Separation logic : expressiveness, complexity, temporal extension

Rémi Brochenin

▶ **To cite this version:**

Rémi Brochenin. Separation logic : expressiveness, complexity, temporal extension. Other [cs.OH]. École normale supérieure de Cachan - ENS Cachan, 2013. English. NNT : 2013DENS0033 . tel-00956587

HAL Id: tel-00956587

https://theses.hal.science/tel-00956587

Submitted on 6 Mar 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Normale Supérieure de Cachan
École Doctorale Sciences Pratiques

Thèse soumise pour l'obtention du diplôme de doctorat,
dans le domaine de l'informatique,
avec une soutenance le 25 Septembre 2013

Logique de séparation: expressivité, complexité, extension temporelle

Rémi Brochenin,
sous la direction de Stéphane Demri et Étienne Lozes

Panel:

Stéphane Demri	Directeur de Recherche, LSV, CNRS	Directeur
Peter Habermehl	Maître de Conférences, LIAFA, Univ. Paris Diderot	Rapporteur
Didier Galmiche	Professeur, LORIA, Univ. Henri Poincarré	Rapporteur
Arnaud Durand	Professeur, IMJ, Univ. Paris Diderot	Examinateur
Jacques Blanc-Talon	Docteur, Direction Générale de l'Armement	Examinateur

Recherche réalisée au Laboratoire Spécification et Vérification
et financée par la Direction Générale de l'Armement

Court Résumé

Cette thèse étudie des formalismes logiques exprimant des propriétés sur des programmes. L'intention originale de ces logiques est de vérifier formellement la correction de programmes manipulant des pointeurs. Dans l'ensemble, il ne sera pas proposé de méthode de vérification applicable dans cette thèse; nous donnons plutôt un éclairage nouveau sur la logique de séparation, une logique pour triplets de Hoare. Pour certains fragments essentiels de cette logique, la complexité et la décidabilité du problème de la satisfiabilité n'étaient pas connus avant ce travail. Aussi, sa combinaison avec certaines autres méthodes de vérification était peu étudiée.

D'une part, dans ce travail nous isolons l'opérateur de la logique de séparation qui la rend indécidable. Nous décrivons le pouvoir expressif de cette logique, en la comparant à des logiques du second ordre. D'autre part, nous essayons d'étendre des fragments décidables de la logique de séparation avec une logique temporelle et avec l'aptitude à décrire les données. Cela nous permet de donner des limites à l'utilisation de la logique de séparation. En particulier, nous donnons des limites à la création de logiques décidables utilisant ce formalisme combiné à une logique temporelle ou à l'aptitude à décrire les données.

École Normale Supérieure de Cachan
École Doctorale Sciences Pratiques

Thesis submitted for the degree of doctorate,
in the field of computer science,
with a defense on September the 25th, 2013

Separation Logic:
Expressiveness, Complexity, Temporal Extension

Rémi Brochenin,
under the supervision of Stéphane Demri and Étienne Lozes

Panel:

Stéphane Demri	Directeur de Recherche, LSV, CNRS	Supervisor
Peter Habermehl	Maître de Conférences, LIAFA, Univ. Paris Diderot	Reviewer
Didier Galmiche	Professeur, LORIA, Univ. Henri Poincarré	Reviewer
Arnaud Durand	Professeur, IMJ, Univ. Paris Diderot	Examiner
Jacques Blanc-Talon	Docteur, Direction Générale de l'Armement	Examiner

Research conducted in Laboratoire Spécification et Vérification
and supported by Direction Générale de l'Armement

Abstract

This thesis studies logics which express properties about programs. These logics were originally intended for the formal verification of programs with pointers. Overall, no automated verification method will be proved tractable here; rather, we give a new insight on separation logic. The complexity and decidability of some essential fragments of this logic for Hoare triples were not known before this work. Also, its combination with some other verification methods was little studied.

Firstly, in this work we isolate the operator of separation logic which makes it undecidable. We describe the expressive power of this logic, comparing it to second-order logics. Secondly, we try to extend decidable subsets of separation logic with a temporal logic, and with the ability to describe data. This allows us to give boundaries to the use of separation logic. In particular, we give boundaries to the creation of decidable logics using this logic combined with a temporal logic or with the ability to describe data.

Contents

Introduction **11**

A Context . . . 11

A.1 Verification . . . 11

A.2 Verification of Programs with Pointers . . . 12

A.3 Separation Logic . . . 13

B Questions Addressed in this Thesis . . . 14

B.1 Complexity of Separation Logic . . . 14

B.2 Expressiveness of Separation Logic . . . 15

B.3 Data . . . 15

B.4 Towards a Temporal Separation Logic . . . 16

C Contributions of this Thesis . . . 17

C.1 Magic Wand and Separation Logic . . . 17

C.2 Ordered Data and Separation Logic . . . 17

C.3 Temporal Separation Logic . . . 17

1 Preliminaries **19**

1.1 Memory Model . . . 20

1.1.1 Memory States . . . 20

1.1.2 Memory Shapes . . . 22

1.1.3 Simple Memory States . . . 22

1.1.4 Simple Memory Shapes . . . 23

1.2 First and Second-Order logic on Simple Memory Shapes . . . 23

1.2.1 Conventions on Variables . . . 23

1.2.2 Second-Order Logic . . . 24

1.2.3 Conventions on Formulas and Languages . . . 25

1.3 Separation Logic . . . 27

1.3.1 Definition . . . 27

1.3.2 Fragments of Separation Logic on Memory Shapes . . . 29

1.3.3 A Separation Logic for Simple Memory States . . . 29

1.3.4 Separation Logic on Simple Memory Shapes . . . 30

1.4 Simple Predicates in Separation Logic . . . 30

1.4.1 Allocated Memory Cells . . . 30

1.4.2 Predecessors and Arithmetical Constraints . . . 30

1.4.3 Reachability and List Predicates . . . 31

2 On the Almighty Wand **33**
2.1 A Decidable Fragment with a Restricted Wand 35
2.1.1 A Complexity Result without Wand 35
2.1.2 A Restricted Use of the Wand 37
2.1.3 Preliminaries to the Translation 38
2.1.4 The Translation 41
2.2 Advanced Arithmetical Constraints with the Wand 43
2.2.1 Comparing Two List Lengths 44
2.2.2 Comparing the Numbers of Predecessors 45
2.3 Equivalence to Second-Order Logic 53
2.3.1 Preliminaries 53
2.3.2 Encoding Environments 57
2.3.3 The Translation 65
2.3.4 Correctness 66
2.4 Extensions with More Than one Selector 71

3 Beyond Shapes: Lists with Ordered Data **77**
3.1 Decidability of Short-Distance Comparisons 78
3.1.1 Method 78
3.1.2 Constraints 80
3.1.3 Recursive Translation 85
3.2 Long-Distance Comparisons 87
3.2.1 An Undecidability Result 87
3.2.2 Decidability of Guarded Long-Distance Comparisons 90
3.3 Magic Wand and Restricted Magic Wand 93

4 Reasoning about Sequences of Heaps **99**
4.1 Preliminaries 100
4.1.1 Temporal Models and Programs 100
4.1.2 Temporal Extension: our Logic 102
4.1.3 Satisfiability and Model-Checking 103
4.1.4 Basic Results 104
4.2 Separation Logic: Complexity and Abstraction 105
4.2.1 Syntactic Measures 105
4.2.2 Complexity of Quantifier-Free Separation Logic 109
4.3 Decidable Problems by Abstracting Computations 115
4.3.1 Symbolic Models 115
4.3.2 Omega-Regularity and Polynomial Space Upper Bound 116
4.3.3 Other Decidable Problems 119
4.4 Undecidability Results 122

Conclusion **131**

Table of Notations **133**

Bibliography **142**

Introduction

A Context

A.1 Verification

Mistakes are frequent in programming and may have dramatic consequences. A famous example is the crash of Ariane 5 due to a division by zero. Formal verification aims at making mathematically certain that programs do what they are intended to. A mathematical proof of the correctness of a program could be much more reliable than testing it intensively, or examining the code carefully. Proving correctness of a program demands however a lot more efforts from the programmers or the people in charge of writing the specifications and assertions.

We are interested in the part of formal verification that attempts at automatically generating mathematical proofs of programs. In other words, it aims at creating programs checking that other programs match a specification. Rice's theorem tells us that this problem is undecidable. Formal verification hence either focuses on specific features of the programs or is not fully automated. Still, the complexity of verification tasks is generally high.

Indeed, programs often have a very elaborate design, which may for instance use concurrency, use recursive procedure calls, rely on arithmetic properties, or manipulate recursive data structures. Such programs usually have an infinite state space which make them hard to verify by naive state space exploration. Additionally, the configurations of such programs may be hard to describe, especially in the case of complex data structures. Nonetheless, the verification problem may be decidable for some infinite-state systems, and it is a very active area of formal verification to identify such infinite-state systems (as explains the book chapter of Burkart et al. [32]), such as Petri nets, timed automata, etc. However, infinite-state systems most of the time have undecidable verification problems, and one usually aims at defining subclasses of programs and specifications for which the verification problem is decidable.

Let us briefly present three formal verification techniques. One method is abstract interpretation, which approximates the steps of the execution of a program in a sound - but not complete - way. The set of possible configurations of the system, possibly infinite, is abstracted into an abstract domain in which each element represents a set of possible configurations. Then the program is simulated on the abstract domain. This technique has been formally described by Cousot and Cousot in [40]. As an example of an existing tool using this method, we can quote Astrée, presented by Blanchet et al. in [14]. Among the abstract interpretation techniques, we should give two examples of special interest regarding this thesis, as they study recursive structures as well as data: the works of Bouajjani et al. in [20] and of Gulwani, McCloskey and Tiwari in [57] both use abstract interpretation as well as guidance from the user so as to

generate annotations of programs.

Another method is model-checking, where the input of the program doing automated formal verification is generally an automaton, or another abstraction of a program. An example of use of the model-checking method is Regular Model-Checking, described by Bouajjani et al. in [23]. In this framework, the models are abstracted by trees or words, for instance a word can naturally represent singly-linked lists. Sets of models are abstracted by tree or word automata. Then, program instructions can be abstracted by transducers. Also, with the model-checking method, temporal logics allow to work in the very convenient framework of programs-as-formulas, and decision procedures for logical problems can be directly used for formal verification. Indeed, in this framework programs as well as formulas can be turned into automata, and conversely; as a consequence, checking that the automaton model of the program satisfies a property is done by computing whether inappropriate states are reached in an automaton. This automata-based approach stems from the famous result showing the equivalence between monadic second-order logic and Büchi automata as far as definability of languages of infinite words are concerned, shown by Büchi in [31]. An example of application of this method is the Spin tool, presented by Holzmann in [60]. Our last chapter is related to this method.

Finally, an interesting framework for formal verification is Hoare logic, introduced by Hoare in [58]. Hoare logic is a proof system based on assertions called Hoare triples of the form $\{f_1\}\texttt{instr}\{f_2\}$ *(Tri)* where `instr` is an instruction or a program, f_1 is the precondition stated in some logical formalism, and f_2 is the postcondition. The precondition is assumed to be true before the execution of `instr`, and the postcondition has to be true after the execution of `instr` under the assumption that f_1 held before. The formulas of these triples are usually provided by the user, hence the input is an annotated program, but they may also be automatically synthesized. Annotated programs are then verified by checking that each triple is valid. In practice, a formula f'_2 can be computed such that $\{f_1\}\texttt{instr}\{f'_2\}$ is valid, and for which the validity of the Hoare triple in *(Tri)* reduces to the one of the logical entailment $f'_2 \vdash f_2$ *(Ent)*. Examples of tools using this method are the Key System, presented by Ahrendt et al. in [1], and Why, presented by Filliâtre and Marché in [50]. Chapters 2 and 3 of this thesis are related to this method.

From this last perspective, formal verification in Hoare logic can be reduced to a purely logical problem, and decidability results for logics are a prior complementary guide for the creation of logics whose aim is formal verification, before the study of tractability.

A.2 Verification of Programs with Pointers

Programming languages with explicit memory management, such as C, expose the programmer to many sources of potential bugs, apart from the more usual problems studied by formal verification. Firstly, there are problems related to the use of recursive data structures such as lists, doubly-linked lists, trees, etc. These are for instance the undesired creation of cycles in a recursive structure, or memory leaks. Secondly, many bugs are due to the nature of pointers. These are for instance: null pointer dereferences, dangling pointers, or undesired aliasing. Avoiding these problems is important for the safety of a program, for its efficiency, for its termination, and last but not least for its security – think about buffer overflow attacks or non-interference requirements.

These specific problems need specific answers, for checking specific properties. Examples of such properties can be very simple to state, for instance that there is no null pointer dereference, or that no block of memory is freed more than once, or that a critical part of the memory can never be reached. It can be also less straightforward to specify, for instance that the output of a program is a binary search tree. Providing formal verification methods for fault detection in such programs that manipulate recursive mutable data structures is a long-standing open problem. From the theoretical point of view, these programs present the same challenges as infinite-state systems, even for singly-linked lists. There is indeed a potentially infinite set of memory states for these programs, due to the recursive nature of lists, and this makes the problem of the reachability of a program point undecidable in the approaches of Bardin, Finkel and Nowak in [8] and of Bouajjani et al. in [17].

We will mostly focus on shape properties. The term "shape" refers to the data structures in which all data are ignored, and only the graph of links matters. Shape properties aim at detecting faults due to in-depth properties of the heap, for instance we may want to check that a program does not create a cycle in an acyclic list. A similar example of shape property is that the memory heap keeps the shape of a tree all along its execution. The non-existence of memory leaks also belongs to this category of properties. Shape analysis focuses on shape properties. It is a well established approach for the static analysis of programs with recursive data structures. The main idea is to summarize a set of objects forming a recursive structure, for instance by storing the fact that there are $\mathtt{n}$ nested nodes, instead of storing all of the $\mathtt{n}$ nodes of a list, while doing the verification. Prominent logics that have been used as abstract domains for such an analysis are pointer assertion logic presented by Jensen et al. in [63], three-valued logic assertions presented by Lev-Ami and Sagiv in [69], or more recently separation logic (leading to the tools Space Invader presented by Yang et al. in [91], and Xisa presented by Rival and Chang in [85]).

Extensions of shape analysis have been proposed for ordering properties, stability properties, and size properties; to cite a few of these extensions, there are the shape graphs by Bouajjani et al. in [17], the three-valued logic approach by Loginov, Reps and Sagiv in [70], and the separation logic approach by Nguyen et al. in [77]. However, fully automatic analyses that are data sensitive are hard to design. The recent approaches already mentioned above of Bouajjani et al. in [20] and of Gulwani, McCloskey and Tiwari in [57], rely on user-defined annotations in expressive logics for graphs with data, and propose to leverage the amount of annotations by guessing some of them by means of shape analyses.

In all of these works, it is insightful to have a good understanding of the expressiveness of the logical formalism which is used, as well as of the complexity of solving the entailment problem *(Ent)* for this formalism.

A.3 Separation Logic

As already mentioned, aliasing is one of the features of pointer-manipulating programs which introduces a lot of complexity into the verification process. For instance, a same field in memory can be accessed by several variables or even by several threads. The complexity of aliasing is particularly sensible in the proofs based on Hoare logic. Separation logic (SL) is an extension of Hoare logic which has the ability to isolate the part of the memory over which a program

works, so that the rest of the memory becomes irrelevant for the proof of the program. This principle, often called local reasoning, makes the formal verification of programs rather modular and achieves a better scalability of both fully automatic shape analyses and user-guided proofs.

The original assertion language SL, which we may call from time to time separation logic as well, extends first-order logic with two substructural connectives. The first one, the separating conjunction ($*$), is the key ingredient for expressing concisely non-aliasing properties. The second one, the separating implication ($-\!*$) also known as the magic wand, is the adjunct of the first one, and finds its roots in the logic of bunched implications which is an ancestor of SL. The logic of Bunched implications has been introduced first by O'Hearn and Pym in [79] and then by Pym in [81]. Separation logic has been introduced as a special case of the bunched implications logic on specific models by Reynolds in [84] and Ishtiaq and O'Hearn in [61]. The operators of SL make the specification of the effect of instructions of programs with pointers very easy and readable for humans.

Several fragments of SL have been studied from the decidability point of view, in particular in the work of Calcagno, Yang and O'Hearn [36, 35]. The early works on the decidability of SL have shown undecidability in large categories of cases, especially for fragments over models with multiple selectors. An interesting fragment of separation logic is of special interest from the complexity point of view, the so-called symbolic heaps introduced by Berdine, Calcagno and O'Hearn in [11]. Although few features of the original separation logic are present, it has deserved a special attention thanks to its implementation in the Smallfoot tool described by Berdine, Calcagno and O'Hearn in [13]. Indeed, its complexity is tractable as it has recently been proved that logical entailment can be decided in polynomial time for this fragment by Cook et al. in [38].

Almost all of the decidable fragments do not include the magic wand connective in their syntax. This restriction makes sense for what user-defined annotations are concerned, since these annotations usually express rather simple properties (such as the presence of two lists without alias). Nevertheless the magic wand can play an important role in many problems that separation logic has to face, and is needed for instance in frame inference, abduction, closure under interferences, or the ramification rule.

One of the most challenging problems in separation logic is to prove decidability for classes of properties that can be expressed with the magic wand as wide as possible. Additionally, it is specially interesting to study the root of this logical language, with all of its features.

B Questions Addressed in this Thesis

This thesis aims at improving the understanding of the assertion language of separation logic from the complexity and expressiveness points of view, with respect to three different aspects: the magic wand, the properties involving data constraints, and the temporal properties.

B.1 Complexity of Separation Logic

The complexity of the satisfiability, the model-checking, the validity and the entailment problems have been intensively studied until quite recently, in particular in the articles mentioned in the previous section [36, 35, 84, 38]. For instance, first-order separation logic over heap models with at least two selectors – or record fields – is known to be undecidable from [36]. This result is there shown even with no separating connectives, by containment of finite satisfiability for classical predicate logic with one binary relation, which is proved undecidable by Trakhtenbrot in [88].

The magic wand connective can make any of the above-mentioned problems quite difficult to decide (note that these problems are often inter-reducible in presence of magic wand). The expressive power of $\mathrel{-\!\!*}$ is increased by the first-order quantification: SL without magic wand is known to be equivalent to a classical propositional logic if first-order quantifiers are disabled, as proved by Lozes in [72], whereas no adjunct elimination holds for SL with first-order quantifiers as proved by Dawar, Gardner and Ghelli in [41] and Lozes in [73]. The same gap exists with respect to decidability: SL without first-order quantifiers is decidable, but it becomes undecidable if first-order quantifiers are taken into account.

These results however crucially rely on cells having two record fields. On the other hand, many of the case studies that are addressed by separation logic tools have to deal with singly-linked lists only. The complexity of SL with the magic wand for memory models with only one record field, despite being a natural question, was open before our work.

B.2 Expressiveness of Separation Logic

Another natural question about separation logic is how it compares with second-order logic (SO) and its fragments. This is a very natural question for at least three reasons. Firstly, separating conjunction and its adjunct are essentially second-order connectives (see also a similar concern on graphs with spatial logics in the work of Dawar, Gardner and Ghelli in [42]), which clearly makes SL be a fragment of SO. Secondly, many properties on heaps require second-order logic, for instance to express recursive predicates, or list and tree properties. Thirdly, SO is usually expressive enough to enforce the completeness of the Hoare logic, and a better understanding of the relationship between SL and SO could serve to derive the completeness of the proof system of separation logic.

There are well-known examples of correspondences between logics inspired by computer science problems and more mathematical logics. The celebrated Kamp's theorem [64] states that linear-time temporal logic (LTL) is as expressive as first-order logic; here LTL has only the strict until and since operators. This result is refined by Etessami, Vardi and Wilke in [49] where it is shown that unary LTL is as expressive as first-order logic restricted to two individual variables. Similarly, the Janin-Waluckiewicz theorem [62] states that the modal mu-calculus is equivalent to the bisimulation-invariant monadic second-order logic.

However, no correspondence between separation logic and any mathematical logic was known before the work that has led to this thesis. Kuncak and Rinard explored the relationships between a logic with separation operators and a second-order logic in [67], but they considered as models arbitrary first-order structures, and not the standard, finite heap model of SL.

B.3 Data

As mentioned above, standard analyses on recursive data structures restrict their attention to shape properties, excluding properties that deal with the actual content of these structures. Decidable logics handling data exist, of which Presburger arithmetic is a standard example, but interactions between data and memory shapes are very hard to handle. Defining decidable formalisms on models with recursive structures and data is very challenging.

For instance, first order logic over finite data words is known to be undecidable, as proved by Bojańczyk et al. in [15], and can be encoded with limited syntactic resources if the model contains lists labelled with data. Additionally, all the formal verification methods we are aware of in this field use approximation techniques, strong restrictions on the syntax or strong restrictions on the data. The approximation techniques are actually in general abstract interpretation techniques which prevent completeness of the method, such as the work of Berdine et al. [10]. An example of work based on strong restrictions of the syntax is the logic of McPeak and Necula in [76], a first order logic which can handle lists but does not contain the negation of the equality between locations. An example of restriction on the model is the work of Yorsh et al. in [92], which can handle complex memory shapes but assumes the data belongs to a finite domain.

An interesting question for logics dealing with data is to identify decidable fragments which are expressive. In the context of separation logic, our question becomes: what are the restrictions that should be considered on SL to make it decidable and still expressive enough for annotating classical list programs with data properties?

B.4 Towards a Temporal Separation Logic

The assertion language SL is a state logic, mostly because it is the assertion language of a Hoare logic, and Hoare logic traditionally deals with state assertions.

It is however tempting to introduce some forms of temporal reasoning in the assertion language of separation logic. There are two main motivations for this; firstly, recent semantics of Hoare logic are based on the interpretation of programs as trace transformers, and not just state transformers, as explains the work of Hoare et al. [59]; secondly, temporal reasoning may help describing recursive data structures by means of properties over the traces of the programs that traverse them.

Among temporal logics, LTL, presented by Pnueli in [80], is often one of the favorites, mostly because of the equivalent decidable decision problems based on automata that have been developed around it, and which are described in the works of Vardi and Wolper [89]. In the context of traces of data-manipulating programs, LTL has been extended so as to express relations between data at different points of the execution. These extensions often complicate the design of equivalent automata for LTL, and may even sometimes introduce undecidability. Although these extensions are relatively well studied for data models such as integers, the classification of these extensions of LTL in computational complexity classes is relatively little studied for heap data structures.

The introduction of temporal reasoning in separation logic raises several questions: First, as arithmetical constraints in temporal logics are known to easily lead to undecidability, (see for instance the works of Bouajjani, Echahed and Habermehl [21], of Comon and Cortier [37], or of Demri and Gascon [44]) how can the logic be kept decidable? Then, what semantics

should the logic have, how can it be encoded into automata, and how expressive should the data constraints across time be?

C Contributions of this Thesis

This thesis presents new results about the decidability, the complexity, and the expressive power of separation logic formalisms that include either the magic wand, data constraints, or temporal reasoning.

C.1 Magic Wand and Separation Logic

In chapter 2, we investigate decidability, complexity and expressive power issues for first-order separation logic with one record field. We consider on the one hand SL without restrictions – including the magic wand, and on the other hand SL without the magic wand. The main result we establish is that SL is as expressive as SO. As a by-product, this shows the undecidability of SL. We refine this result by showing that SL without the separating conjunction is as expressive as SL, in other words that the magic wand can simulate the separating conjunction.

By contrast, we establish that SL without the magic wand is less expressive than the monadic fragment of SO; we also establish that SL without magic wand is decidable, although with a non-elementary complexity. We extend this result for restricted cases where the magic wand occurs in formulas. We also generalize our main result to heaps with an arbitrary number of fields: for $k \in \mathbb{N}$, we show that kSL, the separation logic over heaps with k record fields, is equivalent to kSO, the second-order logic over heaps with k record fields.

C.2 Ordered Data and Separation Logic

In chapter 3, we propose a general approach for reducing the shapes handling ordering properties to pure shapes, and stress some natural limitations we should put on data properties in order to check them automatically. To our knowledge, no predicate dealing with data had ever been integrated to separation logic while preserving decidability as well as correctness before our work. We establish decidability for (first-order) separation logic with a predicate that allows to compare two successive data in a list. We then consider the extension where two data in arbitrary positions may be compared, and establish the undecidability in general. We also replace long distance comparisons by guarded comparisons of data, allowing to compare the data pointed to by a program variable to any other data, which provides an interesting decidable logical fragment. We finally consider the extension with the magic wand and prove that, in contrast with the data-free case, even a very restricted use of the magic wand already introduces undecidability.

C.3 Temporal Separation Logic

In chapter 4, we will introduce a temporal logic $\mathsf{LTL}^{\mathsf{mem}}$ whose underlying assertion language is the quantifier free fragment of separation logic and the temporal logic on the top of it is the standard linear-time temporal logic LTL. We analyze the complexity of various model-checking and satisfiability problems for $\mathsf{LTL}^{\mathsf{mem}}$, considering various fragments of separation logic (including

pointer arithmetic), various classes of models (with or without constant heap), and the influence of making the initial memory shape a part of the input of the problem. We will have a complete picture based on these criteria. Our main decidability result is PSPACE-completeness of the satisfiability problems on two fragments of our logic. We moreover establish Σ^0_1-completeness or Σ^1_1-completeness of various problems by reducing standard problems for Minsky machines, and we eventually give a rather detailed picture of the complexity of this approach to temporal reasoning in separation logic.

Chapter 1

Preliminaries

Introduction

Contents of this Chapter

In this section, we introduce mathematical notions which will be used throughout the whole document, in particular a model of the memory, a definition for separation logic and some examples of its expressiveness. This section will mainly give basic information about separation logic.

Structure of the Chapter

First, we will introduce a general definition of memory states – our model of the memory – of which we will define three subsets used as simpler classes of models: memory shapes, simple memory shapes and simple memory states. Each of these classes of models will be used later in one of the three main chapters. Figure 1.1 are summarized the models and their characteristics. They differ on the ability to contain data in fields of a cell, and on the possibility of the presence of more than one address field in one memory cell.

Then, we introduce first-order and second-order logic on the simplest of these models, simple memory shapes. In the meanwhile, we also explain our conventions on variables as well as general definitions about formulas and logics.

Then, we introduce separation logic. There will first be the formal definition of a general separation logic. This will allows us to define formally its operators able to modify the model, that we have described without being precise yet. Similarly to the models, we will introduce three fragments of separation logic, with or without data, with or without multiple selectors, corresponding to the three main chapters of this thesis. Fragments of these fragments, according to additional characteristics, will be introduced. Figure 1.2 summarizes the fragments of separation logic introduced in the whole document, with their features and their models, for a reference purpose.

Finally, we provide examples of properties that can be expressed in our formalisms. We start with simple properties on allocation of memory cells, and with simple arithmetical constraints on the amount of predecessors of a vertex in the graph one of our models is equivalent to. We will end this section with the definition of reachability predicates and lemmas proving their semantics.

	several selectors	one selector
with data	$\mathtt{Heaps}_{\mathtt{sv}}$	$\mathtt{Heaps}_{\mathtt{v}}$
without data	$\mathtt{Heaps}_{\mathtt{s}}$	$\mathtt{Heaps}$

Figure 1.1: Models

	$\mathrel{-\!\!*}$	$*$	$\exists$	several selectors	pointer arithmetic	data and comparisons	location of definition
SL	✓	✓	✓				1.3
$\mathrm{SL}_{\mathtt{v}}$		✓	✓			✓	1.3
$\mathrm{SL}_{\mathtt{s}}$	✓	✓		✓	✓		1.3
$\mathrm{SL}_{\mathtt{sv}}$	✓	✓	✓	✓	✓	✓	1.3
SL^{*}		✓	✓				1.3
$\mathrm{SL}^{\mathrel{-\!\!*}}$	✓		✓				1.3
$\mathrm{SL}^{*,\mathrel{-\!\!*}_{\mathtt{n}}}$	$\mathrel{-\!\!*}_{\mathtt{n}}$	✓	✓				2.1
$\mathrm{SL}^{<\mathtt{n}}$	restricted	✓	✓				2.1
$\mathrm{SL}_{\mathtt{v}}^{\mathtt{short}}$		✓	✓			$\hookrightarrow_{\leq}$	3.1
$\mathrm{SL}_{\mathtt{v}}^{\mathtt{R},\mathrel{-\!\!*}_{1}}$	$\mathrel{-\!\!*}_{1}$	✓	✓			$\hookrightarrow_{\mathtt{R}}$	3.3
$\mathrm{SL}_{\mathtt{v}}^{\mathtt{guarded}}$		✓	✓			restricted	3.2
$\mathrm{SL}_{\mathtt{v}}^{\mathtt{long}}$		✓	✓			✓	3.2
$\mathrm{SL}_{\mathtt{v}}^{\mathtt{longeq}}$		✓	✓			restricted	3.2
$\mathrm{SL}_{\mathtt{s}}^{*}$		✓		✓	✓		1.3
$\mathrm{SL}_{\mathtt{s}}^{\mathtt{CL}}$				✓	✓		1.3
$\mathrm{SL}_{\mathtt{s}}^{\mathtt{RF}}$	✓	✓		✓			1.3
$\mathrm{SL}_{\mathtt{s}}^{\mathtt{LF}}$							1.3

Figure 1.2: Fragments of separation logic

Note that there is additionally a table of notations at the end of the thesis.

1.1 Memory Model

1.1.1 Memory States

Let us introduce our model of memory. It captures features of programs with pointer variables that use pointer arithmetic and records, as well as data from an ordered set.

Definition

We assume a countably infinite set $\mathtt{Var}$ of first-order variables (although, obviously, for a given formula we need only a finite amount). We will range over variables with $\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}$. For further information about variables, see section 1.2.1.

We assume an infinite set $\mathtt{Loc}$ of locations, thought of as address indexes. We assume that $\mathtt{Loc} = \mathbb{N}$ as we want to model pointer arithmetic. In our abstraction, any integer is a valid address, there is no $\mathtt{nil}$ special address as our logical formalisms will all be able to simulate its presence if necessary, and the memory has an infinite amount of addresses which allows an unbounded size of the stored information. We will range over naturals with $\mathtt{m}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{n}$.

We assume a disjoint, infinite, totally ordered set $(\mathtt{Dat}, \leq)$ of data, and range over a particular datum with $\mathtt{o}$.

In order to model field selectors of a cell, we consider an infinite set $\mathtt{Lab}$ of labels, we will range over labels with $\mathtt{l}, \mathtt{next}, \mathtt{datum}$.

We will use $\mathtt{Pow}_{\mathtt{fin}}(\mathtt{I})$ to denote the set of finite subsets of $\mathtt{I}$. We use $\mathtt{Set}_1 \rightharpoonup_{\mathtt{fin}} \mathtt{Set}_2$ to denote the set of partial functions with finite domain from a subset of $\mathtt{Set}_1$ to $\mathtt{Set}_2$, and $\rightharpoonup_{\mathtt{fin+}}$ the set of the ones of finite non-empty domain.

The sets $\mathtt{Stores}$ of stores and $\mathtt{Heaps}_{\mathtt{sv}}$ of heaps are then defined as follows:

$$\begin{array}{rcl} \mathtt{Stores} & \triangleq & \mathtt{Var} \rightarrow \mathtt{Loc} \\ \mathtt{Heaps}_{\mathtt{sv}} & \triangleq & \mathtt{Loc} \rightharpoonup_{\mathtt{fin}} (\mathtt{Lab} \rightharpoonup_{\mathtt{fin+}} (\mathtt{Loc} \cup \mathtt{Dat})) \end{array}$$

We will range over a store with $\mathtt{s}$ and over a heap with $\mathtt{h}$. We call *memory state* a couple $(\mathtt{s}, \mathtt{h}) \in \mathtt{Stores} \times \mathtt{Heaps}_{\mathtt{sv}}$. A heap can be equivalently understood as a finite subset of $\mathbb{N} \times \mathtt{Lab} \times (\mathtt{Loc} \cup \mathtt{Dat})$. Given a finite set $\mathtt{X}$ of variables (for instance occurring in a given formula), we can assume that a memory state is finite by restricting the domain of the store to $\mathtt{X}$.

In a memory state, each allocated address contains a memory cell, and each cell can contain several fields. Fields of a cell and offsets for pointer arithmetic are both available in our models but are not related, so our models could be more concrete considering labels as offsets and relying on pointer arithmetic. However, for our classification of several problems, it will be useful to consider pointer arithmetic independently. A visual representation of a heap of our general models can be seen in figure 1.3, where the first row represents the addresses for pointer arithmetic, and the boxes below represent the cells, either with field selectors when allocated or with the $\emptyset$ symbol when not allocated.

Subscripts

We will use these models in three different contexts, for which we define three different subsets of $\mathtt{Heaps}_{\mathtt{sv}}$, leading to three different sets of models. We use the subscript $\mathtt{s}$ to denote the heaps which allow several selectors, and the subscript $\mathtt{v}$ to denote that heaps can contain data as well as addresses.

Handling Heaps

We write $\mathtt{Dom}(\mathtt{h})$ to denote the domain of $\mathtt{h}$ and $\mathtt{Im}(\mathtt{h})$ to denote its image. For $\mathtt{I} \subseteq \mathtt{Dom}(\mathtt{h})$, We write $\mathtt{h}_{|\mathtt{I}}$ to denote the restriction of $\mathtt{h}$ to $\mathtt{I}$.

Intuitively, in memory states, each index is thought of as an entry point on some record cell containing several fields. Cells are either not allocated, or allocated with some record stored in. In a memory state $(\mathtt{s}, \mathtt{h})$, the memory cell at index $\mathtt{i}$ is *allocated* if $\mathtt{i} \in \mathtt{Dom}(\mathtt{h})$; in this case the stored record is $\mathtt{h}(\mathtt{i}) = \{\mathtt{l}_1 \mapsto \mathtt{j}_1, \ldots, \mathtt{l}_n \mapsto \mathtt{j}_n\}$. For instance, in the figure 1.3, if we

1	2	3	4	5	6	...
next : 4 datum : o_2	$\emptyset$	l_1 : o_1 l_5 : 1 l_7 : o_3 l_8 : o_4 l_9 : 2	$\emptyset$	$\emptyset$	...	

Figure 1.3: Visual representation of a memory state

call the represented heap $\mathtt{h}$, then $\mathtt{Dom(h)} = \{1, 3\}$; also $\mathtt{h(1)} = \{\mathtt{next} \mapsto 4, \mathtt{datum} \mapsto o_2\}$ and $\mathtt{h(3)} = \{l_1 \mapsto o_1, l_5 \mapsto 1, l_7 \mapsto o_3, l_8 \mapsto o_4, l_9 \mapsto 2\}$.

A heap $\mathtt{h}$ with domain $\{i_1, \ldots, i_n\}$ is sometimes represented by the set of memory cells $\{i_1 \mapsto h(i_1), \ldots, i_n \mapsto h(i_n)\}$.

Two heaps h_1, h_2 are said to be *disjoint*, noted $h_1 \perp h_2$, if their domains are disjoint; when this holds, we write $h_1 \star h_2$ to denote the disjoint union $h_1 \uplus h_2$.

Sizes

The size of the store $\mathtt{s}$ with respect to a finite set of variables $\mathtt{X} \subseteq \mathtt{Var}$, written $\mathtt{size}_{\mathtt{X}}(\mathtt{s})$, is defined as $|\mathtt{X}| \times \max(1 + \log(1 + s(x)) : s(x) \in \mathbb{N}, x \in \mathtt{X})$.

Similarly, the size of the heap $\mathtt{h}$ with respect to a finite set of labels $\mathtt{L} \subseteq \mathtt{Lab}$, which we will write $\mathtt{size}_{\mathtt{L}}(\mathtt{h})$, is defined as $|\mathtt{Dom(h)}| \times |\mathtt{L}| \times \max(1 + \log(1 + h(i)(l)) : i \in \mathtt{Dom(h)}, h(i)(l)$ is defined and $h(i)(l) \in \mathbb{N})$.

The size of the memory state (s, h) with respect to $\mathtt{X}$ and $\mathtt{Y}$, written $\mathtt{size}_{\mathtt{X,L}}((s, h))$, is $\mathtt{size}_{\mathtt{X}}(s) + \mathtt{size}_{\mathtt{L}}(h)$.

1.1.2 Memory Shapes

We define memory shapes as the abstraction of a memory heap forgetting the whole data component of all cells, while retaining the graphical aspect. A *memory shape* is a pair $(s, h) \in \mathtt{Stores} \times \mathtt{Heaps}_s$ where:

$$\mathtt{Heaps}_s \triangleq \mathtt{Loc} \rightharpoonup_{\mathtt{fin}} (\mathtt{Lab} \rightharpoonup_{\mathtt{fin+}} \mathtt{Loc})$$

This model can be seen as a finite directed graph whose edges are labelled, so that two edges originating from the same vertex always have distinct labels.

1.1.3 Simple Memory States

They represent the memory state of programs manipulating singly-linked lists and data. We define a *simple memory state* as a pair $(s, h) \in \mathtt{Stores} \times \mathtt{Heaps}_v$ where $\mathtt{Heaps}_v$ is the set of the heaps in which all the allocated memory cells have exactly two labels, one called $\mathtt{next}$ and always containing a location, the other called $\mathtt{datum}$ and always containing a datum. It can be equivalently defined as:

$$\mathtt{Heaps_v} \triangleq \mathtt{Loc} \rightharpoonup_{\mathtt{fin}} (\mathtt{Loc} \times \mathtt{Dat})$$

This model can also be seen as the graph of a unary function with finite domain, in which each edge is labelled with a datum.

We write $\mathtt{fst}$ and $\mathtt{snd}$ to denote the first and second projection on a product set. As a consequence, $\mathtt{fst(h(i))}$ is the location in the memory cell $\mathtt{h(i)}$ whereas $\mathtt{snd(h(i))}$ is the datum. We can equivalently write $\mathtt{h(i)(next)}$ for $\mathtt{fst(h(i))}$ and $\mathtt{h(i)(datum)}$ for $\mathtt{snd(h(i))}$.

The set $\mathtt{Dat}$ can be instantiated in various ways. As an example, programs manipulating ordered lists of naturals can be modeled choosing $\mathtt{Dat} = \mathbb{N}$ with the standard order. In order to ensure $\mathtt{Dat} \cap \mathtt{Loc} = \emptyset$, we can simply choose $\mathtt{Dat} = \mathbb{N}' = \{0', 1', 2', \ldots\}$ with $\mathtt{i}' \leq \mathtt{j}'$ iff $\mathtt{i} \leq \mathtt{j}$ in $(\mathbb{N}, \leq)$. The same holds for lists of reals, lists of integers, and so on.

Also, $\mathtt{Dat}$ could be thought of as the state of a lock at the current node, that is the identifier of the thread holding the node (or some constant for an available lock). Here, the ordering on data is not relevant, but the equality between data is. For such a model, one may want to express, for instance, that every thread holds the locks of at most two nodes of a list, and that these nodes are necessarily consecutive.

1.1.4 Simple Memory Shapes

They represent the shapes of the memory for programs manipulating singly-linked lists. They are equivalent to a model in which all the allocated memory cells have only one label, $\mathtt{next}$. A *simple memory shape* is a pair $(\mathtt{s}, \mathtt{h}) \in \mathtt{Stores} \times \mathtt{Heaps}$ where:

$$\mathtt{Heaps} \triangleq \mathtt{Loc} \rightharpoonup_{\mathtt{fin}} \mathtt{Loc}$$

This model can be seen as the graph of a unary function with finite domain.

We will write $\mathtt{Shape}(\cdot)$ for the obvious map from heaps of simple memory states to heaps of simple memory shapes – with the domain of $\mathtt{Shape(h)}$ equal to the domain of $\mathtt{h}$

$$\mathtt{Shape(h)} \triangleq \begin{array}{ccc} \mathtt{Loc} & \rightharpoonup & \mathtt{Loc} \\ \mathtt{i} & \mapsto & \mathtt{fst(h(i))} \end{array}$$

1.2 First and Second-Order logic on Simple Memory Shapes

1.2.1 Conventions on Variables

We have already defined the countably infinite set $\mathtt{Var}$.

Program Variables

Variables can be interpreted as both variables from the programs or logical variables quantifying over locations. The main difference between these two types of variables is that program variables are not quantified in formulas. We safely identify them and will use $\mathtt{w}$ to emphasize that a variable should be understood as a program variable. The set of program variables will be called $\mathtt{Progvar}$ and is included in $\mathtt{Var}$.

Special Variables

In this paragraph we define a set of variables and functions providing fresh variables from this set, which will be very useful in several proofs, as having fresh variables will then make things much simpler. We define the special variables $\mathtt{Specialvar}$ as an infinite subset of $\mathtt{Var}$ such that $\mathtt{Var} \setminus \mathtt{Specialvar}$ is also infinite. Unless otherwise stated, a variable should be understood as belonging to $\mathtt{Var} \setminus \mathtt{Specialvar}$. In the remainder, we will assume two fixed injections $(\mathtt{x}, \mathtt{i}) \in \mathtt{Var} \setminus \mathtt{Specialvar} \times \mathbb{N} \mapsto \langle \mathtt{x}, \mathtt{i} \rangle \in \mathtt{Specialvar}$, and $\mathtt{i} \in \mathbb{N} \mapsto \langle \mathtt{i} \rangle \in \mathtt{Specialvar}$ such that for all $\mathtt{x}$, $\mathtt{i}$ and $\mathtt{j}$, $\langle \mathtt{x}, \mathtt{i} \rangle \neq \langle \mathtt{j} \rangle$.

Data Variables

We assume a set $\mathtt{Datvar}$ of data variables, ranged over with $\mathtt{v}$. A valuation interpreting data variables is a function $\mathtt{e} : \mathtt{Datvar} \to \mathtt{Dat}$. In general, the letter $\mathtt{e}$ will be used to describe an environment generated by quantifications. Concerning data variables, they will never be free variables in the formulas which are instances of the problems we will study.

Second-Order Variables

In order to define second-order formulas, we consider a family $\mathtt{Secvar} = (\mathtt{Secvar}_\mathtt{i})_{\mathtt{i} \geq 0}$ of second-order variables, denoted by P, Q that will be interpreted as finite relations over $\mathtt{Loc}$. Each variable in $\mathtt{Secvar}_\mathtt{i}$ is interpreted as an $\mathtt{i}$-ary relation. A *second-order environment* $\mathtt{E}$ is an interpretation of the second-order variables such that for every $\mathsf{P} \in \mathtt{Secvar}_\mathtt{i}$, $\mathtt{E}(\mathsf{P})$ is a finite subset of $\mathtt{Loc}^\mathtt{i}$. Since second-order variables quantify over finite relations, the version of second-order logics we shall consider is usually called *weak*. We will sometimes call environment a second-order environment, when the context is not ambiguous.

The value of a second-order variable, a relation on integers, will be represented with the letter $\mathtt{R}$. If the variable specifically belongs to $\mathtt{Secvar}_1$, it can then be represented by $\mathtt{I}$, $\mathtt{J}$ or $\mathtt{K}$, which will more generally be used to represent sets of integers.

1.2.2 Second-Order Logic

Formulas

We range over formulas describing memory heaps with f or g.

Formulas of (weak) second-order logic $\mathtt{SO}$ are defined by the grammar below:

$$\mathsf{f} := \neg \mathsf{f} \mid \mathsf{f} \wedge \mathsf{f} \mid \exists \mathsf{x}.\, \mathsf{f} \mid \mathsf{x} \hookrightarrow \mathsf{y} \mid \mathsf{x} = \mathsf{y} \mid \exists \mathsf{P}.\, \mathsf{f} \mid \mathsf{Q}(\mathsf{x}_1, \ldots, \mathsf{x}_n)$$

where P, Q are second-order variables and $\mathsf{Q} \in \mathtt{Secvar}_\mathtt{n}$. We write $\mathtt{MSO}$ [resp. $\mathtt{DSO}$] to denote the restriction of $\mathtt{SO}$ to second-order variables in $\mathtt{Secvar}_1$ [resp. $\mathtt{Secvar}_2$]. A *sentence* is defined as a formula with no free occurrence of second-order variables. As free first-order variables are considered as program variables, this is why we can define a formula with free first-order variables as a sentence.

We define the first-order fragment $\mathtt{FO}$, as the restriction of $\mathtt{SO}$ to the formulas with no occurrence of second-order variables.

Let $\mathtt{fct}$ be a unary function. Then $\mathtt{fct}[\mathtt{i} \mapsto \mathtt{j}]$ has as domain $\mathtt{Dom}(\mathtt{fct}) \cup \{\mathtt{i}\}$, and is defined by $\mathtt{fct}[\mathtt{i} \mapsto \mathtt{j}](\mathtt{i}) = \mathtt{j}$ and for all $\mathtt{i}' \in \mathtt{Dom}(\mathtt{fct}) \setminus \{\mathtt{i}\}$, $\mathtt{fct}[\mathtt{i} \mapsto \mathtt{j}](\mathtt{i}') = \mathtt{fct}(\mathtt{i}')$.

Satisfaction Relation

The satisfaction relation for $\mathtt{SO}$ is defined below with an environment $\mathtt{E}$ as argument (below $\mathsf{P} \in \mathtt{Secvar}_n$).

$$
\begin{array}{lll}
(s,h),E \models_{SO} \exists P.\, f & \text{iff} & \text{there is a finite subset } R \text{ of } \mathtt{Loc}^n, \\
 & & \text{such that } (s,h),E[P \mapsto R] \models_{SO} f \\
(s,h),E \models_{SO} P(x_1,\ldots,x_n) & & \\
 & \text{iff} & (s(x_1),\ldots,s(x_n)) \in E(P) \\
(s,h),E \models_{SO} \neg f & \text{iff} & \text{not } (s,h),E \models_{SO} f \\
(s,h),E \models_{SO} f \wedge g & \text{iff} & (s,h),E \models_{SO} f \text{ and } (s,h),E \models_{SO} g \\
(s,h),E \models_{SO} \exists x.\, f & \text{iff} & \text{there is } l \in \mathtt{Loc} \text{ such that } (s[x \mapsto l],h),E \models_{SO} f \\
(s,h),E \models_{SO} x \hookrightarrow y & \text{iff} & h(s(x)) = s(y) \\
(s,h),E \models_{SO} x = y & \text{iff} & s(x) = s(y)
\end{array}
$$

When f is a sentence, we write $(s,h) \models_{SO} f$ to denote $(s,h),E \models_{SO} f$ for any environment $\mathtt{E}$ since $\mathtt{E}$ has no influence on the satisfaction of f. This particularly applies to $\mathtt{FO}$ formulas.

Shorthands

We will write $P \subseteq Q$ for $\forall x.P(x) \Rightarrow Q(x)$, as well as $P \subsetneq Q$ for $P \subseteq Q \wedge \exists x.P(x) \wedge \neg Q(x)$, and use all set operators $P \cap Q$, $P \cup Q$,, etc. defined in a standard way. We will also use the composition of predicates: $xPQy$ for $\exists z.xPz \wedge zQy$. We will make use of standard notations for the derived connectives $\forall, \vee, \Rightarrow, \Leftrightarrow$. Let us also mention that the equality $x = y$ could be encoded by $\forall P.(P(x) \Leftrightarrow P(y))$, obtained by the principle of identity of indiscernibles.

1.2.3 Conventions on Formulas and Languages

Fragments

Let $\mathtt{Frag}$ and $\mathtt{Frag}'$ be two fragments of logics defined on the same set of memory models. We say that $\mathtt{Frag}'$ is at least as expressive as $\mathtt{Frag}$ (written $\mathtt{Frag} \sqsubseteq \mathtt{Frag}'$) whenever for every sentence $f \in \mathtt{Frag}$, there is $f' \in \mathtt{Frag}'$ such that for every memory state (s,h), we have $(s,h) \models f$ iff $(s,h) \models f'$. We write $\mathtt{Frag} \equiv \mathtt{Frag}'$ if $\mathtt{Frag} \sqsubseteq \mathtt{Frag}'$ and $\mathtt{Frag}' \sqsubseteq \mathtt{Frag}$. A *translation* from $\mathtt{Frag}$ to $\mathtt{Frag}'$ is a computable function $\mathtt{tr} : \mathtt{Frag} \to \mathtt{Frag}'$ such that for every sentence $f \in \mathtt{Frag}$, for every memory shape (s,h), we have $(s,h) \models f$ iff $(s,h) \models \mathtt{tr}(f)$.

Free Variables

We write $\mathtt{Freevar}(f)$ to denote the set of free variables occurring in the formula f. The proof of lemma 1.2.3.1 is by an easy verification.

Lemma 1.2.3.1. For all simple memory shape (s, h), $\mathtt{SO}$ formula g, environment $\mathtt{E}$ and store s', if $s_{|\mathtt{Freevar}(g)} = s'_{|\mathtt{Freevar}(g)}$, then $(s,h),E \models_{SO} g$ iff $(s',h),E \models_{SO} g$.

Substitutions

In the latter, we may use the notation $f[g \leftarrow g']$ for the formula f in which the subformula or the variable g' replaces each occurrence of g.

Another useful substitution is $\mathtt{s[i \leftarrow i']}$ [resp. $\mathtt{E[i \leftarrow i']}$], which denotes the store obtained from $\mathtt{s}$ [resp. the environment obtained from $\mathtt{E}$] by replacing every occurrence of $\mathtt{i}$ by $\mathtt{i'}$ in the range of these functions.

Let us define formally $\mathtt{s'} = \mathtt{s[i \leftarrow i']}$. For any $\mathtt{x} \in \mathtt{Var}$, if $\mathtt{s(x)} = \mathtt{i}$ then $\mathtt{s'(x)} = \mathtt{i'}$ otherwise $\mathtt{s'(x)} = \mathtt{s(x)}$.

Let us define formally $\mathtt{E[i \leftarrow i']}$. Let $\mathsf{P} \in \mathtt{Secvar}$. Let $(\mathtt{i}_1, \ldots, \mathtt{i}_n) \in \mathtt{Loc}^n$; let $\mathtt{i}'_k = \mathtt{i}_k$ when $\mathtt{i}_k \neq \mathtt{i}$ and $\mathtt{i}'_k = \mathtt{i'}$ otherwise. Then $(\mathtt{i}'_1, \ldots, \mathtt{i}'_n) \in \mathtt{E[i \leftarrow i'](P)}$ iff $(\mathtt{i}_1, \ldots, \mathtt{i}_n) \in \mathtt{E(P)}$.

Lemma 1.2.3.2. Let $(\mathtt{s}, \mathtt{h})$ be a simple memory shape, $\mathtt{E}$ be an environment, and g be a formula in $\mathtt{DSO}$. Let $\mathtt{i}, \mathtt{i'}$ be locations such that

- $\mathtt{i} \notin \mathtt{Dom(h)} \cup \mathtt{Im(h)}$.
- $\mathtt{i'} \notin \mathtt{Dom(h)} \cup \mathtt{Im(h)} \cup \{\mathtt{s(x)} : \mathtt{x} \in \mathtt{Freevar(g)}\}$.
- $\mathtt{i'}$ is not in the finite graph of $\mathtt{E(P)}$ for any second-order variable P occurring in g.

Then $(\mathtt{s[i \leftarrow i']}, \mathtt{h}), \mathtt{E[i \leftarrow i']} \models_{SO} \mathsf{g}$ iff $(\mathtt{s}, \mathtt{h}), \mathtt{E} \models_{SO} \mathsf{g}$.

Proof. The proof is by a simple induction on the subformulas of g. Let $\mathsf{g'}$ be a subformula of g. Assume that the lemma holds for any strict subformula of $\mathsf{g'}$. We must prove that the lemma holds for $\mathsf{g'}$. The inductive cases, when the outermost connective of $\mathsf{g'}$ is boolean or a quantification, are obvious. Let us study the base case $\mathsf{g'} = \mathsf{x} \hookrightarrow \mathsf{y}$. The other base cases are simpler.

By the semantics, $(\mathtt{s[i \leftarrow i']}, \mathtt{h}), \mathtt{E[i \leftarrow i']} \models_{SO} \mathsf{x} \hookrightarrow \mathsf{y}$ iff $\mathtt{h(s[i \leftarrow i'](x))} = \mathtt{s[i \leftarrow i'](y)}$. As $\mathtt{i} \notin \mathtt{Dom(h)}$ and $\mathtt{i'} \notin \mathtt{Dom(h)}$, we have $\mathtt{s[i \leftarrow i'](x)} \in \mathtt{Dom(h)}$ iff $\mathtt{s(x)} \in \mathtt{Dom(h)}$.

- If $\mathtt{s(x)} \notin \mathtt{Dom(h)}$ then $\mathtt{s[i \leftarrow i'](x)} \notin \mathtt{Dom(h)}$ and none of $(\mathtt{s[i \leftarrow i']}, \mathtt{h}), \mathtt{E[i \leftarrow i']}$ and $(\mathtt{s}, \mathtt{h}), \mathtt{E}$ is a model of $\mathsf{g'}$.

- If $\mathtt{s(x)} \in \mathtt{Dom(h)}$, then, as $\mathtt{i} \notin \mathtt{Dom(h)}$, $\mathtt{s[i \leftarrow i'](x)} = \mathtt{s(x)}$ and $\mathtt{h(s[i \leftarrow i'](x))} = \mathtt{h(s(x))}$.

  * If $(\mathtt{s}, \mathtt{h}), \mathtt{E}$ is a model of $\mathsf{g'}$ then $\mathtt{h(s(x))} = \mathtt{s(y)}$, and as $\mathtt{i} \notin \mathtt{Im(h)}$, we have $\mathtt{s[i \leftarrow i'](y)} = \mathtt{s(y)}$.

  * If $(\mathtt{s}, \mathtt{h}), \mathtt{E}$ is not a model of $\mathsf{g'}$ then $\mathtt{h(s(x))} \neq \mathtt{s(y)}$. If $\mathtt{s(y)} \neq \mathtt{i}$ then $\mathtt{s[i \leftarrow i'](y)} = \mathtt{s(y)}$, so $\mathtt{h(s[i \leftarrow i'](x))} \neq \mathtt{s[i \leftarrow i'](y)}$. If $\mathtt{s(y)} = \mathtt{i}$, then $\mathtt{s[i \leftarrow i'](y)} = \mathtt{i'}$, so since $\mathtt{i'} \notin \mathtt{Im(h)}$ we have $\mathtt{h(s[i \leftarrow i'](x))} \neq \mathtt{s[i \leftarrow i'](y)}$. In both cases $(\mathtt{s[i \leftarrow i']}, \mathtt{h}), \mathtt{E[i \leftarrow i']}$ is not a model of $\mathsf{g'}$.

□

Sizes

The size of the formula f, written $|\, \mathsf{f} \,|$, is the length of the string f for some reasonably succinct encoding of variables and integers with a binary representation. We will use the map $|\cdot|$ for other syntactic objects such as formulas of our temporal logic and formulas of separation logic.

Atomic formulas ($x, x' \in \mathtt{Var},\ i \in \mathbb{N},\ l \in \mathtt{Lab},\ v \in \mathtt{Datvar}$)

atom ::=	$x = x' \mid x + i \hookrightarrow^{l} x'$	(atomic formulas)
	$\mid \mathsf{val}(x) \leq v \mid \mathsf{val}(x) \geq v$	(ordered data)

State formulas

f ::=	atom	
	$\mid f \ast g \mid f \mathrel{-\!\!*} g \mid \mathsf{emp}$	(spatial fragment)
	$\mid f \wedge g \mid \neg f$	(classical fragment)
	$\mid \exists x.f \mid \exists v.f$	(first-order)

Satisfaction

$(s,h), e \models_{SL}$	$\exists x.\ f$	iff there is $i \in \mathtt{Loc}$ such that $(s[x \mapsto i], h), e \models_{SL} f$
$(s,h), e \models_{SL}$	$\exists v.\ f$	iff there is $o \in \mathtt{Dat}$ such that $(s,h), e[v \mapsto o] \models_{SL} f$
$(s,h), e \models_{SL}$	$x = x'$	iff $s(x) = s(x')$
$(s,h), e \models_{SL}$	$x + i \hookrightarrow^{l} x'$	iff $h(s(x) + i)(l) = s(x')$
$(s,h), e \models_{SL}$	$\mathsf{val}(x) \leq v$	iff $h(s(x))(\mathtt{datum}) \leq e(v)$
$(s,h), e \models_{SL}$	$\mathsf{val}(x) \geq v$	iff $h(s(x))(\mathtt{datum}) \geq e(v)$
$(s,h), e \models_{SL}$	emp	iff $\mathtt{Dom}(h) = \emptyset$
$(s,h), e \models_{SL}$	$f_1 \ast f_2$	iff $\exists\, h_1, h_2$ s.t. $h = h_1 \ast h_2,\ (s,h_1) \models_{SL} f_1$ and $(s,h_2) \models_{SL} f_2$
$(s,h), e \models_{SL}$	$f_1 \mathrel{-\!\!*} f_2$	iff for all h', if $h \perp h'$ and $(s,h') \models_{SL} f_1$ then $(s, h \ast h') \models_{SL} f_2$
$(s,h), e \models_{SL}$	$f_1 \wedge f_2$	iff $(s,h) \models_{SL} f_1$ and $(s,h) \models_{SL} f_2$
$(s,h), e \models_{SL}$	$\neg f_1$	iff not $(s,h) \models_{SL} f_1$

Figure 1.4: The syntax and semantics of SL_{sv} with pointer arithmetic and records

1.3 Separation Logic

1.3.1 Definition

We now introduce the separation logic (SL_{sv}). As for the heaps, we will use the subscript s to denote fragments of SL_{sv} which deal with several selectors, and v to denote fragments which deal with data as well as pointers. The syntax of the logic is given in figure 1.4. We range over formulas of separation logic with f, g.

In short, separation logic is about reasoning on disjoint heaps. The models of this logic are the memory states defined above.

Semantics

A formula $f \ast g$ with the *separating conjunction* states that f holds on some portion of the memory heap and g holds on a disjoint portion. A formula $f \mathrel{-\!\!*} g$ with the *separating implication* (usually called the *magic wand*) states that the current heap, when extended with any disjoint heap verifying f, will verify g. Consequently, $\mathrel{-\!\!*}$ is a universal modality whereas $\ast$ has an existential flavour. In a visual representation, one can state this semantics as in figure 1.5.

Boolean operators are understood as such. Derivable connectives $f \vee g$ and $\neg f$ are defined through their straightforward abbreviations of the included boolean operators.

● $\models_{SL}$ $f \ast g$ when there are ◖ and ◗
such that ● = ◐, as well as ◖ $\models_{SL}$ f and ◗ $\models$ g.

◗ $\models_{SL}$ $f \mathrel{-\!\!\ast} g$ when any ◖ such that ◖ $\models_{SL}$ f is also such that ◐ $\models$ g.

Figure 1.5: Visual representation of the semantics of separation operators

---

Formulas atom are *atomic formulas*. The formula $x + i \hookrightarrow^{l} x'$ states that the value of the field l of the record stored at the address pointed by x with offset i is equal to the value of the expression x. If the offset of a pointer is 0, we write $x \hookrightarrow^{l} x'$; if additionally $l = \mathtt{next}$, we can simply write $x \hookrightarrow x'$. Finally, $x \mapsto (y, v)$ will be used for $x \mapsto y \wedge \mathsf{val}(x) = v$. The formula $x = x'$ states the equality between the values of the two variables, and emp means that the current heap has no memory cell allocated.

The semantics of formulas are formally defined by the satisfaction relation $\models_{SL}$ in figure 1.4. We can note that our level of granularity implies that a record cell cannot be decomposed in disjoint parts by separation operators.

Validity and Satisfiability

We will not study formulas with free data variables – except as subformulas of studied formulas. We can write $(s, h) \models_{SL} f$ instead of $(s, h), e \models_{SL} f$ when f has no free data variable. A formula f is valid iff for every memory state (s, h), we have $(s, h) \models_{SL} f$ (written $\models_{SL} f$). Satisfiability is defined dually: f is satisfiable iff there is a memory state (s, h), such that $(s, h) \models_{SL} f$.

Remarks on the Wand

We also introduce a slight variant of the dual connective for the magic wand, also called the *septraction*: $f \mathrel{-\!\!\circledast} g$ is defined as the formula $\neg((f) \mathrel{-\!\!\ast} (\neg(g)))$. it is easy to check that $(s, h) \models_{SL} f_1 \mathrel{-\!\!\circledast} f_2$ iff there is $h' \perp h$ such that $(s, h') \models_{SL} f_1$ and $(s, h \ast h') \models_{SL} f_2$. Septraction is an existential version of magic wand. Hence, the septraction operator is quite natural since it states the existence of a disjoint heap satisfying a formula and for which the addition to the original heap satisfies another formula.

The connective $\mathrel{-\!\!\ast}$ is the *adjunct* of $\ast$, meaning that $(f_1 \ast f_2) \Rightarrow f_3$ is valid iff $f_1 \Rightarrow (f_2 \mathrel{-\!\!\ast} f_3)$ is valid. Still, observe that there is no obvious way to define $\ast$ and $\mathrel{-\!\!\ast}$ from each other since typically the formula $((f_1 \ast f_2) \Rightarrow f_3) \Leftrightarrow (f_1 \Rightarrow (f_2 \mathrel{-\!\!\ast} f_3))$ is not valid. This shall be strengthened in the sequel by establishing that SL without wand is decidable whereas SL without separating conjunction is not.

Shorthands

We use the notation $x \hookrightarrow \square$ for $\exists y. x \hookrightarrow y$. The notation $\square$ will actually have a wider use, always in the meaning of an existential quantification over the variable, label or integer that should be in the place of the square. In the case of figures like figure 2.4, a $\square$ symbol represents a location that is not represented by any other $\square$ symbol, variable (like x in figure 2.4), or integer (like i_1 in figure 2.4)..

We will use equality over vectors $(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$.

If the considered fragment of the language contains first order quantifications and is defined on single a model with single selector ($\mathtt{Heaps}_v$ or $\mathtt{Heaps}$) then emp, which means that the domain of the heap is empty, can be defined as $\mathsf{emp} \triangleq \neg\exists x.\ \neg\exists y.\ x \hookrightarrow y$. In this case it can be omitted from the syntax.

The version of separation logic we have introduced does not contain null, the usual constant for exceptions, interpreted by $\mathtt{nil}$ such that any h is undefined for the value $\mathtt{nil}$. Any formula f possibly with the constant null can be translated into a formula f' of SL such that f is satisfiable iff f' is satisfiable. Indeed, if $g_0 = (\neg\exists z.\ \mathsf{null} \hookrightarrow z)$, then f' can be defined as $\exists\ \mathsf{null}.\ g_0 \wedge f''$ with f'' being f completed with g_0 in each left member of a subformula with the magic wand as outermost connective. null is understood here as a distinguished variable. In the sequel, we will not use the constant null.

1.3.2 Fragments of Separation Logic on Memory Shapes

State Formulas

We define the set of *state formulas* $\mathtt{SL}_s$ with the grammar below. It is a separation logic on memory shapes. It has no quantification as it will be used in chapter 4, where it will be mixed with LTL. In the remainder, we focus on several specific fragments of this separation logic.

$$f := \neg f \mid f \wedge f \mid x + i \hookrightarrow^{l} y \mid x = y \mid \mathsf{emp} \mid f * f \mid f -\!\!* f$$

Note that the size of the information held in a memory cell is neither fixed, nor bounded.

Fragments

We say that a formula is in the *record fragment* ($\mathtt{SL}_s^{RF}$) if all its subformulas of the form $x+i\hookrightarrow^{l}x'$ use $i = 0$. In other words, pointer arithmetic is removed, but all other features are still present, in particular memory cells have multiple selectors through their labels.

We say that a formula is in the *classical fragment* ($\mathtt{SL}_s^{CL}$) if it does not contain any of the connectives $*$ and $-\!\!*$.

The *list fragment* $\mathtt{SL}_s^{LF}$ is part of the classical fragment in which all subformulas $x + i\hookrightarrow^{l}x'$ use $i = 0$ and $l = \mathtt{next}$. In other words, memory cells have a single selector, which is similar to only being able to describe simple memory shapes.

Clearly, the classical and record fragments are incomparable, while the list fragment is included in both of them.

Finally, $\mathtt{SL}_s^{*}$ is $\mathtt{SL}_s$ with no $-\!\!*$ connective.

1.3.3 A Separation Logic for Simple Memory States

Definition

We now define the assertion language $\mathtt{SL}_v$. Formulas of $\mathtt{SL}_v$ are defined by the grammar below. They allow to describe lists with ordered data.

$$f ::= \neg f \mid f \wedge f \mid \exists x.f \mid \exists v.f \mid x \hookrightarrow y \mid \mathsf{val}(x) \leq v \mid \mathsf{val}(x) \geq v \mid x = y \mid f * f$$

Note that due to the memory model $\mathtt{Heaps}_v$, the natural semantics of $\mathsf{val}(x) \leq v$ implies in particular $\exists z.x \hookrightarrow z$.

Comparison Predicates

In the chapter dealing with $\mathtt{SL_v}$, we will define several fragments of it, in particular depending on how restricted are the comparisons that we choose to allow. The comparisons will be restricted to a few predicates, not allowing quantification over data variables outside of these predicates.

The predicate $\mathsf{val}(\mathsf{x}) \leq \mathsf{val}(\mathsf{y}) \triangleq \exists \mathsf{v}.(\mathsf{val}(\mathsf{x}) \leq \mathsf{v} \wedge \mathsf{val}(\mathsf{y}) \geq \mathsf{v})$ asserts that the value stored at the location x is smaller than the one stored at y. We call this predicate *long-distance comparison*. We moreover say that a long-distance comparison is *guarded* if it is $\mathsf{val}(\mathsf{w}) \leq \mathsf{val}(\mathsf{y})$ or $\mathsf{val}(\mathsf{x}) \geq \mathsf{val}(\mathsf{w})$ where w is a program variable, hence a free variable.

We can now define the predicates $\mathsf{x} \hookrightarrow_{\leq} \mathsf{y} \triangleq \mathsf{x} \hookrightarrow \mathsf{y} \wedge \mathsf{val}(\mathsf{x}) \leq \mathsf{val}(\mathsf{y})$ and $\mathsf{x} \hookrightarrow_{\geq} \mathsf{y}$ accordingly; we call these predicates *short-distance comparisons*.

1.3.4 Separation Logic on Simple Memory Shapes

Formulas of first-order separation logic with one selector $\mathtt{SL}$ are defined by the grammar below:

$$\mathsf{f} := \neg \mathsf{f} \mid \mathsf{f} \wedge \mathsf{f} \mid \exists \mathsf{x}.\mathsf{f} \mid \mathsf{x} \hookrightarrow \mathsf{y} \mid \mathsf{x} = \mathsf{y} \mid \mathsf{f} \ast \mathsf{f} \mid \mathsf{f} \mathrel{-\!\!\ast} \mathsf{f}$$

We write $\mathtt{SL}^{\ast}$ [resp. $\mathtt{SL}^{\mathrel{-\!\!\ast}}$] to denote the restriction of $\mathtt{SL}$ without the magic wand [resp. without the separating conjunction].

1.4 Simple Predicates in Separation Logic

1.4.1 Allocated Memory Cells

Let us illustrate the expressive power on simple examples. The formula $\neg\mathsf{emp} \ast \neg\mathsf{emp}$ means that at least two memory cells are allocated. The formula $\mathsf{x} \mapsto^{\mathsf{l}} \mathsf{x}'$, defined as $\neg(\neg\mathsf{emp} \ast \neg\mathsf{emp}) \wedge \mathsf{x} \hookrightarrow^{\mathsf{l}} \mathsf{x}'$, is the local version of $\mathsf{x} \hookrightarrow^{\mathsf{l}} \mathsf{x}'$: $(\mathtt{s}, \mathtt{h}) \models_{\mathtt{SL}} \mathsf{x} \mapsto^{\mathsf{l}} \mathsf{x}'$ iff $\mathtt{Dom}(\mathtt{h}) = \{\mathtt{s}(\mathsf{x})\}$ and $\mathtt{h}(\mathtt{s}(\mathsf{x}))(\mathsf{l}) = \mathtt{s}(\mathsf{x}')$. The formula $(\mathsf{x} \hookrightarrow^{\mathsf{l}} \mathsf{x}) \mathrel{-\!\!\ast} \perp$ is satisfied by $(\mathtt{s}_0, \mathtt{h}_0)$ whenever there is no heap $\mathtt{h}_1$ with $\mathtt{h}_1 \perp \mathtt{h}_0$ such that the variable x is already allocated in the heap $\mathtt{h}_0$. We will call this formula $\mathsf{alloc}(\mathsf{x})$. If the magic wand is not part of the considered logical language, then $\mathsf{alloc}(\mathsf{x})$ can be defined as $\exists \mathsf{y}.\ \mathsf{x} \hookrightarrow \mathsf{y}$. Also, one can specify that the domain of the heap is restricted to the value of x and maps it to that of y: $\mathsf{x} \mapsto \mathsf{y} \triangleq \mathsf{x} \hookrightarrow \mathsf{y} \wedge \neg \exists \mathsf{y}.\ (\mathsf{y} \neq \mathsf{x} \wedge \mathsf{alloc}(\mathsf{y}))$. This last predicate can also be defined as $\mathsf{precisely}(\mathsf{x} \hookrightarrow \mathsf{y})$, where $\mathsf{precisely}(\mathsf{f})$ denotes $\mathsf{f} \wedge \neg(\mathsf{f} \ast \exists \mathsf{x}, \mathsf{y}.\mathsf{x} \hookrightarrow \mathsf{y})$.

1.4.2 Predecessors and Arithmetical Constraints

A *predecessor* of the location $\mathtt{i}$ in the simple memory state $(\mathtt{s}, \mathtt{h})$ is a location $\mathtt{i}'$ such that $\mathtt{h}(\mathtt{i}') = \mathtt{i}$. A predecessor of the variable x is a predecessor of $\mathtt{s}(\mathsf{x})$. Given a memory state $(\mathtt{s}, \mathtt{h})$ and a location $\mathtt{i}$ we write $\tilde{\sharp}\mathtt{i}$ to denote the cardinal of the set $\{\mathtt{i}' \in \mathtt{Loc} : \mathtt{h}(\mathtt{i}') = \mathtt{i}\}$. We call $\tilde{\sharp}\mathtt{i}$ the *number of predecessors* of the location $\mathtt{i}$ in $(\mathtt{s}, \mathtt{h})$).

There are formulas in $\mathtt{SL}^{\ast}$, namely $\sharp \mathsf{x} \geq \mathtt{n}$ and $\sharp \mathsf{x} = \mathtt{n}$, such that $\sharp \mathsf{x} \geq \mathtt{n}$ [resp. $\sharp \mathsf{x} = \mathtt{n}$] holds true exactly in memory states such that x has at least $\mathtt{n}$ predecessors [resp. exactly $\mathtt{n}$

predecessors]. For instance, $\sharp x \geq n$ can be defined in the following ways:

$$\overbrace{(\exists y.\ y \hookrightarrow x) * \ldots * (\exists y.\ y \hookrightarrow x)}^{n \text{ times}} * \top \text{ or } \exists x_1, \ldots, x_n.\ \bigwedge_{i \neq j} x_i \neq x_j \wedge \bigwedge_{i=1}^{n} x_i \hookrightarrow x$$

It is worth noting that the first formula has a unique additional variable y but n occurrences of $*$ whereas the second formula has no separating connectives but n additional variables.

Observe that SL does not contain explicitly arithmetical constraints as in [66, 75, 25]. However, in section 2.2 we show how to compare the number of predecessors of two distinct locations. Similar developments can be performed to compare the lengths of different lists but this will come as a corollary of the equivalence between SL and SO.

1.4.3 Reachability and List Predicates

Reachability in a graph is a standard property that can be expressed in monadic second-order logic. In separation logic, often a built-in predicate for lists is added, sometimes noted $\mathsf{ls}(x, y)$. Adapting some technique used in the spatial logic for graphs [42], we show below how this very predicate can be expressed in SL* as well as the reachability predicate $x \hookrightarrow^{*} y$.

Definitions

A location i' is a *descendant* [resp. *strict descendant*] of i if there is $n \geq 0$ [resp. $n > 0$] such that $h^n(i) = i'$ ($h^n(i)$ is not always defined).

A *cyclic list* in a memory state (s, h) is a non-empty finite sequence $k_1, \ldots, k_n$ ($n \geq 1$) of locations such that $h(k_n) = k_1$ and for every $i \in \{1, \ldots, n-1\}$, $h(k_i) = k_{i+1}$. A memory state (s, h) is a *list segment* between x and y if there are locations $k_1, \ldots, k_n$ ($n \geq 2$) such that $s(x) = k_1$, $s(y) = k_n$, $k_1 \neq k_n$, $\mathrm{Dom}(h) = \{k_1, \ldots, k_{n-1}\}$, and for every $i \in \{1, \ldots, n-1\}$, $h(k_i) = k_{i+1}$.

Formulas

The semantics of the formula below is given in lemma 1.4.3.1, whose proof is given at the end of this section.

$$\begin{aligned} x \hookrightarrow^{\circlearrowleft +} y \ \triangleq\ & \sharp x = 0 \wedge \mathsf{alloc}(x) \\ & \wedge \sharp y = 1 \wedge \neg \mathsf{alloc}(y) \\ & \wedge \forall z.\ z \neq y \Rightarrow (\sharp z = 1 \Rightarrow \mathsf{alloc}(z)) \\ & \wedge \forall z.\ \sharp z \leq 1 \end{aligned}$$

Lemma 1.4.3.1. Let (s, h) be a simple memory shape. $(s, h) \models_{SL} x \hookrightarrow^{\circlearrowleft +} y$ iff h is undefined for $s(y)$ and there are unique heaps h_1, h_2 such that $h_1 * h_2 = h$, (s, h_1) is a list segment between x and y and (s, h_2) can be decomposed uniquely as a (finite) collection of cyclic lists.

Proof. We want to prove lemma 1.4.3.1. A location i is *shared* whenever $\tilde{\sharp} i \geq 2$. A location i is *initial* [resp. *final*] whenever $i \in \mathrm{Dom}(h) \setminus \mathrm{Im}(h)$ [resp. $i \in \mathrm{Im}(h) \setminus \mathrm{Dom}(h)$]. It is easy to show that $(s, h) \models_{SL} x \hookrightarrow^{\circlearrowleft +} y$ if and only if

- $s(x)$ is initial,

- $\mathtt{s(y)}$ is final,
- $\mathtt{s(y)}$ is the only final location,
- $\mathtt{h}$ has no shared location.

It is easy to check that if $\mathtt{h}$ is of the form $\mathtt{h}_1 \star \mathtt{h}_2$ with the properties stated in lemma 1.4.3.1, then it satisfies the formula $\mathsf{x} \hookrightarrow^{\circlearrowleft +} \mathsf{y}$, which shows one implication. Let us prove the other implication.

Assume $(\mathtt{s},\mathtt{h}) \models_{SL} \mathsf{x} \hookrightarrow^{\circlearrowleft +} \mathsf{y}$. Since $\mathtt{Dom(h)}$ is finite, the set of descendants of $\mathtt{s(x)}$ forms either a cyclic list, or a lasso (a list segment followed by a cycle) or a list ended by a final location. Since there are no shared locations, there is no lasso; and since $\mathtt{s(x)}$ is initial, it does not belong to a cyclic list. So $\mathtt{s(x)}$ has a descendant that is final. It can only be $\mathtt{s(y)}$, so $\mathtt{h}$ contains a list segment from $\mathtt{s(x)}$ to $\mathtt{s(y)}$. To end the proof, we must show that the rest of the heap contains cyclic lists only. This is equivalent to say that no location different from $\mathtt{s(x)}$ is initial. The proof is by contradiction. Suppose that $\mathtt{i}$ is an initial location distinct from $\mathtt{s(x)}$. Then by the same reasoning as for $\mathtt{s(x)}$, we have $\mathtt{s(y)}$ is a descendant of $\mathtt{i}$, so two distinct paths reach $\mathtt{s(y)}$, which contradicts the absence of shared locations. □

Now, thanks to this predicate, we can introduce additional formulas in $\mathtt{SL}^\star$ that are useful in the sequel, whose semantics is provided in lemma 1.4.3.2.

$$\begin{array}{rcl} \mathsf{ls}(\mathsf{x},\mathsf{y}) & \triangleq & \mathsf{x} \hookrightarrow^{\circlearrowleft +} \mathsf{y} \wedge \neg(\mathsf{x} \hookrightarrow^{\circlearrowleft +} \mathsf{y} \star \neg \mathsf{emp}) \\ \mathsf{x} \hookrightarrow^{+} \mathsf{y} & \triangleq & (\mathsf{x} = \mathsf{y} \wedge \mathsf{x} \hookrightarrow \mathsf{y}) \vee (\top \star \mathsf{ls}(\mathsf{x},\mathsf{y})) \\ \mathsf{x} \hookrightarrow^{*} \mathsf{y} & \triangleq & \mathsf{x} = \mathsf{y} \vee \mathsf{x} \hookrightarrow^{+} \mathsf{y} \end{array}$$

Lemma 1.4.3.2. Let $(\mathtt{s},\mathtt{h})$ be a simple memory shape.

(I) $(\mathtt{s},\mathtt{h}) \models_{SL} \mathsf{ls}(\mathsf{x},\mathsf{y})$ iff $(\mathtt{s},\mathtt{h})$ is a list segment between x and y.

(II) $(\mathtt{s},\mathtt{h}) \models_{SL} \mathsf{x} \hookrightarrow^{*} \mathsf{y}$ [resp. $(\mathtt{s},\mathtt{h}) \models_{SL} \mathsf{x} \hookrightarrow^{+} \mathsf{y}$] iff y is a descendant [resp. strict descendant] of x.

Remarks

We could also define these formulas as follows::

$$\begin{array}{lll} \mathsf{x} \hookrightarrow^{*} \mathsf{y} & \text{for} & \mathsf{x} = \mathsf{y} \vee \Big(\top \star (\ (\mathsf{x} \hookrightarrow \square) \wedge (\square \hookrightarrow \mathsf{y}) \wedge \neg(\square \hookrightarrow \mathsf{x}) \wedge \neg(\mathsf{y} \hookrightarrow \square) \\ & & \qquad \wedge \forall \mathsf{z}.(\mathsf{x} \neq \mathsf{z} \wedge \mathsf{y} \neq \mathsf{z}) \Rightarrow ((\mathsf{z} \hookrightarrow \square) \Leftrightarrow (\square \hookrightarrow \mathsf{z})))\Big) \\ \mathsf{x} \hookrightarrow^{+} \mathsf{y} & \text{for} & \exists \mathsf{z}.\mathsf{x} \hookrightarrow \mathsf{z} \wedge \mathsf{z} \hookrightarrow^{*} \mathsf{y} \end{array}$$

Additionally we can define the binary predicate $\mathsf{decls}(\mathsf{x},\mathsf{y})$ that characterises a heap composed of a single list segment with data sorted in the decreasing order.

$$\begin{array}{lll} \mathsf{decls}(\mathsf{x},\mathsf{y}) & \text{for} & (\mathsf{x} = \mathsf{y} \wedge \mathsf{emp}) \ \vee \ \mathsf{x} \mapsto \mathsf{y} \\ & & \vee\, \mathsf{precisely}(\exists \mathsf{y}'.\, \mathsf{x} \hookrightarrow^{+} \mathsf{y}' \ \wedge \ \mathsf{y}' \hookrightarrow \mathsf{y} \wedge \forall \mathsf{z}.(\mathsf{z} \hookrightarrow^{+} \mathsf{y}') \Rightarrow (\mathsf{z} \hookrightarrow_{\geq} \square)) \end{array}$$

Chapter 2

On the Almighty Wand

Introduction

Contribution of this Chapter

In this chapter, we address simultaneously the decidability, complexity, expressive power, and minimality of SL with and without magic wand.

We show that SL is as expressive as SO. This is refined by showing that SL without the separating conjunction is as expressive as SL, whence undecidable too. Our proof also shows that the two formalisms have the same conciseness modulo logarithmic space translations. Moreover, we generalize these results to non-linear recursive data structures: we will define kSL, a separation logic over heaps with exactly $k \geq 1$ record fields in each memory cell, and show it equivalent to kSO, the second-order logic over these heaps.

As a by-product, we get that SL is undecidable even if it has a unique selector, (solving an open problem stated in the article of Galmiche and Méry [55] which adopts a proof-theoretic perspective on SL), and that SL is not a minimal logic as the magic wand can simulate the separating conjunction (but it does not have the adjunct elimination).

We also establish that SL without the magic wand is decidable, but with a non-elementary complexity (this lower bound is obtained by reduction from satisfiability for the first-order theory over finite words whose complexity is proved by Stockmeyer in [87], and holds already with three variables). Decidability is shown by reduction to weak monadic second-order theory of one unary total function that is shown decidable by Rabin in [82]. As a by-product, we obtain that the entailment problem considered by Berdine, Calcagno and O'Hearn in [11] for a fragment of SL is decidable. We also establish that decidability can be obtained with a restricted use of the magic wand containing its usage occurring in Hoare-like proof systems involving separation logic.

Figures 2.2 and 2.1 contain together our decidability results concerning models with one selector. Figure 2.2 is a sketch of the expressiveness results concerning undecidable logics – each arrow represents a logarithmic space translation. Figure 2.1 is similar for decidable logics – the solid arrow represents a logarithmic space translation and the dotted arrow is a polynomial time translation.

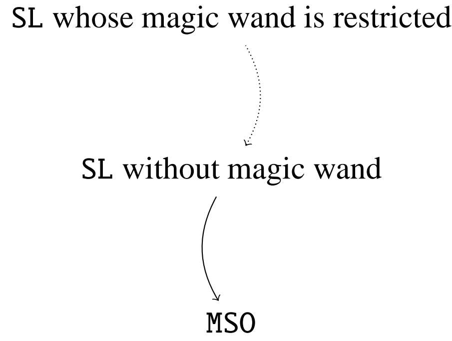


Figure 2.1: Translations proving decidability

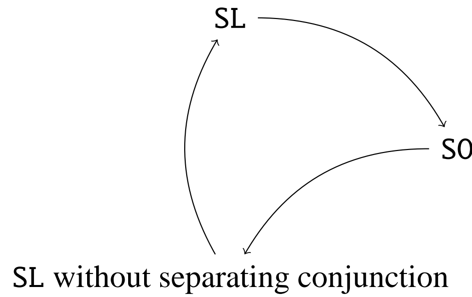


Figure 2.2: Translations proving undecidability

Structure of the Chapter

In section 2.1.1, we show that SL restricted to the separating conjunction (called herein $\mathtt{SL}^{*}$) is decidable with non-elementary complexity. The complexity lower bound is obtained by reduction from the first-order theory over finite words and the decidability is obtained by a logarithmic space reduction into weak monadic second-order theory for one unary function. In section 2.1.2, we extend this decidability result with a restricted use of the magic wand.

Section 2.2 contains many technical contributions about the expressive power of SL, in particular we show how to express advanced arithmetical constraints about the memory heap in SL restricted to the magic wand (called herein $\mathtt{SL}^{-\!*}$). These results are essential to show in section 2.3 that $\mathtt{DSO} \sqsubseteq \mathtt{SL}^{-\!*}$. We conclude from this result that $\mathtt{SL}^{-\!*}$, SL, DSO and SO have the same expressive power (via logarithmic space translations). This implies undecidability of the validity problem for any of these logics, by the undecidability of classical predicate logic with one binary relation proved by Trakhtenbrot in [88]. Section 2.4 extends these results to kSL.

This section presents results originally published in [27], and in [29].

2.1 A Decidable Fragment with a Restricted Wand

In this first section, we first show that $\mathtt{SL}^\star$ satisfiability is decidable but with non-elementary recursive complexity. Then we will study a restricted use of $\mathrel{-\!\!*}$ that will be shown to be decidable through a translation to $\mathtt{MSO}$.

2.1.1 A Complexity Result without Wand

We will here translate $\mathtt{SL}^\star$ to $\mathtt{MSO}$, shown decidable below, before showing the complexity of $\mathtt{SL}^\star$ through a reduction from the first-order theory of finite words. Our result will be explained in theorem 2.1.

In fact, as conjectured in [27], it has been later shown that $\mathtt{MSO}$ is strictly more expressive than $\mathtt{SL}^\star$ by Antonopoulos and Dawar in [3]. They have proved $\mathtt{SL}^\star$ cannot specify that for some $\mathtt{n}$ the a model has $\mathtt{3n}$ allocated cells with no predecessor. They then concluded that since $\mathtt{MSO}$ can, $\mathtt{MSO}$ is strictly more expressive than $\mathtt{SL}^\star$.

Additionally, as a corollary of our result, one can obtain an alternative decidability proof of the entailment problem for the fragment of $\mathtt{SL}$ considered in [11], the *symbolic heaps fragment*. The symbolic heaps fragment is $\mathtt{SL}$ deprived of the $\neg$ and $\vee$ operators and of universal quantification, but containing a list predicate. Its entailment problem was first shown to be in co-NP [11], and more recently in polynomial time in [38]. We have established decidability for a fragment of $\mathtt{SL}$ larger than the symbolic heaps fragment but of higher complexity.

Lemma 2.1.1.1. $\mathtt{MSO}$ satisfiability is decidable.

Proof. The weak monadic second-order theory of unary functions is the theory over structures of the form $(\mathtt{Domain}, \mathtt{fct}, =)$ where $\mathtt{Domain}$ is a countable domain, $\mathtt{fct}$ is a unary function, and $=$ is equality, see [82]. This theory, which we will call $\mathtt{MSO}_{\mathtt{fct}}$ is decidable, see for instance [16, Corollary 7.2.11]. Since in such a logical language it is possible to express that $\mathtt{Domain}$ is infinite and to simulate that $\mathtt{fct}$ is a partial function with finite domain (use a monadic predicate symbol to be interpreted as the finite domain of $\mathtt{fct}$), one can specify that $(\mathtt{Domain}, \mathtt{fct}, =)$ augmented with a first-order valuation is isomorphic to a heap. Based on these elementary facts, we define a translation $\mathtt{tr}_{\mathtt{MSO}\to\mathtt{MSO}_{\mathtt{fct}}}(\mathsf{P}, \cdot)$, computable in logarithmic space, such that a $\mathtt{MSO}$ sentence f is satisfiable iff

$$\overbrace{(\neg\exists \mathsf{P}.\ \forall \mathsf{x}.\ \mathsf{P}(\mathsf{x}))}^{\text{infinity}} \wedge \exists \mathsf{P}.\ \mathtt{tr}_{\mathtt{MSO}\to\mathtt{MSO}_{\mathtt{fct}}}(\mathsf{P}, \mathsf{f})$$

is satisfiable in the weak monadic second-order theory of one unary function. The translation $\mathtt{tr}_{\mathtt{MSO}\to\mathtt{MSO}_{\mathtt{fct}}}(\mathsf{P}, \cdot)$ is defined as follows:

$$\begin{aligned}
\mathtt{tr}_{\mathtt{MSO}\to\mathtt{MSO}_{\mathtt{fct}}}(\mathsf{P}, \mathsf{x} \hookrightarrow \mathsf{y}) &\triangleq \mathsf{P}(\mathsf{x}) \wedge \mathtt{fct}(\mathsf{x}) = \mathsf{y} \\
\mathtt{tr}_{\mathtt{MSO}\to\mathtt{MSO}_{\mathtt{fct}}}(\mathsf{P}, \mathsf{x} = \mathsf{y}) &\triangleq \mathsf{x} = \mathsf{y} \\
\mathtt{tr}_{\mathtt{MSO}\to\mathtt{MSO}_{\mathtt{fct}}}(\mathsf{P}, \mathsf{Q}(\mathsf{x})) &\triangleq \mathsf{Q}(\mathsf{x})
\end{aligned}$$

$\mathtt{tr}_{\mathtt{MSO}\to\mathtt{MSO}_{\mathtt{fct}}}(\mathsf{P}, \cdot)$ is homomorphic for the boolean connectives and for quantifications. □

Using a technique similar to the proof of lemma 2.1.1.1, we now translate $\mathtt{SL}^\star$ into $\mathtt{MSO}$, which will entail decidability for $\mathtt{SL}^\star$.

Lemma 2.1.1.2. $\mathtt{SL}^\star \sqsubseteq \mathtt{MSO}$ via a logarithmic space translation.

Proof. Any formula f in $\mathtt{SL}^\star$ is satisfiable iff

$$\exists P.\ (\forall x.\ P(x) \Leftrightarrow (\exists y.\ x \hookrightarrow y)) \wedge \mathtt{tr}_{\mathtt{SL}^{sep} \to \mathtt{MSO}}(P, f)$$

is satisfiable where $\mathtt{tr}_{\mathtt{SL}^{sep} \to \mathtt{MSO}}(P, \cdot)$ is defined with the following clauses:

- $\mathtt{tr}_{\mathtt{SL}^{sep} \to \mathtt{MSO}}(P, x \hookrightarrow y) \triangleq P(x) \wedge x \hookrightarrow y$,
- $\mathtt{tr}_{\mathtt{SL}^{sep} \to \mathtt{MSO}}(P, x = y) \triangleq x = y$,
- $\mathtt{tr}_{\mathtt{SL}^{sep} \to \mathtt{MSO}}(P, f \ast g) \triangleq \exists Q, Q'.\ P = Q \uplus Q' \wedge \mathtt{tr}_{\mathtt{SL}^{sep} \to \mathtt{MSO}}(Q, f) \wedge \mathtt{tr}_{\mathtt{SL}^{sep} \to \mathtt{MSO}}(Q', g)$ where $P = Q \uplus Q'$ is an abbreviation for $\forall x.\ (P(x) \Leftrightarrow (Q(x) \vee Q'(x))) \wedge \neg(Q(x) \wedge Q'(x))$.

$\mathtt{tr}_{\mathtt{SL}^{sep} \to \mathtt{MSO}}(P, \cdot)$ is homomorphic for the boolean connectives and for first-order quantification. □

As a corollary of the two previous lemmas, $\mathtt{SL}^\star$ satisfiability is decidable.

In order to show that satisfiability in $\mathtt{SL}^\star$ is not elementary recursive, we explain below how to encode finite words as simple memory shapes. Let $\mathtt{A} = \{a_1, \ldots, a_n\}$ be a finite alphabet. A finite word $\mathtt{wd}$ is usually represented as the first-order structure $(\{1, \ldots, |\ \mathtt{wd}\ |\}, <, (P_a)_{a \in \mathtt{A}})$ where P_a is the set of positions labelled by the letter a. Similarly, the word $\mathtt{wd}$ can be represented as a simple memory shape $(s_{\mathtt{wd}}, h_{\mathtt{wd}})$ in which

- $x_{beg} \hookrightarrow^+ x_{end}$ holds true and, x_{beg} and x_{end} are distinguished variables marking respectively, the beginning and the end of the encoding of $\mathtt{wd}$ (they do not encode any of its letters),
- the list segment induced from the satisfaction of $x_{beg} \hookrightarrow^+ x_{end}$ has exactly $|\ \mathtt{wd}\ | + 2$ locations. Also, any location of position $\mathtt{j} \in \{2, \ldots, |\ \mathtt{wd}\ | + 1\}$ in the list segment (hence excluding $s_{\mathtt{wd}}(x_{beg})$ and $s_{\mathtt{wd}}(x_{end})$) has exactly $\mathtt{k}$ predecessors if $P_{a_k}(\mathtt{j})$ holds; additionally we call this location $\mathtt{i}_{\mathtt{j}} - 1$. Since $s_{\mathtt{wd}}(x_{beg})$ and $s_{\mathtt{wd}}(x_{end})$ do not encode any position in $\mathtt{wd}$, there is no constraint on them.

In figure 2.3, we represent a simple memory shape encoding the finite word $a_1 a_2 a_3 a_1$. Throughout the chapter, a simple memory shape (s, h) is encoded as a graph representing the heap such that there is an edge from $\mathtt{i}$ to $\mathtt{i}'$ iff $h(\mathtt{i}) = \mathtt{i}'$. Locations are represented by letters $\mathtt{i}$ (representing themselves), variables x (representing $s(x)$) or a joker location □ (representing an unspecified location different from all the other locations present in the graph). Although the graph of h is fully specified, we may omit irrelevant variables in the representation of (s, h). In figure 2.3, note that each position of the word corresponds to a unique location in the simple memory shape. For instance, the location $\mathtt{i}_4$ has one predecessor encoding the fact that the fourth letter in the word is precisely the first letter a_1. The location $\mathtt{i}_3$ has 3 predecessors encoding that fact that the third letter of the word is precisely the third letter is a_3.

Similarly, any simple memory shape (s, h) containing a list segment between x_{beg} and x_{end} and such that any location on the list segment that is different from $s(x_{beg})$ and $s(x_{end})$ has at most $|\mathtt{A}|$ predecessors corresponds to a unique finite word with the above encoding. In this direction, the simple memory shape may contain other dummy locations but they are irrelevant for the representation of the finite word. Moreover, a simple memory shape can encode only one word since x_{beg} and x_{end} are end-markers and x_{beg} can only have one successor.

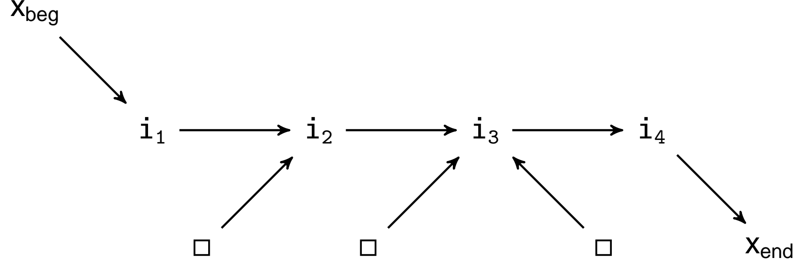


Figure 2.3: Memory state encoding the finite word $a_1 a_2 a_3 a_1$

Theorem 2.1. $SL^\star$ satisfiability is decidable and not elementary recursive. Its restriction with five variables is also not elementary recursive.

Proof. Satisfiability of the first-order theory of finite words [87] is not elementary recursive (this result holds already with three variables). Let us reduce this problem, that we will call FO_{words} to satisfiability in $SL^\star$. Let g_{word} be the formula specifying a word model:

$$(x_{beg} \hookrightarrow^+ x_{end}) \wedge (\forall x.\ (x_{beg} \hookrightarrow^+ x) \wedge (x \hookrightarrow^+ x_{end}) \Rightarrow \sharp x \leq |A|)$$

It is then easy to show that given a first-order formula f over the signature $(<, (P_a)_{a \in A})$, f is satisfiable over finite words iff $g_{word} \wedge tr_{FO_{words} \to SL^{sep}}(f)$ is satisfiable in $SL^\star$ where $tr_{FO_{words} \to SL^{sep}}$ is defined as follows:

$$\begin{aligned}
tr_{FO_{words} \to SL^{sep}}(x < y) &\triangleq (x \hookrightarrow^+ y)\\
tr_{FO_{words} \to SL^{sep}}(\forall x.\ g) &\triangleq \forall x.\ (x_{beg} \hookrightarrow^+ x) \wedge (x \hookrightarrow^+ x_{end}) \Rightarrow tr_{FO_{words} \to SL^{sep}}(g)\\
tr_{FO_{words} \to SL^{sep}}(P_{a_i}(x)) &\triangleq \sharp x = i.
\end{aligned}$$

Remember that $\sharp x = i$ is a shortcut for a formula in $SL^\star$ of size proportional to i (see section 1.4.3). The translation $tr_{FO_{words} \to SL^{sep}}$ is homomorphic for boolean connectives. Similarly, $x \hookrightarrow^+ y$ and $\sharp x \leq |A|$ belongs to $SL^\star$ (see section 1.4.3). One can check that if f contains at most three variables, then $g_{word} \wedge tr_{FO_{words} \to SL^{sep}}(f)$ contains at most five variables. $\square$

It is probable that the number of variables can be reduced further while preserving non-elementarity, for instance by identifying the limits of the words by unique patterns instead of distinguished variables – but it is not very essential at this point.

2.1.2 A Restricted Use of the Wand

We have as of now seen that $SL^\star$ satisfiability is decidable, whereas satisfiability for full SL will be shown to be undecidable. However, $SL^\star$ is certainly not the largest decidable fragment of SL. In the sequel of this section, we investigate another decidable extension of $SL^\star$ thanks to a restricted use of the magic wand; quantification over disjoint heaps is done only for heaps whose domain has cardinality smaller than some fixed n (details will follow). Since the forthcoming extension is closed under negation, this also corresponds to a restricted use of the operator $\overset{\neg}{\ast}$.

Let us define $SL^{\star, -\!\!\ast_n}$ as an extension of $SL^\star$ by adding the binary operators $-\!\!\ast_n$ for every $n \in \mathbb{N}$. Unlike the plain operator $-\!\!\ast$, a formula with outermost connective $-\!\!\ast_n$ quantifies over

disjoint heaps for which the cardinality of the domain is bounded by n. The integer n in the connective $-\!\!*_n$ is encoded in a unary system.

Definition 2.1.2.1. Let $SL^{*,-\!*_n}$ be the logic defined by the grammar below and whose formulas are interpreted over simple memory shapes:

$$f := \neg f \mid f \wedge f \mid \exists x.f \mid x \hookrightarrow y \mid x = y \mid f * f \mid f -\!\!*_n f$$

Additionally, $(s,h) \models_{SL} f_1 -\!\!*_n f_2$ iff for all $h' \perp h$ such that $|Dom(h')| \leq n$, if $(s,h') \models_{SL} f_1$ then $(s, h * h') \models_{SL} f_2$.

$SL^{*,-\!*_n}$ allows to encode the restricted use of the magic wand in the Hoare-like proof systems as in the backward-reasoning form rule (MUBR) recalled below, see also [84]:

$$\overline{\{(\exists z.\ x \mapsto z) * ((x \mapsto y) -\!\!* f)\}\ [x] := y\ \{f\}}$$

The precondition of this rule states with the subformula $\exists z.\ x \mapsto z$ that the variable x is allocated, and states thanks to the separating conjunction that $(x \mapsto y) -\!\!* f$ holds on the model whose heap is modified so that the cell of x is removed. The subformula $(x \mapsto y) -\!\!* f$ states for this modified model that: if a new cell pointing to y is added under x with the magic wand, then f will hold. Removing the cell under x with $*$ so as to replace it with a new cell pointing to y with $-\!\!*$ is a trick to apply the instruction $[x] := y$ to the model. Therefore, the precondition checks that its model modified by $[x] := y$ satisfies f.

It is easy to show that $(x \mapsto y) -\!\!* f$ is equivalent to $(x \mapsto y) -\!\!*_1 f$. Typically whenever the left argument of a formula with outermost connective $-\!\!*$ has only models of bounded size, this trick can be applied again. Let us push a bit further this idea.

2.1.3 Preliminaries to the Translation

Bounding the Cardinal of Heap Domains

Definition 2.1.3.1. Let SL^{*m} be the fragment of SL defined by the grammar below and whose formulas are also interpreted over simple memory shapes:

$$f ::= \perp \mid x \mapsto y \mid emp \mid f * f \mid f \vee f \mid f \wedge f \mid \exists x.f$$

Let $SL^{<n}$ be the logic defined by the grammar below and whose formulas are also interpreted over simple memory shapes:

$$g := \neg g \mid g \wedge g \mid \exists x.g \mid x \hookrightarrow y \mid x = y \mid g * g \mid g -\!\!*_k g \mid f -\!\!* g$$

where $k \in \mathbb{N}$, and $f \in SL^{*m}$.

The satisfaction relation is defined as for SL with the help of definition 2.1.2.1.

Observe that SL^{*m} can express formulas $size = k$ and $size \leq k$ with semantics: $(s,h) \models_{SL} size \leq k$ iff $|Dom(h)| \leq k$ and $(s,h) \models_{SL} size = k$ iff $|Dom(h)| = k$. The formula $size = k$ with $k \geq 1$ is equivalent to the following formula: $\exists x_1, \ldots, x_k.((\exists y.x_1 \mapsto y) * \ldots * (\exists y.x_k \mapsto y))$. Also, the formula $size \leq k$ with $k \geq 1$ is equivalent to the following formula: $emp \vee \bigvee_{j \leq k} size = k$. For $k = 0$, they are both equivalent to emp.

Lemma 2.1.3.2. For any $f \in SL^{*m}$, if $(s,h) \models_{SL} f$ then $|Dom(h)| \leq | f |$.

The proof is by a straightforward structural induction. Since computing $| f |$ from f can be done in polynomial time, we obtain the following reduction that becomes especially interesting after showing decidability of $SL^{*,\multimap_n}$.

Lemma 2.1.3.3. There is a polynomial time reduction from satisfiability for $SL^{<n}$ to satisfiability for $SL^{*,\multimap_n}$.

In order to establish the above lemma, it is sufficient to observe that $f \multimap g$ is equivalent to $f \multimap_{|f|} g$ whenever $f \in SL^{*m}$ and $g \in SL$.

Fictitious Heaps

In order to show decidability for $SL^{*,\multimap_n}$, we define a reduction into SL^{*}. The translation is based on a simple observation: since a formula with outermost connective $\multimap_n$ requires the disjoint heaps to have a domain size of at most n, these new heaps can be encoded by a set of pairs of variables of cardinality n. Hence, a heap of size at most n disjoint from (s,h) can be represented symbolically, or fictitiously, by a set $Sh = \{(y_1,z_1),\ldots,(y_n,z_n)\}$ such that $\{s(y_1),\ldots,s(y_n)\} \cap Dom(h) = \emptyset$ and $s(y_i) = s(y_j)$ implies $s(z_i) = s(z_j)$, naturally encoding the heap $h(Sh) = \{s(y_i) \mapsto s(z_i) : s(y_i) \neq s(y_0),\ 1 \leq i \leq n\}$. We assume a variable y_0 which is not allocated, and will be introduced as such at the beginning of the translation; this can be seen as an equivalent of the null constant. The set $Sh = \{(y_1,z_1),\ldots,(y_n,z_n)\}$ will represent a heap with at most n memory cells, even though Sh contains exactly n pairs. However, whenever $s(y_i) = s(y_0)$, the pair (y_i,z_i) does not encode any new memory cell. In terms of formulas, (y_i,z_i) encodes a memory cell iff $y_i \neq y_0$ holds true. This shall be intensively used in forthcoming formulas. Let us provide now the formal definitions.

Definition 2.1.3.4. A *fictitious heap* Sh for the simple memory shape (s,h) is a finite set of pairs of variables $\{(y_1,z_1),\ldots,(y_n,z_n)\}$ such that

- $\{s(y_1),\ldots,s(y_n)\} \cap Dom(h) = \emptyset$.
- For $1 \leq i,j \leq n$, $s(y_i) = s(y_j)$ implies $s(z_i) = s(z_j)$.

The heap represented by Sh is $h(Sh) \triangleq \{s(y_i) \mapsto s(z_i) : s(y_i) \neq s(y_0),\ 1 \leq i \leq n\}$.

Observe that $|Dom(h(Sh))| \leq n$ and $h(Sh) \perp h$. Sh is said to be of *length* n.

Lemma 2.1.3.5. Given a simple memory shape (s,h) and h' such that $h' \perp h$ and $|Dom(h')| \leq n$, there exists Sh a fictitious heap for (s',h) of length n such that $h' = h(Sh)$, for some s' which may differ from s at most for the variables occurring in Sh.

The proof is by an easy verification by symbolically representing h' with new variables. The use of new variables makes it necessary to use a new store s' for applying the definition of $h(Sh)$ which is dependent on the store. The store s' may have to be different from s so as to have enough variables whose value is different from that of y_0.

Below we introduce simple formulas useful to separate a fictitious heap or to extend a fictitious heap by another fictitious heap. Given the fictitious heaps $Sh = \{(y_1,z_1),\ldots,(y_n,z_n)\}$, $Sh^1 = \{(y^1_1,z^1_1),\ldots,(y^1_{n_0},z^1_{n_0})\}$ and $Sh^2 = \{(y^2_1,z^2_1),\ldots,(y^2_{n_1},z^2_{n_1})\}$, we write $Sh = Sh^1 * Sh^2$ to denote the conjunction of the formulas below:

– $h(Sh)$ is included in $h(Sh^1) \cup h(Sh^2)$:

$$\bigwedge_{1 \leq i \leq n} (\bigvee_{1 \leq j \leq n_0} y^1_j = y_i) \vee (\bigvee_{1 \leq j \leq n_1} y^2_j = y_i)$$

– $h(Sh^1) \cup h(Sh^2)$ is included in $h(Sh)$:

$$(\bigwedge_{1 \leq j \leq n_0} (y^1_j \neq y_0) \Rightarrow (\bigvee_{1 \leq i \leq n} y_i = y^1_j)) \wedge (\bigwedge_{1 \leq j \leq n_1} (y^2_j \neq y_0) \Rightarrow (\bigvee_{1 \leq i \leq n} y_i = y^1_j))$$

– $h(Sh^1)$ and $h(Sh^2)$ encode a function:

$$\bigwedge_{1 \leq j, j' \leq n_0} (y^1_j = y^1_{j'} \Rightarrow z^1_j = z^1_{j'}) \wedge \bigwedge_{1 \leq j, j' \leq n_1} (y^2_j = y^2_{j'} \Rightarrow z^2_j = z^2_{j'})$$

– $h(Sh^1)$ and $h(Sh^2)$ are disjoint:

$$\bigwedge_{1 \leq j \leq n_0} \bigwedge_{1 \leq j' \leq n_1} ((y^1_j \neq y_0) \vee (y^2_{j'} \neq y_0)) \Rightarrow (y^1_j \neq y^2_{j'})$$

We provide a few lemmas whose easy proofs are omitted, except for the last one. All the proofs would be similar and similarly simple, the last one serves as an exemple. The lemmas, will be helpful to prove correctness in section 2.1.4.

Lemma 2.1.3.6. Let Sh be a fictitious heap of length n for (s, h) and let the fictitious heaps $Sh^1 = \{(y^1_1, z^1_1), \ldots, (y^1_{n_0}, z^1_{n_0})\}$ and $Sh^2 = \{(y^2_1, z^2_1), \ldots, (y^2_{n_1}, z^2_{n_1})\}$ be such that their variables do not occur in Sh. Let s' be a store that may differ from s at most for the variables occurring in Sh^1 and Sh^2. Assume moreover that $(s', h) \models_{SL} Sh = Sh^1 * Sh^2$. Then, Sh^1 and Sh^2 are fictitious heaps for (s', h), $h(Sh^1) \perp h(Sh^2)$ and $h(Sh^1) * h(Sh^2) = h(Sh)$.

Again, the proof is by easy verification and we can also get a converse property.

Lemma 2.1.3.7. Let Sh be a fictitious heap of length n for (s, h). Let $h_1 * h_2 = h(Sh)$. There exist fictitious heaps Sh^1 and Sh^2 for (s', h) such that variables in Sh, Sh^1 and Sh^2 are mutually disjoint, s' may differ from s at most for the variables occurring in Sh^1 and Sh^2, $h_1 = h(Sh^1)$, $h_2 = h(Sh^2)$ and $(s', h) \models_{SL} Sh = Sh^1 * Sh^2$.

Let us now consider the corresponding lemmas to build disjoint heaps.

Lemma 2.1.3.8. Let Sh^1 be a fictitious heap for (s, h), Sh and Sh^2 be fictitious heaps for (s, h) whose variables do not occur in Sh^1, and such that $(s', h) \models_{SL} Sh = Sh^1 * Sh^2$, where s' may differ from s at most for the variables occurring in Sh and Sh^2. Then, Sh and Sh^2 are fictitious heaps for (s', h).

We can also get a converse property.

Lemma 2.1.3.9. Let Sh^1 be a fictitious heap for (s, h) and, h' be disjoint from $h * h(Sh^1)$ and the cardinal of its domain is less than n. There exists a fictitious heap Sh^2 of length n for (s', h) such that $h' = h(Sh^2)$, $h' * h(Sh^1) = h(Sh^1 \cup Sh^2)$ and $(s', h) \models_{SL} (Sh^1 \cup Sh^2) = Sh^1 * Sh^2$ (s' may differ from s at most for the variables occurring in Sh^2).

Proof. Let $\mathtt{Sh}^1$ be a fictitious heap for $(\mathtt{s},\mathtt{h})$ and $\mathtt{h}'$ be disjoint from $\mathtt{h}*\mathtt{h}(\mathtt{Sh}^1)$, so that the cardinal of its domain is less than $\mathtt{n}$. Let $\{\mathtt{i}_1,\ldots,\mathtt{i}_k\} = \mathtt{Dom}(\mathtt{h}')$, hence $\mathtt{k} \le \mathtt{n}$. Let $\{(\mathtt{y}^1_1,\mathtt{z}^1_1),\ldots(\mathtt{y}^1_{n_1},\mathtt{z}^1_{n_1})\} = \mathtt{Sh}^1$.

Let $\mathrm{Sh}^2 = \{(\mathtt{y}^2_1,\mathtt{z}^2_1),\ldots(\mathtt{y}^2_n,\mathtt{z}^2_n)\}$, with $\mathtt{y}^2_1,\ldots,\mathtt{y}^2_n,\mathtt{z}^2_1,\ldots,\mathtt{z}^2_n$ new variables. Let $\mathtt{s}'(\mathtt{y}_m)$ be $\mathtt{i}_m$ if $\mathtt{m} \le \mathtt{k}$ and $\mathtt{s}(\mathtt{y}_0)$ otherwise. Let $\mathtt{s}'(\mathtt{z}_m)$ be $\mathrm{h}'(\mathtt{i}_m)$ if $\mathtt{m} \le \mathtt{k}$ and $\mathtt{s}(\mathtt{y}_0)$ otherwise.

Then clearly $\mathrm{h}' = \mathtt{h}(\mathtt{Sh}^2)$. Also:

$$
\begin{aligned}
\mathtt{h}(\mathtt{Sh}^1 \cup \mathtt{Sh}^2) &= \mathtt{h}(\{(\mathtt{y}^1_1,\mathtt{z}^1_1),\ldots(\mathtt{y}^1_{n_1},\mathtt{z}^1_{n_1})\} \cup \{(\mathtt{y}^2_1,\mathtt{z}^2_1),\ldots(\mathtt{y}^2_n,\mathtt{z}^2_n)\}) \\
&= \{\mathtt{s}'(\mathtt{y}^1_k) \mapsto \mathtt{s}'(\mathtt{z}^1_k) : \mathtt{s}'(\mathtt{y}^1_k) \neq \mathtt{s}'(\mathtt{y}_0),\ 1 \le \mathtt{k} \le \mathtt{n}_1\} \\
&\quad \cup \{\mathtt{s}'(\mathtt{y}^2_k) \mapsto \mathtt{s}'(\mathtt{z}^2_k) : \mathtt{s}'(\mathtt{y}^2_k) \neq \mathtt{s}'(\mathtt{y}_0),\ 1 \le \mathtt{k} \le \mathtt{n}\} \\
&= \{\mathtt{s}'(\mathtt{y}^1_k) \mapsto \mathtt{s}'(\mathtt{z}^1_k) : \mathtt{s}'(\mathtt{y}^1_k) \neq \mathtt{s}'(\mathtt{y}_0),\ 1 \le \mathtt{k} \le \mathtt{n}_1\} \\
&\quad * \{\mathtt{s}'(\mathtt{y}^2_k) \mapsto \mathtt{s}'(\mathtt{z}^2_k) : \mathtt{s}'(\mathtt{y}^2_k) \neq \mathtt{s}'(\mathtt{y}_0),\ 1 \le \mathtt{k} \le \mathtt{n}\} \\
&= \mathtt{h}(\mathtt{Sh}^1) * \mathtt{h}(\mathtt{Sh}^2) \\
&= \mathtt{h}' * \mathtt{h}(\mathtt{Sh}^1)
\end{aligned}
$$

Finally, $(\mathtt{Sh}^1 \cup \mathtt{Sh}^2) = \mathtt{Sh}^1 * \mathtt{Sh}^2$ is $(\mathtt{Sh}^1 \cup \mathtt{Sh}^2) \subseteq (\mathtt{Sh}^1 * \mathtt{Sh}^2) \wedge (\mathtt{Sh}^1 * \mathtt{Sh}^2) \subseteq (\mathtt{Sh}^1 \cup \mathtt{Sh}^2)$. By its definition, $(\mathtt{Sh}^1 * \mathtt{Sh}^2) \subseteq (\mathtt{Sh}^1 \cup \mathtt{Sh}^2)$ is $\wedge_{\mathtt{y}\in\{\mathtt{y}^1_1,\ldots,\mathtt{y}^1_{n_1},\mathtt{y}^2_1,\ldots,\mathtt{y}^2_n\}}(\vee_{1\le j\le n_1}\mathtt{y}^1_j = \mathtt{y}) \vee (\vee_{1\le j\le n}\mathtt{y}^2_j = \mathtt{y})$. When $\mathtt{y} \in \{\mathtt{y}^1_1,\ldots,\mathtt{y}^1_{n_1},\mathtt{y}^2_1,\ldots,\mathtt{y}^2_n\}$, then either $\mathtt{y} \in \{\mathtt{y}^1_1,\ldots,\mathtt{y}^1_{n_1}\}$ and $\mathtt{y}^1_j = \mathtt{y}$ is true in $\mathtt{s}'$ for some $\mathtt{j}$, or $\mathtt{y} \in \{\mathtt{y}^2_1,\ldots,\mathtt{y}^2_n\}$ and $\mathtt{y}^2_j = \mathtt{y}$ is true in $\mathtt{s}'$ for some $\mathtt{j}$. As a consequence, $(\mathtt{s},\mathtt{h}) \models_{\mathrm{SL}} (\mathtt{Sh}^1 * \mathtt{Sh}^2) \subseteq (\mathtt{Sh}^1 \cup \mathtt{Sh}^2)$. Similarly, one can show that $(\mathtt{s},\mathtt{h}) \models_{\mathrm{SL}} (\mathtt{Sh}^1 \cup \mathtt{Sh}^2) \subseteq (\mathtt{Sh}^1 * \mathtt{Sh}^2)$. □

2.1.4 The Translation

The recursive translation function is of the form $\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{g}, \mathrm{Sh}, \mathtt{m})$ where g is a subformula to be translated, Sh has the format of some fictitious heap and $\mathtt{m} \in \{0,1\}$ is a flag that specifies whether g is evaluated under $\mathtt{h}(\mathtt{Sh})$ ($\mathtt{m} = 0$) or under $\mathtt{h} * \mathtt{h}(\mathtt{Sh})$ ($\mathtt{m} = 1$).

Definition 2.1.4.1. A formula f is translated into $\exists \mathsf{y}_0.(\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{f}, \emptyset, 1) \wedge \neg\mathsf{alloc}(\mathsf{y}_0))$, where the recursive map $\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}$, is defined as follows:

- $\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{x} = \mathsf{x}', \mathrm{Sh}, \mathtt{m}) \triangleq \mathsf{x} = \mathsf{x}'$.
- $\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{x} \hookrightarrow \mathsf{x}', \mathrm{Sh}, 1) \triangleq (\mathsf{x} \hookrightarrow \mathsf{x}') \vee \mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{x} \hookrightarrow \mathsf{x}', \mathrm{Sh}, 0)$.
- $\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{x} \hookrightarrow \mathsf{x}', \mathrm{Sh}, 0) \triangleq \bigvee_{(\mathsf{y},\mathsf{z})\in\mathrm{Sh}} \mathsf{y} \neq \mathsf{y}_0 \wedge \mathsf{y} = \mathsf{x} \wedge \mathsf{z} = \mathsf{x}'$.
- $\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}$ is homomorphic for boolean connectives and first-order quantification.
- $\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{g} \mathrel{-\!\!*}_0 \mathsf{g}', \mathrm{Sh}, \mathtt{m}) \triangleq \mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{g}, \emptyset, 0) \wedge \mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{g}', \mathrm{Sh}, \mathtt{m})$
- $\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{g} \mathrel{-\!\!*}_n \mathsf{g}', \mathrm{Sh}, \mathtt{m})$ for $\mathtt{n} \ge 1$ is defined as

$$
\begin{aligned}
&\forall \mathsf{y}'_1,\ldots,\mathsf{y}'_n,\mathsf{z}'_1,\ldots,\mathsf{z}'_n. \\
&(((\mathrm{Sh} \cup \mathrm{Sh}') = \mathrm{Sh} * \mathrm{Sh}' \wedge \textstyle\bigwedge_{(\mathsf{y},\mathsf{z})\in\mathrm{Sh}'} \neg\mathsf{alloc}(\mathsf{y})) \\
&\Rightarrow (\mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{g}, \mathrm{Sh}', 0) \wedge \mathtt{tr}_{\mathrm{SL}^{\mathrm{sep,mw-n}} \to \mathrm{SL}^{\mathrm{sep}}}(\mathsf{g}', \mathrm{Sh} \cup \mathrm{Sh}', \mathtt{m})))
\end{aligned}
$$

  where $\mathsf{y}'_1,\ldots,\mathsf{y}'_n,\mathsf{z}'_1,\ldots,\mathsf{z}'_n$ is a sequence of $2\mathsf{n}$ pairs of fresh variables which defines the fictitious heap Sh'.

– $\mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g * g', Sh, m)$ with Sh of length n is defined as

$$\exists y^1_1,\ldots,y^1_n,y^2_1,\ldots,y^2_n,z^1_1,\ldots,z^1_n,z^2_1,\ldots,z^2_n.$$
$$(\mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g, Sh^1, m) * \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g', Sh^2, m)) \wedge Sh = Sh^1 * Sh^2$$

where $y^1_1,\ldots,y^1_n,y^2_1,\ldots,y^2_n,z^1_1,\ldots,z^1_n,z^2_1,\ldots,z^2_n$ is a sequence of 4n pairs of fresh variables which defines the fictitious heaps Sh^1 and Sh^2 of length n.

Even though in the worst-case there is an exponential number of ways to divide a heap into two disjoint heaps, our translation remains in polynomial time as the integer n in the operator $-\!\!*_n$ is encoded in a unary system. The soundness of the translation is guaranteed by the lemma below whose proof is by structural induction and uses the previous lemmas.

Lemma 2.1.4.2. Let Sh be a fictitious heap for (s,h). For all formulas g in $SL^{*,-\!*_n}$, we have $(s,h(Sh)) \models_{SL} g$ iff $(s,h) \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g, Sh, 0)$ and $(s,h*h(Sh)) \models_{SL} g$ iff $(s,h) \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g, Sh, 1)$.

Proof. The proof is by structural induction on g. The base case for atomic formulas is by an easy verification as well as the cases in the induction step for boolean connectives and first-order quantification. We treat below the case $g = g_1 * g_2$, the case $g = g_1 -\!\!*_n g_2$ can be treated analogously using lemmas 2.1.3.8 and 2.1.3.9. The induction hypothesis is of the following form: for every g' whose size is strictly smaller than the size of g, if Sh' be a fictitious heap for (s'',h''), then we have $(s'',h(Sh')) \models_{SL} g'$ iff $(s'',h'') \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g', Sh', 0)$ and $(s'',h''*h(Sh')) \models_{SL} g'$ iff $(s'',h'') \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g', Sh', 1)$.

Suppose $(s,h(Sh)) \models_{SL} g_1 * g_2$. There exist heaps h_1 and h_2 such that $h_1 * h_2 = h(Sh)$, $(s,h_1) \models_{SL} g_1$ and $(s,h_2) \models_{SL} g_2$. By lemma 2.1.3.7, there exist fictitious heaps Sh_1 and Sh_2 (with fresh variables) for (s',h) such that s' may differ from s at most for the variables occurring in $Sh_1 \cup Sh_2$, $h_1 = h(Sh_1)$ and $h_2 = h(Sh_2)$. Since each Sh_i is a fictitious heap for (s',h_i), by the induction hypothesis, $(s',h) \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_1, Sh_1, 0)$ and $(s',h) \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_2, Sh_2, 0)$. Moreover, $(s',h) \models_{SL} Sh = Sh_1 * Sh_2$ (observe that satisfaction of $Sh = Sh_1 * Sh_2$ depends only on the store). Hence,

$$(s,h) \models_{SL} \exists y^1_1,\ldots,y^1_n,y^2_1,\ldots,y^2_n,z^1_1,\ldots,z^1_n,z^2_1,\ldots,z^2_n.$$
$$(\mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_1, Sh_1, 0) * \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_2, Sh_2, 0)) \wedge Sh = Sh_1 * Sh_2$$

where $y^1_1,\ldots,y^1_n,y^2_1,\ldots,y^2_n,z^1_1,\ldots,z^1_n,z^2_1,\ldots,z^2_n$ is the sequence of variables defining Sh_1 and Sh_2. As a consequence, we can state that $(s,h) \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_1 * g_2, Sh, 0)$.

Similarly, suppose $(s,h*h(Sh)) \models_{SL} g_1 * g_2$. There exist h_1, h_2, h'_1 and h'_2 such that $h_1 * h_2 = h(Sh)$, $h'_1 * h'_2 = h$, $(s,h'_1 * h_1) \models_{SL} g_1$ and $(s,h'_2 * h_2) \models_{SL} g_2$. By lemma 2.1.3.7, there exist fictitious heaps Sh_1 and Sh_2 (with fresh variables) for (s',h) such that s' may differ from s at most for the variables occurring in Sh_1, Sh_2, $h_1 = h(Sh_1)$ and $h_2 = h(Sh_2)$. Since each Sh_i is a fictitious heap for (s',h_i), by the induction hypothesis, $(s',h'_1 * h(Sh_1)) \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_1, Sh_1, 1)$ and $(s',h'_2 * h(Sh_2)) \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_2, Sh_2, 1)$. Additionally, $(s',h) \models_{SL} Sh = Sh_1 * Sh_2$. Hence,

$$(s,h) \models_{SL} \exists y^1_1,\ldots,y^1_n,y^2_1,\ldots,y^2_n,z^1_1,\ldots,z^1_n,z^2_1,\ldots,z^2_n.$$
$$(\mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_1, Sh_1, 1) * \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_2, Sh_2, 1)) \wedge Sh = Sh_1 * Sh_2$$

So, $(s,h) \models_{SL} \mathtt{tr}_{SL^{sep,mw-n}\to SL^{sep}}(g_1 * g_2, Sh, 1)$.

Now suppose $(s,h) \models_{SL} tr_{SL^{sep,mw-n} \to SL^{sep}}(g_1 * g_2, Sh, 0)$, that is

$$(s,h) \models_{SL} \exists y^1_1, \ldots, y^1_n, y^2_1, \ldots, y^2_n, z^1_1, \ldots, z^1_n, z^2_1, \ldots, z^2_n. \\ (tr_{SL^{sep,mw-n} \to SL^{sep}}(g, Sh^1, m) * tr_{SL^{sep,mw-n} \to SL^{sep}}(g', Sh^2, m)) \wedge Sh = Sh^1 * Sh^2$$

where $y^1_1, \ldots, y^1_n, y^2_1, \ldots, y^2_n, z^1_1, \ldots, z^1_n, z^2_1, \ldots, z^2_n$ corresponds to the sequence of fresh variables from the fictitious heaps Sh^1 and Sh^2. Hence there exists a store s' that may differ from s at most for the variables occurring in Sh^1 and Sh^2 such that

$$(s',h) \models_{SL} (tr_{SL^{sep,mw-n} \to SL^{sep}}(g_1, Sh^1, 0) * tr_{SL^{sep,mw-n} \to SL^{sep}}(g_2, Sh^2, 0)) \wedge Sh = Sh^1 * Sh^2$$

From the induction hypothesis, we have $(s', h(Sh_1)) \models_{SL} g_1$ and $(s', h(Sh_2)) \models_{SL} g_2$. By lemma 2.1.3.6, $h(Sh_1) \perp h(Sh_2)$ and $h(Sh_1) * h(Sh_2) = h(Sh)$. As a consequence, we have $(s', h(Sh)) \models_{SL} g_1 * g_2$. Since variables in Sh^1 and Sh^2 do not occur in $g_1 * g_2$, we get $(s, h(Sh)) \models_{SL} g_1 * g_2$.

Similarly, from $(s,h) \models_{SL} tr_{SL^{sep,mw-n} \to SL^{sep}}(g_1 * g_2, Sh, 1)$, one can reach the conclusion that $(s, h * h(Sh)) \models_{SL} g_1 * g_2$ by using lemma 2.1.3.6. □

This leads to the main result of this section.

Lemma 2.1.4.3. There is a polynomial time reduction from $SL^{*,\multimap_n}$ satisfiability problem to SL^{*} satisfiability problem.

Proof. By lemma 2.1.4.2, for every simple memory shape (s,h), we have $(s, h * h(\emptyset)) \models_{SL} g$ iff $(s,h) \models_{SL} tr_{SL^{sep,mw-n} \to SL^{sep}}(g, \emptyset, 1)$ where $tr_{SL^{sep,mw-n} \to SL^{sep}}(g, \emptyset, 1)$ is an SL^{*} formula and $\emptyset$ denotes the empty fictitious heap. Moreover, we have seen that $tr_{SL^{sep,mw-n} \to SL^{sep}}(g, \emptyset, 1)$ can be built in polynomial time assuming that the natural numbers are represented with a unary encoding in g. Since $h * h(\emptyset)$ is equal to h, the formulas g and $tr_{SL^{sep,mw-n} \to SL^{sep}}(g, \emptyset, 1)$ hold true at the same states. □

The following theorem is a consequence of lemma 2.1.4.3 by using the decidability of SL^{*} satisfiability (see section 2.1.1).

Theorem 2.2. $SL^{*,\multimap_n}$ satisfiability is decidable.

We then obtain the following interesting corollary.

Theorem 2.3. Satisfiability for $SL^{<n}$ is decidable.

2.2 Advanced Arithmetical Constraints with the Wand

In this section, we show how $SL^{\multimap}$ can be used to express the following property (P-nb):

> The number $\tilde{\sharp} x$ of locations that point to the location $s(x)$ is at most the number $\tilde{\sharp} y$ of locations that point to $s(y)$ augmented by some constant m.

The reason for expressing this property will become clearer in section 2.3. We may however try to provide a few motivations:

– since we want to express all SO properties, we may already train ourselves with expressing this particular SO property;

– most importantly, more than a pure exercise, this property plays a crucial role in the encoding of SO in SL;

The proof that SL^{-*} can express the above property (P-nb) is subject to technical complications, but its essence is not so intricate, and it is better illustrated by encoding other kinds of cardinality constraints. For this reason, we make a slight detour in our presentation by first sketching the encoding of the following property (P-nb'):

The length of the list starting at x is equal to the length of the list starting at y.

The property (P-nb') turns out to be a bit simpler to define than (P-nb), and it already provides the key ingredients for expressing (P-nb). This property (P-nb') will not be used anywhere else and could have been skipped, but we believe it has a pedagogical value to show how it can be expressed. We first sketch the encoding of (P-nb') in section 2.2.1, and then move in section 2.2.2 to the proof, with full details, of the encoding of (P-nb) in SL^{-*}.

2.2.1 Comparing Two List Lengths

Let us restrict our attention to simple memory shapes composed of two acyclic lists starting at x and y respectively, with no other allocated cells, and with the additional constraint that no location is reachable from x and y simultaneously. We aim now at expressing the fact that both lists have the same length n using the magic wand. To do so, we can say that there exist n locations $i_1, \ldots, i_n$ that are not allocated and for which there is a one-one correspondence between these locations and the ones of the list starting at x, and on the other hand there is another one-one correspondence between these same locations and the ones of the list starting at y. As illustrated by figure 2.4, the gain for considering non allocated cells is that the one-one correspondence can be materialized by allocating $i_1, \ldots, i_n$ so that each of them points to the cell it is in correspondence with. The trickiest point is then how to materialize the guess of the locations $i_1, \ldots, i_n$ in such a way that it is possible to refer to them later without confusing them with the cells that were initially allocated. To do so, we may observe that in the original heap, all locations have at most one predecessor. We can thus identify some extra locations $i_1, \ldots, i_n$ if we impose them to admit exactly two predecessors.

As a reminder which applies to all the figures using the symbol □ to indicate a location in this section and in section 2.3, a □ symbol represents a random location that is not represented by any other □ symbol, variable (like x in figure 2.4), or integer (like i_1 in figure 2.4).

With these intuitions in mind, the property that the length of the list starting at x is equal to the length of the list starting at y can be expressed by a formula of the form below:

$$f_2 \mathrel{-\!\!*} ((f_1 \mathrel{-\!\!*} g(x,y)) \wedge (f_1 \mathrel{-\!\!*} g(y,x)))$$

where:

– f_2 expresses that all the locations have either 0 or 2 predecessors,

– f_1 expresses that all the locations have either 0 or 1 predecessor,

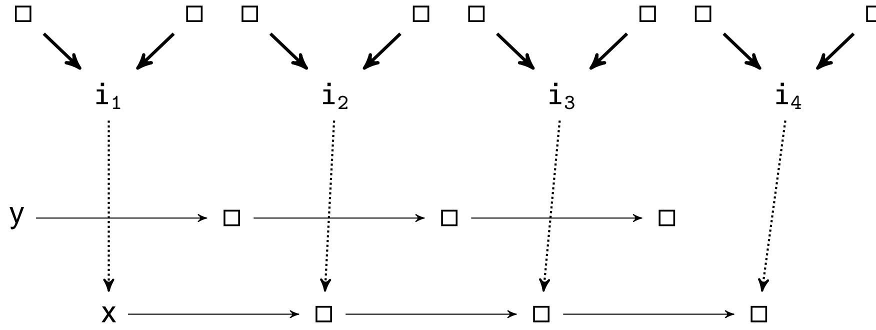


Figure 2.4: How to compare the length of two lists: situation $g(x, y)$, with heap part satisfying f_2 in bold, and heap part satisfying f_1 in dotted line

– $g(x, y)$ expresses the situation depicted in figure 2.4:

1. all the locations reachable from x have exactly two immediate predecessors, except x that has one predecessor only;
2. among the predecessors of the locations reachable from x, the ones which are not reachable from x have exactly two immediate predecessors, and these two immediate predecessors do not have immediate predecessors;
3. all extra allocated locations are only the ones of the list y.

We claim that there exist such formulas f_2, f_1, and $g(x, y)$ in SL, although we do not plan to provide details herein. We shall do it for constraints about the numbers of predecessors. Before doing so, let us first notice that it is not difficult to adapt this technique to express richer constraints on the length of two lists, as for instance the property that one list is one cell longer than another one, and thus using a reduction to counter machines similar with [25], this entails the undecidability of SL^{-*}. However, we were not able to encode SO by using cardinality constraints on list lengths, but rather on comparing the numbers of predecessors of different locations.

Let us also remark that the above construction relies on the fact that in the considered heaps, all the locations have at most one predecessor. In the general case, it could be harder to distinguish the locations that are initially allocated in the heaps, and the ones that correspond to the guessed locations $i_1, \ldots, i_n$. This last point justifies why the construction presented at the next section is a bit more technical. Actually, we shall rely on a reduction to heaps where all the locations have at least three predecessors. However, the key ideas are essentially the same.

2.2.2 Comparing the Numbers of Predecessors

In this section, we show how SL^{-*} can express properties of the form $\tilde{\sharp}x + m \; R \; \tilde{\sharp}y + m'$ with $m, m' \in \mathbb{N}$ and $R \in \{=, \geq, \leq\}$ where $\tilde{\sharp}x$ denotes the number of predecessors of $s(x)$ in a heap. This is a key property in the forthcoming proof establishing that weak second-order logic is equivalent to SL^{-*}. Note that $\tilde{\sharp}x \; R \; m$ can be easily expressed in SL^{-*}, even without magic wand (indeed m is a fixed value), as shown in section 1.4.2. By contrast, expressing a constraint

$\tilde{\sharp}x \mathrel{R} \tilde{\sharp}y + m$ is natural in second-order logic, for instance by introducing an adequate finite binary relation between the predecessors of x and those of y. We show below that this can be done also in SL^{-*} but requires much more work.

In a nutshell, expressing constraints of the form $\tilde{\sharp}x + m \mathrel{R} \tilde{\sharp}y + m'$ will be done as follows. First, thanks to boolean connectives it is sufficient to express properties of the form $\tilde{\sharp}x + m \leq \tilde{\sharp}y + m'$ with $m, m' \in \mathbb{N}$ (strictly speaking, we can assume that $m \times m' = 0$). Moreover, $\tilde{\sharp}x + m \leq \tilde{\sharp}y + m'$ is precisely equivalent to the fact that for all $n \in \mathbb{N}$, $\tilde{\sharp}y - m \leq n$ implies $\tilde{\sharp}x - m' \leq n$ (indeed $i \leq i'$ iff for every $j \geq 0$, we have $i' \leq j$ implies $i \leq j$). Quantification over the set of natural numbers will be simulated by a quantification over disjoint heaps in which n is exactly the cardinal of their domains. Such a quantification is performed thanks to the magic wand and we require that disjoint heaps are segmented and current heap is flooded (to be defined below).

Definition 2.2.2.1. A simple memory shape (s, h) is *segmented* whenever $Dom(h) \cap Im(h) = \emptyset$ and no location has strictly more than one predecessor.

(s, h) is *flooded* when no location has one or two predecessors.

The store s is irrelevant for these concepts. As an example, the heap h_2 in figure 2.6 restricted to cells labelled by 2 is segmented. These conditions on heaps are needed in order to guarantee that the heaps obtained from the original heap and the disjoint heaps easily determine which part of the heap has been added. A nice feature is that the fact of being flooded or segmented can be naturally expressed in SL^{-*} (see lemma 2.2.2.2 below). Finally, any heap such that $\tilde{\sharp}x, \tilde{\sharp}y \geq 3$ can be extended to a flooded heap without modifying the numbers of predecessors for x and y, respectively. This explains why the term 'flooded' has been chosen. In the case $\tilde{\sharp}x \leq 2$ or $\tilde{\sharp}y \leq 2$, we perform a simple case analysis and we obtain boolean combinations of constraints of the form $\tilde{\sharp}x \mathrel{R} m''$ or $\tilde{\sharp}y \mathrel{R} m''$ (that can be easily handled, details will follow).

Lemma 2.2.2.2. There are formulas flooded and seg in SL^{-*} such that for every simple memory shape (s, h),

(I) $(s, h) \models_{SL}$ flooded iff (s, h) is flooded,

(II) $(s, h) \models_{SL}$ seg iff (s, h) is segmented.

Proof. It is easy to check that the formulas below do the job.

- $\mathsf{flooded} \triangleq \forall x.\ (\sharp x = 0 \vee \sharp x > 2)$.
- $\mathsf{seg} \triangleq \forall x, y.\ (x \hookrightarrow y \Rightarrow (\sharp y = 1 \wedge \neg(\exists z.\ z \hookrightarrow x \vee y \hookrightarrow z)))$.

Note that the formulas $\sharp x = 0$, $\sharp x > 2$ and $\sharp y = 1$ are indeed formulas without separating connectives. □

Now, we present a few crucial definitions about specific patterns in simple memory shapes, namely markers.

Definition 2.2.2.3. A [resp. *strict*] *marker* in the model (s, h) is a sequence of distinct locations $i, i_0, \ldots, i_n$ for some $n \geq 0$ such that

- $h(i_0) = i$ [resp. and $Dom(h) = \{i_0, \ldots, i_n\}$],
- for every $j \in \{1, \ldots, n\}$, $h(i_i) = i_0$ and $\tilde{\sharp}i_j = 0$,

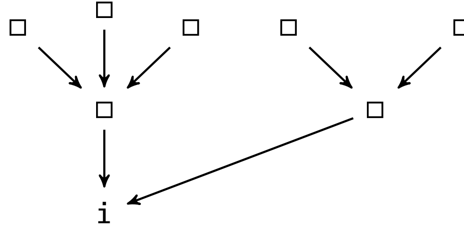


Figure 2.5: A simple memory shape with a 2-marker and a 3-marker

– $\tilde{\sharp}\mathtt{i}_0 = \mathtt{n}$.

The marker is said to be of *degree* $\mathtt{n}$ with *endpoint* $\mathtt{i}$ ($\mathtt{n}$-marker).

Markers have simple structure with natural graphical representation. In figure 2.5, we present a simple memory shape $\mathtt{h}$ containing a 2-marker and a 3-marker, both having the same endpoint $\mathtt{i}$. Note that there are disjoint heaps $\mathtt{h}_1$ and $\mathtt{h}_2$ such that $\mathtt{h} = \mathtt{h}_1 * \mathtt{h}_2$, $\mathtt{h}_1$ has a strict 2-marker and $\mathtt{h}_2$ has a strict 3-marker.

Definition 2.2.2.4. A simple memory shape $(\mathtt{s}, \mathtt{h})$ is said to be $\mathtt{k}$-*marked* whenever there is no location in $\mathtt{Dom}(\mathtt{h})$ that does not belong to a marker of degree $\mathtt{k}$. Moreover, it is *strictly* $\mathtt{k}$-marked when no distinct markers share the same endpoint (no aliasing).

Markers are essential building blocks to express a constraint of the form $\tilde{\sharp}\mathtt{x} - \mathtt{m} \leq \mathtt{n}$ with $\mathtt{m}, \mathtt{n} \in \mathbb{N}$. Before presenting the formal treatment, let us explain the principle of the encoding. Assume that $\mathtt{h}_1$ is a flooded heap (that is, no location has one or two predecessors), and $\mathtt{h}_2$ is a segmented heap such that

1. $\mathtt{h}_1$ and $\mathtt{h}_2$ are disjoint,
2. $|\mathtt{Dom}(\mathtt{h}_2)| = \mathtt{n}$,
3. $\mathtt{h}_1 * \mathtt{h}_2$ does not contain locations with two predecessors,
4. if a location $\mathtt{i}$ has exactly one predecessor $\mathtt{i}'$ in $\mathtt{h}_1 * \mathtt{h}_2$ then $\mathtt{i}'$ has no predecessor and $\mathtt{i}$ does not belong to $\mathtt{Dom}(\mathtt{h}_1 * \mathtt{h}_2)$.

Hence, $\mathtt{h}_1 * \mathtt{h}_2$ is almost flooded since the only reason for not being flooded is possibly to contain isolated memory cells from $\mathtt{h}_2$. Figure 2.6 presents two heaps $\mathtt{h}_1$ and $\mathtt{h}_2$ satisfying the above conditions. Cells of the heap $\mathtt{h}_2$ are labelled by 2. Note also that $\mathtt{h}_1 * \mathtt{h}_2$ is not flooded because of some isolated cells from $\mathtt{h}_2$ such as $\mathtt{i} \mapsto \mathtt{i}'$.

Obviously, $\mathtt{h}_1 * \mathtt{h}_2$ does not contain any 2-marker and in particular no predecessor of any location is the endpoint of some 2-marker.

Definition 2.2.2.5. A $\mathtt{m}$-*completion* of $\mathtt{h}_1 * \mathtt{h}_2$ consists in adding a disjoint heap $\mathtt{h}' = \mathtt{h}'_1 * \mathtt{h}'_2$ such that

1. $\mathtt{h}'_1$ is 1-marked,
2. $\mathtt{h}'_2$ is strictly 2-marked and contains exactly $\mathtt{m}$ distinct 2-markers.

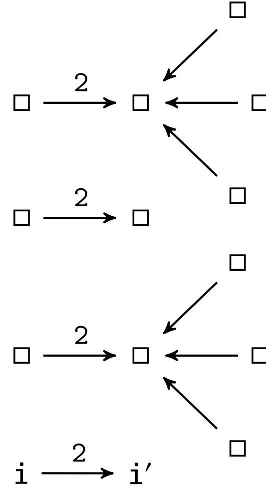


Figure 2.6: h_1 and h_2 satisfying the conditions 1.-4.

Consider the number of 2-markers in the heap $h_1 * h_2 * h'$ resulting from such a completion. First, observe that strictly more than m 2-markers can be present since an isolated memory cell from h_2 and a 1-marker from h'_1 may produce a 2-marker in $h_1 * h_2 * h'$ (see the locations i_1, i_2, i_3 and i_4 in figure 2.7). Second, observe that at least the m 2-markers from h' are still in $h_1 * h_2 * h'$, because the definition of $*$ prevents a 2-marker from combining with a 1-marker to form a 3-marker. Observe also that the insertion of markers of degree strictly less than 3 in the almost flooded heap allows to safely identify them as markers in the new heap. Consequently, there are at most $n + m$ predecessors of $s(x)$ in figure 2.7 that are endpoints of 2-markers in $h_1 * h_2 * h'$.

Definition 2.2.2.6. We say that $h_1 * h_2 * h'$ is x-*completed* whenever all the predecessors of $s(x)$ are endpoints of 2-markers.

Figure 2.7 presents a 2-completion of $h_1 * h_2$ (cells in h_1 are those pointing to x and cells in h_2 are labelled by 2 whereas the cells of the 2-completion are represented by dotted arrows). Moreover, the total resulting heap is x-completed: every predecessor of x is an endpoint of some 2-marker.

It is easy to observe that $\tilde{\sharp} x - m \leq n$ iff there is a m-completion h' of $h_1 * h_2$ such that $h_1 * h_2 * h'$ is x-completed (see the exact statement in lemma 2.2.2.9). Lemma 2.2.2.7 below states that the heaps obtained by completion can be specified in $SL^{-\!*}$.

Lemma 2.2.2.7. There are formulas $\mathsf{completed}(x)$ and $\mathsf{complete}_m$ ($m \geq 0$) in $SL^{-\!*}$ such that for every simple memory shape (s, h),

(III) $(s, h) \models_{SL} \mathsf{completed}(x)$ iff all the predecessors of $s(x)$ are endpoints of 2-markers,

(IV) $(s, h) \models_{SL} \mathsf{complete}_m$ iff there are h_1, h_2 such that $h = h_1 * h_2$, (s, h_1) is 1-marked and (s, h_2) is strictly 2-marked with exactly m distinct 2-markers.

Proof. The formulas below do the job.

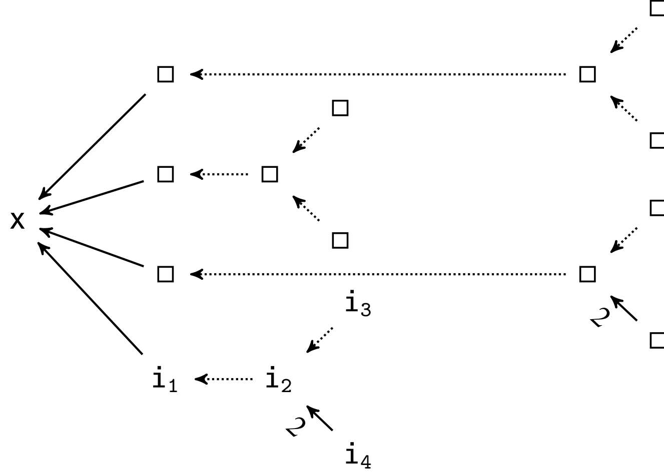


Figure 2.7: A 2-completion that leads to a $\mathtt{x}$-completed heap

(III) $\mathsf{completed}(\mathtt{x})$ is equal to:

$$\forall \mathtt{y}.\ \mathtt{y} \hookrightarrow \mathtt{x} \Rightarrow (\exists \mathtt{z}.\ \mathtt{z} \hookrightarrow \mathtt{y} \wedge \sharp \mathtt{z} = 2 \wedge \forall \mathtt{z}'.\ \mathtt{z}' \hookrightarrow \mathtt{z} \Rightarrow \sharp \mathtt{z}' = 0))$$

(IV) In order to define $\mathsf{complete}_{\mathtt{m}}$ we perform a case analysis and introduce below a few formulas. First, $\mathsf{g}_0 \triangleq \top$ and let $\mathsf{g}_{\mathtt{n}}$ be the formula below:

$$\exists \mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}, \mathtt{y}_1, \ldots, \mathtt{y}_\mathtt{n}.$$

$$(\bigwedge_{\mathtt{i} \neq \mathtt{j}} \mathtt{x}_\mathtt{i} \neq \mathtt{x}_\mathtt{j}) \wedge (\bigwedge_{\mathtt{i}=1}^{\mathtt{n}} ((\mathtt{y}_\mathtt{i} \hookrightarrow \mathtt{x}_\mathtt{i}) \wedge \sharp \mathtt{y}_\mathtt{i} = 2 \wedge \forall \mathtt{z}.\ \mathtt{z} \hookrightarrow \mathtt{y}_\mathtt{i} \Rightarrow \sharp \mathtt{z} = 0))$$

$\mathsf{g}_{\mathtt{m}} \wedge \neg \mathsf{g}_{\mathtt{m}+1}$ states that the heap contains exactly $\mathtt{m}$ 2-markers with disjoint endpoints. Let g_{cases} be the formula below:

$$\forall \mathtt{x}.\ \mathsf{alloc}(\mathtt{x}) \Rightarrow (\mathsf{g}^1_{\mathsf{extr}}(\mathtt{x}) \vee \mathsf{g}^2_{\mathsf{extr}}(\mathtt{x}) \vee \mathsf{g}^1_{\mathsf{end}}(\mathtt{x}) \vee \mathsf{g}^2_{\mathsf{end}}(\mathtt{x}))$$

where $\mathsf{g}^{\mathtt{i}}_{\mathsf{extr}}(\mathtt{x})$ [resp. $\mathsf{g}^{\mathtt{i}}_{\mathsf{end}}(\mathtt{x})$] states that $\mathtt{h}(\mathtt{s}(\mathtt{x}))$ [resp. $\mathtt{h}(\mathtt{h}(\mathtt{s}(\mathtt{x})))$] is the endpoint of some $\mathtt{i}$-marker. By way of example, $\mathsf{g}^1_{\mathsf{extr}}(\mathtt{x})$ is defined as follows:

$$\sharp \mathtt{x} = 1 \wedge (\forall \mathtt{y}.\ (\mathtt{y} \hookrightarrow \mathtt{x}) \Rightarrow \sharp \mathtt{y} = 0) \wedge (\exists \mathtt{y}.\ \mathtt{x} \hookrightarrow \mathtt{y} \wedge \neg \exists \mathtt{z}.\ \mathtt{y} \hookrightarrow \mathtt{z})$$

The formula $\mathsf{complete}_{\mathtt{m}}$ is defined as the conjunction $\mathsf{g}_{\mathtt{m}} \wedge \neg \mathsf{g}_{\mathtt{m}+1} \wedge \mathsf{g}_{cases}$.

□

Note that the heap restricted to dashed edges in figure 2.7 satisfies $\mathsf{complete}_2$ – it is composed of two 2-markers and two 1-markers.

Definition 2.2.2.8. Two heaps $\mathtt{h}_1, \mathtt{h}_2$ are said to be *completely disjoint* if $(\mathtt{Dom}(\mathtt{h}_1) \cup \mathtt{Im}(\mathtt{h}_1)) \cap (\mathtt{Dom}(\mathtt{h}_2) \cup \mathtt{Im}(\mathtt{h}_2)) = \emptyset$. Moreover, a pair of heaps $(\mathtt{h}_1, \mathtt{h}_2)$ is said to be *compatible* whenever

– (s, h_1) is flooded,

– (s, h_2) is segmented,

– h_1 and h_2 are completely disjoint.

Note that h_1 and h_2 from figure 2.6 are not compatible since $Im(h_1) \cap Im(h_2) \neq \emptyset$.

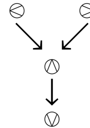
Lemma 2.2.2.9 below presents the formal statement related to the intuitive explanations that were already presented.

Lemma 2.2.2.9. Let s be a store and (h_1, h_2) be a compatible pair of heaps such that x has i predecessors in h_1 for some $i \geq 1$. Then, *(i)* $(s, h_1 \star h_2) \models_{SL} \mathsf{complete}_m \mathrel{-\!\!\circledast} \mathsf{completed}(x)$ iff *(ii)* $|Dom(h_2)| \geq (i - m)$.

Proof. Proof of (i) → (ii).
Assume (i). Let h'_1 be an 1-marked heap, h'_2 be a strict 2-marked heap with exactly m 2-markers, and $h = h_1 \star h_2 \star h'_1 \star h'_2$ with $(s, h) \models_{SL} \mathsf{completed}(x)$. Then, the set of endpoints from 2-markers in h includes $h_1^{-1}(s(x))$ and its cardinal j satisfies $j \geq i$. Markers of degree 2 witnessing the satisfaction of $\mathsf{completed}(x)$ do not come from h_1 since h_1 is flooded. So, either they come directly from h'_2 or they are markers of degree 1 which have been converted into markers of degree 2 thanks to isolated cells from h_2. Let k be the number of converted markers, then $j \leq k + m$. Since none of h_1, h'_2 contributes to the conversion of an 1-marker, the amount of converted markers is bounded by $|Dom(h_2)|$, that is $|Dom(h_2)| \geq k$. Consequently,

$$i - m \leq j - m \leq k \leq |Dom(h_2)|.$$

Proof of (ii) → (i).
Assume (ii). In the sequel, we shall introduce locations that are involved in 2-markers; the exponents below in the locations refer to the following intended positions in the schema for 2-markers (of course $\olessthan$ and $\ogreaterthan$ could have been permuted):



By letting $n_0 = i - m$, we have $|Dom(h_2)| \geq n_0$. The set of locations $h_1^{-1}(s(x))$ (set of predecessors of $s(x)$ in h_1) contains $n_0 + m$ elements that can be written $i_1^{\ovee}, \ldots, i_{n_0+m}^{\ovee}$. Since $|Dom(h_2)| = |Im(h_2)|$, there exist at least n_0 locations $i_1^{\owedge}, \ldots, i_{n_0}^{\owedge}$ in $Im(h_2)$. Moreover, since $K = Dom(h_1 \star h_2) \cup Im(h_1 \star h_2)$ is finite, there exist distinct locations $i_1^{\ogreaterthan}, \ldots, i_{n_0}^{\ogreaterthan}$ that are not in K. Let h'_1 be the heap disjoint from $(h_1 \star h_2)$ with the memory cells below:

$$h'_1 = \{i_1^{\ogreaterthan} \mapsto i_1^{\owedge}, i_1^{\owedge} \mapsto i_1^{\ovee}, \ldots, i_{n_0}^{\ogreaterthan} \mapsto i_{n_0}^{\owedge}, i_{n_0}^{\owedge} \mapsto i_{n_0}^{\ovee}\}$$

Let h'_2 be a heap disjoint from $(h_1 \star h_2 \star h'_1)$ that contains m instances of 2-markers, with endpoints $i_{n_0+1}^{\ovee}, \ldots, i_{n_0+m}^{\ovee}$ respectively. It is easy to check that $(s, h'_1 \star h'_2) \models_{SL} \mathsf{complete}_m$ and $(s, h_1 \star h_2 \star h'_1 \star h'_2) \models_{SL} \mathsf{completed}(x)$, which is sufficient to guarantee (i). □

Satisfying that for all $n \in \mathbb{N}$, $\tilde{\sharp} y - m \leq n$ implies $\tilde{\sharp} x - m' \leq n$ suggests a simple contest between two players: Spoiler aims at disproving that the constraint holds, and Duplicator tries to prove it. The whole play of the contest is depicted on figure 2.8. The steps of contest go as follows:

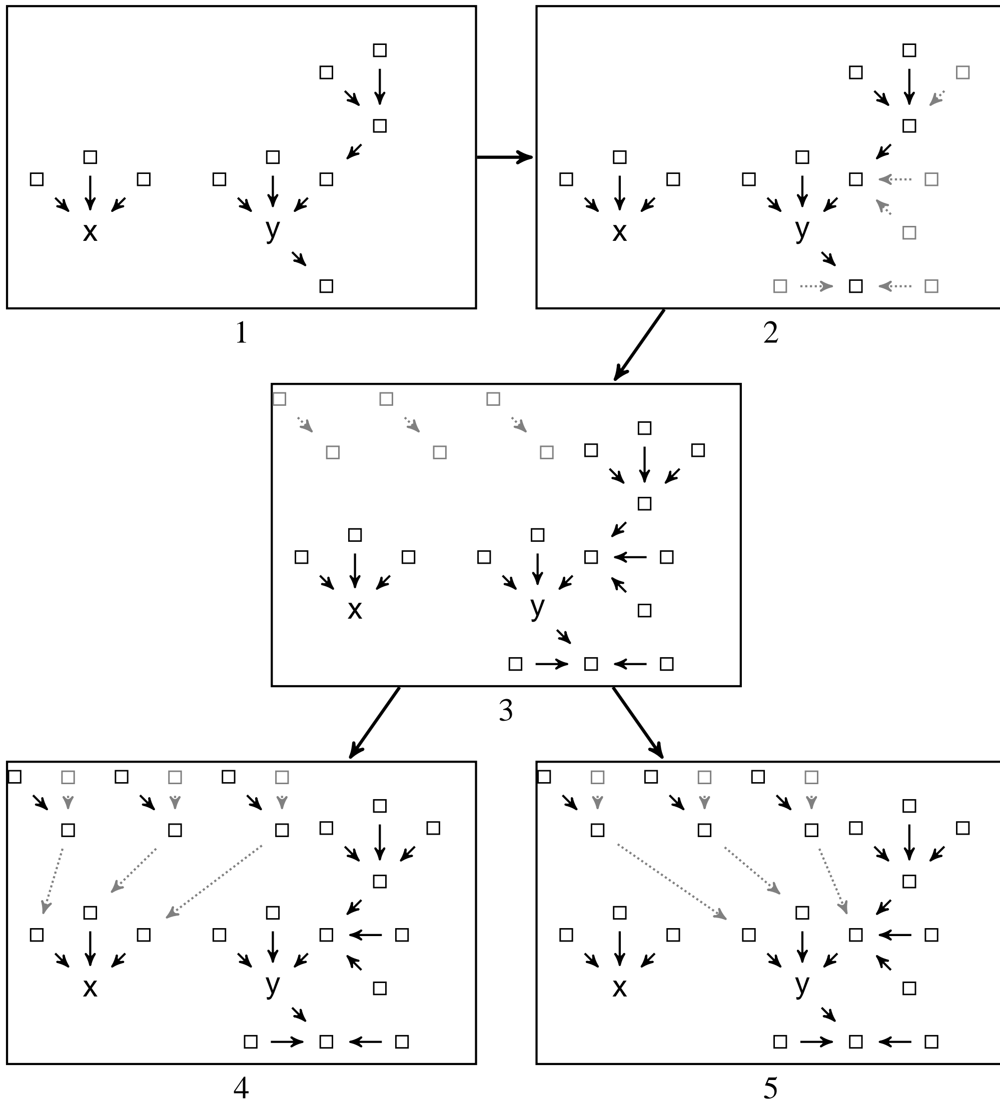


Figure 2.8: A contest won by Duplicator; $n = 3$, $m = m' = 0$

1. We start with an initial heap $\mathtt{h}_0$ without any hypothesis; if $\tilde{\sharp}\mathsf{x} \leq 2$ or $\tilde{\sharp}\mathsf{y} \leq 2$, the contest is over (these cases are handled elsewhere), otherwise the contest may start.
2. Spoiler reduces to the case of a flooded heap $\mathtt{h}_1$ (whole heap on the second frame of figure 2.8) by adding cells (the five black arrows in the second frame) in a controlled way – this will be formalized later.
3. Spoiler picks a segmented heap $\mathtt{h}_2$ (the three black arrows in the third frame) such that $|\mathtt{Dom}(\mathtt{h}_2)|$ equals $\mathtt{n}$ and $(\mathtt{h}_1, \mathtt{h}_2)$ is compatible.
4. Spoiler proves that $\tilde{\sharp}\mathsf{y} - \mathtt{m} \leq \mathtt{n}$ using the previous scenario (frame of the second line).
5. Then Duplicator plays and wins if it can prove $\tilde{\sharp}\mathsf{x} - \mathtt{m}' \leq \mathtt{n}$ (note that Duplicator wins on figure 2.8).

Figure 2.8 summarizes a contest with a successful outcome for Duplicator.

The above contest supposes that it is possible to characterize the heaps $\mathtt{h}_1 * \mathtt{h}_2$ such that $(\mathtt{h}_1, \mathtt{h}_2)$ is compatible.

We now extend little the notion of a compatible pair of heaps to a single heap. Note that a compatible heap according to the following definition is almost flooded.

Definition 2.2.2.10. A heap $\mathtt{h}$ is said to be *compatible* whenever there exist $\mathtt{h}_1$ and $\mathtt{h}_2$ such that $\mathtt{h} = \mathtt{h}_1 * \mathtt{h}_2$ and $(\mathtt{h}_1, \mathtt{h}_2)$ is compatible.

Lemma 2.2.2.11. Let $(\mathtt{s}, \mathtt{h})$ be a simple memory shape. The heap $\mathtt{h}$ is compatible iff $(\mathtt{s}, \mathtt{h}) \models_{SL}$ $\mathsf{compatible}$ with:

$$\mathsf{compatible} \triangleq (\forall \mathsf{x}, \mathsf{y}.\ (\mathsf{x} \hookrightarrow \mathsf{y} \wedge \sharp\mathsf{y} = 1) \Rightarrow (\sharp\mathsf{x} = 0 \wedge \neg\mathsf{alloc}(\mathsf{y}))) \wedge (\neg(\exists \mathsf{x}.\ \sharp\mathsf{x} = 2)).$$

The proof of lemma 2.2.2.11 is by an easy verification. It remains to define the formula $\mathsf{comtest}(\mathsf{x}, \mathsf{y}, \mathtt{m}, \mathtt{m}')$ that defines a contest and that is essential to establish lemma 2.2.2.12 below.

$$\begin{gathered}\mathsf{flooded} \wedge ((\mathsf{seg} \wedge \sharp\mathsf{x} = 0 \wedge \sharp\mathsf{y} = 0) \mathrel{-\!\!*} (\mathsf{compatible} \\ \Rightarrow ((\mathsf{complete}_{\mathtt{m}} \mathrel{\overset{\neg}{-\!\!*}} \mathsf{completed}(\mathsf{y})) \Rightarrow (\mathsf{complete}_{\mathtt{m}'} \mathrel{\overset{\neg}{-\!\!*}} \mathsf{completed}(\mathsf{x}))))).\end{gathered}$$

Lemma 2.2.2.12. For $\mathtt{m}, \mathtt{m}' \geq 0$, there is a formula f in $\mathrm{SL}^{\mathrel{-\!\!*}}$ of quadratic size in $\mathtt{m} + \mathtt{m}'$ such that for every simple memory shape $(\mathtt{s}, \mathtt{h})$, we have $(\mathtt{s}, \mathtt{h}) \models_{SL} \mathsf{f}$ iff $\tilde{\sharp}\mathsf{x} + \mathtt{m} \leq \tilde{\sharp}\mathsf{y} + \mathtt{m}'$.

Proof. By packing the previous developments, we shall show that

(PROP) When $\mathtt{h}$ is flooded, $(\mathtt{s}, \mathtt{h}) \models_{SL} \mathsf{comtest}(\mathsf{x}, \mathsf{y}, \mathtt{m}, \mathtt{m}')$ iff $\tilde{\sharp}\mathsf{x} + \mathtt{m} \leq \tilde{\sharp}\mathsf{y} + \mathtt{m}'$.

Even though $\mathtt{h}$ is not necessarily flooded, when $\tilde{\sharp}\mathsf{x} \geq 3$ and $\tilde{\sharp}\mathsf{y} \geq 3$ it can be safely extended to a flooded heap without modifying the number of predecessors of x and y. When $\tilde{\sharp}\mathsf{x} \leq 2$ or $\tilde{\sharp}\mathsf{y} \leq 2$ such an extension is not anymore possible. Nevertheless, by a simple case analysis, $\tilde{\sharp}\mathsf{x} + \mathtt{m} \leq \tilde{\sharp}\mathsf{y} + \mathtt{m}'$ is equivalent to $\bigvee_{\mathtt{i} \leq 2}(\sharp\mathsf{x} = \mathtt{i} \wedge \sharp\mathsf{y} \geq \mathtt{i} + \mathtt{m} - \mathtt{m}') \vee \bigvee_{\mathtt{i} \leq 2}(\sharp\mathsf{y} = \mathtt{i} \wedge \sharp\mathsf{x} \leq \mathtt{i} + \mathtt{m}' - \mathtt{m})$, which can be easily expressed in $\mathrm{SL}^{\mathrel{-\!\!*}}$. Let us consider $\mathsf{f} \triangleq \mathsf{f}_{\mathsf{special}} \vee \mathsf{f}_{\mathsf{main}}$ with $\mathsf{f}_{\mathsf{main}} \triangleq (\sharp\mathsf{x} = 0 \wedge \sharp\mathsf{y} = 0) \mathrel{\overset{\neg}{-\!\!*}} \mathsf{comtest}(\mathsf{x}, \mathsf{y}, \mathtt{m}, \mathtt{m}')$ and

$$\mathsf{f}_{\mathsf{special}} \triangleq \bigvee_{\mathtt{i} \leq 2}(\sharp\mathsf{x} = \mathtt{i} \ \wedge \sharp\mathsf{y} \geq \mathtt{i} + \mathtt{m} - \mathtt{m}') \vee \bigvee_{\mathtt{i} \leq 2}(\sharp\mathsf{y} = \mathtt{i} \ \wedge \sharp\mathsf{x} \leq \mathtt{i} + \mathtt{m}' - \mathtt{m})$$

First, it is clear that $\tilde{\sharp}\mathsf{x} + \mathtt{m} \leq \tilde{\sharp}\mathsf{y} + \mathtt{m}'$ and ($\tilde{\sharp}\mathsf{x} \leq 2$ or $\tilde{\sharp}\mathsf{y} \leq 2$) is equivalent to $(\mathtt{s}, \mathtt{h}) \models_{SL} \mathsf{f}_{\mathsf{special}}$. Now, suppose that $\tilde{\sharp}\mathsf{x} \geq 3$ and $\tilde{\sharp}\mathsf{y} \geq 3$. Assuming that (PROP) holds, we have the following equivalences:

(1) $(s,h) \models_{SL} (\sharp x = 0 \wedge \sharp y = 0) \multimap \mathsf{comtest}(x,y,m,m')$.

(2) There is a heap $h' \perp h$ such that $(s,h') \models_{SL} (\sharp x = 0 \wedge \sharp y = 0)$ and $(s,h*h') \models_{SL} \mathsf{comtest}(x,y,m,m')$.

(3) There is $h' \perp h$ such that $(s,h') \models_{SL} (\sharp x = 0 \wedge \sharp y = 0)$ and $(s,h*h') \models_{SL} \mathsf{flooded}$ and $\tilde{\sharp}y + m' \geq \tilde{\sharp}x + m$ (in $h*h'$) by (PROP).

(4) $\tilde{\sharp}y + m' \geq \tilde{\sharp}x + m$ in h.

Observe that $\tilde{\sharp}x$ and $\tilde{\sharp}y$ in h are equal to their values in $h*h'$ since $(s,h') \models_{SL} (\sharp x = 0 \wedge \sharp y = 0)$. Moreover, (4) implies (3) since it is always possible to extend a simple memory shape into a flooded one while preserving $\tilde{\sharp}x$ and $\tilde{\sharp}y$ (when $\tilde{\sharp}x \geq 3$ and $\tilde{\sharp}y \geq 3$).

It remains to show that (PROP) holds true. The statements below are equivalent (h is assumed to be flooded):

1. $(s,h) \models_{SL} \mathsf{comtest}(x,y,m,m')$.
2. for every segmented disjoint heap he such that $(s,he) \models_{SL} \sharp x = \sharp y = 0$, if $(s,h*he) \models_{SL} \mathsf{complete}_m \multimap \mathsf{completed}(y)$ and the heap $h*he$ is compatible, then $(s,h*he) \models_{SL} \mathsf{complete}_{m'} \multimap \mathsf{completed}(x)$.
3. for every segmented disjoint heap he such that $(s,he) \models_{SL} \sharp x = \sharp y = 0$, there exist $h'*he' = h*he$ such that (h',he') is compatible and the number of predecessors of x and y in h are equal to those of x and y in h', if $|\mathrm{Dom}(h')| \geq \tilde{\sharp}y - m$, then $|\mathrm{Dom}(h')| \geq \tilde{\sharp}x - m'$.
4. for every $n \geq 0$, we have $n \geq \tilde{\sharp}y - m$ in h implies $n \geq \tilde{\sharp}x - m'$ in h.
5. $\tilde{\sharp}x + m \leq \tilde{\sharp}y + m'$.

Lemma 2.2.2.11 is used from (1) to (2). Lemma 2.2.2.9 is used for the equivalence between (2) and (3). Moreover, one needs to observe that h is flooded, he is a disjoint segmented heap, $(s,he) \models_{SL} \sharp x = \sharp y = 0$ and $h*he$ is compatible iff there are $h'*he' = h*he$ such that (h',he') is compatible and the number of predecessors of x and y in h are equal to those of x and y in h'. Equivalence between (3) and (4) is due to the fact that for every $n \geq 0$ there is a heap he such that $|\mathrm{Dom}(he)| = n$, (h,he) is compatible and $(s,he) \models_{SL} \sharp x = \sharp y = 0$. □

In section 2.3, only constraints of the form $\tilde{\sharp}x + m \leq \tilde{\sharp}y + m'$ with $m, m' \leq 3$ are used. In particular, this means that for the forthcoming formulas using advanced arithmetical constraints, $m + m'$ can be viewed as a constant.

2.3 Equivalence to Second-Order Logic

First, by combining lemma 2.3.1.2 and lemma 2.3.1.1, we recall that DSO is at least as expressive as SL and that there is a logarithmic-space translation from SL into DSO (logarithmic space reductions are closed under compositions). Then, we will show the converse.

2.3.1 Preliminaries

Separation Logic is Less Expressive than Second-Order Logic

Here, we recall standard translations. Before showing advanced results in the sequel of this section, we show below that SO can be encoded in its fragment DSO by representing multiedges

by finite sets of edges (lemma 2.3.1.1), and then we explain how $\mathtt{SL}$ can be encoded into $\mathtt{SO}$ by simply internalizing the semantics (lemma 2.3.1.2).

Lemma 2.3.1.1. There is a logarithmic space translation from $\mathtt{SO}$ to $\mathtt{DSO}$ (hence $\mathtt{SO} \sqsubseteq \mathtt{DSO}$).

Proof. We use the standard graphical representation of a multigraph: a tuple $(i_1, \ldots, i_n)$ is represented by n edges $(i_1, i), \ldots, (i_n, i)$ for some location i. To each variable P in $\mathtt{Secvar}_n$, we associate n distinct variables $P_1, \ldots, P_n$ in $\mathtt{Secvar}_2$. Let us define the map $\mathtt{tr}_{\mathtt{SO}\to\mathtt{DSO}}$, homomorphic for boolean connectives and first-order quantification, such that $\mathtt{tr}_{\mathtt{SO}\to\mathtt{DSO}}$ preserves the semantics:

$$\mathtt{tr}_{\mathtt{SO}\to\mathtt{DSO}}(\exists P.\ g) \triangleq \exists P_1, \ldots, P_n.\ \mathtt{tr}_{\mathtt{SO}\to\mathtt{DSO}}(g)$$

$$\mathtt{tr}_{\mathtt{SO}\to\mathtt{DSO}}(P(x_1, \ldots, x_n)) \triangleq \exists y.\ \bigwedge_{i=1}^{n} P_i(x_i, y).$$

Correctness of the translation is relating on simple properties on relations. Indeed, let $R_1, \ldots, R_n$ be n finite binary relations and R be a finite n-ary relation (over $\mathtt{Loc}$). We say that $(R_1, \ldots, R_n)$ corresponds to R whenever for all $(i_1, \ldots, i_n) \in \mathtt{Loc}^n$, $(i_1, \ldots, i_n) \in R$ iff there is $i \in \mathtt{Loc}$ such that for $1 \leq k \leq n$, $(i_k, i) \in R_k$. We have the following properties:

1. For all finite binary relations $R_1, \ldots, R_n$, there is a finite n-ary relation R such that the n-uple $(R_1, \ldots, R_n)$ corresponds to R.
2. Reciprocally, for every finite n-ary relation R, there are n finite binary relations $R_1, \ldots, R_n$ such that $(R_1, \ldots, R_n)$ corresponds to R.

□

Lemma 2.3.1.2. There is a logarithmic space translation from $\mathtt{SL}$ to $\mathtt{SO}$ (hence $\mathtt{SL} \sqsubseteq \mathtt{SO}$).

Proof. For all variables P, Q, Q' in $\mathtt{Secvar}_2$, let us define the $\mathtt{SO}$ formulas below with free occurrences of P, Q, Q':

- $\mathsf{init}(P) \triangleq \forall x, y.\ xPy \Leftrightarrow x \hookrightarrow y$,
- $\mathsf{heap}(P) \triangleq \forall x, y, z.\ xPy \wedge xPz \Rightarrow y = z$ (functionality),
- $P = Q \star Q' \triangleq \forall x, y.\ (xPy \Leftrightarrow (xQy \vee xQ'y)) \wedge \neg(xQy \wedge xQ'y)$.

Let f be a formula in $\mathtt{SL}$ and P be a variable in $\mathtt{Secvar}_2$. One can show that for every simple memory shape (s, h), we have $(s, h) \models_{\mathtt{SL}} f$ iff $(s, h) \models_{\mathtt{SO}} \exists P.\ \mathsf{init}(P) \wedge \mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}(P, f)$ where $\mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}$ is inductively defined as follows ($\mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}(P, \cdot)$ is homomorphic for boolean connectives and first-order quantification):

$$\begin{aligned}
\mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}(P, x \hookrightarrow y) &\triangleq xPy \\
\mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}(P, g \star g') &\triangleq \exists Q, Q'.\ P = Q \star Q' \wedge \mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}(Q, g) \wedge \mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}(Q', g') \\
\mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}(P, g \mathrel{-\!\!\ast} g') &\triangleq \forall Q.((\exists Q'.\ \mathsf{heap}(Q') \wedge Q' = Q \star P) \wedge \mathsf{heap}(Q) \wedge \mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}(Q, g)) \\
&\quad \Rightarrow (\exists Q'.\ \mathsf{heap}(Q') \wedge Q' = Q \star P \wedge \mathtt{tr}_{\mathtt{SL}\to\mathtt{SO}}(Q', g'))
\end{aligned}$$

In the above clauses, the second-order variables Q and Q' are fresh. □

A Syntactic Convention

In the sequel, without any loss of generality, we require that the sentences in DSO satisfy the Barendregt convention as far as the second-order variables are concerned.

Definition 2.3.1.3. A sentence that contains the second-order variables $P_1, \ldots, P_n$ satisfies the *extended Barendregt convention*, if for all j, any quantification over P_j occurs within the scope of each of $P_1, \ldots, P_{j-1}$.

Typically, we exclude sentences of the form $\exists P_2. \exists P_1. f$. Observe that any sentence in DSO can be transformed in logarithmic space into an equivalent sentence verifying this convention. The *quantifier depth* of the occurrence of a subformula g in f is therefore the maximal i such that this occurrence is in the scope of $\exists P_i$; additionally by convention it is zero if it is not in the scope of any quantification.

Encoding Environments as Specific Parts of the Simple Memory Shape

Before defining the translation of a DSO sentence f, let us explain how environments can be encoded in SL. First, let us introduce some terminology.

Definition 2.3.1.4. We say that a location i is an *extremity* in a given heap if i has at least one predecessor and no predecessor of i has a predecessor.

The following formula states that $s(x)$ is an extremity:

$$\mathsf{extr}(x) \triangleq (\neg\exists y.\ (y \hookrightarrow x \wedge \exists z. z \hookrightarrow y)) \wedge (\exists y.\ y \hookrightarrow x)$$

In the particular case of a marker, an extremity is the location that points to the endpoint of the marker.

Definition 2.3.1.5. An *environment heap* is a heap containing a finite set of markers.

Environment heaps will usually be written h_e. Its markers are usually distinct from a heap h to which we want to add them; then h will be referred to as the original heap.

Environment heaps will be used to encode environments. The main idea is that a pair of locations (i, i') belongs to the interpretation of a dyadic second-order variable if i and i' are the endpoints of two markers of h_e that have respectively degrees k and $k + 1$.

Let us illustrate this idea on a simple example. Assume we want to express in SL the pure SO sentence "all finite orders have a minimal element", stated by the formula $\forall P. f_{min}(P)$, with $f_{min}(P) \triangleq$

$$\begin{pmatrix} & \forall x, y. P(x, y) \Rightarrow (P(x, x) \wedge P(y, y)) \\ \wedge & \forall x, y. (P(x, y) \wedge P(y, x)) \Rightarrow x = y \\ \wedge & \forall x, y, z. (P(x, y) \wedge P(y, z)) \Rightarrow P(x, z) \end{pmatrix} \Rightarrow \exists x. \forall y. P(y, x) \Rightarrow x = y.$$

We could actually illustrate the idea with any other SO sentence using one SO variable only, with this variable quantified in outermost position. Let $\hat{P}(x, y)$ be the SL formula

$$\hat{P}(x, y) \triangleq \exists x', y'. (x' \hookrightarrow x \wedge y' \hookrightarrow y \wedge \sharp x' + 1 = \sharp y').$$

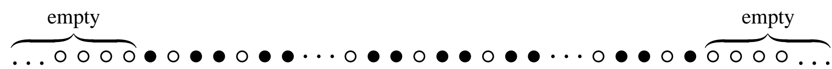
This formula expresses that x and y are the endpoints of two markers of consecutive degrees. To any heap h, we can associate the binary relation $\hat{P}_h$ composed of pairs of such locations. Conversely, any finite binary relation on locations is realized by some $\hat{P}_h$. As a consequence, the SO formula $\forall P.f_{min}(P)$ is valid if and only if the SL formula $\mathsf{emp} \Rightarrow (\top \mathrel{-\!\!*} f_{min}(\hat{P}))$ is valid as well – note that it is also equivalent to $f_{min}(\hat{P})$ being valid, but we want to underline the idea that one SO quantification can be encoded by one application of $\mathrel{-\!\!*}$.

The generalization of this encoding to arbitrary formulas raises several problems. The first problem is to distinguish the environment heap from the original one. As a remark, in the example above, this is solved by restricting ourselves to an original empty heap, but this is not possible in general. In the previous section, we solved this issue by first extending the original heap to a flooded heap, and then by using markers of small degrees (one or two) that were clearly distinct from the original heap. The same approach is not possible here, because one may need arbitrarily large degrees. Transforming an original heap into a flooded one in a controlled way is possible for counting the number of predecessors (see section 2.2), but it might be much more difficult if the property of interest is not just a property on the number of predecessors, but an arbitrary second-order property. For all these reasons, we adopt a different strategy, and we ensure that the degree of a marker in h_e is strictly greater than the maximal number of predecessors of any location from the original heap. Nonetheless, our investigation on counting the number of predecessors is precious (see section 2.2), and will be used when expressing that two endpoints i, i' are consecutively marked.

The second problem is, given a pair (i, i') of locations marked by markers of consecutive degrees, to determine the second-order variable P_j whose interpretation contains (i, i'). In the example above, we only had one second-order variable P, but we may not reduce to the case of a unique second-order variable in general). To do so, we impose some more structure on h_e. First, for any natural number n, there is at most one extremity with degree n in h_e.

Definition 2.3.1.6. The *spectrum* of h_e is the finite set of natural numbers n for which there is a marker of degree n in h_e. A *clean* spectrum is additionally a set of natural numbers of the form $\{n \mid n_0 \le n \le n_1 \text{ and } n \not\equiv n_0 + 1 \pmod 3\}$ for some $n_0, n_1 \in \mathbb{N}$.

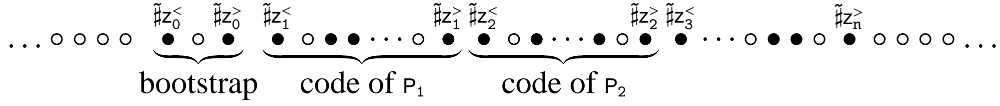
Second, we require that the spectrum of h_e, depicted as a marking of the sequence of naturals, has the following shape, which corresponds to the definition of a clean spectrum:



A symbol '•' on position n indicates the presence of a marker of degree n, and '∘' its absence. This simple and regular structure makes the characterization of well-formed environment heaps easier at every step of the translation (in particular, every time the environment is extended by a new quantified second-order variable). In order to identify markers that are attached to a given second-order variable,

1. we ensure that the markers of a given second-order variable follow each others in a given interval,
2. these intervals do not overlap for two distinct second-order variables,
3. there is no unused space between these intervals.

This is achieved by introducing, for each $\mathsf{P}_{\mathtt{j}}$, two variables $\mathsf{z}_{\mathtt{j}}^{<}$ and $\mathsf{z}_{\mathtt{j}}^{>}$ that are placed on the upper and lower bound of the interval of the interpretation of $\mathsf{P}_{\mathtt{j}}$. For technical reasons mainly related to bootstrapping, we shall also consider the two distinguished variables $\mathsf{z}_{0}^{<}$ and $\mathsf{z}_{0}^{>}$. So, the spectrum of $\mathtt{h}_{\mathtt{e}}$ can be graphically depicted as



2.3.2 Encoding Environments

First, let us show how to express structural properties about the environment heap. Thanks to lemma 2.2.2.12, advanced arithmetical constraints are expressed in the proof of lemma 2.3.2.1 below.

Lemma 2.3.2.1. There is a formula $\mathsf{psenvir}(\mathsf{z},\mathsf{z}')$ in $\mathtt{SL}^{-\!*}$ such that the conditions below hold true iff $(\mathtt{s},\mathtt{h}_{\mathtt{e}}) \models_{\mathrm{SL}} \mathsf{psenvir}(\mathsf{z},\mathsf{z}')$:

- $\tilde{\sharp}\mathsf{z} < \tilde{\sharp}\mathsf{z}'$, $\tilde{\sharp}\mathsf{z} \equiv \tilde{\sharp}\mathsf{z}' + 2 \mod 3$ and z and z' are extremities.
- for all $\mathtt{i}$ in $[\tilde{\sharp}\mathsf{z},\ldots,\tilde{\sharp}\mathsf{z}']$,
    - ∗ if $\mathtt{i} \equiv \tilde{\sharp}\mathsf{z} + 1 \pmod 3$ then there is no extremity $\mathtt{j}$ in $(\mathtt{s},\mathtt{h}_{\mathtt{e}})$ such that $\tilde{\sharp}\mathtt{j} = \mathtt{i}$,
    - ∗ if $\mathtt{i} \not\equiv \tilde{\sharp}\mathsf{z}+1 \pmod 3$, then there is exactly one location $\mathtt{j}$ such that $\mathtt{j}$ is an extremity and $\tilde{\sharp}\mathtt{j} = \mathtt{i}$. This unique location $\mathtt{j}$ belongs to $\mathtt{Dom}(\mathtt{h}_{\mathtt{e}})$.

Proof. The formula $\mathsf{psenvir}(\mathsf{z},\mathsf{z}')$ is the conjunction of the formulas below expressing the following properties:

1. $\tilde{\sharp}\mathsf{z} < \tilde{\sharp}\mathsf{z}'$ and z,z′ are extremities: $\sharp\mathsf{z} < \sharp\mathsf{z}' \wedge \mathsf{extr}(\mathsf{z}) \wedge \mathsf{extr}(\mathsf{z}')$.
2. There is no extremity whose number of predecessors is equal to either $\tilde{\sharp}\mathsf{z} + 1$ or $\tilde{\sharp}\mathsf{z}' - 1$.
$$(\neg\exists \mathsf{x}.\ \mathsf{extr}(\mathsf{x}) \wedge \sharp\mathsf{z} + 1 = \sharp\mathsf{x}) \wedge (\neg\exists \mathsf{x}.\ \mathsf{extr}(\mathsf{x}) \wedge \sharp\mathsf{z}' = 1 + \sharp\mathsf{x})$$
3. There is an extremity whose number of predecessors is equal to $\tilde{\sharp}\mathsf{z} + 2$ [resp. $\tilde{\sharp}\mathsf{z}' - 2$].
$$\exists \mathsf{x}.\ \mathsf{extr}(\mathsf{x}) \wedge \sharp\mathsf{z} + 2 = \sharp\mathsf{x} \wedge \exists \mathsf{x}.\ \mathsf{extr}(\mathsf{x}) \wedge \sharp\mathsf{z}' = 2 + \sharp\mathsf{x}$$
4. For every extremity x whose number of predecessors is strictly between $\tilde{\sharp}\mathsf{z}$ and $\tilde{\sharp}\mathsf{z}'$, there is an extremity whose number of predecessors is equal to either $\tilde{\sharp}\mathsf{x} + 1$ or $\tilde{\sharp}\mathsf{x} - 1$.
$$\forall \mathsf{x}.\ [\mathsf{extr}(\mathsf{x}) \wedge \sharp\mathsf{x} > \sharp\mathsf{z} \wedge \sharp\mathsf{x} < \sharp\mathsf{z}'] \Rightarrow (\exists \mathsf{y}.\ \sharp\mathsf{y} = 1 + \sharp\mathsf{x} \vee \exists \mathsf{y}.\ \sharp\mathsf{y} + 1 = \sharp\mathsf{x})$$
5. Constraint on two extremities whose numbers of predecessors are consecutive:
$$\forall \mathsf{x}.\forall \mathsf{y}.\ [\mathsf{extr}(\mathsf{x}) \wedge \mathsf{extr}(\mathsf{y}) \wedge (\sharp\mathsf{x} > \sharp\mathsf{z}) \wedge (\sharp\mathsf{x} < \sharp\mathsf{z}') \wedge (\sharp\mathsf{y} > \sharp\mathsf{z}) \wedge$$
$$(\sharp\mathsf{y} < \sharp\mathsf{z}') \wedge (\sharp\mathsf{y} + 1 = \sharp\mathsf{x})] \Rightarrow$$
$$[(\neg\exists \mathsf{y}'.\ \sharp\mathsf{y}' = 1 + \sharp\mathsf{x}) \wedge (\exists \mathsf{y}'.\ \sharp\mathsf{y}' = 2 + \sharp\mathsf{x}) \wedge$$
$$(\neg\exists \mathsf{y}'.\ \sharp\mathsf{y}' + 1 = \sharp\mathsf{y}) \wedge (\exists \mathsf{y}'.\sharp\mathsf{y}' + 2 = \sharp\mathsf{y})]$$

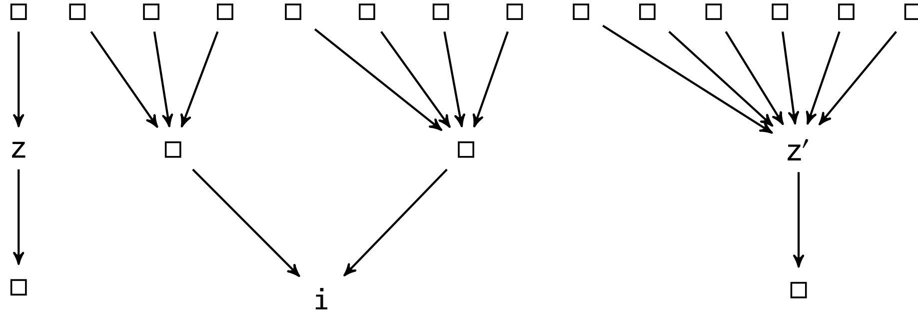


Figure 2.9: A simple environment encoding the pair $(\mathtt{i},\mathtt{i})$

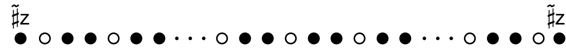
6. There are no two distinct extremities with an equal number of predecessors.

$$\forall x\, [\mathsf{extr}(x) \wedge \sharp x \geq \sharp z \wedge \sharp x \leq \sharp z'] \Rightarrow \neg\exists y.\ (\mathsf{extr}(y) \wedge \sharp x = \sharp y \wedge x \neq y)$$

It is then easy to check that the above conjunction satisfies the statement.

By induction on k ranging from 1 to $(\tilde{\sharp}z' - \tilde{\sharp}z - 2)/3$, one can show that there is no extremity $\mathtt{i}$ in (s, h_e) such that $\tilde{\sharp}\mathtt{i} = \tilde{\sharp}z + 3k - 2$, and there are extremities $\mathtt{i}$ and $\mathtt{i}'$ such that $\tilde{\sharp}\mathtt{i} = \tilde{\sharp}z + 3k - 1$ and $\tilde{\sharp}\mathtt{i} = \tilde{\sharp}z + 3k$. This concludes the proof. □

Consequently, if $(s, h_e) \models_{SL} \mathsf{psenvir}(z, z')$, then h_e has a clean spectrum:



Definition 2.3.2.2. The simple memory shape (s, h_e) is called a *pseudo-environment* between z and z' if $(s, h_e) \models_{SL} \mathsf{psenvir}(z, z')$

Definition 2.3.2.3. An *environment between* z *and* z' is a simple memory shape (s, h_e) such that

(P1) $(s, h_e) \models_{SL} \mathsf{psenvir}(z, z')$.

(P2) If $\mathtt{i} \in \mathrm{Dom}(h_e)$, then either $\mathtt{i}$ or $h_e(\mathtt{i})$ is an extremity in h_e.

(P3) For every extremity $\mathtt{i}$ in h_e, $\mathtt{i} \in \mathrm{Dom}(h_e)$ and $h_e(\mathtt{i}) \notin \mathrm{Dom}(h_e)$.

(P4) For every extremity $\mathtt{i}$ in h_e, $\tilde{\sharp}z \leq \tilde{\sharp}\mathtt{i} \leq \tilde{\sharp}z'$.

Roughly speaking, (s, h_e) is a finite set of markers with the above-mentioned spectrum. Figure 2.9 presents a simple environment with $\tilde{\sharp}z = 1$ and $\tilde{\sharp}z' = 6$, which allows to encode a single pair ($(\mathtt{i},\mathtt{i})$ in the present figure). Note that in full generality, the number of pairs that can be encoded by an environment between z and z' is equal to $(\tilde{\sharp}z' - \tilde{\sharp}z - 2)/3$.

Lemma 2.3.2.4. There exists a formula $\mathsf{env}(z, z') \in \mathrm{SL}^{\rightarrow\!*}$ such that for every simple memory shape (s, h), we have $(s, h) \models_{SL} \mathsf{env}(z, z')$ iff (s, h) is an environment between z and z'.

Proof. Let us consider the conjunction $\mathsf{env}(z, z')$ of the formulas below.

(F1) $\mathsf{psenvir}(\mathtt{z},\mathtt{z}')$.

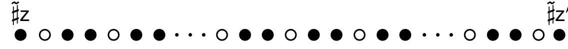
(F2) $\forall \mathtt{x}.\ (\mathsf{alloc}(\mathtt{x}) \Rightarrow (\mathsf{extr}(\mathtt{x}) \vee \exists \mathtt{y}.\ \mathtt{x} \hookrightarrow \mathtt{y} \wedge \mathsf{extr}(\mathtt{y})))$.

(F3) $\forall \mathtt{x}.\ \mathsf{extr}(\mathtt{x}) \Rightarrow (\mathsf{alloc}(\mathtt{x}) \wedge \exists \mathtt{y}.\ \mathtt{x} \hookrightarrow \mathtt{y} \wedge \neg\mathsf{alloc}(\mathtt{y}))$.

(F4) $\forall \mathtt{x}.\ \mathsf{extr}(\mathtt{x}) \Rightarrow (\sharp \mathtt{z} \leq \sharp \mathtt{x} \wedge \sharp \mathtt{x} \leq \sharp \mathtt{z}')$.

Each formula (F*i*) captures the condition (P*i*). □

Consequently, if $(\mathtt{s},\mathtt{h_e}) \models_{\mathrm{SL}} \mathsf{env}(\mathtt{z},\mathtt{z}')$, then $\mathtt{h_e}$ is equal to a set of markers of the clean spectrum



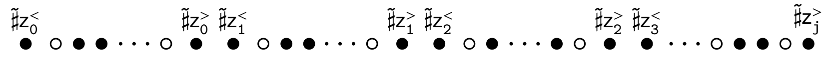
Definition 2.3.2.5. A $\mathtt{j}$-*marked environment* is a simple memory shape $(\mathtt{s},\mathtt{h})$ such that

(PM0) $(\mathtt{s},\mathtt{h})$ is an environment between $\mathtt{z}_0^<$ and $\mathtt{z}_\mathtt{j}^>$.

(PM1) For every variable $\mathtt{x}$ in $\{\mathtt{z}_1^<,\ldots,\mathtt{z}_\mathtt{j}^<\} \cup \{\mathtt{z}_0^>,\ldots,\mathtt{z}_{\mathtt{j}-1}^>\}$, $\mathtt{s}(\mathtt{x})$ is an extremity in $(\mathtt{s},\mathtt{h})$ and $\tilde{\sharp}\mathtt{z}_0^< < \tilde{\sharp}\mathtt{x} < \tilde{\sharp}\mathtt{z}_\mathtt{j}^>$.

(PM2) For $\mathtt{j} \geq \mathtt{i} > 0$, $\tilde{\sharp}\mathtt{z}_{\mathtt{i}-1}^> + 1 = \tilde{\sharp}\mathtt{z}_\mathtt{i}^<$.

Consequently, when $(\mathtt{s},\mathtt{h})$ is a $\mathtt{j}$-marked environment, the spectrum of $\mathtt{h_e}$ contains the following values:



Moreover, if $(\mathtt{s},\mathtt{h}')$ is another $\mathtt{j}$-marked environment with identical store, then $\mathtt{h}$ and $\mathtt{h}'$ have the same spectrum.

Definition 2.3.2.6 below specifies how a heap can be divided into a base part and an environment part with constraints on the values $\tilde{\sharp}\mathtt{z}_0^<,\tilde{\sharp}\mathtt{z}_0^>,\ \ldots,\ \tilde{\sharp}\mathtt{z}_\mathtt{j}^<,\tilde{\sharp}\mathtt{z}_\mathtt{j}^>$. These values are helpful to determine the range of marker degrees that should be considered to encode the interpretation of second-order variables.

Definition 2.3.2.6. A simple memory shape $(\mathtt{s},\mathtt{h})$ is $\mathtt{j}$-*well-formed* for some $\mathtt{j} \geq 0$ iff there are heaps $\mathtt{h_b},\mathtt{h_e}$ with $\mathtt{h} = \mathtt{h_b} * \mathtt{h_e}$ satisfying the properties below:

(WF1) $(\mathtt{s},\mathtt{h_e})$ is a $\mathtt{j}$-marked environment.

(WF2) There is no location $\mathtt{i}$ such that $\tilde{\sharp}\mathtt{i}$ in $(\mathtt{s},\mathtt{h_b})$ is strictly greater than $\tilde{\sharp}\mathtt{z}_0^< - 2$ in $(\mathtt{s},\mathtt{h})$.

(WF3) $\mathtt{Dom}(\mathtt{h_e}) \cap \mathtt{Im}(\mathtt{h_b}) = \emptyset$.

$(\mathtt{s},\mathtt{h_b})$ is called the *base part* and $(\mathtt{s},\mathtt{h_e})$ the *environment part*.

Condition (WF3) guarantees that when $(\mathtt{s},\mathtt{h})$ is $\mathtt{j}$-well-formed, for every extremity $\mathtt{i}$ in $\mathtt{h_e}$, $\tilde{\sharp}\mathtt{i}$ in $\mathtt{h_e}$ is equal to $\tilde{\sharp}\mathtt{i}$ in $\mathtt{h}$. Consequently, any extremity in $\mathtt{h}$ with more than $\tilde{\sharp}\mathtt{z}_0^<$ predecessors has all predecessors in $\mathtt{Dom}(\mathtt{h_e})$. Moreover, $(\mathtt{s},\mathtt{h}) \models_{\mathrm{SL}} \mathsf{psenvir}(\mathtt{z}_0^<,\mathtt{z}_\mathtt{j}^>)$, that is $(\mathtt{s},\mathtt{h})$ is a pseudo-environment between $\mathtt{z}_0^<$ and $\mathtt{z}_\mathtt{j}^>$.

We establish below a few lemmas that are helpful in the sequel.

Lemma 2.3.2.7. Let $\mathtt{h_e}$ be the environment part of some $\mathtt{j}$-well-formed simple memory shape. For every location $\mathtt{i} \in \mathtt{Im}(\mathtt{h_e})$, either $\mathtt{i}$ is an extremity in $\mathtt{h_e}$ or there is $\mathtt{i}'$ such that $\mathtt{h_e}(\mathtt{i}') = \mathtt{i}$ and $\mathtt{i}'$ is an extremity.

Note that the above property holds true for any environment but we shall use it for $\mathtt{j}$-well-formed simple memory shapes only.

Proof. If $\mathtt{i} \in \mathtt{Im}(\mathtt{h_e})$, then there is a location $\mathtt{i}'$ such that $\mathtt{h_e}(\mathtt{i}') = \mathtt{i}$. By (P2) on $(\mathtt{s}, \mathtt{h_e})$, either $\mathtt{h_e}(\mathtt{i}')$ is an extremity or $\mathtt{i}'$ is an extremity. □

Lemma 2.3.2.8 below states unicity of decomposition when a simple memory shape is $\mathtt{j}$-well-formed.

Lemma 2.3.2.8 (Unicity)**.** Whenever $(\mathtt{s}, \mathtt{h})$ is $\mathtt{j}$-well-formed with base part $\mathtt{h_b}$ and environment part $\mathtt{h_e}$, there is no $(\mathtt{h}'_\mathtt{b}, \mathtt{h}'_\mathtt{e}) \neq (\mathtt{h_b}, \mathtt{h_e})$ such that $(\mathtt{s}, \mathtt{h})$ is $\mathtt{j}$-well-formed with base part $\mathtt{h}'_\mathtt{b}$ and environment part $\mathtt{h}'_\mathtt{e}$.

Proof. Let $\mathtt{k}_0 = (\tilde{\sharp}\mathtt{z}_\mathtt{j}^> - \tilde{\sharp}\mathtt{z}_0^< - 2)/3$ and $\mathbb{K} = \{\mathtt{k} : \mathtt{k} \not\equiv 1 \pmod 3 \text{ and } 0 \leq \mathtt{k} \leq 3 \times \mathtt{k}_0 + 2\}$ be the spectrum of $\mathtt{h_e}$ and $\mathtt{h}'_\mathtt{e}$. Indeed, $(\mathtt{s}, \mathtt{h_e})$ and $(\mathtt{s}, \mathtt{h}'_\mathtt{e})$ are both $\mathtt{j}$-marked environments and there are precisely $|\mathbb{K}|$ extremities $\mathtt{i}$ in $(\mathtt{s}, \mathtt{h})$ such that $\tilde{\sharp}\mathtt{z}_0^< \leq \tilde{\sharp}\mathtt{i} \leq \tilde{\sharp}\mathtt{z}_\mathtt{j}^>$. For each $\mathtt{k} \in \mathbb{K}$, we write $\mathtt{i_k}$ to denote the unique extremity such that $\tilde{\sharp}\mathtt{i_k} = \tilde{\sharp}\mathtt{z}_0^< + \mathtt{k}$. Notice that each location $\mathtt{i_k}$ has no predecessor in $\mathtt{h_b}$ by definition 2.3.2.6(WF3), $\mathtt{i}_0 = \mathtt{s}(\mathtt{z}_0^<)$ and $\mathtt{i}_{3\mathtt{k}_0+2} = \mathtt{s}(\mathtt{z}_\mathtt{j}^>)$.

The set $\mathtt{Dom}(\mathtt{h_e})$ contains at least the following locations: for every $\mathtt{k} \in \mathbb{K}$, the location $\mathtt{i_k}$ and the $\tilde{\sharp}\mathtt{z}_0^< + \mathtt{k}$ predecessors of $\mathtt{i_k}$ in $\mathtt{h}$. Let $\mathtt{I}_1$ be the set of the above locations. Assume there is some $\mathtt{i} \in (\mathtt{Dom}(\mathtt{h_e}) \setminus \mathtt{I}_1)$. By (P2), either $\mathtt{i}$ or $\mathtt{h_e}(\mathtt{i})$ is an extremity in $\mathtt{h_e}$ (let us call it $\mathtt{i}'$). Since each predecessor of some location in $\mathtt{I}_1$ is also in $\mathtt{I}_1$ and $\mathtt{i} \notin \mathtt{I}_1$, $\mathtt{i}$ is not a predecessor of an element in $\mathtt{I}_1$. Consequently, $\mathtt{i}'$ is an extremity that does not belong to $\{\mathtt{i_k} : \mathtt{k} \in \mathbb{K}\}$ (let us call this set $\mathtt{I}_2$). Since $\tilde{\sharp}\mathtt{z}_0^< \leq \tilde{\sharp}\mathtt{i}' \leq \tilde{\sharp}\mathtt{z}_\mathtt{j}^>$, either $\mathtt{i}'$ has as many predecessors as an element in $\mathtt{I}_2$ or $\tilde{\sharp}\mathtt{i}' \equiv \tilde{\sharp}\mathtt{z}_0^< + 1 \pmod 3$. This entails that $(\mathtt{s}, \mathtt{h_e})$ does not satisfy $\mathsf{psenvir}(\mathtt{z}_0^<, \mathtt{z}_\mathtt{j}^>)$ which leads to a contradiction. Consequently, $\mathtt{Dom}(\mathtt{h_e}) = \mathtt{I}_1$, $\mathtt{h_e} = \mathtt{h}_{|\mathtt{I}_1}$ (restriction of $\mathtt{h}$ to $\mathtt{I}_1$) and $\mathtt{h_b} = \mathtt{h}_{|(\mathtt{Dom}(\mathtt{h}) \setminus \mathtt{I}_1)}$. □

In the sequel, when $(\mathtt{s}, \mathtt{h})$ is $\mathtt{j}$-well-formed, by default $\mathtt{h_e}$ denotes the environment part and $\mathtt{h_b}$ the base part.

We state below a crucial result, basically stating that adding an environment heap to a $\mathtt{j}$-well-formed simple memory shape leads to a $(\mathtt{j}+1)$-well-formed simple memory shape. This is central to interpret a new second-order variable (extending the environment part) and this can be performed thanks to $\multimap$ (details will follow).

Lemma 2.3.2.9 (Composition)**.** Let $(\mathtt{s}, \mathtt{h})$ be a $\mathtt{j}$-well-formed simple memory shape and $(\mathtt{s}', \mathtt{h}'_\mathtt{e})$ be a simple memory shape such that

1. $\mathtt{h}'_\mathtt{e}$ is disjoint from $\mathtt{h}$ and $\mathtt{s}'$ differs from $\mathtt{s}$ at most for the variables $\mathtt{z}_{\mathtt{j}+1}^<$ and $\mathtt{z}_{\mathtt{j}+1}^>$.
2. $\mathtt{s}'(\mathtt{z}_{\mathtt{j}+1}^<)$ and $\mathtt{s}'(\mathtt{z}_{\mathtt{j}+1}^>)$ do not belong to $\mathtt{Dom}(\mathtt{h}) \cup \mathtt{Im}(\mathtt{h})$.
3. $(\mathtt{s}', \mathtt{h}'_\mathtt{e})$ is an environment between $\mathtt{z}_{\mathtt{j}+1}^<$ and $\mathtt{z}_{\mathtt{j}+1}^>$.
4. $(\mathtt{s}', \mathtt{h} * \mathtt{h}'_\mathtt{e}) \models_{\mathrm{SL}} \sharp\mathtt{z}_\mathtt{j}^> + 1 = \sharp\mathtt{z}_{\mathtt{j}+1}^<$.
5. $\mathtt{Dom}(\mathtt{h}'_\mathtt{e}) \cap \mathtt{Im}(\mathtt{h}) = \emptyset$.

Then, $(\mathtt{s}', \mathtt{h} * \mathtt{h}'_\mathtt{e})$ is $(\mathtt{j}+1)$-well-formed with the base part $\mathtt{h_b}$ and the environment part $\mathtt{h_e} * \mathtt{h}'_\mathtt{e}$.

The proof of lemma 2.3.2.9 is tedious and requires some care. We provide the details below.

Proof. We will refer to an item of the lemma by the word *hypothesis*. For instance, hypothesis (5) is: $\mathtt{Dom}(\mathtt{h}'_e) \cap \mathtt{Im}(\mathtt{h}) = \emptyset$.

The proof mainly rests on establishing the property below.

(PROP1) Any extremity in $\mathtt{h_e}$ or in $\mathtt{h}'_e$ is an extremity in $\mathtt{h} * \mathtt{h}'_e$ with exactly the same number of predecessors.

Consequently, this implies that in the simple memory shape $(\mathtt{s}', \mathtt{h} * \mathtt{h}'_e)$ we have the following relationships:

(PROP2) $\tilde{\sharp}\mathtt{z}_0^< < \tilde{\sharp}\mathtt{z}_j^> = \tilde{\sharp}\mathtt{z}_{j+1}^< - 1 < \tilde{\sharp}\mathtt{z}_{j+1}^> - 1$, $\tilde{\sharp}\mathtt{z}_0^< + 2 \equiv \tilde{\sharp}\mathtt{z}_{j+1}^> \ (\mathtt{mod}\ 3)$ and $\tilde{\sharp}\mathtt{z}_{j+1}^< \equiv \tilde{\sharp}\mathtt{z}_0^< \ (\mathtt{mod}\ 3)$.

Assuming (PROP1) and (PROP2), let us check the conditions from definition 2.3.2.6 for ensuring that $(\mathtt{s}', \mathtt{h} * \mathtt{h}'_e)$ is $(\mathtt{j}+1)$-well-formed with base part $\mathtt{h_b}$. After doing that, we shall establish that (PROP1) holds true.

First, we show that $(\mathtt{s}', \mathtt{h_e} * \mathtt{h}'_e)$ is a $(\mathtt{j}+1)$-marked environment.

(P1) Let us prove that $(\mathtt{s}', \mathtt{h_e} * \mathtt{h}'_e) \models_{SL} \mathsf{psenvir}(\mathtt{z}_0^<, \mathtt{z}_{j+1}^>)$. Below, the numbers of predecessors are relative to $(\mathtt{s}', \mathtt{h_e} * \mathtt{h}'_e)$. Let $\mathtt{i} \in \{\tilde{\sharp}\mathtt{z}_0^<, \ldots, \tilde{\sharp}\mathtt{z}_{j+1}^>\}$.

  * Assume $\mathtt{i} \equiv \tilde{\sharp}\mathtt{z}_0^< + 1 \ (\mathtt{mod}\ 3)$. By contradiction, suppose that there is a location $\mathtt{i}'$ such that $\mathtt{i}'$ is an extremity and $\tilde{\sharp}\mathtt{i}' = \mathtt{i}$. Then $\mathtt{i}'$ is an extremity with $\mathtt{i}$ predecessors either in $\mathtt{h_e}$ or in $\mathtt{h}'_e$, which leads to a contradiction since $(\mathtt{s}', \mathtt{h}'_e)$ is an environment between $\mathtt{z}_{j+1}^<$ and $\mathtt{z}_{j+1}^>$ and $(\mathtt{s}, \mathtt{h_e})$ is an environment between $\mathtt{z}_0^<$ and $\mathtt{z}_j^>$.
  * Assume $\mathtt{i} \not\equiv \tilde{\sharp}\mathtt{z}_0^< + 1 \ (\mathtt{mod}\ 3)$. If $\mathtt{i} \in \{\tilde{\sharp}\mathtt{z}_0^<, \ldots, \tilde{\sharp}\mathtt{z}_j^>\}$, then by (PROP1) there is a unique extremity $\mathtt{i}_k$ such that $\tilde{\sharp}\mathtt{i}_k = \mathtt{i}$. Otherwise ($\mathtt{i} \in \{\tilde{\sharp}\mathtt{z}_{j+1}^<, \ldots, \tilde{\sharp}\mathtt{z}_{j+1}^>\}$), by (PROP1), there is a unique extremity $\mathtt{i}_k^{new}$ such that $\tilde{\sharp}\mathtt{i}_k^{new} = \mathtt{i}$.

(P2) Suppose that $\mathtt{i} \in \mathtt{Dom}(\mathtt{h_e} * \mathtt{h}'_e)$. Two cases are distinguished below.

  * $\mathtt{i} \in \mathtt{Dom}(\mathtt{h_e})$.
    We distinguish again two subcases since $\mathtt{h}$ is $\mathtt{j}$-well-formed.
    · In the case $\mathtt{i}$ is an extremity in $\mathtt{h_e}$, the location $\mathtt{i}$ is an extremity in $\mathtt{h} * \mathtt{h}'_e$ by (PROP1). Consequently, $\mathtt{i}$ is an extremity in $\mathtt{h_e} * \mathtt{h}'$.
    · In the case $\mathtt{h}(\mathtt{i})$ is an extremity in $\mathtt{h_e}$, the proof is analogous.
  * $\mathtt{i} \in \mathtt{Dom}(\mathtt{h}'_e)$.
    The proof is analogous.

(P3) Let $\mathtt{i}$ be an extremity in $\mathtt{h_e}$. Let us show that $\mathtt{h}(\mathtt{i}) \notin \mathtt{Dom}(\mathtt{h_e} * \mathtt{h}'_e)$. Since $(\mathtt{s}, \mathtt{h})$ is $\mathtt{j}$-well-formed, $\mathtt{h}(\mathtt{i}) \notin \mathtt{Dom}(\mathtt{h_e})$. By contradiction, suppose that $\mathtt{h}(\mathtt{i}) \in \mathtt{Dom}(\mathtt{h}'_e)$. Then, either $\mathtt{h}(\mathtt{i})$ is an extremity in $\mathtt{h}'_e$ or $\mathtt{h}(\mathtt{i})$ is a predecessor of an extremity $\mathtt{i}'$ in $\mathtt{h}'_e$. In the first case, it leads to a contradiction since the extremities of $\mathtt{h}'_e$ are not in $\mathtt{Im}(\mathtt{h_e})$, by hypothesis (5). In the second case, $\mathtt{i}'$ is not an extremity in $\mathtt{h} * \mathtt{h}'_e$ which is in contradiction with (PROP1). Consequently, $\mathtt{h}(\mathtt{i}) \notin \mathtt{Dom}(\mathtt{h_e} * \mathtt{h}'_e)$.

  Let $\mathtt{i}$ be an extremity in $\mathtt{h}'_e$. Since $(\mathtt{s}', \mathtt{h}'_e)$ is an environment between $\mathtt{z}_{j+1}^<$ and $\mathtt{z}_{j+1}^>$, we know that $\mathtt{h}'_e(\mathtt{i}) \notin \mathtt{Dom}(\mathtt{h}'_e)$. It remains to check that $\mathtt{h}'_e(\mathtt{i}) \notin \mathtt{Dom}(\mathtt{h_e})$. By contradiction, suppose that $\mathtt{h}'_e(\mathtt{i}) \in \mathtt{Dom}(\mathtt{h_e})$. Then there is $\mathtt{i}' \in \{\mathtt{h}'_e(\mathtt{i}), \mathtt{h}(\mathtt{h}'_e(\mathtt{i}))\}$ such that $\mathtt{i}'$ is an extremity in $\mathtt{h_e}$. By (PROP1), $\mathtt{i}'$ is an extremity in $\mathtt{h_e} * \mathtt{h}'_e$. This leads to a contradiction since $\mathtt{i}$ has predecessors in $\mathtt{h_e} * \mathtt{h}'_e$.

(P4) Let $\mathtt{i}$ be an extremity in $\mathtt{h_e}$. We have $\tilde{\sharp}\mathtt{z}_0^< \leq \tilde{\sharp}\mathtt{i} \leq \tilde{\sharp}\mathtt{z}_j^> < \tilde{\sharp}\mathtt{z}_{j+1}^>$ since $(\mathtt{s},\mathtt{h})$ is $\mathtt{j}$-well-formed and (PROP1). Let $\mathtt{i}$ be an extremity in $\mathtt{h}'_e$. The values $\tilde{\sharp}\mathtt{i}$, $\tilde{\sharp}\mathtt{z}_{j+1}^<$ and $\tilde{\sharp}\mathtt{z}_{j+1}^>$ do not change from $\mathtt{h}'_e$ to $\mathtt{h}*\mathtt{h}'_e$. Since $(\mathtt{s}',\mathtt{h}'_e)$ is an environment between $\mathtt{z}_{j+1}^<$ and $\mathtt{z}_{j+1}^>$, $\tilde{\sharp}\mathtt{z}_{j+1}^< \leq \tilde{\sharp}\mathtt{i} \leq \tilde{\sharp}\mathtt{z}_{j+1}^>$. So in $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$, we have $\tilde{\sharp}\mathtt{z}_0^< < \tilde{\sharp}\mathtt{z}_{j+1}^< \leq \tilde{\sharp}\mathtt{i} \leq \tilde{\sharp}\mathtt{z}_{j+1}^>$.

(PM1) By (PROP1), for each variable $\mathtt{x}$ in $\{\mathtt{z}_0^<,\ldots,\mathtt{z}_{j+1}^<\} \cup \{\mathtt{z}_0^>,\ldots,\mathtt{z}_{j+1}^>\}$, the value $\tilde{\sharp}\mathtt{x}$ remains unchanged from $\mathtt{h}$ or $\mathtt{h}'_e$ to $\mathtt{h}*\mathtt{h}'_e$. Considering that $\mathtt{h}$ is $\mathtt{j}$-well-formed, $\mathtt{h}'_e$ is an environment between $\mathtt{z}_{j+1}^<$ and $\mathtt{z}_{j+1}^>$ and $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e) \models_{SL} \sharp\mathtt{z}_j^> + 1 = \sharp\mathtt{z}_{j+1}^<$, we conclude that for every $\mathtt{x} \in \{\mathtt{z}_1^<,\ldots,\mathtt{z}_{j+1}^<\} \cup \{\mathtt{z}_0^>,\ldots,\mathtt{z}_j^>\}$, $\mathtt{s}(\mathtt{x})$ is an extremity and $\tilde{\sharp}\mathtt{z}_0^< < \tilde{\sharp}\mathtt{x} < \tilde{\sharp}\mathtt{z}_{j+1}^>$.

(PM2) Let $0 < \mathtt{i} \leq \mathtt{j}+1$. If $\mathtt{i} \leq \mathtt{j}$, then since $(\mathtt{s},\mathtt{h})$ is $\mathtt{j}$-well-formed we obtain $(\mathtt{s},\mathtt{h}) \models_{SL} \tilde{\sharp}\mathtt{z}_i^> + 1 = \tilde{\sharp}\mathtt{z}_{i+1}^<$. By (PROP1), $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e) \models_{SL} \tilde{\sharp}\mathtt{z}_i^> + 1 = \tilde{\sharp}\mathtt{z}_{i+1}^<$ ($\mathtt{s}'$ and $\mathtt{s}$ agree for these variables). If $\mathtt{i} = \mathtt{j}+1$, then hypothesis (4) precisely states that $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e) \models_{SL} \sharp\mathtt{z}_j^> + 1 = \sharp\mathtt{z}_{j+1}^<$.

It remains to verify the conditions (WF2) and (WF3).

(WF2) Since $\mathtt{h}$ and $\mathtt{h}*\mathtt{h}'_e$ have the same base part and $(\mathtt{s},\mathtt{h})$ is $\mathtt{j}$-well-formed, we get that there is no location $\mathtt{i}$ such that $\tilde{\sharp}\mathtt{i}$ in $(\mathtt{s},\mathtt{h_b})$ is strictly greater than $\tilde{\sharp}\mathtt{z}_0^< - 2$ in $(\mathtt{s},\mathtt{h}*\mathtt{h_e})$ (equal to $\tilde{\sharp}\mathtt{z}_0^< - 2$ in $(\mathtt{s},\mathtt{h})$ by (PROP1)).

(WF3) Since $(\mathtt{s},\mathtt{h})$ is $\mathtt{j}$-well-formed, we have $\mathtt{Dom}(\mathtt{h_e}) \cap \mathtt{Im}(\mathtt{h_b}) = \emptyset$. By hypothesis (5), $\mathtt{Dom}(\mathtt{h}'_e) \cap \mathtt{Im}(\mathtt{h}) = \emptyset$. Consequently, $\mathtt{Dom}(\mathtt{h_e}*\mathtt{h}'_e) \cap \mathtt{Im}(\mathtt{h_b}) = \emptyset$.

Now, let us prove that (PROP1) holds true. First, we prove the case when an extremity is a location of the form $\mathtt{s}'(\mathtt{z}_k^\diamond)$ with $\mathtt{k} \in \{0,\ldots,\mathtt{j}+1\}$ and $\diamond \in \{<,>\}$. By hypothesis (2), $\mathtt{s}'(\mathtt{z}_{j+1}^<)$ and $\mathtt{s}'(\mathtt{z}_{j+1}^>)$ do not belong to $\mathtt{Im}(\mathtt{h})$. So the values $\tilde{\sharp}\mathtt{z}_{j+1}^<$ and $\tilde{\sharp}\mathtt{z}_{j+1}^>$ remain unchanged from $(\mathtt{s}',\mathtt{h}'_e)$ to $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$. Now let $\mathtt{k} \in \{0,\ldots,\mathtt{j}\}$ and $\diamond \in \{<,>\}$. Assume that $\tilde{\sharp}\mathtt{z}_k^\diamond$ has changed from $(\mathtt{s}',\mathtt{h})$ to $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$. Consequently, $\mathtt{s}'(\mathtt{z}_k^\diamond) \in \mathtt{Im}(\mathtt{h}'_e)$. By lemma 2.3.2.7, there are two possibilities.

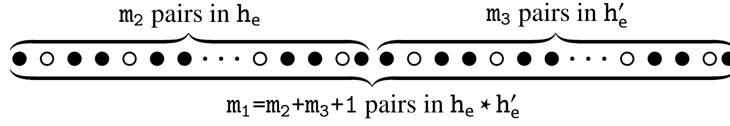
1. $\mathtt{s}'(\mathtt{z}_k^\diamond)$ is an extremity in $(\mathtt{s}',\mathtt{h}'_e)$.
   As $(\mathtt{s}',\mathtt{h}'_e)$ is an environment between $\mathtt{z}_{j+1}^<$ and $\mathtt{z}_{j+1}^>$, every extremity belongs to $\mathtt{Dom}(\mathtt{h}'_e)$, whence $\mathtt{s}'(\mathtt{z}_k^\diamond) \in \mathtt{Dom}(\mathtt{h}'_e)$. This leads to a contradiction since $\mathtt{h}$ and $\mathtt{h}'_e$ are disjoint: $\mathtt{s}'(\mathtt{z}_k^\diamond) \in \mathtt{Dom}(\mathtt{h_e})$ since $(\mathtt{s},\mathtt{h})$ is $\mathtt{j}$-well-formed.
2. There is a location $\mathtt{i}$ such that $\mathtt{h}'_e(\mathtt{i}) = \mathtt{s}'(\mathtt{z}_k^\diamond)$ (also equal to $\mathtt{s}(\mathtt{z}_k^\diamond)$) and $\mathtt{i}$ is an extremity. So $\mathtt{s}'(\mathtt{z}_k^\diamond)$ is not an extremity in $\mathtt{h}*\mathtt{h}'_e$, which also leads to a contradiction.

Consequently, for all $\mathtt{k} \in \{0,\ldots,\mathtt{j}\}$ and $\diamond \in \{<,>\}$, $\tilde{\sharp}\mathtt{z}_k^\diamond$ is unchanged from $\mathtt{h}$ to $\mathtt{h}*\mathtt{h}'_e$. Based on these preservations and since $(\mathtt{s}',\mathtt{h}'_e)$ is an environment between $\mathtt{z}_{j+1}^<$ and $\mathtt{z}_{j+1}^>$, $(\mathtt{s}',\mathtt{h})$ is $\mathtt{j}$-well-formed and $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e) \models_{SL} \sharp\mathtt{z}_j^> + 1 = \sharp\mathtt{z}_{j+1}^<$, we can conclude (PROP2).

Before treating the proof for other types of extremities, let us provide a few basic definitions and facts. We define the natural numbers $\mathtt{m}_1$, $\mathtt{m}_2$ and $\mathtt{m}_3$ as follows:

$$3\mathtt{m}_1 = (\tilde{\sharp}\mathtt{z}_{j+1}^> - \tilde{\sharp}\mathtt{z}_0^<) - 2 \quad 3\mathtt{m}_2 = (\tilde{\sharp}\mathtt{z}_j^> - \tilde{\sharp}\mathtt{z}_0^<) - 2 \quad 3\mathtt{m}_3 = (\tilde{\sharp}\mathtt{z}_{j+1}^> - \tilde{\sharp}\mathtt{z}_{j+1}^<) - 2$$

Notice that $\mathtt{m}_3 = \mathtt{m}_1 - \mathtt{m}_2 - 1$. These values are simply related to the spectrum below where the first value is $\tilde{\sharp}\mathtt{z}_0^<$ and the last one is $\tilde{\sharp}\mathtt{z}_{j+1}^>$.



For $n \geq 1$, let $K_n \triangleq \{k : k \not\equiv 2 \pmod 3 \text{ and } 0 \leq k \leq 3n+2\}$ and we pose $J_{m_1} \triangleq K_{m_1}$, $J_{m_2} \triangleq K_{m_2}$ and $J_{m_3} \triangleq \{n + (3m_2 + 3) : n \in K_{m_3}\}$. Since (s, h_e) is an environment between $z_0^{<}$ and $z_j^{>}$ (remember (s, h) is j-well-formed) and (s', h'_e) is an environment between $z_{j+1}^{<}$ and $z_{j+1}^{>}$, we get that $(s', h * h'_e) \models_{SL} \mathsf{psenvir}(z_0^{<}, z_{j+1}^{>})$. So, for every $k \in J_{m_1}$, there is a location $i_k^{\star}$ verifying the properties below in $(s', h * h'_e)$:

- $\tilde{\sharp} i_k^{\star} = \tilde{\sharp} z_0^{<} + k$,
- $i_k^{\star}$ is an extremity,
- there is no location i such that $\tilde{\sharp} i = \tilde{\sharp} i_k^{\star}$, $i \neq i_k^{\star}$ and i is an extremity.

Notice that $\tilde{\sharp} i_{3\times m_1}^{\star} = \tilde{\sharp} z_{j+1}^{>} - 2$ in $(s', h * h'_e)$, $i_{3\times m_2+2}^{\star} = s'(z_j^{>})$ and $i_{3\times m_2+3}^{\star} = s'(z_{j+1}^{<})$.

Similarly, as $(s', h) \models_{SL} \mathsf{psenvir}(z_0^{<}, z_j^{>})$, for every $k \in J_{m_2}$, there is a location i_k verifying the properties below in (s', h):

- $\tilde{\sharp} i_k = \tilde{\sharp} z_0^{<} + k$,
- i_k is an extremity,
- there is no location i such that $\tilde{\sharp} i = \tilde{\sharp} i_k$, $i \neq i_k$ and i is an extremity.

Observe that all the extremities in h_e are either of the form i_k, or $s'(z_0^{<})$ or $s'(z_j^{>})$. Moreover, $\tilde{\sharp} i_{3m_2} = \tilde{\sharp} z_j^{>} - 2$ in (s', h).

Finally, as $(s', h'_e) \models_{SL} \mathsf{psenvir}(z_{j+1}^{<}, z_{j+1}^{>})$, for every $k \in J_{m_3}$, there is a location i_k^{new} verifying the properties below in (s', h'_e):

- $\tilde{\sharp} i_k^{new} = (\tilde{\sharp} z_{j+1}^{<} - (3m_2 + 3)) + k$,
- i_k^{new} is an extremity,
- there is no location i such that $\tilde{\sharp} i = \tilde{\sharp} i_k^{new}$, $i \neq i_k^{new}$ and i is an extremity.

Observe that all the extremities of h'_e are either of the form i_k^{new}, or $s'(z_{j+1}^{<})$ or $s'(z_{j+1}^{>})$. We can establish additional arithmetical properties: $\tilde{\sharp} i_{3m_3}^{new} = \tilde{\sharp} z_{j+1}^{>} - 2$ in (s', h'_e) and $\tilde{\sharp} i_k^{new}$ in (s', h'_e) is equal to $\tilde{\sharp} z_0^{<} + k$ in $(s', h * h'_e)$.

We are going to prove that for all $k \in J_{m_2}$, $i_k = i_k^{\star}$, and for all $k \in J_{m_3}$, $i_k^{new} = i_k^{\star}$. This will terminate the proof of (PROP1) since the only extremities in h_e are $\{i_k : k \in J_{m_2}\} \cup \{s'(z_0^{<}), s'(z_j^{>})\}$ and the only extremities in h'_e are $\{i_k^{new} : k \in J_{m_3}\} \cup \{s'(z_{j+1}^{<}), s'(z_{j+1}^{>})\}$. The proof is by contradiction and we distinguish two cases (each of them will lead to a contradiction):

(case one) There is $k \in J_{m_2}$ such that $i_k \neq i_k^{\star}$.

(case two) There is $k \in J_{m_3}$ such that $i_k^{new} \neq i_k^{\star}$.

(case one) Let us first establish that $\mathtt{i}^\star_k \in \mathtt{Im}(\mathtt{h}'_e)$ (proof by contradiction). Suppose that $\mathtt{i}^\star_k \notin \mathtt{Im}(\mathtt{h}'_e)$. So, $\tilde{\sharp}\mathtt{i}^\star_k$ remains unchanged from $(\mathtt{s}',\mathtt{h})$ to $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$. As in $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$, we have $\tilde{\sharp}\mathtt{z}^<_0 < \tilde{\sharp}\mathtt{i}^\star_k < \tilde{\sharp}\mathtt{z}^>_j$, and $\mathtt{z}^<_0$ and $\mathtt{z}^>_j$ remain unchanged from $(\mathtt{s}',\mathtt{h})$ to $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$, we can infer that $\tilde{\sharp}\mathtt{z}^<_0 < \tilde{\sharp}\mathtt{i}^\star_k < \tilde{\sharp}\mathtt{z}^>_j$ in $(\mathtt{s}',\mathtt{h})$. Additionally, as in $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$, we have $\tilde{\sharp}\mathtt{i}^\star_k = \tilde{\sharp}\mathtt{z}^<_0 + \mathtt{k}$, this is also true in $(\mathtt{s}',\mathtt{h})$. Finally, as $\mathtt{i}^\star_k$ is an extremity in $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$, it is also an extremity in $(\mathtt{s}',\mathtt{h})$. Consequently, $\mathtt{i}^\star_k = \mathtt{i}_k$, which leads to a contradiction. We have established that $\mathtt{i}^\star_k \in \mathtt{Im}(\mathtt{h}'_e)$. By lemma 2.3.2.7, there are two possibilities:

- $\mathtt{i}^\star_k$ is an extremity in $\mathtt{h}'_e$.
  Consequently, in $\mathtt{h}'_e$, we have $\tilde{\sharp}\mathtt{i}^\star_k > \tilde{\sharp}\mathtt{z}^<_{j+1}$. As $\tilde{\sharp}\mathtt{z}^<_{j+1}$ remains unchanged from $(\mathtt{s}',\mathtt{h}'_e)$ to $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$, in $\mathtt{h}*\mathtt{h}'_e$ we obtain $\tilde{\sharp}\mathtt{i}^\star_k > \tilde{\sharp}\mathtt{z}^<_{j+1} = \tilde{\sharp}\mathtt{z}^>_j + 1$, which leads to a contradiction.
- There is a location $\mathtt{i}_0$ such that $\mathtt{i}_0$ is an extremity in $\mathtt{h}'_e$ and $\mathtt{h}'_e(\mathtt{i}_0) = \mathtt{i}^\star_k$. So $\mathtt{i}^\star_k$ is not an extremity in $\mathtt{h}'_e$, and it cannot either be an extremity in $\mathtt{h}*\mathtt{h}'_e$, which leads to a contradiction.

(case two) Let $\mathtt{k}$ be the smallest element of $\mathtt{J}_{m_3}$ such that $\mathtt{i}^{new}_k \neq \mathtt{i}^\star_k$. In $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$, we know that $\tilde{\sharp}\mathtt{i}^\star_k > \tilde{\sharp}\mathtt{z}^<_{j+1} > \tilde{\sharp}\mathtt{z}^>_j$. Moreover, as $\mathtt{i}^\star_k$ is an extremity in $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$, either $\mathtt{i}^\star_k$ is an extremity in $(\mathtt{s}',\mathtt{h})$ too or $\mathtt{i}^\star_k$ has no predecessor in $(\mathtt{s}',\mathtt{h})$. Since no extremity of $(\mathtt{s}',\mathtt{h})$ has more than $\tilde{\sharp}\mathtt{z}^>_j$ predecessors (in both $\mathtt{h}$ and $\mathtt{h}*\mathtt{h}'_e$), the location $\mathtt{i}^\star_k$ cannot have all of its predecessors in $\mathtt{Dom}(\mathtt{h})$. Let $\mathtt{i}_0$ be one of the predecessors of $\mathtt{i}^\star_k$ that belongs to $\mathtt{Dom}(\mathtt{h}'_e)$, that is $\mathtt{h}'_e(\mathtt{i}_0) = \mathtt{i}^\star_k$.

Let us recall that $(\mathtt{s}',\mathtt{h}'_e)$ is an environment between $\mathtt{z}^<_{j+1}$ and $\mathtt{z}^>_{j+1}$. Since $\mathtt{i}_0 \in \mathtt{Dom}(\mathtt{h}'_e)$, there is $\mathtt{i} \in \{\mathtt{i}_0, \mathtt{h}'_e(\mathtt{i}_0)\}$ such that in $\mathtt{h}'_e$:

(a) $\mathtt{i}$ is an extremity,

(b) $\mathtt{i} \in \mathtt{Dom}(\mathtt{h}'_e)$,

(c) $\tilde{\sharp}\mathtt{z}^<_{j+1} \leq \tilde{\sharp}\mathtt{i} \leq \tilde{\sharp}\mathtt{z}^>_{j+1}$,

(d) no other extremity has exactly $\tilde{\sharp}\mathtt{i}$ predecessors.

Indeed, the condition (a) comes from (P2), the conditions (b) and (d) both come from the $(\mathtt{s}',\mathtt{h}'_e) \models_{SL} \mathsf{psenvir}(\mathtt{z}^<_{j+1}, \mathtt{z}^>_{j+1})$, and (c) from satisfaction of (P4).

In the case $\mathtt{i} = \mathtt{i}_0$, $\mathtt{i}^\star_k$ is not an extremity in $\mathtt{h}'_e$ and hence $\mathtt{i}^\star_k$ is not an extremity in $\mathtt{h}*\mathtt{h}'_e$. This leads to a contradiction. Consequently, we have $\mathtt{i} = \mathtt{h}'_e(\mathtt{i}_0) = \mathtt{i}^\star_k$. Let us conclude the proof.

In $\mathtt{h}'_e$, the location $\mathtt{i}^\star_k$ is an extremity. As $(\mathtt{s}',\mathtt{h}'_e)$ is an environment between $\mathtt{z}^<_{j+1}$ and $\mathtt{z}^>_{j+1}$, we have $\mathtt{i}^\star_k \in \mathtt{Dom}(\mathtt{h}'_e)$ and $\tilde{\sharp}\mathtt{z}^<_{j+1} \leq \tilde{\sharp}\mathtt{i}^\star_k \leq \tilde{\sharp}\mathtt{z}^>_{j+1}$ in $(\mathtt{s}',\mathtt{h}'_e)$. Since $\mathtt{s}(\mathtt{z}^<_{j+1}) \neq \mathtt{i}^\star_k$ and $\mathtt{s}(\mathtt{z}^>_{j+1}) \neq \mathtt{i}^\star_k$, we obtain $\tilde{\sharp}\mathtt{z}^<_{j+1} < \tilde{\sharp}\mathtt{i}^\star_k < \tilde{\sharp}\mathtt{z}^>_{j+1}$ in $\mathtt{h}'$.

So there is $\mathtt{k}_0 \in \mathtt{J}_{m_3}$ such that $\mathtt{i}^\star_k = \mathtt{i}^{new}_{k_0}$. We have that the value $\tilde{\sharp}\mathtt{i}^{new}_{k_0}$ changes from $\mathtt{h}'_e$ to $\mathtt{h}*\mathtt{h}'_e$, and therefore $\mathtt{i}^{new}_{k_0} \neq \mathtt{i}^\star_{k_0}$. Since $\tilde{\sharp}\mathtt{i}^{new}_{k_0}$ can only increase from $\mathtt{h}'_e$ to $\mathtt{h}*\mathtt{h}'_e$, we can conclude that $\tilde{\sharp}\mathtt{i}^{new}_{k_0}$ in $(\mathtt{s}',\mathtt{h}'_e)$ is strictly smaller than $\tilde{\sharp}\mathtt{i}^{new}_{k_0} = \tilde{\sharp}\mathtt{i}^\star_k$ in $(\mathtt{s}',\mathtt{h}*\mathtt{h}'_e)$. By definition of the locations $\mathtt{i}^\star_k$ and $\mathtt{i}^{new}_{k_0}$, we obtain $\mathtt{k}_0 < \mathtt{k}$, which leads to a contradiction by minimality of $\mathtt{k}$. □

2.3.3 The Translation

In this section, we provide the translation from $\mathtt{DSO}$ into $\mathtt{SL}^{\twoheadrightarrow}$. First, we introduce additional formulas that will be useful in the translation process. It is worth observing that in order to translate first-order quantification, we should guarantee that first-order variables $\mathtt{x}$ are not interpreted as locations from the domain of the environment part. Typically, the number of predecessors of $\mathtt{s(x)}$ and $\mathtt{h(s(x))}$ (if it exists) should be less than $\tilde{\sharp}\mathtt{z}_0^>$ and none of these locations is an extremity. The formula $\mathsf{notonmark}(\cdot)$ is introduced for this purpose:

$$\mathsf{notonmark}(\mathtt{x}) \triangleq \neg(\exists \mathtt{y}.\ (\mathtt{y} = \mathtt{x} \vee \mathtt{x} \hookrightarrow \mathtt{y}) \wedge (\sharp \mathtt{y} \geq \sharp \mathtt{z}_0^<) \wedge \mathsf{extr}(\mathtt{y})).$$

Lemma 2.3.3.1. Assume $(\mathtt{s},\mathtt{h})$ is a $\mathtt{j}$-well-formed simple memory shape. Then $(\mathtt{s},\mathtt{h}) \models_{SL} \mathsf{notonmark}(\mathtt{x})$ iff $\mathtt{s(x)} \notin \mathtt{Dom}(\mathtt{h_e})$.

Proof. As $(\mathtt{s},\mathtt{h})$ is $\mathtt{j}$-well-formed, by definition 2.3.2.6, for any location $\mathtt{i}$, we have $\mathtt{i} \in \mathtt{Dom}(\mathtt{h_e})$ iff there is a location $\mathtt{i}' \in \{\mathtt{i},\mathtt{h(i)}\}$ such that in the heap $\mathtt{h_e}$, we have $\tilde{\sharp}\mathtt{i}' \geq \tilde{\sharp}\mathtt{z}_0^<$ and $\mathtt{i}'$ is an extremity. Moreover, by definition 2.3.2.6, we get in the heap $\mathtt{h}$ that $\tilde{\sharp}\mathtt{i}' \geq \tilde{\sharp}\mathtt{z}_0^<$ and $\mathtt{i}'$ is an extremity. Assume that $\mathtt{s(x)} \in \mathtt{Dom}(\mathtt{h_e})$, then thanks to the explanations just above, $(\mathtt{s},\mathtt{h}) \not\models_{SL} \mathsf{notonmark}(\mathtt{x})$.
Now, by contradiction, suppose that $\mathtt{s(x)} \notin \mathtt{Dom}(\mathtt{h_e})$ and $(\mathtt{s},\mathtt{h}) \not\models_{SL} \mathsf{notonmark}(\mathtt{x})$. Then there is $\mathtt{i} \in \{\mathtt{s(x)},\mathtt{h(s(x))}\}$ such that $\tilde{\sharp}\mathtt{i} \geq \tilde{\sharp}\mathtt{z}_0^<$ and $\mathtt{i}$ is an extremity, by definition of $\mathsf{notonmark}$. Furthermore, by definition 2.3.2.6(WF3), the location $\mathtt{i}$ is not an extremity in $\mathtt{h_e}$, all of its predecessors are in $\mathtt{h_b}$. Then by definition 2.3.2.6, $\tilde{\sharp}\mathtt{i} \geq \tilde{\sharp}\mathtt{z}_0^< - 2$, which leads to a contradiction. ◻

The formula $\mathsf{relation}_{\mathtt{j},\mathtt{X}}$ defined below is helpful to build environments.

Lemma 2.3.3.2. Let $\mathtt{j} \geq 0$ and $\mathtt{X}$ be a finite set of variables disjoint from $\{\mathtt{z}_0^<,\mathtt{z}_0^>,\ldots,\mathtt{z}_\mathtt{j}^<,\mathtt{z}_\mathtt{j}^>\}$. Then, there is a formula $\mathsf{relation}_{\mathtt{j},\mathtt{X}}$ such that for every simple memory shape $(\mathtt{s},\mathtt{h})$, we have $(\mathtt{s},\mathtt{h}) \models_{SL} \mathsf{relation}_{\mathtt{j},\mathtt{X}}$ iff $(\mathtt{s},\mathtt{h})$ is an environment between $\mathtt{z}_\mathtt{j}^<$ and $\mathtt{z}_\mathtt{j}^>$ and for every $\mathtt{x} \in \mathtt{X}$, $\mathtt{s(x)} \notin \mathtt{Dom}(\mathtt{h})$.

The formula $\mathsf{relation}_{\mathtt{j},\mathtt{X}}$ is simply

$$\mathsf{relation}_{\mathtt{j},\mathtt{X}} \triangleq \mathsf{env}(\mathtt{z}_\mathtt{j}^<,\mathtt{z}_\mathtt{j}^>) \wedge \bigwedge_{\mathtt{y}\in\mathtt{X}} \neg\mathsf{alloc}(\mathtt{y}).$$

We will additionally need the formula $\mathsf{isol}(\mathtt{x})$, which means that $\mathtt{s(x)} \notin \mathtt{Dom}(\mathtt{h}) \cup \mathtt{Im}(\mathtt{h})$. It is defined as:

$$\mathsf{isol}(\mathtt{x}) \triangleq \neg\exists \mathtt{y}.\ (\mathtt{x} \hookrightarrow \mathtt{y}) \vee (\mathtt{y} \hookrightarrow \mathtt{x})$$

The translation of the formula f, written $\mathtt{translation}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathsf{f})$, is defined with the help of the translation $\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j},\cdot)$ where $\mathtt{j}$ records the quantifier depth. The translation is defined so that $(\mathtt{s(x)},\mathtt{s(y)})$ belongs to the interpretation of $\mathsf{P}_\mathtt{i}$ when $\mathtt{s(x)}$ and $\mathtt{s(y)}$ are end-points of markers with consecutive degrees between $\tilde{\sharp}\mathtt{z}_\mathtt{i}^<$ and $\tilde{\sharp}\mathtt{z}_\mathtt{i}^>$.

$$\begin{array}{c}
\mathtt{translation}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathsf{f}) \triangleq \exists \mathtt{z}_0^<\ \mathtt{z}_0^>.\ \mathsf{isol}(\mathtt{z}_0^>) \wedge \mathsf{isol}(\mathtt{z}_0^<) \wedge \\
[((\forall \mathtt{x}.\ \mathsf{alloc}(\mathtt{x}) \Rightarrow (\mathtt{x} \hookrightarrow \mathtt{z}_0^> \vee \mathtt{x} \hookrightarrow \mathtt{z}_0^< \vee \mathtt{x} = \mathtt{z}_0^> \vee \mathtt{x} = \mathtt{z}_0^<)) \wedge \mathsf{alloc}(\mathtt{z}_0^>) \wedge \mathsf{alloc}(\mathtt{z}_0^<)) \twoheadrightarrow \\
(\forall \mathtt{x}.\mathtt{x} \neq \mathtt{z}_0^> \wedge \mathtt{x} \neq \mathtt{z}_0^< \Rightarrow (\sharp \mathtt{z}_0^< > 2 + \sharp \mathtt{x})) \wedge (\sharp \mathtt{z}_0^> = 2 + \sharp \mathtt{z}_0^<) \wedge \mathsf{extr}(\mathtt{z}_0^<) \wedge \mathsf{extr}(\mathtt{z}_0^>) \wedge \\
\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(0,\mathsf{f}))]
\end{array}$$

In order to define recursively the map $\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}$, note that if $\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \mathsf{P}_\mathtt{i}(\mathsf{x},\mathsf{y}))$ occurs then $\mathtt{i} \geq \mathtt{j}$, and that by the extended Barendregt convention if $\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \exists \mathsf{P}_\mathtt{i}.\mathsf{g})$ occurs then $\mathtt{i} = \mathtt{j} + 1$. Also, $\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \cdot)$ is homomorphic for boolean connectives.

$$
\begin{array}{lll}
\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \mathsf{x} = \mathsf{y}) & \triangleq & \mathsf{x} = \mathsf{y} \\
\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \mathsf{x} \hookrightarrow \mathsf{y}) & \triangleq & \mathsf{x} \hookrightarrow \mathsf{y} \\
\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \mathsf{P}_\mathtt{i}(\mathsf{x},\mathsf{y})) & \triangleq & \exists \mathsf{z}, \mathsf{z}'.\ (\mathsf{z} \hookrightarrow \mathsf{x}) \wedge (\mathsf{z}' \hookrightarrow \mathsf{y}) \wedge (\sharp \mathsf{z} > \sharp \mathsf{z}^{<}_{\mathtt{i}}) \wedge (\sharp \mathsf{z}' < \sharp \mathsf{z}^{>}_{\mathtt{i}}) \wedge \\
 & & (\sharp \mathsf{z}' = 1 + \sharp \mathsf{z}) \wedge \mathsf{extr}(\mathsf{z}) \wedge \mathsf{extr}(\mathsf{z}') \\
\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \exists \mathsf{x}.\ \mathsf{g}) & \triangleq & \exists \mathsf{x}.\ \mathsf{notonmark}(\mathsf{x}) \wedge \mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \mathsf{g}) \\
\mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \exists \mathsf{P}_{\mathtt{j+1}}.\mathsf{g}) & \triangleq & \exists \mathsf{z}^{<}_{\mathtt{j+1}}, \mathsf{z}^{>}_{\mathtt{j+1}}.\ \mathsf{isol}(\mathsf{z}^{<}_{\mathtt{j+1}}) \wedge \mathsf{isol}(\mathsf{z}^{>}_{\mathtt{j+1}}) \wedge \\
 & & (\mathsf{relation}_{\mathtt{j+1},\mathtt{Freevar(g)}} \mathrel{-\!\!\circledast} (\mathsf{psenvir}(\mathsf{z}^{<}_{0}, \mathsf{z}^{>}_{\mathtt{j+1}}) \wedge \sharp \mathsf{z}^{>}_{\mathtt{j}} + 1 = \sharp \mathsf{z}^{<}_{\mathtt{j+1}} \\
 & & \wedge \mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j} + 1, \mathsf{g})))
\end{array}
$$

In order to translate $\exists \mathsf{P}_{\mathtt{j+1}}.\ \mathsf{g}$, we introduced two locations whose numbers of predecessors determine the bounds for the degrees for any marker used to encode a pair for the interpretation of $\mathsf{P}_\mathtt{i}$. There is a way to add markers (expressed thanks to the connective $\mathrel{-\!\!\circledast}$) that guarantees that the new part of the heap encodes the interpretation of the variable $\mathsf{P}_{\mathtt{j+1}}$ by using the above formula $\mathsf{relation}_{\mathtt{j+1},\mathtt{X}}$.

Observe that $\mathtt{translation}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathsf{f})$ and f have the same first-order free variables.

2.3.4 Correctness

Before stating the correctness of the translation $\mathtt{translation}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\cdot)$, we need to formally define how to extract an environment from a $\mathtt{j}$-well-formed simple memory shape (but now, that is easy).

Definition 2.3.4.1. Let $(\mathtt{s}, \mathtt{h})$ be a $\mathtt{j}$-well-formed simple memory shape, and let $\mathtt{h}_\mathtt{e}$ be the associated environment heap. The environment $\mathtt{E}$ *extracted from* $\mathtt{h}$ is

$$\mathtt{E}(\mathsf{P}_\mathtt{i}) \triangleq \{(\mathtt{h}_\mathtt{e}(\mathtt{i}), \mathtt{h}_\mathtt{e}(\mathtt{i}')) \ :\ \tilde{\sharp} \mathsf{z}^{<}_{\mathtt{i}} < \tilde{\sharp}\mathtt{i},\ \tilde{\sharp}\mathtt{i} + 1 = \tilde{\sharp}\mathtt{i}',\ \tilde{\sharp}\mathtt{i}' < \tilde{\sharp}\mathsf{z}^{>}_{\mathtt{i}} \text{ in } \mathtt{h}_\mathtt{e}\}$$

for all $\mathtt{i} \in \{1, \ldots, \mathtt{j}\}$.

Correctness of $\mathtt{translation}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\cdot)$ is based on lemma 2.3.4.2 below. The proof shall use several results established earlier.

Lemma 2.3.4.2. Let f be a $\mathtt{DSO}$ formula using the extended Barendregt convention and g be a subformula of f at quantifier depth $\mathtt{j}$. Let $(\mathtt{s}, \mathtt{h})$ be a $\mathtt{j}$-well-formed simple memory shape, with base part $(\mathtt{s}, \mathtt{h}_\mathtt{b})$ and environment part $(\mathtt{s}, \mathtt{h}_\mathtt{e})$, such that for each $\mathsf{x} \in \mathtt{Freevar}(\mathsf{g})$, $\mathtt{s}(\mathsf{x}) \notin \mathtt{Dom}(\mathtt{h}_\mathtt{e})$. Let $\mathtt{E}_\mathtt{j}$ be the environment extracted from $\mathtt{h}_\mathtt{e}$. Then, $(\mathtt{s}, \mathtt{h}) \models_{\mathtt{SL}} \mathtt{tr}_{\mathtt{DSO}\to\mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \mathsf{g})$ iff $(\mathtt{s}, \mathtt{h}_\mathtt{b}), \mathtt{E}_\mathtt{j} \models_{\mathtt{SO}} \mathsf{g}$.

Proof. Let us start by a preliminary definition. We say that a location $\mathtt{i}$ *occurs* in a binary relation $\mathtt{R}$ when there is a location $\mathtt{i}'$ such that $(\mathtt{i}, \mathtt{i}') \in \mathtt{R}$ or $(\mathtt{i}', \mathtt{i}) \in \mathtt{R}$. Let f be a $\mathtt{DSO}$ sentence satisfying the extended Barendregt convention. We want to show by induction on g that given:

– g is a subformula of f of quantifier depth $\mathtt{j}$,

- (s,h) is j-well-formed with base part h_b and environment part h_e such that for every variable $x \in \mathtt{Freevar}(g)$, we have $s(x) \notin \mathtt{Dom}(h_e)$,
- E_j is the environment $\{P_1 \mapsto R_1, \ldots, P_j \mapsto R_j\}$ extracted from h_e,
- no location occurring in $R_1 \cup \ldots \cup R_j$ belongs to $\mathtt{Dom}(h_e)$,

we have: $(s,h) \models_{SL} tr_{DSO \to SL^{mw}}(j, g)$ iff $(s,h_b), E_j \models_{SO} g$.

Base cases.
The base cases $x = y$ and $x \hookrightarrow y$ are by an easy verification since $tr_{DSO \to SL^{mw}}(j, \cdot)$ restricted to them is the identity map. Let us consider the more interesting base case, that is when $g = P_k(x,y)$ with $k \leq j$.

($\rightarrow$) Suppose that $(s,h) \models_{SL} tr_{DSO \to SL^{mw}}(j, P_k(x,y))$. Then, in the heap h, the locations $s(x)$ and $s(y)$ have predecessors in h that are also extremities, let us call them respectively i_x and i_y. In the heap h, we have $\tilde{\sharp} z_k^< < \tilde{\sharp} i_x = \tilde{\sharp} i_y - 1 < \tilde{\sharp} z_k^> - 1$. By definition 2.3.2.6, both i_x and i_y have predecessors in $\mathtt{Dom}(h_e)$ and all of their predecessors are also in $\mathtt{Dom}(h_e)$. Since $z_k^<$ and $z_k^>$ have also all of their predecessors in $\mathtt{Dom}(h_e)$, we have $\tilde{\sharp} z_k^< < \tilde{\sharp} i_x$, $\tilde{\sharp} i_x + 1 = \tilde{\sharp} i_y$ and $\tilde{\sharp} i_y < \tilde{\sharp} z_k^>$ in h_e. By definition 2.3.4.1, we get $(h(i_x), h(i_y)) \in R_k$, that is $(s(x), s(y)) \in R_k$. Consequently, $(s,h_b), E_j \models_{SO} P_k(x,y)$.

($\leftarrow$) Suppose that $(s,h_b), E_j \models_{SO} P_k(x,y)$. By the definitions of $\models_{SO}$ and E_j, we have $(s(x), s(y)) \in R_k$. So $s(x)$ and $s(y)$ have respectively predecessors i_x and i_y in $\mathtt{Dom}(h_e)$. In the heap h_e, i_x and i_y are extremities and $\tilde{\sharp} z_k^< < \tilde{\sharp} i_x = \tilde{\sharp} i_y - 1 < \tilde{\sharp} z_k^> - 1$. By definition 2.3.2.6, the predecessors of any location among $s(z_k^<)$, i_x, i_y and $s(z_k^>)$ belong to $\mathtt{Dom}(h_e)$. So the above inequalities and equality are also true in h. By definition 2.3.2.6, the locations $s(z_k^<)$, i_x, i_y and $s(z_k^>)$ are extremities in h. So $(s,h) \models_{SL} tr_{DSO \to SL^{mw}}(j, P_k(x,y))$.

Induction step.
Our induction hypothesis is the following: for every subformula g' of size strictly less than the size of g, for $j \in \{0, \ldots, n\}$ (n is the quantifier depth of f) and for any j-well-formed simple memory shape (s,h) such that for every variable $x \in \mathtt{Freevar}(g)$, we have $(s,h) \models_{SL} tr_{DSO \to SL^{mw}}(j, g')$ iff $(s,h_b), E_j \models_{SO} g'$.

Case 1: $g = \exists x.\, g'$.

The statements below are equivalent:

(0) $(s,h) \models_{SL} tr_{DSO \to SL^{mw}}(j, \exists x.\, g')$,

(1) there is $i \in \mathtt{Loc}$ such that $(s',h) \models_{SL} tr_{DSO \to SL^{mw}}(j, g')$ and $(s',h) \models_{SL} \mathsf{notonmark}(x)$ with $s' = s[x \mapsto i]$ (by definition of $tr_{DSO \to SL^{mw}}$),

(2) there is $i \in \mathtt{Loc}$ such that $(s',h) \models_{SL} tr_{DSO \to SL^{mw}}(j, g')$ and $i \notin \mathtt{Dom}(h_e)$ with $s' = s[x \mapsto i]$ (by lemma 2.3.3.1),

(3) there is $i \in \mathtt{Loc}$ such that $(s',h_b), E_j \models_{SO} g'$ and $i \notin \mathtt{Dom}(h_e)$ with $s' = s[x \mapsto i]$ (by induction hypothesis since $\mathtt{Freevar}(g') \subseteq \mathtt{Freevar}(\exists x.\, g') \cup \{x\}$),

(4) there is $i \in \mathtt{Loc}$ such that $(s',h_b), E_j \models_{SO} g'$ with $s' = s[x \mapsto i]$,

(5) $(s,h_b), E_j \models_{SO} g$ (by definition of $\models_{SO}$).

Let us justify below why (4) implies (3). Suppose (4) and $\mathtt{i} \in \mathtt{Dom}(h_e)$. Since $(\mathtt{s}, \mathtt{h})$ is $\mathtt{j}$-well-formed, $\mathtt{i} \notin (\mathtt{Dom}(h_b) \cup \mathtt{Im}(h_b))$. Since $\mathtt{Loc}$ is an infinite set, there is a location $\mathtt{i}' \in (\mathtt{Loc} \setminus (\mathtt{Dom}(h_b) \cup \mathtt{Im}(h_b) \cup \mathtt{Dom}(h_e))$ such that $\mathtt{i}'$ does not occur in $(R_1 \cup \ldots \cup R_j)$. By lemma 1.2.3.2, $(\mathtt{s}[\mathtt{x} \mapsto \mathtt{i}'], h_b), E_j[\mathtt{i} \leftarrow \mathtt{i}'] \models_{SO} \mathsf{g}'$. Suppose by contradiction that $\mathtt{i}$ occurs in R_k for some $1 \leq \mathtt{k} \leq \mathtt{j}$. So, $\mathtt{i}$ has a predecessor that is an extremity in $\mathtt{Dom}(h_e)$ and by (P3), $\mathtt{i} \notin \mathtt{Dom}(h_e)$, which leads to a contradiction. Hence, $E_j[\mathtt{i} \leftarrow \mathtt{i}'] = E_j$. We have established that $(\mathtt{s}[\mathtt{x} \mapsto \mathtt{i}'], h_1), E_j \models_{SO} \mathsf{g}'$ and $\mathtt{i}' \notin \mathtt{Dom}(h_e)$.

Case 2: $\mathsf{g} = \exists \mathsf{P}_{j+1}. \mathsf{g}'$.
$(\leftarrow)$

- *(Introduction)* Suppose that $(\mathtt{s}, h_b), E_j \models_{SO} \exists \mathsf{P}_{j+1}. \mathsf{g}'$. By definition of the satisfaction relation $\models_{SO}$, there is $R \in \mathtt{Pow_{fin}}(\mathtt{Loc}^2)$ such that $(\mathtt{s}, h_b), E_j[\mathsf{P}_{j+1} \mapsto R] \models_{SO} \mathsf{g}'$. Since we aim at having locations in h_e that do not interfere with the store, we need to be more restrictive about R.

- *(Replacing* R *by some* R'*)* We build below a finite binary relation R' from R such that no location in $\mathtt{Dom}(h_e)$ occurs in R' and $(\mathtt{s}, h_b), E_j[\mathsf{P}_{j+1} \mapsto R'] \models_{SO} \mathsf{g}'$. More precisely, R' will be obtained from R by replacing its image under a permutation of the set of locations that leaves the locations in $\mathtt{s}$ and h_b fixed. The relation R' is constructed by successively replacing the locations in $\mathtt{Dom}(h_e)$ that occur also in R. Suppose that for some $\mathtt{i} \in \mathtt{Dom}(h_e)$, $\mathtt{i}$ occurs also in R. By the induction hypothesis, for every variable $\mathtt{x} \in \mathtt{Freevar}(\mathsf{g}')$, $\mathtt{i} \neq \mathtt{s}(\mathtt{x})$. By definition 2.3.2.6 on $(\mathtt{s}, \mathtt{h})$, we have $\mathtt{i} \notin (\mathtt{Dom}(h_b) \cup \mathtt{Im}(h_b))$. So $\mathtt{i} \notin (\mathtt{Dom}(h_b) \cup \mathtt{Im}(h_b) \cup \{\mathtt{s}(\mathtt{x}) : \mathtt{x} \in \mathtt{Freevar}(\mathsf{g}')\})$. As $\mathtt{i} \in \mathtt{Dom}(h_e)$ and E_j is extracted from h_e, $\mathtt{i}$ does not occur in $(R_1 \cup \ldots \cup R_j)$. Moreover, for every location $\mathtt{i}'$ that does not occur in $R_1 \cup \ldots \cup R_j$, we have $E_j[\mathtt{i} \leftarrow \mathtt{i}'] = E_j$.

  Since $\{\mathtt{s}(\mathtt{x}) : \mathtt{x} \in \mathtt{Freevar}(\mathsf{g}')\}$, $\mathtt{Dom}(\mathtt{h})$, $\mathtt{Im}(\mathtt{h})$ and $R_1, \ldots, R_j$ are finite sets, there is $\mathtt{i}' \in \mathtt{Loc}$ such that:

  * $\mathtt{i}' \notin (\mathtt{Dom}(h_b) \cup \mathtt{Im}(h_b) \cup \{\mathtt{s}(\mathtt{x}) : \mathtt{x} \in \mathtt{Freevar}(\mathsf{g}')\})$ and $\mathtt{i}' \notin \mathtt{Dom}(h_e)$,
  * $\mathtt{i}'$ does not occur in $R_1 \cup \ldots \cup R_j$.

  By lemma 1.2.3.2, there is $\mathtt{i}' \notin \mathtt{Dom}(h_e)$ such that $(\mathtt{s}[\mathtt{i} \leftarrow \mathtt{i}'], h_b), E_j[\mathsf{P}_{j+1} \mapsto R][\mathtt{i} \leftarrow \mathtt{i}'] \models_{SO} \mathsf{g}'$. As $\mathtt{i} \notin \{\mathtt{s}(\mathtt{x}) : \mathtt{x} \in \mathtt{Freevar}(\mathsf{g})\}$, we also have $\mathtt{s}[\mathtt{i} \leftarrow \mathtt{i}'] = \mathtt{s}$. Let R'' be $R[\mathtt{i} \leftarrow \mathtt{i}']$. Since $E_j[\mathtt{i} \leftarrow \mathtt{i}'] = E_j$, we obtain $(\mathtt{s}, h_b), E_j[\mathsf{P}_{j+1} \mapsto R''] \models_{SO} \mathsf{g}'$.

  If $k_0 \geq 1$ locations in $\mathtt{Dom}(h_e)$ occur in R, then $k_0 - 1$ locations in $\mathtt{Dom}(h_e)$ occur in R''. By applying the above transformation k_0 times we can build a relation R' such that no location in $\mathtt{Dom}(h_e)$ occurs in R' and $(\mathtt{s}, h_b), E_j[\mathsf{P}_{j+1} \mapsto R'] \models_{SO} \mathsf{g}'$.

  Hence, $(\mathtt{s}, h_b), E_j \models_{SO} \exists \mathsf{P}_{j+1}. \mathsf{g}'$ iff there is a finite binary relation $R \in \mathtt{Pow_{fin}}(\mathtt{Loc}^2)$ such that $(\mathtt{s}, h_b), E_j[\mathsf{P}_{j+1} \mapsto R] \models_{SO} \mathsf{g}'$ and no location in $\mathtt{Dom}(h_e)$ occurs in R.

- *(Defining* $(\mathtt{s}', h'_e)$*)* Let us build $\mathtt{s}'$ and h'_e such that

  (A) $(\mathtt{s}', h'_e)$ is an environment between $z^{<}_{j+1}$ and $z^{>}_{j+1}$.
  (B) $(\mathtt{s}', \mathtt{h} * h'_e)$ is $(\mathtt{j} + 1)$-well-formed with the environment part $h_e * h'_e$.

Suppose that $\mathtt{R}$ contains $\mathtt{m}_1 \geq 0$ pairs, say $\mathtt{R} = \{(\mathtt{i}'_1, \mathtt{i}''_1), \ldots, (\mathtt{i}'_{\mathtt{m}_1}, \mathtt{i}''_{\mathtt{m}_1})\}$. Let us build an environment $(\mathtt{s}', \mathtt{h}'_e)$ whose spectrum, for $\mathtt{m}_2 = \tilde{\sharp}\mathtt{z}^>_j + 1$, can depicted as

$$\overset{\mathtt{m}_2}{\bullet} \circ \overbrace{(\bullet \bullet \circ \ldots \bullet \bullet \circ)}^{\mathtt{m}_1 \text{ times the pattern '}\bullet\bullet\circ\text{'}} \overset{\mathtt{m}_2+3\mathtt{m}_1+2}{\bullet}$$

Its set of natural numbers $\mathtt{J}$ is equal to $\{\mathtt{m}_2, \mathtt{m}_2 + 3\mathtt{m}_1 + 2\} \cup \{\mathtt{m}_2 + 3\mathtt{k} + 2, \mathtt{m}_2 + 3\mathtt{k} + 3 \;:\; 0 \leq \mathtt{k} \leq \mathtt{m}_1 - 1\}$.

A location $\mathtt{i}$ is said to be *fresh* if $\mathtt{i} \notin (\{\mathtt{i}'_k, \mathtt{i}''_k \;:\; 1 \leq \mathtt{k} \leq \mathtt{m}_1\} \cup \mathtt{Dom(h)} \cup \mathtt{Im(h)} \cup \{\mathtt{s(x)} \;:\; \mathtt{x} \in \mathtt{Freevar(g')}\})$. By finiteness of the involved objects, let $\mathtt{K}$ be the following set of fresh locations (there is no need to provide here precise values): $\{\mathtt{i}_{\mathtt{m}_3} \;:\; \mathtt{m}_3 \in \mathtt{J}\} \cup \{\mathtt{i}^{\mathtt{m}'_3}_{\mathtt{m}_3} \;:\; \mathtt{m}_3 \in \mathtt{J},\ 1 \leq \mathtt{m}'_3 \leq \mathtt{m}_3\} \cup \{\mathtt{i}''_0, \mathtt{i}'_{\mathtt{m}_1+1}\}$ The store $\mathtt{s}'$ is defined from $\mathtt{s}$ by only imposing that $\mathtt{s}'(\mathtt{z}^<_{j+1}) = \mathtt{i}''_0$ and $\mathtt{s}'(\mathtt{z}^>_{j+1}) = \mathtt{i}'_{\mathtt{m}_1+1}$. The heap $\mathtt{h}'_e$ has domain $\mathtt{K}$ and it is defined as follows:

* $\mathtt{h}'_e(\mathtt{i}^{\mathtt{m}'_3}_{\mathtt{m}_3}) = \mathtt{i}_{\mathtt{m}_3}$ for $\mathtt{m}_3 \in \mathtt{J}$ and $1 \leq \mathtt{m}'_3 \leq \mathtt{m}_3$,
* $\mathtt{h}'_e(\mathtt{i}_{\mathtt{m}_2+3k+2}) = \mathtt{i}'_k$ and $\mathtt{h}'_e(\mathtt{i}_{\mathtt{m}_2+3k+3}) = \mathtt{i}''_k$ for $0 \leq \mathtt{k} \leq \mathtt{m}_1 - 1$,
* $\mathtt{h}'_e(\mathtt{i}_{\mathtt{m}_2}) = \mathtt{i}''_0$ and $\mathtt{h}'_e(\mathtt{i}_{\mathtt{m}_2+3\mathtt{m}_1+2}) = \mathtt{i}'_{\mathtt{m}_1+1}$.

By an easy (and long) verification, one can check that (A) and (B) hold true. Moreover, the relations extracted from $\mathtt{h}_e * \mathtt{h}'_e$ (see the definition 2.3.4.1) are precisely $\mathtt{R}_1, \ldots, \mathtt{R}_j, \mathtt{R}$ and for every $\mathtt{x} \in \mathtt{Freevar(g')}$, $\mathtt{s}'(\mathtt{x}) \notin \mathtt{Dom}(\mathtt{h}_e * \mathtt{h}'_e)$. By the induction hypothesis, $(\mathtt{s}', \mathtt{h}_b), \mathtt{E}_j[\mathtt{P}_{j+1} \mapsto \mathtt{R}] \models_{SO} \mathtt{g}'$ iff $(\mathtt{s}', \mathtt{h} * \mathtt{h}'_e) \models_{SL} \mathtt{tr}_{DSO \to SL^{mw}}(\mathtt{j}+1, \mathtt{g}')$. Thanks to lemma 1.2.3.1, $(\mathtt{s}', \mathtt{h} * \mathtt{h}'_e) \models_{SL} \mathtt{tr}_{DSO \to SL^{mw}}(\mathtt{j}+1, \mathtt{g}')$.

By (A), $(\mathtt{s}', \mathtt{h}'_e) \models_{SL} \mathsf{relation}_{j+1, \mathtt{Freevar(g')}}$. Additionally, by definition of $\mathtt{i}''_0$ and $\mathtt{i}'_{\mathtt{m}_1+1}$, we have $(\mathtt{s}', \mathtt{h}) \models_{SL} \mathsf{isol}(\mathtt{z}^<_{j+1})$ and $(\mathtt{s}', \mathtt{h}) \models_{SL} \mathsf{isol}(\mathtt{z}^>_{j+1})$. Finally, since $(\mathtt{s}, \mathtt{h} * \mathtt{h}'_e)$ is $(\mathtt{j}+1)$-well-formed, we have $(\mathtt{s}, \mathtt{h} * \mathtt{h}'_e) \models_{SL} \mathsf{psenvir}(\mathtt{z}^<_0, \mathtt{z}^>_{j+1})$ (lemma 2.3.2.1) and $(\mathtt{s}, \mathtt{h} * \mathtt{h}'_e) \models_{SL} \tilde{\sharp}\mathtt{z}^>_j + 1 = \tilde{\sharp}\mathtt{z}^<_{j+1}$. As a conclusion, $(\mathtt{s}, \mathtt{h}) \models_{SL} \mathtt{tr}_{DSO \to SL^{mw}}(\mathtt{j}, \exists \mathtt{P}_{j+1}.\, \mathtt{g}')$.

$(\to)$
Suppose that $(\mathtt{s}, \mathtt{h}) \models_{SL} \mathtt{tr}_{DSO \to SL^{mw}}(\mathtt{j}, \exists \mathtt{P}_{j+1}.\, \mathtt{g}')$. In other words, there are locations $\mathtt{i}, \mathtt{i}' \notin (\mathtt{Im(h)} \cup \mathtt{Dom(h)})$, and a disjoint heap $\mathtt{h}'_e \perp \mathtt{h}$ such that the claims below are true

1. $\mathtt{h}'_e$ is disjoint from $\mathtt{h}$ and $\mathtt{s}'$ differs from $\mathtt{s}$ at most for the variables $\mathtt{z}^<_{j+1}$ and $\mathtt{z}^>_{j+1}$.
2. $\mathtt{s}'(\mathtt{z}^<_{j+1}) = \mathtt{i}$ and $\mathtt{s}'(\mathtt{z}^>_{j+1}) = \mathtt{i}'$ do not belong to $\mathtt{Dom(h)}$.
3. $(\mathtt{s}', \mathtt{h}'_e)$ is an environment between $\mathtt{z}^<_{j+1}$ and $\mathtt{z}^>_{j+1}$.
4. $(\mathtt{s}', \mathtt{h} * \mathtt{h}'_e) \models_{SL} \sharp\mathtt{z}^>_j + 1 = \sharp\mathtt{z}^<_{j+1}$.
5. $\mathtt{Dom}(\mathtt{h}'_e) \cap \mathtt{Im(h)} = \emptyset$.

These claims essentially follow from the definition of formula $\mathtt{tr}_{DSO \to SL^{mw}}(\mathtt{j}, \exists \mathtt{P}_{j+1}.\, \mathtt{g}')$, the only difficult part being claim 5. Let us detail this last point: while merging $\mathtt{h}$ and $\mathtt{h}_e$, no new marker can be created so any marker in $\mathtt{h} * \mathtt{h}_e$ is a marker either from $\mathtt{h}_e$ or from $\mathtt{h}$, with the same degree. Moreover, $\mathtt{h} * \mathtt{h}_e$ satisfies $\mathsf{psenvir}(\mathtt{z}^<_0, \mathtt{z}^>_{j+1})$, so the spectrum of $\mathtt{h}_e$ is included in

the one of $\mathtt{h} \star \mathtt{h_e}$. Combining these two facts, it follows that all markers of $\mathtt{h_e}$ are still markers of the same degree in $\mathtt{h} \star \mathtt{h_e}$, and in particular claim 5 holds.

Now, claims 1-5 are precisely the assumptions from lemma 2.3.2.9 and as a consequence $(\mathtt{s}', \mathtt{h} \star \mathtt{h}'_e)$ is $(\mathtt{j}+1)$-well-formed. Observe that (5) is consequence of (3). Since $(\mathtt{s}', \mathtt{h} \star \mathtt{h}'_e) \models_{\mathtt{SL}} \mathtt{tr}_{\mathtt{DSO} \to \mathtt{SL}^{\mathtt{mw}}}(\mathtt{j}, \mathtt{g}')$ and for every $\mathtt{x} \in \mathtt{Freevar}(\mathtt{g}')$ $\mathtt{s}(\mathtt{x}) \notin \mathtt{Dom}(\mathtt{h_e} \star \mathtt{h}'_e)$, we can then apply the induction hypothesis and obtain $(\mathtt{s}, \mathtt{h_b}), \mathtt{E_{j+1}} \models_{\mathtt{SO}} \mathtt{g}'$, that is $(\mathtt{s}, \mathtt{h_b}), \mathtt{E_j} \models_{\mathtt{SO}} \exists \mathtt{P_{j+1}}.\ \mathtt{g}'$ where $\mathtt{E_{j+1}}$ is extracted from $\mathtt{h_e} \star \mathtt{h}'_e$. ☐

Here is our main result about the expressive power of SL.

Theorem 2.4. $\mathtt{SL}^{\neg *} \equiv \mathtt{SL} \equiv \mathtt{SO} \equiv \mathtt{DSO}$.

Proof. The proof follows from the following properties:

- $\mathtt{SL}^{\neg *} \sqsubseteq \mathtt{SL}$ and $\mathtt{DSO} \sqsubseteq \mathtt{SO}$ by simply considering syntactic fragments.
- $\mathtt{SL} \sqsubseteq \mathtt{DSO}$ and $\mathtt{SO} \sqsubseteq \mathtt{DSO}$ by lemma 2.3.1.1 and lemma 2.3.1.2.
- $\mathtt{DSO} \sqsubseteq \mathtt{SL}^{\neg *}$.

It remains to show that $\mathtt{DSO} \sqsubseteq \mathtt{SL}^{\neg *}$ by using lemma 2.3.4.2. Let f be a DSO sentence. Without any loss of generality, we can assume that f has no free occurrence of first-order variables of the form $\mathtt{z}_n^{\diamond}$ (otherwise, other auxiliary variables are used) and f satisfies the extended Barendregt convention since every DSO sentence can be reduced to an equivalent one in logarithmic space. Let $(\mathtt{s}, \mathtt{h})$ be a simple memory shape. The statements below are equivalent

- $(\mathtt{s}, \mathtt{h}) \models_{\mathtt{SL}} \mathtt{translation}_{\mathtt{DSO} \to \mathtt{SL}^{\mathtt{mw}}}(\mathtt{f})$,
- There are $\mathtt{h}'_e \perp \mathtt{h}$, $\mathtt{i}$, $\mathtt{i}'$ and $\mathtt{s}' = \mathtt{s}[\mathtt{z}_0^{<} \mapsto \mathtt{i}, \mathtt{z}_0^{>} \mapsto \mathtt{i}']$ such that
  - $\mathtt{i}$ and $\mathtt{i}' \notin \mathtt{Dom}(\mathtt{h}) \cup \mathtt{Im}(\mathtt{h})$,
  - $\mathtt{i}, \mathtt{i}' \in \mathtt{Dom}(\mathtt{h}')$ and for every location $\mathtt{i}'' \in \mathtt{Dom}(\mathtt{h}'_e) \setminus \{\mathtt{i}, \mathtt{i}'\}$, we have $\mathtt{h}'_e(\mathtt{i}'') \in \{\mathtt{i}, \mathtt{i}'\}$.
  - In $(\mathtt{s}', \mathtt{h} \star \mathtt{h}'_e)$, $\tilde{\sharp}\mathtt{z}_0^{>} = 2 + \tilde{\sharp}\mathtt{z}_0^{<}$ and for every $\mathtt{i}'' \in \mathtt{Dom}(\mathtt{h})$, we have $\tilde{\sharp}\mathtt{z}_0^{<} \geq 3 + \tilde{\sharp}\mathtt{i}''$.
  - $\mathtt{i}$ and $\mathtt{i}'$ are extremities in $(\mathtt{s}', \mathtt{h} \star \mathtt{h}'_e)$.
  - $(\mathtt{s}', \mathtt{h} \star \mathtt{h}'_e) \models_{\mathtt{SL}} \mathtt{tr}_{\mathtt{DSO} \to \mathtt{SL}^{\mathtt{mw}}}(0, \mathtt{f})$.

  (by definition of $\mathtt{translation}_{\mathtt{DSO} \to \mathtt{SL}^{\mathtt{mw}}}(\cdot)$ and $\models_{\mathtt{SL}}$)
- There are $\mathtt{h}' \perp \mathtt{h}$, $\mathtt{i}$ and $\mathtt{i}'$ such that
  - $(\mathtt{s}', \mathtt{h} \star \mathtt{h}'_e)$ is an environment with $\tilde{\sharp}\mathtt{z}_0^{>} = 2 + \tilde{\sharp}\mathtt{z}_0^{<}$.
  - $(\mathtt{s}', \mathtt{h} \star \mathtt{h}') \models_{\mathtt{SL}} \mathtt{tr}_{\mathtt{DSO} \to \mathtt{SL}^{\mathtt{mw}}}(0, \mathtt{f})$.

  (by definition 2.3.2.6 and lemma 2.3.2.8)
- There are $\mathtt{h}' \perp \mathtt{h}$, $\mathtt{i}$ and $\mathtt{i}'$ such that
  - $(\mathtt{s}', \mathtt{h} \star \mathtt{h}'_e)$ is an environment with $\tilde{\sharp}\mathtt{z}_0^{>} = 2 + \tilde{\sharp}\mathtt{z}_0^{<}$.
  - $(\mathtt{s}', \mathtt{h}), \mathtt{E_0} \models_{\mathtt{SO}} \mathtt{f}$ for any environment $\mathtt{E_0}$ extracted from $\mathtt{h}'_e$.

  (by lemma 2.3.4.2)

– $(s, h) \models_{SO} f$ since

  * the variables $z_0^<$ and $z_0^>$ do not occur in f and f is a sentence.
  * h'_e can always be built since h is essentially a finite structure.

□

Observe that all the equivalences are obtained with logarithmic space translations. Consequently,

Theorem 2.5. SL^{-*} satisfiability problem is undecidable.

Proof. We have seen that for every sentence such that f in DSO, there is an effective way to compute f' in SL^{-*} such that f and f' hold on exactly the same simple memory shapes. In order to show undecidability of SL^{-*}, it is sufficient to provide a reduction from finitary satisfiability for classical predicate logic restricted to a single binary predicate symbol (see [88]) to DSO. Let f be a first-order formula built over the binary predicate symbol Q. One can easily show that f is satisfiable iff

$$\exists P.\ \exists Q.\ (\forall x\, y.\ Q(x, y) \Rightarrow P(x, x) \wedge P(y, y)) \wedge tr_{FO_{fin-pred} \to DSO}(f)$$

is satisfiable. The map $tr_{FO_{fin-pred} \to DSO}$ is the identity map for atomic formulas, homomorphic for boolean connectives, and performs a relativization for first-order quantification as follows: $tr_{FO_{fin-pred} \to DSO}(\forall x.\ g) \triangleq \forall x.\ P(x, x) \Rightarrow tr_{FO_{fin-pred} \to DSO}(g)$. The intention is obviously that $P(x, x)$ holds true whenever x belongs to the finite model. □

Undecidability of SL^{-*} can be obtained much more easily by encoding the halting problem for Minsky machines by using the fact that $\sharp x = \sharp y$ and $\sharp x = \sharp y + 1$ can be expressed in SL^{-*} (section 2.2). Indeed, computations of length n can be encoded as lists of length $3n$; three successive locations encode a configuration of the machine and for two of those locations, counter values are encoded by the numbers of predecessors. Theorem 2.5 is obtained with the stronger result $SL^{-*} \equiv DSO$ since DSO is undecidable.

2.4 Extensions with More Than one Selector

In order to express advanced arithmetical constraints (see section 2.2) or to encode finite sets of pairs of locations (see section 2.3), we have introduced additional parts in the heaps via markers. In order to distinguish these auxiliary markers from the original heap, we have decided to use markers of small degree (as in section 2.2) or markers of large degree (as in section 2.3). However, in the presence of memory cells with strictly more than one selector it is even easier to identify these auxiliary markers; for example, the memory cells $i \mapsto i'$ introduced in a heap to check arithmetical constraints or to encode environments can be replaced by memory cells of the form

$$i \mapsto i', \overbrace{k_0, \ldots, k_0}^{(k-1)\ \text{times}}$$

where k_0 is a location that is not present in the original heap (that is not in $Im(h) \cup Dom(h)$). We write kSL [resp. kSO] to denote the variant of SL [resp. SO] with k selectors. In that case, a heap h is defined as a partial function $h : Loc \rightharpoonup Loc^k$ with finite domain. The atomic formulas

of the form $\mathtt{x} \hookrightarrow \mathtt{y}$ from SL are replaced by $\mathtt{x} \hookrightarrow \mathtt{y}_1, \ldots, \mathtt{y}_k$. Obviously 1SL [resp. 1SO] corresponds to SL [resp. SO]. We write $\mathtt{kSO}^{k'}$ to denote the restriction of kSO to second-order variables in $\mathtt{Secvar}_{k'}$. So $\mathtt{1SO}^2 = \mathtt{DSO}$.

In the rest of this section, we assume that $k > 1$. We dedicate the rest of this section to show theorem 2.6 below can be proved by adapting what we did for a unique selector. We may overload symbols but no confusion should occur. The case $k = 1$ requires special care but a simpler direct proof is possible for $k \neq 1$. Indeed, for $k = 1$ the identification of auxiliary memory cells is performed thanks to structural properties whereas for $k > 1$, this could be done by simply checking the presence of distinguished values.

Theorem 2.6. For every $k > 1$, $\mathtt{kSL} \equiv \mathtt{kSL}^{-*} \equiv \mathtt{kSO}$.

We establish theorem 2.6 by adapting the proof for $k = 1$. However, a simpler proof for $k > 1$ is possible but it would require a different approach. First, an obvious adaptation of the proof of lemmas 2.3.1.2 and 2.3.1.1 allows us to show the statement below.

Lemma 2.4.0.3. $\mathtt{kSL} \sqsubseteq \mathtt{kSO}^{k+1}$ and $\mathtt{kSO}^{k+1} \sqsubseteq \mathtt{kSO}^2$.

It remains to show that $\mathtt{kSO}^2 \sqsubseteq \mathtt{kSL}^{-*}$. The basic observation is that all the auxiliary memory cells $\mathtt{i} \mapsto \mathtt{i}'$ introduced in a heap to check arithmetical constraints or to encode environments are replaced by memory cells of the form

$$\mathtt{i} \mapsto \mathtt{i}', \overbrace{\mathtt{k}_0, \ldots, \mathtt{k}_0}^{(k-1) \text{ times}}$$

where $\mathtt{k}_0$ is a location that is not present in the original heap. Observe that it is easy to check that a memory cell is auxiliary by simply inspecting the presence of $\mathtt{k}_0$. We shall also enforce that in a new memory cell, $\mathtt{i}'$ is different from $\mathtt{k}_0$ and the $(k-1)$ remaining locations are each $\mathtt{k}_0$.

Before explaining the adaptation, we introduce alternative definitions:

Definition 2.4.0.4.

- Given $(\mathtt{s}, \mathtt{h})$ and a location $\mathtt{i}$, we write $\tilde{\sharp}\mathtt{i}$ to denote the cardinal of $\{\mathtt{i}' \in \mathtt{Loc} : \mathtt{h}(\mathtt{i}') = (\mathtt{i}, \ldots)\}$ (number of *1-predecessors* of the location $\mathtt{i}$ in $(\mathtt{s}, \mathtt{h})$).
- We write $\mathtt{x} \hookrightarrow \mathtt{y}$ as a shortcut for $\exists \mathtt{y}_2, \ldots, \mathtt{y}_k.\ \mathtt{x} \hookrightarrow \mathtt{y}, \mathtt{y}_2, \ldots, \mathtt{y}_k$.
- A [resp. *strict*] *marker* in $(\mathtt{s}, \mathtt{h})$ is a sequence of distinct locations $\mathtt{i}, \mathtt{i}_0, \ldots, \mathtt{i}_n$ for some $n \geq 0$ (all distinct from $\mathtt{k}_0$) such that
  - $\mathtt{h}(\mathtt{i}_0) = (\mathtt{i}, \overbrace{\mathtt{k}_0, \ldots, \mathtt{k}_0}^{k-1 \text{ times}})$ [resp. and $\mathtt{Dom}(\mathtt{h}) = \{\mathtt{i}_0, \ldots, \mathtt{i}_n\}$],
  - for every $i \in \{1, \ldots, n\}$, $\mathtt{h}(\mathtt{i}_i) = (\mathtt{i}_0, \overbrace{\mathtt{k}_0, \ldots, \mathtt{k}_0}^{k-1 \text{ times}})$ and $\tilde{\sharp}\mathtt{i}_i = 0$,
  - $\tilde{\sharp}\mathtt{i}_0 = n$.
- We define an *extremity* as a location $\mathtt{i}$ in a heap such that $\mathtt{i}$ has at least one 1-predecessor and no 1-predecessor $\mathtt{i}'$ of $\mathtt{i}$ appears in some tuple from $\mathtt{Im}(\mathtt{h})$.
- Let f_{k_0} be the formula specifying that auxiliary memory cells are of the above shape:

$$f_{k_0} \triangleq \forall \mathtt{x}, \mathtt{x}_1, \ldots, \mathtt{x}_k.\ \mathtt{x} \hookrightarrow \mathtt{x}_1, \ldots, \mathtt{x}_k \Rightarrow (\mathtt{x} \neq \mathtt{x}_{k_0} \wedge \mathtt{x}_1 \neq \mathtt{x}_{k_0} \wedge \bigwedge_{i=2}^{k} \mathtt{x}_i = \mathtt{x}_{k_0})$$

Following the developments from section 2.2, we can show the following lemma.

Lemma 2.4.0.5. For $\mathtt{m}, \mathtt{m}' \geq 0$, there is a formula f in $\mathtt{kSL}^{\twoheadrightarrow}$ of quadratic size in $\mathtt{m} + \mathtt{m}'$ such that for every memory shape $(\mathtt{s}, \mathtt{h})$, we have $(\mathtt{s}, \mathtt{h}) \models_{SL} \mathsf{f}$ iff $\tilde{\sharp}\mathsf{x} + \mathtt{m} \leq \tilde{\sharp}\mathsf{y} + \mathtt{m}'$.

We consider the formula from section 2.2 in which we add to the left argument of any subformula with outermost connective either $\stackrel{\circledast}{\twoheadrightarrow}$ or $\twoheadrightarrow$ the conjunct f_{k_0}. The concerned formulas are those which introduce markers in the heap. Moreover, in some cases, formulas of the form $\mathsf{x} \hookrightarrow \mathsf{y}$ for the one selector case from section 2.2 are replaced by $\mathsf{x} \hookrightarrow \mathsf{y}, \mathsf{x}_{k_0}, \ldots, \mathsf{x}_{k_0}$ when markers are involved.

Let us consider the reduction from $\mathtt{kSO}^2$ into $\mathtt{kSL}$. Given a sentence in $\mathtt{kSO}^2$ satisfying the extended Barendregt convention and with $\mathtt{n}$ second-order variables, its translation $\mathtt{translation}'_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathsf{f})$ is defined below where $\mathtt{translation}''_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathsf{f})$ is a variant of the map $\mathtt{translation}_{\mathtt{DSO} \to \mathtt{SL}^{mw}}(\mathsf{f})$ for the one selector case and where the definition of the inductive auxiliary translation $\mathtt{tr}_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathtt{j}, \mathtt{g})$ is modified as follows.

Note that $\mathsf{notonmark}(\mathsf{x})$ is defined by $\mathsf{notonmark}(\mathsf{x}) \triangleq \neg(\exists \mathsf{y}.\ \mathsf{x} \hookrightarrow \mathsf{y}, \mathsf{x}_{k_0}, \ldots, \mathsf{x}_{k_0} \vee \mathsf{y} \hookrightarrow \mathsf{x}, \mathsf{x}_{k_0}, \ldots, \mathsf{x}_{k_0}) \wedge \mathsf{x} \neq \mathsf{x}_{k_0}$. Also, the formula $\mathsf{isol}(\mathsf{x})$ is now an abbreviation for $\mathsf{isol}(\mathsf{x}) \triangleq \forall\, \mathsf{y}, \mathsf{y}_1, \ldots, \mathsf{y}_k.\ (\mathsf{y} \hookrightarrow \mathsf{y}_1, \ldots, \mathsf{y}_k) \Rightarrow ((\mathsf{y} \neq \mathsf{x}) \wedge \bigwedge_{i=1}^{i=k} (\mathsf{y}_i \neq \mathsf{x}))$.

$$
\begin{array}{lcl}
\mathtt{translation}'_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathsf{f}) & \triangleq & \exists \mathsf{x}_{k_0}. \\
 & & \neg(\exists \mathsf{x}, \mathsf{x}_1, \ldots, \mathsf{x}_k. \\
 & & \quad (\mathsf{x} \hookrightarrow \mathsf{x}_1, \ldots, \mathsf{x}_k) \wedge (\mathsf{x} = \mathsf{x}_{k_0} \vee \bigvee_{i=1}^{k} \mathsf{x}_i = \mathsf{x}_{k_0})) \\
 & & \wedge \mathtt{translation}''_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathsf{f}) \\
\mathtt{translation}''_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathsf{f}) & \triangleq & \exists \mathsf{z}_0^{<}\ \mathsf{z}_0^{>}.\ \mathsf{isol}(\mathsf{z}_0^{>}) \wedge \mathsf{isol}(\mathsf{z}_0^{<}) \wedge \\
 & & [((\forall \mathsf{x}.\ \mathsf{alloc}(\mathsf{x}) \Rightarrow (\mathsf{x} \hookrightarrow \mathsf{z}_0^{>} \vee \mathsf{x} \hookrightarrow \mathsf{z}_0^{<} \vee \mathsf{x} = \mathsf{z}_0^{>} \vee \mathsf{x} = \mathsf{z}_0^{<})) \\
 & & \wedge \mathsf{alloc}(\mathsf{z}_0^{>}) \wedge \mathsf{alloc}(\mathsf{z}_0^{<})) \\
 & & \wedge \mathsf{f}_{k_0} \stackrel{\circledast}{\twoheadrightarrow} \\
 & & ((\forall \mathsf{x}.\mathsf{x} \neq \mathsf{z}_0^{>} \wedge \mathsf{x} \neq \mathsf{z}_0^{<} \Rightarrow (\sharp \mathsf{z}_0^{<} > 2 + \sharp \mathsf{x})) \wedge (\sharp \mathsf{z}_0^{>} = 2 + \sharp \mathsf{z}_0^{<}) \\
 & & \wedge \mathsf{extr}(\mathsf{z}_0^{<}) \wedge \mathsf{extr}(\mathsf{z}_0^{>}) \wedge \mathtt{tr}_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(0, \mathsf{f}))] \\
\mathtt{tr}_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathtt{j}, \mathsf{x} = \mathsf{y}) & \triangleq & \mathsf{x} = \mathsf{y} \\
\mathtt{tr}_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathtt{j}, \mathsf{x} \hookrightarrow \mathsf{y}) & \triangleq & \mathsf{x} \hookrightarrow \mathsf{y} \\
\mathtt{tr}_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathtt{j}, \mathsf{P}_{\mathtt{j}}(\mathsf{x}, \mathsf{y})) & \triangleq & \exists \mathsf{z}, \mathsf{z}'.\ (\mathsf{z} \hookrightarrow \mathsf{x}) \wedge (\mathsf{z}' \hookrightarrow \mathsf{y}) \wedge (\sharp \mathsf{z} > \sharp \mathsf{z}_{\mathtt{j}}^{<}) \wedge (\sharp \mathsf{z}' < \sharp \mathsf{z}_{\mathtt{j}}^{>}) \wedge \\
 & & (\sharp \mathsf{z}' = 1 + \sharp \mathsf{z}) \wedge \mathsf{extr}(\mathsf{z}) \wedge \mathsf{extr}(\mathsf{z}') \\
\mathtt{tr}_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathtt{j}, \exists \mathsf{x}.\ \mathsf{g}) & \triangleq & \exists \mathsf{x}.\ \mathsf{notonmark}(\mathsf{x}) \wedge \mathtt{tr}_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathtt{j}, \mathsf{g}) \\
\mathtt{tr}_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathtt{j}, \exists \mathsf{P}_{\mathtt{j+1}}.\mathsf{g}) & \triangleq & \exists \mathsf{z}_{\mathtt{j+1}}^{iz}, \mathsf{z}_{\mathtt{j+1}}^{>}.\ \mathsf{isol}(\mathsf{z}_{\mathtt{j+1}}^{<}) \wedge \mathsf{isol}(\mathsf{z}_{\mathtt{j+1}}^{>}) \wedge \\
 & & ((\mathsf{relation}_{\mathtt{j+1}, \mathtt{Freevar(g)}} \wedge \mathsf{f}_{k_0}) \stackrel{\circledast}{\twoheadrightarrow} \\
 & & (\mathsf{psenvir}(\mathsf{z}_0^{<}, \mathsf{z}_{\mathtt{j+1}}^{>}) \wedge \sharp \mathsf{z}_{\mathtt{j}}^{>} + 1 = \sharp \mathsf{z}_{\mathtt{j+1}}^{<} \\
 & & \wedge \mathtt{tr}_{\mathtt{kSO}^2 \to \mathtt{kSL}^{mw}}(\mathtt{j} + 1, \mathsf{g})))
\end{array}
$$

Additionally, $\mathsf{relation}_{\mathtt{j+1}, \mathtt{Freevar(g)}}$ and $\mathsf{psenvir}(\mathsf{z}_0^{<}, \mathsf{z}_{\mathtt{j+1}}^{>})$ are slightly updated in order to take into account that the markers are made of memory cells of the form $\mathtt{i} \mapsto \mathtt{i}', \mathtt{k}_0, \ldots, \mathtt{k}_0$.

By adapting definition 2.3.4.1 with 1-predecessors, we can then state a lemma similar to lemma 2.3.4.2 leading to theorem 2.6.

Conclusion

Summary of this Chapter

We have mainly studied first-order separation logic with one selector SL for which we have shown the following results:

1. $\mathtt{SL}^{\star}$ is decidable with non-elementary complexity.
2. $\mathtt{SL}^{\star,\multimap_n}$ and $\mathtt{SL}^{<n}$, extending $\mathtt{SL}^{\star}$ with bounded magic wand are also decidable.
3. SL is as expressive as weak second-order logic SO.
4. $\mathtt{SL}^{\multimap}$ is as expressive as SL as a by-product of our proof technique.
5. SL satisfiability is undecidable.

This solves two central open problems: the decidability status of SL and the characterization of its expressive power. Moreover, the above results about expressive power extend naturally to the case with k selectors, for some $k \geq 1$: $k\mathtt{SL} \equiv k\mathtt{SL}^{\multimap} \equiv k\mathtt{SO}$. Figure 2.10 contains, summarized, our decidability results concerning models with one selector. Figure 2.11 is an updated sketch of the expressiveness results – solid arrows represent a logarithmic space translation and dotted arrows are polynomial time translations.

Related Work

The closest work to ours is certainly the work of Antonopoulos and Dawar [3] on the comparison of the expressive power of monadic second-order logic and the spatial logic for graphs: they showed that the graph logic, and as a consequence $\mathtt{SL}^{\star}$, is strictly less expressive than MSO. Although the questions solved in this work do not overlap the results presented herein, it adopts a point of view quite similar to the one we presented and gives a more complete picture of the topic.

The magic wand is rarely considered by the literature on SL, which our result may explain from the complexity point of view. The magic wand is however often behind the scene in recent developments of SL. For instance, the bi-abduction problem presented by Gorogiannis, Kanovich and O'Hearn in [56] can be seen as a specialized version of the satisfiability problem for SL with magic wand. As a parallel to this work, results stating either the absence of adjunct elimination or the undecidability of satisfiability for logics including a form of magic wand have been independently established for the boolean logic of bunched implications by Larchey-Wendling and Galmiche in [68], propositional SL by Brotherston and Kanovich in [30], or context logic by Calcagno, Gardner and Zarfaty in [34]. The main difference with our work is that the models of these logics include formal propositional variables that can be used to axiomatize the models in any desired way, whereas we are sticking to the heap model.

Even without the magic wand, the decidability we obtained for $\mathtt{SL}^{\star}$ is with non-elementary complexity. The infeasible complexity of separation logic, even propositional as shown by Lozes in [73] explains why, in practice, tools work with symbolic heaps, which have been proved tractable by Cook et al. in [38].

Heap properties are formalized in various logical languages [63, 69, 84, 24, 92] and separation logic is just one prominent example of these logics. However, we focus on expressive power and decidability issues rather than on formal verification. Verification methods and logics for verifying programs with singly-linked lists can be found for instance in [11, 22, 83].

Decidable	$\mathtt{SL}^{\star}$ – theorem 2.1
	$\mathtt{SL}^{\star,\mathrel{-\!\!\star}_n}$ – theorem 2.2
	$\mathtt{SL}^{<n}$ – theorem 2.3
Undecidable	$\mathtt{SL}$, $\mathtt{SL}^{\mathrel{-\!\!\star}}$ – theorem 2.5

Figure 2.10: Decidability results

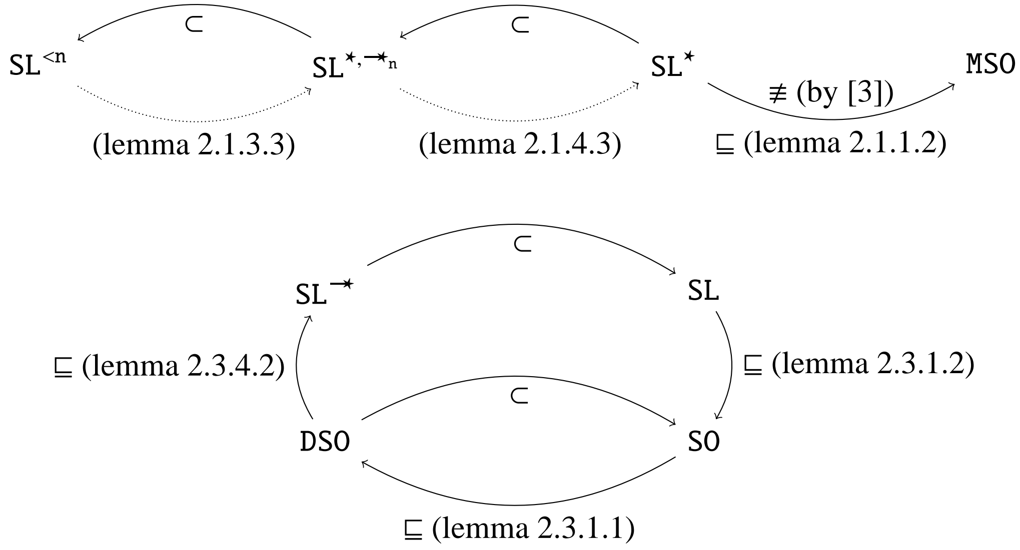


Figure 2.11: Translations

The relationships between logics on graphs with separating features and second-order logic are presented by Dawar, Gardner and Ghelli in [42]. Also, a relationship between separation logic and hyperedge-replacement grammars on a class of hypergraphs representing memory shapes is established by [48].

Perspectives

Note that we used the loose version of points-to and as far as we can judge, our results involving $\mathtt{SL}$ without separating conjunction are dependent on using the loose points-to. It is easy to obtain tight points-to from loose points-to and loose points-to from tight points-to when the separating conjunction belongs to the studied logical fragment, hence any result about a fragment containing the separating conjunction can probably be adapted. But, when the separating conjunction is not present and the points-to predicate is tight, we conjecture that obtaining loose points-to is impossible, as well as expressing that an address has a predecessor in a heap which has strictly more than one allocated address. If these properties are actually impossible to express, it is then difficult to express interesting properties about heaps which contain strictly more than one allocated address. As a consequence, we conjecture that separation logic without separating conjunction and with tight points-to is not as expressive as $\mathtt{SL}^{\mathrel{-\!\!\star}}$.

Finally, we conjecture that $\mathtt{SL}$ with only two variables can encode $\mathtt{SO}$.

Chapter 3

Beyond Shapes: Lists with Ordered Data

Introduction

Contribution of this Chapter

We are now going to study a separation logic on simple memory states, that are models which additionally to their single location selector contain a data field in each cell. Adding data to the model, we hope to extend our decidability results of previous chapter to models which correspond to the memory states of programs manipulating ordered lists. Separation logic was introduced for the verification of programs, and programs generally handle more than a memory shape as they are likely to additionally handle data, for instance in an ordered list. Taking our inspiration from this fact, the decidability results of this chapter will make the results of previous chapter able to deal with such a structure.

In this chapter, we are going to use predicates for comparison of data stored in the model; a reminder of these predicates is available in figure 3.1 for reference.

As we have just shown in the previous chapter that the magic wand $-\!\!*$ brings undecidability, the language SL_{v} we study does not contain the magic wand. On the other hand, we have shown the fragment without $-\!\!*$ is decidable, as well as the fragment with restricted wand $-\!\!*_{\mathtt{n}}$. Here, we will prove that on models with data, a fragment without any wand is decidable too, but the fragment with restricted wand is not. Additionally, the comparison of data has to be restricted to short distance and guarded long distance so as to maintain decidability when the wand is dropped. The results are summarized in figure 3.2.

The decidability result comes from a reduction to monadic second-order logic over functional graphs. The translation is strongly inspired by the one for separation logic over lists without data of chapter 2, but involves some non-trivial complications for ensuring the consistency of data abstraction. The undecidability results are obtained by reduction from first-order logic over finite data words, which was proved undecidable by Bojańsczyk et al. in [15], and was further studied with an approach of temporal logics in the work of Demri, Lazić and Nowak in [46].

Structure of the Chapter

In section 3.1, we establish the decidability of the short distance comparison. Section 3.2 deals with the case of guarded and non-guarded long-distance comparison. Finally, section 3.3

Short distance comparison	$x \hookrightarrow_{\leq} y$ and $x \hookrightarrow_{\geq} y$
Long distance comparison	$\mathsf{val}(x) \leq \mathsf{val}(y)$
Guarded long distance comparison	$\mathsf{val}(x) \leq \mathsf{val}(w)$

Figure 3.1: Comparison predicates

Undecidable	SL_v with long distance comparison SL_v with short distance comparison and the restricted wand
Decidable	SL_v with short distance comparison SL_v with short distance comparison and guarded long distance comparison

Figure 3.2: Decidability and undecidability results

explains the undecidability of the logic in the presence of the restricted magic wand.

This chapter presents results originally published in [6].

3.1 Decidability of Short-Distance Comparisons

In this section, we establish the decidability of a fragment of SL_v with short-distance comparison.

Definition 3.1.0.6. The fragment $\mathsf{SL}_\mathsf{v}^{\mathsf{short}}$ is defined by the following grammar:

$$f ::= \neg f \mid f \wedge f \mid \exists x.f \mid x \hookrightarrow y \mid x \hookrightarrow_{\leq} y \mid x \hookrightarrow_{\geq} y \mid x = y \mid f \star f$$

The semantics of the operators and atomic formulas of this fragment is defined in section 1.3. Note that the operator $\multimap$ does not belong to this grammar.

3.1.1 Method

The decidability of satisfiability for $\mathsf{SL}_\mathsf{v}^{\mathsf{short}}$ is obtained by reduction to the satisfiability of MSO over simple memory shapes.

Colored Shapes

We have to abstract the values taking care of their local comparisons. To do so, we use a colored shape, with three colors on the edges: ‘<’, ‘>’, and ‘=’. Formally, the colors are on vertices, but each edge can be non-ambiguously identified to its source vertex in our model. In logical terms, these colors will be defined by two second-order variables, noted P and Q, and we will observe the color ‘=’ if both P and Q hold for the source location of the edge, ‘<’ if P holds but not Q, and ‘>’ if Q holds but not P. The case where neither P nor Q holds is irrelevant since we assumed a total order on data values, so we should constrain the possible choices for P and Q to avoid this situation. Moreover, some extra constraints will be involved by the necessity to

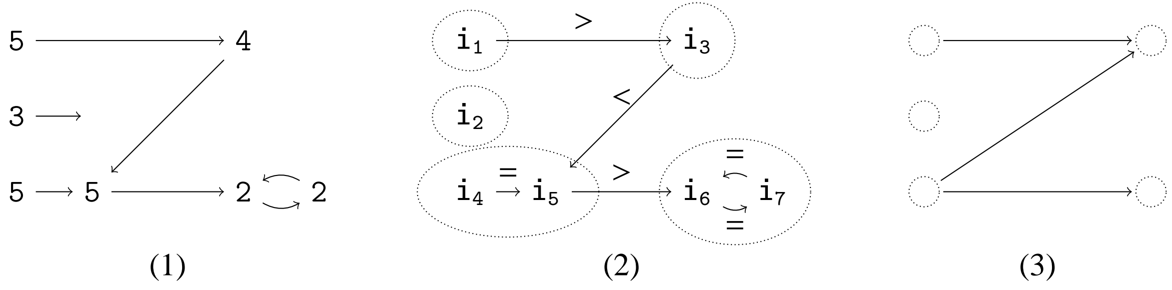


Figure 3.3: A concrete heap (1), its colored abstraction (2), and the associated graph of constraints (3); here $R_P = \{i_3, i_4, i_6, i_7\}$ and $R_Q = \{i_1, i_4, i_5, i_6, i_7\}$

---

manipulate only colored shapes for which it is possible to assign data respecting the colors (for instance, a cycle of '<' cannot be assigned data).

The Graph of Constraints

Given a shape (s, h), and the interpretations $R_P, R_Q \subseteq \mathrm{Dom}(h)$ of the second-order variables mentioned before, we define the associated graph of constraints $\mathbb{G} = (I, J)$ where:

- The set of vertices I is the quotient of $\mathrm{Dom}(h)$ by the equivalence $i \sim i'$ relating locations connected by a non oriented, '='-labeled path in the colored shape. Note that each $\sim$-equivalence class contains at most one location k whose image under h lies outside the equivalence class of k. In such a situation, $[k]$ denotes this equivalence class.
- The set of edges J is the set of pairs of equivalence classes $([k], [k'])$ such that
  * either $h(k) = k'$ and the color on k is '>'
  * or $h(k') = k$ and the color on k' is '<'

Figure 3.3 gives an example of a colored shape and its associated graph of constraints. Note that an edge towards a dangling pointer cannot be colored, and this is in fact the unique situation in which one allows $\neg P \wedge \neg Q$. The graph of constraints helps us to decide whether or not it is possible to assign values to a colored shape: indeed, this problem is equivalent to defining a topological order on the graph of constraints, which is known to be equivalent to this graph being acyclic. What remains to be explained now is:

- how to define the graph of constraints in MSO,
- how to express acyclicity,
- how to treat separating conjunction.

The Reduction

The reduction from SL_v^{short} to MSO is defined by the function $\mathrm{translation}_{SL_v^{short} \to MSO}(f) \triangleq \exists P.\exists Q.\exists Q_0.\mathrm{cons}(P, Q, Q_0) \wedge \mathrm{tr}_{SL_v^{short} \to MSO}(f, P, Q, Q_0)$ where:

– Q_0 is an extra second-order variable that is needed to define the current focus, that is the sub-heap of the original heap on which the (sub)formula is currently evaluated.

– $\mathtt{tr}_{\mathtt{SL}^{\mathtt{short}}_{\mathtt{V}} \to \mathtt{MSO}}$ is an auxiliary reduction that works assuming that P, Q and Q_0 have been correctly guessed, updating these parameters appropriately when $\star$ is translated.

– cons are constraints imposed on P, Q and Q_0 to guarantee that the first guess is a valid one: R_{Q_0} is the domain of the heap, and R_P and R_Q define a colored shape to which one may assign values.

3.1.2 Constraints

We impose three constraints, included in three formulas: $\mathsf{cons}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0) \triangleq \mathsf{cons1}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0) \land \mathsf{cons2}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0) \land \mathsf{cons3}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0)$

1. the only admitted color on a monochromatic cycle is '=' (this is indeed equivalent to the acyclicity condition on the graph of constraints):

$$\mathsf{cons1}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0) \ \triangleq\ \forall \mathsf{Q}_1 \subseteq \mathsf{Q}_0.\ \mathsf{loop}(\mathsf{Q}_1) \ \Rightarrow\ (\mathsf{Q}_1 \subseteq \mathsf{P} \Leftrightarrow \mathsf{Q}_1 \subseteq \mathsf{Q})$$

   where $\mathsf{loop}(\mathsf{Q}_1)$ is defined as $\mathsf{setofloops}(\mathsf{Q}_1) \land \forall \mathsf{Q}_2 \subsetneq \mathsf{Q}_1. \neg \mathsf{setofloops}(\mathsf{Q}_2)$, where $\mathsf{setofloops}(\mathsf{Q}_1)$ is $\forall \mathsf{x}.\mathsf{Q}_1(\mathsf{x}) \Rightarrow \exists \mathsf{y}.\mathsf{Q}_1(\mathsf{y}) \land \mathsf{y} \hookrightarrow \mathsf{x}$

2. every edge that should be colored is colored with '<', '>' or '='

$$\mathsf{cons2}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0) \ \triangleq\ \forall \mathsf{x}.\ (\mathsf{Q}_0(\mathsf{x}) \land (\exists \mathsf{y}.\mathsf{Q}_0(\mathsf{y}) \land \mathsf{x} \hookrightarrow \mathsf{y})) \ \Leftrightarrow\ (\mathsf{P}(\mathsf{x}) \lor \mathsf{Q}(\mathsf{x}))$$

3. R_{Q_0} is the domain of the heap:

$$\mathsf{cons3}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0) \ \triangleq\ \forall \mathsf{x}.(\mathsf{x} \hookrightarrow \Box) \ \Leftrightarrow\ \mathsf{Q}_0(\mathsf{x}).$$

Definition 3.1.2.1. We say that a location $\mathtt{i}$ is an *increasing (resp. decreasing)* node if there are $\mathtt{i}', \mathtt{i}'' \in \mathtt{Loc}$ and $\mathtt{o}_1, \mathtt{o}_2 \in \mathtt{Dat}$ such that $\mathtt{h}'(\mathtt{i}) = (\mathtt{i}', \mathtt{o}_1)$, $\mathtt{h}'(\mathtt{i}') = (\mathtt{i}'', \mathtt{o}_2)$, and $\mathtt{o}_1 \leq \mathtt{o}_2$ (resp. $\mathtt{o}_1 \geq \mathtt{o}_2$). We write $\mathtt{Dom}^+(\mathtt{h}')$ (resp. $\mathtt{Dom}^-(\mathtt{h}')$) to denote the set of increasing (resp. decreasing) nodes of $\mathtt{h}'$, and $E_{\mathtt{h}'}$ denotes the environment $[\mathsf{P} \mapsto \mathtt{Dom}^+(\mathtt{h}'), \mathsf{Q} \mapsto \mathtt{Dom}^-(\mathtt{h}'), \mathsf{Q}_0 \mapsto \mathtt{Dom}(\mathtt{h}')]$.

Definition 3.1.2.2. Given a model $(\mathtt{s}, \mathtt{h})$ and a environment $\mathtt{E}$, we define the *edge labelled graph* $\mathbb{G} = (\mathtt{I}, \mathtt{J}, \mathtt{L})$ *obtained from* $(\mathtt{s}, \mathtt{h})$ and $\mathtt{E}$ as below. Let R_P be $\mathtt{E}(\mathsf{P})$, R_Q be $\mathtt{E}(\mathsf{Q})$ and R_{Q_0} be $\mathtt{E}(\mathsf{Q}_0)$.

– Vertices : $\mathtt{I} = \mathtt{Dom}(\mathtt{h})$

– Edges: $\mathtt{J} = \{(\mathtt{i}, \mathtt{i}') \mid \mathtt{i}, \mathtt{i}' \in \mathtt{I} \text{ and } \mathtt{h}(\mathtt{i}) = \mathtt{i}'\}$. Note that each vertex has at most one outgoing edge.

– Labels: $\mathtt{L}((\mathtt{i}, \mathtt{i}')) \triangleq$

  * '<' if $\mathtt{i} \in R_P$ and $\mathtt{i} \notin R_Q$
  * '=' if $\mathtt{i} \in R_P$ and $\mathtt{i} \in R_Q$

- * '>' if $\mathtt{i} \notin R_P$ and $\mathtt{i} \in R_Q$
- * '#' if $\mathtt{i} \notin R_P$ and $\mathtt{i} \notin R_Q$

We define the equivalence relation $\sim$ on the vertices of $\mathbb{G}$ as follows: $\mathtt{i} \sim \mathtt{i}'$ if there are $\mathtt{i}_0, \mathtt{i}_1, \ldots, \mathtt{i}_n$ with $\mathtt{i} = \mathtt{i}_0$ and $\mathtt{i}_n = \mathtt{i}'$ such that $(\mathtt{i}_{i-1}, \mathtt{i}_i) \in \mathtt{J}$ and $\mathtt{L}((\mathtt{i}_{i-1}, \mathtt{i}_i))$ is '=' for $\mathtt{i} \in \{1, \ldots, \mathtt{n}\}$.

We define $\mathbb{G}' = (\mathtt{I}', \mathtt{J}')$ from $\mathbb{G}$ as follows:

- $\mathtt{I}' = \mathtt{I}/\sim$. Let the equivalence class of $\mathtt{i}$ be denoted by $[\mathtt{i}]$.
- $([\mathtt{i}], [\mathtt{i}']) \in \mathtt{J}'$ if and only if $(\mathtt{i}, \mathtt{i}') \in \mathtt{J}$ with $\mathtt{L}((\mathtt{i}, \mathtt{i}')) =$'<' or $(\mathtt{i}', \mathtt{i}) \in \mathtt{J}$ with $\mathtt{L}((\mathtt{i}', \mathtt{i})) =$'>'.

Definition 3.1.2.3. A graph $\mathbb{G}$ is said to have a *cycle* if there exist a sequence of vertices $\mathtt{j}_1, \mathtt{j}_2, \ldots \mathtt{j}_n$ such that each of $(\mathtt{j}_1, \mathtt{j}_2), (\mathtt{j}_2, \mathtt{j}_3), \ldots (\mathtt{j}_{n-1}, \mathtt{j}_n), (\mathtt{j}_n, \mathtt{j}_1)$ is an edge. In an edge labelled graph a cycle is said to be *increasing* (*decreasing*) if each edge in the cycle is labelled '<' or '=' ('>' or '='). It is said to be a *strictly increasing* (*strictly decreasing*) if in addition there is at least one edge which is marked '<' ('>'). A graph is said to be *acyclic* if there is no cycle in the graph.

A graph $\mathbb{G} = (\mathtt{I}, \mathtt{J})$ is said to have a *topological order* if there exists a map $\mathtt{ord} : \mathtt{I} \to \mathtt{Dat}$ such that if $(\mathtt{k}, \mathtt{j})$ is an edge then $\mathtt{ord}(\mathtt{k}) < \mathtt{ord}(\mathtt{j})$. An edge labelled graph $\mathbb{G} = (\mathtt{I}, \mathtt{J}, \mathtt{L})$ can be *assigned values respecting edge labels* if there is a map $\mathtt{ord} : \mathtt{I} \to \mathtt{Dat}$ such that:

- if $\mathtt{L}((\mathtt{i}, \mathtt{i}')) =$'<' then $\mathtt{ord}(\mathtt{i}) < \mathtt{ord}(\mathtt{i}')$,
- if $\mathtt{L}((\mathtt{i}, \mathtt{i}')) =$'>' then $\mathtt{ord}(\mathtt{i}) > \mathtt{ord}(\mathtt{i}')$,
- if $\mathtt{L}((\mathtt{i}, \mathtt{i}')) =$'=' then $\mathtt{ord}(\mathtt{i}) = \mathtt{ord}(\mathtt{i}')$,
- if $\mathtt{L}((\mathtt{i}, \mathtt{i}')) =$'#' then $\mathtt{ord}(\mathtt{i})$ and $\mathtt{ord}(\mathtt{i}')$ are incomparable.

Let us now state here the following well-known property of topological orders, see for instance [39].

Lemma 3.1.2.4. A directed graph is acyclic if and only if it has a topological order.

We can now state lemmas which will lead us to prove the soundness and completeness of the three constraints.

Lemma 3.1.2.5.

(a) $(\mathtt{s}, \mathtt{h}), \mathtt{E} \models_{\mathrm{SO}} \mathsf{cons1}(\mathsf{P}, \mathsf{Q}, \mathsf{Q}_0)$ if and only if $\mathbb{G}$ has no strictly increasing or strictly decreasing cycle (strictly monotonic).

(b) $(\mathtt{s}, \mathtt{h}), \mathtt{E} \models_{\mathrm{SO}} \mathsf{cons2}(\mathsf{P}, \mathsf{Q}, \mathsf{Q}_0)$, if and only if $R_P \cup R_Q = \{\mathtt{i} \in R_{Q_0} | \exists \mathtt{i}' \in R_{Q_0}, \mathtt{h}(\mathtt{i}) = \mathtt{i}'\}$.

(c) $(\mathtt{s}, \mathtt{h}), \mathtt{E} \models_{\mathrm{SO}} \mathsf{cons3}(\mathsf{P}, \mathsf{Q}, \mathsf{Q}_0)$ if and only if $R_{Q_0} = \mathtt{Dom}(\mathtt{h})$.

Proof.

(a) (Forward direction) Consider a directed graph where each vertex has at most outdegree one. In addition let each vertex have indegree at least one. As the sum of indegrees must be equal to sum of outdegrees, each vertex must have indegree and outdegree exactly equal to one. It is easy to see that such a graph is made of disjoint directed cycles.

Consider $R_{Q_1} \subseteq R_{Q_0}$. If $(s, h), [Q_1 \mapsto R_{Q_1}] \models_{SO} \mathsf{loop}(Q_1)$ then R_{Q_1} represents the vertices of a cycle in $\mathbb{G}$ as $\mathsf{setofloops}(Q_1)$ states that each vertex has indegree at least one. We have already noted that $\mathbb{G}$ has outdegree at most one. Hence R_{Q_1} must represent a set of disjoint cycles. Since no subset of R_{Q_1} satisfies this property, R_{Q_1} is a single cycle. Hence we have proved the forward direction of the lemma.

The formula $\mathsf{cons1}(P, Q, Q_0)$ says that: if all the edges of a cycle of $\mathbb{G}$ are labelled by '<' or '=' – in other words a cycle is an increasing cycle, then the edges are all labelled by '=' – in other words it is not a strictly increasing cycle. Hence, there are no strictly increasing cycles. Similarly, we can prove there are no strictly decreasing cycles.

(Other direction) Let $\mathbb{G}$ have no strictly monotonic cycle. Let $R_{Q_1} \subseteq R_{Q_0}$ represent vertices of a cycle in $\mathbb{G}$. $R_{Q_1} \subseteq R_P$ means that each edge is labelled with '<' or '=': the cycle is increasing. Since it can not be strictly increasing, all the edges are labelled with '='. Hence $R_{Q_1} \subseteq R_Q$. This shows that $(s, h), E[Q_1 \mapsto R_{Q_1}] \models_{SO} \mathsf{loop}(Q_1) \Rightarrow ((\forall x.Q_1(x) \Rightarrow P(x)) \Rightarrow (\forall x.Q_1(x) \Rightarrow Q(x)))$. Doing similarly for strictly decreasing cycles, we obtain an equivalence. Hence, if $\mathbb{G}$ has no cycle then $(s, h), E \models_{SO} \forall Q_1 \subseteq Q_0. \mathsf{loop}(Q_1) \Rightarrow ((\forall x.Q_1(x) \Rightarrow P(x)) \Leftrightarrow (\forall x.Q_1(x) \Rightarrow Q(x)))$.

(b) (Forward direction) From the given condition we know that a location i is in R_P or R_Q if and only if i is in R_{Q_0} and there is another location i' in R_{Q_0} such that $h(i) = i'$, which is the same as saying $R_P \cup R_Q$ is equal to $\{i \in R_{Q_0} | \exists i' \in R_{Q_0}, h(i) = i'\}$.

(Other direction) Let $R_P \cup R_Q = \{i \in R_{Q_0} | \exists i' \in R_{Q_0}, h(i) = i'\}$. Then $(s[x \mapsto i], h), E \models_{SO} Q_0(x) \wedge (\exists y.Q_0(y) \wedge x \hookrightarrow y)$ if and only if $i \in R_{Q_0}$ and there is $i' \in R_{Q_0}$ such that $h(i) = i'$, which happens if and only if $i \in R_P \cup R_Q$. Equivalently, $(s[x \mapsto i], h), E \models_{SO} P(x) \vee Q(x)$. Hence, $(s, h), E \models_{SO} \forall x.(Q_0(x) \wedge (\exists y.Q_0(y) \wedge x \hookrightarrow y)) \Leftrightarrow P(x) \vee Q(x)$.

(c) (Forward direction) From the given condition we know that for any location i, $(s[x \mapsto i], h), E \models_{SO} x \hookrightarrow \square \Leftrightarrow Q_0(x)$. Hence, $i \in R_{Q_0}$ if and only if there is i' such that $h(i) = i'$. In other words, $i \in R_{Q_0}$ if and only if $i \in \mathrm{Dom}(h)$.

(Other direction) Let $R_{Q_0} = \mathrm{Dom}(h)$. Then $(s[x \mapsto i], h), [Q_0 \mapsto R_{Q0}] \models_{SO} Q_0(x)$ if and only if $i \in R_{Q_0}$, which holds if and only if $i \in \mathrm{Dom}(h)$, which is equivalent to $(s[x \mapsto i], h) \models_{SO} x \hookrightarrow \square$. Hence, if $E(Q_0) = \mathrm{Dom}(h)$ then $(s[x \mapsto i], h), E \models_{SO} \forall x.Q_0(x) \Leftrightarrow x \hookrightarrow \square$.

□

Lemma 3.1.2.6. If $(s, h), E \models_{SO} \mathsf{cons}(P, Q, Q_0)$ then $\mathbb{G}$ has no edges labelled '#' and $\mathbb{G}'$ is acyclic.

Proof. Let (i, i') be an edge in $\mathbb{G}$. As $(s, h), E \models_{SO} \mathsf{cons}(P, Q, Q_0)$ and hence $(s, h), E \models_{SO} \mathsf{cons2}(P, Q, Q_0)$, from lemma 3.1.2.5 (b) we know that i in $R_P \cup R_Q$. Hence the possibility that (i, i') is labelled '#' is ruled out.

As $(s,h),E \models_{SO} cons(P,Q,Q_0)$ and hence $(s,h),E \models_{SO} cons1(P,Q,Q_0)$, we know from lemma 3.1.2.5 (a) that $\mathbb{G}$ has no strictly monotonic cycle. By contradiction, assume that $[i_1],[i_2],\ldots,[i_n]$ is a cycle in $\mathbb{G}'$. Hence there are locations $i'_0, i_1, i'_1, i_2, i'_2, \ldots, i_n, i'_n = i'_0$ such that $i_j \sim i'_j$, (i'_{j-1}, i_j) is labelled '<' in J or (i_j, i'_{j-1}) is labelled '>' in J for j in $\{1,2\ldots n\}$. Hence, there is an undirected cycle in $\mathbb{G}$: a path from i_1 to i'_1 and (i'_1, i_2), followed by a path from i_2 to i'_2 and $(i'_2, i_3), \ldots$, followed by a path from i_n to i'_n and (i'_n, i_1). Since all vertices in $\mathbb{G}$ have outdegree at most one, we are going to show that this undirected cycle is actually a cycle in $\mathbb{G}$ itself. Let $j_0, j_1, \ldots, j_n = j_0$ be such an undirected cycle. Without any loss of generality assume (j_1, j_0) is a directed edge.

Let k be the smallest index such that (j_{k+1}, j_k) is not an edge but (j_k, j_{k-1}) is. Then both (j_k, j_{k+1}) and (j_k, j_{k-1}) are edges which leads to a contradiction. Such a k exists as otherwise each of (j_k, j_{k-1}) would be an edge, which defines a directed cycle. This shows that each of the (i'_{k-1}, i_k) is an edge or each of the (i_k, i'_{k-1}) is an edge. In either case, it is a strictly monotonic cycle in $\mathbb{G}$, which is a contradiction. As a consequence, the assumption that there is a cycle in $\mathbb{G}'$ was wrong. □

Lemma 3.1.2.7. Let $\mathbb{G}$ have no edges labelled with a '#'. $\mathbb{G}'$ has a topological order if and only if $\mathbb{G}$ can be assigned values respecting edge labels.

Proof. Let $\mathbb{G}' = (I', J')$ have a topological order. Hence, there exists a function $ord' : I' \to Dat$ such that if $([i],[i']) \in J'$ then $ord'([i]) < ord'([i'])$. We define the map $ord : I \to Dat$ by $i \mapsto ord'([i])$. We now show ord assigns values respecting edge labels for $\mathbb{G} = (I, J, L)$. Let $j = (i, i') \in J$.

- If $L(j) =$'=' then $i \sim i'$. Hence $[i] = [i']$, which means $ord(i) = ord'([i]) = ord'([i']) = ord(i')$.
- If $L(j) =$'<' then $ord'([i]) < ord'([i'])$. As $ord(i) = ord'([i])$ and $ord(i') = ord'([i'])$ we obtain $ord(i) < ord(i')$.
- If $L(j) =$'>' then $ord'([i']) < ord'([i])$. As $ord(i) = ord'([i])$ and $ord(i') = ord'([i'])$ we obtain $ord(i) > ord(i')$.

Let $\mathbb{G} = (I, J, L)$ be a graph which can be assigned values respecting edge labels using $ord : I \to Dat$. Consider $ord' : I' \to Dat$ which maps $[i]$ to $ord(i)$. First we need to check that this map is well defined. Let $i \sim i'$, then there are $i_0, i_1, \ldots, i_n$ with $i = i_0$ and $i_n = i'$ such that $(i_{k-1}, i_k) \in J$ and $L((i_{k-1}, i_k))$ is '=', or $ord(i_{k-1}) = ord(i_k)$, for $k \in \{1,\ldots,n\}$. Hence $ord(i_0) = ord(i_1) = \ldots = ord(i_n)$. As expected, we showed if $i \sim i'$ then $ord(i) = ord(i')$. Finally, we need to check that ord is a topological order. If $([i],[i']) \in J'$, then $(i, i') \in J$ is labelled '<' or $(i', i) \in J$ is labelled '>'. In both cases, $ord(i) < ord(i')$. □

Lemma 3.1.2.8. Assume $R_{Q_0} = Dom(h)$, and $R_P \cup R_Q = \{i \in R_{Q_0} \mid \exists i' \in R_{Q_0}, h(i) = i'\}$. $\mathbb{G} = (I, J, L)$ can be assigned values respecting edge labels if and only if there is h' satisfying $Shape(h') = h$, $R_P = Dom^+(h')$, $R_Q = Dom^-(h')$ and $R_{Q_0} = Dom(h')$.

Proof.

(Forward direction) Given $\mathbb{G} = (\mathtt{I}, \mathtt{J}, \mathtt{L})$ that can be assigned values respecting edge labels using the map $\mathtt{ord} : \mathtt{I} \to \mathtt{Dat}$, we define a heap of simple memory state $\mathtt{h}'$ whose domain is the set $\mathtt{I}$ as follows:

$$\begin{array}{rcl} \mathtt{h} : \mathtt{Loc} & \rightharpoonup & \mathtt{Loc} \times \mathtt{Dat} \\ \mathtt{i} & \mapsto & (\mathtt{h(i)}, \mathtt{ord(i)}) \end{array}$$

It is well defined as $\mathtt{I} = \mathtt{Dom(h)} = \mathtt{Dom(h')}$. By definition $\mathtt{Shape(h')} = \mathtt{h}$. Also, $\mathrm{R}_{\mathrm{Q}_0} = \mathtt{Dom(h)}$, hence $\mathrm{R}_{\mathrm{Q}_0} = \mathtt{Dom(h')}$.

- If $\mathtt{i} \in \mathtt{Dom}^+\mathtt{(h')}$, then there are $\mathtt{i}', \mathtt{i}'' \in \mathtt{Loc}$ and $\mathtt{m}', \mathtt{m}'' \in \mathtt{Dat}$ such that $\mathtt{h'(i)} = (\mathtt{i}', \mathtt{m}')$, $\mathtt{h'(i')} = (\mathtt{i}'', \mathtt{m}'')$ and $\mathtt{m}' \leq \mathtt{m}''$. As $\mathtt{m}' = \mathtt{ord(i)}$, $\mathtt{m}'' = \mathtt{ord(i')}$ and $\mathtt{h(i)} = \mathtt{i}'$, it is clear that $\mathtt{ord(i)} \leq \mathtt{ord(i')}$. Hence, by the definition of $\mathbb{G}$, $\mathtt{i} \in \mathrm{R}_{\mathrm{P}}$.

- Let $\mathtt{i} \in \mathrm{R}_{\mathrm{P}}$. Let $\mathtt{i}'$ be such that $\mathtt{h(i)} = \mathtt{i}'$. As $\mathrm{R}_{\mathrm{P}} \subseteq \mathrm{R}_{\mathrm{Q}_0} = \mathtt{Dom(h)}$, actually $\mathtt{i}' \in \mathtt{Dom(h)}$. As $\mathtt{i} \in \mathrm{R}_{\mathrm{P}}$, the edge $(\mathtt{i}, \mathtt{i}') \in \mathtt{J}$ is labelled by '<' or '='. Hence $\mathtt{ord(i)} \leq \mathtt{ord(i')}$. Hence, $\mathtt{i} \in \mathtt{Dom}^+\mathtt{(h')}$.

This proves $\mathrm{R}_{\mathrm{P}} = \mathtt{Dom}^+\mathtt{(h')}$. The proof for $\mathrm{R}_{\mathrm{Q}} = \mathtt{Dom}^-\mathtt{(h)}$ is identical.

(Other direction) Given $\mathtt{h}'$, let us define $\mathtt{ord}$ as follows:

$$\begin{array}{rcl} \mathtt{ord} : \mathtt{I} & \to & \mathtt{Dat} \\ \mathtt{i} & \mapsto & \mathtt{snd(h(i))} \end{array}$$

If $(\mathtt{i}, \mathtt{i}') \in \mathtt{J}$ is labelled '=', then $\mathtt{i} \in \mathrm{R}_{\mathrm{P}}$ and $\mathtt{i} \in \mathrm{R}_{\mathrm{Q}}$. Hence, $\mathtt{i} \in \mathtt{Dom}^+\mathtt{(h')}$ and $\mathtt{i} \in \mathtt{Dom}^-\mathtt{(h')}$. As $\mathtt{Shape(h')} = \mathtt{h}$, we can state that $\mathtt{h'(i)} = (\mathtt{i}', \mathtt{ord(i)})$ and $\mathtt{h'(i')} = (\mathtt{i}'', \mathtt{ord(i')})$ for some $\mathtt{i}''$ in $\mathtt{Loc}$. From the definitions of $\mathtt{Dom}^+$ and $\mathtt{Dom}^-$ we know that $\mathtt{ord(i)} \leq \mathtt{ord(i')}$ and $\mathtt{ord(i)} \geq \mathtt{(i')}$. Hence $\mathtt{ord(i)} = \mathtt{ord(i')}$ as desired. The cases when the label is '<' or '>' are very similar and omitted. The case of label being '#' cannot happen as $\mathtt{Dat}$ has a total order. □

Lemma 3.1.2.9 (Constraints soundness)**.** If $(\mathtt{s}, \mathtt{h}), \mathtt{E} \models_{\mathrm{S0}} \mathsf{cons}(\mathrm{P}, \mathrm{Q}, \mathrm{Q}_0)$ then there is a heap $\mathtt{h}' : \mathtt{Loc} \rightharpoonup \mathtt{Loc} \times \mathtt{Dat}$ such that $\mathtt{Shape(h')} = \mathtt{h}$, $\mathtt{E}(\mathrm{Q}_0) = \mathtt{Dom(h')}$, $\mathtt{E}(\mathrm{P}) = \mathtt{Dom}^+\mathtt{(h')}$ and $\mathtt{E}(\mathrm{Q}) = \mathtt{Dom}^-\mathtt{(h')}$.

Proof. Let $(\mathtt{s}, \mathtt{h}), \mathtt{E} \models_{\mathrm{S0}} \mathsf{cons}(\mathrm{P}, \mathrm{Q}, \mathrm{Q}_0)$. By lemma 3.1.2.6, $\mathbb{G}$ has no edges labelled '#' and $\mathbb{G}'$ is acyclic. By lemma 3.1.2.4, $\mathbb{G}'$ has a topological ordering. By lemma 3.1.2.7, $\mathbb{G}$ can be assigned values respecting edge labels. From lemma 3.1.2.5 (b) and lemma 3.1.2.5 (c) we can satisfy the hypothesis of lemma 3.1.2.8. Then, by applying lemma 3.1.2.8, there is $\mathtt{h}'$ such that $\mathtt{Shape(h')} = \mathtt{h}$, $\mathtt{E}(\mathrm{P}) = \mathtt{Dom}^+\mathtt{(h')}$, $\mathtt{E}(\mathrm{Q}) = \mathtt{Dom}^-\mathtt{(h')}$ and $\mathtt{E}(\mathrm{Q}_0) = \mathtt{Dom(h')}$. □

We can now state that our encoding of the shape in the constraints is complete, in other words that any model with data can be encoded in a model without data and an environment satisfying our requirements.

Lemma 3.1.2.10 (Constraints completeness)**.** For all simple memory states with data $(\mathtt{s}, \mathtt{h}')$:

$$(\mathtt{s}, \mathtt{Shape(h')}), \mathrm{E}_{\mathtt{h}'} \models_{\mathrm{S0}} \mathsf{cons}(\mathrm{P}, \mathrm{Q}, \mathrm{Q}_0).$$

Proof. Let $\mathbb{G}$ be the edge labelled graph obtained from $(\mathtt{s}, \mathtt{Shape(h')}, \mathtt{E_{h'}})$. Let us show that $\mathbb{G}$ has no strictly increasing or decreasing cycle. By contradiction, let $\mathtt{i_0}, \mathtt{i_1}, \dots \mathtt{i_n}$ with $\mathtt{i_n} = \mathtt{i_0}$ be a strictly increasing or a strictly decreasing cycle. Without loss of generality, we can assume that it is a strictly increasing cycle, that is: for all $\mathtt{j} \in \{1, \dots \mathtt{n}\}$, $\mathtt{Shape(h')(i_{j-1})} = \mathtt{i_j}$ and for all $\mathtt{i_j} \in \mathtt{E_{h'}}(\mathsf{P})$, and at least one location is not in $\mathtt{E_{h'}}(\mathsf{Q})$. In other words, for all $\mathtt{j}$, $\mathtt{i_j} \in \mathtt{Dom^+(h')}$ and there is $\mathtt{k}$ such that $\mathtt{j_{k-1}} \notin \mathtt{Dom^-(h')}$. Let $\mathtt{m_1}, \dots, \mathtt{m_n}$ be the constants such that $\mathtt{h'(i_{j-1})} = (\mathtt{i_j}, \mathtt{m_j})$. Then, for all $\mathtt{j}$, $\mathtt{m_{j-1}} \leq \mathtt{m_j}$, in other words $\mathtt{m_n} \leq \mathtt{m_1} \leq \mathtt{m_2} \leq \dots \leq \mathtt{m_n}$. Hence all the $\mathtt{m_j}$ must be equal. This is a contradiction with $\mathtt{m_{k-1}} < \mathtt{m_k}$. Hence, $\mathbb{G}$ has no strictly increasing or decreasing cycle. By lemma 3.1.2.5 (a), $(\mathtt{s}, \mathtt{Shape(h')}), \mathtt{E_{h'}} \models_{\mathtt{SO}} \mathsf{cons1}(\mathsf{P}, \mathsf{Q}, \mathsf{Q_0})$.

Also, $\mathtt{E_{h'}}(\mathsf{P}) \cup \mathtt{E_{h'}}(\mathsf{Q}) = \mathtt{Dom^+(h')} \cup \mathtt{Dom^-(h')} = \{\mathtt{i} \mid$ there are $\mathtt{i'}, \mathtt{i''} \in \mathtt{Loc}$ and $\mathtt{o}, \mathtt{o'} \in \mathtt{Dat}$ such that $\mathtt{h'(i)} = (\mathtt{i'}, \mathtt{o})$, $\mathtt{h'(i')} = (\mathtt{i''}, \mathtt{o'})$ and $\mathtt{o} \leq \mathtt{o'}\} \cup \{\mathtt{i} \mid$ there are $\mathtt{i'}, \mathtt{i''} \in \mathtt{Loc}$ and $\mathtt{o}, \mathtt{o'} \in \mathtt{Dat}$ such that $\mathtt{h'(i)} = (\mathtt{i'}, \mathtt{o})$, $\mathtt{h'(i')} = (\mathtt{i''}, \mathtt{o'})$ and $\mathtt{o} \geq \mathtt{o'}\} = \{\mathtt{i} \mid$ there are $\mathtt{i'} \in \mathtt{Loc}$ and $\mathtt{o} \in \mathtt{Dat}$ such that $\mathtt{h'(i)} = (\mathtt{i'}, \mathtt{o})\} = \{\mathtt{i} \mid$ there is $\mathtt{i'} \in \mathtt{Loc}$ such that $\mathtt{Shape(h')(i)} = \mathtt{i'}\}$. By lemma 3.1.2.5 (b), $(\mathtt{s}, \mathtt{Shape(h')}), \mathtt{E_h} \models_{\mathtt{SO}} \mathsf{cons2}(\mathsf{P}, \mathsf{Q}, \mathsf{Q_0})$.

$\mathtt{Dom(Shape(h'))} = \mathtt{Dom(h')} = \mathtt{E_{h'}}(\mathsf{Q_0})$. By lemma 3.1.2.5 (c), $(\mathtt{s}, \mathtt{Shape(h')}), \mathtt{E_{h'}} \models_{\mathtt{SO}} \mathsf{cons3}(\mathsf{P}, \mathsf{Q}, \mathsf{Q_0})$.

As a consequence of the simultaneous satisfaction of $\mathsf{cons1}(\mathsf{P}, \mathsf{Q}, \mathsf{Q_0})$, $\mathsf{cons2}(\mathsf{P}, \mathsf{Q}, \mathsf{Q_0})$ and $\mathsf{cons3}(\mathsf{P}, \mathsf{Q}, \mathsf{Q_0})$ we can conclude that $(\mathtt{s}, \mathtt{Shape(h')}), \mathtt{E_{h'}} \models_{\mathtt{SO}} \mathsf{cons}(\mathsf{P}, \mathsf{Q}, \mathsf{Q_0})$. □

3.1.3 Recursive Translation

The auxiliary recursive translation $\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}$ is defined as follows: (1) it is homomorphic on the cases of $\mathsf{f} \wedge \mathsf{g}$, $\neg\mathsf{f}$, $\exists\mathsf{x}.\mathsf{f}$, and $\mathsf{x} = \mathsf{y}$, and (2) for other connectives, parameters $\mathsf{P}, \mathsf{Q}, \mathsf{Q_0}$ come into play:

$$
\begin{array}{lcl}
\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{f} \wedge \mathsf{g}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) & \triangleq & \mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) \\
 & & \wedge\ \mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{g}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) \\
\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\neg\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) & \triangleq & \neg\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) \\
\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\exists\mathsf{x}.\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) & \triangleq & \exists\mathsf{x}.\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) \\
\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{x} = \mathsf{y}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) & \triangleq & \mathsf{x} = \mathsf{y} \\
\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{x} \hookrightarrow \mathsf{y}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) & \triangleq & \mathsf{Q_0}(\mathsf{x}) \wedge \mathsf{x} \hookrightarrow \mathsf{y} \\
\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{x} \hookrightarrow_{\leq} \mathsf{y}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) & \triangleq & \mathsf{Q_0}(\mathsf{x}) \wedge \mathsf{Q_0}(\mathsf{y}) \wedge \mathsf{P}(\mathsf{x}) \wedge \mathsf{x} \hookrightarrow \mathsf{y} \\
\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{x} \hookrightarrow_{\geq} \mathsf{y}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) & \triangleq & \mathsf{Q_0}(\mathsf{x}) \wedge \mathsf{Q_0}(\mathsf{y}) \wedge \mathsf{Q}(\mathsf{x}) \wedge \mathsf{x} \hookrightarrow \mathsf{y} \\
\mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{f_1} * \mathsf{f_2}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) & \triangleq & \exists\mathsf{Q_{01}}, \mathsf{Q_{02}}. \\
 & & \mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{f_1}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_{01}}) \\
 & & \wedge\ \mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{f_2}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_{02}}) \\
 & & \wedge\ \mathsf{Q_0} = \mathsf{Q_{01}} \cup \mathsf{Q_{02}}\ \wedge\ \mathsf{Q_{01}} \cap \mathsf{Q_{02}} = \emptyset
\end{array}
$$

Lemma 3.1.3.1 (Reduction Lemma)**.** For all $\mathtt{s}, \mathtt{h'}$, for all $\mathtt{R_{Q_0}} \subseteq \mathtt{Dom(h')}$,

$$(\mathtt{s}, \mathtt{Shape(h')}), \mathtt{E_{h'}}[\mathsf{Q_0} \mapsto \mathtt{R_{Q_0}}] \models_{\mathtt{SO}} \mathtt{tr}_{\mathtt{SL^{short}_v} \to \mathtt{MSO}}(\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{Q_0}) \text{ if and only if } (\mathtt{s}, \mathtt{h'}_{|\mathtt{R_{Q_0}}}) \models_{\mathtt{SL}} \mathsf{f}.$$

Proof. In order to prove the lemma by induction, let us prove it for a formula f, assuming that it holds for all subformulas of f. Let $\mathtt{R_{Q_0}} \subseteq \mathtt{Dom(h')}$ and let $(\mathtt{s}, \mathtt{h'})$ be a simple memory state. We will show that the lemma holds also for f. Hence, by structural induction we will have proved the claim for f.

(Case when f is $\mathsf{f}_1 \wedge \mathsf{f}_2$.)

$$
\begin{array}{ll}
 & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{f}_1 \wedge \mathsf{f}_2, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \\
\text{iff} & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{f}_1, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \wedge \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{f}_2, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \\
\text{iff} & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{f}_1, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \text{ and } (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{f}_2, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \\
\text{iff} & (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}}) \models_{\mathrm{SL}} \mathsf{f}_1 \text{ and } (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}}) \models_{\mathrm{SL}} \mathsf{f}_2 \text{ (using the induction hypothesis)} \\
\text{iff} & (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}}) \models_{\mathrm{SL}} \mathsf{f}_1 \wedge \mathsf{f}_2
\end{array}
$$

(Case when f is $\neg\mathsf{g}$ or $\exists \mathsf{x}.\mathsf{g}$.) The proof is very similar and omitted.
(Case when f is $\mathsf{x} \hookrightarrow \mathsf{y}$.)

$$
\begin{array}{ll}
 & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{x} \hookrightarrow \mathsf{y}, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \\
\text{iff} & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathsf{Q}_0(\mathsf{x}) \wedge \mathsf{x} \hookrightarrow \mathsf{y} \\
\text{iff} & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathsf{Q}_0(\mathsf{x}) \text{ and } (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathsf{x} \hookrightarrow \mathsf{y} \\
\text{iff} & \mathtt{s}(\mathsf{x}) \in \mathrm{R}_{\mathsf{Q}_0} \text{ and } \mathtt{h}(\mathtt{s}(\mathsf{x})) = \mathtt{s}(\mathsf{y}) \\
\text{iff} & \mathtt{h}_{|\mathrm{R}_{\mathsf{Q}_0}}(\mathtt{s}(\mathsf{x})) = \mathtt{s}(\mathsf{y}) \\
\text{iff} & \mathtt{Shape}(\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}})(\mathtt{s}(\mathsf{x})) = \mathtt{s}(\mathsf{y}) \\
\text{iff} & (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}}) \models_{\mathrm{SL}} \mathsf{x} \hookrightarrow \mathsf{y}
\end{array}
$$

(Case when f is $\mathsf{x} \hookrightarrow_{\leq} \mathsf{y}$.)

$$
\begin{array}{ll}
 & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{x} \hookrightarrow \mathsf{y}, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \\
\text{iff} & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathsf{Q}_0(\mathsf{x}) \wedge \mathsf{Q}_0(\mathsf{y}) \wedge \mathsf{P}(\mathsf{x}) \wedge \mathsf{x} \hookrightarrow \mathsf{y} \\
\text{iff} & \mathtt{s}(\mathsf{x}) \in \mathtt{E}(\mathsf{P}) \cap \mathrm{R}_{\mathsf{Q}_0},\ \mathtt{s}(\mathsf{y}) \in \mathrm{R}_{\mathsf{Q}_0}, \text{ and } \mathtt{h}(\mathtt{s}(\mathsf{x})) = \mathtt{s}(\mathsf{y}) \\
\text{iff} & \mathtt{s}(\mathsf{x}) \in \mathtt{E}(\mathsf{P}),\ \mathtt{s}(\mathsf{y}) \in \mathrm{R}_{\mathsf{Q}_0}, \text{ and } \mathtt{h}_{|\mathrm{R}_{\mathsf{Q}_0}}(\mathtt{s}(\mathsf{x})) = \mathtt{s}(\mathsf{y}) \\
\text{iff} & \mathtt{s}(\mathsf{x}) \in \mathtt{Dom}^{+}(\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}}) \text{ and } \mathtt{Shape}(\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}})(\mathtt{s}(\mathsf{x})) = \mathtt{s}(\mathsf{y}) \\
\text{iff} & \text{there are } \mathsf{o}, \mathsf{o}' \in \mathtt{Dat} \text{ and } \mathtt{i}'' \in \mathtt{Loc} \text{ such that } \mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}}(\mathtt{s}(\mathsf{x})) = (\mathtt{s}(\mathsf{y}), \mathsf{o}), \\
 & \mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}}(\mathtt{s}(\mathsf{y})) = (\mathtt{i}'', \mathsf{o}') \text{ and } \mathsf{o} \leq \mathsf{o}' \\
\text{iff} & (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}}) \models_{\mathrm{SL}} \mathsf{x} \hookrightarrow_{\leq} \mathsf{y}
\end{array}
$$

(Case when f is $\mathsf{x} \hookrightarrow_{\geq} \mathsf{y}$.) The proof is identical and omitted.
(Case when f is $\mathsf{x} = \mathsf{y}$.) As the heap is not involved, the lemma holds obviously.
(Case when f is $\mathsf{f}_1 \star \mathsf{f}_2$.)

$$
\begin{array}{ll}
 & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{f}_1 \star \mathsf{f}_2, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \\
\text{iff} & \text{there are } \mathrm{R}_{\mathsf{Q}_{01}}, \mathrm{R}_{\mathsf{Q}_{02}} \subseteq \mathtt{Loc} \text{ such that } \mathrm{R}_{\mathsf{Q}_0} = \mathrm{R}_{\mathsf{Q}_{01}} \cup \mathrm{R}_{\mathsf{Q}_{02}},\ \mathrm{R}_{\mathsf{Q}_{01}} \cap \mathrm{R}_{\mathsf{Q}_{02}} = \emptyset, \\
 & (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{f}_1, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_{01}) \text{ and } (\mathtt{s},\mathtt{h}),\mathtt{E} \models_{\mathrm{SO}} \mathtt{tr}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{f}_1, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_{01}) \\
\text{iff} & \text{there are } \mathrm{R}_{\mathsf{Q}_{01}}, \mathrm{R}_{\mathsf{Q}_{02}} \subseteq \mathtt{Loc} \text{ such that } \mathrm{R}_{\mathsf{Q}_0} = \mathrm{R}_{\mathsf{Q}_{01}} \cup \mathrm{R}_{\mathsf{Q}_{02}},\ \mathrm{R}_{\mathsf{Q}_{01}} \cap \mathrm{R}_{\mathsf{Q}_{02}} = \emptyset, \\
 & (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_{01}}}), \models_{\mathrm{SL}} \mathsf{f}_1 \text{ and } (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_{02}}}), \models_{\mathrm{SL}} \mathsf{f}_2 \\
\text{iff} & \text{there are } \mathtt{h}_1, \mathtt{h}_2, \text{ such that } \mathtt{h}_{|\mathrm{R}_{\mathsf{Q}_0}} = \mathtt{h}_1 \star \mathtt{h}_2, \text{ and setting } \mathrm{R}_{\mathsf{Q}_{0i}} = \mathtt{Dom}(\mathtt{h}_{\mathtt{i}}) : \\
 & (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_{01}}}), \models_{\mathrm{SL}} \mathsf{f}_1 \text{ and } (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_{02}}}), \models_{\mathrm{SL}} \mathsf{f}_2 \\
\text{iff} & (\mathtt{s},\mathtt{h}'_{|\mathrm{R}_{\mathsf{Q}_0}}), \models_{\mathrm{SL}} \mathsf{f}_1 \star \mathsf{f}_2
\end{array}
$$

□

Lemma 3.1.3.2. For all formulas f of $\mathrm{SL}_v^{\mathtt{short}}$, there is $(\mathtt{s},\mathtt{h}')$ such that $(\mathtt{s},\mathtt{h}') \models_{\mathrm{SL}} \mathsf{f}$ if and only if there is $(\mathtt{s},\mathtt{h})$ such that $(\mathtt{s},\mathtt{h}) \models_{\mathrm{SO}} \mathtt{translation}_{\mathrm{SL}_v^{\mathtt{short}}\to\mathtt{MSO}}(\mathsf{f})$.

Proof. We are going to prove the two directions of the if and only if condition. Let f be a formula of $\mathtt{SL}_{\mathtt{v}}^{\mathtt{short}}$
(Forward direction.) Assume there is $(\mathtt{s},\mathtt{h}')$ such that $(\mathtt{s},\mathtt{h}') \models_{SL} \mathsf{f}$. Using lemma 3.1.2.10 we obtain:

$$(\mathtt{s}, \mathtt{Shape}(\mathtt{h}')), [\mathsf{P} \mapsto \mathtt{Dom}^{+}(\mathtt{h}'), \mathsf{Q} \mapsto \mathtt{Dom}^{-}(\mathtt{h}'), \mathsf{Q}_0 \mapsto \mathtt{Dom}(\mathtt{h}')] \models_{SO} \mathsf{cons}(\mathsf{P}, \mathsf{Q}, \mathsf{Q}_0)$$

Using lemma 3.1.3.1 we know that:

$$(\mathtt{s}, \mathtt{Shape}(\mathtt{h}')), [\mathsf{P} \mapsto \mathtt{Dom}^{+}(\mathtt{h}'), \mathsf{Q} \mapsto \mathtt{Dom}^{-}(\mathtt{h}'), \mathsf{Q}_0 \mapsto \mathtt{Dom}(\mathtt{h}')] \\ \models_{SO} \mathtt{tr}_{\mathtt{SL}_{\mathtt{v}}^{\mathtt{short}} \to \mathtt{MSO}}(\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0)$$

Combining above two statements we can state:

$$(\mathtt{s}, \mathtt{Shape}(\mathtt{h}')) \models_{SO} \exists \mathsf{P}. \exists \mathsf{Q}. \exists \mathsf{Q}_0 . \mathsf{cons}(\mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \wedge \mathtt{tr}_{\mathtt{SL}_{\mathtt{v}}^{\mathtt{short}} \to \mathtt{MSO}}(\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0)$$

In other words, $\mathtt{s}$ itself and $\mathtt{h} = \mathtt{Shape}(\mathtt{h}')$ are such that $(\mathtt{s}, \mathtt{h}) \models_{SO} \mathtt{translation}_{\mathtt{SL}_{\mathtt{v}}^{\mathtt{short}} \to \mathtt{MSO}}(\mathtt{f})$.
(Other direction.) Assume there is $(\mathtt{s}, \mathtt{h})$ such that $(\mathtt{s}, \mathtt{h}) \models_{SO} \mathtt{translation}_{\mathtt{SL}_{\mathtt{v}}^{\mathtt{short}} \to \mathtt{MSO}}(\mathtt{f})$. Hence there are sets R_{P}, R_{Q} and R_{Q_0} such that

$$(\mathtt{s}, \mathtt{h}), [\mathsf{P} \mapsto R_{\mathsf{P}}, \mathsf{Q} \mapsto R_{\mathsf{Q}}, \mathsf{Q}_0 \mapsto R_{\mathsf{Q}_0}] \models_{SO} \mathsf{cons}(\mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \quad (3.1)$$

$$(\mathtt{s}, \mathtt{h}), [\mathsf{P} \mapsto R_{\mathsf{P}}, \mathsf{Q} \mapsto R_{\mathsf{Q}}, \mathsf{Q}_0 \mapsto R_{\mathsf{Q}_0}] \models_{SO} \mathtt{tr}_{\mathtt{SL}_{\mathtt{v}}^{\mathtt{short}} \to \mathtt{MSO}}(\mathsf{f}, \mathsf{P}, \mathsf{Q}, \mathsf{Q}_0) \quad (3.2)$$

Using equation 3.1 and lemma 3.1.2.9 we can state that there is $\mathtt{h}' : \mathtt{Loc} \rightharpoonup \mathtt{Loc} \times \mathtt{Dat}$ with $\mathtt{Shape}(\mathtt{h}') = \mathtt{h}$, $R_{\mathsf{Q}_0} = \mathtt{Dom}(\mathtt{h}')$, $R_{\mathsf{P}} = \mathtt{Dom}^{+}(\mathtt{h}')$ and $R_{\mathsf{Q}} = \mathtt{Dom}^{-}(\mathtt{h}')$. Note that $\mathtt{h}_{|R_{\mathsf{Q}_0}}$ is $\mathtt{h}$ as $\mathtt{Dom}(\mathtt{h}') = \mathtt{Dom}(\mathtt{h})$.

Knowing equation 3.2, we can use lemma 3.1.3.1 with $\mathtt{h}'$, $\mathtt{s}$, $\mathtt{h}$ and $\mathtt{E} = [\mathsf{P} \mapsto R_{\mathsf{P}}, \mathsf{Q} \mapsto R_{\mathsf{Q}}, \mathsf{Q}_0 \mapsto R_{\mathsf{Q}_0}]$, which allows us to conclude that $(\mathtt{s}, \mathtt{h}') \models_{SL} \mathsf{f}$. □

Thanks to lemma 3.1.3.2 and lemma 2.1.1.1, we have established the announced result:

Theorem 3.1. The satisfiability problem for $\mathtt{SL}_{\mathtt{v}}^{\mathtt{short}}$ is decidable.

3.2 Long-Distance Comparisons

3.2.1 An Undecidability Result

We consider now a fragment of $\mathtt{SL}_{\mathtt{v}}$ with long-distance comparison.

Definition 3.2.1.1. We call $\mathtt{SL}_{\mathtt{v}}^{\mathtt{long}}$ the long-distance fragment of $\mathtt{SL}_{\mathtt{v}}$ defined by the following grammar:

$$\mathsf{f} ::= \neg \mathsf{f} \mid \mathsf{f} \wedge \mathsf{f} \mid \exists \mathsf{x}.\mathsf{f} \mid \exists \mathsf{v}.\mathsf{f} \mid \mathsf{x} \hookrightarrow \mathsf{y} \mid \mathsf{val}(\mathsf{x}) \leq \mathsf{v} \mid \mathsf{val}(\mathsf{x}) \geq \mathsf{v} \mid \mathsf{x} = \mathsf{y} \mid \mathsf{f} \star \mathsf{f}.$$

We are going to show that, without any further restriction, long-distance comparisons yield undecidability, even for a simpler fragment defined below.

Definition 3.2.1.2. We call $\mathtt{SL}_{\mathtt{v}}^{\mathtt{longeq}}$ the equality long-distance fragment of $\mathtt{SL}_{\mathtt{v}}^{\mathtt{long}}$ defined by the following grammar:

$$\mathsf{f} ::= \neg \mathsf{f} \mid \mathsf{f} \wedge \mathsf{f} \mid \exists \mathsf{x}.\mathsf{f} \mid \mathsf{x} \hookrightarrow \mathsf{y} \mid \mathsf{val}(\mathsf{x}) = \mathsf{val}(\mathsf{y}) \mid \mathsf{x} = \mathsf{y} \mid \mathsf{f} \star \mathsf{f}.$$

The proof of the undecidability of $\mathtt{SL}_{\mathtt{v}}^{\mathtt{longeq}}$ goes by reduction to the satisfiability problem of first-order formulas over data words. Before giving the intuition of the reduction, we first recall this logic. Note that, so far, we assumed a total order on $\mathtt{Dat}$, but this aspect is not essential for this reduction as equality only is considered, and one may think here of $\mathtt{Dat}$ as any arbitrary infinite set.

Definition 3.2.1.3. We assume a finite set $\mathtt{A}$. A finite *data word* is a sequence $\mathtt{wd} = \mathtt{wd}_1 \ldots \mathtt{wd}_n$, where $\mathtt{wd}_i = (a_i, o_i) \in \mathtt{A} \times \mathtt{Dat}$; we write $|\,\mathtt{wd}\,|$ to denote the length $n \in \mathbb{N}$ of $\mathtt{wd}$.

First-order logic over data words is defined by:

$$f ::= \neg f \mid f \wedge f \mid \exists x.f \mid a(x) \mid x = y + 1 \mid x \sim_{\mathsf{data}} y$$

where $a \in \mathtt{A}$. Variables are interpreted as positions in the word through a valuation $\mathtt{va} : \mathtt{Var} \to \{1, \ldots, |\,\mathtt{wd}\,|\}$, $+1$ is the successor function over $\mathbb{N}$, and $\sim_{\mathsf{data}}$ relates positions holding the same datum. More formally:

$\mathtt{wd}, \mathtt{va} \models_{\mathtt{FO}} \exists x.f$ if there is $n \in \{1, \ldots, |\,\mathtt{wd}\,|\}$ such that $\mathtt{wd}, \mathtt{va}[x \mapsto n] \models_{\mathtt{FO}} f$
$\mathtt{wd}, \mathtt{va} \models_{\mathtt{FO}} a(x)$ if $a_{\mathtt{va}(x)} = a$
$\mathtt{wd}, \mathtt{va} \models_{\mathtt{FO}} x = y + 1$ if $\mathtt{va}(x) = \mathtt{va}(y) + 1$
$\mathtt{wd}, \mathtt{va} \models_{\mathtt{FO}} x \sim_{\mathsf{data}} y$ if $o_{\mathtt{va}(x)} = o_{\mathtt{va}(y)}$

Lemma 3.2.1.4 (see [15], Prop. 27)**.** The satisfiability problem for a closed sentence of first-order logic over data words is undecidable.

In order to prove the undecidability of $\mathtt{SL}_{\mathtt{v}}^{\mathtt{longeq}}$ with the help of lemma 3.2.1.4, we are going to define a translation from First-order logic over data words to $\mathtt{SL}_{\mathtt{v}}^{\mathtt{longeq}}$ such that a formula f admits a data word model if and only if its translation admits a simple memory state model. A data word of length n is encoded as a list segment of length $2n$, placing the sequence of letters of $\mathtt{A}$ in the even positions, and the data sequence in odd positions. Then $x = y+1$ can be encoded by $y \hookrightarrow^2 x$, and $x \sim_{\mathsf{data}} y$ can be encoded by $\mathsf{val}(x) = \mathsf{val}(y)$.

Theorem 3.2. The satisfiability problem for $\mathtt{SL}_{\mathtt{v}}^{\mathtt{longeq}}$ is undecidable.

Proof. Without any loss of generality, we assume $\mathtt{A} = \{1, \ldots, n\}$. Let $\mathtt{wd} = (\mathtt{wd}(i), o_i)_{i=1,\ldots,m} \in (\mathtt{A} \times \mathtt{Dat})^*$ be a data word over $(\mathtt{A}, \mathtt{Dat})$. We are going to use the following distinct variables: $x_1, \ldots, x_n, z_3, z_4, y_1, \ldots, y_{2|\mathtt{wd}|}$.

We define the set of its heap representation as the set $\mathtt{He}(\mathtt{wd})$ of models (s, h) such that $\mathtt{Dom}(h) = s(\{x_1, \ldots, x_n, z_3, z_4, y_1, \ldots, y_{2|\mathtt{wd}|}\})$ and:

- $(s, h) \models_{\mathtt{SL}} \mathsf{val}(y_1) = \mathsf{val}(x_{\mathtt{wd}(1)}) \wedge \ldots \wedge \mathsf{val}(y_{2|\mathtt{wd}|-1}) = \mathsf{val}(x_{\mathtt{wd}(|\mathtt{wd}|)}) \wedge \bigwedge_{i \neq k} \mathsf{val}(x_i) \neq \mathsf{val}(x_k)$
- $\mathtt{fst}(h(s(x_1))) = s(x_2), \ldots, \mathtt{fst}(h(s(x_{n-1}))) = s(x_n)$
- $\mathtt{fst}(h(s(x_n))) = s(z_3)$
- $\mathtt{fst}(h(s(z_3))) = s(y_1)$
- Odd positions of $y_\square$: $\mathtt{fst}(h(s(y_1))) = s(y_2), \ldots, \mathtt{fst}(h(s(y_{2|\mathtt{wd}|-1})))) = s(y_{2|\mathtt{wd}|})$
- Even positions of $y_\square$: $h(s(y_2)) = (s(y_3), o_1), \ldots, h(s(y_{2|\mathtt{wd}|})) = (s(z_4), o_{|\mathtt{wd}|})$

Now we define a formula wms that recognizes exactly the memory states that encode the $(\mathtt{A},\mathtt{Dat})$ words. If we abbreviate $\mathsf{singles} \triangleq \forall z, z'. z \hookrightarrow z' \Rightarrow \neg z' \hookrightarrow \Box$. and $\mathsf{even}(x) \triangleq (\mathsf{singles} \wedge x \hookrightarrow \Box) \star (\mathsf{singles} \wedge x \hookrightarrow z_4)$:

$$\begin{aligned}\mathsf{wms} \triangleq\ & ((\mathsf{ls}(z_3, z_4) \ \wedge ((\mathsf{singles} \wedge z_3 \hookrightarrow \Box) \star (\mathsf{singles} \wedge \Box \hookrightarrow z_4))) \\ & \star (\mathsf{ls}(x_1, z_3) \wedge \textstyle\bigwedge_{i \neq k} \mathsf{val}(x_i) \neq \mathsf{val}(x_k) \wedge \bigwedge_{i<n} x_i \hookrightarrow x_{i+1} \wedge x_n \hookrightarrow z_3)) \\ & \wedge (\forall x. \mathsf{even}(x) \Rightarrow \textstyle\bigvee_i \mathsf{val}(x) = \mathsf{val}(x_i))\end{aligned}$$

The proof is organized as the successive proof of three propositions leading to the end of the proof.

(Proposition 1) $(\mathtt{s},\mathtt{h}) \models_{SL} \mathsf{wms}$ if and only if $(\mathtt{s},\mathtt{h}) \in \mathtt{He}(\mathtt{wd})$ for some data word $\mathtt{wd}$.

Assume $(\mathtt{s},\mathtt{h}) \models_{SL} \mathsf{wms}$. Then $(\mathtt{s},\mathtt{h})$ is a list segment from x_1 to z_3 made of two parts:

* A list segment from x_1 to z_3, of length $\mathtt{n}$, with, if we call z_3 x_{n+1}, for all $\mathtt{i} \leq \mathtt{n}$: $\mathtt{h}(\mathtt{s}(x_i)) = (\mathtt{s}(x_{i+1}), \mathtt{o}_i)$, for some $\mathtt{o}_i$. The values $\mathtt{o}_i$ are all distinct.
* A list segment from z_3 to z_4 such that:
  - There is in this list segment a 2-partition of the allocated locations such that: two consecutive locations do not belong to the same part, and z_3 and the predecessor of z_4 do not belong to the same partition.
  - In the even positions, any datum belongs to the set $\{\mathtt{o}_1, \ldots, \mathtt{o}_n\}$.

Then the length of the list segment $\mathsf{ls}(z_3, z_4)$ is even, and one can read on it a $(\mathtt{A},\mathtt{Dat})$ word $\mathtt{wd}$, for which $(\mathtt{s},\mathtt{h}) \in \mathtt{He}(\mathtt{wd})$ trivially holds. The converse implication is proved with the same arguments. This ends the proof of (Proposition 1)

We now associate to every formula f of first-order logic over data words a formula of $\mathtt{SL}_v^{\mathtt{longeq}}$ that we call $\mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\mathsf{f})$ as follows:

$$\begin{aligned}
\mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\neg \mathsf{f}) &\triangleq \neg \mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\mathsf{f}) \\
\mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\mathsf{f}_1 \wedge \mathsf{f}_2) &\triangleq \mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\mathsf{f}_1) \wedge \mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\mathsf{f}_2) \\
\mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\exists x. \mathsf{f}) &\triangleq \exists x. \mathsf{odd}(x) \wedge \mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\mathsf{f}) \\
\mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(a(x)) &\triangleq \mathsf{val}(x) = \mathsf{val}(x_a) \\
\mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(x = y + 1) &\triangleq \exists z. y \hookrightarrow z \wedge z \hookrightarrow x \\
\mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(x \sim_{\mathsf{data}} y) &\triangleq \exists x', y'. x \hookrightarrow x' \wedge y \hookrightarrow y' \wedge \mathsf{val}(x') = \mathsf{val}(y')
\end{aligned}$$

where $\mathsf{odd}(x)$ is $(\mathsf{singles} \wedge x \hookrightarrow \Box) \star (\mathsf{singles} \wedge z_3 \hookrightarrow \Box)$.

Let $\mathtt{s}_{\mathtt{va}}$ denote the valuation that maps x to the $(2 \times \mathtt{va}(x) - 1)$-th successor of z_3.

(Proposition 2) For all data words $\mathtt{wd}$ and valuations for first-order logic over data words $\mathtt{va}$, for all $(\mathtt{s},\mathtt{h})$ such that $(\mathtt{s},\mathtt{h}) \in \mathtt{He}(\mathtt{wd})$ and $\mathtt{s} = \mathtt{s}_{\mathtt{va}}$: $(\mathtt{s},\mathtt{h}) \models_{SL} \mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\mathsf{f})$ is equivalent to $\mathtt{wd}, \mathtt{va} \models_{\mathtt{FO}} \mathsf{f}$.

The proof of (Proposition 2) is by straightforward induction.

(Proposition 3) A formula f is satisfiable in first-order logic over data words if and only if $\mathsf{ls}(z_3, z_4) \ \wedge \mathsf{wms} \wedge \mathtt{tr}_{\mathtt{FO}_{\mathtt{data-wd}} \to \mathtt{SL}_v^{\mathtt{longeq}}}(\mathsf{f})$ admits a heap model in separation logic.

(Proposition 3) is a consequence of (Proposition 1) and (Proposition 2).

Let us finally stress that the formula in (Proposition 3) belongs to $\mathtt{SL}_v^{\mathtt{longeq}}$. This proves that the reduction is correct and ends the proof. □

3.2.2 Decidability of Guarded Long-Distance Comparisons

We now consider the fragment of formulas where each quantification over values is restricted to values stored in a finite set of cells. We have chosen these cells to be those pointed by the program variables.

Definition 3.2.2.1. We call $\mathtt{SL}_v^{\mathtt{guarded}}$ the guarded long-distance fragment of $\mathtt{SL}_v$ defined by the following grammar:

$$\begin{array}{ll} f ::= & \neg f \mid f \wedge f \mid \exists x.f \mid \exists v.\mathsf{val}(w) = v \wedge f \\ & \mid x \hookrightarrow_{\leq} y \mid x \hookrightarrow_{\geq} y \mid x \hookrightarrow y \mid \mathsf{val}(x) \leq v \mid \mathsf{val}(x) \geq v \mid x = y \mid f \star f. \end{array}$$

Note that guarded long-distance comparisons are quite weak: they just store the value of a program variable before the use of $\star$. Hence we need to add short-distance comparisons as basic predicates if we still want to use them.

Theorem 3.3. The satisfiability problem for $\mathtt{SL}_v^{\mathtt{guarded}}$ is decidable.

Let us first sketch the proof. We adapt the proof of theorem 3.1.3.2 by extending the notions of colored shapes and graphs of constraints. Every formula to be translated will have all its free variables in a finite subset $\mathtt{W} = \{w_1, \ldots, w_n\}$ of $\mathtt{Progvar}$. To every variable $w \in \mathtt{W}$, we associate two second-order variable P_w, Q_w. A colored shape will contain the same sets extended from the short-distance comparisons with sets R_w^P, R_w^Q which contain the allocated locations containing a datum respectively higher and lower than the one stored in w. A colored shape is then a tuple:

$$\mathtt{Cs} = ((\mathtt{s},\mathtt{h}),\ R^P, R^Q,\ R_{w_1}^P, R_{w_1}^Q, \ldots, R_{w_n}^P, R_{w_n}^Q)$$

where R_w^P, R_w^Q are finite sets of locations; it is *well defined* if $R^P \cup R^Q = \mathtt{Dom}(\mathtt{h}) \cap \mathtt{h}^{-1}(\mathtt{Dom}(\mathtt{h}))$ and $R_w^P \cup R_w^Q = \mathtt{Dom}(\mathtt{h})$ for every program variable w such that $\mathtt{s}(w) \in \mathtt{Dom}(\mathtt{h})$. Let $(\mathtt{s},\mathtt{h})$ be a fixed shape. We define the relation $\sim$ on $\mathtt{Dom}(\mathtt{h})$ as the smallest equivalence relation such that:

- if $\mathtt{k} \in R_w^P \cap R_w^Q$ and $\mathtt{s}(w) \in \mathtt{Dom}(\mathtt{h})$, then $\mathtt{s}(w) \sim \mathtt{k}$;
- if $\mathtt{h}(\mathtt{k}) = \mathtt{k}'$, and $\mathtt{k} \in R^P \cap R^Q$, then $\mathtt{k} \sim \mathtt{k}'$.

The graph of constraints associated to $\mathtt{Cs}$ is the pair $(\mathtt{J},\mathtt{K})$ where the vertex set $\mathtt{J}$ is the quotient of $\mathtt{Dom}(\mathtt{h})$ by $\sim$, and there is an edge from the equivalence class $[\mathtt{k}_1]$ to $[\mathtt{k}_2]$ if at least one of the following conditions holds:

- either there is $\mathtt{s}(w) \in [\mathtt{k}_1]$ and $\mathtt{k} \in [\mathtt{k}_2]$ such that $\mathtt{k} \in R_w^Q - R_w^P$;
- or there is $\mathtt{s}(w) \in [\mathtt{k}_2]$ and $\mathtt{k} \in [\mathtt{k}_1]$ such that $\mathtt{k} \in R_w^P - R_w^Q$;
- or there is $\mathtt{k} \in [\mathtt{k}_1], \mathtt{k}' \in [\mathtt{k}_2]$ such that $\mathtt{h}(\mathtt{k}) = \mathtt{k}'$ and $\mathtt{k} \in R^Q - R^P$;
- or there is $\mathtt{k} \in [\mathtt{k}_1], \mathtt{k}' \in [\mathtt{k}_2]$ such that $\mathtt{h}(\mathtt{k}') = \mathtt{k}$ and $\mathtt{k}' \in R^P - R^Q$.

It is possible to check that the graph of constraints and the acyclicity condition on it are $\mathtt{MSO}$ definable. We will then adapt the reduction of section 3.1: we guess the R_w^Ps and R_w^Qs at start and check we made a valid guess, and we extend the recursive translation $\mathtt{tr}_{\mathtt{SL}_v^{\mathtt{short}} \to \mathtt{MSO}}(f)$ to a new recursive translation $\mathtt{tr}_{\mathtt{SL}_v^{\mathtt{guarded}} \to \mathtt{MSO}}(f)$ with the following updates:

$$\begin{array}{lcl} \mathtt{tr}_{\mathtt{SL}_v^{\mathtt{guarded}} \to \mathtt{MSO}}(\exists v.\mathsf{val}(w) = v \wedge f) & \triangleq & \mathtt{tr}_{\mathtt{SL}_v^{\mathtt{guarded}} \to \mathtt{MSO}}(f[v \leftarrow \mathsf{val}(w)]) \\ \mathtt{tr}_{\mathtt{SL}_v^{\mathtt{guarded}} \to \mathtt{MSO}}(\mathsf{val}(x) \leq \mathsf{val}(w)) & \triangleq & Q_0(x) \wedge Q_0(w) \wedge Q_w(x) \wedge \neg P_w(x) \\ \mathtt{tr}_{\mathtt{SL}_v^{\mathtt{guarded}} \to \mathtt{MSO}}(\mathsf{val}(x) \geq \mathsf{val}(w)) & \triangleq & Q_0(x) \wedge Q_0(w) \wedge P_w(x) \wedge \neg Q_w(x) \end{array}$$

Proof of the Theorem

To prove theorem 3.3, we will associate, to every formula f of the guarded long-distance fragment, a formula $\mathtt{translation}_{SL_v^{guarded}\to MSO}(f)$, with:

$$\mathtt{translation}_{SL_v^{guarded}\to MSO}(f) \triangleq$$
$$\exists P, Q, P_{w_1}, Q_{w_1}, \dots, P_{w_n}, Q_{w_n}.\exists Q_0.\mathsf{cons}(P, Q, P', Q', Q_0) \wedge \mathtt{tr}_{SL_v^{guarded}\to MSO}(f, Q_0)$$

where $\mathsf{cons}(P, Q, P', Q', Q_0)$ ensures that we guessed a coloring that defines a colored shape for which it is possible to assign values, and $\mathtt{tr}_{SL_v^{guarded}\to MSO}(f, Q_0)$ is the translation of the formula on memory states to the colored shape.

(Constraints) Let us first introduce some abbreviations we will use in the sequel. If R_1 a unary predicate and R_2 is a binary relation, we abbreviate $\mathsf{Mona}_1(R_2, R_1)$ and $\mathsf{Mona}_2(R_1, R_2)$ for the predicates defined by a kind of composition:

- given R_2 binary and R_1 unary, $\mathsf{Mona}_1(R_2, R_1)(x)$ holds if and only if $\exists y.xR_2y \wedge R_1(y)$,
- similarly, given R_1 unary and R_2 binary, $\mathsf{Mona}_2(R_1, R_2)(x)$ holds if and only if $\exists y.yR_2x \wedge R_1(y)$.

We define the formula

$$\begin{array}{rcl} x \;\mathsf{tbc}\; y & \triangleq & Q_0(x) \wedge Q_0(y) \wedge \\ & \big(& \bigvee_{w\in \mathtt{Progvar}} (x = w \wedge P_w(y) \wedge Q_w(y)) \\ & \vee & (x \hookrightarrow y \wedge P(x) \wedge Q(x))\big) \end{array}$$

that defines the binary relation whose reflexive, symmetric, transitive closure is the equivalence $\sim$ we defined in section 3.2. We then define the formula:

$$x \sim y \;\triangleq\; \forall P_0.(P_0(x) \wedge \mathsf{Mona}_2(P_0, \mathsf{tbc}) \subseteq P_0 \wedge \mathsf{Mona}_1(\mathsf{tbc}, P_0) \subseteq P_0) \;=>\; P_0(y)$$

which characterizes $\sim$. Indeed, let R be a relation over integers and j be an integer. Assume that for some $k \in \mathbb{N}$ we have that for all $I \subseteq \mathbb{N}$, if $\mathsf{Mona}_1(R, I) \subseteq I$ and $I \subseteq \mathsf{Mona}_2(I, R)$ and $j \in I$, then $k \in I$. Then k belongs to the intersection of all the sets I such that $\mathsf{Mona}_1(R, I) \subseteq I$, $I \subseteq \mathsf{Mona}_2(I, R)$ and $j \in I$. Any of these sets contains the equivalence class of j for the reflexive, symmetric and transitive closure of R. Also the equivalence class of j itself is such a set. So the intersection of all these sets is the equivalence class of j. So k belongs to the equivalence class of j for the reflexive, symmetric and transitive closure of R.

Then we define the edge relation on the graph of constraints:

$$\begin{array}{rcl} x \;\mathsf{edge}\; y & \triangleq & \exists x', y'.x \sim x' \wedge y \sim y' \wedge \\ & \big(& (x' \hookrightarrow y' \wedge P(x') \wedge \neg Q(x')) \\ & \vee & (y' \hookrightarrow x' \wedge Q(y') \wedge \neg P(y')) \\ & \vee & \bigvee_{w\in \mathtt{Progvar}} (x' = w \wedge P_w(y') \wedge \neg Q_w(y')) \\ & \vee & \bigvee_{w\in \mathtt{Progvar}} (y' = w \wedge Q_w(x') \wedge \neg P_w(x'))\big) \end{array}$$

and its transitive closure $x \;\mathsf{edge}^+\; y \;\triangleq\; \exists z.\; x \;\mathsf{edge}\; z \;\wedge\; \forall P_0.(P_0(z) \wedge \mathsf{Mona}_2(P_0, \mathsf{edge}) \subseteq P_0) => P_0(y)$.

We finally define the $\mathsf{cons}(P, Q, P', Q', Q_0)$ formula as a conjunction :

$$\mathsf{cons1}(P, Q, P', Q', Q_0) \wedge \mathsf{cons2}(P, Q, P', Q', Q_0) \wedge \mathsf{cons3}(Q_0)$$

– the graph of constraints is acyclic $\mathsf{cons1}(\mathsf{P},\mathsf{Q},\mathsf{P}',\mathsf{Q}',\mathsf{Q}_0) \triangleq \neg\exists x.x \ \mathsf{edge}^+ \ x$.

– each edge that should be colored is colored with '<', '>' or '=':

$$\begin{array}{ll} \mathsf{cons2}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0,\mathsf{P}',\mathsf{Q}') \triangleq & \\ \forall x.(& (\mathsf{Q}_0(x) \Leftrightarrow (\mathsf{P}(x) \vee \mathsf{Q}(x)) \oplus (\exists y.x \hookrightarrow y \wedge \neg\mathsf{Q}_0(y))) \\ & \wedge(\mathsf{Q}_0(x) \Leftrightarrow (\mathsf{P}_{w_1}(x) \vee \mathsf{Q}_{w_1}(x)) \oplus (\neg\mathsf{Q}_0(w_1) \wedge \mathsf{P}_{w_1} = \mathsf{Q}_{w_1} = \emptyset)) \\ & \wedge \dots \\ & \wedge(\mathsf{Q}_0(x) \Leftrightarrow (\mathsf{P}_{w_n}(x) \vee \mathsf{Q}_{w_n}(x)) \oplus (\neg\mathsf{Q}_0(w_n) \wedge \mathsf{P}_{w_n} = \mathsf{Q}_{w_n} = \emptyset))) \end{array}$$

(where $\oplus$ denotes the exclusive or).

– R_{Q_0} is the domain of the heap: $\mathsf{cons3}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0) \triangleq \forall x.(x \hookrightarrow \square) \Leftrightarrow \mathsf{Q}_0(x)$.

With these definitions, we know by construction that $(\mathtt{s},\mathtt{h}),\mathtt{E} \models_{SO} \mathsf{cons}(\mathsf{P},\mathsf{Q},\mathsf{Q}_0,\mathsf{P}',\mathsf{Q}')$ if and only if the colored shape $\mathtt{Cs}(\mathtt{s},\mathtt{h},\mathtt{E}) \triangleq (\mathtt{s},\mathtt{h},\mathtt{E}(\mathsf{P}),\mathtt{E}(\mathsf{Q}),\mathtt{E}(\mathsf{P}_{w_1}),\dots,\mathtt{E}(\mathsf{Q}_{w_n}))$ is well defined and its associated graph of constraints is acyclic.

(Soundness and completeness) For a given memory state $(\mathtt{s},\mathtt{h})$, and a given program variable w, we define

$$\mathsf{P}_w^{(s,h)} = \begin{cases} \mathtt{i} \in \mathtt{Dom}(\mathtt{h}) : \mathtt{snd}(\mathtt{h}(\mathsf{l})) \geq \mathtt{snd}(\mathtt{h}(\mathtt{s}(\mathtt{w})))\} \text{ if } \mathtt{s}(\mathtt{w}) \in \mathtt{Dom}(\mathtt{h}) \\ \emptyset \text{ otherwise} \end{cases}$$

Moreover, we define $\mathsf{P}^{(s,h)} = \mathtt{Dom}^+(\mathtt{h})$, and $\mathsf{Q}_w^{(s,h)}$, $\mathsf{Q}^{(s,h)}$ correspondingly. This allows to define the colored shape associated to a memory state $\mathtt{s},\mathtt{h}$:

$$\mathtt{Cs}(\mathtt{s},\mathtt{h}) \triangleq (\mathtt{s},\mathtt{Shape}(\mathtt{h}),\mathsf{P}^{(s,h)},\mathsf{Q}^{(s,h)},\mathsf{P}_{w_1}^{(s,h)},\dots,\mathsf{Q}_{w_n}^{(s,h)}).$$

Finally, to a memory state $(\mathtt{s},\mathtt{h})$, we associate the environment

$$\mathtt{E}^{(s,h)} \triangleq [\mathsf{P} \mapsto \mathtt{Dom}^+(\mathtt{h}),\mathsf{Q} \mapsto \mathtt{Dom}^-(\mathtt{h}),\mathsf{P}_{w_1} \mapsto \mathsf{P}_{w_1}^{(s,h)},\dots,\mathsf{Q}_{w_n} \mapsto \mathsf{Q}_{w_n}^{(s,h)},\mathsf{Q}_0 \mapsto \mathtt{Dom}(\mathtt{h})].$$

Let us prove two propositions that ensure soundness and completeness of cons.

– (Constraints soundness) If $(\mathtt{s},\mathtt{h}),\mathtt{E} \models_{SO} \mathsf{cons}(\mathsf{P},\mathsf{Q},\mathsf{P}',\mathsf{Q}',\mathsf{Q}_0)$ then there is a $\mathtt{h}' : \mathtt{Loc} \rightharpoonup \mathtt{Loc} \times \mathtt{Dat}$ such that $\mathtt{Cs}(\mathtt{s},\mathtt{h},\mathtt{E}) = \mathtt{Cs}(\mathtt{s},\mathtt{h}')$ and $\mathtt{E} = \mathtt{E}^{(s,h')}$ on relevant variables.

  Assume $(\mathtt{s},\mathtt{h}),\mathtt{E} \models_{SO} \mathsf{cons}(\mathsf{P},\mathsf{Q},\mathsf{P}',\mathsf{Q}',\mathsf{Q}_0)$. Then the graph of constraints associated is acyclic. Thus it admits a topological ordering $\mathtt{ord} : (\mathtt{Dom}(\mathtt{h})/\sim) \mapsto \mathtt{Dat}$. It can be lifted to $\hat{\mathtt{ord}} : \mathtt{Dom}(\mathtt{h}) \mapsto \mathtt{Dat}$, and one defines $\mathtt{h}(\mathtt{i}) = (\mathtt{h}(\mathtt{i}),\hat{\mathtt{ord}}(\mathtt{i}))$. It is then straightforward to check that $\mathtt{Cs}(\mathtt{s},\mathtt{h},\mathtt{E}) = \mathtt{Cs}(\mathtt{s},\mathtt{h})$.

– (Constraints completeness) For all simple memory state $(\mathtt{s},\mathtt{h})$, its shape and its environment satisfy cons, that is $(\mathtt{s},\mathtt{Shape}(\mathtt{h})),\mathtt{E}^{(s,h)} \models_{SO} \mathsf{cons}(\mathsf{P},\mathsf{Q},\mathsf{P}',\mathsf{Q}',\mathsf{Q}_0)$.

  If $\mathtt{i} \sim \mathtt{i}'$ in the graph of constraints $\mathtt{Cs}(\mathtt{s},\mathtt{Shape}(\mathtt{h}),\mathtt{E})$, then there is a path from $\mathtt{i}$ to $\mathtt{i}'$ labeled with '=', hence $\mathtt{snd}(\mathtt{h}(\mathtt{i})) = \mathtt{snd}(\mathtt{h}(\mathtt{i}'))$. This allows to define $\mathtt{ord} : (\mathtt{Dom}(\mathtt{h})/\sim) \mapsto \mathtt{Dat}, [\mathtt{i}] \mapsto \mathtt{snd}(\mathtt{h}(\mathtt{i}))$. This obviously defines a topological order, thus $\mathtt{Cs}(\mathtt{s},\mathtt{Shape}(\mathtt{h}),\mathtt{E})$ is acyclic. Other conditions are obviously satisfied.

Reduction Let us now state the full translation $\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f, Q_0)$ of a formula f of separation logic to a formula of MSO. The invariant we achieve through this translation is :

Invariant: For all s, h, for all $R_{Q_0} \subseteq \mathtt{Dom(h)}$,
$(s, \mathtt{Shape(h)}, E_{s,h'}[Q_0 \mapsto R_{Q_0}] \models_{S0} \mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f, Q_0)$ iff $(s, h_{|R_{Q_0}}) \models_{SL} f$.

$$
\begin{array}{rcl}
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f \wedge g, Q_0) & \triangleq & \mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f, Q_0) \wedge \mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(g, Q_0) \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(\neg f, Q_0) & \triangleq & \neg\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f, Q_0) \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(\exists x.f, Q_0) & \triangleq & \exists x.\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f, Q_0) \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(x \hookrightarrow y, Q_0) & \triangleq & Q_0(x) \wedge x \hookrightarrow y \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(x \hookrightarrow_{\leq} y, Q_0) & \triangleq & Q_0(x) \wedge Q_0(y) \wedge P(x) \wedge x \hookrightarrow y \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(x \hookrightarrow_{\geq} y, Q_0) & \triangleq & Q_0(x) \wedge Q_0(y) \wedge Q(x) \wedge x \hookrightarrow y \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(x = y, Q_0) & \triangleq & x = y \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(\exists v.\mathsf{val}(w) = v \wedge f, Q_0) & \triangleq & \mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f[v \leftarrow \mathsf{val}(w)], Q_0) \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(\mathsf{val}(x) \leq \mathsf{val}(w), Q_0) & \triangleq & Q_0(x) \wedge Q_0(w) \wedge Q_w(x) \wedge \neg P_w(x) \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(\mathsf{val}(x) \geq \mathsf{val}(w), Q_0) & \triangleq & Q_0(x) \wedge Q_0(w) \wedge P_w(x) \wedge \neg Q_w(x) \\
\mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f_1 * f_2, Q_0) & \triangleq & \exists Q_{01}.\exists Q_{02}.\ Q_0 = Q_{0_1} \cup Q_{02}\ \wedge\ Q_{01} \cap Q_{02} = \emptyset \\
& & \wedge\ \mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f_1, P, Q, P', Q', Q_{01}) \\
& & \wedge\ \mathtt{tr}_{\mathrm{SL}_v^{\mathrm{guarded}}\to\mathtt{MSO}}(f_2, P, Q, P', Q', Q_{02})
\end{array}
$$

3.3 Magic Wand and Restricted Magic Wand

Even without data, the logic with the operator $-\!\!*$ was proved to be undecidable in the previous chapter. A decidable separation logic with a restricted magic wand was presented. Let us write again the definition of this binary operator, $-\!\!*_n$ (for n an integer). Unlike the plain operator $-\!\!*$, the quantification on disjoint heaps of $-\!\!*_n$ considers only heaps for which the cardinality of the domain is bounded by n. More formally, we define that $(s, h) \models_{SL} f_1 -\!\!*_n f_2$ if and only if for all h' such that $h' \perp h$ and $|\mathtt{Dom}(h')| \leq n$, if $(s, h') \models_{SL} f_1$ then $(s, h * h') \models_{SL} f_2$. It can be seen as an abbreviation of $(f_1 \wedge \neg\exists x_1, \ldots, x_{n+1}. \bigwedge_{i \neq j} x_i \neq x_j \wedge \bigwedge_i \exists y.x_i \hookrightarrow y) -\!\!* f_2$. In the sequel, we will prove that, in the context of heaps with data, $-\!\!*_1$ is sufficient to obtain undecidability.

Let R denote an arbitrary binary relation on Dat. For a given value o_1, we write $\{o, o\ RR\ o_1\}$ to denote the set of values $o \in \mathtt{Dat}$ such that: there is o_2 such that both $o\ R\ o_2$ and $o_2\ R\ o_1$. Let us call $\sim_R$ the equivalence relation defined as $o_1 \sim_R o'_1$ iff $\{o, o\ RR\ o_1\} = \{o, o\ RR\ o'_1\}$. We consider the atomic formula $\mathsf{val}(x)\ R\ \mathsf{val}(y)$ stating that values stored in x and y compare through R. Formally, $(s, h) \models_{SL} \mathsf{val}(x)\ R\ \mathsf{val}(y)$ iff there are $o_1, o_2 \in \mathtt{Dat}$ and $i, i' \in \mathtt{Loc}$ such that $h(s(x)) = (i, o_1)$, $h(s(y)) = (i', o_2)$, and $o_1\ R\ o_2$. We now introduce the relation $x \hookrightarrow_R y$ for $x \hookrightarrow_R y \triangleq x \hookrightarrow y \wedge \mathsf{val}(x)\ R\ \mathsf{val}(y)$

Definition 3.3.0.2. The logic $\mathrm{SL}_v^{R,-\!*_1}$ is defined by the grammar:

$$f ::= \neg f \mid f \wedge f \mid \exists x.f \mid x \hookrightarrow y \mid x \hookrightarrow_R y \mid x = y \mid f * f \mid f -\!\!*_1 f.$$

We are going to prove that satisfiability and validity problems are undecidable for $\mathrm{SL}_v^{R,-\!*_1}$, for any $R \in \{\leq, \geq, =, <, >\}$ - recall that in this chapter the studied orders are in any case total

orders. We will rely on the previous section, especially theorem 3.2, by simulating a long-distance equality. We first need the following fact:

Lemma 3.3.0.3. Let $R \in \{\leq, \geq, =, <, >\}$. Then $\sim_R$ has an infinite number of equivalence classes.

Proof. If $R \in \{<, >\}$. Let us first notice that $\mathtt{Dat}$ is infinite. Let $(o_i)_{i\in\mathbb{N}}$ be an infinite sequence such that $o_i R o_{i+1}$ for all i or such that $o_{i+1} R o_i$ for all i. If $(o_i)_{i\in\mathbb{N}}$ is such that $o_i R o_{i+1}$ for all i, then for all $i \in \mathbb{N}$ and for all $j > 0$, $o_i \in \{o', o' RR o_{i+2\times j}\}$, but $o_i \notin \{o', o' RR o_i\}$. So $\{\{o', o' RR o_i\}, i \in \mathbb{N}\}$ is an infinite set of distinct classes. Similarly, if $(o_i)_{i\in\mathbb{N}}$ is such that $o_{i+1} R o_i$ for all i then $\{\{o', o_i RR o'\}, i \in \mathbb{N}\}$ is an infinite set of distinct classes.

If $R \in \{\leq, \geq, =\}$. Then $o = o'$ implies clearly $\{o_1, o_1 RR o\} = \{o_1, o_1 RR o'\}$. Let us prove the other implication. Assume $\{o_1, o_1 RR o\} = \{o_1, o_1 RR o'\}$. We know that RR is reflexive. So $o \in \{o_1, o_1 RR o\}$, and since $\{o_1, o_1 RR o\} \subseteq \{o_1, o_1 RR o'\}$ we have $o RR o'$. Since R is transitive $o RR o'$ is true iff $o R o'$. Similarly $o' R o$. So $o = o'$. Hence $\sim_R$ is equality, and has infinitely many equivalence classes since $\mathtt{Dat}$ is infinite. □

Let $\sim$ be an equivalence relation on $\mathtt{Dat}$ with infinitely many equivalence classes.

Definition 3.3.0.4. Let us define the *equivalence long-distance fragment* by the grammar:

$$f ::= \neg f \mid f \wedge f \mid \exists x.f \mid x \hookrightarrow y \mid \mathsf{val}(x) \sim \mathsf{val}(y) \mid x = y \mid f * f \mid f \mathrel{-\!\!*_1} f.$$

Next lemma, a slight variation of theorem 3.2, also holds in this generalised framework:

Lemma 3.3.0.5. The satisfiability problem for the equivalence long-distance fragment is undecidable.

Proof. By the same encoding as the one of theorem 3.2, one may reduce a satisfiability problem of a first-order sentence over data words, where data is taken from the infinite quotient set $\mathtt{Dat}/\sim_R$, to the satisfiability problem for the equivalence long-distance fragment. □

Lemma 3.3.0.6. There is a formula $f_R(x, x') \in SL_V^{R, -\!*_1}$ such that for all simple memory states (s, h) with $\{s(x), s(x')\} \subseteq \mathtt{Dom}(h)$:

$$(s, h) \models_{SL} f_R(x, x') \text{ iff } (s, h) \models_{SL} \mathsf{val}(x) \sim_R \mathsf{val}(x')$$

Let us first sketch the proof. $f \stackrel{\neg}{-\!*}_1 g$ will abbreviate $\neg(f \mathrel{-\!\!*_1} \neg g)$. Then $(s, h) \models_{SL} f \stackrel{\neg}{-\!*}_1 g$ iff there is h' such that $(s, h') \models_{SL} f$, $(s, h * h') \models_{SL} g$ and $\mid \mathtt{Dom}(h') \mid \leq 1$. The operators $-\!*_1$ and $\stackrel{\neg}{-\!*}_1$ will be used to simulate restricted quantifications over $\mathtt{Dat}$, respectively universal and existential. Consider the formula f'_R:

$$\begin{array}{c} \exists x_1.\exists x_2.(\neg\exists x_3.x_1 \hookrightarrow x_3 \vee x_2 \hookrightarrow x_3) \\ \wedge (x_1 \hookrightarrow x_2) \mathrel{-\!\!*_1} ((\mathsf{val}(x_1)\ RR\ \mathsf{val}(x)) \Leftrightarrow (\mathsf{val}(x_1)\ RR\ \mathsf{val}(x'))) \end{array}$$

where $\mathsf{val}(x_1)\ RR\ \mathsf{val}(x)$ abbreviates $(x_2 \hookrightarrow x) \stackrel{\neg}{-\!*}_1 [x_1 \hookrightarrow_R x_2 \wedge x_2 \hookrightarrow_R x]$. The formula f'_R expresses that for all o_1, there is o_2 such that $o_1\ R\ o_2\ R\ \mathtt{snd}(h(s(x)))$ if and only if there is o_2 such that $o_1\ R\ o_2\ R\ \mathtt{snd}(h(s(x')))$, that is $\mathsf{val}(x) \sim_R \mathsf{val}(x')$. By lemma 3.3.0.3, proving that the semantics of the formula f'_R is actually the same as that of $\mathsf{val}(x) \sim_R \mathsf{val}(x')$ implies that the satisfiability problem of $SL_V^{R, -\!*_1}$ is an instance of the satisfiability problem of the equivalence long distance fragment. By lemma 3.3.0.5, the satisfiability problem of $SL_V^{R, -\!*_1}$ is undecidable. We now begin the full proof of lemma 3.3.0.6, where the actual f_R is a little different from f'_R.

Proof.
(Preliminary definitions) We assume $\{\mathtt{s(x)}, \mathtt{s(x')}\} \subseteq \mathtt{Dom(h)}$; let $\mathtt{o} = \mathtt{snd(h(s(x)))}$ and $\mathtt{o'} = \mathtt{snd(h(s(x')))}$.

We are going to first define a formula $f_0(x, x')$ which expresses that $\mathsf{val}(x) \mathrm{R} \mathsf{val}(x')$, where $o\mathrm{R}o'$ iff $\{o_1, o_1 RRo\} \subseteq \{o_1, o_1 RRo'\}$. Assume we have such a formula, then it is be easy to obtain $f(x, x') \triangleq f_0(x, x') \wedge f_0(x', x)$, meaning $\{o_1, o_1 RRo\} = \{o_1, o_1 RRo'\}$. Let us define this formula:

$$f_0(x, x') \triangleq \exists x_1. \exists x_2. [\neg \exists x_3. x_1 \hookrightarrow x_3 \vee x_2 \hookrightarrow x_3] \wedge [(x_1 \hookrightarrow x_2) \mathrel{-\!\!\circledast}_1 \\ (((x_2 \hookrightarrow x) \mathrel{-\!\!\circledast}_1 (x_1 \hookrightarrow_R x_2 \wedge x_2 \hookrightarrow_R x)) \Rightarrow ((x_2 \hookrightarrow x') \mathrel{-\!\!\circledast}_1 (x_1 \hookrightarrow_R x_2 \wedge x_2 \hookrightarrow_R x')))]$$

We define also $f_1 = ((x_2 \hookrightarrow x) \mathrel{-\!\!\circledast}_1 (x_1 \hookrightarrow_R x_2 \wedge x_2 \hookrightarrow_R x))$ and $f_2 = ((x_2 \hookrightarrow x') \mathrel{-\!\!\circledast}_1 (x_1 \hookrightarrow_R x_2 \wedge x_2 \hookrightarrow_R x'))$.

The formula $f_0(x, x')$ is satisfied by (s, h) iff there are i_1 and i_2 not belonging to $\mathtt{Dom(h)}$ such that, with $s' = s[x_1 \mapsto i_1; x_2 \mapsto i_2]$, for all h' with domain size at most 1 such that $(s', h') \models_{SL} x_1 \hookrightarrow x_2$: $(s', h' * h) \models_{SL} f_1 \Rightarrow f_2$.

The formula $f_0(x, x')$ is satisfied by (s, h) iff there are i_1 and i_2 not belonging to $\mathtt{Dom(h)}$ such that, with $s' = s[x_1 \mapsto i_1; x_2 \mapsto i_2]$, for all h' with domain size at most 1 such that $h'(s'(x_1)) = (s'(x_2), o_1)$ for some o_1: $(s', h' * h) \models_{SL} f_1 \Rightarrow f_2$.

The formula $f_0(x, x')$ is satisfied by (s, h) iff there are i_1 and i_2 not belonging to $\mathtt{Dom(h)}$ such that, for all $o_1 \in \mathtt{Dat}$, if $s' = s[x_1 \mapsto i_1; x_2 \mapsto i_2]$ and $h' = [i_1 \mapsto (i_1, o_1)]$ then: $(s', h' * h) \models_{SL} f_1 \Rightarrow f_2$.

(Semantics of f_1 *and* f_2*)* Assume $(s', h') \models_{SL} x_1 \hookrightarrow x_2$ and h' is a good candidate for $-\!\!*_1$, that is $h'(s'(x_1)) = (s'(x_2), o_1)$ for some o_1, and $\mathtt{Dom}(h') = \{s(x_1)\}$. Let us study f_1, and prove that $(s', h' * h) \models_{SL} f_1$ iff there is o_2 such that $o_1 R o_2$ and $o_2 R o$.

Assume $(s', h' * h)$ satisfies f_1. Then there is h''_1 with domain size at most 1 such that $(s', h''_1) \models_{SL} x_2 \hookrightarrow x$ and $(s', h * h' * h''_1) \models_{SL} x_1 \hookrightarrow_R x_2 \wedge x_2 \hookrightarrow_R x$. Since the domain size of h''_1 is at most 1 and $(s', h''_1) \models_{SL} x_2 \hookrightarrow x$, we know that $\mathtt{Dom}(h''_1) = \{s'(x_2)\}$ and $\mathtt{fst}(h''_1(s'(x_2))) = s'(x)$. So, since $(s', h * h' * h''_1) \models_{SL} x_1 \hookrightarrow_R x_2 \wedge x_2 \hookrightarrow_R x$, if we call $o_2 = \mathtt{snd}(h(i_2))$, we obtain $o_1 R o_2$ and $o_2 R o$. As a consequence, there is o_2 such that $o_1 R o_2$ and $o_2 R o$.

Assume there is o_2 such that $o_1 R o_2$ and $o_2 R o$. Since i_2 is unallocated in h, it is possible to find h''_1 such that $\mathtt{Dom}(h''_1) = \{s'(x_2)\}$ and $h''_1 \perp h$. Then let h''_1 be such a heap, with $h''_1(s'(x_2)) = (s'(x), o_2)$. Hence h''_1 has domain size 1, and is a good candidate for the extension of the heap in a formula with outermost operator $\mathrel{-\!\!\circledast}_1$. It actually satisfies the left-hand side of the operator: $(s', h''_1) \models_{SL} x_2 \hookrightarrow x$. Also, since x is allocated in h with $\mathtt{snd}(h(s'(x))) = o$ and $o_2 R o$: $(s', h * h' * h''_1) \models_{SL} x_2 \hookrightarrow_R x$. Finally, since x_2 is allocated in h''_1, with $\mathtt{snd}(h''_1(s'(x_2))) = o_2$ and $o_1 R o_2$: $(s', h * h' * h''_1) \models_{SL} x_1 \hookrightarrow_R x_2$. So $(s', h * h' * h''_1)$ satisfies f_1.

By the same reasoning, one can prove that $(s', h' * h)$ satisfies f_2 iff there is o_2 such that $o_1 R o_2$ and $o_2 R o'$. As a consequence, $(s', h' * h)$ satisfies $f_1 \Rightarrow f_2$ iff: if $o_1 RRo$ then $o_1 RRo'$.

(Conclusion) We have shown in (Preliminary definitions) that the formula $f_0(x, x')$ is satisfied by (s, h) iff there are i_1 and i_2 not belonging to $\mathtt{Dom(h)}$ such that, for all $o_1 \in \mathtt{Dat}$, if $s' = s[x_1 \mapsto i_1; x_2 \mapsto i_2]$ and $h' = [i_1 \mapsto (i_1, o_1)]$ then: $(s', h' * h) \models_{SL} f_1 \Rightarrow f_2$.

So, $(s, h) \models_{SL} f_0(x, x')$ iff there are i_1 and i_2 not belonging to $\mathtt{Dom(h)}$ such that, for all $o_1 \in \mathtt{Dat}$, if $s' = s[x_1 \mapsto i_1; x_2 \mapsto i_2]$ and $h' = [i_1 \mapsto (i_1, o_1)]$ then: if $o_1 RRo$ then $o_1 RRo'$.

In other words, $(s,h) \models_{SL} f_0(x,x')$ iff there are $i_1 \notin Dom(h)$ and $i_2 \notin Dom(h)$ such that for all o_1, if $o_1 RRo$ then $o_1 RRo'$.

Since it is always possible to find unallocated i_1 and i_2 as Loc is infinite and $Dom(h)$ is finite, and since i_1 and i_2 are not used in "for all o_1, if $o_1 RRo$ then $o_1 RRo'$", we can forget about them. $(s,h) \models_{SL} f_0(x,x')$ iff for all o_1, if $o_1 RRo$ then $o_1 RRo'$.

That is to say, $(s,h) \models_{SL} f_0(x,x')$ iff $\{o_1, o_1 RRo\} \subseteq \{o_1, o_1 RRo'\}$. This is exactly: $(s,h) \models_{SL} f_0(x,x')$ iff $val(x) \mathrel{R} val(x)'$. Then $f_R \triangleq f_0(x,x') \wedge f_0(x',x)$ has the semantics we expected. □

The following theorem is a direct consequence of lemma 3.3.0.6.

Theorem 3.4. For any $R \in \{\leq, \geq, <, >, =\}$, the validity and satisfiability problems for $SL_v^{R,-\!\!*_1}$ are undecidable.

Conclusion

Summary of this Chapter

We have given a wide picture of the decidability status of the satisfiability problem for separation logic dealing with data.

With the ability to describe lists and quantify over locations, allowing long-distance comparisons brings undecidability, and so does allowing the operator $-\!\!*$, even strongly restricted. Yet, there is a positive result: dropping these two features makes the satisfiability problem decidable, still being able to specify local reasoning and express properties about ordered recursive structures. The decidability also holds when a finite set of references can be compared to all the rest of the memory. The results are summarized in figure 3.4.

Related Work

First-order separation logic over heap models with at least two selectors is known to be undecidable even with no separating connectives, from the result of Calcagno, Yang and O'Hearn in [36] by containment of finite satisfiability for classical predicate logic with one binary relation – see Trakhtenbrot [88]. On the other hand we have proved first-order separation logic over heaps with one selector to be decidable when the magic wand is dropped in the previous chapter. We have studied in this chapter separation logic on models more complicated than one selector but simpler than two or more selectors, that are models with one selector plus data. To our knowledge, nothing was known about first-order separation logic with data before the initial publication of these results, during the research that led to this thesis.

Soon after was published the logic Strand by Madhusudan, Parlato and Qiu [74], which is dealing with a very similar model to SL_v; it describes recursive structures labelled with data thanks to monadic second-order variables representing labels plus one second-order variable representing the memory shape by edges. On top of this model of the memory, Strand allows monadic second-order quantification. Its satisfiability is decidable when provided with a class of models, for instance the class of the models representing one tree. The composite structures logic of Bouajjani et al. [18] is also related, as it deals with composite data structures, with easily computable postconditions and decidable satisfiability for a fragment; it is a more general framework as it can handle data structures with several selectors. The Celia tool of Bouajjani et

Undecidable	$\mathtt{SL}_{\mathtt{v}}^{\mathtt{long}}$ and $\mathtt{SL}_{\mathtt{v}}^{\mathtt{longeq}}$ – theorem 3.2
	$\mathtt{SL}_{\mathtt{v}}^{\mathtt{R},\rightarrow^{*}_{1}}$ for any $\mathtt{R} \in \{\leq, \geq, <, >, =\}$ – theorem 3.4
Decidable	$\mathtt{SL}_{\mathtt{v}}^{\mathtt{short}}$ – theorem 3.1
	$\mathtt{SL}_{\mathtt{v}}^{\mathtt{guarded}}$ – theorem 3.3

Figure 3.4: Decidability and undecidability results

al. [19, 20] deals more precisely with lists with data, and obtains decidability through abstract domains.

Our fragment $\mathtt{SL}_{\mathtt{v}}^{\mathtt{guarded}}$ is actually able to store the data in locations pointed by program variables so as to compare them with other data after a separation makes them unavailable. Hence, one can relate this work to other logics handling data, especially logics which can store an element from the data values so as to be able to compare it to others later – this feature can be called the *freeze quantifier*. As examples of such logics, one should mention LTL with freeze, studied by Demri and Lazić in [45]. About logics dealing with data, this work also relates with logics on data words, as we used the results of the work of Bojańczyk, Muscholl, Schwentick, Segoufin and David in [15] so as to prove theorem 3.3. These logics give clear boundaries on the expressiveness of logics containing lists and data if their creator wants them to remain decidable.

Perspectives

Some ways to restrict the full language are still unexplored, for instance bounding the amount of quantified variables. With the same hope to obtain decidability for satisfiability problems, one may look at extensions of our decidable fragment.

In particular, we expect our decidability results to extend to more complex data structures that would have a decidable MSO theory (trees, doubly-linked lists, lists of lists, and more generally tree-width bounded structures), and to more complex short-distance comparisons (such as n-th successor or brothers). The restrictions we set may for instance be sufficient to handle search-trees. About more complex data structures, the proof should be very similar, as long as a graph of constraints similar to the one we defined can also be encoded in sets. These sets would again be the value of MSO variables. About more complex short-distance comparisons, more categories of addresses would have to be defined and encoded in sets. Instead of only categorizing compared to the immediate successor with $\{<, >, =\}$ thanks to two sets as we described, we could categorize compared to the successor and its successor with $\{<, >, =\} \times \{<, >, =\}$ thanks to four sets, or even further with more sets.

We did not explore this possibility. If these ideas actually provide the expected results, it would show that a graph of constraints is a good general concept for logics dealing with sorted data structures.

Finally, our results are general for any totally ordered infinite set, and questions remain open about partially ordered sets.

Chapter 4

Reasoning about Sequences of Heaps

Introduction

Contribution of this Chapter

Our aim is to combine the features of temporal logics with the conciseness of separation logic for describing the behavior of programs manipulating pointers. We introduce a linear-time temporal logic to specify sequences of memory shapes with underlying assertion language based on the quantifier free separation logic $\mathrm{SL_s}$ inspired from the definition of Reynolds in [84]. Its models, sequences of memory shapes, can be seen as an abstraction of the evolution of the memory of a program during its execution. It is a many-dimensional logic, as it has a spatial dimension to describe memory shapes as well as a temporal dimension.

The formal definition of our logic is in figure 4.1. The explanation of this definition will be given in section 4.1. We call the obtained formalism, with both separation and temporal aspects, $\mathrm{LTL}^{\mathrm{mem}}$. Our logic addresses a very general notion of models, including the aspects of pointer arithmetic and recursive structures with records. We distinguish the satisfiability problems from the model-checking problems, as well as distinct subclasses of interesting programs, as for instance the programs without destructive update.

The most surprising result is the PSPACE-completeness of the satisfiability problems where the heap can vary in time, and we either drop the pointer arithmetic or the separation connectives. This result is especially tight, as both propositional LTL and static separation logic are already PSPACE-complete, as proved by Sistla and Clarke in [86] for propositional LTL and by Calcagno, Yang and O'Hearn in [36] for static separation logic. These results are obtained by reduction to the non-emptiness problem for Büchi automata on an alphabet made of symbolic memory shapes obtained by an abstraction that we show sound and complete, with a similar technique to that of Lozes in [71], used also by Calcagno, Gardner and Hague in [33].

Surprisingly, this abstraction method does not scale to the whole logic, due to a subtle interplay between separation connectives and pointer arithmetic. Moreover, we will show undecidability results for several problems, for instance satisfiability problems when the heap cannot vary. A summary of the numerous results can be found in the conclusion, and the path we will follow to prove half of the results is presented in figure 4.7.

Expressions	$\mathsf{expr} ::=$	$\mathsf{x} \mid \bigcirc \mathsf{expr}$
Atomic formulas	$\mathsf{atom} ::=$	$\mathsf{expr} = \mathsf{expr}' \mid \mathsf{expr} + \mathtt{i} \hookrightarrow^{\mathtt{l}} \mathsf{expr}'$
State formulas	$\mathsf{f} ::=$	$\mathsf{atom} \mid \mathsf{emp} \mid \mathsf{f} * \mathsf{g} \mid \mathsf{f} \mathrel{-\!\!*} \mathsf{g} \mid \mathsf{f} \wedge \mathsf{g} \mid \neg \mathsf{f}$
Temporal formulas	$\mathsf{t} ::=$	$\mathsf{f} \mid \bigcirc \mathsf{t} \mid \mathsf{t}\ \mathsf{until}\ \mathsf{u} \mid \mathsf{t} \wedge \mathsf{u} \mid \neg \mathsf{t}$

Semantics

$\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}}$	$\bigcirc \mathsf{t}$	iff	$\mathtt{mod}, \mathtt{n}+1 \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathsf{t}$.
$\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}}$	$\mathsf{t}\ \mathsf{until}\ \mathsf{u}$	iff	there is $\mathtt{n}_1 \geq \mathtt{n}$ s.t. $\mathtt{mod}, \mathtt{n}_1 \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathsf{u}$ and $\mathtt{mod}, \mathtt{n}' \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathsf{t}$ for all $\mathtt{n}' \in [\mathtt{n}, \mathtt{n}_1[$.
$\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}}$	$\mathsf{t} \wedge \mathsf{u}$	iff	$\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathsf{t}$ and $\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathsf{u}$.
$\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}}$	$\neg \mathsf{t}$	iff	not $\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathsf{t}$.
$\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}}$	f	iff	$\mathtt{s}'_n, \mathtt{h}_n \models_{\mathtt{SL}} \mathsf{f}[\bigcirc^k \mathsf{x} \leftarrow \langle \mathsf{x}, \mathtt{k} \rangle]$ where $\mathtt{mod} = (\mathtt{s}_n, \mathtt{h}_n)_{n \geq 0}$ and $\mathtt{s}'_n$ is defined by $\mathtt{s}'_n(\langle \mathsf{x}, \mathtt{k} \rangle) = \mathtt{s}_{n+k}(\mathsf{x})$.

Figure 4.1: The syntax and semantics of $\mathtt{LTL}^{\mathtt{mem}}$

Structure of the Chapter

We define our logic $\mathtt{LTL}^{\mathtt{mem}}$ and several of its fragments as well as decision problems in section 4.1. Section 4.2 introduces the symbolic memory shapes (useful in section 4.3) and presents the PSPACE-completeness of the satisfiability and model-checking problems for $\mathtt{SL}_s$ with pointer arithmetic. Section 4.3 is dedicated to the decidability proof of satisfiability for various fragments and its consequences for other problems. In section 4.4, we mention several seemingly optimal undecidability results by encoding computations of Minsky machines.

This chapter presents results originally published in [26], and in [28].

4.1 Preliminaries

4.1.1 Temporal Models and Programs

Temporal Models

Temporal models are infinite sequences of memory shapes, which means they are elements in $(\mathtt{Stores} \times \mathtt{Heaps}_s)^{\mathbb{N}}$ and they are understood as infinite computations of programs with pointer variables. We range over $\mathtt{mod}$ for a given model, and its $\mathtt{i}^{\text{th}}$ state $\mathtt{mod}(\mathtt{i})$ will be noted $(\mathtt{s}_\mathtt{i}, \mathtt{h}_\mathtt{i})$. In order to analyze computations from programs without destructive update, we shall also consider models with constant heap, that is elements in $\mathtt{Stores}^{\mathbb{N}} \times \mathtt{Heaps}_s$.

$$
\begin{array}{lll}
[\![w := w']\!] & (s,h) & \ni (s[w \mapsto s(w')],h). \\
[\![w := w' \to l]\!] & (s,h*\{i \mapsto \{l \mapsto j,\ldots\}\}) & \ni (s[w \mapsto j],h*\{i \mapsto \{l \mapsto j,\ldots\}\}) \\
 & & \text{with } s(w') = i \\
[\![w \to l := w']\!] & (s,h*\{i \mapsto \{l \mapsto j,\ldots\}\}) & \ni (s,h*\{i \mapsto \{l \mapsto s(w'),\ldots\}\}) \\
 & & \text{with } s(w) = i \\
[\![w := \mathtt{cons}(l_1 : w_1,\ldots,l_k : w_k)]\!] & (s,h) & \ni (s[w \mapsto i],h*\{i \mapsto \{l_1 \mapsto s(w_1), \ldots,l_k \mapsto s(w_k)\}\}) \\
 & & \text{with } i \notin \mathtt{Dom}(h) \\
[\![\mathtt{free}\ w]\!] & (s,h*\{i \mapsto \square\}) & \ni (s,h) \\
 & & \text{with } s(w) = i \\
[\![\mathtt{skip}]\!] & (s,h) & \ni (s,h) \\
[\![w := w'[i]]\!] & (s,h*\{i+i' \mapsto \{\mathtt{next} \mapsto j\}\}) & \ni (s[w \mapsto j], h*\{i+i' \mapsto \{\mathtt{next} \mapsto j\}\})) \\
 & & \text{with } s(w') = i' \\
[\![w[i] := w']\!] & (s,h*\{i'+i \mapsto \{\mathtt{next} \mapsto j\}\}) & \ni (s,h*\{i'+i \mapsto \{\mathtt{next} \mapsto s(w')\}\}) \\
 & & \text{with } s(w) = i' \\
[\![w := \mathtt{malloc}(i)]\!](s,h) & & \ni (s[w \mapsto i'], h*\{i' \mapsto \{\mathtt{next} \mapsto i''_1\},\ldots, i'+(i-1) \mapsto \{\mathtt{next} \mapsto i''_{i-1}\}\}) \\
 & & \text{with } i',\ldots,i'+(i-1) \notin \mathtt{Dom}(h) \\
 & & \text{and } i''_1,\ldots,i''_{i-1} \notin \mathtt{Dom}(h) \\
[\![\mathtt{free}\ w,i]\!] & (s,h*\{i'+i \mapsto \square\}) & \ni (s,h) \\
 & & \text{with } s(w) = i'
\end{array}
$$

Figure 4.2: Semantics for instructions

Instructions

The set Ins of *instructions* used in the programs is defined by the grammar below:

$$
\begin{array}{lll}
\mathtt{instr} ::= & x := y \mid \mathtt{skip} \\
 & \mid x := y \to l \mid x \to l := y \mid x := \mathtt{cons}(l_1 : x_1,\ldots,l_k : x_k) \mid \mathtt{free}\ x \\
 & \mid x := y[i] \mid x[i] := y \mid x := \mathtt{malloc}(i) \mid \mathtt{free}\ x,i
\end{array}
$$

The denotational semantics $[\![\mathtt{instr}]\!]$ of an instruction instr is defined as a binary relation $[\![\mathtt{instr}]\!] \subseteq (\mathtt{Stores} \times \mathtt{Heaps}_s) \times (\mathtt{Stores} \times \mathtt{Heaps}_s)$ in order to deal with the non-deterministic allocation of new memory cells. It can also be seen as a function from $(\mathtt{Stores} \times \mathtt{Heaps}_s)$ to $\mathtt{Pow}(\mathtt{Stores} \times \mathtt{Heaps}_s)$, and we write $[\![\mathtt{instr}]\!](s,h)$ to denote the image of (s,h) through this function. We list in figure 4.2 the formal denotational semantics of our instruction set.

Observe that the instructions $x := y[i]$, $x := \mathtt{malloc}(i)$ and $x[i] := y$ deal with the specific label next. Boolean combinations of equalities between variables are called guards and their set is denoted by Guards.

Programs

A program $\mathbb{P}$ is defined as a triple $(\mathtt{B}, \mathtt{d}, \mathtt{b_I})$ such that $\mathtt{B}$ is a finite set of control states, $\mathtt{b_I}$ is the initial state and $\mathtt{d}$ is the transition relation, a subset of $\mathtt{B}\times\mathtt{Guards}\times\mathtt{Ins}\times\mathtt{B}$. We use $\mathtt{b} \xrightarrow{\mathtt{g,instr}} \mathtt{b'}$ to denote a transition. We say that a program is *without destructive update* if transitions are labeled only with instructions of the form $\mathtt{x} := \mathtt{y}$, $\mathtt{x} := \mathtt{y} \to \mathsf{l}$, and $\mathtt{x} := \mathtt{y[i]}$. We write $\mathtt{Prog}$ to denote the set of programs and $\mathtt{Prog^{ct}}$ to denote the set of programs without destructive update.

A program is a finite object whose interpretation can be viewed as an infinite-state system. More precisely, given a program $\mathbb{P} = (\mathtt{B}, \mathtt{d}, \mathtt{b_I})$, the transition system $\mathbb{S}_{\mathbb{P}} = (\mathtt{Config}, \to)$ is defined as follows: $\mathtt{Config} = \mathtt{B}\times(\mathtt{Stores}\times\mathtt{Heaps_s})$ (set of configurations) and $(\mathtt{b}, (\mathtt{s}, \mathtt{h})) \to (\mathtt{b'}, (\mathtt{s'}, \mathtt{h'}))$ iff there is a transition $\mathtt{b} \xrightarrow{\mathtt{g,instr}} \mathtt{b'} \in \mathtt{d}$ such that $(\mathtt{s}, \mathtt{h}) \models \mathtt{g}$ and $(\mathtt{s'}, \mathtt{h'}) \in [\![\ \mathtt{instr}\ ]\!](\mathtt{s}, \mathtt{h})$. Note that $\mathbb{S}_{\mathbb{P}}$ is not necessarily deterministic. A computation (or execution) of $\mathbb{P}$ is defined as an infinite path in $\mathbb{S}_{\mathbb{P}}$ starting with control state $\mathtt{b_I}$.

4.1.2 Temporal Extension: our Logic

Our logic is a combination of $\mathtt{LTL}$ and $\mathtt{SL_s}$, a quantifier free fragment of $\mathtt{SL}$. These logics are combined so that the propositional variables of $\mathtt{LTL}$ are replaced by the formulas of $\mathtt{SL_s}$, which allow to describe the heap. While the operators of $\mathtt{LTL}$ allow to navigate in time, the separation logic formulas describe the present configuration of the heap only – with a limited ability to relate consecutive models. As the separation operators can only be under the scope of temporal operators, it is impossible to extend the compositionality principle of separation logic to $\mathtt{LTL^{mem}}$.

Formulas of $\mathtt{LTL^{mem}}$ are defined in figure 4.1 under the name *temporal formulas*. Their semantics is defined in the same figure; the satisfaction relation is $\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL^{mem}}} \mathtt{t}$, where $\mathtt{mod}$ is a temporal model, $\mathtt{n} \in \mathbb{N}$ and $\mathtt{t}$ is a formula. One can note that we use the notation $\bigcirc$ for the predicate called *next*, which is one of the notations found in the literature about $\mathtt{LTL}$ along with the capital letter X. We prefer this choice so as to avoid the use of single capital letters which describe sets in this thesis. Similarly, we use until rather than the capital letter U, and later we will use $\mathsf{sometimes}$ rather than the capital letter F and always rather than the capital letter G. Finally, one can note that we clearly distinguish state formulas from temporal formulas by using $\mathtt{f}$ and $\mathtt{g}$ for state formulas, as for separation logic formulas in the rest of the document, but $\mathtt{t}$ and $\mathtt{u}$ for temporal formulas.

The temporal operators are the standard next-time operator $\bigcirc$ and until operator until present in $\mathtt{LTL}$, see for instance [53, 86]. The formula $\bigcirc\mathtt{t}$ means that $\mathtt{t}$ holds in the next time state. The formula $\mathtt{t}\ \mathsf{until}\ \mathtt{u}$ means that either $\mathtt{u}$ holds or $\mathtt{t}$ holds and will continue holding until $\mathtt{u}$ holds at some moment in the future, and there exists be a moment in the future such that $\mathtt{u}$ holds. We use standard abbreviations such as $\mathsf{sometimes}(\mathtt{t})$ for $(\top\ \mathsf{until}\ \mathtt{t})$ or $\mathsf{always}(\mathtt{t})$ for $\neg\mathsf{sometimes}(\neg\mathtt{t})$.

State formulas of $\mathtt{LTL^{mem}}$ are formulas from the quantifier free separation logic $\mathtt{SL_s}$, except that variables can be prefixed by the symbol '$\bigcirc$' – see below for further explanation about this feature.

Contrarily to the two previous chapters, the atomic formulas allow pointer arithmetic, and an unbounded amount of fields through the labels. We allow to reach the pointer which is located $\mathtt{i}$ cells further than a variable, but we do not allow to check that a variable is located $\mathtt{i}$ cells further than another variable, as would a formula $\mathtt{x} + \mathtt{i} = \mathtt{y}$ which is not in our grammar.

It is important to notice that cells can be seen as having several selectors through this pointer arithmetic, but this is not related to the multiple selectors that may be available through labels.

The expression $\bigcirc\mathtt{x}$ is interpreted by the value of $\mathtt{x}$ at the next memory state. While keeping in mind that encoding $\bigcirc^{\mathtt{i}}\mathtt{x}$ requires memory space proportional to $\mathtt{i}$, we use the notation

$$\bigcirc^{\mathtt{i}}\mathtt{x} \triangleq \overbrace{\bigcirc \ldots \bigcirc}^{\mathtt{i}\ \text{times}}\mathtt{x}$$

One should note that the symbol '$\bigcirc$' is used in two different ways. When it is a temporal operator, it allows to state properties about next time state, forgetting about the present one. But when it is used in a state formula to prefix a variable, it is a way to compare the future value of a variable with the present memory state. As an example, $(\mathtt{s_i}, \mathtt{h_i})_{\mathtt{i}\in\mathbb{N}}, \mathtt{n} \models_{\mathtt{LTL^{mem}}} \bigcirc(\mathtt{x} \hookrightarrow \mathtt{y})$ holds when $\mathtt{h}_{\mathtt{n}+1}(\mathtt{next})(\mathtt{s}_{\mathtt{n}+1}(\mathtt{x})) = \mathtt{s}_{\mathtt{n}+1}(\mathtt{y})$, but $(\mathtt{s_i}, \mathtt{h_i})_{\mathtt{i}\in\mathbb{N}}, \mathtt{n} \models_{\mathtt{LTL^{mem}}} \bigcirc\mathtt{x} \hookrightarrow \bigcirc\mathtt{y}$ holds when $\mathtt{h_n}(\mathtt{next})(\mathtt{s}_{\mathtt{n}+1}(\mathtt{x})) = \mathtt{s}_{\mathtt{n}+1}(\mathtt{y})$. In the formula $\bigcirc(\mathtt{x} \hookrightarrow \mathtt{y})$ the symbol '$\bigcirc$' is a temporal operator. An example of formula using a temporal operator and the second use of '$\bigcirc$' can be $\mathsf{always}(\mathtt{x} = \bigcirc\mathtt{x})$, which means that the value of the variable $\mathtt{x}$ will never change. Indeed, $(\mathtt{s_i}, \mathtt{h_i})_{\mathtt{i}\in\mathbb{N}}, 0 \models_{\mathtt{LTL^{mem}}} \mathsf{always}(\mathtt{x} = \bigcirc\mathtt{x})$ iff for all $\mathtt{n}$, we have $\mathtt{s_n}(\mathtt{x}) = \mathtt{s}_{\mathtt{n}+1}(\mathtt{y})$.

Given an atomic formula $\mathtt{f}$, we write $\mathtt{f}[\bigcirc^{\mathtt{k}}\mathtt{x} \leftarrow \langle\mathtt{x}, \mathtt{k}\rangle]$ to denote the $\mathtt{SL_s}$ formula in which every occurrence of a term of the form $\bigcirc^{\mathtt{k}}\mathtt{x}$ is replaced by the variable $\langle\mathtt{x}, \mathtt{k}\rangle$. Similarly, given a formula $\mathtt{f}$, we write $\mathtt{f}[\mathtt{x} \leftarrow \langle\mathtt{x}, 0\rangle]$ to denote the state formula in which every occurrence of a variable $\mathtt{x}$ is replaced by $\langle\mathtt{x}, 0\rangle$.

We can freely use propositional variables, having in mind that a propositional variable should be understood as an ordinary variable, for instance $\mathtt{x} \in \mathtt{Var}$, whose equality tests with a fixed special variable $\mathtt{x}_\top \in \mathtt{Specialvar}$ encode the boolean value.

4.1.3 Satisfiability and Model-Checking

The fragments of the quantifier free separation logic $\mathtt{SL_s}$ we are going to mention are defined in section 1.3, Given a fragment $\mathtt{Frag}$ of $\mathtt{SL_s}$, such as $\mathtt{SL_s^{RF}}$ or $\mathtt{SL_s^{LF}}$, $\mathtt{LTL^{mem}}(\mathtt{Frag})$ is the restriction of $\mathtt{LTL^{mem}}$ to formulas in which occur only state formulas built over $\mathtt{Frag}$ with extended variables $\bigcirc^{\mathtt{k}}\mathtt{x}$. We write $\mathtt{Sat}(\mathtt{Frag})$ to denote the satisfiability problem for $\mathtt{LTL^{mem}}(\mathtt{Frag})$: given a temporal formula $\mathtt{t}$ in $\mathtt{LTL^{mem}}(\mathtt{Frag})$, is there a model $\mathtt{mod}$ such that $\mathtt{mod}, 0 \models_{\mathtt{LTL^{mem}}} \mathtt{t}$? The variant problem in which we require that the model has a constant heap is denoted by $\mathtt{Sat^{cons}}(\mathtt{Frag})$. The variant problem in which we require that the initial memory state is chosen beforehand as an input of the problem is denoted by $\mathtt{Sat_{init}}(\mathtt{Frag})$. The problem $\mathtt{Sat^{cons}_{init}}(\mathtt{Frag})$ is defined analogously.

The computations of a program can be viewed as $\mathtt{LTL^{mem}}$ models, using propositional variables to encode the extra information about the control states. As said above, so as to encode propositional variables, we use the special variable $\mathtt{x}_\top$. For each control state $\mathtt{b}$ we choose one corresponding variable $\mathtt{x_b}$, so that the propositional variable is true iff $\mathtt{x}_\top = \mathtt{x_b}$. Then one and only one of these propositional variables is true at a given moment, the one corresponding to the state in which the program is.

Model-checking aims at checking properties expressible in $\mathtt{LTL^{mem}}$ along computations of programs. To a logical fragment ($\mathtt{SL_s}$, $\mathtt{SL_s^{CL}}$, $\mathtt{SL_s^{RF}}$, or $\mathtt{SL_s^{LF}}$), we associate a set of programs :

- for $\mathtt{SL_s}$ and $\mathtt{SL_s^{CL}}$, all programs;

- for $\mathtt{SL}_s^{\mathtt{RF}}$, as this logic cannot handle pointer arithmetic, programs with instructions having the offset $\mathtt{i} = 0$;
- for $\mathtt{SL}_s^{\mathtt{LF}}$, as this logic can neither handle pointer arithmetic nor multiple labels in a cell, programs with instructions having the offset $\mathtt{i} = 0$ and only the label $\mathtt{next}$.

Given one of these fragments $\mathtt{Frag}$ of $\mathtt{SL_s}$, we write $\mathtt{Mc(Frag)}$ to denote the model-checking problem for $\mathtt{LTL^{mem}(Frag)}$: given a temporal formula $\mathtt{t}$ in $\mathtt{LTL^{mem}}$ with state formulas built over $\mathtt{Frag}$ and a program $\mathbb{P}$ of the associated set of programs, is there an infinite computation $\mathtt{mod}$ of $\mathbb{P}$ such that $\mathtt{mod}, 0 \models_{\mathtt{LTL^{mem}}} \mathtt{t}$ (which we write $\mathbb{P} \models_{\mathtt{LTL^{mem}}} \mathtt{t}$)? This is the existential variant of the problem. The variant problem in which we require that the program is without destructive update is denoted by $\mathtt{Mc^{cons}(Frag)}$. The variant problem in which we require that the initial memory state is chosen beforehand as an input of the problem is denoted by $\mathtt{Mc_{init}(Frag)}$. The problem $\mathtt{Mc^{cons}_{init}(Frag)}$ is defined analogously. We may write $\mathbb{P}, (\mathtt{s}, \mathtt{h}) \models_{\mathtt{LTL^{mem}}} \mathtt{t}$ to emphasize what is the initial memory state.

Our notations $\mathtt{Mc(Frag)}$ and $\mathtt{Sat(Frag)}$ for the problems about $\mathtt{LTL^{mem}(Frag)}$ should not be mistaken with the notations $\mathtt{Mcheck(Frag)}$ and $\mathtt{Satis(Frag)}$ for the problems about $\mathtt{Frag}$ itself, which will be defined in section 4.2.2.

4.1.4 Basic Results

Using extended variables $\bigcirc \mathtt{x}$, we may express some programs as formulas. This actually holds only for programs without destructive update, that is for the semantics with constant heap. Intuitively, we express the control of the program with propositional variables, and define a formula that encodes the transitions. As a consequence, the following result can be derived.

Lemma 4.1.4.1. Let $\mathtt{Frag}$ be a fragment among $\mathtt{SL_s}$, $\mathtt{SL}_s^{\mathtt{CL}}$, $\mathtt{SL}_s^{\mathtt{RF}}$, or $\mathtt{SL}_s^{\mathtt{LF}}$. There is a logarithmic space reduction from $\mathtt{Mc^{cons}(Frag)}$ to $\mathtt{Sat^{cons}(Frag)}$ (resp. from $\mathtt{Mc^{cons}_{init}(Frag)}$ to $\mathtt{Sat^{cons}_{init}(Frag)}$).

Proof. We adapt the proof in [86] for reducing $\mathtt{LTL}$ model-checking to $\mathtt{LTL}$ satisfiability. To a program $\mathbb{P} = (\mathtt{B}, \mathtt{d}, \mathtt{b_I})$, we associate the formula $\mathtt{t}_{\mathbb{P}}$ below built over the propositional variables in $\mathtt{B}$:

$$\mathtt{t}_{\mathbb{P}} \triangleq \mathtt{b_I} \wedge \mathsf{always} \bigwedge_{\mathtt{b} \in \mathtt{B}} (\mathtt{b} \Rightarrow (\bigwedge_{\mathtt{b}' \in \mathtt{B} \setminus \{\mathtt{b}\}} \neg \mathtt{b}' \wedge \bigvee_{\mathtt{transit} \in \mathtt{d}_{\mathtt{b}}^{+}} \mathtt{t_{transit}}))$$

where $\mathtt{t_{transit}}$ expresses that transition $\mathtt{transit}$ is fired between the current state and the next state and, $\mathtt{d}_{\mathtt{b}}^{+}$ is the set of transitions starting at the state $\mathtt{b}$. In order to define $\mathtt{t_{transit}}$, we need to translate instructions and guards into the logic (remember that there are limitations on the instructions). We translate instructions of the form

- $\mathtt{x} := \mathtt{y}$ into $\bigcirc \mathtt{x} = \mathtt{y}$,
- $\mathtt{x} := \mathtt{y} \rightarrow \mathtt{l}$ into $\mathtt{y} \hookrightarrow^{\mathtt{l}} \bigcirc \mathtt{x}$,
- $\mathtt{x} := \mathtt{y[i]}$ into $\mathtt{y} + \mathtt{i} \hookrightarrow^{\mathtt{next}} \bigcirc \mathtt{x}$.

Guards are translated accordingly. It is then standard to show that $\mathbb{P} \models_{\mathtt{LTL^{mem}}} \mathtt{t}$ iff $\mathtt{t} \wedge \mathtt{t}_{\mathbb{P}}$ is satisfiable. Indeed, it is sufficient to prove that for all models $\mathtt{mod}$, we have $\mathtt{mod} \models_{\mathtt{LTL^{mem}}} \mathtt{t}_{\mathbb{P}}$ iff $\mathtt{mod}$ is a computation of $\mathbb{P}$. It is obvious that computations of $\mathbb{P}$ satisfy $\mathtt{t}_{\mathbb{P}}$. Additionally, by a simple induction on time, one can easily show that $\mathtt{mod} \models_{\mathtt{LTL^{mem}}} \mathtt{t}_{\mathbb{P}}$ is a computation of $\mathbb{P}$. □

We now describe the reasons for which all the problems we have defined are PSPACE-hard and contained in Σ_1^1. In the analytical hierarchy, a problem (or equivalently a set of integers) is in Σ_1^1 if it is definable by a formula of second-order arithmetic with only existential set quantifiers in a prenex normal form. More information on this topic can be found in [78], for instance. All the model-checking and satisfiability problems defined in this chapter belong to Σ_1^1 in the analytical hierarchy. Indeed, the models and computations of programs can be viewed as functions $\mathbb{N} \to \mathbb{N}$: by encoding memory states and configurations by natural numbers, our infinite sequences of memory states can be encoded in sequences of integers. These mathematical objects are countable, and finding an actual injection from the set of memory states to $\mathbb{N}$ is not a challenge, although details would be tedious. Then, the satisfaction relation between models and $\mathtt{LTL}^{\mathtt{mem}}$ formulas and the transition relations obtained from programs can be encoded by a first-order formula. This guarantees that these problems are in Σ_1^1. Additionally, all the problems can easily be shown PSPACE-hard since they all generalize LTL satisfiability and model-checking [86].

4.2 Separation Logic: Complexity and Abstraction

After defining an abstraction for the fragment $\mathtt{SL}_{\mathtt{s}}^{\mathtt{RF}}$ of $\mathtt{SL}_{\mathtt{s}}$, which will be proved sound, we will be able to decide the complexity of model-checking and satisfiability problems for $\mathtt{SL}_{\mathtt{s}}$.

The main approach to get decision procedures to verify infinite-state systems consists of introducing a symbolic representation for infinite sets of configurations. The symbolic representation defined below is motivated by a similar goal. Given a formula t of $\mathtt{LTL}^{\mathtt{mem}}$, we are going to define its measure $\mathtt{mes}_{\mathsf{t}}$, understood as pieces of information about the syntactic resources involved in t. Indeed, forthcoming symbolic states are finite objects parametrized by such syntactic measures.

4.2.1 Syntactic Measures

Definitions of measures and related concepts

The method described below, using test formulas (see after the definition of measures), is inspired by [71].

We introduce a series of syntactic limitations of $\mathtt{LTL}^{\mathtt{mem}}$ formulas.

- For a state formula f of $\mathtt{LTL}^{\mathtt{mem}}$, the size of memory potentially examined by f, written $\mathtt{maxsize}_{\mathsf{f}}$, is inductively defined as follows: $\mathtt{maxsize}_{\mathsf{f}}$ is 2 for atomic formulas, $\mathtt{maxsize}_{\mathsf{f}_1}$ for $\neg\mathsf{f}_1$, and $\mathtt{maxsize}_{\mathsf{f}_1} + \mathtt{maxsize}_{\mathsf{f}_2}$ for $\mathsf{f}_1 * \mathsf{f}_2$, $\mathsf{f}_1 \wedge \mathsf{f}_2$ or $\mathsf{f}_1 \mathrel{-\!\!*} \mathsf{f}_2$. Observe that $\mathtt{maxsize}_{\mathsf{f}} \leq \mid \mathsf{f} \mid$, and that $\mathtt{maxsize}_{\mathsf{f}}$ is actually twice the amount of atomic formulas contained by f.
- $\mathtt{Lab}_{\mathsf{f}} \in \mathtt{Pow}_{\mathtt{fin}}(\mathtt{Lab})$ is the set of labels from $\mathtt{Lab}$ occurring in f.
- $\mathtt{Var}_{\mathsf{f}} \in \mathtt{Pow}_{\mathtt{fin}}(\mathtt{Var})$ is the set of variables from $\mathtt{Var}$ occurring in f.
- $\mathtt{Offsets}_{\mathsf{f}}$ ($\in \mathtt{Pow}_{\mathtt{fin}}(\mathbb{N})$) is the set of natural numbers $\mathtt{i}$ such that $\bigcirc^k \mathsf{x} + \mathtt{i} \hookrightarrow^l \mathsf{x}'$ occurs in f, where $\mathtt{Pow}_{\mathtt{fin}}(\mathtt{I})$ denotes the set of finite subsets of some set $\mathtt{I}$.

A *measure* $\mathtt{mes}$ can now be defined as a tuple $(\mathtt{Offsets}_{\mathtt{mes}}, \mathtt{maxsize}_{\mathtt{mes}}, \mathtt{Lab}_{\mathtt{mes}}, \mathtt{Var}_{\mathtt{mes}}) \in \mathtt{Pow}_{\mathtt{fin}}(\mathbb{N}) \times \mathbb{N} \times \mathtt{Pow}_{\mathtt{fin}}(\mathtt{Lab}) \times \mathtt{Pow}_{\mathtt{fin}}(\mathtt{Var})$. If $|\mathtt{Var}_{\mathtt{mes}}| \leq \mathtt{maxsize}_{\mathtt{mes}}$ then we say that $\mathtt{mes}$ is a *good measure*.

The set of measures has a natural lattice structure for the pointwise order, which we write $\mathtt{mes} \leqslant \mathtt{mes}'$. More precisely, this can be written $(\mathtt{Offsets}_{\mathtt{mes}}, \mathtt{maxsize}_{\mathtt{mes}}, \mathtt{Lab}_{\mathtt{mes}}, \mathtt{Var}_{\mathtt{mes}}) \leqslant (\mathtt{Offsets}_{\mathtt{mes}'}, \mathtt{maxsize}_{\mathtt{mes}'}, \mathtt{Lab}_{\mathtt{mes}'}, \mathtt{Var}_{\mathtt{mes}'})$ iff $\mathtt{Offsets}_{\mathtt{mes}} \subseteq \mathtt{Offsets}_{\mathtt{mes}'}$, $\mathtt{maxsize}_{\mathtt{mes}} \leq \mathtt{maxsize}_{\mathtt{mes}'}$, $\mathtt{Lab}_{\mathtt{mes}} \subseteq \mathtt{Lab}_{\mathtt{mes}}$ and $\mathtt{Var}_{\mathtt{mes}} \subseteq \mathtt{Var}_{\mathtt{mes}}$. We also write $\mathtt{mes}[\mathtt{maxsize} \leftarrow 0]$ to denote the measure $\mathtt{mes}$ except that the second component $\mathtt{maxsize}$ is 0. We write $\mathtt{size}(\mathtt{mes})$ to denote the size of the measure $\mathtt{mes}$ in some reasonable succinct encoding.

The measure of a state formula f, is the tuple $\mathtt{mes}_{\mathsf{f}} \triangleq (\mathtt{Offsets}_{\mathsf{f}}, \mathtt{maxsize}_{\mathsf{f}}, \mathtt{Lab}_{\mathsf{f}}, \mathtt{Var}_{\mathsf{f}})$. Note that for any formula f, we have $|\mathtt{Var}_{\mathtt{mes}_{\mathsf{f}}}| \leq \mathtt{maxsize}_{\mathtt{mes}_{\mathsf{f}}}$, so for any formula $\mathtt{mes}_{\mathsf{f}}$ is a good measure. The *measure of a temporal formula* t of $\mathtt{LTL}^{\mathtt{mem}}$, written $\mathtt{mes}_{\mathsf{t}}$, is the tuple $(\mathtt{Offsets}_{\mathsf{t}}, \mathtt{maxsize}_{\mathsf{t}}, \mathtt{Lab}_{\mathsf{t}}, \mathtt{Var}_{\mathsf{t}})$ where:

- $\mathtt{Offsets}_{\mathsf{t}} \triangleq \bigcup_{\mathsf{f} \text{ occurs in } \mathsf{t}} \mathtt{Offsets}_{\mathsf{t}}$,
- $\mathtt{maxsize}_{\mathsf{t}} \triangleq \sum_{\mathsf{f} \text{ occurs in } \mathsf{t}} \mathtt{maxsize}_{\mathsf{t}}$,
- $\mathtt{Lab}_{\mathsf{t}} \triangleq \bigcup_{\mathsf{f} \text{ occurs in } \mathsf{t}} \mathtt{Lab}_{\mathsf{t}}$,
- $\mathtt{Var}_{\mathsf{t}} \triangleq \bigcup_{\mathsf{f} \text{ occurs in } \mathsf{t}} \mathtt{Var}_{\mathsf{t}}$,

Note that the measure $\mathtt{mes}_{\mathsf{t}}$ of a temporal formula t is always a good measure. This notion will represent the syntactic resources used by a formula accurately enough for our purposes.

We now introduce the set of *test formulas*: each of them contains a piece of information about the model, and a set of these formulas will be used to abstract a heap. They are SL_{s} formulas of the forms below:

- $\mathsf{alloc}(\mathsf{x} + \mathtt{i}) \triangleq (\mathsf{x} + \mathtt{i} \mapsto^{\mathtt{next}} \mathsf{x}) \multimap \perp$ ($\mathsf{x} + \mathtt{i}$ is allocated).
- $\mathsf{size} \geq \mathtt{k} \triangleq \neg\mathsf{emp} \ast \ldots \ast \neg\mathsf{emp}$ with $\mathtt{k}$ times $\neg\mathsf{emp}$ (at least $\mathtt{k}$ indexes are allocated).
- $\mathsf{x} + \mathtt{i} \hookrightarrow^{\mathsf{l}} \mathsf{x}'$, $\mathsf{x} = \mathsf{x}'$ (see figure 1.4 for notations).

Given a measure $\mathtt{mes} = (\mathtt{Offsets}, \mathtt{maxsize}, \mathtt{Lab}_{\mathtt{mes}}, \mathtt{Var}_{\mathtt{mes}})$, we write $\mathtt{F}_{\mathtt{mes}}$ to denote the finite set of test formulas u defined as follows:

$$\mathsf{u} ::= \mathsf{x} + \mathtt{i} \hookrightarrow^{\mathsf{l}} \mathsf{x}' \mid \mathsf{alloc}(\mathsf{x}) \mid \mathsf{x} = \mathsf{x}' \mid \mathsf{size} \geq \mathtt{k}$$

with $\mathtt{i} \in \mathtt{Offsets}$, $\mathsf{l} \in \mathtt{Lab}_{\mathtt{mes}}$, $\mathtt{k} \in [0, \mathtt{maxsize}[$ and $\mathsf{x}, \mathsf{x}' \in \mathtt{Var}_{\mathtt{mes}}$.

Given a measure $\mathtt{mes} = (\mathtt{Offsets}, \mathtt{maxsize}, \mathtt{Lab}_{\mathtt{mes}}, \mathtt{Var}_{\mathtt{mes}})$ and a memory shape $(\mathtt{s}, \mathtt{h})$, we write $\mathtt{Abs}_{\mathtt{mes}}(\mathtt{s}, \mathtt{h}) = \{\mathsf{f} \in \mathtt{F}_{\mathtt{mes}} : (\mathtt{s}, \mathtt{h}) \models_{\mathsf{SL}} \mathsf{f}\}$ to denote the abstraction of $(\mathtt{s}, \mathtt{h})$ with respect to $\mathtt{mes}$. Given a measure $\mathtt{mes}$ and two memory shapes $(\mathtt{s}, \mathtt{h})$ and $(\mathtt{s}', \mathtt{h}')$, we write $(\mathtt{s}, \mathtt{h}) \simeq_{\mathtt{mes}} (\mathtt{s}', \mathtt{h}')$ iff $\mathtt{Abs}_{\mathtt{mes}}(\mathtt{s}, \mathtt{h}) = \mathtt{Abs}_{\mathtt{mes}}(\mathtt{s}', \mathtt{h}')$, that is, formulas in $\mathtt{F}_{\mathtt{mes}}$ cannot distinguish the two memory shapes. We will later show that a formula f such that $\mathtt{mes}_{\mathsf{f}} \leq \mathtt{mes}$ can not distinguish $(\mathtt{s}, \mathtt{h})$ and $(\mathtt{s}', \mathtt{h}')$ if $(\mathtt{s}, \mathtt{h}) \simeq_{\mathtt{mes}} (\mathtt{s}', \mathtt{h}')$.

Soundness of the Abstraction

Observe that the cardinal of $\mathtt{F}_{\mathtt{mes}_t}$ is polynomial in $\mid \mathtt{t} \mid$. The variable $\langle \mathtt{x}, \mathtt{k} \rangle$ will be used in subsequent developments to deal with the interpretation of the term $\bigcirc^k \mathtt{x}$ in the formulas of the temporal logic.

The proof of lemma 4.2.1.4 below is based on three technical lemmas. Before stating them and proving them, in lemmas 4.2.1.1-4.2.1.3, we assume that the measure has $\mathtt{Offsets} = \{0\}$ since these lemmas will be used for dealing with $\mathtt{SL}_s^{\mathtt{RF}}$. Moreover, we introduce the following definition:

$$(\mathtt{Offsets}, \mathtt{maxsize}_1, \mathtt{Lab}_0, \mathtt{Var}_1) + (\mathtt{Offsets}, \mathtt{maxsize}_2, \mathtt{Lab}_0, \mathtt{Var}_2) = \\ (\mathtt{Offsets}, \mathtt{maxsize}_1 + \mathtt{maxsize}_2, \mathtt{Lab}_0, \mathtt{Var}_1 \cup \mathtt{Var}_2)$$

Lemma 4.2.1.1 (Distributivity)**.** Let $\mathtt{mes}$ be a measure and $\mathtt{mes}_1$ and $\mathtt{mes}_2$ be good measures, with $\mathtt{mes} = \mathtt{mes}_1 + \mathtt{mes}_2$ and all sets of offsets equal to $\{0\}$. Let $(\mathtt{s}, \mathtt{h})$ and $(\mathtt{s}', \mathtt{h}')$ be memory shapes such that $(\mathtt{s}, \mathtt{h}) \simeq_{\mathtt{mes}} (\mathtt{s}', \mathtt{h}')$ and, $\mathtt{h}_1$, $\mathtt{h}_2$ be heaps such that $\mathtt{h} = \mathtt{h}_1 * \mathtt{h}_2$. Then, there exist heaps $\mathtt{h}'_1$ and $\mathtt{h}'_2$ with $\mathtt{h}' = \mathtt{h}'_1 * \mathtt{h}'_2$, $(\mathtt{s}, \mathtt{h}_1) \simeq_{\mathtt{mes}_1} (\mathtt{s}', \mathtt{h}'_1)$ and $(\mathtt{s}, \mathtt{h}_2) \simeq_{\mathtt{mes}_2} (\mathtt{s}', \mathtt{h}'_2)$.

Proof. Let $(\mathtt{s}, \mathtt{h})$, $(\mathtt{s}', \mathtt{h}')$, $\mathtt{h}_1$, $\mathtt{h}_2$ and the measures $\mathtt{mes} = (\mathtt{Offsets}, \mathtt{maxsize}, \mathtt{Lab}_0, \mathtt{Var}_0)$, $\mathtt{mes}_1 = (\mathtt{Offsets}, \mathtt{maxsize}_1, \mathtt{Lab}_0, \mathtt{Var}_1)$ and $\mathtt{mes}_2 = (\mathtt{Offsets}, \mathtt{maxsize}_2, \mathtt{Lab}_0, \mathtt{Var}_2)$ satisfy the hypotheses of the lemma.

We shall define the disjoint heaps $\mathtt{h}'_1$ and $\mathtt{h}'_2$ by distinguishing the four disjoint sets of locations $\mathtt{K}_1$, $\mathtt{K}_2$, $\mathtt{I}_1$ and $\mathtt{I}_2$ corresponding to the following sets:

- $\mathtt{K}_1 = \mathtt{Dom}(\mathtt{h}'_1) \cap \mathtt{Im}(\mathtt{s}')$, $\mathtt{K}_2 = \mathtt{Dom}(\mathtt{h}'_2) \cap \mathtt{Im}(\mathtt{s}')$,
- $\mathtt{I}_1 = \mathtt{Dom}(\mathtt{h}'_1) \setminus \mathtt{Im}(\mathtt{s}')$, $\mathtt{I}_2 = \mathtt{Dom}(\mathtt{h}'_2) \setminus \mathtt{Im}(\mathtt{s}')$.

Let us first separate $\mathtt{Dom}(\mathtt{h}') \cap \mathtt{Im}(\mathtt{s}')$ into two parts $\mathtt{K}_1$ and $\mathtt{K}_2$. For $\mathtt{i} \in \{1, 2\}$, we define $\mathtt{K}_\mathtt{i} \triangleq \mathtt{s}'(\mathtt{s}^{-1}(\mathtt{Dom}(\mathtt{h}_\mathtt{i}) \cap \mathtt{Var}_0)$, and we need to show that $\mathtt{K}_1$ and $\mathtt{K}_2$ are disjoint. Let us assume by contradiction that they are not, thus there are some variables $\mathtt{x}, \mathtt{y} \in \mathtt{Var}_0$ such that $\mathtt{s}'(\mathtt{x}) = \mathtt{s}'(\mathtt{y}) \in \mathtt{K}_1 \cap \mathtt{K}_2$, and $\mathtt{s}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h}_1)$ whereas $\mathtt{s}(\mathtt{y}) \in \mathtt{Dom}(\mathtt{h}_2)$. Since $\mathtt{h}_1 \perp \mathtt{h}_2$, $\mathtt{s}(\mathtt{x}) \neq \mathtt{s}(\mathtt{y})$, so $\mathtt{s}, \mathtt{h} \not\models_{\mathtt{SL}} \mathtt{x} = \mathtt{y}$, but we already know that $\mathtt{s}', \mathtt{h}' \models_{\mathtt{SL}} \mathtt{x} = \mathtt{y}$, hence the contradiction.

Now, we shall separate the set $\mathtt{Dom}(\mathtt{h}') \setminus \mathtt{Im}(\mathtt{s}')$ into two parts $\mathtt{I}_1$ and $\mathtt{I}_2$. Let $\mathtt{J} = \mathtt{Dom}(\mathtt{h}) \setminus \mathtt{Im}(\mathtt{s})$, $\mathtt{J}_1 = \mathtt{Dom}(\mathtt{h}_1) \setminus \mathtt{Im}(\mathtt{s})$ and $\mathtt{J}_2 = \mathtt{Dom}(\mathtt{h}_2) \setminus \mathtt{Im}(\mathtt{s})$. We have $|\mathtt{J}_1| + |\mathtt{K}_1| = |\mathtt{Dom}(\mathtt{h}_1)|$ and $|\mathtt{J}_2| + |\mathtt{K}_2| = |\mathtt{Dom}(\mathtt{h}_2)|$. The sets $\mathtt{I}_1$ and $\mathtt{I}_2$ shall contain respectively $|\mathtt{I}_1|$ and $|\mathtt{I}_2|$ random elements of $\mathtt{Dom}(\mathtt{h}') \setminus \mathtt{Im}(\mathtt{s}')$ so that $|\mathtt{I}_\mathtt{i}| = |\mathtt{Dom}(\mathtt{h}_\mathtt{i})| - |\mathtt{K}_\mathtt{i}|$ if $|\mathtt{Dom}(\mathtt{h})| < \mathtt{maxsize}$; otherwise $|\mathtt{I}_\mathtt{i}| = \mathtt{min}(\mathtt{maxsize}_\mathtt{i}, |\mathtt{Dom}(\mathtt{h}_\mathtt{i})|) - |\mathtt{K}_\mathtt{i}|$. In order to select the elements of $\mathtt{I}_1$ and $\mathtt{I}_2$, we distinguish different cases depending on $|\mathtt{Dom}(\mathtt{h}_1)|$ and $|\mathtt{Dom}(\mathtt{h}_2)|$.

Case 1: $|\mathtt{Dom}(\mathtt{h})| < \mathtt{maxsize}$.

Since $(\mathtt{s}, \mathtt{h}) \simeq_{\mathtt{mes}} (\mathtt{s}', \mathtt{h}')$, we have $|\mathtt{Dom}(\mathtt{h}')| = |\mathtt{Dom}(\mathtt{h})|$. Hence, $\mathtt{Dom}(\mathtt{h}') \setminus \mathtt{Im}(\mathtt{s}')$ can be divided into two parts $\mathtt{I}_1$, $\mathtt{I}_2$ such that $\mathtt{I}_1 \uplus \mathtt{I}_2 = \mathtt{Dom}(\mathtt{h}') \setminus \mathtt{Im}(\mathtt{s}')$, $|\mathtt{I}_1| = |\mathtt{Dom}(\mathtt{h}_1)| - |\mathtt{K}_1|$ and $|\mathtt{I}_2| = |\mathtt{Dom}(\mathtt{h}_2)| - |\mathtt{K}_2|$.

Case 2: $|\mathtt{Dom}(\mathtt{h})| \geq \mathtt{maxsize}$.

Consequently $|\mathtt{Dom}(\mathtt{h}')| \geq \mathtt{maxsize}$.

Case 2.1: $|\mathtt{Dom}(\mathtt{h}_1)| \geq \mathtt{maxsize}_1$ and $|\mathtt{Dom}(\mathtt{h}_2)| \geq \mathtt{maxsize}_2$.

There exist $\mathtt{I}_1$, $\mathtt{I}_2$ such that $\mathtt{I}_1 \uplus \mathtt{I}_2 = \mathtt{Dom}(\mathtt{h}') \setminus \mathtt{Im}(\mathtt{s}')$, $|\mathtt{I}_1| = |\mathtt{Dom}(\mathtt{h}_1)| - |\mathtt{K}_1| \geq \mathtt{maxsize}_1 - |\mathtt{K}_1|$ and $|\mathtt{I}_2| = |\mathtt{Dom}(\mathtt{h}_2)| - |\mathtt{K}_2| \geq \mathtt{maxsize}_2 - |\mathtt{K}_2|$.

Case 2.2: For some $\mathtt{i} \in \{1,2\}$, $|\mathtt{Dom}(\mathtt{h_i})| < \mathtt{maxsize_i}$ and $|\mathtt{Dom}(\mathtt{h_{3-i}})| \geq \mathtt{maxsize_{3-i}}$. There exist $\mathtt{I_1}$, $\mathtt{I_2}$ such that $\mathtt{I_1} \uplus \mathtt{I_2} = \mathtt{Dom}(\mathtt{h'}) \setminus \mathtt{Im}(\mathtt{s'})$, $|\mathtt{I_i}| = |\mathtt{Dom}(\mathtt{h_i})| - |\mathtt{K_i}|$. Then $|\mathtt{I_{3-i}}| = |\mathtt{Dom}(\mathtt{h_{3-i}})| - |\mathtt{K_{3-i}}| \geq \mathtt{maxsize_{3-i}} - |\mathtt{K_{3-i}}|$.

The heap $\mathtt{h'_1}$ is defined as $\mathtt{h'}_{|\mathtt{I_1} \cup \mathtt{K_1}}$ and the heap $\mathtt{h'_2}$ is defined as $\mathtt{h'}_{|\mathtt{I_2} \cup \mathtt{K_2}}$. Since $\mathtt{I_1}$, $\mathtt{I_2}$, $\mathtt{K_1}$ and $\mathtt{K_2}$ are disjoint sets, we have that $\mathtt{I_1} \cup \mathtt{K_1}$ and $\mathtt{I_2} \cup \mathtt{K_2}$ are disjoint. Moreover, $\mathtt{I_1} \cup \mathtt{I_2} \cup \mathtt{K_1} \cup \mathtt{K_2} = \mathtt{Dom}(\mathtt{h'})$. So $\mathtt{h'} = \mathtt{h'_1} * \mathtt{h'_2}$. Observe that for $\mathtt{i} \in \{1,2\}$, we have $|\mathtt{Dom}(\mathtt{h_i})| \geq \mathtt{maxsize_i}$ iff $|\mathtt{Dom}(\mathtt{h'_i})| \geq \mathtt{maxsize_i}$; also $|\mathtt{Dom}(\mathtt{h_i})| < \mathtt{maxsize_i}$ implies $|\mathtt{Dom}(\mathtt{h_i})| = |\mathtt{Dom}(\mathtt{h'_i})|$.

It remains to show that $(\mathtt{s},\mathtt{h_1}) \simeq_{\mathtt{mes_1}} (\mathtt{s'},\mathtt{h'_1})$ and $(\mathtt{s},\mathtt{h_2}) \simeq_{\mathtt{mes_2}} (\mathtt{s'},\mathtt{h'_2})$. The above considerations about cardinality entail that for all $\mathtt{i} \in \{1,2\}$ and $\mathtt{k} < \mathtt{maxsize_i}$, we have $\mathsf{size} \geq \mathtt{k} \in \mathtt{Abs}_{\mathtt{mes_i}}(\mathtt{s},\mathtt{h_i})$ iff $\mathsf{size} \geq \mathtt{k} \in \mathtt{Abs}_{\mathtt{mes_i}}(\mathtt{s'},\mathtt{h'_i})$. It is also easy to check that for all $\mathtt{x} = \mathtt{x'} \in \mathtt{F}_{\mathtt{mes_i}}$, $\mathtt{x} = \mathtt{x'} \in \mathtt{Abs}_{\mathtt{mes_i}}(\mathtt{s},\mathtt{h_i})$ iff $\mathtt{x} = \mathtt{x'} \in \mathtt{Abs}_{\mathtt{mes_i}}(\mathtt{s'},\mathtt{h'_i})$.

As $\mathtt{K_i} = \mathtt{s'}(\mathtt{s}^{-1}(\mathtt{Dom}(\mathtt{h_i})) \cap (\mathtt{Var_0})$, if $\mathtt{s}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h_i})$ then $\mathtt{s'}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h'_i})$. Conversely, assume $\mathtt{s'}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h'_i})$. Then $\mathtt{s'}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h'})$. As the measure $\mathtt{mes}$ is a good measure, for any variable $\mathtt{y} \in \mathtt{Var_0}$, we have $\mathtt{s}(\mathtt{y}) \in \mathtt{Dom}(\mathtt{h})$ iff $\mathtt{s'}(\mathtt{y}) \in \mathtt{Dom}(\mathtt{h'})$, so $\mathtt{s}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h})$. As $\mathtt{h} = \mathtt{h_1} * \mathtt{h_2}$, there is $\mathtt{j}$ such that $\mathtt{s}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h_j})$, which implies $\mathtt{x} \in \mathtt{s}^{-1}(\mathtt{Dom}(\mathtt{h_j}) \cap \mathtt{Var_0})$. By the definition of $\mathtt{K_j}$ we have $\mathtt{s'}(\mathtt{x}) \in \mathtt{K_j}$, and since $\mathtt{s'}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h'_i})$ we have $\mathtt{s'}(\mathtt{x}) \in \mathtt{K_j}$. So as $\mathtt{K_1} \cap \mathtt{K_2} = \emptyset$, we have $\mathtt{j} = \mathtt{i}$ and $\mathtt{s}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h_i})$. So $\mathtt{s}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h_i})$ iff $\mathtt{s'}(\mathtt{x}) \in \mathtt{Dom}(\mathtt{h'_i})$. So $\mathsf{alloc}(\mathtt{x}) \in \mathtt{Abs}_{\mathtt{mes_i}}(\mathtt{s},\mathtt{h_i})$ iff $\mathsf{alloc}(\mathtt{x}) \in \mathtt{Abs}_{\mathtt{mes_i}}(\mathtt{s'},\mathtt{h'_i})$.

The proof for test formulas of the form $\mathtt{x} \hookrightarrow^{\mathtt{l}} \mathtt{x'}$ is very similar. □

In the proof of lemma 4.2.1.4, we need lemma 4.2.1.2 below, which is indeed an instance of lemma 4.2.2.2.

Lemma 4.2.1.2. Let $\mathtt{mes}$ be a measure such that $\mathtt{Offsets_{mes}} = \{0\}$. If $(\mathtt{s},\mathtt{h}) \simeq_{\mathtt{mes}} (\mathtt{s'},\mathtt{h'})$, then for all $\mathtt{h_0} \perp \mathtt{h}$, there is $\mathtt{h'_0} \perp \mathtt{h'}$ such that $(\mathtt{s},\mathtt{h_0}) \simeq_{\mathtt{mes}} (\mathtt{s'},\mathtt{h'_0})$.

Lemma 4.2.1.3 (Congruence)**.** Let $(\mathtt{s},\mathtt{h_0})$, $(\mathtt{s'},\mathtt{h'_0})$, $(\mathtt{s},\mathtt{h_1})$, $(\mathtt{s'},\mathtt{h'_1})$ be memory shapes such that $\mathtt{h_0} \perp \mathtt{h_1}$, $\mathtt{h'_0} \perp \mathtt{h'_1}$. Let $\mathtt{mes}$ be a measure such that $\mathtt{Offsets_{mes}} = \{0\}$, and assume that $(\mathtt{s},\mathtt{h_0}) \simeq_{\mathtt{mes}} (\mathtt{s'},\mathtt{h'_0})$ and $(\mathtt{s},\mathtt{h_1}) \simeq_{\mathtt{mes}} (\mathtt{s'},\mathtt{h'_1})$. Then, $(\mathtt{s},\mathtt{h_0} * \mathtt{h_1}) \simeq_{\mathtt{mes}} (\mathtt{s'},\mathtt{h'_0} * \mathtt{h'_1})$.

Proof. Let $\mathtt{mes}$ be the measure $(\mathtt{Offsets},\mathtt{maxsize},\mathtt{Lab_{mes}},\mathtt{Var_{mes}})$. We shall show that $(\mathtt{s},\mathtt{h_0} * \mathtt{h_1}) \simeq_{\mathtt{mes}} (\mathtt{s'},\mathtt{h'_0} * \mathtt{h'_1})$. By symmetry of $\simeq_{\mathtt{mes}}$, it is sufficient to prove one inclusion, we will prove that $\mathtt{Abs_{mes}}(\mathtt{s},\mathtt{h_0} * \mathtt{h_1}) \subseteq \mathtt{Abs_{mes}}(\mathtt{s'},\mathtt{h'_0} * \mathtt{h'_1})$. Let $\mathsf{f} \in \mathtt{Abs_{mes}}(\mathtt{s},\mathtt{h_0} * \mathtt{h_1})$. We make a case analysis according to f.

- If $\mathsf{f} = \mathsf{size} \geq \mathtt{k}$ for some $\mathtt{k} < \mathtt{maxsize}$, then $\mathtt{k} \leq |\mathtt{Dom}(\mathtt{h_0} * \mathtt{h_1})|$. We want to show that $\mathtt{k} \leq |\mathtt{Dom}(\mathtt{h'_0} * \mathtt{h'_1})|$ which implies that $\mathsf{f} \in \mathtt{Abs_{mes}}(\mathtt{s},\mathtt{h'_0} * \mathtt{h'_1})$.
  - If $|\mathtt{Dom}(\mathtt{h_1})| \geq \mathtt{maxsize}$ or $|\mathtt{Dom}(\mathtt{h_0})| \geq \mathtt{maxsize}$, then $|\mathtt{Dom}(\mathtt{h'_1})| \geq \mathtt{maxsize}$ or $|\mathtt{Dom}(\mathtt{h'_0})| \geq \mathtt{maxsize}$, respectively. So we have $|\mathtt{Dom}(\mathtt{h'_0} * \mathtt{h'_1})| \geq \mathtt{maxsize}$ and $|\mathtt{Dom}(\mathtt{h'_0} * \mathtt{h'_1})| \geq \mathtt{k}$ as $\mathtt{k} < \mathtt{maxsize}$.
  - If $|\mathtt{Dom}(\mathtt{h_1})| < \mathtt{maxsize}$ and $|\mathtt{Dom}(\mathtt{h_0})| < \mathtt{maxsize}$, then we have $|\mathtt{Dom}(\mathtt{h_0} * \mathtt{h_1})| = |\mathtt{Dom}(\mathtt{h_1})| + |\mathtt{Dom}(\mathtt{h_0})| = |\mathtt{Dom}(\mathtt{h'_1})| + |\mathtt{Dom}(\mathtt{h'_0})| = |\mathtt{Dom}(\mathtt{h'_0} * \mathtt{h'_1})|$. So $\mathtt{k} \leq |\mathtt{Dom}(\mathtt{h'_0} * \mathtt{h'_1})|$.
- If f is $\mathtt{x} = \mathtt{x'}$, then $\mathtt{s}(\mathtt{x}) = \mathtt{s}(\mathtt{x'})$. Moreover, $\mathsf{f} \in \mathtt{Abs_{mes}}(\mathtt{s},\mathtt{h_1})$ iff $\mathsf{f} \in \mathtt{Abs_{mes}}(\mathtt{s'},\mathtt{h'_1})$. Therefore $\mathtt{s'}(\mathtt{x}) = \mathtt{s'}(\mathtt{x'})$ and $\mathsf{f} \in \mathtt{Abs_{mes}}(\mathtt{s'},\mathtt{h'_0} * \mathtt{h'_1})$.

– If $f = x \hookrightarrow^{l} x'$ then $(h_0 * h_1)(s(x))(l) = s(x')$. Hence, there is $i \in \{0,1\}$ such that $h_i(s(x))(l) = s(x')$. Since $(s, h_i) \simeq_{mes} (s', h'_i)$, we can state that $h'_i(s'(x))(l) = s'(x')$ and $(h'_0 * h'_1)(s'(x))(l) = s'(x')$. So $f \in Abs_{mes}(s, h'_1 * h'_0)$.
– If $f = alloc(x)$ then $s(x) \in Dom(h_0 * h_1)$. Hence, there is $i \in \{0,1\}$ such that $s(x) \in Dom(h_i)$. Since $(s, h_i) \simeq_{mes} (s', h'_i)$, $s'(x) \in Dom(h'_i)$ and $s'(x) \in Dom(h'_0 * h'_1)$, which entails $f \in Abs_{mes}(s, h'_0 * h'_1)$.

□

Lemma 4.2.1.4 below states that our abstraction is correct for the fragments SL_s^{CL} and SL_s^{RF}.

Lemma 4.2.1.4 (Soundness of the abstraction)**.** Let mes be a good measure, (s, h) and (s', h') be two memory shapes such that $(s, h) \simeq_{mes} (s', h')$ [resp. $(s, h) \simeq_{mes[maxsize \leftarrow 0]} (s', h')$]. For any SL_s formula f such that $mes_f \leqslant mes$ and f belongs to SL_s^{RF} [resp. SL_s^{CL}], we have $(s, h) \models_{SL} f$ iff $(s', h') \models_{SL} f$.

Proof. The proof of lemma 4.2.1.4 for the classical fragment is rather straightforward. Indeed, any SL_s^{CL} formula is a boolean combination of test formulas. In order to deal with the record fragment, more efforts are needed. First note that if $mes_f \leqslant mes$ then $(s, h) \simeq_{mes} (s', h')$ implies $(s, h) \simeq_{mes_f} (s', h')$.

By structural induction on f, we show that if $(s, h) \simeq_{mes_f} (s', h')$, then $(s, h) \models_{SL} f$ iff $(s', h') \models_{SL} f$. The base case when f has one of the forms $x = x'$, $x + i \hookrightarrow^{l} x'$ and emp is by an easy verification. Similarly, in the induction step, the cases when the outermost connective is boolean are straightforward.

– Assume that $(s, h) \models_{SL} f$ with $f = g_1 * g_2$. There are heaps h_1 and h_2 such that $h = h_1 * h_2$, $(s, h_1) \models_{SL} g_1$ and $(s, h_2) \models_{SL} g_2$. As $mes_f \geqslant mes_{g_1} + mes_{g_2}$ and as mes_{g_1} and mes_{g_2} are good measures since they are measures of a state formula, by application of lemma 4.2.1.1, there are heaps h'_1 and h'_2 verifying $h' = h'_1 * h'_2$, $(s, h_1) \simeq_{mes_{g_1}} (s', h'_1)$ and $(s, h_2) \simeq_{mes_{g_2}} (s', h'_2)$. By the induction hypothesis, we get $(s', h'_1) \models_{SL} g_1$ and $(s', h'_2) \models_{SL} g_2$. Consequently, $(s', h') \models_{SL} f$ since $h' = h'_1 * h'_2$ and $f = g_1 * g_2$.
– Finally, assume that $f = g_1 \mathrel{-\!\!*} g_2$. Let $h'_1 \perp h'$ be such that $(s', h'_1) \models_{SL} g_1$. Then by lemma 4.2.1.2, there is a heap h_1 such that $(s, h_1) \simeq_{mes_f} (s', h'_1)$ and $h_1 \perp h$, and so $(s, h_1) \models_{SL} g_1$ by the induction hypothesis. Then we have $(s, h * h_1) \models_{SL} g_2$, and by lemma 4.2.1.3, $(s', h' * h'_1) \models_{SL} g_2$. Hence $(s', h') \models_{SL} g_1 \mathrel{-\!\!*} g_2$.

□

4.2.2 Complexity of Quantifier-Free Separation Logic

In this section, we show that model-checking, satisfiability, and validity, for SL_s, are PSPACE-complete. We use the abbreviations $Mcheck(SL_s)$, $Satis(SL_s)$ and $Valid(SL_s)$ for the respective problems. These abbreviations are extended to any fragment of separation logic, for instance $Satis(SL_s^{RF})$ denotes the satisfiability problem for the record fragment.

PSPACE-hardness of $Mcheck(SL_s^{LF})$ and $Satis(SL_s^{LF})$ is a consequence of [36, Sect. 5.2]. As SL_s strictly contains SL_s^{LF}, this entails the PSPACE-hardness of $Mcheck(SL_s)$ and $Satis(SL_s)$. Since SL_s is closed under negation, PSPACE-completeness of $Valid(SL_s)$ will follow from PSPACE-completeness of $Satis(SL_s)$.

In order to show that $\mathtt{Mcheck}(\mathtt{SL_s})$ and $\mathtt{Satis}(\mathtt{SL_s})$ are in PSPACE, we establish the lemmas below. Lemma 4.2.2.1 establishes a reduction from $\mathtt{Mcheck}(\mathtt{SL_s})$ to $\mathtt{Mcheck}(\mathtt{SL_s^{RF}})$, so that we only need to consider $\mathtt{SL_s^{RF}}$ in order to find the complexity of model-checking. Then, in lemma 4.2.2.2, we will provide a small model property for $\mathtt{SL_s^{RF}}$, leading to the PSPACE-easiness of $\mathtt{Mcheck}(\mathtt{SL_s^{RF}})$ (see lemma 4.2.2.3). Finally, we characterize the computational complexity of the satisfiability problem thanks to lemma 4.2.2.6, which entails a reduction from $\mathtt{Satis}(\mathtt{SL_s})$ to $\mathtt{Mcheck}(\mathtt{SL_s})$.

Lemma 4.2.2.1. There is a logarithmic space reduction from $\mathtt{Mcheck}(\mathtt{SL_s})$ to $\mathtt{Mcheck}(\mathtt{SL_s^{RF}})$.

Proof. Let $\mathtt{tr}_{\mathtt{SL_s}\to\mathtt{SL_s^{RF}}}(\mathsf{f})$ be the formula obtained from f in $\mathtt{SL_s}$ by replacing each occurrence of $\mathtt{x}+\mathtt{i}\hookrightarrow^{\mathsf{l}}\mathtt{x}'$ by $\langle \mathtt{x},\mathtt{i}\rangle\hookrightarrow^{\mathsf{l}}\mathtt{x}'$. The formula $\mathtt{tr}_{\mathtt{SL_s}\to\mathtt{SL_s^{RF}}}(\mathsf{f})$ belong to $\mathtt{SL_s^{RF}}$. Given a store $\mathtt{s}$, we write $\mathtt{tr}_{\mathtt{SL_s}\to\mathtt{SL_s^{RF}}}(\mathtt{s})$ to denote the store such that $\mathtt{tr}_{\mathtt{SL_s}\to\mathtt{SL_s^{RF}}}(\mathtt{s})(\langle \mathtt{x},\mathtt{i}\rangle) = \mathtt{s}(\mathtt{x})+\mathtt{i}$. One can show that for every heap $\mathtt{h}$, we have $(\mathtt{s},\mathtt{h}) \models_{\mathtt{SL}} \mathsf{f}$ iff $(\mathtt{tr}_{\mathtt{SL_s}\to\mathtt{SL_s^{RF}}}(\mathtt{s}),\mathtt{h}) \models_{\mathtt{SL}} \mathtt{tr}_{\mathtt{SL_s}\to\mathtt{SL_s^{RF}}}(\mathsf{f})$. The proof is by structural induction on f. □

We need to establish a quite technical lemma. Given a heap $\mathtt{h}$, let $\mathtt{Im}^2(\mathtt{h})$ be the set of natural numbers $\mathtt{i}$ such that there are $\mathtt{i}'$ and l for which $\mathtt{h}(\mathtt{i}')(\mathsf{l}) = \mathtt{i}$.

Lemma 4.2.2.2. Let $\mathtt{mes} = (\{0\},\mathtt{maxsize},\mathtt{Lab_{mes}},\mathtt{Var_{mes}})$ be a measure, and l_0 be a label that does not belong to the finite set of labels $\mathtt{Lab_{mes}}$. If $(\mathtt{s},\mathtt{h}) \simeq_{\mathtt{mes}} (\mathtt{s}',\mathtt{h}')$ and $\mathtt{h}_0\perp\mathtt{h}$ is a heap, then there is a heap $\mathtt{h}_0'$ such that:

- $\mathtt{h}_0' \perp \mathtt{h}'$,
- $(\mathtt{s},\mathtt{h}_0) \simeq_{\mathtt{mes}} (\mathtt{s}',\mathtt{h}_0')$,
- $|\mathtt{Dom}(\mathtt{h}_0')| \leq \mathtt{max}(\mathtt{maxsize},|\mathtt{Var_{mes}}|)$,
- $\mathtt{max}(\mathtt{Dom}(\mathtt{h}_0') \cup \mathtt{Im}^2(\mathtt{h}_0')) \leq \mathtt{max}(\mathtt{s}'(\mathtt{Var_{mes}}) \cup \mathtt{Dom}(\mathtt{h}')) + \mathtt{maxsize} + 1$,
- for all $\mathtt{n} \in \mathtt{Dom}(\mathtt{h}_0')$, $\{\mathsf{l} : \mathtt{h}_0'(\mathtt{n})(\mathsf{l}) \text{ is defined}\} \subseteq \mathtt{Lab_{mes}} \uplus \{\mathsf{l}_0\}$.

The heap $\mathtt{h}_0'$ is said to be a small disjoint heap with respect to $\mathtt{mes}$ and $(\mathtt{s}',\mathtt{h}')$ and it can be represented in polynomial space in $\mathtt{size}(\mathtt{mes}) + \mathtt{size}_{\mathtt{Var_{mes}}}(\mathtt{h}_0) + \mathtt{size}_{\mathtt{Var_{mes}},\mathtt{Lab_{mes}}}((\mathtt{s}',\mathtt{h}'))$.

Proof. Assume that $(\mathtt{s},\mathtt{h}) \simeq_{\mathtt{mes}} (\mathtt{s}',\mathtt{h}')$ and $\mathtt{h}_0\perp\mathtt{h}$. We introduce two disjoint heaps $\mathtt{h}_{01}$ and $\mathtt{h}_{02}$ such that $\mathtt{Dom}(\mathtt{h}_{01}) = \mathtt{Dom}(\mathtt{h}_0) \cap \mathtt{Im}(\mathtt{s})$, $\mathtt{Dom}(\mathtt{h}_{02}) = \mathtt{Dom}(\mathtt{h}_0) \setminus \mathtt{Im}(\mathtt{s})$ and $\mathtt{h}_0 = \mathtt{h}_{01} * \mathtt{h}_{02}$. We define the heap $\mathtt{h}_0'$ as the disjoint union $\mathtt{h}_{01}' * \mathtt{h}_{02}'$ where $\mathtt{h}_{01}'$ and $\mathtt{h}_{02}'$ are defined so as to satisfy $\mathtt{Dom}(\mathtt{h}_{01}') = \mathtt{Dom}(\mathtt{h}_0') \cap \mathtt{Im}(\mathtt{s}')$ and $\mathtt{Dom}(\mathtt{h}_{02}') = \mathtt{Dom}(\mathtt{h}_0') \setminus \mathtt{Im}(\mathtt{s}')$.

In the sequel, $\mathtt{n}_0 = \mathtt{max}(\mathtt{s}'(\mathtt{Var_{mes}}) \cup \mathtt{Dom}(\mathtt{h}')) + \mathtt{maxsize} + 1$ is a location which can be seen as an equivalent of the value $\mathtt{nil}$ of the null constant – chosen large enough so as not to be mistaken with any other location by test formulas of $\mathtt{F_{mes}}$.

- In order to define $\mathtt{h}_{01}'$, let $\mathtt{Y}_1,\ldots,\mathtt{Y}_{\mathtt{k}_0}$ be the equivalence classes over the set $\mathtt{Var_{mes}}$ for the relation $\sim_{\mathtt{s}}$ defined by $\mathtt{x} \sim_{\mathtt{s}} \mathtt{y}$ if $\mathtt{s}(\mathtt{x}) = \mathtt{s}(\mathtt{y})$. Since $(\mathtt{s},\mathtt{h}) \simeq_{\mathtt{mes}} (\mathtt{s}',\mathtt{h}')$, the relation $\sim_{\mathtt{s}'}$ defines the same set of equivalence classes. For each class $\mathtt{Y_k}$, let $\mathtt{i_k}$ be the image of the variables of $\mathtt{Y_k}$ through $\mathtt{s}$, and $\mathtt{i_k}'$ through $\mathtt{s}'$. Then, for each $\mathtt{k} \in [1,\mathtt{k}_0]$ and $\mathsf{l} \in \mathtt{Dom}(\mathtt{h}_{01}(\mathtt{i_k}))$, the heap $\mathtt{h}_{01}'$ is defined as follows:
  - if $\mathsf{l} \notin \mathtt{Lab_{mes}}$, then $\mathtt{h}_{01}'(\mathtt{i_k}')(\mathsf{l}_0) = \mathtt{n}_0$ and $\mathtt{h}_{01}'(\mathtt{i_k}')(\mathsf{l})$ is undefined,
  - if $\mathsf{l} \in \mathtt{Lab_{mes}}$ and $\mathtt{h}_{01}(\mathtt{i_k})(\mathsf{l}) = \mathtt{i_n}$ for some $\mathtt{n}$, then $\mathtt{h}_{01}'(\mathtt{i_k}')(\mathsf{l}) = \mathtt{i_n}'$,

* if $l \in \texttt{Lab}_{\texttt{mes}}$ and $\texttt{h}_{01}(\texttt{i}_k)(l) \neq \texttt{i}_n$ for all n, then $\texttt{h}'_{01}(\texttt{i}'_k)(l) = \texttt{n}_0$.

The domain of $\texttt{h}'_{01}$ is included in $\texttt{Im}(\texttt{s}')$, since $\texttt{Im}(\texttt{s}'_{|\texttt{Var}_{\texttt{mes}}}) = \{\texttt{i}'_1, \ldots, \texttt{i}'_a\}$.

– In order to define $\texttt{h}'_{02}$, let $\texttt{k}_1 = \texttt{max}(0, \texttt{min}(|\texttt{Dom}(\texttt{h}_{02})|, \texttt{maxsize} - |\texttt{Dom}(\texttt{h}_{01})|))$ and $\texttt{j}'_1, \ldots, \texttt{j}'_{\texttt{k}_1}$ be the $\texttt{k}_1$ smallest natural numbers disjoint from $\{\texttt{i}'_1, \ldots, \texttt{i}'_a\} \cup \texttt{Dom}(\texttt{h}')$. Hence, when $|\texttt{Dom}(\texttt{h}_{01})| \geq \texttt{maxsize}$, we have $\texttt{k}_1 = 0$ and therefore there are no such natural numbers. Otherwise, $\texttt{Dom}(\texttt{h}'_{02}) = \{\texttt{j}'_1, \ldots, \texttt{j}'_{\texttt{k}_1}\}$ and for each $\texttt{k} \in [1, \texttt{k}_1]$, we define $\texttt{h}'_{02}(\texttt{j}'_k)(l_0) = \texttt{n}_0$.

As announced, we define $\texttt{h}'_0$ as the heap $\texttt{h}'_{01} * \texttt{h}'_{02}$. Let us show that the heap $\texttt{h}'_0$ has all the desired properties.

– Let us check that $\texttt{h}' \perp \texttt{h}'_0$. First, $\texttt{h}' \perp \texttt{h}'_{01}$ since $\texttt{h} \perp \texttt{h}_{01}$. Second, $\texttt{h}' \perp \texttt{h}'_{02}$ by construction.
– Let us check that $(\texttt{s}, \texttt{h}_0) \simeq_{\texttt{mes}} (\texttt{s}', \texttt{h}'_0)$. We proceed by a case analysis on the form of the test formulas.

($x = x'$) Since $(\texttt{s}, \texttt{h}) \simeq_{\texttt{mes}} (\texttt{s}', \texttt{h}')$, $\texttt{s}(x) = \texttt{s}(x')$ iff $\texttt{s}'(x) = \texttt{s}'(x')$.

($\mathsf{alloc}(x)$) We have equivalences between the propositions below:

* $\mathsf{alloc}(x) \in \texttt{Abs}_{\texttt{mes}}(\texttt{s}, \texttt{h}_0)$,
* $\texttt{s}(x) \in \texttt{Dom}(\texttt{h}_0)$,
* there is k such that $x \in \texttt{Y}_k$ and $\texttt{i}_k \in \texttt{Dom}(\texttt{h}_0)$,
* there is k such that $x \in \texttt{Y}_k$ and $\texttt{i}'_k \in \texttt{Dom}(\texttt{h}'_{01})$,
* $\mathsf{alloc}(x) \in \texttt{Abs}_{\texttt{mes}}(\texttt{s}', \texttt{h}'_0)$.

($\mathsf{size} \geq \texttt{k}$) First, observe that $|\texttt{Dom}(\texttt{h}_{01})| = |\texttt{Dom}(\texttt{h}'_{01})|$. Moreover, by construction, if $|\texttt{Dom}(\texttt{h}_0)| < \texttt{maxsize}$, then $|\texttt{Dom}(\texttt{h}_0)| = |\texttt{Dom}(\texttt{h}'_0)|$. When $|\texttt{Dom}(\texttt{h}_0)| \geq \texttt{maxsize}$, the construction of $\texttt{h}'_0$ guarantees that $|\texttt{Dom}(\texttt{h}'_0)| \geq \texttt{maxsize}$. So, for all formulas $\mathsf{size} \geq \texttt{k}$ with $\texttt{k} < \texttt{maxsize}$, $\mathsf{size} \geq \texttt{k} \in \texttt{Abs}_{\texttt{mes}}(\texttt{s}, \texttt{h}_0)$ iff $\mathsf{size} \geq \texttt{k} \in \texttt{Abs}_{\texttt{mes}}(\texttt{s}', \texttt{h}'_0)$.

($x \hookrightarrow^l x'$) We have the following implications:

* $x \hookrightarrow^l x' \in \texttt{Abs}_{\texttt{mes}}(\texttt{s}, \texttt{h}_0)$,
* there is k such that $x \in \texttt{Y}_k$ and $\texttt{h}_0(\texttt{i}_k)(l) = \texttt{s}(x')$,
* there are k, k′ such that $\texttt{h}_0(\texttt{i}_k)(l) = \texttt{i}_{k'}$,
* there are k, k′ such that $x \in \texttt{Y}_k$ and $\texttt{h}'_{01}(\texttt{i}'_k)(l) = \texttt{i}'_{k'}$,
* $x \hookrightarrow^l x' \in \texttt{Abs}_{\texttt{mes}}(\texttt{s}', \texttt{h}'_0)$.

Now suppose that $x \hookrightarrow^l x' \notin \texttt{Abs}_{\texttt{mes}}(\texttt{s}, \texttt{h}_0)$. We distinguish three cases.

1. $\texttt{s}(x) \notin \texttt{Dom}(\texttt{h}_0)$.
   From the above case with $\mathsf{alloc}(x)$, $\texttt{s}'(x) \notin \texttt{Dom}(\texttt{h}'_0)$ and therefore $x \hookrightarrow^l x' \notin \texttt{Abs}_{\texttt{mes}}(\texttt{s}', \texttt{h}'_0)$.
2. $\texttt{s}(x) \in \texttt{Dom}(\texttt{h}_0)$ (with $\texttt{i}_k = \texttt{s}(x)$), $l \in \texttt{Dom}(\texttt{h}_0(\texttt{i}_k))$ and $\texttt{h}_0(\texttt{i}_k)(l) \neq \texttt{s}(x')$.
   If $\texttt{h}_0(\texttt{i}_k)(l) = \texttt{i}_{k'}$ for some $k' \in [1, \texttt{k}_0]$, then $\texttt{s}'(x') \neq \texttt{i}'_{k'}$. If for all $k' \in [1, \texttt{k}_0]$, $\texttt{i}_{k'} \neq \texttt{h}_0(\texttt{i}_k)(l)$ (in particular $\texttt{h}_0(\texttt{i}_k)(l)$ cannot be equal to $\texttt{n}_0$, chosen large enough for this purpose), then by construction, and as $\texttt{h}'_{01}(\texttt{i}'_k)(l) \notin \{\texttt{i}'_1, \ldots, \texttt{i}'_{\texttt{k}_0}\}$. In both cases, $x \hookrightarrow^l x' \notin \texttt{Abs}_{\texttt{mes}}(\texttt{s}', \texttt{h}'_0)$.
3. $\texttt{s}(x) \in \texttt{Dom}(\texttt{h}_0)$ (with $\texttt{i}_k = \texttt{s}(x)$) and $l \notin \texttt{Dom}(\texttt{h}_0(\texttt{i}_k))$. Consequently, $l \notin \texttt{Dom}(\texttt{h}'_0(\texttt{i}'_k))$ and therefore $x \hookrightarrow^l x' \notin \texttt{Abs}_{\texttt{mes}}(\texttt{s}', \texttt{h}'_0)$.

function $\mathtt{Mcheck}((s,h),f,\mathtt{mes})$

(base-cases) If f is atomic, then return $(s,h) \models_{SL} f$;

(boolean-cases) If f is a conjunction $f_1 \wedge f_2$, then return ($\mathtt{Mcheck}((s,h),f_1,\mathtt{mes})$ and $\mathtt{Mcheck}((s,h),f_2,\mathtt{mes})$);

Other boolean operators are treated analogously.

($*$ case) If $f = f_1 * f_2$, then return $\perp$ if there are no h_1, h_2 such that $h = h_1 * h_2$ and $\mathtt{Mcheck}((s,h_1),f_1,\mathtt{mes})$ and $\mathtt{Mcheck}((s,h_2),f_2,\mathtt{mes})$);

($-\!*$ case) If $f = f_1 -\!* f_2$, then return $\perp$ if for some small disjoint heap h' with respect to $\mathtt{mes}$ and (s,h) verifying $\mathtt{Mcheck}((s,h'),f_1,\mathtt{mes})$, we have not $\mathtt{Mcheck}((s,h*h'),f_2,\mathtt{mes})$;

Return $\top$;

Figure 4.3: Model-checking algorithm

Therefore (s,h_0) and (s',h'_0) have the same abstraction.

- Let us check that $|\mathtt{Dom}(h'_0)| \leq \max(\mathtt{maxsize}, |\mathtt{Var}_{\mathtt{mes}}|)$. We already know that $k_0 \leq |\mathtt{Var}_{\mathtt{mes}}|$. If $|\mathtt{Dom}(h'_{01})| \geq \mathtt{maxsize}$, then h'_{02} is the empty heap and therefore $|\mathtt{Dom}(h'_0)| \leq k_0$. Otherwise, by construction $|\mathtt{Dom}(h'_{01})| + |\mathtt{Dom}(h'_{02})| \leq \mathtt{maxsize}$. Consequently, $|\mathtt{Dom}(h'_0)| \leq \max(\mathtt{maxsize}, |\mathtt{Var}_{\mathtt{mes}}|)$.
- Let us check that $\max(\mathtt{Dom}(h'_0) \cup \mathtt{Im}^2(h'_0)) \leq \max(\{s'(x) : x \in \mathtt{Var}_{\mathtt{mes}}, s'(x) \in \mathbb{N}\} \cup \mathtt{Dom}(h')) + \mathtt{maxsize}$. We have chosen the domain and image of h'_{01} to be included in the image of s' plus n_0, and therefore $\mathtt{Dom}(h'_{01})$ satisfies the above condition. The image of h'_{02} is $\{n_0\}$. The domain of h'_{02} is composed of the smallest natural numbers which neither belong to $s'(\mathtt{Var}_{\mathtt{mes}})$, nor to $\mathtt{Dom}(h')$. As $\mathtt{Dom}(h'_{02})$ has less than $\mathtt{maxsize}$ elements, it is bounded by the $\mathtt{maxsize}^{\text{th}}$ such natural number, which is bounded by $\mathtt{maxsize} + \max(s'(\mathtt{Var}_{\mathtt{mes}}) \cup \mathtt{Dom}(h'))$.
- Let us check that for every $n \in \mathtt{Dom}(h'_0)$, $\{l : h'_0(n)(l) \text{ is defined}\} \subseteq \mathtt{Lab}_{\mathtt{mes}} \uplus \{l_0\}$. This condition is satisfied by construction of h'_{01} and h'_{02}.

□

Lemma 4.2.2.3. $\mathtt{Mcheck}(\mathrm{SL}^{\mathrm{RF}}_{s})$ is in PSPACE.

Proof. The algorithm is described in figure 4.3. First of all, the algorithm can be implemented in polynomial space since the quantifications are over sets of exponential size in $|f| + \mathtt{size}_{\mathtt{Var}_{\mathtt{mes}},\mathtt{Lab}_{\mathtt{mes}}}((s,h))$ where $\mathtt{mes}_f = (\ldots, \mathtt{Lab}_{\mathtt{mes}}, \mathtt{Var}_{\mathtt{mes}})$, and the recursion depth is linear in $|f|$. Hence, all the heaps considered in the algorithm are of polynomial size in $|f| + \mathtt{size}_{\mathtt{Var}_{\mathtt{mes}},\mathtt{Lab}_{\mathtt{mes}}}((s,h))$. It remains to be shown that the algorithm is correct: given $\mathtt{mes}$ a good measure and f with $\mathtt{mes}_f \preccurlyeq \mathtt{mes}$, then $(s,h) \models_{SL} f$ iff $\mathtt{Mcheck}((s,h),f,\mathtt{mes})$ returns $\top$. The only point to check in the proof by structural induction is the case when the outermost connective is the operator $-\!*$. Whenever $(s,h) \not\models_{SL} f_1 -\!* f_2$, there is a heap $h_0 \perp h$ such that $(s,h_0) \models_{SL} f_1$ and $(s,h*h_0) \not\models_{SL} f_2$. By lemma 4.2.2.2 with $(s',h') = (s,h)$, there is a small disjoint heap h'_0 with respect to $\mathtt{mes}$ and (s,h) such that $(s,h'_0) \simeq_{\mathtt{mes}} (s,h_0)$. Since the measure of f_1 is less than $\mathtt{mes}$, lemma 4.2.1.4 entails $(s,h'_0) \models_{SL} f_1$. Moreover, by lemma 4.2.1.3,

$(s, h * h'_0) \not\models_{SL} f_2$. Consequently, $(s, h) \not\models_{SL} f_1 \multimap f_2$ iff there is a small heap h'_0 such that $(s, h'_0) \models_{SL} f_1$ and $(s, h * h'_0) \not\models_{SL} f_2$. □

The rest of the section is dedicated to the characterization of the complexity of decision problems for SL_s. To do so, we need another technical lemma.

Definition 4.2.2.4. Given a permutation $pmt : Loc \to Loc$ and a heap h, we write $pmt \cdot h$ to denote the partial function which maps i to the partial function $pmt \circ (h(i))$. When viewing heaps as finite subsets of $\mathbb{N} \times Lab \times Loc$, $pmt \cdot h$ is equal to $\{(i, l, pmt(j)) : (i, l, j) \in h\}$. We write $pmt \circledcirc h$ to denote the heap $pmt \cdot (h \circ pmt^{-1})$, which corresponds to $\{(pmt(i), l, pmt(j)) : (i, l, j) \in h\}$.

For instance, given a label l and an address i, $(pmt \circledcirc h)(i)(l) = pmt(h(pmt^{-1}(i))(l))$. The operation $\circledcirc$ allows us to rename all the addresses according to the permutation: the memory graph keeps the same shape, but vertices are placed on different addresses. We shall use the properties below that can be easily checked:

- For all permutations pmt and disjoint heaps h_1 and h_2,
  $pmt \circledcirc (h_1 * h_2) = (pmt \circledcirc h_1) * (pmt \circledcirc h_2)$.
- For all permutations pmt and heaps h,
  $pmt^{-1} \circledcirc (pmt \circledcirc h) = h$.

Lemma 4.2.2.5. Let $mes = (Offsets, maxsize, Lab_{mes}, Var_{mes})$ be a measure, f be a state formula with measure mes and (s, h) be a memory shape. For all permutations $pmt : Loc \to Loc$ such that for all $x \in Var_{mes}$ and $i \in Offsets$, $pmt(s(x) + i) = pmt(s(x)) + i$, we have $(s, h) \models_{SL} f$ iff $(pmt \circ s, pmt \circledcirc h) \models_{SL} f$.

Proof. Let f be an SL_s formula, mes be a measure greater than mes_f, s be a store and h be a heap. It is sufficient to show one direction of the equivalence since the other direction is obtained by application of the first one with the store $pmt \circ s$ and the well-defined inverse bijection pmt^{-1}. Indeed, for all $x \in Var_{mes}$, $pmt^{-1}((pmt \circ s)(x) + i) = pmt^{-1}((pmt \circ s)(x)) + i$. Assume that $(s, h) \models_{SL} f$. We show that $(pmt \circ s, pmt \circledcirc h) \models_{SL} f$. We are going to prove this by induction on f. The cases with boolean operators are trivial and are omitted. If f is an atomic formula, then we proceed by a case analysis.

f is $x = x'$: $s(x) = s(x')$ iff $pmt(s(x)) = pmt(s(x'))$ since pmt is a bijection on Loc.

f is $x + i \hookrightarrow^{l} x'$: then $h(s(x) + i)(l) = s(x')$, and we have $pmt \circledcirc h(pmt \circ s(x) + i)(l) = pmt \cdot h(pmt^{-1}(pmt(s(x)) + i))(l) = pmt \cdot h(pmt^{-1}(pmt(s(x) + i)))(l) = pmt \cdot h(s(x) + i)(l) = pmt(h(s(x) + i)(l)) = pmt(s(y)) = pmt \circ s(x')$,

f is emp: $Dom(pmt \circledcirc h)$ is empty iff $Dom(h)$ is empty.

If $f = f_1 * f_2$, then there are h_1 and h_2 such that $h = h_1 * h_2$ and $(s, h_1) \models_{SL} f_1$ and $(s, h_2) \models_{SL} f_2$. For each measure mes_{f_i}, we have $mes_{f_i} \leqslant mes_f \leqslant mes$. Then, by induction, $(pmt \circ s, pmt \circledcirc h_i) \models_{SL} f_i$. Since $pmt \circledcirc h = pmt \circledcirc (h_1 * h_2) = (pmt \circledcirc h_1) * (pmt \circledcirc h_2)$, we can conclude that $(pmt \circ s, pmt \circledcirc h) \models_{SL} f$.

If $f = f_1 \multimap f_2$, then let h_0 be a heap which is orthogonal to $pmt \circledcirc h$. Assume that $(pmt \circ s, h_0) \models_{SL} f_1$. By induction, $(pmt^{-1} \circ (pmt \circ s), pmt^{-1} \circledcirc h_0) \models_{SL} f_1$, that is $(s, pmt^{-1} \circledcirc h_0) \models_{SL} f_1$.

So $(s, h * (pmt^{-1} \circledcirc h_0)) \models_{SL} f_2$, and by induction $(pmt \circ s, pmt \circledcirc (h * (pmt^{-1} \circledcirc h_0))) \models_{SL} f_2$, that is $(pmt \circ s, (pmt \circledcirc h) * (pmt \circledcirc (pmt^{-1} \circledcirc h_0))) \models_{SL} f_2$, and finally $(pmt \circ s, (pmt \circledcirc h) * h_0) \models_{SL} f_2$. So, $(pmt \circ s, pmt \circledcirc h) \models_{SL} f$. □

We state below a small memory shape property that happens to be central to establish the results about the forthcoming PSPACE upper bounds.

Lemma 4.2.2.6 (Small memory shape property)**.** A state formula f in SL_s is satisfiable iff there is a store s such that $(s, \emptyset) \models_{SL} \neg(f \twoheadrightarrow \perp)$ and for each variable $x \in Var_f$, $s(x) \leq (|Var_f| + 1) \times (1 + max(Offsets_f))$, where $\emptyset$ stands for the heap with empty domain, Var_f is the set of variables occuring in f, and $Offsets_f$ is the set of indexes i such that $x + i$ occurs in f for some variable x. If $Offsets_f$ is empty, we can replace $max(Offsets_f)$ by 0.

Proof. First, it is straightforward to show that f in SL_s is satisfiable iff there is a store s such that $(s, \emptyset) \models_{SL} \neg(f \twoheadrightarrow \perp)$, where $\emptyset$ is the heap with empty domain. So, we only have to prove that given an SL_s state formula f and a store s such that $(s, \emptyset) \models_{SL} f$, there is a store s′ such that $(s', \emptyset) \models_{SL} f$ and for each $x \in Var_f$, $s'(x) \leq (|Var_f| + 1) \times (1 + max(Offsets_f))$ (the interpretation of other variables is irrelevant). In order to obtain this small store, we are going to decrease the value of the variables in several steps. Each step consists of applying a permutation to the memory graph.

Assume that $(s, \emptyset) \models_{SL} f$ and let $maxoffset = 1 + max(Offsets_f)$. Let x_0 be a dummy variable such that $s(x_0) = 0$, and $x_1, \ldots, x_n$ be an ordering of the variables occurring in f such that for $j \in [0, n-1]$, $s(x_j) \leq s(x_{j+1})$. If there is no k such that $s(x_{k+1}) \geq s(x_k) + maxoffset$, then for all $x \in Var_f$, $s(x) \leq (n+1) \times (1 + maxoffset)$.

Otherwise, let k be the smallest index such that $s(x_{k+1}) \geq s(x_k) + maxoffset$. Let $m = s(x_{k+1}) - (s(x_k) + maxoffset)$. Let us define the permutation pmt based on m:

- If $j \leq s(x_k) + maxoffset$ then $pmt(j) = j$;
- If $s(x_{k+1}) \leq j \leq s(x_n) + maxoffset$, then $pmt(j) = j - m$;
- If $j \geq s(x_n) + maxoffset$ then $pmt(j) = j$;
- If $s(x_k) + maxoffset < j < s(x_{k+1})$ then we have to complete this function so as to obtain a bijection, $pmt(j) = j - (s(x_k) + maxoffset) + (s(x_n) + maxoffset - m)$.

Observe that for all $x \in Var_f$ and $i \in Offsets_f$, $pmt(s(x) + i) = pmt(s(x)) + i$. This permutation satisfies the hypotheses of lemma 4.2.2.5, and thus may be applied to $(s, \emptyset)$, which then still satisfies f. We apply this type of permutation until there is no k such that $s(x_{k+1}) \geq s(x_k) + maxoffset$. So, by simple multiplication, for all $x \in Var_f$, $s(x) \leq (n+1) \times maxoffset$. □

Lemma 4.2.2.7. The model-checking, satisfiability, and validity problems for SL_s are PSPACE-complete.

Proof. PSPACE-hardness results are consequences of [36, Sect. 5.2]. The PSPACE upper bound for $Mcheck(SL_s)$ is a consequence of lemmas 4.2.2.1 and 4.2.2.3. The PSPACE upper bound for $Satis(SL_s)$ is obtained by enumerating the small memory shapes of $\neg(f \twoheadrightarrow \perp)$ with empty heap (see lemma 4.2.2.6) and then using lemma 4.2.2.3. □

4.3 Decidable Problems by Abstracting Computations

In this section, we establish the PSPACE-completeness of $\mathtt{Sat}(\mathtt{SL}_s^{\mathtt{CL}})$ and $\mathtt{Sat}(\mathtt{SL}_s^{\mathtt{RF}})$. To do so, we abstract memory shapes whose size is a priori unbounded by symbolic memory shapes whose size is bounded. As usual with linear temporal logic, temporal infinity in models is handled by Büchi automata recognizing infinite sequences, the method is describe in [89]. We propose below an abstraction that is correct for $\mathtt{SL}_s^{\mathtt{CL}}$ (allowing pointer arithmetic) and for $\mathtt{SL}_s^{\mathtt{RF}}$ (allowing all operators from separation logic) taken separately but that is not exact for the full language $\mathtt{SL}_s$.

4.3.1 Symbolic Models

We define below symbolic models, which are abstractions of models from $\mathtt{LTL}^{\mathtt{mem}}$, and a symbolic satisfiability relation.

Definitions

Given a measure $\mathtt{mes}$, we write $\mathtt{A}_{\mathtt{mes}}$ to denote the power set of $\mathtt{F}_{\mathtt{mes}}$; $\mathtt{A}_{\mathtt{mes}}$ is thought of as an alphabet, and elements $\mathtt{a} \in \mathtt{A}_{\mathtt{mes}}$ are called *letters*. A symbolic model with respect to $\mathtt{mes}$ is defined as an infinite sequence $\mathtt{Symbmod} \in \mathtt{A}_{\mathtt{mes}}^{\mathbb{N}}$.

Given a model $\mathtt{mod} : \mathbb{N} \to \mathtt{Stores} \times \mathtt{Heaps}_s$ and a measure $\mathtt{mes}$, we write $\mathtt{Abs}_{\mathtt{mes}}(\mathtt{mod}) : \mathbb{N} \to \mathtt{A}_{\mathtt{mes}}$ to denote the symbolic model with respect to $\mathtt{mes}$ such that for every $\mathtt{n} \in \mathbb{N}$, $\mathtt{Abs}_{\mathtt{mes}}(\mathtt{mod})(\mathtt{n}) \triangleq \{\mathtt{f} \in \mathtt{F}_{\mathtt{mes}} : \mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{f}[\langle \mathtt{x}, \mathtt{k}\rangle \leftarrow \bigcirc^{\mathtt{k}}\mathtt{x}]\}$.

To a letter $\mathtt{a}$, we associate the formula $\mathtt{conj}(\mathtt{a}) = \bigwedge_{\mathtt{f}\in\mathtt{a}} \mathtt{f} \wedge \bigwedge_{\mathtt{f}\in(\mathtt{F}_{\mathtt{mes}}\setminus\mathtt{a})} \neg\mathtt{f}$. For all symbolic models $\mathtt{Symbmod}$ and formulas $\mathtt{t}$ such that $\mathtt{mes}_{\mathtt{t}} \leqslant \mathtt{mes}$, we inductively define the symbolic satisfaction relation $\mathtt{Symbmod}, \mathtt{n} \models_{\mathtt{mes}} \mathtt{t}$ the same way as the satisfaction relation for temporal models except for the clause about state subformulas (see a few sentences further). For instance $\mathtt{Symbmod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{t} \wedge \mathtt{u}$ iff $\mathtt{Symbmod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{t}$ and $\mathtt{Symbmod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{u}$; also $\mathtt{Symbmod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{t}\ \mathtt{until}\ \mathtt{u}$ iff there is $\mathtt{n}_1 \geq \mathtt{n}$ such that $\mathtt{Symbmod}, \mathtt{n}_1 \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{u}$ and $\mathtt{Symbmod}, \mathtt{n}' \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{t}$ for all $\mathtt{n}' \in [\mathtt{n}, \mathtt{n}_1[$. The clause about state subformulas is updated as follows: $\mathtt{Symbmod}, \mathtt{n} \models_{\mathtt{mes}} \mathtt{f}$ iff $\models_{\mathtt{SL}} \mathtt{conj}(\mathtt{Symbmod}(\mathtt{n})) \Rightarrow \mathtt{f}[\bigcirc^{\mathtt{k}}\mathtt{x} \leftarrow \langle \mathtt{x}, \mathtt{k}\rangle]$. We write $\mathtt{Lang}^{\mathtt{mes}}(\mathtt{t})$ to denote the set of symbolic models $\mathtt{Symbmod}$ with respect to $\mathtt{mes}$ such that $\mathtt{Symbmod}, 0 \models_{\mathtt{mes}} \mathtt{t}$.

Soundness

As a corollary of lemma 4.2.1.4, we get a soundness result for our abstraction:

Lemma 4.3.1.1. Let $\mathtt{t}$ be a formula of $\mathtt{LTL}^{\mathtt{mem}}(\mathtt{SL}_s^{\mathtt{RF}})$ [resp. of $\mathtt{LTL}^{\mathtt{mem}}(\mathtt{SL}_s^{\mathtt{CL}})$] and $\mathtt{mes}$ a good measure such that $\mathtt{mes}_{\mathtt{t}} \leqslant \mathtt{mes}$. For any model $\mathtt{mod}$, we have $\mathtt{mod} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{t}$ if and only if $\mathtt{Abs}_{\mathtt{mes}}(\mathtt{mod}) \models_{\mathtt{mes}} \mathtt{t}$ [resp. $\mathtt{Abs}_{\mathtt{mes}[\mathtt{maxsize}\leftarrow 0]}(\mathtt{mod}) \models_{\mathtt{mes}} \mathtt{t}$].

Proof. We treat the case $\mathtt{t} \in \mathtt{LTL}^{\mathtt{mem}}(\mathtt{SL}_s^{\mathtt{RF}})$ (for the case $\mathtt{t} \in \mathtt{LTL}^{\mathtt{mem}}(\mathtt{SL}_s^{\mathtt{CL}})$, replace below $\mathtt{mes}$ by $\mathtt{mes}[\mathtt{maxsize} \leftarrow 0]$). The induction step for the cases with boolean and temporal operators is by an easy verification. Let us check the base case, for a state formula. Suppose that $\mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{g}$ for an atomic formula $\mathtt{g}$ of $\mathtt{LTL}^{\mathtt{mem}}$. By definition, $\mathtt{Abs}_{\mathtt{mes}}(\mathtt{mod})(\mathtt{n}) \triangleq \{\mathtt{f} \in \mathtt{F}_{\mathtt{mes}} : \mathtt{mod}, \mathtt{n} \models_{\mathtt{LTL}^{\mathtt{mem}}} \mathtt{f}[\langle \mathtt{x}, \mathtt{k}\rangle \leftarrow \bigcirc^{\mathtt{k}}\mathtt{x}]\}$. Let us show that $\models_{\mathtt{SL}} \mathtt{conj}(\mathtt{Abs}_{\mathtt{mes}}(\mathtt{mod})(\mathtt{n})) \Rightarrow \mathtt{g}[\bigcirc^{\mathtt{k}}\mathtt{x} \leftarrow \langle \mathtt{x}, \mathtt{k}\rangle]$. If for some memory shape $(\mathtt{s}, \mathtt{h}) \models_{\mathtt{SL}} \mathtt{conj}(\mathtt{Abs}_{\mathtt{mes}}(\mathtt{mod})(\mathtt{n}))$, then by he lemma 4.2.1.4, $(\mathtt{s}, \mathtt{h}) \models_{\mathtt{SL}} \mathtt{g}[\bigcirc^{\mathtt{k}}\mathtt{x} \leftarrow \langle \mathtt{x}, \mathtt{k}\rangle]$.

Suppose now that $\mathtt{Abs_{mes}(mod)}, n \models_{mes} g$. Then, $\models_{SL} \mathsf{conj}(\mathtt{Abs_{mes}(mod)(n)}) \Rightarrow g[\bigcirc^k x \leftarrow \langle x, k\rangle]$. Since $\mathtt{mod}, n \models_{SL} \mathsf{conj}(\mathtt{Abs_{mes}(mod)(n)})[\langle x, k\rangle \leftarrow \bigcirc^k x]$, we have $\mathtt{mod}, n \models_{LTL^{mem}} (g[\bigcirc^k x \leftarrow \langle x, k\rangle])[\langle x, k\rangle \leftarrow \bigcirc^k x]$. This means that $\mathtt{mod}, n \models_{LTL^{mem}} g$. □

Note that $\mathtt{Abs_{mes}}$ is not surjective; we note $\mathtt{Lang^{mes}_{sat}}$ the set of symbolic models with respect to $\mathtt{mes}$ that are abstractions of some model for $\mathtt{LTL^{mem}}$. Consequently, t in $\mathtt{LTL^{mem}}(\mathtt{SL^{RF}_s})$ is satisfiable iff $\mathtt{Lang^{mes_t}}(t) \cap \mathtt{Lang^{mes_t}_{sat}}$ is nonempty.

4.3.2 Omega-Regularity and Polynomial Space Upper Bound

In order to show that $\mathtt{Sat}(\mathtt{SL^{RF}_s})$ and $\mathtt{Sat}(\mathtt{SL^{CL}_s})$ are in PSPACE we shall explain why testing the non-emptiness of $\mathtt{Lang^{mes_t}}(t) \cap \mathtt{Lang^{mes_t}_{sat}}$ can be done in PSPACE. Below we always treat the case for $\mathtt{SL^{RF}_s}$. For $\mathtt{SL^{CL}_s}$, replace every occurrence of $\mathtt{mes_t}$ by $\mathtt{mes_t[maxsize \leftarrow 0]}$ and every occurrence of $\mathtt{mes}$ by $\mathtt{mes[maxsize \leftarrow 0]}$.

We are going to show that each of the languages $\mathtt{Lang^{mes_t}}(t)$ and $\mathtt{Lang^{mes_t}_{sat}}$ can be recognized by a Büchi automaton with exponential size. This Büchi automaton will additionally satisfy the right properties to establish the PSPACE upper bound. If $\mathbb{A}$ is a Büchi automaton, we note $\mathtt{Lang}(\mathbb{A})$ the language recognized by $\mathbb{A}$. Following [89, 43], let $\mathbb{A}$ be the generalized Büchi automaton defined by the structure (A, B, d, B_I, B_F) such that ($\mathtt{mes} \geq \mathtt{mes_t}$):

- the set of states B is the set of so-called atoms of t, that are sets of temporal formulas included in the so-called closure set $\mathtt{cl}(t)$ (see [89]). Let us briefly recall that the closure set $\mathtt{cl}(t)$ is the smallest set containing t, closed under subformulas, negations (double negations are eliminated) and such that: if $(u \ \mathsf{until}\ u') \in \mathtt{cl}(t)$, then $\bigcirc(u \ \mathsf{until}\ u') \in \mathtt{cl}(t)$. A set $T \subseteq \mathtt{cl}(t)$ is an atom if
  - for $u \wedge u' \in \mathtt{cl}(t)$, we have $u \in T$ and $u' \in T$ iff $u \wedge u' \in T$;
  - for $u \in \mathtt{cl}(t)$, we have $u \in T$ iff $\neg u \notin T$;
  - we have $(u \ \mathsf{until}\ u') \in T$ iff ($u' \in T$ or ($u, \bigcirc(u \ \mathsf{until}\ u') \in T$)) whenever $(u \ \mathsf{until}\ u') \in \mathtt{cl}(t)$.
- the set of initial states B_I is $\{T \in B : t \in T\}$.
- the alphabet A is $A_{\mathtt{mes}}$.
- the transition relation is defined by $T' \in d(T, a)$ iff
  1. for every atomic formula f of T, $\models_{SL} \mathsf{conj}(a) \Rightarrow f[\bigcirc^n x \leftarrow \langle x, n\rangle]$.
  2. for every $\bigcirc t' \in \mathtt{cl}(t)$, $\bigcirc t' \in T$ iff $t' \in T'$.
- The generalized acceptance condition, that is a set of sets of states such that a run is accepted iff for any set of states in the generalized acceptance condition there is a state in this set such that this state is visited infinitely often, is defined as follows. Let $\{(t_1 \ \mathsf{until}\ t'_1), \ldots, (t_n \ \mathsf{until}\ t'_n)\}$ be the set of until formulas in $\mathtt{cl}(t)$. Let B_F be equal to $\{B^1_F, \ldots, B^n_F\}$ where $B^j_F = \{T \in B : (t_j \ \mathsf{until}\ t'_j) \notin T \text{ or } t'_j \in T\}$ for $j \in \{1, \ldots, n\}$. If the formula does not contain any until operator, then the set is empty, and any run in the automaton is accepting, hence any word for which there is a run is accepted.

Let $\mathbb{A}_t^{\mathtt{mes}}$ be the Büchi automaton equivalent to the generalized Büchi automaton $\mathbb{A}$. This automaton can be obtained by working with $|\mathtt{B_F}|$ copies of the generalized automaton. The set of states of $\mathbb{A}_t^{\mathtt{mes}}$ is $\mathtt{B} \times [0, |\mathtt{B_F}| - 1]$. The initial states are $\mathtt{B_I} \times \{0\}$. The final states are $\mathtt{B_F^1} \times \{0\}$. There is a transitions $((\mathtt{b}, \mathtt{j}), \mathtt{a}, (\mathtt{b}', \mathtt{j}'))$ in $\mathbb{A}_t^{\mathtt{mes}}$ iff there is a transition $(\mathtt{b}, \mathtt{a}, \mathtt{b}')$ in $\mathbb{A}$ and:

- if $\mathtt{b} \in \mathtt{B_F^j}$ then $\mathtt{j}'$ is $\mathtt{j} + 1$ modulo $|\mathtt{B_F}|$,
- otherwise $\mathtt{j}'$ is $\mathtt{j}$.

It is easy to observe that $\mathbb{A}_t^{\mathtt{mes}}$ is equivalent to $\mathbb{A}$. It is also easy to observe that $\mathbb{A}_t^{\mathtt{mes_t}}$ has an exponential amount of states in the size of t and its transition relation can be checked in polynomial space in the size of t. Moreover:

Lemma 4.3.2.1. Let t be a formula in $\mathtt{LTL^{mem}(SL_s^{RF})}$ [resp. $\mathtt{LTL^{mem}(SL_s^{CL})}$] and let $\mathtt{mes}$ be a good measure such that $\mathtt{mes} \geq \mathtt{mes_t}$ [resp. $\mathtt{mes}[\mathtt{maxsize} \leftarrow 0] \geq \mathtt{mes_t}[\mathtt{maxsize} \leftarrow 0]$]. Then, $\mathtt{Lang}(\mathbb{A}_t^{\mathtt{mes}}) = \mathtt{Lang^{mes}}(\mathtt{t})$ [resp. $\mathtt{Lang}(\mathbb{A}_t^{\mathtt{mes}[\mathtt{maxsize}\leftarrow 0]}) = \mathtt{Lang}^{\mathtt{mes}[\mathtt{maxsize}\leftarrow 0]}(\mathtt{t})$].

We can also build a Büchi automaton $\mathbb{A}_{\mathtt{sat}}^{\mathtt{mes}}$ such that $\mathtt{Lang}(\mathbb{A}_{\mathtt{sat}}^{\mathtt{mes}}) = \mathtt{Lang}_{\mathtt{sat}}^{\mathtt{mes}}$. $\mathbb{A}_{\mathtt{sat}}^{\mathtt{mes}}$ is defined as $(\mathtt{A}, \mathtt{B}', \mathtt{d}', \mathtt{B}_\mathtt{I}', \mathtt{B}_\mathtt{F}')$, where $\mathtt{A} = \mathtt{A_{mes}}$, $\mathtt{B}' = \mathtt{A_{mes}}$, $\mathtt{B}_\mathtt{F}' = \mathtt{B}_\mathtt{I}' = \mathtt{B}'$ and $\mathtt{a} \xrightarrow{\mathtt{a}'} \mathtt{a}''$ iff:

1. $\mathsf{conj}(\mathtt{a}), \mathsf{conj}(\mathtt{a}'')$ are satisfiable, and $\mathtt{a} = \mathtt{a}'$,
2. for every formula $\langle \mathtt{x}, \mathtt{n}\rangle = \langle \mathtt{x}', \mathtt{n}'\rangle \in \mathtt{F_{mes}}$ with $\mathtt{n}, \mathtt{n}' \geq 1$, $\langle \mathtt{x}, \mathtt{n}\rangle = \langle \mathtt{x}', \mathtt{n}'\rangle \in \mathtt{a}$ iff $\langle \mathtt{x}, \mathtt{n} - 1\rangle = \langle \mathtt{x}', \mathtt{n}' - 1\rangle \in \mathtt{a}''$,

If $\mathtt{mes} = \mathtt{mes_t}$, then $\mathbb{A}_{\mathtt{sat}}^{\mathtt{mes}}$ is of exponential size in the size of t and the transition relation can be checked in polynomial space in the size of t. More importantly, this automaton recognizes satisfiable symbolic models.

Lemma 4.3.2.2. Let t in $\mathtt{LTL^{mem}(SL_s^{RF})}$ [resp. $\mathtt{LTL^{mem}(SL_s^{CL})}$] and $\mathtt{mes} = \mathtt{mes_t}$ [resp. $\mathtt{mes} = \mathtt{mes_t}[\mathtt{maxsize} \leftarrow 0]$]. Then, $\mathtt{Lang}(\mathbb{A}_{\mathtt{sat}}^{\mathtt{mes}}) = \mathtt{Lang}_{\mathtt{sat}}^{\mathtt{mes}}$.

Proof. It is immediate that the abstraction with respect to $\mathtt{mes}$ of any model necessarily belongs to $\mathtt{Lang}(\mathbb{A}_{\mathtt{sat}}^{\mathtt{mes}})$. Therefore, the set $\mathtt{Lang}_{\mathtt{sat}}^{\mathtt{mes}}$ of abstractions of models with respect to $\mathtt{mes}$ is included in $\mathtt{Lang}(\mathbb{A}_{\mathtt{sat}}^{\mathtt{mes}})$.

The other inclusion is shown by induction. Let $\mathtt{mes} = (\mathtt{Offsets}, \mathtt{maxsize}, \mathtt{Lab_0}, \mathtt{Var_0})$ be the measure $\mathtt{mes_t}$, $\mathtt{n_0}$ be $\mathtt{max}(\{\mathtt{n} :$ there is $\mathtt{x} \in \mathtt{Var}$ such that $\bigcirc^{\mathtt{n}} \mathtt{x}$ occurs in t$\})$ and $\mathtt{maxi}$ be $\mathtt{max}(\mathtt{Offsets}) + 1$. Let $(\mathtt{a_i})_{\mathtt{i}\in\mathbb{N}}$ be an infinite sequence of symbolic memory shapes in $\mathtt{Lang}(\mathbb{A}_{\mathtt{sat}}^{\mathtt{mes}})$. We shall build a sequence $(\mathtt{s_i}, \mathtt{h_i})_{\mathtt{i}\in\mathbb{N}}$ such that $\mathtt{Abs_{mes}}((\mathtt{s_i}, \mathtt{h_i})_{\mathtt{i}\in\mathbb{N}}) = (\mathtt{a_i})_{\mathtt{i}\in\mathbb{N}}$. So, for $\mathtt{i} \in \mathbb{N}$, $\mathtt{a_i} = \{\mathtt{f} \in \mathtt{F_{mes}} : \mathtt{mod}, \mathtt{i} \models_{\mathtt{LTL^{mem}}} \mathtt{f}[\langle \mathtt{x}, \mathtt{n}\rangle \leftarrow \bigcirc^{\mathtt{n}}\mathtt{x}]\}$. The construction is by induction on the position $\mathtt{i} \in \mathbb{N}$.

Let us study the base case of the induction that will provide a value for $\mathtt{s_0}, \ldots, \mathtt{s_{n_0}}, \mathtt{h_0}$. Since $(\mathtt{a_i})_{\mathtt{i}\in\mathbb{N}} \in \mathtt{Lang}(\mathbb{A}_{\mathtt{sat}}^{\mathtt{mes}})$, $\mathsf{conj}(\mathtt{a_0})$ is satisfiable. There are $\mathtt{s}_0'$ and $\mathtt{h}_0'$ satisfying $(\mathtt{s}_0', \mathtt{h}_0') \models_{\mathtt{SL}} \mathsf{conj}(\mathtt{a_0})$. When dealing with the record fragment ($\mathtt{Offsets} = \{0\}$), the objects are appropriate for the initialization: $\mathtt{h_0} = \mathtt{h}_0'$ and for $\mathtt{n} \in [0, \mathtt{n_0}]$ and $\mathtt{x} \in \mathtt{Var_0}$, we set $\mathtt{s_n}(\mathtt{x}) = \mathtt{s}_0'(\langle \mathtt{x}, \mathtt{n}\rangle)$. When $\mathtt{Offsets} \neq \{0\}$ ($\mathtt{maxsize} = 0$ and we are dealing with the fragment $\mathtt{SL_s^{CL}}$), there is no constraint on the size of the heap. We apply a permutation $\mathtt{pmt}$ which maps all the images of variables to multiples of $\mathtt{maxi}$. For $\mathtt{n} \in [0, \mathtt{n_0}]$, we consider the store $\mathtt{s_n}$ such that for $\mathtt{x} \in \mathtt{Var_0}$, $\mathtt{s_n}(\mathtt{x}) = \mathtt{pmt}(\mathtt{s}_0'(\langle \mathtt{x}, \mathtt{n}\rangle))$. The heap $\mathtt{h_0}$ is defined by enumerating the test formulas $\langle \mathtt{x}, \mathtt{n}\rangle + \mathtt{j} \hookrightarrow^{\mathtt{l}} \langle \mathtt{x}', \mathtt{n}'\rangle$ and $\mathsf{alloc}(\langle \mathtt{x}, \mathtt{n}\rangle + \mathtt{j})$ of $\mathtt{a_0}$, and by defining the heap accordingly. When

$\langle x, n\rangle + j \hookrightarrow^{l} \langle x', n'\rangle \in a_0$, we define $h_0(s_n(x) + j)(l) = s_{n'}(x')$; when $\mathsf{alloc}(\langle x, n\rangle + j) \in a_0$, we define $h_0(s_n(x) + j)(l_0) = s_n(x) + j$, for some $l_0 \notin \mathtt{Lab}_0$. Thanks to the distance $\mathtt{maxi}$ imposed between the values of variables, and as $l_0 \notin \mathtt{Lab}_0$, test formulas about the heap which are not in a_0 are not satisfied. Equalities $x = x'$ are preserved since the store has only been modified by a permutation.

For the inductive step, suppose that we have already defined the stores $s_0, \ldots, s_{k+n_0}$ and heaps $h_0, \ldots, h_k$ for some position $k \geq 0$ satisfying the conditions below: for every $i \leq k$,

- for all $f \in F_{mes}$, $(s_i^+, h_i) \models_{SL} f$ iff $f \in a_i$, where $s_i^+ : \langle x, n\rangle \mapsto s_{i+n}(x)$;
- $\mathtt{Im}(s_i^+) \subseteq \mathtt{maxi}\mathbb{N}$, where $k\mathbb{N}$ for $k \in \mathbb{N}$ is $\{k \times i, i \in \mathbb{N}\}$.

Let us build the store s_{k+n_0+1} and the heap h_{k+1}. Since $(a_i)_{i\in\mathbb{N}} \in \mathtt{Lang}(\mathbb{A}^{mes}_{sat})$, $\mathsf{conj}(a_{k+1})$ is satisfiable. There exists a memory shape (s', h') satisfying $(s', h') \models_{SL} \mathsf{conj}(a_{k+1})$ and for all $x \in \mathtt{Var}$ and $n \in [0, n_0 - 1]$, $s'(\langle x, n\rangle) = s_{k+1+n}(x)$. By definition of $\mathbb{A}^{mes}_{sat}$, for all $n, n' \in [0, n_0 - 1]$ we have $\langle x, n + 1\rangle = \langle x', n' + 1\rangle \in a_k$ iff $\langle x, n\rangle = \langle x', n'\rangle \in a_{k+1}$. Consequently, for all $n, n' \in [0, n_0 - 1]$, $s_{k+1+n}(x) = s_{k+1+n'}(x')$ iff $s'(\langle x, n\rangle) = s'(\langle x', n'\rangle)$. So, there is a permutation $\mathtt{pmt}$ identical for the variables $\langle x, n\rangle$ with $n \in [0, n_0 - 1]$ such that $\mathtt{Im}(\mathtt{pmt} \circ s') \subseteq \mathtt{maxi}\mathbb{N}$. By construction, for $\langle x, n\rangle \in \mathtt{Var}_{mes}$, $\mathtt{pmt}(s'(\langle x, n\rangle)) \in \mathtt{maxi}\mathbb{N}$. For $x \in \mathtt{Var}_0$, we set $s_{k+1+n_0}(x) = \mathtt{pmt}(s'(\langle x, n_0\rangle))$.

If we consider $\mathtt{SL}^{RF}_s$, this permutation satisfies the prerequisites of lemma 4.2.2.5, since $\mathtt{Offsets} = \{0\}$. We can define $h_{k+1} = \mathtt{pmt} \circledcirc h'$. Thanks to lemma 4.2.2.5, we know that both of these memory shapes satisfy the same test formulas, which are exactly those in a_{k+1}.

If we are dealing with $\mathtt{SL}^{CL}_s$, then the definition of s_{k+n_0+1} ensures that the equalities satisfied are exactly those of a_{k+1}. This time the prerequisites of lemma 4.2.2.5 are not satisfied unless $\mathtt{Offsets} = \{0\}$. We know that $\mathtt{maxsize} = 0$, which means that the only test formula about size in a_{k+1} is $\mathsf{size} \geq 0$; therefore there is no constraint on the size of the heap. The heap is defined by enumerating the test formulas of the form $\langle x, n\rangle + j \hookrightarrow^{l} \langle x', n'\rangle$ of a_{k+1}, and defining for each of them $h_{k+1}(s_{k+1+n}(x) + j)(l) = s_{k+1+n'}(x')$; and then for each of the test formulas of the form $\mathsf{alloc}(\langle x, n\rangle + j)$ of a_{k+1}, we define $h_{k+1}(s_{k+1+n}(x) + j)(l_0) = s_{k+1+n}(x) + j$, for some $l_0 \notin \mathtt{Lab}_0$. Thanks to the distance $\mathtt{maxi}$ between variables, the test formulas about the heap which are not in a_{k+1} are not satisfied. Equalities $x = x'$ are preserved since the store has only been modified by a permutation. □

Note that we can extend lemma 4.2.1.4 to $\mathtt{SL}_s$ by considering test formulas of the form $x + i = x' + i'$. Sadly, the lemma above is essential and it is not possible to extend it to the whole logic $\mathtt{LTL}^{mem}$, even by allowing test formulas of the form $x + i = x' + i'$, as the resulting sets of formulas cannot be handled by Büchi automata. Indeed, we conjecture that automata with counters could handle these sets of formulas, but even this result would not be helpful to reach a decidable procedure, as their non-emptiness is undecidable.

Now, we can state our main complexity result.

Theorem 4.1. $\mathtt{Sat}(\mathtt{SL}^{RF}_s)$ and $\mathtt{Sat}(\mathtt{SL}^{CL}_s)$ are PSPACE-complete.

As a consequence, since $\mathtt{LTL}^{mem}(\mathtt{SL}^{LF}_s)$ is a syntactic fragment of $\mathtt{LTL}^{mem}(\mathtt{SL}^{RF}_s)$, the problem $\mathtt{Sat}(\mathtt{SL}^{LF}_s)$ is in PSPACE, and hence is PSPACE-complete.

Proof. The lower bound is from $\mathtt{LTL}$ [86]. Let t be an instance formula of $\mathtt{Sat}(\mathtt{SL}^{RF}_s)$ (for $\mathtt{Sat}(\mathtt{SL}^{CL}_s)$ replace below $\mathtt{mes}_t$ by $\mathtt{mes}_t[\mathtt{maxsize} \leftarrow 0]$). As seen earlier, t is satisfiable iff $\mathtt{Lang}^{\mathtt{mes}_t}(t) \cap \mathtt{Lang}^{\mathtt{mes}_t}_{sat}$ is nonempty. Hence, t is satisfiable iff $\mathtt{Lang}(\mathbb{A}^{\mathtt{mes}_t}_t) \cap \mathtt{Lang}(\mathbb{A}^{\mathtt{mes}_t}_{sat}) \neq \emptyset$.

The intersection automaton is of exponential size in the size of t and can be checked nonempty by a non-deterministic on-the-fly algorithm. This algorithm, for the non-emptiness problem of Büchi automata, is in NLOGSPACE, see [89]. The transition relation in the intersection automaton can be checked in polynomial space in the size of t. As a consequence, we obtain a non-deterministic polynomial space algorithm for testing satisfiability of t. As a non-deterministic polynomial space algorithm can be turned into a polynomial space algorithm, see for instance [4], we get the PSPACE upper bound. □

4.3.3 Other Decidable Problems

Let $\mathtt{Frag}$ be either the classical fragment or the record fragment. Lemma 4.1.4.1 provides a reduction from $\mathtt{Mc}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{Frag})$ to $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{Frag})$ based on a program-as-formula encoding. As we will see now, we may also reduce $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{Frag})$ to $\mathtt{Sat}(\mathtt{Frag})$ internalizing an approximation of the initial memory shape which the logical language cannot distinguish from the initial memory shape. As a consequence, the PSPACE upper bound for $\mathtt{Sat}(\mathtt{Frag})$ will entail the PSPACE upper bound for both $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{Frag})$ and $\mathtt{Mc}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{Frag})$.

Theorem 4.2. The problems $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}})$ and $\mathtt{Mc}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}})$ are PSPACE-complete. The problems $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{SL}^{\mathrm{CL}}_{\mathrm{s}})$ and $\mathtt{Mc}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{SL}^{\mathrm{CL}}_{\mathrm{s}})$ are also PSPACE-complete.

As a consequence, since $\mathtt{LTL}^{\mathrm{mem}}(\mathtt{SL}^{\mathrm{LF}}_{\mathrm{s}})$ is a syntactic fragment of $\mathtt{LTL}^{\mathrm{mem}}(\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}})$, the problems $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{SL}^{\mathrm{LF}}_{\mathrm{s}})$ and $\mathtt{Mc}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{SL}^{\mathrm{LF}}_{\mathrm{s}})$ are in PSPACE, and hence are PSPACE-complete.

Proof. We begin with the fragment $\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}}$. By lemma 4.1.4.1 and since $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}})$ is known to be PSPACE-hard, it remains to establish the PSPACE upper bound for $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}})$.

Given a formula t and an initial memory shape $(\mathtt{s},\mathtt{h})$, we shall build in polynomial time an instance of $\mathtt{Sat}(\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}})$, that is a formula $t^{ct}_{s,h} \in \mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}}$ such that t is satisfiable in a model with initial memory shape $(\mathtt{s},\mathtt{h})$ and constant heap iff $t^{ct}_{s,h}$ is satisfiable by a general model. Since we have shown that $\mathtt{Sat}(\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}})$ is in PSPACE, this guarantees that $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}})$ is in PSPACE. The idea of the proof is to internalize the initial memory shape and the fact that the heap is constant in the logic $\mathtt{Sat}(\mathtt{SL}^{\mathrm{RF}}_{\mathrm{s}})$. Actually, we will not exactly express that the heap is constant but the approximation we use will be sufficient for our purpose.

Apart from the variables of t, the formula $t^{ct}_{s,h}$ is built over additional variables in $\mathtt{X} = \{\mathtt{x}_\mathtt{i} : \mathtt{i} \in \mathtt{Dom(h)} \cup \mathtt{Im(s)}\} \cup \{\mathtt{x}_{\mathtt{i},l} : \mathtt{i} \in \mathtt{Dom(h)}, l \in \mathtt{Dom(h(i))}\}$ from $\mathtt{Specialvar}$. The formula $t^{ct}_{s,h}$ is of the form $\mathsf{always}(u_1 \wedge u_2 \wedge u_3) \wedge u_s \wedge u'$, where the subformulas are defined as follows.

- u_1 states that the heap is almost equal to $\mathtt{h}$ since we cannot forbid additional labels in the logical language. If $\mathtt{Dom(h)} = \{\mathtt{i}_1, \ldots, \mathtt{i}_k\}$ we define:

$$u_1 \triangleq (\bigwedge_{l \in \mathtt{Dom(h(i_1))}} \mathtt{x}_{\mathtt{i}_1} \overset{l}{\mapsto} \mathtt{x}_{\mathtt{i}_1,l}) * \ldots * (\bigwedge_{l \in \mathtt{Dom(h(i_k))}} \mathtt{x}_{\mathtt{i}_k} \overset{l}{\mapsto} \mathtt{x}_{\mathtt{i}_k,l})$$

- u_2 states which variables are equal and which ones are not, depending on the initial memory shape. It is a conjunction of simple formulas. As an example, for $\mathtt{i} \neq \mathtt{j} \in \mathtt{Dom(h)}$, a simple formula of u_2 is $\mathtt{x}_\mathtt{i} \neq \mathtt{x}_\mathtt{j}$. Similarly, if $\mathtt{h(i)}(l) = \mathtt{j}$ and $\mathtt{j} \in \mathtt{Dom(h)}$, then $\mathtt{x}_{\mathtt{i},l} = \mathtt{x}_\mathtt{j}$ is a simple formula of u_2. Details are omitted.

– u_3 states that the variables of X remain constant:

$$\bigwedge_{x \in X} x = \bigcirc x$$

– The formula u' is obtained from t by replacing each occurrence of $x \hookrightarrow^{l} x'$ by

$$x \hookrightarrow^{l} x' \wedge \bigwedge_{i \in Dom(h), l \notin Dom(h(i))} x \neq x_i .$$

The additional conjunction is useful because our logical language cannot state that a label is not in the domain of some allocated address.

– u_s states constraints about the initial store s: $u_s \triangleq \bigwedge_{x \in t} x = x_{s(x)}$.

It is then easy to check that t is satisfiable by a model with initial memory shape (s, h) and constant heap iff $t^{ct}_{s,h}$ is satisfiable by a general model.

As far as the results for the classical fragment are concerned, by lemma 4.1.4.1, there is a logarithmic space reduction from $Mc^{cons}_{init}(SL^{CL}_{s})$ to $Sat^{cons}_{init}(SL^{CL}_{s})$. Also, as done above, one can reduce $Sat^{cons}_{init}(SL^{CL}_{s})$ to $Sat(SL^{CL}_{s})$, with the additional concern of the arithmetic links between allocated locations, which is easy to handle. First we must add more variables, the variables x_{i-j} for $i \in Dom(h)$ and $\bigcirc_n x + i \hookrightarrow^{l} x'$ occurring in t for some n and some l. It is then sufficient to add a formula u_4 in the scope of the always operator in $t^{ct}_{s,h}$. This formula will describe all the pointers accessible with the syntactic resources of t, with the additional ability of using pointer arithmetic. We do not need to be concerned with the equalities among these new variables as the syntax of LTL^{mem} does not allow us to check equality between, say, x and $y + i$.

$$u_4 \triangleq \bigwedge_{i \in Dom(h)} \bigwedge_{l \in Dom(h(i))} \bigwedge_{\{j, \bigcirc^k z + j \hookrightarrow^{l} y \text{ occurs in } t \text{ for some } y,\ z,\ l \text{ and } k\}} x_{i-j} + j \hookrightarrow^{l} x_i$$

□

Theorem 4.3. $Mc^{cons}_{init}(SL_s)$ is PSPACE-complete.

As a consequence, since $LTL^{mem}(SL^{LF}_{s})$ is a syntactic fragment of $LTL^{mem}(SL^{RF}_{s})$, the problem $Mc^{cons}_{init}(SL^{\star}_{s})$ is in PSPACE, and hence is PSPACE-complete.

Proof. Since $Mc^{cons}_{init}(SL^{RF}_{s})$ is a subproblem of $Mc^{cons}_{init}(SL_s)$, theorem 4.2 entails the PSPACE-hardness. It remains to prove the PSPACE upper bound. The proof goes by designing a polynomial space reduction to the model-checking problem for propositional LTL. Let $(\mathbb{P}, s_0, h_0, t)$ be an instance of $Mc^{cons}_{init}(SL^{RF}_{s})$, where $\mathbb{P} = (B, d, b_I)$ is a program without destructive updates, (s_0, h_0) is an initial memory shape, and t is a temporal formula in $LTL^{mem}(SL_s)$. Let S be the finite set of stores $\{s : Im(s) \subseteq Im(s_0) \cup Im(h_0)\}$ restricted to variables occurring in $\mathbb{P}$ and t. Its cardinality is bounded by $(|Im(s_0) \cup Im(h_0)|)^{|t|+|\mathbb{P}|}$. All the memory shapes in the transition system $\mathbb{S}_{\mathbb{P}}$ restricted to the configurations reachable from the initial memory shape (s_0, h_0) are in $S \times \{h_0\}$, since $\mathbb{P}$ is without destructive updates.

Let wdw be one plus the maximal natural number j such that $\bigcirc^j x$ appears in t (size of the window made of consecutive states that need to be considered simultaneously). We define the transition graph $\mathbb{G} = (B_{\mathbb{G}}, \rightarrow, B_I)$ such that: $B_{\mathbb{G}} = B \times S^{wdw}$, B_I is the set of tuples

$(b_I, s_1, s_2, \ldots, s_{wdw})$ such that $(s_1, h_0), \ldots, (s_{wdw}, h_0)$ is a prefix of a run of $\mathbb{P}$ with initial memory shape (s_0, h_0), and the transition relation $\rightarrow$ is defined as follows:

$$(b, s_1, \ldots, s_{wdw}) \rightarrow (b', s'_1, \ldots, s'_{wdw})$$
$$\text{iff: } \begin{cases} s_{k+1} = s'_k,\ k = 1, \ldots, wdw - 1, \text{ and } \exists b \xrightarrow{g, instr} b' \in d \\ \text{such that } (s_1, h_0) \models g \text{ and } (s_2, h_0) \in [\![instr]\!](s_1, h_0). \end{cases}$$

We now define the propositional LTL model by associating to each vertex of the transition graph a set of propositional variables that are true. We define Prop to be the set of atomic formulas occurring in t, so that t can be seen as a propositional LTL formula over Prop. Then the LTL model is the vertex-labeled transition graph $\mathbb{G}' = (\mathbb{G}, L)$, with

$$L : B_{\mathbb{G}} \rightarrow Pow(Prop),\ (b, s_1, \ldots, s_{wdw}) \mapsto \{f \in Prop : s_1, \ldots, s_{wdw}, h_0 \models_{SL} f\}.$$

By construction, $\mathbb{G}', (b_I, s_1, s_2, \ldots, s_{wdw}) \models_{LTL^{mem}} t$ in LTL for some $(b_I, s_1, s_2, \ldots, s_{wdw}) \in B_I$ (existential version) if and only if $\mathbb{P}, (s_0, h_0) \models_{LTL^{mem}} t$. The model $\mathbb{G}'$ can be computed in polynomial space in the size of $(\mathbb{P}, s_0, h_0, t)$ in the sense that the (non-deterministic) transition function and the labelling function are computable in polynomial space. $\mathbb{G}'$ has an exponential size in the size of $(\mathbb{P}, s_0, h_0, t)$, but let us explain now why the existence of $(b_I, s_1, s_2, \ldots, s_{wdw}) \in B_I$ such that $\mathbb{G}', (b_I, s_1, s_2, \ldots, s_{wdw}) \models_{LTL^{mem}} t$ can be checked in polynomial space. Let $\mathbb{A}_t$ be the automaton recognizing the models of t over the set Prop of propositions: it has an exponential size in the size of $(\mathbb{P}, s_0, h_0, t)$, and so is the product with $\mathbb{G}'$. Now the existence of $(b_I, s_1, s_2, \ldots, s_{wdw}) \in B_I$ such that $\mathbb{G}', (b_I, s_1, s_2, \ldots, s_{wdw}) \models_{LTL^{mem}} t$ reduces to check the non-emptiness of $\mathbb{A}_t \cap \mathbb{G}'$, which is decidable in space proportional to $\log(|\ \mathbb{A}_t\ |) + \log(|\ \mathbb{G}'\ |)$ by a non-deterministic on-the-fly algorithm. The problem can therefore be solved in polynomial space in the size of $(\mathbb{P}, s_0, h_0, t)$ by a non-deterministic algorithm, and by Savitch's theorem this can be turned into a deterministic polynomial space algorithm. □

Theorem 4.4. $Sat^{cons}_{init}(SL^{\star}_s)$ is PSPACE-complete.

Proof. PSPACE-hardness is a consequence of the PSPACE-hardness of $Sat^{cons}_{init}(SL^{CL}_s)$ since SL^{CL}_s is a fragment of $SL^{\star}_s$. In order to get the PSPACE upper bound, we are going to reduce the problem $Sat^{cons}_{init}(SL^{\star}_s)$ to $Sat^{cons}_{init}(SL^{RF}_s)$. Let $(s_0, h), t$ be an instance of $Sat^{cons}_{init}(SL^{\star}_s)$. We shall build an instance $(s'_0, h), t'$ of $Sat^{cons}_{init}(SL^{RF}_s)$.

Let $I = Dom(h) \cup \{k - i \in \mathbb{N} : k \in Dom(h) \text{ and } \bigcirc^n x + i \text{ occurs in t}\}$. We use the injections $\langle k \rangle$ for each $k \in I$, and $\langle x, i \rangle$ for all x and i occuring in t in an expression of the form $\bigcirc^n x + i$ (possibly n or i is equal to zero). These extra variables defined in section 1.2.1 do not occur in t.

The initial store s'_0 is the extension of s_0 which maps $\langle k \rangle$ to k, and $\langle x, i \rangle$ to $s_0(x) + i$. Finally:

$$\begin{aligned} t' = \ & t[\bigcirc^n x + i \leftarrow \bigcirc^n \langle x, i \rangle] \\ & \wedge \text{always} \bigwedge_{k \in I} (\langle k \rangle = \bigcirc \langle k \rangle) \\ & \wedge \text{always} \bigwedge_{x+i \in t} \ \bigwedge_{(k+i) \in Dom(h)} (x = \langle k \rangle \Leftrightarrow \langle x, i \rangle = \langle k + i \rangle) \end{aligned}$$

s'_0 and t' have a polynomial size in the size of the instance $(s_0, h), t$.

Assume that $(s_0, h), t$ is accepted by the problem $Sat^{cons}_{init}(SL^{\star}_s)$. Then there is $(s_i)_{i \in \mathbb{N}}$ such that $(s_i, h)_{i \in \mathbb{N}} \models_{LTL^{mem}} t$. Let s'_i be s_i extended so as to map $\langle k \rangle$ to k and $\langle x, j \rangle$ to $s_i(x) + j$.

Clearly $(s'_i, h)_{i\in\mathbb{N}} \models_{LTL^{mem}} t[\bigcirc^n x + i \leftarrow \bigcirc^n\langle x, i\rangle]$. Our definition of each s'_i also ensures that $(s'_i, h)_{i\in\mathbb{N}} \models_{LTL^{mem}}$ always $\bigwedge_{k\in I}(\langle k\rangle = \bigcirc \langle k\rangle)$ since the value of a variable $\langle k\rangle$ is constantly equal to k, and that $(s'_i, h)_{i\in\mathbb{N}} \models_{LTL^{mem}}$ always $\bigwedge_{x+i\in t}\bigwedge_{(k+i)\in Dom(h)}(x = \langle k\rangle \Leftrightarrow \langle x, i\rangle = \langle k + i\rangle)$ since for all positions, the value of $\langle k + i\rangle$ is that of $\langle k\rangle$ plus i and the value of $\langle x, i\rangle$ is that of x plus i. So $(s'_i, h)_{i\in\mathbb{N}} \models_{LTL^{mem}} t'$, and therefore $(s'_0, h), t'$ is accepted by $Sat^{cons}_{init}(SL^{RF}_s)$.

Now, assume that $(s'_0, h), t'$ is accepted by $Sat^{cons}_{init}(SL^{RF}_s)$. Then there is a sequence $(s'_i)_{i\in\mathbb{N}}$ such that $(s'_i, h)_{i\in\mathbb{N}} \models_{LTL^{mem}} t'$. Then $(s'_i, h)_{i\in\mathbb{N}} \models_{LTL^{mem}}$ always $\bigwedge_{k\in I}(\langle k\rangle = \bigcirc \langle k\rangle)$, and so, at each time state i_0, we have $s'_{i_0}(\langle k\rangle) = s'_0(\langle k\rangle) = k$. Moreover, $(s'_i, h)_{i\in\mathbb{N}} \models_{LTL^{mem}}$ always $\bigwedge_{x+i\in t}\bigwedge_{(k+i)\in Dom(h)}(x = \langle k\rangle \Leftrightarrow \langle x, i\rangle = \langle k + i\rangle$, and so, if $k \in Dom(h)$ and $\bigcirc^n x + i$ occurs in t, we have $s'_{i_0+n}(x) = k - i$ iff $s'_{i_0+n}(\langle x, i\rangle) = k$ (Property I).

We write $h' \leq h$ when there is another heap h'' for which $h = h' * h''$. Let us prove by induction on subformulas t_0 of t that for all $i_0 \in \mathbb{N}$ and $h' \leq h$, we have $(s'_i, h')_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} t_0$ iff $(s'_i, h')_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} t_0[\bigcirc^n x + i \leftarrow \bigcirc^n\langle x, i\rangle]$. This will ensure that $(s'_i, h)_{i\in\mathbb{N}}, 0 \models_{LTL^{mem}} t$, so that $(s'_0, h), t$ is accepted by $Sat^{cons}_{init}(SL^{\star}_s)$, from which we will conclude that $(s_0, h), t$ is also accepted; indeed if Y_0 is the set of variables occurring in t the restriction $s'_{0|Y_0}$ is equal to $s_{0|Y_0}$. Here is the proof by induction:

- If t_0 is $\bigcirc^n x + i \hookrightarrow^l \bigcirc^{n'} y$, let $k = s'_{i_0+n}(\langle x, i\rangle)$.
  * Suppose that $k \notin Dom(h)$. We are going to prove that neither $(s'_i, h')_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} t_0[\bigcirc^n x + i \leftarrow \bigcirc^n \langle x, i\rangle]$, nor $(s'_i, h')_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} t_0$. First, it is clear that $(s'_i, h')_{i\in\mathbb{N}}, i_0 \not\models_{LTL^{mem}} t_0[\bigcirc^n x + i \leftarrow \bigcirc^n \langle x, i\rangle]$. Second, assume there is $k' \in Dom(h)$ such that $k' = s'_{i_0+n}(x) + i$. Thanks to (Property I), from $s'_{i_0+n}(x) = k' - i$, we get $s'_{i_0+n}(\langle x, i\rangle) = k'$, and so $k = k' \in Dom(h)$, which leads to a contradiction. So there is no such k', and not $(s'_i, h')_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} t_0$.
  * Now suppose that $k \in Dom(h)$. We have $s'_{i_0+n}(x) = k = s'_{i_0+n}(\langle x, i\rangle) - i$ thanks to (Property I). Then, $h'(s'_{i_0+n}(x) + i)(l) = s'_{i_0+n'}(y)$ iff $h'(s'_{i_0+n}(\langle x, i\rangle))(l) = s'_{i_0+n}(\langle y, 0\rangle)$. And, $(s'_i, h')_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} t_0$ iff $(s'_i, h')_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} t_0[\bigcirc^n x + i \leftarrow \bigcirc^n\langle x, i\rangle]$.
- If $t_0 = f_1 * f_2$, then there are two heaps h'_1 and h'_2 such that $(s'_i, h'_1)_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} f_1$ and $(s'_i, h'_2)_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} f_2$. By the induction hypothesis, and since $h = (h'_1 * h'_2) * h'' = h'_1 * (h'_2 * h'')$, we can state that: $(s'_i, h'_1)_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} f_1[\bigcirc^n x + i \leftarrow \bigcirc^n \langle x, i\rangle]$ iff $(s'_i, h'_1)_{i\in\mathbb{N}}, i_0 \models_{LTL^{mem}} f_1$; and the same equivalence is true for h'_2. From the two equivalences for h'_1 and h'_2, we can conclude the same equivalence for $h' = h'_1 * h'_2$.

Other cases of the induction are straightforward. □

If we allow the operator $-\!\!*$ in the above theorem 4.4, the proof may not be adapted, since we would have to deal with heaps which are not sub-heaps of h in the induction step.

4.4 Undecidability Results

As a preliminary remark, we will use the standard abbreviation Σ^0_1 for the set of recursively enumerable sets. A formal definition and more information about this topic can be found in [78].

In this section, we show several undecidability results by using reduction from problems for Minsky machines. So, we first give the definition of a Minsky machine.

Definition 4.4.0.1. A Minsky machine $\mathbb{M}$ consists of two counters $\mathtt{c}_1$ and $\mathtt{c}_2$, and a sequence of $\mathtt{n} \geq 1$ instructions of one of the forms below. The instructions can be seen as control states, we call them $\mathtt{b}$, $\mathtt{b}'$ or $\mathtt{b}''$ below.

b: $\mathtt{c}_\mathtt{i} := \mathtt{c}_\mathtt{i} + 1$; goto b′

b: if $\mathtt{c}_\mathtt{i} = 0$ then goto b′ else $\mathtt{c}_\mathtt{i} := \mathtt{c}_\mathtt{i} - 1$; goto b″.

In a non-deterministic machine, after an increment or a decrement, a non-deterministic choice of the form "goto $\mathtt{b}'$ or goto $\mathtt{b}''$" is performed. The configurations of $\mathbb{M}$ are triples $(\mathtt{b}, \mathtt{m}_1, \mathtt{m}_2)$, where $\mathtt{b} \in \{\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{n}\}$ and $\mathtt{m}_1, \mathtt{m}_2 \geq 0$ are the current values of the control state and the two counters $\mathtt{c}_1$ and $\mathtt{c}_2$, respectively. The consecution relation on configurations is defined in the obvious way. A computation of $\mathbb{M}$ is a sequence of related configurations, starting with the initial configuration $(1, 0, 0)$.

We will reduce the halting problem and the recurring problem for Minsky machines. The halting problem consists of determining whether the machine can reach a configuration with control state $\mathtt{b}_\mathtt{n}$. The recurring problem consists in determining whether the machine has a computation with the control state $\mathtt{b}_\mathtt{n}$ repeated infinitely often.

Different encodings of counters are used here. For instance, one is inspired from [7]: a counter $\mathtt{c}$ with value $\mathtt{n}$ is represented by a list of length $\mathtt{n}$ pointed to by a variable $\mathtt{x}_\mathtt{c}$ dedicated to the counter $\mathtt{c}$. The same idea is used in the proof of theorem 4.5 below. Alternatively, in order to show undecidability of $\mathtt{Sat}(\mathtt{SL}_\mathtt{s})$, we encode counters by relying on pointer arithmetic and properties of heaps. In the case of a problem that involves an existential quantification on the initial heap, the maximal value of the counters can be guessed, as illustrated in the proof of the theorem below. Finally, the programs without destructive updates can simulate finite computations of Minsky machines on counters bounded by the size of some parts of the heap (the length of a list).

Theorem 4.5. For any fragment $\mathtt{Frag} \in \{\mathtt{SL}_\mathtt{s}^\mathtt{LF}, \mathtt{SL}_\mathtt{s}^\star, \mathtt{SL}_\mathtt{s}^\mathtt{CL}, \mathtt{SL}_\mathtt{s}^\mathtt{RF}\}$, the problems $\mathtt{Sat}^{\mathrm{cons}}(\mathtt{Frag})$ and $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{Frag})$ are Σ_1^0-complete. The problem $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s})$ is also Σ_1^0-complete.

Proof. First, let us prove that these problems are in Σ_1^0. By theorem 4.2, $\mathtt{Sat}^{\mathrm{cons}}_{\mathtt{init}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ is decidable in polynomial space using a finite abstraction argument. Hence, $\mathtt{Sat}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ is in Σ_1^0 by adding an existential quantification over the initial memory shape. Similarly, by theorem 4.2, $\mathtt{Mc}^{\mathrm{cons}}_{\mathtt{init}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ is decidable in polynomial space. Hence, $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ is also in Σ_1^0. It is possible to reason in the exact same way for $\mathtt{Sat}^{\mathrm{cons}}(\mathtt{Frag})$ and $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{Frag})$ with $\mathtt{Frag} \in \{\mathtt{SL}_\mathtt{s}^\mathtt{CL}, \mathtt{SL}_\mathtt{s}^\mathtt{RF}\}$ from the decidability results of theorem 4.2, for $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\star)$ and $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s})$ from the decidability results of theorem 4.3, and for $\mathtt{Sat}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\star)$ from the decidability results of theorem 4.4.

Now, let us prove that the problems are Σ_1^0-hard. As $\mathtt{Sat}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ is a subproblem of all the studied $\mathtt{Sat}^{\mathrm{cons}}$ problems, proving the Σ_1^0-hardness of $\mathtt{Sat}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ will entail the Σ_1^0-hardness of all the others. Similarly, as $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ is a subproblem of all the studied $\mathtt{Mc}^{\mathrm{cons}}$ problems, proving the Σ_1^0-hardness of $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ will entail the Σ_1^0-hardness of all the others. As a consequence, we only need to prove that $\mathtt{Sat}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ and $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ are Σ_1^0-hard. Additionally, by lemma 4.1.4.1, we only need to show that $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$ is Σ_1^0-hard.

We reduce the Σ_1^0-complete halting problem for Minsky machines to $\mathtt{Mc}^{\mathrm{cons}}(\mathtt{SL}_\mathtt{s}^\mathtt{LF})$. The halting problem consists of determining whether $\mathbb{M}$ can reach a configuration with control state $\mathtt{b}_\mathtt{n}$.

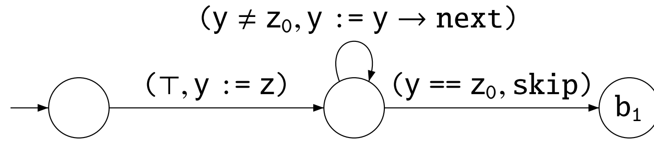


Figure 4.4: Checking that $\mathtt{z}$ points to a list

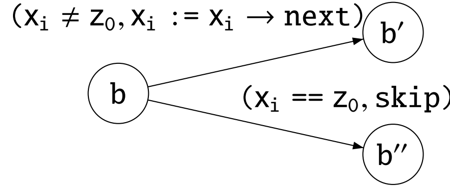


Figure 4.5: Simulating a decrement

Let us build a formula t and a program $\mathbb{P}$ in $\mathtt{Prog}^{\mathtt{ct}}$ such that the existence of some memory shape $(\mathtt{s}_0, \mathtt{h}_0)$ for which $\mathbb{P}, (\mathtt{s}_0, \mathtt{h}_0) \models_{\mathrm{LTL}^{\mathrm{mem}}} \mathsf{t}$ is equivalent to the fact that the machine $\mathbb{M}$ reaches a configuration with control state $\mathtt{b}_n$. In order to encode the values of counters, we consider a variable $\mathtt{z}$ pointing to a list ending on $\mathtt{x}_0$ (as shown below) in the initial memory shape $(\mathtt{s}_0, \mathtt{h}_0)$:

$$\mathtt{z} \hookrightarrow^{\mathtt{next}} \Box \hookrightarrow^{\mathtt{next}} \ldots \Box \hookrightarrow^{\mathtt{next}} \Box \hookrightarrow^{\mathtt{next}} \mathtt{z}_0$$

The variables $\mathtt{z}$ and $\mathtt{z}_0$ remain constant along any execution of $\mathbb{P}$ and the length of the list is greater than the maximal value of the counters in some finite computation (hopefully ending at the instruction corresponding to control state $\mathtt{b}_n$). We consider also the variables $\mathtt{x}_1$ and $\mathtt{x}_2$ and along any execution of $\mathbb{P}$, each variable $\mathtt{x}_i$ points to a cell of the above sequence: the length of the list starting at $\mathtt{x}_i$ encodes the value of the counter $\mathtt{c}_i$. Hence, in $\mathbb{P}$, each $\mathtt{x}_i$ is initialized as equal to $\mathtt{z}_0$.

The program $\mathbb{P}$ is structured by the following stages:

1. Check that $\mathtt{z}$ points to a list;
2. Initialize the variables;
3. Simulate $\mathbb{M}$.

Figure 4.4 shows how to perform stage 1 with a simple loop, which can be seen as a while loop. Observe that checking whether a counter is equal to zero corresponds in $\mathbb{P}$ to an equality test with $\mathtt{z}_0$. In order to simulate $\mathbb{M}$, its structure can be embedded in the control graph of $\mathbb{P}$. For instance, a decrement instruction is encoded in $\mathbb{P}$ by the transitions shown in figure 4.5. An increment instruction requires a bit more care and its encoding in $\mathbb{P}$ is presented in figure 4.6. Indeed, the auxiliary variables $\mathtt{y}$ and $\mathtt{y}'$ initialized to $\mathtt{z}$ visit the list until meeting $\mathtt{x}_i$.

In the above encoding, every instruction $\mathtt{b}$ in $\mathbb{M}$ corresponds to a control state of $\mathbb{P}$. Hence, the formula t is simply $\mathsf{sometimes}(\mathtt{b}_n)$: as stated earlier, we may encode propositional variable $\mathtt{b}_n$ by additional variables dedicated only for this purpose.

It is then easy to show that there is an initial memory shape $(\mathtt{s}_0, \mathtt{h}_0)$ such that $\mathbb{P}$ reaches the control $\mathtt{b}_n$ starting with $(\mathtt{s}_0, \mathtt{h}_0)$ iff the machine $\mathbb{M}$ reaches the control state $\mathtt{b}_n$. For this purpose, observe that both $\mathbb{P}$ and $\mathbb{M}$ are deterministic. $\Box$

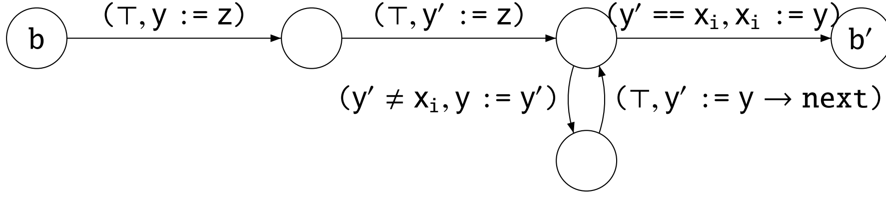


Figure 4.6: Simulating an increment

By contrast, programs with destructive update can work with unbounded heaps, and by using the representation of counters as above, they can faithfully simulate a Minsky machine, even if the initial heap is an empty heap, without any bound on the counters. Then, as LTL can express repeated accessibility, Σ_1^1-hardness can be obtained.

Theorem 4.6. The problems $\mathtt{Mc}(\mathrm{SL}_s^{\mathrm{LF}})$ and $\mathtt{Mc}_{\mathtt{init}}(\mathrm{SL}_s^{\mathrm{LF}})$ are Σ_1^1-complete.

As a consequence for more expressive logics, the problems $\mathtt{Mc}(\mathtt{Frag})$ and $\mathtt{Mc}_{\mathtt{init}}(\mathtt{Frag})$ for $\mathtt{Frag} \in \{\mathrm{SL}_s^{\mathrm{CL}}, \mathrm{SL}_s^{\mathrm{RF}}, \mathrm{SL}_s^{\star}, \mathrm{SL}_s\}$ are Σ_1^1-hard, and hence Σ_1^1-complete.

Proof. It is possible to reduce the recurring problem for non-deterministic Minsky machines to $\mathtt{Mc}(\mathrm{SL}_s^{\mathrm{LF}})$ and to $\mathtt{Mc}_{\mathtt{init}}(\mathrm{SL}_s^{\mathrm{LF}})$. This problem is Σ_1^1-hard [2]. The question is whether the machine has a computation with the control state b_n repeated infinitely often; and this can be expressed by $\mathsf{always}(\mathsf{sometimes}(b_n))$ in $\mathrm{LTL}^{\mathrm{mem}}$.

The proof is quite similar to the proof of theorem 4.5 except that there is no maximal value of the counters, the initial heap is empty (which can be expressed in $\mathrm{LTL}^{\mathrm{mem}}$), and the behavior of counters is encoded by updating the memory shape. For instance, incrementing c_i amounts to execute $x_i := \mathtt{cons}(\mathtt{next} : x_i)$ (the length of the list pointed by x_i is incremented), decrementing c_i amounts to execute $x_i := x_i \rightarrow \mathtt{next}$. Zero tests are encoded by equality tests with z_0 and the initial values of the variables is equal to z_0. Details are omitted since there are no technical difficulties. □

Now, let us explain how to encode increment and decrement with separating connectives and pointer arithmetic. Observe that expressions of the form $x = y + 1$ are not allowed in $\mathrm{LTL}^{\mathrm{mem}}$. We circumvent this obstacle in two different ways: using non-aliasing expressed by the separating conjunction, and using the precise pointing assertion $x \mapsto^{\mathtt{next}} y$ stating that the heap contains only one cell, in conjunction with the $-\!\!*$ operator. We assume a variable z_0, with constant value ($\mathsf{always}(\bigcirc z_0 = z_0)$), which can be considered as a substitute for the null constant.

$$\begin{array}{lcl}
t^{\star}_{x++} & = & (\bigcirc x \hookrightarrow^{\mathtt{next}} z_0 \wedge x + 1 \hookrightarrow^{\mathtt{next}} z_0) \wedge \neg(\bigcirc x \hookrightarrow^{\mathtt{next}} z_0 \star x + 1 \hookrightarrow^{\mathtt{next}} z_0) \\
t^{\star}_{x--} & = & (\bigcirc x + 1 \hookrightarrow^{\mathtt{next}} z_0 \wedge x \hookrightarrow^{\mathtt{next}} z_0) \wedge \neg(\bigcirc x + 1 \hookrightarrow^{\mathtt{next}} z_0 \star x \hookrightarrow^{\mathtt{next}} z_0) \\
t^{-\!*}_{x++} & = & (\bigcirc x \mapsto^{\mathtt{next}} z_0) -\!\!* x + 1 \mapsto^{\mathtt{next}} z_0 \\
t^{-\!*}_{x--} & = & (x \mapsto^{\mathtt{next}} z_0) -\!\!* \bigcirc x + 1 \mapsto^{\mathtt{next}} z_0
\end{array}$$

The formulas based on the separating conjunction correctly express increment and decrement when the cells at indexes $x + 1$ and $\bigcirc x$ are allocated, whereas formulas based on the operator $-\!\!*$ work when the heap is empty.

Theorem 4.7. The problems $\mathtt{Sat}(\mathrm{SL}_s)$, $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathrm{SL}_s)$, $\mathtt{Sat}^{\mathtt{cons}}(\mathrm{SL}_s)$ and $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}(\mathrm{SL}_s)$ are all Σ_1^1-complete.

Proof. Let $\mathtt{Sat}^{\cdots}_{\cdots}(\mathtt{SL_s})$ be any satisfiability problem among the four studied variants. As announced, we are going to reduce the recurrence problem for non-deterministic Minsky machines to $\mathtt{Sat}^{\cdots}_{\cdots}(\mathtt{SL_s})$. In this proof, any model will be written $\mathtt{mod} = (\mathtt{s_n}, \mathtt{h_n})_{n\geq 0}$.

Let t_0 be the formula $\mathsf{always}(\mathsf{emp} \wedge (\bigcirc z_0 = z_0))$, which ensures that the heap will always be empty – the part about z_0 being constant is unnecessary but makes the idea of z representing null more consistent. Increment and decrement are performed thanks to the formulas $t^{\twoheadrightarrow\star}_{x++}$ and $t^{\twoheadrightarrow\star}_{x--}$, respectively. For any model $\mathtt{mod}$ such that $\mathtt{mod}, 0 \models_{\mathrm{LTL^{mem}}} t_0$, and for any $\mathtt{i_0}$, we have $\mathtt{mod}, \mathtt{i_0} \models_{\mathrm{LTL^{mem}}} t^{\twoheadrightarrow\star}_{x_i++}$ iff $\mathtt{s_{i_0}(x_i) + 1 = s_{i_0+1}(x_i)}$. Indeed, the formula $t^{\twoheadrightarrow\star}_{x_i++}$ is satisfied by the model iff the pointer existing in the right member of the formula is satisfied by the current model extended by any heap satisfying the right member. Since the model has an empty heap by t_0, this happens if and only if the pointer has been added by the wand, which means if and only if it also satisfies the left member of the wand. By the definition of the semantics of $\mathtt{LTL^{mem}}$, both sides of the wand are satisfied by the same heap if and only if the same memory cell is located both at the image of $\langle x, 1\rangle$ through the store $\mathtt{s'_{i_0}}$ and at one plus the image of $\mathtt{x}$ through the store $\mathtt{s_{i_0}}$, where $\mathtt{s'_{i_0}}(\langle x, 1\rangle) = \mathtt{s_{i_0+1}(x)}$. In other words, if and only if $\mathtt{s_{i_0}(x_i) + 1 = s_{i_0+1}(x_i)}$. Hence, we have a means to encode increment.

Very similarly, $\mathtt{mod}, \mathtt{i_0} \models_{\mathrm{LTL^{mem}}} t^{\twoheadrightarrow\star}_{x_i--}$ and $\mathtt{s_{i_0}(x_i) > 0}$ iff $\mathtt{s_{i_0}(x_i) - 1 = s_{i_0+1}(x_i)}$. The fact that a counter does not change is encoded by $\mathtt{x_i} = \bigcirc \mathtt{x_i}$. Given that $t_1 = \mathsf{always}(x_{zero} = \bigcirc x_{zero})$ holds, zero tests are encoded by $\mathtt{x_i} = x_{zero}$.

Given a non-deterministic Minsky machine $\mathbb{M}$, we write u_b to denote the formula encoding instruction b. For instance for the instruction "b: if $c_1 = 0$ then goto b' else $c_1 := c_1 - 1$; goto b'' or goto b''';" the formula u_b is equal to the formula below:

$$\mathsf{always}((b \wedge x_1 \neq x_{zero}) \Rightarrow (x_2 = \bigcirc x_2 \wedge (\bigcirc b'' \vee \bigcirc b''') \wedge t^{\twoheadrightarrow\star}_{x_1--})) \wedge$$

$$\mathsf{always}((b \wedge x_1 = x_{zero}) \Rightarrow (x_1 = \bigcirc x_1 \wedge x_2 = \bigcirc x_2 \wedge \bigcirc b')).$$

Finally, let t_2 be a formula stating that each position corresponds to a unique configuration and the first instruction is b_1: $t_2 = \mathsf{always}(\bigwedge_b (\bigwedge_{b' \neq b} (b \rightarrow \neg b'))) \wedge b_1$.

Hence, $(x_1 = x_2 = x_{zero}) \wedge t_0 \wedge t_1 \wedge \bigwedge_b u_b \wedge \mathsf{always}(\mathsf{sometimes}(b_n))$ is satisfiable iff $\mathbb{M}$ has a computation with instruction b_n repeated infinitely often. □

Theorem 4.8. The problem $\mathtt{Sat}(\mathtt{SL^{\star}_s})$ is Σ^1_1-complete.

The proof of theorem 4.8 is similar to the proof of theorem 4.7 except that increment and decrement are performed with the formulas $t^{\star}_{x++}$ and $t^{\star}_{x--}$ respectively, and the heap is not always empty: at each increment or decrement, it has size precisely 1.

Conclusion

Summary of this Chapter

We have introduced a temporal logic $\mathtt{LTL^{mem}}$ whose assertion language is a quantifier free separation logic, for which we have introduced five fragments and seven decision problems for each fragment. We have categorized all of these problems in terms of complexity. Figure 4.8 contains a summary of the results. All problems are categorized as complete in their class.

In a nutshell, we have shown that our model-checking problems, which encode a halting problem thanks to until, are undecidable when the program has access to a memory heap of unbounded size ($\mathtt{Mc}$, $\mathtt{Mc}^{\mathtt{cons}}$ and $\mathtt{Mc}_{\mathtt{init}}$), and decidable otherwise ($\mathtt{Mc}^{\mathtt{cons}}_{\mathtt{init}}$).

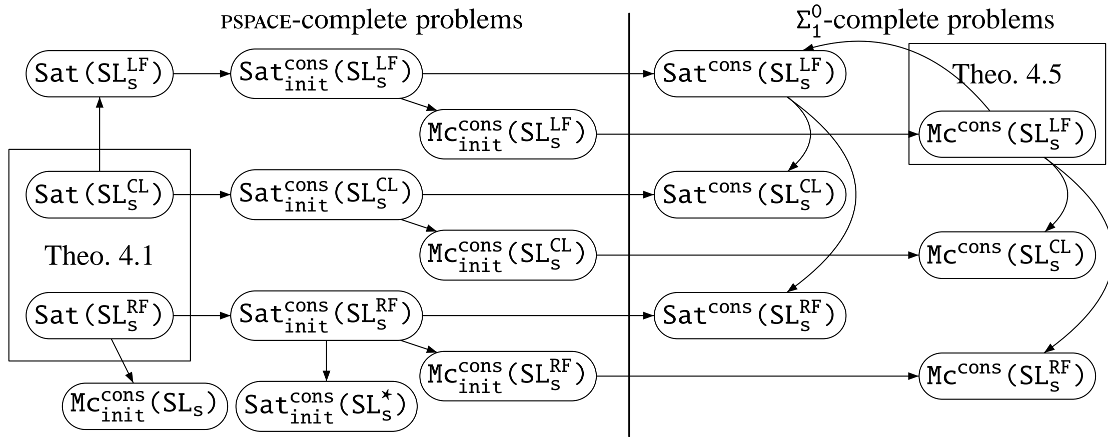
Concerning our satisfiability problems, a special care must be granted to a subtle interplay between the temporal features, the separation connectives and the pointer arithmetic we use. When this interaction is possible, it leads to undecidability ($\mathtt{Sat(SL}^{\star}_{\mathtt{s}}\mathtt{)}$ and $\mathtt{Sat}^{\cdots}_{\cdots}\mathtt{(SL_s)}$). When this interaction is not possible, the problems are decidable if and only if the control on the heap is effective on a bounded amount of locations. This amount of locations can be bounded for two reasons. It is clearly bounded when the initial memory heap is part of the problem and constant ($\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}$). It can also be bounded because the memory cells which are not the image of a variable of the studied formula through the store can change uncontrollably between two consecutive memory states: in this last case we have been able to abstract the heap with a set of formulas describing a heap bounded by the syntactic resources of the studied formula ($\mathtt{Sat(SL}^{\mathtt{LF}}_{\mathtt{s}}\mathtt{)}$, $\mathtt{Sat(SL}^{\mathtt{CL}}_{\mathtt{s}}\mathtt{)}$ and $\mathtt{Sat(SL}^{\mathtt{RF}}_{\mathtt{s}}\mathtt{)}$).

We obtained all these results from few direct proofs and many subsequent reductions between decision problems. Figure 4.7 shows the reductions between problems leading to half of the results. Curved lines represent reductions for proving hardness in a class. Straight lines represent reductions for showing that a problem belongs to its class.

Related Work

The interest of the model-checking of programs with heap updates stems from early works on automata-based verification. Decision procedures are obtained at the cost of limitations: to restrict the programming language, see for instance Bardin et al. in [7], or to define approximations as done in [90, 47]. Previous temporal logics designed for pointer verification include in particular: the evolution temporal logic of Yahav et al. [90], based on the three-valued logic abstraction method that made the success of the three-valued logic assertion engine presented by Lev-Ami and Sagiv in [69]; and the navigation temporal logic of Distefano, Katoen and Rensink [47], based on a tableau method quite similar to our automata-based reduction. In these works, the assertion language for states is quite rich, as it includes, for instance, a list predicate, the quantification over addresses, and a freshness predicate. Because of this high expressive power, only incomplete abstractions are proposed, whereas we stick to exact methods. Similarly, we should also mention the work of Katoen, Noll and Rieger [65], published in the same year as the work presented in this chapter, which presents sound heuristics for the problem that we call $\mathtt{Mc}_{\mathtt{init}}\mathtt{(SL}^{\mathtt{LF}}_{\mathtt{s}}\mathtt{)}$. As an additional difference with these works, our work addresses models with constant heaps and pointer arithmetic, which has not been done so far and leads to a different perspective.

The abstractions we made of memory states are similar to resource graphs of Galmiche and Méry from [54, 55]. We have chosen to use them following the work of Lozes [73]. The use we make of them is a variant of the automata-based approach introduced by Vardi and Wolper in [89] for plain LTL and further developed with concrete domains of interpretation by Demri and D'Souza in [43]. From a logical perspective, the logic $\mathtt{LTL}^{\mathtt{mem}}$ can be seen as a many-dimensional logic as for instance Gabbay et al. in [52] since $\mathtt{LTL}^{\mathtt{mem}}$ contains a temporal dimension and the spatial dimension for memory shapes. Interesting examples of many-dimensional logics can be found in [9, 5, 52, 43].



Straight lines prove a problem belongs to a class, curved lines prove hardness

Figure 4.7: Reductions

	Mc	Mc^{cons}	Mc$^{cons}_{init}$	Mc$_{init}$	Sat	Sat^{cons}	Sat$^{cons}_{init}$
SL$_s^{LF}$	Σ^1_1-c. th. 4.6	Σ^0_1-c. th. 4.5	PSPACE-C. th. 4.2	Σ^1_1-c. th. 4.6	PSPACE-C. th. 4.1	Σ^0_1-c. th. 4.5	PSPACE-C. th. 4.2
SL$_s^{RF}$	Σ^1_1-c. th. 4.6	Σ^0_1-c. th. 4.5	PSPACE-C. th. 4.2	Σ^1_1-c. th. 4.6	PSPACE-C. th. 4.1	Σ^0_1-c. th. 4.5	PSPACE-C. th. 4.2
SL$_s^{CL}$	Σ^1_1-c. th. 4.6	Σ^0_1-c. th. 4.5	PSPACE-C. th. 4.2	Σ^1_1-c. th. 4.6	PSPACE-C. th. 4.1	Σ^0_1-c. th. 4.5	PSPACE-C. th. 4.2
SL$_s^{\star}$	Σ^1_1-c. th. 4.6	Σ^0_1-c. th. 4.5	PSPACE-C. th. 4.3	Σ^1_1-c. th. 4.6	Σ^1_1-c. th. 4.8	Σ^0_1-c. th. 4.5	PSPACE-C. th. 4.4
SL$_s$	Σ^1_1-c. th. 4.6	Σ^0_1-c. th. 4.5	PSPACE-C. th. 4.3	Σ^1_1-c. th. 4.6	Σ^1_1-c. th. 4.7	Σ^1_1-c. th. 4.7	Σ^1_1-c. th. 4.7

Figure 4.8: Complexity of reasoning about programs with pointer variables

Perspectives

Assuming that the heap is constant is subject to promising developments. Indeed, it is then possible to define spatial operators at the same syntactic level as temporal operators, and write formulas as for instance $(x_1 = x \wedge ((x \hookrightarrow \bigcirc x) \text{ until } (x \hookrightarrow x_0))) * (y \mapsto x_0)$. Observe that this formula does not belong to LTL^{mem}. This might be a way to specify the modularity of programs without destructive updates, but there are other points of interest we will try to advocate now.

Recursion with Local Parameters

The constant heap semantics provides an original viewpoint for recursion with local parameters and local quantification. The design of decision procedures in the presence of general recursive predicates was introduced by Berdine et al. in [10], as well as incomplete methods of inference

even though they are apparently good in practice. Complete methods have been proposed for some standard recursive structures such as trees, lists, or doubly-linked lists as by Berdine, Calcagno and O'Hearn in [12]. But we are not aware of complete methods for a general form of recursive data structures defined on top of separation logic, and we believed that a logic close to ours could give an alternative way of specifying recursion, although we did not manage to characterize an interesting decidable fragment.

In order to be a bit more precise, let us consider the fragment of recursive separation logic where all recursive formulas are of the form:

$$\mu P(x_1,\ldots,x_k).\, t(x_1,\ldots,x_k) \vee \exists x'_1,\ldots,x'_k.\, u(x_1,\ldots,x_k,x'_1,\ldots,x'_k) \wedge P(x'_1,\ldots,x'_k). \quad (4.1)$$

This fragment is rich enough to express singly-linked lists, cyclic lists, and doubly-linked lists. However, we conjecture that it is not expressive enough for trees and directed acyclic graphs. We conjecture that deciding satisfiability in the fragment of recursive separation logic mentioned above reduces to $\mathtt{Sat}^{\mathtt{cons}}(\mathtt{SL}_s)$, and the model-checking problem reduces to $\mathtt{Sat}^{\mathtt{cons}}_{\mathtt{init}}$, considering that (4.1) can be rewritten as:

$$(t(x_1,\ldots,x_k,\bigcirc x_1,\ldots,\bigcirc x_k))\ \mathsf{until}\ (u(x_1,\ldots,x_k)).$$

In this perspective, from our results could arise interesting decidability results for the model-checking problem of some of the recursive separation logic with local quantifiers. For satisfiability, we expect to define decidable fragments for $\mathtt{Sat}^{\mathtt{cons}}(\mathtt{SL}_s)$, for instance considering the techniques for proving decidability of checking temporal properties of so-called flat programs without destructive updates introduced by Finkel, Lozes and Sangnier in [51]. Another interesting fragment of recursive separation logic is probably the one where recursion is guarded by the separation operator $*$, but we do not currently see how to treat it in the temporal logic perspective.

Programs as formulas

Let us speculate a bit more. We may take advantage of expressing programs as formulas in order to reduce model-checking problems to satisfiability problems, a known approach since the work of Sistla and Clarke in [86]. For programs without destructive update, we take advantage of lemma 4.1.4.1. Moreover, we believe we can extend this result to programs with updates, but with a slightly different perspective. The constant heap semantics can be helpful to define the input-output relation of programs, even with destructive updates, provided some conditions on the way the program read and write over the memory are satisfied. To do so, we could study the extension of $\mathtt{LTL}^{\mathtt{mem}}$ in which two predicates $\hookrightarrow_0$ and $\hookrightarrow_1$ are used instead of the single $\hookrightarrow$, and for which the models are couples of state sequences with constant heap, that is tuples $((s_i)_{i\geq 0}, h_0, h_1)$. Let us define the input-output relation $R^{\mathtt{IO}}_{\mathbb{P}}$ of a program $\mathbb{P}$ as : for all (s_0,h_0), (s_1,h_1), $(s_0,h_0)R^{\mathtt{IO}}_{\mathbb{P}}(s_1,h_1)$ if there is a run of $\mathbb{P}$ that starts with (s_0,h_0) and ends with (s_1,h_1). Then we conjecture that for an interesting class of programs, this relation is definable in $\mathtt{LTL}^{\mathtt{mem}}$ extended with $\hookrightarrow_0$ and $\hookrightarrow_1$. Basically, the encoding of the control of the program will be the same as for programs without destructive updates, but the encoding of the instructions will be different. For instance, $x \to \mathsf{l} := y$ implies $(\bigcirc x) \hookrightarrow^{\mathsf{l}}_1 y$ whereas $x := y \to \mathsf{l}$ implies $y \hookrightarrow^{\mathsf{l}}_0 \bigcirc x$. A precise encoding remains to be found.

Conclusion

In chapter 2, we have shown that first-order separation logic with one selector $\mathtt{SL}$ is undecidable. Also, $\mathtt{SL}$ without wand is decidable with non-elementary complexity, as well as its extension with a restricted magic wand sufficiently interesting to replace all the occurrences of the ordinary magic wand in the usual Hoare-style rules that use it. Finally, we have characterized the expressive power of first-order separation logic over models with any number of selectors.

In chapter 3, we have given a wide picture of the decidability status of the satisfiability problem for separation logic dealing with data. An interesting result arose: dropping the restricted operator $-\!\!*_{\mathtt{n}}$ and restricting data comparisons to local comparisons makes the satisfiability problem decidable, still being able to specify local reasoning and express properties about ordered recursive structures.

In chapter 4, we have introduced a temporal logic $\mathtt{LTL}^{\mathtt{mem}}$ for which assertion language is quantifier free separation logic, and provided a complete characterization of the complexity of 35 satisfiability and model-checking problems we have defined. This draws clear borders not to be crossed if one wants to adapt separation logic to temporal reasoning while defining a decidable logic.

We have ideas about how to extend each of the chapters presented. First, we conjecture that $\mathtt{SL}$ with only two variables can encode $\mathtt{SO}$. Then, we expect our decidability results for $\mathtt{SL}_{\mathtt{v}}^{\mathtt{guarded}}$ to extend to more complex data structures that would have a decidable $\mathtt{MSO}$ theory (trees, doubly-linked lists, lists of lists, and more generally tree-width bounded structures), and to more complex short-distance comparisons (such as $\mathtt{n}$-th successor or brothers). Finally, we expect our decidability result for $\mathtt{Sat}(\mathtt{SL}_{\mathtt{s}}^{\mathtt{RF}})$ to extend to branching time, and we think that there is a way to extend $\mathtt{LTL}^{\mathtt{mem}}$ so as to express properties in a way more related to a Hoare logic with preconditions and postconditions.

Tables of Notations

Unless otherwise indicated below, the capital of a letter denotes a set whose elements are denoted by the lower case of the same letter– for instance $\mathtt{a}$ is a letter and $\mathtt{A}$ is an alphabet. An exception to this rule are the capital letters of the font $\mathbb{A}$, $\mathbb{B}$, etc. Similarly, the first letter a notation made of multiple letters is a capital letter iff they denote a set. When a mathematical object is denoted by a single letter, figure A allows in any case to know which type of object is denoted.

The notations are divided in four figures. Figure A summarizes the notations for a certain type of mathematical object, such as integers or letters. Figure B summarizes general notations for a single item or mathematical tool, uniquely defined. Figure C summarizes the names for logical formalisms. Figure D summarizes the notations for the main logical operators.

a	A letter
$\mathbb{A}$	An automaton
atom	An atomic formula
b	A state
c	A counter
d	A transition function
e	A data environment
E	A second-order environment
expr	An expression
f	A formula describing a memory state
Frag	A fragment of a logic
g	A formula describing a memory state
$\mathbb{G}$	A graph
h	A heap
i	An integer
instr	An instruction
j	An integer
k	An integer
l	A label
Lang	A language
m	An integer
$\mathbb{M}$	A Minsky machine
mes	A measure
mod	A temporal model
n	An integer
o	A data value
P	A second-order variable
$\mathbb{P}$	A program
Q	A second-order variable
R	A relation
s	A store
$\mathbb{S}$	A transition system
t	A temporal formula
tr	A translation between logics
u	A temporal formula
v	A data variable
w	A program variable
wd	A word
x	A first-order variable
y	A first-order variable
z	A first-order variable

Figure A: Notations for a type of object

Dat	The set of all data values (see section 1.1.1)
datum	The specific label for data
Datvar	The set of all data variables (see section 1.2.1)
$\mathtt{Dom}(\cdot)$	The domain of a function
$\mathtt{Freevar}(\cdot)$	The free variables of a formula (see section 1.2.3)
$\mathtt{fst}(\cdot)$	The first element of a pair
$\mathtt{Heaps}_{\mathtt{sv}}$	The set of all heaps (see section 1.1.1)
$\mathtt{Heaps}_{\mathtt{s}}$	The set of all shape heaps (see section 1.1.2)
$\mathtt{Heaps}_{\mathtt{v}}$	The set of all simple heaps (see section 1.1.3)
Heaps	The set of all simple shape heaps (see section 1.1.4)
Ins	The set of all program instructions (see section 4.1.1)
$\mathtt{Im}(\cdot)$	The image of a function
Lab	The set of all labels (see section 1.1.1)
Loc	The set of all locations (see section 1.1.1)
$\mathtt{max}(\cdot)$	The maximum of a set
$\mathtt{min}(\cdot)$	The minimum of a set
$\mathbb{N}$	The set of all integers
next	The specific label for the successor
$\mathtt{Pow}(\cdot)$	The powerset of a set
$\mathtt{Pow}_{\mathtt{fin}}(\cdot)$	The finite powerset of a set
Prog	The set of all programs (see section 4.1.1)
$\mathtt{Prog}^{\mathtt{ct}}$	The set of all programs without destructive update (see section 4.1.1)
Progvar	The set of all program variables (see section 1.2.1)
Secvar	The set of all second-order variables (see section 1.2.1)
$\mathtt{Shape}(\cdot)$	The shape of a heap (see section 1.1.4)
$\mathtt{snd}(\cdot)$	The second element of a pair
Specialvar	The set of all special variables (see section 1.2.1)
Stores	The set of all stores (see section 1.1.1)
$\mathtt{sup}(\cdot)$	The supremum of a set
Var	The set of all first-order variables (see section 1.1.1)
$\lvert\cdot\rvert$	The cardinal of a set or the length of a word
$\langle\cdot,\cdot\rangle$ and $\langle\cdot\rangle$	The functions providing fresh variables (see section 1.2.1)
$[\cdot,\cdot]$	The interval between two integers
$\cdot \rightharpoonup \cdot$	The set of partial functions from one set to another
$\cdot \rightharpoonup_{\mathtt{fin}} \cdot$	The set of partial functions with finite domain
$\cdot \rightharpoonup_{\mathtt{fin+}} \cdot$	The set of partial functions with nonempty finite domain
$\star$	The disjoint union of heaps (see section 1.1.1) and a logical operator
$\square$	The wildcard symbol (see section 1.3.1)

Figure B: Names and notations of mathematical objects

DSO	The dyadic second-order logic on simple memory shapes	1.2
LTL	The linear-time temporal logic	
$\mathtt{LTL}^{\mathtt{mem}}$	The temporal logic for sequences on memory shapes we introduce	4.1
MSO	The monadic second-order logic on simple memory shapes	1.2
SL	The separation logic on simple memory shapes	1.3
$\mathtt{SL}^{*}$	SL without wand	1.3
$\mathtt{SL}^{-*}$	SL without the separating conjunction	1.3
$\mathtt{SL}^{*,-*_n}$	SL with restricted wand	2.1
$\mathtt{SL}^{<n}$	SL with restricted use of the wand	2.1
$\mathtt{SL}_{\mathtt{v}}$	The separation logic on simple memory states	1.3
$\mathtt{SL}_{\mathtt{v}}^{\mathtt{short}}$	$\mathtt{SL}_{\mathtt{v}}$ with short-distance comparisons	3.1
$\mathtt{SL}_{\mathtt{v}}^{\mathtt{R},-*_1}$	$\mathtt{SL}_{\mathtt{v}}$ with restricted wand	3.3
$\mathtt{SL}_{\mathtt{v}}^{\mathtt{guarded}}$	$\mathtt{SL}_{\mathtt{v}}$ with guarded long-distance comparisons	3.2
$\mathtt{SL}_{\mathtt{v}}^{\mathtt{long}}$	$\mathtt{SL}_{\mathtt{v}}$ with long-distance comparisons	3.2
$\mathtt{SL}_{\mathtt{v}}^{\mathtt{longeq}}$	$\mathtt{SL}_{\mathtt{v}}$ with equality long-distance comparisons	3.2
$\mathtt{SL}_{\mathtt{s}}$	The separation logic on memory shapes	1.3
$\mathtt{SL}_{\mathtt{s}}^{*}$	$\mathtt{SL}_{\mathtt{s}}$ without wand	1.3
$\mathtt{SL}_{\mathtt{s}}^{\mathtt{CL}}$	The classical fragment of $\mathtt{SL}_{\mathtt{s}}$	1.3
$\mathtt{SL}_{\mathtt{s}}^{\mathtt{RF}}$	The record fragment of $\mathtt{SL}_{\mathtt{s}}$	1.3
$\mathtt{SL}_{\mathtt{s}}^{\mathtt{LF}}$	The list fragment of $\mathtt{SL}_{\mathtt{s}}$	1.3
$\mathtt{SL}_{\mathtt{sv}}$	The separation logic on memory states	1.3
SO	The second-order logic on simple memory shapes	1.2

Figure C: Names of logical formalisms – the last column is the section containing the definition

always	The always operator
emp	The empty constant
sometimes	The sometimes operator
until	The until operator
val(·)	The value stored in a variable
$\bigcirc$	The next operator and the next symbol in expressions
$-*$	The magic wand operator
$\exists\!\!-*$	The existential magic wand operator
$-*_{\mathtt{n}}$	The restricted wand operator
$\exists\!\!-*_{\mathtt{n}}$	The existential restricted wand operator
$*$	The separating conjunction operator and the disjoint union of heaps
$\top$	The true constant
$\hookrightarrow$	The points-to predicate
$\mapsto$	The precise points-to predicate

Figure D: Main notations for logical operators

Bibliography

[1] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. *Verification of object-oriented software: the Key approach*. Springer, 2000.

[2] R. Alur and Th. Henzinger. A really temporal logic. *Journal of the Association for Computing Machinery*, 41(1):181–204, 1994.

[3] T. Antonopoulos and A. Dawar. Separating graph logic from MSO. In *FOSSACS'09*, volume 5504 of *LNCS*, pages 63–77. Springer, 2009.

[4] S. Arora and B. Barak. *Computational complexity – a modern approach*. Cambridge University Press, 2009.

[5] P. Balbiani and J. F. Condotta. Computational complexity of propositional linear temporal logics based on qualitative spatial or temporal reasoning. In *FROCOS'02*, volume 2309 of *LNAI*, pages 162–173. Springer, 2002.

[6] K. Bansal, R. Brochenin, and E. Lozes. Beyond shapes: lists with ordered data. In *FOSSACS'09*, volume 5504 of *LNCS*, pages 425–439. Springer, 2009.

[7] S. Bardin, A. Finkel, E. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. Presented at *AVIS'06*, 2006.

[8] S. Bardin, A. Finkel, and D. Nowak. Toward symbolic verification of programs handling pointers. Presented at *AVIS'04*, 2004.

[9] B. Bennett, F. Wolter, and M. Zakharyaschev. Multi-dimensional modal logic as a framework for spatio-temporal reasoning. *Applied Intelligence*, 17(3):239–251, 2002.

[10] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV'07*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.

[11] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *FST&TCS'04*, volume 3328 of *LNCS*, pages 97–109. Springer, 2004.

[12] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: modular automatic assertion checking with separation logic. In *FMCO'05*, volume 4111 of *LNCS*, pages 115–137. Springer, 2005.

[13] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS'05*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.

[14] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The essence of computation: complexity, analysis, transformation.*, volume 2566 of *LNCS*, pages 85–108. Springer, 2002.

[15] M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS'06*, pages 7–16. IEEE, 2006.

[16] E. Börger, E. Grädel, and Y. Gurevich. *The classical decision problem*. Perspectives in Mathematical Logic. Springer, 1997.

[17] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV'06*, volume 4144 of *LNCS*, pages 517–531. Springer, 2006.

[18] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In *CONCUR'09*, volume 5710 of *LNCS*, pages 178–195. Springer, 2009.

[19] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI'11*, pages 578–589. ACM, 2011.

[20] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *ATVA'12*, volume 7561 of *LNCS*, pages 167–182. Springer, 2012.

[21] A. Bouajjani, R. Echahed, and P. Habermehl. On the verification problem of nonregular properties for nonregular processes. In *LICS'95*, pages 123–133. IEEE, 1995.

[22] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structured in regular model-checking. In *TACAS'05*, volume 3440 of *LNCS*, pages 13–29. Springer, 2005.

[23] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer*, 14(2):167–191, 2012.

[24] M. Bozga, R. Iosif, and Y. Lakhnech. On logics of aliasing. In *SAS'04*, volume 3148 of *LNCS*, pages 344–360. Springer, 2004.

[25] M. Bozga, R. Iosif, and S. Perarnau. Quantitative separation logic and programs with lists. In *IJCAR'08*, volume 5195 of *LNCS*, pages 34–49. Springer, 2008.

[26] R. Brochenin, S. Demri, and E. Lozes. Reasoning about sequences of memory states. In *LFCS'07*, volume 4514 of *LNCS*, pages 100–114. Springer, 2007.

[27] R. Brochenin, S. Demri, and E. Lozes. On the almighty wand. In *CSL'08*, volume 5213 of *LNCS*, pages 322–337. Springer, 2008.

[28] R. Brochenin, S. Demri, and E. Lozes. Reasoning about sequences of memory states. *Annals of Pure and Applied Logic*, 161(3):305–323, 2009.

[29] R. Brochenin, S. Demri, and E. Lozes. On the almighty wand. *Information & Computation*, 211:106–137, 2012.

[30] J. Brotherston and M. I. Kanovich. Undecidability of propositional separation logic and its neighbours. In *LICS'10*, pages 130–139. IEEE, 2010.

[31] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford, 1960.

[32] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification of infinite structures. In *Handbook of Process Algebra*, pages 545–623. Elsevier, 2001.

[33] C. Calcagno, P. Gardner, and M. Hague. From separation logic to first-order logic. In *FOSSACS'05*, volume 3441 of *LNCS*, pages 395–409. Springer, 2005.

[34] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In *POPL'07*, pages 123–134. ACM, 2007.

[35] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language. In *APLAS'01*, volume 3302 of *LNCS*, pages 289–300. Springer, 2001.

[36] C. Calcagno, H. Yang, and P. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FST&TCS'01*, volume 2245 of *LNCS*, pages 108–119. Springer, 2001.

[37] H. Comon and V. Cortier. Flatness is not a weakness. In *CSL'00*, volume 1862 of *LNCS*, pages 262–276. Springer, 2000.

[38] B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR'11*, volume 6901 of *LNCS*, pages 235–249. Springer, 2011.

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2009.

[40] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.

[41] A. Dawar, P. Gardner, and G. Ghelli. Adjunct elimination through games in static ambient logic. In *FST&TCS'04*, volume 3328 of *LNCS*, pages 211–223. Springer, 2004.

[42] A. Dawar, P. Gardner, and G. Ghelli. Expressiveness and complexity of graph logic. *Information & Computation*, 205(3):263–310, 2007.

[43] S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *Information & Computation*, 205(3):380–415, 2007.

[44] S. Demri and R. Gascon. The effects of bounding syntactic resources on Presburger LTL. *Journal of Logic and Computation*, 19(6):1541–1575, 2009.

[45] S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*, 10(3):16:1–16:30, 2009.

[46] S. Demri, R. Lazić, and D. Nowak. On the freeze quantifier in constraint LTL: decidability and complexity. In *TIME'05*, pages 113–121. IEEE, 2005.

[47] D. Distefano, J. P. Katoen, and A. Rensink. Who is pointing when to whom? on the automated verification of linked list structures. In *FST&TCS'04*, volume 3328 of *LNCS*, pages 250–262. Springer, 2004.

[48] M. Dodds and D. Plump. From separation logic to hyperedge replacement and back. In *ICGT'08*, volume 5214 of *LNCS*, pages 484–486. Springer, 2008.

[49] K. Etessami, M. Vardi, and T. Wilke. First-order logic with two variables and unary temporal logic. *Information & Computation*, 179(2):279–295, 2002.

[50] J. C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV'07*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[51] A. Finkel, E. Lozes, and A. Sangnier. Towards model-checking programs with lists. In *ILC'07*, volume 5489 of *LNCS*, pages 56–86. Springer, 2009.

[52] D. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev. *Many-dimensional modal logics: theory and applications*. CUP, 2003.

[53] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *POPL'80*, pages 163–173. ACM, 1980.

[54] D. Galmiche and D. Méry. Characterizing provability in BI's pointer logic through resource graphs. In *LPAR'05*, volume 3835 of *LNCS*, pages 459–473. Springer, 2005.

[55] D. Galmiche and D. Méry. Tableaux and resource graphs for separation logic. *Journal of Logic and Computation*, 20(1):189–231, 2010.

[56] N. Gorogiannis, M. I. Kanovich, and P. W. O'Hearn. The complexity of abduction for separated heap abstractions. In *SAS'11*, volume 6887 of *LNCS*, pages 25–42. Springer, 2011.

[57] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. *ACM SIGPLAN Notices*, 43(1):235–246, 2008.

[58] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[59] C. A. R. Hoare, A. Hussain, B. Möller, P. W. O'Hearn, R. L. Petersen, and G. Struth. On locality and the exchange law for concurrent processes. In *CONCUR'11*, volume 6901 of *LNCS*, pages 250–264. Springer, 2011.

[60] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, 1991.

[61] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL'01*, pages 14–26. ACM, 2001.

[62] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *CONCUR'96*, volume 1119 of *LNCS*, pages 263–277. Springer, 1996.

[63] J. Jensen, M. Jorgensen, N. Klarlund, and M. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI'97*, pages 226–236. ACM, 1997.

[64] J. Kamp. *Tense logic and the theory of linear order*. PhD thesis, UCLA, USA, 1968.

[65] J. P. Katoen, T. Noll, and S. Rieger. Verifying concurrent list-manipulating programs by LTL model-checking. Technical report, RWTH Aachen University, Germany, 2007.

[66] F. Klaedtke and H. Rueb. Monadic second-order logics with cardinalities. In *ICALP'03*, volume 2719 of *LNCS*, pages 681–696. Springer, 2003.

[67] V. Kuncak and M. Rinard. On spatial conjunction as second-order logic. Technical report, MIT CSAIL, USA, 2004.

[68] D. Larchey-Wendling and D. Galmiche. The undecidability of boolean BI through phase semantics. In *LICS'10*, pages 140–149. IEEE, 2010.

[69] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS'00*, volume 1824 of *LNCS*, pages 280–301. Springer, 2000.

[70] A. Loginov, T. Reps, and M. Sagiv. Refinement-based verification for possibly-cyclic lists. In *Program analysis and compilation, theory and practice*. Springer, 2007.

[71] E. Lozes. *Expressivité des logiques spatiales*. PhD thesis, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, France, 2004.

[72] E. Lozes. Separation logic preserves the expressive power of classical logic. Presented at *SPACE'04*, 2004.

[73] E. Lozes. Elimination of spatial connectives in static spatial logics. *Theoretical Computer Science*, 330(3):475–499, 2005.

[74] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *POPL'11*, pages 611–622. ACM, 2011.

[75] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS'07*, volume 4634 of *LNCS*, pages 419–436. Springer, 2007.

[76] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV'05*, pages 476–490. Springer, 2005.

[77] H. H. Nguyen, C. David, S. Qin, and W. N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI'07*, volume 4349 of *LNCS*, pages 251–266. Springer, 2007.

[78] P. Odifreddi. *Classical recursion theory*. Studies in Logic. Elsevier, 1989.

[79] P.W. O'Hearn and D. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[80] A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–57. IEEE, 1977.

[81] D. Pym. *The semantics and proof theory of the logic of bunched implications*. Applied Logic Series. Kluwer academic publishers, 2002.

[82] M. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 41:1–35, 1969.

[83] S. Ranise and C. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *SEFM'06*, pages 206–215. IEEE, 2006.

[84] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE, 2002.

[85] X. Rival and B. Y. E. Chang. Calling context abstraction with shapes. In *POPL'11*, pages 173–186. ACM, 2011.

[86] A. Sistla and E. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 32(3):733–749, 1985.

[87] L. Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, Department of Electrical Engineering, MIT, USA, 1974.

[88] B. A. Trakhtenbrot. The impossibility of an algorithm for the decision problem for finite models. *Doklady Akademii Nauk SSSR*, 70:596–572, 1950. English translation in: AMS Transl. Ser. 2, vol.23 (1063), 1–6.

[89] M. Vardi and P. Wolper. Reasoning about infinite computations. *Information & Computation*, 115:1–37, 1994.

[90] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP'03*, volume 2618 of *LNCS*, pages 204–22. Springer, 2003.

[91] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV'08*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.

[92] G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data structures. In *FOSSACS'05*, volume 3441 of *LNCS*, pages 94–110. Springer, 2005.