTURES

Shape:

\( \text{a-singly-ll} \ ::= \ "a \text{ points to an acyclic singly-linked list or it points to null}" \)

\[ \text{a} \xrightarrow{\text{next}} \text{null} \lor \text{a} = \text{null} \]

Notice that at the end of the program the lists pointed to by \( \text{gr} \) and \( \text{sm} \) are disjoint. Moreover, all the other pointer variables point to null except maybe for \( a \) which has the same value either as \( \text{gr} \) or as \( \text{sm} \). This is expressed by the following formula:

\[ \text{disjoint-gr-sms} ::= (\text{gr} \xrightarrow{\text{next}} \text{null} \lor \text{gr} = \text{null}) \land (\text{sm} \xrightarrow{\text{next}} \text{null} \lor \text{sm} = \text{null}) \land \text{disjoint}(\text{sm}, \text{gr}) \land (a = \text{sm} \lor a = \text{gr}), \]

where

\[ \text{disjoint}(v, w) ::= (v \neq w) \land \forall y_1, y_2. (v \xrightarrow{\text{next}} y_1 \land w \xrightarrow{\text{next}} y_2) \implies y_1 \neq y_2. \]

Data:

\( \text{gr-greater-v} ::= \ "\text{all elements in the list pointed to by gr have the values of the data field dt greater than v}" \)

\( (\text{gr} = \text{null}) \lor (\text{gr} \neq \text{null} \land \text{dt(gr)} \geq v) \lor \forall y. \text{gr} \xrightarrow{\text{next}} y \xrightarrow{\text{next}} \text{null} \implies \text{dt}(y) \geq v \)

Program Fibonacci: The language of assertions for the program given in Figure 4.17 is defined by gCSL formulas over the type system \( \Sigma \) in Figure 4.18 extended to data and location variables. The difference with the previous program is that the specification and the assert/assume statements of each procedure are defined in gCSL over \( \Sigma^L \) restricted to the local variables of the procedure. For example, to express the specification of the procedure Fibonacci one cannot use the variable \( n \) because it is a variable local to the main procedure. The postcondition of this procedure (head points to a Fibonacci sequence) is expressed by the following gCSL formula:
4.4. A GENERAL LOGIC gCSL FOR REASONING OVER PROGRAM CONFIGURATIONS

Fibo-seq ::= head = null \lor
            \left(\begin{array}{l}
            \text{head} \xrightarrow{\text{next}} \text{null} \land \text{dt}(\text{head}) = 1 \land \\
            \forall y. \left(\begin{array}{l}
            \text{head} \xrightarrow{\text{next}, 1} y \Rightarrow \text{dt} = 1
            \end{array}\right) \\
            \forall y_1, y_2. \left(\begin{array}{l}
            \text{head} \xrightarrow{\text{next}, 1} y_1 \land \\
            y_1 \xrightarrow{\text{next}, 1} y_2 \Rightarrow (\text{dt} = 1 \land \text{dt} = 2)
            \end{array}\right) \\
            \forall y_1, y_2, y_3. \left(\begin{array}{l}
            \text{head} \xrightarrow{\text{next}} y_1 \land \\
            y_1 \xrightarrow{\text{next}, 1} y_2 \land \\
            y_2 \xrightarrow{\text{next}, 1} y_3 \land \\
            y_3 \xrightarrow{\text{next}} \text{null} \Rightarrow \text{dt}(y_3) = \text{dt}(y_1) + \text{dt}(y_2)
            \end{array}\right)
            \end{array}\right)\right)

Program AddV: The specification of the procedure \text{addV} and the assertion from the program in Figure 4.19 are described using gCSL formulas over the type system \(\Sigma^L\), defined from \(\Sigma\) given in Figure 4.20. The postcondition of the procedure \text{addV} expresses a relation between the configuration of the memory at the beginning of the procedure and the configuration of the memory at the end of the procedure. As we have explained in Section 4.3.1, the corresponding formula should describe a heap containing two parts: one part describes the input configuration and one part describes the output configuration. The variables in the input configuration are superscripted with 0 in order to distinguish them from the variables in the output configuration. To express relation we extend \(\Sigma^L\) with a set of variables \(\text{loc}^0\) such that for each procedure variable \(\text{loc} \in \text{loc}\) there is a unique variable \(\text{loc}^0 \in \text{loc}^0\) and \(\tau(\text{loc}^0) = \tau(\text{loc})\). The formula corresponding to the postcondition of this procedure is expressed in gCSL over \(\Sigma\) restricted to \{\text{head, headi, aux}\} \cup \{\text{head}^0, \text{headi}^0, \text{aux}^0\}\) (such that \(\tau(\text{head}^0) = \tau(\text{head}) = \text{list}, \tau(\text{aux}^0) = \tau(\text{aux}) = \text{list}\) and \(\tau(\text{headi}^0) = \tau(\text{headi}) = \text{list}\):

\[
\text{add-V ::= (head = null \land head}^0 = \text{null) \lor} \\
\left(\begin{array}{l}
\text{head}^0 \xrightarrow{\text{next}} \text{null} \land \text{head} \xrightarrow{\text{next}} \text{null} \land \\
\forall y. \left(\begin{array}{l}
\text{head}^0 \xrightarrow{\text{next}} y \Rightarrow \neg \text{head} \xrightarrow{\text{next}} y
            \end{array}\right) \land \\
\forall y. \left(\begin{array}{l}
\text{head} \xrightarrow{\text{next}} y \Rightarrow \neg \text{head}^0 \xrightarrow{\text{next}} y
            \end{array}\right) \land \\
\forall y^0, y, \forall l. \left(\begin{array}{l}
\text{head}^0 \xrightarrow{\text{next}, 1} y^0 \land \text{head} \xrightarrow{\text{next}, 1} y \Rightarrow \text{dt}(y) = \text{dt}(y^0) + v
            \end{array}\right)
            \end{array}\right)\right)
\]
CHAPTER 4. PROGRAMS WITH DYNAMIC DATA STRUCTURES

The sortedness property on singly-linked lists is expressed by the following formula:

\[
\text{list-sorted} ::= \text{head} = \text{null} \lor \\
(\text{head} \xrightarrow{\text{next}} \text{null} \land \\
\forall y, y'. ((\text{head} \xrightarrow{\text{next}} y \land (y \xrightarrow{\text{next}} y' \lor y = y') \land y' \xrightarrow{\text{next}} \text{null})) \lor \\
(\text{head} = y \land (y = y' \lor y \xrightarrow{\text{next}} y' \land y' \xrightarrow{\text{next}} \text{null})) \\
\implies \text{dt}(y') \geq \text{dt}(y)
\]

A model for this formula does not constrain the initial configuration of the main procedure.

**Program Quicksort:** The specification and the assertions from the program given in Figure 4.21 are expressible in \text{gCSL} over the type system induced by the program extended with logic variables. It is important to notice that the equality of two singly-linked lists is expressible in \text{gCSL} by:

\[
\text{list-eq} ::= (\text{h1} = \text{null} \land \text{h2} = \text{null}) \lor \\
(\text{h1} \xrightarrow{\text{next}} \text{null} \land \text{h2} \xrightarrow{\text{next}} \text{null} \land \\
\text{h1} \xrightarrow{\text{next}, l_1} \text{null} \land \text{h2} \xrightarrow{\text{next}, l_2} \text{null} \land l_1 = l_2 \land \text{disjoint(h1,h2)} \\
\forall y_1, y_2 \forall l. (\text{h1} \xrightarrow{\text{next}, l} y_1 \land \text{h2} \xrightarrow{\text{next}, l} y_2) \implies \text{dt}(y_1) = \text{dt}(y_2).
\]
Chapter 5

Program verification using fragments of $gCSL$

The verification of SimpleC programs requires reasoning about complex unbounded size structures, that may carry data ranging over infinite domains. In a logic based framework, the verification problem relies on decision procedures for logics on data structures. The main issue within this framework is to define a logic which is expressive enough to express assertions about program configurations and has a decidable satisfiability problem.

In this chapter, we propose a fragment of the $gCSL$ logic, called CSL (Composite Structures Logic), which, we believe, offers a good compromise between expressiveness and decidability. This logic is able to express properties of multi-linked lists, arrays and composite structures like lists of lists or arrays of lists. The models of CSL are graphs (of arbitrary size and shape) representing memory configurations. Vertices represent objects of record type, while edges represent pointer fields; each vertex is typed according to the type of the record object it represents. CSL is a multi-sorted first order logic over graphs, parametrized by a first order logic on the data domain, denoted $FO(\mathbb{D}, \mathbb{O}, \mathbb{P})$. Its formulas use reachability predicates, linear constraints and predicates from $FO(\mathbb{D}, \mathbb{O}, \mathbb{P})$.

The reachability predicate describes paths formed of edges labeled by sets of pointer fields. For example, $n \xrightarrow{f, g} m$ states that there exists a path between the nodes denoted by $n$ and $m$, and each edge on this path is labeled by the pointer fields $f$ and $g$.

The linear constraints are used to reason about the size of the allocated structure. For example, $n \xrightarrow{f} \mathsf{null} \land m \xrightarrow{g, \ell} \mathsf{null} \land \ell = \ell'$ states that two singly-linked lists with an equal number of elements are allocated.

Finally, data constraints are expressed using predicates from the underlying logic. These predicates are build over terms representing vertices (record objects), and data fields. For example, $p \xrightarrow{data} \mathsf{null} \land data(p) = 3$ states that $p$ points to a memory cell that contains the integer 3 (accessed by the data field $data$) and a reference to $null$.

The CSL allows a restricted form of quantifier alternation. To obtain a logic with a decidable satisfiability problem, the quantification is restricted according to an ordered partition on the vertex types. This partition decomposes the graph into classes and assigns to each vertex in the graph a level. (Notice, that we do not require that the induced graph by this partition be a tree.) Then, roughly speaking, by associating a level to positions and quantifiers over them, the quantification part of CSL formulas has the form $\exists^*_k \forall^*_k \exists^*_{k-1} \forall^*_{k-1} \ldots \exists^*_1 \forall^*_1$. Allowing such quantifier alternation increases the expressiveness power of the handled initial conditions and invariants.

We prove that the satisfiability problem of CSL can be reduced to the satisfiability problem of its underlying data logic, based on a small model property. To prove this property, in general the idea used when there are no length constraints is that data can impose only upper bounds on the size of models, so the size of the minimal models is
determined by the number of existentially quantified variables. For example, a formula \( \phi \), describing a strictly sorted list such that the value of the first element is 1 and the value of the last one is 5, has as models lists with at most 5 elements. The first and the last element of the list are denoted by some existentially quantified variables. The small model for this formula is a list with two elements, whose data fields have value 1, respectively 5.

On the other hand, length constraints may impose lower and upper bounds on the size of the models. Due to the lower bounds the size of the (small) model is not determined only by the number of existentially quantified variables. Therefore, the question that arises is how to compute the bound on the size of the models in the presence of length constraints. For example, consider that we take the conjunction between the previous formula \( \phi \) and \( l_1 + l_2 \leq 8 \land l_2 - l_1 \geq 2 \land l_1 \geq 1 \), where \( l_2 \) denotes the length of the list constrained by \( \phi \) and \( l_1 \) is an integer variable. The idea is to define a system of linear inequations starting from the CSL formula and to solve it using multi-objective integer linear programming (MOIPL), where each length variable is a different objective to minimize. Using MOIPL we obtain for the considered example the following solutions: \((l_1 = 1, l_2 = 7), (l_1 = 2, l_2 = 6), (l_1 = 3, l_2 = 5)\). Moreover, notice that \( l_1 = 1 \) and \( l_2 = 7 \) is not a good solution \( l_2 \) is constrained to be less than or equal to 5. Therefore, the decidability proof must take into consideration the fact that upper bounds imposed by the length constraints might not coincide with the upper bounds imposed by the data constraints. The quantifier alternation is handled by applying a small model argument on every level of the considered ordered partition.

Another important fact we prove is that CSL is effectively closed under the computation of post images (i.e., strongest post condition). We show how this result, together with the decidability of the satisfiability problem can be used in pre-post condition reasoning and invariant checking.

Outline: In Section 5.1 we describe the verification framework and we state the undecidability of \( \text{gCSL} \), which is the main reason for introducing the fragment CSL. The syntax of CSL is defined in Section 5.2.1. In Section 5.2.3 we give the proof for the decidability of the satisfiability problem of CSL\(_1\), a fragment of CSL based on a trivial partition that puts all the vertex types in the same class. This result is extended to the full CSL in Section 5.2.5. In Section 5.2.2 we introduce a fragment of CSL closed under negation, called ICSL, that can be used to express inductive invariants. The proof for the closure under the post image computation is given in Section 5.2.6. Finally, in Section 5.2.7 we give several example of verified assertions. Among them, we consider a program that manipulates an array of doubly-linked lists in order to illustrate the pre-post condition reasoning using CSL.

5.1 Proving partial correctness using a logic based framework

Given a program \( P \) let \( \Sigma \) be the associated type system. In this chapter, we consider that all the program annotations are \( \text{gCSL} \) formulas over \( \Sigma^L \), the type system obtained by extending \( \Sigma \) to logic variables. Also, we assume that program annotations use a set of location variables which have the same name as the pointer variables in \( P \text{Var} \). A location variable \( p \in P \text{Var} \) is always interpreted to a vertex that is labeled by the pointer variable \( p \).
Furthermore, because of this assumption, we consider that gCSL formulas do not contain labeling predicates of the form $p(x)$, where $p \in PVar$ and $x$ is a location variable. Every gCSL formula that appears in an assert/assume statement or as a loop annotation has as free variables only the local variables of the procedure where the annotation is placed. The formulas from the procedure specification have as free variables only the formal input and output parameters of the procedure.

**Programs with basic statements:** Let $P_{\Pi}$ be a program without loops and procedure calls. To prove an assert statement one has to apply the framework of pre/post-condition reasoning described in Section 3.2. Let $Pre$ be the set of initial memory configurations of $P_{\Pi}$, $Post$ the set of memory configurations described by the formula in the assert statement, $s_{P_{\Pi}}$ the starting node of the CFG, $n_a$ the node of CFG that corresponds to the assert statement, and $post$ is the operator that gives the successors of a set of states in the transition system associated to the program $P_{\Pi}$ (defined in Section 4.2.3).

Then, for every sequence of statements $a_1 \cdot a_2 \ldots \cdot a_p$ that label the edges of a path between $s_{P_{\Pi}}$ and $n_a$ (in the CFG of $P_{\Pi}$), the following should hold:

$$post(a_p, post(a_{p-1}, \ldots (post(a_1, (s_{P_{\Pi}}, Pre)) \ldots )) \subseteq \{n_a\} \times Post.$$ 

The postconditions from the specification of $P_{\Pi}$ are proved in a similar manner (in this case, the control point $n_a$ is replaced by the exit point of $P_{\Pi}$).

Let $\varphi_{pre}, \varphi_{post} \in gCSL$ with $Pre = \langle \varphi_{pre} \rangle$ (where $\langle \varphi_{pre} \rangle$ is the set of models of $\varphi_{pre}$) and $Post = \langle \varphi_{post} \rangle$ be the formulas which represent the set of initial memory configurations, respectively the set of memory configurations described by the assert statement. To carry out pre/post condition reasoning in a logic-based framework:

1. the logic must be closed under the computation of the strongest post-condition, that is, for any gCSL formula $\varphi$ and any edge $pc \stackrel{a}{\rightarrow} pc'$ in the CFG of $P_{\Pi}$, one can compute a gCSL formula denoted $post(a, \varphi)$ such that $post(a, (pc, [\varphi])) = (pc', [post(a, \varphi_{pre})]);$

2. the logic must have a decidable satisfiability problem: the inclusion between two sets of states (e.g., $Pre' \subseteq Post$) corresponds to the validity of the implication between two gCSL formulas (e.g., $\varphi_{pre} \implies \varphi_{post}$, where $\varphi_{pre} = post(a_p, post(a_{p-1}, \ldots (post(a_1, \varphi_{pre})) \ldots ))$). The validity of an implication between two gCSL formulas $\varphi_1 \implies \varphi_2$ is equivalent to the unsatisfiability of $\varphi_1 \land \neg \varphi_2$.

The assertion checking can be soundly approximated if $post(a, \varphi_{pre})$ is a formula that over-approximates the strongest post-condition, i.e., $post(a, (pc, [\varphi])) \subseteq [(pc', post(a, \varphi))]$, where $pc \stackrel{a}{\rightarrow} pc'$ is an edge in the CFG.

**Programs with loops:** Let $P_{\Pi}$ be a program with while loops but no procedure calls. We assume that all while loops are annotated with loop invariants (otherwise, we consider by default the trivial annotation /*@loop invariant true */).

Then, the loop
CHAPTER 5. PROGRAM VERIFICATION USING FRAGMENTS OF GCSL

```plaintext
while(cond){
    /* loop invariant ϕ */
    ...
}
```

is replaced by the statement `assume( ¬cond ∧ ϕ)` provided that \( \bigcup_{n \in CFG|_{loop}} \{(n, [ϕ])\} \), where \( CFG|_{loop} \) denotes the set of nodes in the CFG of \( P_Π \) that correspond to statements in the loop, is an inductive invariant for the transition system obtained by (1) restricting the transition system of the program to the set of states of the form \((n, H)\) with \( n \in CFG|_{loop} \), (2) taking as initial states, all the reachable states of the form \((n_c, H)\), where \( n_c \) is the first control point of the while loop and \( H \) is a heap.

Let \( Path(s_{P_Π}, n_c) \) be the set of all sequences of statements that label paths in CFG of \( P_Π \) between the starting node \( s_{P_Π} \) and \( n_c \), and \( ϕ_{pre} \) be the precondition of \( P_Π \). Also, let

\[
ϕ_{Init} = \bigvee_{a_1…a_n_c \in Path(s_{P_Π}, n_c)} post(a_n_c, … post(a_1, ϕ_{pre})).
\]

Then, \( ϕ \) is an inductive invariant for the loop if:

1. \( (n_c, [ϕ_{Init}]) \subseteq \{(n, [ϕ]) \mid n \in CFG|_{loop}\} \) which is equivalent to the validity of \( ϕ_{Init} \implies ϕ \) (notice that \( n_c \in CFG|_{loop} \)) or the unsatisfiability of \( ϕ_{Init} \land ¬ϕ \), and

2. \( post(a, (pc, [ϕ])) \subseteq (pc', [ϕ]), \) for every \( pc \xrightarrow{a} pc' \) with \( n, n' \in CFG|_{loop} \), which is equivalent to the validity of \( post(a, ϕ) \implies ϕ \) or the unsatisfiability of \( post(a, ϕ) \land ¬ϕ \).

The loops are replaced with `assume` statements. For nested loops, we start by replacing the inner most ones (for more details, see [Bradley 2007]). The verification of the specification for programs with loops is not complete because loop invariants describe an over-approximation of the set of memory configurations reached by the program. This over-approximation might be too coarse for proving the assertions. Moreover, it may happen that there exists no sufficiently precise invariant that can be expressed in gCSL.

**Programs with procedures:** Let \( P_Π \) be a program with procedures (defined as in Section [4.3.1]). We assume that every procedure \( P_π \) of the program \( P_Π \) is annotated by its specification (by default, we assume that the annotations are `true`). The annotations in the specification describe sets of memory configurations, that is, they are formulas in gCSL over \( Σ^C \) (for the verification problem, we do not consider formulas that describe relations between memory configurations using copies of the local variables superscripted with zero). The post-condition describes the values of the input and output parameters at the end of the procedure. (We assume that input parameters are passed by value but this does not prevent updates on their data and pointer fields, which are visible at the end of the procedure.)

Let \( P_π(\text{fi}, \text{fo}) \) be a procedure, where \( \text{fi} \) are the formal input parameters and \( \text{fo} \) are the formal output parameters, and let \( ϕ_{pre} \) and \( ϕ_{post} \) be two gCSL formulas representing the pre-condition and the post-condition of \( P_π \). Provided that the specification of \( P_π \) is correct, any procedure call \( P_π(\text{ai}, \text{ao}) \) is replaced by two statements

\[
\begin{align*}
\text{assert} & \left( \overline{ϕ_{pre}} \right); \\
\text{assume}_{proc} & \left( ϕ_{post} \right);
\end{align*}
\]
where $\varphi_{\text{pre}}$ and $\varphi_{\text{post}}$ are obtained from $\varphi_{\text{pre}}$ and respectively $\varphi_{\text{post}}$, by replacing formal parameters with actual parameters. The statement $\text{assume proc}$ composes the context of the call, given by the states reachable at this control point, and the output of the procedure described by the formula $\varphi_{\text{post}}$ (it corresponds to the concrete transformer for $\text{return } P_\pi(ai, ao)$; more details are given in Section 5.2.6).

To prove the specification of a (recursive) procedure $P_\pi$ we apply an inductive reasoning: we assume the given specification, we replace every procedure call by the two statements given in (5.1.1) and any loop by its invariant and then, we try to prove that the specification using a pre/post condition reasoning w.r.t. basic statements.

This is just a sound method for proving the specification. It may happen that we can not prove the post-condition of a procedure because the logic is not sufficiently expressive or because of weak invariants and incomplete specifications (for more details, see [Bradley 2007]).

In the next chapter, we will present a method to synthesize loop annotations (which are by construction inductive invariants) and procedure specifications when they are missing.

### Undecidability of gCSL

The logic gCSL is quite expressive, it is closed under the strongest post-condition but the satisfiability problem is undecidable: The satisfiability problem for gCSL is undecidable, because gCSL includes several fragments that have been previously shown undecidable, e.g. in [Börger 1997, Bradley 2006, Bouajjani 2007, Habermehl 2008].

**Theorem 5.1.1.** The satisfiability problem of gCSL is undecidable.

**Proof.** When the data domain is finite, the satisfiability problem of gCSL is undecidable in general since it subsumes the first-order logic on graphs with reachability [Börger 1997].

If we consider that the models are linear heap structures (i.e., one dimensional arrays or singly-linked lists) then, the fragment of gCSL containing formulas with quantifier prefix $\forall^* \exists^*$ is undecidable [Bradley 2006, Bouajjani 2007] even with a data logic like $(\mathbb{N}, =)$ (i.e., an enumerable data domain with only equality) and without using length constraints. The proof is based on a reduction from the halting problem of Turing machines to the satisfiability of a gCSL formula.

If we restrict the quantifier alternation in gCSL over linear heap structures to $\exists^* \forall^*$ and we consider that the underlying data logic is Presburger arithmetics then the undecidability of this fragment can be proven by a reduction from the halting problem for 2-counter automata [Minsky 1967]. To write formulas that describe exactly the halting runs of a 2-counter automaton we need either constraints of the form $j = j' + 1$, where $j$ and $j'$ are universally quantified index variables (see Lemma 4 in [Habermehl 2008]), or equivalently constraints of the form $y \equiv_{(f_{j+1},j)} y'$, where $y$ and $y'$ are universally-quantified location variables.

### 5.2 Composite Structure Logic. CSL

In the following, we define a syntactic fragment of gCSL which is relevant for program verification, that is it closed under post-computation (see Section 5.2.6) and it has a decidable satisfiability problem (see Section 5.2.5).
The decidability of the satisfiability problem is proven, using a small model argument. To this, (1) we restrict the class of the formulas involving quantifier alternation of the form \( \forall^* \exists^* \) and (2) we impose syntactic restrictions mainly to avoid formulas which are able to describe properties relating immediate successors of a list or array. Notice that, it is important to allow formulas with some form of \( \forall^* \exists^* \) quantifier alternation since they may correspond to natural assertions (such as doubly-ll, dll-len, and dll-len2 in Example 4.4.3).

### 5.2.1 Syntax

We define hereafter the **Composite Structures Logic** (CSL) as a fragment of gCSDL. First we define the restriction on the quantifier alternation. Let us consider a gCSDL formula of the form

\[
\exists^* \forall^* \exists^* \forall^* \ldots \exists^* \forall^*. \phi
\]

where \( \phi \) is quantifier free. Roughly, we require that if a variable \( x \) is existentially quantified within the scope of a universal quantification of some variable \( x' \), then the type of \( x' \) must be different from the one of \( x \) and from the types of all the variables which are (universally or existentially) quantified after (under the scope of) \( x \). This restriction is defined through the notion of ordered partition of the set of types.

**Definition 5.2.1** (Ordered partition of types). Let \( N \) be a natural number such that \( 1 \leq N \leq |RT| \). Then, an ordered partition over the set of types \( RT \) is a mapping \( \sigma : RT \to \{1, \ldots, N\} \). A type \( Rt \in RT \) is of level \( k \), for some \( k \in \{1, \ldots, N\} \), iff \( \sigma(Rt) = k \). We extend an ordered partition \( \sigma \) to the basic type \( D \) and we assume that it associates level 0 to all elements of \( D \).

Ordered partitions of the set of types induce ordered partitions on heap graphs: the notion of level is transferred from types to vertices in heap graphs. These partitions correspond to natural decompositions of heap structures into sub-structures according to the type definitions in the program. For example, in the data structure of Example 4.1(a), we may consider two levels: the first level contains the vertices denoting elements of the doubly-linked lists (i.e., \( \sigma(dll_{ty}) = 1 \)) and the second level contains the vertices denoting the array’s elements (i.e., \( \sigma(a_{ty}) = 2 \)). Notice that the quotient graph induced by this partition is cyclic: the edges labeled by dll go from level 2 to level 1, and edges labeled by root go from level 1 to level 2.

**k-stratified formulas** Given an ordered partition \( \sigma \) over \( RT \), we extend the definition of \( \sigma \) to variables from the logic: we consider that \( \sigma(x) = \sigma(\tau(x)) \) and \( \sigma(a) = \sigma(\tau(a)) \) for any location variable \( x \) or array variable \( a \). For an index variable \( i \), we let the level \( \sigma(i) \) to be fixed arbitrarily. Then, for each level \( k \), with \( 1 \leq k \leq N \), let \( Q_k \) be a sequence of quantifiers over variables of level \( k \) defined as follows:

\[
Q_k = \exists a^k \exists x^{\leq k} \exists d^k \forall b^k \forall y^k \forall j^k
\]

where \( a^k \) and \( b^k \) are sets of array variables of level \( k \), \( x^{\leq k} \) is a set of location variables of level less than or equal to \( k \), \( y^k \) is a set of location variables of level \( k \), \( i^k \) and \( j^k \) are sets of index variables of level \( k \), and \( d^k \) is a set of data variables. (There is no level restriction on data variables.)

**Definition 5.2.2** (k-stratified formula). A gCSDL formula is k-stratified if it is of the form \( Q_k Q_{k-1} \ldots Q_1 Q. \phi \), where \( Q \) is a set of quantifiers over data variables and \( \phi \) is a quantifier-free formula.
5.2. COMPOSITE STRUCTURE LOGIC. CSL

The fragment CSL<sub>k</sub> To define the CSL fragment, we consider in addition to stratification some restrictions on the occurrences of universally quantified location, array, and index variables. All these restrictions, except for the last one, are necessary to avoid atomic formulas over universal variables which are essential for writing formulas that describe halting runs of a two-counter automaton. The last restriction is important for our proof technique, which is based on establishing a small model property, to show the decidability of the satisfiability problem.

In the following, we formally define these restrictions and we give the reasons for introducing them:

UnivIdx: the atomic formulas over two universally quantified index variables \( j \) and \( j' \), without location terms, are of the form \( j - j' < 0 \) or \( j - j' = 0 \). Moreover, any such variable \( j \) must appear in at least one location term \( a[j] \);

Motivation: Without this restriction the logic becomes undecidable because it allows constraints of the form \( j' = j + 1 \), with \( j \) and \( j' \) universally quantified index variables.

Reach1: in the reachability sub-formula \( x^{A,B,ind} \rightarrow x' \), the location variables \( x \) and \( x' \), and the index variable \( ind \), are free or existentially quantified variables;

Motivation: Without this restriction the logic becomes undecidable because it allows constraints of the form \( y^{(f,\emptyset,1)} \rightarrow y' \), with \( y \) and \( y' \) universally quantified location variables.

Reach2: any reachability sub-formula \( y^{A,B} \rightarrow y' \) such that \( A \) contains more than one pointer field and \( y \) and \( y' \) are universally quantified appears under an even number of negations;

Motivation: This restriction is motivated by the small model property used to obtain the decidability of CSL. Roughly, when there are no constraints on the lengths of the paths in the heap graph, the small model property says that a CSL formula is satisfiable iff it has a model of size bounded by the number of existential variables. This does not hold if atomic formulas of the form \( y^{A,B} \rightarrow y' \), where \( A \) contains more than one pointer field, appear under an odd number of negations. For example, there exist no model of size two for the following formula:

\[
\varphi := \exists x, x'. \left( x^{(f,\emptyset)} \rightarrow x' \land x^{(g,\emptyset)} \rightarrow x' \land \forall y, y'. \left( y^{(f,\emptyset)} \rightarrow y' \land y^{(g,\emptyset)} \rightarrow y' \land \neg y^{(f,g,\emptyset)} \rightarrow y' \right) \right).
\]

Lev: the constraints on lengths of lists and array indices must involve only one level, that is: (1) for any atomic formula \( x^{A,B,i} \rightarrow x' \), \( \sigma(i) = \sigma(x) \), (2) for any term \( a[i] \), \( \sigma(i) = \sigma(a) \), and (3) for any atomic formulas \( Exp < ct \) or \( Exp = ct \), all index variables in \( Exp \) have the same level.

Motivation: This restriction is motivated by our proof technique which consists in computing, for any CSL<sub>k</sub> formula \( \varphi \), an equi-satisfiable formula \( \varphi' \) without universal variables of level \( k \). This is based on computing a bound \( \alpha_k \) such that if \( \varphi \) is satisfiable then it has a model \( G \) such that the number of vertices of level \( k \) is less than \( \alpha_k \). The bound \( \alpha_k \) depends only on the atomic formulas over variables of level \( k \).
Definition 5.2.3 (CSL\(_k\)). We define the fragment CSL\(_k\) to be the smallest set of formulas which is closed under disjunction and conjunction, and which contains the set of all \(k\)-stratified formulas satisfying the constraints UnivIdx, Reach1, Reach2, and Lev.

The set of formulas of CSL is the union of the fragments CSL\(_k\) for all \(k \in \{1, \ldots, N\}\). Despite the syntactical restrictions in CSL, the logic is still quite powerful. Notice that all formulas given in Example 4.4.3 are in CSL (for the ordered partition defined above).

We prove that CSL has a decidable satisfiability problem. For the sake of readability, we present the proof of this result in two steps. First, we prove it for the fragment CSL\(_1\) (i.e., all location/array variables are in the same class of the considered ordered partition of the types), and then we show the extension to the full CSL.

5.2.2 ICSL. A fragment of CSL closed under negation

For pre/post-condition reasoning and for inductive invariance checking (assuming we have an annotated program with assertions and loop invariants) the precondition/postcondition of procedures, and the loop annotation must be in a fragment which is closed by negation (see Section 5.1).

The logic CSL is closed under disjunction and conjunction, but not under negation. We introduce hereafter such a fragment.

For each level \(k\), let ICSL\(_k\) be the smallest fragment of CSL which is closed under all boolean operations and which contains formulas of the form:

\[
S_k S_{k-1} \ldots S_1 S, \phi
\]

where \(S_k \in \{\exists a^k \exists i^k \exists x^k, \forall b^k \forall y^k \forall j^k\}\), \(S\) is a set of quantifiers over data variables and \(\phi\) is a quantifier-free formula in CSL such that:

- **BoundIdx**: the atomic formulas over two quantified index variables \(i\) and \(i'\), without location terms, are of the form \(i - i' < 0\) or \(i - i' = 0\). Moreover, any such variable \(i\) must appear in at least one location term \(a[i]\);

- **Reach1’**: in the reachability sub-formula \(x \xrightarrow{A,B,ind} x'\), the location variables \(x\) and \(x'\), and the index variable \(ind\), are free variables;

- **Reach2’**: any reachability sub-formula \(x \xrightarrow{A,B} x'\) or \(x \xrightarrow{A,B,ind} x'\) such that \(A\) contains more than one pointer field appears (1) under an even number of negations, when \(x, x'\) are universally quantified and (2) under an odd number of negations, when \(v, v'\) are existentially quantified.

Then, let ICSL = \(\bigcup_{1 \leq k \leq N} \text{ICSL}_k\). Notice that ICSL allows alternation between quantifiers provided that they concern different levels.

Examples of ICSL formulas are those given in Example 4.4.3 except doubly-11 and dll-len. Actually, it is possible to give an equivalent formula to doubly-11 which is in ICSL (see Section 5.2.7). The CSL formula dll-len can be used to constrain the initial condition of the program.

5.2.3 Deciding the satisfiability problem of CSL\(_1\)

In this section, we prove the decidability of the satisfiability problem for CSL\(_1\), when this problem is decidable for the underlying data logic. We first give an overview of the proof and then the details for the main steps. The main result of this section is the following:
5.2. COMPOSITE STRUCTURE LOGIC. CSL

**Theorem 5.2.1.** The satisfiability problem for closed CSL$_1$ formulas is decidable provided that the satisfiability problem of FO($\mathcal{D}, \mathcal{O}, \mathcal{P}$) is decidable.

5.2.3.1 Overview of the proof

Let $\varphi$ be a closed CSL$_1$ formula. We reduce its satisfiability to the satisfiability of a FO($\mathcal{D}, \mathcal{O}, \mathcal{P}$) formula. The reduction is done in four steps: (1) we build an equivalent formula obtained by enumerating all the truth valuations for the atomic formulas over existential variables, (2) we prove that if $\varphi$ has a model (i.e., a heap), then it must have a model of bounded size where the bound can be effectively computed from the syntax of $\varphi$, (3) for any bounded model $G$ of $\varphi$, we construct an equi-satisfiable formula without universal quantification over location, array, and index variables, and finally (4) we construct an equi-satisfiable formula without existential quantification over location, array, and index variables which is in FO($\mathcal{D}, \mathcal{O}, \mathcal{P}$).

Let us explain some of the issues in doing this reduction on an example. Consider the following formula (defined over FO($\mathbb{N}, \mathbb{N}, \{<\}$, where $\mathbb{N}$ denotes the set of natural numbers):

$$\psi_1 := \exists x, q, z. \left( q \neq z \land x^{(f)} q \land x^{(f)} \Rightarrow z \land dt(x) = 0 \land dt(q) = 2 \land \forall y, y'. \left( x^{(f)} y \land y^{(f)} \Rightarrow y' \Rightarrow dt(x) < dt(y) < dt(y') \right) \right)$$

(5.2.1) which says that there is an $f$-path from $x$ to $q$ and an $f$-path from $x$ to $z$, the data attached with $x$ (resp. $q$) is 0 (resp. 2), and the data are strictly ordered along $f$-paths that start in $x$ ($f$ is a recursive pointer field in $\mathcal{P}_{R}$). It can be checked that $\psi_1$ is satisfiable and has two minimal models of size three, either the graph $x \xrightarrow{f} q \xrightarrow{f} z$, or the graph $x \xrightarrow{f} z \xrightarrow{f} q$ (here we identify vertices with the variables they represent). Notice that the three vertices corresponding to $x$, $q$, and $z$ must belong to the same $f$-path since heap graphs are deterministic by definition. So, the size of a minimal model is the number of existentially quantified variables. Consider now the following formula obtained from $\psi_1$ by constraining the lengths of the paths connecting $x$ to $q$ and $x$ to $z$:

$$\psi_2 := \exists x, q, z. \exists i_1, i_2. \left( q \neq z \land x^{(f)} q \land x^{(f)} \Rightarrow z \land dt(x) = 0 \land dt(q) = 2 \land i_1 + i_2 \geq 8 \land \forall y, y'. \left( x^{(f)} y \land y^{(f)} \Rightarrow y' \Rightarrow dt(x) < dt(y) < dt(y') \right) \right)$$

(5.2.2) Again, since $x$, $q$, and $z$ must be part of a same $f$-path, there are two different possible heap graph “templates” to consider:

$$x \xrightarrow{i_1} q \xrightarrow{i_2} z \quad (5.2.3)$$

$$x \xrightarrow{i_2} z \xrightarrow{i_1} q \quad (5.2.4)$$

where each edge represents an $f$-path (of some length denoted $i_1$, $i_2$, or $l$), and we need to determine the minimal lengths of each of these paths, taking into account the constraints from the formula $\psi_2$. First, it can be seen that with the first (resp. second) template above there is an associated constraint on the lengths which is $i_1 + l = i_2 \land i_1 + i_2 \geq 8$ (resp. $i_2 + l = i_1 \land i_1 + i_2 \geq 8$). For the first (resp. second) template, the set of minimal solutions w.r.t. the usual ordering on vectors of natural numbers for the constraints on $i_1$, $i_2$, and $l$ is $\{(3, 5, 2), (2, 6, 4), (1, 7, 6)\}$ (resp. $\{(5, 3, 2), (6, 2, 4), (7, 1, 6)\}$). However, not all of these minimal solutions lead to minimal models of the formula. This is due
to the fact that data constraints can also impose constraints on the path lengths. From \( dt(x) = 0, dt(q) = 2 \), and the fact that the \( f \)-path between \( x \) and \( q \) is strictly sorted w.r.t. the values of \( dt \), it follows that the value of \( i_1 \) should be at most 2. This means that if the formula \( \psi_2 \) has a model, a minimal model of it should correspond to the first template with either \( i_1 = 2, i_2 = 6 \), and \( l = 4 \), or \( i_1 = 1, i_2 = 7 \), and \( l = 6 \). For the solution \((2,6,4)\), the minimal model is obtained from the template by inserting one vertex between \( x \) and \( q \) and five vertices between \( q \) and \( z \).

In fact, our proof shows that we can build from \( \psi_2 \) an equivalent formula \( \psi'_2 \) such that, if \( \psi_2 \) has a model, then it must have a model of size less or equal than the number of existentially quantified variables in \( \psi'_2 \). The outline of the proof is the following.

In the first step, detailed in Section 5.2.3.3, we enumerate all truth valuations for the atomic formulas over existentially quantified variables. This gives us the number of vertices imposed by the reachability constraints of the formula through existentially quantified location variables and terms of the form \( a[i] \), where \( a \) and \( i \) are existentially quantified.

In the second step, detailed in Section 5.2.3.4, we compute the vertices imposed by the length constraints, and we define a bound on the size of the minimal models. We prove that if there exists a model for the initial formula then, there exists a model whose size is less than or equal to the bound that we computed. Notice that this fact does not lead directly to an enumerative decision procedure for the satisfiability problem since the number of models of a bounded size is infinite in general, due to infinite underlying data domain.

In the third step, given in Section 5.2.3.5, we define an equi-satisfiable formula without universal quantification over location, array, and index variables.

Finally, in Section 5.2.3.6, we construct an equi-satisfiable formula without existential quantification over location, array, and index variables which is in FO(\( \mathbb{D}, \mathbb{O}, \mathbb{P} \)).

In these steps we may have already proved that the formula is unsatisfiable. Otherwise, its satisfiability is reduced to the satisfiability of a formula in the underlying data logic.

### 5.2.3.2 Running examples

To illustrate the constructions in the proof, we consider as running examples the following formulas (defined over FO(\( \mathbb{N}, \mathbb{N}, \{<\} \)):

- the formula \( \psi_3 \) describes two lists like in \( \psi_2 \) and an array whose data values between positions \( i_1 \) and \( i_1 + i_3 \), are sorted:

\[
\psi_3 := \exists a \exists x, q, z. \exists i_1, i_2, i_3. \left( q \neq z \land x \frac{1}{0} \frac{y}{1} \frac{a[i]}{1} \land q \land x \frac{1}{0} \frac{y}{1} \frac{a[i]}{1} \Rightarrow z \right) (5.2.5)
\]

\[
\land dt(x) = 0 \land dt(q) = 2 \land i_1 + i_2 \geq 8
\]

\[
\land \forall y, y'. \left( (x \frac{1}{0} \frac{y}{1} \Rightarrow y \land y \frac{1}{0} \Rightarrow y') \Rightarrow dt(x) < dt(y) < dt(y') \right)
\]

\[
\land dt'(a[i]) = 0 \land dt'(a[i_2]) = 4 \land i_1 + i_3 < i_2
\]

\[
\land \forall j, j'. i_1 \leq j < j' \leq i_1 + i_3 \Rightarrow dt'(a[j]) < dt'(a[j']) \leq 1
\]

We consider that the location variables \( x, q, z \) have some type \( Rt \) and that the type of the array variable \( a \) is \( Rt' \neq Rt \). The array field corresponding to \( Rt' \) is denoted as usual by \( aRt' \) and \( \tau(aRt') = Rt' \rightarrow Rt' \). The recursive pointer

104
5.2. COMPOSITE STRUCTURE LOGIC. CSL

Field \( f \) is of type \( \tau(f) = Rt \rightarrow Rt \). Also, \( dt \) and \( dt' \) are two data fields with \( \tau(dt) : Rt \rightarrow \mathbb{N} \) and \( \tau(dt') : Rt' \rightarrow \mathbb{N} \).

Besides the problems described in the overview of the proof, this example is used to illustrate the interaction between arrays and lists.

- The formula \( \psi_4 \) states that the heap graph contains a vertex \( x \) labeled by the value 1 and two arrays denoted by \( a \) and \( c \), whose data are less than 7 and sorted between positions 0 and \( i_1 + i_2 + i_3 \) (the sortedness property of the two arrays \( a \) and \( c \) is implied by the constraints on the universal array variable \( b \)):

\[
\psi_4 ::= \exists a, c \exists i_1, i_2, i_3 \forall j, j', j''. \left( dt(x) = 1 \land dt'(a[i_1]) \leq dt'(c[i_1]) \land dt'(a[i_2]) \leq dt'(c[i_3]) \land (j \leq j' \leq i_1 + i_2 + i_3) \implies dt'(b[j]) \leq dt'(b[j']) \land (dt'(a[j'']) \leq 7 \land dt'(c[j'']) \leq 7) \right).
\]

We consider that the location variable \( x \) is of type \( Rt \) and that the array variables \( a, b, \) and \( c \) are of type \( Rt' \). As in the previous example, the array field corresponding to \( Rt' \) is denoted by \( aRt' \). Also, \( dt \) and \( dt' \) are two data fields as above. This example is considered in order to illustrate the manipulation of the universally quantified array variables.

- The formula \( \psi_5 \) states that between vertices denoted by \( x \) and \( z \) there is a path following the pointer fields \( \{f_1, f_2\} \), a path following the pointer fields \( \{f_2, f_3\} \), and a path following the pointer fields \( \{f_4\} \). The universal quantification adds a dependency between the paths starting in \( x \) labeled by \( f_4 \) and the ones labeled by \( f_1 \):

\[
\psi_5 ::= \exists x, z \exists i, i', i'' \forall y. \left( x f_1 f_2 f_3 f_4 i, i', i'' \mapsto z \land z f_2 f_3 f_4 i, i', i'' \mapsto z \land f_4 1 f_4 i, i' \mapsto y \implies f_4 1 f_4 i, i' \mapsto y \right).
\]

We consider that the variables \( x, y, z \) are of type \( Rt \) and that \( f_i \) is a recursive pointer field with \( \tau(f_i) = Rt \rightarrow Rt \), for all \( 1 \leq i \leq 4 \). Using this example, we show how the procedure handles the new locations imposed by the atomic formulas over existentially quantified location and index variables.

5.2.3.3 Enumerating truth valuations of atomic formulas over existential variables

In this section, we give a procedure that enumerates all the truth valuations for the equality, reachability, and index atomic formulas over existential variables. Valuations of equality predicates are given by equivalence relations over terms, valuations of reachability predicates are given by transitive relations over equivalence classes of terms, and finally, valuations of index constraints are given by functions from atomic formulas to the set of truth values \{true, false\}.

Let

\[
\phi ::= \exists a \exists i \exists x \exists d \forall b \forall j \forall y \{ \exists d, \forall d \}^* \phi
\]
be a closed CSL₁ formula (φ is quantifier-free).

First, we introduce some notations for sets of terms from the formula φ. For every record type Rt ∈ RT, we denote by \( L_{Rt}(\varphi) \) the set of location terms of type Rt that appear in φ which are built over the existential variables (we use \( L_{Rt} \) when φ is clear from the context):

\[
L_{Rt}(\varphi) ::= \{ x \in x \mid \tau(x) = Rt \} \cup \{a[0] \mid a \in a, \tau(a) = Rt \} \cup \{a[i] \mid a \in a, \tau(a) = Rt, i \in i, a[i] \text{ is a term of } \varphi \}.
\]

(5.2.8)

We denote by \( L(\varphi) \) the set of all equivalence relations over the existentially quantified array variables in \( \varphi \). We lift these equivalence relations to set of all location terms and we define, for any \( e \in E \in E \):

\[
I(\varphi) ::= \{i \in i \mid a[i] \text{ is a term of } \varphi\} \cup I_{exp}(\varphi) \cup \{0 \mid a \neq \emptyset \},
\]

\[I_{exp}(\varphi) ::= \{exp \mid exp \text{ is an expression over the existential variables in } i \text{ and } \varphi \text{ contains an atomic formula that compares some } j \in j \text{ with } exp \}.\]

**Example 5.2.1.** The set of location terms in \( \psi_3 \) are \( L_{Rt}(\psi_3) = \{x, q, z\} \) and \( L_{Rt'}(\psi_3) = \{a[0], a[i_1], a[i_2]\} \). Also, \( I_{exp}(\psi_3) = \{i_1, i_1 + i_3\} \) and \( I(\psi_3) = \{0, i_1, i_2, i_1 + i_3\} \). The set of location terms in \( \psi_5 \) is \( L(\psi_5) = \{x, z\} \) and \( I(\psi_5) = \emptyset \).

### 5.2.3.3.1 Equalities between existential variables:

Let \( E_a \) be the set of all equivalence relations over the existentially quantified array variables in \( \varphi \), such that for any \( e \in E_a \) two array variables of different type are not equivalent w.r.t \( e \), i.e. for all \( a, a' \in a \), if \( \tau(a) = Rt, \tau(a') = Rt' \) and \( Rt \neq Rt' \) then \( (a, a') \notin e \).

Let \( E_i \) be the set of all equivalence relations over terms in \( I(\varphi) \). For any \( e_i \in E_i \) and \( e_a \in E_a \) let \( E_{Rt}(e_i, e_a) \) be the set of all equivalence relations \( e_{Rt} \) over the terms in \( L_{Rt}(\varphi) \) such that:

1. for any \( a \) and \( a' \) two array variables equivalent w.r.t. \( e_a \), \( a[i] \) and \( a'[i] \) are in the same equivalence class under \( e_{Rt} \), for every \( i \in i \),
2. for any \( i \) and \( i' \) two index variables, \( i \) and \( i \) are equivalent w.r.t. \( e_i \) iff \( a[i] \) and \( a[i'] \) are in the same equivalence class under \( e_{Rt} \), and
3. any two terms of the form \( a[i] \) and \( a'[i'] \) with \( (a, a') \notin e_a \) do not belong to the same equivalence class under \( e_{Rt} \); if there exist two terms \( a[i] \) and \( a'[i'] \) which are not equivalent w.r.t. \( e_{Rt} \) then \( (a, a') \notin e_a \).

We lift these equivalence relations to set of all location terms and we define, for any \( e_i \in E_i, e_a \in E_a \)

\[
E_{Rt}(e_i, e_a) ::= \{e_{Rt_1} \cup \ldots \cup e_{Rt_n} \mid e_{Rt_k} \in E_{Rt_k}(e_i, e_a), \text{ for any } 1 \leq k \leq n\}
\]

where \( Rt_1, \ldots, Rt_n \) are the record types used in \( \varphi \).
5.2. COMPOSITE STRUCTURE LOGIC. CSL

The equivalence relations in $\mathcal{E}_i(e_i, e_a)$, for any $e_i$ and $e_a$, give all the possible truth valuations of the equality predicates between existential location terms from $\varphi$. We obtain that $\varphi$ is equivalent to

$$\varphi_1 := \bigvee_{e_a \in \mathcal{E}_a, e_i \in \mathcal{E}_i, e \in \mathcal{E}_l(e_i, e_a)} \exists a \exists i \exists x \exists d \forall b \forall y \{\exists d, \forall d\}^* \cdot \phi_e \land Eq(e),$$

where

$$Eq(e) := \bigwedge_{(t,t') \in e \cup e_i} t = t' \land \bigwedge_{(t,t') \notin e \cup e_i} t \neq t' \land \bigwedge_{(a,a') \in e_a} a[0] = a'[0] \land \bigwedge_{(a,a') \notin e_a} a[0] \neq a'[0]$$

and $\phi_e$ is obtained from $\phi$ by replacing with $true$ the equalities between equivalent location terms (i.e., $t = t'$ with $(t,t') \in e$) and with $false$ the equalities between non-equivalent location terms. Notice that two arrays are different iff the terms representing their first elements are different.

Example 5.2.2. In the case of $\psi_3$, there are only two equivalence relations that lead to a model. One in which all the equivalence classes are singletons and one in which 0 and $i_1$ are in the same equivalence class. The latter implies that $a[0]$ and $a[i_1]$ are also in the same equivalence class. For all the other equivalence relations, the corresponding disjuncts are unsatisfiable.

5.2.3.3.2 Order relations over existential index expressions: For any equivalence relation over index expressions in $I(\varphi)$, $e_i \in \mathcal{E}_i$, we define $\mathcal{R}(I(\varphi)/e_i)$ to be the set of all total order relations between equivalence classes under $e_i$. Then, $\varphi$ is equivalent to

$$\varphi_2 := \bigvee_{e_a \in \mathcal{E}_a, e_i \in \mathcal{E}_i, e \in \mathcal{E}_l(e_i, e_a)} \exists a \exists i \exists x \exists d \forall b \forall y \{\exists d, \forall d\}^* \cdot \phi_{e,r} \land Eq(e) \land Ord(r),$$

where

$$Ord(r) := \bigwedge_{([t],[t']) \in r} t < t'$$

and $\phi_{e,r}$ is obtained from $\phi_e$ by replacing any comparison of the form $t < t'$, for some $t, t' \in I(\varphi)$, with $true$ if $([t],[t']) \in r$ and with $false$, otherwise.

Example 5.2.3. Let $e$, $e_a$, and $e_i$ be some equivalence relations for the formula $\psi_3$ such that the induced equivalence classes are singletons. An order relation on equivalence classes from $I(\psi_3)/e_i$ which leads to a model of $\psi_3$ is defined by $0 < i_1 < i_1 + i_3 < i_2$. Notice that the formula $\psi_5$ contains no array variables and consequently, $I(\psi_3) = \emptyset$, $\mathcal{E}_i = \emptyset$, and $\mathcal{R}(I(\psi_3)/e_i) = \emptyset$.

The existential index variables in CSL define lower bounds for the lengths of the arrays. A formula of the form $\exists i. \psi$, where $\psi$ contains $a[i]$, is satisfied by a heap graph containing an array of size at least $i$. Consequently, an expression over index variables which is constrained to be less than an existential index variable as above may also denote a position of an array. For example, in $\psi_3$, $i_1 + i_3 < i_2$ implies that $i_1 + i_3$ corresponds to a position in the array denoted by $a$ in every model of $\psi_3$ (the existence of the position...
CHAPTER 5. PROGRAM VERIFICATION USING FRAGMENTS OF GCSL

$i_2$ implies the existence of all positions less than $i_2$. In this case, we introduce a fresh existential index variable and a fresh existential location variable to explicitly denote this position. A similar case is when an index variable $i$ may denote a position on some array $a$ although it is not used explicitly in a term of the form $a[i]$. For example, if we consider the formula $\psi_4$ with $0 < i_1 < i_2 < i_3 < i_1 + i_2 + i_3$ then $i_2$ always represents a position of the array $c$. We introduce a fresh existential location variable $x^c_{i_2}$ such that $x^c_{i_2} = c[i_2]$ to make explicit the element at the position $i_2$ in the array $c$.

Now, let’s consider $\psi_4$ and $e_i, e_a, e$ some equivalence relations such that every equivalence class is a singleton. If the order between the index variables is given by $0 < i_1 < i_2 < i_3 < i_1 + i_2 + i_3$ then the expression $i_1 + i_2 + i_3$ does not represent an array position in some models of $\psi_4$ (e.g., the ones in which the size of the array $a$ is exactly the value of $i_2$ and the size of the array $c$ is exactly the value of $i_3$.)

In the following, we formally define a formula equivalent to $\varphi_2$ from (5.2.9) such that the array positions denoted by expressions over existential index variables are denoted also by fresh existential index variables.

Given $e_i \in \mathcal{E}_i$, $e_a \in \mathcal{E}_a$, $e \in \mathcal{E}_i(e_i, e_a)$, and $r \in \mathcal{R}(\mathcal{I}(\varphi)/e_i)$, let $I'_{\exp}(\varphi)$ denote the set of expressions over index variables which are smaller, w.r.t. $r$, than some existential index variable $i$ with $a[i]$ a term in $\varphi$. Formally,

\[
I'_{\exp}(\varphi) := \{ \text{exp} \in I(\varphi) \setminus i \mid \exists i \in I(\varphi) \text{ such that } ([\text{exp}], i) \in r, \text{ and } a[i] \text{ is a term in } \varphi \text{ for some } a \in a\}.
\]

For every $\text{exp} \in I'_{\exp}(\varphi)$, we introduce a fresh existentially quantified index variable $i_{\exp}$. Also, we introduce a fresh existential location variable $x_{\exp}^a$, for every array variable $a \in a$ such that $\exp$ is smaller than the greatest existential index variable, w.r.t $r$, used with $a$. The latter is denoted by $\max_{\text{a,r}}$ and it is formally defined by

\[
\max_{\text{a,r}} := \max_r \{ i \mid i \in i \text{ and } a[i] \text{ is a term of } \varphi \},
\]

(5.2.10)

where $\max_r$ denotes the maximum w.r.t. the order $r$. Notice that, for every $a \in a$ and every model $H = (G, \delta)$ of $\varphi_2$, $[0, \max_{\text{a,r}}(\varphi)] \subseteq \text{Dom}(\theta(a))$, where $\theta(a)$ is the interpretation of $a$ in $G$.

We obtain that a disjunct of $\varphi_2$

\[
\exists a \exists i \exists x \exists d \forall b \forall j \forall y \{ \exists d, \forall d \}^* \cdot \varphi_{e,r} \land \text{Eq}(\varphi) \land \text{Ord}(r)
\]

is equivalent to

\[
\exists a \exists i \exists \text{exp} \exists x_{\exp} a \exists x a \exists d \forall b \forall j \forall y \{ \exists d, \forall d \}^* \cdot \varphi_{e,r} \land \text{Eq}(\varphi) \land \text{Ord}(r) \land \text{Ext}(r),
\]

(5.2.11)

where

\[
\text{Ext}(r) := \bigwedge_{\text{exp} \in I'_{\exp}(\varphi)} i_{\exp} = \exp \land \bigwedge_{\text{exp} \in I'_{\exp}(\varphi), a \in a, \text{max}_{\text{a,r}}(\varphi) \in r} x_{\exp}^a = a[i_{\exp}] \land \bigwedge_{a \in a, (i, \max_{\text{a,r}}) \in r, a[i] \text{ is not a term of } \varphi} x_i^a = a[i],
\]

\[i_{\exp}\] is the set of variables $i_{\exp}$ in $\text{Ext}(r)$, \[x_{\exp}^a\] is the set of variables $x_{\exp}^a$ in $\text{Ext}(r)$, and \[x_i^a\] is the set of variables $x_i^a$ in $\text{Ext}(r)$.

Let $\varphi_3$ be the formula obtained from $\varphi_2$ by replacing every disjunct with the corresponding equivalent from (5.2.11).

We extend every equivalence relation $e \in \mathcal{E}_i(e_i, e_a)$, for some $e_i$ and $e_a$, to the terms $i_{\exp}, a[i_{\exp}], x_{\exp}^a$, and $x_i^a$ introduced above. Thus,
5.2. COMPOSITE STRUCTURE LOGIC. CSL

- we create a new equivalence class under \( e \) for every pair of terms \( x_i^a \) and \( a[i] \), where \( a \in a, (i, i_{\text{max}}) \in r \), and \( a[i] \) is not a term of \( \varphi \),
- we add \( i_{\text{exp}} \) to every equivalence class under \( e_i \) containing already \( \text{exp} \),
- we add \( x_{i_{\text{exp}}}^a \) and \( a[i_{\text{exp}}] \) to the equivalence class under \( e \) containing \( a[i] \), where \( i \in i \) belongs to the same equivalence class as \( \text{exp} \) (w.r.t. \( e_i \)),
- we create new equivalence classes under \( e \) for the terms \( x_{i_{\text{exp}}}^a \) and \( a[i_{\text{exp}}] \) not considered before.

Let \( \mathcal{L}'(\varphi) = \mathcal{L}(\varphi) \cup \mathcal{L}(\text{Ext}) \), where \( \mathcal{L}(\text{Ext}) \) denotes the location terms in \( \text{Ext} \). Also, for some \( e_i \in \mathcal{E}_i \) and \( e_a \in \mathcal{E}_a \), let \( \mathcal{E}_i'(e_i, e_a) \) denote the set of equivalence relations extended to terms in \( \mathcal{L}'(\varphi) \) as above.

**Example 5.2.4.** The equivalent formula obtained for \( \psi_3 \) and the order relation in Example 5.2.3 is

\[
\exists a \exists x, q, z, x_{i_1+i_3}^{i_1}, t_{i_2}, t_{i_3}, x_{i_1+i_3}. (\phi \land l_{i_1+i_3} = i_1 + i_3 \land x_{i_1+i_3} = a[l_{i_1+i_3}]),
\]

where \( \phi \) is the sub-formula of \( \psi_3 \) without the existential quantifiers. Also, \( \mathcal{L}'(\psi_3) = \mathcal{L}(\psi_3) \cup \{x_{i_1+i_3}, a[l_{i_1+i_3}]\} \). The equivalence relation over \( \mathcal{L}(\psi_3) \) is extended by adding a new equivalence class that contains \( x_{i_1+i_3} \) and \( a[l_{i_1+i_3}] \).

### 5.2.3.3 Reachability relations between existential location variables

We give a procedure that enumerates all the truth valuations for the reachability predicates over terms in \( \mathcal{L}'(\varphi) \). Notice that the reachability relation w.r.t. a set of pointer fields is a transitive relation. So, a truth valuation for the reachability predicates corresponds to a set of partial transitive relations, one for each set of pointer fields in \( \mathcal{P}\mathcal{F}^* \) of the same type. Because the models of a CSL formula contain only well-formed heap graphs, the array fields create acyclic distinct paths. Therefore, we consider transitive relations w.r.t. sets of pointer fields that contain at most one array field.

Let \( e \in \mathcal{E}_i'(e_i, e_a) \), for some \( e_i \) and \( e_a \), be some equivalence relation and \( r \in \mathcal{R}(I(\varphi)/e_i) \) some order relation over index terms. First, we consider sets of recursive pointer fields.

Let \( R_t \in \mathcal{R_T} \) be a record type and let \( F \subseteq \mathcal{P}\mathcal{F}^* \) be a set of recursive pointer fields such that \( F \) contains at most one array field and \( \tau(f) = R_t \rightarrow R_t \), for every \( f \in F \). We define \( \mathcal{R}_F(e, r, \mathcal{L}'(\varphi)) \) to be the set of all transitive relations \( R_F \) between equivalence classes of terms of type \( R_t \) in \( \mathcal{L}'(\varphi)/e \) such that:

1. \( R_F \) preserves the determinism of the pointer fields, i.e., for any \( [t], [t'], [t''] \in \mathcal{L}'(\varphi)/e \) if \( ([t], [t']) \in R_F \) and \( ([t], [t'']) \in R_F \) then either \( ([t'], [t'']) \in R_F \) or \( ([t''], [t']) \in R_F \) or \( [t'] = [t''] \), and

2. if \( \tau \) contains some array field \( a \) then \( R_F \) satisfies the order relation over terms of the form \( a[i] \) induced by \( r \), i.e., if \( ([i], [i']) \in r \) and \( \{a[i], a[i']\} \subseteq \mathcal{L}'(\varphi) \) then \( ([a[i]], [a[i']]) \in R_F \).

Next, let \( H \subseteq \mathcal{P}\mathcal{F}^* \) be a set of non-recursive pointer fields of type \( R_t \rightarrow R_t' \), where \( R_t \) and \( R_t' \) are two record types. We define \( \mathcal{R}_H(e, r, \mathcal{L}'(\varphi)) \) to be the set of all partial functions from \( \mathcal{L}'_{R_t}(\varphi)/e \) to \( \mathcal{L}'_{R_t'}(\varphi)/e \), where \( \mathcal{L}'_{R_t}(\varphi) \), resp. \( \mathcal{L}'_{R_t'}(\varphi) \), denotes the set of all location terms in \( \mathcal{L}'(\varphi) \) of type \( R_t \), resp. \( R_t' \).
We lift these transitive relations to the set of all location terms and we define, for any $e \in E_i((e_i, e_a))$ and $r \in R(I(\varphi)/e)$,

$$R(e, r, L^I(\varphi)) = \left\{ R = \bigcup_{F \subseteq PF^*} R_F \mid R_F \in R_F(e, r, L^I(\varphi)) \text{ and } R \text{ is well-defined} \right\}$$

where $R$ is well-defined if for all $F, F' \in PF^*$ and $([t], [t']) \in R_F$:

1. if $F' \subseteq F$ then $([t], [t']) \in R_{F'}$,
2. if $F \cap F' \neq \emptyset$ then $([t'], [t]) \in R_{F'}$, and
3. if $F \cap F'$ and $([t], [t']) \in R_{F'}$ then $([t], [t']) \in R_{F \cup F'}$.

The well-definedness of $R$ is implied by the semantics of the reachability predicates in CSL and, intuitively, it states that, for any pointer fields $f, g$, and $h$ of the same type, (1) if $x \record{\{f, g, h\}} \rightarrow z$ then $x \record{\{f\}} \rightarrow z$ and $x \record{\{g\}} \rightarrow z$, (2) if $x \record{\{f, g\}} \rightarrow z$ then $z \record{\{f\}} \rightarrow x$, and (3) if $x \record{\{f\}} \rightarrow z$ and $x \record{\{h\}} \rightarrow z$ then $x \record{\{g, h\}} \rightarrow z$.

### 5.2.1 Remark

The transitive relations in $R(e, r, L^I(\varphi))$ don't completely determine the truth value of the reachability predicates with length constraints. For example, if $([t_1], [t_2]) \not\in R(f)$, for some $t_1, t_2$ of record type $Rt$ and $\tau(f) = Rt \rightarrow Rt$ then $t_1 \record{\{f\}} \rightarrow t_2$ is interpreted as false but if $([t_1], [t_2]) \in R(f)$ then the value of the predicate $t_1 \record{\{f\}} \rightarrow t_2$ is still not known. The complete evaluation for these constraints is discussed later.

### Example 5.2.5

For the formula $\psi_3$ and some equivalence relations such that the induced equivalence classes are singletons, two transitive relations in $R_{\{f\}}$ are $\{(x, q), (q, z), (x, z)\}$ and $\{(x, q), (z, q), (x, z)\}$. Notice that only the first one will lead to a model of $\psi_3$. By Remark 5.2.1, for any transitive relation in $R_{\{f\}}$, the atomic formulas $x \record{\{f\}} \rightarrow q$ and $x \record{\{f\}} \rightarrow z$ cannot be evaluated.

### Example 5.2.6

For the formula $\psi_5$ and an equivalence relation over $L^I(\varphi_5) = \{x, z\}$ that distinguishes $x$ from $z$, we consider the following well-defined relation

$$R = \bigcup_{F \subseteq \{f_1, f_2, f_3, f_4\}} R_F,$$

where (1) $R_F = \{(x, z)\}$, if $F \subseteq \{f_1, f_2, f_3\}$ or $F = \{f_4\}$, (2) $R_F = \{(z, x)\}$, if $F \subseteq \{f_1, f_2, f_3\}$ or $F = \{f_4\}$, and (3) $R_F = \emptyset$, otherwise. Because $R$ is well-defined, $(x, z) \in R_{\{f_1, f_2\}}$ and $(x, z) \in R_{\{f_2, f_3\}}$ imply that $(x, z) \in R_{\{f_1, f_2, f_3\}}$.

Then, $R(e, r, L^I(\varphi))$ gives all the possible truth valuations of the reachability predicates without length constraints. For any $R \in R(e, r, L^I(\varphi))$, $\max \text{max}(\{t\}, [t'])$ is the maximum set of pointer fields $F \subseteq PF^*$ such that $([t], [t']) \in R_F$ and $\min \text{max}(\{t\}, [t'])$ is the minimum set of pointer fields $F \subseteq PF^*$ such that $([t], [t']) \not\in R_F$. It follows that $\varphi_3$ is equivalent to the formula $\varphi_4$ given by:

$$\bigvee_{e_a \in E_a} \exists a \exists i \exists x \exists a \exists x \bigvee_{e \in E_i(e_i, e_a)} \exists a \exists x \exists x \bigvee_{e \in E_i(e_i, e_a)} \exists a \exists x \exists x \bigvee_{e \in E_i(e_i, e_a)}$$

\(\wedge Eq(e) \wedge Ord(r) \wedge Ext(r) \wedge Reach(R),\)
5.2. COMPOSITE STRUCTURE LOGIC. CSL

where

\[
\text{Reach}(R) := \bigwedge_{\max R([t], [t']) = F} t \xrightarrow{F} t' \land \bigwedge_{\max R([t], [t']) = F} t \xrightarrow{\neg F} t' \land \bigwedge_{\min R([t], [t']) = F} \neg t \xrightarrow{F} t' \land \bigwedge_{F \subseteq \mathbb{PF}_r \cup \overline{\mathbb{PF}_r}} \neg t \xrightarrow{\neg F} t' \land \bigwedge_{F \cap (\mathbb{PF}_r \cup \overline{\mathbb{PF}_r}) = \emptyset} \neg t \xrightarrow{\neg F} t' \land \bigwedge_{min^{-R}([t], [t']) = F} t \xrightarrow{\neg F} t' \land \bigwedge_{F \cap (\mathbb{PF}_r \cup \overline{\mathbb{PF}_r}) = \emptyset} \neg t \xrightarrow{\neg F} t'
\]

(5.2.13)

and \(\phi_{x,R}^{\ast} \) is obtained from \(\phi_{x,R} \) by replacing (1) all atomic formulas of the form \(t \xrightarrow{F} t'\) or \(t \xrightarrow{\neg F} t'\) with true if \([t],[t']\) \(\in R_F\) and (2) all atomic formulas of the form \(t \xrightarrow{F} t'\), \(t \xrightarrow{\neg F} t'\) or \(t \xrightarrow{\neg F} t'\) with false if \([t],[t']\) \(\not\in R_F\). The atomic formulas of the form \(x \xrightarrow{F,F,B} z'\) with \(B \neq \emptyset\) will be eliminated later when we enumerate the truth valuations for the unary label predicates.

Since each \(R_F\) with \(F \subseteq \mathbb{PF}_r \cup \overline{\mathbb{PF}_r}\) denotes a deterministic reachability relation and \(R_{F'} \subseteq R_F\) whenever \(F \subseteq F'\), is enough to consider in \(\text{Reach}(R)\) only the maximal sets of pointer fields \(F\) for which \(R_F\) holds between two equivalence classes \([t]\) and \([t']\) and only the minimal sets of pointer fields \(F\) for which \(R_F\) does not hold between \([t]\) and \([t']\).

Notice that negations of atomic formulas of the form \(\neg t \xrightarrow{F} x'\) with \(F \subseteq \mathbb{PF}_r \cup \overline{\mathbb{PF}_r}\) and \(|F| \geq 2\) may impose the existence of some vertices in the heap graph which are not denoted by existential location variables. For example, the formula

\[
\exists x, x'. x \xrightarrow{\{f\}, \emptyset} x' \land x \xrightarrow{\{g\}, \emptyset} x' \land \neg x \xrightarrow{\{f,g\}, \emptyset} x'
\]

is satisfied by heap graphs which, besides the vertices denoted by \(x\) and \(x'\), contain at least one more vertex on the \(f\)-path or on the \(g\)-path between \(x\) and \(x'\). It can be easily seen that a heap graph with only two vertices denoted by \(x\) and \(x'\) and two edges between \(x\) and \(x'\), one labeled by \(f\) and one labeled by \(g\), satisfies also \(x \xrightarrow{\{f,g\}, \emptyset} x'\).

In the following, we add to \(\varphi_4\) fresh existential location variables to denote explicitly the vertices required by literals of the form \(\neg t \xrightarrow{F} x'\), where \(F \subseteq \mathbb{PF}_r \cup \overline{\mathbb{PF}_r}\) and \(|F| \geq 2\). Then, all these literals are replaced by formulas that use only reachability predicates w.r.t. a single pointer field. We use the following lemma.

**Lemma 5.2.1.** A literal \(\alpha\) of the form \(\neg x \xrightarrow{F} x'\) with \(F \subseteq \mathbb{PF}_r \cup \overline{\mathbb{PF}_r}\) and \(|F| \geq 2\) is equivalent to the formula \(\rho_\alpha\) given by:

\[
\bigvee_{f \in F} \neg x \xrightarrow{\{f\}, \emptyset} x' \lor \bigvee_{f \in F, g \in F} \exists z_1^\alpha, z_2^\alpha \exists l_1^\alpha, l_2^\alpha \left( x \xrightarrow{\{f,g\}, \emptyset} z_1^\alpha \xrightarrow{\{f\}, \emptyset} z_2^\alpha \xrightarrow{\{f\}, \emptyset} x' \land l_1^\alpha = 1 \land \land (\neg z_2^\alpha \lor (z_1^\alpha \land l_2^\alpha \geq 2)) \right),
\]

where \(x \xrightarrow{\{f\}, \emptyset} z_1^\alpha\) is a syntactic sugar for \(x \xrightarrow{\{f\}, \emptyset} z_1^\alpha \lor x = z_1^\alpha\).

**Proof.** For simplicity, let \(F = \{f,g\}\) with \(f, g \in \mathbb{PF}_r \cup \overline{\mathbb{PF}_r}\). We prove only that \(\alpha\) implies \(\rho_\alpha\) (the reverse is obvious). Let \(H = (G, \delta)\) be a model of \(\neg x \xrightarrow{\{f,g\}, \emptyset} x'\) by some valuation \(\mu\) defined by \(\mu(x) = v\) and \(\mu(x') = v'\). We consider the following cases:

111
Let $z^\alpha$ (resp. $l^\alpha$) denote the set of all location variables $z_1^\alpha$ and $z_2^\alpha$ (resp. $l_1^\alpha$ and $l_2^\alpha$) from $\rho_\alpha$, where $\alpha$ is a literal of the form $\neg t_{E^\emptyset}^{E^\emptyset} t'$ with $|F| \geq 2$ in $\text{Reach}(R)$. Notice that if $\alpha$ is a literal of $\text{Reach}(R)$ then $([t], [t']) \in R_f$, for all $f \in F$.

The set of variables $z^\alpha$ is added to $\mathcal{L}'(\varphi)$. We consider extensions of the equivalence relations $e \in \mathcal{E}(e_i, e_\alpha)$ and of the relations $R \in \mathcal{R}(e, r, \mathcal{L}'(\varphi))$ to the variables $z^\alpha$ such that the reachability constraints from the formulas $\rho_\alpha$ are satisfied. First, let $\mathcal{E}_-(e)$ be all the extensions of the equivalence relation $e \in \mathcal{E}(e_i, e_\alpha)$ over the set of variables $z^\alpha$ such that for every literal $\alpha$ of the form $\neg t_{E^\emptyset}^{E^\emptyset} t'$, $z_2^\alpha$ is not equivalent to $t$ and $z_1^\alpha$. Second, let $\mathcal{R}_-(R)$ be all the extensions of the transitive relation $R \in \mathcal{R}(e, r, \mathcal{L}'(\varphi))$ to the new equivalence classes containing variables in $z^\alpha$ such that for every literal $\alpha$ as above, the reachability constraints from at least one disjunct of $\rho_\alpha$ are satisfied (the relations in $\mathcal{R}_-(R)$ are well-defined).

It follows that $\varphi_4$ is equivalent to

$$
\varphi_4 := \bigvee_{e_\alpha \in \mathcal{E}_-(e)} \bigvee_{r \in \mathcal{R}(l(\varphi)/e_\alpha)} \bigvee_{e' \in \mathcal{E}_-(e)} \exists a \exists x' \exists d \forall b \forall y \forall \{\exists d, \forall d\}^* \cdot \phi_{e, r, R} \quad (5.2.14)
$$
5.2. COMPOSITE STRUCTURE LOGIC. CSL

\[ \land \text{Eq}(e') \land \text{Ord}(r) \land \text{Ext}(r) \land \text{Reach}_-(R'), \]

where \( i' = i \cup i_{\exp} \cup \Gamma_0 \), \( x' = x \cup \mathbf{x}_{\exp}^a \cup \mathbf{x}_i^a \cup \mathbf{z}^a \), for every \( e' \in \mathcal{E}_-(e) \),

\[ \text{Eq}(e') ::= \text{Eq}(e) \land \bigwedge_{z, z' \in \mathbf{z}} (z = z' \land \bigvee_{(z, z') \in e'} z \neq z'), \quad (5.2.15) \]

and, for every \( R' \in \mathcal{R}_-(R) \),

\[ \text{Reach}_-(R') ::= \bigwedge_{\text{max}R'([t], [t']) = F} t \xrightarrow{\ell F_0} t' \land \bigwedge_{\text{min}R'([t], [t']) = F} t \xrightarrow{\ell F_0} t' \land \bigwedge_{\ell \notin \mathbf{P}_F (t) \notin \mathbf{R}_F} \neg t \xrightarrow{\ell F_0} t' \land \bigwedge_{\text{Length}(\rho_\alpha)} \quad (5.2.16) \]

The formula \( \text{Length}(\rho_\alpha) \) is obtained from \( \rho_\alpha \) by substituting every atomic formula of the form \( x \xrightarrow{\ell F} x' \) with \textit{true} or \textit{false} according to the relation \( R' \in \mathcal{R}_-(R) \).

5.2.3.4 Unary label predicates: Given an equivalence relation \( e' \in \mathcal{E}_-(e) \), for some \( e \in \mathcal{E}_i'(c_i, c_a) \), the evaluations of the label predicates are all the possible mappings \( \mathcal{B}(e') \) from \( \mathcal{P}_\text{Var} \) to \( \mathcal{L}'(\varphi)/_{e'} \). Then, every disjunct of \( \varphi_4' \) is equivalent to

\[ \bigvee_{b \in \mathcal{B}(e')} \exists a \exists e' \exists d \forall y \forall \{d, \forall d\}^*, \phi_{e', R, b} \land \text{Ord}(r) \land \text{Ext}(r) \land \text{Reach}_-(R') \land \text{Lab}(b), \quad (5.2.17) \]

where \( \phi_{e, R, b} \) is obtained from \( \phi_{e, R, b} \) by replacing (1) every atomic formula of the form \( \ell(t) \) with \textit{true} if \( b(\ell) = [t] \) and \textit{false}, otherwise, (2) every atomic formula of the form \( x \xrightarrow{\ell F} x' \) or \( x \xrightarrow{\ell F, b} x' \) with \textit{false} if \( ([x], [x']) \notin \mathbf{R}_F \) or there is some \( \ell \in \mathbf{B} \) and a location variable \( z \) such that \( ([x], [z]) \in \mathbf{R}_F \), \( ([z], [x']) \in \mathbf{R}_F \), and \( b(\ell) = [z] \), and (3) every atomic formula of the form \( x \xrightarrow{\ell F, b} x' \) is replaced with \textit{true} if \( ([x], [x']) \in \mathbf{R}_F \) and for any \( \ell \in \mathbf{B} \), there exist no location variable \( z \) such that \( ([x], [z]) \in \mathbf{R}_F \), \( ([z], [x']) \in \mathbf{R}_F \), and \( b(\ell) = [z] \).

Also,

\[ \text{Lab}(b) ::= \bigwedge_{[\ell] \in \mathcal{L}'(\varphi)/_{e'}} \ell(t) \land \bigwedge_{[\ell] \in \mathcal{L}'(\varphi)/_{e'}} \neg \ell(t) \quad (5.2.18) \]

Let \( \varphi_5 \) be the formula obtained from \( \varphi_4' \) by replacing every disjunct with the corresponding equivalent formula from \( \text{Lab}(b) \).

5.2.3.5 Length constraints: Finally, we enumerate all the possibilities of satisfying the length constraints. By length constraints, we understand: (1) linear constraints over the values of the index variables and (2) constraints that associate index variables to paths in the graph: \( x \xrightarrow{\ell F} x' \) saying that \( i \) denotes the length of the path, whose edges are labeled by the pointer fields \( F \), between the vertices denoted by \( x \) and \( x' \).
Notice that \( \varphi_5 \) contains linear constraints over index variables only in the sub-formulas denoted by \( \phi_{e,r,R,b} \) and \( \text{Reach}_\sim(R') \). Thus, let \( \text{atomsI}(\phi_{e,r,R,b} \land \text{Reach}_\sim(R')) \) denote the linear constraints in \( \phi_{e,r,R,b} \land \text{Reach}_\sim(R') \) built over existential index variables. Also, let \( \Pi_{\text{Ind}}(\phi_{e,r,R,b} \land \text{Reach}_\sim(R')) \) be the set of all possible truth valuations \( \pi : \text{atomsI}(\phi_{e,r,R,b} \land \text{Reach}_\sim(R')) \rightarrow \{ \text{true}, \text{false} \} \). We obtain that every disjunct of \( \varphi_5 \) is equivalent with:

\[
\bigvee_{\pi \in \Pi_{\text{Ind}}(\phi_{e,r,R,b} \land \text{Reach}_\sim(R'))} \exists a \exists b \exists c \exists d \forall x \forall y \{ \exists d, \forall d \}^{\ast} \left( \phi_{e,r,R,b,\pi} \land \text{Reach}_\sim(R') \right) \tag{5.2.19}
\]

where \( \phi_{e,r,R,b,\pi} \) (resp. \( \text{Reach}_\sim(R') \)) is the formula \( \phi_{e,r,R,b} \) (resp. \( \text{Reach}_\sim(R') \)) where every atomic formula from \( \text{atomsI}(\phi_{e,r,R,b} \land \text{Reach}_\sim(R')) \) is replaced by the truth value given by \( \pi \), and

\[
\text{Ind}(\pi) := \bigwedge_{\pi(\alpha) = \text{true}} \alpha.
\]

Let \( \varphi_6 \) be the formula obtained from \( \varphi_5 \) by replacing any of its disjuncts with the equivalent formula from (5.2.19).

Notice that \( \varphi_6 \) contains atomic formulas of the form \( x^{\mathcal{F},B,i} \Rightarrow x' \) with \( x, x' \in \mathcal{X} \), \( \mathcal{F} \subseteq \mathcal{P}\mathcal{F}^* \), and \( i \in \mathcal{I} \) only in the sub-formulas denoted by \( \phi_{e,r,R,b,\pi} \) and \( \text{Reach}_\sim(R') \).

Let \( \text{atomsR}(\phi_{e,r,R,b,\pi} \land \text{Reach}_\sim(R')) \) be the set of such atomic formulas in \( \phi_{e,r,R,b,\pi} \land \text{Reach}_\sim(R') \). By construction, if \( x^{\mathcal{F},B,i} \Rightarrow x' \) is an atomic formula in \( \phi_{e,r,R,b,\pi} \land \text{Reach}_\sim(R') \) then the relation \( R' \) and the valuation \( b \) imply that there exists an \( F \)-path between the vertices denoted by \( x \) and \( x' \) which does not pass through vertices labeled by pointer variables in \( B \) (i.e., \( ([x],[x']) \in R'_F \) and for any \( \ell \in B \), there exist no location variable \( z \) such that \( ([x],[z]) \in R'_F \), \( ([z],[x']) \in R'_F \), and \( b(\ell) = [z] \)).

Let \( \Pi_{\text{Loc}}(\phi_{e,r,R,b,\pi} \land \text{Reach}_\sim(R')) \) be the set of all mappings \( \pi' : \text{atomsR}(\phi_{e,r,R,b,\pi} \land \text{Reach}_\sim(R')) \rightarrow \{ \text{true}, \text{false} \} \).

**Example 5.2.7.** For the formula \( \psi_3 \), \( \text{atomsI}(\psi_3) = \{ i_1 + i_3 < i_2, i_1 + i_2 \geq 8 \} \) and \( \text{atomsR}(\psi_3) = \{ x^{f_1} \Rightarrow x, x^{f_1} \Rightarrow q \} \). For the formula \( \psi_5 \), \( \text{atomsI}(\psi_5) = \{ i + i' \geq 4 \} \) and \( \text{atomsR}(\psi_5) = \{ x^{f_1,f_2} \Rightarrow x, x^{f_1,f_2} \Rightarrow z, x^{f_1,f_2} \Rightarrow x', x^{f_1,f_2} \Rightarrow x'' \} \). In order to prove the satisfiability of these formulas, one has to consider valuations \( \pi \) and \( \pi' \) that map all these atomic formulas to true.

The truth valuations considered above may induce new constraints on the existential index variables. Intuitively, (1) a valuation that maps to true the atomic formulas \( x^{f_1} \Rightarrow x' \) and \( x^{f_1} \Rightarrow x'' \) implies that \( i_1 = i_2 \) and (2) a valuation that maps to true the atomic formulas \( x^{f_1} \Rightarrow x' \), \( x^{f_1} \Rightarrow x'' \), \( x^{f_1} \Rightarrow x'' \), and \( x^{f_1} \Rightarrow x'' \) implies that \( i_2 = i_1 + l \) and \( l > 0 \), where \( l \) is the length of the \( f \)-path from \( x' \) to \( x'' \). To discover all these constraints we define a symbolic representation for the truth valuations considered until now, called template.

**Definition 5.2.4 (Template).** A template for a formula \( \varphi \) is a pair \( (T_G, T_I) \), where \( T_G = (V, E, L_V, L_E) \) is a labeled directed multi-graph and \( T_I \) is a conjunction of linear constraints over index variables such that:
5.2. COMPOSITE STRUCTURE LOGIC. CSL

- \( V \) is a set of typed vertices, \( E \) is a multi-set of edges over \( V \),

- \( L_V : V \rightarrow 2^{L'(\varphi)} \times 2^{PVar} \) is a labeling of vertices with sets of terms from \( L'(\varphi) \) and sets of labels in \( PVar \) such that \( L_V(v_1) |_1 \cap L_V(v_2) |_1 = \emptyset \) and \( L_V(v_1) |_2 \cap L_V(v_2) |_2 = \emptyset \), for any two different vertices \( v_1 \neq v_2 \),

- \( L_E : E \rightarrow 2^{PF} \times I_T, \) with \( I_T \) index variables in \( \text{Ind} \), is a labeling of edges with sets of pointer fields and index variables. We require that \( T \) is deterministic, every edge satisfies the typing constraints, and any two different edges are labeled by different index variables.

Given a formula \( \varphi \) and given \( e_i \in E_i, e_a \in E_a, e \in E(e_i, e_a), r \in R(I(\varphi)/e_a) \), \( R \in R(e, r, L'(\varphi)) \), \( e' \in E_e(e) \), \( R' \in R_e(R) \), \( b \in B(e') \), \( \pi \in \Pi_{\text{Ind}} \), and \( \pi' \in \Pi_{\text{Len}} \), we define the template \( T_{e', e, r, b, \pi, \pi'} = (V, E, L_V, L_E) \) as follows:

1. for every equivalence class \([t]\) of \( e'\) over \( L'(\varphi) \), \( V \) contains a distinct vertex denoted \( v_t \) which is labeled by all the terms in \([t]\), i.e.,
   \[
   V = \{ v_t \mid [t] \in L'(\varphi)/e' \} \text{ and } L_V(v_t) = \left\{ \{ t' \mid t' \in [t] \}, \{ \ell \in PVar \mid b(\ell) = [t] \} \right\};
   \]

2. for each pointer field type \( R_t \rightarrow R_{t'} \) (\( R_t \) may be equal to \( R_{t'} \)), we consider sets of pointer fields of type \( R_t \rightarrow R_{t'} \) in a top-down manner, starting from the largest one, i.e., \( \{ f \in PF^* \mid \pi(f) = R_t \rightarrow R_{t'} \} \), until we reach sets of size 1. For each set of pointer fields \( F \), we add edges corresponding to the transitive relation \( R_k^F \). Formally, for every \(([t], [t']) \in R_k^F \) let \( v_t \), resp. \( v_{t'} \), denote the vertices in \( V \) corresponding to the equivalence classes \([t]\), resp. \([t']\). If the graph built so far does not contain a path between \( v_t \) and \( v_{t'} \) whose edges are labeled by \( F \), we add a new edge \( e \) and we label it with \((F, l_e)\), where \( l_e \) is a fresh index variable. Let \( I_{TP} \) be the set of all index variables introduced in this way;

3. \( I_{TP} \) is a conjunction of linear constraints that includes \( \text{Ind}(\pi) \) and that relates the index variables labeling the edges of the template with the index variables that represent lengths of paths according to \( \pi' \). More precisely, \( \mathcal{I}_T = \text{Ind}(\pi) \land \text{Len}(\pi_{R_k^F}) \) is the conjunction of:

   - \( l = l_1 + \ldots + l_n \), for every \( x \xrightarrow{F, B, l} \rightarrow z \in \text{atomsR}(e, e, R, b, \pi \land \text{Reach}_\pi(R')) \) such that \( \pi'(x \xrightarrow{F, B, l} \rightarrow z) = \text{true} \) and there exists a path \([v_0, \ldots, v_n] \) in \( T_G \) with \( x \in L_V(v_0) |_1, \; z \in L_V(v_n) |_1, \) and, for every \( 1 \leq k \leq n \), either \( F \subseteq L_E(v_{k-1}, v_k) |_1 \) and \( L_E(v_{k-1}, v_k) |_2 = l_k^F \) or \( F \subseteq L_E(v_k, v_{k-1}) |_1 \) and \( L_E(v_k, v_{k-1}) |_2 = l_k^R \). Intuitively, the path \([v_0, \ldots, v_n] \) contains edges labeled by all the pointer fields in \( F \) and by some index variable \( l_k^F \). This path is unique because \( R_k' \) defines deterministic pointer fields.

   - \( l \neq l_1 + \ldots + l_n \) for every \( x \xrightarrow{F, B, l} \rightarrow z \in \text{atomsR}(e, e, R, b, \pi \land \text{Reach}_\pi(R')) \) such that \( \pi'(x \xrightarrow{F, B, l} \rightarrow z) = \text{false} \), \(([x], [z]) \in R_k^F \), and there exists a path \([v_0, \ldots, v_n] \) in \( T_G \) as in the previous case.

   - \( l_T = 1 \), for every index variable \( l_t \) such that there exists an edge \( e \in E \) with \( L_E(e) = (F, l_T) \), where \( F \) is a set of non-recursive pointer fields.

   - \( l_T \geq 1 \), for every \( l_T \in I_{TP} \);
Example 5.2.8. Consider the formula $\psi_3$, some equivalence relations such that the induced equivalence classes are singletons, the order on index variables $0 < i_1 < i_2 < i_3$ and $R_{\{f\}} = \{(x,q),(q,z),(x,z)\}$. Notice that $R_{\pi}(R) = R$. Also, let $\pi$ and $\pi'$ be some valuations as above which assign true to any atomic formula in $\text{atomsI}(\psi_3)$ and $\text{atomsR}(\psi_3)$. The template corresponding to all these valuations is given in Figure 5.2. Since the formula does not use labels in $\text{PVar}$, vertices are labeled only by location terms.

Example 5.2.9. Consider the formula $\psi_4$, some equivalence relations such that the induced equivalence classes are singletons, and the order relation $0 < i_1 < i_2 < i_3 < i_1 + i_2 + i_3$. Figure 5.3(a) pictures the template corresponding to these valuations. Notice that we have added a vertex on the array denoted by $c$ corresponding to $c[i_2]$.

Example 5.2.10. Consider the formula $\psi_5$, some equivalence relations such that the induced equivalence classes are singletons, and the transitive relation $R$ defined in 5.2.6. Notice that $R_{\pi}(R) = R$. Also, let $\pi$ and $\pi_R$ be some valuations as above which assign true to any atomic formula in $\text{atomsI}(\psi_5)$ and $\text{atomsR}(\psi_5)$. The template representing these valuations is given in Figure 5.3(b). As in the previous case, the formula does not use labels in $\text{PVar}$ and the vertices are labeled only by location terms.

Figure 5.2: A template in $\mathcal{T}_{\psi_3,3}$

Figure 5.3: Templates from Example 5.2.10
5.2. COMPOSITE STRUCTURE LOGIC. CSL

From the definitions given above, we obtain that

\[ \varphi \text{ is equivalent with } \bigvee_{T \in \mathcal{T}_{\varphi, \exists}} \psi_T, \text{ where } \quad (5.2.20) \]

where \( I_T = \text{Ind}(\pi) \land \text{Len}(\pi') \) is the second component of the template \( T \),

\[ \rho_T = Eq(e') \land \text{Ord}(r) \land \text{Ext}(r) \land \text{Reach}^1_{\pi, \exists}(R') \land \text{Lab}(b), \]

and \( \text{Reach}^1_{\pi, \exists}(R') \) is obtained from \( \text{Reach}_{\pi, \exists}(R') \) by replacing every literal \( \ell_{F,B} \rightarrow t \) with \( t_{F,B} \rightarrow t' \), for any \( t, t' \in \mathcal{L}(\varphi) \) such that there is an edge \((v, v') \) in \( T \) with \( L_E(e) = (F, \ell_{F,B}) \), \( t \in L_V(v) \mid 1 \), and \( t' \in L_V(v') \mid 1 \). The formula \( \phi_{e,r,R,b,\pi,\pi'} \) is obtained from \( \phi_{e,r,R,b,\pi} \) by replacing \( x_{A,B} \rightarrow x' \) either with true if \( \pi'(x_{A,B} \rightarrow x') = \text{true} \) or with false, otherwise.

5.2.3.4 Small model property

By (5.2.20), for every model \( H \) of the formula \( \varphi \), there is a template \( T \in \mathcal{T}_{\varphi, \exists} \) such that \( H \models \psi_T \). So the satisfiability of \( \varphi \) is reduced to the satisfiability of \( \psi_T \), for some template \( T \in \mathcal{T}_{\varphi, \exists} \).

To decide the satisfiability of \( \psi_T \), we use a small model property: for every model of \( \psi_T \), there exists a smaller model whose size is bounded by some \( \alpha \). In general \( \alpha \) does not depend only on the number of existential quantified variables in \( \psi_T \). In the following, we show how to compute \( \alpha \) and the small models for \( \psi_T \).

From the semantics of the existential quantifier and from the construction of the template, it is straightforward that \( \alpha \) is greater than the number of vertices in the template. The paths in the template represent all the paths imposed by the formula \( \psi_T \) through existentially quantified variables. Concerning the lengths of these paths, we know that they satisfy the constraints in \( I_T \) but we don’t have some concrete values for them. To define \( \alpha \), we compute the minimal values for the index variables denoting the lengths of these paths. In fact, it is sufficient to compute minimal values only for the lengths of the paths represented by edges in the template.

Then, for every minimal solution \( m \) for the lengths of the paths denoted by edges in the template, \( \alpha \) equals the number of vertices in the template (determined by the number of existentially quantified variables) plus the sum of the values in \( m \).

The set of small models corresponding to \( m \) are obtained by adding to the template as many vertices as imposed by the values in \( m \).

5.2.3.4.1 Computing the bounds on the small models

Given a template \( T \in \mathcal{T}_{\varphi, \exists} \) we compute a finite set of solutions, denoted \( M_T \), for the index constraints in \( I_T \).

A solution for \( I_T \) is a function from index variables to integers such that the integers associated to the index variables satisfy \( I_T \). Let \( \text{Sol}(I_T) \) be the set of solutions for \( I_T \). Given the set of index variables \( I_T \) labeling edges of \( T \), we define the order relation \( \preceq_T \) on the solutions for \( I_T \) by \( s_1 \preceq_T s_2 \) iff for any \( l_T \in I_T \), \( s_1(l_T) \leq s_2(l_T) \).

Definition 5.2.5 (Minimal solutions). Given a template \( T \in \mathcal{T}_{\varphi, \exists} \) the set of minimal solutions \( M_T \subseteq \text{Sol}(I_T) \) for \( I_T \) w.r.t. \( I_T \) is the set of solutions such that
• for any \( s \in \text{Sol}(I_T) \), there exists \( m \in M_T \) such that \( m \preceq_T s \) and

• for any \( m_1 \) and \( m_2 \) in \( M_T \), \( m_1 \preceq_T m_2 \) and \( m_2 \preceq_T m_1 \).

They are also called Pareto minimal solutions.

To compute minimal values for the paths described by edges in the template, we compute the set of minimal solutions for \( I_T \) w.r.t. the index variables \( I_T \). Any model \( H \) of \( \psi_T \) defines a solution \( s_H \) for \( I_T \). Then, there exists some \( m_H \in M_T \) such that \( m_H \preceq s_H \).

We prove that for any model \( H \) of \( \psi_T \) there exists a smaller model \( H' \) where any path denoted by an edge in the template (labeled by \( l_T \)) contains exactly \( m_H(l_T) - 1 \) vertices.

We need to consider all the minimal solutions because in general two models of \( \psi_T \) might correspond to two incomparable minimal solutions \( m_{H_1} \) and \( m_{H_2} \) from \( M_T \).

The set \( M_T \) is computed using a multi-objective integer linear program (MOILP, for short) of the following form:

\[
\text{minimize } \{l \mid l \in I_T\} \text{ subject to } I_T.
\]

**Example 5.2.11.** For the formula \( \psi_5 \) and the constraint \( I_T \) from Figure 5.3(b), there are four minimal solutions w.r.t. \( I_T \) and \( l' \) (every vector represents a valuation for \( (l', l', i, l') \)): \( M_T = \{m_1 = (4, 1, 4, 1), m_2 = (3, 2, 3, 2), m_3 = (2, 3, 3, 3), m_4 = (1, 4, 1, 4, 4)\} \).

**Example 5.2.12.** For the formula \( \psi_3 \) and the constraints \( I_T \) from Figure 5.2, there are three minimal solutions w.r.t \( l_1, \ldots, l_5 \) (every vector represents only the values for \( (l_1, l_2, l_3, l_4, l_5) \)): \( M_T = \{m_1 = (1, 6, 1, 1, 5), m_2 = (2, 4, 2, 1, 4), m_3 = (3, 2, 3, 1, 3)\} \). The values of the other index variables are defined uniquely by the values of \( l_1, l_2, l_3, l_4, l_5 \).

Given \( T \in T_{\varphi, \exists} \) and \( m \in M_T \), we define a procedure, called template expanding, that constructs a set of templates \( T_{\varphi, T, m} \) in two steps as follows:

• **Step 1**, detailed in Section 5.2.3.4.2, consists in replacing each edge in \( T \) by a path of some length given by \( m \). For this, new vertices are introduced, each of them being labeled with a fresh location variable. Also, each new edge is labeled with a fresh index variable. Let \( x_m \) and \( i_m \) be the sets of these new location and index variables, respectively;

• **Step 2**, detailed in Section 5.2.3.4.3, consists in guessing labels from \( P\text{Var} \) for each new vertex and guessing additional edges between the new vertices or between the new and the old vertices, while preserving determinism.

For any \( \psi_T \) with \( T \in T_{\varphi, \exists} \), the proof of the small model property has the following scheme:

• first, Lemma 5.2.2 proves that \( \psi_T \) is equivalent to \( \bigvee_{m \in M_T} \bigvee_{T' \in T_{\varphi, T, m}} \psi_{\varphi, T', m} \)

where \( \psi_{\varphi, T', m} \) is a obtained from \( \psi_T \) by extending the relations represented by the template \( T \) over the new vertices. We denote by \( T_{\varphi, T, m} \) the set of templates obtained in this way;

• then, Lemma 5.2.3 proves that any satisfiable formula \( \psi_{\varphi, T', m} \) has a model whose size is given by the number of existentially quantified location variables.
5.2. COMPOSITE STRUCTURE LOGIC. CSL

5.2.3.4.2 First step of template expanding Let \( T = (V, E, L_V, L_E, \mathcal{I}_T) \) be a template in \( \mathcal{T}_\phi \) and \( m \in \mathcal{M}_T \) a minimal solution for \( \mathcal{I}_T \) w.r.t. \( \mathcal{I}_T \). We expand the graph of \( T \) by adding the vertices imposed by \( m \), and define a new template \( \mathcal{T}_m = ((V_m, E_m, L_{E_m}, L_{V_m}), \mathcal{I}_{\mathcal{T}_m}) \). Roughly, to build \( \mathcal{T}_m \) we replace each edge of \( T \) labeled by some index variable \( l \) with a path that has the same start and end point and passes through \( m(l) - 1 \) new vertices. Then, we define the formulas \( \rho_{\mathcal{T}_m} \) and \( \mathcal{I}_{\mathcal{T}_m} \) from \( \rho_T \) and \( \mathcal{I}_T \) by adding constraints on the length of the new edges.

Formally, let \( \epsilon \) be an edge in \( E \), with \( L_E(\epsilon) = (F, l) \) and \( v, v' \in V \). If \( \tau(v) \neq \tau(v') \) then \( m(l) = 1 \) and, consequently, no vertex should be added on this path. This happens because not all the pointer fields in \( F \) are recursive. In the following, we consider edges \( \epsilon = (v, v') \in E \) with \( \tau(v) = \tau(v') \). Then,

- we add \( m(l) - 1 \) new vertices to \( V_m \), denoted \( v_m^\epsilon \), such that for every \( v_m^\epsilon \in v_m^\epsilon \)

\[ \tau(v) = \tau(v'), \] for every \( 1 \leq i \leq m(l) - 1 \). We label each vertex \( v_m^\epsilon \) by a unique variable denoted \( x_m^\epsilon \) such that \( \tau(x_m^\epsilon) = \tau(v) \).

- if \( L_E(\epsilon) \cap \mathcal{AF} = \{ a \} \) then let \( a[i] \in L_V(v) \) and \( a[i'] \in L_V(v') \). We add fresh index variables \( i_m^\epsilon \) such that every \( v_m^\epsilon \) is labeled also by an unique \( a[i_m^\epsilon] \) with \( i_m^\epsilon \in i_m^\epsilon \);

- we replace \( \epsilon \) by a path of length \( m(l) \), formed of \( \epsilon_0 = (v, v_m^{\epsilon_0}) \), \( \epsilon_k = (v_m^{\epsilon_k-1}, v_m^{\epsilon_k}) \), and \( \epsilon_m(l-1) = (v_m^{\epsilon_m(l-1)}, v') \), with \( 2 \leq k \leq m(l) - 2 \), such that \( L_E(m(l)) = (F, l_m^\epsilon) \) where \( l_m^\epsilon \) is a fresh index variable for every \( 0 \leq p \leq m(l) - 1 \);

For every graph \( \mathcal{T}_m \) obtained by expanding \( T \in \mathcal{T}_\phi \) w.r.t \( m \in \mathcal{M}_T \) we must redefine the index constraints from \( T \) to integrate the constraints on the new edges. Therefore, we define \( \rho_{\mathcal{T}_m} = \rho_T[\gamma] \) and \( \mathcal{I}_{\mathcal{T}_m} = \mathcal{I}_T \land \phi_m \) where:

- \( \gamma \) replaces every atomic formula \( x^{F, \emptyset, l} \rightarrow z \) in \( \rho_T \) by

\[ x^{F, \emptyset, l_m^\epsilon} \rightarrow x_m^{\epsilon_0} \land \bigwedge_{1 \leq i < k} x_m^{\epsilon_{i-1}} F, \emptyset, l_m^{i-1} \rightarrow x_m^{\epsilon_i} \land x_m^{\epsilon_k} F, \emptyset, l_m^q \rightarrow z \]

where \( x \in L_V(v_x) \) and \( z \in L_V(v_z) \) with \( v_x, v_z \in V \) such that there is an edge \( \epsilon = (v_x, v_z) \in E \) with \( L_E(\epsilon) = (F, l) \) and every \( x_m^{\epsilon_i} \) labels the vertex \( v_m^{\epsilon_i} \) in \( V_m \setminus V \) for every \( 1 \leq i \leq q \) and \( q = m(l) - 1 \).

If \( F \cap \mathcal{AF} = \{ a \} \) then \( \rho_{\mathcal{T}_m} \) has a conjunct of the following form:

\[ x_m^{\epsilon_k} = a[i_m^\epsilon], \] for every vertex \( 1 \leq k \leq m(l) - 1 \).

- \( \phi_m \) contains for every \( x^{F, \emptyset, l} \rightarrow z \) in \( \rho_T \) such that \( x \in L_V(v_x) \) and \( z \in L_V(v_z) \) with \( v_x, v_z \in V \) and there is an edge \( \epsilon = (v_x, v_z) \in E \) with \( L_E(\epsilon) = (F, l) \), an atomic formula of the form

\[ l = l_m^1 + \ldots + l_m^q, \]

where \( q = m(l) - 1 \), \( x_m^{\epsilon_k} \) denotes the location variable labeling the vertices introduced between \( v_x \) and \( v_z \) corresponding to the edge \( \epsilon \), and \( l_m^j \) denotes the index variables labeling a new edge on the new path corresponding to \( \epsilon \), for every \( 1 \leq k \leq m(l) - 1 \).
Also, if \( F \cap \mathcal{A}F = \{a\} \) let \( I_m = \{i^k_m \mid 1 \leq k \leq m(l) - 1\} \) such that if \( x^k_m \in L_V(v) \) then \( a[x^k_m] \in L_V(v) \). Then, \( \phi_m \) contains also the atomic formulas:

\[
i^k_m = l^1_m + \ldots + l^k_m, \text{ for every } 1 \leq k \leq m(l) - 1.\]

**Example 5.2.13.** The expanded templates from Figure 5.4 are defined for the formula \( \psi_5 \), the template \( T \) from Figure 5.3(b) and all the minimal solutions from Example 5.2.11.

(a) The template \( T^1_m \) corresponding to \( m = (4, 1, 4, 1, 1) \), where \( m(l') = 4 \) and \( m(l) = 1 \).

(b) The template \( T^2_m \) corresponding to \( m = (3, 2, 3, 2, 2) \), where \( m(l') = 3 \) and \( m(l) = 2 \).

(c) The template \( T^3_m \) corresponding to \( m = (2, 3, 2, 3, 3) \), where \( m(l') = 2 \) and \( m(l) = 3 \).

(d) The template \( T^4_m \) corresponding to \( m = (1, 4, 1, 4, 4) \), where \( m(l') = 1 \) and \( m(l) = 4 \).

Figure 5.4: Expanded templates for the formula \( \psi_5 \).

**Example 5.2.14.** In Figure 5.5, we give an example for the expanding of the template \( T \) from Figure 5.3 corresponding to \( \psi_3 \) with respect to the minimal solution \( m \) with \( m(l_1) = 2 \), \( m(l_2) = 4 \), \( m(l_3) = 2 \), \( m(l_4) = 1 \), \( m(l_5) = 4 \). This is the only template that leads to a
5.2. COMPOSITE STRUCTURE LOGIC. CSL

model for $\psi_3$, because the array must be strictly sorted between position $i_1$ and $l_{i_1+i_3}$ with values between 0 and 1.

\[
\begin{array}{cccccccccc}
x & \{\{a\}, \ell_m^5\} & x_m^5 & \{\{a\}, \ell_m^6\} & x_m^6 \rightarrow & \{\{a\}, \ell_m^{10}\} & x_m^7 \rightarrow & \{\{a\}, \ell_m^{11}\} & z \\
\{\{a\}, \ell_m^1\} & \{\{a\}, \ell_m^2\} & \{\{a\}, \ell_m^3\} & \{\{a\}, \ell_m^4\} & \{\{a\}, \ell_m^5\} & \{\{a\}, \ell_m^6\} & \{\{a\}, \ell_m^7\} & \{\{a\}, \ell_m^8\} & \{\{a\}, \ell_m^9\} & \{\{a\}, \ell_m^{10}\} & \{\{a\}, \ell_m^{11}\} & \{\{a\}, \ell_m^{12}\} & \{\{a\}, \ell_m^{13}\}
\end{array}
\]

\[
i_1 + i_2 \geq 8 \land i_3 < i_2 \land l_{i_1+i_3} = i_1 + i_3 \land
\]

\[
I_m :=
\begin{align*}
l_1 &= i_1 \land l_2 = i_2 - i_1 \land \\
l_m^7 + l_m^9 &= l_1 \land l_m^3 + l_m^4 + i_m^8 &= l_2 \land \\
l_1 + l_m^3 + l_m^4 + i_m^8 &= l_2 \land \\
\land_{i_1 \leq i_3} l_m^9 \geq 1
\end{align*}
\]

Figure 5.5: An expanded template $T_m$ for the formula $\psi_3$ and $m = (2,4,2,1,4)$

Let us consider the sub-graph in Figure 5.5 that consists only in the path labeled by $f$. This is actually the expanded template corresponding to the formula $\psi_2$, given in the overview (Section 5.2.3.1), for the template $x \overset{f_1 \land l_1 \rightarrow}{\rightarrow} q \overset{f_2 \land l_2 \rightarrow}{\rightarrow} z$ and the minimal solution $m = (2,6,4)$.

5.2.3.4 Second step of template expanding We extend the equivalence relations $e_i \in E_i, e_a \in E_a, e \in E_i(e_i, e_a), e' \in E_-(e)$ and the relations $r \in R(I(\varphi)/e), R \in R(e, r, L'(\varphi)), R' \in R_-(R)$ represented by $T$ to the new location variables introduced in the formula $\rho_m$.

Let

\[
L_m(\varphi) = \{x_m^\epsilon, a[i_m^\epsilon] | x_m^\epsilon \in X_m, i_m^\epsilon \in I_m, \epsilon \in E \text{ and } a[i_m^\epsilon] \text{ is a term of } \rho_m\}
\]

be the set of new location terms labeling the vertices in $T_m$. Also, let $L''(\varphi) = L'(\varphi) \cup L_m(\varphi)$.

Let $E(m, T, L''(\varphi))$ denote the set of equivalence relations over $L''(\varphi)$ such that every $e_m \in E(m, T, L''(\varphi))$ has the following properties:

- $e' \subseteq e_m$ and $e_m \setminus e'$ contains only pairs of terms from $L_m(\varphi)$;
- any two location terms of different type are not equivalent;
- $(x_m^\epsilon, a[i_m^\epsilon]) \in e_m$ for every $x_m^\epsilon, a[i_m^\epsilon]$ from $L_m$ such that there is a vertex in $V_m$ labeled by these two terms;
- $(t_m, t_m') \notin e_m$ for any terms $t_m, t_m' \in L_m(\varphi)$ such that there are two vertices $v_m$ and $v_m'$ in $V_m$ with $t_m \in L_{V_m}(v_m)$ and $t_m' \in L_{V_m}(v_m')$, and there is a path in $T_m$ between $v_m$ and $v_m'$ following some set of pointer fields $F \subseteq PF^*$.

Let $e_m \in E(m, T, L''(\varphi))$ be an equivalence relation as above. We extend the transitive relation $R'$ represented by $T$ over the terms in $L_m$ as follows.
CHAPTER 5. PROGRAM VERIFICATION USING FRAGMENTS OF GCSL

First, we consider recursive pointer fields. Let \( Rt \in \mathcal{RT} \) be a record type and \( F \subseteq \mathcal{PF}^* \) a set of recursive pointer fields such that \( F \) contains at most one array field and \( \tau(f) = Rt \rightarrow Rt \), for every \( f \in F \). We define \( \mathcal{R}^F_m(R_t, \mathcal{L}''(\varphi)) \) to be the set of all transitive relations that extend \( R'_F \) to the equivalence classes in \( \mathcal{L}''(\varphi)/e_m \) of record type \( Rt \) such that every \( R^F_m \in \mathcal{R}_m^F(R_t, \mathcal{L}''(\varphi)) \) has the following properties:

- for every \( t, t' \in \mathcal{L}''(\varphi) \), \( ([t], [t']) \in R'_F \) iff \( ([t], [t']) \in R^F_m \) and
- \( R^F_m \) preserves the determinism of the pointer fields.

Next, for every set of non-recursive pointer fields \( H \subseteq \mathcal{PF}^* \) of type \( Rt \rightarrow Rt' \), where \( Rt \) and \( Rt' \) are two record types, we define \( \mathcal{R}^H_m(R_t, \mathcal{L}''(\varphi)) \) to be the set of all partial functions from \( \mathcal{L}''_R(\varphi)/e_m \) to \( \mathcal{L}''(\varphi)/e_m \) that extend \( R'_H \) to the terms in \( \mathcal{L}_m(\varphi) \) such that every \( R^H_m \in \mathcal{R}_m^H(R_t, \mathcal{L}''(\varphi)) \) has the property that:

- for every \( t, t' \in \mathcal{L}''(\varphi) \), \( R^H_m([t]) = [t'] \) iff \( R^H_m([t]) = [t'] \).

Since each \( R^F_m \) with \( F \subseteq \mathcal{PF}^* \) denotes a deterministic reachability relation and \( R^H_m \subseteq R^F_m \) whenever \( F \subseteq F' \), it is enough to consider only the maximal sets of pointer fields \( F \) for which \( R^F_m \) holds between two equivalence classes \([t]\) and \([t']\) and only the minimal sets of pointer fields \( F \) for which \( R^F_m \) does not hold between \([t]\) and \([t']\).

We lift these transitive relations to the set of all location terms and we define, for any template \( T \in \mathcal{T}_\varphi, \beta \) and any \( m \in \mathcal{M}_T \),

\[
\mathcal{R}(m, T, \mathcal{L}''(\varphi)) = \left\{ R_m = \bigcup_{F \subseteq \mathcal{PF}^*} R^F_m \quad \bigg| \quad \begin{array}{l} R^F_m \in \mathcal{R}^F_m(R_t, \mathcal{L}''(\varphi)) \text{ and} \\
R_m \text{ is well-defined} \end{array} \right. 
\]

where the property that \( R_m \) is well-defined is defined as in Section 5.2.3.3.3.

Given \( b \in \mathcal{B}(e') \) the labeling represented by \( T, \mathcal{B}(m, T, \mathcal{L}''(\varphi)) \) denotes the set of all possible extensions of \( b \) to equivalence classes in \( \mathcal{L}_m(\varphi)/e_m \).

For every \( T \in \mathcal{T}_\varphi, \beta \) and \( m \in \mathcal{M}_T \), \( \mathcal{T}_\varphi, T, m \) denotes the set of templates such that every \( \tilde{T} \in \mathcal{T}_\varphi, T, m \) represents an equivalence relation \( \tilde{e} \in \mathcal{E}(m, T, \mathcal{L}''(\varphi)) \), a transitive relation \( \tilde{R} \) in \( \mathcal{R}(m, T, \mathcal{L}''(\varphi)) \) and a labeling \( \tilde{b} \in \mathcal{B}(m, T, \mathcal{L}''(\varphi)) \).

Each \( \tilde{T} \in \mathcal{T}_\varphi, T, m \) is built from \( T_m \) (the template defined in the first step starting from \( T \) and \( m \)) by collapsing vertices labeled with equivalent location variables w.r.t. \( \tilde{e} \), adding edges according to \( \tilde{R} \) and adding labels on the vertices according to \( \tilde{b} \).

Notice that \( \tilde{e} \) induces an equivalence relation, denoted by \( \sim_{\tilde{e}} \), over the vertices that are in \( V_m \):

\[
\text{if } t \in L_{V_m}(v) \mid_1 \text{ and } t' \in L_{V_m}(v') \mid_1 \text{ such that } (t, t') \in \tilde{e} \text{ then } v \sim_{\tilde{e}} v'.
\]

From the definition of \( \tilde{e} \) any term from \( \mathcal{L}_m(\varphi) \) is not equivalent with any term from \( \mathcal{L}'(\varphi) \) and two terms from \( \mathcal{L}'(\varphi) \) are equivalent w.r.t. \( \tilde{e} \) iff they are equivalent w.r.t. \( e' \). This means that \( \sim_{\tilde{e}} \) is a relation such that for every \( v \sim_{\tilde{e}} v' \) either \( v = v' \in V \), where \( V \) is the set of vertices of the template \( T \), or \( v, v' \in V_m \setminus V \).

To represent \( \tilde{e} \) we collapse any equivalent vertices \( v_m \sim_{\tilde{e}} v'_m \) where \( v_m, v'_m \in V_m \setminus V \). Formally, we define \( \tilde{T} = ((\tilde{V}, \tilde{E}, L_{\tilde{V}}, L_{\tilde{E}}), I_{\tilde{T}}) \) such that:

- a vertex in \( \tilde{V} \) represents an equivalence class of vertices from \( V_m \) w.r.t. \( \tilde{e} \), i.e., \( \tilde{V} = V_m/\sim_{\tilde{e}} \);
5.2. COMPOSITE STRUCTURE LOGIC. CSL

• for every $v \in \tilde{V}$, $v$ is labeled by all the terms labeling some vertex in the equivalence class it represents and by all the labels $\ell$ imposed by $\tilde{b}$. Formally,

$$L_{\tilde{V}}(v) = \left( \bigcup_{v' \sim_{\tilde{e}} v} L_{V_m}(v') \cup L_{V_m}(v), \bigcup_{\ell \in P\text{Var} \mid t \in L_{\tilde{V}}(v)} \ell \right) .$$

• the reachability relation from $T_m$ is redefined over the vertices in $\tilde{V}$. That is:

- if $\epsilon = (v_m, v'_m) \in E_m$ and $v_m \not\sim_{\tilde{e}} v'_m$ then $\tilde{\epsilon} = ([v_m], [v'_m]) \in \tilde{E}$ and $L_{\tilde{E}}(\tilde{\epsilon}) = L_{E_m}(\epsilon) ;$

Moreover, to represent the transitive relation $\tilde{R}$, for every $([t], [t']) \in \tilde{R}_F$, we add an edge $\epsilon = (v_t, v_{t'})$ to $\tilde{E}$, where $t \in L_{\tilde{V}}(v_t) \mid 1$, $t' \in L_{\tilde{V}}(v_{t'}) \mid 1$, such that $L_{\tilde{E}}(\epsilon) = (F, \tilde{l})$ with $\tilde{l}$ a fresh index variable.

• $I_{\tilde{T}}$ is the conjunction between $I_{T_m}$ and

$$\bigwedge_{\tilde{l} = L_{\tilde{E}}(\epsilon) \mid 2, \text{for some } \epsilon \in \tilde{E}} \tilde{l} \geq 1$$

The collapsing of nodes and the new edges does not introduce new constraints on index variables, besides $\tilde{l} \geq 1$, for any $\tilde{l}$ as above, because, in CSL, reachability predicates over universally quantified location variables don’t have length constraints.

Let $\rho_{\tilde{T}}$ be the following formula:

$$\rho_{T_m} \land \bigwedge_{(t,t') \in \tilde{\epsilon}} t = t' \land \bigwedge_{\max \tilde{R}([t],[t']) = F} t \xrightarrow{F} t' \land \bigwedge_{\max \tilde{R}([t],[t']) = F} t \xleftarrow{F} t' \land \bigwedge_{t' \in L_m(\varphi) \lor t \in L_m(\varphi)}$$

(5.2.22)
Example 5.2.15. Consider the templates $T^i_m$, $1 \leq i \leq 4$, from Figure 5.4 built for the formula $\psi_5$. $T^i_m$ and $T^i_k$, from Figure 5.4(a) and Figure 5.4(d) keep the same number of vertices in the second step of the expanding (no collapsing is allowed). We add edges between the vertex labeled by $x^m_1$, $x^m_2$, and $x^m_3$, $x^m_1$ and $z$, or $x^m_2$ and $z$, or $x^m_3$ and $z$. Figure 5.6(a) and Figure 5.6(d) give the templates obtained by applying the second step of template expanding to $T^4_m$ and $T^4_k$. For the templates $T^2_m$ and $T^2_k$ from Figure 5.4(b) and Figure 5.4(c), if we consider the equivalence relation $e$ such that $(x_m, x'_m) \in e$ then the vertices labeled by these variables are equivalent and therefore, we can create new templates by collapsing them. Notice that due to the constraints over the universally quantified variable $y$, only the templates $T^3_m$ and $T^4_m$ pictured in Figure 5.6(c) and Figure 5.6(d) lead to a model for $\psi_5$.

Using the definitions given above, we prove that:

Lemma 5.2.2. For any template $T \in \mathcal{T}_{\rho,\exists}$, the formula $\psi_T$ in (5.2.21) is equivalent to the formula $\psi$ given by:

$$\bigvee_{m \in M_T} \bigvee_{\bar{e} \in \mathcal{T}_{\rho,\exists}} \exists a \exists i \exists \bar{m} \exists m \exists x' \exists x_m \exists d \forall b \forall y \forall j \{\exists d, \forall d\}^* \cdot \phi_{e,r,R,b,x',\pi} \land \rho_T \land \mathcal{I}_T$$

where $x_m$ and $\bar{m}$ correspond to the newly added vertices and $l_m$ denote the index variables labeling the new edges of $\bar{e}$.

Proof. Let $H = (G, \delta)$ be a model for $\psi_T$. Then, for each $x \in \mathcal{X}$ there exists some vertex $v_x$ in $G$, for each $a \in a$ there exists some array vertex $v_a$ in $G$, for each $i \in i'$ there exists some natural number $n_i$, and for each data variable $d \in \mathcal{D}$ there exists some element $e_{d} \in \mathcal{D}$ such that

$$H \models_{\mu, \theta, \nu, \kappa} \forall b \forall y \forall j \{\exists d, \forall d\}^* \cdot \phi_{e,r,R,b,x',\pi} \land \rho_T \land \mathcal{I}_T$$

where $\mu = [x \leftarrow v_x]_{x \in \mathcal{X}}, \theta = [a \leftarrow v_a]_{a \in a}, \nu = [i \leftarrow n_i]_{i \in i'},$ and $\kappa = [d \leftarrow e_{d}]_{d \in \mathcal{D}}$.

After the elimination of the existential quantification, the formulas $\rho_T$ and $\mathcal{I}_T$ have only free variables (they don’t contain any atomic formulas over the universally quantified ones). Therefore,

$$H \models_{\mu, \theta, \nu, \kappa} \rho_T \land \mathcal{I}_T \land \forall b \lor \forall y \forall j \{\exists d, \forall d\}^* \cdot \phi_{e,r,R,b,x',\pi}$$

(5.2.23)

We may remark that, from the definition of the minimal solutions, there exists $m \in M_T$ such that $m(i) \leq \nu(i)$, for every index variable $i \in i' \subseteq i'$ which labels an edge in $T$. Then,

$$H \models_{\mu, \theta, \nu, \kappa} \exists m \exists \bar{m} \cdot (\rho_{T_m} \land \mathcal{I}_{T_m}) \land \forall b \lor \forall y \forall j \{\exists d, \forall d\}^* \cdot \phi_{e,r,R,b,x',\pi}$$

(5.2.24)

where $x_m$ and $l_m$ denote the vertices added to the template $T_m$, $l_m$ denote the lengths of the new edges, and $\rho_{T_m}$, $\mathcal{I}_{T_m}$ are the formulas associated with $T_m$. This holds for the following reasons:

- the formula $\rho_T$ has (at least) an atomic formula of the form $x^{F,B,L} \rightarrow x'$ with $F \in \mathcal{P}_F \cup \mathcal{P}_R$ or $x \xrightarrow{H} x'$ with $H \cup \mathcal{P}_F \cup \mathcal{P}_R = \emptyset$ for every edge in the template $T$. Then, $H \models_{\mu, \theta, \nu, \kappa} x^{F,B,L} \rightarrow x'$ implies that there exists an $F$-path of length $\nu(l) \geq m(l)$ in $G$ between $\mu(x)$ and $\mu(x')$ that does not pass through vertices labeled by pointer variables in $B$. Also, $H \models_{\mu, \theta, \nu, \kappa} x \xrightarrow{H} x'$ implies that either there exists an edge from $\mu(x)$ to $\mu(x')$ whose label contains $H$ or there exists an edge from $\mu(x')$ to $\mu(x)$ whose label contains $\overline{H}$. 

124
5.2. COMPOSITE STRUCTURE LOGIC. CSL

(a) A template $\tilde{T}^1_m$ in $\mathcal{T}_{\psi_5, T^1_m}$, where $m = (4, 1, 4, 1, 1)$, $m(l') = 4$, and $m(l) = 1$.

(b) A template $\tilde{T}^2_m$ in $\mathcal{T}_{\psi_5, T^2_m}$, where $m = (3, 2, 3, 2, 2)$, $m(l') = 3$, and $m(l) = 2$.

(c) A template $\tilde{T}^3_m$ in $\mathcal{T}_{\psi_5, T^3_m}$, where $m = (2, 3, 2, 3, 3)$, $m(l') = 2$, and $m(l) = 3$.

(d) A template $\tilde{T}^4_m$ in $\mathcal{T}_{\psi_5, T^4_m}$, where $m = (1, 4, 1, 4, 4)$, $m(l') = 1$, and $m(l) = 4$.

Figure 5.6: Expanded templates for the formula $\psi_5$. 

$I_{\tilde{T}^1_m} := \begin{cases} l = i \land l' = i' \land i < i' \land i = i'' \land i'' \land l = t_{m} + t_{l} + t_{m} + t_{l} \end{cases}$

$I_{\tilde{T}^2_m} := \begin{cases} l = i \land l' = i' \land i > i' \land i = i'' \land i'' \land l = t_{m} + t_{l} + l_{m} + l_{l} \end{cases}$

$I_{\tilde{T}^3_m} := \begin{cases} l = i \land l' = i' \land i < i' \land i = i'' \land i'' \land l = t_{m} + t_{l} + l_{m} + l_{l} \end{cases}$

$I_{\tilde{T}^4_m} := \begin{cases} l = i \land l' = i' \land i > i' \land i = i'' \land i'' \land l = t_{m} + t_{l} + l_{m} + l_{l} \end{cases}$
for every atomic formula of the form \( x^{F\emptyset I} \rightarrow x' \) in \( \rho_T \) there exists a conjunction of atomic formulas \( x^{F\emptyset I} \rightarrow x_m' \), \( x^{k\emptyset I} \rightarrow x_{m+1} \) and \( x_m^{F\emptyset I} \rightarrow x' \) in \( \rho_T \), with \( 1 \leq k \leq q - 1 \) and \( q = m(l) - 1 \). To prove (5.2.24),

- the interpretation \( \mu' \) of the variables \( x_m^{k} \), \( 1 \leq k \leq q \), is defined by \( \mu'(x_m^{k}) = v^k \), where each \( v^k \) is a distinct vertex on the \( F \)-path from \( \mu(x) \) to \( \mu(x') \), and
- the interpretation \( \nu' \) of the variables \( t_m \), \( 0 \leq j \leq q \), is defined by \( \nu'(t_m^{j}) = d_{st} \), where \( d_{st} \) is the distance between \( v^j \) and \( v^{j+1} \), for any \( 1 \leq j \leq q - 1 \), and \( d_{st_0} \), resp. \( d_{st_q} \), are interpreted into the distance between \( \mu(x) \) and \( v^1 \), resp \( v^q \) and \( \mu(x') \).

- for every \( a[i_1] \) and \( a[i_2] \) terms in \( \mathcal{L}''(\varphi) \) such that \( i_1 \) and \( i_2 \) are consecutive elements in \( \mathcal{L}''(\varphi) \) used to define \( T \), let \( l = i_2 - i_1 \) represent the number of elements on the array \( a \) between positions \( i_1 \) and \( i_2 \) ( \( l = i_2 - i_1 \) is also a formula in \( \mathcal{L}''(\varphi) \)). By the definition of \( m \), we have that \( \nu(l) \geq m(l) \). Consequently, there exist \( q = m(l) - 1 \) positions \( i_1^1, \ldots, i_1^q \) with \( i_1 < i_1' < \ldots < i_1^q < i_2 \) such that \( a[i_1^m] \) denotes a vertex of the array \( a \) between the positions denoted by \( i_1 \) and \( i_2 \), for every \( 1 \leq k \leq q \).

Let \( e_m \) be the equivalence relation over \( \mathcal{L}_m(\varphi) \) induced by the interpretation \( \mu' \). That is, for any \( t_m, t_m' \in \mathcal{L}_m(\varphi) \) if \( \mu'(t_m) = \mu'(t_m') \) then \( (t_m, t_m') \in e_m \). The equivalence relation \( e' \) which includes \( e_m \) and the equivalence relation \( e'' \) represented by \( T \) belongs to \( \mathcal{E}(m, T, \mathcal{L}''(\varphi)) \) because 1) \( \mu' \) is consistent with the typing system and 2) from its definition, for any \( x_m, x_m' \) in \( \mathcal{L}_m(\varphi) \) such that \( x^{F\emptyset I} \rightarrow x_m \land x_m^{F\emptyset I} \rightarrow x_m' \land x_m^{F\emptyset I} \rightarrow x' \) is a formula of \( \rho_T \), \( \mu' \) maps \( x_m, x_m' \) into distinct vertices along the path between \( \mu(x) \) and \( \mu(x') \).

Also, let \( \tilde{R} \) be the transitive relation induced by \( G \) over \( \mathcal{L}''(\varphi) \). The transitive relation \( \tilde{R} \) belongs to \( \mathcal{R}(m, T, \mathcal{L}''(\varphi)) \) because all the restrictions imposed on the transitive relations from \( \mathcal{R}(m, T, \mathcal{L}''(\varphi)) \) derive from the fact that these relations must represent the reachability relation w.r.t. some pointer fields. Since \( \tilde{R} \) is defined from a model, then \( \tilde{R} \in \mathcal{R}(m, T, \mathcal{L}''(\varphi)) \).

The same holds for the labeling of the vertices \( \mu'(x_m) \) in \( G \). Then, we conclude that

\[
H \models_{\mu, \emptyset, \nu, r} \exists x_m \exists x_m. (\rho_\tilde{T} \land I_\tilde{T}) \land \forall b \forall y \{ \exists d, \forall d \}^*. \phi_{e, r, R, b, \nu, \pi, \pi'}
\]

where \( \tilde{T} \) is an expanded template which is defined for \( T \) and \( m \), and which represents \( \tilde{e}, \tilde{R} \) and \( \tilde{b} \). Finally, using the fact that \( l_m \) and \( x_m \) are not bound in \( \phi_{e, r, R, b, \nu, \pi, \pi'} \) we obtain that

\[
H \models_{\mu, \emptyset, \nu, r} \exists x_m \exists x_m. \forall b \forall y \{ \exists d, \forall d \}^*. (\phi_{e, r, R, b, \nu, \pi, \pi'} \land \rho_\tilde{T} \land I_\tilde{T})
\]

Consequently, the claim of the lemma holds.

Then, we can prove the following fact:

**Lemma 5.2.3.** Let

\[
\varphi_{T,m,T'} \ ::= \exists a \exists b \exists y \exists x_m \exists x_m' \exists x_m \exists y \exists y \{ \exists d, \forall d \}^*. \phi_{e, r, R, b, \nu, \pi, \pi'} \land \rho_\tilde{T} \land I_\tilde{T}
\]

be a disjunct of \( \psi_{\tilde{T}, \tilde{T}_T} \) as in Lemma 5.2.2. If \( \varphi_{T,m,T'} \) is satisfiable then it must have a model of size less than or equal to the number of its existentially quantified index, array, and location variables.
5.2. COMPOSITE STRUCTURE LOGIC. CSL

Proof. Let \( H = (G, \delta) \), where \( G = (V, E, L_D, L_E) \), be a model of
\[
\exists a \exists i_T \exists i_m \exists x_T \exists x_m \exists d \forall y \forall j \{\exists d, \forall d\}^* \cdot \phi_{e, r, R, b, \pi, \pi'} \land \rho \land \mathcal{I}_T.
\]

Then, there exist valuations \( \mu : x' \cup x_m \rightarrow V \), \( \nu : i'_T \cup i_m \cup i_m \rightarrow \mathbb{N} \), \( \theta : a \rightarrow AV \), \( \kappa : d \rightarrow \mathbb{D} \), where \( AV \) is the set of array vertices in \( G \), such that
\[
H \models \mu, \theta, \nu, \kappa \forall y \forall j \{\exists d, \forall d\}^* \cdot \phi_{e, r, R, b, \pi, \pi'} \land \rho \land \mathcal{I}_T.
\]

For each array variable \( a \in a \), let \( i_a \) be the set of index variables \( i \) in \( i'_T \cup i_m \) that appear in terms of the form \( a[i] \) in \( \phi_{e, r, R, b, \pi, \pi'} \land \rho \land \mathcal{I}_T \).

We define a heap graph \( G' \) which contains: (1) the vertices in \( G \) from the image of \( \mu \), (2) the array vertices from the image of \( \theta \), and (3) for each array variable \( a \in a \) and each \( i \in i_a \), it contains the vertex in \( G \) at distance \( \nu(i) \) from the array vertex \( \theta(a) \). The labeling of the vertices in \( G' \) is exactly the one from \( G \).

Let \( \mu(x) \) and \( \mu(x') \) be two vertices of \( G \) having the same type. We consider all the paths in \( G \) between \( \mu(x) \) and \( \mu(x') \). For every path that is a witness for some reachability predicate \( x F 0 x' \) or \( x \xrightarrow{F} x' \), where \( F \in \mathcal{P}F^* \) is maximal, we add to \( G' \) an edge labeled by \( F \) between \( \mu(x) \) and \( \mu(x') \). The edges labeled by array fields are added as follows. Let \( n = \langle a[i] \rangle_{H, \mu, \theta, \nu} \) and \( n' = \langle a[i] \rangle_{H, \mu, \theta, \nu} \) be the interpretations of \( a[i] \) and \( a[i] \) in \( G \), where \( a \in A \) and \( i_1, i_2 \in I_a \). We add to \( G' \) an edge between \( n \) and \( n' \) labeled by \( a \), if the path in \( G \) between \( n \) and \( n' \) following the array field \( a \) does not contain some node \( \langle a[i] \rangle_{H, \mu, \theta, \nu} \) with \( i \in I_a \), \( i \neq i_1 \), and \( i \neq i_2 \). All the edges in \( G \) labeled by sets of non-recursive pointer fields between nodes in the image of \( \mu \) are copied to \( G' \) (together with their labeling). Notice that every node, respectively edge, in \( G' \) corresponds to a node, respectively edge, in the template \( T \).

Example 5.2.16. Figure 5.7(a) shows a model \( G \) for the formula \( \psi_4 \) and Figure 5.7(b) shows the graph \( G' \) built from \( G \) according to the following valuations: (1) \( \theta \) that maps \( a \), resp \( c \), to the first vertex \( v_a \), resp. \( v_c \), of the path labeled by the array field \( a \), resp. \( c \), (2) \( \mu \) maps \( x \) to the first vertex of the path labeled by the pointer field \( f \), and (3) \( \nu \) is defined by \( \nu(i_1) = 2 \), \( \nu(i_2) = 5 \), and \( \nu(i_3) = 6 \). The valuation of the data variables from \( DVar \) is omitted.

![Figure 5.7: Models of \( \psi_4 \)](image)
CHAPTER 5. PROGRAM VERIFICATION USING FRAGMENTS OF GCSL

Let \( I_m \subseteq i' \cup I_m \) be the index variables that label the edges of the template \( \hat{T} \) and \( I'_m = \bigcup_{a \in A} I_a \). Let \( \nu' \) be a valuation for the index variables in \( i' \cup I_m \cup I_m \) such that \( \nu'(i) = 1 \), for all \( i \in I'_m \). By the construction of the template \( \hat{T} \), the latter determines uniquely the evaluations for all \( i \in I_a \), for any \( a \in A \). Moreover, the set \( \{ \nu'(i) \mid i \in I_a \} \) contains all the natural numbers between 0 and some \( \alpha \in \mathbb{N} \). The construction of \( G' \) implies that \( \langle\langle a[i]\rangle\rangle_{H',\mu,\theta,\nu'} = \langle\langle a[i]\rangle\rangle_{H',\mu,\theta,\nu'} \), where \( H' = (G', \delta) \). In the following, we prove that

\[
H' \models \mu,\theta,\nu',\kappa \forall \nu \forall j \{ \exists d, \forall d \}^* \cdot (e, r, R, b, \pi') \land \hat{I}_T. \tag{5.2.25}
\]

To this, let \( \theta_b : b \to AV \) be any valuation of the array variables in \( b \) (into the set of array vertices of \( G' \), denoted by \( AV' \), \( \mu_y : y \to V' \) any valuation of the location variables in \( y \) (into the set of vertices of \( G' \), denoted by \( V' \)), and \( \nu'_j : j \to N \) any valuation of the index variables in \( j \). We have to prove that either there exists a sub-term \( t \) of \( \{ \exists d, \forall d \}^* \cdot (e, r, R, b, \pi') \land \hat{I}_T \) such that \( \langle\langle t\rangle\rangle_{H',\mu,\theta,\nu,\nu'_j,\kappa} = \perp \) or

\[
H' \models \mu,\mu_y,\theta,\theta_b,\nu',\nu'_j,\kappa \{ \exists d, \forall d \}^* \cdot (e, r, R, b, \pi') \land \hat{I}_T. \tag{5.2.26}
\]

Let \( \nu'_j : j \to N \) be a mapping of the universal index variables such that \( \nu'_j(j) \leq \min\{ \text{max}_{G'}^\nu | v \in \text{Arr}(\theta \cup \theta_b, j) \} \) where

\[
\text{Arr}(\theta \cup \theta_b, j) = \{ v \in V' \mid (e, r, R, b, \pi') \land \hat{I}_T \text{ contains a term } a[j] \text{ with } a \in A \text{ and } \theta(a) = v, \text{ or } \text{a term } b[j], \text{ with } b \in b \text{ and } \theta_b(b) = v \}
\]

and \( \text{max}_{G'}^\nu \) is the length of the maximum path in \( G' \) that starts in \( v \) and whose edges are labeled by some array field (by the array well-formedness, this path is unique).

By the semantics of gCSL, for any \( \nu'_j \) that does not satisfy the properties above, there exists a term \( t \) such that \( \langle\langle t\rangle\rangle_{H',\mu,\mu_y,\theta,\theta_b,\nu',\nu'_j,\kappa} = \perp \). Also, by the construction of \( \hat{T} \), for any \( j \in j, \nu'_j(j) \) equals \( \nu'(i) \) for some index variable \( i \in I'_m \).

**Example 5.2.17.** Consider again the formula \( \psi_4 \) and the graph \( G' \) in Example 5.2.16. Given a valuation \( \theta_b \) defined by \( \theta_b(b) = v_a \), \( \text{Arr}(j, \theta \cup \theta_b) = \text{Arr}(j', \theta \cup \theta_b) = \{ v_a \} \), and \( \text{Arr}(j'', \theta \cup \theta_b) = \{ v_a, v_c \} \). Also, \( \text{max}_{G'}^\nu = 2 \) and \( \text{max}_{G'}^\nu = 3 \). The mappings \( \nu'_j \) described above (which does not induce undefined terms) assign to \( j \) and \( j' \) values between 0 and 2, and to \( j'' \) values between 0 and 2. For any other values for the universal index variables, one of the location terms \( b[j], b[j'] \), and \( c[j''] \) is undefined.

We prove that property (5.2.26) holds. We use the fact that \( H \models \mu,\theta,\nu,\kappa \forall b \forall y \forall j \{ \exists d, \forall d \}^* \cdot (e, r, R, b, \pi') \land \hat{I}_T \). Since the nodes in \( G' \) are a subset of the nodes in \( G \), \( \mu_y \) and \( \theta_b \) are also valuations of the variables in \( y \), resp. \( b \), in \( G \). Notice that, for any \( y \in y \), there exists \( x \in x' \cup x_m \) such that \( \mu_y(y) = \mu(x) \). Moreover, let \( \nu_j : j \to N \) be a valuation such that \( \nu_j(j) = \nu(i) \) iff \( \nu'_j(j) = \nu'(i) \). By the semantics of the universal quantifiers, we have that

\[
H \models \mu,\mu_y,\theta,\theta_b,\nu,\nu_j,\kappa \{ \exists d, \forall d \}^* \cdot (e, r, R, b, \pi') \land \hat{I}_T. \tag{5.2.28}
\]

For simplicity, let \( \phi' \) be the disjunctive normal form of \( (e, r, R, b, \pi') \land \hat{I}_T \). We prove by structural induction that

\[
H \models \mu,\theta,\nu,\kappa \{ \exists d, \forall d \}^* \cdot \phi' \implies H' \models \mu,\theta,\nu,\kappa \{ \exists d, \forall d \}^* \cdot \phi', \tag{5.2.29}
\]

128
5.2. COMPOSITE STRUCTURE LOGIC. CSL.

where \( \overline{p} = \mu \cup \mu', \overline{\theta} = \theta \cup \theta_b, \overline{\nu} = \nu \cup \nu', \text{ and } \overline{\nu'} = \nu' \cup \nu' \).

The following cases are to be discussed:

- if the formula \( \{3d, \forall d\}^* \phi' \), denoted by \( \omega \), is an atomic formula then:

  - if \( \omega : \ell(x), \omega : -\ell(x) \), or \( \omega : x = x' \) with \( x \in \text{Loc} \), then \( \text{[5.2.29]} \)
    holds because \( \overline{p}(x) \) in \( G' \) is a copy of \( \overline{p}(x) \) in \( G \) (together with the labeling);
  
  - if \( \omega : x \xrightarrow{H} x' \) or \( \omega : -x \xrightarrow{H} x' \) with \( x, x' \in \text{Loc} \) and \( H \subseteq (\mathcal{PF} \setminus \mathcal{PF}_r) \cup \overline{\mathcal{PF}} \setminus \mathcal{PF}_r \) then \( \text{[5.2.29]} \)
    holds because the graph \( G' \) contains both \( \overline{p}(x) \) and \( \overline{p}(x') \) and by definition, \( G' \) contains an edge between \( \overline{p}(x) \) and \( \overline{p}(x') \) whose label contains \( H \) if \( G \) contains the same edge;
  
  - if \( \omega : x \xrightarrow{A,B} x' \) with \( x, x' \in \text{Loc} \) then \( \text{[5.2.29]} \)
    holds because the truth value of \( \omega \) in \( G' \) and \( G \) is established under the same valuation for the location variables and because, by definition, \( G' \) preserves the reachability relations from \( G \). That is, for every \( A \)-path between \( \overline{p}(x) \) and \( \overline{p}(x') \) in \( G \) there exists a similar path in \( G' \). Moreover, since paths in \( G' \) are obtained from paths in \( G \) by replacing sub-paths with edges, the constraint of not passing through vertices labeled by pointer variables in \( B \) holds;
  
  - if \( \omega : -x \xrightarrow{A,B} x' \) with \( x, x' \in \text{Loc} \) and \( f \in \mathcal{PF}_r \) then \( \text{[5.2.29]} \)
    holds because by construction, \( G \) contains a path between \( \overline{p}(x) \) and \( \overline{p}(x') \) formed of edges labeled by the pointer field \( f \) iff \( G' \) contains a similar path.
  
  - if \( \omega : x \in \text{Ind} \) then \( \text{[5.2.29]} \)
    holds because \( \{a[i]\}\overline{\mathcal{H}},\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}} = \{a[i]\}\overline{\mathcal{H}'}\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}} \).
  
  - if \( \omega : \text{Exp} < ct, \omega : \text{Exp} = ct, \text{ or } \omega : -\text{(Exp} = \text{ct}) \) with \( \text{Exp} \) an index expression over index variables in \( i' \cup i_m \) then \( \text{[5.2.29]} \)
    holds because the valuations \( \nu \) and \( \nu' \) are both solutions for the constraint in \( \omega (\nu' \text{ is a minimal optimal solution for } \omega) \):
  
  - if \( \omega : j < j', \omega : j \leq j', \omega : j = j' \) or \( \omega : -(j = j') \) with \( j, j' \in \mathbb{J} \) then \( \text{[5.2.29]} \)
    holds because: (1) by the definition of \( \overline{\nu}, \overline{\nu}'(j) = \nu'(j') \), \( \overline{\nu}(j) = \nu(i) \), and \( \overline{\nu}(j') = \nu(i') \) for some \( i, i' \in i' \cup i_m \), (2) the order or the equality between the values of the index variables \( i \) and \( i' \) is fixed in the formula \( \mathcal{I}, \) i.e., \( \nu(i) < \nu(i') \iff \nu'(i) < \nu'(i') \) and \( \nu(i) = \nu(i') \iff \nu'(i) = \nu'(i') \);
  
  - if \( \omega : p(d_1, \ldots, d_t) \) or \( \omega : -p(d_1, \ldots, d_t) \), where \( d_{k}, 1 \leq k \leq t \), is a data term, then \( \text{[5.2.29]} \)
    holds because location variables are interpreted into nodes having the same data labeling and because \( \{a[i]\}\overline{\mathcal{H}'}\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}} = \{a[i]\}\overline{\mathcal{H}},\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}} \).
  
- if \( \omega : \varphi_1 \lor \varphi_2 \) (resp. \( \omega : \varphi_1 \land \varphi_2 \)) then \( H \models_{\overline{\mathcal{H}},\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}},\overline{\nu},\overline{\nu}' \omega \) implies that \( H \models_{\overline{\mathcal{H}},\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}},\overline{\nu}} \varphi_1 \) or (resp. and) \( H \models_{\overline{\mathcal{H}},\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}},\overline{\nu}} \varphi_2 \). Using the induction hypothesis we obtain that \( H' \models_{\overline{\mathcal{H}'}\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}},\overline{\nu},\overline{\nu}' \varphi_1 \) or (resp. and) \( H' \models_{\overline{\mathcal{H}'}\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}},\overline{\nu}} \varphi_2 \) and consequently, \( H' \models_{\overline{\mathcal{H}},\overline{\mathcal{I}},\overline{\mathcal{F}},\overline{\mathcal{P}},\overline{\nu}} \omega \).
CHAPTER 5. PROGRAM VERIFICATION USING FRAGMENTS OF GCSL

- if \( \omega : \exists d. \varphi \) or \( \omega : \forall d. \varphi \) with \( d \in \text{Data} \) then (5.2.29) is proved by applying the induction hypothesis on \( \varphi \).

\[ \begin{array}{c}
\begin{array}{c}
\text{v}_1 \\
\text{f,} \\
\text{g}
\end{array} \\
\begin{array}{c}
\text{v}_2 \\
\text{f,} \\
\text{g}
\end{array}
\end{array} \]

(a) \hspace{2cm} \begin{array}{c}
\text{v}_1 \\
\text{f,} \\
\text{g}
\end{array} \\
\begin{array}{c}
\text{v}_2 \\
\text{f,} \\
\text{g}
\end{array}

(b)

Figure 5.8: The graphs \( G \) and \( G' \) from Remark 5.2.2

5.2.2 Remark. Notice that, by construction, the formula \( \{\exists d, \forall d\}^* \phi' \) does not contain literals of the form \( \neg x^{A,B} \rightarrow x' \) with \( x, x' \in \text{Loc} \) and \( f \in \text{PF}_r \). Also, by the definition of CSL, it does not contain literals of the form \( \neg x^{A,B,i} \rightarrow x' \) with \( A \) containing at least two pointer fields.

Property (5.2.29) may not hold if \( \omega \) contains the formula \( \neg x^{A,B} \rightarrow x' \), where \( A \) contains at least two pointer fields. Figure 5.8(a) shows a graph \( G \) that satisfies \( x^{\{f\}}_i \rightarrow x' \wedge x^{\{g\}}_i \rightarrow x' \) w.r.t. the valuation \( \mu \) defined by \( \mu(x) = v_1 \) and \( \mu(x) = v_2 \) and Figure 5.8(b) shows a graph \( G' \) defined as above such that \( G' \) does not satisfy \( x^{\{f,g\}}_i \rightarrow x' \) according to the same valuation \( \mu \).

Having computed a bound on the small models of \( \varphi \), we can apply the transformations given in Section 5.2.3.5 and 5.2.3.6 (i.e., elimination of the universal and then of the existential quantification over location, array, and index variables) in order to compute an equi-satisfiable formula in \( \text{FO}(D, \mathcal{O}, P) \).

5.2.3.5 Elimination of universally quantified array, index and location variables

Based on a small model property, we define a transformation that eliminates the universally quantified location, array and index variables from \( \varphi \) and outputs an equi-satisfiable formula. The main difficulty comes from the fact that universally quantified index variables don’t commute with respect to the conjunction. The solution is to eliminate first the universally quantified array variables. Then, from every conjunct obtained in this way we eliminate the universally quantified index variables. Intuitively, this corresponds to renaming the universally quantified index variables in every conjunct.

Proposition 5.2.1. Let \( \varphi \) be CSL formula in prenex normal form. There exists a formula \( \varphi^3 \) without universally quantified location, array and index variables such that \( \varphi \) is equi-satisfiable to \( \varphi^3 \).

Proof. Let \( \varphi := \exists a_i \exists x \exists d \forall b \forall j \{\exists d, \forall d\}^* \phi \). By (5.2.20),

\[ \varphi \text{ is equivalent to } \bigvee_{T \in \mathcal{T}_r} \bigvee_{\exists \mathcal{M}_T} \bigvee_{\bar{r} \in \mathcal{T}_r,T,m} \psi_{\bar{r}}, \]

where

\[ \psi_{\bar{r}} = \exists a_i \exists l_i \exists i_m \exists x_m \exists x' x_m \exists d \forall b \forall y \forall j \{\exists d, \forall d\}^* \phi_{e,r,b,\pi,\pi',\bar{r}} \land \rho_{\bar{r}} \land I_{\bar{r}}. \]
5.2. COMPOSITE STRUCTURE LOGIC. CSL

In the following, we denote $\varphi_{e,r,R,b,\pi,\pi'} \land \rho_{\bar{T}} \land \mathcal{I}_{\bar{T}}$ by $\psi_1$.

Let $H = (G, \delta)$, where $G = (V, E, L, E_L, L_D)$, be a model for the formula $\varphi$. Then, there exists $T \in \mathcal{T}_{\varphi, \exists}$, $m \in \mathcal{M}_T$, and $\bar{T} \in \mathcal{T}_{\varphi, \exists, \forall}$ such that

$$H \models \varphi \text{ iff } H \models \psi_{\bar{T}}.$$  \hspace{1cm} (5.2.30)

Then, for each $x \in x \cup x_m$ there exists some vertex $v_x$, for each $a \in a$ there exists some array vertex $v_a$, for each $i \in i' \cup i_m$ there exists some natural number $n_i$, and for each data variable $d \in d$ there exists some element $e_d \in \mathbb{D}$ such that:

$$H \models \mu, \theta, \nu, \kappa \forall b \forall y \forall j \{\exists d, \forall d\}^x. \psi_1,$$

where $\mu = [x \leftarrow v_x]_{x \in x}$, $\theta = [a \leftarrow v_a]_{a \in a}$, $\nu = [i \leftarrow n_i]_{i \in i'}$, and $\kappa = [d \leftarrow e_d]_{d \in d}$.

From $\rho_{\bar{T}}$ we can deduce the set of location variables $x^a \subseteq x \cup x_m$ which are interpreted to vertices that don’t have any outgoing/incoming edges from/into labeled by an array field. These the location variables represent arrays of size one. Formally, $x \in x^a$ iff for any term $a[i]$ or $a[0]$ of $\psi_1$, $(x, a[i]) \notin e$ and $(x, a[0]) \notin e$, where $e$ is the equivalence relation on location terms represented by $\bar{T}$.

If $\Lambda$ is the set of all possible (partial or total) mappings between the variables $b$ and $a \cup x^a$ which preserve types (that is, $b$ and $\lambda(b)$ have the same type, for any $b \in b$ and $\lambda \in \Lambda$) then, $\forall b \forall y \forall j \{\exists d, \forall d\}^x. \psi_1$ is equivalent to

$$\bigwedge_{\lambda \in \Lambda} \forall b \left( \bigwedge_{b \in \text{dom}(\lambda)} b[0] = \lambda(b)[0] \land \bigwedge_{b \notin \text{dom}(\lambda)} \left( \bigwedge_{a \in a} b[0] \neq a[0] \land \bigwedge_{x \in x^a} b[0] \neq x \right) \right) \Rightarrow \psi_2,$$

where $\psi_2 = \forall y \forall j \{\exists d, \forall d\}^x. \psi_1$.

Let $i_a \subseteq i' \cup i_m$ be the set of index variables that appears in a term of the form $a[i]$ in $\psi_2$. From $\rho_{\bar{T}}$ we can deduce, for each $a \in a$, the index variable $i_{\text{max}}^{a}$ that is the greatest existential index variable in $i_a$. For every $\lambda \in \Lambda$ we define $\text{Ind}(\lambda, j)$ the set of index variables $\{i | i \in \{0\} \cup i' \cup i_m$ and $i \in i_a$ for some $a \in a$ such that $i$ is less than the smallest $i_{\text{max}}^{a}$ with $a[j]$ or $\lambda(b)[j]$ a term in $\psi_2$ and $\lambda(b) = a$.

Let $\Omega$ be the set of all possible (partial or total) mappings between the variables in $j$ and $i' \cup i_m$ such that for all $\omega \in \Omega$, $j \in j$, $\omega(j) \in \text{Ind}(\lambda, j)$. The semantics of CSL demands that we evaluate the an universally quantified index variables $j \in j$ over positions that are defined on all the arrays where $j$ is used. This is why, every evaluation of the universally quantified array variables $b$ induces a different evaluation for the universally quantified index variables $j$.

Then, $\psi_2$ is equivalent to

$$\bigwedge_{\omega \in \Omega} \forall j \left( \bigwedge_{j \in \text{dom}(\omega)} j = \omega(j) \land \bigwedge_{j \notin \text{dom}(\omega)} \bigwedge_{i \in \text{Ind}(\lambda, j)} j \neq i \right) \Rightarrow \psi_3 \text{ where}$$

$$\psi_3 = \forall y \{\exists d, \forall d\}^x. \psi_1$$

Finally, let

$$\text{Loc} = x \cup x_m \cup \bigcup_{i \in i_a} \{a[i] \mid a[i] \text{ is a term in } \psi_3 \} \cup \bigcup_{a \in a} a[0].$$
If $\Xi$ is the set of all possible (partial or total) mappings between the variables $y$ and $\mathbf{Loc}$, which preserve types (that is, $y$ and $\xi(y)$ have the same type, for any $y \in y$ and $\xi \in \Xi$), then $\psi$ is equivalent to
\[
\bigwedge_{\xi \in \Xi} \forall y \left( \bigwedge_{y \in \text{dom} (\xi)} y = \xi(y) \right) \land \left( \bigwedge_{y \notin \text{dom} (\xi)} \right) \Rightarrow \{ \exists d, \forall d \}^* \cdot \psi_1
\]

Let $G'$ be the small model obtained from $G$ by restricting it to the image of $\mu, \nu, \theta$ and $H' = (G', \delta)$. By Lemma 5.2.3, we obtain that
\[
H' \models_{\mu, \nu, \omega, \kappa} \bigwedge_{\lambda \in \Lambda} \bigwedge_{\omega \in \Omega} \bigwedge_{\xi \in \Xi} \{ \exists d, \forall d \}^* \cdot \psi_1^{\lambda, \omega, \xi}
\]
where,
\[
\psi_1^{\lambda, \omega, \xi} = \psi_1^{\left[\begin{array}{ccc}
y & \xi(y) \\
j & \omega(j) \\
b & \lambda(b)
\end{array}\right]}_{y \in y, j \in j, b \in b}
\]
Moreover,
\[
H' \models_{\emptyset, \emptyset, \emptyset, 0} \exists a \exists x \exists m \exists i \exists m \exists d. \bigwedge_{\lambda \in \Lambda} \bigwedge_{\omega \in \Omega} \bigwedge_{\xi \in \Xi} \{ \exists d, \forall d \}^* \cdot \psi_1^{\lambda, \omega, \xi}
\]
Conversely, every minimal model of the formula above is necessarily a model of the formula $\varphi$.

**Example 5.2.18.** Let’s consider the formula $\psi_4$, its template $T$ from Figure 5.3(a). The minimal solution $m(i_1) = 1, m(i_2 - i_1) = 1, m(i_3 - i_2) = 1$ and finally let’s consider $T = T_m$ (the transitive relation on location terms is exactly the one represented by $T_m$). After the elimination of universally quantified location, array and index variables we obtain the following equi-satisfiable formula: $\exists x, c \exists x \exists y \exists i, i_2, i_3. \left( \psi_4^x \land \psi_4^c \land \psi_4^y \right)$ where $\psi_4^x$ is the formula corresponding to the case when $b$ interprets into the same array as $a$:
\[
\psi_4^x = dt(x) = 1 \land dt'(a[i_1]) \leq dt'(c[i_1]) \land dt'(a[i_2]) \leq dt'(c[i_3]) \land
\bigwedge_{0 \leq q \leq q'} (q \leq i_1 + i_2 + i_3 \land q' \leq i_1 + i_2 + i_3) \Rightarrow dt(a[q]) \leq dt(a[q'])
\]
where
\[
\psi_4^c = dt'(c[i_1]) \leq dt'(c[i_3]) \land (0 \leq i_1 + i_2 + i_3 \land \bigwedge_{0 \leq q \leq 2} (dt'(a[q]) \leq 7 \land dt(c[q]) \leq 7).
\]
The formula $\psi_4^y$, which corresponds to the case when $b$ interprets into the same array as $c$, is defined similarly. $\psi_4^y$ is the formula corresponding to the case when $b$ interprets into the array vertex $x$:
\[
\psi_4^y = dt(x) = 1 \land dt'(a[i_1]) \leq dt'(c[i_1]) \land dt'(a[i_2]) \leq dt'(c[i_3]) \land
\bigwedge_{0 \leq q \leq 2} (dt'(a[q]) \leq 7 \land dt'(c[q]) \leq 7).
\]
5.2. COMPOSITE STRUCTURE LOGIC. CSL

In the formula $\psi^x_4$, because $x$ is a location variable, we consider it like an array on size one. Therefore, $j$ is interpreted only into 0 and $dt(x[0])$ is simply $dt(x)$.

Example 5.2.19. Consider the formula $\psi_2$ given as example in the overview of the proof, Section [5.2.3.1]. The equi-satisfiable formula without universally quantified variables that we obtain when we consider the template from Example [5.2.14] is the following:

$$\psi'_{2} = \exists x, q, z \exists x^5_m, x^6_m, x^7_m, (q \neq z \land x[1] \rightarrow q \land x[1] \rightarrow z \land dt(x) = 0 \land dt(q) = 2 \land i_1 + i_2 \geq 8 \land dt(x) < dt(y) < dt(y')).$$

5.2.3.6 Computing an equi-satisfiable formula in $\text{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$

In order to eliminate the existentially quantified index, location, and array variables we are going to introduce a set of fresh data variables $d_{\text{new}}$ which are existentially quantified in front of the formula. We use these variables to denote data from $\mathbb{D}$ attached to the vertices denoted by location terms.

We eliminate all variables that are not interpreted in $\mathbb{D}$ and we obtain an equi-satisfiable formula to $\varphi$:

$$\varphi' = \bigvee_{T \in T_e} \bigvee_{T \in T_f} \bigvee_{T \in T_m} \exists d_{\text{new}} \exists d \{ \exists d, \forall d \}^* \bigwedge_{T \in T_e, \tau, m} \bigwedge_{T \in T_f, \tau, m} \bigwedge_{T \in T_m} \bigwedge_{T \in T_e, \tau, m} \{ \exists d, \forall d \}^* \cdot \psi_2^{\lambda, \omega, \xi},$$

in the underlying data logic $\text{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$, where $\psi_2^{\lambda, \omega, \xi}$ is obtained from $\psi_1^{\lambda, \omega, \xi}$ as follows:

- the atomic formulas in $\psi_1^{\lambda, \omega, \xi}$ which are not of the form $o(dt, \ldots, dt)$ are replaced by $\text{true}$ or $\text{false}$ according to $\rho_\tau$. Then, the formula $\rho_\tau$ is deleted from $\psi_2^{\lambda, \omega, \xi}$;
- for each data field $dt$ of type $\tau(dt) = R \rightarrow \mathbb{D}$ and each node $v$ of the template $\tilde{T}$ of type $R$ we consider a fresh data variable $d_{dt,v}$. Then, we replace each term of the form $dt(x)$, where $x \in L_{\psi}(v)|_2$, for some vertex $v$ of $\tilde{T}$, by the corresponding data variable. The set of all new data variables is denoted $d_{\text{new}}$.

5.2.4 Complexity

Let us define the size of a CSL formula to be the number of existential variables, operators, and predicates, plus the integer constants.

Theorem 5.2.2. Let $\varphi$ be a closed CSL$_1$ formula. The satisfiability of $\varphi$ can be reduced to the satisfiability of a FO($\mathbb{D}, \mathbb{O}, \mathbb{P}$) formula $\varphi'$ in nondeterministic double exponential time. The size of $\varphi'$ is doubly-exponential in the size of $\varphi$. If the number of universally quantified variables of $\varphi$ is fixed then the size of $\varphi'$ is exponential in the size of $\varphi$ and the complexity of the reduction procedure is nondeterministic exponential time.

Proof. Let $x$, $a$, and $i$ be the existentially quantified location, array and index variables of $\varphi$. In the following, we give the complexity of each step of the reduction to FO($\mathbb{D}, \mathbb{O}, \mathbb{P}$).
CHAPTER 5. PROGRAM VERIFICATION USING FRAGMENTS OF GCSL

Step 1: Choosing the equivalence classes on terms of the formula, \( e_i \in E_i, e_a \in E_a, e \in E \), is done in polynomial time. To express the constraints imposed by the chosen equivalence relations the size of the formula grows polynomially (we add \( O(|x|^2 + |i|^2 + |a|^2) \) new atomic formulas).

Choosing an order \( r \in R(I(\varphi)/e_i) \), on the index expressions in \( I(\varphi) \) is done in polynomial time. At this step we introduce a linear number of fresh existential index and location variables.

The number of sets of pointer fields is exponential in the number of pointer fields used in the formula, i.e., \( 2^{\left| \mathcal{PF}(\varphi) \right|} \), where \( \mathcal{PF}(\varphi) \) denotes the pointer fields used in the formula \( \varphi \). Notice that \( |\mathcal{PF}(\varphi)| \) is linear in the number of reachability atomic formulas in \( \varphi \). Since the pointer fields represent deterministic reachability relations and \( R_{F'} \subseteq R_F \) whenever \( F \subseteq F' \), choosing a relation \( R \in R(e, r, L'(\varphi)) \) can be done in polynomial time as follows: for any two equivalence classes in \( L'(\varphi)/e_i \), we choose a set of relations \( R_F \) with \( F \subseteq \mathcal{PF}(\varphi) \) to which they belong, whose number is linear in \( |\mathcal{PF}| \).

To express the constraints imposed by the chosen transitive relations the size of the formula grows polynomially (we add \( O(|\mathcal{PF}(\varphi)| \cdot |x|^2 + |i|^2) \) new atomic formulas).

When we eliminate negations of reachability atomic formulas of the form \( \neg z.F_0 \rightarrow z' \) (in Lemma 5.2.1), we add a polynomial number of existential location variables, i.e. \( O(2 \cdot |x|^2 \cdot |\mathcal{PF}(\varphi)|) \). Moreover, choosing the transitive relations on these new location variables is done in polynomial time, i.e. \( O(|\mathcal{PF}(\varphi)| \cdot |x|^2) \), in the size of the original formula \( \varphi \). To express the new transitive relations, the size of the formula grows polynomially in the size of the formula from the previous step (we add a sub-formula polynomial in the size of \( \varphi \)).

Choosing a truth valuation for the unary label predicates and for the length constraints is done in polynomial time. Moreover, the size of the formula grows polynomially in the size of the initial formula.

The number of nodes and edges in the template is polynomial in the size of \( \varphi \) (the number of edges between two vertices is bounded by the number of pointer fields in \( \varphi \)). The template construction is done in polynomial time. The size of the formula grows polynomially because for every edge we add a fresh index variable, and the corresponding constraints. Overall, the complexity of the first step is polynomial time and the size of the formula grows polynomially.

Step 2: MOILP can be solved in non-deterministic exponential time. This is implied by the simple exponential bound \( \beta \) on the components of a minimal solution for a set of linear constraints given in [Pottier 1991]. We can give a procedure that guesses some solution \( m \) for the set of linear constraints and then checks that all the vectors of integers which are less than \( m \) are not solutions for these constraints; the number of vectors less than \( m \) is the number of index variables of the set of linear constraints times \( \beta \).

Given a template \( T = (T_G, T_F) \in T_{\varphi, 3} \) and a minimal solution \( m \) for \( T_F \), the corresponding expanded template \( T_m \) has an exponential number of vertices (in the size of \( \varphi \), i.e \( O(\beta \cdot |x|^2) \)). The new vertices are denoted using an exponential number of index variables \( I_m \cup I_m \) and an exponential number of location variables...
5.2. COMPOSITE STRUCTURE LOGIC. CSL

The size of the resulting formula is exponential in the size of $\varphi$ because $T$ has a polynomial number of edges, and each atomic formula associated with an edge is replaced by an exponential number of atomic formulas. Choosing an equivalence relation over the fresh variables, is done in polynomial time w.r.t. the size of the current formula. We have to choose an exponential number of transitive relations over the fresh variables (2$|\mathcal{PF}(\varphi)|$) but, as before, they are determined in a unique way by a linear number of transitive relations. Each transitive relation can be chosen in polynomial time w.r.t. the size of the current formula and exponential time in the size of $\varphi$. Consequently, to represent these transitive relations, the size of the formula grows with $\mathcal{O}((|\mathcal{PF}(\varphi)| \cdot (\beta \cdot |x|^2)^2)$. Overall, the complexity of the second step is non-deterministic exponential time in the size of $\varphi$, and the size of the formula is exponential in the size of $\varphi$.

Step 3: To transform the universal quantifications into conjunctions, we need to enumerate all the mappings of universal variables into positions of the expanded template. The number of these mappings is doubly-exponential in the size of the initial formula (this is implied by the exponential bound on the solutions of MOILP). If the number of universal variables is fixed then, the number of mappings becomes only single-exponential in the size of $\varphi$.

Step 4: The elimination of existentially quantified variables leads to a formula in $\text{FO}(D, O, P)$ whose size is doubly exponential in the size of $\varphi$.

5.2.5 Deciding the satisfiability problem of CSL

In this section we give the generalization of Theorem 5.2.1 to CSL.

**Theorem 5.2.3.** The satisfiability of a CSL formula can be reduced to the satisfiability of a formula in the underlying data logic $\text{FO}(D, O, P)$.

**Proof.** Let $\varphi = \exists a^k \exists i^k \exists x^k \exists d^k \forall b^k \forall j^k \forall y^k Q_{k-1} \ldots Q_1 Q. \psi$ be a CSL$_k$ formula. We suppose w.l.o.g. that $\varphi$ is closed (otherwise, the free variables are existentially quantified at the beginning of the formula). The main idea of the proof is that levels can be analyzed in a top down manner. The restriction $\text{Lev}$ implies that the constraints on the inferior levels do not influence the size of the models on the superior levels. More precisely, we can compute a bound on the models, by computing first a bound on the vertices of level $k$, then we use this bound to define bound on the vertices of level $k - 1$ and so on until we define a bound on the entire model. The steps of the proof are the following:

1. using a small model property, we prove that for any $k \geq 1$ and any formula $\phi \in \text{CSL}_k$ we can compute an equi-satisfiable formula without universally quantified location, array and index variables of level $k'$, where $k' \leq k$ is the greatest level of universally quantified variables in $\phi$; for $\varphi$ we obtain that is equi-satisfiable with

   \[ \exists a^{k'} \exists i^{k'} \exists x^{k'} \exists d^{k'} Q_{k-1} \ldots Q_1 Q. \psi' ; \]

2. we apply iteratively the universal quantifier elimination in step 1, until we end up with an equi-satisfiable formula of level $k$ without universally quantified location, array and index variables;
3. we eliminate all existentially quantified location, array and index variables and we obtain an equi-satisfiable formula in $\mathsf{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$.

The first step of the proof is showed by the following lemma:

**Lemma 5.2.4.** For every $k \geq 1$, the satisfiability of a $\mathsf{CSL}_k$ formula $\varphi$ can be reduced to the satisfiability of a $\mathsf{CSL}_{k'}$ formula without universal quantification of level $k'$, where $k' \leq k$ is the maximal level of the universally quantified variables in $\varphi$.

**Proof.** Let’s consider first the $k = k'$ and let

$$\varphi = \exists a^k \exists i^k \exists x^k \forall b^k \forall j^k \forall y^k Q_{k-1} \ldots Q_1 Q. \psi$$

We suppose w.l.o.g. that $\varphi$ is closed (otherwise, the free variables are existentially quantified at the beginning of the formula).

$\varphi$ can relate vertices of different levels only by atomic formulas of the form $x \xrightarrow{H} y$, where $H$ is set of non-recursive pointer fields, and by data constraints.

Consequently, the shape of a model of $\varphi$ restricted to vertices of level $k$ can be obtained independently of the shape of the inferior levels. We can define a set of templates, $\mathcal{T}_{\varphi, \exists, k}$, representing the level $k$ of the models of $\varphi$, similarly to $\mathcal{T}_{\varphi, \exists}$ in the proof of Theorem 5.2.1. These templates represent equivalence and transitive relations between terms, respectively atomic formulas, of level $k$. That is, they represent evaluations for all atomic formulas of the form $\ell(x_k)$, $x_k = t_k$, $x_k \xrightarrow{A.B} x_{k'}$, $x_k \xrightarrow{A} x_{k'}$, $x_k \xrightarrow{H} x_{k'}$, $E_k < ct$, and $E_k = ct$, where $x_k$ and $x'_{k}$ are location variables of level $k$.

In the following a template that represent an equivalence relation and a transitive relation over terms of level $k$ is called *template of level $k$.*

Similarly to $\mathsf{CSL}_1$, for every $T \in \mathcal{T}_{\varphi, \exists, k}$, a template of level $k$, we compute the set of minimal solutions, denoted $\mathcal{M}_T$, for the length constrains $\mathcal{I}_T$. For every $T \in \mathcal{T}_{\varphi, \exists, k}$ and $m \in \mathcal{M}_T$, let $\tilde{T}_{\varphi, T, m, k}$ be the set of expanded templates defined like in the case of $\mathsf{CSL}_1$. Using a generalization of Lemma 5.2.2 to any level $k$ instead of level 1, we obtain that $\varphi$ is equivalent to

$$\bigvee_{T \in \mathcal{T}_{\varphi, \exists, k}} \bigvee_{m \in \mathcal{M}_T} \bigvee_{T' \in \mathcal{T}_{\varphi, T, m, k}} \exists a^k \exists i^k \exists x^k \exists x^k_m \exists j^k \exists y^k \forall \psi_{T, k, m}^{T, \exists, k} (5.2.32)$$

where $t^k$ is the set of index variables from the definition of $T$, $x^k_m$ and $i^k_m$ denote the vertices added to the template $T$, and $\psi_{T, k, m}^{T, \exists, k}$ is the conjunction between the formula describing $\tilde{T}$, i.e. $\rho_{\tilde{T}} \land \mathcal{I}_{\tilde{T}}$, and the formula $\psi$ where the atomic formulas over existentially quantified location, array and index variables are replaced by *true* or *false* according to the relations represented by $\tilde{T}$.

Suppose that $\varphi$ is satisfiable and let $G$ be a model for it. Then, there exists $\tilde{T} \in \mathcal{T}_{\varphi, T, m, k}$ such that $G$ is a model for the disjunct of (5.2.32) corresponding to $\tilde{T}$. From $G$ we can build a “small model with respect to level $k$” for $\varphi$ as follows:

1. we copy all the vertices from levels less than $k$, together with all the edges between them;

2. on level $k$ we copy only the vertices which correspond to vertices in $\tilde{T}$ and we edges between them, by preserving the paths between these vertices;
3. we copy all the edges from $G$ between vertices on levels less than \( k \) and the vertices of level \( k \) selected in the previous step.

This is possible because the rest of the formula (speaking about inferior levels) cannot impose the existence of supplementary positions on level \( k \).

Notice that the size of level \( k \) of this model is bounded by the number of existentially quantified location and array variables of level \( k \) (i.e. \( a_k \cup \{ x' \leq k : \tau(x) = k \} \cup x_m \)).

Having a model bounded on level \( k \) we can eliminate the universally quantified variables of level \( k \) by mapping them in all possible ways over the existentially quantified ones.

If \( k > k' \) then we build the templates for every level between \( k \) and \( k' \) (inclusive) and then we apply the quantifier elimination. Actually, when the formula has only existential quantified variables of levels \( k, k-1, \ldots, k'+1 \), we can consider it as a formula of level \( k' \), i.e. we consider that all record types of level \( k, \ldots, k'+1 \) are record types of level \( k' \).

The second step consists in applying Lemma 5.2.4 until a CSL\(_k\) formula without universally quantified location, array and index variables is obtained. To obtain an equi-satisfiable FO(\(D, O, P\)) we apply the same technique to eliminate existentially quantified variables as for CSL\(_1\) (see Section 5.2.3.6).

Let \( \exp_t(n) \) be the \( t \)-fold iterated exponential function with base 2: \( \exp_0(n) = n \) and \( \exp_{t+1}(n) = 2^{\exp_t(n)} \), for any \( t \geq 0 \). The following result is a direct consequence of Theorem 5.2.2.

**Theorem 5.2.4.** Let \( \varphi \) be a closed CSL\(_k\) formula. The satisfiability of \( \varphi \) can be reduced to the satisfiability of a FO(\(D, O, P\)) formula \( \varphi' \) in nondeterministic \( O(\exp_{2k}(n)) \) time, where \( n \) is the size of \( \varphi \). Moreover, the size of \( \varphi' \) is \( O(\exp_{2k}(n)) \).

### 5.2.6 Closure of CSL under post image computation

We call basic statement any program statement of the class of programs defined in Section 4.2.1 that is different from a while loop. Then, we can prove the following fact.

**Theorem 5.2.5.** For any basic statement \( S \) and any CSL formula \( \varphi \), post\((S, \varphi)\) is CSL-definable and it can be computed in linear time.

The theorem above is important for carrying out pre/post-condition reasoning (assuming we have an annotated program with assertions) and for inductive invariance checking (assuming we have an annotated program with loop invariants).

Let \( \varphi(x, i, a) \in CSL \) be a formula with free variables \( x \in Loc \cup Data, i \in Ind \) and \( a \in Arr \), representing the pointer, data, index and array variables of the program. The set \( x \) contains a special variable \( null \), for each record type \( Rt \in RT \). This variable is always interpreted into a vertex, s.t. there is a loop formed of a single edge labeled by all the pointer fields in this vertex. We give in the following the definition of post computation for the statements of our programs. We use the notation \( \varphi[v/x] \) for syntactic substitution of \( x \) by \( v \). This notation is useful in any computation to keep track of the old value of \( x \).
CHAPTER 5. PROGRAM VERIFICATION USING FRAGMENTS OF GCSL

New statements: The statement \( x = \text{new } Rt() \) where \( x \) is a pointer variable of type \( Rt \), creates a new location, i.e., a location (1) which is not pointed by any field, (2) whose pointer/array fields are set to \text{null}, and (3) whose data field is set to the default value of \( \mathbb{D} \).

\[
\begin{align*}
\text{post}(x = \text{new }(), \varphi) & \equiv x \neq \text{null} \land \text{isolated}(x) \land \exists v. \varphi[v/x], \\
\text{isolated}(x) & \equiv \bigwedge_{f:Rt \rightarrow Rt'} \text{isolated}(x, f) \land \bigwedge_{f':Rt' \rightarrow Rt} \text{isolated}(x, f') \\
\text{default}(x, dt) & \equiv dt(x) = c_{\mathbb{D}}
\end{align*}
\]

The number of atomic formulas of \( \text{post} \) is linear in the number of atomic formulas of \( \varphi \) and the number of pointer fields, i.e., for each pointer field an atomic formula and at most one quantified variable are added to \( \varphi \).

When \( a \) is of type pointer to an array, the statement \( a = \text{new } Rt[c] \) creates an array of object of type \( Rt \) with at least \( c \) elements.

\[
\begin{align*}
\text{post}(a = \text{new } Rt[c], \varphi) & \equiv a[0] \neq \text{null} \land c \geq 0 \land (\exists e. e = a[c - 1]) \land \exists v. \varphi[v/a].
\end{align*}
\]

Free statements: The effect of freeing \( x \) of type pointer to record is make \( x \) to point towards the special node \( \text{null} \); the precondition of this operation is that \( x \) was not already freed, i.e., equal to \( \text{null} \) (\( v \) is a location variable representing the old value of \( x \)).

\[
\begin{align*}
\text{post}(\text{free}(x), \varphi) & \equiv x = \text{null} \land \exists v. (\varphi[x/v] \land v \neq \text{null}).
\end{align*}
\]

Analogously, if \( x \) is a pointer to an array, the first node of the array, \( x[0] \), becomes equal to \( \text{null} \) (\( v \) is an array variable representing the old value of \( x \)).

\[
\begin{align*}
\text{post}(\text{free}(x), \varphi) & \equiv x[0] = \text{null} \land \exists v. (\varphi[x/v] \land v[0] \neq \text{null}).
\end{align*}
\]

Variable assignment: Let us consider now assignments of the form \( x = t \). For assignment with program variables, we use the equality between terms. For assignment with a record field, we use either (pointer or data) term equality, or the reachability predicate. For assignment with an array element, we use a special field \( \delta \) defined for each array and modeling the array dereference operation; then \( \delta(a[i]) \) represents the data attached to the
5.2. COMPOSITE STRUCTURE LOGIC. CSL

Field assignment: The strongest post-condition computation for the statement $x\rightarrow f=y$ is straightforward when $f \notin \mathcal{PF}_r$. If $f$ is a recursive field, this computation is not trivial because of the reachability predicate $\mathcal{A}_{df}$. Indeed, a local change in the field $f$ of $x$ may produce a global change in the value of this predicate.

The simplest case is when $f$ is a data field, i.e., $f \in \mathcal{DF}$:

$$
\text{post}(x\rightarrow f=y, \varphi) \equiv x \neq \text{null} \land f(x) = y \land \exists v. \varphi[x/v]
$$

The application $\Pi$ goes through the formula $\varphi$ and changes atomic formulas of the form $r(dt_1, \ldots, dt_n)$ build over some term $f(w)$, by replacing it with the variable $v$ when $w = x$.

$\Pi$ transforms only the atomic formulas $r(dt_1, \ldots, dt_n)$:

$$
\Pi(r(dt_1, \ldots, dt_n), x, f, v, y) \equiv \begin{cases} 
\psi[u_i] = \Pi(dt_1, x, f, v, y)[u_1], \ldots, \\
\psi[u_n] = \Pi(dt_n, x, f, v, y)[u_n] \text{ with each } u_i \text{ fresh} \\
\text{in } r(u_1, \ldots, u_n) \land \bigwedge_{1 \leq i \leq n} \psi[u_i]
\end{cases}
$$

$$
\Pi(f(w), x, f, v, y)[u] \equiv (w = x \land u = v) \lor (w \neq x \land u = f(w))
$$

$$
\Pi(o(dt'_1, \ldots, dt'_n), x, f, v, y)[u] \equiv \begin{cases} 
\psi[w_i] = \Pi(dt'_1, x, f, v, y)[w_1], \ldots, \\
\psi[w_n] = \Pi(dt'_n, x, f, v, y)[w_n] \text{ with each } w_i \text{ fresh} \\
\text{in } u = o(w_1, \ldots, w_n) \land \bigwedge_{1 \leq i \leq n} \psi[w_i]
\end{cases}
$$

Notice that, the size of the strongest post-condition is linear in the size of $\varphi$.

In the case of a recursive field $f$, i.e., $f \in \mathcal{PF}_r$, the assignment is more technically involved. We add an existentially quantified variable $v_{f(x)}$ representing the old value of
CHAPTER 5. PROGRAM VERIFICATION USING FRAGMENTS OF GCSL

\( f(x) \) and a set of existentially quantified index variables \( I \), and we apply a transformation \( \Gamma \) to \( \varphi \) in order to compute the global change of \( \varphi \):

\[
\text{post}(x \rightarrow f = y, \varphi) \equiv x \neq \text{null} \land x \xrightarrow{f} y \land \exists v_{f(x)} \exists I. \Gamma(\varphi, f, x, v_{f(x)}, y, I)
\]

The application \( \Gamma \) is defined inductively on formulas. It captures the effect of replacing the old value of \( f(x) \) represented by \( v_{f(x)} \) with the value of \( y \). Its definition is not straightforward for the reachability predicates. We assume w.l.o.g. that negations are pushed on atomic formulas. Then, for positive occurrences of the reachability predicate, \( \Gamma \) is defined by:

\[
\Gamma(v_1 \overrightarrow{AB} v_2, f, x, v, y) \equiv \\
\text{if } f, f \notin A \text{ or } x \in B \text{ then } v_1 \overrightarrow{AB} v_2 \\
\text{else if } f \in A \text{ (for } f \in A \text{ take } v_2 \overrightarrow{AB} v_1) \\
\quad v_1 \overrightarrow{AB} (x) \rightarrow v_2 \\
\quad \lor v_2 = x \land v_1 \neq v_2 \land v_1 \overrightarrow{AB} v_2 \\
\quad \lor v_1 = x \land v_1 = v_2 \land v_1 \overrightarrow{AB \setminus f} v_2 \land v \overrightarrow{AB} v_2 \land x \overrightarrow{A \setminus f_1} v \\
\quad \lor v_1 = x \land v_1 = v_2 \land v_1 \overrightarrow{AB \setminus f} v_2 \land v \overrightarrow{AB \setminus f} v_2 \land x \overrightarrow{A \setminus f_1} v \\
\quad \lor v_1 \neq x \land v_2 \neq x \land v_1 \overrightarrow{AB} x \land v_1 \overrightarrow{AB \setminus f} v_2 \land v \overrightarrow{AB \setminus f} v_2 \land x \overrightarrow{A \setminus f_1} v
\]

The two first disjuncts capture the cases where no change in the path between \( v_1 \) and \( v_2 \) due either to the absence of \( x \) on this path or to \( x \) being the last node of a path without cycle. The third disjunct captures the case where the path between \( v_1 \) and \( v_2 \) is a cycle and \( x \) points to the first location of the path. Then, if \( f \) is not the only field of the path between \( v_1 \) and \( v_2 \), the path is still there but label \( f \) is removed. The path between \( v \) and \( v_2 \) is either reduced to a point or it stays labeled by \( A \). These two properties are captured respectively by the two predicates below which are extensions of the reachability predicate:

\[
\overrightarrow{AB} v \equiv \text{ if } A = \emptyset \text{ then } \text{true else } v \overrightarrow{AB} v \\
\overrightarrow{-A} v \equiv v = v \lor (v \neq v \land u \overrightarrow{AB} v)
\]

Also, the location \( v \) remains the successor of \( x \) for fields in \( A \) different from \( f \). When \( A = \{f\} \), it is not possible to exactly identify \( v \), except when \( v \) (i.e., \( x \rightarrow f \)) is one of the program variable. However, the latter is very usual in programs: before destroying a link in a list, the programmer put the target of the link in a program variable for further use. The fourth disjunct is similar to the third but for a path without cycle. Then, we have to capture that the remaining path between \( v \) and \( v_2 \) does not goes through \( x \). The last case captures the case where \( x \) and \( v \) are between \( v_1 \) and \( v_2 \).

The syntax of CSL does not allow negative occurrences of the predicate \( \overrightarrow{-A} \).

Next, we give the definition of \( \Gamma \) on reachability predicates with lengths:
5.2. COMPOSITE STRUCTURE LOGIC. CSL

\[ \Gamma(v_1^{A,B,l} \rightarrow v_2, f, x, v, y) = \]

- if \( f, \vec{f} \notin A \) or \( \{x\} \in B \) then \( v_1^{A,B,l} \rightarrow v_2 \)
- else case \( f \in A \) (for \( f \in A \) take \( v_2^{A,B,l} \rightarrow v_1 \))
  \[ v_1^{A,B,l} \rightarrow v_2 \]
  \[ \lor \quad v_2 = x \land v_1 \neq v_2 \land v_1^{A,B,l} \rightarrow v_2 \]
  \[ \lor \quad v_1 = x \land v_1 = v_2 \land v_1^{A,f,l} \rightarrow v_2 \land v_1^{A,f,l} \rightarrow v_2 \land \]
  \[ l_1 + l_2 + 1 = l \land x^{A,f,l} \rightarrow v \]

Otherwise, if \( f \) is a non-recursive pointer field, i.e. \( f \in \mathcal{PF} \setminus \mathcal{PF}_r \), then \( \Gamma \) transforms positive the atomic formulas \( u^{f,l} \rightarrow w \) as follows:

\[ \Gamma(v_1^{A \rightarrow} v_2, f, x, v, y) = \]

- if \( f, \vec{f} \notin A \) or \( x \in B \) then \( v_1^{A \rightarrow} v_2 \)
- else if \( f \in A \) (for \( f \in A \) take \( v_2^{A,B,l} \rightarrow v_1 \))
  \[ v_1^{A \rightarrow} v_2 \land v_1 \neq x \]
  \[ \lor \quad v_1 = x \land v_1^{A \rightarrow} v_2 \]

For negative occurrences of the predicate \( \cdot^{A \rightarrow} \cdot \), with \( A \subseteq \mathcal{PF} \setminus \mathcal{PF}_r \cup \overline{\mathcal{PF}} \setminus \overline{\mathcal{PF}_r} \), \( \Gamma \) introduces the case (second disjunct below) when the field assignment \( x \rightarrow f \neq y \) creates a path (of label \( f \)) between \( v_1 \) and \( v_2 \):

\[ \Gamma(- (v_1^{A \rightarrow} v_2), f, x, v, y) = \]

\[ (- (v_1^{A \rightarrow} v_2) \land x \neq v_1) \]
\[ \lor \quad v_1^f \neq v_2 \land v_1 = x \land v_2 = y \land (v_1^{A,f,l} \rightarrow v_2) \]

Index assignment: Due to the constraints on existential indices, the assignment of indices is trivial to compute:

\[ \text{post}(i^{=ie}, \varphi) \equiv (i - ie) = 0 \land \exists i'. \varphi[i'/i] \]

Boolean conditions: The application \( \alpha \) translates boolean expressions into CSL formulas; this translation introduces fresh variables in \( v \).

\[ \text{post}(be, \varphi) \equiv \exists v \varphi \land \alpha(\text{be}) \]

where \( \alpha \) is defined by:

\[ \alpha(t = t') \equiv v = v' \land \beta(t, v) \land \beta(t', v') \quad \alpha(\text{be} \& \text{be}') \equiv \alpha(\text{be}) \land \alpha(\text{be}') \]
\[ \alpha(ie = ie') \equiv ie - ie' = 0 \quad \alpha(\text{! be}) \equiv \neg \alpha(\text{be}) \]
\[ \alpha(ie < ie') \equiv ie - ie' < 0 \]

The application \( \beta \) builds a formula from a location term \( t \).

\[ \beta(x, v) \equiv x = v \quad \beta(x[i], v) \equiv i \geq 0 \land x[i] = v \]

\[ \beta(x \rightarrow f, v) = \begin{cases} x \rightarrow f, v, & \text{if } f \in \mathcal{PF} \setminus \mathcal{PF}_r \\ x(f), \text{else} \rightarrow v, & \text{otherwise} \end{cases} \]

141
CHAPTER 5. PROGRAM VERIFICATION USING FRAGMENTS OF GCSL

**Assertion checking:** Let \( \varphi \) be an ICSL formula over the type system \( \Sigma \). The strongest post-condition w.r.t. an `assert` statement is defined by:

\[
\text{post}(\text{assert } \varphi, \psi) \equiv \begin{cases} 
\psi, & \text{if } \psi \implies \varphi \\
\text{false}, & \text{otherwise.}
\end{cases}
\]

Now, given a CSL formula \( \varphi \), the strongest post-condition w.r.t. an `assume` statement is defined by:

\[
\text{post}(\text{assume } \varphi, \psi) \equiv \varphi.
\]

In the following, we define the strongest postcondition w.r.t. an `assume proc` statement. Let \( P_\pi \) be a procedure of \( P_\Pi \) and let \( (\varphi_{\text{pre}}, \varphi_{\text{post}}) \) be a pair of CSL formulas which give the specification of \( P_\pi \) (\( \varphi_{\text{pre}} \) is the pre-condition and \( \varphi_{\text{post}} \) is the post-condition). Recall that any call \( P_\pi(\mathbf{a}, \mathbf{a}_o) \) is replaced by `assert \( \varphi_{\text{pre}} \); assume proc \( \varphi_{\text{post}}`, where \( \varphi_{\text{pre}} \) and \( \varphi_{\text{post}} \) are obtained from \( \varphi_{\text{pre}} \) and respectively \( \varphi_{\text{post}} \), by replacing formal parameters with actual parameters.

Roughly, \( \text{post}(\text{assume proc } \varphi_{\text{post}}, \psi) \) replaces the values of the actual parameters from the context of the call \( \psi \) with the ones described by the post-condition of the procedure \( P_\pi, \varphi_{\text{post}} \), and preserves the constraints in \( \psi \) that do not concern the actual parameters, i.e. the constraints on the non-local part of the context of the call. It corresponds to the semantics of the concrete transformer for `return P_\pi(\mathbf{a}, \mathbf{a}_o)`. Formally, the output of \( \text{post}(\text{assume proc } \varphi_{\text{post}}, \psi) \), denoted by \( \psi' \), is obtained by:

1. we rename in \( \varphi_{\text{post}} \) the variables which represent actual parameters into their primed version, that is, every \( a \in \mathbf{a} \cup \mathbf{a}_o \) is renamed into \( a' \).

2. in the formula \( \varphi_{\text{post}} \), we make explicit the successors of the actual input parameters w.r.t. all pointer fields. To this, we introduce a distinguished set of pointer variables \( p_{a',f} \), for any primed actual input parameter \( a' \) in \( \varphi_{\text{post}} \) and any pointer field \( f \) in the record type of \( a' \). The variable \( p_{a',f} \) represents the successor of the primed actual input parameter \( a' \) w.r.t. the pointer field \( f \). The values of these variables are defined by the strongest postcondition of \( \varphi_{\text{post}} \) w.r.t. a program that contains all the statements of the form \( p_{a',f} = a' \rightarrow f \), where \( a' \) and \( f \) are as above. This program is denoted by \( P_s \). We define:

\[
\varphi_1 = \text{post}(P_s, \varphi_{\text{post}})
\]

(we use the same name for the extension of post to sequences of basic statements).

Similarly, we introduce fresh data variables \( d_{a',dt} \), for any primed actual input parameter \( a' \) in \( \varphi_{\text{post}} \) and any data field \( dt \) in the record type of \( a' \). The variable \( d_{a',dt} \) denotes the value of the data field \( dt \) in the object pointed to by the parameter \( a' \). The values of these variables are defined by the strongest postcondition of \( \varphi_1 \) w.r.t. a program that contains all the statements of the form \( d_{a',dt} = a' \rightarrow dt \), where \( a' \) and \( f \) are as above. If we denote this program by \( P_d \) then we define:

\[
\varphi_2 = \text{post}(P_d, \varphi_1).
\]

Notice that \( \varphi_2 \) and \( \varphi_{\text{post}} \) have the same set of models except for the labeling of vertices. The formula \( \varphi_2 \) contains fresh variables that are interpreted into vertices that already exists in the models of \( \varphi_{\text{post}} \).
5.2. COMPOSITE STRUCTURE LOGIC. CSL

3. to combine the post-condition with the context of the call, we define $\varphi^3 = \varphi^2 \land \psi$;

4. we change the values of the pointer and data fields of the actual input parameters (from the context of the call) to match the ones from the output of the procedure. This is done by computing the strongest postcondition w.r.t. a program that contains all the statements of the form $a \rightarrow f = p_{a',f}$ and $a \rightarrow dt = d_{a',dt}$ with $a$ an actual input parameter, $f$ a pointer field in the record type of $a'$, and $dt$ a data field in the record type of $a'$. Let $P_{ni}$ denote this program. We define:

$$\varphi^4 = \text{post}(P_{ni}, \varphi^3).$$

5. we change the values of the actual output parameters according to the post-condition of the procedure. This is done by computing the strongest post-condition w.r.t. a program that contains all the assignments of the form $ao = ao'$, where $ao$ is some actual output parameter. If $P_{ao}$ denotes this program, then:

$$\varphi^5 = \text{post}(P_{ao}, \varphi^4).$$

6. finally, $\psi'$ is obtained from $\varphi^5$ by eliminating all the primed actual parameters (they are assigned to $\text{null}$). Thus, if $P'$ denotes the program that contains all the statements of the form $a' = \text{null}$, where $a'$ is a primed actual parameter, then

$$\psi' = \text{post}(P', \varphi^5).$$

5.2.7 Examples

Program Dispatch: Let us consider the program Dispatch give in Figure 4.5. Let us assume that the while loop is annotated with a formula, $\varphi_{\text{disp Inv}}$, saying that the list pointed by gr, respectively by sm, is either empty or all its elements have the values of their data fields greater than, respectively less than or equal to, $v$:

$$\varphi_{\text{disp Inv}} = \left( (\text{gr-singly-ll} \land \text{gr-greater-v}) \lor (\text{gr} = \text{null}) \right) \land \left( (\text{sm-singly-ll} \land \text{sm-less-v}) \lor (\text{sm} = \text{null}) \right).$$

We can prove that $\varphi_{\text{disp Inv}}$ is an inductive invariant for the loop, and using it we can prove the assert statement at line 34.

Program AddV: For the program AddV given in Figure 4.19 using CSL we are able to prove a weaker specification for the procedure addV:

$$\varphi_{\text{pre}}^{\text{add}} ::= \left( \forall y. (\text{head}_{\text{next}} \rightarrow \text{null} \land (\text{head}_{\text{next}} \rightarrow y \implies dt(y) \leq 0)) \right) \lor \text{head} = \text{null}$$

$$\varphi_{\text{post}}^{\text{add}} = \left( \forall y. (\text{head}_{\text{next}} \rightarrow \text{null} \land (\text{head}_{\text{next}} \rightarrow y \implies dt(y) \leq v)) \right) \lor \text{head} = \text{null}$$

The specification given in Section 4.4.3 it is not a CSL formula (due to the length constraints over universally quantified location variables).

Program Quicksort: Another procedure whose specification we could prove if annotated with the appropriate invariant (similar with the loop invariant $\varphi_{\text{disp Inv}}$ of the program Dispatch) is the specification of the procedure split in Figure 4.21. Still, we are not able to prove the specification of quicksort. In terms of sorting algorithms, we are able to prove for example the partial correctness of classical implementation of insertion sort (over lists or arrays) (annotated with a corresponding invariant).
**Program Insert:** Let us focus in the following, on proving the partial correctness for the program in Figure 4.3. This program contains an infinite loop which accesses in a random manner one of the array elements and its corresponding doubly-linked list and it inserts an element on the second position of the list. We aim to prove that every iteration of the loop preserves structure described by loop assertion, i.e. a points to an array of doubly-linked lists. The array of doubly-linked lists can be described in a CSL fragment defined over the ordered partition considered in Section 5.2.1, that is, over \( \sigma : \{ \text{dll}_\text{ty}, \text{a}_\text{ty} \} \rightarrow \{1, 2\} \) given by \( \sigma(\text{dll}_\text{ty}) = 1 \) and \( \sigma(\text{a}_\text{ty}) = 2 \). More precisely, it can be described by the formula \( \varphi = \varphi_1 \land \varphi_2 \land \varphi_3 \), where:

- \( \varphi_1 \) says that a is an array of 10 elements:
  \[
  \varphi_1 \equiv \exists i \exists loc \exists val. i = 10 \land a[i] = loc \land id(loc) = val,
  \]
  where \( i \) is some index variable, \( loc \) is some location variable, and \( val \) is some data variable. Using syntactic sugar, it can be rewritten as \( \varphi_1 \equiv \exists val. \text{id}(a[10]) = val \).

- \( \varphi_2 = \varphi'_2 \land \varphi''_2 \) says that each array element has a dll link towards a non-empty doubly-linked list. \( \varphi'_2 \) requires that each array element is linked by dll to a location from which we have a list using next towards null and a list using \( \text{prev} \) towards null (null is some distinguished location variable):
  \[
  \varphi'_2 \equiv \forall j \exists x. a[i] \xrightarrow{\text{dll}} x \land x \neq \text{null} \land x \{\text{next}\} \xrightarrow{\text{null}} x \land x \{\text{prev}\} \xrightarrow{\text{null}} x
  \]
  This formula is clearly not sufficient. We use \( \varphi''_2 \) to impose that the lists starting from \( x \) form a doubly-linked list:
  \[
  \varphi''_2 \equiv \forall j \forall y \forall q. (a[j] \xrightarrow{\text{dll}} y \land y \{\text{next}\} \xrightarrow{\text{null}} q \land q \neq \text{null}) \implies y \{\text{next, prev}\} \xrightarrow{q} q
  \land \forall j \forall y \forall q. (a[j] \xrightarrow{\text{dll}} y \land y \{\text{prev}\} \xrightarrow{\text{null}} q) \implies q = \text{null}
  \]

- \( \varphi_3 \) says that all the elements of the doubly-linked lists are linked by root to the corresponding node from the array a:
  \[
  \varphi_3 = \forall j \forall x, z. \quad (a[j] \xrightarrow{\text{dll}} x \land x \neq \text{null}) \implies
  (x \xrightarrow{\text{root}} a[j] \land (x \{\text{next}\} \xrightarrow{\text{null}} z \land z \neq \text{null}) \implies (z \xrightarrow{\text{root}} a[i]))
  \]

We want to prove that before executing the statement at line 20, the configurations of the program are models for the formula \( \varphi \). That is, the configurations of the program satisfy:

\[
\text{Prop} \equiv pc = 20 \implies \varphi,
\]

where the index variable \( pc \) models the program counter. We suppose that before entering the loop, the heap has the structure from above but, each doubly-linked list contains exactly two elements. This is expressed by the formula \( \text{Init} \equiv pc = 17 \land \varphi_1 \land \varphi_2 \land \varphi_3 \land \varphi_4 \), where \( \varphi_4 \) characterizes the length of the lists:
5.2. COMPOSITE STRUCTURE LOGIC. CSL

\[ \varphi_4 \equiv \forall j \exists z, t \forall y. a[j] \xrightarrow{\text{dll}} z \land z^{\text{next,prev}} \land t \neq \text{null} \land z \neq \text{null} \]

\[ \land t^{\text{next,prev}} \land \text{null} \land z \xrightarrow{\text{root}} a[j] \land t \xrightarrow{\text{root}} a[j]. \]

The formula Init is given as an annotation before line 17. To prove (5.2.33), we need the annotation loop invariant Inv before line 20 which gives a candidate for an inductive invariant of the loop. We consider Inv = \( \varphi_1 \land \varphi_2 \land \varphi_3 \land \phi \), where \( \phi = \land_{1 \leq i \leq 7} \phi_i \) with:

\[ \phi_1 = (pc \geq 22) \implies v[k] \xrightarrow{\text{dll}} \text{tmp1}, \]

\[ \phi_2 = ((pc \geq 23 \land pc \leq 26) \implies \text{succ} (\text{tmp1}, \{\text{next}\}, \text{tmp2})) \land \]

\[ ((pc \geq 23) \implies (\forall y. (\text{tmp2}^{\text{next}} \implies y \land y \neq \text{null}) \implies (\text{tmp2}^{\text{next,prev}} \implies y \land y \neq \text{null}) \land y \longrightarrow \text{a[k]})) \]

\[ \phi_3 = (pc = 24) \implies (\text{node} \neq \text{null} \land \forall y. (\neg \text{node}^{\text{next}} \implies y \land \neg \text{node}^{\text{prev}} \implies y)), \]

\[ \phi_4 = (pc \geq 25) \implies \text{succ} (\text{node}, \{\text{prev}\}, \text{tmp1}), \]

\[ \phi_5 = (pc \geq 26) \implies \text{node} \xrightarrow{\text{root}} \text{a[k]}, \]

\[ \phi_6 = (pc \geq 27) \implies \text{tmp1}^{\text{next,prev}} \implies \text{node} \land \forall z. (\text{tmp1}^{\text{next}} \implies z) \implies (\text{node}^{\text{next,prev}} \implies z) \]

\[ \phi_7 = (pc \geq 28) \implies \text{succ} (\text{tmp2}, \{\text{prev}\}, \text{node}), \]

where succ(x, f, y) is a predicate saying that x and t are successors w.r.t. the pointer field f, that is:

\[ \text{succ}(x, f, y) \equiv x \xrightarrow{f} y \land \forall z. x \xrightarrow{f} z \implies (y = z \lor y \xrightarrow{f} z). \]

If we prove that Inv is an inductive invariant then, clearly, Inv \implies \text{Prop} and we are finished. We discuss the main features of this proof in the rest of this section.

First, we must prove that Init \implies Inv is valid, that is, Init \land \neg(\varphi_1 \land \varphi_2 \land \varphi_3 \land \phi), is unsatisfiable. Thus, we must prove that

\[(\text{Init} \land \neg \varphi_1) \lor (\text{Init} \land \neg \varphi_2) \lor (\text{Init} \land \neg \varphi_3) \lor (\text{Init} \land \neg \phi)\]

is unsatisfiable. The first three disjuncts are clearly unsatisfiable, while the fourth is unsatisfiable because \neg\phi requires that pc \geq 22 which contradicts the atomic formula pc = 17 from Init.

Second, we must prove that post(S, Inv) \implies Inv, for any statement S of the loop. After the first statement of the loop, k receives some random value within the array bounds, that is,

\[ \text{post}(k = \text{rand}() \mod 10, \text{Inv}) \equiv \exists k'. \exists c. \text{Inv}[k \leftarrow k'] \land 1 \leq c \land c \leq 10 \land k = c \land pc = 21. \]

To show that post(k = \text{rand}() \mod 10, Inv) \implies Inv is valid, we must prove that

\[(\text{post}(k = \text{rand}() \mod 10, \text{Inv}) \land \neg \varphi_1) \lor (\text{post}(k = \text{rand}() \mod 10, \text{Inv}) \land \neg \varphi_2) \lor (\text{post}(k = \text{rand}() \mod 10, \text{Inv}) \land \neg \varphi_3) \lor (\text{post}(k = \text{rand}() \mod 10, \text{Inv}) \land \neg \phi)\]

is unsatisfiable. Since none of the formulas \varphi_1, \varphi_2, or \varphi_3 do not characterize the value of k, the unsatisfiability of the first three disjuncts given above can be obtained
very easily. With respect to the fourth disjunct we obtain a contradiction because 
\( \text{post}(k = \text{rand}() \mod 10, \text{Inv}) \) implies \( pc = 21 \) and \( \neg \phi \) requires \( pc \geq 22 \).

A more interesting case is the one corresponding to the statement at line 27. We have to show that 
\( \text{post}(\text{tmp2} \rightarrow \text{prev} = \text{node}, \text{Inv}) \land \neg \text{Inv} \) is unsatisfiable. We have that

\[
\text{post}(\text{tmp2} \rightarrow \text{prev} = \text{node}, \text{Inv}) \equiv \text{tmp2} \neq \text{null} \land \text{succ}(\text{tmp2}, \{\text{prev}\}, \text{node}) \land pc = 16 \land \Gamma(\text{Inv}, \text{prev}, \text{tmp2}, \text{tmp1}, \text{node}),
\]

where \( \Gamma \) is a function that transforms the formula \( \text{Inv} \) recursively on its structure (\( \text{tmp1} \) is given as an argument because it represents the old value of the pointer field \( \text{prev} \) in \( \text{tmp2} \)).

According to the definition of \( \Gamma \) we have that

\[
\Gamma(\text{Inv}, \text{prev}, \text{tmp2}, \text{tmp1}, \text{node}) = \Gamma(\varphi_1, \text{prev}, \text{tmp2}, \text{tmp1}, \text{node})
\land \Gamma(\varphi_2, \text{prev}, \text{tmp2}, \text{tmp1}, \text{node}) \land \Gamma(\varphi_3, \text{prev}, \text{tmp2}, \text{tmp1}, \text{node})
\land \Gamma(\phi, \text{prev}, \text{tmp2}, \text{tmp1}, \text{node}).
\]

Thus, we have to prove that each conjunct of \( \text{post}(\text{tmp2} \rightarrow \text{prev} = \text{node}, \text{Inv}) \) conjugated with one of the formulas \( \neg \varphi_1, \neg \varphi_2, \neg \varphi_3, \) or \( \neg \phi \) is unsatisfiable. For example, to show that

\[
\Gamma(\varphi_1, \text{prev}, \text{tmp2}, \text{tmp1}, \text{node}) \land \neg \varphi_1
\]

is unsatisfiable is very easy because \( \Gamma(\varphi_1, \text{prev}, \text{tmp2}, \text{tmp1}, \text{node}) = \varphi_1 \).

### 5.3 Conclusions

We have introduced an expressive logic for reasoning about programs manipulating composite data structures storing data values from an unbounded domain. The logic allows a restricted form of quantifier alternation, reachability predicates, length and data constraints. Our main result is the decidability of its satisfiability problem in the presence of length constraints and quantifier alternation. It is established using a reduction to the satisfiability problem of the underlying data logic. We were able to prove the partial correctness of interesting programs with destructive updates, such as the program \textbf{Insert} given in Figure 4.3, the program \textbf{Dispatch} given in Figure 4.5 and sorting algorithms like insertion sort.
Chapter 6

Analysis of programs
manipulating singly-linked lists

We address the problem of automatic synthesis of assertions on sequential programs with
singly-linked lists containing data over infinite domains such as integers or reals. Our
approach is based on an accurate abstract inter-procedural analysis. We define composi-
tional techniques for computing procedure summaries describing various aspects such as
the shape of the heap, the size of the lists in the heap, and the data contained in the list
cells.

Program configurations or relations between program configurations are represented
by graphs where vertices represent list segments without sharing. Then, we define a
family of abstract domains for sets of relations between program configurations. Each
abstract domain combines a specific finite-range abstraction for the shape of the heap
with a (possibly infinite) abstract domain for sequences of data.

We instantiate our framework by introducing different abstractions on data sequences
allowing to reason about various aspects such as their sizes, the sums (see Section 6.6),
the multisets of their elements (see Section 6.5), or relations on their data at different
(linearly ordered or successive) positions. To express the latter relations, we define a new
abstract domain whose elements correspond to an expressive class of first order universally
quantified formulas (see Section 6.4).

Our analysis computes the effect of each procedure in a local manner, by considering
only the part of the heap reachable from its actual parameters. In order to avoid losses of
information, we introduce a mechanism allowing to strengthen the analysis in the domain
of first-order formulas by the analysis in the multisets domains (see Section 6.7). The same
mechanism is used for strengthening the sound (but incomplete) entailment operator of
the domain of first-order formulas.

The mechanism for generating universally quantified formulas and the mechanism of
strengthening universally quantified formulas are based on the same principle of unfold-
ing/folding operations on the manipulated data structures.

Outline: In Section 6.1 we introduce several simplifications of the representation of
relations between memory configurations, and, also, of the program assertions language.
These simplifications are induced by the fact that the considered programs manipulate
only singly-linked lists. Section 6.3 defines an abstraction of the graphs modeling the
heap memory, in terms of pairs of graphs and sets of words. Moreover, in this Section we
introduce the finite heap backbones that the analysis manipulates (see Section 6.3.1.2),
as well as the interface of an abstract domain representing words that can be combined
with the heap backbone (see Section 6.3.3). The next three Sections introduce three such
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

instances of abstract domains representing words of integers. Thus, Section 6.4 introduces
an abstract domain of universally quantified first-order formulas, Section 6.5 discusses an
abstract domain of multiset constraints, and Section 6.6 presents an abstract domain over
linear constraints describing the sum of their values. Section 6.7 presents some techniques
used in order to combine the analysis with different abstract domains, techniques based
on partial reduction operators. These techniques are implemented in a tool called Celia,
whose description is given in Section 6.8.

6.1 Programs over singly-linked lists

The programs handling singly-linked lists form a sub-class of the the language SimpleC
declared in Chapter 4. In this section we introduce the simplifications implied by this
restriction in the definition of the type system, of the representation of program configu-
rations and of the language of assertions.

Syntax We consider a class of strongly typed imperative programs as defined in Chap-
ter 4 that manipulate dynamic singly linked lists. The program syntax is the one given
in Section 4.3.1, where the type definition is restricted. We suppose that all manipulated
lists have the same type, i.e., reference to a record called list including one reference
field next and one data field dt of some type D. While the generalization to records with
several data fields is straightforward, the presence of a single reference field is important
for this work.

A program P_H manipulating only singly-linked lists induces a type system of the form:

Σ_{P_H} = (\{D\}, \text{list}, \{dt\}, \{next\}, \emptyset, DVar, PVar, \tau) \quad (6.1.1)

where DVar and PVar are the program data and pointer variables (the variables in DVar
are interpreted as values of type D and the variables in PVar are interpreted as addresses
of values of type list). The programs don’t have array variables, therefore the set of
array fields \mathcal{AF} is empty.

Semantics The semantics of program instructions is the one given in Section 4.3.3 but,
in this case, memory configurations can be represented by a particular class of heap graphs
where (1) each vertex, except for \$ and \$’, has exactly one successor, (2) all the edges
are labeled by next, (3) each vertex, except for \$ and \$’, is labeled by exactly one data
value in D, (4) all vertices are of type list, and (5) there are no array vertices and array
fields. In order to simplify the notation, the heap graphs in this class are represented by
SLL heap graphs where the edges are defined by a partial function and the data labeling
is defined by a function from vertices to D.

Definition 6.1.1 (SLL heap graph). An SLL-heap graph over a type system \Sigma as in 6.1.1
is an oriented graph G = (V, S, L, D), where:

- V is a finite set of vertices with \$ and \$” \in V,
- S : V \rightarrow V is a partial function that defines the successor of each vertex in V. It
  is undefined only for \$ and \$”.
- L : PVar \cup \{null\} \rightarrow V is a labeling function for vertices with sets of program
  variables, such that L(null) = \$.

148
6.1. PROGRAMS OVER SINGLY-LINKED LISTS

- \( D : V \rightarrow \mathbb{D} \) is a partial function that labels vertices with values from \( \mathbb{D} \) such that it is undefined only for \( \sharp \) and \( \sharp' \).

**Definition 6.1.2** (SLL heap). An SLL heap over a type system \( \Sigma \) is a pair \( H = (G, \delta) \), where \( G \) is an SLL heap graph over \( \Sigma \) and \( \delta : DVar \rightarrow \mathbb{D} \) is a valuation for the program data variables. \( \mathcal{H}_{SLL}(\Sigma) \) denotes the set of all SLL heaps over \( \Sigma \).

We define now several notions used in the following.

**Definition 6.1.3** (Sharing vertex). A vertex \( v \) of an SLL heap graph \( G = (V, S, L, D) \) is called a sharing vertex if \( v \) has at least two predecessors, i.e. there are two different vertices \( v_1 \) and \( v_2 \) in \( G \) such that \( S(v_1) = v \) and \( S(v_2) = v \).

**Definition 6.1.4** (Crucial vertex). A vertex \( v \) of an SLL heap graph \( G = (V, S, L, D) \) is called crucial vertex if \( v \) is either a sharing vertex or it is labeled by a program variable (i.e. \( L(p) = v \), for some \( p \in PVar \)).

**Definition 6.1.5** (Simple vertex). A vertex \( v \) of an SLL heap graph \( G = (V, S, L, D) \) is called a simple vertex if \( v \) is not a crucial vertex.

We denote by \( \text{Simple}(G) \) the set of all simple vertices of \( G \). For any crucial vertex \( v \), we define \( \text{Simple}(G, v) = [n_1, \ldots, n_t] \), where \( n_s \in \text{Simple}(G) \) for any \( 1 \leq s < t \), to be the maximal vector of adjacent simple vertices such that (1) \( S(v) = n_1 \), (2) \( S(n_s) = n_{s+1} \) for any \( 1 \leq s \leq t-1 \), and (3) \( S(n_t) \notin \text{Simple}(G) \).

![Figure 6.1: An SLL heap graph denoting a memory configuration (here, \( \mathbb{D} = \mathbb{Z} \)](image)

**Example 6.1.1.** Figure 6.1 shows a memory configuration possible at line 8 of the procedure \( \text{main} \) from the program \( P_{11} \) given in Figure 4.16. It is a garbage free SLL heap graph over the type system \( \Sigma_{P_{11}} \) where the base type \( \mathbb{D} \) are the integers and \( PVar = \{x, xi, z, zi\} \). This SLL heap graph has only one sharing vertex, the one labeled by the integer 3. This sharing vertex together with the two vertices labeled by pointer variables form the set of crucial vertices. If \( v \) denotes the vertex labeled by \( z \), then \( \text{Simple}(G, v) = [v_1, v_2] \), where \( v_1 \) is the vertex labeled by 5 and \( v_2 \) is the vertex labeled by 7.

**6.1.1 Remark.** For any garbage free SLL heap graph, the number of sharing vertices is bounded by the number of program pointer variables.

Notice that garbage free SLL heap graphs are still potentially unbounded because they can have an unbounded number of simple vertices. On the other hand, SLL heap graphs that are not garbage free might have an unbounded number of sharing vertices.

Following the inter-procedural semantics given in Section 4.3.3, a pair of two SLL heaps \( H_1 \) and \( H_2 \) over the same type system \( \Sigma_{P_{11}} \), induced by some program \( P_{11} \), is denoted by \( [H_1, H_2] \) and it can represented by a unique SLL heap denoted by \( H_1 \oplus H_2 \). The SLL
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

heap $H_1 \oplus H_2$ is defined over a type system $\Sigma_{P_H}$ obtained from $\Sigma_{P_H}$ by adding one copy (superscripted with 0) of each program variable. Therefore,

$$\Sigma_{P_H} = (\{D\}, \{\text{list}\}, \{\text{dt}\}, \{\text{next}\}, \emptyset, DVar \cup DVar^0, PVar \cup PVar^0, \tau).$$

We recall that $H_1 \oplus H_2$ is obtained by substituting the variables in $H_1$ with the corresponding copies from $DVar \cup PVar^0$ and by taking the union of the two heaps (see Section 4.3.3.2 for more details).

![Diagram](image)

(a) The SLL heap $H$

(b) The contracted 0-SLL heap $H^0_w$

(c) The contracted 1-SLL heap $H^1_w$

(d) The contracted 0-SLL heap $H^0_w$

Figure 6.2: A 0-SLL-heap graph, a 1-SLL-heap graph and the corresponding SLL heap denoting the same pair of memory configurations where $D = \mathbb{Z}$

**Example 6.1.2.** The SLL heap graph from Figure 6.2(a) describes the relation between some input configuration of the procedure list-shared, given in Figure 4.16, (the part of the graph containing vertices reachable from the ones labeled by $\text{ain}^0$ and $\text{bin}^0$) and the corresponding output configuration (the part of the graph containing vertices reachable from the ones labeled by a and b).

We define an isomorphism relation between SLL heaps based on the isomorphism of their underling graphs.

**Definition 6.1.6** (SLL heap isomorphism). Two SLL heaps over $\Sigma$, $H = (G = (V, S, L, D), \delta)$ and $H' = (G' = (V', S', L', D'), \delta')$ are isomorphic, denoted $H \sim H'$, iff

1. the underlying graphs $G$ and $G'$ are isomorphic, i.e. there is a bijection $h : V \to V'$ such that any two vertices $v_1$ and $v_2$ from $V$ are adjacent in $G$ iff $h(v_1)$ and $h(v_2)$ are adjacent in $G'$, for any $p \in PVar$, $L(p) = v$ iff $L'(p) = h(v)$ and, for any $v \in V$, $D(v) = D'(h(v))$ and,
6.2. LANGUAGE OF ASSERTIONS

2. for any \( d \in DVar \), \( \delta(d) = \delta'(d) \).

Let \( P_H \) be a program over a type system \( \Sigma \). The concrete lattice of memory configurations for the program \( P_H \), denoted by \( C(\Sigma) \), is defined by:

\[
C(\Sigma) = (P(H_{\text{SLL}}(\Sigma)/\sim), \subseteq /\sim, \cup, \cap, \emptyset, H_{\text{SLL}}(\Sigma)/\sim),
\]

where \( H_{\text{SLL}}(\Sigma)/\sim \) is the quotient set of \( H_{\text{SLL}}(\Sigma) \) by the isomorphism relation \( \sim \) and \( \subseteq /\sim \) is the quotient of the order \( \subseteq \) by \( \sim \). That is, \( H \subseteq H' \) iff for every \( H \in H \) there is \( H' \in H' \) such that \( H \sim H' \).

If we extend \( \Sigma \) to \( \Sigma' \) then, the concrete lattice of pairs of memory configurations for \( P_H \), denoted by \( C(\Sigma') \), is defined by:

\[
C(\Sigma') = (P(H_{\text{SLL}}(\Sigma')/\sim), \subseteq /\sim, \cup, \cap, \emptyset, H_{\text{SLL}}(\Sigma')/\sim).
\]

We extend \( C(\Sigma) \) and \( C(\Sigma') \) to the lattice of tuples of concrete memory configurations and, respectively, tuples of pairs of memory configurations by:

\[
\begin{align*}
C_{\text{pc}}(\Sigma) &= (P((\mathcal{H}_{\text{SLL}}(\Sigma)/\sim)^p), \subseteq /\sim, \cup, \cap, \emptyset, (\mathcal{H}_{\text{SLL}}(\Sigma)/\sim)^p) \\
C_{\text{pc}}(\Sigma') &= (P((\mathcal{H}_{\text{SLL}}(\Sigma')/\sim)^p), \subseteq /\sim, \cup, \cap, \emptyset, (\mathcal{H}_{\text{SLL}}(\Sigma')/\sim)^p),
\end{align*}
\]

where \( p \) is the number of nodes in the (inter-procedural) control flow graph associated with \( P_H \). Each element of the tuple denotes a set of memory configurations (a set of pairs of memory configurations) at a control point of the program \( P_H \).

The elements of \( C_{\text{pc}}(\Sigma) \) (\( C_{\text{pc}}(\Sigma') \)) are sets of states from the transition system corresponding to the program \( P_H \) defined in Section 4.2.3 (respectively, Section 4.3.3). The transformers \( F[a] \), where \( a \) is an edge label of the intra-procedural control flow graph, from Section 4.2.3.3 are defined over elements in \( C(\Sigma) \). Also, the transformers \( U[a] \) and \( \overline{U}[a] \), where \( a \) is an edge label of the inter-procedural control flow graph, from Section 4.3.3.3 are defined over elements in \( C(\Sigma') \). The intra-procedural reachability collecting semantics is an element of \( C_{\text{pc}}(\Sigma) \) defined by the system of equations from Section 4.2.3.4 and the inter-procedural reachability collecting semantics is an element of \( C_{\text{pc}}(\Sigma') \) defined by the system of equations from Section 4.3.3.4.1.

6.2. Language of assertions

The specification and the assertions of the programs introduced in the previous section are first-order formulas in a logic called SL3. An SL3 formula is a quantifier free or existentially quantified disjunction of formulas which are conjunctions between (1) a formula that describes the shape of the memory configuration (the graph), (2) a formula that labels vertices of the graph with program pointer variables, and (3) a formula that constrains the values from \( D \) associated with vertices in the graph and the program variables of basic type.

The sub-formula that describes the shape of the memory configuration is a \( gCSL \) formula that uses some syntactic sugar introduced hereafter. can be simplified because we can assume w.l.o.g. that there is only one (recursive) pointer field, that is, \( PF = PF_r = \{ f \} \).

First, because the program uses only one (recursive) pointer field (that is, \( PF = PF_r = \{ f \} \)) we omit pointer fields in the reachability predicates: we write \( v \leadsto v' \), \( v \leftarrow v' \), and
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

\[ \nu \xrightarrow{L,R} \nu \text{ instead of } \nu \xrightarrow{L} \nu' \text{, } \nu \xrightarrow{L} \nu' \text{ and } \nu \xrightarrow{L,R} \nu' \text{, respectively. Also, sometimes, we write } \nu \xrightarrow{z} \nu' \text{ instead of } \nu \xrightarrow{z} \nu \land \nu \xrightarrow{z} \nu'. \]

Figure 6.3: A representation for program heaps with singly-linked lists

Since the record types contain only one pointer field, the \textit{gCSL} formulas over the corresponding type system describe graphs where each vertex has at most one successor (due to the determinism of the pointer fields). Let us consider the graph in Figure 6.3. There are several formulas that hold over this graph. For example,

\[ \exists v_1, v_2, v_3. x(v_1) \land z(v_2) \land v_1 \xrightarrow{\#} \# \land v_2 \xrightarrow{\#} \# \land v_1 \xrightarrow{} v_3 \land v_2 \xrightarrow{} v_3 \land \text{null}(\#) \]

or

\[ \exists v_1, v_2, v_3. x(v_1) \land z(v_2) \land v_1 \xrightarrow{} v_3 \land v_2 \xrightarrow{} v_3 \land v_3 \xrightarrow{} \# \land \text{null}(\#). \]

In fact, the two formulas have the same set of models. Figure 6.3 gives an example of such a model. Notice that the second formula has fewer reachability predicates and it distinguishes the parts of the lists which are shared from the ones without sharing. Any heap graph over a single pointer field can be described as a union of list segments that may share only their ends.

To express list segments in a concise and intuitive way, we introduce:

- a predicate \( l_s(n,m) \), where \( n \) and \( m \) are location variables, that denotes a list segment (a path in the graph), and
- a binary operator, denoted \( \star \), defined over predicates of the form \( l_s(n,m) \) that expresses the disjointness of its arguments (the considered paths don’t share more than their ends). Intuitively, \( \star \) states that the graphs described by its arguments share at most the vertices with no predecessors or the vertices with no successors.

In the following, we consider the predicate \( l_s(x,x') \) to be equivalent with \( x \xrightarrow{} x' \). The formulas describing the shape of the graph in \textit{SL3} formulas are defined by the following grammar:

\[
\varphi^{\text{peq}} := n = m \mid n \neq m \mid \varphi^{\text{peq}} \land \varphi^{\text{peq}}
\]

\[
\varphi^{l_s} := l_s(n,m) \mid \varphi^{l_s} \land \varphi^{l_s}
\]

\[
\varphi^{\text{SLL}} := \varphi^{l_s} \land \varphi^{\text{peq}} \tag{6.2.1}
\]

The operator \( \star \) is associative and commutative. Therefore, a general \( \varphi^{l_s} \) formula (with more than two \( l_s \) predicates) is equivalent with the \textit{gCSL} formula that is a conjunction of formulas \( \varphi^{\star}(l_s(n,m),l_s(n',m')) \) like in 6.2.2 one for every pair of predicates \( (l_s(n,m),l_s(n',m')) \) that appear in \( \varphi^{l_s} \). \( p \), \( p' \), \( q \), and \( q' \) are ghost pointer variables pointing to the vertices \( m \), respectively \( m' \). The ghost pointer variables are used only in the assertions (and not in the source code) in order to avoid inconsistencies induced by cycles. The term \( \text{len}(n) \) denotes the length of the list segment denoted by \( l_s(n,m) \). It
6.2. LANGUAGE OF ASSERTIONS

is like an index variable, and it does not depend on $m$ because every node has an unique successor.

$\varphi^*(\mathsf{ls}(n,m),\mathsf{ls}(n',m')) = n^{\mathsf{ls}(n)} \rightarrow m \land n'^{\mathsf{len}(n')} \rightarrow m' \land$

$p(m) \land p'(m') \land q(n) \land q'(m) \land$

$n^{\mathsf{E} \rightarrow} m \land n'^{\mathsf{E} \rightarrow} m \land n'^{\mathsf{E} \rightarrow} m' \land n'^{\mathsf{E} \rightarrow} m' \land$

$\forall y,y'. (n \rightarrow y^{\{p\}} \rightarrow m \land n' \rightarrow y^{\{p'\}} \rightarrow m') \implies y \neq y'$.

Notice that, we can combine $\varphi^\mathsf{SLL}$ formulas with usual $\mathsf{gCSL}$ formulas. The operator $*$ is defined only over $\mathsf{ls}$ predicates and since its arguments are only $*$ applications, it has no influence on the rest of the formula. For example, Figure 6.4 shows several models for the formula: $\varphi = \exists n, m, n', m', x, z. \mathsf{ls}(n,m) \ast \mathsf{ls}(n',m') \land x \rightarrow z$. In this figure, the vertices are labeled also by location variables from $\varphi$, denoting a valuation for the existential variables that satisfies $\varphi$. Notice that $x$ can be interpreted into a node on the path between $n$ and $m$, as long as this node does not belong to the path between $n'$ and $m'$.

**Syntax of SL3**: An SL3 formula is an existentially-quantified disjunction of formulas which are conjunctions between:

1. a formula $\varphi^\mathsf{SLL}$ as in [6.2.1]; we recall that such formulas are $*$ applications over predicates of the form $\mathsf{ls}(n,m)$ conjuncted with equalities and inequalities like $n = m$ or $n \neq m$, where $n$ and $m$ are location variables. All the location variables representing vertices which are not labeled by program pointer variables are existentially quantified. The other location variables (representing vertices labeled by pointer variables) are considered as free variables. The name of the location variable labeled by the constant $\mathsf{null}$ is fixed to $\sharp$. This location variable represents a vertex with no successors;
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

2. a formula $\varphi^p$ which is a conjunction of predicates of the form $p(n)$, where $p$ is a pointer variable and $n$ is a location variable;

3. a formula $\varphi^W$ over the same location variables as $\varphi^{SLL}$, that constrains (only) the data associated with the list segments in $\varphi^{SLL}$ and the values of the program data variables. $\varphi^W$ is a formula that has as model a set of sequences. That is, the models of $\varphi^W$ are graphs that are a union of distinct acyclic paths whose vertices are labeled by values of type $D$.

Therefore, an $SL3$ formula, $\varphi^{SL3}$, has the form:

$$\varphi^{SL3} ::= \exists x. \bigvee \left( \varphi^{SLL} \land \varphi^p \land \varphi^W \right),$$

(6.2.3)

where $x$ is a vector of location variables. We require that $x$ does not contain location variables $n$ for which there exists a predicate $p(n)$ with $p \in PVar$ in some formula $\varphi^p$.

Notice that, for any predicate $ls(n,m)$ of $\varphi^{SLL}$, any node on the (unique) path between $n$ and $m$, except for $n$ and $m$, cannot be the interpretation of any location variable from $\varphi^{SLL}$ in any model of $\varphi^{SLL}$. This particularity of $\varphi^{SLL}$ formulas implies that, from a modeling point of view, list segments are very similar to arrays. Therefore, for the sake of readability, we make an abuse of notation and we consider that $\varphi^W$ is a formula where every location variable in $\varphi^{SLL}$ is used as an array variable in $\varphi^W$ and $\varphi^W$ is a formula on arrays, that does not quantify over location variables. The models of $SL3$ formulas are heaps as defined in Section 4.2.3. The semantics of $SL3$ formulas interprets all location variables into vertices of the heap (not into array nodes). $SL3$ does not quantify over array variables, nor use them as free variables. Only the notation from array terms is borrowed in $\varphi^W$ in order to simplify the syntax of the formulas.

**Example 6.2.1.** Let $\varphi_{SLL}$ be a $gCSL$ formula. Making an abuse of notation, we consider $x$ to be an array variable in $\varphi^W$ and, the equivalent $SL3$ formula has the following syntax: $\varphi_{SLL} \land \forall j. 1 \leq j < \text{len}(x) \implies x[j] > 3$, where $\text{len}(x)$ denotes the length of the list between $x$ and $x'$ (the reachability predicate corresponding to $ls(x, x')$ is $x \overset{\text{len}(x)}{\rightarrow} x'$).

In the next sections (Sections 6.4, 6.5, and 6.6) we introduce three logic fragments that model sets of sequences, denoted generically by $A_W$.

**Definition 6.2.1 (SL3).** The specification and the program assertions are formulas in $SL3$ parametrized by $A_W$, that is

$$\varphi^{SL3} ::= \exists x. \bigvee \left( \varphi^{SLL} \land \varphi^p \land \varphi^W \right),$$

where $\varphi^W \in A_W$.

In the following, we list some $SL3$ formulas that describe the specification and the assertions of the programs given as example in Section 4.2.2 and Section 4.3.2.
6.3. A FAMILY OF ABSTRACT DOMAINS

- Program Dispatch given in Figure 4.15
  “\(\text{gr}\) points to a acyclic singly-linked list and the value of the data field of each list node reachable from \(\text{gr}\) is greater than \(v\)”
  \[\begin{align*}
  &\text{ls}(x, \sharp) \land \text{gr}(x) \\
  &\text{dt}(x) > v \land \\
  &\forall y. 1 \leq y < \text{len}(x) \implies x[y] > v
  \end{align*}\]

- Program Fibonacci given in Figure 4.17
  – the specification of the Fibonacci procedure is:
    “\(\text{head}\) points to an acyclic singly-linked list or \(\text{head}\) points to null”
    \(\begin{align*}
    &\left(\text{ls}(\text{head}, \sharp) \land \text{head}(\text{head})\right) \lor \\
    &\left(\text{head} = \sharp \land \text{head}(\text{head})\right)
    \end{align*}\)

  “\(\text{head}\) points to an acyclic singly-linked list whose elements form a Fibonacci sequence”
  \(\begin{align*}
  &\left(\text{ls}(\text{head}, \sharp) \land \text{head}(\text{head})\right) \land \\
  &\forall y_1, y_2, y_3. (0 \leq y_1 \land y_2 = y_1 + 1 \land \\
  &y_3 = y_2 + 1 \land y_3 < \text{len}(\text{head}) \implies \text{head}[y_3] = \text{head}[y_2] + \text{head}[y_1])
  \end{align*}\)

  – the specification of the alloc procedure is:
    “\(\text{head}\) points to an acyclic singly-linked list of length size” or \(\text{head}\) points to null”
    \(\begin{align*}
    &\left(\text{ls}(\text{head}, \sharp) \land \text{head}(\text{head})\right) \land \\
    &\text{len}(\text{head}) = \text{size} \lor \\
    &\left(\text{head} = \sharp \land \text{head}(\text{head})\right)
    \end{align*}\)

- Program addV given in Figure 4.19
  The specification of the procedure addV is:
  “\(\text{head}\) points to an acyclic singly-linked list or \(\text{head}\) points to null”
  \(\begin{align*}
  &\left(\text{ls}(\text{head}, \sharp) \land \text{head}(\text{head})\right) \lor \\
  &\left(\text{head} = \sharp \land \text{head}(\text{head})\right)
  \end{align*}\)

  “\(\text{head}\) points to an acyclic singly-linked list the values of the data fields are incremented by \(v\) w.r.t. the input ones”
  \(\begin{align*}
  &\left(\text{ls}(\text{head}', \sharp) \land \text{head}'(\text{head}')\right) \land \\
  &\left(\text{ls}(\text{head}, \sharp) \land \text{head}(\text{head})\right) \land \\
  &\text{len}(\text{head}') = \text{len}(\text{head}) \land \\
  &\text{dt}(\text{head}) = \text{dt}(\text{head}') + v \land \\
  &\forall y_1, y_2. (0 \leq y_1 < \text{len}(\text{head}) \land y_1 = y_2 \land \\
  &1 \leq y_2 < \text{len}(\text{head}') \implies \text{head}[y_1] = \text{head}'[y_2] + v \lor \\
  &\left(\text{head}(\sharp) \land \text{head}'(\sharp)\right)
  \end{align*}\)

6.3 A family of abstract domains

6.3.1 A precise heap abstraction

We introduce a precise representation for sets of SLL heaps, called data word k-SLL heap sets. We define the lattice of data word k-SLL heap sets which is connected through a concretization function to \(C(\Sigma)\). To this, first we define the lattice of contracted k-SLL heaps. This lattice is connected through a concretization function to the lattice \(C(\Sigma)\), introduced in Section 4.3.3 Finally, we introduce an abstract semantics on data word k-SLL heap sets which corresponds to the concrete semantics on SLL heaps.
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

6.3.1.1 Lattice of contracted \( k \)-SLL heaps

Given an SLL heap \( H = (G, \delta) \) and \( k \) a fixed natural number, we define an equivalent representation for \( H \), called \emph{contracted \( k \)-SLL heap}, as a graph where vertices are labeled by words over \( \mathbb{D} \) together with the valuation \( \delta \). Roughly, this graph contains all the crucial vertices and at most \( k \) simple vertices from \( H \). Notice that the number of vertices of this graph is linear in \( k \) and the number of program pointer variables.

\begin{definition}[Contracted \( k \)-SLL heap] A contracted \( k \)-SLL heap over \( \Sigma \) is a tuple \( H_w = (N, S, L, \nu, \delta) \) where
\begin{itemize}
  \item \( G = (N, S, L, \nu, \delta) \), where \( D(v) \in \mathbb{D} \), for all \( v \in N \), is an SLL heap graph over \( \Sigma \) with at most \( k \) simple vertices,
  \item \( \nu : N \to \mathbb{D}^* \) is a partial function that associates with each node in the graph a word from \( \mathbb{D}^* \). It is undefined only for \( \nu \) and \( \nu' \).
  \item \( \delta : DVar \to \mathbb{D} \) is a valuation for the program data variables from \( \Sigma \).
\end{itemize}

Let \( \mathcal{H}_{k-\text{SLL}}(\Sigma) \) be the set of all contracted \( k \)-SLL heaps over \( \Sigma \).
\end{definition}

In the following, the vertices of a contracted \( k \)-SLL heap are called \emph{nodes}. Also, for a word \( w \in \mathbb{D}^* \), \( \text{len}(w) \) denotes the length of \( w \). We consider that the positions of a word are numbered starting from 0 and we denote \( w = [w_0, w_1, \ldots, w_{\text{len}(w)-1}] \) where \( w_i \in \mathbb{D} \), for all \( 0 \leq i < \text{len}(w) \).

As in the case of SLL heaps, we define an isomorphism relation between contracted \( k \)-SLL heaps based on the notion of graph isomorphism.

\begin{definition}[Contracted \( k \)-SLL heap isomorphism] Two contracted \( k \)-SLL heaps over \( \Sigma \), \( H_w = (N, S, L, \nu, \delta) \) and \( H'_w = (N', S', L', \nu', \delta') \) are isomorphic, denoted \( H_w \sim H'_w \), iff
\begin{enumerate}
  \item the underlying graphs \( G = (N, S, L) \) and \( G' = (N', S', L') \) are isomorphic, i.e. there is a bijection \( h : N \to N' \) such that any two nodes \( n_1 \) and \( n_2 \) from \( N \) are adjacent in \( G \) iff \( h(n_1) \) and \( h(n_2) \) are adjacent in \( G' \) and for any \( p \in PVar \), \( L(p) = n \) iff \( L'(p) = h(n) \),
  \item for any \( n \in N \), \( \nu(n) = \nu'(h(n)) \) and,
  \item for any \( d \in DVar \), \( \delta(d) = \delta'(d) \).
\end{enumerate}
\end{definition}

We consider a distinguished contracted \( k \)-SLL heap in \( \mathcal{H}_{k-\text{SLL}}(\Sigma) \), \( H_{err} \), that is not isomorphic with any other contracted \( k \)-SLL heap in \( \mathcal{H}_{k-\text{SLL}}(\Sigma) \).

Given a type system \( \Sigma \) and a natural number \( k \), we define the lattice \( \mathcal{C}_{k-\text{SLL}}(\Sigma) \) to be the lattice of sets of contracted SLL heaps in \( \mathcal{H}_{k-\text{SLL}}(\Sigma) \):

\[ \mathcal{C}_{k-\text{SLL}}(\Sigma) = (P(\mathcal{H}_{k-\text{SLL}}(\Sigma)/\sim), \subseteq / \sim, \cup, \cap, \emptyset, \mathcal{H}_{k-\text{SLL}}(\Sigma)/\sim), \]

where, for any two \( HS_w, HS'_w \in P(\mathcal{H}_{k-\text{SLL}}(\Sigma)/\sim) \)

\[ HS_w \subseteq / \sim HS'_w \ \text{iff} \ \text{for every} \ H_w \in HS_w \ \text{there is} \ H'_w \in HS'_w \ \text{such that} \ H_w \sim H'_w. \]  (6.3.1)
6.3. A FAMILY OF ABSTRACT DOMAINS

In the following we work only with equivalence classes and we make an abuse of notation by writing \( H_{k-SLL}(\Sigma) \) inserted of \( H_{k-SLL}(\Sigma)/\sim \).

A contracted k-SLL heap \( H_w \) represents an SLL heap \( H \) obtained by transforming every word \( \nu(n) = w \), for some node \( n \), into a path that starts in the node \( n \) and ends in the unique successor of \( n \) in \( H_w \). This path is created by adding \( \text{len}(w) - 1 \) new vertices that are labeled with values from \( D \) such that if we concatenate the labels on this path we obtain \( w \). In particular, \( n \) is labeled by the first letter of \( w \) and the last vertex added to the graph is labeled by the last letter of \( w \). This is formally defined by a concretization function \( \gamma^{k-SLL} : H_{k-SLL}(\Sigma) \rightarrow H_{SLL}(\Sigma) \).

**Concretization function for contracted k-SLL heaps:** Let \( \gamma^{k-SLL} : H_{k-SLL}(\Sigma) \rightarrow H_{SLL}(\Sigma) \) be defined by: for any \( H_w = (N,S,L,\nu,\delta) \),

\[
\gamma^{k-SLL}(H_w) = (G = (V',S,L,D),\delta),
\]

if \( H_w \) is different from \( H_{err} \) then

- \( V = N \cup V' \) such that \( |V'| = \sum_{n \in N} \text{len}(\nu(n)) - 1 \) and \( V' \cap N = \emptyset \);
- for every \( n,n' \in N \) such that \( S(n) = n' \) and \( \nu(n) = [w_0,\ldots,w_{l-1}] \), (1) we add \( l - 1 \) new vertices \( \{v_1,\ldots,v_{l-1}\} \) to \( V' \), (2) we define \( S(n) = v_1 \), \( S(v_i) = v_{i+1} \), for any \( 1 \leq i < l-1 \), and \( S(v_{l-1}) = n' \), and (3) we define \( D(n) = w_0 \) and \( D(v_i) = w_i \), for any \( 1 \leq i \leq l-1 \).

Otherwise \( \gamma^{k-SLL}(H_{err}) = H_{err} \).

We extend \( \gamma^{k-SLL} \) to sets of contracted k-SLL heaps as usual.

**Example 6.3.1.** In Figure 6.2(b) \( H^0_w \) is a contracted 0-SLL heap (without any simple nodes) whose concretization is the heap in Figure 6.2(a). The words in the image of \( \nu \) are represented as labels of the nodes. For example, \( \nu(n) = [1,8,9] \) where \( n \) is the node labeled by a. Similarly, \( H^1_w \) from Figure 6.2(c) is a contracted 1-SLL heap having the same concretization as \( H^0_w \). Both contracted k-SLL heaps represent the relation between some input configuration of the procedure list_shared in Figure 4.16 and the corresponding output configuration. Since \( k \) is an upper bound on the number of simple nodes, \( H^0_w \) is also a contracted 1-SLL heap. The contracted 0-SLL heap in Figure 6.2(d) represent the relation between another input configuration of the same procedure list_shared and the corresponding output configuration. This procedure does not manipulate any data variables and, therefore, we don’t represent the function \( \delta \).

The lattice \( C_{k-SLL}(\Sigma) \) is connected through the concretization function \( \gamma^{k-SLL} \) to the lattice \( C(\Sigma) \):

\[
\gamma^{k-SLL} : \mathcal{P}(H_{k-SLL}(\Sigma)) \rightarrow \mathcal{P}(H_{SLL}(\Sigma)).
\]

It is straightforward that the concretization function \( \gamma^{k-SLL} \) is monotone. Moreover, we can define an abstraction function

\[
\alpha^{k-SLL} : \mathcal{P}(H_{SLL}(\Sigma)) \rightarrow \mathcal{P}(H_{k-SLL}(\Sigma))
\]

which associates to every concrete heap \( H \) a set of contracted k-SLL heaps, one for each different set of at most \( k \) simple vertices in \( H \). Given a set \( N' \) of vertices in \( H = (G = (V,S,L,D),\delta) \) that contains all the crucial vertices and at most \( k \) simple vertices, we can
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

de define a contracted \( k \)-SLL heap \( H_{N'} = (N', S', L, \nu, \delta) \) such that (1) \( S'(n) = n' \) iff \( n' \) is reachable from \( n \) in \( H \) and (2) \( \nu(n) = [w_0, \ldots, w_{l-1}] \) iff there exists a path \([n_0, n_1, \ldots, n_l]\) in \( H \) between \( n = n_0 \) and its successor in \( H_{N'} \), such that \( D(n_i) = w_i \), for any \( 0 \leq i \leq l - 1 \). Notice that the pair \((\alpha^{k-\text{SLL}}, \gamma^{k-\text{SLL}})\) forms a Galois connection between \( C_{k-\text{SLL}}(\Sigma) \) and \( C(\Sigma) \).

6.3.1.2 Lattice of data word \( k \)-SLL heap sets

The set of SLL heaps over some type system \( \Sigma \) is in general unbounded because it contains graphs of unbounded size whose vertices are labeled by values from an unbounded data domain \( \mathbb{D} \). For any \( k \), the set of contracted \( k \)-SLL heaps over \( \Sigma \) is also unbounded. In the case of contracted \( k \)-SLL heaps, the unboundedness comes from (1) the image of the function \( \nu \) which is a set of words of unbounded length over a possible unbounded data domain and (2) the image of the function \( \delta \) which is a set of values from \( \mathbb{D} \). Contrary to SLL heaps, the number of vertices in the graph is bounded by \( k \) and the number of program pointer variables.

We define an equivalent representation for sets of contracted \( k \)-SLL heaps that contains a bounded number of objects, called data word contracted \( k \)-SLL heaps. A data word \( k \)-SLL heap contains (1) a family of valuations for the program data variables and (2) a graph together with a family of vertex labelings with words over \( \mathbb{D} \). Any set of contracted \( k \)-SLL heaps \( \mathcal{H}_u \) with isomorphic underlying graphs (the underlying graph contains all but the labeling with words over \( \mathbb{D} \)) can be represented by one data word \( k \)-SLL heap which contains (1) the valuation for the program data variables from each contracted \( k \)-SLL heap and (2) the underlying graph of the contracted \( k \)-SLL heaps in \( \mathcal{H}_u \) (which is unique by the isomorphism assumption) plus the vertex labeling from each contracted \( k \)-SLL heap in \( \mathcal{H}_u \). Since the number of underlying graphs contained in all the contracted \( k \)-SLL heaps is bounded, we obtain that any set of contracted \( k \)-SLL heaps can be represented by a bounded number of data word \( k \)-SLL heaps.

Definition 6.3.3 (Data word \( k \)-SLL heap). A data word \( k \)-SLL heap over \( \Sigma \) (heap, for short) is a tuple \( H = (G, (\nu_i, \delta_i)_{i \in I}) \), where:

- \( G = (N, S, L) \) is a labeled graph, where \( N \) is a set of nodes containing two distinguished nodes \( \sharp \) and \( \sharp' \), \( S : N \rightarrow N \) is the successor function (it is undefined only for \( \sharp \) and \( \sharp' \)), and \( L : PVar \cup \{\text{null}\} \rightarrow N \) is a labeling function for nodes with sets of program variables, such that \( L(\text{null}) = \sharp \),

- \( (\nu_i, \delta_i)_{i \in I} \subseteq \mathcal{P}([N \rightarrow \mathbb{D}^+] \times [DVar \rightarrow \mathbb{D}]) \) is a family of pairs of functions such that \( \nu_i \), for all \( i \in I \), is undefined only for \( \sharp \) and \( \sharp' \). The functions \( \nu_i \) with \( i \in I \) label nodes with non-empty words over \( \mathbb{D} \) and the functions \( \delta_i \) with \( i \in I \) represent valuations for the program data variables.

The set of all data word \( k \)-SLL heaps over \( \Sigma \) is denoted by \( \mathcal{H}(\Sigma, k) \).

We define a notion of isomorphism on data word \( k \)-SLL heaps, based on graph isomorphism, which ignores the labelings with words from \( \mathbb{D} \).

Definition 6.3.4 (Data word \( k \)-SLL heap isomorphism). Two data word \( k \)-SLL heaps over \( \Sigma \) \( H = (G = (N, S, L), (\nu_i, \delta_i)_{i \in I}) \) and \( H' = (G' = (N', S', L'), (\nu'_i, \delta'_i)_{i \in I'}) \) are isomorphic, denoted \( H \sim H' \), iff
6.3. A FAMILY OF ABSTRACT DOMAINS

- the underlying graphs \( G = (N, S, L) \) and \( G' = (N', S', L') \) are isomorphic, i.e.

there is a bijection \( h : N \to N' \) such that any two nodes \( n_1 \) and \( n_2 \) from \( N \) are

adjacent in \( G \) iff \( h(n_1) \) and \( h(n_2) \) are adjacent in \( G' \) and for any \( p \in PVar, \)

\( L(p) = n \) iff \( L'(p) = h(n) \).

We use an abuse of notation and we consider \( H_{err} \) to be an error data word \( k\text{-SLL} \)
heap, that is not isomorphic with any other data word \( k\text{-SLL} \) heap in \( \mathcal{H}(\Sigma, k) \).

Notice that \( \sim \) is an equivalence relation and let \( \mathcal{H}(\Sigma, k)/\sim \) be the quotient of \( \mathcal{H}(\Sigma, k) \)
by \( \sim \).

**Definition 6.3.5** (Data word \( k\text{-SLL} \) heap set). A set of data word \( k\text{-SLL} \) heaps is called
a data word \( k\text{-SLL} \) heap set (heap set, for short) if it is a subset of \( \mathcal{H}(\Sigma, k)/\sim \). The set of
all data word \( k\text{-SLL} \) heap sets over a type system \( \Sigma \) is denoted by \( \mathcal{HS}(\Sigma, k) \).

Notice that a data word \( k\text{-SLL} \) heap set is a set of data word \( k\text{-SLL} \) heaps that does
not contain two isomorphic data word \( k\text{-SLL} \) heaps.

We define the lattice of data word \( k\text{-SLL} \) heap sets:

\[
\mathcal{C}_{\mathcal{HS}(\Sigma, k)} = (\mathcal{HS}(\Sigma, k), \subseteq, \sqcup, \sqcap, \emptyset, \mathcal{HS}(\Sigma, k)), \text{ where}
\]

- for any two heap sets \( HS', HS'' \in \mathcal{HS}(\Sigma, k) \), the order relation \( \subseteq \) is defined as follows:

\[
HS' \subseteq HS'' \text{ iff for every } H' = (G', (\nu_i', \delta_i')_{i \in I'}) \in HS',
\]  

there exists \( H'' = (G'', (\nu_i'', \delta_i'')_{i \in I''}) \in HS'' \)

such that \( H' \sim H'' \text{ w.r.t. an isomorphism } h \text{ and } ((\nu_i', \delta_i'[h])_{i \in I'}) \subseteq (\nu_i'', \delta_i''[h])_{i \in I''} \).

For every \( i \in I' \), \( (\nu_i', \delta_i'[h]) = (\nu_i'[h], \delta_i'), \) where \( \nu_i'[h] : h(N') \to D^* \) such that
\( \nu_i'[h](h(n)) = \nu_i'(n) \) for every \( n \in N' \).

The inclusion between families of the form \( (\nu_i, \delta_i)_{i \in I} \) is defined as usual modulo a
renaming of the index domain of \( (\nu_i, \delta_i)_{i \in I} \), i.e. \( (\nu_i', \delta_i'[h])_{i \in I'} \subseteq (\nu_i'', \delta_i''[h])_{i \in I''} \) iff for
any \( i \in I' \) there exists \( j \in I'' \) such that \( \nu_i'[h] = \nu_j'' \) and \( \delta_i' = \delta_j'' \). In the following,
we sometimes make an abuse of notation and omit the isomorphism \( h \).

- for any \( HS', HS'' \in \mathcal{HS}(\Sigma, k) \), \( HS' \sqcup HS'' \) is defined by (1) taking the union of \( HS' \) and \( HS'' \) and (2) any two isomorphic heaps \( H' \) and \( H'' \) are replaced by one heap \( H \) having the same graph s.t. the family of vertex labellings (resp. valuations for
program data variables) in \( H \) is union of the vertex labellings (resp. valuations for
program data variables) in \( H' \) and \( H'' \). Formally, \( HS' \sqcup HS'' \) is defined by:

\[
\{ H = (G, (\nu_i, \delta_i)_{i \in I}) | \text{there exists } H' = (G', (\nu_i, \delta_i)_{i \in I}) \in HS' \text{ s.t. } H' \sim H \text{ and } H'' \not \sim H, \forall H'' \in HS'' \}
\]

or

\[
\text{there exists } H'' = (G'', (\nu_i, \delta_i)_{i \in I}) \in HS'' \text{ s.t. } H'' \sim H \text{ and } H' \not \sim H, \forall H' \in HS'
\]

or

\[
\{ H' = (G', (\nu_i', \delta_i')_{i \in I'}) \in HS' \text{ and } H'' = (G'', (\nu_i'', \delta_i'')_{i \in I''}) \in HS'' \text{ s.t. } H' \sim H, H'' \sim H \text{, and } (\nu_i, \delta_i)_{i \in I} = (\nu_i', \delta_i'[h'])_{i \in I'} \cup (\nu_i'', \delta_i''[h'])_{i \in I''},
\]

where \( h' : N' \to N \) and \( h'' : N'' \to N \) are the bijections between nodes induced by the isomorphism relation.

159
Chapter 6. Analysis of Programs Manipulating Singly-Linked Lists

- For any $HS', HS'' \in H\mathcal{S}(\Sigma, k)$, $HS' \cap HS''$ is the set of all heaps $H = (G, (\nu_i, \delta_i)_{i \in I})$ such that there exist two heaps $H' = (G', (\nu_i', \delta_i')_{i \in I'})$ in $HS'$ and $H'' = (G'', (\nu_i'', \delta_i'')_{i \in I''})$ in $HS''$ such that $H' \sim H$, $H'' \sim H$, and $(\nu_i, \delta_i)_{i \in I} = (\nu_i', \delta_i')_{i \in I'} \cap (\nu_i'', \delta_i'')_{i \in I''}$, where $h' : N' \rightarrow N$ and $h'' : N'' \rightarrow N$ are the bijections between nodes induced by the isomorphism relation.

In the following, we make an abuse of notation. For any two isomorphic data word $k$-SLL heaps $H \sim H'$ with $H = (G, (\nu_i, \delta_i)_{i \in I})$ and $H' = (G', (\nu_i', \delta_i')_{i \in I'})$, we assume implicit the renaming of nodes induced by the isomorphism when using $(\nu_i', \delta_i')_{i \in I'}$ and $(\nu_i, \delta_i)_{i \in I}$.

We define a concretization function $\gamma : H\mathcal{S}(\Sigma, k) \rightarrow \mathcal{P}(H_{k-SLL}(\Sigma))$ by:

$$\gamma((G, (\nu_i, \delta_i)_{i \in I})) = \{(G, \nu_i, \delta_i) \mid i \in I\}.$$ 

This function is extended to data word $k$-SLL heap sets as usual. Notice that the function $\gamma$ defined on heap sets is a bijection.

![Figure 6.5: The data word k-SLL heap H.](image_url)

Example 6.3.2. The data word 0-SLL heap $H$ given in Figure 6.5 represents the set of contracted 0-SLL heaps from Figures 6.2(b) and 6.2(d). We have that $\gamma(H) = \{H_0^0, H_0^1\}$.

The fact that $\sqcup$ and $\sqcap$ define least upper bounds and greatest lower bounds is obtained from the following relations (1) $HS' \sqsubseteq HS''$ iff $\gamma(HS') \subseteq \gamma(HS'')$, (2) $HS' \sqcup HS'' = \gamma^{-1}(\gamma(HS') \sqcup \gamma(HS''))$, and (3) $HS' \sqcap HS'' = \gamma^{-1}(\gamma(HS') \cap \gamma(HS''))$.

The function $\gamma$ defined above is a concretization function from the lattice $C_{H\mathcal{S}(\Sigma, k)}$ to the lattice $C_{k-SLL}(\Sigma)$. Because it is a bijection we have that $(\gamma^{-1}, \gamma)$ is a Galois insertion between $C_{H\mathcal{S}(\Sigma, k)}$ and $C_{k-SLL}(\Sigma)$. Notice that $\gamma^{k-SLL} \circ \gamma$ is a concretization function from the lattice $C_{H\mathcal{S}(\Sigma, k)}$ to the lattice $C(\Sigma)$. Also, $(\gamma^{-1} \circ \alpha^{k-SLL}, \gamma^{k-SLL} \circ \gamma)$ is a Galois connection between $C_{H\mathcal{S}(\Sigma, k)}$ and $C(\Sigma)$:

![Diagram](image_url)

Example 6.3.3. Consider the contracted 1-SLL heaps $H_0^0$, $H_0^1$, and $H_1^1$ from Figure 6.2(b), Figure 6.2(d), and Figure 6.2(e), respectively. Notice that any contracted
Section 4.3.3.4 we defined the transformers \(Pr\) undefined only in \(\#\) \(N\) structured into a lattice \(C\) where \(\\{H, H^1_w\}\) because these heaps have non-isomorphic underlying graphs.

6.3.1 Remark. Any data word \(k\)-SLL heap \(H \in \mathcal{H}(\Sigma, k)\) has an underlying graph whose number of nodes is linear in the number of program pointer variables plus \(k\).

As a consequence, each data word \(k\)-SLL heap set \(HS \in \mathcal{HS}(\Sigma, k)\) contains a bounded number of data word \(k\)-SLL heaps. This bound is exponential in the number of pointer variables in the program and \(k\).

Based on Remark 6.3.1, we suppose that the names of the nodes of a data word \(k\)-SLL heap come from a fixed set \(\mathcal{N}_{\Sigma, k}\) which includes the two distinguished nodes \(\#\) and \(\#'\). A function \(\nu\) from a data word \(k\)-SLL heap associates a non-empty word to every node in \(\mathcal{N}_{\Sigma, k}\), except for \(\#\) and \(\#'\). From this point of view, we can think of \(\mathcal{N}_{\Sigma, k}\) as a set of variables, called \textit{data word variables} (word variables, for short), interpreted as non-empty words over \(\Sigma\) according to \(\nu\). The families \((\nu, \delta_i)_{i \in I}\) from data word \(k\)-SLL heaps can be structured into a lattice \(C_{W}(\Sigma, \mathcal{N}_{\Sigma, k})\), called \textit{the concrete data words domain over} \(\Sigma\) and \(\mathcal{N}_{\Sigma, k}\), defined by:

\[
C_{W}(\Sigma, \mathcal{N}_{\Sigma, k}) = (\mathcal{P}([N_{\Sigma, k} \rightarrow \mathbb{D}^+]) \times \{DVar \rightarrow \mathbb{D}\}), \subseteq, \cup, \cap, \emptyset, [N_{\Sigma, k} \rightarrow \mathbb{D}^+] \times \{DVar \rightarrow \mathbb{D}\}),
\]

where \([N_{\Sigma, k} \rightarrow \mathbb{D}^+]\) denotes the set of all partial functions \(\nu : N_{\Sigma, k} \rightarrow \mathbb{D}^+\), which are undefined only in \(\#\) and \(\#'\) and \([DVar \rightarrow \mathbb{D}]\) denotes the set of all functions \(\delta : DVar \rightarrow \mathbb{D}\).

6.3.2 Abstract semantics

Let \(Pr\) be a program over a type system \(\Sigma_{Pr}\) as in 6.1.1. In Section 4.2.3.3 and Section 4.3.3.4 we defined the transformers \(U[a] : \mathcal{P}(R(\Sigma)) \rightarrow \mathcal{P}(R(\Sigma))\) and \(\overline{U}[a] : \mathcal{P}(R(\Sigma)) \times \mathcal{P}(R(\Sigma)) \rightarrow \mathcal{P}(R(\Sigma))\) over pairs of memory configurations, where \(a\) is an edge label of the ICFG associated with \(Pr\). In the previous section, we have defined the lattice \(C(\Sigma_{Pr}, k)\) of data word \(k\)-SLL heap sets which is connected through the Galois connection \((\alpha_{k-SLL} \circ \gamma^{-1}, \gamma \circ \gamma^{k-SLL})\) to the lattice \(C'(\Sigma)\) of sets of program relations. For programs with singly-linked lists, the lattice \(C(\Sigma_{Pr})\) is exactly the lattice \(C'(\Sigma)\) of subsets of \(R(\Sigma)\) introduced in Section 4.3.3.4.

Next, we define the abstract transformers \(U^k_{\mathcal{HS}}[a] : \mathcal{HS}(\Sigma_{Pr}, k) \rightarrow \mathcal{HS}(\Sigma_{Pr}, k)\) and \(\overline{U}^k_{\mathcal{HS}}[a] : \mathcal{HS}(\Sigma_{Pr}, k) \times \mathcal{HS}(\Sigma_{Pr}, k) \rightarrow \mathcal{HS}(\Sigma_{Pr}, k)\) corresponding to \(U[a]\) and \(\overline{U}[a]\). These abstract transformers are precise, i.e. \(\gamma' \circ U^k_{\mathcal{HS}}[a] = U[a] \circ \gamma'\) and \(\gamma' \circ \overline{U}^k_{\mathcal{HS}}[a] = \overline{U}[a] \circ \gamma'\), for any \(a \neq \textbf{assert} \psi\) and \(a \neq \textbf{assume} \psi\), where \(\gamma' = \gamma^{k-SLL} \circ \gamma\) is the concretization function from the lattice of heap sets to the lattice of SLL heaps. For \(a \neq \textbf{assert} \psi\) and \(a \neq \textbf{assume} \psi\) we define sound transformers, that is \(\gamma' \circ \overline{U}^k_{\mathcal{HS}}[a] \subseteq \overline{U}[a] \circ \gamma'\).

Let HS be a heap set in \(\mathcal{HS}(\Sigma_{Pr}, k)\). For every edge label \(a\) of the ICFG, corresponding to assignments of pointer variables or data variables, boolean conditions, formulas from \texttt{assume/assert} statements, and procedure calls, we define:

\[
U^k_{\mathcal{HS}}[a](HS) = \bigcup_{H \in HS} U^k_{\mathcal{H}}[a](H),
\]

where \(U^k_{\mathcal{H}}[a] : \mathcal{H}(\Sigma_{Pr}, k) \rightarrow \mathcal{HS}(\Sigma_{Pr}, k)\) is the transformer \(U^k_{\mathcal{HS}}[a]\) restricted to heaps (or equivalently, restricted to heap sets containing only one heap). Analogously, for every
edge label $a$ which is a procedure return, we define

$$\mathcal{U}_H^k[a](HS, HS') = \bigcup_{H \in HS} \mathcal{U}_H^k[a](H, H'),$$

where $\mathcal{U}_H^k[a] : \mathcal{H}(\Sigma_{Pr}, k) \times \mathcal{H}(\Sigma_{Pr}, k) \rightarrow \mathcal{H}(\Sigma_{Pr}, k)$ is the restriction of $\mathcal{U}_H^k[a]$ to heaps.

Next, we formally define $U_H^k[a](H)$ and $\mathcal{U}_H^k[a](H, H')$ for any $H = (G = (N, S, L), (\nu, \delta_i)_{i \in I})$ and $H' = (G' = (N', S', L'), (\nu', \delta'_i)_{i' \in I'})$ two heaps in $\mathcal{H}(\Sigma_{Pr}, k)$. Their definition uses a set of transformers on the elements of $\mathcal{C}_W(\Sigma_{Pr}, N_{\Sigma_{Pr}, k})$ which are sets of pairs between a node labeling with words over $D$ and a valuation for the variables in $DVar$.

$$\mathcal{C}_W(\Sigma, N_{\Sigma, k}) = (P([N_{\Sigma, k} \rightarrow D^+] \times [DVar \rightarrow D]), \subseteq, \cup, \cap, \emptyset, [N_{\Sigma, k} \rightarrow D^+] \times [DVar \rightarrow D]),$$

The transformers on $\mathcal{C}_W(\Sigma_{Pr}, N_{\Sigma_{Pr}, k})$ are introduced when they are used for the first time in the definition of $U_H^k[a]$. In the following, $\text{len}(w)$ denotes the length of $w$ (i.e., $w \in D^{\text{len}(w)}$). Also, $[]$ (|[|]) denotes the empty word (the word with one element $e$) and $\bullet$ denotes the word concatenation operator. Finally, $\text{hd}(w)$ and $\text{tl}(w)$ denote the first element and respectively, the tail of the word $w$ (i.e., $\text{hd}(w) \in D$ and $w = \text{hd}(w) \bullet \text{tl}(w)$) for $w \neq []$.

Figure 6.6: The transformers $U_H^k[p = \text{new list}()]$ and $\text{sglt}$

**New statement** When $a$ is a statement of the form $p := \text{new list}()$ that creates a new object of type list, $U_H^k[a](H)$ is built from $H$ by (1) adding a new node $n$ to the graph $G$, labeled with $p$, having no predecessor, and with $\sharp$ as successor, and (2) extending the definition of each $\nu_i$ with $i \in I$ to the new node $n$ such that $\nu_i(n)$ is a arbitrary word of length one. Extending the definition of $\nu_i$ in this way corresponds to a transformer in the domain $\mathcal{C}_W$ called $\text{sglt}$ with

$$\text{sglt} : N_{\Sigma_{Pr}, k} \times P([N \rightarrow D^+] \times [DVar \rightarrow D]) \rightarrow P([N \rightarrow D^+] \times [DVar \rightarrow D]).$$

The rules 6.3.5 and 6.3.3 from Figure 6.3.2 give the formal definition of $\text{sglt}$ and $U_H^k[a]$, respectively.

In the following, we consider assignments of the form $d = dt$ or $p->\text{data=} dt$, where $dt$ is some data expression as in Figure 4.2. For simplicity, we define only $U_H^k[d = p->dt]$.
and $U^k_H[d\rightarrow dt = d]$, where $p$ is a pointer variable and $d$ is a data variable. The extension to general data expressions is straightforward. As a general remark, notice that $p\rightarrow dt$ denotes a value in $\mathbb{D}$, which is the first element of the word associated with $L(p)$.

**Assignments $d = p\rightarrow dt$** If $L(p) = \emptyset$ then $U^k_H[d = p\rightarrow dt](H) = H_{err}$ because null pointer dereferencing has occurred ($H_{err}$ is a distinguished heap that represents runtime errors).

Otherwise, the transformer $U^k_H[d = p\rightarrow dt]$ modifies the value of the data variable $d$ using a transformer $\text{updDvar}$ on the lattice $C_W$. This transformer takes as input a data variable $d \in DVar$, a word variable $n = L(p) \in \mathcal{N}_{pr,k}$, and a set of pairs of functions $(\nu_i, \delta_i)_{i \in I}$. For each $i \in I$, it sets $\delta_i(d)$ to be equal to the first element of the word associated by $\nu_i$ with $n$, denoted by $hd(\nu(L(p)))$. The formal definitions are given in Figure 6.7.

$$H = (\langle N, S, L \rangle, (\nu_i, \delta_i)_{i \in I}) \quad L(p) = \emptyset$$
$$U^k_H[d = p\rightarrow dt](H) = H_{err}$$

$$H = (\langle N, S, L \rangle, (\nu_i, \delta_i)_{i \in I}) \quad L(p) \neq \emptyset$$
$$U^k_H[d = p\rightarrow dt](H) = (\langle N, S, L \rangle, \text{updDvar}(d, L(p)(\nu_i, \delta_i)_{i \in I}))$$

for any $i \in I$, $\nu_i$ is defined in $n$$\text{updDvar}(d, n, (\nu_i, \delta_i)_{i \in I}) = \bigcup_{i \in I} \text{updDvar}(d, n, (\nu_i, \delta_i))$

$$\nu(n) = [w_0, w_1, \ldots, w_{n-1}]$$
$$\text{updDvar}(d, n, \nu, \delta) = (\nu, \delta[d \mapsto w_0])$$

Figure 6.7: The transformers $U^k_H[d = p\rightarrow dt]$ and $\text{updDvar}$.

**Assignment $p\rightarrow dt = d$** Like previously, $U^k_H[p\rightarrow dt = d](H) = H_{err}$ in case of a null pointer dereferencing, that is $L(p) = \emptyset$.

The transformer $U^k_H[p\rightarrow dt = d]$ updates the value of the first element in the word associated with $L(p)$ to be equal with the value of $d$, in any pair $(\nu_i, \delta_i)$ with $i \in I$. To change the concrete valuations $(\nu_i, \delta_i)_{i \in I}$, it uses a transformer over the concrete data words domain, denoted $\text{updFirst}$. This transformer takes as input a data variable $d$, a word variable $n \in \mathcal{N}_{pr,k}$, and a set of pairs of functions $(\nu_i, \delta_i)_{i \in I}$ from $C_W$ and modifies the definition of each $\nu_i(n)$ such that $\nu_i(n) = [\delta_i(d), w_1, \ldots, w_{n-1}]$, where $[w_0, w_1, \ldots, w_{n-1}]$ is the old value of $\nu_i(n)$. The formal definition of these transformers is given in Figure 6.8.

**Example 6.3.4.** Consider the assignment $\text{aux}\rightarrow dt = \text{aux}\rightarrow dt + v$ from the procedure $\text{addV}$ in Figure 4.19 (at the control point 31). Figure 6.9(a) pictures a heap, denoted by $H$, which represents a pair of memory configurations possible at the control point preceding this assignment. Then, Figure 6.9(b) shows the heap obtained after applying the transformer corresponding to this statement. The new value for $\nu(n_3)$ is obtained by applying the rule for $\text{updFirst}$ given in Figure 6.8. equation 6.3.6 where the variable $d$ is replaced by the expression $\text{aux}\rightarrow dt + v$ (this expression is evaluated to 7).

Similarly, if we apply the transformer corresponding to the statement $\text{headi}\rightarrow dt = \text{headi}\rightarrow dt + v$ over the heap in Figure 6.9(b) then we obtain the same graph but the value of $\nu$ in $n_4$ is changed to $[4, 9, 3]$. 

163


CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

\[
\begin{align*}
H &= ((N, S, L), (\nu_i, \delta_i)_{i \in I}) \quad L(p) = S \\
U^k_H[p->dt = d](H) &= H_{err}
\end{align*}
\]

\[
\begin{align*}
H &= ((N, S, L), (\nu_i, \delta_i)_{i \in I}) \quad L(p) \neq S \\
U^k_H[p->dt = d](H) &= ((N, S, L), \text{ updFirst(d, } L(p), (\nu_i, \delta_i)_{i \in I}))
\end{align*}
\]

for any \( i \in I, \nu_i \) is defined in \( n \)

\[
\text{ updFirst(d, n, (\nu_i, \delta_i)_{i \in I})} = \bigcup_{i \in I} \text{ updFirst(d, n, (\nu_i, \delta_i))},
\]

\[
\nu(n) = [w_0, w_1, \ldots, w_{n-1}], \quad w' = [\delta(d), w_1, \ldots, w_{n-1}]
\]

Figure 6.8: The transformers \( U^k_H[p->dt = d] \) and \( \text{ updFirst} \).

Concerning pointer assignments, we recall that every assignment of the form \( p = \ldots \) and \( p->\text{next} = \ldots \) is preceded by \( p = \text{null} \) and, respectively, \( p->\text{next} = \text{null} \). Consequently, the only statements that create garbage are \( p = \text{null} \) and \( p->\text{next} = \text{null} \).

**Assignment** \( q = p->\text{next} \) If \( p \) labels \( \neq \) in \( H \) then \( U^k_H[q = p->\text{next}](H) = H_{err} \) because null pointer dereferencing has occurred.

Now consider the case when \( L(p) = n \) and \( n \neq \neq \). Recall that \( H \) represents a set of SLL heaps, one for each pair of functions \( (\nu_i, \delta_i) \) with \( i \in I \). The term \( p->\text{next} \) denotes the immediate successor of \( n \) in each of these SLL heaps. Given a pair of functions \( (\nu_i, \delta_i) \), \( p->\text{next} \) is either interpreted as \( S(n) \) if the word associated by \( \nu_i \) to \( n \) is a word of length one or as a new node that does not appear in \( N \) labeled with the tail of the word \( \nu_i(n) \) (the tail of a word is obtained by deleting its first element).

To be able to interpret the term \( p->\text{next} \) we define a transformer

\[
\text{ unfold : } N_{\Sigma_{pr}, k} \times \mathcal{H}(\Sigma_{pr}, k) \to \mathcal{H}S(\Sigma_{pr}, k + 1),
\]

that takes as argument a heap \( H \) and a node \( n \) of \( H \) (remember that we have fixed a set of node names \( N_{\Sigma_{pr}, k} \)) and returns a heap set \( HS \in \mathcal{H}S(\Sigma_{pr}, k + 1) \), such that (1) \( HS \) and \( H \) denote the same set of SLL heaps, i.e. \( \gamma'(H) = \gamma'(HS) \) and (2) in any heap \( H' \in HS \), all vertex labellings associate with \( n \) a word of length one.
6.3. A FAMILY OF ABSTRACT DOMAINS

\[ H = ((N,S,L),(v_i,\delta_i)_{i\in I})) \quad L(p) = \varepsilon \]

\[ U^k_H[q = p->next](H) = H_{err} \]

\[ H = ((N,S,L),(v_i,\delta_i)_{i\in I}), \quad n' \notin N, \]
\[ H_1 = \left( ((N,S,L),selectSglt(n,(v_i,\delta_i)_{i\in I})) \right), \]
\[ H_{>1} = \left( (N \cup \{n'\},S[n'\mapsto S(n),n\mapsto n'],L),split(n,n',selectNonSglt(n,(v_i,\delta_i)_{i\in I})) \right) \]
\[ \text{unfold}(n,H) = \{H_1,H_{>1}\} \quad (6.3.7) \]

\[ L(p) \neq \varepsilon \]
\[ \text{unfold}(L(p),H) = \{H_1 = ((N_1,S_1,L_1),(v^1_i,\delta^1_i)_{i\in I_1})\}, \]
\[ H_{>1} = ((N_{>1},S_{>1},L_{>1}),(v^{>1}_i,\delta^{>1}_i)_{i\in I_{>1}})) \]
\[ H'_1 = ((N_1,S_1,L_1[q \mapsto S(L_1(p))]),(v^1_i,\delta^1_i)_{i\in I_1}) \]
\[ H'_{>1} = ((N_{>1},S_{>1},L_{>1}[q \mapsto S(L_{>1}(p))]),(v^{>1}_i,\delta^{>1}_i)_{i\in I_{>1}}) \quad (6.3.8) \]
\[ U^k_H[q = p->next](H) = \{H'_1,H'_{>1}\} \]

Figure 6.10: The transformers \( U^k_H[q = p->next](H) \) and unfold.

The transformer unfold is formally defined in Figure 6.10. The output of unfold\((n,H)\) is a heap set with at most two heaps \( H_1 \) and \( H_{>1} \).

The heap \( H_1 \) represents the SLL heaps from the concretization of \( H \) in which \( n \) and \( S(n) \) are immediate successors. These SLL heaps correspond to the set of valuations \((v_i,\delta_i)\) that associate with \( n \) words of length one. To compute these valuations, we define the transformer selectSglt on the elements of the concrete data words domain, \( C_W \). Its definition is given in Figure 6.11 by the rule 6.3.9. It has the same graph as \( H \) and it contains the vertex labelings that associate with \( n \) words of length one (if any).

The heap \( H_{>1} \) represents the SLL heaps from the concretization of \( H \) in which \( n \) and \( S(n) \) are not immediate successors. These SLL heaps correspond to the set of valuations \((v_i,\delta_i)\) that associate with \( n \) words of length greater than one. To compute these valuations, we introduce a transformer selectNonSglt on the elements of the lattice \( C_W \). Its definition is given in Figure 6.11 by the rule 6.3.10. Then, \( H_{>1} \) is obtained by adding a new node \( n' \notin N \) to the graph, between \( n \) and \( S(n) \). The labeling with words is redefined, such that if \( n \) was labeled by \( w \) in \( H \) then \( n \) is labeled by \( hd(w) \) and \( n' \) is labeled by \( tl(w) \) in \( H_{>1} \). This transformation is done using the transformer split, given in Figure 6.12 which is applied on elements of \( C_W \) that associate \( n \) with a word of length greater than one.

Finally, we define \( U^k_H[q = p->next](H) = HS \), where HS is obtained from HS' = unfold\((H,L(p))\) by labeling the successor of \( L(p) \) with q in each heap in HS'. This transformation does not increase the number of simple nodes because (1) the simple node introduced by unfold is labeled by q and (2) q labels \( \varepsilon \) in H (this holds because any assignment \( q = p->next \) is preceded by \( q = \text{null} \)). The latter implies also that this transformation does not produce garbage.

Example 6.3.5. Consider the assignment headi=aux->next from the procedure addV in Figure 4.19 (at the control point 30). Figure 6.13(a) pictures a heap, denoted by \( H \),

165
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

(ν_j, δ_j)_{j \in J} \subseteq (ν_i, δ_i)_{i \in I}
for any \( i \in I \), \( \text{len}(ν_i(n)) = 1 \) iff there is \( j \in J \) s.t. \((ν_j, δ_j) = (ν_i, δ_i)\) \hspace{1cm} (6.3.9)

selectSglt(n, (ν_i, δ_i)_{i \in I}) = (ν_j, δ_j)_{j \in J}
for any \( i \in I \), \( \text{len}(ν_i(n)) = 1 \) iff there is \( j \in J \) s.t. \((ν_j, δ_j) = (ν_i, δ_i)\) \hspace{1cm} (6.3.10)

selectNonSglt(n, (ν_i, δ_i)_{i \in I}) = (ν_j, δ_j)_{j \in J}

Figure 6.11: The transformers selectSglt and selectNonSglt.

\[
\text{split}(n, n', (ν, δ)) = (ν[n] → \text{hd}(ν[n]), n' → \text{tl}(ν[n]), δ)
\]

\[
\text{len}(ν(n)) > 1 \quad \text{and} \quad ν \text{ is undefined in } n'
\]

\[
\text{split}(n, n', (ν, δ)) = \emptyset
\]

\[
\text{split}(n, n', (ν_i, δ_i)_{i \in I}) = \bigcup_{i \in I} \text{split}(n, n', (ν_i, δ_i))
\]

Figure 6.12: The transformer split.

that represents a pair of memory configurations possible at the control point preceding this assignment. The heap \( H' \) representing the pair of memory configurations reached at the next control point is given in the Figure 6.13(b). To define the words associated with the nodes labeled by aux and headi in \( H' = \bigcup_{i \in I} \text{headi} = \text{aux→next}|(H) \), the transformer \( \text{split}(n_3, n_4, (ν, δ)) \) is used. This transformer splits the word \( ν(n_3) = [5, 2, 9, 3] \) into \( \text{hd}([5, 2, 9, 3]) = [5] \) and \( \text{tl}([5, 2, 9, 3]) = [2, 9, 3] \), which label \( n_3 \) and \( n_4 \), respectively.

Figure 6.13: Applying the transformer for the statement headi=aux->next from the procedure addV in Figure 4.19

Example 6.3.6. Consider the assignment tmp=x->next from the procedure main in Figure 4.5 (at the control point 17). The heap in Figure 6.14(a), denoted by \( H^a \), pictures a pair of memory configurations possible at the control point preceding this assignment (because the procedure main has no parameters, the input configuration is empty). Then, the pair of memory configurations reachable at the control point 18 is represented by the heap

Figure 6.14: The transformers selectSglt and selectNonSglt.
set in Figure 6.14(b). This heap set contains two heaps. The first one, $H^a_1$, is obtained from $H$ by restricting it to $(\nu_2, \delta_2)$ (according to rules 6.3.7 and 6.3.9). It corresponds to the case when the node labeled by $x$ in $H^a$, denotes a list segment of length one. The second heap, $H^a_{>1}$ represents the SLL heaps where the length of the list pointed by $x$ is greater than one. In this case, to obtain the words associated with $n_1$ and the new node $n_4$, the transformer split$(n_1, n_4, (\nu_1, \delta_1))$ over pairs of valuations is used. According to the definition of split given in Figure 6.12, $n_1$ is associated with the first element of $\nu_1(n_1)$ and $n_4$ is associated with the rest of this word.

Figure 6.14: Applying the transformer for the statement $\text{tmp} = x->\text{next}$ from the procedure $\text{main}$ in Figure 4.5

**Garbage collector** The transformers associated with the statements $p = \text{null}$ and $p->\text{next} = \text{null}$ use a function called $\text{RemGrb}$ that takes as input a heap that might contain nodes which are not reachable from any node labeled by a program variable and deletes these nodes:

$$\text{RemGrb}(H) = \text{proj}(\text{getGarbage}(H), H)$$ (6.3.11)
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

The operator RemGrb\(H\) removes from \(H = ((N, S, L), (\nu_i, \delta_i)_{i \in I})\) the garbage nodes using two operators:

1. \textit{getGarbage}(\(H\)) returns the complete set of garbage nodes (computed, e.g., by a graph traversal algorithm starting from the nodes pointed to by program variables);

2. \textit{proj}(\(M, H\)) is a transformer that removes from \(H\) a set of nodes \(M \subset N\). The resulting heap is \(((N', S', L), \text{proj}(M, (\nu_i, \delta_i)_{i \in I}))\) such that \(N' = N \setminus M\), the successor function \(S'\) is the restriction of \(S\) to the nodes in \(N'\), the labeling with pointer variables is unchanged because no labeled nodes are deleted, and \textit{proj} is a transformer on elements of \(C_W\) that removes the variables in \(M\) from the domain of each \(\nu_i\) with \(i \in I\) (defined by rule 6.3.13). The definition of \textit{proj}(\(M, H\)) is given by the rule 6.3.12:

\[
H = ((N, S, L), (\nu_i, \delta_i)_{i \in I}) \\
N' = N \setminus M, S' = S|_{N'}, L' = L \\
\text{proj}(M, H) = ((N', S', L'), \text{proj}(M, (\nu_i, \delta_i)_{i \in I}))
\]  \hspace{1cm} (6.3.12)

\[
\text{the domain of } \nu_i \text{ is } N, \text{ for all } i \in I \\
\text{proj}(M, (\nu_i, \delta_i)_{i \in I}) = (\nu_i|_{N \setminus M}, \delta_i)_{i \in I}
\]  \hspace{1cm} (6.3.13)

Assignments \(p = \text{null}\) The transformer corresponding to \(p = \text{null}\) moves the label \(p\) to the node \(\sharp\). If the node labeled by \(p\) in \(H\) is reachable from any other node in \(H\) (i.e., it is not eliminated by the garbage collector) then it becomes a simple node. Therefore, if \(H\) has exactly \(k\) simple nodes, the resulting heap would have \(k + 1\) simple nodes and it would not be an element of \(H(\Sigma_{Pr}, k)\). In this situation, we perform an additional transformation called \textit{folding} that, given a data word \(k'\)-SLL heap with \(k' > k\), transforms it into a data word \(0\)-SLL heap (i.e., a heap with no simple nodes).

We define hereafter the transformer \(U^p_H[p = \text{null}](H)\). There are two situations when the number of simple nodes increases:

1. the first one is when the node \(n = L(p)\) is reachable from a node labeled by some other program variable \(p' \in \text{PVar}\). If \(p\) is the only pointer variable labeling \(n\) then removing it transforms \(n\) into a simple node;

2. the second one is when the node \(n = L(p)\) is not reachable from any other node labeled by some program variable but, there is a sharing node \(n'\) such that \(n'\) is reachable from \(n\). If \(n'\) has only two predecessors in \(H\) then, by making \(p\) point to \(\sharp\), \(n'\) is no longer a sharing node, it becomes a simple node.

In each of these cases, if \(H\) has exactly \(k\) simple nodes then labeling \(\sharp\) with \(p\) transforms it into a heap with \(k + 1\) simple nodes. Therefore, \(U^p_H[p = \text{null}](H)\) first labels \(\sharp\) with \(p\) and removes all the garbage if any; let \(H'\) denote the heap obtained after these transformations. Then, it folds all simple nodes in \(H'\), i.e. \(U^p_H[p = \text{null}](H) = \text{fold}(H')\). The rule 6.3.14 in Figure 6.15 formally defines these transformation.

We define the transformer \(\text{fold}: HS(\Sigma_{Pr}, k') \rightarrow HS(\Sigma_{Pr}, k)\) with \(k' > k\) by

\[
\text{fold}(HS) = \bigcup_{H \in HS} \text{fold}(H),
\]  \hspace{1cm} (6.3.15)
6.3. A FAMILY OF ABSTRACT DOMAINS

\[ U^k_{H[p = \text{null}]}(H) = \text{fold}(\text{RemGrb}((N, S, L[p \mapsto \sharp]), (\nu_i, \delta_i)_{i \in I})) \]  \hspace{1cm} (6.3.14)

Figure 6.15: The transformer \( U^k_{H[p = \text{null}]} \)

where

\[ \text{fold}(H) = \begin{cases} H, & \text{if } H \in \mathcal{H}S(\Sigma^r, k) \\ H', & \text{otherwise}, \end{cases} \]  \hspace{1cm} (6.3.16)

where \( H' \) is the heap obtained after the elimination of all simple nodes from \( H \), defined by the rule 6.3.17 from Figure 6.16. In this figure \( \text{last}(\text{Simple}(H, n)) \) denotes the last element of the vector \( \text{Simple}(H, n) \). Roughly, \( H' \) is obtained by:

1. adding a fresh node \( \pi \) for every crucial node \( n \) of \( H \) with \( \text{Simple}(H, n) \neq \emptyset \);
2. for every \( \pi \), for every \( (\nu_i, \delta_i) \), \( \nu_i(\pi) \) is obtained by applying \( \text{concat}(M, (\nu_i, \delta_i)) \), where \( M \) is an array of pair, such that the first element of each pair is a node \( n \) and the second is a sequence of nodes \( n \cdot \text{Simple}(H, n) \), such that the word associated to \( \pi \) corresponds to the concatenation of these words associated to this sequence of nodes;
3. any predecessor of \( n \) becomes a predecessor of \( \pi \);
4. any successor of \( \text{last}(\text{Simple}(H, n)) \) becomes a successor of \( \pi \);
5. any label on \( n \) becomes a label of \( \pi \);
6. all simple nodes are deleted from \( H \).

It uses a transformer \( \text{concat} \) on elements of \( \mathcal{C}_W \), defined by the rule 6.3.18 from Figure 6.16, whose signature is:

\[ \text{concat}: \mathcal{P}(N_{\Sigma^r, p, k} \times N_{\Sigma^r, p, k}) \times \mathcal{P}([N \rightarrow D^+] \times [DVar \rightarrow D]) \rightarrow \mathcal{P}([N \rightarrow D^+] \times [DVar \rightarrow D]). \]

The transformer \( \text{concat} \) receives as input a set of pairs \((\overline{\nu}_i, V_i), 1 \leq i \leq p \), where \( \overline{\nu}_i \) is a word variable and \( V_i \) is a vector of word variables, and an element of \( \mathcal{C}_W \), \((\nu_i, \delta_i)_{i \in I}\), and returns an element of \( \mathcal{C}_W \) obtained by extending the definition of each \( \nu_i \) to the set of variables \( \{\overline{\nu}_1, \ldots, \overline{\nu}_p\} \) such that \( \nu(\pi) \) equals the concatenation of the words associated to the variables in \( V_i \).

Example 6.3.7. Consider the assignment \( \text{aux} = \text{null} \) from the procedure \( \text{addV} \) in Figure 4.19 (at the control point 32). Suppose that we execute this statement on the heap from Figure 6.9(b) denoted by \( H' \). Moreover, suppose that the bound on the number of simple nodes is 1, i.e. \( k = 1 \). Then, the transformer \( U^k_{H[\text{aux} = \text{null}]}(H') \) moves the label \( \text{aux} \) to \( \sharp \) and the node \( n_3 \) will become a simple node. The result is a heap with one simple node. If we continue the execution then, the first time we reach the control point 32, we obtain a pair of memory configurations described by the data word \( 1-\text{SLL} \) heap \( H \) in Figure 6.17(a). Applying \( U^1_{H[\text{aux} = \text{null}]}(H) \) will first change the node labeling. By moving the label \( \text{aux} \) to \( \sharp \), the graph becomes a heap with 2 simple nodes. Thus, the second phase of \( U^1_{H[\text{aux} = \text{null}]}(H) \) folds all the simple nodes. We have that
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

\[ H = ((N, S, L), (\nu_i, \delta_i)_{i \in I}) \]

\[ |Simple(H)| > k \quad V' = \{n \in N \setminus Simple(H) \mid Simple(H, n) \neq \emptyset\} \]

\[ \text{proj}(Simple(H) \cup V', H) = H' = ((N', S', L), (\nu'_i, \delta'_i)_{i \in I}) \]

\[ M = \{ (\pi, n \cdot Simple(H, n)) \mid n \in V' \} \]

for any \( n \in N \), \( S^{-1}(n) = n' \) iff \( S(n') = n \)

\[
\text{fold}(H) = \left((N' \cup \{ \pi \mid n \in V' \},
S[S^{-1}(n) \mapsto \pi, \pi \mapsto S(\text{last}(Simple(H, n))))_{n \in V'}, L'[^p \mapsto \pi]_{n \in V', L(p) = n} ,
\text{concat}(M, (\nu_i, \delta_i)_{i \in I})\right)
\]

(6.3.17)

\[
\text{concat}(M, (\nu_i, \delta_i)_{i \in I}) = \bigcup_{i \in I} \text{concat}(M, (\nu_i, \delta_i))
\]

(6.3.18)

Figure 6.16: The transformer \text{fold} over \( \mathcal{H}(\Sigma_{p_p}, k) \) and the transformer \text{concat} over \( \mathcal{C}_W \)

\[
\nu(n_1) = [1, 6, 7, 5, 2, 9, 3]
\]

\[
\nu(n_2) = [3, 8, 9]
\]

\[
\nu(n_3) = [4]
\]

\[
\nu(n_4) = [9, 3]
\]

\[
\delta(v) = 2
\]

(a) A heap \( H \)

\[
\nu(n_1) = [1, 6, 7, 5, 2, 9, 3]
\]

\[
\nu(n_2) = [3, 8, 9, 7, 4]
\]

\[
\nu(n_3) = [9, 3]
\]

\[
\delta(v) = 2
\]

(b) The heap \( U'_k[\text{aux} = \text{null}](H) \)

Figure 6.17: Applying the transformer for the statement aux = null from the procedure \text{add}V in Figure 4.19

Simple(\( H \)) = Simple(\( H, n_2 \)) = n_3 \cdot n_4. Therefore, the words associated with the simple nodes to eliminate are concatenated to the word associated with \( n_2 \) such that the information they store is not lost (rules 6.3.17 and 6.3.18). The definition of the transformer \text{concat}(\{n_2 n_3 n_4\}(\nu, \delta)) is straightforward: it concatenates the words \( \nu(n_2), \nu(n_3), \) and \( \nu(n_4) \) in the order given by the vector of word variables received as argument. The result of the concatenation is associated with \( n_2 \). The order in which words are concatenated is important and it corresponds to the order induced by the reachability relation between the corresponding nodes in the graph.

Example 6.3.8. Consider the program Dispatch in Figure 4.3. As in the previous example, \( k = 1 \). Let us continue the execution of this procedure using data word 1-SLL heaps
6.3. A FAMILY OF ABSTRACT DOMAINS

Figure 6.18: Applying the transformer for the statement $x = \text{null}$ from the program Dispatch in Figure 4.5

starting from the heap set \{${H_1^a, H_{>1}^a}$\} in Figure 6.14(b). After one more loop iteration, $H_1^a$ is transformed into the heap $H_1^b$ given in Figure 6.18 and the heap $H_{>1}^a$ is modified by adding a simple node between the node labeled by sm and $\delta$. From $H_1^b$, there are no more new configurations to reach because the condition of the loop is no longer true. On the other hand, the loop condition still holds on the heap obtained from $H_{>1}^a$. The heap $H_{>1}^b$ in Figure 6.18 represents the memory configuration reached from $H_{>1}^a$ after one more loop iteration at the control point 30. At this point, applying $U^1_H[x = \text{null}](H_{>1}^b)$, first changes the node labeling by moving $x$ to $\delta$ which produces a heap with two simple nodes. Again, all simple nodes are folded into their first predecessor which is a crucial node. The set of simple nodes is \text{Simple}(H_{>1}^b) = \{n_2, n_3\} with \text{Simple}(H_{>1}^b, n_1) = \{n_3\} and \text{Simple}(H_{>1}^b, n_4) = \{n_2\}. Consequently, the node $n_3$ is folded into $n_1$ and the node $n_2$ is folded into $n_4$ ($n_1$ and $n_4$ are crucial nodes). According to rule 6.3.17 we obtain the heap $H$ in Figure 6.18 where $\nu(n_1) = \nu_1(n_1) \bullet \nu_1(n_3)$ and $\nu(n_4) = \nu_1(n_4) \bullet \nu_1(n_2)$ is obtained by applying concat(\{\{n_1 n_3, n_4 n_2\}\}, (\nu_1, \delta_1)).

Example 6.3.9. Consider the data word 0-SLL heap from Figure 6.19(a), denoting a memory configuration possible at the control point 30 of the procedure list_share from Figure 4.16. We suppose that $k = 0$. By applying the transformer $U^0_H[zi = \text{null}](H)$
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

![Diagrams](image)

(a) A data word 0-SLL heap \(H\)

(b) The heap \(H\) after removing the garbage caused by \(zi = null; z = null\)

(c) The heap \(H\) after folding the simple nodes introduced by \(zi = null; z = null\)

Figure 6.19: Applying the transformers corresponding to \(zi = null; z = null\).

only the node labeling changes, that is \(zi\) is moved to \(\sharp\). The node \(n_2\) is labeled only by \(z\) and the result remains a data word 0-SLL heap. Let us continue to apply the transformer \(U_k[H | p = q]\) on the heap previously obtained. Then, after moving \(z\) to \(\sharp\), the node \(n_3\) is not reachable from any node labeled by a program variable. Therefore, the transformer corresponding to the garbage collector eliminates it. Furthermore, the node \(n_2\) becomes a simple node. Therefore, the current heap has one simple node (it is given in Figure 6.19(b)). To transform it into a data word 0-SLL heap, we apply fold and we obtain the heap in Figure 6.19(c).

Assignments \(p = q\) For any heap \(H\), the transformer \(U_k[H | p = q](H)\) changes the labeling of the graph such that the node labeled by \(q\) is also labeled by \(p\). Because this assignment is preceded by \(p = null\), this transformation does not introduce garbage nor new simple nodes. Figure 6.20 gives the formal definition of this transformer.

\[
L(q) = n \\
U_k[H | p = q](H) = ((N, S, L \rightarrow n)), (\nu_i, \delta_i)_{i \in I})
\]  

(6.3.19)

Figure 6.20: The transformer \(U_k[H | p = q]\)

Assignments \(p->next = null\) and \(p->next = q\) First, notice that if \(L(p) = \sharp\) then \(U_k[H | p->next = null](H) = H_{err}\), because null pointer dereferencing has occurred.

Otherwise, we begin by applying a transformer \(F_k[p->next = null](H)\) which redefines the successor of \(L(p)\) to be \(\sharp\) (see rule 6.3.20 in Figure 6.21). Then, to define the result of \(U_k[H | p->next = null](H)\) we apply RemGrb on the output of \(F_k[p->next = null](H)\). Removing the garbage is not enough to ensure that the current heap \(H'\) is a data word \(k\)-SLL heap. This is due to the fact that \(H'\) may contain more than \(k\) simple nodes, all reachable from some node labeled by a program variable. If there is a sharing node \(n_s\) in \(H\) such that \(n_s\) has exactly two immediate predecessors, one of them being \(n = L(p)\),
6.3. A FAMILY OF ABSTRACT DOMAINS

\[
\begin{align*}
L(p) &= \sharp \\
U^k_H[p\rightarrow \text{next} = \text{null}](H) &= H_{\text{err}} \\
L(p) &\neq \sharp \\
U^k_H[p\rightarrow \text{next} = \text{null}](H) &= \text{fold}\left(\text{RemGrb}\left(F^k_H[p\rightarrow \text{next} = \text{null}](H)\right)\right)
\end{align*}
\]

Figure 6.21: The transformer \( U^k_H[p\rightarrow \text{next} = \text{null}] \)

then after removing the edge between \( n \) and \( n_s \) the latter becomes a simple node. Figure 6.22 gives an example for this situation (the sharing node with two predecessors is \( n_2 \) and the statement to be executed is \( z\rightarrow \text{next} = \text{null} \)). Therefore, after removing the garbage, we apply \( \text{fold} \) to eliminate all simple nodes if their number exceeded \( k \). The formal definition of \( U^k_H[p\rightarrow \text{next} = \text{null}] \) is given in Figure 6.21.

The definition of the transformer \( U^k_H[p\rightarrow \text{next} = q] \) is similar and is given in Figure 6.3.2. It uses this statement is precluded by \( p\rightarrow \text{next} \) points to \( \text{null} \).

Figure 6.22: Applying the transformer for the statement \( z\rightarrow \text{next} = \text{null} \).

\[
H = ((N,S,L),(\nu_i,\delta_i)_{i\in I}) \\
F^k_H[p\rightarrow \text{next} = q](H) = ((N,S[L(p) \mapsto \sharp],L),(\nu_i,\delta_i)_{i\in I})
\]

Figure 6.23: The transformer \( U^k_H[p\rightarrow \text{next} = q] \)

**Assert statement** We consider first only SL3 formulas that have only one disjunct. Therefore, let \( \varphi^{\text{SL3}} = \varphi^{\text{SLL}} \land \varphi^{p\varphi^{\text{SW}}} \) be a SL3 formula. Then,

\[
U^k_H[\text{assert } \varphi^{\text{SL3}}](H) = H \quad \text{iff} \quad \forall (G,\delta) \in (\gamma^{k\text{SLL}} \circ \gamma)(H), \quad G \models_\delta \varphi^{\text{SL3}}. \quad (6.3.21)
\]

That is the formula \( \varphi^{\text{SL3}} \) is modeled by any SLL heap graph (see Definition 6.1.1) in the concretization of \( H \). To avoid using the concretization function, we introduce an equivalent way of checking this entailment, using the rules 6.3.22, otherwise if this rule cannot be applied \( U^k_H[\text{assert } \varphi^{\text{SL3}}](H) = H_{\text{err}} \).

The rule 6.3.22 says that the formula \( \varphi^{\text{SL3}} \) holds in \( H \) iff there exists an evaluation \( \mu : x \cup P\text{Var} \to \mathbb{N} \), such that
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

- $\mu$ evaluates all predicates $ls(x, z)$ either in list segments from $H$ (in this case $p = 2$ and $\mu(x) = n_1$, $\mu(z) = n_2$), or in a path of length greater then two between $\nu(x) = n_1$ and $\mu(z) = n_p$, with $p > 2$;
- all the equality and labeling constraints of $\varphi^{SLL}$ are fulfilled by the interpretation $\mu$;
- for the data constraints if $ls(x, z)$ is evaluated into a path, so the formula $\varphi^{W}$ constrains the path starting in $x$; therefore, we have to check that for every $i \in I$, the word obtained by concatenating the words associated by $\nu_i$ with the nodes on this path (except for $\mu(z)$) satisfies the constraints imposed by $\varphi^{W}$. To this, we first apply $\text{concat}$ over $(\nu_i, \delta_i)$ and the sequences of nodes that are on a path, of length greater then 2, between $\nu(x)$ and $\mu(z)$ for any predicate $ls(x, z)$ of $\varphi^{SLL}$, and then we check that the resulting words are a model for $\varphi^{W}$. Notice that if $ls(x, z)$ is interpreted into a list segment there is nothing to concatenate.

\[ H = ((N, S, L), (\nu_i, \delta_i)_{i \in I}), \quad \varphi^{SL3} = \exists x. \varphi^{SLL} \land \varphi^{W} \]

there is $\mu : x \cup \text{PVar} \to N$ such that

1) if $ls(x, z)$ is a predicate of $\varphi^{SLL}$ then there is a path formed of $\{n_1, \ldots, n_p\} \in N$ with $\mu(x) = n_1, \mu(z) = n_p, S(n_i) = n_{i+1} 1 \leq i \leq p - 1$ and $p \geq 2$

2) if $p(x)$ is a predicate of $\varphi^{SLL}$ then $L(p) = \mu(x)$

3) if $x = z$ is a predicate of $\varphi^{SLL}$ then $\mu(x) = \mu(z)$

4) let $M = \bigcup_{i \in I} l_i(x, z)$ is a predicate in $\varphi^{SL3}$

\[ \nu(x) = n_1 \text{ and } \nu(z) = n_{p-1} \text{ and } p > 2 \]

and $(\nu', \delta') = \text{concat}((M, (\nu_i, \delta_i))$ for every $i \in I$ then

\[ (\nu', \delta') \models \varphi^{W}[x \mapsto \pi^x] \text{ for every } i \in I. \]

\[ U_H^k[\text{assert } \varphi^{SL3}](H) = H \]  

(6.3.22)

Let $\varphi^{SL3} = \bigvee_{j \in J} (\varphi_j^{SLL} \land \varphi_j^{W})$. Then, $U_H^k[\text{assert } \varphi^{SL3}](H) = H$ if there exists some $j \in J$ such that $U_H^k[\text{assert } (\varphi_j^{SLL} \land \varphi_j^{W})](H) = H$, otherwise $U_H^k[\text{assert } \varphi^{SL3}](H) = H_{err}$.

Finally, let $HS \in H\mathcal{S}$ be a k-Sll heap set. Then, $U_H^k[\text{assert } \varphi^{SL3}](HS) = HS$ iff for every $j \in J$ there exists $H \in HS$ such that $U_H^k[\text{assert } (\varphi_j^{SLL} \land \varphi_j^{W})](H) = H$.

Assume statement \quad We consider first only SL3 formulas that have only one disjunct. Therefore, let $\varphi^{SL3} = \varphi^{SLL} \land \varphi^P \land \varphi^W$ be a SL3 formula. Then,

\[ U_H^k[\text{assume } \varphi^{SL3}](H) = H' \text{ such that } \forall (G, \delta) \in (\gamma^{k\text{-SLL}} \circ \gamma)(H'), \ G \models \delta \varphi^{SL3}. \]

The transformer defines a heap $H'$ whose concretization contains all the SLL heaps that are models for $\varphi^{SL3}$.

For simplicity we consider that the node variables from $\varphi^{SL3}$ with different names denoted different nodes. Otherwise, we could consider all the possible equality relations.
over the node variables, and for each to build a heap \((H')\). The result would be the union of all heaps with non-isomorphic underlying graphs.

Let \(\varphi^{SLL} = \exists x. \varphi^{SLL} \wedge \varphi^p \wedge \varphi^W\). We define \(H' = (G', (\nu'_i, \delta'_i), i \in I)\) first by building \(G' = (N', S', L')\) from \(\varphi^{SLL} \wedge \varphi^p\) using the rule given in 6.3.23.

Roughly, for every node variable used in \(\varphi^{SLL}\) we add a node to the graph. Then, for every predicate of the form \(\text{ls}(n, m)\) where \(n\) and \(m\) are node variables we add an edges. Finally, for every literal \(p(n)\) in \(\varphi^p\) we label the node corresponding to \(n\) with the pointer variable \(p\).

\[
N' = \{n_1, \ldots, n_t\} \quad \text{and} \quad \mu : x \cup PVar \to N' \quad \text{is a bijection such that}
\]

\[
\begin{align*}
\text{if } \text{ls}(x, z) & \quad \text{is a predicate of } \varphi^{SLL} \text{ then } \\
S'(\mu(x)) & = z \\
\text{if } p(x) & \quad \text{is a predicate of } \varphi^{SLL} \text{ then } L'(p) = \mu(x) \\
\text{and } (\nu'_i, \delta'_i)_{i \in I} & = [\varphi^W[x \mapsto \mu x]] \quad (6.3.23)
\end{align*}
\]

If a node should have two successors according to this definition then the corresponding resulting heap is \(H_{err}\).

Let \(\varphi^{SLL}_k = \bigvee_{j \in J} (\varphi^{SLL}_j \wedge \varphi^W_j)\). Then, \(U^k_H[\text{assume } \varphi^{SLL}_k](H) = \bigcup_{j \in J} U^k_H[\text{assume } (\varphi^{SLL}_j \wedge \varphi^W_j)](H)\). Finally, let \(HS \in H\mathcal{S}\) be a k-SLL heap set. Then, \(U^k_H[\text{assume } \varphi^{SLL}_k](HS) = \bigcup_{H \in HS} U^k_H[\text{assume } \varphi^{SLL}_k](H)\). Actually, it would be enough to consider \(U^k_H[\text{assume } \varphi^{SLL}_k](HS) = U^k_H[\text{assume } \varphi^{SLL}_k](H)\) for some \(H \in HS\) because the definition of \(\text{assume}\) does not depend on the current state. Notice that \(\text{assume}\) statements may specify simple nodes through existential quantified node variables.

**Procedure calls** We define hereafter the transformer \(U^k_H[\text{call } P(ai, ao)]\) over heaps. Recall that the definition of the concrete transformer \(U[\text{call } P(ai, ao)]\) over heap graphs/SLL heaps is based on the fact that we consider cut-point free programs (see Section 4.3.3.4 for more details). To determine the existence of cut-points and to compute the local graph for a procedure call, all that is needed is to know precisely the reachability relation between vertices labeled by program variables. This relation is precisely preserved in the data word k-SLL heaps. It corresponds to the reachability relation between nodes labeled by program variables. The vertices that have been removed from an SLL heap in order to build the corresponding data word k-SLL heap (which are transformed into positions in some words attached to the nodes of the data word k-SLL heap) are of no importance in the cut-point detection and in the construction of the local graph. This is implied by the fact that we consider only garbage free SLL heaps and any simple node is reachable from a node labeled by some program variable.

**Verifying cut-point freedom:** We recall that we assume that all output parameters are set to \texttt{null} before the call. We define the predicate \(\text{cpf}(H, ai)\) that equals \texttt{true} if the invocation of \(P(ai, ao)\) in \(H\) is cut-point free, and equals \texttt{false} otherwise.

Let \(N_t\) denote the set of nodes from \(N\) that are either labeled by a pointer variable in \(ai\) or reachable from a node labeled by a pointer variable in \(ai\). Then, the set of cut-points for the procedure call \(P(ai, ao)\), denoted by \(N_{cp}\), is the set of nodes \(n_{cp}\) such that (1) \(n_{cp} \in N_t\), \(n \notin ai\), and (2) there is a path from a node \(n \notin N_t\) which does not contain
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

nodes labeled by pointer variables in \( \text{ai} \) (the set \( N_{cp} \) is computed using a traversal of the graph).

6.3.2 Remark. Let \( \gamma' \) be the concretization function from \( \mathcal{H}(\Sigma^*, k) \) to \( \mathcal{C}(\Sigma_{p_0}) \) and \( P(\text{ai}, \text{ao}) \) a procedure call. Any cut-point of a data word \( k \cdot \text{SLL} \) heap \( H \) corresponds to a cut-point in each \( \text{SLL} \) heap from \( \gamma'(H) \). This follows from the fact that the concretization \( \gamma' \) builds \( \text{SLL} \) heaps by adding vertices and edges to \( H \).

Moreover, any cut-point of an \( \text{SLL} \) heap \( G = (V, S, L, D) \) correspond to a cut-point in any \( H \) such that \( G \in \gamma'(H) \). This happens because \( H \) will keep all the crucial vertices from \( G \) which include also the set of cut-points.

If \( N_{cp} = 0 \) then \( \text{cpf}(H, \text{ai}) = \text{true} \), otherwise, \( \text{cpf}(H, \text{ai}) = \text{false} \).

The transformer \( U_H^k[\text{call } P(\text{ai}, \text{ao})] \): The transformer corresponding to a call to start edge \( \text{call } P(\text{ai}, \text{ao}) \), where \( P = (\text{fpi}, \text{fpo}, \text{loc}, G) \), computes heap sets over the set of program variables \( \text{loc}^1 \cup \text{loc} \). It has the same effect as the transformer corresponding to a call to start edge \( \text{call } q = P(\text{ai}, \text{ao}) \).

Let \( H = H^c = ((N^c, S^c, L^c), (\nu^c_i, \delta^c_i)_{i \in I}) \) be a heap representing the context of the call. First, we define the local heap \( \text{local}(H^c, \text{ai}) \) which contains only nodes reachable from actual parameters. Let \( N^\ell \) denote the set of nodes from \( N^c \) reachable from a node labeled by a variable in \( \text{ai} \) (including nodes labeled by some variable in \( \text{ai} \)). We define

\[
\text{local}(H^c, \text{ai}) = \text{proj}(D\text{Var} \setminus \text{ai}, N^c \setminus N^\ell, H^c) = H^\ell = ((N^\ell, S^\ell, L^\ell), (\nu^\ell_i, \delta^\ell_i)_{i \in I})
\]

where \( \text{proj}(D\text{Var} \setminus \text{ai}, N^c \setminus N^\ell, H^c) \) is an extension of the transformer \( \text{proj} \) used in \( \text{RemGrb} \) that, besides a set of nodes, removes the set of program data variables received as argument from the domain of each valuation \( \delta_i \). The set \( (\nu^\ell_i, \delta^\ell_i)_{i \in I} \) is defined by \( (\nu^\ell_i, \delta^\ell_i)_{i \in I} = \text{proj}(D\text{Var} \setminus \text{ai}, N^c \setminus N^\ell, (\nu^c_i, \delta^c_i)_{i \in I}) \), where \( \text{proj} \) is the transformer on the concrete data words domain that removes the data variables in \( D\text{Var} \setminus \text{ai} \) and the word variables in \( N^c \setminus N^\ell \) from the definition of each pair of functions in \( (\nu^c_i, \delta^c_i)_{i \in I} \).

Finally,

\[
U_H^k[\text{call } P(\text{ai}, \text{ao})](H) = H^c = ((N^c, S^c, L^c), (\nu^c_i, \delta^c_i)_{i \in I})
\]

(6.3.24)

where

- \( N^c \) contains two copies of the nodes in \( N^\ell \), i.e. \( N^c = N^\ell \cup N^{\ell,0} \) with \( N^{\ell,0} = \{n^0 \mid n \in N^\ell\} \);

- \( S^c(n_1) = n_2 \) if (1) \( n_1, n_2 \) are two nodes from \( N^\ell \) and there is an edge between them in \( H^\ell \), i.e., \( S^\ell(n_1) = n_2 \), or (2) \( n_1, n_2 \) are two nodes from \( N^{\ell,0} \), that is, \( n_1 \) is the copy of a node \( n \) and \( n_2 \) is the copy of a node \( n' \), and \( S^\ell(n) = n' \);

- for any \( p \in \text{fpi} \cap \text{PVar} \), \( L^\ell(p) = L^\ell(q) \), where \( q \) is the actual parameter corresponding to the formal parameter \( p \), and, for any \( p^0 \in \text{fpi} \cap \text{PVar} \), \( L^\ell(p^0) = n^0 \), where \( n^0 \) is the copy of \( n \);

- \( (\nu^c_i, \delta^c_i)_{i \in I} = \text{Eq}(M_{N^\ell}, M_{N^{\ell,0}}, D, D^0, \text{addDims}(N^{\ell,0} \cup D^0, (\nu^\ell_i, \delta^\ell_i)_{i \in I})) \), where \( M_{N^\ell} = n_1 \ldots n_s \) is a vector containing all the nodes in \( N^\ell \), \( M_{N^{\ell,0}} = n_0^1 \ldots n_0^n \), \( D = d_1 \ldots d_s \) is a vector containing all the data variables in \( \text{fpi} \), \( D^0 = d^0_1 \ldots d^0_s \), and \( \delta^\ell_i \) is obtained from \( \delta^c_i \) by renaming actual parameters into formal parameters. The transformer \( \text{addDims} \) over \( C_W \) takes as input a set of data and word variables and associates a random data or a random word with each of them.

176
6.3. A FAMILY OF ABSTRACT DOMAINS

The transformer $\mathsf{Eq}$ over elements of $\mathcal{C}_W$ corresponds to a set of assignments of the form $n^0 = n$, where $n^0$ and $n$ are word variables or $d^0 = d$, where $d \in W\text{Var}$ is a program data variable. It receives as input (1) two vectors of word variables $M$ and $M^0$ and it copies the content of $\nu_i(M[j])$ to $\nu_i(M^0[j])$, for any $j$ and $i \in I$, and (2) two vectors of program data variables $D$ and $D^0$ and it copies the content of $\delta_i(D[j])$ into $\delta_i(D^0[j])$, for any $j$ and $i \in I$. The formal definition of $\mathsf{Eq}$ is given in Figure 6.24.

$$
M = n_1 \ldots n_s, \quad M^0 = n^0_1 \ldots n^0_s, \\
D = d_1 \ldots d_t, \quad D^0 = d^0_1 \ldots d^0_t, \\
\nu'(n^0_j) = \nu(n_j), \text{ for any } 1 \leq j \leq s, \text{ and } \nu'(n') = \nu'(n'), \text{ for any } n' \notin M^0, \\
\delta'(d^0_j) = \delta(d_j), \text{ for any } 1 \leq j \leq t, \text{ and } \delta'(d') = \delta'(d'), \text{ for any } d' \notin D^0
$$

$$
\mathsf{Eq}(M, M^0, D, D^0, (\nu, \delta)) = (\nu^0, \delta^0)
$$

Figure 6.24: The transformer $\mathsf{Eq}$ over $\mathcal{C}_W$

(a) The data word 0-SLL heap $H^c$ (b) The local heap for the representing the context of the call. call of $\text{quicksort(left)}$ on $H^c$. (c) The heap $U^c_H[\text{call left} = \text{quicksort(left)}](H^c)$

Example 6.3.10. Let us consider the procedure $\text{quicksort}$ given in Figure 4.27. Suppose that the data word 0-SLL heap $H^c$, given in Figure 6.25(a), represents the pair of memory configurations reached by the program at the control point 40. At this point, in the context defined by $H^c$, a recursive call to $\text{quicksort}$ is done on the list pointed to by $\text{left}$. This call is cut-point free and the corresponding local heap is given in Figure 6.25(b). Then, the output of the transformer $U^c_H[\text{call left} = \text{quicksort(left)}](H^c)$ is given in Figure 6.25(c). The word associated with the node $n^0_0$ is obtained using the transformer $\mathsf{Eq}$, which is imposes the equality between two word variables.
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

**Procedure returns** Consider now an exit to return edge, labeled by return \( q = P(ai, ao) \), or return \( P(ai, ao) \), from the exit point of \( P = (\text{fpi, fpo, loc, G}) \), denoted \( e_P \), to some control point \( r \) in the CFG of a procedure \( Q \) that made the call. Let \( c \) be the call point associated to \( r \) and \( H^c = (N^c, S^c, L^c, (\nu^c, \delta^c)_{i \in I^c}) \) a heap associated with \( c \) and \( H^x = (N^x, S^x, L^x, (\nu^x, \delta^x)_{i \in I^x}) \) a heap associated with \( e_P \). We recall that the nodes of \( H^x \) and \( H^c \) are labeled with variables local to \( P \), and respectively \( Q \), and their superscript zero versions. Also, \( H^x \) and \( H^c \) contain valuations for the data variables local to \( P \), and respectively \( Q \), and their superscript zero versions.

Like for the transformer \( \overline{U}[\text{return } P(ai, ao)] \) over concrete SLL heaps, we have to check that the input configurations for the procedure \( P \) from the context of the call \( H^c \) match the input configurations from the procedure summary \( H^x \). The input configurations from the procedure summary \( H^x \) are represented by a heap \( H^{x,0} = (N^{x,0}, S^{x,0}, L^{x,0}, (\nu^x, \delta^x)_{i \in I^x}) \) where \( N^{x,0} \) is the set of nodes in \( N^x \) reachable from nodes labeled by variables in \( \text{loc}^0 \cap P\text{Var}^0 \), \( S^{x,0}, L^{x,0}, \nu^x, \delta^x \) are the restrictions of \( S^x, L^x, \) and \( \nu^x \) to \( N^{x,0} \), and \( \delta^x \) is the restriction of \( \delta^x \) to \( \text{loc}^0 \cap D\text{Var}^0 \).

We say that the the heap \( H^{x,0} \) matches the local heap corresponding to \( \text{call } P(ai, ao) \) in \( H^c \), local(\( H^c, ai \)), if the following hold:

- local(\( H^c, ai \)) is isomorphic to the heap \( \overline{H}^{x,0} \) obtained from \( H^{x,0} \) by renaming formal parameters into actual parameters and by removing all variables local to the procedure \( P \), i.e. \( \text{local}(H^c, ai) \sim \overline{H}^{x,0} \), and

- the set of pairs of valuations from local(\( H^c, ai \)) is included, modulo the isomorphism, into the set of pairs of valuations from \( H^{x,0} \). Formally, given \( h \) an isomorphism between local(\( H^c, ai \)) = \( (N^c, S^c, L^c, (\nu^c, \delta^c)_{i \in I^c}) \) and \( H^{x,0} = (N^{x,0}, S^{x,0}, L^{x,0}, (\nu^x, \delta^x)_{i \in I^x}) \), we have that: for any pair of valuations \( (\nu^c, \delta^c) \) with \( i \in I^c \) there exists a pair of valuations \( (\overline{\nu}^c, \overline{\delta}^x) \) with \( j \in I^x \) such that \( \overline{\nu}^c(h(n)) = \overline{\nu}^x(d) \), for any \( n \in N^c \), and \( \overline{\delta}^c(d) = \overline{\delta}^x(d) \), for any \( d \in ai \cap D\text{Var} \).

If the heap \( H^{x,0} \) does not match local(\( H^c, ai \)) then \( \overline{U}^k[\text{return } P(ai, ao)](H^c, H^x) = \emptyset \).

Otherwise, the transformer \( \overline{U}^k[\text{return } P(ai, ao)](H^c, H^x) \) returns a heap \( H^r = (N^r, S^r, L^r, (\nu^r, \delta^r)_{i \in I^r}) \) which is associated with the returning control point \( r \) in the CFG of \( Q \), computed as follows:

1. let \( H^{\text{out}} = (N^{\text{out}}, S^{\text{out}}, L^{\text{out}}, (\nu^{\text{out}}, \delta^{\text{out}})_{i \in I^r}) \) be the heap obtained from \( H^x \) by:
   - assigning every variable in \( P\text{Var} \) which is not a parameter or the returned variable to \( \text{null} \) (using the transformer \( U^k_N[p = \text{null}] \) which includes a call to \( \text{RemGrb} \) for removing the garbage). At this moment, we do not modify the variables from \( P\text{Var}^0 \) superscripted with 0.
   - removing from the domain of the set of functions \( (\nu, \delta) \) all the data variables that are not input or output parameters (using the transformer \( \text{proj} \) over \( C_W \)).

2. let \( H^{\text{aux}} = ((N^{\text{aux}}, S^{\text{aux}}, L^{\text{aux}}, (\nu^{\text{aux}}, \delta^{\text{aux}})_{i \in I^r}) \) be a heap defined as the union of \( H^c \) and \( H^{\text{out}} \) (if the procedure call is recursive, i.e., \( P = Q \), then the program variables in \( H^{\text{out}} \) are renamed so their values can be distinguished from the values of the same variables in \( H^c \)).
6.3. A FAMILY OF ABSTRACT DOMAINS

- \( N_{\text{aux}} = N_{\text{out}} \cup (N^c \setminus N^l) \), where \( N^l \) are of the nodes of the local heap \( \text{local}(H^c, ai) \).
- \( S_{\text{aux}}(n) = n' \) iff (1) \( n, n' \in N^c \setminus N^l \) and \( S^c(n) = n' \), (2) \( n, n' \in N_{\text{out}} \) and \( S_{\text{out}}(n) = n' \), or (3) \( n \in N^c \setminus N^l, n' \in N_{\text{out}}, S^c(n) = n'' \), for some node \( n'' \) labeled by some actual pointer parameter \( api \), and \( n' \) is labeled by the formal parameter corresponding to \( api \).
- \( L^c_{\text{aux}}(p) = n \) iff (1) \( p \) is a variable local to \( Q \) but not an actual parameter from \( ai \cup ao \) and \( L^c(p) = n \), (2) \( p \) is an actual parameter corresponding to the formal parameter \( q \) and \( L_{\text{out}}(q^0) = n \), or (3) \( p \) is a formal parameter or the returned variable of \( P \) and \( L_{\text{out}}(p) = n \).
- the set of pairs of valuations \( (\nu^c_i, \delta^c_i)_{i \in I} \) is defined by:

\[
(\nu^c_i, \delta^c_i)_{i \in I} = \text{combine}(h_n, h_d, (\nu^c_i, \delta^c_i)_{i \in I^c}, (\nu^c_i, \delta^c_i)_{i \in I^l}),
\]

where \( h_n \) is the isomorphism between the local heap \( \text{local}(H^c, ai) \) and \( H^x_{\text{aux}} \) and \( h_d : \text{api} \cap DVar \to \text{fpi} \cap DVar \) is a bijection that relates actual parameters to formal parameters. The transformer \( \text{combine} \) on elements of \( C_W \) receives as input a bijection \( h_n \) between word variables, a bijection \( h_d \) between data variables, and two elements of \( C_W \) as above, and returns

\[
\{ (\nu, \delta) \mid \text{there exists } (\nu^c_i, \delta^c_i) \text{ with } i \in I^c \text{ and } (\nu^c_i, \delta^c_i) \text{ with } j \in I^l \text{ s.t.} \\
\quad \text{for any } n \in \text{domain of } h, \nu^c_i(n) = \nu^c_j(h(n)), \\
\quad \text{for any } d \in \text{domain of } h_d, \delta^c_i(d) = \delta^c_j(h(d)^0), \text{ and} \\
\quad (\nu, \delta) = \text{proj}(\text{dom}(h_d), \text{dom}(h_n), (\nu^c_i \cup \nu^c_j, \delta^c_i \cup \delta^c_j)) \}
\]

(6.3.25)

Intuitively, for any two pairs of valuations \( (\nu^c_i, \delta^c_i) \) and \( (\nu^c_j, \delta^c_j) \), that agree on the values of the variables related by the bijections \( h_n \) and \( h_d \), \( \text{combine} \) takes their union and then removes the variables from the domain of \( h_d \) and \( h_n \).

3. all actual pointer parameters are assigned to the corresponding formal pointer parameters, using \( U^c_H[ai = fi] \) and \( U^c_H[ao = fo] \), for any \( ai \in ai \) and \( ao \in ao \) (\( fi \) and \( fo \) represent values from the output configuration of the summary \( H^x \)). If we are in the case of an exit to return edge labeled by \( \text{return } q = P(ai, ao) \) then the variable \( q \) is assigned to the corresponding returned value of the procedure \( P \).

4. finally, (1) we remove the node labels \( q^0 \), where \( q \) is a formal parameter of \( P \) (which represent the values of the formal parameters from the input configuration of the summary \( H^x \)), and the data variables \( d^0 \), where \( d \) is a formal parameter of \( P \), and (2) we apply \( \text{RemGrb} \) and then \( \text{fold} \) on the heap obtained in the previous step.

Example 6.3.11. Consider again the procedure \texttt{quicksort} given in Figure 4.21. Let \( H^c \) be the heap in Figure 6.26(a) which represents the context of the call \( \text{left} = \text{quicksort}(\text{left}) \) (at control point 40). Also, let \( H^x \) be the heap in Figure 6.26(b) which represents a procedure summary for the procedure \texttt{quicksort} (i.e., a heap reachable...


CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

\[
\nu(n_0^a) = [6 \ 4 \ 9 \ 2 \ 7] \\
\nu(n_i) = [4 \ 2] \\
\nu(n) = [9 \ 7] \\
\nu(p) = [6]
\]

(a) The heap \(H^c\) representing the context of the call.

\[
\nu'(n_0^a) = [6 \ 4 \ 9 \ 2 \ 7] \\
\nu'(n_i) = [4 \ 2] \\
\nu'(n) = [9 \ 7] \\
\nu'(p) = [6]
\]

(b) The heap \(H^x\) representing the procedure summary.

\[
\nu(\text{left}) = [6 \ 4 \ 9 \ 2 \ 7] \\
\nu(m_a) = [4 \ 2] \\
\nu(m) = [2 \ 4] \\
\nu(\text{res}) = [4 \ 2]
\]

(c) The heap \(H^{aux}\).

\[
\nu(\text{left}) = [6 \ 4 \ 9 \ 2 \ 7] \\
\nu(m_a) = [4 \ 2] \\
\nu(m) = [2 \ 4] \\
\nu(\text{res}) = [4 \ 2]
\]

(d) The heap \(U^k\{\text{return left = quicksort(left)}\}(H^c, H^x)\).

Figure 6.26: Applying the transformer corresponding to the procedure return \(\text{return left = quicksort(left)}\) from the procedure \(\text{quicksort}\) in Figure 4.21

at the exit point of the procedure \(\text{quicksort}\). For simplicity, we have ignored the value of the data variable \(d\).

The heap \(H^{aux}\) is given in Figure 6.26(c). It is obtained as the union between the heap \(H^c\) without the node pointed to by the actual parameter \(\text{left}\) and \(\text{null}\) (which form the local heap for this call) and the heap \(H^x\) in which pointer variables are renamed in order to be distinguished from the variables in the context heap \(H^c\). The valuation for the word variables \(\nu'\) is the result of

\[
\text{combine}\{h_d, h_n, (\nu, \delta), \{(\nu_1, \delta_1), (\nu_2, \delta_2)\}\},
\]

where (1) \(h_d\) is the empty function (there are no parameters which are data variables in DVar) and (2) \(h_n : \{n, \text{null}\} \rightarrow \{m_0, m_0\}\) with \(h_n(n_i) = m_0^0\) and \(h_n(\text{null}) = \nu\) is the isomorphism between the local heap for this call and the input configuration in \(H^x\).

The heap \(U^k\{\text{return left = quicksort(left)}\}(H^c, H^x)\) is given in Figure 6.26(d).

It is obtained from \(H^{aux}\) by assigning the actual parameter \(\text{left}\) to the returned variable \(\text{res}\) and by removing the garbage nodes \(m_0^0\) and \(m_a\).

6.3.3 Concrete data words domains

The heap sets from \(\mathcal{H}S(\Sigma, k)\) are sets of graphs to which we add an element from the concrete data word domain \(C_W(\Sigma, N_{\Sigma,k})\) that associates words over \(\Sigma\) to the nodes of
6.3. A FAMILY OF ABSTRACT DOMAINS

the graph and interprets the program data variables. Recall that a concrete data word domain over a type system \( \Sigma \) and a set of data word variables \( \mathcal{N} \) is the lattice

\[
\mathcal{C}_W(\Sigma, \mathcal{N}) = \left( \mathcal{P}(\mathcal{N} \rightarrow \mathbb{D}^+) \times [DVar \rightarrow \mathbb{D}] \right), \subseteq, \cup, \cap, \emptyset, [\mathcal{V} \rightarrow \mathbb{D}^+] \times [DVar \rightarrow \mathbb{D}],
\]

where \([\mathcal{N} \rightarrow \mathbb{D}^+]\) denotes the set of all partial functions \( \nu : \mathcal{N} \rightarrow \mathbb{D}^+ \) which are undefined only in \( \sharp \) and \( \sharp' \) and \([DVar \rightarrow \mathbb{D}]\) denotes the set of all functions \( \delta : DVar \rightarrow \mathbb{D} \).

As we have seen in the previous section, the program semantics over heap sets is defined using the following set of transformers on the concrete data words domain \( (n, n' \in \mathcal{N}, \mathbf{d} \in DVar, W = (\nu_i, \delta_i)_{i \in I} \) and \( W' = (\nu'_i, \delta'_i)_{i \in I'} \):

- \( \text{sglt}(n, W) \) extends the function \( \nu \) from each pair \( (\nu, \delta) \) of \( W \) with \( \nu(n) = w \), where \( w \) is a random word of length one; the formal definition is given by the rule 6.3.5 in Figure 6.3.2.

- \( \text{updDvar}(\mathbf{d}, n, W) \) changes the \( \delta \) component of each pair \( (\nu, \delta) \) in \( W \) s.t. \( \delta(\mathbf{d}) \) takes the value of \( \text{hd}(\nu(n)) \); the formal definition is given in Figure 6.7.

- \( \text{updFirst}(\mathbf{d}, n, W) \) changes the \( \nu \) component of each pair \( (\nu, \delta) \) in \( W \) s.t. \( \text{hd}(\nu(n)) \) takes the value of \( \delta(\mathbf{d}) \); the formal definition is given in Figure 6.8.

- \( \text{selectSglt}(n, W) \) (resp. \( \text{selectNonSglt}(n, W) \)) selects from \( W \) the pairs \( (\nu, \delta) \) for which the word \( \nu(n) \) has one element (resp. at least two elements); the formal definition is given in Figure 6.11.

- \( \text{split}(n, n', W) \) removes all the pairs \( (\nu, \delta) \) in \( W \) for which the length of \( \nu(n) \) is 1 and modifies the other pairs by extending the definition of \( \nu \) to \( \nu(n') = \text{tl}(w) \) and by setting \( \nu(n) \) to \( \text{hd}(w) \); the formal definition is given in Figure 6.12.

- \( \text{proj}(D, N, W) \) removes the variables in \( N \) from the domain of \( \nu \) and the variables in \( D \) from the domain of \( \delta \), for each \( (\nu, \delta) \) in \( W \).

- \( \text{concat}(M, W) \), where \( M = \{(n_1, \mathbf{v}_1), \ldots, (n_p, \mathbf{v}_p)\} \) is a set of pairs between word variables and vectors of word variables, changes the \( \nu \) component of each pair \( (\nu, \delta) \) of \( W \) by (1) assigning to \( n_i \) the concatenation of the words represented by the variables in \( \mathbf{v}_i \), \( \nu(\mathbf{v}_i[0]) \bullet \cdots \bullet \nu(\mathbf{v}_i[\text{len}(\mathbf{v}_i) - 1]) \). Its definition is given in Figure 6.16.

- \( \text{addDims}(X, W) \), where \( X \subseteq DVar \cup \mathcal{N} \) is a set of variables, extends the domain of the functions \( \nu \) in \( W \) by associating random values from \( \mathbb{D}^+ \) with \( \nu(n) \), for every \( n \in X \cap \mathcal{N} \), and it extends the domain of the functions \( \delta \) in \( W \) by associating random values from \( \mathbb{D} \) with \( \delta(\mathbf{d}) \), for every \( \mathbf{d} \in X \cap DVar \).

- \( \text{Eq}(V, V', D, D', W) \), where \( V = n_1 \ldots n_s \) and \( V' = n'_1 \ldots n'_t \) are vectors of data word variables of equal length and \( D = d_1 \ldots d_t \) and \( D' = d'_1 \ldots d'_t \) are vectors of data variables of equal length, (1) modifies the \( \nu \) component of each pair \( (\nu, \delta) \) of \( W \) by setting \( \nu(n'_i) = \nu(n_i) \), for all \( 1 \leq i \leq s \), and (2) modifies the \( \delta \) component of each pair \( (\nu, \delta) \) of \( W \) by setting \( \delta(d'_j) = \delta(d_j) \), for all \( 1 \leq j \leq t \); its formal definition is given in Figure 6.24.
Chapter 6. Analysis of Programs Manipulating Singly-Linked Lists

- **combine**($h_n, h_d, W, W'$), where $h_n$ is a bijection between word variables and $h_d$ is a bijection between data variables, contains all the pairs of valuations obtained by (1) taking the union of two pairs $(\nu, \delta) \in W$ and $(\nu', \delta') \in W'$ that agree on the values of the variables related by the bijections $h_n$ and $h_d$ and (2) removing the variables from the domain of $h_d$ and $h_n$; its definition is given in 6.3.25.

Let $\mathcal{F}_C$ denote the set of transformers described above.

### 6.3.4 Abstraction schema for heap sets

Abstractions of data word $\kappa$-SLL heaps can be defined by representing the values of the integer program variables and the words over $\mathbb{D}$ using formulas in some first-order logic. The logics we use capture various aspects such as constraints on the sizes, the multisets of letters, or the data at different positions of the words. For example, the graph and the multiset formula $\psi$ in Figure 6.27(a) represent an abstraction of the data word heap in Figure 6.27(b). The same holds for the graph and the multiset formula $\psi'$ in Figure 6.27(b).

Such a pair between a graph and a formula is called abstract heap. In the formula $\psi$, $n_a, n_p, n_1, n_2,$ and $n_p$ are variables interpreted as words over $\mathbb{N}$, $\text{hd}(n_1)$ denotes the first letter of the word denoted by $n_1$, $\text{len}(n_2)$ denotes the length of the word denoted by $n_2$, $y$ is a variable interpreted as a position in some word (as usual, the positions of a word $a_0, a_1, \ldots, a_n$ are the integers from 0 to $n$), $y \in \text{tl}(n_1)$ means that $y$ belongs to the tail of $n_1$ (i.e., the suffix starting with the second letter), and $n_1[y]$ is a term interpreted as the value at the position $y$ of $n_1$. The formula $eq_d(n_a, n_d^0)$ states that the words denoted by $n_a$ and $n_d^0$ are identical:

$$eq_d(n_a, n_d^0) ::= \forall y_1, y_2. [(y_1) \in \text{tl}(n_a) \land (y_2) \in \text{tl}(n_d^0) \land y_1 = y_2) \implies n_a[y_1] = n_d^0[y_2]$$

$$\land \text{hd}(n_a) = \text{hd}(n_d^0) \land \text{len}(n_a) = \text{len}(n_d^0).$$

We distinguish the first letter of a word from its tail because program assignments can update only this first letter (the statement $p\rightarrow\text{data}=\ldots$ updates the first letter of the word associated to the node labeled by $p$). In the multiset formula, $\text{ms}(n_a)$ denotes the multiset containing all the letters of $n_a$.

The formulas over data word variables and program data variables used in abstract heaps are elements of a *data words abstract domain* ($\mathbb{D}W$-domain, for short). The elements of a $\mathbb{D}W$-domain are abstractions of elements from the concrete data words domain $C_W$. Formally,

**Definition 6.3.6 ($\mathbb{D}W$-domain).** A data words abstract domain over a type system $\Sigma$ and $\mathcal{N}$ ($\mathbb{D}W$-domain, for short) is a lattice $\mathcal{A}_\mathbb{D}W = (\mathbb{A}_\mathbb{D}W, \sqsubseteq_\mathbb{D}W, \sqcap_\mathbb{D}W, \sqcup_\mathbb{D}W, \top_\mathbb{D}W, \bot_\mathbb{D}W)$ such that there exists a concretization function $\gamma_\mathbb{D}W : \mathbb{A}_\mathbb{D}W \rightarrow \mathcal{P}([\mathcal{N} \rightarrow \mathbb{D}^+] \times [\mathbb{D}Var \rightarrow \mathbb{D}])$ from $\mathcal{A}_\mathbb{D}W$ to $C_W(\Sigma, \mathcal{N})$.

Then, an abstract $\kappa$-SLL heap is obtained from a data word $\kappa$-SLL heap by replacing the set of pairs of valuations $(\nu_i, \delta_i)_{i \in I}$ with an abstract element from a $\mathbb{D}W$-domain.

**Definition 6.3.7 (Abstract $\kappa$-SLL heap).** An abstract $\kappa$-SLL heap over a type system $\Sigma$ and a $\mathbb{D}W$-domain $\mathcal{A}_\mathbb{D}W$ is a tuple $\tilde{H} = (N, S, L, \varphi)$ where $N, S, L$ are as in the definition of heaps, and $\varphi$ is an element of $\mathcal{A}_\mathbb{D}W$ over $\Sigma$ and the data word variables $N \setminus \{\}$ (we assume that for each node in $N$ there exists a data word variable with the same name).
6.3. A FAMILY OF ABSTRACT DOMAINS

\[ \nu(n_0^a) = [6 \ 4 \ 9 \ 2 \ 7] \]

\[ \nu(n_a) = [6 \ 4 \ 9 \ 2 \ 7] \]

\[ \nu(n_l) = [4 \ 2] \]

\[ \nu(n_r) = [9 \ 7] \]

\[ \nu(n_p) = [6] \]

(a) The heap \( H^c \).

Universally quantified formula:
\[ \psi_c : \text{hd}(n_l) \leq \text{hd}(n_p) \land \text{hd}(n_r) > \text{hd}(n_p) \land \text{len}(n_a) = \text{len}(n_l) + \text{len}(n_r) + \text{len}(n_p) \land \forall y. y \in \text{tl}(n_l) \implies n_l[y] \leq \text{hd}(n_p) \land \forall y. y \in \text{tl}(n_r) \implies n_r[y] > \text{hd}(n_p) \land d = \text{hd}(n_p) \land eq_v(n_a, n_0^a) \land \text{len}(n_p) = 1 \]

Multiset formula:
\[ \psi'_c : \text{ms}(n_a) = \text{ms}(n_l) \cup \text{ms}(n_r) \cup \text{ms}(n_p) \land \text{ms}(n_0^a) = \text{ms}(n_a) \]

(b) Abstract heaps representing \( H^c \).

Figure 6.27: Abstracting data word k-SLL heaps.

Two abstract heaps are isomorphic if their underlying graphs are isomorphic (denotes the isomorphism relation). Recall that a set of data word k-SLL heaps which does not contain two isomorphic heaps is called a data word k-SLL heap set. Similarly, we define the notion of abstract k-SLL heap set.

**Definition 6.3.8** (Abstract k-SLL heap set). An abstract k-SLL heap set over a type system \( \Sigma \) and a \( \mathbb{D}W \)-domain \( A_W \) is a finite set of non-isomorphic abstract k-SLL heaps over \( \Sigma \) and \( A_W \). Let \( A_{\text{HS}}(\Sigma, k, A_W) \) denote the set of all abstract k-SLL heap sets over \( \Sigma \) and \( A_W \).

The lattice of abstract k-SLL heap sets over \( \Sigma \) and \( A_W \), denoted by \( A_{\text{HS}}(\Sigma, k, A_W) \) is defined by
\[
A_{\text{HS}}(\Sigma, k, A_W) = \left( A_{\text{HS}}(\Sigma, k, A_W), \subseteq, \cup, \cap, \emptyset, \text{HS}(\Sigma, k) \right).
\]

The lattice operators are obtained from those in \( C_{\text{HS}}(\Sigma, k) = (\text{HS}(\Sigma, k), \subseteq, \cup, \cap, \emptyset, \text{HS}(\Sigma, k)) \) (the lattice of data word k-SLL heap sets) by replacing \( \subseteq \), \( \cup \), and \( \cap \) between sets of pairs of valuations (elements of the lattice \( C_W \)) with similar operators from the \( \mathbb{D}W \)-domain \( A_W \). More precisely,

- for any two abstract k-SLL heap sets \( \text{HS}, \overline{\text{HS}} \in A_{\text{HS}}(\Sigma, k, A_W) \), the order relation
\[ \leq \text{HS} \text{ is defined as follows:} \]

\[ \overrightarrow{HS} \leq \text{HS} \overrightarrow{HS}' \text{ iff for every } \overrightarrow{H} = (N, S, L, \varphi) \text{ in } \overrightarrow{HS} \text{ there exists } \overrightarrow{H}' = (N', S', L', \varphi') \text{ in } \overrightarrow{HS}' \text{ such that } \overrightarrow{H} \sim \overrightarrow{H}' \text{ and } \varphi \leq W \varphi'. \]

- for any \( \overrightarrow{HS}, \overrightarrow{HS}' \in A_{\text{HS}}(\Sigma, k, A_W) \), \( \overrightarrow{HS} \sqcup \overrightarrow{HS}' \) is defined by (1) taking the union of \( \overrightarrow{HS} \) and \( \overrightarrow{HS}' \) and (2) any two isomorphic abstract heaps \( \overrightarrow{H}' = (N, S, L, \varphi') \) and \( \overrightarrow{H}'' = (N, S, L, \varphi'') \) are replaced by one abstract heap \( \overrightarrow{H} = (N, S, L, \varphi) \) having the same graph s.t. \( \varphi \) is the join between \( \varphi' \) and \( \varphi'' \) in \( A_W \), i.e., \( \varphi = \varphi' \sqcap W \varphi'' \).

- for any \( \overrightarrow{HS}, \overrightarrow{HS}' \in A_{\text{HS}}(\Sigma, k, A_W) \), \( \overrightarrow{HS} \sqcap \overrightarrow{HS}' \) is the set of all abstract heaps \( \overrightarrow{H} = (N, S, L, \varphi) \) such that there exist two abstract heaps \( \overrightarrow{H}' = (N, S, L, \varphi') \in \overrightarrow{HS} \) and \( \overrightarrow{H}'' = (N, S, L, \varphi'') \in \overrightarrow{HS}' \) isomorphic to \( \overrightarrow{H} \) with \( \varphi = \varphi' \sqcap W \varphi'' \).

We define a widening operator for \( A_{\text{HS}}(\Sigma, k, A_W) \), denoted \( \overrightarrow{W} \), which is similar to the join operator \( \overrightarrow{\sqcup} \) except that it uses a widening operator for \( A_W \), \( \overrightarrow{W} \), instead of the join operator \( \overrightarrow{\sqcup} \). The operator \( \overrightarrow{W} \) is indeed a widening operator because abstract k-SLL heaps have a bounded number of nodes (linear in the number of program pointer variables and k) and because \( \overrightarrow{W} \) is a widening operator for \( A_W \).

The lattice \( A_{\text{HS}}(\Sigma, k, A_W) \) is connected to the lattice of data word k-SLL heap sets through a concretization function \( \gamma_1 : A_{\text{HS}}(\Sigma, k, A_W) \rightarrow \mathcal{HS}(\Sigma, k) \) defined by

\[ \gamma_1(\overrightarrow{HS}) = \{(N, S, L, (\nu_i, \delta_i)_{i \in I}) | (N, S, L, \varphi) \in \overrightarrow{HS} \text{ and } (\nu_i, \delta_i)_{i \in I} = \gamma_W(\varphi)\} \]

In Section 6.3.1.2 we have defined a concretization function from data word k-SLL heap sets to sets of SLL heaps, denoted \( \gamma' \). Consequently, the lattice \( A_{\text{HS}}(\Sigma, k, A_W) \) is connected to the lattice of sets of SLL heaps \( \mathcal{C}(\Sigma) \) through the concretization function \( \gamma_1 \circ \gamma' \).

### 6.3.5 Analyses over the domain of abstract heap sets

Let \( P_1 \) be a program over a type system \( \Sigma_{P_1} \) as in 6.1.1. In Section 4.2.3.3 and Section 4.3.3.4 we defined the transformers \( U[a] : \mathcal{P}(\mathcal{R}(\Sigma)) \rightarrow \mathcal{P}(\mathcal{R}(\Sigma)) \) and \( U[a] : \mathcal{P}(\mathcal{R}(\Sigma)) \times \mathcal{P}(\mathcal{R}(\Sigma)) \rightarrow \mathcal{P}(\mathcal{R}(\Sigma)) \) over pairs of memory configurations, where \( a \) is an edge label of the ICFG associated with \( P_1 \). In Section 6.3.2 we have defined abstract versions of these operators over data word k-SLL heap sets \( U_{\mathcal{HS}}[a] : \mathcal{HS}(\Sigma_{P_1}, k) \rightarrow \mathcal{HS}(\Sigma_{P_1}, k) \) and \( \overline{U}_{\mathcal{HS}}[a] : \mathcal{HS}(\Sigma_{P_1}, k) \times \mathcal{HS}(\Sigma_{P_1}, k) \rightarrow \mathcal{HS}(\Sigma_{P_1}, k) \) (\( U_{\mathcal{HS}}[a] \) is an abstract version of \( U[a] \) and \( \overline{U}_{\mathcal{HS}}[a] \) is an abstract version of \( \overline{U}[a] \)).

The lattice \( A_{\text{HS}}(\Sigma, k, A_W) \) of abstract k-SLL heap sets is connected through a concretization function to the lattice \( \mathcal{C}(\Sigma_{P_1}) \) of sets of program relations.

The abstract versions of the operators \( U[a] \) and \( \overline{U}[a] \) over abstract k-SLL heap sets are defined from \( U_{\mathcal{HS}}[a] \) and \( \overline{U}_{\mathcal{HS}}[a] \) by replacing concrete transformers over elements of \( \mathcal{C}_W(\Sigma_{P_1}, \mathcal{N}_{\Sigma_{P_1}}, k) \) with abstract transformers over \( A_W \). The abstract version of \( U[a] \), resp. \( \overline{U}[a] \), is denoted by \( U_{\mathcal{HS}}^\#[a] \), resp. \( \overline{U}_{\mathcal{HS}}^\#[a] \).

In the following, we introduce several \( \mathcal{D}W \)-domains and their corresponding abstract transformers.
6.4 A $\mathbb{D}W$-domain with universally quantified formulas

We define the $\mathbb{D}W$-domain $A_U = (A^U, \sqsubseteq^U, \sqcap^U, \sqcup^U, \top^U, \bot^U)$ whose elements are first-order formulas having as free variables a set of data word variables $\mathcal{V}$ and a set of program data variables $DVar$. We suppose that $\mathbb{D}$ is the set of integers and consequently, word variables are interpreted as non-empty words over $\mathbb{Z}$ and data variables are interpreted as integers.

6.4.1 Lattice definition

The formulas in $A^U$ contain a quantifier-free part and a conjunction of universally quantified formulas of the form $\forall \mathbf{y}. P(\mathbf{y}) \Rightarrow U$, where $\mathbf{y}$ is a vector of integer variables representing positions in the words, the guard $P(\mathbf{y})$ is a constraint over the values of $\mathbf{y}$, and $U$ is a constraint over the data values at the positions denoted by the variables $\mathbf{y}$. The syntax of the guards from the universally quantified formulas depends on a finite set of guard patterns $\mathcal{P}$, which is a parameter of $A_U$. The formulas in $A_U$ have the form:

$$E(\mathcal{V}) \land \bigwedge_{P \in \mathcal{P}(\mathcal{V})} \forall \mathbf{y}. P(\mathbf{y}, \mathcal{V}) \Rightarrow U_P(\mathbf{y}, \mathcal{V}),$$

where $\mathcal{V}$ is a set of data word variables and $\mathcal{P}(\mathcal{V})$ is the set of formulas obtained from patterns in $\mathcal{P}$ by substituting all the data word variables with variables from $\mathcal{V}$, and $E(\mathcal{V})$, $U_P(\mathbf{y}, \mathcal{V})$ are formulas in some numerical abstract domain $A_Z$. There is no quantification on the word variables, only variables representing positions in the words can be (universally) quantified.

To understand some of the choices we have done in the design of the lattice $A_U$, let us denoted by ArrLogic the fragment of gCSL restricted to arrays. Suppose that the implication between ArrLogic is decidable and that ArrLogic is closed under disjunction and conjunction. Then, we can define a lattice $A$ whose elements are formulas from ArrLogic such that: (1) the order relation is given by the logical implication, (2) the join (resp., meet) operator is the disjunction (resp., conjunction) between formulas, and (3) the top (resp., bottom) element is true (resp., false). However, the complexity of checking the entailment can be quite high, from NP-time (if we consider the Array Property fragment from [Bradley 2006]) to non-elementary (if we consider the UABE logic from [Zhou 2010]), and the design of an widening operator is not obvious. Furthermore, an analysis over this lattice using a join operator that introduces disjunctions is not scalable: the size of the formulas double at each iteration of the fixpoint computation and consequently, the size of the formulas handled by the analysis is exponential in the number of iteration steps. Finally, the fact that the implication is decidable may have consequences on the expressiveness of ArrLogic formulas.

The lattice $A_U$ addresses these concerns as follows: (1) formulas use disjunction in a limited manner, (2) it uses an order relation which is a sound approximation of the logical implication, (3) it defines a join operator which may introduce disjunctions only in the quantifier-free part and in the right part of the universal formulas, and (4) it defines a widening operator based on the widening operator from $A_Z$. Notice that, in general, the logical implication between formulas in $A_U$ is undecidable.
6.4.1.1 Syntax of guard patterns $P$

Let $\mathcal{O} \subseteq \mathcal{N}$ be a set of distinguished data word variables and $\omega_1, \ldots, \omega_n \in \mathcal{O}$. Let $y_1, \ldots, y_n$ be non-empty vectors of position variables interpreted as positions in the words denoted by $\omega_1, \ldots, \omega_n$ (these variables are universally quantified in the elements of $A_U$). We assume that these vectors are pairwise disjoint and that $\omega_i \neq \omega_j$, for any $i \neq j$. We denote by $y^j_i$ the $j$th element of the vector $y_i$, $1 \leq j \leq |y_i|$. Let $\Omega \subseteq \mathcal{O}$ be a set of variables not necessarily distinct from $\omega_1, \ldots, \omega_n$.

The guard patterns are conjunctions between (1) a formula that associates vectors of position variables with data word variables, (2) an arithmetical constraint on the values of some position variables and (3) order constraints between the position variables associated with the same data word variable.

Formally,

$$P(y_1, \ldots, y_q, \omega_1, \ldots, \omega_q, \Omega) ::= \bigwedge_{1 \leq i \leq q} y_i \in tl(\omega_i) \land P_L(y^1_1, \ldots, y^1_q, \Omega) \land \bigwedge_{1 \leq i \leq q} P_R(y^i)$$

$$P_R(y^1y^2 \ldots y^m) ::= y^1 \prec_1 y^2 \prec_2 \ldots \prec_m y^m$$

where

1. for each vector $y_i$, $y_i \in tl(\omega_i)$ states that the positions denoted by $y_i$ belong to the tail of word denoted by $\omega_i$,

2. for each vector $y \in \{y_1, \ldots, y_q\}$, the formula $P_R(y)$ is an order constraint over the variables in $y$, where $\prec_i \in \{\leq, <, <\}$ with $x < y$ iff $y = x + 1$,

3. $P_L$ is a boolean combination of linear constraints over the variables $y^i_1$ with $1 \leq i \leq q$ and the terms $\mathsf{len}(\omega)$ with $\omega \in \Omega$.

6.4.1.2 Numerical abstract domain $A_Z$

Let $\mathcal{V}ars$ be a set of variables and let $C_Z$ be the concrete lattice of functions from $\mathcal{V}ars \to \mathbb{Z}$, that is $C_Z = (P([\mathcal{V} \to \mathbb{D}]), \subseteq, \cup, \cap, \emptyset, [\mathcal{V} \to \mathbb{D}])$.

An abstract domain for $C_Z$ is denoted by $A_Z = (A^Z, \subseteq^Z, \cap^Z, \cup^Z, \top^Z, \bot^Z)$,

where $A^Z$ is a set of constraints over $\mathcal{V}ars$, $\subseteq^Z$ is an order relation over them, $\cup^Z, \cap^Z$ are the least upper bound, respectively greatest lower bound operators.

The concretization function $\gamma_Z$ associates to a set of constraints in $A_Z$ the set of valuations $\mathcal{V}ars \to \mathbb{Z}$ that satisfy the constraints.

An abstract transformer over $A_Z$ is an operator defined on $A_Z$ that returns an abstract element of $A_Z$. We assume in the following a set of basic abstract transformers on $A_Z$:

- $\mathsf{update}^#(C^k, v, \mathit{exp})$ is the transformer corresponding to an assignment of a variable $v \in \mathcal{V}ars$; $\mathit{exp}$ is an expression build over the variables in $\mathcal{V}ars$ and the constants in $\mathbb{Z}$ using the operators allowed by $A_Z$.
6.4. A $\mathbb{D}\mathbb{W}$-DOMAIN WITH UNIVERSALLY QUANTIFIED FORMULAS

- **add**$(\mathbb{C}, (v_1, \ldots, v_2))$ is the abstract transformer that add new unconstrained variables to $\mathbb{C}$;

- $\mathbb{C} \uparrow (v_1, \ldots, v_n)$ is the projection operator, that eliminates the variables $v_1, \ldots, v_n$ from the constraints composing $\mathbb{C}$.

6.4.1.3 Elements of $\mathcal{A}_U$

Let $\mathcal{V} \subseteq \mathcal{N}$ and let $\mathcal{P}$ be a set of guard patterns. We define $\mathcal{P}(\mathcal{V})$ to be the set of all formulas $P(y_1, \ldots, y_q, w_1, \ldots, w_q, \mathcal{W})$ obtained from some $P(y_1, \ldots, y_q, \omega_1, \ldots, \omega_q, \Omega) \in \mathcal{P}$ by substituting $\omega_i$ with $w_i \in \mathcal{V}$, for any $1 \leq i \leq q$, and $\Omega$ with $\mathcal{W} \subseteq \mathcal{V}$. We assume that $w_i \neq w_j$, for any $i \neq j$. Then, an element of $\mathcal{A}_U$ has the following syntax:

$$\varphi(\mathcal{V}) := E(\mathcal{V}) \land \bigwedge_{P(y_1, \ldots, y_q, w_1, \ldots, w_q, \mathcal{W}) \in \mathcal{P}(\mathcal{V})} \forall y. P(y_1, \ldots, y_q, w_1, \ldots, w_q, \mathcal{W}) \Rightarrow U_P(\mathcal{V}, y)$$  \hfill (6.4.1)

where

- $E(\mathcal{V})$ is a quantifier-free arithmetical formula over $DVar$ and terms $\text{hd}(w)$, $\text{len}(w)$ with $w \in \mathcal{V}$ ($\text{hd}(w)$ denotes the first element of the word denoted by $w$ and $\text{len}(w)$ denotes the length of the word denoted by $w$),

- $U_P$ is a quantifier-free arithmetical formula over the terms in $E(\mathcal{V})$ and the terms $w[y]$ and $y$ with $w \in \mathcal{V}$ and $y$ a position variable in $P$ ($w[y]$ denotes the data at the position $y$ in the word denoted by $w$).

$E$ and $U_P$ are elements of some numerical abstract domain $\mathcal{A}_\mathcal{Z} = (\mathbb{Z}^\mathbb{N}, \subseteq, \cap, \cup, \Rightarrow, \bot)$ which is a parameter of $\mathcal{A}_U$. Their syntax is given by the syntax of the numerical abstract domain. For instance, if $E$ and $U_P$ are elements of the Polyhedra domain [Cousot 1978] then, they are conjunctions of linear constraints. Each of the terms $\text{hd}(w)$, $\text{len}(w)$, $y$, or $w[y]$ correspond to a unique integer variable constraint by the abstract value from the Polyhedra domain.

Each variable in $w \in \mathcal{V}$ describes a word over $\mathbb{Z}$. The terms $\text{len}(w)$, $\text{hd}(w)$, $\text{tl}(w)$ represent the length, the first symbol, and respectively the tail (the last $\text{len}(w) - 1$ symbols), of the word represented by $w$. The quantified variable $y$ denotes an integer, and the predicate $y \in \text{tl}(w)$ evaluates to true only if $y$ represents an integer corresponding to a position defined in the word represented by $w$, i.e. $1 \leq y \leq \text{len}(w)$. Finally, $w[y]$ denotes the $(y + 1)$th symbol of the word represented by $w$.

**Examples:** The following formula is an element of $\mathcal{A}_U$ parametrized by $\mathcal{P} = \{[y_1] \in \text{tl}(\omega_1) \land [y_2] \in \text{tl}(\omega_2) \land y_1 = y_2\}$ and the Polyhedra domain. It expresses the fact that the word denoted by $w_1$ is a copy of the word denoted by $w_2$:

$$\text{len}(w_1) = \text{len}(w_2) \land \text{hd}(w_1) = \text{hd}(w_2)$$

$$\forall y_1, y_2. (\,[y_1] \in \text{tl}(w_1) \land [y_2] \in \text{tl}(w_2) \land y_1 = y_2 \Rightarrow w_1[y_1] = w_2[y_2]\)$$  \hfill (6.4.2)

The following element of $\mathcal{A}_U$ over $\mathcal{P} = \{[y_1, y_2, y_3] \in \text{tl}(\omega) \land y_1 <_1 y_2 <_1 y_3, [y] \in \text{tl}(\omega) \land y = 1, [y] \in \text{tl}(\omega) \land y = 2\}$ and the Polyhedra domain represents words $w$ whose
data are in the Fibonacci sequence:

\[ \text{hd}(w) = 1 \land \forall y. \left( [y] \in t1(w) \land y = 1 \right) \Rightarrow w[y] = 1 \land \forall y. \left( [y] \in t1(w) \land y = 2 \right) \Rightarrow w[y] = 2 \]

\[ \forall y_1, y_2, y_3. \left( [y_1, y_2, y_3] \in t1(w) \land y_1 < 1 \land y_2 < 1 \land y_3 \right) \Rightarrow w[y_3] = w[y_1] + w[y_2]. \]

### 6.4.1.4 Lattice operators

The lattice operators are approximations for the usual boolean connectors, that exploit the syntactic form of the considered formulas. The value \( \top^U \) (resp. \( \bot^U \)) corresponds to a formula which is equivalent true (resp. false) and it is defined by the formula in which \( E \) and all \( U_P \) are \( \top^Z \) (resp. \( \bot^Z \)). Let

\[
\varphi(V_1) = E(V_1) \land \bigwedge_{P(y,w,y) \in P(V_1)} \forall y. \left( P(y,w,y) \Rightarrow U_P(V_1,y) \right)
\]

\[
\varphi'(V_2) = E'(V_2) \land \bigwedge_{P(y,w,y) \in P(V_2)} \forall y. \left( P(y,w,y) \Rightarrow U'_P(V_2,y) \right)
\]

where \( y \) is a position of variable positions and \( w \) is a vector of data word variables.

If the two formulas do not use exactly the same set of guard patterns then, before applying any lattice operator, we add to \( \varphi \) (resp. \( \varphi' \)) universally quantified formulas \( \forall y. \left( P(y,w,y) \Rightarrow \top^Z \right) \), for any \( P(y,w,y) \in P(V_1) \cup P(V_2) \) (resp. \( P(y,w,y) \in P(V_2) \setminus P(V_1) \)).

Then,

\[
\varphi \sqsubseteq^U \varphi' \text{ iff } (1) \quad E \sqsubseteq^Z E', \text{ and (2) for each } P(y,w,y) \in P(V_1) \cup P(V_2), \left( E \sqsubseteq^Z U_P \right) \sqsubseteq^Z U'_P.
\]

Also, the union \( \varphi \cup^U \varphi' \) is defined by

\[
\left( E \cup^Z E' \right) \land \bigwedge_{P(y,w,y) \in P(V_1) \cup P(V_2)} \forall y. \left( P(y,w,y) \Rightarrow \left( U_P \cup^Z U'_P \right) \right).
\]

Finally, the operators \( \sqcap^U \) and \( \sqcup^U \) are defined in a similar way, by replacing \( \sqsubseteq^Z \) with \( \sqcap^Z \) and, respectively \( \sqcup^Z \).

**Proposition 6.4.1.** Let \( \mathbb{P} \) be a set of guard patterns and \( \mathbb{A}_Z \) a numerical abstract domain. Given two formulas \( \varphi_1 \) and \( \varphi_2 \) in \( \mathbb{A}_U(\mathbb{P}, \mathbb{A}_Z) \) if \( \varphi_1 \sqsubseteq^U \varphi_2 \) then \( \varphi_1 \implies \varphi_2 \), where \( \implies \) is the logical implication. The reverse does not hold.

**Proof.** The first claim follows from the fact that (1) for any two abstract values \( X, X' \in \mathbb{A}_Z \), if \( X \sqsubseteq^Z X' \) then \( X \implies X' \) and (2) for any \( \varphi_1, \varphi_2, \varphi'_1, \) and \( \varphi'_2 \in \mathbb{A}_U \), if \( \varphi_1 \implies \varphi'_1 \) and \( \varphi_2 \implies \varphi'_2 \) are valid then \( \left( \varphi_1 \land \varphi_2 \right) \implies \left( \varphi'_1 \land \varphi'_2 \right) \) is also valid. The reverse of the latter is not true and consequently, \( \varphi_1 \implies \varphi_2 \) can not always imply that \( \varphi_1 \sqsubseteq^U \varphi_2 \).

**Proposition 6.4.2.** The operators \( \sqcup^U \) and \( \sqcap^U \) correspond to the least upper bound and the greatest lower bound w.r.t. the order relation defined by \( \sqsubseteq^U \). Also, \( \sqcup^U \) is a widening operator and \( \top^U \) (respectively, \( \bot^U \)) is the greatest (respectively, the smallest) element of the lattice w.r.t. \( \sqsubseteq^U \).

**Proof.** The properties of \( \sqcup^U \) and, respectively \( \sqcap^U \), follow directly from similar properties of \( \sqcup^Z \) and \( \sqcap^Z \). The proof for the widening operator relies on the properties of the widening operator \( \sqcup^Z \) and on the fact that the set of patterns \( \mathbb{P} \) is fixed.
6.4. A $\mathbb{D}W$-DOMAIN WITH UNIVERSALLY QUANTIFIED FORMULAS

Let $\varphi$ and $\varphi'$ be two formulas defined over the same set of word variables and the same set of patterns:

$$
\varphi = E \land \bigwedge_{P(y,w) \in P(V_1)} \forall y. (P(y,w,W) \implies U_P)
$$

$$
\varphi' = E' \land \bigwedge_{P(y,w) \in P(V_2)} \forall y. (P(y,w,W) \implies U'_P).
$$

Then,

$$
\varphi \sqcup^U \varphi' = E \sqcup Z \land \bigwedge_{P(y,w) \in P(V_1)} \forall y. (P(y,w,W) \implies U_P \sqcup Z U_P).
$$

Suppose, that there exists

$$
\varphi'' = E'' \land \bigwedge_{P(y,w) \in P(V_2)} \forall y. (P(y,w,W) \implies U''_P)
$$

such that $\varphi \sqsubseteq^U \varphi''$ and $\varphi' \sqsubseteq^U \varphi''$. Then, from the definition of $\sqsubseteq^U$ it follows that $E' \sqsubseteq^Z E'', E \sqsubseteq^Z E'', U_P \sqsubseteq^Z U''_P$, and $U'_P \sqsubseteq^Z U''_P$. Using the property of least upper bound of $\sqcup^Z$ we obtain that $E \sqcup^Z E' \sqsubseteq^Z E''$ and $U_P \sqcup Z U'_P \sqsubseteq^Z U''_P$. Then, according to the order relation on $A_U$ we obtain that $\varphi \sqsubseteq^Z \varphi' \sqsubseteq^U \varphi''$. Similarly, $\cap^U$ defines a greatest lower bound operator. The proof that the widening operator insure the convergences of any infinite ascending chain, can be done by contradiction. As in the case of $\sqcup^Z$ one can prove that if applying $\nabla^U$ to an ascending chain of formulas this sequence does not converge then, there exists an infinite ascending chain of element from $A_Z$ whose convergence is not insured by $\nabla^Z$. This fact contradict that $\nabla^Z$ is a widening operator on $A_Z$. \hfill \Box

6.4.2 Concretization based abstraction

The lattice $A_U$ is connected to the concrete data words domain $C_W$ over $\mathbb{D} = \mathbb{Z}$ through the concretization function $\gamma^U$ defined hereafter. The concretization function $\gamma^U$ is defined from formulas in $A_U$ to sets of pairs of functions representing models of the formulas in $A_U$. The concretization function corresponds to the usual semantics of the universally quantified formulas. Let $\varphi(V)$ be an element of $A_U$ over a set of patters $P$ and a numerical abstract domain $A_Z$ given by:

$$
\varphi(V) = E(V) \land \bigwedge_{P(y_1,...,y_q,w_1,...,w_q,W) \in P(V)} \forall y. P(y_1,...,y_q,w_1,...,w_q,W) \implies U_P.
$$

Then,

$$
\gamma^U(\varphi) = \{(v, \delta) \in [V \rightarrow \mathbb{Z}^+] \times [DVar \rightarrow \mathbb{Z}] \mid (v, \delta) \text{ satisfies the constraints (I) and (II)}\},
$$

where

(I) if $V = \{n_1,...,n_p\}$ and $DVar = \{d_1,...,d_s\}$ then,

$$
(\text{len}(\nu(n_1)),...,\text{len}(\nu(n_p)), \text{hd}(\nu(n_1)),...,\text{hd}(\nu(n_p)), \delta(d_1),...,\delta(d_s)) \in \gamma^Z(E).
$$

where for all $1 \leq i \leq p$, $\text{len}(\nu(n_i))$ represents the value of the term $\text{len}(n_i)$ in $E$, $\text{hd}(\nu(n_i))$ represents the value of the term $\text{hd}(n_i)$ in $E$, and for all $1 \leq i \leq s$, $\delta(d_i)$ represents the value of the variable $d_i$. 

189
(II) for every pattern $P(y_1, \ldots, y_q, w_1, \ldots, w_q, W) \in P(V)$, for every interpretation $\mu$ of the variables in $y_1 \cup \ldots \cup y_q$ and of the terms in $\{\text{len}(n_1), \ldots, \text{len}(n_p)\}$ as natural numbers such that $\mu(\text{len}(n_i)) = \text{len}(\nu(n_i))$, for all $1 \leq i \leq p$, if

\[
\mu \models P(y_1, \ldots, y_q, w_1, \ldots, w_q, W)
\]

then

\[
(\text{len}(\nu(n_1)), \ldots, \text{len}(\nu(n_p)), \text{hd}(\nu(n_1)), \ldots, \text{hd}(\nu(n_p)), \delta(d_1), \ldots, \delta(d_s), \\
\mu(y_1), \ldots, \mu(y_q), \nu(w_1)[\mu(y_1)], \ldots, \nu(w_q)[\mu(y_q)]) \in \gamma^Z(U_P),
\]

where (1) for all $1 \leq i \leq p$, $\text{len}(\nu(n_i))$ represents the value of the term $\text{len}(n_i)$ in $U_P$ and $\text{hd}(\nu(n_i))$ represents the value of the term $\text{hd}(n_i)$ in $U_P$, (2) for all $1 \leq i \leq s$, $\delta(d_i)$ represents the value of the variable $d_i$, and (3) for all $1 \leq i \leq q$, $\mu(y_i)$ represent the values of the variables $y_i$ in $U_P$ and $\nu(w_i)[\mu(y_i)]$ represent the integer values from the word $\nu(w_i)$ at the positions from $\mu(y_i)$.

Based on the monotonicity of the concretization function $\gamma^Z$, and on the definition of $\sqsubseteq^U$, we conclude that the concretization function $\gamma^U$ defined above is monotone.

### 6.4.3 Canonical representations

In general, an element from $C_W(N, DVar)$ can have several distinct abstractions in $A_U$. These abstractions may not be even comparable with respect to the order relation in $A_U$.

In the following, we give a procedure called saturation, and denoted $\text{sat}^\#$, which, for any $\varphi \in A_U$, returns an abstract element $\varphi'$ such that $\varphi' \sqsubseteq^U \varphi$ and $\gamma^U(\varphi) = \gamma^U(\varphi')$. We prove that for some particular class of patterns, the output of $\text{sat}^\#$ is the minimal element $\varphi'$ (w.r.t. $\sqsubseteq^U$) such that $\gamma^U(\varphi) = \gamma^U(\varphi')$.

Let $\varphi$ be an element of $A_U$ of the form

\[
E(V) \land \bigwedge_{P(y) \in P(V)} \forall y. P(y) \implies U_P(V, y).
\]

The procedure $\text{sat}^\#$ strengthens the quantifier-free part of $\varphi$ based on the constraints from the universal sub-formulas and then, it strengthens each universal sub-formula based on the constraints from the other universal sub-formulas. It uses three external operators:

1. the projection operator of $A_Z$, denoted $\uparrow$; for any abstract value $U_P \in A_Z$ that appears in the right part of a universal sub-formula of $\varphi$, $U_P \uparrow y'$ denotes the abstract value obtained from $U_P$ by projecting out all the terms of the form $w[y]$ and $y$ with $y \in y'$;

2. a procedure for existential quantifier elimination on boolean combinations of patterns,

3. a procedure for checking entailments between a numerical abstract element (seen as a formula over integers) and a Presburger formula. ($\implies$ denotes in the following the entailment in Presburger arithmetics;)

190
6.4. A \(\mathbb{DW}\)-DOMAIN WITH UNIVERSALLY QUANTIFIED FORMULAS

As a consequence, the precision of \(\text{sat}\#\) depends on the precision of these operators. After describing the procedure \(\text{sat}\#\), we will characterize its precision w.r.t. the syntax of the patterns and the numerical abstract domain \(A_Z\) parameterizing \(A_U\).

**Strengthening the quantifier-free part:** To strengthen the quantifier-free part of \(\varphi\), \(\text{sat}\#\) uses formulas obtained from the right part of an universal formula by projecting out terms that contain position variables.

We start by an example. Let \(\varphi\) be the following formula:

\[
\begin{align*}
\text{len}(n) & \geq 1 \land \forall y. \ [y] \in \text{tl}(n) \implies (\text{hd}(n) \geq 3 \land n[y] \geq 3) \\
\forall y_1, y_2. \ ([y_1, y_2] \in \text{tl}(n) \land y_1 < y_2) & \implies (\text{hd}(n) \leq 5 \land n[y_1] < n[y_2]).
\end{align*}
\]

The strongest constraint implied by \(\varphi\) should contain the property \(\text{hd}(n) \geq 3\) which appears in the right part of the first universal formula. However, it should not contain the constraint \(\text{hd}(n) \leq 5\) from the second universal formula. This is because there are models of \(\varphi\), where \(\text{len}(n) = 1\), s.t. there exist no values for \(y_1\) and \(y_2\) which satisfy the guard of the second universal formula.

For any guard \(P(y) \in \mathbb{P}(\mathcal{Y})\), we test if the quantifier-free part implies that there are values for \(y\) which satisfy \(P(y)\), i.e. \(E \implies \exists y. P(y)\). For any guard \(P(y)\) that satisfies this condition, we take the meet between the quantifier-free part and the formula obtained from the right part of the universal formula by projecting out terms that contain universally-quantified variables, i.e. terms of the form \(y\) or \(n[y]\) with \(y \in \mathcal{Y}\). Let \(E'\) denote the formula obtained in this way.

**Strengthening the universal formulas:** Given a guard \(P(y)\) in \(\mathbb{P}(\mathcal{Y})\), \(\text{sat}\#\) computes an universal formula of the form \(\forall y. P(y) \implies U'_p(y)\) as follows. Initially, \(U'_p(y) = U_p(y)\), if \(\varphi\) contains a formula of the form \(\forall y. P(y) \implies U_p(y)\), or \(U'_p(y) = \text{true}\), otherwise.

Then, the procedure has three steps:

- **Step 1** consist in strengthening \(U'_p\) using \(U_P\) and \(E\);
- **Step 2** consist in strengthening \(U'_p\) using the right hand side of other guards one by one;
- **Step 3** consist in strengthening \(U'_p\) using combinations of the right hand sides of tuples of guards.

**Step 1:** First, it strengthens each universal formula \(\forall y. P(y) \implies U'_p(y)\) without looking to the other universal formulas in \(\varphi\). If \(E' \land P(y)\) is unsatisfiable then we put \(U'_p(y)\) to \(\perp^Z\). Otherwise,

- we apply \(U'_p = U'_p \sqcap^Z E'\);
- if \(P(y)\) can be represented precisely by an object in \(A_Z\) then we apply \(U'_p = U'_p \sqcap^Z P(y)\), and
- for any \(y, y' \in \mathcal{Y}\) representing positions on the same word (i.e., there exists a data-word variable \(n\) s.t. \(y \in \text{tl}(n)\) and \(y' \in \text{tl}(n)\) are sub-formulas of \(P(y)\)), if

\[
(E' \land y = y') \implies (\exists y \setminus \{y, y'\}. P(y))
\]

then we modify \(U'_p\) to

\[
U'_p \sqcap^Z (U'_p [y' \leftarrow y]) \sqcap^Z (U'_p [y \leftarrow y']).
\]
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

This strengthening typically applies to patterns of the form \( y, y' \in tl(n) \ldots y \leq y' \). For example, \( \forall y, y' \in tl(n). y \leq y' \implies d(y) \leq 4 \) it is strengthened to \( \forall y, y' \in tl(n). y \leq y' \implies d(y) \leq 4 \wedge d(y') \leq 4 \).

To exemplify the following constructions, we consider as running example the following formula, denoted \( \varphi_1 \):

\[
\begin{align*}
\text{len}(n) &\geq \text{len}(m) \geq 1 \\
\wedge &\forall y_1, y_2. \left[ [y_1, y_2] \in tl(n) \wedge y_1 <_1 y_2 \right] \implies n[y_2] = n[y_1] + 1 \\
\wedge &\forall y_1, y_2. \left[ [y_1] \in tl(m) \wedge [y_2] \in tl(n) \wedge y_1 = y_2 \right] \implies n[y_2] = m[y_1] \\
\wedge &\forall y_1, y_2. \left[ [y_1, y_2] \in tl(m) \wedge y_1 <_1 y_2 \right] \implies \top
\end{align*}
\]

Next, the procedure \texttt{sat}\# strengthens the universal formulas using other universally quantified formulas.

**Step 2:** In the first step, the saturation procedure strengthens a conjunct of the form \( \forall y. P(y) \implies U'_p \) using each of the other universally quantified conjuncts independently. Intuitively, for every conjunct \( \forall y_a. P_a(y_a) \implies U'_p \), we consider all the ways of renaming universal variables from \( P_a(y_a) \) to variables in \( y \) such that the obtained formula is weaker than the guard \( P(y) \). For every such renaming, we conjunct to \( U'_p \) by applying the same renaming of universal variables and by projecting out all the terms containing universal variables in \( y_a \setminus y \). Formally,

- for every (partial) function \( \rho : y_a \to y \), if \( E' \wedge P(y) \implies P_a[\rho] \), where \( P_a[\rho] \) denotes the formula obtained from \( P_a(y_a) \) by substituting every \( y \in y_a \) with \( \rho(y) \),

\[
U'_p = U'_p \sqcap (U'_p[\rho])(y_a \setminus y)
\]

**Step 3:** In the next steps, the saturation procedure (1) searches for combination of guards from \( \varphi \) which are weaker than the guard \( P(y) \) and (2) for each such a combination it builds a numerical abstract element that is intersected with \( U'_p \). More precisely:

1. we consider existentially quantified positive boolean combinations (without negation) of guards from \( \mathbb{P}(V) \), which are a satisfiable w.r.t. the length constraints in \( E' \).

Let \( \alpha \) be a fixed but arbitrary natural number. Also, let \( \overline{V} \) be a set of position variables defined as the union between \( y \) and a set of \( \alpha \) fresh position variables \( y_1, \ldots, y_\alpha \). Then, let \( \mathbb{P}(V, \overline{V}) \) be the set of all guards obtained from guards in \( \mathbb{P}(V) \) by substituting all position variables with elements from \( \overline{V} \).

For each positive boolean combinations (without negation) of guards \( F(\overline{V}) \) from \( \mathbb{P}(V, \overline{V}) \), \texttt{sat}\# tests if

\[
(E' \wedge P(y)) \implies \exists \overline{V} \setminus y. F(\overline{V}) \text{ is valid.} \tag{6.4.3}
\]

For the formula \( \varphi_1 \) given above, let \( \alpha = 2 \). We consider a set \( \overline{V} \) with 4 position variables: \( y_1, y_2, u_1, \) and \( u_2 \). Then, we enumerate all the satisfiable disjunctions of conjunctions of guards from \( \varphi_1 \) and we find that:

\[
\begin{align*}
\text{len}(n) &\geq \text{len}(m) \geq 1 \land P_m(u_1, u_2) \implies \\
\exists y_1, y_2. \left[ P_n(y_1, y_2) \land P_{m,n}(u_1, y_1) \land P_{m,n}(u_2, y_2) \right] \\
&= F(y_1, y_2, u_1, u_2)
\end{align*}
\]

192
2. For any formula $\exists y \setminus y$. $F(y)$ such that \[6.4.3\] holds, sat#$^{\#}$ computes an universal formula of the form

$$\forall y. F(y) \implies U_F.$$ 

For every guard $Q(y) \in \mathcal{P}(V, y)$ that appears in $F(y)$, let $\forall y_q. Q(y_q) \implies U_Q$ be a conjunct in $\varphi$ such that $Q(y)$ is obtained from $Q(y_q)$ by applying some substitution $\rho_Q : y_q \rightarrow y$. Suppose that $F(y)$ is a boolean combination of a set of guards $\{Q_1(y), \ldots, Q_t(y)\}$. Then, $U_F$ is obtained by taking the boolean combination over $U_{Q_i}[\rho_{Q_i}]$ with $1 \leq i \leq t$, where $\lor$ is replaced by $\lor^Z$ and $\land$ is replaced by $\land^Z$, corresponding to the boolean combination over $Q_i(y)$ with $1 \leq i \leq t$ that defines $F(y)$.

Then, the property implied by $P(y)$ is strengthened by $U'_P = U'_P \lor^Z (U_F \downarrow(y \setminus y))$.

In the example given above, we can compute a formula of the form

$$\forall u_1, u_2, y_1, y_2. F(u_1, u_2, y_1, y_2) \implies U_F(u_1, u_2, y_1, y_2),$$

where

$$U_F(u_1, u_2, y_1, y_2) := n[y_2] = n[y_1] + 1 \land n[y_1] = m[u_1] \land n[y_2] = m[u_2]$$

Then, we will strengthen the formula having as guard $P_m(u_1, u_2)$ by taking the meet between $\top^Z$ and

$$U_F(u_1, u_2, y_1, y_2) \downarrow(y_1, y_2) := m[u_2] = m[u_1] + 1.$$

Consequently, the output of sat#$^{\#}$ contains the universal formula:

$$\forall y_1, y_2. ([y_1, y_2] \in tl(m) \land y_1 <_1 y_2) \implies m[u_2] = m[u_1] + 1.$$ 

After strengthening all the universal formulas, it is possible to add more constraints to $E'$ using the same operations that defined $E'$ from $E$. In particular, for any formula $\forall y. P(y) \implies \bot^Z$ computed above, if $E' \sqsubseteq^Z \exists y. P(y)$ then $E' = \bot^Z$.

Finally, if there exists a closure procedure for the numerical abstract domain $A_Z$ that parametrizes $A_U$ (a procedure that for any $X \in A_Z$ returns the minimal element $X'$ w.r.t $\sqsubseteq^Z$ such that $\gamma(X) = \gamma(X')$) then we apply it on every abstract value in $A_Z$ obtained above.

The next theorem proves that, by applying sat#$^{\#}$ to some abstract value $\varphi \in A_U$, we obtain an abstract value having the same concretization but smaller than $\varphi$ w.r.t. $\sqsubseteq^U$.

**Theorem 6.4.1.** For any abstract value $\varphi \in A_U$, sat#$^{\#}(\varphi) \sqsubseteq^U \varphi$ and $\gamma^U(sat#$$(\varphi)) = \gamma^U(\varphi)$.

**Proof.** The first property holds because sat#$^{\#}$ modifies the numerical abstract values from $\varphi$ by applying the meet between the old value and some other abstract values in $A_Z$.

Then, $\gamma^U(sat#$$(\varphi)) = \gamma^U(\varphi)$ follows from the following:

- the quantifier-free part of sat#$^{\#}(\varphi)$, $E'$, is obtained as the meet between $E$ and $U_P[y]$ for any $\forall y. P(y) \implies U_P$ such that $E \implies \exists y. P(y)$. We have that

$$\gamma^U(E' \land \psi) = \gamma^U(E') \land \gamma^U(\psi) = \gamma^U(E) \cap \left( \bigcap_{\forall y. P(y) \implies U_P. \exists y. P(y) \implies E} \gamma^U(U_P[y]) \right) \land \gamma^U(\psi),$$
where ϕ is the conjunction of all universal sub-formulas of φ. Notice that if $E \implies \exists y. P(y)$ then $\gamma^U(\phi) \subseteq \gamma^U(U_P[y])$. Consequently,

$$\gamma^U(E) \cap \gamma^U(\phi) \subseteq \bigcap_{\forall y. P(y) \implies U_P \in \psi} \gamma^U(U_P[y])$$

and $\gamma^U(E' \land \psi) = \gamma^U(E \land \psi)$.

- concerning the strengthening of universal sub-formulas, a key fact is that the concretization of φ is included in the concretization of all formulas $\forall y. F(y) \implies U_P$ built in the second step, where $\exists y. F(y)$ is a formula such that (6.4.3) holds. This is a direct consequence of the following inference rules (we suppose that $P_1(y_1)$ and $P_2(y_2)$ are satisfiable):

  - if $\forall y_1. P_1(y_1) \implies U_1(y_1) \land \forall y_2. P_2(y_2) \implies U_2(y_2)$ then

    $\forall y_1, y_2. (P_1(y_1) \land P_2(y_2)) \implies (U_1(y_1) \land U_2(y_2))$

    $\forall y_1, y_2. (P_1(y_1) \lor P_2(y_2)) \implies (U_1(y_1) \lor U_2(y_2))$

  - if $\forall y. P(y) \implies U(y)$ then, for any $y \in y$, $\forall y \setminus \{y\}. (\exists y. P(y)) \implies (\exists y. U(y))$

The operations which slow down the performance of this saturation procedure are (1) the satisfiability test in Presburger logic between the quantifier free part of φ and a guard, (2) the enumeration of boolean combinations of patterns, (3) an exact procedure for quantifier elimination over guards, and (4) testing implication in Presburger between a guard and an existentially quantified boolean combination of guards. In the following, we identify different sub-classes of patterns, such that the tests above are over-approximated and Presburger satisfiability is avoided, while the result of the saturation procedure is sound or even exact (sat# computes the minimal abstract value with the same concretization as φ).

**Definition 6.4.1** (Simple values). A pattern $P(y_1, \ldots, y_q, w_1, \ldots, w_q)$ with $y_1 = y_1^1 \ldots y_1^{p_1}$, for any $1 \leq i \leq q$, of the form

$$\bigwedge_{1 \leq i \leq q} y_i \in \text{tl}(w_i) \land y_i^1 \leq y_i^2 \leq \ldots \leq y_i^{p_i}$$

is called a simple pattern. An abstract value $E \land \bigwedge_{P(y,w) \in P(V)} \forall y. P(y,w) \implies U_P$ is called a simple abstract value if it belong to the abstract domain $A_U$ parametrized by a set of simple patterns and every $U_P$ does not constraint any position variable $y \in y$.

Next, we describe the simplification that can be applied to the saturation procedure if we consider only formulas over simple patterns:

1. the strengthening of the quantifier-free part of the input formula φ tests if $E \implies \exists y. P(y, w), w$. For any simple guard $P(y, w)$, $\exists y. P(y, w)$ is equivalent to the conjunction $\bigwedge_{w \in w} \text{len}(w) > 1$. This constraint can be represented by an element of the numerical abstract domain of intervals (see Section 3.4). Therefore, when we consider $A_{\mathbb{Z}}$ to be a numerical abstract domain at least as expressive as the abstract domain of intervals, the implication $E \implies \exists y. P(y, w)$ is over-approximated using the entailment $\subseteq^{\mathbb{Z}}$ in $A_{\mathbb{Z}}$ by $E \subseteq^{\mathbb{Z}} \bigwedge_{w \in w} \text{len}(w) \geq 2$. 

194
2. the strengthening of universally quantified formulas: let $\forall y. P(y, w) \implies U_P$ be the universal formula to be strengthened. Then,

- let $P_a(y_a, w_a)$ be the guard of some universal formula in $\varphi$, where $w_a$ are the data word variables in $P_a$. To test that there is at least one valuation for the position variables $y_a$ that satisfies $P(y_a, w_a)$, we use the fact that the only constraint imposed by the guard is that the length of the words denoted by the variables in $w_a$ is greater than one. Then, the satisfiability of $E \land P_a(y_a, w_a)$ is reduced to checking that $E \subseteq \mathbb{Z} \land_{w \in w_a} \text{len}(w) > 1$;

- all guards $P_a(\text{vec}y_a, w_a)$ that do not satisfy the constraint $E \subseteq \mathbb{Z} \land_{w \in w_a} \text{len}(w) \leq 1$, for some $w \in w_a$, then every such guard $P_a(y_a, w_a)$ implies $\bot \in \text{sat}^{\#}(\varphi)$, that the corresponding $U_P$ formula equals $\bot$;

- the elimination of position variables from simple patterns is straightforward due to their syntax. For any pattern $P_a(y_a, w_a)$ and $y'_a \subseteq y_a$, $\exists y'_a. P_a(y_a, w_a)$ is a pattern where for every $y'_a \in y'_a$, (1) we remove the atomic formula $y'_a \in t_1(w)$ and (2) we replace the sub-formula $y_i \leq \ldots \leq y_i \leq y'_{i+1} \leq \ldots \leq y_i$ of $P_a$ by $y_1 \leq \ldots \leq y_i \leq y_{i+1} \leq \ldots \leq y_i$;

- if $P(y, w)$ is a guard defined over only one word variable $w$, that is, all the position variables $y$ belong to the word denoted by $w$, the saturation procedure applies only the first step from the strengthening of $\forall y. P(y, w) \implies U_P$. Moreover, since we consider only simple patterns, the implication $E \land P(y, w) \implies P_a[\rho]$, where $P_a$ is a guard over the position variables $y_a$, and $\rho : y_a \to y$, can be checked very easily. It suffices to check that for any two variables $y, y' \in y$, if $y \leq y'$ is an atomic formula in $P(y, w)$ then the order constraint in $P_a(y_a)$ implies that $\rho^{-1}(y)$ is less than or equal to $\rho^{-1}(y')$.

- if $P(y, w)$ is a guard defined over more than one word variable then, it is enough to strengthen $P(y, w)$ using conjunctions of guards from $\varphi$. That is, the formulas $F(\overline{y})$ from the second step of the strengthening of $\forall y. P(y) \implies U_P$ are conjunctions of guards from $\varphi$ modulo a renaming of the position variables.

The reason for which disjunctions are not relevant is that, if $P(y) \implies \exists y. \left( P_1(y) \lor P_2(y) \right)$ then $P(y) \implies \exists y. P_1(y)$ and $P(y) \implies \exists y. P_2(y)$ (in general this is not true; for example, $y \in t_1(w) \implies (y \in t_1(w) \land y > 3) \lor (y \in t_1(w) \land y > 3)$ does not imply $y \in t_1(w) \implies (y \in t_1(w) \land y > 3)$).

The following theorem shows that the saturation procedure for elements $\varphi$ of a domain $\mathcal{A}_U$ parametrized by simple patterns returns the minimal value w.r.t. $\subseteq \mathbb{Z}$ having the same concretization as $\varphi$ (when the numerical abstract domain has an exact projection operator and a closure procedure). For a set of abstract values $\mathcal{X}$ of some abstract domain $A$, $\inf\mathcal{X}$ denotes the minimal element of $\mathcal{X}$ w.r.t. the order relation $\subseteq$ in $\mathcal{A}$.

**Theorem 6.4.2.** Let $\mathcal{A}_U$ be an abstract domain of universally quantified formulas parametrized by simple patterns and by a numerical abstract domain $\mathcal{A}_Z$ with an exact projection operator and a closure procedure. For every $\varphi \in \mathcal{A}_U$, $\text{sat}^{\#}(\varphi) = \inf\{\varphi' | \gamma_U(\varphi') = \gamma_U(\varphi)\}$. 

195
6.4.1 Remark. The theorem above can not be extended to more general patterns. If we consider patterns of the form
\[ \bigwedge_{1 \leq i \leq q} y_i \in \text{tl}(w_i) \land y_i^1 < y_i < \ldots < y_i^{p_i} \]
then some of the universal properties depend on the length of the words. For example, let \( \mathcal{V} = \{ w \} \), \( \mathbb{P}(\mathcal{V}) = \{ [y_1, y_2] \in \text{tl}(w) \land y_1 < y_2, [y_1] \in \text{tl}(w) \} \), and \( \varphi \) be the abstract element
\[
\text{hd}(w) = 0 \land \text{len}(w) = 6
\]
\[
\land \forall y_1. [y_1] \in \text{tl}(w) \implies (1 \leq w[y_1] \leq 5)
\]
\[
\land \forall y_1, y_2. ([y_1, y_2] \in \text{tl}(w) \land y_1 < y_2) \implies w[y_1] < w[y_2].
\]

Notice that the two universal formulas in \( \varphi \) and the fact that the length of \( w \) is 6 induce the following property
\[ \forall y_1. [y_1] \in \text{tl}(w) \implies (w[y_1] = y_1) \] (6.4.4)
and consequently, a smaller abstract value than \( \varphi \) (w.r.t. \( \sqsubseteq^U \)) is
\[
\text{hd}(w) = 0 \land \text{len}(w) = 6
\]
\[
\land \forall y_1. [y_1] \in \text{tl}(w) \implies (w[y_1] = y_1 \land 1 \leq w[y_1] \leq 5)
\]
\[
\land \forall y_1, y_2. ([y_1, y_2] \in \text{tl}(w) \land y_1 < y_2) \implies w[y_1] < w[y_2].
\]

The procedure that we have presented uses the projection from \( \mathcal{A}^Z \), which in this case cannot induce a relation between positions and data.

If we consider the patterns describing consecutive positions, i.e. of the form
\[ \bigwedge_{1 \leq i \leq q} y_i \in \text{tl}(w_i) \land y_i^1 < y_i^2 < \ldots < y_i^{p_i} \]
we encounter a similar difficulty. For example, let \( \mathcal{V} = \{ w \} \), \( \mathbb{P}(\mathcal{V}) = \{ [y_1, y_2] \in \text{tl}(w) \land y_1 < y_2, [y_1] \in \text{tl}(w) \} \), and \( \varphi \) be the abstract element
\[
\text{hd}(w) = 0 \land \text{len}(w) = 6
\]
\[
\land \forall y_1. [y_1] \in \text{tl}(w) \implies (1 \leq w[y_1] \leq 5)
\]
\[
\land \forall y_1, y_2. ([y_1, y_2] \in \text{tl}(w) \land y_1 < y_2) \implies w[y_2] = w[y_1] + 1.
\]

As in the previous case, the universal formulas and the fact that the length of \( w \) is 6 induce the following property
\[ \forall y_1. [y_1] \in \text{tl}(w) \implies (w[y_1] = y_1) \]
which is not obtained from projections since it does not take into consideration the length of \( w \) and the successor relation between \( y_1 \) and \( y_2 \).

6.4.4 Abstract semantics
In the following, we define the abstract transformers corresponding to the concrete ones given in Section 6.3.3. For each concrete transformer \( F \in \mathcal{F}^C \) we denote by \( \mathcal{F}^A \) the corresponding abstract version in the domain of universally quantified formulas \( \mathcal{A}^U \) parametrized by a set of patterns \( \mathbb{P} \). We start by describing the most interesting transformers of \( \mathcal{A}^U \) which are \( \text{concat}^\#_{\mathbb{P}} \) and \( \text{split}^\#_{\mathbb{P}} \).
6.4. A $\mathbb{D} \mathbb{W}$-DOMAIN WITH UNIVERSALLY QUANTIFIED FORMULAS

6.4.1 Overview by example of the main abstract transformers

Let’s consider two formulas $\varphi_1 = \text{sorted}(n_1) \land \text{less}(n_1)$ and $\varphi_2 = \text{sorted}(n_2) \land \text{less}(n_1)$ expressing the sortedness property of two different words $n_1$ and $n_2$, where

\[
\text{sorted}(n) := \forall y_1, y_2. [y_1, y_2] \in \text{tl}(n) \land y_1 \leq y_2 \implies n[y_1] \leq n[y_2]
\]

\[
\text{less}(n) := \forall y. [y] \in \text{tl}(n) \implies \text{hd}(n) \leq n[y].
\]

Suppose that we want to apply $\text{concat}_P^\#(\{(n, [n_1, n_2])\}, \varphi_1 \land \varphi_2)$ where $P = \{[y_1, y_2] \in \text{tl}(\omega) \land y_1 \leq y_2, [y] \in \text{tl}(\omega)\}$. The result should contain properties of the word obtained by concatenating $n_1$ and $n_2$, which is denoted by $n$. The universal formulas that describe $n$ should have as guards constraints from $P(\{n\})$.

A simple definition for $\text{concat}_P^\#$, based on the same principle used for defining the lattice operators, could take the union of the properties expressed using the same pattern on each word ($n_1$ and $n_2$) and define it as a property of the concatenation $n$. Unfortunately, this definition is unsound. For example, $\text{concat}_P^\#(\{(n, [n_1, n_2])\}, \varphi_1 \land \varphi_2) = \text{sorted}(n) \land \text{less}(n)$ is not sound because the concatenation of two sorted words is not always sorted (for example, $[1, 3, 5] \in \gamma^U(n_1)$, $[2, 4, 6] \in \gamma^U(n_2)$, and the concatenation $[1, 3, 5, 2, 4, 6]$ is not sorted).

In the following, we give a sound definition for $\text{concat}_P^\#$ which is guided by the guards of the universal formulas and which is based on a relation between patterns and sets of patterns, called $\text{Closure}$ (defined in Section 6.4.4.2). If we go back to the sortedness of $n_1$ and $n_2$, then $\text{sorted}(n_1) \land \text{less}(n_1)$ characterizes the data values in the first part of the concatenation and $\text{sorted}(n_2) \land \text{less}(n_2)$ characterizes the data values in the second part. But, out of two positions in the concatenation, one might be in $n_1$ (different from the first element of $n_1$) and the other one in $n_2$. Therefore, to define a sound $\text{concat}_P^\#$ transformer, we need a property expressed with the pattern

\[
\forall y_1, y_2. [y_1] \in \text{tl}(n_1) \land [y_2] \in \text{tl}(n_2).
\]

Furthermore, the semantics we have chosen for $y_1 \in \text{tl}(n_1)$ interprets $y_1$ as a position of $n_1$ different from the first one (similar for $n_2$). So this pattern does not capture the situation when $y_1$ is a position of $n_1$ and $y_2$ is the first element of $n_2$, nor the situation when $y_1$ is the first element of $n_2$ and $y_2$ is a position in the tail on $n_2$. The pattern syntax forbids formulas of the form $\forall y_1, y_2. [y_1] \in \text{tl}(n_1) \land y_2 = \text{fst}(n_2) \implies U_{P_1}$ and $\forall y_1, y_2. y_1 = \text{fst}(n_2) \land [y_2] \in \text{tl}(n_2) \implies U_{P_2}$, where $\text{fst}(n_2)$ denotes the first position in the word $n_2$, because the data value associated with the first node of $n_2$ is already represented by $\text{hd}(n_2)$. In the definition of $\text{concat}_P^\#$, we associate with each pattern $P \in P$ a set of extended patterns which depend on the number of concatenated segments, called extended closure and denoted $\text{Closure}_P^\ext(P, u_1, \ldots, u_q)$ ($u_1, \ldots, u_q$ are vectors of data word variables and $q$ is the number of words to which $P$ associates position variables) the extended closure is formally defined in Section 6.4.4.2. For the sake of readability, these patterns respect the syntax given in Section 6.4.1, but, they can use atomic formulas of the form $y = \text{fst}(n)$, for some position variable $y$ and some data word variable. For example, the extended closure of $P_{\leq} := [y_1, y_2] \in \text{tl}(n) \land y_1 \leq y_2$ for $u = [n_1, n_2]$ is:

\[
\text{Closure}_P^\ext(P_{\leq}, [n_1, n_2]) = \{ P_1(n_1, n_2) := [y_1] \in \text{tl}(n_1) \land [y_2] \in \text{tl}(n_2), \\
P_2(n_1, n_2) := [y_1] \in \text{tl}(n_1) \land y_2 = \text{fst}(n_2), \\
P_3(n_2, n_2) := y_1 = \text{fst}(n_2) \land [y_2] \in \text{tl}(n_2), \\
P_{\leq}(n_1) := [y_1, y_2] \in \text{tl}(n_1) \land y_1 \leq y_2, \\
P_{\leq}(n_2) := [y_1, y_2] \in \text{tl}(n_2) \land y_1 \leq y_2, \}
\]
Similarly, the extended closure of \( P := [y] \in \text{tl}(n) \) for \( u = [n_1, n_2] \) is:

\[
\text{Closure}^{\text{ext}}(P, [n_1, n_2]) = \{ P_4(n_1) := y = \text{fst}(n_1), \\
\quad P(n_1) := y \in \text{tl}(n_1), \\
\quad P(n_2) := y \in \text{tl}(n_2) \}.
\]

To come back to the syntax of \( A_U \) elements, for any pattern \( P' \) in the extended closure, we define a pattern \( \sigma(P') \) in the syntax given in Section 6.4.1.1. Roughly, \( \sigma(P') \) eliminates all the position variables \( y \) which appear in atoms of the form \( y = \text{fst}(n) \) in \( P' \). If \( \forall y'. \sigma(P')(y') \implies U_{\sigma(P')} \) is a sub-formula of some element \( \varphi \in A_U \) then we can compute a numerical abstract element \( U_{\sigma(P)} \), starting from \( U_{\sigma(P')} \) and the quantifier-free part of \( \varphi \), such that all the elements from the concretization of \( \varphi \) satisfy the formula \( \forall y. P'(y) \implies U_{\sigma(P')} \).

For example, \( \sigma(P_2(n_1, n_2)) := [y_1] \in \text{tl}(n_1) \). Recall that the \( A_U \) element \( \varphi_1 \land \varphi_2 \) contains the formula \( \forall y_1, y_1 \in \text{tl}(n_1) \implies U_{\sigma(P_2)} \), where \( U_{\sigma(P_2)} := \text{hd}(n_1) \leq n_1[y_1] \). We compute a numerical abstract element

\[
U_{P_2} := \text{hd}(n_1) \leq n_1[y] \land n_2[y_2] = \text{hd}(n_2)
\]

by adding to \( U_{\sigma(P_2)} \) the constraint \( n_2[y_2] = \text{hd}(n_2) \) such that \( \forall y_1, y_2. P_2(n_1, n_2) \implies U_{P_2} \) is implied by \( \varphi_1 \land \varphi_2 \). Notice that \( \sigma(P_4(n_1)) := \text{true} \) and consequently, the definition of \( U_{P_3} \) depends only on the constraints from the quantifier free part.

Let \( \text{Closure}(P, u_1, \ldots, u_q) \) be the set of patterns obtained by applying the transformation \( \sigma \) to every pattern in \( \text{Closure}^{\text{ext}}(P, u_1, \ldots, u_q) \). For the patterns considered above,

\[
\text{Closure}(P_{\leq}, [n_1, n_2]) = \{ P_1(n_1, n_2) := [y_1] \in \text{tl}(n_1) \land [y_2] \in \text{tl}(n_2), \\
\quad P(n_1) := [y] \in \text{tl}(n_1), P(n_2) := [y] \in \text{tl}(n_2), \\
\quad P_{\leq}(n_1) := [y_1, y_2] \in \text{tl}(n_1) \land y_1 \leq y_2, \\
\quad P_{\leq}(n_2) := [y_1, y_2] \in \text{tl}(n_2) \land y_1 \leq y_2 \}
\]

\[
\text{Closure}(P_1, [n_1, n_2]) = \{ P(n_1) := y \in \text{tl}(n_1), P(n_2) := y \in \text{tl}(n_2) \}
\]

Roughly, the \( \text{concat}^{\#} \) transformer associates with the pattern \( P_{\leq}(n) \) (resp., \( P(n) \)), where \( n \) represents the concatenation of \( n_1 \) and \( n_2 \), the union of the properties expressed with the patterns in the extended closure of \( P_{\leq} \) (resp., \( P \)) modulo some substitution denoted \( \rho_{P_{\leq}} \) (resp. \( \rho_P \)). These substitutions reflect the fact that any position \( i \) in \( n_1 \) corresponds to the position \( i \) in \( n \) (the concatenation of \( n_1 \) and \( n_2 \)), and any position \( j \) in \( n_2 \) corresponds to the position \( \text{len}(n_1) + j \) in \( n \). For each \( P' \in \text{Closure}^{\text{ext}}(P_{\leq}, [n_1, n_2]) \), let \( \rho_{P'} \) denote the substitution applied to \( U_{P'} \). So,

\[
\text{concat}^{\#}([\{n, [n_1, n_2]\}], \varphi_1 \land \varphi_2) := \forall y_1, y_2. P_{\leq}(n) \implies U_{P_{\leq}} \land \forall y. P(n) \implies U_P,
\]

\[
U_{P_{\leq}} = \bigcup_{P' \in \text{Closure}^{\text{ext}}(P_{\leq}, [n_1, n_2])} U_{P'}[\rho_{P'}] \quad \text{and} \quad U_P = \bigcup_{P' \in \text{Closure}^{\text{ext}}(P, [n_1, n_2])} U_{P'}[\rho_{P'}].
\]

As expected,

\[
\text{concat}^{\#}([n_1, n_2], \varphi_1 \land \varphi_2) := \forall y_1, y_2. [y_1, y_2] \in \text{tl}(n) \land y_1 \leq y_2 \implies \top^Z \\
\land \forall y. [y] \in \text{tl}(n) \implies \top^Z
\]

198
6.4. A $\mathcal{D}W$-DOMAIN WITH UNIVERSALLY QUANTIFIED FORMULAS

because

\[
\left(\text{hd}(n_1) \leq n[y_1] \land n[y_2] = \text{hd}(n_2)\right) \cup^\mathbb{Z} (n[y_1] \leq n[y_2]) = T^\mathbb{Z} \sqsubseteq^\mathbb{Z} U_{P^\mathbb{Z}}
\]

and

\[
\left(E[p_{\mathcal{L}}] \cup^\mathbb{Z} (\text{hd}(n_1) \leq n[y_1])\right) = T^\mathbb{Z} \sqsubseteq^\mathbb{Z} U_P
\]

(for the $A_\mathcal{U}$ element $\varphi_1 \land \varphi_2$, the quantifier-free part $E$ is $T^\mathbb{Z}$). The formal definition of $\text{concat}_\mathcal{P}$ is given in Section 6.4.4.3.

Let us consider $\varphi \in A_\mathcal{U}, [m_1, \ldots, m_j]$ a vector of word variables and $m$ a data word variable. The definition of $\text{concat}_\mathcal{P}^\sharp((m, [m_1, \ldots, m_j]), \varphi)$ assumes that $\varphi$ contains a universal sub-formula for each pattern $P'(y)$ in $\text{Closure}(P, [m_1, \ldots, m_j])$, for every $P \in \mathcal{P}$. If this is not the case, a sound approach is to add to $\varphi$ a formula of the form $\forall y. P'(y) \implies T^\mathbb{Z}$ for every missing pattern $P'$. However, this approach might suffer from great lack of precision. For example, let us consider

\[
\varphi := \text{hd}(n_1) \leq \text{hd}(n_2) \land \\
\forall y_1, y_2. P_{\leq}(n_1) \implies (n_1[y_1] \leq n_2[y_2] \land \text{hd}(n_1) \leq n_1[y_1] \land n_1[y_2] \leq \text{hd}(n_2))
\]

\[
\forall y_1, y_2. P_{\leq}(n_2) \implies (n_2[y_1] \leq n_2[y_2] \land \text{hd}(n_2) \leq n_2[y_1]).
\]

If we add to $\varphi$ the sub-formulas $\forall y. P'(y) \implies T^\mathbb{Z}$ for every pattern $P'(y) \in \text{Closure}(P_{\leq}, [n_1, n_2])$ then

\[
\text{concat}_\mathcal{P}^\sharp((\{n, [n_1, n_2]\}), \varphi) = T^\mathbb{Z} \land \forall y_1, y_2. P_{\leq}(n) \implies T^\mathbb{Z} \land \forall y. P(n) \implies T^\mathbb{Z}.
\]

To gain precision, before applying the $\text{concat}_\mathcal{P}^\sharp$ abstract transformer, we apply the saturation procedure $\text{sat}_\mathcal{P}^\sharp$ defined in Section 6.4.3. This procedure strengthens the right hand side of the universally-quantified implications, including the ones that we added and have as right part $T^\mathbb{Z}$. For the formula $\varphi$ considered above, let

\[
\varphi' := \varphi \\
\wedge \forall y. [y] \in tl(n_1) \implies T^\mathbb{Z} \land \forall y. [y] \in tl(n_2) \implies T^\mathbb{Z} \\
\wedge \forall y_1, y_2. ([y_1] \in tl(n_1) \land [y_2] \in tl(n_2)) \implies T^\mathbb{Z}.
\]

By applying $\text{sat}_\mathcal{P}^\sharp$ to $\varphi'$, we obtain the formula:

\[
\varphi'' := \varphi \\
\wedge \forall y. [y] \in tl(n_1) \implies \text{hd}(n_1) \leq n_1[y] \leq \text{hd}(n_2) \\
\wedge \forall y. y \in tl(n_2) \implies \text{hd}(n_2) \leq n_2[y] \\
\wedge \forall y_1, y_2. ([y_1] \in tl(n_1) \land [y_2] \in tl(n_2)) \implies \\
(\text{hd}(n_1) \leq n_1[y] \leq \text{hd}(n_2) \land \text{hd}(n_2) \leq n_2[y])
\]

Then, $\text{concat}_\mathcal{P}^\sharp(\{(n, [n_1, n_2])\}, \text{sat}_\mathcal{P}^\sharp(\varphi'))$ is a formula stating that the word denoted by $n$ is sorted.
6.4.4.2 Formal definition of $\text{Closure}(P, u_1, \ldots, u_q)$

Let $P(y_1, \ldots, y_q, w_1, \ldots, w_q, W)$ be a pattern

$$
\left( \bigwedge_{1 \leq i \leq q} \left( y_i \in \text{tl}(w_i) \land y_i^1 \sim^1_i y_i^2 \sim^2_i \cdots \sim^{p_i-1}_i y_i^{p_i} \right) \right) \land P_L(y_1^1, \ldots, y_q^1, W)
$$

and let $u_1, \ldots, u_q$ be a set of vectors of data word variables. Suppose that $w_i$ represents the concatenation of the words denoted by $u_i$, $1 \leq i \leq q$.

The procedure that computes $\text{Closure}(P, u_1, \ldots, u_q)$ has two steps:

Step 1 first, it computes $\text{Closure}^\text{ext}(P, u_1, \ldots, u_q)$: it defines all the patterns over position variables in $y_1 \cup \cdots \cup y_q$ such that (1) the position variables denote positions belonging to one of the words in $u_1 \cup \cdots \cup u_q$ and (2) the values of the position variables satisfy the ordering constraints in $P$ and the linear constraints in $P_L$;

Step 2 then, it transforms the patterns from the extended closure such that they are in the syntax of $\mathcal{A}_U$ formulas: $\text{Closure}(P, u_1, \ldots, u_q) = \sigma(\text{Closure}^\text{ext}(P, u_1, \ldots, u_q))$.

**Step 1:** Let $u_i = [u_i^1, \ldots, u_i^{m_i}]$ and let $P_R^i(y_i, w_i)$ be the sub-formula of $P$:

$$
y_i \in \text{tl}(w_i) \land y_i^1 \sim^1_i y_i^2 \sim^2_i \cdots \sim^{p_i-1}_i y_i^{p_i},
$$

where $\sim^1_i, \ldots, \sim^{p_i-1}_i \in \{<, \leq, <_1\}$. Intuitively, the procedure considers all the possible ways of choosing $p_i$ positions satisfying the order constraint in $P_R^i$ on the word representing the concatenation of the words $u_i^1, \ldots, u_i^{m_i}$. It will consider all the possible ways of choosing an arbitrary number of positions on the word $u_i^1$, an arbitrary number of positions on the word $u_i^2$, etc. such that, at the end, the number of chosen positions is $p_i$.

For every $1 \leq i \leq q$, let $\Pi_i$ be a set of functions $\pi_i : y_i \rightarrow \{\text{fst}(u_i^l), \text{tl}(u_i^l) \mid 1 \leq l \leq m\}$ such that for any $1 \leq j \leq p_i - 1$,

1. if $\sim^j_i$ is $\leq$ then

   $$
   \pi_i(y_i^j) \in \{\text{fst}(u_i^l), \text{tl}(u_i^l)\},
   \pi_i(y_i^{j+1}) \in \{\text{fst}(u_i^{l'}), \text{tl}(u_i^{l'})\}, \text{ and } l \leq l'.
   $$

   Also, if $\pi_i(y_i^j) = \text{tl}(u_i^l)$ then $\pi_i(y_i^{j+1}) \neq \text{fst}(u_i^l)$.

2. if $\sim^j_i$ is $<$ then all the conditions from the previous case hold. Moreover, if $\pi_i(y_i^j) = \text{fst}(u_i^l)$ then $\pi_i(y_i^{j+1}) \neq \text{fst}(u_i^l)$.

3. if $\sim^j_i$ is $<_1$ then one of the following is true:

   (1) $\pi_i(y_i^j) = \text{fst}(u_i^l)$, $\pi_i(y_i^{j+1}) = \text{fst}(u_i^{l'})$, $l' = l + 1$, and $\text{len}(u_i^l) = 1$;

   (2) $\pi_i(y_i^j) = \text{fst}(u_i^l)$, $\pi_i(y_i^{j+1}) = \text{tl}(u_i^l)$, and $y_i^{j+1} = 1$;

   (3) $\pi_i(y_i^j) = \text{tl}(u_i^l)$, $\pi_i(y_i^{j+1}) = \text{fst}(u_i^{l'})$, $l' = l + 1$, and $y_i^{j+1} = \text{len}(u_i^l) - 1$;

   (4) $\pi_i(y_i^j) = \text{tl}(u_i^l)$ and $\pi_i(y_i^{j+1}) = \text{tl}(u_i^l)$;

200
Let $\Pi$ be the set of all families of functions $(\pi_i)_{1 \leq i \leq q}$. Then, every $\pi = (\pi_i)_{1 \leq i \leq q}$ describes a unique pattern over the position variables $y_1 \cup \cdots \cup y_q$, denoted $P^\pi$, which is a conjunction between $P^{\pi_i}(y_i, u_i)$, for every $1 \leq i \leq q$, where

$$P^{\pi_i}(y_i, u_i) = \bigwedge_{y \in y_i, \pi_i(y) = u_i} y \in \pi_i(y) \land \bigwedge_{y \in y_i, \pi_i(y) = b_i} y = \pi_i(y) \land y \sim^1 y_1^2 \sim^2 \ldots \sim^{n-1} y_i \land P^\pi_{i-1},$$

where $P^\pi_{i-1}$ is the conjunction between constraints like $\text{len}(u) = 1$, $y = 1$, and $y = \text{len}(u) - 1$ that are used in the definition of $\pi_i$, for every $1 \leq i \leq q$.

Finally, we define

$$\text{Closure}^{\text{ext}}(P, u_1, \ldots, u_q) = \{ P^\pi \land P^\pi_L[\theta_\pi] \mid \pi \in \Pi \},$$

where $\theta_\pi$ is a substitution that reflects the fact that a position $y$ in the concatenation $u_i$ corresponds to $y - \text{len}(u_i) - \ldots - \text{len}(u_{i-1})$ in $u_i$.

Formally, $\theta_\pi$ is defined by (for any $1 \leq i \leq q$, $u_i = (u_{i1}, u_{i2}, \ldots, u_{il})$):

- for any $1 \leq i \leq q$, $\theta_\pi$ substitutes $\text{len}[u_i]$ with $\text{len}[u_{i1}] + \text{len}[u_{i2}] + \cdots + \text{len}[u_{il}]$,
- for any $1 \leq i \leq q$, if $y_{i0}^1$ is a position in the tail of the word $u_{i0}^{o+1}$, for some $1 \leq o < j_i$, then $\theta_\pi$ substitutes $y_{i1}^o$ with $y_{i1}^o + \text{len}[u_i] + \cdots + \text{len}[u_{oi}]$,
- for any $1 \leq i \leq q$, if $y_{i1}^o$ is the first position of the word $u_{i0}^{o+1}$, for some $1 \leq o < j_i$, then we substitute $y_{i1}^o$ with $\text{len}[u_{i1}] + \cdots + \text{len}[u_{oi}]$.

The linear constraints from the patterns obtained in the previous step involve either the first or the last position variable associated to some word variable $u$. We eliminate all conditions on the last position variable, denoted $y$, unless (1) it is the only variable in $u$ or (2) the pattern contains a constraint of the form $y_1 < y_2 < \ldots < y_l$, where $y_1, \ldots, y_l$ are all the position variables denoting positions of $u$. In the last case, we substitute $y_l$ with $y_l + 1$.

**Step 2:** Let $P'$ be a pattern in $\text{Closure}^{\text{ext}}(P, u_1, \ldots, u_q)$ of the form

$$P' := \bigwedge_{1 \leq i \leq q} P^{\pi_i}(y_i, u_i) \land P^\pi_L[\theta_\pi].$$

Then, $\sigma(P')$ is obtained from $P'$ by eliminating all occurrences of a position variable $y$ that appears in an atomic formula of the form $y = \text{fst}(n)$, where $n \in u_i$, for some $1 \leq i \leq q$, as follows:

1. replacing $y = \text{fst}(n)$ with $\text{true}$,
2. replacing $y' \sim y$ or $y \sim y'$ with $\text{true}$, for any $y'$ position variable in $P'$,
3. eliminating other occurrences of $y$,
4. eliminate all length constraints of the form $\text{len}(n) = ct$ where $ct$ is an integer constant and $n$ is a word variable without any position variables associated to it.

Then,

$$\text{CLOSURE}(P, u_1, \ldots, u_q) = \bigcup_{P' \in \text{Closure}^{\text{ext}}(P, u_1, \ldots, u_q)} \sigma(P').$$
### CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

<table>
<thead>
<tr>
<th>Pattern $P^\text{ext}_{\text{fib}}(n_1, n_2)$</th>
<th>Closed(Closure$^\text{ext}(P_{\text{fib}}, [n_1, n_2])$)</th>
<th>$P_1(n_1, n_2)$</th>
<th>$P_2(n_1, n_2)$</th>
<th>$P_3(n_1)$</th>
<th>$P_4(n_2)$</th>
<th>$P_{\text{fib}}(n_1)$</th>
<th>$P_{\text{fib}}(n_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P^\text{ext}_1(n_1, n_2)$</td>
<td>$[y_1] \in t_1(n_1) \land y_1 = \text{len}(n_1) - 1 \land y_2 = \text{fst}(n_2) \land [y_3] \in t_2(n_2) \land y_3 = 1$</td>
<td>$[y_1] \in t_1(n_1) \land y_1 = \text{len}(n_1) - 1$</td>
<td>$[y_3] \in t_2(n_2) \land y_3 = 1$</td>
<td>$P_1(n_1, n_2)$</td>
<td>$P_2(n_1, n_2)$</td>
<td>$P_3(n_1)$</td>
<td>$P_4(n_2)$</td>
</tr>
<tr>
<td>$P^\text{ext}_2(n_1, n_2)$</td>
<td>$[y_1] \in t_1(n_1) \land y_1 = \text{len}(n_1) - 1 \land y_2 = \text{fst}(n_2) \land [y_3] \in t_2(n_2) \land y_3 = 1$</td>
<td>$[y_1] \in t_1(n_1) \land y_1 = \text{len}(n_1) - 1$</td>
<td>$[y_3] \in t_2(n_2) \land y_3 = 1$</td>
<td>$P_1(n_1, n_2)$</td>
<td>$P_2(n_1, n_2)$</td>
<td>$P_3(n_1)$</td>
<td>$P_4(n_2)$</td>
</tr>
<tr>
<td>$P^\text{ext}_3(n_1, n_2)$</td>
<td>$[y_1] \in t_1(n_1) \land y_1 = \text{len}(n_1) - 2 \land y_2 = \text{fst}(n_2) \land [y_3] \in t_2(n_2) \land y_3 = 1$</td>
<td>$[y_1] \in t_1(n_1) \land y_1 = \text{len}(n_1) - 2$</td>
<td>$[y_3] \in t_2(n_2) \land y_3 = 1$</td>
<td>$P_1(n_1, n_2)$</td>
<td>$P_2(n_1, n_2)$</td>
<td>$P_3(n_1)$</td>
<td>$P_4(n_2)$</td>
</tr>
<tr>
<td>$P^\text{ext}_4(n_2)$</td>
<td>$[y_1] \in t_1(n_1) \land y_1 = \text{len}(n_1) - 2 \land y_2 = \text{fst}(n_2) \land [y_3] \in t_2(n_2) \land y_3 = 1$</td>
<td>$[y_1] \in t_1(n_1) \land y_1 = \text{len}(n_1) - 2$</td>
<td>$[y_3] \in t_2(n_2) \land y_3 = 1$</td>
<td>$P_1(n_1, n_2)$</td>
<td>$P_2(n_1, n_2)$</td>
<td>$P_3(n_1)$</td>
<td>$P_4(n_2)$</td>
</tr>
</tbody>
</table>

Figure 6.28: Closure$^\text{ext}(P_{\text{fib}}, [n_1, n_2])$ and Closure(Closure$^\text{ext}(P_{\text{fib}}, [n_1, n_2])$)

For a given pattern $P$ and a set of vectors $M$, Closure$^\text{ext}(P, M)$ and Closure$^\text{ext}(P, M)$ are sets of patterns of size $|M|^y$, where $|M|$ is the total number of word variables in $M$ and $y$ is the number of position variables of the pattern $P$.

**Example 6.4.1.** Let $P_{\text{fib}}$ be the pattern $(y_1, y_2, y_3) \in t_1(w) \land y_1 < y_2 < y_3$. Suppose that $w$ is interpreted as the concatenation of two words denoted by $n_1$ and $n_2$. If we want to deduce a property of $w$ of the form $\forall y_1, y_2, y_3, P_{\text{fib}} \Rightarrow U$ from properties of $n_1$ and $n_2$ then, we have to use universally quantified implications having as guards the patterns given in the second column of Figure 6.28 (the first column gives the extended closure Closure$^\text{ext}(P_{\text{fib}}, [n_1, n_2])$):

- $P_1(n_1, n_2) ::= [y_1] \in t_1(n_1) \land y_1 = \text{len}(n_1) - 1 \land [y_3] \in t_2(n_2) \land y_3 = 1$, 
- $P_2(n_2) ::= [y_3] \in t_2(n_2) \land y_3 = 1$
- $P_3(n_1) ::= [y_1, y_2] \in t_1(n_1) \land y_1 < y_2 \land y_1 = \text{len}(n_1) - 2$
- $P_4(n_2) ::= [y_2, y_3] \in t_2(n_2) \land y_2 < y_3 \land y_2 = 1$, 
- $P_{\text{fib}}(n_1)$ and $P_{\text{fib}}(n_2)$.

These patterns characterize any three consecutive positions in the word denoted by $w$. Using $P_1$ and $P_2$, we capture the case when $y_1$ is the last position of $n_1$ and $y_2, y_3$ are the first two positions of $n_2$. Because the patterns characterize positions in the tail of words, $y_2$ does not appear explicitly. Its data can be characterized using the quantifier-free part. The case when $y_1$ and $y_2$ are the last positions of $n_1$ and $y_3$ is the first position of $n_2$ is considered using $P_2$. The pattern $P_3$ describes the case when $y_1, y_2$, and $y_3$ are the first three positions of $n_2$. Finally, $P_{\text{fib}}(n_1)$ and $P_{\text{fib}}(n_2)$ consider the situations when all the positions belong to the same word.

**Example 6.4.2.** Let $P_{eq} = y_1 \in t_1(r) \land y_2 \in t_1(m) \land y_1 = y_2$ and suppose that $r$ is the concatenation of two words denoted by $r_1$ and $r_2$ and $m$ is the concatenation of two
### 6.4. A $\mathbb{D}W$-DOMAIN WITH UNIVERSALLY QUANTIFIED FORMULAS

<table>
<thead>
<tr>
<th></th>
<th>Closure$^+$($P_{eq}, {r_1, r_2, [m_1, m_2]}$)</th>
<th>CLOSURE($P_{eq}, {r_1, r_2, [m_1, m_2]}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{cex}^1$</td>
<td>$[y_1] \in \text{tl}(r_1) \land [y_2] \in \text{tl}(m_1) \land y_1 = y_2$</td>
<td>$[y_1] \in \text{tl}(r_1) \land [y_2] \in \text{tl}(m_1) \land y_1 = y_2$</td>
</tr>
<tr>
<td>$P_{cex}^2$</td>
<td>$[y_1] \in \text{tl}(r_1) \land y_2 = \text{fst}(m_2) \land y_1 = \text{len}(m_1)$</td>
<td>$[y_1] \in \text{tl}(r_1) \land y_1 = \text{len}(m_1)$</td>
</tr>
<tr>
<td>$P_{cex}^3$</td>
<td>$[y_1] \in \text{tl}(r_1) \land [y_2] \in \text{tl}(m_2) \land y_1 = y_2 + \text{len}(m_1)$</td>
<td>$[y_1] \in \text{tl}(r_1) \land [y_2] \in \text{tl}(m_2) \land y_1 = y_2 + \text{len}(m_1)$</td>
</tr>
<tr>
<td>$P_{cex}^4$</td>
<td>$y_1 = \text{fst}(r_2) \land [y_2] \in \text{tl}(m_1) \land \text{len}(r_1) = y_2$</td>
<td>$[y_2] \in \text{tl}(m_1) \land \text{len}(r_1) = y_2$</td>
</tr>
<tr>
<td>$P_{cex}^5$</td>
<td>$y_1 = \text{fst}(r_2) \land y_2 = \text{fst}(m_2) \land \text{len}(r_1) = \text{len}(m_1)$</td>
<td>$\text{true}$</td>
</tr>
<tr>
<td>$P_{cex}^6$</td>
<td>$y_1 = \text{fst}(r_2) \land [y_2] \in \text{tl}(m_2) \land \text{len}(r_1) = y_2 + \text{len}(m_1)$</td>
<td>$[y_2] \in \text{tl}(m_2) \land \text{len}(r_1) = y_2 + \text{len}(m_1)$</td>
</tr>
<tr>
<td>$P_{cex}^7$</td>
<td>$[y_1] \in \text{tl}(r_2) \land [y_2] \in \text{tl}(m_1) \land y_1 + \text{len}(r_1) = y_2$</td>
<td>$[y_1] \in \text{tl}(r_2) \land [y_2] \in \text{tl}(m_1) \land y_1 + \text{len}(r_1) = y_2$</td>
</tr>
<tr>
<td>$P_{cex}^8$</td>
<td>$[y_1] \in \text{tl}(r_2) \land [y_2] \in \text{tl}(m_2) \land y_1 + \text{len}(r_1) = \text{len}(m_1)$</td>
<td>$[y_1] \in \text{tl}(r_2) \land [y_2] \in \text{tl}(m_2) \land y_1 + \text{len}(r_1) = \text{len}(m_1)$</td>
</tr>
</tbody>
</table>

Figure 6.29: Closure$^+$($P_{eq}, \{r_1, r_2, [m_1, m_2]\}$) and CLOSURE($P_{eq}, \{r_1, r_2, [m_1, m_2]\}$)
words denoted by \( m_1 \) and \( m_2 \). The extended closure and the closure of \( P_{eq} \) are given in Figure 6.29.

In the following, we describe the abstract transformers associated to a domain \( A_U \) parametrized by a set of patterns \( \mathcal{P} \) and a numerical abstract domain \( A_Z \). Let \( \varphi \) be an element in \( A_U \) as defined in Section 6.4.1.3.

### 6.4.4.3 The abstract transformer \( \text{concat}_p^\#(M, \varphi) \)

Let \( M = \{ (n_1, \mathbf{v}_1), \ldots, (n_p, \mathbf{v}_p) \} \) be a set of pairs between word variables \( (n_i, 1 \leq i \leq p) \) and vectors of word variables \( (\mathbf{v}_i, 1 \leq i \leq p) \). \( \text{concat}_p^\#(M, \varphi) \) adds to \( \varphi \) properties of \( n_i \) which represents the concatenation of the words denoted by the variables in \( \mathbf{v}_i \), for every \( 1 \leq i \leq p \). The free data word variables of \( \text{concat}_p^\#(M, \varphi) \) are \( \mathcal{V} \cup \{ n_1, \ldots, n_p \} \).

There are two main steps in the definition of this transformer:

- **generation of universally quantified formulas:** The data word variables \( n_1, \ldots, n_p \) from the input of \( \text{concat}_p^\# \) are distinct from the data word variables in \( \varphi \). Therefore, for every guard

\[
P(\mathbf{y}_1, \ldots, \mathbf{y}_q, w_1, \ldots, w_q, \mathbf{W}) \in \mathcal{P}(\mathcal{V} \cup \{ n_1, \ldots, n_p \})
\]

such that

\[
\{ w_1, \ldots, w_q \} \cap \{ n_1, \ldots, n_p \} \neq \emptyset
\]

(i.e., \( P \) describes the data of at least one word denoted by a variable in \( \{ n_1, \ldots, n_p \} \)), \( \text{concat}_p^\#(M, \varphi) \) contains a new universally-quantified formula

\[
\forall \mathbf{y}_1, \ldots, \mathbf{y}_q. P(\mathbf{y}_1, \ldots, \mathbf{y}_q, w_1, \ldots, w_q, \mathbf{W}) \Rightarrow U_p
\]

computed from formulas in \( \varphi \) over the data word variables in \( \mathcal{V} \). For simplicity, suppose that \( w_1 = n_1, \ldots, w_q = n_q \) (the extension to the general case is straightforward). Intuitively, we need to glue in every possible way that satisfies \( P \), the properties on the nodes in \( \mathbf{v}_i \) with \( 1 \leq i \leq q \). Actually, all the possible ways to glue them is given exactly by \( \text{Closure}_{\text{ext}}(P, \mathbf{v}_1, \ldots, \mathbf{v}_q) \). Therefore, the procedure that defines \( U_p \) has the following steps:

**Step 1:** compute \( \text{Closure}_{\text{ext}}(P, \mathbf{v}_1, \ldots, \mathbf{v}_q) \) according to the procedure defined in Section 6.4.4.2.

**Step 2:** for every pattern \( P'(\mathbf{y}) \in \text{Closure}_{\text{ext}}(P, \mathbf{v}_1, \ldots, \mathbf{v}_q) \) define \( U_{P'} \in A_Z \) as follows:

1. define \( U_{P'} \) as implied by \( \varphi \), that is: given \( \forall \mathbf{y}'. \sigma(P')(\mathbf{y}') \Rightarrow U_{\sigma(P')} \) a conjunct of \( \varphi \) (\( \sigma(P') \) is a pattern in \( \text{Closure}(P, \mathbf{v}_1, \ldots, \mathbf{v}_q) \) defined as in 6.4.5), \( U_{P'} \) is obtained from \( E \) and \( U_{\sigma(P')} \) such that \( \forall \mathbf{y}. P'(\mathbf{y}) \Rightarrow U_{P'} \) is a formula implied (in \( \text{gCSL} \)) by \( E \wedge \forall \mathbf{y}'. \sigma(P')(\mathbf{y}') \Rightarrow U_{\sigma(P')} \);
2. modify the constraints on the position variables in the \( U_{P'} \) formula resulting from the previous step, to reflect the fact that they represent positions on the word resulting after the concatenation,
3. because the first position of \( n_i \) corresponds to the first position of \( v_i^1 \), where \( v_i^1 \) is the first element of \( \mathbf{v}_i \), we substitute \( \text{hd}(v_i^1) \) with \( \text{hd}(n_i) \) in the formula obtained from the previous step;
6.4. A $\mathbb{W}$-DOMAIN WITH UNIVERSALLY QUANTIFIED FORMULAS

Step 3: the abstract value $U_P$ is obtained by taking the join of the numerical elements $U_{P'}$ obtained in Step 2, for any $P'(\mathbf{y}) \in \text{Closure}^{\text{ext}}(P, \mathbf{v}_1, \ldots, \mathbf{v}_q)$.

- update $E$ and the existing universally quantified formulas: for every $P(\mathbf{y}_1, \ldots, \mathbf{y}_q, w_1, \ldots, w_q, \mathbf{W}) \Rightarrow U_P$ in $\varphi$
  
  such that
  
  \[ \{w_1, \ldots, w_q\} \cap \{n_1, \ldots, n_p\} = \emptyset \]

  we modify $E$ and $U_P$ such that

  - $\text{len}(n_i)$ corresponds to the sum of the lengths of the words in $\mathbf{v}_i$, for all $1 \leq i \leq p$;
  - $\text{hd}(n_i)$ equals $\text{hd}(v^i_1)$ where $v^i_1$ is the first element of $\mathbf{v}_i$, for all $1 \leq i \leq p$.

We consider two running examples, introduced in Example 6.4.3 and Example 6.4.4, to illustrate each step of the definition of $\text{concat}^\#_P$.

Example 6.4.3. Let $\varphi_1$ be a formula in $\mathcal{A}_3$, parametrized by the polyhedra abstract domain and the patterns $P = \text{Closure}(P_{fb}, \{[n_1, n_2]\})$, given in Figure 6.28, where $P_{fb} = (y_1, y_2, y_3) \in \text{tl}(n) \land y_1 < y_2 < y_3$, defined as follows:

\[
\begin{align*}
\varphi_1 &= E_1 \land \phi_1^1 \land \phi_1^2, \text{ with } \\
E_1 &= \text{len}(n_1) = 5 \land \text{hd}(n_1) = 1 \land \text{len}(n_2) = 8 \land \text{len}(n_3) = 13 \\
\phi_1^1 &= \forall y_1, y_2, y_3. [y_1, y_2, y_3] \in \text{tl}(n_1) \Rightarrow n_1[y_1] + n_1[y_2] = n_1[y_3] \\
\phi_1^2 &= \forall y_1, y_2. [y_1, y_2] \in \text{tl}(n_1) \land y_1 < y_2 \land y_1 = \text{len}(n_1) - 2 \Rightarrow n_1[y_1] = 3 \land n_1[y_2] = 5.
\end{align*}
\]

We take as running example $\text{concat}^\#_P((\{n_1, [n_1, n_2]\}), \varphi_1)$.

Example 6.4.4. Let $\varphi_2$ be a formula in $\mathcal{A}_3$, parametrized by the polyhedra abstract domain and the patterns $P = \text{Closure}(P_{eq}, \{[r_1, r_2], [m_1, m_2]\})$, given in Figure 6.28, where $P_{eq} = y_1 \in \text{tl}(r) \land y_1 \in \text{tl}(m) \land y_1 = y_2$, defined as follows:

\[
\begin{align*}
\varphi_2 &= E_2 \land \phi_2^1 \land \phi_2^2 \land \phi_2^3 \land \phi_2^4, \text{ with } \\
E_2 &= \text{len}(r_1) < \text{len}(m_1) \land \text{len}(m_1) > 1 \land \text{len}(r_2) > 1 \land \\
&\quad \text{len}(r_1) + \text{len}(r_2) = \text{len}(m_1) + \text{len}(m_2) \\
\phi_2^1 &= \forall y_1, y_2. [y_1] \in \text{tl}(r_1) \land [y_2] \in \text{tl}(m_1) \land y_1 = y_2 \Rightarrow r_1[y_1] = m_1[y_2] \\
\phi_2^2 &= \forall y_1, y_2. [y_1] \in \text{tl}(r_1) \land y_2 = \text{len}(r_1) \Rightarrow m_1[y_2] = \text{hd}(r_2) \\
\phi_2^3 &= \forall y_1, y_2. [y_1] \in \text{tl}(r_1) \land [y_2] \in \text{tl}(m_1) \land y_1 + \text{len}(r_1) = y_2 \Rightarrow r_2[y_1] = m_1[y_2] \\
\phi_2^4 &= \forall y_1, y_2. [y_1] \in \text{tl}(r_2) \land \text{len}(r_1) = \text{len}(m_1) \Rightarrow r_2[y_1] = \text{hd}(m_2) \\
\phi_2^5 &= \forall y_1, y_2. [y_1] \in \text{tl}(r_2) \land [y_2] \in \text{tl}(m_2) \land \\
&\quad y_1 + \text{len}(r_1) = y_2 + \text{len}(m_1) \Rightarrow r_2[y_1] = m_2[y_2].
\end{align*}
\]

Figure 6.30 shows for each conjunct of $\varphi_2$ the part of the words $r_1$, $r_2$, $m_1$ and $m_2$ that it constraints. We consider as running example $\text{concat}^\#_P((\{r, [r_1, r_2]\}, (m, [m_1, m_2]\)), \varphi_1)$.
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

![Diagram of linked list](image)

Figure 6.30: \( \varphi_2 \)

**Generation of universally quantified formulas** We formalize the details of step 2 and step 3 since the computation of \( \text{Closure}^{\text{ext}} \) is given in Section 6.4.4.2.

**Step 2:** Let \( P(y_1, \ldots, y_n, n_1, \ldots, n_q, W) \in P(\mathcal{V} \cup \{n_1, \ldots, n_p\}) \), let \( P'(y) \in \text{Closure}^{\text{ext}}(P, v_1, \ldots, v_q) \) and let \( \sigma(P') \in \text{CLOSURE}(P, v_1, \ldots, v_q) \). Let \( U_{\sigma(P')} \) be the element of \( \mathcal{A}_Z \) such that:

- \( \forall y'. \sigma(P')(y') \implies U_{\sigma(P')} \) is a conjunct of \( \varphi \), if \( \sigma(P')(y') \) belongs to the set of patterns \( \mathcal{P} \) that parametrizes \( \mathcal{A}_U \).
- Let \( U_{\sigma(P')} = \top_Z \), otherwise.

(1) If \( U_{\sigma(P')} = \top_Z \) then \( U_{P'} = \top_Z \). Otherwise, notice that we can write \( P'(y) = \sigma(P')(y') \land P'' \), where \( P'' \) is a conjunction between atomic formulas of the form \( \text{len}(w) = ct \), where \( ct \) is an integer constant, and atomic formulas of the form \( y = \text{fst}(w) \) where \( y \) is not used in \( \sigma(P') \). The conjunct of \( \varphi \)

\[
\forall y'. \sigma(P')(y') \implies U_{\sigma(P')}
\]

implies that

\[
\forall y. (\sigma(P') \land P') \implies (U_{\sigma(P')} \land P'').
\]

The syntax of the universally quantified implications from \( \mathcal{A}_U \) does not allow terms like \( y = \text{fst}(w) \) in the right part. However, such a formula can be equivalently represented by the conjunction \( y = 0 \land w[y] = \text{hd}(w) \).

Formally, we define \( U_{P'} \) as follows:

- initially \( U_{P'} = U_{\sigma(P')} \)
- for every atomic formula of the form \( y = \text{fst}(w) \) in \( P' \), \( y \) represents the first position of \( w \) in \( U_P \). Therefore, we add two new dimensions to \( U_{P'} \) denoted \( y \) and \( w[y] \) such that \( y = 0 \) and \( w[y] = \text{hd}(w) \):
  
  \[
  U_{P'} = \text{add}^#(y, w[y], U_{P'});
  U_{P'} = \text{update}^#(w[y], \text{hd}(w), U_{P'})
  U_{P'} = \text{update}^#(y, 0, U_{P'}),
  \]

where \( \text{update}^# \) and \( \text{add}^# \) are abstract transformers in \( \mathcal{A}_Z \).

- we intersect the abstract value obtained previously with

1. the quantifier-free part \( E \) and
2. the formula $P'_1 \text{len}$ which is a conjunction between: (1) $P''$ if it is representable in $A_Z$, otherwise we consider only the constraints of the form $\text{len}(w) = ct \in \mathbb{Z}$ from $P''$, (2) $y < \text{len}(w)$, with $y \in y$, and (3) constraints of the form $\text{len}(w) > l$, for every $w$ such that $y \in tl(w)$ is an atomic formula of $P'$, for some $y$. $l$ is a constant corresponding to the number of $y$ in the vector $y \in tl(w)$ that are connected to their successor in $y$ by $<_1$ or $<_1$;

$$U_P = U_P \cap E \cap \exists E' P'_1 \text{len}.$$  \hspace{1cm} (6.4.6)

The intersection of $U_P$ with $E$ increases the precision of the resulting formula. Let us illustrate through an example.

**Example 6.4.5.** Consider that

$$P(y_1, y_2) ::= [y_1, y_2] \in tl(n) \land y_1 < y_2$$

and a call to $\text{concat}_P^{\#}(\{(n, [v_1, v_2, v_3])\}, \varphi)$ where $\varphi$ is an abstract value from $A_U$ parametrized by $P = \text{Closure}(P, [v_1, v_2, v_3])$ ($\varphi$ contains universally quantified formulas having as guards instantiations of all patterns in $\text{Closure}(P, [v_1, v_2, v_3])$). Let’s take a closer look to the following extended pattern:

$$P_1((y_1, y_2) ::= y_1 = \text{fst}(v_2) \land y_2 = \text{fst}(v_3) \land \text{len}(v_2) = 1,$$

$$P_2((y_1, y_2) ::= [y_1] \in tl(v_2) \land y_1 = \text{len}(v_2) - 1 \land y_2 = \text{fst}(v_3).$$

If the quantifier-free part $E$ of $\varphi$ implies that $\text{len}(v_2) \geq 1$ then, for each of these patterns, there exists at least one instance for the variable $y_1$ denoting a position in the word $v_2$. But, if $E$ implies that $\text{len}(v_2) = 1$ then there is no instance for the universal variable $y_1$ which satisfies the second pattern $P'_2 ([y_1] \in tl(v_2))$ implies that the length of $v_2$ is greater than or equal to $2$.

If we do not apply the assignment in (6.4.6) and $\forall y_1. \sigma(P'_2) \implies \top \mathbb{Z}$ is a subformula in $\varphi$ then $\text{concat}_P^{\#}(\{(n, [v_1, v_2, v_3])\}, \varphi)$ adds to $\varphi$ a property of the form $\forall y_1, y_2. P(y_1, y_2) \implies \top \mathbb{Z}$. This is correct but it might be too imprecise.

When applying this assignment, the abstract element implied by $P'_2$ is intersected with the quantifier-free part $E$ and the length constraints $P'_2 \text{len}$ imposed by the guard $P'_2$. In this case, we obtain that $U_P = U_{\sigma(P'_2)} \cap E \text{len}(v_2) > 1 \cap \exists E' \text{len}(v_2) = 1$ which is equivalent to $\top \mathbb{Z}$. Therefore, the value associated with a pattern that has no instances under the constraints in $E$ has no effect on the final value associated with $P$.

**Example 6.4.6.** We define the formulas implied by the patterns in $\text{Closure}^{\text{ext}}(\{(n, [n_1, n_2])\}, \varphi_1)$. Because $E_1$ implies that $\text{len}(n_2) = 1$ then the $P^{\text{ext}}_1(n_1, n_2)_2 \text{len}$ and $P^{\text{ext}}_3(n_1, n_2)_2 \text{len}$ and $P^{\text{ext}}_4(n_2)_2 \text{len}$ and $P^{\text{ext}}_5(n_2)_2 \text{len}$ imply $\top \mathbb{Z}$. For the other two patterns we obtain that

$$\forall y_1, y_2, y_3. P^{\text{ext}}_3(n_1, n_2) \implies n_1[y_1] = 3 \land n_1[y_2] = 5 \land$$

$$y_3 = 0 \land n_2[y_3] = \text{hd}(n_2) \land \text{len}(n_1) > 2 \land \text{len}(n_1) > 2 \land \text{len}(n_1) = 5 \land \text{hd}(n_2) = 8 \land \text{len}(n_2) = 1 \land \text{hd}(n_1) = 1 \land \text{len}(n_1) = 5 \land$$

$$E$$

The formula corresponding to $P^{\text{fib}}_5(n_2)$ is $\phi^2_1$ from Example 6.4.3.
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

Example 6.4.7. We define the formulas implied by the patterns in \( \text{Closure}^{\text{ext}} \{ \{r, [v_1, v_2]\}, \{m, [m_1, m_2]\}\} \). Due to the intersection with \( E_2 \) and more precisely because \( E_2 \) implies that \( \text{len}(r_1) < \text{len}(m_1) \) the guards \( P^{\text{ext}}_2, P^{\text{ext}}_3, P^{\text{ext}}_5, P^{\text{ext}}_6 \) imply \( \bot \). For example, to compute the formula implied by \( P^{\text{ext}}_2 \) we have to intersect \( P^{\text{ext}}_{2, \text{len}} = y_1 < \text{len}(r_1) \land y_1 = \text{len}(m_1) \) with \( E_2 \) (\( E_2 \) implies \( \text{len}(r_1) < \text{len}(m_1) \)). Since this intersection is \( \bot \) then \( \forall y_1, y_2, P^{\text{ext}}_2 \implies \bot \). We will define the formula implied by \( P^{\exp}_7 \), the others are defined similarly. The pattern \( P^{\exp}_7 \) corresponds to \( P_7 \) which is the guard of \( \phi^3_2 \). Then,

\[
\forall y_1, y_2, P^{\exp}_7 \implies r_2[y_1] = m_1[y_2] \land \exists Z E_2 \land Z
\]

that is

\[
\forall y_1, y_2, P^{\exp}_7 \implies r_2[y_1] = m_1[y_2] \land y_1 + \text{len}(r_1) = y_2 \quad (6.4.8)
\]

(2) Let \( U_{P'} \) be the formula implied by \( P'(y) \in \text{Closure}^{\text{ext}}(P, v_1, \ldots, v_q) \) built so far. Next,

- every term \( v[y] \) with \( v \) in some \( v_j \), \( 1 \leq j \leq q \), is substituted with \( n_j[y] \), because any position of \( v \) (if \( y \) appears in a term \( v[y] \) then either \( y \in tl(v) \) or \( y = \text{fst}(v) \)) is a position of \( n_j \)

\[
U_{P'} = U_{P'}[v[y] \mapsto n_j[y] \mid v \in v_j \text{ and } 1 \leq j \leq q]
\]

- let \( v_j = [v_j^1, \ldots, v_j^q] \) with \( q^j \leq 1 \), for every \( 1 \leq j \leq q \). If \( l \) is a position of \( v_j^l \in V_j \) with \( 1 < i \leq q^j \), then \( l + \text{len}(v_j^l) + \ldots + \text{len}(v_j^{i-1}) \) is the corresponding position in the concatenation denoted by \( n_j \). Therefore, every \( y \) such that \( P' \) contains an atomic formula of the form \( y \in tl(v_j^i) \) or \( y = \text{fst}(v_j^i) \) with \( 1 < i \leq q^j \) is substituted with \( y - (\text{len}(v_j^i) + \ldots + \text{len}(v_j^{i-1})) \):

\[
U_{P'} = U_{P'}[y \mapsto y - (\text{len}(v_j^1) + \ldots + \text{len}(v_j^{i-1})) \mid y \in tl(v_j^i) \text{ or } y = \text{fst}(v_j^i) \text{ in } P']
\]

(3) Finally, the first position of \( n_j \) corresponds to the first position of \( v_j^1 \) and consequently, we substitute in \( U_{P'} \), \( \text{hd}(v_j^1) \) with \( \text{hd}(n_j) \).

Example 6.4.8. The formula \( [6.4.7] \) is transformed into the following formula (we omit several conjuncts irrelevant for the transformation) that characterizes data associated to position in \( n \), in the concatenation of \( n_1 \) and \( n_2 \):

\[
\forall y_1, y_2, y_3, P^{\text{ext}}_3 \implies n[y_1] = 3 \land n[y_2] = 5 \land y_3 = \text{len}(n_1) + \text{len}(n_2) \land n[y_3] = \text{hd}(n_2) \land \text{hd}(n_2) = 8.
\]

Example 6.4.9. The formula \( [6.4.8] \) is transformed into the following formula (we omit several conjuncts irrelevant for the transformation) that characterizes position in \( r \) and \( m \), the concatenation of \( r_1 \) and \( r_2 \), respectively \( m_1 \) and \( m_2 \):

\[
\forall y_1, y_2, P^{\text{ext}}_7 \implies r[y_1] = m[y_2] \land y_1 - \text{len}(r_1) + \text{len}(r_1) = y_2.
\]

208
Then, we define Example 6.4.10. During the computation of \( \text{concat}_P^\# \left( \{(n, [n_1, n_2])\}, \varphi_1 \right) \), we obtain that all the extended patterns imply either \( \bot \mathbb{Z} \) or a property stronger than \( n[y_1] + n[y_2] = n[y_3] \) (it might contain several other constraints on the length of the segments). Therefore, we obtain that the concatenation of \( n_1 \) with \( n_2 \) preserves the Fibonacci relation between values of consecutive positions:

\[
\forall y_1, y_2, y_3. P_{\text{fib}}(n) \implies n[y_1] + n[y_2] = n[y_3].
\]

Example 6.4.11. During the computation of \( \text{concat}_P^\# \left( \{(r, [r_1, r_2]), (m, [m_1, m_2])\}, \varphi_2 \right) \), we obtain that all the extended patterns imply either \( \bot \mathbb{Z} \) or a property stronger then \( r[y_1] = m[y_2] \land y_1 = y_2 \) (it might contain several other constraints on the length of the segments). Therefore, we obtain that by concatenating \( m_1 \) with \( m_2 \) and \( r_1 \) with \( r_2 \) we obtain two equal words:

\[
\forall y_1, y_2. P_{\text{eq}}(r, m) \implies r[y_1] = m[y_2] \land y_1 = y_2.
\]

**Update \( E \) and the existing universally quantified formulas.** To update the length constraints, we assign in all the abstract values from \( A_\mathbb{Z} \), \( \text{len}(n_i) \) to \( \text{len}(v_j) + \ldots + \text{len}(v_j^q) \), for every \( 1 \leq j \leq q \):

\[
E := \text{update}^\#(\text{len}[n_j], \text{len}[v_j] + \ldots + \text{len}[v_j^q], E), \quad \text{for every } P \in \mathbb{P}(V \cup \{n_1, \ldots, n_p\})
\]

\[
U_P := \text{update}^\#(\text{len}[n_j], \text{len}[v_j] + \ldots + \text{len}[v_j^q], U_P).
\]

Also, we update the value of \( \text{hd}(n_i) \) for every \( 1 \leq i \leq q \):

\[
E := \text{update}^\#(\text{hd}[n_j], \text{hd}[v_j] + \ldots + \text{hd}[v_j^q], E), \quad \text{for every } P \in \mathbb{P}(V \cup \{n_1, \ldots, n_p\})
\]

\[
U_P := \text{update}^\#(\text{hd}[n_j], \text{hd}[v_j] + \ldots + \text{hd}[v_j^q], U_P).
\]

**Theorem 6.4.3.** Let \( A_{\mathbb{U}} \) be as above such that the numerical abstract domain \( A_{\mathbb{Z}} \) has an exact meet operator. Then, the transformer \( \text{concat}_P^\# \) is sound.

**Proof.** Let \( M = \{(n_1, v_1), \ldots, (n_p, v_p)\} \) be a set of pairs between word variables \( (n_i, 1 \leq i \leq p) \) and vectors of word variables \( (v_i, 1 \leq i \leq p) \) and let \( \varphi \) be an abstract value from \( A_{\mathbb{U}}(\mathbb{P}) \).

We have to prove that

\[
\text{concat}(M, \gamma^\mathbb{U}(\varphi)) \subseteq \text{concat}_P^\#(M, \varphi). \tag{6.4.9}
\]

The formula \( \text{concat}_P^\#(M, \varphi) \) is obtained from \( \varphi \) by (1) adding to the quantifier-free part equalities of the form \( \text{len}(n_i) = \Sigma_{v \in V_i} \text{len}(v) \) and \( \text{hd}(n_i) = \text{hd}(v_i^1) \), where \( v_i^1 \) is the first element of \( v_i \), for any \( 1 \leq i \leq p \) and (2) adding universally-quantified formulas over
the data word variables \( n_1, \ldots, n_p \) (which do not appear in \( \varphi \)). Recall that the concrete transformer \( \text{concat} \) defines the word denoted by \( n_i \) as the concatenation of the words denoted by variables in \( v_i \), for any \( 1 \leq i \leq p \). Clearly, the equalities added to the quantifier-free part of \( \varphi \) are implied by the transformation performed by \( \text{concat} \). In the following, we will prove that the same holds for the universally-quantified formulas added to \( \varphi \).

Let \( P(y_1, \ldots, y_q, n_1, \ldots, n_q, W) \) be a pattern from \( \mathbb{P}(\mathcal{V} \cup \{n_1, \ldots, n_p\}) \), where \( \mathcal{V} \) is the set of data word variables from \( \varphi \) (for simplicity, we have considered patterns that describe only positions of words from \( \{n_1, \ldots, n_p\} \); the case of patterns that describe relations between positions of words from \( \mathcal{V} \) and positions of words from \( \{n_1, \ldots, n_p\} \) is similar). Also, let

\[
\psi_P := \forall y_1, \ldots, y_q. P(y_1, \ldots, y_q, n_1, \ldots, n_q, W) \implies \bigcup_{P' \in \text{Closure}^\text{ext}(P, v_1, \ldots, v_q)} \mathbb{Z} U_{P'}
\]

be the formula in \( \text{concat}^\#_P(M, \varphi) \) corresponding to the pattern \( P \). We prove that

\[
\text{concat}(M, \gamma^U(\varphi)) \subseteq \gamma^U(\psi_P).
\]  \hspace{1cm} (6.4.10)

The starting point is the fact that \( \text{Closure}^\text{ext}(P, v_1, \ldots, v_q) \) has the following property:

\[
\left( \bigvee_{P' \in \text{Closure}^\text{ext}(P, v_1, \ldots, v_q)} \mathbb{P}[\xi] \land \bigwedge_{1 \leq i \leq q} \text{len}(n_i) = \Sigma_{v \in v_i} \text{len}(v) \right) \iff P(y_1, \ldots, y_q, n_1, \ldots, n_q, W),
\]  \hspace{1cm} (6.4.11)

where, (1) for any pattern \( P, \overline{P} \) is obtained from \( P \) by replacing every atomic formula \( y \in \omega \) with \( \bigwedge_{y \in v} 1 \leq y \leq \text{len}(\omega) \), (2) \( \xi \) is a substitution that replaces every \( y \) in \( P' \) representing a position of \( v'_j \), the \( j \)-th element of \( v_i \), with \( y - \text{len}(v'_1) + \ldots + \text{len}(v'_{j-1}) \), and (3) \( \iff \) is the equivalence in Presburger arithmetics (the terms \( \text{len}(n) \) are treated as integer variables).

For every \( P' \in \text{Closure}^\text{ext}(P, v_1, \ldots, v_q) \), the definition of \( U_{P'} \) from 6.4.6 implies that

\[
\gamma^U(\varphi) \subseteq \gamma^U(\forall y_{P'}. P' \implies U_{P'}),
\]

where \( y_{P'} \) is the set of position variables from \( P' \) (this property uses the fact that the meet operator of \( A_\mathbb{Z} \) is exact). Then, from (6.4.11) and the fact that \( \sqcup^U \) is an over-approximation of the disjunction between formulas (i.e., \( \gamma^U(\varphi_1) \cup \gamma^U(\varphi_2) \subseteq \gamma(\varphi_1 \cup^U \varphi_2) \), for any \( \varphi_1 \) and \( \varphi_2 \) in \( A_\mathbb{U} \)) we obtain that (6.4.10) holds.

### 6.4.4.4 Abstract transformer \( \text{split}^\#_P(n, v, \varphi) \)

The transformer \( \text{split}^\#_P(n, v, \varphi) \) is an approximation for the splitting of the word denoted by \( n \) into its head and its tail; the head is assigned to the variable \( n \) and the tail to \( v \). For presentation reasons, we add two fresh word variables \( u \) and \( v \) and we constrain \( u \) to represent a word of length one equal with the head of \( n \) and \( v \) to represent a copy of the tail of \( n \). We introduce formulas that describe \( u \) and \( v \) starting from the ones that describe \( n \).

Notice that \( \text{split}^\#_P \) corresponds to a transformation which is the reverse of the concatenation \( n = u \circ v \), that is \( n \) is replaced by \( u \) and \( v \). Before updating \( n \) we want to infer as much information as possible about the resulting words \( u \) and \( v \). Therefore,
for every guard $P$ over $n$, that appears in some universally quantified implication of $\varphi$, we consider all the possible ways in which we can distribute the universally quantified variables denoting positions in $n$ over the words $u$ and $v$. If the position variables of $P$ are associated to a set of words $n, w_2, \ldots, w_{q-1}$ then the set of extended patterns $\text{Closure}^\text{ext}(P, [u,v], [w_2], \ldots, [w_{q-1}])$ defines exactly all these situations. Actually, there is no need to consider all the patterns in this extended closure. It is enough to consider only those patterns where position variables are not mapped to $u$. This follows from the fact that the tail of $u$ is empty ($u$ represents words of length 1).

The procedure that defines the result of $\text{split}^\#_P(n,v,\varphi)$ consists in:

1. **generating new universally quantified formulas:** for every guard in over $n$, $P(y_1, \ldots, y_q, n, w_2, \ldots, w_{q-1}, W) \in \mathbb{P}(V)$ such that

   $$\forall y_1 \ldots \forall y_q \cdot P(y_1, \ldots, y_q, n, w_2, \ldots, w_{q-1}, W) \implies U_P,$$

   is a conjunct of $\varphi$:

   - **Step 1:** for every $P'(y') \in \text{Closure}^\text{ext}(P, [u,v], [w_2], \ldots, [w_{q-1}])$, we define a numerical abstract value $U_{P'}$ such that $\varphi$ implies $\forall y'. P'(y') \implies U_{P'}$ (or, equivalently, the concretization of $\varphi$ is included into the concretization of $\forall y'. P'(y') \implies U_{P'}$);

   - **Step 2:** we add constraints on $\text{hd}(v)$ to $E$ using the formulas $U_{P'}$ computed in Step 1;

   - **Step 3:** we add new universally quantified formulas that describe $\text{tl}(v)$ using the formulas $U_{P'}$ computed in Step 1; we add only the universal formulas of the form

     $$\forall y'. \sigma(P') \implies U_{P'},$$

     where $\sigma(P') \in \mathbb{P}(V \cup \{u,v\})$ is a guard obtained from a parameter of $A_U$ ($\sigma(P')$ is defined as in (6.4.5)).

2. **updating $E$ and $U_P$ for every guard $P(y)$ in the current formula** finally, the result of $\text{split}^\#_P(n,v,\varphi)$ is obtained by:

   1. updating $\text{hd}(u)$ to be equal with $\text{hd}(v)$, $\text{len}(u)$ to be equal with 1, and $\text{len}(v)$ to be equal with $\text{len}(n) - 1$, in $E$ and $U_P$, for every guard $P(y)$ in the current formula;

   2. projecting out the data word variable $n$;

   3. replacing all occurrences of $u$ with $n$.

In the following we detail only the step the generates universally quantified variables and discovers the constraints on $\text{hd}(v)$ using $\text{Closure}^\text{ext}$. We consider the running examples, for the definition the abstract transformer $\text{split}^\#_P(n,v,\varphi)$:

**Example 6.4.12.** Let $\varphi$ be a formula in $A_U$, parametrized by the polyhedra abstract domain and the patterns $\mathbb{P} = \text{Closure}(P_{eq}, \{[r_1,r_2],[m_1,m_2]\})$, given in Figure 6.28, where $P_{eq} = [y_1] \in \text{tl}(r) \land [y_2] \in \text{tl}(m) \land y_1 = y_2$, defined as follows:

$$\varphi = E \land \phi, \text{ with }$$

$$E = \text{len}(n) > 5 \land \text{hd}(r) = \text{hd}(m) \land \text{len}(r) = \text{len}(m)$$

$$\phi = \forall y_1, y_2. [y_1] \in \text{tl}(r) \land [y_2] \in \text{tl}(m) \land y_1 = y_2 \implies r[y_1] = m[y_2]$$
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

Let \( P(y_1, \ldots, y_q, n, w_2, \ldots, w_{q-1}, W) \in \mathbb{P}(V) \) such that
\[
\forall y_1 \ldots \forall y_q \cdot P(y_1, \ldots, y_q, n, w_2, \ldots, w_{q-1}, W) \implies U_P,
\]
is a conjunct of \( \varphi \), and let \( P'(y') \in \text{Closure}^{\text{ext}}(P, [u, v], [w_2], \ldots, [w_{q-1}]) \).

**Step 1:** Computing \( \overline{U}_{P'} \) The formula \( \overline{U}_{P'} \) is obtained from \( U_P \) by:

1. first, define the length of \( u \) to be equal to one:
\[
\overline{U}_{P'} = \text{update}^\#(\text{len}(u), 1, \text{add}^\#(\text{len}(u), \text{hd}(u), \text{len}(v), \text{hd}(v), U_P));
\]
2. for every position variable \( y \), if \( y = \text{fst}(v) \) is an atomic formula of \( P' \) then \( \text{hd}(v) \) equals \( n[y] \) and \( y \) equals \( \text{len}(u) \). Consequently, we update \( \overline{U}_{P'} \) as follows:
\[
\overline{U}_{P'} = \text{update}^\#(y, \text{len}(u), \overline{U}_{P'}),
\]
\[
\overline{U}_{P'} = \text{update}^\#(n[y], \text{hd}(v), \overline{U}_{P'})
\]
3. for every position variable \( y \), if \( y \in \text{tl}(v) \) is an atomic formula of \( P' \) then \( y \) is substituted with \( y - 1 \) (in \( U_P \), \( y \) denotes a position of \( n \) and any position \( p \) of the word denoted by \( n \) corresponds to the position \( p - 1 \) of the word denoted by \( v \)); also, \( n[y] \) is substituted with \( v[y] \) (a data value of \( n \) corresponds to a data value of \( v \));

**Example 6.4.13.** Let us consider \( \text{split}^\#(r, \varphi) \) where \( \varphi \) is introduced in Example 6.4.12.

Notice that is useless to consider the patterns where \( \text{len}(u) > 1 \). Therefore, we define

- \( P^\text{ext}_4 \implies \overline{U}_{P^\text{ext}_4} \), where \( P^\text{ext}_4(r_1, r_2) = y_1 = \text{fst}(r_2) \land y_2 \in \text{tl}(m) \land y_2 = \text{len}(r_1) \)
  and

\[
\overline{U}_{P^\text{ext}_4} = \text{len}(r_1) = 1 \land y_1 = \text{len}(r_1) \land y_2 = \text{len}(r_1) \land r_1[y_1] = \text{hd}(r_2) \land r_1[y_1] = m[y_2]
\]

- \( P^\text{ext}_7 \implies \overline{U}_{P^\text{ext}_7} \), where \( P^\text{ext}_7(r_2, m) = \forall y_1, y_2. [y_1] \in \text{tl}(r_2) \land [y_2] \in \text{tl}(m) \land y_1 = y_2 \)
  and

\[
\overline{U}_{P^\text{ext}_7} = \text{len}(r_1) = 1 \land r_2[y_1] = m[y_2] \land y_1 + \text{len}(r_1) = y_2
\]

**Step 2:** Adding constraints on \( \text{hd}(v) \) First, notice that \( \sigma(P') \) with \( P' \) in the extended closure of some pattern \( P \) as above might be a pattern with no position variables. In this case, the formula \( \sigma(P') \implies U_{\sigma(P')} \) strengthens the quantifier-free part. If we add the implication as it is then we introduce disjunctions in the formulas, which most abstract numerical domains can not represent precisely. Therefore, let \( E' \) denote the updated quantifier-free part. Initially, \( E' = E \). For every \( P' \) such that \( \sigma(P') \) is a pattern with no position variables, if \( E \sqsubseteq \overline{U}_{P'} \) then we intersect \( E' \) with \( U_{P'} \), that is, \( E' = E' \cap \overline{U}_{P'} \).

Another situation when we can infer constraints on \( \text{hd}(v) \) from formulas of the form \( \forall y' \). \( P' \implies \overline{U}_{P'} \) is when \( P' \) contains an atomic formula of the form \( y = \text{fst}(v) \). As described above, in this case, \( \overline{U}_{P'} \) might constraint \( \text{hd}(v) \). Therefore, if \( E' \sqsubseteq \overline{U}_{P'} \), where \( P'_L \) are the length constrains in \( P' \), we strengthen \( E' \) with the formula obtained from \( \overline{U}_{P'} \) by forgetting all the constraints on the position variables and the data associated to them:

\[
E' = E' \cap \overline{U}_{P'} \upharpoonright \{y, v[y] \mid y \in \text{tl}(v) \text{ atomic formula in } P'\}.
\]
Most numerical abstract domains consider an over-approximation for the entailment relation between a numerical abstract object and a linear constraint (a conjunction of linear constraints). Even if they don’t, we can do the test above using some SMT solver.

**Example 6.4.14.** For the guard \( P_{4}^{\text{ext}} \), \( \sigma(P_{4}^{\text{ext}}) \) implies no length constraints that do not involve any position variables, so \( E \) is intersected with \( U_{P_{4}^{\text{ext}}} \). Similarly for the other patterns, there are interesting constraints that are added to \( E \).

**Step 3: Adding constraints on \( \text{tl}(v) \)** The universal formulas that describe \( \text{tl}(v) \) are computed as follows. Initially, for every guard \( \sigma(P') \) in \( P(\mathcal{V} \cup \{v\}) \), where \( P' \in \text{Closure}^{\text{ext}}(P, [u, v], [w_{2}],..., [w_{q-1}]) \) and \( P(y_{1}, ..., y_{q}, n, w_{1},..., w_{q-1}, W) \in \mathcal{P}(\mathcal{V}) \), we consider a universal formula of the form \( \forall v'. \sigma(P') \implies \top \) (\( v' \) are the position variables in \( \sigma(P') \)).

Then, for every \( P' \in \text{Closure}^{\text{ext}}(P, [u, v], [w_{2}],..., [w_{q-1}]) \) with \( P'(y_{1}, ..., y_{q}, n, w_{1},..., w_{q-1}, W) \in \mathcal{P}(\mathcal{V}) \), the righthand side of the implication \( \forall y''. \sigma(P') \implies U_{\sigma(P')} \) is strengthened with \( U_{P_{r}} \), that is, we apply

\[
U_{\sigma(P')} = U_{\sigma(P') \cap \neg \exists} U_{P_{r}}.
\]

**Example 6.4.15.** After applying the modification on \( \text{tl}(r_{2}) \) we obtain the following universally quantified formulas:

\[
\phi_{1} = \forall y_{2}. [y_{2}] \in \text{tl}(m) \land y_{2} = \text{len}(r_{1}) \implies m[y_{2}] = \text{hd}(r_{2}) \text{ obtained from } P_{4}^{\text{ext}}
\]

\[
\phi_{2} = \forall y_{1}, y_{2}. [y_{1}] \in \text{tl}(r_{2}) \land [y_{2}] \in \text{tl}(m) \land y_{2} = y_{1} + \text{len}(r_{1}) \implies m[y_{2}] = r_{2}[y_{1}] \text{ obtained from } P_{7}^{\text{ext}}.
\]

The formula computed by \( \text{split}_{P}^{\#}(\{(r, [r_{1}, r_{2}])\}, \varphi) \) is

\[
\text{len}(r_{1}) + \text{len}(r_{2}) = \text{len}(m) \land \text{len}(r_{1}) = 1 \land \text{hd}(r_{1}) = \text{hd}(m) \land \phi_{1} \land \phi_{2}.
\]

**Theorem 6.4.4.** Let \( \mathcal{A}_{\mathcal{V}} \) be as above such that the numerical abstract domain \( \mathcal{A}_{\mathcal{V}} \) has an exact meet operator. Then, the transformer \( \text{split}_{P}^{\#} \) is sound.

**Proof.** Because \( \text{split}_{P}^{\#} \) corresponds to a transformation which is the reverse of the concatenation from \( \text{concat}_{P}^{\#} \), the proof is similar to the one of Theorem 6.4.3. Again, we use the property of \( \text{Closure}^{\text{ext}} \) from [6.4.11]. \( \square \)

### 6.4.4.5 Projection, Singleton and Update abstract transformers

In the following we formally define all the others abstract transformers. Their correctness is stated at the end in Theorem 6.4.5.

**The projection operator** \( \text{proj}_{P}^{\#}(D, N, \varphi) \) Given a set of data variables \( D \) and a set of word variables \( N \), the result of \( \text{proj}_{P}^{\#}(D, N, \varphi) \) is an abstract element that does not contain any occurrences of the variables in \( D \cup N \) and its concretization is an over-approximation of the concretization of \( \varphi \) when considering only words and integers denoted by variables in \( \mathcal{V} \setminus N \) and \( D\text{Var} \setminus D \).

We start by projecting out from \( E \) (using the projection operator in \( \mathcal{A}_{\mathcal{V}} \)) (1) the variables \( \text{len}(w) \) and \( \text{hd}(w) \) corresponding to the length and the first symbol of \( w \), for any \( w \in N \), and (2) the variables \( d \in D \). Then, the universal quantified formulas are modified as follows:
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

1. for any universal sub-formula of \( \varphi \) of the form

\[
\forall y_1 \ldots \forall y_q. \left( \bigwedge_{1 \leq i \leq q} P^R_i(y_i, w_i) \land P_L(y_1^1, \ldots, y_q^1, W) \right) \implies U_P, \tag{6.4.12}
\]

(1) we project out from the right part of the implication, \( U_P \), the terms built over the variables in \( D \cup N \) (\( \text{len}(w), \text{hd}(w) \)) with \( w \in N \) and \( d \in D \) and (2) we apply the meet operator in \( A_{\geq} \) between the obtained abstract value and the quantifier-free part of \( \varphi \). Formally:

\[
U_P := E \sqcap \uparrow (U_P \uparrow (\{d \mid d \in D\} \cup \{\text{len}(w), \text{hd}(w) \mid w \in N\})).
\]

2. for any universal sub-formula of \( \varphi \) as in (6.4.12) such that \( w_i \notin N \), for all \( 1 \leq i \leq q \), and \( W \cap N = \{w\} \), let \( P' \) be the pattern

\[
\bigwedge_{1 \leq i \leq q} P^R_i(y_i, w_i) \land P_L(y_1^1, \ldots, y_q^1, W) \uparrow \text{len}(w),
\]

where \( P_L(y_1^1, \ldots, y_q^1, W) \uparrow \text{len}(w) \) is the quantifier-free Presburger formula corresponding to \( \exists \text{len}(w), P_L(y_1^1, \ldots, y_q^1, W) \). If there exists \( P'' \in \mathbb{P}(\mathbb{V} \setminus N) \) with the same number of data words variables as \( P' \) and the same number of position variables on each data word such that \( P'' \implies P' \) then we modify the universal formula corresponding to \( P'' \) by

\[
U_{P''} := U_P \sqcap \uparrow U_P
\]

3. for any universal sub-formula of \( \varphi \) as in (6.4.12) such that (1) \( w_j \in N \), for some \( 1 \leq j \leq q \), (2) \( w_i \notin N \), for all \( i \neq j \), and (3) \( W \cap N = \emptyset \), let \( P' \) be the pattern

\[
\bigwedge_{1 \leq i \leq q, i \neq j} P^R_i(y_i, w_i) \land P_L(y_1^1, \ldots, y_q^1, W) \downarrow \{\text{len}(w_j), y_j^1\}.
\]

If there exists \( P'' \in \mathbb{P}(\mathbb{V} \setminus N) \) with the same number of data words variables as \( P' \) and the same number of position variables on each data word such that \( P'' \implies P' \) then we modify the universal formula corresponding to \( P'' \) by

\[
U_{P''} = U_{P''} \sqcap \uparrow (U_P \uparrow (\{y_j \cup \{w_j[y] \mid y \in y_j\} \cup \{\text{len}(w_j), \text{hd}(w_j)\}))).
\]

4. universal sub-formulas of \( \varphi \) as in (6.4.12) such that more than one variable in \( \{w_1, \ldots, w_q\} \) belongs to \( N \) and \( W \) contains more than one variable from \( N \) are modified in a similar manner.

The abstract transformer \( \text{selectSglt}_P^n(n, \varphi) \) Given \( n \) a data word variable, the output of \( \text{selectSglt}_P^n(n, \varphi) \) is

\[
E' \land \bigwedge_{P \in \mathbb{P}(\mathbb{V})} \forall y_1 \ldots \forall y_q. P(y_1, \ldots, y_q, w_1, \ldots, w_q, W) \implies U'_P,
\]

where (1) \( E' \) is obtained from \( E \) by adding two dimensions for \( \text{len}(n) \) and \( \text{hd}(n) \) and by applying \( E' = E \sqcap \uparrow \text{len}(n) = 1 \), and (2) for every \( P \in \mathbb{P}(\mathbb{V}) \), the corresponding \( U'_P \) is obtained from \( U_P \) by adding two dimensions for \( \text{len}(n) \) and \( \text{hd}(n) \) and by applying \( U'_P = U_P \sqcap \uparrow \text{len}(n) = 1 \).
6.4. A $\mathbb{D}\mathbb{W}$-DOMAIN WITH UNIVERSALLY QUANTIFIED FORMULAS

The abstract transformer $\text{sglt}_E^\#(n, \varphi)$ Given $n$ a data word variable, the output of $\text{sglt}_E^\#(n, \varphi)$ is

$$E' \land \bigwedge_{P \in \mathbb{P}(V)} \forall y_1 \ldots \forall y_q. P(y_1, \ldots, y_q, w_1, \ldots, w_q, W) \Rightarrow U'_P,$$

where (1) $E'$ is obtained from $E$ by applying $E' = E \land \# \text{len}(n) > 1$, and (2) for every $P \in \mathbb{P}(V)$, the corresponding $U'_P$ is obtained from $U_P$ by applying $U'_P = U_P \land \# \text{len}(n) > 1$.

The abstract transformer $\text{selectNonSgl}_{E}^\#(n, \varphi)$ Given $n$ a data word variable, the output of $\text{selectSgl}_{E}^\#(n, \varphi)$ is

$$E' \land \bigwedge_{P \in \mathbb{P}(V)} \forall y_1 \ldots \forall y_q. P(y_1, \ldots, y_q, w_1, \ldots, w_q, W) \Rightarrow U'_P,$$

where (1) $E'$ is obtained from $E$ by applying $E' = E \land \# \text{len}(n) = 1$, and (2) for every $P \in \mathbb{P}(V)$, the corresponding $U'_P$ is obtained from $U_P$ by applying $U'_P = U_P \land \# \text{len}(n) = 1$.

The abstract transformer $\text{updFirst}_E^\#(d, n, \varphi)$ Given $d$ a data variable and $n$ a data word variable, the abstract value $\text{updFirst}_E^\#(d, n, \varphi)$ is obtained from $\varphi$ by replacing the quantifier-free part with

$$\text{update}(E, \text{hd}(n) = d)$$

and for each pattern $P \in \mathbb{P}(V)$, the corresponding abstract element $U_P$ from $A_Z$ is

$$\text{update}(U_P, \text{hd}(n) = d),$$

where $\text{update}$ is the abstract transformer in $A_Z$ corresponding to assignments between integer variables.

The abstract transformer $\text{updDvar}_E^\#(d, n, \varphi)$ Given $d$ a data variable and $n$ a data word variable, the abstract value $\text{updDvar}_E^\#(d, n, \varphi)$ is obtained from $\varphi$ by replacing the quantifier-free part with

$$\text{update}(E, d = \text{hd}(n))$$

and for each pattern $P \in \mathbb{P}(V)$, the corresponding abstract element $U_P$ from $A_Z$ is

$$\text{update}(U_P, d = \text{hd}(n)).$$

The abstract transformer $\text{Eq}_E^\#(V, V', D, D', \varphi)$ Given $V = n_1 \ldots n_s$ and $V' = n'_1 \ldots n'_s$ vectors of data word variables of equal length and $D = d_1 \ldots d_t$ and $D' = d'_1 \ldots d'_t$ vectors of data variables of equal length, the abstract value $\text{Eq}_E^\#(V, V', D, D', \varphi)$ is obtained from $\varphi$ by:

- replacing the quantifier-free part with
  $$E \land \bigwedge_{1 \leq i \leq s} \text{hd}(n_i) = \text{hd}(n'_i) \land \bigwedge_{1 \leq i \leq s} \text{len}(n_i) = \text{len}(n'_i) \land \bigwedge_{1 \leq i \leq t} d_i = d'_i$$
- if the set of patterns $\mathbb{P}$ contains a pattern of the form $[y_1] \in \text{tl}(\omega) \land [y_2] \in \text{tl}(\omega') \land y_1 = y_2$ then we add to $\varphi$ the following formula:
  $$\bigwedge_{1 \leq i \leq s} \forall y_1, y_2. ([y_1] \in \text{tl}(n_i) \land [y_2] \in \text{tl}(n'_i) \land y_1 = y_2) \Rightarrow n_i[y_1] = n'_i[y_2],$$

215
The abstract transformer \( \text{combine}_P^\#(h_n, h_d, \varphi, \varphi') \) Given \( h_n \) a bijection between word variables, \( h_d \) a bijection between data variables, and \( \varphi, \varphi' \) two elements of \( A_U \), the abstract value \( \text{combine}_P^\#(h_n, h_d, \varphi, \varphi') \) is obtained by taking the intersection between (1) a formula obtained from \( \varphi \) by applying the substitutions given by \( h_n \) and \( h_d \) and (2) the formula \( \varphi' \). Formally,

\[
\text{combine}_P^\#(h_n, h_d, \varphi, \varphi') = (\varphi[h_n][h_d]) \cap \varphi',
\]

where \( \varphi[h_n][h_d] \) is obtained from \( \varphi \) by renaming (1) every variable \( n \) in the domain of \( h_n \) to \( h_n(n) \) and (2) every variable \( d \) in the domain of \( h_d \) to \( h_d(d) \).

The soundness of these transformers is given by the next theorem. The only requirement is that the numerical abstract domain \( A_Z \) has an exact meet operator, that is,

\[
\gamma^Z(X \cap^Z Y) = \gamma^Z(X) \cap \gamma^Z(Y).
\]

The latter is true for all the classical numerical abstract domains.

**Theorem 6.4.5.** Let \( A_U \) be as above such that the numerical abstract domain \( A_Z \) has an exact meet operator. Then, the transformers \( \text{proj}_P^\#, \text{sglt}_P^\#, \text{selectSglt}_P^\#, \text{selectNonSglt}_P^\#, \text{updDvar}_P^\#, \text{updFirst}_P^\#, \text{Eq}_P^\#, \) and \( \text{combine}_P^\# \) are sound.

**Proof.** This is a direct consequence of the semantics of the formulas in \( A_U \). \( \square \)

### 6.4.5 SL3\(^U\): a gCSL fragment representing elements of \( A_{\text{BS}}(A_U) \)

The abstract elements of \( A_{\text{BS}}(A_U) \) are representable with SL3\(^U\) formulas, which are SL3 formulas, as in Definition 6.2.1 from Section 6.1, parametrized by formulas in \( A_U \). Thus, SL3\(^U\) formulas have the form:

\[
\exists x. \bigvee \left( \varphi_{\text{SLL}} \land \varphi^P \land \varphi^U \right), \quad \text{where } \varphi^U \in A_U.
\]

**Semantics of SL3\(^U\):** The models of SL3\(^U\) formulas are SLL heaps. Thus, an SLL heap \( (G \triangleright (V, S, L, D), \delta) \) satisfies an SL3\(^U\) formula \( \varphi \) as in (6.4.13) if there exists an interpretation \( \mu : x \to V \) of the node variables and a disjunct \( \varphi_{\text{SLL}}^U \land \varphi^P \land \varphi^U \) of \( \varphi \), where

\[
\varphi^U := E(x) \land \bigwedge_{P(y_1, \ldots, y_q, w_1, \ldots, w_q, W) \in P(x)} \forall y. P(y_1, \ldots, y_q, w_1, \ldots, w_q, W) \implies U_P(x, y)
\]

such that:

1. \((G, \delta)\) satisfies the gCSL formula \( \varphi_{\text{SLL}}^U \land \varphi^P \) w.r.t. the valuation \( \mu \);

2. let \( \xi \) be a valuation of the terms \( \text{hd}(n), \text{len}(n) \) with \( n \in x \), and \( d \in DVar \) defined by \( \xi(\text{hd}(n)) = D(\mu(n)), \xi(d) = \delta(d) \), and \( \xi(\text{len}(n)) = l \), where \( \text{ls}(n, m) \) is an atomic formula in \( \varphi_{\text{SLL}} \) and \( l \) is the length of the path between \( \mu(n) \) and \( \mu(m) \) in \( G \).

    • \( \xi \) is a model of \( E(x) \);
    • for any universal formula \( y. P(y_1, \ldots, y_q, w_1, \ldots, w_q, W) \implies U_P(x, y) \) in \( \varphi^U \), the following holds:
6.4. A ΩW-Domain with Universally Quantified Formulas

- for any valuation $\chi : y_1 \cup \ldots \cup y_q \rightarrow \mathbb{N}$, if the union of the valuations $\xi$ and $\chi$ satisfies $P(y_1, \ldots, y_q, w_1, \ldots, w_q, W)$ (a formula $y \in tl(n)$ is satisfied if $1 \leq \chi(y) < \xi(\text{len}(n))$) then the union of $\xi$, $\chi$, and $\zeta$ satisfies $Up(x, y)$, where $\zeta$ interprets every term $n[y]$ as the integer $D(v_{n,y})$ with $v_{n,y}$ the vertex in $V$ at distance $\chi(y)$ from $\mu(n)$.

**Representing heap sets with $\text{SL}^3_U$ formulas:** Let $\tilde{H}S \in A^{\text{HS}}(\Sigma, k, A_W)$. For every $\tilde{H} \in Hs$ we define a formula $\varphi_{\tilde{H}}^{\text{SL}^3} \in \text{SL}^3_U$ such that the models of $\varphi_{\tilde{H}}^{\text{SL}^3}$ are exactly the heaps in the concretization of $H$, i.e., $\llbracket \varphi_{\tilde{H}}^{\text{SL}^3} \rrbracket = \gamma_1(\tilde{H})$ where $\gamma_1 : A^{\text{HS}}(\Sigma, k, A_W) \rightarrow \mathcal{HS}(\Sigma, k)$. Then, the $\text{SL}^3_U$ formula defining the abstract $k$-SLL heap set $\tilde{H}S$ is

$$\varphi_{\tilde{H}}^{\text{SL}^3} = \bigvee_{\tilde{H} \in \tilde{H}S} \varphi_{\tilde{H}}^{\text{SL}^3}.$$

Let $\tilde{H} = (N, S, L, \varphi^U)$ be an abstract heap as in Definition 6.3.7 and $\varphi^U \in \text{A}_U^U$ (the syntax of $\varphi^U$ is given in (6.4.1) from Section 6.4.1.3). The corresponding formula $\varphi_{\tilde{H}}^{\text{SL}^3} = \varphi_{\text{SL}^3} \land \varphi^p \land \varphi^U$ is built as follows:

- the location variables of $\varphi_{\text{SL}^3}$ correspond to nodes in $N$: for every node $n \in N$, $\varphi_{\text{SL}^3}$ has a unique free location variable with the same name;
- $\varphi_{\text{SL}^3}$ has a predicate $\text{is}(n, m)$ for every $n, m \in N$ such that $S(n) = m$; all these predicates are connected by $\land$;
- $\varphi^p$ is the conjunction of the predicates $p(n)$ for every $p \in PVar$ and $n \in N$ such that $L(p) = n$;
- $\varphi^U$ is the $A_U$ formula from $\tilde{H}$.

**Examples of assertions generate with $A_{\text{HS}}(A_U)$** The only program, among the ones given as examples in Section 4.2.2 and Section 4.3.2 that cannot be handled by the analysis because of the complex data structure it manipulates is **Insert** given in Figure 4.43. In the following, we present the most interesting summaries and assertions that are obtained, by performing an analysis with $A_{\text{HS}}(A_U)$, for the programs given as examples in Section 4.2.2 and Section 4.3.2.

**Example 6.4.16 (Fibonacci).** Let us consider the procedure Fibonacci given in Figure 4.17 from Section 4.3.2. The summary for this procedure generated by the analysis with $A_{\text{HS}}(A_U)$ is given in Figure 6.31.

$$\varphi^U := \text{len}(n) = \text{len}(n^0) \land \text{hd}(n) = 1 \land$$
$$\forall y \in tl(n) \land y = 1 \Rightarrow n[y] = 1 \land$$
$$\forall y_1, y_2, y_3 \in tl(n) \land y_1 < y_2 < y_3 \Rightarrow n[y_1] + n[y_2] = n[y_3]$$

Figure 6.31: Summary for the procedure Fibonacci obtained with $A_{\text{HS}}(A_U)$
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

The corresponding SL3U formula is the following:

\[ \varphi_{Fibo} := \varphi_{SLL} \land \varphi^p \land \varphi_U. \]

Example 6.4.17 (Dispatch). Let us consider the program Dispatch given in Figure 4.5 from Section 4.2.2. We recall that the program dispatches the elements of a list with respect to the value of some integer variable v. The invariant synthesized for the while loop, using \(A_{HS}(A_U)\) without allowing any simple nodes (i.e., \(k = 0\)) and parametrized by the pattern \(y \in tl(\omega)\), is given in (6.4.14). It states that the variable gr, respectively sm, points to a list whose elements are greater than, respectively less than or equal to, v.

\[ \varphi_{Disp} := \varphi_{SLL} \land \varphi_U \land (\forall y \in tl\(gr\) \implies gr[y] > v) \land (\forall y \in tl\(sm\) \implies sm[y] \leq v) \lor \]

\[ \varphi_{SLL} \land \varphi_U \land (\forall y \in tl\(gr\) \implies gr[y] > v) \land (\forall y \in tl\(sm\) \implies sm[y] \leq v) \]

(6.4.14)

By an abuse of notation, the formula \(\varphi_{Disp}\) uses the same names for pointer variables and node variables. It captures the relation between the shape and the data content of the lists. It distinguishes two possible values for the pointer variables ai: the case when it points to null and when it is different from null. Actually, the formula \(\varphi_{Disp}\) does not contain all the disjuncts from the invariant generated for this loop. The missing cases are: sm points to null and ai points to null, sm points to null and ai is different from null, gr points to null and ai points to null, gr points to null and ai is different from null, and the case when the initial list is empty and all variables point to null. When sm (resp., gr) points to null, the corresponding disjunct of the invariant has no universally quantified formula describing the content of the list pointed to by sm (resp., gr), i.e., there is no formula having as guard \(y \in tl(sm)\) (resp., \(y \in tl(gr)\)).

Example 6.4.18 (AddV). Let us consider the program AddV given in Figure 4.19 from Section 4.3.2. We recall that the procedure addV receives as input a (possibly empty) list and an integer variable v, and modifies the values of the data fields of the input list by incrementing them with v. The summary for this procedure synthesized with \(A_{HS}(A_U)\) using \(k = 1\) and parametrized by the pattern \(y \in tl(\omega) \land y' \in tl(\omega') \land y = y'\) is the formula
6.4. A \( \mathbb{D} \mathbb{W} \)-DOMAIN WITH UNIVERSALLY QUANTIFIED FORMULAS

\( \varphi^{addV} \), given in (6.4.15).

\[
\varphi^{addV} := \varphi^{SLL} \land \varphi^p \land \varphi^U
\]

\[
= \left( \text{ls}(\text{head}^0, \text{null}) \ast \text{ls}(\text{head}, \text{null}) \land 
\right.

\[
\left. (\text{len}(\text{head}^0) = \text{len}(\text{head}) \land \text{hd}(\text{head}) = \text{hd}(\text{head}^0) + 2
\land \forall y, y'. (y \in \text{tl}(\text{head}^0) \land y' \in \text{tl}(\text{head}) \land y = y') \implies \text{head}[y'] = \text{head}^0[y] + v) \right)
\]

\[
\lor \left( \text{head} = \text{null} \land \text{head}^0 = \text{null} \right)
\]

(6.4.15)

The pattern given as a parameter, guides the analysis towards computing the relation between the data of the input and the output list.

Example 6.4.19 (Sorting algorithms). We consider sorting procedures having the following prototype: \( \text{list}* \text{sort}(\text{list}* \text{input}) \). The specification of a sorting algorithm includes the property \( \varphi^{sort} \) given in (6.4.16), where \( \text{res} \) is the output sorted list.

\[
\varphi^{sort} = \text{ls}(\text{res}, \text{null}) \land 
\forall y, y'. (y, y' \in \text{tl}(\text{res}) \land y \leq y') \implies \text{res}[y] \leq \text{res}[y']
\]

(6.4.16)

Using the domain \( A_{\mathbb{HS}}(A_U) \) parametrized by the patterns \( y, y' \in \text{tl}(\omega) \land y \leq y' \) and \( y \in \text{tl}(\omega) \) we analyzed several sorting algorithms. The analysis was able to synthesize the summary given in (6.4.16) for sorting algorithms like insertion sort or merge sort, but not for the procedure \( \text{quicksort} \) as given in Figure 4.21. In Section 6.7.1, we explain why this analysis fails to synthesize the expected summary and we describe a solution for this.

Example 6.4.20 (Initialization with the first even numbers). Consider a procedure \( \text{void init}(\text{list}* \text{list}^0) \), that initializes a list with the first even numbers \([0,2,4,\ldots]\). The analysis with \( A_{\mathbb{HS}}(A_U) \) parametrized by the pattern \( y \in \text{tl}(\omega) \) is able to synthesize the summary for such a procedure, given by (6.4.17). This summary contains a relation between the values of a list and its length. In (6.4.17) \( \text{list}^0 \) denotes the input list and \( \text{list}^1 \) the value of the parameter \( \text{list}^0 \) at the end of the procedure.

\[
\varphi^{init} := \text{ls}(\text{list}^0, \text{null}) \ast \text{ls}(\text{list}^1, \text{null}) \land
\text{len}(\text{list}^0) = \text{len}(\text{list}^1) \land \text{hd}(\text{list}^1) = 0 \land
\forall y, y \in \text{tl}(\text{list}^1) \implies \text{list}^1[y] = 2 \ast y.
\]

(6.4.17)

Relation between SL3\(^U\) formulas and gCSL formulas. Any SL3 formula \( \varphi^{SL3} = \varphi^{SLL} \land \varphi^p \land \varphi^U \) is a shorthand for a gCSL formula, denoted \( \varphi^{gCSL} \), obtained by (1) adding to \( \varphi^{SLL} \land \varphi^p \) a conjunct \( \varphi^\sharp \) that expresses the fact that \( \sharp \) has no successors, i.e., \( \varphi^\sharp := \forall y. \neg \sharp \ast y \), and (2) applying the following transformations to \( \varphi^U \):

1. for any position variable \( y \), we introduce a fresh location variable \( \nu_y \), that has the same quantification as \( y \) (i.e., it is universally quantified);
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

2. atomic formulas of the form \( y \in t_1(x) \) from the guard of the universal formulas are replaced by

\[
\bigwedge_{y \in y} x \rightarrow v_y,
\]

3. for any term \( \text{len}(x) \) that appears in \( \varphi^U \) we introduce a fresh index variable, denoted \( l_x \), a reachability predicate \( x \downarrow \rightarrow x' \) where \( \text{ls}(x, x') \) is a predicate of \( \varphi^\text{SLL} \), and we replace \( \text{len}(x) \) by \( l_x \);

4. any term \( \text{hd}(x) \) that appears in \( \varphi^U \) is replaced by \( dt(x) \), where \( dt \) is the data field;

5. every data predicate \( P(dt_1, \ldots, dt_n) \) in \( \varphi^U \) is replaced by \( P(dt'_1, \ldots, dt'_n) \), where \( dt'_i = dt_i \) if \( dt_i \in DVar \), \( dt'_i = dt(v_j) \) if \( dt_i = x[j] \), for any \( 1 \leq i \leq n \), and \( dt'_i = \text{len}(x) \) if \( dt_i = \text{len}(x) \) for any \( 1 \leq i \leq n \).

If \( \varphi^\text{SL3} \) is a disjunction then we apply this transformation to each disjunct.

**Proposition 6.4.3.** Let \( G = (V, S, L, D) \) be a an SLL-heap graph and let \( \varphi^\text{SL3} \) be an SL3\(^U\) formula. Then,

\[
(G, \delta) \models_{\mu, \theta, \nu, \kappa} \varphi^\text{SL3} \iff (G, \delta) \models_{\mu, \theta, \nu, \kappa} \varphi^\text{gCSL}_g\text{CSL}
\]

where \( \varphi^\text{gCSL}_g\text{CSL} \) is the gCSL formula corresponding to \( \varphi^\text{SL3} \).

The proof of this proposition follows from the definition of \( \varphi^\text{gCSL}_g\text{CSL} \).

**Example 6.4.21.** The gCSL formula corresponding to \( \varphi^\text{Fib} \) given in Example 6.4.16 is the following:

\[
\varphi^\text{Fib}_{\text{gCSL}} := \text{head}(n) \land n \downarrow \rightarrow \text{null} \land \forall y. n \downarrow \rightarrow y \implies dt(y) = 1 \land \forall y_1, y_2, y_3. (n \downarrow \rightarrow y_1 \land y_1 \downarrow \rightarrow y_2 \land y_2 \downarrow \rightarrow y_3 \land y_3 \downarrow \rightarrow \text{null}) \implies dt(y_3) = dt(y_1) + dt(y_2)
\]

**SL3\(^{ICSL} \)** a fragment of SL3\(^U\) and ICSL The fragment of gCSL defined by SL3\(^U\) formulas is incomparable with ICSL. ICSL captures more complex shape properties while SL3 restricts the handled data structures allowing more complex data properties. To improve the verification of ICSL assertions, we have identified a fragment at the intersection between SL3\(^U\) and ICSL, called SL3\(^{ICSL} \). It is a fragment of SL3\(^U\) obtained by restricting the syntax of the formulas from \( \mathcal{A}_U \).

**Definition 6.4.2 (SL3\(^{ICSL} \)).** SL3\(^{ICSL} \) consists of formulas of the following from:

\[
\bigvee (\varphi^\text{SLL} \land \varphi^p \land \varphi^U),
\]

where \( \varphi^U \) is a formula in \( \mathcal{A}_U \) parametrized by the patterns that use only constraints of the form \( y \leq y' \), i.e., \( \{ y \in t_1(\omega), y_1, \ldots, y_n \in t_1(\omega) \land \bigwedge_{1 \leq i < n} y_i \leq y_{i+1} \mid \text{for every } n \geq 2 \} \), and for any conjunct \( \forall y. P(y, N) \implies U_P(y, N) \) of \( \varphi^U \), the position variables \( y \) appear only in terms of the form \( n[y] \) in \( U_P(y, N) \), with \( n \in N \) and \( y \in \text{y} \) (\( N \) are the data word variables corresponding to the location variables in \( \varphi^\text{SLL} \land \varphi^p \)).

220
6.5. A $\mathcal{D}\mathcal{W}$-domain of multiset formulas

We present the abstract domain $\mathcal{A}_M = (\mathcal{A}_M, \sqsubseteq_M, \sqcap_M, \sqcup_M, \top_M, \bot_M)$ whose elements are multiset constraints with free variables in $\mathcal{N} \cup DVar$. In the following we define the elements of the multiset domain and the lattice operators. Then, we briefly describe the abstract transformers corresponding to the ones in Section 6.3.3.

6.5.1 Lattice definition

A multiset formula is a conjunction of equalities between unions of basic multiset terms. A basic multiset term is

$$u ::= \text{ms\_hd}(n) \mid \text{ms\_tl}(n) \mid \text{ms}(d),$$

where $n \in \mathcal{N}$ and $d \in DVar$. The term $\text{ms\_hd}(n)$ (resp. $\text{ms}(d)$) represents the singleton containing the first letter of the word associated to $n$ (resp. the value of $d$). The term $\text{ms\_tl}(n)$ represents the multiset containing all the data values of the word associated to $n$ except the first one. This term can be interpreted into the empty multiset, while $\text{ms\_hd}(n)$ and $d$ are interpreted always as singletons. A multiset term is a finite union between distinct basic multiset terms

$$t ::= u_1 \cup \ldots \cup u_s,$$

where any $u_i, u_j$ with $1 \leq i \neq j \leq s$ are distinct and $\cup$ is the usual union operator between multisets. As a shorthand, $\text{ms}(n)$ denotes the term $\text{ms\_hd}(n) \cup \text{ms\_tl}(n)$. A multiset constraint is a formula of the form

$$\varphi ::= \text{true} \mid \text{false} \mid t = t' \mid \varphi \land \varphi.$$

In general we denote a multiset constraint by

$$\varphi^M ::= \bigwedge_c \varphi_c,$$

where $\varphi_c$ is of the form $t = t'$ where $t$ and $t'$ are multiset terms. The sets $\mathcal{N}$ and $DVar$ are bounded by the number of program variables, therefore the number of basic terms is also bounded. Moreover, we consider multiset terms that don’t contain multiple occurrences of basic terms, therefore the lattice of multiset formulas is finite.

Let $\top^M$ be the constant $true$ and $\bot^M$ be the constant $false$. 

To synthesize SL$_3^{\text{ICSL}}$ assertions, we analyze the program with the abstract domain $\mathcal{A}_\mathcal{HS}(A_U)$ parametrized by the patterns that use only constraints of the form $y \leq y'$. Moreover, each abstract transformer $U^\mathcal{HS}_a$, resp. $U^\mathcal{HS}_a$, is composed with the projection of the abstract transformer to eliminate all constraints on $y$, from the right part of the universally quantified implications. More precisely, this transformer takes as input the abstract element returned by $U^\mathcal{HS}_a$ or $U^\mathcal{HS}_a$, i.e. an $A_U$ formula denoted $\varphi^U$, and replaces any conjunct $\forall y. P(y, N) \Rightarrow U^U P(y, N)$ of $\varphi^U$ with

$$\forall y. P(y, N) \Rightarrow U^U P(y, N)\upharpoonright y.$$
To define lattice operators, we start by defining a procedure $\text{sat}^\#(\varphi)$, where $\varphi \in \mathcal{A}^M$, which applies the commutativity of $=$ and $\cup$, the associativity of $\cup$, and substitutions in order to obtain new atomic formulas that are implied by the existing ones. The substitutions are applied as follows:

- if $t_1 = t_2$ is an atomic formula of $\varphi$ then we add a new atomic formula
  - for every conjunct of $\varphi$ of the form $t_3 \cup t_1 = t'$, by substituting $t_1$ with $t_2$.
  - and for every conjunct of $\varphi$ of the form $t_3 \cup t_2 = t'$, by substituting $t_2$ with $t_1$.

- if $t = t'$ and the number of basic term of the form $\text{ms.\text{hd}}(n)$ and $\text{ms}(d)$ is not equal in $t$ and $t'$ then we add false as a conjunct.

Let $\varphi_1$ and $\varphi_2$ be two elements in $\mathcal{A}^M$. The order relation between multiset formulas is an over-approximation of the usual implication:

$$\varphi_1 \preceq^M \varphi_2 \quad \text{iff} \quad (A) \text{false is a conjunct of } \text{sat}^\#(\varphi_1) \text{ or }$$

$$\text{(A) } \varphi_2 = \text{true or }$$

$$\text{(C) for every conjunct } t_1 = t_2 \text{ in } \varphi_2 \text{ there exists in } \varphi_1:$$

- $t_1^1, t_2^2, \ldots, t_1^p$ such that $t_1 = t_1^1 \cup t_2^2 \cdots \cup t_1^p$,
- $t_2^1, t_2^2, \ldots, t_2^p$ such that $t_2 = t_2^1 \cup t_2^2 \cdots \cup t_2^p$, and
- $\text{sat}^\#(\varphi_1)$ has a conjunct of the form $t_1^i = t_2^i$, for any $1 \leq i \leq p$.

We define $\varphi_1 \sqcap^M \varphi_2$ to be the conjunction of atomic formulas that appear in both $\text{sat}^\#(\varphi_1)$ and $\text{sat}^\#(\varphi_2)$. The meet operator $\varphi_1 \sqcap^M \varphi_2$ is defined to be (1) the conjunction of atomic formulas that appear in $\text{sat}^\#(\varphi_1)$ or $\text{sat}^\#(\varphi_2)$ if false does not belong to neither $\text{sat}^\#(\varphi_1)$ nor $\text{sat}^\#(\varphi_2)$ and (2) $\sqcap^M$, otherwise. Since the abstract domain contains a finite number of elements, there is no need to consider a widening operator $\sqcup^M$. The fact that $\sqcap^M$ and $\sqcup^M$ define the least upper bound and respectively, the greatest lower bound w.r.t. $\preceq^M$ is a direct consequence of their definition and the definition of $\text{sat}^\#$.

### 6.5.2 Concretization based abstraction

The concretization function $\gamma^M$ is defined from formulas in $\mathcal{A}_d$ to pairs of functions in $\mathcal{C}_V$. The concretization corresponds to the semantics of the formulas. Let $\varphi$ be a multiset constraint in $\mathcal{A}_d$ over the word variables $V$ and the data variables $DVar$. Then,

$$\gamma^M(\varphi) = \{(\nu, \delta) \mid \theta \models \varphi \} \text{ where, (1) for every } n \in V, \nu(n) \text{ is a word of length } p \geq 1$$

$$\theta(\text{ms.\text{hd}}(n)) = \{\text{hd}(\nu(n))\}, \text{ and }$$

$$\theta(\text{ms.\text{tl}}(n)) = \{\nu(n)[2], \ldots, \nu(n)[p]\},$$

(2) for every $d \in DVar$, $\theta(\text{ms}(d)) = \{\delta(d)\}\}$.

### 6.5.3 Abstract semantics

Next, we define the abstract transformers associated to this $\mathbb{D}^V(Vec)$-domain, corresponding to the concrete ones given in Section 6.3.3 Thus, for any $\varphi \in \mathcal{A}^M$, $d \in DVar$, and $n, n' \in N$,
6.5. A $\mathbb{D}M$-DOMAIN OF MULTISET FORMULAS

- $\text{sglt}^#_M(n, \varphi)$ adds to $\varphi$ two dimensions for $\mathbf{ms}_{\mathbf{hd}}(n)$ and $\mathbf{ms}_{\mathbf{tl}}(n)$ such that
  \[
  \text{sglt}^#_M(n, \varphi) = \varphi \land \mathbf{ms}_{\mathbf{hd}}(n) \cup \mathbf{ms}_{\mathbf{tl}}(n) = \mathbf{ms}_{\mathbf{hd}}(n);
  \]

- $\text{selectSglt}^#_M(n, \varphi) = \varphi[\rho_n]$, where $\rho_n$ is a function from terms to terms which corresponds to defining $\mathbf{ms}_{\mathbf{tl}}(n)$, and any other terms which are equal to $\mathbf{ms}_{\mathbf{tl}}(n)$, to be the empty set. $\text{selectSglt}^#_M$ returns $\bot^M$ if it detects a contradiction, such as $\mathbf{ms}(d)$ equals empty set or $\mathbf{ms}_{\mathbf{hd}}(n)$ equals empty set. Formally, $\rho_n$
  1. removes $\mathbf{ms}_{\mathbf{tl}}(n)$ from any term of $\varphi$;
  2. for any atomic formula of the form $t = \mathbf{ms}_{\mathbf{tl}}(n)$ deduced from $\varphi$ using the transitivity of the equality relation, $\rho_n$ removes all the occurrences of the term $t$ from $\varphi$, if $t$ does not contain any terms of the form $\mathbf{ms}_{\mathbf{hd}}(n')$ or $d$.

- Otherwise, $\text{selectSglt}^#_M(n, \varphi) = \bot^M$;

- $\text{selectNonSglt}^#_M(n, \varphi)$ replaces in $\varphi$ the multiset variable $\mathbf{ms}_{\mathbf{tl}}(n)$ with the term $\mathbf{ms}_{\mathbf{tl}}(n) \cup \mathbf{ms}_{\mathbf{hd}}(n') \cup \mathbf{ms}_{\mathbf{tl}}(n')$;

- $\text{proj}^#(D, N, \varphi)$ considers the following cases:
  - for any $n \in N$, (1) it deletes all conjuncts of the form $t = t'$, such that either $t$ or $t'$ but not both are union terms that include $\mathbf{ms}_{\mathbf{tl}}(n)$ or $\mathbf{ms}_{\mathbf{hd}}(n)$, (2) it deletes the term $\mathbf{ms}_{\mathbf{hd}}(n)$, respectively $\mathbf{ms}_{\mathbf{tl}}(n)$, from any conjunct of the form $t = t'$, such that $\mathbf{ms}_{\mathbf{hd}}(n)$, respectively $\mathbf{ms}_{\mathbf{tl}}(n)$, is a term used in both $t$ or $t'$;
  - for any $d \in D$ (1) it deletes all conjuncts of the form $t = t'$, such that either $t$ or $t'$ but not both are union terms that include $\mathbf{ms}(d)$ (2) it deletes the term $\mathbf{ms}(d)$ from any conjunct of the form $t = t'$, such that $\mathbf{ms}(d)$ is a term used in both $t$ or $t'$;

- $\text{updFirst}^#_M(d, n, \varphi)$ first projects out the term $\mathbf{ms}_{\mathbf{hd}}(n)$ from $\varphi$ and then adds to $\varphi$ the atomic formula $\mathbf{ms}_{\mathbf{hd}}(n) = \mathbf{ms}(d)$;

- $\text{updDvar}^#_M(d, n, \varphi)$ first projects out the term $\mathbf{ms}(d)$ from $\varphi$ and then adds to $\varphi$ the atomic formula $\mathbf{ms}(d) = \mathbf{ms}_{\mathbf{hd}}(n)$;

- $\text{concat}^#_M(M, \varphi)$, where $M = \{(n_1, v_1), \ldots, (n_p, v_p)\}$ is a set of pairs between word variables and vectors of word variables, applies $\text{sat}^#(\varphi)$ and then, for any $v_i = [v_{i,1}, v_{i,2}, \ldots, v_{i,k_i}]$, $1 \leq i \leq p$, it replaces the union term:
  \[
  \mathbf{ms}_{\mathbf{tl}}(v_i^1) \cup \mathbf{ms}_{\mathbf{hd}}(v_i^2) \cup \mathbf{ms}_{\mathbf{tl}}(v_i^2) \cup \cdots \cup \mathbf{ms}_{\mathbf{hd}}(v_i^{k_i}) \cup \mathbf{ms}_{\mathbf{tl}}(v_i^{k_i})
  \]
  with $\mathbf{ms}_{\mathbf{tl}}(n_i)$.

223
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

- \( \text{Eq}^\#_M(V, V', D, D', \varnothing) \), where \( V = n_1 \ldots n_s \) and \( V' = n'_1 \ldots n'_s \) are vectors of data word variables of equal length, and \( D = d_1 \ldots d_t \) and \( D' = d'_1 \ldots d'_t \) are vectors of data variables of equal length, adds to \( \varnothing \) the following formula

\[
\bigwedge_{1 \leq i \leq t} \text{ms}(d_i) = \text{ms}(d'_i) \land \bigwedge_{1 \leq i \leq s} (\text{ms.hd}(n_i) = \text{ms.hd}(n'_i) \land \text{ms.tl}(n_i) = \text{ms.tl}(n'_i))
\]

- \( \text{combine}^\#_M(h_n, h_d, \varnothing, \varnothing') \), where \( h_n \) is a bijection between word variables, \( h_d \) is a bijection between data variables, and \( \varnothing, \varnothing' \) are two elements of \( A_M \), takes the intersection between (1) a formula obtained from \( \varnothing \) by applying the substitutions given by \( h_n \) and \( h_d \) and (2) the formula \( \varnothing' \). Formally,

\[
\text{combine}^\#_M(h_n, h_d, \varnothing, \varnothing') = (\varnothing[h_n][h_d]) \sqcap_M \varnothing',
\]

where \( \varnothing[h_n][h_d] \) is obtained from \( \varnothing \) by renaming (1) every variable \( n \) in the domain of \( h_n \) to \( h_n(n) \) and (2) every variable \( d \) in the domain of \( h_d \) to \( h_d(d) \).

The definition of the lattice \( A_M \) and of the abstract transformers could be extended in a straightforward manner to multiset formulas that have a constant number of occurrences of the same basic term.

6.5.4 \( A_M \) formulas representation

Multiset constraints can be represented by a polyhedron with a dimension for any basic term: a formula \( u_1 \cup u_2 \cup \ldots \cup u_s = v_1 \cup v_2 \cup \ldots \cup v_t \) is represented by the linear constraint \( u_1 + u_2 + \cdots + u_s = v_1 + v_2 + \cdots + v_t \). The entailment relation between the multiset constraints is defined by the entailment relation between the corresponding polyhedra. The lattice of multiset constraints is finite (for finite \( N \) and \( DVar \)) and consequently, there is no need for a widening operator. The abstract transformers are implemented using substitutions, projections, and transformers of the Polyhedra domain. For example, \( \text{selectSgl}_M = (n, \varnothing) \) corresponds to intersection (meet) with the constraint \( \text{ms.tl}(n) = 0 \) and \( \text{selectNonSgl}_M(n, \varnothing) \) is implemented as the intersection with the constraint \( \text{ms.tl}(n) > 0 \). Moreover, this representation allows the multiset domain to handle also assignments with expression that is, \( \text{updFirst}_M(d, exp, \varnothing) \) and \( \text{updDvar}_M(d, exp, \varnothing) \) where \( exp \) is a linear expression over \( DVar \) and \( \text{ms.hd}(n) \).

6.5.5 \( SL3^M \): a logic for representing the elements of \( A_{\mathcal{HS}}(A_M) \)

The abstract elements of \( A_{\mathcal{HS}}(A_M) \) are representable with \( SL3^M \) formulas, which are \( SL3 \) formulas, as given in Definition 6.2.1 from Section 6.1 parametrized by formulas in \( A_M \).

Let \( \tilde{H} \in A_{\mathcal{HS}}(W, k, A_M) \). For every \( H \in \tilde{H} \) we define a formula \( \varnothing^M_{\tilde{H}} \in SL3^M \) such that the models of \( \varnothing^M_{\tilde{H}} \) are exactly the heaps in the concretization of \( \tilde{H} \). Then, the \( SL3^M \) formula defining the abstract \( k \)-SLL heap set \( \widetilde{HS} \) is \( \varnothing^M_{\tilde{H}}_{\tilde{H}} = \bigvee_{H \in \tilde{H}} \varnothing^M_{\tilde{H}} \).

Let \( \tilde{H} = (N, S, L, \varnothing^M) \) be an abstract heap as in Definition 6.3.7 and \( \varnothing^M \in A_M \). The corresponding \( SL3^M \) is \( \varnothing^M \land \varnothing^P \land \varnothing^M \), where the formula \( \varnothing^M \) is build by associating to each edge of the graph a predicate \( 1s(n, m) \) where \( n, m \in N \) are the ends of the edge, and \( \varnothing^P \) is build by taking the conjunction of a set of positive predicates \( p(n) \), one for each label \( p \) of a node \( n \in N \).
6.6. A \( \mathbb{DW} \)-domain of sum formula

Examples of generated assertions with \( A_{\mathbb{HS}}(A_M) \) Consider the program given in Section 4.3.2 in Figure 4.16. An analysis with \( A_{\mathbb{HS}}(A_M) \) generates the following summary for the procedure `concat`:

\[
\varphi^M_{\text{concat}} ::= \text{ls}(a^0, \text{null}) \star \text{ls}(b^0, \text{null}) \star \text{ls}(a, \text{null}) \land \\
\text{ms}(a^0) \cup \text{ms}(b^0) = \text{ms}(a)
\]

where \( a^0 \) and \( b^0 \) are the values of the input parameters for the procedure `concat` and \( a \) is the value of the parameter at the end of the procedure.

The summary for the procedure `list_share` given in the same Figure 4.16 is:

\[
\varphi^M_{\text{list_share}} ::= \exists n. \text{ls}(a, \text{null}) \star \text{ls}(b, \text{null}) \star \text{ls}(a, n) \star \text{ls}(b, n) \star \text{ls}(n, \text{null}) \land \\
\text{ms}(a) \cup \text{ms}(b) = \text{ms}(a) \cup \text{ms}(b) \cup \text{ms}(n)
\]

The node variable \( n \) denotes the sharing point of the lists pointed by \( a \) and \( b \) which is not pointed to by any program variable.

The last example that we consider is the summary for the sorting procedure `quicksort` given in Section 4.3.2 in Figure 4.21:

\[
\varphi^M_{\text{quicksort}} ::= \text{ls}(a^0, \text{null}) \star \text{ls}(a, \text{null}) \land \\
\text{ms}(a^0) = \text{ms}(a)
\]

where \( a^0 \) represents the input value for the parameter \( a \) and \( a \) is its output value.

6.6 A \( \mathbb{DW} \)-domain of sum formula

To reason about the sum of data in a word, we define the \( \mathbb{DW} \)-domain \( A_S = (A^S, \sqsubseteq^S, \sqcap^S, \sqcup^S, \top^S, \bot^S) \) parametrized by a numerical abstract domain whose dimensions represent integer program variables or terms of the form \( \text{hd}(n), \text{len}(n), \text{sum}_t(n) \), with \( n \in \mathcal{N} \). In the following we define the elements of the sum domain and the lattice operators. Then, we briefly describe the abstract transformers corresponding to the ones in Section 6.3.3.

6.6.1 Lattice definition

Let \( \mathcal{N} \) be a set of word variables and \( DVar \) be a set of program data variables. An element \( \varphi(\mathcal{N}, DVar) \) of \( A_S \) is a constraint from \( A_Z \) defined over \( \{ \text{hd}(n), \text{len}(n), \text{sum}_t(n) \mid n \in \mathcal{N} \} \cup DVar \). For any word variable \( n \in \mathcal{N} \), \( \text{hd}(n) \) denotes the value associated with the first position of the word denoted by \( n \), \( \text{len}(n) \) denotes the length of the word (it is always constraint to be positive) and \( \text{sum}_t(n) \) denotes the sum of all values associated with positions of \( n \) except for the first one. All these terms represent integers. The lattice order relation \( \sqsubseteq^S \) is the order relation on the numerical abstract domain \( \sqsubseteq^Z \) and also \( \sqcap^S, \sqcup^S, \top^S \) are the numerical domain operators \( \sqcap^Z, \sqcup^Z, \top^Z \).

6.6.2 Concretization based abstraction

The concretization function is defined using the concretization of the numerical abstract domain, \( \gamma^Z \), such that the sum of all values of a word \( n \) corresponds to the value associated
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

by $\gamma^Z$ to $\text{sum}_\text{tl}(n)$ plus the value associated by $\gamma^Z$ to $\text{hd}(n)$. More precisely,

$$\gamma^S(\varphi(N, D\text{Var})) = \{(\nu_\theta, \delta_\theta) \mid \theta \in \gamma^Z(\varphi) \text{ such that }$$

$$\text{len}(\nu_\theta(n)) = \theta(\text{len}(n)), \text{hd}(\nu_\theta(n)) = \theta(\text{hd}(n)), \delta_\theta(d) = \theta(d),$$

$$\nu_\theta(n)[2] + \ldots + \nu_\theta[\theta(\text{len}(n))] = \theta(\text{sum}_\text{tl}(n))\}.$$

6.6.3 Abstract semantics

Next, we define the abstract transformers associated to $A_S$, corresponding to the concrete ones given in Section 6.3.3. Their definition is based on the definition of the standard abstract transformers in $A_Z$. Thus, for any $\varphi \in A^S$, $d \in D\text{Var}$, and $n, n' \in N$,

- $\text{sglt}_S(n, \varphi)$ adds to $\varphi$ two dimensions for $\text{hd}(n)$ and $\text{sum}_\text{tl}(n)$ and defines $\text{sum}_\text{tl}(n) = 0$, while $\text{hd}(n)$ is not constraint:

  $$\text{sglt}_S(n, \varphi) = \text{update}^\#(\text{sum}_\text{tl}(n), 0, \text{addDims}(\text{hd}(n), \text{sum}_\text{tl}(n), \varphi)),$$

  where $\text{update}^\#$, respectively $\text{addDims}$, is the numerical abstract transformer for assignment, respectively adding dimensions;

- $\text{selectSgt}_S(n, \varphi)$ is a $\varphi \cap^Z (\text{len}(n) = 1)$; if $\varphi$ implies that $\text{len}(n) \leq 1$ then, by intersecting it with $\text{len}(n) = 1$ we select only those concretizations where $n$ is a singleton; if $\varphi$ implies that $\text{len}(n) > 1$ then the returned value is $\bot^S$;

- $\text{selectNonSgt}_S(n, \varphi)$ is a $\varphi \cap^Z \text{len}(n) > 1$; interesting $\varphi$ with the constraint $\text{len}(n) > 1$ we select all the concretizations where $n$ is not a singleton, if any;

- $\text{split}_S(n', \varphi)$ uses the substitution in $A_Z$ and replaces the term $\text{sum}_\text{tl}(n)$ with $\text{sum}_\text{tl}(n) + \text{hd}(n') + \text{sum}_\text{tl}(n')$;

- $\text{updFirst}_S(d, n, \varphi)$ uses the $\text{update}^\#$ transformer in $A_Z$ to assign $\text{hd}(n) = d$ in $\varphi$;

- $\text{updDVar}_S(d, n, \varphi)$ uses the $\text{update}^\#$ transformer in $A_Z$ to assign $d = \text{hd}(n)$ in $\varphi$;

- $\text{proj}_S(D, N, \varphi)$ considers the following cases:

  - for any $n \in N$, it applies the projection in $A_Z$ to eliminate $\text{hd}(n)$ and $\text{sum}_\text{tl}(n)$ from $\varphi$;

  - for any $d \in D$, it applies the projection in $A_Z$ to eliminate $d$ from $\varphi$;

- $\text{concat}_S(M, \varphi)$, where $M = \{(n_1, v_1), \ldots, (n_p, v_p)\}$ is a set of pairs between word variables and vectors of word variables such that $v_i = [v^{i1}_1, v^{i2}_1, \ldots, v^{ik_i}_1]$, for any $1 \leq i \leq p$, assigns to $\text{sum}_\text{tl}(n_i)$:

  $$\text{sum}_\text{tl}(v^{i1}_1) + \text{hd}(v^{i2}_1) + \text{sum}_\text{tl}(v^{i3}_1) + \ldots + \text{hd}(v^{ik_i}_1) + \text{sum}_\text{tl}(v^{ik_i}_1).$$

- $\text{Eq}_S(V, V', D, D', \varphi)$, where $V = n_1 \ldots n_s$ and $V' = n'_1 \ldots n'_s$ are vectors of data word variables of equal length, and $D = d_1 \ldots d_t$ and $D' = d'_1 \ldots d'_t$ are vectors of data variables of equal length, intersects $\varphi$ with

  $$\bigwedge_{1 \leq i \leq t} d_i = d'_i \land \bigwedge_{1 \leq i \leq s} (\text{hd}(n_i) = \text{hd}(n'_i) \land \text{sum}_\text{tl}(n_i) = \text{sum}_\text{tl}(n'_i))$$

226
6.7 COMBINING ABSTRACT DOMAINS

- **combine**$_S^\#(h_n, h_d, \varphi, \varphi')$, where $h_n$ is a bijection between word variables, $h_d$ is a bijection between data variables, and $\varphi, \varphi'$ are two elements of $A_S$, takes the intersection between (1) a formula obtained from $\varphi$ by applying the substitutions given by $h_n$ and $h_d$ and (2) the formula $\varphi'$. Formally,

$$
\text{combine}_S^\#(h_n, h_d, \varphi, \varphi') = (\varphi[h_n][h_d]) \cap^S \varphi',
$$

where $\varphi[h_n][h_d]$ is obtained from $\varphi$ by renaming (1) every variable $n$ in the domain of $h_n$ to $h_n(n)$ and (2) every variable $d$ in the domain of $h_d$ to $h_d(d)$.

We have implemented the $\text{DW}$-domain $A_S$ based on the polyhedra domain [Cousot 1978]. This allows us to combine the length constraints to sum constraints.

6.6.4 SL3$: a logic for representing the elements of $A_{\text{HS}}(A_S)$

The abstract elements of $A_{\text{HS}}(A_S)$ are representable with SL3 formulas, which are SL3 formulas, as given in Definition 6.2.1 from Section 6.1 parametrized by formulas in $A_S$.

Let $H_S \in A_{\text{HS}}(\Sigma, k, A_S)$. For every $\tilde{H} \in H_S$ we define a formula $\varphi_{\tilde{H}}^{\text{SL3}} \in \text{SL3}^S$ such that the models of $\varphi_{\tilde{H}}^{\text{SL3}}$ are exactly the heaps in the concretization of $\tilde{H}$. Then, the SL3$^S$ formula defining the abstract k-SLL heap set $H_S$ is $\varphi_{\tilde{H}}^{\text{SL3}} = \bigvee_{\tilde{H} \in H_S} \varphi_{\tilde{H}}^{\text{SL3}}$.

Let $\tilde{H} = (N, S, L, \varphi^S)$ be an abstract heap as in Definition 6.3.7 and $\varphi^S \in A_S$. The corresponding SL3$^S$ is $\varphi^{\text{SL3}} \wedge \varphi^p \wedge \varphi^S$, where the formula $\varphi^{\text{SL3}}$ is built by associating to each edge of the graph a predicate $\text{ls}(n, m)$ where $n, m \in N$ are the ends of the edge, and $\varphi^p$ is built by taking the conjunction of a set of positive predicates $p(n)$, one for each label $p$ of a node $n \in N$.

Examples of generated assertions with $A_{\text{HS}}(A_S)$ Consider the procedure Fibonacci given in Section 4.2.2 in Figure 4.17. Using $A_{\text{HS}}(A_S)$ the generated assertion at line 45 is expressing the relation between the last two elements of the sequence, denoted by $m_1$ and $m_2$ and the sum of the other elements in the sequence:

$$
\varphi^S_{44} := \text{ls(head, null)} \wedge \text{sum_tl(head)} = 2 \times m_2 + m_1 - 1 \quad (6.6.1)
$$

If we consider the procedure addV given in Section 4.2.2 in Figure 4.19 the summary for this procedure generated using $A_{\text{HS}}(A_S)$ is the following:

$$
\varphi_{\text{addV}}^S := \text{ls}(\text{head}^0, \text{null}) \wedge \text{ls}(\text{head}, \text{null}) \wedge \\
\text{hd(head)} + \text{sum_tl(head)} = \text{hd(head}^0) + \text{sum_tl(head}^0) + v \times \text{len(head)},
$$

where $\text{head}^0$ represent the input list, $\text{head}$ the output one, and $v$ is the variable whose value is added to each element of the input list.

6.7 Combining abstract domains

In this section we exploit different ways to combine analyses with different abstract domains in order to obtain an accurate and scalable inter-procedural analysis. To gain precision, we introduce a partial reduction operator, called strengthen$_{\text{HS}}$ between the domain of abstract heap sets over universally quantified formulas and abstract heap sets.
over multiset formulas, and then, we analyze the program in a reduced product of the two domains \((\mathcal{A}_{HS}(\mathcal{A}_U) \times \mathcal{A}_{HS}(\mathcal{A}_M))\) using \text{strengthen}^{HS}.

Also, in order to increase the scalability of the analysis with the domain of abstract heap sets over universally quantified formulas, \(\mathcal{A}_{HS}(\mathcal{A}_U)\), each procedure should be analyzed with a different set of patterns. Carrying around more patterns than needed increases the size of the manipulated formulas (the formulas have many useless conjuncts irrelevant for the procedure specification) and consequently, the complexity of the analysis. This means that during the analysis, at procedure calls, we must use an operator called \text{convert}, that transforms a formula in \(\mathcal{A}_U(F)\), parametrized by some set of patterns \(P\), to a formula parametrized by another set of patterns \(P_1\) or \(P_2\) as shown in Figure 6.32.

At procedure returns, we must do the reverse: transform a formula in \(\mathcal{A}_U(P_1)\) or \(\mathcal{A}_U(P_2)\) to a formula in \(\mathcal{A}_U(F)\).

![Figure 6.32: Compositional analysis with patterns.](image)

The two problems exposed above have in common the task of computing the intersection (or, an over-approximation of the intersection) between abstract elements from different abstract domains. Therefore, we define a partial reduction operator \text{strengthen}^{HS} that takes as argument two elements from two potentially different abstract domains (e.g., heaps sets over first-order formulas and heap sets over multiset constraints, or heap sets over first-order formulas built with different sets of patterns) and returns an over-approximation of their intersection. This operator is defined as a fixpoint computation of a simpler partial reduction operator, based on a mechanism of unfolding and folding on words.

### 6.7.1 Motivation

Our main objective is to compute procedure summaries in the abstract domain where words are described by universally-quantified formulas. The fact that we consider a semantics based on local heaps raises some issues with respect to the precision of the analysis. We describe them on the computation of the procedure summary for \text{quicksort} and an initial configuration which contains an arbitrary list.

Let us take a closer look to the analysis of the \text{quicksort} procedure from Figure 4.21 with the domain of universally-quantified formulas, \(\mathcal{A}_{HS}(\mathcal{A}_U)\), over the set of guard patterns

\[
[y] \in tl(\omega), \quad [y_1, y_2] \in tl(\omega) \land y_1 \leq y_2, \\
[y_1] \in tl(\omega_1) \land [y_2] \in tl(\omega_2) \land y_1 = y_2.
\]

The analysis manipulates universally-quantified implications where the left part is one
6.7. COMBINING ABSTRACT DOMAINS

Figure 6.33: Sequence of summaries computed for the procedure quicksort

of the formulas above, where the ω’s are data word variables. Typically, in the case of recursive procedures, the analysis starts by computing procedure summaries for input lists of length 0 and then, for input lists of length 1, and so on until it reaches a fixpoint (to terminate it applies the widening operator).

We recall that we consider an implementation of split that allocates memory for the lists left and right, which contain copies of the elements of a.

The analysis first synthesizes the summary corresponding to the case when the input list is empty. It is given in Figure 6.33(a). Then, the summary for lists of length one is obtained by using $\psi_{\text{sum,0}}^{U}$ when calling “left=quicksort(left)” and “right=quicksort(right)”. The context of these calls is an abstract heap where both left and right point to null. The resulting summary is given in the upper part of Figure 6.33(b). It contains the property that the only value in the input list equals the only value in the output list ($\text{hd}(m_{\text{res}}) = \text{hd}(m_{a})$). By applying the join operator between this summary and the one for lists of length 0 we obtain the abstract heap set $\psi_{\text{sum,}\leq 1}^{U}$ in Figure 6.33(b). This abstract heap set represents the summary of quicksort for input lists of length at most 1.

The next step is to compute a summary for lists of length two. For this, the previously computed summaries are used when returning from the recursive calls “left=quicksort(left)” and “right=quicksort(right)” in the context where either (1) left points to null and right points to a list of length 1 whose value is greater than the pivot, or (2) right point to null and left points to a list of length 1 whose value is less than or equal to the pivot. The words associated to the nodes labeled by left and right are of length 1 and consequently, there are no universal formulas to manipulate during the calls. When returning from the two recursive calls, the relation between the pivot and the value in the list pointed to by left or right is preserved. Therefore, when returning from concat and assigning left, right, and pivot to null, the sortedness property is generated by fold# (the transformer that removes simple nodes). The new summary is joined to $\psi_{\text{sum,}\leq 1}^{U}$ and the result is given in Figure 6.33(c).
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

ψ U T := eq(na, na0) ∧ 1 ≤ len(na) ∧

hd(na0) ≤ hd(np) ∧ hd(np) < hd(nr)∧
∀y. [y] ∈ tl(na0) ⇒ na0[y] ≤ hd(np)∧
∀y. [y] ∈ tl(nr) ⇒ hd(np) < nr[y]

ψ U sum := sorted(mres) ∧

eq(ma, ma0) ∧
1 ≤ len(ma) ∧ len(ma) = len(mres)

ψ U aux := eq(na, na0) ∧ 1 ≤ len(na) ∧

hd(ma0) ≤ hd(np) ∧ hd(np) < hd(na)∧
∀y. [y] ∈ tl(ma0) ⇒ ma0[y] ≤ hd(np)∧
∀y. [y] ∈ tl(na) ⇒ hd(np) < na[y]

sorted(mres) ∧ eq(ma, ma0) ∧
1 ≤ len(ma) ∧ len(ma) = len(mres)

ψ U := eq(na, na0) ∧ 1 ≤ len(na) ∧

hd(na0) < hd(np) ∧ hd(np) < hd(na)∧
∀y. [y] ∈ tl(na) ⇒ hd(np) < na[y] ∧
sorted(mres)

(a) The heap \( \tilde{H}_U \) representing the context of the call.

(b) The heap \( \tilde{H}_U \) representing the procedure summary.

(c) The heap \( \tilde{H}_{aux} \).

(d) The heap \( \tilde{H}_{U \#} \) corresponding to the procedure return \( \text{return left = quicksort(left)} \) from the procedure quicksort in Figure 4.21.

Figure 6.34: Applying the transformer of \( A_{HS}(A_U) \) corresponding to the procedure return \( \text{left = quicksort(left)} \) from the procedure quicksort in Figure 4.21.
6.7. COMBINING ABSTRACT DOMAINS

Let us apply an inductive reasoning where we assume that we have obtained the summary \( \tilde{H}^U \) given in Figure 6.34(b) (the heap containing only \( \# \) is omitted), which says that the output list is sorted, and we want to prove that the computation stabilizes with this summary. Therefore in the next iteration, the context of the first recursive call “\( \text{left} = \text{quicksort}(\text{left}) \)” is given by the graph and the formula \( \psi^U_c \) in Figure 6.34(a).

The abstract transformer for procedure return computes first the intermediary abstract heap given in Figure 6.34(c), by matching the local graph of the call with the input graph of the summary (\( \tilde{a}, a^0 \) and \( res \) denote the input and output formal parameters of the summary). The formula \( \psi^U_{aux} \) is obtained by applying \( \text{combine} \) between \( \psi^U_c \) and \( \psi^U_{sum} \), which takes the intersection (\( \cap^U \)) of the two formulas and substitutes \( n_l \) with \( m^0_a \) (in \( \psi_c \)). Then, \( \text{left} \) is assigned to the node \( m_{res} \) labeled by the output parameter \( res \) and all pointer variables in the summary are eliminated. Finally, the garbage collector is applied. Since the nodes \( m_a \) and \( m^0_a \) are not reachable from any program variable, the garbage collector calls \( \text{proj}(m_a, m^0_a, \tilde{H}^U_{aux}) \) which removes them from the abstract heap.

The formula \( \psi^U_r \) is obtained by applying \( \text{proj} \) between \( m_a, m^0_a, \psi^U_c \) and \( \psi^U_{sum} \).

This projection will remove all the constraints on \( m_a \) and \( m^0_a \) which represent the initial and current value of the actual input parameter. Therefore, \( \psi^U_r \) lost the property that all the elements of \( \text{left} \) are less than the pivot. The same holds for the list pointed to by \( \text{right} \) and the second recursive call. The relation obtained after the two calls is given in Figure 6.35. We have lost the property that all the elements of \( \text{left} \) (respectively \( \text{right} \)) are less than or equal to (respectively greater than) the pivot. Consequently, after calling \( \text{concat} \), and assigning \( \text{left}, \text{pivot}, \) and \( \text{right} \) to \( \text{null} \), we can’t obtain that the list pointed to by \( \text{res} \) is sorted.

Figure 6.35: The relation synthesized at line 29 of \( \text{quicksort} \) with \( A_{HS}(A_U) \)

In this section, we give a solution for this problem based on a combination of abstract analyses. More precisely, we define a partial reduction operator called \( \text{strengthen}^{HS} \) between \( A_{HS}(A_U) \) and \( A_{HS}(A_M) \), and we analyze \( \text{quicksort} \) in a partial reduced product \( A_{HS}(A_U) \times A_{HS}(A_M) \).
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

ψ^M_{c} ::= \text{ms}(n_{a}) = \text{ms}(n_{0}^{a}) \land \\
\text{ns}(n_{l}) \cup \text{ms}_{hd}(n_{p}) \cup \text{ms}(n_{r}) = \text{ms}(n_{a})

ψ^M_{\text{sum}} ::= \text{ms}(m_{a}) = \text{ms}(m_{0}^{a}) \land \\
\text{ns}(m_{a}) = \text{ms}(m_{\text{res}})

ψ^M_{\text{aux}} ::= \text{ms}(n_{a}) = \text{ms}(n_{0}^{a}) \land \\
\text{ns}(m_{a}) = \text{ms}(m_{0}^{a}) \land \text{ms}(m_{a}) = \text{ms}(m_{\text{res}})

(a) The heap $\tilde{H}^M_{c}$ representing the context of the call.
(b) The heap $\tilde{H}^M_{\text{sum}}$ representing the procedure summary.
(c) The heap $\tilde{H}^M_{\text{aux}}$.

Figure 6.36: Applying the transformer of $A_{\text{HS}}(A_{M})$ corresponding to the procedure return
return left = quicksort(left) from the procedure quicksort in Figure 4.21

232
6.7. COMBINING ABSTRACT DOMAINS

We slightly modify the definition of the abstract transformer for return statements such that, it applies strengthen$_{HS}$ to strengthen the universally quantified formulas associated with $H^U_{aux}$ using the multiset constraints, before eliminating the nodes $m_a$ and $m'_a$. Therefore, suppose that $(\tilde{H}_x^U, \tilde{H}_x^M)$ is the summary obtained so far by the analysis in $A_{HS}(A_U) \times A_{HS}(A_M)$, where Figure 6.36(b) shows the summary $\tilde{H}_x^M$ computed using the multiset domain $(\tilde{H}_x^M$ contains the fact that the multiset of data in the output list equals the multiset of data in the input list). Suppose that we want to apply the abstract transformer for the procedure return $\text{return left} = \text{quicksort(left)}$ in the context $(\tilde{H}_c^U, \tilde{H}_c^M)$, where $\tilde{H}_c^M$ is given in Figure 6.36(a). Recall that during the analysis with $A_{HS}(A_U)$, the relation between the elements of the list pointed to by left and the pivot is lost when the projection operator is applied. Let $\tilde{H}_x^U$ and $\tilde{H}_x^M$ be the abstract heaps obtained from $\tilde{H}_{aux}^U$ and $\tilde{H}_{aux}^M$ by assigning left to res and removing the labels $\pi$ and $\sigma^0$. Then, before applying $\text{proj}(m_a, m'_a, (\tilde{H}_s^U, \tilde{H}_s^M))$ (during garbage collection), the partial reduction operator is used to strengthen the two abstract elements using the information collected by the other domain. That is, the garbage collector applies $\text{proj}(m_a, m'_a, (\tilde{H}_s^U, \tilde{H}_s^M))$, where

$$
(\tilde{H}_s^U, \tilde{H}_s^M) = \text{strengthen}_{HS}(\tilde{H}_x^U, \tilde{H}_x^M).
$$

Roughly, to recover the property that all the elements of the word associated to $m_{res}$ are less than the pivot, strengthen$_{HS}$ uses a partial reduction operator strengthen$_{M}$ defined over the data words abstract domains $A_U$ and $A_M$.

Thus, at the return from the first recursive call, the abstract transformer computes

$$
\text{strengthen}_M\left(\psi^U_c[n_t \mapsto m^0_a] \mathbin{\exists}^\psi^U \psi^M_{\text{sum}}, \psi^M_c[n_t \mapsto m^0_a] \mathbin{\exists}^\psi^M \psi^M_{\text{sum}}\right). \tag{6.7.1}
$$

The operator strengthen$_M$ returns the universally quantified formula:

$$
(\psi^U_c[n_t \mapsto m^0_a] \mathbin{\exists}^\psi^U \psi^M_{\text{sum}}) \land \text{hd}(m_{res}) \leq \text{hd}(n_p) \land \forall y. [y] \in \text{tl}(m_{res}) \implies m_{res}[y] \leq \text{hd}(n_p),
$$

and consequently, by eliminating the constraints on $m_a$ and $m'_a$, the new formula $\psi^U_c$ preserves the fact that the elements of the list pointed to by left are smaller than or equal to the pivot. Similarly, after the second recursive call, the analysis is able to maintain the fact that the elements of the list pointed to by right are greater than the pivot. Now, after calling concat, and assigning left, pivot, and right to null, we obtain that the list pointed to by res is sorted which proves that the fixpoint computation stabilizes with the expected result.

6.7.2 General framework for combining analyses

In order to increase the precision, we analyze programs in a partially reduced product [[Cousot 1979]] between $A_{HS}(A_U)$ and $A_{HS}(A_M)$. This analysis uses a partial reduction operator strengthen$_{HS}$ which is parametrized by a partial reduction operator [[Cousot 1979]] between the $D \times W$-domains $A_U$ and $A_M$.

The operator strengthen$_{HS}$ actually an instance of a more general one, denoted strengthen$_{HS} : A_{HS}(A_U) \times A_{HS}(A_W) \rightarrow A_{HS}(A_U) \times A_{HS}(A_W)$. This operator is parametrized by a partial reduction operator over the $D \times W$-domains $A_U$ and $A_W$. Then, strengthen$_{HS}$ corresponds to strengthen$_{HS}$ when $W = M$ and strengthen$_{HS}$ corresponds to strengthen$_{HS}$ when $W = U$. 

233
6.7.2.1 Definition of strengthen

The operator

\[ \text{strengthen}^\text{HS} : (A^\text{HS}(A_U) \times A^\text{HS}(A_W)) \rightarrow (A^\text{HS}(A_U) \times A^\text{HS}(A_W)) \]

propagates information between the two abstract domains and, like any partial reduction operator, for any \(H_1 \in A^\text{HS}(A_U)\) and \(H_2 \in A^\text{HS}(A_W)\), the corresponding output \(\text{strengthen}^\text{HS}(H_1, H_2) = (H_1^*, H_2^*)\) should satisfy the following properties:

1. \(H_1^* \subseteq \text{HS} H_1, \ H_2^* \subseteq \text{HS} H_2, \text{and} \)
2. \(\gamma^U((H_1^*) \cap \gamma^W(H_2^*)) = \gamma^U(H_1) \cap \gamma^W(H_2). \)

Let \(\text{HS}_1 \in A^\text{HS}(A_U), \text{HS}_2 \in A^\text{HS}(A_W), \) and \(\cup^\text{HS}\) be the component-wise extension of \(\cup\) to pairs of elements (i.e., \(\{(X, Y)\} \cup^\text{HS} \{(X', Y')\} = (X \cup^\text{HS} X', Y \cup^\text{HS} Y')\)), for any \(X, X'\) in \(A^\text{HS}(A_U)\) and \(Y, Y'\) in \(A^\text{HS}(A_W)\). We define the operator \(\text{strengthen}^\text{HS}(\text{HS}_1, \text{HS}_2)\) on abstract heap sets, by applying \(\cup^\text{HS}\) between \(\text{strengthen}^\text{HS}(\text{HS}_1, \text{HS}_2)\) for any two abstract heaps \(\text{HS}_1, \text{HS}_2\):

\[ \text{strengthen}^\text{HS}(\text{HS}_1, \text{HS}_2) = \bigsqcup_{\text{HS}_1, \text{HS}_2} \text{strengthen}^\text{HS}(\text{HS}_1, \text{HS}_2). \]

The definition of \(\text{strengthen}^\text{HS}\) over abstract heaps uses a partial reduction operator over \(D^W\)-domains: \(\text{strengthen}^\text{W} : A^W \times A^W \rightarrow A^W \times A^W\). For any two abstract heaps \(\text{HS}_1 \in A^\text{HS}(A_U)\) and \(\text{HS}_2 \in A^\text{HS}(A_W)\) which are not isomorphic, we define \(\text{strengthen}^\text{HS}(\text{HS}_1, \text{HS}_2) = (\text{HS}_1, \text{HS}_2)\). Otherwise, let \(\text{HS}_1 = (N, S, L, \varphi_1) \in A^\text{HS}(A_U)\) and \(\text{HS}_2 = (N, S, L, \varphi_2) \in A^\text{HS}(A_W)\) be two isomorphic abstract heaps (we suppose that the nodes related by the isomorphism have the same name). We define

\[ \text{strengthen}^\text{HS}(\text{HS}_1, \text{HS}_2) = ((N, S, L, \text{strengthen}^\text{W}(\varphi_1, \varphi_2) |_1), (N, S, L, \text{strengthen}^\text{W}(\varphi_1, \varphi_2) |_2)), \]

where \(\text{strengthen}^\text{W}(\text{HS}_1, \text{HS}_2) |_i\) is the \(i\)th component of \(\text{strengthen}^\text{W}(\text{HS}_1, \text{HS}_2)\), for any \(1 \leq i \leq 2\).

Proposition 6.7.1. If \(\text{strengthen}^\text{W}\) is a partial reduction operator over two \(D^W\)-domains \(A_U\) and \(A_W\) then, \(\text{strengthen}^\text{HS}\) is also a partial reduction operator between the domains of abstract heaps sets over \(A_U\) and \(A_W\).

Proof. Let \((\text{HS}_1^*, \text{HS}_2^*) = \text{strengthen}^\text{HS}(\text{HS}_1, \text{HS}_2)\). For any \(\text{HS}_1^* \in \text{HS}_1^*,\) either (1) there exists only one abstract heap \(H \in \text{HS}_1 \cup \text{HS}_2\) such that \(H\) is isomorphic to \(H_1^*\) or (2) there exist two abstract heaps \(H_1 = (N, S, L, \varphi_1) \in \text{HS}_1\) and \(H_2 = (N, S, L, \varphi_2) \in \text{HS}_2\)
6.7. COMBINING ABSTRACT DOMAINS

isomorphic to \( \tilde{H}_1^s \). By definition, in the first case we have that \( \tilde{H}_1^s = \tilde{H} \) and in the second case we have that

\[
\tilde{H}_1^s = (N, S, L, \text{strengthen}_W(\varphi_1, \varphi_2) \mid 1)
\]

The fact that \( \text{strengthen}_W \) is a partial reduction operator implies that for any \( \tilde{H}_1^s = (N, S, L, \varphi_1) \in HS_1^s \), there exists \( \tilde{H}_1 = (N, S, L, \varphi_1) \in HS_1 \) isomorphic to \( \tilde{H}_1^s \) such that \( \varphi_1 \subseteq U \varphi_1 \). Consequently, \( HS_1^s \subseteq HS \). Similarly, it can be proved that \( HS_2^s \subseteq HS \).

To prove the equality between the intersection of the concretizations,

\[
\gamma^{HS}(\tilde{H}_1^s) \cap \gamma^{HS}(\tilde{H}_2^s) = \gamma^{HS}(\tilde{H}_1) \cap \gamma^{HS}(\tilde{H}_2), \tag{6.7.2}
\]

we use the fact that, by the definition of \( \gamma^{HS} \),

\[
\gamma^{HS}(\tilde{H}_1^s) \cap \gamma^{HS}(\tilde{H}_2^s) = \bigcup_{\tilde{H}_1^s, \tilde{H}_2^s} \gamma(\tilde{H}_1^s) \cap \gamma(\tilde{H}_2^s) \quad \text{and}
\gamma^{HS}(\tilde{H}_1) \cap \gamma^{HS}(\tilde{H}_2) = \bigcup_{\tilde{H}_1, \tilde{H}_2} \gamma(\tilde{H}_1) \cap \gamma(\tilde{H}_2).
\]

The equality in (6.7.2) follows from the fact that, for any \( \tilde{H}_1 \in \tilde{H}S_1 \) and \( \tilde{H}_2 \in \tilde{H}S_2 \),

\[
\gamma(\tilde{H}_1) \cap \gamma(\tilde{H}_2) = \gamma(\tilde{H}_1^s) \cap \gamma(\tilde{H}_2^s),
\]

where \( \tilde{H}_1^s \), resp. \( \tilde{H}_2^s \), is the unique abstract heap in \( \tilde{H}S_1^s \), resp. \( \tilde{H}S_2^s \), isomorphic to \( \tilde{H}_1 \), resp. \( \tilde{H}_2 \).

\( \Box \)

6.7.2 Combined analysis between \( A_{HS}(A_U) \) and \( A_{HS}(A_M) \) using \( \text{strengthen}_{HS}^{HS} \)

The elements of the partial reduced product are pairs from \( A_{HS}(A_U) \times A_{HS}(A_M) \). Almost all the abstract transformers in this analysis are defined by:

\[
U^{\#}_{H,S,U}[a](\tilde{H}S, \tilde{H}S') = (U^{\#}_{H,S,U}[a](\tilde{H}S), U^{\#}_{H,S,M}[a](\tilde{H}S')),
\]

where \( U^{\#}_{H,S,U}[a] \) is the abstract transformer in \( A_{HS}(A_U) \) and \( U^{\#}_{H,S,M}[a] \) is the abstract transformer in \( A_{HS}(A_M) \).

The only exceptions are the \textit{assert} statements and the statements \( a \) such that the definition of \( U^{\#}_{H,S,U}[a] \) or \( U^{\#}_{H,S,M}[a] \) (respectively, \( U^{\#}_{H,S,M}[a] \) or \( U^{\#}_{H,M}[a] \)) contains a call to the procedure \( \text{proj}(D, N, H) \) that removes a set of nodes \( N \) and a set of data variables \( D \) from an abstract heap \( H \) (the procedure \( \text{proj} \) calls \( \text{proj}_P \) or \( \text{proj}_M \) depending on the \( D\text{W} \) domain of \( H \) in order to remove the data word variables \( N \) and the program data variables \( D \) from the data constraint associated to \( H \)). These statements are \( p = \text{null}, \ p->\text{next} = \text{null} \), procedure calls and procedure returns. Notice that the call to \( \text{proj} \) does not depend on the \( D\text{W} \) domain used in the analysis. Consequently, we assume that the transformers for these statements over abstract heaps \( U^{\#}_{H,U}[a] \) and \( U^{\#}_{H,M}[a] \), respectively

\[
U^{\#}_{H,U}[a] = F_3 \circ \text{proj}_P \circ F_4 \quad \text{and} \quad U^{\#}_{H,M}[a] = F_3' \circ \text{proj}_P \circ F_4'.
\]

235
 CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

Under this assumption, we define \( U^\#_{\mathcal{M}}[a] \) and \( U^\#_{\mathcal{H}}[a] \), the transformers in the reduced product \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_{\mathcal{U}}) \times \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_{\mathcal{M}}) \) by (1) applying \( F_2 \) and \( F_3 \) on the inputs, (2) applying \( \text{strengthen}^\mathcal{M}_{\mathcal{HS}} \) on the output of the previous step, (3) applying \( \text{proj} \) in \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_{\mathcal{U}}) \) (respectively, \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_{\mathcal{M}}) \)) on the first (respectively, the second) element of the pair outputted in the previous step, and (4) applying \( F_1 \) and \( F_1 \) on the result of the previous step. Formally, we define:

\[
\begin{align*}
(\tilde{H}_1, \tilde{H}_1') &= (F_2(\tilde{H}_2), F_2'(\tilde{H}_2')) \\
(\tilde{H}_2, \tilde{H}_2') &= \text{strengthen}^\mathcal{M}_{\mathcal{HS}}(\tilde{H}_1, \tilde{H}_1') \\
(\tilde{H}_3, \tilde{H}_3') &= (\text{proj}(\tilde{H}_2), \text{proj}(\tilde{H}_2')) \\
U^\#_{\mathcal{H}}[a](\tilde{H}_1, \tilde{H}_1') &= (F_1(\tilde{H}_3), F_1'(\tilde{H}_3')) \\
U^\#_{\mathcal{M}}[\psi](\tilde{H}_1, \tilde{H}_2) &= (\tilde{H}_1, \tilde{H}_2), \text{ where} \\
(\tilde{H}_1, \tilde{H}_2) &= (U^\#_{\mathcal{HS}, U}[\psi](\tilde{H}_1'), U^\#_{\mathcal{HS}, M}[\psi](\tilde{H}_2')) \\
(\tilde{H}_1, \tilde{H}_2) &= \text{strengthen}^\mathcal{M}_{\mathcal{HS}}(\tilde{H}_1, \tilde{H}_2).
\end{align*}
\]

We do not apply the partial reduction operator after each step for efficiency reasons.

6.7.2.3 Combined analysis using \( \mathcal{DW} \)-domains \( \mathcal{A}_{\mathcal{U}} \) over different sets of patterns

Let \( \text{Pr} \) be a program and let \( P_i \) with \( 1 \leq i \leq q \) be all the procedures defined in \( \text{Pr} \). To each procedure \( P_i \) we associate a set of patterns \( \mathcal{P}_i \) and we design an analysis over \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_{\mathcal{U}}) \), where \( \mathcal{A}_{\mathcal{U}} \) is parametrized by all the patterns in \( \mathcal{P} = \bigcup_{1 \leq i \leq q} \mathcal{P}_i \), such that the formulas from the abstract heaps associated to a control point of the procedure \( P_i \) are built over the set of patterns \( \mathcal{P}_i \). We modify the definition of the abstract transformers as follows:

1. the procedures \( \text{fold} \) and \( \text{unfold} \), when used in the abstract transformer corresponding to a statement of the procedure \( P_i \), use the transformers \( \text{concat}^\mathcal{P}_i \) and \( \text{split}^\mathcal{P}_i \) that output formulas built over the patterns in \( \mathcal{P}_i \) (instead of applying \( \text{concat}^\# \) and \( \text{split}^\# \) that output formulas built over all the patterns in \( \mathcal{P} \)),

2. consider a call to the procedure \( P_2 \) made from the procedure \( P_1 \) in some context \( \tilde{H}_1^1 \in \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_{\mathcal{U}}(\mathcal{P}_1)) \). The transformer for procedure calls (see Section 6.3.2) (1) first, converts the context \( \tilde{H}_1^1 \) into an abstract element in \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_{\mathcal{U}}(\mathcal{P}_2)) \) that uses the patterns in \( \mathcal{P}_2 \), denoted \( \tilde{H}_2^c \) and then, (2) applies the transformer for procedure calls defined previously on the abstract heap set \( \tilde{H}_2^c \). That is, the transformer for a procedure call \( \text{call} P_2(\text{ai, ao}) \) in the context \( \tilde{H}_2^c \) is defined by:

\[
U^\#_{\mathcal{HS}}[\text{call} P_2(\text{ai, ao})](\text{convert}[\mathcal{P}_1 \to \mathcal{P}_2](\tilde{H}_1^1)),
\]

where \( \text{convert}[\mathcal{P}_1 \to \mathcal{P}_2] \) is an operator that transforms any abstract value \( A_1 \) in \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_{\mathcal{U}}(\mathcal{P}_1)) \) over a set of patterns \( \mathcal{P}_1 \) into an abstract value \( A_2 \) in a domain \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_{\mathcal{U}}(\mathcal{P}_2)) \) over a set of patterns \( \mathcal{P}_2 \neq \mathcal{P}_1 \).
6.7. COMBINING ABSTRACT DOMAINS

3. consider again the call to the procedure $P_2$ made from the procedure $P_1$ in some context $\tilde{\mathcal{S}}^1 \in \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_1))$. Also, let $\tilde{\mathcal{S}}^2 \in \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_2))$ be the summary associated to this procedure call. Then, the transformer for procedure returns (see Section 6.3.2) (1) first, converts the summary $\tilde{\mathcal{S}}^2$ into an abstract element in $\mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_1))$ that uses the patterns in $\mathbb{P}_1$, denoted $\tilde{\mathcal{S}}^1$, and then, (2) applies the previously defined transformer for procedure returns on the abstract heap sets $\tilde{\mathcal{S}}^1$ and $\tilde{\mathcal{S}}^2$. Formally, the transformer for a procedure return $\text{return } P_2(\mathbf{ai}, \mathbf{ao})$ in the context $\tilde{\mathcal{S}}^1$ with the procedure summary $\tilde{\mathcal{S}}^2$ is defined by:

$$U^\#_{\mathbb{H}}[\text{return } P_2(\mathbf{ai}, \mathbf{ao})](\tilde{\mathcal{S}}^1, \text{convert}[P_2 \rightarrow P_1](\tilde{\mathcal{S}}^2)),$$

where $\text{convert}[P_2 \rightarrow P_1]$ transforms any abstract value in $\mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_2))$ into an abstract value in $\mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_1))$.

We recall that the procedure $\text{unfold}$ is used within the abstract transformer for the statement $q = p->\text{next}$ and the procedure $\text{fold}$ is used within the abstract transformers for the statements $p = \text{null}$, $p->\text{next} = \text{null}$, and procedure returns.

For any pair of sets of patterns $(P_1, P_2)$ we define an operator

$$\text{convert}[P_1 \rightarrow P_2] : \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_1)) \rightarrow \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_2))$$

such that, for any $\tilde{\mathcal{S}}^1 \in \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_1))$, $\text{convert}[P_1 \rightarrow P_2](\tilde{\mathcal{S}}^1)$ is an over-approximation of $\tilde{\mathcal{S}}^1$ in the domain $\mathcal{A}_{\mathbb{U}}(P_2)$. Intuitively, $\text{convert}[P_1 \rightarrow P_2](\tilde{\mathcal{S}}^1)$ returns an abstract heap set $\tilde{\mathcal{S}}^2 \in \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_2))$ which contains (1) constraints from $\tilde{\mathcal{S}}^1$ using the patterns in $P_1 \cap P_2$ and (2) constraints using the patterns in $P_2 \setminus P_1$ implied by $\tilde{\mathcal{S}}^1$. Formally, we define $\text{convert}$ based on a partial reduction operator $\text{strengthen}_{\mathbb{U}} : \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_1)) \times \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_2)) \rightarrow \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_1)) \times \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_2))$:

$$\text{convert}[P_1 \rightarrow P_2](\tilde{\mathcal{S}}^1) = \text{strengthen}_{\mathbb{U}}(\tilde{\mathcal{S}}^1, \top_{\mathbb{H}}) \upharpoonright 2,$$  \hspace{1cm} (6.7.3)

where $\top_{\mathbb{H}}$ is the top element in $\mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_2))$ and $\text{strengthen}_{\mathbb{U}}(\tilde{\mathcal{S}}^1, \top_{\mathbb{H}}) \upharpoonright 2 \in \mathcal{A}_{\mathbb{H}}(\mathcal{U}(P_2))$ is the second element of the pair defined by $\text{strengthen}_{\mathbb{U}}(\tilde{\mathcal{S}}^1, \top_{\mathbb{H}})$.

The property that $\text{convert}[P_1 \rightarrow P_2](\tilde{\mathcal{S}}^1)$ is an over-approximation of $\tilde{\mathcal{S}}^1$, i.e., $\gamma(\tilde{\mathcal{S}}^1) \subseteq \gamma(\text{convert}[P_1 \rightarrow P_2](\tilde{\mathcal{S}}^1))$, follows from the fact that $\text{strengthen}_{\mathbb{U}}$ is a partial reduction operator.

6.7.3 A partial reduction operator between $\mathbb{D}^{\mathbb{W}}$-domains $\text{strengthen}_{\mathbb{W}}$

We define hereafter a partial reduction operator, $\text{strengthen}_{\mathbb{W}}$, between two different $\mathbb{D}^{\mathbb{W}}$-domains.

6.7.3.1 Definition of $\text{strengthen}_{\mathbb{W}}$

Given, $\mathcal{A}_{\mathbb{W}_1}$ and $\mathcal{A}_{\mathbb{W}_2}$ two $\mathbb{D}^{\mathbb{W}}$-domains, we define

$$\text{strengthen}_{\mathbb{W}} : (\mathcal{A}_{\mathbb{W}_1} \times \mathcal{A}_{\mathbb{W}_2}) \rightarrow (\mathcal{A}_{\mathbb{W}_1} \times \mathcal{A}_{\mathbb{W}_2})$$

$$\text{strengthen}_{\mathbb{W}}(\varphi_1, \varphi_2) = (\varphi_1, \varphi_2) \cap_{\mathbb{W}_1 \times \mathbb{W}_2} \text{infer}(\varphi_1, \varphi_2)$$

where $\text{infer} : \mathcal{A}_{\mathbb{W}_1} \times \mathcal{A}_{\mathbb{W}_2} \rightarrow \mathcal{A}_{\mathbb{W}_1} \times \mathcal{A}_{\mathbb{W}_2}$.
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

Let \( \varphi_1 \in A_{W_1} \) and \( \varphi_2 \in A_{W_2} \). Intuitively, \textit{strengthen}_w(\varphi_1, \varphi_2) adds to \( \varphi_1 \), respectively to \( \varphi_2 \), constraints which are implied by both \( \varphi_1 \) and \( \varphi_2 \). To this, it uses the procedure \textit{infer} which, for any two abstract values \( \varphi_1 \in A_{W_1} \) and \( \varphi_2 \in A_{W_2} \), returns an over-approximation of their conjunction expressed in \( A_{W_1} \) and \( A_{W_2} \), i.e. the concretization of each abstract element in the returned pair includes the intersection between the concretizations of \( \varphi_1 \) and \( \varphi_2 \).

\textbf{Definition 6.7.1 (\textit{infer}/\textit{strengthen}_w)}. Let \( A_{W_1} \) and \( A_{W_2} \) be two \( DW \)-domains and let \( \text{infer} : A_{W_1} \times A_{W_2} \rightarrow A_{W_1} \times A_{W_2} \) be a function such that for any \( \varphi_1 \in A_{W_1} \) and \( \varphi_2 \in A_{W_2} \) if \( \text{infer}(\varphi_1, \varphi_2) = (\varphi_1', \varphi_2') \) then
\[
(\gamma^{W_1}(\varphi_1) \cap \gamma^{W_2}(\varphi_2)) \subseteq \gamma^{W_1}(\varphi_1') \quad \text{and} \quad (\gamma^{W_1}(\varphi_1) \cap \gamma^{W_2}(\varphi_2)) \subseteq \gamma^{W_2}(\varphi_2').
\]

The function \textit{strengthen}_w : \( A_{W_1} \times A_{W_2} \rightarrow A_{W_1} \times A_{W_2} \) is defined by
\[
\text{strengthen}_w(\varphi_1, \varphi_2) = (\varphi_1 \cap^{W_1} \varphi_1', \varphi_2 \cap^{W_2} \varphi_2'),
\]
where \( (\varphi_1', \varphi_2') = \text{infer}(\varphi_1, \varphi_2) \).

\textbf{Theorem 6.7.1}. Given two \( DW \)-domains \( A_{W_1} \) and \( A_{W_2} \) that contain an exact meet operator and a function \( \text{infer} : A_{W_1} \times A_{W_2} \rightarrow A_{W_1} \times A_{W_2} \) like in Definition 6.7.1 \textit{strengthen}_w : \( A_{W_1} \times A_{W_2} \rightarrow A_{W_1} \times A_{W_2} \) is a partial reduction operator.

\textbf{Proof}. Let \( (\varphi_1, \varphi_2) \in A_{W_1} \times A_{W_2} \) and let \( \text{infer}(\varphi_1, \varphi_2) = (\varphi_1', \varphi_2') \). Then, according to Definition 6.7.1
\[
\text{strengthen}_w(\varphi_1, \varphi_2) = (\varphi_1 \cap^{W_1} \varphi_1', \varphi_2 \cap^{W_2} \varphi_2').
\]

Since \( \cap^{W_1} \) and \( \cap^{W_2} \) are (greatest) lower bounds, it follows that \( \varphi_1 \cap^{W_1} \varphi_1' \subseteq^{W_1} \varphi_1 \) and \( \varphi_2 \cap^{W_2} \varphi_2' \subseteq^{W_2} \varphi_2 \). To prove that \textit{strengthen}_w is a partial reduction operator we must also prove the equality of the intersections between concretizations, that is:
\[
\gamma^{W_1}(\varphi_1) \cap^{W_2} \gamma^{W_2}(\varphi_2) = \gamma^{W_1}(\varphi_1 \cap^{W_1} \varphi_1') \cap^{W_2} \gamma^{W_2}(\varphi_2 \cap^{W_2} \varphi_2'). \tag{6.7.5}
\]

Because the meet operator of \( A_{W_1} \) and \( A_{W_2} \) is exact,
\[
\gamma^{W_1}(\varphi_1 \cap^{W_1} \varphi_1') \cap^{W_2} \gamma^{W_2}(\varphi_2 \cap^{W_2} \varphi_2') = \gamma^{W_1}(\varphi_1) \cap^{W_1} \gamma^{W_1}(\varphi_1') \cap^{W_2} \gamma^{W_2}(\varphi_2) \cap^{W_2} \gamma^{W_2}(\varphi_2').
\]

Then, the definition of \textit{infer} \( (\varphi_1, \varphi_2) = (\varphi_1', \varphi_2') \) implies that
\[
\gamma^{W_1}(\varphi_1) \cap^{W_2} \gamma^{W_2}(\varphi_2) \subseteq \gamma^{W_1}(\varphi_1') \quad \text{and} \quad \gamma^{W_1}(\varphi_1) \cap^{W_2} \gamma^{W_2}(\varphi_2) \subseteq \gamma^{W_1}(\varphi_2').
\]

Therefore, we can conclude that (6.7.5) holds. Notice that, without intersecting the result of \textit{infer} with \( (\varphi_1, \varphi_2) \) the equality of the intersections between concretizations (of the input and the output values of \textit{strengthen}_w) is not guaranteed.

\hfill \Box

\textit{6.7.1 Remark}. Remark that the elements of all the \( DW \)-domains we have defined, \( A_U \), \( A_M \), and \( A_S \) are conjunctions of constraints. Therefore the meet operator of all these \( DW \)-domains is exact.
6.7. COMBINING ABSTRACT DOMAINS

In the following, we denote by $\text{infer}_U$ the infer function when $A_{W_1}$ is the domain of universally quantified formulas $A_U(P_1)$ over a set of patterns $P_1$ and $A_{W_2}$ is the domain of universally quantified formulas $A_U(P_2)$ over a set of patterns $P_2$. Also, we denote by $\text{infer}_M$ the infer function when $A_{W_1}$ is $A_U(P)$ over a set of patterns $P$ and $A_{W_2}$ is $A_M$.

We give a general definition for infer, for any two $D \times W$ domains $A_{W_1}$ and $A_{W_2}$, $\text{infer}_U$ and $\text{infer}_M$ being just two instances of this definition. We begin by two examples that demonstrate the computation of $\text{infer}_M$ and one example that demonstrates the computation of $\text{infer}_U$.

Example 6.7.1 (Computing $\text{infer}_M$). Consider the application of $\text{strengthen}_M(\varphi_1, \varphi_2)$ from the quicksort example, (6.7.1), where,

$$
\varphi_1 := \text{hd}(m_0^0) \leq \text{hd}(n_p) \land \text{hd}(n_r) > \text{hd}(n_p) \land \\
\text{len}(m_0^0) + \text{len}(n_r) + 1 = \text{len}(n_a) \land eq(n_a, n_0^0) \land \\
\forall y. [y] \in tl(n_r) \implies n_r[y] > \text{hd}(n_p) \land \\
\forall y. [y] \in tl(m_0^0) \implies m_0^0[y] \leq \text{hd}(n_p) \\
$$

$$
\varphi_2 := \text{ms}(n_a) = \text{ms}(n_0^0) \land \text{ms}(m_0^0) \cup \text{ms}(n_r) \cup \text{ms} \text{hd}(n_p) = \text{ms}(m_a) \land \\
\text{ms}(m_a) = \text{ms}(m_{res}) \land \text{ms}(m_a) = \text{ms}(m_0^0)
$$

(6.7.6)

For the purpose of this example, we simplify the arguments of $\text{strengthen}_M$. We consider that

$$
\varphi_1 := \text{hd}(m_0^0) \leq \text{hd}(n_p) \land \forall y. y \in tl(m_0^0) \implies n_l[y] \leq \text{hd}(n_p) \\
\varphi_2 := \text{ms}(m_{res}) = \text{ms}(m_0^0)
$$

The formula $\varphi_1$ says that all the elements of the word $m_0^0$ are less than $\text{hd}(n_p)$ and the formula $\varphi_2$ says that the words $m_0^0$ and $m_{res}$ have equal multisets of elements. These conjuncts are sufficient to infer the desired property. Adding the other ones only make the formulas more involved.

The result of $\text{strengthen}_M$ is computed using $\text{infer}_M$ applied on the same inputs. The inputs are constraints over unbounded length words. Roughly, to compute $\text{infer}_M$, we reduce the problem of over-approximating the intersection of constraints over unbounded words to the intersection of constraints over words of fixed length. To this, (1) we unfold a bounded length prefix $p_1$, resp. $p_2$, of the word $m_{res}$, respectively $m_0^0$, thus, $m_{res} = p_1 \cdot s_1$ and $m_0^0 = p_2 \cdot s_2$, where $\cdot$ denotes the concatenation of words and $s_1, s_2$ are possibly empty words over $\mathbb{Z}$ (2) we infer the properties of data in $p_1$ and $p_2$ implied by the conjunction between the constraint in the domain of universally-quantified formulas and the constraint in the multiset domain and (3) we fold the prefixes $p_1$ and $p_2$ and collect the information on these words using a universally-quantified formula. Then, we continue to apply the same transformations on the words $s_1$ and $s_2$ until we reach a fixpoint.

This unfolding/folding mechanism reduces the initial problem (of inferring universally-quantified constraints implied by the conjunction of the inputs) to the problem of inferring quantifier-free constraints implied by the conjunction of the inputs.

Some steps from the computation of $\text{infer}_M$ are given in Figure 6.37. The unfolding of a prefix of length 2 is given in Figure 6.37(a). Above each sub-word we give the positions from the initial word it contains. The sub-words are colored if their elements satisfy
the property that all the values they contain are less than \( \text{hd}(n_p) \). At this step, only the sub-words of \( m_0^i \) are colored. For any \( i \geq 0 \), the node \( m_i^1 \) denotes the singleton word corresponding to the position \( i \) of \( m_{res} \). Similarly, \( m_i^1 \) denotes the singleton word corresponding to \( m_0^i \) at position \( i \). On this unfolding we apply another partial reduction operator, denoted \( \sigma_M^M \), which deduces new properties on the unfolded prefix based on the multiset constraints. Here, it deduces that, for any \( 0 \leq i \leq 1 \),

\[
\text{ms}_{hd}(m_1^i) \cap \text{ms}_{hd}(m_0^i) \cap \text{ms}(s_1) = \text{ms}(m_{res})
\]

\[
\text{ms}_{hd}(m_1^0) \cup \text{ms}_{hd}(m_1^1) \cup \text{ms}(s_1) = \text{ms}(m_{res})
\]

\[
\text{ms}_{hd}(m_0^0) \cup \text{ms}_{hd}(m_1^1) \cup \text{ms}(s_2) = \text{ms}(m_{res})
\]

\[
\text{ms}_{hd}(m_0^1) \cup \text{ms}_{hd}(m_1^1) \cup \text{ms}(s_2) = \text{ms}(m_{res})
\]

\[
\text{ms}(p_1) \cup \text{ms}_{hd}(m_1^2) \cup \text{ms}_{hd}(m_0^1) \cup \text{ms}(s_1) = \text{ms}(m_{res})
\]

\[
\text{ms}(p_2) \cup \text{ms}_{hd}(m_2^1) \cup \text{ms}_{hd}(m_1^1) \cup \text{ms}(s_2) = \text{ms}(m_{res})
\]

\[
\text{ms}(p_1) \cup \text{ms}(s_1) = \text{ms}(m_{res})
\]

\[
\text{ms}(p_2) \cup \text{ms}(s_2) = \text{ms}(m_{res})
\]

\[
\text{ms}_{hd}(m_1^1) \in \text{ms}(m_0^i) \land \text{hd}(m_0^i) \leq \text{hd}(n_p) \land \forall y. [y] \in \text{tl}(m_0^y) \implies m_0^y[y] \leq \text{hd}(n_p).
\]

The result of applying \( \sigma_M^M \) is given in Figure 6.37(b). Now, we can apply a folding operation in the two abstract domains whose result is given in Figure 6.37(c). As a result we obtain that all the values in the prefix \( p_1 \) are less than \( \text{hd}(n_p) \). Then, we continue by unfolding another prefix of length 2 from the sub-words of \( m_{res} \) and \( m_0^0 \) that start with the position 2 (this is pictured in Figure 6.37(d)) (we unfold the first two positions of \( s_2 \)). We repeat these steps, by continuing the traversing of \( m_{res} \) and \( m_0^0 \), and applying the partial reduction operator \( \sigma_M^M \) after each unfolding, until the fixed-point computation terminates. The result will be:

\[
\text{infer}_M(\varphi_1, \varphi_2) = \left( \varphi_1 \land \text{hd}(m_{res}) \leq \text{hd}(n_p) \land \forall y. [y] \in \text{tl}(m_{res}) \implies m_{res}[y] \leq \text{hd}(n_p), \varphi_2 \right).
\]

Because \( \text{infer}_M \) keeps all the constraints from the input formulas, \( \text{strengthen}_M(\varphi_1, \varphi_2) \) is exactly \( \text{infer}_M(\varphi_1, \varphi_2) \).

**Example 6.7.2** (Computing \( \text{infer}_M \)). Let \( A_U(P) \) be a domain with universal formulas parametrized by a set of patterns \( P \) and \( A_M \) be the multiset abstract domain. Like previously, based on the principle of folding and unfolding, we iterate the definition of a
6.7. COMBINING ABSTRACT DOMAINS

basic partial reduction operator $\sigma^M$ defined over these two abstract domains, in order to over-approximate the intersection of two constraints. In this example, infer$_M$ soundly translates a universally-quantified formula into a multiset formula. Let

$$\text{P} = \{[y_1] \in \text{tl}(\omega_1) \land [y_2] \in \text{tl}(\omega_2) \land y_1 = y_2\},$$

$n_1$, $n_2$ be two word variables and $\varphi_1 \in \text{A}_U(\text{P})$, $\varphi_2 \in \text{A}_M$ be two formulas over $n_1$ and $n_2$, where

$$\varphi_1 = \text{eq}(n_1, n_2) ::= \text{hd}(n_1) = \text{hd}(n_2) \land \text{len}(n_1) = \text{len}(n_2)$$

$$\forall y_1, y_2. [y_1] \in \text{tl}(n_1) \land [y_2] \in \text{tl}(n_2) \implies n_1[y_1] = n_2[y_2].$$

and

$$\varphi_2 = \text{true}$$

Let $(\varphi'_1, \varphi'_2)$ denote the result of infer$_M(\varphi_1, \varphi_2)$. Notice that $\varphi'_1 = \varphi_1$ because $\varphi_2$ is the trivial formula true. In the following we show the computation of the $A_M$ formula $\varphi'_2$. We unfold simultaneously a bounded-length prefix of $n$ and $n'$. For instance, if we unfold a prefix of length 2, the formula $\varphi_2$ remains unchanged (that is, it equals true) and $\varphi_1$ is transformed into a formula $\varphi_1'$ given by:

$$\varphi'_1 ::= \text{hd}(n[0]) = \text{hd}(n'[0]) \land \text{hd}(n[1]) = \text{hd}(n'[1]) \land \text{eq}(n_1', n_2'),$$

where (1) $n_1'$ and $n_2'$ are word variables, (2) $n_1[i]$, resp. $n_2[i]$, for any $0 \leq i \leq 1$, denote the singletons corresponding to the first two elements of $n_1$, resp. $n_2$ (i.e., $n_1 = n_1[0] \cdot n_1[1] \cdot n_1'$ and $n_2 = n_2[0] \cdot n_2[1] \cdot n_2'$). Also, $\text{eq}(n_1', n_2')$ denotes the fact that the formula eq true starts from the second position of the words $n_1$ and $n_2$. Then, we apply a partial reduction operator, $\sigma^M$, which deduces multiset constraints on the unfolded prefix, based on the constraints from the quantifier-free part of the universally quantified formula. The idea is that for every two word variables $m$ and $m'$ such that $\varphi'_U \subseteq U \text{hd}(m) = \text{hd}(m')$, we add to $\varphi_2$ the atomic formula $\text{ms}_{\text{hd}}(m') = \text{ms}_{\text{hd}}(m)$. The obtained formula is

$$\varphi'_2 ::= \text{ms}_{\text{hd}}(n[0]) = \text{ms}_{\text{hd}}(n'[0]) \land \text{ms}_{\text{hd}}(n[1]) = \text{ms}_{\text{hd}}(n'[1]).$$

Afterwards, we apply a folding operation and the initial words $n_1$ and $n_2$ become $p_1 \cdot n_1'$ and respectively, $p_2 \cdot n_2'$, where $p_1$ ($p_2$) represents the concatenation of the first 2 elements of $n_1$ ($n_2$). The multiset formula obtained after the folding operation is

$$\varphi_2 ::= \text{ms}(p_1) = \text{ms}(p_2).$$

As in the previous cases, by iterating these three steps into a fixedpoint computation that traverses all the elements of $n_1$ and $n_2$, we obtain that

$$\text{infer}(\varphi_1, \varphi_2) = (\varphi_1, \text{ms}(n_1) = \text{ms}(n_2)).$$

Consequently, strengthen$_M(\varphi_1, \varphi_2) = (\varphi_1, \text{ms}(n_1) = \text{ms}(n_2)).$

Example 6.7.3 (Computing infer$_U$). Let $\text{A}_U(\text{P}_1)$, resp. $\text{A}_U(\text{P}_2)$, be a domain with universal formulas parametrized by a set of patterns $\text{P}_1$, resp. $\text{P}_2$. Using a similar mechanism (of unfold/fold) we compute, for any $\varphi_1 \in \text{A}_U(\text{P}_1)$, an over-approximation of $\varphi_1$ in $\text{A}_U(\text{P}_2)$, denoted $\text{convert}(\text{P}_1 \rightarrow \text{P}_2)(\varphi_1)$. For example, let

$$\text{P}_1 = \{[y] \in \text{tl}(\omega), [y_1, y_2] \in \text{tl}(\omega) \land y_1 \leq y_2\}$$

$$\text{P}_2 = \{[y] \in \text{tl}(\omega) \land y = 1, [y] \in \text{tl}(\omega) \land y = \text{len}(\omega) - 1,$$

$$[y_1, y_2] \in \text{tl}(\omega) \land y_1 < y_2\},$$

241
and let \( \varphi_1 \) be the sortedness property

\[
\text{sorted}(n) ::= \forall y_1, y_2. ([y_1, y_2] \in \text{tl}(n) \land y_1 \leq y_2) \implies \text{hd}(n) \leq n[y_1] \leq n[y_2].
\]

Computing \( \text{convert} [P_1 \rightarrow P_2](\varphi) \) allows us to prove that \( \varphi_1 \) implies

\[
\text{sorted}^{<1} ::= \forall y_1, y_2. ([y_1, y_2] \in \text{tl}(n) \land y_1 < 1 y_2) \implies n[y_1] \leq n[y_2]
\]

using the entailment relation \( \subseteq^U \) defined in \( A_U(P_2) \).

More precisely, first, we use \( \text{convert} [P_1 \rightarrow P_2](\varphi_1) \) to translate \( \varphi_1 \) into a property, denoted \( \varphi_2 \), built over the patterns in \( P_2 \) such that \( \gamma^U(\varphi_1) \subseteq \gamma^U(\varphi_2) \) and then, we prove that \( \varphi_1 \) implies \( \text{sorted}^{<1} \) using the fact that \( \varphi_2 \subseteq^U \text{sorted}^{<1} \).

We define \( \text{convert} [P_1 \rightarrow P_2](\varphi_1) = \varphi_2 \), where \( \varphi_2 \) is computed using the \( \text{strengthen}_U \) operator, that is, \( (\varphi_2, \varphi_1) = \text{strengthen}_U(\top_U, \varphi_1) \). Notice that the properties of \( \text{strengthen}_U \) imply that \( \gamma_U(\varphi_1) \subseteq \gamma_U(\text{convert} [P_1 \rightarrow P_2](\varphi_1)) \).

To compute \( \text{infer}_U(\top_U, \varphi_1) \) (used to define \( \text{strengthen}_U(\top_U, \varphi_1) \)) we start by unfolding a bounded-length prefix of \( n \). For instance, if we unfold a prefix of length \( 3 \), the formula \( \top_U \) remains unchanged while \( \varphi_1 \) is transformed into a formula \( \varphi'_1 \) given by:

\[
\varphi'_1 ::= n[0] \leq n[1] \land n[1] \leq n[2] \land n[2] \leq n[3] \land \text{sorted}(n[3]),
\]

where \( \text{sorted}(n[3]) \) denotes the fact that the formula \( \text{sorted} \) is true starting from the third position of \( n \). Then, we apply a partial reduction operator, \( \sigma^U \), that takes the existential part of \( \varphi'_1 \) and adds it to the first argument of \( \text{infer}_U, \top_U \). Afterwards, we apply a folding operation and the initial word \( n \) becomes \( n' \cdot n'' \), where \( n' \) represents the concatenation of the first 3 elements of \( n \). The formula in \( A_U(P_2) \) becomes

\[
\varphi_2 ::= \forall y. ([y] \in \text{tl}(n') \land y = 1) \implies \text{hd}(n') \leq n'[y] \land
\quad \forall y_1, y_2. ([y_1, y_2] \in \text{tl}(n') \land y_1 < 1 y_2) \implies n'[y_1] \leq n'[y_2] \land
\quad \forall y. ([y] \in \text{tl}(n') \land y = \text{len}(n') - 1) \implies n'[y] \leq \text{hd}(n'') \land
\quad \forall y_1, y_2. ([y_1, y_2] \in \text{tl}(n'') \land y_1 < 1 y_2) \implies \text{true}.
\]

As in the previous case, by iterating these two steps into a fixpoint computation that traverses all the elements of \( n \), we obtain that \( \text{infer}_U(\top_U, \varphi_1) = (\varphi_2, \varphi_1) \), where

\[
\text{convert} [P_1 \rightarrow P_2](\varphi_1) = \varphi_2 ::= \forall y. ([y] \in \text{tl}(n) \land y = 1) \implies \text{hd}(n) \leq n[y]
\quad \land \forall y_1, y_2. ([y_1, y_2] \in \text{tl}(n) \land y_1 < 1 y_2) \implies n[y_1] \leq n[y_2].
\]

From the definition of \( \subseteq^U \) it follows that \( \text{convert} [P_1 \rightarrow P_2](\varphi_1) \subseteq^U \text{sorted}^{<1} \) which proves that \( \varphi_1 \) implies \( \text{sorted}^{<1} \).

**6.7.3.2 The procedure infer**

For every abstract value \( \varphi_1 \in A_{W_1} \) and \( \varphi_2 \in A_{W_2} \), the output of \( \text{infer} : A_{W_1} \times A_{W_2} \rightarrow A_{W_1} \times A_{W_2} \) is defined by the analysis of a program without procedures over an abstract domain which is a partially reduced product between \( A_{HS}(A_{W_1}) \) and \( A_{HS}(A_{W_2}) \). The definition of \( \text{infer} \) is parametrized by a partial reduction operator denoted \( \sigma^{HS(W)} \) on \( A_{HS}(A_{W_1}) \times A_{HS}(A_{W_2}) \). In the next section, we define two instances of these operators.
6.7. COMBINING ABSTRACT DOMAINS

6.7.3.2.1 Partially reduced product

The elements of the partially reduced product are pairs from \( \mathcal{A}_{HS}(A_{W_1}) \times \mathcal{A}_{HS}(A_{W_2}) \). The analysis computes an invariant for the reachable program configurations at each control point. Almost all the abstract transformers in this analysis, denoted by \( U_{HS}^\#[a] \), are defined by:

\[
U_{HS}^\#[a] \left( (\tilde{H}_1, \tilde{H}_2) \right) = (U_{HS,W_1}^\#[a](\tilde{H}_1), U_{HS,W_2}^\#[a](\tilde{H}_2)),
\]

where \( U_{HS,W_1}^\#[a] \) is the abstract transformer in \( \mathcal{A}_{HS}(A_{W_1}) \) and \( U_{HS,W_2}^\#[a] \) is the abstract transformer in \( \mathcal{A}_{HS}(A_{W_2}) \). The only exception is the statement \( p\sigma q \rightarrow \text{next} \) whose abstract transformer calls a partial reduction operator \( \sigma^{HS(W)} : \mathcal{A}_{HS}(A_{W_1}) \times \mathcal{A}_{HS}(A_{W_2}) \rightarrow \mathcal{A}_{HS}(A_{W_1}) \times \mathcal{A}_{HS}(A_{W_2}) \). This operator propagates information between the two abstract domains.

Thus, for \( a' := p\sigma q \rightarrow \text{next} \),

\[
U_{HS}^\#[a'] \left( (\tilde{H}_1, \tilde{H}_2) \right) = \sigma^{HS(W)}(U_{HS,W_1}^\#[a'](\tilde{H}_1), U_{HS,W_2}^\#[a'](\tilde{H}_2));
\]

6.7.3.2.2 Strengthening

We define a partial reduction operator \( \sigma^{HS(W)} : \mathcal{A}_{HS}(A_U) \times \mathcal{A}_{HS}(A_W) \rightarrow \mathcal{A}_{HS}(A_U) \times \mathcal{A}_{HS}(A_W) \) over abstract heap sets, parametrized by a partial reduced operator, denoted \( \sigma^W : A_U \times A_W \rightarrow A_U \times A_W \), over data words abstract domains. Its definition is similar to the one of \( \text{strengthen}^{HS} \). We use different names because \( \text{strengthen}^{HS} \) and \( \sigma^{HS(W)} \) are parametrized by different partial reduction operators over \( \mathbb{D}^W \)-domains and also because they are used in different contexts (\( \sigma^{HS(W)} \) is used to define \( \text{strengthen}^{HS} \)).

The operator \( \sigma^{HS(W)} \) applied to a pair of abstract heap sets \( (\tilde{H}_S, \tilde{H}_S') \) takes the join of \( \sigma^{HS(W)}(\tilde{H}, \tilde{H}') \), for any two abstract heaps \( \tilde{H} \in \tilde{H}_S \) and \( \tilde{H}' \in \tilde{H}_S' \):

\[
\sigma^{HS(W)}(\tilde{H}_S, \tilde{H}_S') = \bigcup_{\tilde{H} \in \tilde{H}_S} \sigma^{HS(W)}(\tilde{H}, \tilde{H}'),
\]

where \( \sqcup^{HS} \) is the extension of \( \sqcup \) to pairs of abstract heaps.

Given two non-isomorphic abstract heaps \( \tilde{H} \) and \( \tilde{H}' \), \( \sigma^{HS(W)}(\tilde{H}, \tilde{H}') = (\tilde{H}, \tilde{H}') \). The definition of \( \sigma^{HS(W)} \) over two isomorphic abstract heaps uses a partial reduction operator over \( \mathbb{D}^W \)-domains: \( \sigma^W : A_U \times A_W \rightarrow A_U \times A_W \). For any \( \varphi \in A_U \) and \( \varphi' \in A_W \), let \( \sigma^W(\varphi, \varphi') = (\sigma^W_1(\varphi, \varphi'), \sigma^W_2(\varphi, \varphi')) \) (that is, \( \sigma^W_i(\varphi, \varphi') \) is ith component of \( \sigma^W(\varphi, \varphi') \), for any \( 1 \leq i \leq 2 \)). Then, let \( H = (N, S, L, \varphi) \in \mathcal{A}_{HS}(A_U) \) and \( H' = (N, S, L, \varphi') \in \mathcal{A}_{HS}(A_W) \) be two isomorphic abstract heaps (we suppose that the nodes related by the isomorphism have the same name). We define

\[
\sigma^{HS(W)}(\tilde{H}, \tilde{H}') = \left( (N, S, L, \sigma^W_1(\varphi, \varphi')), (N, S, L, \sigma^W_2(\varphi, \varphi')) \right).
\]

The partial reduction operator \( \sigma^W \) is denoted by \( \sigma^U \) if \( W = \mathbb{U} \) and by \( \sigma^M \) if \( W = \mathbb{M} \).

**Proposition 6.7.2.** If \( \sigma^W \) is a partial reduction operator over two \( \mathbb{D}^W \)-domains \( A_U \) and \( A_W \) then, \( \sigma^{HS(W)} \) is also a partial reduction operator between abstract heaps sets over the same \( \mathbb{D}^W \)-domains as \( \sigma^W \).

**Proof.** Similar to the proof of Theorem 6.7.1.
The partial reduction operator $\sigma^U$ Let $\mathbb{P}$ and $\mathbb{P}'$ be two (possibly different) sets of patterns and $\mathcal{A}_Z$ a numerical abstract domain. Also, let $\mathcal{A}_U = \mathcal{A}_U(\mathbb{P})$ be the abstract domain of universally quantified formulas parametrized by $\mathbb{P}$ and $\mathcal{A}_Z$ and $\mathcal{A}_U' = \mathcal{A}_U(\mathbb{P}')$ be the abstract domain of universally quantified formulas parametrized by $\mathbb{P}'$ and $\mathcal{A}_Z$.

We define $\sigma^U : \mathcal{A}_U \times \mathcal{A}_U' \rightarrow \mathcal{A}_U \times \mathcal{A}_U'$ for any two abstract values $\varphi$ in $\mathcal{A}_U$ and $\varphi'$ in $\mathcal{A}_U'$ defined over the same data word variables. If the two formulas constrain different sets of data word variables then we add to each formula new trivial constraints (equivalent to true) over the missing data word variables. Let $V$ be the set of data word variables used in $\varphi$ and $\varphi'$, and let $\varphi = E(V) \land \bigwedge_{P(y) \in \mathbb{P}(V)} \forall y. P(y) \implies U_P$ and $\varphi' = E'(V) \land \bigwedge_{P(y') \in \mathbb{P}(V)} \forall y'. P'(y') \implies U'_P$.

Then,

$$
\sigma^U(\varphi, \varphi') = (\varphi_r, \varphi'_r), \quad \text{where}
\begin{align*}
\varphi_r &:= (E \cap Z E') \land \bigwedge_{P(y) \in \mathbb{P}(V)} \forall y. P(y) \implies U_P \tag{6.7.8} \\
\varphi'_r &:= (E' \cap Z E') \land \bigwedge_{P(y') \in \mathbb{P}(V)} \forall y'. P'(y') \implies U'_P.
\end{align*}
$$

In the following, $\sigma^U_1(\varphi, \varphi')$ (resp. $\sigma^U_2(\varphi, \varphi')$) denotes the first (resp. second) component of $\sigma^U(\varphi, \varphi')$.

**Proposition 6.7.3.** If the meet operator $\cap Z$ of the numerical abstract domain $\mathcal{A}_Z$ is exact then the function $\sigma^U$ defined in (6.7.8) is a partial reduction operator.

**Proof.** The fact that $\varphi_r \subseteq^U \varphi$ and $\varphi'_r \subseteq^U \varphi'$ holds because the meet operator $\cap Z$ defines lower bounds, i.e., $E \cap Z E' \subseteq Z E$ and $E \cap Z E' \subseteq Z E$, and because the universally quantified sub-formulas are not modified.

To obtain the equality between the intersections of the concretizations, $\gamma^U(\varphi) \cap \gamma^U(\varphi') = \gamma^U(\varphi_r) \cap \gamma^U(\varphi'_r)$, we use the fact that $\cap Z$ is exact, that is, $\gamma^Z(E) \cap \gamma^Z(E') = \gamma^Z(E \cap Z E')$.

The partial reduction operator $\sigma^M$ Let $\mathbb{P}$ be a set of patterns and $\mathcal{A}_Z$ a numerical abstract domain and let $\mathcal{A}_U$ be the abstract domain of universally quantified formulas over $\mathbb{P}$ and $\mathcal{A}_Z$. We define $\sigma^M : \mathcal{A}_U \times \mathcal{A}_M \rightarrow \mathcal{A}_U \times \mathcal{A}_M$ for any two abstract values $\varphi$ in $\mathcal{A}_U$ and $\varphi'$ in $\mathcal{A}_M$ defined over the same data word variables (if the two formulas constrain different sets of data word variables, we add trivial constraints over the missing variables as in the definition of $\sigma^U$). Let $V$ be the set of data word variables used in $\varphi$ and $\varphi'$, and let $\varphi = E \land \phi$ with $\phi = \bigwedge_{P(y) \in \mathbb{P}(V)} \forall y. P(y) \implies U_P$.

Then,

$$
\sigma^M(\varphi, \varphi') = (\varphi_r, \varphi'_r), \quad \text{where}
\begin{align*}
\varphi_r &:= E_r \land \phi \\
\varphi'_r &:= \varphi' \land \bigwedge_{n, n' \in V \text{ s.t. } ms_{\mathbf{hd}}(n) = ms_{\mathbf{hd}}(n')} E \subseteq^Z \mathbf{hd}(n) = \mathbf{hd}(n'). \tag{6.7.9}
\end{align*}
$$

In the following, $\sigma^M_1(\varphi, \varphi')$ (resp. $\sigma^M_2(\varphi, \varphi')$) denotes the first (resp. second) component of $\sigma^M(\varphi, \varphi')$.

The $\mathcal{A}_M$ formula $\varphi'_r$ is obtained by adding to $\varphi'$ an equality $ms_{\mathbf{hd}}(n) = ms_{\mathbf{hd}}(n')$ for any equality $\mathbf{hd}(n) = \mathbf{hd}(n')$ implied by the quantifier-free part of $\varphi$. The $\mathcal{A}_U$ formula $\varphi_r$ is obtained from $\varphi$ by adding new constraints to the quantifier-free part of $\varphi$, constraints
on the values of $\text{hd}(n)$ with $n \in \mathcal{V}$, implied by the multiset constraint $\varphi'$. The universally quantified conjuncts of $\varphi_r$ are the same as in $\varphi$. To help the intuition, we start by an example.

**Example 6.7.4.** Let $\varphi ::= \forall y. y \in \text{tl}(n) \implies n[y] > 5$ and $\varphi' ::= \text{ms\_hd}(n_1) \cup \text{ms\_tl}(n_2) = \text{ms\_tl}(n)$. W.l.o.g. we assume that $\varphi$ has an implicit conjunct of the form $\forall y. y \in \text{tl}(n_i) \implies \top^\mathbb{Z}$, with $1 \leq i \leq 2$. The quantifier-free part of $\varphi$ equals $\top^\mathbb{Z}$ and consequently, $\varphi'_r = \varphi'$.

To define $\varphi_r$, notice that, the multiset of values associated with positions in $\text{tl}(n)$ is $\text{ms\_tl}(n)$. Because $\varphi$ has no constraints on the position variable $y$ from the tail of $n$, by an abuse of notation, we can rewrite $\varphi$ as $\forall y, \text{val} \in \text{ms\_tl}(n), \text{val} < 5$, where $\text{val}$ is a variable interpreted as an integer. Remark that $\varphi'$ implies that the value $\text{ms\_hd}(n_1)$, denoting the integer on the first position of the word $n_1$, belongs to the multiset of integers denoted by $\text{ms\_tl}(n)$. Consequently, $\varphi \land \varphi'$ implies that $\text{hd}(n_1) < 5$. This deduction is formalized by the first inference rule in Figure 6.38. So,

$$\varphi_r = (\top^\mathbb{Z} \cap \top^\mathbb{Z} \text{hd}(n_1) < 5) \land \forall y. y \in \text{tl}(n) \implies n[y] > 5$$

and $\sigma^M(\varphi, \varphi') = (\varphi_r, \varphi')$.

In general, a multiset constraint induces multiple choices w.r.t. the multisets to which the singletons belong to. For example if $\varphi' ::= \text{ms\_hd}(n_1) \cup \text{ms\_tl}(n_2) = \text{ms\_tl}(n) \cup \text{ms\_tl}(m)$ then $\varphi'$ implies that either $\text{hd}(n_1) \in \text{ms\_tl}(n)$ or $\text{hd}(n_1) \in \text{ms\_tl}(m)$. In each case, the property on $\text{hd}(n_1)$ added to the formula $\varphi$ might be different. However, there are a finite number of choices. For each of them, we construct a strengthening of $\varphi$ and then we define $\sigma^M(\varphi, \varphi')$ as the join of all of these strengthenings.

For the clarity of the presentation, we extend the syntax of the multiset constraints with the operator $\subseteq$, which corresponds to the inclusion between multisets. To deduce new constraints on $\text{hd}(n)$, for some $n \in \mathcal{V}$, (to be added to $\varphi$) we first search for all the facts “$\text{ms\_hd}(n)$ is equal to $\text{ms\_hd}(n')$” and “$\text{ms\_hd}(n)$ is included into $\text{ms\_tl}(n')$” with $n' \in \mathcal{V}$, which are implied by $\varphi'$. For any conjunct $\varphi'_c$ of $\varphi$, we deduce the set of all conjunctions of the form

$$\psi_c ::= \text{ms\_hd}(n_1) \subseteq \text{bmt}_1 \land \ldots \land \text{ms\_hd}(n_k) \subseteq \text{bmt}_k,$$

where $\text{bmt}_i$ is a basic multiset term, for any $1 \leq i \leq k$, such that (1) $\varphi'_c$ implies the disjunction of all formulas $\psi_c$ as above and (2) $\psi_c$ contains exactly once all the terms of the form $\text{ms\_hd}(n_i)$ from $\varphi'_c$. Note that if, for example, $\text{bmt}_1 = \text{ms\_hd}(n')$ then the conjunction will not contain any other atomic formula over the term $\text{ms\_hd}(n')$.

Formally, let $\varphi' = \bigwedge_c \varphi'_c$ be a multiset constraint like in Section 6.5 where $\varphi'_c$ is a conjunction of the form:

$$\varphi'_c ::= \bigcup_{i \in I_c} \text{ms\_hd}(n_i) \cup \bigcup_{j \in J_c} \text{ms\_tl}(n_j) = \bigcup_{i \in I'_c} \text{ms\_hd}(n_i) \cup \bigcup_{j \in J'_c} \text{ms\_tl}(n_j).$$

Let $\Pi_c$ be the set of all pairs of functions $(\pi, \pi')$ with $\pi : I_c \to I'_c \cup J'_c$ and $\pi' : I'_c \to I_c \cup J_c$ such that for any $i \in I_c$ and $i' \in I'_c$, $\pi(i) = i'$ iff $\pi'(i') = i$. Then, using the semantics of the multiset formulas, it follows that

$$\varphi'_c$$

is equivalent with $\bigvee_{(\pi, \pi') \in \Pi_c} \varphi'_{c, (\pi, \pi')}$, where
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

\[
\begin{align*}
\sigma^M_1(n) & \subseteq \sigma^M_1(n') \quad \text{and} \quad \varphi, n \in U \implies U \\
\sigma^M_1(n) & = \sigma^M_1(n') \\
hd(n) & = \hd(n')
\end{align*}
\]

Figure 6.38: Inference rules for \(A_U\); \(\sigma^M_1(y, (U[n'[y] \leftarrow \hd(n)))\) projects the variable \(y\) from \(U[n'[y] \leftarrow \hd(n))\).

\[
\begin{align*}
\varphi'_{\pi(x, \pi')} &= \bigwedge_{i \in I_c, \bar{i}' \in I_c', \pi(i) = \bar{i}'} \sigma^M_1(n_i) = \sigma^M_1(n_{i'}) \quad \land \quad \bigwedge_{i \in I_c \text{ and } \pi(i) = j \in J_c', \text{or} \ i \in I_c' \text{ and } \pi(i) = j \in J_c} \sigma^M_1(n_i) \subseteq \sigma^M_1(n_j) \quad \land \quad \varphi'_c.
\end{align*}
\]

In terms of concretization we have that

\[
\gamma^M(\varphi'_c) = \bigcup_{(\pi, \pi') \in \Pi_c} \gamma^M(\varphi'_{\pi(x, \pi')}).
\]

Each pair of functions \((\pi, \pi')\) constrains the value of every term \(\sigma^M_1(n_i)\) in the left (resp. right) side of \(\varphi'_c\) such that it belongs to the multiset represented by some basic term in the right (resp. left) side of the equality \(\varphi'_c\). The fact that for any \(i \in I_c\) and \(\bar{i}' \in I_c'\), \(\pi(i) = \bar{i}'\) if \(\pi'({\bar{i}'}) = i\) means that once \(\sigma^M_1(n_i)\) equals \(\sigma^M_1(n_{i'}')\), no other constraints can be imposed on \(\sigma^M_1(n_i)\) and \(\sigma^M_1(n_{i'})\). Then,

1. for every \((\pi, \pi') \in \Pi_c\), we define \(\varphi'_{\pi(x, \pi')}\) a formula in \(A_U\) stronger than \(\varphi\). The formula \(\varphi'_{\pi(x, \pi')}\) is obtained from \(\varphi\) by adding constraints to the quantifier-free part. These constraints are deduced by applying the inference rules in Figure 6.38 for every atomic formula \(\sigma^M_1(n_i) \subseteq \sigma^M_1(n_j)\), respectively \(\sigma^M_1(n_i) = \sigma^M_1(n_{i'})\), added to \(\varphi'_c\) in order to define \(\varphi'_{\pi(x, \pi')}\);

2. we define \(\varphi_c\) to be the join of all formulas \(\varphi'_{\pi(x, \pi')}\) with \((\pi, \pi') \in \Pi_c\) obtained in the previous step;

3. we define \(\sigma^M_1(\varphi, \varphi')\) (i.e. \(\varphi_\gamma\)) as the intersection of all the formulas \(\varphi_c\) obtained in the previous step (each formula \(\varphi_c\) corresponds to one conjunct \(\varphi'_c\) of \(\varphi\)).

Before formalizing these two steps let us take a closer look to the inference rules from Figure 6.38. The correctness of the second rule is immediate from the semantics of the terms. The first rule states that the property obtained from the right part of the universal formula by substituting \(n'[y]\) with \(\hd(n)\) and by projecting the universal variable \(y\) is implied by the premises (this is correct because \(\sigma^M_1(n) = \{\hd(n)\}\) and \(\sigma^M_1(n) \subseteq \sigma^M_1(n')\) means that \(\text{head}(n)\) equals a data value in the tail of \(n')\). The position variable \(y\) is eliminated from \(U\) because \(\sigma^M_1(n)\) equals a data value \(d\) in the tail of \(n'\), without being related to the position of \(d\) in the word \(n'\).

Therefore, for every \((\pi, \pi') \in \Pi_c\), we define

\[
E_{\pi(x, \pi')} = \bigcap_{i \in I_c, \bar{i}' \in I_c', \pi(i) = \bar{i}'} \sigma^M_1(n_i) = \sigma^M_1(n_{i'}) \land \bigcap_{i \in I_c \text{ and } \pi(i) = j \in J_c', \text{or} \ i \in I_c' \text{ and } \pi(i) = j \in J_c} \sigma^M_1(n_i) \subseteq \sigma^M_1(n_j) \implies U
\]

\[
\bigcap_{i \in I_c \text{ and } \pi(i) = j \in J_c', \text{or} \ i \in I_c' \text{ and } \pi(i) = j \in J_c} \sigma^M_1(y, (U[n_j[y] \leftarrow \hd(n_i))))
\]
and \( \varphi_{c,(\pi,\pi')} = E_{c,(\pi,\pi')} \land E \land \phi \). Then, \( \varphi_c = \bigcup_{(\pi,\pi') \in \Pi_c} \varphi_{c,(\pi,\pi')} \), which by the definition of \( \bigcup \) implies that
\[
\varphi_c = E_c \land E \land \phi, \quad \text{with} \quad E_c = \bigcup_{(\pi,\pi') \in \Pi_c} Z_{E_{c,(\pi,\pi')}}.
\]

Finally, \( \varphi_r = \varphi \cap \bigcup \varphi_c \), which by the definition of \( \bigcap \) implies that
\[
\varphi_r = E_r \land \ldots, \quad \text{with} \quad E_r = \bigcap_c Z_{E_c}.
\]

### 6.7.2 Remark.
If \( y \in t1(\omega) \) is not in the set of patterns \( P \) that parametrizes \( A_U \) or if \( \varphi \) contains only formulas of the form \( \forall y. y \in t1(n) \implies \top \) with \( n \in V \) then \( \sigma^M(\varphi,\varphi') = (\varphi,\varphi') \). However, in order to obtain more precise results, we could first saturate the universally quantified formulas (i.e., apply the procedure \( \text{sat}^\# \)) and then apply the partial reduction operator between \( \text{sat}^\#(\varphi) \) and \( \varphi' \).

### Proposition 6.7.4.
If the meet operator \( \bigcap^Z \) of the numerical abstract domain \( A_Z \) is exact then the function \( \sigma^M \) defined in [6.7.9] is a partial reduction operator.

**Proof.** Let \( \sigma^M(\varphi,\varphi') = (\varphi_r,\varphi'_r) \) such that \( \varphi_r = \varphi \cap \bigcup \varphi_{aux} \) and \( \varphi'_r = \varphi' \cap \bigcup \varphi'_{aux} \), where
\[
\varphi_{aux} = \bigcap_c \varphi_c \quad \text{and} \quad \varphi'_{aux} = \bigcap_{n,n' \in V \text{ s.t.} \quad E \subseteq^Z \text{hd}(n) = \text{hd}(n')} \bigcap_{\sigma} \ms_{\text{hd}}(n).
\]
The fact that \( \varphi \subseteq \bigcup \varphi_r \) and \( \varphi' \subseteq \bigcap \varphi'_r \) follows from the fact that meet is the greatest lower bound operator. Then, to prove the equality between the intersections of the concretizations, we prove that:
\[
\begin{align*}
(1) \quad \gamma^U(\varphi) \cap \gamma^M(\varphi') & \subseteq \gamma^U(\varphi_{aux}) \quad \text{and} \\
(2) \quad \gamma^U(\varphi) \cap \gamma^M(\varphi') & \subseteq \gamma^M(\varphi'_{aux})
\end{align*}
\]
(6.7.12)

Let us detail the proof of the first inclusion since it is more technically involved (the second one can be proved in a similar manner). The fact that \( \bigcap^Z \) is exact implies that \( \bigcup^U \) is also exact. Thus, we first prove that:
\[
(\forall \varphi'_c \in \varphi_r, \quad \gamma^U(\varphi) \cap \gamma^M(\varphi') \subseteq \gamma^U(\varphi_c)).
\]
(6.7.13)

Notice that \( \gamma^M(\varphi') = \bigcap_c \gamma^M(\varphi'_c) \) and, for any conjunct \( \varphi'_c \) of \( \varphi' \),
\[
\bigcap_c \gamma^M(\varphi'_c) \cap \gamma^U(\varphi) \subseteq \gamma^M(\varphi'_c) \cap \gamma^U(\varphi).
\]

Thus, in order to validate (6.7.13), it is sufficient to prove that:
\[
\gamma^M(\varphi'_c) \cap \gamma^U(\varphi) \subseteq \gamma^U(\varphi_c).
\]
(6.7.14)

By (6.7.11), the inclusion from (6.7.14) is equivalent to
\[
\bigcup_{(\pi,\pi') \in \Pi_c} \gamma^M(\varphi'_{c,(\pi,\pi')}) \cap \gamma^U(\varphi) \subseteq \gamma^U(\varphi_c)
\]
(6.7.15)
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

void main() {
    list *zm, *zn;
    zm = m0;
    zn = n0;
    while((zm!=NULL) && (zn!=NULL)) {
        zm = zm->next;
        zn = zn->next;
    }
    while(zm!=NULL) zm = zm->next;
    while(zn!=NULL) zn = zn->next;
}

Figure 6.39: The program $P_{\varphi_1,\varphi_2}$ and its initial configuration used in infer($\varphi_1,\varphi_2$) when $M = \{m,n\}$

The correctness of the inference rules implies that:

$$\text{for any } (\pi,\pi') \in \Pi_c, \quad \gamma^M(\varphi_{c,(\pi,\pi')}) \cap \gamma^U(\varphi) \subseteq \gamma^U(\varphi_{c,(\pi,\pi')}).$$

(6.7.16)

The fact that $\bigcup^U$ is a least upper bound operator implies that:

$$\bigcup_{(\pi,\pi') \in \Pi_c} \gamma^U(\varphi_{c,(\pi,\pi')}) \subseteq \gamma^U(\bigcup_{(\pi,\pi') \in \Pi_c} \varphi_{c,(\pi,\pi')}) = \gamma^U(\varphi_c).$$

(6.7.17)

The inclusions in (6.7.16) and (6.7.17) finalize the proof of (6.7.14).

6.7.3.2.2 The program analyzed to compute infer Let $\varphi_1 \in A_{W_1}$ and $\varphi_2 \in A_{W_2}$. In order to define infer($\varphi_1,\varphi_2$) we analyze a program denoted $P_{\varphi_1,\varphi_2}$ in the partially reduced product defined previously.

Let $M$ denote the set of data word variables which appear in both $\varphi_1$ and $\varphi_2$. The program $P_{\varphi_1,\varphi_2}$ manipulates a set of disjoint lists, one for each variable in $M$, and consists in while loops that traverse these lists. The number of program variables and the number of loops depends only on the number of variables in $M$. In practice, we can heuristically choose to consider only some of the data word variables that appear in both $\varphi_1$ and $\varphi_2$; the result of infer($\varphi_1,\varphi_2$) would still be sound.

The initial configuration of the program is a pair of abstract heaps $(\tilde{H}_1, \tilde{H}_2)$ which contain the same graph $(N, S, L)$ such that

1. $N$ contains exactly one node for each data word variable that appears in both $\varphi_1$ and $\varphi_2$ plus the distinguished nodes $\sharp$ and $\sharp'$; the successor of every node $n \in N$ with $n \neq \sharp$ is $\sharp$ (i.e. $S(n) = \sharp$),

2. every node $n \in N$ is labeled by two pointer variables, denoted n0 and zn,

3. $\tilde{H}_1 = (N, S, L, \varphi_1)$ and $\tilde{H}_2 = (N, S, L, \varphi_2)$.

The first loop of the program traverses simultaneously all list segments, using the variables zn. Since the list segments may not be of equal lengths, for every possibility that one of them ended (some zn equals null), the traversal of the other lists continues following the
same principle: all the lists such that the corresponding variable $zn$ does not equal null, are traversed simultaneously. Therefore, the first loop is followed by $i$ loops that traverse $i - 1$ lists, where $i = |M|$ is the number of word variables belonging to both $\varphi_1$ and $\varphi_2$. After these $i$ loops, the program has $i \times (i - 1)$ loops corresponding to all the possibilities that two of the variables $zn$ reached null. The total number of loops is $\sum_{k=0}^{\lceil \frac{i}{2} \rceil}$.

**Example 6.7.5.** Let $M = \{m, n\}$. The program used in infer$(\varphi_1, \varphi_2)$ is given in Figure 6.39. The initial configuration is given by the two heaps $H_1$ and $H_2$ in Figure 6.39. The first loop traverses simultaneously the two list segments. At each iteration, the pointer variables $zn$ and $zn$ are advanced to the next element. The variables $n0$ and $m0$ point to the beginning of the lists denoting $n$ and $m$. Their values are never changed by the program. Since the lists may not have the same length, we add another two while loops to continue the traversal of the unfinished list starting from where the previous loop stopped.

**6.7.3.2.3 Abstract values computed by infer** Let $(\tilde{HS}_1, \tilde{HS}_2) \in \mathcal{A}_{HS}(\mathcal{A}_{W_1}) \times \mathcal{A}_{HS}(\mathcal{A}_{W_2})$ be the postcondition of the program $P_{\varphi_1, \varphi_2}$ (i.e., the pair of abstract heap sets associated to the last control point). Notice that any underlying graph $\tilde{H}_1' \in \tilde{HS}_1$ and $\tilde{H}_2' \in \tilde{HS}_2$ has the following properties:

- the variables $zn$ label the node $\sharp$;
- the variables $n0$ label the nodes $n$ that have no predecessors;
- the graphs have no crucial nodes other than the ones labeled by program variables.

The absence of sharing nodes is a simple consequence of the fact that the program traverses the lists without performing any assignments to the pointer fields (also, it does not perform any assignments to the data fields).

If the program is analyzed with 0-SLL abstract heap sets (i.e., the graphs do not contain simple nodes) then any $\tilde{H}_1'' \in \tilde{HS}_1$, resp. $\tilde{H}_2'' \in \tilde{HS}_2$, has an underlying graph isomorphic to the underlying graph of $\tilde{H}_1$, resp. $\tilde{H}_2$, from the initial configuration. If the analysis is performed over domains of abstract $k$-SLL heap sets with $k \geq 1$ then, the underlying graphs of $\tilde{H}_1$ and $\tilde{H}_2$ may contain simple nodes (on the paths starting in nodes labeled by variables $n0$ and ending in $\sharp$). For example, if the program given in Figure 6.39 is analyzed using abstract 1-SLL heap sets then the graphs that may appear in its postcondition are pictured in Figure 6.40. In order to define infer, we eliminate all simple nodes from the postcondition by applying fold#. In this way, any $\tilde{H}_1'' \in$ fold#$(\tilde{HS}_1)$, resp. $\tilde{H}_2'' \in$ fold#$(\tilde{HS}_2)$, has an underlying graph isomorphic to the underlying graph of $\tilde{H}_1$, resp. $\tilde{H}_2$, from the initial configuration.

**Definition 6.7.2.** Given two abstract values $\varphi_1 \in \mathcal{A}_{W_1}$ and $\varphi_2 \in \mathcal{A}_{W_2}$, let $P_{\varphi_1, \varphi_2}$ be the program defined as above and $(\tilde{H}_1, \tilde{H}_2)$ its initial configuration. Also, let $(\tilde{HS}_1, \tilde{HS}_2) \in \mathcal{A}_{HS}(\mathcal{A}_{W_1}) \times \mathcal{A}_{HS}(\mathcal{A}_{W_2})$ be the abstract element associated with the last control point by the analysis of $P_{\varphi_1, \varphi_2}$ in the partially reduced product $\mathcal{A}_{HS}(\mathcal{A}_{W_1}) \times \mathcal{A}_{HS}(\mathcal{A}_{W_2})$ induced by the partial reduction operator $\sigma_{HS(W)}$. Then,

$$\text{infer}(\varphi_1, \varphi_2) = (\varphi_1', \varphi_2'),$$

where

$$\text{fold}^#(\tilde{HS}_1) = \{(N_1, S_1, L_1, \varphi_1')\} \quad \text{and} \quad \text{fold}^#(\tilde{HS}_2) = \{(N_2, S_2, L_2, \varphi_2')\}.$$
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

Figure 6.40: Graphs from the post-condition of $P_{\varphi_1,\varphi_2}$ when analyzed with abstract 1-SLL heap sets.

The following proposition is a direct consequence of the definitions of $P_{\varphi_1,\varphi_2}$ and of the partial reduced product between $A_{\text{SSS}}(A_{W_1})$ and $A_{\text{HS}}(A_{W_2})$.

**Proposition 6.7.5.** Given two abstract values $\varphi_1 \in A_{W_1}$ and $\varphi_2 \in A_{W_2}$, let $\text{infer}(\varphi_1, \varphi_2) = (\varphi'_1, \varphi'_2)$. Then,

$$
\gamma^{W_1}(\varphi_1) \cap \gamma^{W_2}(\varphi_2) \subseteq \gamma^{W_1}(\varphi'_1) \quad \text{and} \quad \gamma^{W_1}(\varphi_1) \cap \gamma^{W_2}(\varphi_2) \subseteq \gamma^{W_2}(\varphi'_2).
$$

### 6.7.3.3 A new saturation procedure for $A_U$ using $\text{strengthen}_U$

In Section 6.4.3 we have introduced a saturation procedure, called $\text{sat}^\#$, which strengthens any formula $\varphi \in A_U$, such that $\text{sat}^\#(\varphi) \sqsubseteq^U \varphi$ and $\gamma(\varphi) = \gamma(\text{sat}^\#(\varphi))$. In this Section, we define a new saturation procedure $\text{sat}^\#_s$ based on the partial reduction operator $\text{strengthen}_U$.

**Definition 6.7.3 ($\text{sat}^\#_s$).** Let $\varphi$ be an abstract element in $A_U$ parametrized by a set of patterns $\mathbb{P}$. Then,

$$
\text{sat}^\#_s : A_U(\mathbb{P}) \to A_U(\mathbb{P}), \quad \text{by}
$$

$$
\text{sat}^\#_s(\varphi) = \varphi' \quad \text{with} \quad \text{strengthen}_U(\varphi, \top^U) = (\varphi, \varphi'),
$$

where $A_U(\mathbb{P})$ is the lattice $A_U$ parametrized by the set of patterns $\mathbb{P}$, $\top^U$ is its the top element, and $\text{strengthen}_U : A_U(\mathbb{P}) \times A_U(\mathbb{P}) \to A_U(\mathbb{P}) \times A_U(\mathbb{P})$.

The fact that $\text{sat}^\#_s(\varphi) \sqsubseteq^U \varphi$ and that $\gamma(\varphi) = \gamma(\text{sat}^\#_s(\varphi))$ follow from the definition of $\text{strengthen}_U$.

The two saturation procedures are incomparable. The definition of $\text{sat}^\#_s$ is based on a fixed point computation that terminates due to a widening operator. Therefore there are no theoretical guarantees that the resulting formula, i.e. $\text{sat}^\#_s(\varphi)$, is stronger than $\varphi$ w.r.t. the order relation $\sqsubseteq^U$.

Nevertheless, there are formulas in $A_U$ which $\text{sat}^\#$ cannot strengthen w.r.t. the order relation $\sqsubseteq^U$. For example, let us consider the abstract domain $A_U$ parametrized by the patterns $\{y \in \text{tl}(\omega), y \in \text{tl}(\omega) \land y = 1, \gamma, y' \in \text{tl}(\omega) \land y < 1 \land y'\}$. Then, the formula

$$
\varphi^a := \text{hd}(n) = 0 \land \forall y, y \in \text{tl}(n) \Rightarrow \top^Z
$$

$$
\land \forall y, y', y, y' \in \text{tl}(n) \land y < 1 \land y' \Rightarrow n[y'] = n[y] + 1
$$

$$
\land \forall y, y \in \text{tl}(n) \land y = 1 \Rightarrow n[y] = 1.
$$

states that the word denoted by $n$ denotes the sequence of the first natural numbers, that is $\gamma^U(\varphi^a) = \{0, 1, 2, 3, \ldots\}$. 

250
6.8. CELIA: A TOOL FOR INTER-PROCEDURAL ANALYSIS

If we apply the saturation procedure given in Section 6.4.3 we obtain that \( \text{sat}^\#(\varphi^a) = \varphi^a \). But, using \( \text{sat}^\#_s \) we obtain the following formula:

\[
\text{sat}^\#_s(\varphi^a) := \text{hd}(n) = 0 \land \forall y. y \in \text{tl}(n) \implies n[y] = y \land \\
\land \forall y, y'. y, y' \in \text{tl}(n) \land y <_1 y' \implies n[y'] = n[y] + 1 \\
\land \forall y. y \in \text{tl}(n) \land y = 1' \implies n[y] = 1.
\]

Intuitively, \( \text{sat}^\#_s \) does not strengthen \( \varphi^a \) because there is no explicit relation between position variables and data.

There are also formulas that the procedure \( \text{sat}^\#_s \) does not strengthen but a stronger w.r.t. \( \sqsubseteq_U \) formula is obtained using \( \text{sat}^\# \). Such an example is the formula:

\[
\varphi^b := \forall y. y \in \text{tl}(n) \implies n[y] = \text{len}[m] \land \\
\forall y', y' \in \text{tl}(m) \implies \top \land \\
\land \forall y, y'. y \in \text{tl}(n) \land y \in \text{tl}(m) \land y = y' \implies m[y'] = n[y].
\]

\( \varphi^b \) states the two words denoted by \( n \) and \( m \) are equal and that all the values of the tail on \( n \) are equal with the length of the word denoted \( n \). The procedure \( \text{sat}^\# \) returns

\[
\varphi^b := \forall y. y \in \text{tl}(n) \implies d(y) = \text{len}[m] \land \\
\forall y', y' \in \text{tl}(m) \implies m[y] = \text{len}[m] \land \\
\land \forall y, y'. y \in \text{tl}(n) \land y' \in \text{tl}(m) \land y = y' \implies m[y'] = n[y].
\]

while \( \text{sat}^\#_s(\varphi^b) = \varphi^b \). Intuitively, \( \text{sat}^\#_s \) does not strengthen \( \varphi^b \) because the length of the list segment associated with \( m \) changes during the traversal of the two lists and the widening loses the connection between data and length.

6.8 CELIA: A tool for inter-procedural analysis

CELIA is a plug-in of the Frama-C platform [CEA] which implements a static analysis for C programs manipulating (singly linked) lists. More precisely, CELIA computes for each line \( l \) of the program, reachable from the main function an assertion describing the program configurations reachable at \( l \) and verifies the assertions given in the C input file, if any. For this, CELIA performs a symbolic reachability analysis based on abstract interpretation [Cousot 1977a] which uses the abstract domains introduced in this chapter.

6.8.1 Architecture

CELIA is based on several tools released on LGPL licence: the Frama-C platform is used for parsing and typing the input C programs, the Fixpoint library [Jeannet] is used as a symbolic fixed point engine on inter-procedural control flow graphs, the CINV tool implements the abstract domains on lists introduced in this chapter, and the Apron library [Jeannet 2009] implements the numerical abstract domains. Its architecture diagram is given in Figure 6.41.

CELIA receives as inputs a list of C files containing the procedures to be analyzed and a configuration file, where the parameters of the analysis and the entry point are specified (the default entry point is the procedure main).

First, CELIA uses Frama-C to obtain a normalized version of the original C input files. Roughly, during the normalization phase, Frama-C performs a number of local code transformations. These transformations aim at making further work easier for the analyzers.
For example, it performs syntactic folding of constant expressions, or it replaces a loop statement by a \texttt{while(true)} statement followed by an \texttt{if} statement. Also, to control points of the program are associated unique identifiers, recalled in comments.

Celia builds a system of equations, given as an inter-procedural control flow graph (ICFG) in the Fixpoint library format. This graph is obtained from a normalized version of the original C input files. The module Clim that defines the system of equations is written in Ocaml. The system of equations computed by Clim, corresponding to the input program, is given as input to CINV, which uses Fixpoint to solve the system of equations. Fixpoint is an OCaml library that implements a generic fixpoint engine, based on the iteration strategies defined in [Bourdoncle 1993]. The interface is parameterized by the abstract domain on which fixpoint computations are performed. Celia also translates (if possible) the assertions written in the program into values of the abstract domain which is the parameter of Fixpoint.

CINV implements a generic abstract domain $\mathcal{A}_{HS}$ for reasoning about dynamic lists with unbounded data which includes an abstraction on the shape of the heap and which is parametrized by some abstract domain on finite sequences of data (a data words abstract domain). The $\mathcal{DW}$-domains implemented by CINV are

- $\mathcal{A}_M$, a version of the abstract domain introduced in Section 6.5 which captures
properties on the data, the length and the multiset of the elements in the lists;

- \( \mathcal{A}_S \), a version of the abstract domain introduced in Section 6.6 which captures properties on the data, the length and the sum of the elements in the lists;

- \( \mathcal{A}_U \), introduced in Section 6.4, an abstract domain with universally quantified formulas to reason about the content of the lists.

The \( \mathbb{DW} \)-domains from CINV use the numerical abstract domains implemented by APRON. The abstract domains \( \mathcal{A}_M \) and \( \mathcal{A}_S \) are implemented using the polyhedra abstract domain. We recall that the elements of \( \mathcal{A}_U \) are formulas of the form 
\[
E \land \bigwedge_{P(y) \in \mathcal{P}(y)} \forall y. P(y \Rightarrow U_P),
\]
developed in Section 6.4.1.1. In our implementation, \( E \) and \( U_g \) are elements from a numerical abstract domain implemented by APRON. The current implementation supports the following set of patterns, together with their closure:

- \( P_1 = \forall y \in tl(\omega) \),
- \( P_2 = \forall y, y'. y, y' \in tl(\omega). y \leq y' \), and
- \( P_\infty = \forall y, y'. y \in tl(\omega) \land y' \in tl(\omega') \land y = y' \).

The APRON library provides numerical abstract domains implementations, defined under a uniform API. The abstract domains defined in APRON are the interval abstract domain (called box), the octagon abstract domain, called octagons, and the convex polyhedra domain, called \texttt{NewPolka}, described in Section 3.4. Moreover, independent libraries, e.g. \[ PPL \] (an implementation for convex polyhedra that support linear congruences), can be connected using a wrapper.

**Input:** The C files analyzed by Celia respect the program syntax given in Section 4.2.1 and Section 4.3.1 (for loops and do .. while loops are also allowed). The C files given as input to Celia can be annotated with specifications in the ACSL logic of Frama-C. ACSL is a standard specification logic that supports user defined predicates. We use these predicates to define \( SL3 \) formulas.

The configuration file “cinv.properties” contains the parameters of the analysis: it is a list of lines of the form key=value, where key specifies the option and value specifies the value of the option. The following tuples of keys and values are known to Celia:

- \texttt{dwdomain} denotes the abstract domain to be used for sequences of integers; possible values are \texttt{lsum} (default) for the sums abstract domain \( \mathcal{A}_S \), \texttt{mset} for the multiset abstract domain \( \mathcal{A}_M \), and \texttt{ucons} for the abstract domain of universally quantified variables \( \mathcal{A}_U \). In order to use products of abstract domains the parameter \texttt{dwdomain} may be defined with multiple values.;

- \texttt{maxanon} is the maximum number of anonymous nodes per segment (default 0); this number can be changed for each function (see option \texttt{funspec});

- \texttt{maxasegm} is the maximum number of segments with anonymous nodes (default 1); this number can be changed for each function (see option \texttt{funspec});

- \texttt{funspec} defines the name of configuration file in which are given specific parameters of analysis of each function;
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

- **main** is the name of the function where the analysis starts (default: *main*).
- in this file, one can also define the parameters of Fixpoint, **depth**, **wdelay**, and **wdesc** which are used to define the parameters of the iteration strategy.

The format of the function specification file (given by *funspec*) is a sequence of lines, each line having the form: `function_name: n s p`, where: `function_name` id the name of the function for which

- `n` defines the value of the **maxanon** parameter (a positive integer),
- `s` defines the value of the **maxsegm** parameter (a strictly positive integer),
- `p` defines the patterns to be used in the **ucons** domain (if **ucons** is considered among the selected **DW**-domains); it is an integer value obtained from the binary or of the following values: 1 for the pattern $P_1$ (default value), 2 for the pattern $P_2$, 4 for the pattern $P_3$.

Also, in a different file “cinv.txt” the numerical abstract domain used for the representation of $E$ and $U_P$ is defined, if the **DW**-domains is **ucons**. The file contains one of the strings: *box*, *oct*, *polyk*, or *polyp*, corresponding to the abstract domain of boxes, octagons, polka and PPL, respectively.

Output: Celia outputs the following files:

- *pan-nm.c* contains the C code normalized by Frama-C where each control point has a unique identifier (given by a comment containing $sid$).
- *pan.eq* contains the inter-procedural control flow graph (ICFG) considered by the analysis;
- *pan.abs* contains the result of the analysis, i.e., for each control point $l$ it gives a number, denoted here *XXX* such that the file *f_XXX.shp* contains the invariant that describes the set of program configuration reachable in $l$;
- the files *f_XXX.shp* can be visualized using *shp2dot.sh*.

6.8.2 Experimental results

Benchmark: We have applied Celia to a benchmark of C programs which is available on the web site of Celia. The benchmark contains all the basic functions that are used in usual libraries on singly-linked lists, for example the GTK *glist* library which is part of the Linux distribution. Table 6.1 gives a sample of functions in this benchmark, split in six classes. The class **sll** includes C functions performing elementary operations on lists: adding/deleting the first/last element, initializing a list of some length. The classes **map** and **map2** include C functions performing a traversal of one resp. two lists, without modifying their structure, but modifying their data. The classes **fold** and **fold2** include C functions computing from one resp. two input lists some output parameters of type list or integer. Finally, the **sort** class includes sorting algorithms on lists. For the procedures
in classes map* and fold* we consider both iterative and recursive versions. The third column of Table 6.1 specifies the versions considered (iterative/recursive) and the number of nested loops or recursive calls.

The benchmark also contains programs which do several calls of the above introduced functions. For example, we handle some programs manipulating chaining hash tables. For that, we use abstraction techniques (slicing, unfolding fixed-size arrays) available through the Frama-C platform.

Computing procedure summaries: Table 6.1 describes some of our experimental results on the synthesis of procedure summaries.

Columns 4–5 provides the running time of the analysis depending on the used abstract domain, including the calls to the APRON library. (All experiments have been done on an Intel i3-370M with 2.4GHz and 2GB of RAM.) Column 5 gives also the set of patterns used for the analysis with $A_{HS}(A_U)$. The pattern $P_\_\_\_\_$ is used by default for each analysis since it is needed to capture the relation of equality between actual and formal function parameters. The choice of $P_1$ and $P_2$ is made according to a heuristics that is based on syntactical criteria such as the number of nested loops or the number of recursive calls in the body of the program. (These numbers are reported in column 3 of Table 6.1.)

The pattern $P_1$ is used for programs with at least one loop (resp. recursive call) and one iteration variable over lists. The pattern $P_2$ is used for nested loops, more than one recursive call, or two iteration variables. Column 6 of Table 6.1 shows samples of procedure summaries that Celia can synthesize. (We use the & sign to denote, like in C, the output parameters.)

All examples in our benchmark corresponding to common functions for list manipulation (classes sllfold2 in Table 6.1 except the function merge) are analyzed in less than 1 second. During the analysis of these programs, the manipulated relations are represented using at most 6 abstract heaps, each of them having at most 16 nodes. For the rest of the examples, these relations have at most 18 abstract heaps. The sorting algorithms are time consuming due to (1) complexity of the inter-procedural control flow graph, and (2) the frequent use of the strengthen operation in examples such as quicksort.

In all examples we have considered, the summary $\rho^#$ is represented by at most 6 abstract heap graphs, each of them having at most 16 nodes. $\rho^#_M$ is the summary obtained with the multiset domain and $\rho^#_U$ is the summary obtained using $A_U$.

Besides dealing with recursion, compositional inter-procedural allows to have scalable analysis. For instance, consider a program that calls the init(v) function on 10 different lists. Our analysis computes once the summary of this function and reuse it, while the analysis after inlining computes successively the effect of all the calls. Thus, the inter-procedural analysis is ten times faster for this example than the intra-procedural analysis.

Combination of abstract domains: The use of the partial reduction operator strengthen$^{HS}$ is needed in many examples of programs with procedure calls. For instance, as we have seen in Section 6.7.1, the analysis of the recursive sorting algorithm quicksort requires combining universal formulas with multiset constraints. Without this combination, the quicksort procedure must be transformed to have two parameters (the first and the last element of the list) like in Rinetzky 2005b. Therefore, the pivot is given as a parameter which helps to recover at the return from the recursive calls the property that all elements are less/greater than the pivot.
### Chapter 6. Analysis of Programs Manipulating Singly-Linked Lists

<table>
<thead>
<tr>
<th>class</th>
<th>fun</th>
<th>nesting (loop, rec)</th>
<th>$\mathcal{A}_{\text{S}}$ $t$ (s)</th>
<th>$\mathcal{A}_{\text{U}}$ $t$ (s)</th>
<th>Examples of summaries synthesized</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll</td>
<td>alloc</td>
<td>(0, −)</td>
<td>0.013</td>
<td>$P_m, P_1$</td>
<td>0.021</td>
</tr>
<tr>
<td></td>
<td>addfst</td>
<td>−</td>
<td>0.003</td>
<td>$P_m$</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>addllst</td>
<td>(0, 1)</td>
<td>0.031</td>
<td>$P_m$</td>
<td>0.033</td>
</tr>
<tr>
<td></td>
<td>delfst</td>
<td>−</td>
<td>0.001</td>
<td>$P_m$</td>
<td>0.001</td>
</tr>
<tr>
<td></td>
<td>dellst</td>
<td>(0, 1)</td>
<td>0.034</td>
<td>$P_m$</td>
<td>0.042</td>
</tr>
<tr>
<td>map</td>
<td>init(v)</td>
<td>(0, 1)</td>
<td>0.024</td>
<td>$P_m, P_1$</td>
<td>0.034</td>
</tr>
<tr>
<td></td>
<td>initSeq</td>
<td>(0, 1)</td>
<td>0.024</td>
<td>$P_m, P_1$</td>
<td>0.034</td>
</tr>
<tr>
<td></td>
<td>add(v)</td>
<td>(0, 1)</td>
<td>0.021</td>
<td>$P_m$</td>
<td>0.032</td>
</tr>
<tr>
<td>map2</td>
<td>add(v)</td>
<td>(0, 1)</td>
<td>0.089</td>
<td>$P_m$</td>
<td>0.517</td>
</tr>
<tr>
<td></td>
<td>copy</td>
<td>(0, 1)</td>
<td>0.063</td>
<td>$P_m$</td>
<td>0.078</td>
</tr>
<tr>
<td>fold</td>
<td>delPred</td>
<td>(0, 1)</td>
<td>0.062</td>
<td>$P_m, P_1$</td>
<td>0.145</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>(0, 1)</td>
<td>0.031</td>
<td>$P_m, P_1$</td>
<td>0.048</td>
</tr>
<tr>
<td></td>
<td>clone</td>
<td>(0, 1)</td>
<td>0.071</td>
<td>$P_m$</td>
<td>0.315</td>
</tr>
<tr>
<td></td>
<td>split</td>
<td>(0, 1)</td>
<td>0.245</td>
<td>$P_m, P_1$</td>
<td>0.871</td>
</tr>
<tr>
<td>fold2</td>
<td>equal</td>
<td>(0, 1)</td>
<td>0.127</td>
<td>$P_m$</td>
<td>0.261</td>
</tr>
<tr>
<td></td>
<td>concat</td>
<td>(0, 1)</td>
<td>0.217</td>
<td>$P_m, P_1, P_2$</td>
<td>0.806</td>
</tr>
<tr>
<td></td>
<td>merge</td>
<td>(0, 1)</td>
<td>1.014</td>
<td>$P_m, P_1, P_2$</td>
<td>2.306</td>
</tr>
<tr>
<td>sort</td>
<td>bubble</td>
<td>(1, −)</td>
<td>0.387</td>
<td>$P_m, P_1, P_2$</td>
<td>2.190</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>(1, −)</td>
<td>0.557</td>
<td>$P_m, P_1, P_2$</td>
<td>3.292</td>
</tr>
<tr>
<td></td>
<td>quick</td>
<td>(−, 2)</td>
<td>1.541</td>
<td>$P_m, P_1, P_2$</td>
<td>121.1</td>
</tr>
<tr>
<td></td>
<td>merge</td>
<td>(−, 2)</td>
<td>1.547</td>
<td>$P_m, P_1, P_2$</td>
<td>95.94</td>
</tr>
</tbody>
</table>

Table 6.1: Experimental results for functions in our benchmark.
Non-recursive programs may also need strengthening operations for their analysis due to the fact that different sets of patterns may be used for different procedure calls. To experiment that, we have considered programs performing multiple calls to procedures given in Table 6.1, taking \{P_e, P_1, P_2\} as set of patterns for the analysis of the main procedure. For example, we have considered a procedure that calls \texttt{bubblesort} on a list \(x\), and then copies it in a variable \(y\) using the procedure \texttt{clone} (the procedure \texttt{bubblesort} is analyzed using \{P_e, P_1, P_2\} and the procedure \texttt{clone} using \{P_e\}). For the call to \texttt{clone}, we obtain that the two lists \(x\) and \(y\) are equal, but the sortedness property of \(x\) is not transferred to \(y\). However, this property can be recovered at the return of \texttt{clone} (using the \texttt{strengthen} operation) from the fact that \(y\) is equal to \(x\) and that \(x\) is sorted.

**Equivalence checking:** We have experimented the analysis described in this chapter for checking equivalence between sorting algorithms. In order to check the equivalence of two procedures, we need to maintain a copy of their inputs. The two procedures are equivalent if whenever they are executed on two copies of the same input, they return equal outputs.

To explain this, consider the program given in Figure 4.22 from Section 4.3.2. The program checks the equivalence of two sorting procedures, \texttt{quicksort} and \texttt{mergesort}.

The first part of the program allocates a list pointed to by \(h_1\) and initializes it with some random values. Then, \(h_2\) points to a copy of the list pointed to by \(h_1\). Each of these lists is going to be given as input to one of the sorting procedures. Before calling the two procedure, the program makes a copy of the inputs. These copies are pointed to by \(hi_1\) and \(hi_2\). Then, the procedure \texttt{quicksort} is called and \(h_1\) points to the list returned by \texttt{quicksort}. Similarly, the procedure \texttt{mergesort} is called and \(h_2\) points to the list returned by \texttt{mergesort}.

We reduce the problem of checking the equivalence of the two procedures to checking the validity of the \texttt{assert} statement at line 29 in all program runs, which states that \(h_1\) and \(h_2\) point to two different but equal lists.

The analysis with \(A_{HS}(A_U)\), parametrized by the patterns \{P_e, P_1, P_2\}, generates the following assertions:

- at line 17:
  \[
  \varphi_{17} := \text{ls}(h_1, \text{null}) \ast \text{ls}(h_2, \text{null}) \land eq(h_1, h_2), \tag{6.8.1}
  \]

- at line 22
  \[
  \varphi_{22} := \text{ls}(hi_1, \text{null}) \ast \text{ls}(h_1, \text{null}) \ast \text{ls}(hi_2, \text{null}) \ast \text{ls}(h_2, \text{null}) \land
  eq(hi_1, h_1) \land eq(hi_2, h_2) \tag{6.8.2}
  \]

- at line 28:
  \[
  \varphi_{28} := \text{ls}(hi_1, \text{null}) \ast \text{ls}(hi_2, \text{null}) \ast \text{ls}(h_1, \text{null}) \ast \text{ls}(h_2, \text{null}) \land
  eq(hi_1, hi_2) \land \text{sorted}(h_1) \land \text{sorted}(h_2) \tag{6.8.3}
  \]

where

\[
\begin{align*}
  eq(n, m) & := \text{len}(n) = \text{len}(m) \land \text{hd}(n) = \text{hd}(m) \land \\
  & \forall y, y'. y \in \text{tl}(n) \land y' \in \text{tl}(m) \land y = y' \implies n[y] = m[y], \tag{6.8.4}
\end{align*}
\]
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS

\[
\begin{align*}
\text{sorted}(n) &:= \forall y, y'. (y, y' \in tl(n) \land y \leq y') \implies n[y] \leq n[y'] \land \\
&\quad \forall y, y \in tl(n) \implies hd(n) \leq n[y].
\end{align*}
\]

To prove of the assert statement at line 18 is immediate because the associated formula is exactly \( \varphi_{17} \). The next assert statement is a more difficult. The formula asserted at line 23 is

\[
\varphi_{23} := ls(hi1, null) \ast ls(hi2, null) \land eq(hi1, hi2).
\]

Then, \( \varphi_{22} \supseteq_{\mathcal{HS}(U)} \varphi_{23} \) because although the underlying graphs are isomorphic the associated formulas from \( \mathcal{A}_U \) do not imply each other. The formula \( \varphi_{22} \) contains only the trivial conjunct with the guard \( P_\ast = (hi1, hi2) \) which is

\[
\forall y, y'. (y \in tl(hi1) \land y' \in tlhi2 \land y = y') \implies \top.
\]

Since \( \top \supseteq Z \{ hi1[y] = hi2[y'] \} \) the assert would fail without strengthening \( \varphi_{22} \). In this case, to prove the assert statement we use the procedure \( sat^\# \) defined in Section 6.4.3

\[
sat^\#(eq(hi1, h1) \land eq(hi2, h2)) := eq(hi1, h1) \land eq(hi2, h2) \land eq(hi1, hi2).
\]

Therefore,

\[
\varphi_{22} = \left( ls(hi1, null) \ast ls(hi1, null) \ast ls(hi2, null) \ast ls(h2, null) \land \\
\quad sat^\#(eq(hi1, h1) \land eq(hi2, h2)) \right) \subseteq_{\mathcal{HS}(U)} \varphi_{23},
\]

which proves the assert statement at line 22.

Alternatively, one could apply \( sat^\#_U \) to saturate the constraint expressed by \( \varphi_{22} \). In this case,

\[
\text{strengthen}_{\mathcal{HS}_U}(\varphi_{22}, \top \supseteq_{\mathcal{HS}(U)} \varphi_{23},
\]

where \( \text{strengthen}_{\mathcal{HS}} \) is applied over \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_U(P_\ast)) \times \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_U(P_\ast)). \)

Finally, the most interesting (and difficult) assert statement to prove is the one at line 29, which states the following formula:

\[
\varphi_{29} := ls(hi1, null) \ast ls(h2, null) \land eq(h1, h2).
\]

Unfortunately, the formula \( \varphi_{28} \) is too weak to imply \( \varphi_{29} \), w.r.t. \( \subseteq_{\mathcal{HS}} \). Moreover, none of the saturation procedures (neither the one given in Section 6.4.3 nor the one based on the partial reduction operator \( \text{strengthen}_{\mathcal{HS}} \)) would not strengthen \( \varphi_{28} \) enough to prove the desired entailment relation. The solution to prove this assert statement is to combine the analysis in \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_U) \) with the analysis in \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_M) \) using \( \text{strengthen}_{\mathcal{HS}_M} \).

The assertion generated at line 28 with \( \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_M) \) is:

\[
\varphi_{28}^M := ls(hi1, null) \ast ls(hi2, null) \ast ls(h1, null) \ast ls(h2, null) \land \\
\quad ms(hi1) = ms(h1) \land ms(hi2) = ms(h2) \land ms(hi1) = ms(hi2)
\] (6.8.5)

Then, we apply \( \text{strengthen}_{\mathcal{HS}_M} : \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_U) \times \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_M) \rightarrow \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_U) \times \mathcal{A}_{\mathcal{HS}}(\mathcal{A}_M) \) where \( \mathcal{A}_U \) is parametrized by \( \{ P_\ast, P_1, P_2 \} \) to the pair \( (\varphi_{28}, \varphi_{28}^M) \) which returns:

\[
\text{strengthen}_{\mathcal{HS}_M}(\varphi_{28}, \varphi_{28}^M) = (\varphi_{28}', \varphi_{28}^M)
\]

where

\[
\varphi_{28}' = \varphi_{28} \land eq(h1, h2)
\]

and \( \varphi_{28} \supseteq_{\mathcal{HS}_U} \varphi_{29} \) which proves the assert statement at line 29. So, it concludes the proof of the equivalence of the two sorting procedures.

258
6.9 Conclusion

We have defined an approach based on abstract inter-procedural analysis for the automatic synthesis of complex assertions about recursive programs manipulating singly-linked data structures with unbounded data. This approach is powerful enough to deal with a wide spectrum of programs including programs performing list traversal to search or to update data, programs with destructive updates and changes in the shape (e.g., list dispatch or reversal, sorting algorithms such as insertion sort), and programs computing complex arithmetical relations.

The techniques introduced are able to synthesize ordering constraints, data preservation constraints like those in (1.2.4) from Section 1 or (6.5.1) from Section 6.5, relations between data and lengths of lists, e.g. (6.6.2), and complex arithmetical relations, e.g. (6.6.1) from Section 6.6. Besides constraints which affect only one list, they are able to synthesize relations on data from different lists. For example, the program that creates a copy of a list, generates the post-condition given in (6.8.4).
CHAPTER 6. ANALYSIS OF PROGRAMS MANIPULATING SINGLY-LINKED LISTS
In this thesis we have introduced a logic-based framework for reasoning about programs manipulating dynamic data structures. We have investigated the problem of proving partial correctness of such programs, and we have provided solutions for different classes of specification languages. More precisely, we have tackled the problem of checking user defined assertions but also the problem of synthesizing assertions that describe the behavior of the program.

Our framework is based on a general logic, called $gCSL$, that is able to capture properties of various data structures, such as the values of the pointer fields, the size of the allocated data structures and the values of the data fields and data variables. We have identified fragments of $gCSL$ in which checking the validity of Hoare triples is possible, either due to decidability results (the fragment is called $CSL$), or by using approximate methods (the fragment is called $SL3$). Moreover, we introduced techniques to automatically synthesize assertions in $SL3$.

Decidable logics for program verification: In Chapter 5 we introduced a logic, called $CSL$, for describing shape related properties of complex (heap) data structures involving pointers and arrays. Specifically, $CSL$ allows the expression of some form of reachability properties, linear constraints to reason about the size of the allocated memory, and also of constraints on the data contained in the memory locations. The satisfiability problem for this logic is shown to be decidable provided the underlying logic of the data properties is itself decidable. Also, the strongest post-condition is definable in $CSL$, which makes this fragment suited for program verification. To express invariants and post-conditions one has to use the fragment of $CSL$ which is closed under negation.

The main contributions of $CSL$ lie in the combination of reachability and data constraints with size constraints and in the fact that is supports some form of quantifier alternation. The decidability of the satisfiability problem for $\exists^*\forall^*$ formulas combining reachability and data predicates with size constraints is obtained based on a small model property. The cruxes of this result is the use of multi-objective integer linear programming to determine a bound of the models.

The $\forall^*\exists^*$ quantifier alternation is defined according to an ordered partition over the record types used in the program. This partition decomposes the heap graphs into classes and assigns to each vertex in the graph a level. Then, roughly speaking, by associating a level to each variable, the quantification part of $CSL$ formulas has the form $\exists^*_k\forall^*_k\exists^*_{k-1}\forall^*_k\ldots\exists^*_1\forall^*_1$. The decidability of this form of quantifier alternation is based on the fact that, the partitioning allows us to apply the small model property (for $\exists^*\forall^*$) iteratively, starting from the $k$-th level to the first one.

Future work: The definition of $CSL$ is based on syntactical restrictions allowing to prove a small model property for the logic. This property is lost if any of these restrictions...
is relaxed (in the sense that the size of the minimal models cannot be computed from
the syntax of the formula, regardless of the considered data domain: it can rather be
arbitrarily large depending on this domain). Nevertheless, it could be possible that some
of these restrictions be relaxed without loss of decidability. For instance, to allow data
properties that relate any two successive vertices in the heap graph, one could adopt an
alternative approach such as the one used in [Habermehl 2008] which consists in reducing
the satisfiability problem to the reachability problem in a class of automata.

Another aspect that can be the object of further investigation is to identify different
classes of linear constraints for which the complexity of multi-objective linear program-
ing is not exponential. For example, solving MOIPL for systems of difference bound
constraints can be done in polynomial time. However, this would not be enough to im-
prove the complexity of checking the validity of Hoare triples. One needs to identify
classes of programs that preserve the size constraints in the considered form during the
computation of the strongest post-condition.

Program analysis: We introduce an accurate inter-procedural abstract analysis for
synthesizing assertions of programs with lists and without procedure calls.

We define abstract domains whose elements, called abstract k-SLL heap sets, are sets of
pairs composed of a heap backbone and an abstract data constraint. The heap backbone is
an abstraction of the heap graph (the graph representing the allocated memory) where only
a bounded number of nodes are kept, including all sharing nodes and all the nodes pointed
to by program variables. An edge in the backbone represents a path (without sharing
nodes) relating the source and target nodes in the original heap. The data constraint
is given as an element of some abstract domain, and allows to specify properties of the
data sequences represented by the edges of the heap backbone. The provided analysis is
based on (a) unfolding the structures in order to reveal the properties of some internal
nodes in the lists, which makes necessary to introduce in the structures some nodes, called
simple nodes, others than the sharing nodes or those pointed to by variables, and then (b)
folding the structures by eliminating the simple nodes and in the same time collecting the
informations on these nodes using a formula that speaks about sequences of data. The
analysis is iterated several times, which may lead to additional unfoldings and foldings.

Then, widening techniques on numerical domains are used in order to force termination.

Several abstract domains are defined for the analysis, where elements are (1) formulas
in a universally quantified fragment of the first-order logic over data words, denoted \( \mathcal{A}_U \),
(2) conjunctions of equality constraints between unions of multisets of data of the words,
denoted \( \mathcal{A}_M \), or (3) linear constraints on the sums of data of the words, denoted \( \mathcal{A}_S \).

The formulas in the abstract domain \( \mathcal{A}_U \) contain a (quantified) universal part which
is a conjunction of formulas \( \forall \mathbf{y}. \ (P \Rightarrow U) \), where \( \mathbf{y} \) is a vector of variables interpreted as
positions in the words, \( P \) is a constraint on the positions (seen as integers) associated with
the \( \mathbf{y} \)’s, and \( U \) is a constraint on the data values at these positions. It is assumed that \( P \)
is obtained from a finite set of fixed patterns corresponding to, e.g., order constraints or
difference constraints.

Abstract k-SLL heap sets over first order formulas are represented by formulas in SL3,
a fragment of gCSL. Consequently, the invariants and the procedure summaries generated
by an analysis over such abstract k-SLL heap sets are formulas in SL3. Moreover, this
analysis can be tuned to synthesize assertions in the intersection between SL3 and ICSL.
In this way the analysis can be seen as an oracle for the program verification task.
For procedure calls we use the local semantics introduced in [Rinetzky 2005a], which allows to compute procedure summaries by considering only the reachable part of the heap from its actual parameters. In order to increase the precision of the computed summaries, we strengthen the analysis in the domain of first-order formulas by the analysis in the multiset domains, via an instance of a generic partial reduction operator, called \texttt{strengthen}^{HS}. Another instance of \texttt{strengthen}^{HS} is used to improve the entailment relation between \textsc{sl3} formulas. This also allows to soundly convert any \textsc{sl3} formula into an \textsc{icsl} formula.

**Future work:** Several extensions of this work are foreseen. In particular, we have discarded in this work the issue of dealing with cut-points in order to concentrate on the problems related to data manipulation. These issues are indeed somehow orthogonal. The extension of our work to programs with a bounded number of cut-points is possible without major technical problems. However, the extension to the case of programs with an unbounded number of cut-points is a challenging problem, regardless from the problem of dealing with data in the structures. We think however that using abstract domains with first-order constraints can be useful to tackle this problem since formulas could be used to characterize the positions of the cut-points in the structures.

In this thesis we have restricted the class of data structures to singly-linked lists. An interesting problem is to investigate extensions to a larger class of structures where multiple linking is allowed (such as doubly-linked lists, trees or hash tables) as well as compositions of these multi-linked data structures. One idea to do this is to define embeddings of the heap graph into finite abstract graphs whose nodes represent sub-structures described by some abstract data constraints.

Finally, concerning the analysis, we have shown that multiset domains are useful for strengthening the analysis with universally quantified first-order formulas. The multiset constraints capture bijections modulo equality of elements. The same techniques can be extended to the case of bijections modulo other kinds of relations such as bounded difference, etc. Then, it could be interesting to define more general partial reduction operators allowing to transfer more complex relations between abstract domains in order to compose accurately assertions during the inter-procedural analysis.


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


