Technical University of Denmark

A Succinct Approach to Static Analysis and Model Checking

Filipiuk, Piotr; Nielson, Hanne Riis; Nielson, Flemming

Publication date:
2012

Document Version
Publisher's PDF, also known as Version of record

A Succinct Approach to Static Analysis and Model Checking

Piotr Filipiuk

Kongens Lyngby 2012
IMM-PHD-2012-278

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN

Summary

In a number of areas software correctness is crucial, therefore it is often desirable to formally verify the presence of various properties or the absence of errors. This thesis presents a framework for concisely expressing static analysis and model checking problems. The framework facilitates rapid prototyping of new analyses and consists of variants of ALFP logic and associated solvers.

First, we present a Lattice based Least Fixed Point Logic (LLFP) that allows interpretations over complete lattices satisfying Ascending Chain Condition. We establish a Moore Family result for LLFP that guarantees that there always is single best solution for a problem under consideration. We also develop a solving algorithm, based on a differential worklist, that computes the least solution guaranteed by the Moore Family result.

Furthermore, we present a logic for specifying analysis problems called Layered Fixed Point Logic. Its most prominent feature is the direct support for both inductive computations of behaviors as well as co-inductive specifications of properties. Two main theoretical contributions are a Moore Family result and a parametrized worst-case time complexity result. We develop a BDD-based solving algorithm, which computes the least solution guaranteed by the Moore Family result with worst-case time complexity as given by the complexity result.

We also present magic set transformation for ALFP, known from deductive databases, which is a clause-rewriting strategy for optimizing query evaluation. In order to compute the answer to a query, the original ALFP clauses are rewritten at compile time, and then the rewritten clauses are evaluated bottom-up. It is usually more efficiently than computing entire solution followed by selection

of the tuples of interest, which was the case in the classical formulation of ALFP logic.

Finally, we show that the logics and the associated solvers can be used for rapid prototyping. We illustrate that by a variety of case studies from static analysis and model checking.

Resumé

Inden for mange områder er det essentielt at softwaren er korrekt. Inden for datalogien er der udviklet en række formelle verifikation teknikker, herunder statisk analyse og model tjek, som gør det muligt at analysere softwaren og sikre at den har forskellige egenskaber. I denne afhandling præsenteres en ramme inden for hvilken man hurtigt og elegant kan specificere specielt statisk analyse og model tjek problemer i logisk form. Denne ramme understøttes af en række generiske værktøjer, som gør at man givet en egenskab automatisk kan konstruere et system som kan analysere softwaren for den pågældende egenskab.

I den første del af afhandlingen præsenteres en gitter-baseret logik kaldet Least Fixed Point Logic (LLFP). Dens semantiske fundament er en matematiske struktur af fuldstændige gitre som tilfredsstiller den såkaldte Ascending chain condition. Vi viser at LLFP har en Moore familie egenskab; det betyder at ethvert problem udtrykt i LLFP altid har præcist en løsning som er bedre en alle andre løsninger på problemet. Vi udvikler derefter en implementation som beregner denne løsning; den er baseret på den såkaldte differential worklist tilgangsvinkel.

Den næste logik der præsenteres i afhandlingen er Layered Fixed Point Logic. Denne logik adskiller sig fra den forrige ved at den direkte understøtter induktive såvel som co-induktive specifikationer af problemer. Også for denne logik viser vi en Moore familie egenskab; ydermere etablerer vi et worst-case tids kompleksitets resultat. Denne gang udvikler vi en implementation baseret på BDD tilgangsvinklen; implementationen beregner den bedste løsning på problemet, som angivet af Moore familie resultatet, og har en køretid svarende til det teoretiske kompleksitets resultat.

Efterfølgende studerer vi en optimeringsstrategi, kaldet magic sets transforma-

tioner, fra deduktive databaser og dens anvendelse på logikken ALFP. Ideen er at omskrive den oprindelige formulering af egenskaben til en form som muliggør en mere effektiv beregning; specielt er det ikke nødvendigt at beregne hele løsningen hvis der kun er brug for en mindre del af den.

I den sidste den af afhandlingen illustrerer vi hvordan logikkerne og de tilhørende implementationer kan bruges til hurtig konstruktion af prototyper. Specielt ser vi på forskellige eksempler fra statisk analyse og model tjek.

Preface

This thesis was prepared at DTU Informatics at Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in Computer Science.

The Ph.D. study has been carried out under the supervision of Professor Hanne Riis Nielson and Professor Flemming Nielson in the period of August 2009 to July 2012.

Most of the work behind this dissertation has been carried out independently and I take full responsibility for its contents. Part of the work presented is based on published articles co-authored with my supervisors. In particular, LLFP logic introduced in Chapter 3 was published in [27]. The LFP logic presented in Chapter 4 was accepted for publication in [28], whereas the work on solving algorithms for ALFP logic described in Chapter 5 was published in [29]. The implementation of the solving algorithms from Chapter 5 was released under an open-source license and is available at https://github.com/piotrfilipiuk/succinct-solvers.

Kongens Lyngby, July 2012

Piotr Filipiuk

Acknowledgements

First, I would like to thank my family for their love, support, and encouragement.

I also thank my supervisors Professor Hanne Riis Nielson and Professor Flemming Nielson for their guidance and support. They offered insightful comments and constructive feedback during the entire course of my PhD studies.

I would also like to thank Professor Alan Mycroft for many useful discussions and invaluable feedback during my stay at the University of Cambridge.

I am grateful to Michał Terepeta for proof-reading this dissertation.

I would also like to thank the rest of the LBT section: Jose Nuno Carvalho Quaresma, Han Gao, Alejandro Mario Hernandez, Sebastian Alexander Mödersheim, Henrik Pilegaard, Christian W. Probst, Matthieu Stéphane Benoit Queva, Carroline Dewi Puspa Kencana Ramli, Nataliya Skrypnyuk, Roberto Vigo, Fan Yang, Ender Yeksel, Kebin Zeng, Fuyuan Zhang, Lijun Zhang, for creating an inspiring working environment.

Contents

CHAPTER 1

Introduction

1.1 System analysis and verification

Nowadays, we rely more and more on software systems. At the same time the systems become bigger and more complex. They are present in almost every aspect of our daily life via e.g. online banking, shopping and embedded systems such as cameras or mobile phones. Due to our extensive use of software systems, it is very important that they are reliable, offer good performance and are free of errors. We are sure that many Windows OS users were annoyed by the 'Blue Screen' that is displayed when the system encountered an unrecoverable error. Game players may also be familiar with a software bug in the Pac-Man game, which was caused by integer overflow [1] and made further play impossible. There are well known examples of errors that had damaging financial consequences. The most prominent is probably the bug in the control software of the Ariane-5 missile, which crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value. Another critical error, which caused the death of six people due to the exposure to an overdose of radiation, was the bug in the control part of the radiation therapy machine Therac-25.

As systems grow in size and complexity, the number of potential errors increases. At the same time market pressure and demand on software systems, make the task of delivering reliable and fast software hard and challenging. In order to achieve more reliable systems, peer reviews and testing may be applied. A peer review is a manual inspection of the source code by a software engineer, who preferably was not the author of the part of the system being reviewed. The main drawback of the method is that subtle errors such as corner cases or concurrency problems are very hard to detect. Another technique commonly used in practice is software testing. In contrast to peer reviews, which is a static method and does not execute the software, testing is a dynamic technique that runs the software. During software testing the software being tested is executed for given inputs, and the actual output is then compared against the expected one. The main problem of software testing is its incompleteness, due to the fact that exhaustive testing that covers all execution paths is infeasible in practice. It is well summarized by Edsger W. Dijkstra: 'Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence' [24].

A much stronger approach for ensuring reliability and correctness of systems is verification. Its aim is to prove that a system under consideration possess certain properties such as deadlock freedom or lack of memory leaks. In order to verify a system, the specification (model) is needed, along with the property (or properties) that are to be checked. The system is considered correct with respect to some properties, if it satisfies all of them. Consequently, this understanding of correctness is relative to the specification and properties.

This dissertation deals with two formal verification techniques called static analysis [41, 2] and model checking [4, 31]. Both static analysis and model checking apply mathematics to model, analyse and verify systems. Research in formal methods led to the development of promising techniques, which in turn led to an increasing use of formal verification in practice. There are many powerful verification tools that could have detected the errors in, e.g., the Ariane-5 missile, Intel's Pentium II processor, and the Therac-25 therapy radiation machine.

Model checking is an automated verification technique that systematically explores all possible system behaviors. Thanks to the exhaustive exploration, one is sure that the system satisfies certain properties. However, since model checking verifies the model, not the actual system, the technique is only as good as the model of the system.

State-of-the-art model checkers are able to analyse real systems whose corresponding models have $10^9 - 10^{12}$ states. Using specialized algorithms and data structures suited for a specific problem, even larger state spaces (10^{20} and beyond) can be handled [13]. Importantly, they have found errors that were undetected using peer reviews and testing.

Examples of properties that can be verified using model checking vary from qualitative properties such as: "Can the system deadlock?" to more sophisticated quantitative properties such as: "Is the response delivered within 0.1 seconds?".

The model of the system under consideration is usually automatically extracted from the description of the system, which commonly is a programming language. The properties to be checked need to be precise and unambiguous, and are usually expressed as logical formulae using modal logic. As already mentioned the model checker explores all possible system behaviors and checks whether the properties of interest are satisfied. In the case a state violating some property is encountered, the model checker produces a counterexample that shows how the state can be reached. More precisely, the counterexample represents a path from the initial state to the state violating the property.

Static analysis is a technique for reasoning about system behavior without executing it. It is performed statically at compile-time, and it computes safe approximations of values or behaviors that may occur at run-time. Static Analysis is recognized as a fundamental technique for program verification, bug detection, compiler optimization and software understanding.

Static analysis bug-finding tools have evolved over the last several decades from basic syntactic checkers to those that find complex errors by reasoning about the semantics of code. They help developers find hard-to-spot, yet potentially crash-causing defects early in the software development life cycle, reducing the cost, time, and risk of software errors. State-of-the-art static analysis engines are able to identify critical bugs and they scale to millions of lines of code. They also provide low rate of false positives, which makes them extremely useful.

Another important application of static analysis is compiler optimization [2]. Compiler optimization applies static program analysis techniques and aims at producing the output so that e.g. the execution time of the program or the memory used are minimized. It is usually accomplished using a sequence of transformation passes - algorithms which take a representation of the program and transform it to produce a semantically equivalent output that uses fewer resources.

There are many different analyses embedded in compiler optimization frameworks, results of which are used to perform these optimizations. Probably the most common technique is data-flow analysis, which computes information about the possible set of values at various points in the analysed program. Classical data flow analyses include reaching definition, live variable and available expression analyses. Other important analyses performed by compilers are pointer analysis, which computes which pointers may point to which variables or heap locations, as well as array bound analysis that determines whether an array index is always within the bounds of the array.

It is known that some optimization problems are NP-complete, or even undecidable. The optimizers are also limited by the time and memory requirements; hence the optimization rarely produces “optimal” output in any sense. In order for static analysis results to be computable, the technique can usually only provide approximate answers. Hence, program analysis usually provides a possibly larger set of values or behaviors than what would be feasible during execution of the system. The great challenge is not to produce too many spurious results, since then the analysis will become useless.

1.2 Contributions

This dissertation deals with two verification techniques namely static analysis and model checking, and presents a framework for concisely expressing problems from both areas. The framework facilitates rapid prototyping and consists of variants of ALFP logic [44] and associated solvers. In particular it consists of the following logics and solving algorithms

- Alternation-free Least Fixed Point Logic (ALFP) developed by Nielson et al. [44], which has successfully been used as the constraint language for sophisticated analyses of many programming paradigms including imperative, functional, concurrent and mobile languages and more recently for model checking [10, 42]. Our contribution is the development of a BDD-based solving algorithm for ALFP, which computes the least model of a given ALFP formula.
- Lattice based Least Fixed Point Logic (LLFP) that allows interpretations over complete lattices satisfying Ascending Chain Condition. We establish a Moore Family result for LLFP that guarantees that there always is single best solution for a problem under consideration. We also develop a solving algorithm, based on differential worklist, that computes the least solution guaranteed by the Moore Family result.
- Layered Fixed Logic (LFP), which has direct support for both inductive computations of behaviors as well as co-inductive specifications of properties. Two main theoretical contributions are a Moore Family result and a parametrized worst-case time complexity result. We also develop a BDD-based solving algorithm, which computes the least solution guaranteed by the Moore Family result with worst-case time complexity as given by the complexity result.

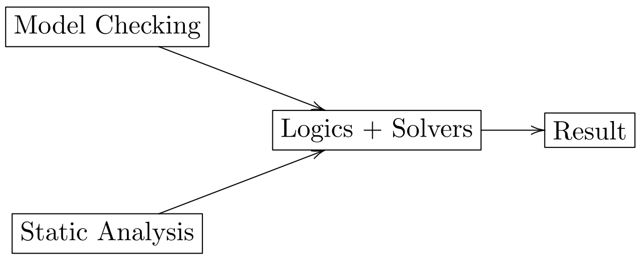


Figure 1.1: Succinct Analysis Framework in a nutshell.

The defining feature of this framework is the use of logic for specifying the analysis problems, which has many benefits. We believe that logical (declarative) specifications of analysis problems are superior to their imperative counterparts. This of mostly because they are clearer and simpler to analyse for complexity and correctness than imperative ones. Furthermore, they give a clear distinction between specification of the analysis, and the computation of the best analysis result. The applicability of the framework is illustrated by presenting case studies from static analysis and model checking. Due to the fact that problems from both areas can be succinctly expressed within one framework, we believe that this dissertation enhanced our understanding of the interplay between static analysis and model checking — to the extent that they can be seen as essentially equivalent to each other.

The main thesis of this dissertation is to show that:

Variants of ALFP logic and their associated solvers can be used for efficient rapid prototyping and have a wide variety of applications within static analysis and model checking.

The overview of the approach presented in this dissertation is depicted in Figure 1.1. The main idea is to have a unified framework that can handle both static analysis and model checking problems in a succinct manner. Our approach consists of two steps. In the first one, we transform the analysis problem into a logical formula expressed in some variant of ALFP logic. In the second step, a corresponding solver is used to obtain the analysis result.

1.3 Related Work

The use of logic for specifying static analysis problems intrigued many researchers and resulted in an immense body of work. Dawson, Ramakrishnan, and Warren [22] showed how some program analyses can be cast in the form of evaluating minimal models of logic programs. In their case study they used formulations of groundness and strictness analyses, and they used the XSB system as a solving engine. Their results suggested that practical analysers can be build using general purpose logic programming systems. They also argued that logic programming is expressive enough to formulate many common analyses.

It was also demonstrated by Reps [49] that many data flow analyses may be formulated as graph reachability. Based on the correspondence between context-free languages and declarative programs that recognize them, his approach implies existence of declarative specifications of these analyses. The paper presented the application of the approach to interprocedural dataflow analysis, interprocedural program slicing and shape analysis.

There is also an immense amount of work on pointer analysis using logic programming; hence we restrict our discussion to a few representatives. Whaley et al. [59, 58, 34] developed an implementation of datalog based on BDDs, called bddbddb. Thanks to the use of BDDs, they were able to exploit redundancy in the analysis relations in order to solve large problems efficiently.

PADDLE framework [36] is a highly flexible framework for context-sensitive analyses. It is also based on BDDs and represents the state of the art in context sensitive pointer analyses, in terms of both semantic completeness (language features support) and scalability.

Finally, DOOP [11] raised the bar for precise context sensitive analyses. It is a purely declarative points-to analysis framework and achieves remarkable performance due to a novel optimization methodology of Datalog programs. Unlike two previous frameworks, DOOP uses an explicit representation of relations and thus it enhances our understanding on how to implement efficient points-to analyses.

There is also interesting work on formalizing model checking using logic programming. Ramakrishna et al. [47] presented an implementation of a model checker called XMC using logic programming system XSB. In their system, a CCS-like language is used to describe the model of the system under consideration, whereas properties are expressed in the alternation free fragment of the modal μ-calculus. The results presented in [46] indicate that XMC, although implemented in a general purpose logic programming system, can compete with the state-of-the-art model checkers.

An alternative approach to model checking using logic programming is described by Charatonik and Podelski [15], where they demonstrated verification technique for infinite-state systems. Their approach uses set-based analysis to compute approximations of CTL formulae. Furthermore, Delzanno and Podelski [23] explored formulation of safety and liveness properties in terms of logic programs. Their approach uses constraint logic programming to encode both the transition system and the properties to be checked. Using their approach, they were able to verify well-known examples of infinite-state programs over integers.

Another line of related work is concerned with the interplay between static analysis and model checking. On one hand we have a developments by Steffen and Schmidt that showed that static analysis is model checking of formulae in some modal logic. In [52] they used abstract interpretation to generate program traces, and modal μ-calculus to specify trace properties. In particular they presented formulation of data flow equations for bit-vector frameworks as modal μ-calculus formulae. In [53] they presented how abstract interpretation, flow analysis and model checking intersect and support each other. The methodology they presented consists of three stages. First a model, in a form of a state-transition system, is constructed from the operational semantics and a program of interest. Then the program model is abstracted by reducing the amount of information in the model's states and edges. Finally, the model is verified against properties of its states and paths using a variant of Computation Tree Logic (CTL). On the other hand there is work by Nielson and Nielson [42] showing that model checking amounts to a static analysis of the modal formulae. They used Alternation-free Least Fixed Point Logic (ALFP) to encode modal formulae expressed in Action Computation Tree Logic.

CHAPTER 2

Preliminaries

This chapter presents necessary background and notation used throughout the dissertation. The chapter is organised as follows. In Section 2.1 we summarize some properties of the partially ordered sets that play a crucial role in the developments of the further chapters. Section 2.2 presents various ways of modelling systems such as transition systems, control flow graphs and program graphs. We present an Alternation free Least Fixed Point Logic (ALFP) in Section 2.3, which constitutes a starting point for the formalisms developed in this dissertation.

2.1 Partially Ordered Sets

Since partially ordered sets and complete lattices play a crucial role in this dissertation, we summarize some of their properties. We present simple techniques for constructing complete lattices from other complete lattices, and state the definitions for Ascending and Descending Chain Conditions. We begin with a definition of a partial order.

Definition 2.1 A partial ordering $\sqsubseteq$ is a binary relation on L that is:

- reflexive, i.e. $\forall l : l \sqsubseteq l$,

- transitive, i.e. $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$,
- anti-symmetric, i.e. $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$.

The above definition gives rise to a structured sets, whose elements are related to each other according to a partial order relation.

Definition 2.2 A partially ordered set (poset) $(L, \sqsubseteq)$ is a set L equipped with a partial ordering $\sqsubseteq$.

The following two definitions introduce notions of *least upper bounds* as well as *greatest lower bounds*.

Definition 2.3 An element $l \in L$ is an upper bound of a subset $Y \subseteq L$ if $\forall l' \in Y : l' \sqsubseteq l$. A least upper bound l of Y is an upper bound of Y that satisfies $l \sqsubseteq l_0$ whenever l_0 is another upper bound of Y.

Definition 2.4 An element $l \in L$ is a lower bound of a subset $Y \subseteq L$ if $\forall l' \in Y : l \sqsubseteq l'$. A greatest lower bound l of Y is a lower bound of Y that satisfies $l_0 \sqsubseteq l$ whenever l_0 is another lower bound of Y.

Note that a subset Y of a poset does not necessary have least upper bounds nor greatest lower bounds, however if they exist they are unique and are denoted $\bigsqcup Y$ and $\bigsqcap Y$, respectively. Alternatively, $\bigsqcup$ is called *meet*, whereas $\bigsqcap$ is called *join*.

Definition 2.5 A complete lattice $L = (L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is a partially ordered set $(L, \sqsubseteq)$ such that all subsets have least upper bounds and greatest lower bounds. Moreover, $\bot = \bigsqcup \emptyset = \bigsqcap L$ is the least element and $\top = \bigsqcap \emptyset = \bigsqcup L$ is the greatest element.

Definition 2.6 A Moore family is a subset Y of a complete lattice $L = (L, \sqsubseteq)$ that is closed under greatest lower bounds: $\forall Y' \subseteq Y : \bigsqcap Y' \in Y$.

It follows that a Moore family always contains a least element, $\bigsqcap Y$, and a greatest element, $\bigsqcap \emptyset$, which equals the greatest element, $\top$, from L; in particular, a Moore family is never empty. The property is also called the model intersection property, since whenever we take a *meet* of a number of models we still get a model.

New complete lattices may be created by combining the existing ones. We review three methods for construction of new complete lattices based on cartesian products, as well as total and monotone function spaces.

Let $L_1 = (L_1, \sqsubseteq_1)$ and $L_2 = (L_2, \sqsubseteq_2)$ be two partially ordered sets. Then $L = (L, \sqsubseteq)$ defined by

$$L = L_1 \times L_2$$

and

$$(l_1, l_2) \sqsubseteq (l'_1, l'_2) \Leftrightarrow l_1 \sqsubseteq_1 l'_1 \wedge l_2 \sqsubseteq_2 l'_2$$

is also a partially ordered set. Furthermore, if each $L_i = (L_i, \sqsubseteq_i) = (L_i, \sqsubseteq_i, \bigsqcup_i, \bigsqcap_i, \perp_i, \top_i)$, $i \in \{1, 2\}$, is a complete lattice, then so is $L = (L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \perp, \top)$. The least upper bound of the lattice is as follows

$$\bigsqcup Y = \left(\bigsqcup_1 \{l_1 \mid \exists l_2 : (l_1, l_2) \in Y\}, \bigsqcup_2 \{l_2 \mid \exists l_1 : (l_1, l_2) \in Y\} \right)$$

the bottom element $\perp$ is given by $\perp = (\perp_1, \perp_2)$. All other components are defined analogously.

Now, we present construction of complete lattices based on a total function space. Let $L_1 = (L_1, \sqsubseteq_1)$ be a partially ordered set and let S be a set. Then $L = (L, \sqsubseteq)$ defined by

$$L = \{f : S \to L_1 \mid f \text{ is total}\}$$

and

$$f \sqsubseteq f' \Leftrightarrow \forall s \in S : f(s) \sqsubseteq_1 f'(s)$$

is also a partially ordered set. Furthermore, if $L_1 = (L_1, \sqsubseteq_1) = (L_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1, \perp_1, \top_1)$ is a complete lattice, then so is $L = (L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \perp, \top)$. The components of the lattice are as follows

$$\bigsqcup Y = \lambda s. \bigsqcup_1 \{f(s) \mid f \in Y\}$$

the bottom element $\perp$ is given by $\perp = \lambda s. \perp_1$. All other components are defined analogously.

The construction of complete lattices based on monotone function space is as follows. Let $L_1 = (L_1, \sqsubseteq_1)$ and $L_2 = (L_2, \sqsubseteq_2)$ be two partially ordered sets. Then $L = (L, \sqsubseteq)$ defined by

$$L = \{f : L_1 \to L_2 \mid f \text{ is monotone}\}$$

and

$$f \sqsubseteq f' \Leftrightarrow \forall l_1 \in L_1 : f(l_1) \sqsubseteq_2 f'(l_1)$$

is also a partially ordered set. Furthermore, if each $L_i = (L_i, \sqsubseteq_i) = (L_i, \sqsubseteq_i, \bigsqcup_i, \bigsqcap_i, \perp_i, \top_i)$, $i \in \{1, 2\}$, is a complete lattice, then so is $L = (L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcup, \bigsqcap, \perp, \top)$. The components of the lattice are as follows

$$\bigsqcup Y = \lambda l_1. \bigsqcup_2 \{f(l_1) \mid f \in Y\}$$

the bottom element $\perp$ is given by $\perp = \lambda l_1.\perp_2$. All other components are defined analogously.

It is common for static analysis algorithms to iteratively compute analysis information, which usually are an elements of a complete lattice. In each iteration the algorithm obtains better information, hence the information computed in each iteration essentially forms a sequence of lattice elements. Now we define these sequences, called chains, and state some of their properties.

Definition 2.7 A subset $Y \subseteq L$ of a partially ordered set $L = (L, \sqsubseteq)$ is a chain if $\forall l_1, l_2 \in Y : (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$.

It follows that a chain is a (possibly empty) totally ordered subset of a partially ordered set.

A sequence $(l_n)_n = (l_n)_{n \in \mathbb{N}}$ of elements in L is an ascending chain if

$$n \leq m \Rightarrow l_n \sqsubseteq l_m$$

Similarly, a sequence $(l_n)_n$ is a descending chain if

$$n \leq m \Rightarrow l_m \sqsubseteq l_n$$

Clearly ascending and descending chains are also chains.

We say that a sequence $(l_n)_n$ eventually stabilise if and only if

$$\exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow l_n = l_{n_0}$$

A partially ordered set has finite height if and only if all chains are finite. A partially ordered set satisfies the Ascending Chain Condition if and only if all ascending chains eventually stabilise. Analogously, it satisfies the Descending Chain Condition if and only if all descending chains eventually stabilise. These give rise to the following Lemma.

Lemma 2.8 *A partially ordered set $L = (L, \sqsubseteq)$ has finite height if and only if it satisfies both the Ascending and Descending Chain Conditions.*

Proof. See Appendix A.3 in [41].

2.2 Modelling Systems

A prerequisite for the analysis is a model of the system under consideration. In this section we present such models. We first present transition systems that are a standard way of representing hardware and software systems in model checking. Then we introduce control flow graphs, which are usually used by compilers to represent programs. Finally, we present *program graphs*, in which actions label the edges rather than the nodes. The main benefit of using program graphs is that we can model concurrent systems in a straightforward manner. Moreover since a model of a concurrent system is also a program graph, all the results are applicable both in the sequential as well as in the concurrent setting.

2.2.1 Transition Systems

Transition systems are used to model behavior of systems. They are basically directed graphs, where nodes represent *states* of the system, whereas edges represent *transitions* i.e. changes of states. More precisely, a state describes certain information about current state of the system, and transitions state how the system may go from one state to another.

There are many different types of transition systems. The one we present here uses named transitions, and labels each state with a set of atomic propositions. The transition names can be used to denote the kind of action, or in the case of concurrent systems may be used for communication. The atomic propositions describe some basic facts about the given state. The formal definition of the transition system is given by Definition 2.9.

Definition 2.9 A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- S is a set of states,
- Act is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

A transition system TS is called finite if S, Act, and AP are finite.

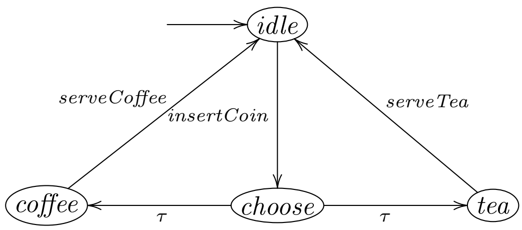


Figure 2.1: Transition system representing a vending machine.

As an example of a transition system, let us consider a simplified model of a vending machine depicted in Figure 2.1. The machine first accepts a coin and then nondeterministically serves either coffee or tea. The state space is $S = \{idle, choose, coffee, tea\}$. The set of initial states is $I = \{idle\}$, which in the figure is indicated by having an incoming arrow without a source. The set of actions is $Act = \{insertCoin, serveCoffee, serveTea, \tau\}$. The action *insertCoin* corresponds to the user of the machine inserting the coin. The action τ is an internal action performed by the machine that is not visible to the environment, which in this case is the user of the machine. The actions *serveCoffee* and *serveTea* represent delivery of coffee and tea, respectively. Let the set of atomic propositions be $AP = \{paid, deliver\}$ with the labeling function given by

$$L(idle) = \emptyset, L(choose) = \{paid\}, L(coffee) = L(tea) = \{deliver\}$$

Now we are able to reason about the behavior of the vending machine by expressing properties such as "The machine never delivers a beverage without inserting a coin."

In order to formally express the behavior of a transition system, we introduce a notion of an *execution* or a *run*. The execution represents a possible behavior of the transition system by resolving the nondeterminism in the transition system.

Definition 2.10 Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *finite* execution fragment ϱ of TS is an alternating sequence of states and actions ending with a state

$$\varrho = s_0\alpha_0 s_1\alpha_1 \ldots \alpha_{n-1}s_n \text{ such that } s_i \xrightarrow{\alpha_i} s_{i+1} \text{ for all } 0 \leq i < n,$$

where $n \geq 0$. We refer to n as the length of the execution fragment ϱ. An *infinite* execution fragment ρ of TS is an infinite alternating sequence of states and actions

$$\rho = s_0\alpha_0 s_1\alpha_1 s_2\alpha_2 \ldots \text{ such that } s_i \xrightarrow{\alpha_i} s_{i+1} \text{ for all } 0 \leq i.$$

The sequence s, where $s \in S$, is a valid execution fragment of length $n = 0$. The j-th state of $\rho = s_0\alpha_0 s_1\alpha_1 s_2\alpha_2\ldots$ is denoted by $\rho_S[j] = s_j$, whereas the j-th action is denoted by $\rho_{Act}[j] = \alpha_j$, where $j \geq 0$. The notion is defined analogously for finite execution fragments. Sometimes we write $\varrho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n$ and $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$ for $\varrho = s_0\alpha_0 s_1\alpha_1\ldots\alpha_{n-1}s_n$ and $\rho = s_0\alpha_0 s_1\alpha_1 s_2\alpha_2\ldots$, respectively.

An execution fragment is called maximal if it cannot be prolonged. Formally, it is defined by the following definition.

Definition 2.11 A *maximal* execution fragment is either a finite execution fragment that ends in the terminal state, or an infinite execution fragment.

Hence according to the above definition the execution fragment is maximal if it cannot be prolonged i.e. either it is infinite, or it is finite and ends in a state having no outgoing transitions. In the following we use $Execs(s)$ to denote a set of maximal execution fragments that start in s, i.e. $\rho_S[0] = s$.

Example execution fragments for the vending machine from Figure 2.1 are

$$\rho = idle \xrightarrow{insertCoin} choose \xrightarrow{\tau} tea \xrightarrow{serveTea} \ldots$$
$$\varrho = idle \xrightarrow{insertCoin} choose \xrightarrow{\tau} coffee$$

Notice that the execution fragment ρ is maximal, whereas ϱ is not.

In some instances the actions are not important and hence can be omitted. In particular this will be the case in Section 8.1, where we consider the CTL model checking. The result of omitting the actions from an execution fragment is called a *path fragment*. Similarly to the case of execution fragments, we define *path fragments* as well as *maximal path fragments*. All the definitions are obtained in a straightforward manner by omitting the actions.

Definition 2.12 Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. A *finite* path fragment $\widehat{\pi}$ of TS is a sequence of states

$$\widehat{\pi} = s_0 s_1 \ldots s_n \text{ such that } s_i \xrightarrow{\alpha} s_{i+1} \text{ for all } 0 \leq i < n,$$

where $n \geq 0$. We refer to n as the length of the path fragment $\widehat{\pi}$. An *infinite* path fragment π of TS is an infinite sequence of states

$$\pi = s_0 s_1 s_2 \ldots \text{ such that } s_i \xrightarrow{\alpha} s_{i+1} \text{ for all } 0 \leq i.$$

The j-th state of $\pi = s_0 s_1\ldots$ is denoted by $\pi[j] = s_j$, where $j \geq 0$. The notion is defined analogously for finite paths.

Now let us define a notion of a *maximal* path fragment.

Definition 2.13 A *maximal* path fragment is either a finite path fragment that ends in the terminal state, or an infinite path fragment.

Hence according to the above definition the path fragment is maximal if it cannot be prolonged i.e. either it is infinite, or it is finite and ends in a state having no outgoing transitions. In the following we use $Paths(s)$ to denote a set of maximal path fragments that start in s, i.e. $\pi[0] = s$.

Let us again consider the vending machine depicted in Figure 2.1. Example path fragments are

$$\pi = idle\ choose\ tea \ldots$$
$$\widehat{\pi} = idle\ choose\ coffee$$

Notice that the path fragment π is maximal, whereas $\widehat{\pi}$ is not since it ends in a state having an outgoing transition. We also have $\pi[1] = \mathit{choose}$ and $\widehat{\pi}[2] = \mathit{coffee}$.

2.2.2 Control Flow Graphs

Control Flow Graphs (CFGs) are usually used to model a program under consideration. They essentially are directed graphs where nodes represent statements in the program, whereas edges model the flow of control between these statements. We also assume that a control flow graph has two special nodes that are not associated with any statement. Thus, we distinguish a unique initial node, which does not have any incoming edges, and one final node having no outgoing edges. The decision for distinguishing unique initial and final nodes is motivated by simplifications in the specifications of the analyses. The formal definition of the control flow graph is given by Definition 2.14.

Definition 2.14 A Control Flow Graph is a directed graph with one entry node (having no incoming edges) and one exit node (having no outgoing edges), called extremal nodes. The remaining nodes represent program statements and conditions. Furthermore, the edges represent the control flow of the program.

2.2.3 Program Graphs

This section introduces program graphs, a representation of software (hardware) systems that is often used in model checking [4] to model concurrent and distributed systems. Compared to the classical flow graphs [33, 41], the main difference is that in the program graphs the actions label the edges rather than the nodes.

Definition 2.15 A program graph over a space S has the form

$$(\mathsf{Q}, Act, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, S)$$

where

- Q is a finite set of states;
- Act is a finite set of actions;
- $\rightarrow \subseteq \mathsf{Q} \times Act \times \mathsf{Q}$ is a transition relation;
- $\mathsf{Q}_I \subseteq \mathsf{Q}$ is a set of initial states;
- $\mathsf{Q}_F \subseteq \mathsf{Q}$ is a set of final states; and
- $\mathcal{A} : Act \rightarrow S$ specifies the meaning of the actions.

Now let us consider a number of processes each specified as a program graph $PG_i = (\mathsf{Q}_i, Act_i, \rightarrow_i, \mathsf{Q}_{Ii}, \mathsf{Q}_{Fi}, \mathcal{A}_i, S)$ that are executed independently of one another except that they can exchange information via shared variables. The combined program graph $PG = PG_1 \mid\mid\mid \cdots \mid\mid\mid PG_n$ expresses the interleaving between n processes.

Definition 2.16 The interleaved program graph over S

$$\mathsf{PG} = \mathsf{PG}_1 \mid\mid\mid \cdots \mid\mid\mid \mathsf{PG}_n$$

is defined by $(\mathsf{Q}, Act, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, S)$ where

- $\mathsf{Q} = \mathsf{Q}_1 \times \cdots \times \mathsf{Q}_n$,
- $Act = Act_1 \uplus \cdots \uplus Act_n$ (disjoint union),
- $\langle q_1, \cdots, q_i, \cdots, q_n \rangle \xrightarrow{a} \langle q_1, \cdots, q_i', \cdots, q_n \rangle$ if $q_i \xrightarrow{a}_i q_i'$,
- $\mathsf{Q}_I = \mathsf{Q}_{I1} \times \cdots \times \mathsf{Q}_{In}$,
- $\mathsf{Q}_F = \mathsf{Q}_{F1} \times \cdots \times \mathsf{Q}_{Fn}$,
- $\mathcal{A}[\![a]\!] = \mathcal{A}_i[\![a]\!]$ if $a \in Act_i$.

Note that $\mathcal{A}_i : Act_i \to S$ for all i and hence $\mathcal{A} : Act \to S$.

Note that the ability to create interleaved program graphs allows us to model concurrent systems using the same methods as in the case of sequential programs. This will be used to analyse and verify the algorithm in Section 8.1.

2.3 The logic ALFP

This section presents the Alternation-free Least Fixed Point Logic (ALFP) originally introduced in [44]. The logic is a starting point for the formalisms developed in further chapters, and itself it is an extension of *definite* Horn clauses allowing both existential and universal quantifications in preconditions, negative queries, disjunctions of preconditions, and conjunctions of conclusions. In order to deal with negative queries, we restrict ourselves to *alternation-free* formulae that are subject to a notion of *stratification* defined below.

Definition 2.17 Given a fixed countable set $\mathcal{X}$ of variables, a non-empty and finite universe $\mathcal{U}$ and a finite alphabet $\mathcal{R}$ of predicate symbols, we define the set of ALFP formulae (or clause sequences), cls, together with clauses, cl, and preconditions, pre, by the grammar:

$$
\begin{array}{lcl}
u & ::= & x \mid a \\
pre & ::= & R(u_1, \ldots, u_k) \mid \neg R(u_1, \ldots, u_k) \mid pre_1 \wedge pre_2 \\
 & \mid & pre_1 \vee pre_2 \mid \exists x : pre \mid \forall x : pre \\
cl & ::= & R(u_1, \ldots, u_k) \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid pre \Rightarrow cl \mid \forall x : cl \\
cls & ::= & cl_1, \cdots, cl_s
\end{array}
$$

Here $u_i \in (\mathcal{X} \cup \mathcal{U})$, $a \in \mathcal{U}$, $x \in \mathcal{X}$, $R \in \mathcal{R}$ and $s \geq 1, k \geq 0$.

Occurrences of R and $\neg R$ in preconditions are called *positive queries* and *negative queries*, respectively, whereas the other occurrences of R are called *assertions*. An *atom* is written as $R(\vec{u})$, where R is a predicate name and $\vec{u}$ is a non empty list of arguments. A *literal* is either an atom, or a negated atom i.e. $\neg R(\vec{u})$. We say that an atom $R(\vec{u})$ is *ground* when all of its arguments are constants. A *ground clause* is a clause containing only ground atoms. A *fact* is a ground clause without a precondition. A *definition* of a predicate is a clause sequence asserting that predicate. We say that a predicate is a *base predicate* if it is defined only by facts. A clause that is not a fact is a *derivation clause*, and a predicate that is defined only by derivation clauses is a *derived predicate*. We write **1** for the always true clause.

In order to ensure desirable theoretical and pragmatic properties in the presence of negation, we impose a notion of *stratification* similar to the one in Datalog [3, 14]. Intuitively, stratification ensures that a negative query is not performed until the predicate has been fully asserted. This is important for ensuring that once a precondition evaluates to true it will continue to be true even after further assertions of predicates.

Definition 2.18 The formula $cls = cl_1, \cdots, cl_s$ is stratified if there exists a function rank : $\mathcal{R} \to \{0, \cdots, s\}$ such that for all $i = 1, \cdots, s$:

- $\text{rank}(R) = i$ for every assertion R in cl_i;
- $\text{rank}(R) \leq i$ for every positive query R in cl_i; and
- $\text{rank}(R) < i$ for every negative query $\neg R$ in cl_i.

Example 1 *As an example let us define equality predicate E, and non-equality predicate N as follows:*

$$(\forall x : E(x, x)) \land (\forall x : \forall y : \neg E(x, y) \Rightarrow N(x, y))$$

Let us assign $rank(E) = 1$ and $rank(N) = 2$. It is straight-forward to verify that the stratification conditions are fulfilled.

To specify the semantics of ALFP we shall introduce the interpretations $\rho : \mathcal{R} \to \bigcup_k \mathcal{P}(\mathcal{U}^k)$ and $\sigma : \mathcal{X} \to \mathcal{U}$ for predicate symbols and variables, respectively. We shall write $\rho(R)$ for the set of k-tuples $(a_1, \ldots, a_k)$ from $\mathcal{U}^k$ associated with the k-ary predicate R and we write $\sigma(x)$ for the atom of $\mathcal{U}$ bound to x. In the sequel we view the free variables occurring in a formula as constants from the finite universe $\mathcal{U}$. The satisfaction relations for preconditions *pre*, clauses *cl* and clause sequences *cls* are given by:

$$(\rho, \sigma) \models pre, \quad (\rho, \sigma) \models cl \quad \text{and } (\rho, \sigma) \models cls$$

The formal definition is given in Table 2.1; here $\sigma[x \mapsto a]$ stands for the mapping that is as σ except that x is mapped to a.

Now we present the Moore family result for ALFP. A Moore family was formally defined is Section 2.1.

Let $\Delta = \{\rho \mid \rho : \mathcal{R} \to \bigcup_k \mathcal{P}(\mathcal{U}^k)\}$ denote the set of interpretations ρ of predicate symbols in $\mathcal{R}$ over $\mathcal{U}$. We define a lexicographical ordering $\sqsubseteq$ defined by $\rho_1 \sqsubseteq \rho_2$

$$
\begin{array}{llcll}
(\rho,\sigma) & \models & R(\vec{u}) & \underline{\texttt{iff}} & \sigma(\vec{u}) \in \rho(R) \\
(\rho,\sigma) & \models & \neg R(\vec{u}) & \underline{\texttt{iff}} & \sigma(\vec{u}) \notin \rho(R) \\
(\rho,\sigma) & \models & pre_1 \wedge pre_2 & \underline{\texttt{iff}} & (\rho,\sigma) \models pre_1 \text{ and } (\rho,\sigma) \models pre_2 \\
(\rho,\sigma) & \models & pre_1 \vee pre_2 & \underline{\texttt{iff}} & (\rho,\sigma) \models pre_1 \text{ or } (\rho,\sigma) \models pre_2 \\
(\rho,\sigma) & \models & \exists x : pre & \underline{\texttt{iff}} & (\rho,\sigma[x \mapsto a]) \models pre \text{ for some } a \in \mathcal{U} \\
(\rho,\sigma) & \models & \forall x : pre & \underline{\texttt{iff}} & (\rho,\sigma[x \mapsto a]) \models pre \text{ for all } a \in \mathcal{U} \\
\\
(\rho,\sigma) & \models & R(\vec{u}) & \underline{\texttt{iff}} & \sigma(\vec{u}) \in \rho(R) \\
(\rho,\sigma) & \models & \mathbf{1} & \underline{\texttt{iff}} & \texttt{true} \\
(\rho,\sigma) & \models & cl_1 \wedge cl_2 & \underline{\texttt{iff}} & (\rho,\sigma) \models cl_1 \text{ and } (\rho,\sigma) \models cl_2 \\
(\rho,\sigma) & \models & pre \Rightarrow cl & \underline{\texttt{iff}} & (\rho,\sigma) \models cl \text{ whenever } (\rho,\sigma) \models pre \\
(\rho,\sigma) & \models & \forall x : cl & \underline{\texttt{iff}} & (\rho,\sigma[x \mapsto a]) \models cl \text{ for all } a \in \mathcal{U} \\
\\
(\rho,\sigma) & \models & cl_1, \cdots, cl_s & \underline{\texttt{iff}} & (\rho,\sigma) \models cl_i \text{ for all } i, 1 \leq i \leq s
\end{array}
$$

Table 2.1: Semantics of ALFP.

if and only if there is some $0 \leq j \leq s$, where s is the order of the formula, such that the following properties hold:

(a) $\rho_1(R) = \rho_2(R)$ for all $R \in \mathcal{R}$ with $\text{rank}(R) < j$,

(b) $\rho_1(R) \subseteq \rho_2(R)$ for all $R \in \mathcal{R}$ with $\text{rank}(R) = j$,

(c) either $j = s$ or $\rho_1(R) \subset \rho_2(R)$ for some relation $R \in \mathcal{R}$ with $rank(R) = j$.

Lemma 2.19 $\sqsubseteq$ *defines a partial order.*

Lemma 2.20 $(\Delta, \sqsubseteq)$ *is a complete lattice with the greatest lower bound given by*

$$(\bigsqcap M)(R) = \bigcap\{\rho(R) \mid \rho \in M \wedge \forall R' \ rank(R') < rank(R) : \rho(R') = \rho(R)\}$$

which is well defined by induction on the value of $rank(R)$*.*

Proposition 2.21 *Assume cls is a stratified ALFP formula,* σ_0 *is an interpretation of the free variables in cls. Then* $\{\rho \mid (\rho, \sigma_0) \models cls\}$ *is a Moore family.*

The result ensures that the approach falls within the framework of Abstract Interpretation [18, 19]; hence we can be sure that there always is a single best solution for the analysis problem under consideration, namely the one defined in Proposition 2.21.

Example 2 *As an example we can formulate a ALFP clause defining a predicate R that holds on all states in the graph from which no cycle can be reached. The clause is as follows*

$$\forall s : (\forall s' : \neg T(s, s') \vee R(s')) \Rightarrow R(s)$$

where T is a predicate defining the edges of the graph. Note that the above example cannot be defined in Datalog, due to the use of universal quantification in precondition.

Chapter 3

Lattice based Least Fixed Point Logic

Prior work on using logic for specifying static analysis focused on analyses defined over some powerset domain [49, 54, 59]. However, this can be quite limiting. Therefore, in this chapter we present a logic that lifts this restriction, called Lattice based Least Fixed Point Logic (LLFP), that allows interpretations over any complete lattice satisfying Ascending Chain Condition. The main theoretical contribution is a Moore Family result that guarantees that there always is a unique least solution for a given problem.

The chapter is organized as follows. In Section 3.1 we define the syntax and semantics of Lattice based Least Fixed Point Logic. In Section 3.2 we establish a Moore Family result. We continue in Section 3.3 with presenting the relationship between ALFP and LLFP logics. Section 3.4 introduces the implementation of the LLFP logic called $\text{LLFP}^{\#}$. Finally, in Section 3.5 we present an extension of LLFP with monotone functions.

3.1 Syntax and Semantics

In Section 2.3 we presented ALFP logic developed by Nielson et al. [44], which is interpreted over a finite universe of atoms. In this section we present an extension of ALFP called Lattice based Least Fixed Point Logic (LLFP) allowing interpretations over complete lattices satisfying the Ascending Chain Condition. Due to the use of negation in the logic, we need to introduce a complement operator, $\complement$, in the underlying complete lattice. The only condition that we impose on the complement is anti-monotonicity i.e. $\forall l_1, l_2 \in \mathcal{L} : l_1 \sqsubseteq l_2 \Rightarrow \complement l_1 \sqsupseteq \complement l_2$, which is necessary for establishing Moore Family result. The following definition introduces the syntax of LLFP.

Definition 3.1 Given fixed countable and pairwise disjoint sets $\mathcal{X}$ and $\mathcal{Y}$ of variables, a non-empty and finite universe $\mathcal{U}$ and a finite alphabet $\mathcal{R}$ of predicate symbols, we define the set of LLFP formulae (or clause sequences) *cls*, together with clauses *cl*, preconditions *pre*, terms u and V by the grammar:

$$
\begin{array}{lcl}
u & ::= & x \mid a \\
V & ::= & Y \mid [u] \\
pre & ::= & R(\vec{u}; V) \mid \neg R(\vec{u}; V) \mid Y(u) \mid pre_1 \wedge pre_2 \mid pre_1 \vee pre_2 \\
 & \mid & \exists x : pre \mid \exists Y : pre \\
cl & ::= & R(\vec{u}; V) \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid pre \Rightarrow cl \mid \forall x : cl \mid \forall Y : cl \\
cls & ::= & cl_1, \cdots, cl_s
\end{array}
$$

Here $x \in \mathcal{X}$, $a \in \mathcal{U}$, $Y \in \mathcal{Y}$, $R \in \mathcal{R}$, and $s \geq 1$. Furthermore, $\vec{u}$ abbreviates a tuple $(u_1, \cdots, u_k)$ for some $k \geq 0$.

We write $fv(\cdot)$ for the set of free variables in the argument $\cdot$. Occurrences of $R(\vec{u}; V)$ and $\neg R(\vec{u}; V)$ in preconditions are called *positive*, resp. *negative*, queries and we require that $fv(\vec{u}) \subseteq \mathcal{X}$ and $fv(V) \subseteq \mathcal{Y} \cup \mathcal{X}$; these variables are *defining* occurrences. Occurrences of $Y(u)$ in preconditions must satisfy $Y \in \mathcal{Y}$ and $fv(u) \subseteq \mathcal{X}$; Y is an *applied* occurrence, u is a defining occurrence. Clauses of the form $R(\vec{u}; V)$ are called *assertions*; we require that $fv(\vec{u}) \subseteq \mathcal{X}$ and $fv(V) \subseteq \mathcal{Y} \cup \mathcal{X}$ and we note that these variables are *applied* occurrences. A clause *cl* satisfying these conditions together with $fv(cl) = \emptyset$ is said to be *well-formed*; we are only interested in clause sequences *cls* consisting of well-formed clauses.

In order to deal with negation correctly we impose a stratification condition. The definition is exactly the same as the one for ALFP (given in Definition 2.18) and hence omitted.

The following example illustrates the use of negation in an LLFP formula.

Example 3 *Using the notion of stratification we can define equality E and non-equality N predicates in LLFP as follows*

$$(\forall x : E(x; [x])), (\forall x : \forall Y : \neg E(x; Y) \Rightarrow N(x; Y))$$

According to Definition 2.18 the formula is stratified, since predicate E is fully asserted before it is negatively queried in the clause asserting predicate N. As a result we can dispense with an explicit treatment of $=$ and $\neq$ in the development that follows. On the other hand the Definition 2.18 rules out

$$(\forall x : \forall Y : \neg P(x; Y) \Rightarrow Q(x; Y)), (\forall x : \forall Y : \neg Q(x; Y) \Rightarrow P(x; Y))$$

This is because relations P and Q depend negatively on each other. More precisely, it is impossible to have $rank(P) < rank(Q) \wedge rank(Q) < rank(P)$.

To specify the semantics of LLFP we introduce the interpretations ϱ and ς of predicate symbols and variables, respectively. Formally we have

$$\begin{array}{ll} \varrho : & \prod_k \mathcal{R}_{/k} \to \mathcal{U}^k \to \mathcal{L} \\ \varsigma : & (\mathcal{X} \to \mathcal{U}) \times (\mathcal{Y} \to \mathcal{L}_{\neq\perp}) \end{array}$$

In the above $\mathcal{R}_{/k}$ stands for a set of predicate symbols of arity k, and $\mathcal{R}$ is a disjoint union of $\mathcal{R}_{/k}$, hence $\mathcal{R} = \biguplus_k \mathcal{R}_{/k}$. We write $\varsigma(x)$ for the element from $\mathcal{U}$ bound to $x \in \mathcal{X}$ and $\varsigma(Y)$ for the element of $\mathcal{L}_{\neq\perp}$ bound to $Y \in \mathcal{Y}$, where $\mathcal{L}_{\neq\perp} = \mathcal{L} \setminus \{\perp\}$. We do not allow variables from $\mathcal{Y}$ to be mapped to $\perp$ in order to establish a relationship between ALFP and LLFP in the case of a powerset lattice, i.e. $\mathcal{P}(\mathcal{U})$, which we present in Section 3.3. The interpretation of terms is generalized to sequences $\vec{u}$ of terms in a pointwise manner by taking $\varsigma(a) = a$ for all $a \in \mathcal{U}$, thus $\varsigma(u_1, \cdots, u_k) = (\varsigma(u_1), \cdots, \varsigma(u_k))$. In order to give the interpretation of $[u]$, we introduce a function $\beta : \mathcal{U} \to \mathcal{L}$. The β function is called a *representation function* and the idea is that β maps a value from the universe $\mathcal{U}$ to the *best* property describing it. For example in the case of a powerset lattice, β could be defined by $\beta(a) = \{a\}$ for all $a \in \mathcal{U}$. The interpretation of $[u]$ is given by $\varsigma([u]) = \beta(\varsigma(u))$.

The satisfaction relations for preconditions *pre*, clauses *cl* and clause sequences *cls* are given in Table 3.1; here $\varsigma[x \mapsto a]$ stands for the mapping that is as ς except that x is mapped to a and similarly $\varsigma[Y \mapsto l]$ stands for the mapping that is as ς except that Y is mapped to $l \in \mathcal{L}_{\neq\perp}$.

Example 4 *As an example we can formulate the classical live variables analysis in LLFP. Let the complete lattice be $(\mathcal{P}(\mathcal{U}), \subseteq, \cup, \cap, \emptyset, \mathcal{U})$. The complement operator is defined as a set complement, $\complement$, and the representation function is given by $\beta(a) = \{a\}$ for all $a \in \mathcal{U}$. Assume that we have a program graph (defined in Section 2.2.3) with three kinds of actions: $x := e$, e, and skip. Then we*

(ϱ, ς)	$\models_\beta$	$R(\vec{u}; V)$	<u>iff</u>	$\varrho(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(V)$
(ϱ, ς)	$\models_\beta$	$\neg R(\vec{u}; V)$	<u>iff</u>	$\complement(\varrho(R)(\varsigma(\vec{u}))) \sqsupseteq \varsigma(V)$
(ϱ, ς)	$\models_\beta$	$Y(u)$	<u>iff</u>	$\beta(\varsigma(u)) \sqsubseteq \varsigma(Y)$
(ϱ, ς)	$\models_\beta$	$pre_1 \wedge pre_2$	<u>iff</u>	$(\varrho, \varsigma) \models_\beta pre_1$ and $(\varrho, \varsigma) \models_\beta pre_2$
(ϱ, ς)	$\models_\beta$	$pre_1 \vee pre_2$	<u>iff</u>	$(\varrho, \varsigma) \models_\beta pre_1$ or $(\varrho, \varsigma) \models_\beta pre_2$
(ϱ, ς)	$\models_\beta$	$\exists x : pre$	<u>iff</u>	$(\varrho, \varsigma[x \mapsto a]) \models_\beta pre$ for some $a \in \mathcal{U}$
(ϱ, ς)	$\models_\beta$	$\exists Y : pre$	<u>iff</u>	$(\varrho, \varsigma[Y \mapsto l]) \models_\beta pre$ for some $l \in \mathcal{L}_{\neq\perp}$
(ϱ, ς)	$\models_\beta$	$R(\vec{u}; V)$	<u>iff</u>	$\varrho(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(V)$
(ϱ, ς)	$\models_\beta$	$\mathbf{1}$	<u>iff</u>	`true`
(ϱ, ς)	$\models_\beta$	$cl_1 \wedge cl_2$	<u>iff</u>	$(\varrho, \varsigma) \models_\beta cl_1$ and $(\varrho, \varsigma) \models_\beta cl_2$
(ϱ, ς)	$\models_\beta$	$pre \Rightarrow cl$	<u>iff</u>	$(\varrho, \varsigma) \models_\beta cl$ whenever $(\varrho, \varsigma) \models_\beta pre$
(ϱ, ς)	$\models_\beta$	$\forall x : cl$	<u>iff</u>	$(\varrho, \varsigma[x \mapsto a]) \models_\beta cl$ for all $a \in \mathcal{U}$
(ϱ, ς)	$\models_\beta$	$\forall Y : cl$	<u>iff</u>	$(\varrho, \varsigma[Y \mapsto l]) \models_\beta cl$ for all $l \in \mathcal{L}_{\neq\perp}$
(ϱ, ς)	$\models_\beta$	$cl_1, \cdots, cl_s$	<u>iff</u>	$(\varrho, \varsigma) \models_\beta cl_i$ for all $i, 1 \leq i \leq s$

Table 3.1: Semantics of LLFP.

can define the KILL and GEN predicates for the assignment action $q_s \xrightarrow{x:=e} q_t$ *by the two clauses*

$$\forall Y : FV(q_s; Y) \Rightarrow GEN(q_s; Y) \wedge KILL(q_s; [x])$$

where $FV(q_s; Y)$ *captures a set of free variables* Y *occurring in the expression* e*. We also define the GEN predicate for an action* $q_s \xrightarrow{e} q_t$ *as follows*

$$\forall Y : FV(q_s; Y) \Rightarrow GEN(q_s; Y)$$

The analysis itself is defined by the predicate LV; whenever we have $q_s \xrightarrow{x:=e} q_t$ *in the program graph we generate the clause*

$$\forall Y : (LV(q_t; Y) \wedge \neg KILL(q_s; Y)) \vee GEN(q_s; Y) \Rightarrow LV(q_s; Y)$$

Similarly whenever we have $q_s \xrightarrow{e} q_t$ *or* $q_s \xrightarrow{skip} q_t$ *in the program graph we generate the clause*

$$\forall Y : LV(q_t; Y) \Rightarrow LV(q_s; Y)$$

3.2 Moore family result for LLFP

In this section we establish a Moore family result for LLFP that guarantees that there always is a unique best solution for LLFP clauses. A Moore family was formally defined in Section 2.1.

Assume cls has the form $cl_1, \ldots, cl_s$, and let $\Delta = \{\varrho : \prod_k \mathcal{R}_{/k} \rightarrow \mathcal{U}^k \rightarrow \mathcal{L}\}$ denote the set of interpretations ϱ of predicate symbols in $\mathcal{R}$. We also define the lexicographical ordering $\preceq$ such that $\varrho_1 \preceq \varrho_2$ if and only if there is some $1 \leq j \leq s$, where s is the order of the formula, such that the following properties hold:

(a) $\varrho_1(R) = \varrho_2(R)$ for all $R \in \mathcal{R}$ with $\text{rank}(R) < j$,

(b) $\varrho_1(R) \sqsubseteq \varrho_2(R)$ for all $R \in \mathcal{R}$ with $\text{rank}(R) = j$,

(c) either $j = s$ or $\varrho_1(R) \sqsubset \varrho_2(R)$ for at least one $R \in \mathcal{R}$ with $\text{rank}(R) = j$.

We say that $\varrho_1(R) \sqsubseteq \varrho_2(R)$ if and only if $\forall \vec{a} \in \mathcal{U}^k : \varrho_1(R)(\vec{a}) \sqsubseteq \varrho_2(R)(\vec{a})$, where $k \geq 0$ is the arity of R. Notice that in the case $s = 1$, the above ordering coincides with the lattice ordering $\sqsubseteq$. Intuitively, the lexicographical ordering $\preceq$ orders the relations strata by strata starting with the strata 0. It is essentially analogous to the lexicographical ordering on strings, which is based on the alphabetical order of their characters. The conditions (a)-(c) are exactly the same as the ones in the definition of partial order for ALFP (see Section 2.3 for details). The only distinction follows from the difference in the definitions of the interpretations of predicate symbols ϱ in LLFP and ρ in ALFP.

Lemma 3.2 $\preceq$ *defines a partial order.*

PROOF. See Appendix A.1.

Assume cls has the form $cl_1, \ldots, cl_s$ where cl_j is the clause corresponding to stratum j, and let $M \subseteq \Delta$ denote a set of assignments which map relation symbols to relations.

Lemma 3.3 $\Delta = (\Delta, \preceq)$ *is a complete lattice with the greatest lower bound* $\sqcap_\Delta$ *given by*

$$\left(\sqcap_\Delta M\right)(R) = \lambda \vec{a}. \sqcap \left\{\varrho(R)(\vec{a}) \mid \varrho \in M_{rank(R)}\right\}$$

where

$$M_j = \left\{\varrho \in M \mid \forall R' \;\; rank(R') < j : \varrho(R') = \left(\sqcap_\Delta M\right)(R')\right\}$$

PROOF. See Appendix A.2

Note that $\bigsqcap_\Delta M$ is well defined by induction on j observing that $M_0 = M$ and $M_j \subseteq M_{j-1}$. Intuitively, $\bigsqcap_\Delta M$ is defined strata by strata starting with strata 0. For strata j and relation R of rank j we define $(\bigsqcap_\Delta M)(R)$ as a function that takes a tuple $\vec{a}$ as argument and returns a lattice element that is a greatest lower bound of all these $\varrho(R)(\vec{a})$ whose interpretation $\varrho \in M$ matches $\bigsqcap_\Delta M$ for all relations with rank less than j.

Proposition 3.4 *Assume cls is a stratified LLFP clause sequence, and let ς_0 be an interpretation of free variables in cls. Furthermore, ϱ_0 is an interpretation of all relations of rank 0. Then*

$$\{\varrho \mid (\varrho, \varsigma_0) \models_\beta cls \land \forall R : rank(R) = 0 \Rightarrow \varrho_0(R) \sqsubseteq \varrho(R)\}$$

is a Moore family.

Proof. See Appendix A.3.

The result ensures that the approach falls within the framework of Abstract Interpretation [18, 19]; hence we can be sure that there always is a single best solution for the analysis problem under consideration, namely the one defined in Proposition 3.4.

3.3 The relationship to ALFP

This section aims to establish the relationship between ALFP and LLFP logics. Here, we consider the Datalog fragment of ALFP, i.e. we do not allow universal quantification in preconditions in the ALFP formulae. In the remainder of this chapter when we refer to ALFP, we mean the Datalog fragment of ALFP. As the reader may have already noticed, in case the underlying complete lattice is $\mathcal{P}(\mathcal{U})$ the two logics are essentially equivalent. Therefore, we can translate every LLFP formula into a corresponding ALFP one and vice versa. In this section we show this transformation. In particular we present a transformation from LLFP to ALFP and prove its correctness; finally we show how ALFP can be embedded in LLFP.

3.3.1 From LLFP to ALFP

Let the underlying complete lattice be $\mathcal{P}(\mathcal{U})$. The aim is now to transform clauses in LLFP into ALFP. In order to map elements of the universe $\mathcal{U}$ into the elements of the powerset lattice $\mathcal{P}(\mathcal{U})$, we define the representation function $\beta : \mathcal{U} \rightarrow \mathcal{P}(\mathcal{U})$ as $\beta(a) = \{a\}$ for all $a \in \mathcal{U}$. The idea is that a relation R in LLFP with interpretation $\varrho(R) : \mathcal{U}^k \rightarrow \mathcal{P}(\mathcal{U})$ is replaced by a relation in ALFP (also named R) with interpretation $f(\varrho)(R) \in \mathcal{P}(\mathcal{U}^{k+1})$. More precisely

$$f(\varrho)(R) = \{(\vec{a}, b) \in \mathcal{U}^{k+1} \mid \beta(b) \subseteq \varrho(R)(\vec{a})\} \tag{3.1}$$

Note that if $\varrho(R)(\vec{a}) = \bot$ then $f(\varrho)(R)$ does not contain any tuples with $\vec{a}$ as the first k components. Furthermore, in order to correctly transform interpretations of predicate symbols in case of negations, we define the complement operator, $\complement$, as a set complement on the universe $\mathcal{U}$.

In the transformation from LLFP to ALFP we need to replace the variables ranging over the complete lattice with variables ranging over the universe and we need to ensure that these variables only take values corresponding to those of the complete lattice. To capture this for each variable $Y \in \mathcal{Y}$ we introduce a special variable $x_Y \in \mathcal{X}$, and ensure that the interpretation of x_Y in ALFP will correspond to one of the potential values in the interpretation of Y in LLFP. Thus for each mapping ς in LLFP we have a number of mappings σ in ALFP; this is formalized by

$$f(\varsigma) = \left\{ \sigma : \mathcal{X} \rightarrow \mathcal{U} \;\middle|\; \begin{array}{ll} \sigma(x) = \varsigma(x) & \text{whenever } x \in \mathcal{X} \\ \beta(\sigma(x_Y)) \subseteq \varsigma(Y) & \text{whenever } Y \in \mathcal{Y} \end{array} \right\} \tag{3.2}$$

The replacement of the variables Y with variables x_Y is generalized to a transformation on terms by taking

$$\begin{array}{rcl} f(Y) & = & x_Y \\ f([u]) & = & u \end{array}$$

The latter reflects that $[u]$ represents a singleton set in the powerset lattice $\mathcal{P}(\mathcal{U})$ in LLFP. The preconditions, clauses, and clause sequences of LLFP are now transformed into preconditions, clauses, and clause sequences of ALFP using the function f defined in Table 3.2. The transformation is defined in a syntax directed manner with the quantification over a variable $Y \in \mathcal{Y}$ and $Y(u)$ being the only non-trivial cases. Firstly, each quantification over a variable $Y \in \mathcal{Y}$ is transformed into a quantification over variable x_Y; this is necessary because as mentioned above all occurrences of the variables $Y \in \mathcal{Y}$ are replaced by x_Y. This means that the quantification over sets (variables from $\mathcal{Y}$) in LLFP corresponds to quantification over the elements of these sets in ALFP. Furthermore, the $Y(u)$ construct in LLFP amounts to checking whether the lattice element corresponding to the constant bound to u is less or equal to a lattice element bound to Y. In the current setting the semantics essentially boils down to checking whether $\{a\} \subseteq \varsigma(Y)$, where a is an element of $\mathcal{U}$ bound

$$
\begin{array}{lcl}
f(R(\vec{u};V)) & = & R(\vec{u}, f(V)) \\
f(\neg R(\vec{u};V)) & = & \neg R(\vec{u}, f(V)) \\
f(Y(u)) & = & x_Y = u \\
f(pre_1 \wedge pre_2) & = & f(pre_1) \wedge f(pre_2) \\
f(pre_1 \vee pre_2) & = & f(pre_1) \vee f(pre_2) \\
f(\exists x : pre) & = & \exists x : f(pre) \\
f(\exists Y : pre) & = & \exists x_Y : f(pre) \\
\\
f(R(\vec{u};V)) & = & R(\vec{u}, f(V)) \\
f(1) & = & 1 \\
f(cl_1 \wedge cl_2) & = & f(cl_1) \wedge f(cl_2) \\
f(pre \Rightarrow cl) & = & f(pre) \Rightarrow f(cl) \\
f(\forall x : cl) & = & \forall x : f(cl) \\
f(\forall Y : cl) & = & \forall x_Y : f(cl) \\
\\
f(cl_1, \cdots, cl_s) & = & f(cl_1), \cdots, f(cl_s)
\end{array}
$$

Table 3.2: Transformation from LLFP to ALFP.

to u, i.e. $\varsigma(u) = a$. Therefore, since the variables $Y \in \mathcal{Y}$ are replaced by x_Y, we transform $Y(u)$ into the test $x_Y = u$ checking essentially the same condition in ALFP.

Example 5 *Continuing Example 4, the LLFP specification of the live variables analysis can be transformed into ALFP using function f defined in Table 3.2. The resulting clause for an assignment $q_s \xrightarrow{x:=e} q_t$ is of the form:*

$$\forall x_Y : (LV(q_t, x_Y) \wedge \neg KILL(q_s, x_Y)) \vee GEN(q_s, x_Y) \Rightarrow LV(q_s, x_Y)$$

Similarly for a test $q_s \xrightarrow{e} q_t$ we get

$$\forall x_Y : LV(q_t, x_Y) \Rightarrow LV(q_s, x_Y)$$

The resulting clauses are exactly the same as if they were written directly in ALFP.

The following result captures the relationship between ALFP and LLFP.

Proposition 3.5 *If ϕ is a well formed LLFP formula (a precondition, clause or a clause sequence), the underlying complete lattice is $\mathcal{P}(\mathcal{U})$ and $\beta : \mathcal{U} \to \mathcal{L}$ is defined as $\beta(a) = \{a\}$ for all $a \in \mathcal{U}$, then*

$$(\varrho, \varsigma) \models_\beta \phi \quad \Leftrightarrow \quad \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\phi)$$

Proof. See Appendix A.4.

3.3.2 From ALFP to LLFP

Now we show how the ALFP logic can be transformed into LLFP. Let $\mathcal{L} = (\{\bot, \top\}, \sqsubseteq)$ be a complete lattice such that $\complement\top = \bot$ and $\complement\bot = \top$. Moreover let function β map all elements of the universe into $\top$; namely $\forall a \in \mathcal{U} : \beta(a) = \top$. First we define the transformation for ALFP clauses and preconditions, which is accomplished by adding $[a]$, where $a \in \mathcal{U}$, as the second component of relations. The transformation for positive queries and assertions is $f'(R(\vec{u})) = R(\vec{u}; [a])$; for negative queries it is $f'(\neg R(\vec{u})) = \neg R(\vec{u}; [a])$. For all other syntactic categories it is an identity function. Moreover the transformation of the interpretations of predicate symbols is defined as

$$f'(\rho)(R) = \lambda\vec{u}.\begin{cases} \top & \text{whenever } \vec{u} \in \rho(R) \\ \bot & \text{otherwise.} \end{cases}$$

Since all occurrences of predicates are transformed by adding $[a]$ in the second component there are no variables in $\mathcal{Y}$. Hence the transformation of interpretations of variables is defined as $f'(\varsigma) = (\varsigma, [\,])$, where $[\,]$ stands for the empty mapping.

Example 6 *As an example let us consider ALFP clauses for transitive closure*

$$(\forall x : \forall y : E(x, y) \Rightarrow T(x, y)) \wedge (\forall x : \forall z : (\exists y : E(x, y) \wedge T(y, z)) \Rightarrow T(x, z))$$

Let a be some constant from $\mathcal{U}$; then the above clause is transformed into following LLFP formula

$$\begin{array}{c}(\forall x : \forall y : E(x, y; [a]) \Rightarrow T(x, y; [a]))\wedge \\ (\forall x : \forall z : (\exists y : E(x, y; [a]) \wedge T(y, z; [a])) \Rightarrow T(x, z; [a]))\end{array}$$

Based on the above example, we can see that ALFP can easily be embedded in LLFP by adding a 'dummy' lattice component. Note, that neither nesting depth of quantifiers nor the size of the corresponding formula changes.

3.4 Implementation of LLFP

In this section we present LLFP$^{\#}$ logic, which the implementation of LLFP. The reason for introducing LLFP$^{\#}$ is to achieve an efficient implementation of the logic. The main differences between LLFP and LLFP$^{\#}$ are as follows

- LLFP$^\#$ formulae do not contain disjunctions of preconditions,
- each variable $Y \in \mathcal{Y}$ quantified in an LLFP$^\#$ clause has at most one defining occurrence in that clause.

The intuition behind the above restrictions is that we want the interpretations of variables $Y \in \mathcal{Y}$ to be as large as possible. However, since in LLFP the given variable may be used as an argument in a number of queries, it cannot be maximal for all instances. Furthermore, the restrictions allow to handle the $Y(u)$ construct in an easy and efficient manner. This will become evident in Section 3.4.3 where we ensure that a given variable $Y \in \mathcal{Y}$ has at most one defining occurrence in the given clause.

The rest of this section is organized as follows. We begin with presenting a syntactic transformation of LLFP formulae into a Horn format in Section 3.4.1. Section 3.4.2 presents the syntax and semantics of LLFP$^\#$. Finally in Section 3.4.3 we show how LLFP clauses in Horn format can be transformed into LLFP$^\#$ clauses, and establish the semantical equivalence between LLFP and LLFP$^\#$.

3.4.1 From LLFP to Horn format

As a first step towards getting an implementation of LLFP we transform the clauses into Horn format. This transformation is fairly straightforward since the Datalog fragment of ALFP (without universal quantifications in preconditions) corresponds to Horn clauses.

Definition 3.6 An LLFP precondition, clause or clause sequence is in *Horn format* if it is defined by the grammar:

$$\begin{array}{rcl} u & ::= & x \mid a \\ V & ::= & Y \mid [u] \\ pre' & ::= & R(\vec{u}; V) \mid \neg R(\vec{u}; V) \mid Y(u) \mid pre'_1 \wedge pre'_2 \\ cl'' & ::= & R(\vec{u}; V) \mid \mathbf{1} \mid pre' \Rightarrow R(\vec{u}; V) \\ cl' & ::= & \forall x : cl'' \mid \forall Y : cl'' \mid cl'' \\ cls' & ::= & cl'_1, \cdots, cl'_s \end{array}$$

This means that all clauses can be written in the form

$$(\forall \vec{\alpha}_1 : cl''_1) \wedge \cdots \wedge (\forall \vec{\alpha}_m : cl''_m)$$

where each cl''_j has the form $R(\vec{u}; V)$, $\mathbf{1}$, or $pre' \Rightarrow R(\vec{u}; V)$ and $\vec{\alpha_j}$ is a (possibly empty) sequence of quantifiers over variables in $\mathcal{X} \cup \mathcal{Y}$. Furthermore, no preconditions pre' contain disjunctions.

The transformation proceeds in a number of stages:

1. First the variables introduced by the quantifiers are renamed so that they are pairwise distinct. This is needed in order to avoid name captures.
2. All existential quantifiers in preconditions are turned into universal quantifiers for clauses. Thus $(\exists x : pre) \Rightarrow cl$ becomes $\forall x : (pre \Rightarrow cl)$ and similarly $(\exists Y : pre) \Rightarrow cl$ becomes $\forall Y : (pre \Rightarrow cl)$.
3. Preconditions of all clauses are transformed into the form $pre'_1 \vee \cdots \vee pre'_k$ where each of the pre'_i is a conjunction of queries of the form $R(\vec{u}; \vec{V})$, $\neg R(\vec{u}; \vec{V})$, and $Y(\vec{u})$ (so they adhere to the grammar for pre' given above).
4. Then the clauses are transformed so that they do not use disjunction in preconditions, that is, all occurrences of $(pre'_1 \vee \cdots \vee pre'_k) \Rightarrow cl$ are replaced by the k conjuncts $(pre'_1 \Rightarrow cl) \wedge \cdots \wedge (pre'_k \Rightarrow cl)$.
5. All (universal) quantifiers are moved to the outermost level in the clauses. Thus $pre' \Rightarrow (\forall x : cl)$ becomes $\forall x : (pre' \Rightarrow cl)$ and similarly $pre' \Rightarrow (\forall Y : cl)$ becomes $\forall Y : (pre' \Rightarrow cl)$.
6. Finally all clauses of the form $\forall \vec{\alpha} : (pre' \Rightarrow cl_1 \wedge cl_2)$ are replaced by clauses of the form $(\forall \vec{\alpha} : (pre' \Rightarrow cl_1)) \wedge (\forall \vec{\alpha} : (pre' \Rightarrow cl_2))$ and all clauses of the form $\forall \vec{\alpha} : (pre' \Rightarrow (pre'' \Rightarrow cl))$ are replaced by clauses of the form $\forall \vec{\alpha} : (pre' \wedge pre'' \Rightarrow cl)$. Since there can be more than one conjunction or implication in the conclusion, this step is performed iteratively until no more transformations can be done.

We can then establish:

Lemma 3.7 *If cl is an LLFP clause, then $h(cl)$ is in Horn format and:*

$$(\varrho^*, \varsigma^*) \models_\beta cl \quad \Leftrightarrow \quad (\varrho^*, \varsigma^*) \models_\beta h(cl)$$

PROOF. See Appendix A.5.

Example 7 *Continuing Example 4, we can transform the Live Variables Analysis specification into the Horn format. The LLFP clause for an assignment $q_s \xrightarrow{x:=e} q_t$ can be written in the Horn format as follows:*

$$\begin{array}{l}\forall Y : LV(q_t; Y) \wedge \neg KILL(q_s; Y) \Rightarrow LV(q_s; Y) \\ \forall Y : GEN(q_s; Y) \Rightarrow LV(q_s; Y)\end{array}$$

Similarly for $q_s \xrightarrow{e} q_t$ *we have:*

$$\forall Y : LV(q_t; Y) \Rightarrow LV(q_s; Y)$$

3.4.2 The Logic LLFP$^{\#}$

In this section we introduce the variant LLFP$^{\#}$ of the logic. In LLFP the semantics of a positive query $R(\vec{u}; Y)$ states that $\varrho(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(Y)$; in LLFP$^{\#}$ we want to have $\varrho(R)(\varsigma(\vec{u})) = \varsigma(Y)$ so that the interpretation of Y is as large as possible.

However, a number of queries using the same variable Y implicitly impose some restrictions on the interpretation of the variables meaning that it cannot be chosen to be maximal in all instances; e.g. this is the case in the clause in Example 4. In LLFP$^{\#}$ these implicit operations have to be explicit. As a consequence an LLFP clause as

$$\forall Y : (LV(q_s; Y) \wedge \neg KILL(q_s; Y)) \vee GEN(q_s; Y) \Rightarrow LV(q_t; Y)$$

now has to be rewritten as

$$\begin{array}{l} \forall Y_1 : \forall Y_2 : LV(q_s; Y_1) \wedge KILL(q_s; Y_2) \Rightarrow LV(q_t; Y_1 \sqcap \complement Y_2) \quad \wedge \\ \forall Y_3 : GEN(q_s; Y_3) \Rightarrow LV(q_t; Y_3) \end{array}$$

so that it is in Horn format, uses distinct variables, and performs the explicit operations on the variables in the assertions.

To summarize the logic, LLFP$^{\#}$ is patterned after the Horn format introduced above but extended to allow terms W to be used in applied positions:

Definition 3.8 Given fixed countable and pairwise disjoint sets $\mathcal{X}$ and $\mathcal{Y}$ of variables, a non-empty and finite universe $\mathcal{U}$, and a finite alphabet $\mathcal{R}$ of predicate symbols, we define the set of LLFP$^{\#}$ formulae (or clause sequences) cls, together with clauses cl, and preconditions pre, by the grammar:

$$\begin{array}{lcl} u & ::= & x \mid a \\ V & ::= & Y \mid [u] \\ W & ::= & V \mid W_1 \sqcap W_2 \mid \complement W \mid L \\ pre & ::= & R(\vec{u}; V) \mid \neg R(\vec{u}; [u]) \mid W(u) \mid pre_1 \wedge pre_2 \\ cl' & ::= & R(\vec{u}; W) \mid \mathbf{1} \mid pre \Rightarrow R(\vec{u}; W) \\ cl & ::= & \forall x : cl \mid \forall Y : cl \mid cl' \\ cls & ::= & cl_1, \cdots, cl_s \end{array}$$

Here $x \in \mathcal{X}$, $a \in \mathcal{U}$, $Y \in \mathcal{Y}$, $L \in \mathcal{L}$, $R \in \mathcal{R}$ and $s \geq 1$.

The term W can be either a variable Y, a lattice element $[u]$, the greatest lower bound of two lattice elements, the complement of a lattice element, or it can be the top element as denoted by constant L. The terms W can be used in all applied occurrences, that is the queries $Y(u)$ of LLFP have been generalized to $W(u)$ and the assertions $R(\vec{u}; V)$ have been generalized to $R(\vec{u}; W)$. Note that negated queries are only of the form $\neg R(\vec{u}; [u])$, which is a consequence of the transformation form LLFP in Horn format into LLFP$^{\#}$. The details of the transformation are given in the next section.

To specify the semantics of LLFP$^{\#}$ we make use of the interpretations ϱ and ς used for LLFP. We again make use of function $\beta : \mathcal{U} \to \mathcal{L}$; the details are given in Table 3.3; here we extend the interpretation ς of variables to the terms W as follows:

$$\begin{array}{rcl} \varsigma([u]) & = & \beta(\varsigma(u)) \\ \varsigma(W_1 \sqcap W_2) & = & \varsigma(W_1) \sqcap \varsigma(W_2) \\ \varsigma(\complement W) & = & \complement \varsigma(W) \\ \varsigma(L) & = & \top \end{array}$$

The interpretation for $W(u)$ then amounts to $\beta(\varsigma(u)) \sqsubseteq \varsigma(W)$. The interpretation for assertions $R(\vec{u}; W)$ amounts to $\varsigma(W) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$ as shown in Table 3.3.

3.4.3 From LLFP in Horn format to LLFP$^{\#}$

We have seen that LLFP clauses can be transformed into Horn format and now we show that LLFP clauses in Horn format can be transformed into LLFP$^{\#}$ clauses. So let us consider a clause cl of the form

$$\forall \vec{\alpha} : \forall \vec{Y} : pre \Rightarrow R(\vec{u}; V)$$

where $\vec{\alpha}$ is a (possibly non-empty) sequence of variables from $\mathcal{X}$, and $\vec{Y}$ is (possibly non-empty) sequence of variables $Y \in \mathcal{Y}$ – we can without loss of generality assume that variables from $\mathcal{Y}$ are the last variables in the sequence of universally quantified variables. Let us assume that pre contains $k \geq 0$ defining occurrences of the variable $Y \in \mathcal{Y}$ and let $Y_1, \cdots, Y_k$ be k fresh variables from $\mathcal{Y}$. The LLFP$^{\#}$ clause $g(cl)$ is then obtained as follows:

1. Rename the k defining occurrences of Y in pre to be $Y_1, \cdots, Y_k$, and call the resulting precondition pre'.

$$
\begin{array}{llll}
(\varrho,\varsigma) & \models^{\#}_{\beta} & R(\vec{u};V) & \underline{\texttt{iff}} \left\{ \begin{array}{l} \varrho(R)(\varsigma(\vec{u})) = \varsigma(V), \text{ if } V \in \mathcal{Y}. \\ \varrho(R)(\varsigma(u)) \sqsupseteq \varsigma(V), \text{ otherwise} \end{array} \right. \\
(\varrho,\varsigma) & \models^{\#}_{\beta} & \neg R(\vec{u};[u]) & \underline{\texttt{iff}} \ \complement\varrho(R)(\varsigma(u)) \sqsupseteq \varsigma([u]) \\
(\varrho,\varsigma) & \models^{\#}_{\beta} & W(u) & \underline{\texttt{iff}} \ \beta(\varsigma(u)) \sqsubseteq \varsigma(W) \\
(\varrho,\varsigma) & \models^{\#}_{\beta} & pre_1 \wedge pre_2 & \underline{\texttt{iff}} \ (\varrho,\varsigma) \models^{\#} pre_1 \text{ and } (\varrho,\varsigma) \models^{\#}_{\beta} pre_2 \\
(\varrho,\varsigma) & \models^{\#}_{\beta} & R(\vec{u};W) & \underline{\texttt{iff}} \ \varrho(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(W) \\
(\varrho,\varsigma) & \models^{\#}_{\beta} & \mathbf{1} & \underline{\texttt{iff}} \ \texttt{true} \\
(\varrho,\varsigma) & \models^{\#}_{\beta} & pre \Rightarrow cl & \underline{\texttt{iff}} \ (\varrho,\varsigma) \models^{\#} cl \text{ whenever } (\varrho,\varsigma) \models^{\#}_{\beta} pre \\
(\varrho,\varsigma) & \models^{\#}_{\beta} & \forall x : cl & \underline{\texttt{iff}} \ (\varrho,\varsigma[x \mapsto a]) \models^{\#} cl \text{ for all } a \in \mathcal{U} \\
(\varrho,\varsigma) & \models^{\#}_{\beta} & \forall Y : cl & \underline{\texttt{iff}} \ (\varrho,\varsigma[Y \mapsto l]) \models^{\#} cl \text{ for all } l \in \mathcal{L}_{\neq\perp} \\
(\varrho,\varsigma) & \models^{\#}_{\beta} & cl_1,\cdots,cl_s & \underline{\texttt{iff}} \ (\varrho,\varsigma) \models^{\#}_{\beta} cl_i \text{ for all } i, 1 \leq i \leq s
\end{array}
$$

Table 3.3: Semantics of LLFP$^{\#}$.

2. Replace all occurrences of $\neg R(\vec{u};Y)$ by $R(\vec{u};Y_i)$
3. Replace all occurrences of $Y(u')$ by $W^{Y}_{pre'}(u')$ (defined below).

Then the transformation g will return the clause

$$g(cl) = \forall\vec{\alpha} : \forall Y_1 : \cdots \forall Y_k : pre' \Rightarrow R(\vec{u}; W^{V}_{pre'})$$

where $W^{V}_{pre'}$ (and $W^{Y}_{pre'}$) is a term that captures how the variables $Y_1, \cdots, Y_k$ are used in the precondition pre' to produce V (and Y, respectively). The term is defined as follows:

$$
\begin{array}{rcl}
W^{Y}_{R'(\vec{u'};V)} & = & \left\{ \begin{array}{ll} Y_i & \text{if } V = Y_i \text{ for some } i \\ L & \text{otherwise} \end{array} \right. \\
W^{Y}_{\neg R'(\vec{u'};V)} & = & \left\{ \begin{array}{ll} \complement Y_i & \text{if } V = Y_i \text{ for some } i \\ L & \text{otherwise} \end{array} \right. \\
W^{V}_{Y(u)} & = & L \\
W^{Y}_{pre_1 \wedge pre_2} & = & W^{Y}_{pre_1} \sqcap W^{Y}_{pre_2} \\
W^{[u]}_{pre'} & = & [u]
\end{array}
$$

The idea is that if Y does not occur in a defining position in pre' then $W^{Y}_{pre'}$ is equal to L meaning that no restrictions have been imposed on the interpretation of Y. If Y occurs in a positive query as Y_i then we record Y_i and if it occurs in a negative query as Y_i then we record $\complement Y_i$. In case of a conjunction of two preconditions we take the greatest lower bound of the terms from the two conjuncts. The last clause in the definition above takes care of the special case where an assertion has the form $R(\vec{u}; [u])$ and $W^{[u]}_{pre'}$ just have to record $[u]$ independently of the form of pre'.

Lemma 3.9 *Assume that pre contains k defining occurrences of Y and that*

$$(\varrho, \varsigma[Y \mapsto l]) \models_\beta pre$$
$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_\beta g(pre)$$

where $l_i \sqsubseteq l$ *for* $1 \leq i \leq k$*. Then*

$$\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k](W^Y_{pre}) = l$$

If Y does not occur in a defining position in pre then $W^Y_{pre} = L$.

Proof. See Appendix A.6.

The following result shows that satisfiability of the LLFP$^{\#}$ clause $g(cl)$ implies satisfiability of the LLFP clause cl:

Lemma 3.10 *Assume that* $(\varrho, \varsigma) \models^{\#}_\beta g(cl)$*. Then* $(\varrho, \varsigma) \models_\beta cl$*.*

Proof. The proof is by a case analysis of the conclusion of the clause cl and makes use of Lemma 3.9 above. The details of the proof can be found in Appendix A.6.

Lemma 3.11 *Assume that* $(\varrho, \varsigma) \models_\beta cl$*. Then* $(\varrho, \varsigma) \models^{\#}_\beta g(cl)$*.*

Proof. The proof is by a case analysis of the conclusion of the clause cl and makes use of Lemma 3.9 above. The details of the proof can be found in Appendix A.6.

Example 8 *As an example we can formulate the detection of signs analysis in LLFP*$^{\#}$*. Assume that we have a program graph with three kinds of actions:* $x := y \star z$*, e, and skip.*

The analysis is defined by the predicate SA; whenever we have $q_s \xrightarrow{x:=y\star z} q_t$ *in the program graph we generate the clauses*

$$\forall v : \forall S : SA(q_s, v; S) \wedge v \neq x \Rightarrow SA(q_t, v; S) \wedge$$
$$\forall s_y : \forall s_z : \forall S : SA(q_s, y; [s_y]) \wedge SA(q_s, z; [s_z]) \wedge R_\star(s_y, s_z; S) \Rightarrow SA(q_t, x; S)$$

where we assume that we have a relation $R_\star$ for each arithmetic operation $\star$. The first conjunct expresses that for all variables v and signs S, if the variable is different than x and at state q_s it has sign S, then it will have the same sign at state q_t. The second conjunct states that for all possible values S, s_y and s_z, if at state q_s the signs of variables y and z are s_y and s_z, respectively, and the sign of the result of evaluating the arithmetic operation $\star$ is S, then at state q_t variable x will have sign S. Similarly whenever $q_s \xrightarrow{e} q_t$ or $q_s \xrightarrow{skip} q_t$ in the program graph we generate the clause

$$\forall v : \forall S : SA(q_s, v; S) \Rightarrow SA(q_t, v; S)$$

The clause simply propagates the signs of all variables along the edge of the program graph, without altering it.

In the next section we show an alternative specification of the detection of signs analysis using function symbols. Notice, that since the analysis is defined over a powerset domain, it could also be expressed in ALFP or Datalog.

3.5 Extension with monotone functions

In this section we present an extension of LLFP that allows function terms as arguments of relations. For convenience we refer to the extension as LLFP. Since functions over the universe $\mathcal{U}$ can be represented as relations, we do not consider them here. Instead, we focus on functions over a complete lattice $[\![f]\!] : \mathcal{L}^k \to \mathcal{L}$, and we restrict our attention to monotone functions only. Recall that a function $[\![f]\!] : \mathcal{L}_1 \to \mathcal{L}_2$ between partially ordered sets $\mathcal{L}_1 = (\mathcal{L}_1, \sqsubseteq_1)$ and $\mathcal{L}_2 = (\mathcal{L}_2, \sqsubseteq_2)$ is monotone if

$$\forall l, l' \in \mathcal{L}_1 : l \sqsubseteq_1 l' \Rightarrow [\![f]\!](l) \sqsubseteq_2 [\![f]\!](l')$$

The following definition introduces the syntax of LLFP.

Definition 3.12 Given fixed countable and pairwise disjoint sets $\mathcal{X}$ and $\mathcal{Y}$ of variables, a non-empty and finite universe $\mathcal{U}$, finite alphabets $\mathcal{R}$ and $\mathcal{F}$ of predicate and function symbols, respectively, we define the set of LLFP formulae (or clause sequences), cls, together with clauses, cl, preconditions, pre, terms u and lattice terms V and V' by the grammar:

$$
\begin{array}{lll}
u & ::= & x \mid a \\
V & ::= & Y \mid [u] \\
V' & ::= & V \mid f(\vec{V'}) \\
pre & ::= & R(\vec{u}; V) \mid \neg R(\vec{u}; V) \mid Y(u) \mid pre_1 \wedge pre_2 \mid pre_1 \vee pre_2 \\
 & \mid & \exists x : pre \mid \exists Y : pre \\
cl & ::= & R(\vec{u}; V') \mid \mathbf{1} \mid cl_1 \wedge cl_2 \mid pre \Rightarrow cl \mid \forall x : cl \mid \forall Y : cl \\
cls & ::= & cl_1, \ldots, cl_s
\end{array}
$$

Here $x \in \mathcal{X}$, $a \in \mathcal{U}$, $Y \in \mathcal{Y}$, $R \in \mathcal{R}$, $f \in \mathcal{F}$, and $s \geq 1$. Furthermore, $\vec{u}$ and $\vec{V'}$ abbreviate tuples $(u_1, \ldots, u_k)$ and $(V'_1, \ldots, V'_k)$ for some $k \geq 0$, respectively.

Comparing to the Definition 3.1 we added a set $\mathcal{F}$ of function symbols. Furthermore, we extended syntax of terms with function terms over a complete lattice; denoted by $f(\vec{V'})$. Note that we allow function terms only as arguments of assertions.

In order to give a semantics of the logic, in addition to the interpretations ϱ and ς of predicate symbols and variables from Section 3.1, we introduce an interpretation of function symbols ζ. The interpretation is defined as follows

$$\zeta : \prod_k \mathcal{F}_{/k} \to \mathcal{L}^k \to \mathcal{L}$$

where $\mathcal{F}_{/k}$ is a set of function symbols of arity k. The set $\mathcal{F}$ is then defined as a disjoint union of $\mathcal{F}_{/k}$; namely $\mathcal{F} = \biguplus_k \mathcal{F}_{/k}$.

The interpretation of variables from $\mathcal{X}$ is given by $[\![x]\!](\zeta, \varsigma) = \varsigma(x)$, where $\varsigma(x)$ is the element from $\mathcal{U}$ bound to $x \in \mathcal{X}$. The interpretation of variables from $\mathcal{Y}$ is given by $[\![Y]\!](\zeta, \varsigma) = \varsigma(Y)$, where $\varsigma(Y)$ is the element from $\mathcal{L}_{\neq\perp} = \mathcal{L} \setminus \{\perp\}$ bound to $Y \in \mathcal{Y}$. In order to give the interpretation of $[u]$, we again make use of the function $\beta : \mathcal{U} \to \mathcal{L}$. The interpretation is given by $\varsigma([u]) = \beta(\varsigma(u))$. The interpretation of function terms is defined as $[\![f(\vec{V'})]\!](\zeta, \varsigma) = \zeta(f)([\![\vec{V'}]\!](\zeta, \varsigma))$. As already mentioned, we restrict our attention to the monotone functions over the complete lattice only. The interpretation of terms is generalized to sequences $\vec{u}$ of terms in a point-wise manner by taking $\varsigma(a) = a$ for all $a \in \mathcal{U}$, thus $\varsigma(u_1, \ldots, u_k) = (\varsigma(u_1), \ldots, \varsigma(u_k))$. The interpretation of lattice terms V' is generalized to sequences $\vec{V'}$ of lattice terms in the similar way.

The satisfaction relations for preconditions pre, clauses cl, and clause sequences cls are denoted by:

$$(\varrho, \varsigma) \models_\beta pre, \quad (\varrho, \zeta, \varsigma) \models_\beta cl \quad \text{and } (\varrho, \zeta, \varsigma) \models_\beta cls$$

The formal definition is given in Table 3.4.

(ϱ, ς)	$\models_\beta$	$R(\vec{u}; V)$	<u>`iff`</u>	$\varrho(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(V)$
(ϱ, ς)	$\models_\beta$	$\neg R(\vec{u}; V)$	<u>`iff`</u>	$\complement(\varrho(R)(\varsigma(\vec{u}))) \sqsupseteq \varsigma(V)$
(ϱ, ς)	$\models_\beta$	$Y(u)$	<u>`iff`</u>	$\beta(\varsigma(u)) \sqsubseteq \varsigma(Y)$
(ϱ, ς)	$\models_\beta$	$pre_1 \wedge pre_2$	<u>`iff`</u>	$(\varrho, \varsigma) \models_\beta pre_1$ and $(\varrho, \varsigma) \models_\beta pre_2$
(ϱ, ς)	$\models_\beta$	$pre_1 \vee pre_2$	<u>`iff`</u>	$(\varrho, \varsigma) \models_\beta pre_1$ or $(\varrho, \varsigma) \models_\beta pre_2$
(ϱ, ς)	$\models_\beta$	$\exists x : pre$	<u>`iff`</u>	$(\varrho, \varsigma[x \mapsto a]) \models_\beta pre$ for some $a \in \mathcal{U}$
(ϱ, ς)	$\models_\beta$	$\exists Y : pre$	<u>`iff`</u>	$(\varrho, \varsigma[Y \mapsto l]) \models_\beta pre$ for some $l \in \mathcal{L}_{\neq\bot}$
$(\varrho, \zeta, \varsigma)$	$\models_\beta$	$R(\vec{u}; V')$	<u>`iff`</u>	$\varrho(R)([\![\vec{u}]\!](\zeta, \varsigma)) \sqsupseteq [\![V']\!](\zeta, \varsigma)$
$(\varrho, \zeta, \varsigma)$	$\models_\beta$	$\mathbf{1}$	<u>`iff`</u>	`true`
$(\varrho, \zeta, \varsigma)$	$\models_\beta$	$cl_1 \wedge cl_2$	<u>`iff`</u>	$(\varrho, \zeta, \varsigma) \models_\beta cl_1$ and $(\varrho, \zeta, \varsigma) \models_\beta cl_2$
$(\varrho, \zeta, \varsigma)$	$\models_\beta$	$pre \Rightarrow cl$	<u>`iff`</u>	$(\varrho, \zeta, \varsigma) \models_\beta cl$ whenever $(\varrho, \varsigma) \models_\beta pre$
$(\varrho, \zeta, \varsigma)$	$\models_\beta$	$\forall x : cl$	<u>`iff`</u>	$(\varrho, \zeta, \varsigma[x \mapsto a]) \models_\beta cl$ for all $a \in \mathcal{U}$
$(\varrho, \zeta, \varsigma)$	$\models_\beta$	$\forall Y : cl$	<u>`iff`</u>	$(\varrho, \zeta, \varsigma[Y \mapsto l]) \models_\beta cl$ for all $l \in \mathcal{L}_{\neq\bot}$
$(\varrho, \zeta, \varsigma)$	$\models_\beta$	$cl_1, \cdots, cl_s$	<u>`iff`</u>	$(\varrho, \zeta, \varsigma) \models_\beta cl_i$ for all $i, 1 \leq i \leq s$

Table 3.4: Semantics of LLFP.

Now we establish a Moore family result for the logic extended with the function terms.

Proposition 3.13 *Assume cls is a stratified LLFP clause sequence, ς_0 and ζ_0 are interpretations of free variables and function symbols in cls, respectively. Furthermore, ϱ_0 is an interpretation of all relations of rank 0. Then*

$$\{\varrho \mid (\varrho, \zeta_0, \varsigma_0) \models_\beta cls \wedge \forall R : rank(R) = 0 \Rightarrow \varrho_0(R) \sqsubseteq \varrho(R)\}$$

is a Moore family.

Proof. See Appendix A.7.

Now, let us present the LLFP specification of the detection of signs analysis in the extension of the logic using function terms over the underlying complete lattice.

Example 9 *Analogously to Example 8 we assume that we have a program graph with three kinds of actions: $x := y \star z$, e, and skip. The analysis is defined by the predicate SA, and whenever we have $q_s \xrightarrow{x:=y\star z} q_t$ in the program graph we generate the clauses*

$$\forall v : \forall S : SA(q_s, v; S) \wedge v \neq x \Rightarrow SA(q_t, v; S) \wedge$$
$$\forall S_y : \forall S_z : SA(q_s, y; S_y) \wedge SA(q_s, z; S_z) \Rightarrow SA(q_t, x; f_\star(S_y, S_z))$$

where we assume that we have a function $f_\star$ for each arithmetic operation $\star$. The first conjunct expresses that for all variables v and signs S, if the variable is different than x and at state q_s it has possible signs S, then it will have the same signs at state q_t. The second conjunct states that for all possible values S_y and S_z, if at state q_s the sets of possible signs of variables y and z are S_y and S_z, respectively, then at state q_t the set of possible signs of variable x is updated with the set being the result of evaluating the arithmetic operation $\star$. Notice that in contrast to Example 8 the variables S_y and S_z belong to $\mathcal{Y}$, hence they range over sets of signs, not a single sign. Furthermore, the transfer function is captured by $f_\star$ instead of predicate $R_\star$. Similarly whenever $q_s \xrightarrow{e} q_t$ or $q_s \xrightarrow{skip} q_t$ in the program graph we generate the clause

$$\forall v : \forall S : SA(q_s, v; S) \Rightarrow SA(q_t, v; S)$$

The clause simply propagates the signs of all variables along the edge of the program graph, without altering it, and is exactly as in the Example 8.

CHAPTER 4

Layered Fixed Point Logic

In this chapter we present a logic for the specification of static analysis problems that goes beyond the logics traditionally used. Its most prominent feature is the direct support for both inductive computations of behaviors as well as co-inductive specifications of properties. Two main theoretical contributions are a Moore Family result and a parametrized worst case time complexity result.

The chapter is organized as follows. In Section 4.1 we define the syntax and semantics of Layered Fixed Point Logic. In Section 4.2 we establish a Moore Family result and estimate the worst case time complexity. We continue in Section 4.3 with an application to the Constraint Satisfaction Problem.

4.1 Syntax and Semantics

In this section, we introduce Layered Fixed Point Logic (abbreviated LFP). The LFP formulae are made up of layers. Each layer can either be a *define* formula which corresponds to the inductive definition, or a *constrain* formula corresponding to the co-inductive specification. The following definition introduces the syntax of LFP.

Definition 4.1 Given a fixed countable set $\mathcal{X}$ of variables, a non-empty universe $\mathcal{U}$, a finite set of function symbols $\mathcal{F}$, and a finite alphabet $\mathcal{R}$ of predicate symbols, we define the set of LFP formulae, *cls*, together with clauses, *cl*, conditions, *cond*, constrains, *con*, definitions, *def*, and terms u by the grammar:

$$
\begin{array}{lll}
u & ::= & x \mid f(\vec{u}) \\
\mathit{cond} & ::= & R(\vec{x}) \mid \neg R(\vec{x}) \mid \mathit{cond}_1 \wedge \mathit{cond}_2 \mid \mathit{cond}_1 \vee \mathit{cond}_2 \\
 & \mid & \exists x : \mathit{cond} \mid \forall x : \mathit{cond} \mid \mathit{true} \mid \mathit{false} \\
\mathit{def} & ::= & \mathit{cond} \Rightarrow R(\vec{u}) \mid \forall x : \mathit{def} \mid \mathit{def}_1 \wedge \mathit{def}_2 \\
\mathit{con} & ::= & R(\vec{u}) \Rightarrow \mathit{cond} \mid \forall x : \mathit{con} \mid \mathit{con}_1 \wedge \mathit{con}_2 \\
\mathit{cl}_i & ::= & \mathit{define}(\mathit{def}) \mid \mathit{constrain}(\mathit{con}) \\
\mathit{cls} & ::= & \mathit{cl}_1, \ldots, \mathit{cl}_s
\end{array}
$$

Here $x \in \mathcal{X}$, $R \in \mathcal{R}$, $f \in \mathcal{F}$ and $1 \leq i \leq s$. We say that s is the order of the LFP formula $cl_1, \ldots, cl_s$.

We allow to write $R(\vec{u})$ for $\mathit{true} \Rightarrow R(\vec{u})$, $\neg R(\vec{u})$ for $R(\vec{u}) \Rightarrow \mathit{false}$ and we abbreviate zero-arity functions $f()$ as $f \in \mathcal{U}$. Occurrences of $R(\vec{x})$ and $\neg R(\vec{x})$ in conditions are called positive and negative queries, respectively. Occurrences of $R(\vec{u})$ on the right hand side of the implication in define formulas are called defined occurrences. Occurrences of $R(\vec{u})$ on the left hand side of the implication in constrain formulas are called constrained occurrences. Defined and constrained occurrences are jointly called assertions. In the following we refer to ALFP relations interchangeably as relations or predicates.

In order to ensure desirable properties in the presence of negation, we impose a notion of *stratification* similar to the one in ALFP and LLFP.

Definition 4.2 The formula $cl_1, \ldots, cl_s$ is stratified if for all $i = 1, \ldots, s$ the following properties hold:

- Relations asserted in cl_i must not be asserted in $cl_{i+1}, \ldots, cl_s$
- Relations occurring in positive queries in cl_i must not be asserted in $cl_{i+1}, \ldots, cl_s$
- Relations occurring in negative queries in cl_i must not be asserted in $cl_i, \ldots, cl_s$

The function rank : $\mathcal{R} \to \{0, \ldots, s\}$ is then uniquely defined as

$$
\mathrm{rank}(R) = \begin{cases} i & \text{if } R \text{ is asserted in } cl_i, \\ 0 & \text{otherwise.} \end{cases}
$$

Intuitively, the definition states that every relation can be asserted in at most one clause. Furthermore is ensures that a negative query is not performed until the predicate has been fully asserted. The following example illustrates the use of negation in the LFP formulae.

Example 10 *Using the notion of stratification we can define equality eq and non-equality neq predicates as follows*

$$\begin{aligned} &\mathit{define}(\forall x : \mathit{true} \Rightarrow \mathit{eq}(x,x)),\\ &\mathit{define}(\forall x : \forall y : \neg \mathit{eq}(x,y) \Rightarrow \mathit{neq}(x,y)) \end{aligned}$$

According to Definition 4.2 the formula is stratified, since predicate eq is negatively used only in the layer above the one that defines it. More precisely, the predicate eq is fully defined before it is negatively queried in the clause asserting predicate neq.

Alternatively, we can use a greatest fixed point specification (using constrain clause) to define equality and non-equality predicates

$$\begin{aligned} &\mathit{constrain}(\forall x : \mathit{neq}(x,x) \Rightarrow \mathit{false}),\\ &\mathit{constrain}(\forall x : \forall y : \neg \mathit{neq}(x,y) \Rightarrow \mathit{eq}(x,y)) \end{aligned}$$

Again the formula is stratified, since predicate neq is negatively used only in the layer above the one that asserts it.

To specify the semantics of LFP we introduce the interpretations ϱ, ζ and ς of predicate symbols, function symbols and variables, respectively. Formally we have

$$\begin{array}{ll} \varrho : & \prod_k \mathcal{R}_{/k} \to \mathcal{P}(\mathcal{U}^k) \\ \zeta : & \prod_k \mathcal{F}_{/k} \to \mathcal{U}^k \to \mathcal{U} \\ \varsigma : & \mathcal{X} \to \mathcal{U} \end{array}$$

In the above $\mathcal{R}_{/k}$ stands for a set of predicate symbols of arity k, then $\mathcal{R}$ is a disjoint union of $\mathcal{R}_{/k}$, hence $\mathcal{R} = \biguplus_k \mathcal{R}_{/k}$. Similarity $\mathcal{F}_{/k}$ is a set of function symbols of arity k and $\mathcal{F} = \biguplus_k \mathcal{F}_{/k}$. The interpretation of variables is given by $[\![x]\!](\zeta,\varsigma) = \varsigma(x)$, where $\varsigma(x)$ is the element from $\mathcal{U}$ bound to $x \in \mathcal{X}$. Furthermore, the interpretation of function terms is defined as $[\![f(\vec{u})]\!](\zeta,\varsigma) = \zeta(f)([\![\vec{u}]\!](\zeta,\varsigma))$. It is generalized to sequences $\vec{u}$ of terms in a point-wise manner by taking $[\![a]\!](\zeta,\varsigma) = a$ for all $a \in \mathcal{U}$, and $[\![(u_1,\ldots,u_k)]\!](\zeta,\varsigma) = ([\![u_1]\!](\zeta,\varsigma),\ldots,[\![u_k]\!](\zeta,\varsigma))$.

The satisfaction relations for conditions *cond*, definitions *def* and constrains *con* are denoted by:

$$(\varrho,\varsigma) \models \mathit{cond}, \quad (\varrho,\zeta,\varsigma) \models \mathit{def} \quad \text{and } (\varrho,\zeta,\varsigma) \models \mathit{con}$$

The formal definition is given in Table 4.1; here $\varsigma[x \mapsto a]$ stands for the mapping that is as ς except that x is mapped to a.

$$
\begin{array}{lcllll}
(\varrho,\varsigma) & \models & R(\vec{x}) & \underline{\texttt{iff}} & [\![\vec{x}]\!]([\,],\varsigma) \in \varrho(R) \\
(\varrho,\varsigma) & \models & \neg R(\vec{x}) & \underline{\texttt{iff}} & [\![\vec{x}]\!]([\,],\varsigma) \notin \varrho(R) \\
(\varrho,\varsigma) & \models & cond_1 \wedge cond_2 & \underline{\texttt{iff}} & (\varrho,\varsigma) \models cond_1 \text{ and } (\varrho,\varsigma) \models cond_2 \\
(\varrho,\varsigma) & \models & cond_1 \vee cond_2 & \underline{\texttt{iff}} & (\varrho,\varsigma) \models cond_1 \text{ or } (\varrho,\varsigma) \models cond_2 \\
(\varrho,\varsigma) & \models & \exists x : cond & \underline{\texttt{iff}} & (\varrho,\varsigma[x \mapsto a]) \models cond \text{ for some } a \in \mathcal{U} \\
(\varrho,\varsigma) & \models & \forall x : cond & \underline{\texttt{iff}} & (\varrho,\varsigma[x \mapsto a]) \models cond \text{ for all } a \in \mathcal{U} \\
(\varrho,\varsigma) & \models & true & \underline{\texttt{iff}} & \mathsf{true} \\
(\varrho,\varsigma) & \models & false & \underline{\texttt{iff}} & \mathsf{false} \\
\\
(\varrho,\zeta,\varsigma) & \models & R(\vec{u}) & \underline{\texttt{iff}} & [\![\vec{u}]\!](\zeta,\varsigma) \in \varrho(R) \\
(\varrho,\zeta,\varsigma) & \models & def_1 \wedge def_2 & \underline{\texttt{iff}} & (\varrho,\zeta,\varsigma) \models def_1 \text{ and } (\varrho,\zeta,\varsigma) \models def_2 \\
(\varrho,\zeta,\varsigma) & \models & cond \Rightarrow R(\vec{u}) & \underline{\texttt{iff}} & (\varrho,\zeta,\varsigma) \models R(\vec{u}) \text{ whenever } (\varrho,\varsigma) \models cond \\
(\varrho,\zeta,\varsigma) & \models & \forall x : def & \underline{\texttt{iff}} & (\varrho,\zeta,\varsigma[x \mapsto a]) \models def \text{ for all } a \in \mathcal{U} \\
\\
(\varrho,\zeta,\varsigma) & \models & R(\vec{u}) & \underline{\texttt{iff}} & [\![\vec{u}]\!](\zeta,\varsigma) \in \varrho(R) \\
(\varrho,\zeta,\varsigma) & \models & con_1 \wedge con_2 & \underline{\texttt{iff}} & (\varrho,\zeta,\varsigma) \models con_1 \text{ and } (\varrho,\zeta,\varsigma) \models con_2 \\
(\varrho,\zeta,\varsigma) & \models & R(\vec{u}) \Rightarrow cond & \underline{\texttt{iff}} & (\varrho,\varsigma) \models cond \text{ whenever } (\varrho,\zeta,\varsigma) \models R(\vec{u}) \\
(\varrho,\zeta,\varsigma) & \models & \forall x : con & \underline{\texttt{iff}} & (\varrho,\zeta,\varsigma[x \mapsto a]) \models con \text{ for all } a \in \mathcal{U} \\
\\
(\varrho,\zeta,\varsigma) & \models & cl_1, \ldots, cl_s & \underline{\texttt{iff}} & (\varrho,\zeta,\varsigma) \models cl_i \text{ for all } 1 \leq i \leq s
\end{array}
$$

Table 4.1: Semantics of LFP.

4.2 Optimal Solutions

Moore Family First we establish a Moore family result for LFP, which guarantees that there always is a unique best solution for LFP formulae. A Moore family was formally defined in Section 2.1.

Let $\Delta = \{\varrho \mid \varrho : \prod_k \mathcal{R}_{/k} \to \mathcal{P}(\mathcal{U}^k)\}$ denote the set of interpretations ϱ of predicate symbols in $\mathcal{R}$ over $\mathcal{U}$. We define a lexicographical ordering $\sqsubseteq$ defined by $\varrho_1 \sqsubseteq \varrho_2$ if and only if there is some $0 \leq j \leq s$, where s is the order of the formula (number of layers), such that the following properties hold:

(a) $\varrho_1(R) = \varrho_2(R)$ for all $R \in \mathcal{R}$ with $\text{rank}(R) < j$,

(b) $\varrho_1(R) \subseteq \varrho_2(R)$ for all $R \in \mathcal{R}$ with $\text{rank}(R) = j$ and either $j = 0$ or R is a *defined* relation,

(c) $\varrho_1(R) \supseteq \varrho_2(R)$ for all $R \in \mathcal{R}$ with $\text{rank}(R) = j$ and R is a *constrained* relation,

(d) either $j = s$ or $\varrho_1(R) \neq \varrho_2(R)$ for some relation $R \in \mathcal{R}$ with $\text{rank}(R) = j$.

Notice that in the case $s = 1$ the ordering $\sqsubseteq$ coincides with the ordering $\subseteq$ for *defined* relations and with the ordering $\supseteq$ for the *constrained* relations. The use of the dual orderings for *defined* and *constrained* relations stems from the fact that we are interested in the smallest solution for the *defined* relations and the greatest solution for the *constrained* ones. Notice also that in the case of *defined* relations the definition of the partial order is equivalent to the ones for ALFP and LLFP. Intuitively, the lexicographical ordering $\sqsubseteq$ orders the relations layer by layer starting with the layer 0. It is essentially analogous to the alphabetical ordering on strings, which is based on the alphabetical order of their characters.

Lemma 4.3 $\sqsubseteq$ *defines a partial order.*

Proof. See Appendix A.8. □

Lemma 4.4 $(\Delta, \sqsubseteq)$ *is a complete lattice with the greatest lower bound given by*

$$(\bigsqcap M)(R) = \begin{cases} \bigcap\{\varrho(R) \mid \varrho \in M_{rank(R)}\} & \text{if } rank(R) = 0 \text{ or } R \text{ is} \\ & \text{a defined relation.} \\ \bigcup\{\varrho(R) \mid \varrho \in M_{rank(R)}\} & \text{if } R \text{ is a constrained relation.} \end{cases}$$

where

$$M_j = \{\varrho \in M \mid \forall R' : rank(R') < j \Rightarrow (\bigsqcap M)(R') = \varrho(R')\}$$

Proof. See Appendix A.9. □

Note that $\bigsqcap M$ is well defined by induction on j observing that $M_0 = M$ and $M_j \subseteq M_{j-1}$.

Proposition 4.5 *Assume cls is a stratified LFP formula, ς_0 and ζ_0 are interpretations of the free variables and function symbols in cls, respectively. Furthermore, ϱ_0 is an interpretation of all relations of rank 0. Then $\{\varrho \mid (\varrho, \zeta_0, \varsigma_0) \models cls \wedge \forall R : rank(R) = 0 \Rightarrow \varrho(R) \supseteq \varrho_0(R)\}$ is a Moore family.*

Proof. See Appendix A.10. □

Complexity The least model for LFP formulae guaranteed by Proposition 4.5 can be computed efficiently as summarized in the following result.

Proposition 4.6 *For a finite universe $\mathcal{U}$, the best solution ϱ such that $\varrho_0 \sqsubseteq \varrho$ of a LFP formula $cl_1, \ldots, cl_s$ (w.r.t. an interpretation of the constant symbols) can be computed in time*

$$\mathcal{O}(|\varrho_0| + \sum_{1 \leq i \leq s} |cl_i||\mathcal{U}|^{k_i})$$

where k_i is the maximal nesting depth of quantifiers in the cl_i and $|\varrho_0|$ is the sum of cardinalities of predicates $\varrho_0(R)$ of rank 0. We also assume unit time hash table operations (as in [39]).

Proof. See Appendix A.11. □

For *define* clauses a straightforward method that achieves the above complexity proceeds by instantiating all variables occurring in the input formula in all possible ways. The resulting formula has no free variables thus it can be solved by classical solvers for alternation-free Boolean equation systems [25] in linear time.

In case of *constrain* clauses we first dualize the problem by transforming the co-inductive specification into the inductive one. The transformation increases the size of the input formula by a constant factor. Thereafter, we proceed in the same way as for the define clauses.

In addition we need to take into account the number of known facts, which equals to the cardinality of all predicates of rank 0. As a result we get the complexity from Proposition 4.6.

4.3 Application to Constraint Satisfaction

Arc consistency is a basic technique for solving Constraint Satisfaction Problems (CSP) and has various applications within e.g. Artificial Intelligence. Formally a CSP [38, 60] problem can be defined as follows.

Definition 4.7 A Constraint Satisfaction Problem (N, D, C) consists of a finite set of variables $N = \{x_1, \ldots, x_n\}$, a set of finite non-empty domains $D = \{D_1, \ldots, D_n\}$, where x_i ranges over D_i, and a set of constraints $C \subseteq \{c_{ij} \mid i, j \in N\}$, where each constraint c_{ij} is a binary relation between variables x_i and x_j.

For simplicity we consider binary constraints only. Furthermore, we can represent a CSP problem as a directed graph in the following way.

Definition 4.8 A constraint graph of a CSP problem (N, D, C) is a directed graph $G = (V, E)$ where $V = N$ and $E = \{(x_i, x_j) \mid c_{ij} \in C\}$.

Thus vertices of the graph correspond to the variables and an edge in the graph between nodes x_i and x_j corresponds to the constraint $c_{ij} \in C$.

The arc consistency problem is formally stated in the following definition.

Definition 4.9 Given a CSP (N, D, C), an arc (x_i, x_j) of its constraint graph is arc consistent if and only if $\forall x \in D_i$, there exists $y \in D_j$ such that $c_{ij}(x, y)$ holds, as well as $\forall y \in D_j$, there exists $x \in D_i$ such that $c_{ij}(x, y)$ holds. A CSP (N, D, C) is arc consistent if and only if each arc in its constraint graph is arc consistent.

The basic and widely used arc consistency algorithm is the AC-3 algorithm proposed in 1977 by Mackworth [38]. The complexity of the algorithm is $O(ed^3)$,

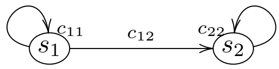


Figure 4.1: Arc consistency.

where e is the number of constraints and d the size of the largest domain. The algorithm is used in many constrains solvers due to its simplicity and fairly good efficiency [57].

Now we show the LFP specification of the arc consistency problem. A domain of a variable x_i is represented as a unary relation D_i, and for each constraint $c_{ij} \in C$ we have a binary relation $C_{ij} \subseteq D_i \times D_j$. Then we obtain

$$\mathit{constrain}\left(\bigwedge_{c_{ij} \in C} \begin{array}{l} (\forall x : D_i(x) \Rightarrow \exists y : D_j(y) \wedge C_{ij}(x,y)) \wedge \\ (\forall y : D_j(y) \Rightarrow \exists x : D_i(x) \wedge C_{ij}(x,y)) \end{array} \right)$$

which exactly captures the conditions from Definition 4.9.

According to the Proposition 4.6 the above specification gives rise to the worst-case complexity $\mathcal{O}(ed^2)$. The original AC-3 algorithm was optimized in [60] where it was shown that it achieves the worst-case optimal time complexity of $\mathcal{O}(ed^2)$. Hence LFP specification has the same worst-case time complexity as the improved version of the AC-3 algorithm.

Example 11 *As an example let us consider the following problem. Assume we have two processes P_1 and P_2 that need to be finished before 8 time units have elapsed. The process P_1 is required to run for 3 or 4 time units, the process P_2 is required to run for precisely 2 time units, and P_2 should start at the exact moment when P_1 finishes.*

The problem can be defined as an instance of CSP (N, D, C) where $N = \{s_1, s_2\}$ denoting the starting times of the corresponding process. Since both processes need to be completed before 8 time units have elapsed we have $D_1 = D_2 = \{0, \ldots, 8\}$. Moreover, we have the following constrains $C = \{c_{12} = (3 \leq s_2 - s_1 \leq 4), c_{11} = (0 \leq s_1 \leq 4), c_{22} = (0 \leq s_2 \leq 6)\}$. We can represent the above CSP problem as a constraint graph depicted in Figure 4.1. Furthermore it can be specified as the following LFP formulae

$$\begin{array}{l} \mathit{define}\left(\bigwedge_{0 \leq x \leq 4} C_1(x) \wedge \bigwedge_{0 \leq y \leq 6} C_2(y) \wedge \bigwedge_{3 \leq z \leq 4} C_{12}(z) \right), \\ \mathit{constrain}\left(\begin{array}{l} (\forall x : D_1(x) \Rightarrow \exists y : D_2(y) \wedge C_{12}(y - x)) \wedge \\ (\forall y : D_2(y) \Rightarrow \exists x : D_1(x) \wedge C_{12}(y - x)) \end{array} \right) \end{array}$$

where we write $y - x$ for a function $f_{sub}(y, x)$.

Chapter 5

Solvers

In this chapter we describe the design and implementation of the solvers for ALFP, LLFP, and LFP. In Section 5.1 we present an abstract algorithm that captures similarities and gives an overall structure for the algorithms presented later in this chapter. In Section 5.2 we introduce a differential worklist algorithm for ALFP, originally developed in [44], that is based on a representation of relations as prefix trees [44]. Section 5.3 presents another algorithm for ALFP, being a continuation passing style one that is based on a BDD representation of relations [12]. BDDs, originally designed for hardware verification, have already been used in a number of program analyses [59, 9] and proven to be very efficient. We introduce a differential worklist algorithm for LLFP in Section 5.4. The algorithm is fairly similar to the one presented in Section 5.2; thus our main focus in that section is to emphasize the distinguishing features of the LLFP algorithm. Finally, in Section 5.5, we report on a BDD-based algorithm for LFP, which extends the algorithm from Section 5.3 with direct support for co-inductive specifications. The implementation of the solving algorithms described in this chapter was released under an open-source license and is available at https://github.com/piotrfilipiuk/succinct-solvers.

5.1 Abstract algorithm

Now, we present an abstract algorithm for solving clause sequences, which forms the basis for the concrete algorithms presented in the following sections. Although the underlying data structures of the concrete algorithms are very different they share the same overall structure that is captured by the abstract algorithm. We leave a detailed discussion of the concrete algorithms to the next sections.

The abstract algorithm operates with (intermediate) representations of the two interpretations ς and ϱ of the semantics; we shall call them `env` and `result`, respectively, in the following. The `result` is an imperative data structure that is updated as we progress. The data structure `env` is supplied as a parameter to the functions of the algorithms.

We have one function for each of the three syntactic categories. The function SOLVE takes a *clause sequence* as input and calls the function EXECUTE on each of the individual clauses. The pseudo code is as follows

$$\textsc{solve}(cl_1, \ldots, cl_s) \quad = \quad \textsc{execute}(cl_1)[\,]; \ldots; \textsc{execute}(cl_s)[\,]$$

where we write [ ] for the empty environment reflecting that we have no free variables in clause sequences.

The function EXECUTE takes a *clause cl* as a parameter and a representation `env` of the interpretation of the variables. We have one case for each of the forms of *cl*:

$$\begin{array}{rcl}
\textsc{execute}(R(u_1, \ldots, u_k))\texttt{env} & = & \ldots \\
\textsc{execute}(\mathbf{1})\texttt{env} & = & () \\
\textsc{execute}(cl_1 \wedge cl_2)\texttt{env} & = & \textsc{execute}(cl_1)\texttt{env}; \textsc{execute}(cl_2)\texttt{env} \\
\textsc{execute}(pre \Rightarrow cl)\texttt{env} & = & \textsc{check}(pre, \textsc{execute}(cl))\texttt{env} \\
\textsc{execute}(\forall x : cl)\texttt{env} & = & \textbf{let } \texttt{env}' = \ldots \textbf{ in } \textsc{execute}(cl)\texttt{env}'
\end{array}$$

In the case of assertions the details depend on the actual algorithm and we return to those later. The case of conjunction is straightforward as we have to inspect both clauses. In the case of implication we make use of the function CHECK that in addition to the precondition and the environment also takes the continuation EXECUTE(cl) as an argument. In the case of universal quantification we perform a recursive call using an updated environment, the details of which depend on the actual algorithm.

The function CHECK takes a *precondition*, a continuation, and an environment as parameters. The treatment of queries depends on the actual algorithm and so does the treatment of disjunction and universal quantification; except from the fact that the overall structure is:

$$\begin{array}{rcl}
\textsc{check}(R(u_1,\ldots,u_k), next)\texttt{env} & = & \ldots \\
\textsc{check}(\neg R(u_1,\ldots,u_k), next)\texttt{env} & = & \ldots \\
\textsc{check}(pre_1 \wedge pre_2, next)\texttt{env} & = & \textsc{check}(pre_1, \textsc{check}(pre_2, next))\texttt{env} \\
\textsc{check}(pre_1 \vee pre_2, next)\texttt{env} & = & \ldots \\
\textsc{check}(\exists x : cl, next)\texttt{env} & = & \textbf{let } next' = next \circ \ldots \\
 & & \textbf{let } \texttt{env}' = \ldots \\
 & & \textbf{in } \textsc{check}(cl, next')\texttt{env}' \\
\textsc{check}(\forall x : cl, next)\texttt{env} & = & \ldots
\end{array}$$

For conjunction we exploit a continuation passing programming style and for existential quantification we perform a recursive call using an updated environment and an updated continuation, the details of which depend on the actual algorithm.

In the following sections we give more details of the data structures used by the actual (concrete) algorithms and the missing cases in the above definitions.

5.2 Differential algorithm for ALFP

In this section we present the main data structures and the details of the differential worklist algorithm for ALFP developed by Nielson et al. [44, 43]. The algorithm computes the relations in increasing order on their rank, and therefore negations present no obstacles. It combines the top-down solving approach of Le Charlier and van Hentenryck [16] with the propagation of differences [26], an optimization technique for distributive frameworks that is also known in the area of deductive databases [5] or as reduction of strength transformations for program optimization [45]. As mentioned above the main data structures are `env` and `result` representing the (partial) interpretation of variables and predicates, respectively.

Here `env` is implemented as a map from variables to their possible values. Thus, for a given variable it returns either *None*, which means that the variable is undefined or *Some*(a), which means that the variable is bound to $a \in \mathcal{U}$. The main operation on `env` is the function `unify`. It is given by

$$\texttt{unify}(\texttt{env}, u, a) = \begin{cases} \texttt{env} & \text{if } (u \in \mathcal{X} \wedge \texttt{env}[u] = Some(a)) \vee u = a \\ \texttt{env}[u \mapsto Some(a)] & \text{if } u \in \mathcal{X} \wedge \texttt{env}[u] = None \\ \text{fail} & \text{otherwise} \end{cases}$$

It is extended to k-tuples in a straightforward way. The function UNIFIABLE

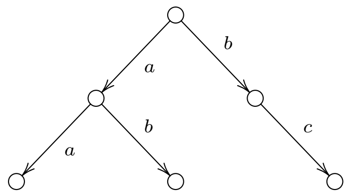


Figure 5.1: Prefix tree representing interpretation of relation R.

will, when applied to `env` and a tuple $(u_1, \ldots, u_k)$, return the subset of $\mathcal{U}^k$ for which `unify` will succeed.

The interpretation of the predicate symbols ρ is given by the global data structure `result`, which is updated incrementally during computations. It is represented as a mapping from predicate names to the prefix trees that for each predicate R record the tuples currently known to belong to R. The prefix trees themselves are implemented as arbitrarily branching trees and are defined using the following data structure

$$\texttt{RTrie} = \texttt{RNode}\ (\texttt{Map U RTrie})$$

where `Map k v` is a mapping (dictionary) from keys `k` to values `v`. Therefore, each node in the prefix tree contains a mapping from elements of the universe to its successor nodes (children). The terminal nodes in the tree are represented simply as nodes without successors (children), represented by the empty mapping.

As an example, consider the following interpretation ρ of a relation R

$$\rho(R) = \{(a, a), (a, b), (b, c)\}$$

The corresponding prefix tree representation is depicted in Figure 5.1. The operations on prefix trees boil down to tree traversal. For example the content of the relation is retrieved by the traversal of the prefix tree from the root to the leaves.

There are three main operations on the data structure `result`: the operation `result.`HAS checks whether a tuple of atoms from the universe is associated with a given predicate, the operation `result.`SUB returns a list of the tuples associated with a given predicate and the operation `result.`ADD adds a tuple to the interpretation of a given predicate.

Since ρ is not completely determined from the beginning, it may happen that a query $R(u_1, \ldots, u_k)$ inside a precondition fails to be satisfied at a given point

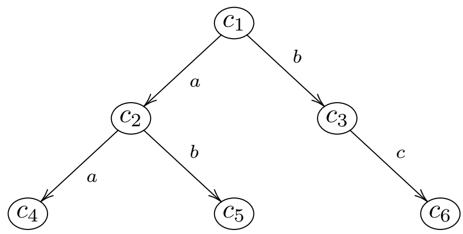


Figure 5.2: Prefix tree representing consumers of relation R.

in time, but may hold in the future when a new tuple $(a_1, \ldots, a_k)$ has been added to the interpretation of R. If we are not careful we will then lose the consequences that adding $(a_1, \ldots, a_k)$ to R will have on the contents of other predicates. This gives rise to introducing yet another global data structure `infl` that records computations that have to be resumed for the new tuples; these future computations will be called *consumers*. The `infl` data structure is also represented as a mapping from the predicate names to prefix trees that for each predicate R records consumers that have to be resumed when the interpretation of R is updated. The data structure used is defined as

$$\texttt{ITrie} = \texttt{INode cons } (\texttt{Map U ITrie})$$

In addition to the mapping into the successor nodes, each node contains a set of consumers, denoted by `cons`. The tree representation of the `ITrie` for the relation from Figure 5.1 is depicted in Figure 5.2.

There are two main operations on the data structure `infl`. The operation `infl.`REGISTER adds a new consumer for a given predicate. The other operation is `infl.`CONSUMERS, which retrieves all the consumers currently associated with a given predicate.

Let us now return to the description of the function EXECUTE for the cases that are specific for the differential algorithm for ALFP, that is, the case of assertion and the case of universal quantification. In case of assertions the algorithm is as follows

EXECUTE$(R(u_1, \ldots, u_k))$`env` =
    let ITERFUN $(a_1, \ldots, a_k)$ =
        match `result.`HAS$(R, (a_1, \ldots, a_k))$ **with**
            | *true* $\rightarrow$ ()
            | *false* $\rightarrow$
                `result.`ADD$(R, (a_1, \ldots, a_k))$
                ITER (**fun** $f \rightarrow f\ (a_1, \ldots, a_k)$) (`infl.`CONSUMERS R)

in ITER ITERFUN (UNIFIABLE(env,$(u_1, \ldots, u_k)$))

The function uses the auxiliary function ITER, which applies the function ITERFUN to each element of the list of k-tuples that can be unified with the argument $(u_1, \ldots, u_k)$. Given a tuple $(a_1, \ldots, a_k)$, the function ITERFUN adds the tuple to the interpretation of R stored in `result` if it is not already present. If the ADD operation succeeds, we first create a list of all the consumers currently registered for predicate R by calling the function `infl`.CONSUMERS. Thereafter, we resume the computations by iterating over the list of consumers and calling corresponding continuations.

In the case of universal quantification, we simply extend the environment to record that the value of the new variable is unknown and then we recurse

EXECUTE($\forall x : cl$)env = EXECUTE(cl)(env[$x \mapsto None$])

Turning to the CHECK function let us first consider the algorithm in the case of positive queries

CHECK($R(u_1, \ldots, u_k), next$)env =
  let CONSUMER $(a_1, \ldots, a_k)$ =
    match UNIFY(env, $(u_1, \ldots, u_k), (a_1, \ldots, a_k)$) **with**
      | fail → ()
      | env' → $next$ env'
  in `infl`.REGISTER(R,CONSUMER); ITER CONSUMER (`result`.SUB R)

We first ensure that the consumer is registered in `infl`, by calling function REGISTER, so that future tuples associated with R will be processed. Thereafter, the function inspects the data structure `result` to obtain the list of tuples associated with the predicate R. Then, the auxiliary function CONSUMER unifies each tuple with $(u_1, \ldots, u_k)$; and if the operation succeeds, the continuation $next$ is invoked on the updated new environment.

In the case of negated query, the algorithm is of the form

CHECK($\neg R(u_1, \ldots, u_k), next$)env =
  let ITERFUN $(a_1, \ldots, a_k)$ =
    match `result`.HAS($R, (a_1, \ldots, a_k)$) **with**
      | $true$ → ()
      | $false$ → $next$ (UNIFY(env, $(u_1, \ldots, u_k), (a_1, \ldots, a_k)$))
  in ITER ITERFUN (UNIFIABLE(env, $(u_1, \ldots, u_k)$))

The function first computes the tuples unifiable with $(u_1, \ldots, u_k)$ in the environment `env`. Then, for each tuple it checks if the tuple is already in R and if not, the tuple is unified with $(u_1, \ldots, u_k)$ to produce new environment in which the continuation *next* is evaluated.

The CHECK function for disjunction of preconditions is as follows

$\text{CHECK}(pre_1 \vee pre_2, next)$`env` =
  $\text{CHECK}(pre_1, next)$`env`; $\text{CHECK}(pre_2, next)$`env`

The function simply checks preconditions pre_1 and pre_2 respectively in the current environment `env`. In order to be efficient we use memoization; this means that if both checks yield the same bindings of variables, the second check does not need to consider the continuation, as it has already been done.

The algorithm for existential quantification checks the precondition *pre* in the environment extended with the quantified variable. The continuation that is passed is a composition of functions *next* and REMOVE x, where the function REMOVE removes the variable passed as the first argument from the environment passed as the second argument. The algorithm is as follows

$\text{CHECK}(\exists x : pre, next)$`env` =
  $\text{CHECK}(pre, next \circ (\text{REMOVE x}))$(`env`$[x \mapsto None]$)

In the case of universal quantification the function CHECK needs to inspect all atoms from the universe and find the extensions of `env` that are compatible with the precondition *pre*. In order to do that we iterate over the entire universe, successively binding the atoms to x and modifying the partial environments to be compatible with the precondition *pre*. We enumerate the universe using the auxiliary function LOOP, which is initially called with the complete list of atoms in the universe. The recursive structure of the function LOOP reflects the fact that universal quantification is a conjunction over the entire universe. The pseudo code for the case is as follows:

$\text{CHECK}(\forall x : pre, next)$`env` =
  let LOOP U' `env'` =
    match U' **with**
      | $hd :: tl \rightarrow \text{CHECK}(pre, \text{LOOP } tl)$ (`env'`$[x \mapsto Some(hd)]$)
      | $[\,] \rightarrow next$ `env'`
  in LOOP U (`env`$[x \mapsto None]$)

5.3 BDD-based algorithm for ALFP

We now turn our attention to the BDD-based algorithm for ALFP. This algorithm also makes use of the data structures `env` and `result`, but this time they are represented as binary decision diagrams, or, to be more precise, by reduced ordered binary decision diagrams (ROBDDs) [12]. The use of BDDs allows us to operate on entire relations, rather than on individual tuples (as in the differential worklist algorithm). Furthermore, the cost of the BDD operations depends on the size of the BDD and not the number of tuples in the relation; hence dense relations can be computed efficiently as long as their encoded representations are compact.

Each BDD is defined over a finite sequence of distinct domain names. The main operations on BDDs, to be used in the following, are given by means of operations on the relations they represent. Given two relations with the same domain names, the operations *union*, $\cup$, and *non-equality testing*, $\neq$, are defined as corresponding operations on the set of their tuples. The *projection* operation, π, selects the subset of domains from the relation and removes all other domains. The *select* operation, σ_b, selects all tuples from the relation for which the given condition b holds. The *complement* operation, $\complement$, on the relation R returns a new relation containing tuples that are not in R. Given two relations with pairwise disjoint domain names, the *product* operation, $\times$, is defined as a Cartesian product of their tuples. The operation $\forall_{d_i}$ is the universal quantification of variables in domain d_i. It removes tuples from the relation by universal quantification over domain d_i.

The environment `env` and the interpretation of the predicates in `result` are represented as ROBDD data structures. We need to keep track of the domain names of the BDDs so the environments and predicates will be annotated with subscript $[d_1, \ldots, d_k]$ denoting a list of pairwise disjoint domain names. In the case of environments $\texttt{env}_{[x_1,\ldots,x_n]}$ the domain names represent the variables currently in the scope.

Note, that in contrast to the differential algorithm for ALFP, in the BDD-based one, due to the use of BDDs, an environment $\texttt{env}_{[x_1,\ldots,x_n]}$ represents a set of mappings of variables to their corresponding values, not a single mapping. Consequently the BDD-based algorithm propagates sets of mappings at a time, not individual ones.

Also in the BDD algorithm we need to resume computations when the interpretation of the given predicate is updated. Therefore, we again define a data structure `infl`. It is implemented as a mapping from predicate names to consumers representing computations to be resumed. The `infl` data structure has two main operations REGISTER and RESUME for adding new consumers and invoking registered computations, respectively.

We now present the parts of the algorithm that are specific for the BDD-based algorithm for ALFP. We begin with the case of assertion for the EXECUTE function, which is defined as follows

$\textsc{execute}(R_{[d_1,\ldots,d_k]}(u_1,\ldots,u_k))\texttt{env}_{[x_1,\ldots,x_n]} =$
    for $i = 1$ to k **do**
        $\texttt{env}_{[x_1,\ldots,x_n,d_1,\ldots,d_i]} \leftarrow \sigma_{u_i=d_i}(\texttt{env}_{[x_1,\ldots,x_n,d_1,\ldots,d_{i-1}]} \times U_{[d_i]})$
    $oldR_{[d_1,\ldots,d_k]} \leftarrow \texttt{result}[R]$
    $\texttt{result}[R] \leftarrow oldR_{[d_1,\ldots,d_k]} \cup \pi_{[d_1,\ldots,d_k]}(\texttt{env}_{[x_1,\ldots,x_n,d_1,\ldots,d_k]})$
    if $oldR_{[d_1,\ldots,d_k]} \neq \texttt{result}[R]$ **then**
        $\texttt{infl}.\textsc{resume}(R)$

In the *for* loop the function incrementally builds a product of the current environment and a relation representing the universe, and simultaneously selects the tuples compatible with the arguments $(u_1,\ldots,u_k)$. Then, the resulting relation is projected to the domain names of R, and the content of R is updated with the newly derived tuples. Additionally, if the interpretation of predicate R has changed, we invoke the consumers registered for predicate R in the data structure `infl` by calling the RESUME function.

The case of universal quantification is of the following form:

$\textsc{execute}(\forall x : cl)\texttt{env}_{[x_1,\ldots,x_n]} =$
    $\textsc{execute}(cl)(\texttt{env}_{[x_1,\ldots,x_n]} \times U_{[x]})$

The function extends the current environment with a domain for the quantified variable, and then executes the clause *cl*.

Turning to the CHECK function, we first present the case for the query, which is as follows

$\textsc{check}(R_{[d_1,\ldots,d_k]}(u_1,\ldots,u_k), next)\texttt{env}_{[x_1,\ldots,x_n]}=$
    $\texttt{infl}.\textsc{register}\ R\ \textsc{consumer}$
    $\texttt{env'}_{[x_1,\ldots,x_n,d_1,\ldots,d_k]} \leftarrow \texttt{env}_{[x_1,\ldots,x_n]} \times \texttt{result}[R]$
    for $i = 1$ to k **do**
        $\texttt{env'}_{[x_1,\ldots,x_n,d_1,\ldots,d_k]} \leftarrow \sigma_{u_i=d_i}(\texttt{env'}_{[x_1,\ldots,x_n,d_1,\ldots,d_k]})$
    $\texttt{env'}_{[x_1,\ldots,x_n]} \leftarrow \pi_{[x_1,\ldots,x_n]}(\texttt{env'}_{[x_1,\ldots,x_n,d_1,\ldots,d_k]})$
    $next(\texttt{env'}_{[x_1,\ldots,x_n]})$

First, the function registers a consumer for the relation R. Then, it creates an auxiliary relation, which is a product of the relations representing the current environment and the predicate R. The *for* loop selects tuples that are compatible with the arguments $(u_1,\ldots,u_k)$ producing a new relation that is then projected to the domain names of $\texttt{env}_{[x_1,\ldots,x_n]}$. The resulting relation is then applied to continuation *next*.

The case of negated query is similar, except that the predicate is complemented first. The algorithm for this case is of the following form

$\textsc{check}(\neg R_{[d_1,\ldots,d_k]}(u_1,\ldots,u_k), next)\texttt{env}_{[x_1,\ldots,x_n]} =$

$\texttt{env'}_{[x_1,\ldots,x_n,d_1,\ldots,d_k]} \leftarrow \texttt{env}_{[x_1,\ldots,x_n]} \times (\complement\ \texttt{result}[R])$
for $i = 1$ to k **do**
$\quad \texttt{env'}_{[x_1,\ldots,x_n,d_1,\ldots,d_k]} \leftarrow \sigma_{u_i=d_i}(\texttt{env'}_{[x_1,\ldots,x_n,d_1,\ldots,d_k]})$
$\texttt{env'}_{[x_1,\ldots,x_n]} \leftarrow \pi_{[x_1,\ldots,x_n]}(\texttt{env'}_{[x_1,\ldots,x_n,d_1,\ldots,d_k]})$
$next(\texttt{env'}_{[x_1,\ldots,x_n]})$

Notice that in the case of negative queries we do not register a consumer for the relation R. This is because the stratification condition introduced in Definition 2.18 ensures that the relation is fully evaluated before it is queried negatively. Thus, there is no need to register future computations since the interpretation of R will not change.

The CHECK function for disjunction of preconditions is defined as follows

$$\begin{aligned}&\textsc{check}(pre_1 \vee pre_2, next)\texttt{env}_{[x_1,\ldots,x_n]} =\\ &\quad \textsc{check}(pre_1, \lambda\texttt{env}^1_{[x_1,\ldots,x_n]}.\\ &\qquad \textsc{check}(pre_2, \lambda\texttt{env}^2_{[x_1,\ldots,x_n]}.\\ &\qquad\quad next(\texttt{env}^1_{[x_1,\ldots,x_n]} \cup \texttt{env}^2_{[x_1,\ldots,x_n]}))\texttt{env}_{[x_1,\ldots,x_n]})\texttt{env}_{[x_1,\ldots,x_n]}\end{aligned}$$

The function first checks both preconditions pre_1 and pre_2 in the current environment $\texttt{env}_{[x_1,\ldots,x_n]}$. Unlike in the differential algorithm, the continuation *next* is evaluated in the union of $\texttt{env}^1_{[x_1,\ldots,x_n]}$ and $\texttt{env}^2_{[x_1,\ldots,x_n]}$, which were produced by calls to the procedure CHECK for preconditions pre_1 and pre_2 respectively. The difference stems from the fact that the BDD-based algorithm works on sets of environments, not individual ones.

In the case of existential quantification in a precondition, the algorithm is defined as follows

$$\begin{aligned}&\textsc{check}((\exists x : pre, next)\texttt{env}_{[x_1,\ldots,x_n]} =\\ &\quad \textsc{check}(pre, next \circ \pi_{[x_1,\ldots,x_n]})(\texttt{env}_{[x_1,\ldots,x_n]} \times U_{[x]})\end{aligned}$$

The function first extends the current environment, in which the precondition is checked. Furthermore, before calling the continuation *next*, the domain for the quantified variable is projected out.

The universal quantification is dealt with in the following way

$$\begin{aligned}&\textsc{check}(\forall x : pre, next)\texttt{env}_{[x_1,\ldots,x_n]} =\\ &\quad \textsc{check}(pre, next \circ (\forall_x))(\texttt{env}_{[x_1,\ldots,x_n]} \times U_{[x]})\end{aligned}$$

The algorithm utilizes universal quantification of variables in a given domain, denoted by $\forall_x$, which is a standard BDD operation provided by the BDD package that we use [37]. The operation removes tuples from the given relation by performing universal quantification over the given domain. Hence in the case of the BDD-based algorithm it is enough to extend the current environment with a quantified variable, check the precondition in the extended environment and then perform universal quantification on the returned environment.

5.4 Algorithm for LLFP

In this section we present the algorithm for solving LLFP clause sequences. The algorithm has many similarities to the differential worklist algorithm for ALFP, and is again based on the abstract algorithm presented in Section 5.1.

Similarly to the ALFP algorithms the main data structures are `env` and `result` representing the (partial) interpretation of variables and predicates, respectively. The partial environment `env` is implemented as a map from variables to their optional values. In the case the variable is undefined it is mapped into *None*. Otherwise, depending on the type of the variable is mapped to $\mathit{Some}(a)$ or $\mathit{Some}(l)$, which means that the variable is bound to $a \in \mathcal{U}$, or $l \in \mathcal{L}_{\neq\perp}$, respectively. The main operation on `env` is the function UNIFY, defined as follows

$$\textsc{unify}(\beta, \texttt{env}, (\vec{u}; V), (\vec{a}; l)) = \begin{cases} \emptyset & \text{if } \textsc{unify}_{\text{U}}(\texttt{env}, \vec{u}, \vec{a}) = \text{fail} \\ \textsc{unify}_{\text{L}}(\beta, \texttt{env}', V, l) & \text{if } \textsc{unify}_{\text{U}}(\texttt{env}, \vec{u}, \vec{a}) = \texttt{env}' \end{cases}$$

It uses two auxiliary functions that perform unifications on each component of the relation. For the first component, which ranges over the universe $\mathcal{U}$, the function is given by

$$\textsc{unify}_{\text{U}}(\texttt{env}, u, a) = \begin{cases} \texttt{env} & \text{if } (u \in \mathcal{X} \land \texttt{env}[u] = \mathit{Some}(a)) \lor u = a \\ \texttt{env}[u \mapsto \mathit{Some}(a)] & \text{if } u \in \mathcal{X} \land \texttt{env}[u] = \mathit{None} \\ \text{fail} & \text{otherwise} \end{cases}$$

It performs a unification of an argument u with an element $a \in \mathcal{U}$ in the environment `env`. In case the unification succeeds the modified environment is returned, otherwise the function fails. The funcion is extended to k-tuples in a straightforward way. The definition of the function for the lattice component is more complicated, and is given by

$$\textsc{unify}_{\text{L}}(\beta, \texttt{env}, V, l) = \begin{cases} \{\texttt{env}[V \mapsto \mathit{Some}(l \sqcap l_V)]\} \\ \quad \text{if } V \in \mathcal{Y} \land \texttt{env}[V] = \mathit{Some}(l_V) \land l \sqcap l_V \neq \perp \\ \{\texttt{env}[V \mapsto \mathit{Some}(l)]\} \\ \quad \text{if } V \in \mathcal{Y} \land \texttt{env}[V] = \mathit{None} \land l \neq \perp \\ \{\texttt{env}\} \text{ if } V = [u] \land \\ \quad ((u \in \mathcal{X} \land \texttt{env}[u] = \mathit{Some}(a)) \lor u = a) \land \beta(a) \sqsubseteq l \\ \{\texttt{env}[u \mapsto \mathit{Some}(a)] \mid \beta(a) \sqsubseteq l\} \\ \quad \text{if } V = [u] \land u \in \mathcal{X} \land \texttt{env}[u] = \mathit{None} \\ \emptyset \quad \text{otherwise} \end{cases}$$

The function is parametrized with β, which is defined in Section 3.1 and maps constants from the universe $\mathcal{U}$ into elements of the lattice $\mathcal{L}$.

Now, let us explain different cases in the definition of $\textsc{unify}_{\mathrm{L}}$. If the argument is a variable from $\mathcal{Y}$ and the environment maps that variable to the element $l_V \in \mathcal{L}_{\neq\bot}$, then the environment is updated with a new mapping for that variable. The value for the variable is set to be a greatest lower bound of l and l_V as long as it is not equal to $\bot$. In the case the argument V is a variable from $\mathcal{Y}$ that is uninitialized in the environment `env`, the function returns a singleton set containing the modified environment where that variable is mapped to l (provided that $l \neq \bot$). The third and fourth case handle the situation where the argument V is of the form $[u]$. If u is either a variable from $\mathcal{X}$ and the environment maps it to $Some(a)$, or it is a constant $a \in \mathcal{U}$, then provided that $\beta(a) \sqsubseteq l$ holds, the singleton set containing the unchanged environment `env` is returned. Otherwise, if u is an uninitialized variable from $\mathcal{X}$, then a set of modified environments is returned. The environments contained in the returned set are as `env` except that u is mapped to these $a \in \mathcal{U}$ for which $\beta(a) \sqsubseteq l$ holds. If none of the above cases holds, the empty set of environments is returned.

The other important operation on the partial environment is given by the function $\textsc{unifiable}$. When applied to `env` and a tuple $(\vec{u}; V)$, the function returns a set of tuples for which $\textsc{unify}$ would succeed. The function is defined by means of two auxiliary functions, formally we have

$$\textsc{unifiable}(\texttt{env}, (\vec{u}; V)) = (\textsc{unifiable}_{\mathrm{U}}(\texttt{env}, \vec{u}); \textsc{unifiable}_{\mathrm{L}}(\texttt{env}, V))$$

where

$$\textsc{unifiable}_{\mathrm{U}}(\texttt{env}, u) = \begin{cases} \{a\} & \text{if } (u \in \mathcal{X} \wedge \texttt{env}[u] = Some(a)) \vee u = a \\ \mathcal{U} & \text{if } u \in \mathcal{X} \wedge \texttt{env}[u] = None \end{cases}$$

and

$$\textsc{unifiable}_{\mathrm{L}}(\texttt{env}, V) = \begin{cases} l & \text{if } V \in \mathcal{Y} \wedge \texttt{env}[V] = Some(l) \\ \top & \text{if } V \in \mathcal{Y} \wedge \texttt{env}[V] = None \\ \beta(a) & \text{if } V = [u] \wedge (u = a \vee \\ & \quad (u \in \mathcal{X} \wedge \texttt{env}[u] = Some(a))) \\ \bigsqcup\{\beta(a) \mid a \in \mathcal{U}\} & \text{if } V = [u] \wedge u \in \mathcal{X} \wedge \\ & \quad \texttt{env}[u] = None \\ [\![f]\!](l) & \text{if } V = f(\vec{V}) \wedge \\ & \quad l = \textsc{unifiable}_{\mathrm{L}}(\texttt{env}, \vec{V}) \end{cases}$$

Both auxiliary funcions are extended to k-tuples in a straightforward way.

The interpretation of the predicate symbols ϱ from the semantics is given by the global data structure `result`, which is updated incrementally during computations. It is represented as a mapping from predicate names to the prefix

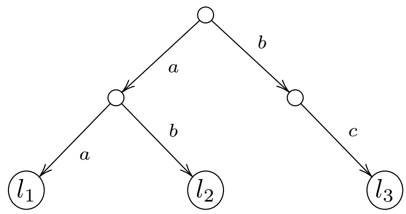


Figure 5.3: Prefix tree representing interpretation of relation R.

trees that for each predicate R record the tuples currently known to belong to R. The prefix trees themselves are implemented as arbitrarily branching trees; the formal definition is as follows

$$\texttt{RTrie} = \texttt{RNode}\ (\texttt{Map U RTrie}) \mid \texttt{RLeaf L}$$

We have two constructors; one for an internal node that contains mapping from elements of $\mathcal{U}$ to the successors nodes. The second constructor represents a terminal node (leaf) and it contains a lattice element $l \in \mathcal{L}$. Note, that the implementation of prefix trees in the LLFP solver differs from the one in the solver for ALFP. This is due to the difference in the definition of the interpretations of predicate symbols between ALFP and LLFP. Recall that for a k-ary relation in ALFP we have $\rho(R) \subseteq \mathcal{U}^k$ whereas in LLFP the interpretation is given by $\varrho(R) : \mathcal{U}^k \to \mathcal{L}$.

As an example, consider the following interpretation ϱ of a relation R

$$\varrho(R)(a, a) = l_1$$
$$\varrho(R)(a, b) = l_2$$
$$\varrho(R)(b, c) = l_3$$

The corresponding prefix tree representation is depicted in Figure 5.3. Notice that tuples are retrieved by the traversal of the prefix tree from the root to the leaves.

There are three main operations on the data structure `result`. The operation `result.`HAS acts as a lookup and checks whether a given tuple $(\vec{a}, l)$ is associated with a given predicate. More precisely, it checks whether the lattice element in the leaf of the branch labelled with $\vec{a}$ is greater or equal to l. Therefore, it checks the exact condition in the semantics of assertions and queries in LLFP. The operation `result.`SUB returns a list of the tuples associated with a given predicate and the operation `result.`ADD adds a tuple to the interpretation of a given predicate. More precisely in the case the prefix tree for the predicate

R does not contain a branch labeled with $\vec{a}$, the operation `result`.ADD $R\ (\vec{a}, l)$ adds such a branch and sets the value of the lattice element in the leaf to l. Alternatively, if the prefix tree contains a branch labeled with $\vec{a}$ ending with a leaf l', then the value of the leaf is set to the least upper bound of l and l', i.e. $l \sqcup l'$.

Since ϱ is updated as the algorithm progresses, we again make use of the data structure `infl` to record computations that have to be resumed for the new tuples. Similarly to the differential algorithm, described in Section 5.2, `infl` is represented as a prefix tree, and two main operations are `infl`.REGISTER and `infl`.CONSUMERS.

Similarly to the algorithms from Section 5.1, we have one function for each of the three syntactic categories. The function SOLVE takes a *clause sequence* as input and will call the function EXECUTE on each of the individual clauses

$$\text{SOLVE}(cl_1, \ldots, cl_s) \quad = \quad \text{EXECUTE}(cl_1)[\,]; \ldots; \text{EXECUTE}(cl_s)[\,]$$

where we write [ ] for the empty environment reflecting that we have no free variables in the clause sequences.

Let us now turn to the description of the function EXECUTE. Again, the function takes a *clause cl* as a parameter and a representation `env` of the interpretation of the variables. We have one case for each of the forms of cl; and let us consider the case of an assertion first. The algorithm is as follows

```
EXECUTE(R(u⃗; V))env =
    let ITERFUN (a⃗; l) =
        match result.HAS(R, (a⃗; l)) with
            | true → ()
            | false →
                result.ADD(R, (a⃗; l))
                ITER (fun f → f (a⃗; l)) (infl.CONSUMERS R)
    in ITER ITERFUN (UNIFIABLE(env,(u⃗; V)))
```

The function uses the auxiliary function ITER, which applies the function ITERFUN to each element of the list of tuples that can be unified with the argument $(\vec{u}; V)$. Given a tuple $(\vec{a}; l)$, the function ITERFUN adds the tuple to the interpretation of R stored in `result` if it is not already present. If the ADD operation succeeds, we first create a list of all the consumers currently registered for predicate R by calling the function `infl`.CONSUMERS. Thereafter, we resume the computations by iterating over the list of consumers and calling corresponding continuations.

The cases of the always true clause, **1**, conjunction of clauses, and implication are exactly as defined in Section 5.1, whereas the case of universal quantification follows the definition from Section 5.2.

Now, let us present the function CHECK. It takes a *precondition*, a continuation, and an environment as parameters. We first consider the algorithm in the case of positive queries

$\textsc{check}(R(\vec{u};V), next)\texttt{env} =$
    let $\textsc{consumer}\ (\vec{a};l) =$
        match $\textsc{unify}(\texttt{env}, (\vec{u};V), (\vec{a};l))$ **with**
            | fail $\rightarrow$ ()
            | `envs` $\rightarrow$ ITER *next* `envs`
    in `infl`.REGISTER(R,CONSUMER); ITER CONSUMER (`result`.SUB R)

We first ensure that the consumer is registered in `infl`, by calling function REGISTER, so that future tuples associated with R will be processed. Thereafter, the function inspects the data structure `result` to obtain the list of tuples associated with the predicate R. Then, the auxiliary function CONSUMER unifies each tuple with $(\vec{u};V)$; and if the operation succeeds, the continuation *next* is invoked on each of the updated new environments in the returned set `envs`.

In the case of negated query, the algorithm is of the form

$\textsc{check}(\neg R(\vec{u};V), next)\texttt{env} =$
    let ITERFUN $(\vec{a};l) =$
        match `result`.HAS$(R, (\vec{a};l))$ **with**
            | *true* $\rightarrow$ ()
            | *false* $\rightarrow$ ITER *next* $(\textsc{unify}(\texttt{env}, (\vec{u};V), (\vec{a};l)))$
    in ITER ITERFUN $(\textsc{unifiable}(\texttt{env}, (\vec{u};V)))$

The function first computes the tuples unifiable with $(\vec{u};V)$ in the environment `env`. Then, for each tuple it checks whether the tuple is already in R and if not, the tuple is unified with $(\vec{u};V)$ to produce a set of new environments. Thereafter, the continuation *next* is evaluated in each of the environments contained in the returned set.

Now, let us consider the function CHECK in the case of $Y(x)$, where $x \in \mathcal{X}$. The algorithm is as follows

$\textsc{check}(Y(x), next)\texttt{env} =$
    let `env'` $=$ **if** $\texttt{env}(Y) = Some(l)$ **then** `env` **else** $\texttt{env}[Y \mapsto \top]$
    in let F a $=$ **if** $Some(\beta(a)) \sqsubseteq \texttt{env'}(Y)$ **then** *next* $\texttt{env'}[x \mapsto a]$ **else** ()
    in match `env'`(x) **with**
        | $Some(a)$ $\rightarrow$ F a
        | $None$ $\rightarrow$ ITER F U

The function begins with creating an environment `env'` that is exactly as `env` except that the binding for the variable Y is set to $\top$ in the case Y is undefined in `env`. Then, we define an auxiliary function that checks whether `env'`(Y) over-approximates the abstraction of an argument a, denoted by $\beta(a)$, and if so the continuation is called in the environment `env'`$[x \mapsto a]$. Finally, the function checks the binding for the variable x in the environment `env'` and if it is bound to $Some(a)$ the function F applied to a is called. Otherwise, the function F is called for each element of the universe, using the ITER function. In the case the argument of Y is a constant $a \in \mathcal{U}$ the function is given by

CHECK$(Y(a), next)$`env` =
  let `env'` = **if** `env`$(Y) = Some(l)$ **then** `env` **else** `env`$[Y \mapsto \top]$
  in if $Some(\beta(a)) \sqsubseteq$ `env'`(Y) **then** $next$ `env'` **else** ()

This is essentially the same as the case explained above, except that we do not have to handle the case when $x \in \mathcal{X}$ is undefined in `env`.

All the other cases are exactly as defined in Section 5.2, and hence omitted.

5.5 Algorithm for LFP

In this section we present an algorithm for solving LFP formulae. The algorithm is based on a BDD representation of relations and it is fairly similar to the BDD-based algorithm for ALFP, presented in Section 5.3.

Similarly to the case of the ALFP algorithm, it operates with (intermediate) representations of the two interpretations ς and ϱ of the semantics presented in Table 4.1; we call them `env` and `result`, respectively. In the algorithm `result` is an imperative data structure that is updated as we progress. The data structure `env` is supplied as a parameter to the functions of the algorithms. Both data structures are represented as reduced ordered binary decision diagrams (ROBDDs). Consequently the algorithm operates on entire relations, rather than on individual tuples.

In the following we use exactly the same BDD operations as the ones introduced in Section 5.3. Similarly in order to keep track of the domain names of the BDDs the environments and predicates are annotated with subscript $[d_1, \cdots, d_k]$ denoting a list of pairwise disjoint domain names.

In the algorithm the `infl` data structure is implemented as a mapping from predicate names to functions, again called consumers, that are used to resume computations when the interpretation of the given predicate is updated. The `infl` data structure has two main operations REGISTER and RESUME for adding new consumers and invoking registered computations, respectively.

We have one function for each of the syntactic categories. The function SOLVE takes a *clause sequence* as input and calls the function EXECUTE on each of the individual clauses. Hence it is exactly the same as the abstract algorithm presented in Section 5.1.

Similarly to other algorithms, we have one function for each of the syntactic categories. The function EXECUTE takes a *clause cl* as a parameter and calls the appropriate function depending on whether a given clause is a *define* or a *constrain* clause. The pseudo code is as follows

$$\begin{aligned} \text{EXECUTE}(\mathit{define}(cl)) &= \text{EXECUTE}_{\text{DEF}}(cl)[\,] \\ \text{EXECUTE}(\mathit{constrain}(cl)) &= \text{EXECUTE}_{\text{CON}}(cl)[\,] \end{aligned}$$

where we write [ ] for the empty environment reflecting that we have no free variables in the clause sequences.

The function $\text{EXECUTE}_{\text{DEF}}$ is defined exactly as the EXECUTE function in the BDD-based algorithm for ALFP, and hence omitted. The novelty of the LFP logic and thus the algorithm is its direct support for co-inductive specifications. Therefore, let us focus on the function $\text{EXECUTE}_{\text{CON}}$, which handles the *constrain* clauses. Let us first consider the case of the assertion, which is the most interesting. The function is defined as follows

$$\begin{aligned} &\text{EXECUTE}_{\text{CON}}\ R_{[d_1,\ldots,d_k]}(u_1,\ldots,u_k)\ \mathtt{env}_{[x_1,\ldots,x_n]} = \\ &\quad \mathtt{env}'_{[x_1,\ldots,x_n]} \leftarrow \complement\mathtt{env}_{[x_1,\ldots,x_n]} \\ &\quad \textbf{for } i = 1 \textit{ to } k \textbf{ do} \\ &\quad\quad \mathtt{env}'_{[x_1,\ldots,x_n,d_1,\ldots,d_i]} \leftarrow \sigma_{u_i=d_i}(\mathtt{env}'_{[x_1,\ldots,x_n,d_1,\ldots,d_{i-1}]} \times U_{[d_i]}) \\ &\quad oldR_{[d_1,\ldots,d_k]} \leftarrow \mathtt{result}[R] \\ &\quad \mathtt{result}[R] \leftarrow oldR_{[d_1,\ldots,d_k]} \cap \complement(\pi_{[d_1,\ldots,d_k]}(\mathtt{env}'_{[x_1,\ldots,x_n,d_1,\ldots,d_k]})) \\ &\quad \textbf{if } oldR_{[d_1,\ldots,d_k]} \neq \mathtt{result}[R] \textbf{ then} \\ &\quad\quad \mathtt{infl}.\text{RESUME } R \end{aligned}$$

The function begins with complementing the current environment and assigning the result to variable $\mathtt{env}'_{[x_1,\ldots,x_n]}$. In the *for* loop the function incrementally builds a product of the complemented environment and a relation representing the universe, and simultaneously selects the tuples compatible with the arguments $(u_1,\cdots,u_k)$. Since we aim at computing the greatest set of tuples for relation R, we assign the content of R with an intersection of the current interpretation of R and the complement of the relation denoted by $\mathtt{env}'_{[x_1,\ldots,x_n,d_1,\ldots,d_i]}$ projected to the domain names of R. Additionally, if the interpretation of predicate R has changed, we invoke the consumers registered for predicate R in the data structure `infl` by calling the RESUME function.

The case of conjunction is straightforward as we again have to inspect both clauses in the same environment $\texttt{env}_{[x_1,\ldots,x_n]}$. The pseudo code is as follows

$$\begin{aligned}&\text{EXECUTE}_{\text{CON}}(con_1 \wedge con_2)\texttt{env}_{[x_1,\ldots,x_n]} = \\&\quad \text{EXECUTE}_{\text{CON}}(con_1)\texttt{env}_{[x_1,\ldots,x_n]}; \text{EXECUTE}_{\text{CON}}(con_2)\texttt{env}_{[x_1,\ldots,x_n]}\end{aligned}$$

In the case of implication we again make use of the function CHECK that in addition to the condition and the environment also takes the continuation $\text{EXECUTE}_{\text{CON}}$ partially applied to $R_{[d_1,\ldots,d_k]}(u_1,\ldots,u_k)$ as an argument. The function is defined as

$$\begin{aligned}&\text{EXECUTE}_{\text{CON}}(R_{[d_1,\ldots,d_k]}(u_1,\ldots,u_k) \Rightarrow cond)\texttt{env}_{[x_1,\ldots,x_n]} = \\&\quad \text{CHECK}(cond, \text{EXECUTE}_{\text{CON}}(R_{[d_1,\ldots,d_k]}(u_1,\ldots,u_k)))\texttt{env}_{[x_1,\ldots,x_n]}\end{aligned}$$

The case of universal quantification is of the following form:

$$\text{EXECUTE}_{\text{CON}}(\forall x : con)\ \texttt{env}_{[x_1,\ldots,x_n]} = \text{EXECUTE}_{\text{CON}}\ con\ (\texttt{env}_{[x_1,\ldots,x_n]} \times U_{[x]})$$

The function extends the current environment with a domain for the quantified variable, and then executes the *constrain* clause *con*.

The function CHECK takes a *condition*, a continuation and an environment as parameters. The definition is the same as the one presented in Section 5.3 and hence omitted.

In order to conclude this chapter, in Figure 5.4 we provide an overview of the data structures used in the presented algorithms. Clearly that the differential ALFP solver and LLFP solver use very similar data structures. The difference in the implementation of `RTrie` follows from the definition of the interpretation of predicate symbols. In the ALFP algorithm `RTrie` represents a set of tuples, whereas in the LLFP one, for each tuple in the prefix tree we additionally have a corresponding lattice value. In both algorithms, the environment `env` maps variables to their optional values. However, since in ALFP values of variables range over a finite universe $\mathcal{U}$, we have U as the range of the mapping. In LLFP, on the other hand, we distinguish between variables ranging over a universe $\mathcal{U}$ and a complete lattice $\mathcal{L}$, hence the range of the mapping is given by an auxiliary data structure `Val`. When we compare the BDD-based ALFP solver with the LFP one, it is evident that they use the same data structures. This is because the underlying logics are very similar — the main difference is the direct support for co-inductive specifications in the case of LFP. This allows us to share some of the code-base of the two solvers.

	differential ALFP solver	BDD-based ALFP solver	LLFP solver	LFP solver
`result`	`Map R RTrie`	`Map R BDD`	`Map R RTrie`	`Map R BDD`
`RTrie`	`RNode (Map U RTrie)`	-	`RNode (Map U RTrie)` | `RLeaf L`	-
`infl`	`Map R ITrie`	`Map R cons`	`Map R ITrie`	`Map R cons`
`ITrie`	`INode cons (Map U ITrie)`	-	`INode cons (Map U ITrie)`	-
`env`	`Map Var U`	`BDD`	`Map Var Val`	`BDD`
			`Val = ValU U` | `ValL L`	

Figure 5.4: Overview of the data structures.

Chapter 6

Magic set transformation for ALFP

In this chapter we present how magic set transformation [7, 8, 50], known from deductive databases [30], can be applied to increase the efficiency of bottom-up evaluation of analysis problems expressed in ALFP. The transformation presented is essentially equivalent to the magic set transformation for Datalog. The novelty of our method lies in handling universal quantification in preconditions of ALFP formulae, which goes beyond expressiveness of Datalog. Even though, the developments of this chapter are presented for ALFP, we believe that they could also be applied to the other logics presented in this dissertation.

In the classical formulation of the ALFP logic in order to answer a specific query, the entire solution has to be computed, followed by selection of tuples of interest. This is inefficient since many irrelevant tuples are discovered during the computations.

Here we present a remedy for that problem by first adding the ability to specify queries. Secondly, in order to avoid generating irrelevant tuples, we perform the magic set transformation, which is a compile-time transformation of the original clauses based on the supplied query. More precisely, having a query q the idea is to transform the ALFP clause sequence cls into a clause sequence cls_q such that they both give the same answer set to the query q. As a result we narrow down the exploration of the state space and the bottom-up computation focuses on

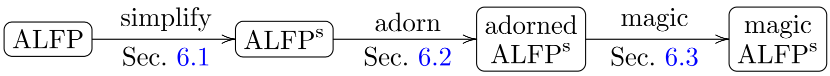


Figure 6.1: Overview of the chapter.

relevant tuples. The transformation presented in this chapter proceeds strata by strata, and hence negation presents no obstacles. For a stratified ALFP formula, the result of the transformation is also stratified. Moreover, the theoretical worst-case time complexity of solving the resulting formula increases linearly.

The structure of this chapter is depicted in Figure 6.1. In order to simplify the transformation, but with no loss of expressive power, we introduce in Section 6.1 a restricted syntax of ALFP logic. We present the first step of the transformation, namely the *adorn* algorithm, in Section 6.2. The magic set transformation algorithm is presented in Section 6.3.

6.1 The restricted syntax of ALFP logic

In order to make the presentation of the magic set transformation easier, we assume that the ALFP formula is in a restricted syntax called ALFP$^{\mathrm{s}}$. The following definition introduces the syntax of ALFP$^{\mathrm{s}}$.

Definition 6.1 Given a fixed countable set $\mathcal{X}$ of variables, a non-empty and finite universe $\mathcal{U}$ and a finite alphabet $\mathcal{R}$ of predicate symbols, we define the set of ALFP$^{\mathrm{s}}$ formulae (or clause sequences), *cls*, together with clauses, *cl*, and preconditions, *pre*, by the grammar:

$$
\begin{array}{lll}
u & ::= & x \mid a \\
pre & ::= & R(\vec{u}) \mid R(\vec{u}) \mid pre_1 \wedge pre_2 \mid \forall x : pre \mid \exists x : pre \\
cl' & ::= & R(\vec{u}) \mid \mathbf{1} \mid pre \Rightarrow R(\vec{u}) \\
cl & ::= & \forall x : cl \mid cl' \\
cls & ::= & cl_1, \ldots, cl_s
\end{array}
$$

Here $u \in (\mathcal{X} \cup \mathcal{U})$, $a \in \mathcal{U}$, $x \in \mathcal{X}$, $R \in \mathcal{R}$ and $s \geq 1$.

The transformation of ALFP formulae into ALFP$^{\mathrm{s}}$ ones proceeds in a number of stages:

1. First the variables introduced by the quantifiers are renamed so that they are pairwise distinct. This is needed in order to avoid name captures.

2. All universal quantifications in preconditions $\forall x : pre$ are transformed as follows. We introduce a fresh predicate name P and create a new clause asserting P of the following form $\forall \vec{y} : pre \Rightarrow P(\vec{y})$, where $\vec{y}$ is a sequence of free variables in pre excluding constants from $\mathcal{U}$. Furthermore, we modify the original precondition into $\forall x : P(\vec{y})$. This step is needed in order to be able to further transform the preconditions so that they do not contain disjunctions (which is done in step 5). In particular if the precondition pre does not contain disjunctions this step may be omitted. More formally, the step is defined as follows

   $f^i_\forall(\forall x : pre) =$
   let $\vec{y}$ be a vector of free variables in pre
   let P be a fresh predicate symbol
   insert a clause $\forall \vec{y} : pre \Rightarrow P(\vec{y})$ before cl_i into cls
   transform $\forall x : pre$ into $\forall x : P(\vec{y})$

   where the superscript i of the function denotes the index of the current clause.

3. Similarly all existential quantifiers in preconditions of the form $\exists x : pre_1 \vee pre_2$ are transformed into $(\exists x : pre_1) \vee (\exists x : pre_2)$. Formally

   $$f^i_\exists(\exists x : pre_1 \vee pre_2) = (\exists x : pre_1) \vee (\exists x : pre_2)$$

4. The preconditions of all clauses are transformed into the form $pre'_1 \vee \cdots \vee pre'_k$ where each of the pre'_i is a conjunction of preconditions of the form $R(\vec{u})$, $\neg R(\vec{u})$, $\exists x : pre$ and $\forall x : pre$ (so they adhere to the grammar for pre given above).

5. The clauses are transformed so that they do not use disjunction in preconditions, that is, all occurrences of $(pre'_1 \vee \cdots \vee pre'_k) \Rightarrow cl$ are replaced by the k conjuncts $(pre'_1 \Rightarrow cl) \wedge \cdots \wedge (pre'_k \Rightarrow cl)$.

6. All (universal) quantifiers are moved to the outermost level in the clauses. Thus $pre' \Rightarrow (\forall x : cl)$ becomes $\forall x : (pre' \Rightarrow cl)$.

7. All clauses of the form $\forall \vec{\alpha} : (pre' \Rightarrow cl_1 \wedge cl_2)$ are replaced by clauses of the form $(\forall \vec{\alpha} : (pre' \Rightarrow cl_1)) \wedge (\forall \vec{\alpha} : (pre' \Rightarrow cl_2))$ and all clauses of the form $\forall \vec{\alpha} : (pre' \Rightarrow (pre'' \Rightarrow cl))$ are replaced by clauses of the form $\forall \vec{\alpha} : (pre' \wedge pre'' \Rightarrow cl)$. Since there can be more than one conjunction or implication in the conclusion, this step is performed iteratively until no more transformations can be done.

The worst-case time complexity of solving the corresponding ALFP^s clauses increases linearly. This is because the maximal nesting depth of quantifiers does not change and the number of fresh predicates and clauses asserting them is proportional to the number of original ALFP clauses.

The following lemma states that the transformation is semantics preserving.

Lemma 6.2 *Let cls be a closed and stratified clause sequence in ALFP, and cls′ be a corresponding clause sequence in ALFP^s. Then*

$$(\rho, \sigma) \models cls \Leftrightarrow (\rho, \sigma) \models cls'$$

Proof. See Appendix A.12.

6.2 Adorned ALFP^s clauses

In this section we present the first step of the transformation; namely creating the adorned ALFP^s clauses [55]. Informally, the adorned clauses show the flow of sideways information between relations in the formula. They are created by adding annotations to predicates and for the purpose of the magic set transformation we annotate derived predicates only. As already mentioned in Section 2.3 a derived predicate is one that is defined solely by clauses with non-empty preconditions. The goal of a magic set transformation is to reduce the number of irrelevant tuples generated during the bottom-up computation. We do not adorn base predicates, since they are fully evaluated, and querying them retrieves only relevant tuples anyway.

For an occurrence of a literal $R(u_1, \ldots, u_n)$ the adornment α is a n-tuple of characters b and f standing for bound and free, respectively. In the case the argument u_i is bound then there is b at the i-th position in α, otherwise there is an f. As an example take a query $R(a, w, c)$, where a and c are constants and w is an unbound variable. Then the adorned version of the query is $R^{bfb}(a, w, c)$.

The adorned clauses are the result of a sideways information passing strategy (SIPS) [8], which captures what information is passed by a predicate (or a set of predicates) to another predicate. More precisely, for an ALFP^s clause a SIPS represents a strategy for evaluating the given clause. Namely, it describes in which order the literals in the precondition are evaluated, and how values of variables are passed from literal to literal during the bottom-up computation of the least model. As emphasized in [6], the SIPS does not say how the information is passed i.e. it does not specify whether information is passed one tuple at a time or a set of tuples at a time. In [8] this component is called a control component and it is not discussed here.

It is important to note that the generated SIPS depends on the form of the supplied query. The query form can be seen as a generic way of representing a set of queries, and it can be written as an adorned predicate name. For example having two different queries $R(a, w)$ and $R(c, w)$, where a and c are constants and w is a free variable, the resulting adorned predicate name is exactly the same, namely R^{bf}. On the other hand, the two queries $R(w, a)$ and $R(c, w)$ give rise to two different query forms: R^{fb} and R^{bf}, respectively.

As mentioned above SIPS describes how bindings of an asserted predicate are passed and used to evaluate the precondition. Hence, a SIPS depicts how the clause is evaluated when a given set of arguments of the asserted predicate is bound to constants. As an example, let us consider the following ALFP^{s} clauses specifying the transitive closure of a relation E.

$$\begin{array}{l}\forall x : \forall y : E(x, y) \Rightarrow T(x, y) \wedge \\ \forall x : \forall y : (\exists z : E(x, z) \wedge T(z, y)) \Rightarrow T(x, y)\end{array}$$

Let us also assume that we are interested in computing all the states reachable from s_1. This corresponds to a query $T(s_1, w)$, where w is a free variable. Since the first argument is bound to s_1, by unification the variable x in the second clause is bound to s_1. The second clause can be evaluated using that binding and as a result we obtain the bindings for z, due to $E(x, z)$. These are passed to literal $T(z, y)$ and generate new subgoals that have the same binding pattern, namely T^{bf}. We can generalize the above reasoning and may say that a SIPS aims to evaluate a set of predicates, and use the result to bind the variables appearing as arguments of other predicates.

The formal definition of SIPS is adopted from [6] and is given below. We first define a notion of *connectedness* as follows.

Definition 6.3 Let *cl* be an ALFP^{s} clause containing literals $R(\vec{u})$ and $S(\vec{v})$. We say that $R(\vec{u})$ and $S(\vec{v})$ are *connected* if

- $R(\vec{u})$ and $S(\vec{v})$ share a common variable as an argument; or
- there exists a literal $Q(\vec{w})$ in *cl* such that $R(\vec{u})$ is *connected* to $Q(\vec{w})$ and $Q(\vec{w})$ is *connected* to $S(\vec{v})$.

Thus connectedness is essentially a transitive closure of sharing a common argument.

Let Pre(cl) be the set of literals in the precondition of cl, in the case cl does not have a precondition the set Pre(cl) is empty. Formally we define Pre(cl) as follows

$$\begin{array}{lcl}
\mathrm{Pre}(R(\vec{v})) & = & \emptyset \\
\mathrm{Pre}(\mathbf{1}) & = & \emptyset \\
\mathrm{Pre}(pre \Rightarrow cl) & = & \mathrm{Pre}'(pre) \\
\mathrm{Pre}(\forall x : cl) & = & \mathrm{Pre}(cl) \\
 & & \\
\mathrm{Pre}'(R(\vec{v})) & = & \{R(\vec{v})\} \\
\mathrm{Pre}'(\neg R(\vec{v})) & = & \{\neg R(\vec{v})\} \\
\mathrm{Pre}'(pre_1 \wedge pre_2) & = & \mathrm{Pre}'(pre_1) \cup \mathrm{Pre}'(pre_2) \\
\mathrm{Pre}'(\forall x : pre) & = & \mathrm{Pre}'(pre) \\
\mathrm{Pre}'(\exists x : pre) & = & \mathrm{Pre}'(pre)
\end{array}$$

The SIPS are defined as follows.

Definition 6.4 Let $R^{\alpha}(\vec{u})$ be an adorned version of a literal asserted in cl. Sideways Information Passing Strategy (SIPS) for cl is a ternary relation $\mathcal{G} \subseteq \mathcal{V}_1 \times \mathcal{P}(\mathcal{X}) \times \mathcal{V}_2$ where $\mathcal{V}_1 = \mathcal{P}(\mathrm{Pre}(cl) \cup \{R^{\alpha}(\vec{u})\})$, $\mathcal{V}_2 = \mathrm{Pre}(cl)$, and where the following conditions hold

1. Each tuple is of the form $(V, \mathcal{W}, P^{\beta}(\vec{v}))$, where $V \in \mathcal{V}_1$ and $P^{\beta}(\vec{v}) \in V_2$. Furthermore, $\mathcal{W}$ stands for a nonempty set of variables that satisfies the following conditions
   (a) each variable in $\mathcal{W}$ appears in the argument of $P^{\beta}(\vec{v})$, and in either a bound argument position of $R^{\alpha}(\vec{u})$, or a positive literal in V (or both),
   (b) each literal in V is *connected* to $P^{\beta}(\vec{v})$.
2. There exists a total order $\leq$ on $\mathrm{Pre}(cl) \cup \{R^{\alpha}(\vec{u})\}$ in which:
   (a) For all $Q^{\gamma}(\vec{w}) \in \mathrm{Pre}(cl) : R^{a}(\vec{u}) \leq Q^{\gamma}(\vec{w})$,
   (b) $\forall Q^{\gamma}(\vec{w}) \in (\mathrm{Pre}(cl) \cup \{R^{\alpha}(\vec{u})\}) : \forall S^{\beta}(\vec{y}) \notin (\mathrm{Pre}(cl) \cup \{R^{\alpha}(\vec{u})\}) : Q^{\gamma}(\vec{w}) \leq S^{\beta}(\vec{y})$,
   (c) for each tuple $(V, \mathcal{W}, P^{\beta}(\vec{v}))$ we have $\forall Q^{\gamma}(\vec{w}) \in V : Q^{\gamma}(\vec{w}) \leq P^{\beta}(\vec{v})$.

A tuple $(V, \mathcal{W}, P^{\beta}(\vec{v})) \in \mathcal{G}$ means that by evaluating the join of the literals in V, where some of the arguments are bound to constants, we obtain values for the variables in $\mathcal{W}$, which are then passed to $P^{\beta}(\vec{v})$ and used to restrict the retrieved tuples. For that reason it is required by condition (1a) that all variables appearing in $\mathcal{W}$ appear in the argument of $P^{\beta}(\vec{v})$, since including in $\mathcal{W}$ variables that do not appear in the argument of $P^{\beta}(\vec{v})$ does not influence the evaluation of $P^\beta(\vec{v})$. Furthermore, by condition (1b) only literals that are connected to the $P^\beta(\vec{v})$ are included in V. This is because a literal that is not connected does not serve any useful role when performing the join. The intuition behind imposing the condition (2) is to provide a consistency condition on SIPS. More precisely, it forbids cyclic dependencies in the SIPS; namely the situation where different strategies make cyclic assumptions about variables being bound.

Now, we explain how the tuples of the SIPS are used to evaluate ALFP^s clauses. Assume that we want to evaluate a clause asserting a predicate $R^\alpha(\vec{u})$ with some arguments in $\vec{u}$ bound to constants. The evaluation begins with the literals that do not appear as the third component of any tuple in the SIPS. These literals are evaluated with all arguments free, except if a given argument is a constant. Then, literals appearing as the third component in the tuples are evaluated with values supplied by the second component of a given tuple. Finally, having all predicates evaluated, we perform a join followed by the projection to the arguments of the asserted predicate $R^\alpha(\vec{u})$.

In the following we write a tuple $(V, \mathcal{W}, P^\beta(\vec{v})) \in \mathcal{G}$ as $V \rightarrow_{\mathcal{W}} P^\beta(\vec{v})$ to follow existing notation [6]. As an example we again consider the clauses specifying the transitive closure of a relation E, and the corresponding query is $T(s_1, w)$, where w is a free variable. The adorned version of the query is $T^{bf}(s_1, w)$. The SIPS for the first clause is

$$\{T^{bf}(x, y)\} \rightarrow_{\{x\}} E(x, y)$$

whereas for the second one we have two different SIPS

$$\begin{aligned} &\{T^{bf}(x, y)\} \rightarrow_{\{x\}} E(x, z) \\ &\{T^{bf}(x, y), E(x, z)\} \rightarrow_{\{z\}} T^{bf}(z, y) \end{aligned} \tag{6.1}$$

and

$$\begin{aligned} &\{T^{bf}(x, y)\} \rightarrow_{\{x\}} E(x, z) \\ &\{E(x, z)\} \rightarrow_{\{z\}} T^{bf}(z, y) \end{aligned} \tag{6.2}$$

It is important to point out the difference between the above two SIPS; thus let us focus on (6.1) and (6.2). In the first one, the literal $T^{bf}(z, y)$ is evaluated based on the information passed from evaluating the conjunction of $T^{bf}(x, y)$ and $E(x, z)$, whereas in the second one it is based on the information passed from $E(x, z)$ alone. As a consequence, (6.1) may be more efficient since it may restrict more irrelevant tuples. As we will see in the next section, in practice it is always best to use the *normalized* SIPS, defined in Definition 6.5. Notice that in both SIPS the predicate E is not adorned, since it is a base predicate (it is defined only by facts).

6.2.1 The adorn algorithm

In order to create an adorned version of the original clauses, we begin by creating an adorned query in the way described at the beginning of this section. The adorned version of the original clauses is created by the adorn algorithm, which is based on the algorithm by Balbin et al. [6], and whose pseudo-code is given in Figure 6.2. The algorithm takes three arguments: the adorned predicate name, a clause sequence and SIPS. It maintains a worklist W containing adorned predicate names that still need to be processed. The worklist is initialized with the adorned version of the query, and the algorithm terminates when the worklist becomes empty. In each iteration, an adorned predicate name is removed from the worklist and added to the set of processed items. Then for each clause that asserts the given predicate, the algorithm creates an adorned version of the clause by adorning predicates in the precondition. Additionally, it adds newly created adorned predicate names into a worklist. Since there is a finite number of clauses and a finite number of possible adorned predicate names, the algorithm is guaranteed to terminate. The algorithm in Figure 6.2 makes use of a function h that creates adornments of predicates

$$h(\mathcal{W}, v) = \begin{cases} b & \text{if } v \in \mathcal{W} \\ f & \text{otherwise} \end{cases}$$

The function is extended to tuples $\vec{v}$ in a straightforward manner.

In order to simplify the algorithm, we assume that all SIPS are normalized in a way described in [6], and formally defined in the following definition (adopted from [6]).

Definition 6.5 A *normalized* SIPS has the property that whenever there are n tuples in a SIPS, $n > 0$, $(V_i, \mathcal{X}_i, R(\vec{u}))$, $0 \leq i \leq n$, agreeing in the third component then $(\bigcup V_i, \bigcup \mathcal{X}_i, R(\vec{u}))$ is also valid SIPS.

The above definition states that different strategies for sideways information passing can be combined together in the case they match on the last component, producing a normalized strategy. As a result we know that for a given clause there is at most one edge in the SIPS with a given adorned version of a predicate, or in other words there is a following functional dependency

$$\mathcal{V}_2 \rightarrow_{cl} \mathcal{V}_1 \times \mathcal{P}(\mathcal{X})$$

which simplifies the magic set algorithm presented in Section 6.3.

Continuing with the running example the adorned clauses defining the transitive

$\text{adorn}(R^\alpha, cls, S)$
    $W := \{R^\alpha\}$
    let cls' be an empty sequence of clauses
    $S' := D := \emptyset$
    while $W \neq \emptyset$ **do**
        let P^β be an adorned predicate name from W
        $W := W \setminus P^\beta$
        $D := D \cup P^\beta$
        let cls_P be a copy of the clauses from cls asserting P
        foreach cl of the form $\forall\vec{x} : pre \Rightarrow P(\vec{v})$ **in** cls_P **do**
            let S_{cl} be a copy of the SIPS associated with clause cl
            $cl := \forall\vec{x} : pre \Rightarrow P^\beta(\vec{v})$
            foreach derived literal $Q(\vec{v})$ in pre **do**
                $\gamma := h(\mathcal{W}, \vec{v})$ where $(V, \mathcal{W}, Q(\vec{v})) \in S_{cl}$
                $pre := pre[Q^\gamma(\vec{v})/Q(\vec{v})]$
                replace all occurrences of $Q(\vec{v})$ in S_{cl} by $Q^\gamma(\vec{v})$.
                if $Q^\gamma \notin D$ **then** $W := W \cup \{Q^\gamma\}$
             od
            add cl into cls'
            $S' := S' \cup \{S_{cl}\}$
         od
     od
    return (cls', S')

Figure 6.2: Algorithm for creating adorned ALFP$^{\text{s}}$ clauses.

closure of the relation E are as follows

$$\begin{array}{l}\forall x : \forall y : E(x,y) \Rightarrow T^{bf}(x,y) \wedge \\ \forall x : \forall y : (\exists z : E(x,z) \wedge T^{bf}(z,y)) \Rightarrow T^{bf}(x,y)\end{array}$$

Notice that, as mentioned before, only derived predicates are adorned (in this case predicate T).

As pointed out in [6], this normalization is essentially equivalent to the approach taken in [8]. Their approach handles multiple edges leading to a given predicate by introducing auxiliary predicates, and then by defining additional clause to join the information from these predicates. We believe that the normalization approach from [6] is superior, since it eliminates the need for introducing auxiliary predicates and further simplifies the magic set algorithm.

Definition 6.6 Let cls_1 and cls_2 be two closed and stratified ALFP$^{\mathrm{s}}$ formulae over a universe $\mathcal{U}$, and $R_1(\vec{v})$ and $R_2(\vec{v})$ two queries of arity k on cls_1 and cls_2, respectively. Let ρ_1 and ρ_2 be two least models such that $(\rho_1, [\,]) \models cls_1$ and $(\rho_2, [\,]) \models cls_2$. We say that formulae cls_1 and cls_2 are equivalent with respect to $R_1(\vec{v})$ and $R_2(\vec{v})$, written $cls_1 \equiv^{R_1(\vec{v})}_{R_2(\vec{v})} cls_2$, if

$$\forall \vec{a} \in \varsigma_{\vec{v}}(\mathcal{U}) : \vec{a} \in \rho_1(R_1) \Leftrightarrow \vec{a} \in \rho_2(R_2)$$

where

$$\varsigma_v(\mathcal{S}) = \begin{cases} \{v\} & \text{if } v \in \mathcal{S} \\ \mathcal{S} & \text{otherwise} \end{cases}$$

and

$$\varsigma_{(v_1,\ldots,v_k)}(\mathcal{S}) = \varsigma_{v_1}(\mathcal{S}) \times \ldots \times \varsigma_{v_k}(\mathcal{S})$$

Proposition 6.7 *Let cls be a closed and stratified ALFP^{s} formula and $R(\vec{v})$ be a query on cls. Let cls′ and $R^{\alpha}(\vec{v})$ be the result of the adorn algorithm and the corresponding adorned query, respectively. Then $cls \equiv^{R(\vec{v})}_{R^{\alpha}(\vec{v})} cls'$.*

Proof. See Appendix A.13.

6.3 Magic sets algorithm

In this section we extend the magic set transformation algorithm to ALFP$^{\mathrm{s}}$. The transformation is done at compile time and it transforms the adorned ALFP$^{\mathrm{s}}$ clauses with respect to a given query, into clauses that give the same answer to the query as the original ones. The intuition behind the transformation is that during the bottom-up evaluation of the transformed ALFP^{s} formula, facts that may contribute to the answer to the query are discovered (the transformation narrows down the search space). The transformation makes the following changes to the adorned clauses:

- It adds new predicates, called magic predicates, into the preconditions of the existing clauses,
- It defines additional clauses, called magic clauses, asserting the magic predicates.

The magic predicates are created for derived relations only; ideally for only some of them. In particular, for a predicate $R^{\alpha}(\vec{u})$ where adornment α contains exactly k b's, we create a predicate $MagicR^{\alpha}(\vec{u}')$ whose arity is k and the arguments are those u_i that are indicated as bound in α. More formally, the magic predicates are created by function mkMagic defined as follows

$$\text{mkMagic}(R^{\alpha}(\vec{u})) = MagicR^{\alpha}(u_i \mid \alpha_i = b, \vec{u} = u_1, \ldots, u_n, \alpha = \alpha_1, \ldots, \alpha_n)$$

Thus, for an adorned predicate R^{α} we create the magic predicate $MagicR^{\alpha}$, whose rank is equal to the rank of R^{α} i.e. $\text{rank}(R^{\alpha}) = \text{rank}(MagicR^{\alpha})$. The ranking function was formally defined in Section 2.3. The arguments of the magic predicate are these u_i that are indicated as bound by the adornment α. The set of tuples computed during the bottom-up evaluation of the transformed clauses is called magic set. Even though the aim of the algorithm is to transform the original clauses such that during bottom-up evaluation only relevant facts are discovered, irrelevant tuples are usually generated. The reason for that is that it is not possible to know in advance which tuples are relevant and which are not. The pseudo code of the magic set transformation is given in Figure 6.3, and is based on the algorithm by Balbin et al. [6]. The algorithm takes an adorned query, adorned clause sequence and normalized SIPS as arguments. It uses function mkPre, that creates a precondition for the magic clauses and is defined as follows

$$\text{mkPre}_Q(V) = \bigwedge_{\substack{R^{\alpha}(\vec{u}) \in V \wedge \\ \text{rank}(R) \leq \text{rank}(Q)}} R^{\alpha}(\vec{u}) \wedge \bigwedge_{\substack{\neg R^{\alpha}(\vec{u}) \in V \wedge \\ \text{rank}(R) < \text{rank}(Q)}} \neg R^{\alpha}(\vec{u})$$

The function simply creates a conjunction of positive literals whose corresponding predicate has rank less or equal to the rank of the predicate Q as well as these negative literals whose corresponding predicate has rank strictly less than the rank of the predicate Q. The above restrictions on literals occurring in the resulting precondition are needed in order to ensure that for a stratified ALFP^{s}

doMagic($R^{\alpha}(\vec{u}), cls, S$)
  let cls' be an empty sequence of clauses
  foreach cl of the form $\forall \vec{x} : pre \Rightarrow P^{\beta}(\vec{v})$ **in** cls **do**
    $cls' := cls', \forall \vec{x} : \text{mkMagic}(P^{\beta}(\vec{v})) \wedge pre \Rightarrow P^{\beta}(\vec{v})$
    foreach $(V, \mathcal{Y}, Q^{\gamma}(\vec{w}))$ **in** S_{cl} **do**
      if $P^{\beta}(\vec{v}) \in V \wedge \text{rank}(P^{\beta}) \leq \text{rank}(Q^{\gamma})$ **then**
        $cls' := cls', \forall \vec{y} : \text{mkMagic}(P^{\beta}(\vec{v})) \wedge \text{mkPre}_Q(V) \Rightarrow \text{mkMagic}(Q^{\gamma}(\vec{w}))$
      else
        $cls' := cls', \forall \vec{y} : \text{mkPre}_Q(V) \Rightarrow \text{mkMagic}(Q^{\gamma}(\vec{w}))$
      fi
    od
  od
  return cls'

Figure 6.3: Magic set transformation.

clauses, the resulting clauses are also stratified. This approach essentially corresponds to performing the magic set transformation strata by strata. Notice that in the case when none of the literals satisfies the conditions, the result of mkPre is *true*. In the algorithm we also use S_{cl} to denote the subset of SIPS S associated with clause cl i.e. $S_{cl} \subseteq S$. The algorithm iterates over the adorned clauses, transforms them and accumulates the result in the set cls'. For each adorned clause cl, it first creates a new clause by inserting the corresponding magic predicate into the precondition. Furthermore, for each tuple in the SIPS associated with cl it creates a magic clause defining the corresponding magic predicate.

As mentioned at the beginning of this chapter, the novelty of the magic set algorithm for ALFP is the ability to handle universal quantification in preconditions. In our approach we treat all kinds of preconditions uniformly, which greatly simplifies the algorithm. The drawback is that some magic predicates may contain more irrelevant tuples. To illustrate this let us present the following example.

Example 12 *We consider two clauses. The first one uses existential quantification in precondition*

$$\forall x : (\exists y : E(x, y) \wedge S(y)) \Rightarrow S(x)$$

and the other uses universal quantification in precondition

$$\forall x : (\forall y : E(x, y) \wedge R(y)) \Rightarrow R(x)$$

Assume that E is a base predicate, and the queries we are interested in are $S(a)$ and $R(a)$ for some $a \in \mathcal{U}$. The adorned version of the queries are $S^b(a)$ and $R^b(a)$, and the adorned versions of the clauses are

$$\forall x : (\exists y : E(x, y) \wedge S^b(y)) \Rightarrow S^b(x)$$

and

$$\forall x : (\forall y : E(x, y) \wedge R^b(y)) \Rightarrow R^b(x)$$

The magic set transformation results in the following. For both queries we create seeds $magicS^b(a)$ and $magicR^b(a)$, respectively, and magic clauses

$$\forall x : \forall y : E(x, y) \wedge magicS^b(x) \Rightarrow magicS^b(y)$$

and

$$\forall x : \forall y : E(x, y) \wedge magicR^b(x) \Rightarrow magicR^b(y)$$

Furthermore, the original clauses are modified into

$$\forall x : magicS^b(x) \wedge (\exists y : E(x, y) \wedge S^b(y)) \Rightarrow S^b(x)$$

and

$$\forall x : magicR^b(x) \wedge (\forall y : E(x, y) \wedge R^b(y)) \Rightarrow R^b(x)$$

Since the magic clauses are exactly the same (modulo magic predicate names) in both cases, the magic predicates $magicS^b$ and $magicR^b$ contain exactly the same sets of tuples. Intuitively, the magic predicate $magicR^b$ should contain fewer tuples, since the precondition of the clause asserting predicate R is more restrictive. In the current version of the transformation, however, it is not the case, and in the presence of universal quantification in precondition the magic predicates may contain more irrelevant tuple.

Notice that for a stratified input formula, the result of the magic set transformation is also stratified. First we note that, as already mentioned, the rank of the magic predicate is equal to the rank of the corresponding derived predicate. The transformation alters the input clauses in two ways. First, it inserts magic predicate into the precondition of the clause asserting the given derived predicate. Since the ranks of both predicates are the same, the resulting clause is stratified. The second change is the addition of the clauses defining magic predicates. This clauses are also stratified by definition of the function mkPre and due to the fact that the magic literal mkMagic($P^\beta(\vec{v})$) is included in the precondition only in the case when rank(P^β) $\leq$ rank(Q^γ). Since both changes preserve stratification, the resulting clause sequence is stratified.

Notice also that the worst-case time complexity of solving the resulting clauses increases linearly. This is firstly because the maximal nesting depth of quantifiers does not change. Secondly, the number of new clauses (magic clauses) is proportional to the number of input clauses.

Continuing with the running example; the result of the magic set transformation is

$$\begin{aligned}
&MagicT^{bf}(s_1) \wedge \\
&\forall x : \forall z : MagicT^{bf}(x) \wedge E(x,z) \Rightarrow MagicT^{bf}(z) \qquad (6.3)\\
&\forall x : \forall y : MagicT^{bf}(x) \wedge E(x,y) \Rightarrow T^{bf}(x,y) \wedge \\
&\forall x : \forall y : MagicT^{bf}(x) \wedge (\exists z : E(x,z) \wedge T^{bf}(z,y)) \Rightarrow T^{bf}(x,y)
\end{aligned}$$

The first clause is the seed; created based on the supplied query. It is simply an assertion of the magic predicate with the constant corresponding to the bound argument of the query of interest. The second clause, (6.3), is a magic clause defining the magic predicate $MagicT^{bf}$. It corresponds to the following tuple in the SIPS

$$\{T^{bf}(x,y), E(x,z)\} \rightarrow_{\{z\}} T^{bf}(z,y)$$

and is obtained by applying function mkMagic to the adorned versions of derived predicates in the first and third component of the tuple. The precondition of the clause is a conjunction of literals in the first component of the tuple. The last two clauses are modified versions of the corresponding adorned clauses. They are a result of inserting the magic predicate corresponding to the asserted predicate into the precondition of each clause.

Notice that the last clause is equivalent to the following one

$$\forall x : \forall y : \forall z : MagicT^{bf}(x) \wedge E(x,z) \wedge T^{bf}(z,y) \Rightarrow T^{bf}(x,y) \qquad (6.4)$$

The result of the magic set transformation can be further optimized by performing common subexpression elimination. As an example notice that during evaluation of clause (6.4) the precondition from clause (6.3) is reevaluated. The inefficiency can be avoided by applying a supplementary magic sets algorithm that was introduced in [51] and then generalized in [8]. The general idea is to use this supplementary predicates to store the intermediate results that can be used later, and hence eliminate redundant computations. Continuing with the example, the result of applying the supplementary magic sets algorithm (using supplementary predicate Sup) to the above clauses results in the following clauses

$$\begin{aligned}
&MagicT^{bf}(s_1) \wedge \\
&\forall x : \forall z : MagicT^{bf}(x) \wedge E(x,z) \Rightarrow Sup(x,z) \\
&\forall x : \forall y : Sup(x,y) \Rightarrow MagicT^{bf}(y) \\
&\forall x : \forall y : MagicT^{bf}(x) \wedge E(x,y) \Rightarrow T^{bf}(x,y) \wedge \\
&\forall x : \forall y : \forall z : Sup(x,z) \wedge T^{bf}(z,y) \Rightarrow T^{bf}(x,y)
\end{aligned}$$

In the above, the result of evaluating the precondition of clause (6.3) is stored in the supplementary predicate Sup, which is then used to define predicate

$MagicT^{bf}$. Furthermore, in order to avoid redundant computations during evaluation of the precondition of clause (6.4) we use the supplementary predicate Sup and join it with predicate T^{bf}.

Even though, we do not present the supplementary magic sets algorithm here, in order to achieve better efficiency the algorithm should always be applied. For the detailed presentation of the algorithm, we the reader should refer to [8].

Proposition 6.8 *Let cls be a closed and stratified adorned ALFP*^{s} *formula and* $R^\alpha(\vec{v})$ *be a query on cls. Let cls′ be the result of the magic set transformation. Then* $cls \equiv_{R^\alpha(\vec{v})}^{R^\alpha(\vec{v})} cls'$.

Proof. See Appendix A.14.

Example 13 *As another example let us consider the following ALFP*^{s} *clauses*

$$\begin{array}{l} \forall x : P(x) \Rightarrow S(x) \\ \forall x : \forall y : S(y) \wedge E(x,y) \Rightarrow S(x) \end{array}$$

and let the query be $S(error)$. *The adorned version of the above clauses is as follows*

$$\begin{array}{l} \forall x : P(x) \Rightarrow S^b(x) \\ \forall x : \forall y : S^b(y) \wedge E(x,y) \Rightarrow S^b(x) \end{array}$$

The SIPS used to create the adorned clauses are

$$\begin{array}{l} \{S^b(x)\} \rightarrow_{\{x\}} P(x) \\ \{S^b(x)\} \rightarrow_{\{x\}} E(x,y) \\ \{S^b(x), E(x,y)\} \rightarrow_{\{y\}} S^b(y) \end{array}$$

The result of magic set transformation is

$$\begin{array}{l} MagicS^b(error) \\ \forall x : \forall y : MagicS^b(x) \wedge E(x,y) \Rightarrow MagicS^b(y) \\ \forall x : MagicS^b(x) \wedge P(x) \Rightarrow S^b(x) \\ \forall x : \forall y : MagicS^b(x) \wedge E(x,y) \wedge S^b(y) \Rightarrow S^b(x) \end{array}$$

The result of applying the supplementary magic sets algorithm (using supple-

mentary predicate Sup) to the above clauses is as follows

$$
\begin{aligned}
&MagicS^b(error)\\
&\forall x : \forall y : MagicS^b(x) \wedge E(x,y) \Rightarrow Sup(x,y)\\
&\forall x : \forall y : Sup(x,y) \Rightarrow MagicS^b(y)\\
&\forall x : MagicS^b(x) \wedge P(x) \Rightarrow S^b(x)\\
&\forall x : \forall y : Sup(x,y) \wedge S^b(y) \Rightarrow S^b(x)
\end{aligned}
$$

CHAPTER 7

Case study: Static Analysis

This chapter aims to show how logics and associated solvers introduced in this dissertation can be used for rapid prototyping of new static analyses. More precisely, we present that many analysis problems can be specified as logic programs using logics of this thesis. Since analysis specifications are generally written in a declarative style, logic programming presents an attractive model for producing executable specifications of analyses, and has been successfully used in practice [59, 11]. Furthermore, the declarative style of the specifications makes them easy to analyse for complexity and correctness.

This chapter is organized as follows. We begin in Section 7.1 by presenting LFP specification of the Bit-Vector Frameworks. We continue in Section 7.2 with LFP formulation of context-insensitive points-to analysis. Section 7.3 presents the constant propagation analysis and the corresponding LLFP specification. Finally, we present the specification of the interval analysis by means of LLFP in Section 7.4.

7.1 Bit-Vector Frameworks

Datalog has already been used for program analysis in compilers [59, 48, 56]. In this section we present how the LFP logic can be used to specify analyses that are instances of Bit-Vector Frameworks, which are a special case of the Monotone Frameworks [41, 32].

A Monotone Framework consists of (a) a property space that usually is a complete lattice L satisfying the Ascending Chain Condition, and (b) transfer functions, i.e. monotone functions from L to L. The property space is used to represent the data flow information, whereas transfer functions capture the behavior of actions. In the Bit-Vector Framework, the property space is a power set of some finite set and all transfer functions are of the form $f_n(x) = (x \backslash kill_n) \cup gen_n$.

Throughout this section we assume that a program is represented as a control flow graph [33, 41], which is a directed graph with one entry node (having no incoming edges) and one exit node (having no outgoing edges), called extremal nodes. The remaining nodes represent statements and have transfer functions associated with them. The control flow graphs were formally defined in Section 2.2.2.

Backward may analyses. Let us first consider backward may analyses expressed as an instance of the Monotone Frameworks. In the analyses, we require the least sets that solve the equations and we are able to detect properties satisfied by at least one path leading to the given node. The analyses use the reversed edges in the flow graph; hence the data flow information is propagated *against* the flow of the program starting at the exit node. The data flow equations are defined as follows

$$A(n) = \begin{cases} \iota & \text{if } n = n_{exit} \\ \bigcup\{f_n(A(n') \mid (n, n') \in E\} & \text{otherwise} \end{cases}$$

where $A(n)$ represents data flow information at the entry to the node n, E is a set of edges in the control flow graph, and ι is the initial analysis information. The first case in the above equation, initializes the exit node with the initial analysis information, whereas the second one joins the data flow information from different paths (using the revered flow). We use $\bigcup$ since we want be able detect properties satisfied by at least one path leading to the given node.

The LFP specification for backward may analyses consists of two conjuncts corresponding to two cases in the data flow equations. Since in case of may analyses we aim at computing the least solution, the specification is defined in terms of a *define* clause. The formula is given by

$$define \left(\begin{array}{c} \forall x : \iota(x) \Rightarrow A(n_{exit}, x) \\ \bigwedge_{(s,t) \in E} \forall x : (A(t, x) \wedge \neg kill_s(x)) \vee gen_s(x) \Rightarrow A(s, x) \end{array} \right)$$

The first conjunct initializes the exit node with initial analysis information, denoted by the predicate ι. The second one propagates data flow information agains the edges in the control flow graph, i.e. whenever we have an edge (s,t) in the control flow graph, we propagate data flow information from t to s, by applying the corresponding transfer function. More precisely, x holds at the entry to node s if it either holds at the entry to the node t (the successor of s) and is not killed at node s or it is generated at node s.

Notice that there is no explicit formula for combining analysis information from different paths, as it is the case in the data flow equations, but rather it is done implicitly. Suppose there are two distinct edges (s,p) and (s,q) in the flow graph, then we get

$$\forall x : \underbrace{(A(p,x) \wedge \neg kill_s(x)) \vee gen_s(x)}_{cond_p(x)} \Rightarrow A(s,x)$$
$$\forall x : \underbrace{(A(q,x) \wedge \neg kill_s(x)) \vee gen_s(x)}_{cond_q(x)} \Rightarrow A(s,x)$$

which is equivalent to

$$\forall x : cond_p(x) \vee cond_q(x) \Rightarrow A(s,x)$$

Forward must analyses. Let us now consider the general pattern for defining forward must analyses. Here we require the largest sets that solve the equations and we are able to detect properties satisfied by all paths leading to a given node. The analyses propagate the data flow information along the edges of the flow graph starting at the entry node. The data flow equations are defined as follows

$$A(n) = \begin{cases} \iota & \text{if } n = n_{entry} \\ \bigcap\{f_n(A(n')) \mid (n',n) \in E\} & \text{otherwise} \end{cases}$$

where $A(n)$ represents analysis information at the exit from the node n. Since we require the greatest solution, the greatest lower bound $\bigcap$ is used to combine information from different paths.

The corresponding LFP specification is obtained as follows

$$constrain \begin{pmatrix} \forall x : A(n_{entry}, x) \Rightarrow \iota(x) \\ \bigwedge_{(s,t)\in E} \forall x : A(t,x) \Rightarrow (A(s,x) \wedge \neg kill_t(x)) \vee gen_t(x) \end{pmatrix}$$

Since we aim at computing the greatest solution, the analysis is given by means of *constrain* clause. The first conjunct initializes the entry node with the initial analysis information, whereas the second one propagates the information along the edges in the control flow graph, i.e. whenever we have an edge (s,t) in the control flow graph, we propagate data flow information from s to t, by applying the corresponding transfer function. More precisely, if x holds at the exit from the node t then it either holds at the exit from the node s (the successor of t) and is not killed at node t or it is generated at node t.

The specifications of forward may and backward must analyses follow similar pattern. In the case of forward may analyses the data flow information is propagated along the edges of the flow graph and since we aim at computing the least solution, the analyses are given by means of *define* clauses

$$define \left(\begin{array}{c} \forall x : \iota(x) \Rightarrow A(n_{entry}, x) \\ \bigwedge_{(s,t)\in E} \forall x : (A(s,x) \wedge \neg kill_t(x)) \vee gen_t(x) \Rightarrow A(t,x) \end{array} \right)$$

Backward must analyses, on the other hand, use reversed edges in the flow graph and are specified using *constrain* clauses, since the analysis results are represented by the greatest sets satisfying given specifications

$$constrain \left(\begin{array}{c} \forall x : A(n_{exit}, x) \Rightarrow \iota(x) \\ \bigwedge_{(s,t)\in E} \forall x : A(s,x) \Rightarrow (A(t,x) \wedge \neg kill_s(x)) \vee gen_s(x) \end{array} \right)$$

In order to compute the least solution of the data flow equations, one can use a general iterative algorithm for Monotone Frameworks. The worst case complexity of the algorithm is $\mathcal{O}(|E|h)$, where $|E|$ is the number of edges in the control flow graph, and h is the height of the underlying lattice [41]. For Bit-Vector Frameworks the lattice is a powerset of a finite set $\mathcal{U}$; hence h is $\mathcal{O}(|\mathcal{U}|)$. This gives the complexity $\mathcal{O}(|E||\mathcal{U}|)$.

According to Proposition 4.6 the worst case time complexity of the LFP specification is $\mathcal{O}(|\varrho_0| + \sum_{1\leq i\leq |E|} |\mathcal{U}||cl_i|)$. Since the size of the clause cl_i is constant and the sum of cardinalities of predicates of rank 0 is $\mathcal{O}(|N|)$, where N is the number of nodes in the control flow graph, we get $\mathcal{O}(|N| + |E||\mathcal{U}|)$. Provided that $|E| > |N|$ we achieve $\mathcal{O}(|E||\mathcal{U}|)$ i.e. the same worst case complexity as the standard iterative algorithm.

It is common in compiler optimization that various analyses are performed at the same time. Since LFP logic has direct support for both least fixed points and greatest fixed points, we can perform both may and must analyses at the same time by splitting the analyses into separate layers.

At this point it is worth mentioning that the Bit-Vector Frameworks can also be expressed using ALFP. The specification of forward may analysis could be given by

$$\begin{array}{c} \forall x : \iota(x) \Rightarrow A(n_{entry}, x) \\ \bigwedge_{(s,t)\in E} \forall x : (A(s,x) \wedge \neg kill_t(x)) \vee gen_t(x) \Rightarrow A(t,x) \end{array}$$

Hence, it is exactly as the corresponding LFP specification except that there is no explicit *define* keyword. It is implicit, since in the case of ALFP we always compute the least solution. In the case of the backward must analysis the corresponding ALFP specification is more complicated since there is no direct support for the greatest fixpoints in the logic. To remedy that problem we need to dualize the specification, hence we give a specification that defines the complement of the relation of interest i.e. A^{C}. The idea is based on the following condition: $A^{\mathsf{C}}(n, x)$ holds if and only if $\neg A(n, x)$ holds. The complement relation A^{C} is defined as follows

$$\forall x : \neg\iota(x) \Rightarrow A^{\mathsf{C}}(n_{exit}, x)$$
$$\bigwedge_{(s,t)\in E} \forall x : (A^{\mathsf{C}}(t, x) \vee kill_s(x)) \wedge \neg gen_s(x) \Rightarrow A^{\mathsf{C}}(s, x)$$

Now we obtain the definition of the relation of interest by complementing the A^{C} predicate for each node in the corresponding control flow graph as follows

$$\bigwedge_{n\in N} \forall x : \neg A^{\mathsf{C}}(n, x) \Rightarrow A(n, x)$$

Based on the ALFP specification, it is evident that the use of LFP has many benefits. Firstly, the LFP specifications are particularly intuitive and can easily be extracted from the classical data flow equations, which is a great convenience for the programmer writing the specification. Secondly, due to direct support for the co-inductive specifications, in practice the analysis result can be computed more efficiently using LFP solver. This is because we do not need to compute the complement of the relation of interest first and then complement it, which in the case the relation A^{C} is sparse can be very expensive.

7.2 Points-to analysis

In this section we present LFP specification of points-to analysis, which forms the basis for many higher-level program analyses and is an integral part of compiler optimization frameworks. The analysis computes static approximation of the data that a pointer variable may reference at run time. Here we consider context-insensitive points-to analysis, which means that the control flow of the program is ignored and that statements can be executed in any order. We assume that the call graph is computed prior to the analysis.

Let us begin with a simple specification using three relations

$$Allocate(var, heap), Assign(to, from) \text{ and } PointsTo(var, heap)$$

Assume that the underlying program in addition to condition checking has two types of actions. First one is an allocation of a heap object and its assignment

$$
\begin{aligned}
&A\ a;\\
&A\ b = \textbf{new}\ A();\\
&a = b;
\end{aligned}
$$

Figure 7.1: Example program for points-to analysis.

to a variable, denoted $x = \textbf{new}\ X()$. The second one is an assignment of a variable $x = y$.

The relation $\mathit{Allocate}(\mathit{var}, \mathit{heap})$ expresses that a variable *var* references a heap object *heap*. More precisely, whenever we have an action $x = \textbf{new}\ X()$ in the program graph we generate a fact $\mathit{Allocate}(\mathit{var}, \mathit{heap})$, where *var* represents a variable x and *heap* corresponds to an allocated heap object.

Furthermore, $\mathit{Assign}(\mathit{to}, \mathit{from})$ captures that a variable represented by *to* is assigned a value of a variable represented by *from*. Thus, for each action $x = y$ in the program graph, we generate a fact $\mathit{Assign}(\mathit{to}, \mathit{from})$, where *to* represents variable x and *from* corresponds to variable y.

The relation *PointsTo* represents points-to information; it captures *possible* points-to relations from variables to heap objects. $\mathit{PointsTo}(\mathit{var}, \mathit{heap})$ is true if a variable *var* may point to a heap object *heap* an any point during program execution.

Based on these relations, a simple points-to analysis can be expressed using following LFP clauses

$$
\mathit{define}\left(\begin{array}{c}
(\forall x : \forall h : \mathit{Allocate}(x,h) \Rightarrow \mathit{PointsTo}(x,h)) \wedge \\
(\forall x : \forall h : (\exists y : \mathit{Assign}(x,y) \wedge \mathit{PointsTo}(y,h)) \Rightarrow \\
\hfill \mathit{PointsTo}(x,h))
\end{array}\right) \tag{7.1}
$$

The first clause initializes the *PointsTo* relation with the initial points-to information. The second clause computes the transitive closure of the *PointsTo* relation. Whenever a variable y can point to a heap object h and it is assigned to a variable x, then x can also point to a heap object h. Since the points-to information is a least model for the above clauses, the specification is given by means of *define* clauses.

As an example let us consider a simple program from Figure 7.1. Assume that the universe $\mathcal{U}$ contains the following values

$$\mathcal{U} = \{v_a, v_b, h_2\}$$

The values v_a and v_b represent variables a and b, respectively, whereas value h_2 represents heap object allocated in line 2 in the program. The object allocation in line 2 results in

$$\mathit{define}(\mathit{Allocate}(v_b, h_2))$$

The program has one assignment action, represented by

$$\mathit{define}(\mathit{Assign}(v_a, v_b))$$

A possible evaluation of the specification given in (7.1) based on the above facts is as follows. We begin by evaluating first clause, and we derive that $\mathit{PointsTo}(v_b, h_2)$ is true. Finally, by evaluating second clause we discover that $\mathit{PointsTo}(v_a, h_2)$ is true, since both $\mathit{Assign}(v_a, v_b)$ and $\mathit{PointsTo}(v_b, h_2)$ are true. As a result, it is clear that both variables a and b can point to the same heap object h_2.

Now, let us add fields to the language, and consider a refinement of the above analysis, where we add *field sensitivity* to the analysis. Namely the analysis will keep track of storing and loading objects to and form object instance fields. As a result we add three additional relations. The relation *Store* represents store actions $x.a = y$. More precisely, $\mathit{Store}(\mathit{base}, \mathit{field}, \mathit{from})$ expresses a store field action from a variable captured by *from*, to the object referenced by variable *base* in the field identified by *field*. If, for instance, the program graph contains an action labeled $x.a = y$, then *Store* contains a tuple with *base* being a representation of the variable x, *field* identifying a field a and *from* corresponding to y.

Similarly, the *Load* relation represents the load action. More precisely, the fact $\mathit{Load}(\mathit{to}, \mathit{base}, \mathit{field})$ expresses a load field action to a variable represented by *to*, from the object referenced by variable *base* in the field identified by *field*. Thus, whenever we have an action labeled $x = y.b$ in the program graph, then the relation *Load* contains a tuple (x, y, b).

The third relation, *FieldPointsTo*, is used to capture which heap object can point to which other heap object through a given field. Hence we have that $\mathit{FieldPointsTo}(\mathit{base}, \mathit{field}, \mathit{heap})$ is true if a heap object represented by *base* may point to a heap object *heap* through a given *field* at any point during program execution.

Now the above specification of points-to analysis can be extended by adding additional clauses as follows

$$
\begin{array}{l}
\mathit{define}(\\
\forall x : \forall h : \mathit{Allocate}(x,h) \Rightarrow \mathit{PointsTo}(x,h) \wedge \\
\forall x : \forall h : (\exists y : \mathit{Assign}(x,y) \wedge \mathit{PointsTo}(y,h)) \Rightarrow \mathit{PointsTo}(x,h) \wedge \\
\forall x : \forall h_x : \\
\quad (\exists y : \exists h_y : \exists f : \\
\qquad \mathit{Load}(x,y,f) \wedge \mathit{PointsTo}(y,h_y) \wedge \mathit{FieldPointsTo}(h_y,f,h_x)) \\
\qquad\qquad\qquad \Rightarrow \mathit{PointsTo}(x,h_x) \wedge \\
\forall h_x : \forall h_y : \forall f : \\
\quad (\exists x : \exists y : \mathit{Store}(x,f,y) \wedge \mathit{PointsTo}(x,h_x) \wedge \mathit{PointsTo}(y,h_y)) \\
\qquad\qquad\qquad \Rightarrow \mathit{FieldPointsTo}(h_x,f,h_y))
\end{array}
$$

where first two clauses remained unchanged. The third clause handles the load actions. Namely, given a statement $x = y.f$, if y may point to h_y and $h_y.f$ may point to h_x then x may point to h_x. The last clause models the effect of store actions. Whenever we have an action $x.f = y$, x may point to h_x and y may point to h_y then $h_x.f$ may point to h_y.

In order to illustrate the analysis in action, let us consider the simple program listed in Figure 7.2. Assume that the universe $\mathcal{U}$ contains the following values

$$\mathcal{U} = \{v_a, v_b, v_c, h_2, h_3, f\}$$

The values v_a, v_b and v_c represent variables a, b and c, respectively. Values h_2 and h_3 represent object allocated in lines 2 and 3 in the program. The field f of the type C is represented by value f. The initial facts for the program are as follows. The object allocations in lines 2 and 3 result in

$$\mathit{define}(\mathit{Allocate}(v_b, h_2) \wedge \mathit{Allocate}(v_c, h_3))$$

The program has one assignment action, represented by

$$\mathit{define}(\mathit{Assign}(v_a, v_b))$$

and one store action given by

$$\mathit{define}(\mathit{Store}(v_c, f, v_a))$$

The result of the points-to analysis corresponds to the least model of the specifiation above. Thus, let us show an example evaluation of the above clauses. We begin by evaluating first clause, and we derive that $\mathit{PointsTo}(v_b, h_2)$ and $\mathit{PointsTo}(v_c, h_3)$ are true. Next, by evaluating second clause we discover that $\mathit{PointsTo}(v_a, h_2)$ is true, since both $\mathit{Assign}(v_a, v_b)$ and $\mathit{PointsTo}(v_b, h_2)$ are true. Finally, using the last clause we find out that $\mathit{FieldPointsTo}(h_3, f, h_2)$ is true based on the truth of $\mathit{Store}(v_c, f, v_a)$, $\mathit{PointsTo}(v_c, h_3)$, and $\mathit{PointsTo}(v_a, h_2)$.

$$
\begin{aligned}
&A\ a;\\
&A\ b = \textbf{new}\ A();\\
&C\ c = \textbf{new}\ C();\\
&a = b;\\
&c.f = a;
\end{aligned}
$$

Figure 7.2: Example program for points-to analysis.

7.3 Constant propagation analysis

In this section we present an LLFP specification of constant propagation analysis. The purpose of the analysis is to determine for each program point whether or not a variable has a constant value whenever that point is reached during run-time execution. The analysis results can be used to perform an optimization called *constant folding*, which replaces variables that evaluate to a constant by that constant. In contrast to the analyses discussed in Section 7.1, constant propagation analysis is not distributive [41]. Recall that a function $f : \mathcal{L}_1 \to \mathcal{L}_2$ between partially ordered sets $\mathcal{L}_1$ and $\mathcal{L}_2$ is distributive (also called additive) if

$$\forall l_1, l_2 \in \mathcal{L}_1 : f(l_1 \sqcup_1 l_2) = f(l_1) \sqcup_2 f(l_2)$$

where $\sqcup_1$ and $\sqcup_2$ are binary least upper bound operators on corresponding posets.

As an example, consider the following statements

$$
\begin{aligned}
&x := 5;\\
&y := x + 10;\\
&\textbf{print}\ y;
\end{aligned}
$$

By performing constant propagation followed by constant folding the above statements results in

$$
\begin{aligned}
&x := 5;\\
&y := 15;\\
&\textbf{print}\ 15;
\end{aligned}
$$

which can be further optimized by dead code elimination of both x and y.

The analysis is defined over the following complete lattice

$$CP = ((Var \to \mathbb{Z}_\bot^\top), \sqsubseteq, \sqcup, \sqcap, \lambda x.\bot, \lambda x.\top)$$

where Var is a finite set of variables appearing in the program and $\mathbb{Z}_\bot^\top$ is a set of integers (possible values of these variables) extended with bottom and top elements i.e. $\mathbb{Z}_\bot^\top = \mathbb{Z} \cup \{\bot, \top\}$. The ordering is defined as follows

$$\forall z \in \mathbb{Z} : \bot \sqsubseteq_\mathbb{Z} z \sqsubseteq_\mathbb{Z} \top$$
$$\forall z_1, z_2 \in \mathbb{Z} : z_1 \sqsubseteq_\mathbb{Z} z_2 \Leftrightarrow z_1 = z_2$$

We use the bottom element $\bot$ to denote that a given variable is not initialized (no information is available about the possible value of the variable), which may mean e.g. unreachable code. The top element $\top$ indicates that a given variable is non-constant, whereas all other values $z \in \mathbb{Z}$ indicate that a value is z. Thus, $Var \to \mathbb{Z}_\bot^\top$ is an environment that maps variables to constants, in case a given variable is a constant, or indicates that a variable is uninitialized or non-constant, by means of $\bot$ and $\top$ elements, respectively.

The set $\mathbb{Z}_\bot^\top$ is indeed a complete lattice with the binary least upper bound operator given by

$$\forall z \in \mathbb{Z} : \bot \sqcup_\mathbb{Z} z = z = z \sqcup_\mathbb{Z} \bot$$
$$\forall z \in \mathbb{Z} : \top \sqcup_\mathbb{Z} z = \top = z \sqcup_\mathbb{Z} \top$$
$$\forall z_1, z_2 \in \mathbb{Z} : z_1 \neq z_2 \Rightarrow z_1 \sqcup_\mathbb{Z} z_2 = \bot$$
$$\forall z_1, z_2 \in \mathbb{Z} : z_1 = z_2 \Rightarrow z_1 \sqcup_\mathbb{Z} z_2 = z_1 = z_2$$

The partial order in the analysis lattice is defined as

$$\forall \sigma_1, \sigma_2 \in (Var \to \mathbb{Z}_\bot^\top) : \sigma_1 \sqsubseteq \sigma_2 \Leftrightarrow (\forall x \in Var : \sigma_1(x) \sqsubseteq_\mathbb{Z} \sigma_2(x))$$

The binary least upper bound operator is defined as follows

$$\forall \sigma_1, \sigma_2 \in (Var \to \mathbb{Z}_\bot^\top) : \forall x \in Var : (\sigma_1 \sqcup \sigma_2)(x) = \sigma_1(x) \sqcup_\mathbb{Z} \sigma_2(x)$$

In the following we again assume that the program of interest is represented as a control flow graph.

Since constant propagation is a forward analysis, and we are interested in the least solution, the data flow equations are defined as follows

$$A(n) = \begin{cases} \iota & \text{if } n = n_{entry} \\ \bigsqcup\{f_n(A(n')) \mid (n', n) \in E\} & \text{otherwise} \end{cases}$$

$[x := a]^n$	$f_n(\sigma) = \sigma[x \mapsto \mathcal{A}_{CP}[\![a]\!]\sigma]$
$[\mathbf{skip}]^n$	$f_n(\sigma) = \sigma$
$[b]^n$	$f_n(\sigma) = \sigma$

Table 7.1: Transfer functions for Constant Propagation Analysis.

$$
\begin{aligned}
\mathcal{A}_{CP}[\![x]\!]\sigma &= \sigma(x)\\
\mathcal{A}_{CP}[\![n]\!]\sigma &= n\\
\mathcal{A}_{CP}[\![a_1 \star a_2]\!]\sigma &= \mathcal{A}_{CP}[\![a_1]\!]\sigma \ \star_{\mathbb{Z}_\bot^\top}\ \mathcal{A}_{CP}[\![a_2]\!]\sigma
\end{aligned}
$$

Table 7.2: Function for analyzing expressions.

where $A(n)$ represents analysis information at the exit from the node n. The initial analysis information ι is defined as $\lambda x.\top$, hence it initializes all variables with non-constant value. Since we require the least solution, the least upper bound $\bigsqcup$ is used to combine information from different paths. The mapping of nodes to transfer functions is given in Table 7.1. The definition of transfer functions uses function $\mathcal{A}_{CP}$ for evaluating expressions, which is defined in Table 7.2. The operations on $\mathbb{Z}$ are lifted to $\mathbb{Z}_\bot^\top$ in the following way

$\star_{\mathbb{Z}_\bot^\top}$	$\bot$	z_2	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
z_1	$\bot$	$z_1 \star_{\mathbb{Z}} z_2$	$\top$
$\top$	$\bot$	$\top$	$\top$

where $\star_{\mathbb{Z}}$ is the corresponding arithmetic operation on $\mathbb{Z}$. Let us briefly explain the definition of the transfer functions in Table 7.1. In the case of an assignment the environment is updated with a new mapping for the assigned variable and mappings for all other variables remain unchanged. The transfer functions for **skip** and boolean conditions b are just identities; they propagate the environment unaltered.

Now let us turn into the LLFP specification of the analysis. For simplicity we assume that the assignments are in three-address form. The universe $\mathcal{U}$ is the union of all variables $x \in \mathit{Var}$, constants $z \in \mathbb{Z}$ appearing in the analysed program and the set of nodes in the underlying control flow graph. The complete lattice used is $(\mathbb{Z}_\bot^\top, \sqsubseteq_{\mathbb{Z}})$. The representation function $\beta : \mathcal{U} \to \mathbb{Z}_\bot^\top$ maps each constant $z \in \mathbb{Z}$ in the universe $\mathcal{U}$ into a lattice value z, whereas all other elements of the universe are mapped to $\perp$. Formally we have

$$\beta(a) = \begin{cases} a & \text{if } a \in \mathbb{Z} \\ \perp & \text{otherwise} \end{cases}$$

As we will see, in the case of constant propagation only the first case of the above definition is ever applied i.e. the application to an integer appearing in the analysed program.

The LLFP specification for the analysis is given by the predicate A. It consists of two kinds of clauses corresponding to the two cases in the data flow equations. The first equation that initializes the entry node with the initial analysis information corresponds to a conjunction of facts of the form

$$A(n_{entry}, x; \top)$$

for all $x \in Var$; hence it initializes all variables with non-constant value. Notice that for all other nodes and all variables the predicate A is implicitly initialized with the bottom element $\perp$. For the second equation we distinguish three cases depending on the kind of action, corresponding to the transfer functions. Whenever we have an edge $(s,t) \in E$ where $[x := y \star z]^t$ in the control flow graph we generate the clauses

$$\begin{array}{l} \forall v_y : \forall v_z : A(s, y; v_y) \wedge A(s, z; v_z) \Rightarrow A(t, x; f_\star(v_y, v_z)) \wedge \\ \forall w : \forall v : w \neq x \wedge A(s, w; v) \Rightarrow A(t, w; v) \end{array}$$

where function $f_\star : \mathbb{Z}_\perp^\top \times \mathbb{Z}_\perp^\top \to \mathbb{Z}_\perp^\top$ evaluates arithmetic operations. Similarly, for assignments of the form $[x := y \star c]^t$ and $[x := c]^t$ the corresponding LLFP clauses are

$$\begin{array}{l} \forall v_y : A(s, y; v_y) \Rightarrow A(t, x; f_\star(v_y, [c])) \wedge \\ \forall w : \forall v : w \neq x \wedge A(s, w; v) \Rightarrow A(t, w; v) \end{array}$$

and

$$A(t, x; [c]) \wedge (\forall w : \forall v : w \neq x \wedge A(s, w; v) \Rightarrow A(t, w; v))$$

respectively. Notice, that in the case the expressions in the control flow graph contain constants, we make use of the lattice terms $[u]$ in the resulting LLFP clauses. We do that in order to map constants from the universe into the corresponding lattice values. Moreover, whenever we have an edge $(s,t) \in E$ where $[b]^t$ or $[\textbf{skip}]^t$ in the control flow graph we generate the clause

$$\forall w : \forall v : A(s, w; v) \Rightarrow A(t, w; v)$$

$[x := 3]^{n_1};$
$\textbf{if } [x = y]^{n_2} \textbf{ then } [y := x + 2]^{n_3}; \textbf{ else } [y := x - 2]^{n_4};$
$[\textbf{skip}]^{n_5};$

Figure 7.3: Example program.

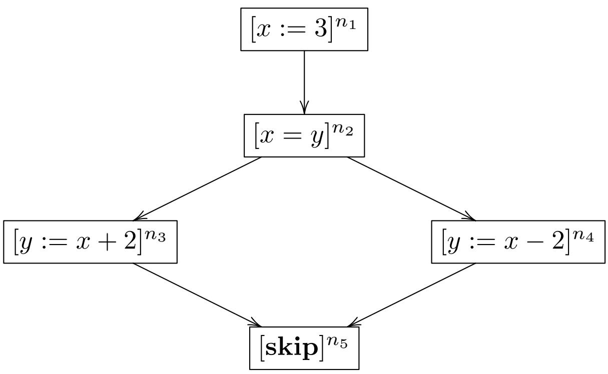


Figure 7.4: Control flow graph corresponding to program from Figure 7.3.

In order to present the analysis in action, consider a simple program from Figure 7.3. The corresponding control flow graph is depicted in Figure 7.4.

The LLFP clauses for constant propagation for this program are as follows. We have two clauses asserting the analysis predicate for the entry node

$$A(n_{entry}, x; \top) \wedge A(n_{entry}, y; \top)$$

The assignment in node n_1 gives rise to

$$A(n_1, x; 3) \wedge (\forall w : \forall v : w \neq x \wedge A(n_{entry}, w; v) \Rightarrow A(n_1, w; v))$$

The condition in node n_2 simply propagates the analysis information

$$\forall w : \forall v : A(n_1, w; v) \Rightarrow A(n_2, w; v)$$

The assignments in nodes n_3 and n_4 are captured by

$$\begin{array}{l} \forall v_x : A(n_2, x; v_x) \Rightarrow A(n_3, y; \mathrm{sum}(v_x, [2])) \wedge \\ \forall w : \forall v : w \neq y \wedge A(n_2, w; v) \Rightarrow A(n_3, w; v) \wedge \end{array}$$

and

$$\begin{array}{l} \forall v_x : A(n_2, x; v_x) \Rightarrow A(n_4, y; \mathrm{sum}(v_x, [-2])) \wedge \\ \forall w : \forall v : w \neq y \wedge A(n_2, w; v) \Rightarrow A(n_4, w; v) \wedge \end{array}$$

respectively. Finally the information from both branches of the *if* statement is joined using the following clauses

$$\begin{array}{l} \forall w : \forall v : A(n_3, w; v) \Rightarrow A(n_5, w; v) \wedge \\ \forall w : \forall v : A(n_4, w; v) \Rightarrow A(n_5, w; v) \end{array}$$

All the clauses are a straightforward application of the specification defined above. The result of the analysis, being the least model, is presented in Table 7.3. The assignment at node n_1 results in variable x being mapped to value 3. The assignments in nodes n_3 and n_4 cause variable y to be mapped to values 5 and 1, respectively. Finally, both mappings for variable y are joined at node n_5 resulting in the mapping to a non-constant value denoted by $\top$, which is as expected.

	x	y
n_{entry}	$\top$	$\top$
n_1	3	$\top$
n_2	3	$\top$
n_3	3	5
n_4	3	1
n_5	3	$\top$

Table 7.3: Analysis result.

7.4 Interval analysis

In this section we present Interval Analysis and show how it can be specified in LLFP. The purpose of the analysis is to determine for each program point an interval containing possible values of variables whenever that point is reached during run-time execution. The analysis results can be used for Array Bound Analysis, which determines whether an array index is always within the bounds of the array. If this is the case, a run-time check can safely be eliminated, which makes code more efficient.

We begin with defining the complete lattice that we later use to define the analysis. The lattice $(Interval, \sqsubseteq_I)$ of intervals is defined as follows. The underlying set is

$$Interval = \bot \cup \{[z_1, z_2] \mid z_1 \leq z_2, z_1 \in \mathbf{Z} \cup \{-\infty\}, z_2 \in \mathbf{Z} \cup \{\infty\}\}$$

where $\mathbf{Z}$ is a finite subset of integers, $\mathbf{Z} \subseteq \mathbb{Z}$, and the integer ordering $\leq$ on $\mathbb{Z}$ is extended to an ordering on $\mathbf{Z}' = \mathbf{Z} \cup \{-\infty, \infty\}$ by taking for all $z \in \mathbf{Z}$: $-\infty \leq z$, $z \leq \infty$ and $-\infty \leq \infty$. In the above definition, $\bot$ denotes an empty interval, whereas $[z_1, z_2]$ is the interval from z_1 to z_2 including the end points, where $z_1, z_2 \in \mathbf{Z}$. The interval $[-\infty, \infty]$ is equivalent to $\top$. In the following we use i to denote an interval from $Interval$.

The partial ordering $\sqsubseteq_I$ in $Interval$ uses operations inf and sup

$$\inf(i) = \begin{cases} \infty & \text{if } i = \bot \\ z_1 & \text{if } i = [z_1, z_2] \end{cases}$$

$$\sup(i) = \begin{cases} -\infty & \text{if } i = \bot \\ z_2 & \text{if } i = [z_1, z_2] \end{cases}$$

and is defined as

$$i_1 \sqsubseteq_I i_2 \text{ iff } \inf(i_2) \leq \inf(i_1) \wedge \sup(i_1) \leq \sup(i_2)$$

The intuition behind the partial ordering $\sqsubseteq$ in $Interval$ is that

$$i_1 \sqsubseteq_I i_2 \Leftrightarrow \{z \mid z \text{ belongs to } i_1\} \subseteq \{z \mid z \text{ belongs to } i_2\}$$

Hence the intervals over-approximate the set of possible values. For example, the set of possible values $\{2, 5, 7\}$ is represented by the interval $[2, 7]$.

The least upper bound operator is defined as follows

$$\bigsqcup_I Y = \begin{cases} \bot & \text{if } Y \subseteq \{\bot\} \\ [\inf'(\{\inf(i) \mid i \in Y\}), \sup'(\{\sup(i) \mid i \in Y\})] & \text{otherwise} \end{cases}$$

where $\inf'$ and $\sup'$ are the infimum and supremum operators on $\mathbf{Z}'$ corresponding to the ordering $\leq$ on $\mathbf{Z}'$. They are defined as

$$\inf'(Z) = \begin{cases} \infty & \text{if } Z = \emptyset \\ z' \in Z & \text{if } \forall z \in Z : z' \leq z \\ -\infty & \text{otherwise} \end{cases}$$

Similarly, the supremum operator is given by

$$\sup'(Z) = \begin{cases} -\infty & \text{if } Z = \emptyset \\ z' \in Z & \text{if } \forall z \in Z : z \leq z' \\ \infty & \text{otherwise} \end{cases}$$

The interval analysis is defined over the following complete lattice

$$IA = ((Var \to Interval), \sqsubseteq, \sqcup, \sqcap, \lambda x.\bot, \lambda x.\top)$$

where Var is a finite set of variables appearing in the program. The underlying set acts as an environment, mapping variables to intervals. Thus for each variable in the program it gives the interval of possible values. The partial ordering on environments is defined as follows

$$\forall \sigma_1, \sigma_2 \in (Var \to Interval) : \sigma_1 \sqsubseteq \sigma_2 \Leftrightarrow (\forall x \in Var : \sigma_1(x) \sqsubseteq_I \sigma_2(x))$$

where $\sqsubseteq_I$ is the partial ordering on intervals defined at the beginning of this section. The binary least upper bound is defined as follows

$$\forall \sigma_1, \sigma_2 \in (Var \to Interval) : \forall x \in Var : (\sigma_1 \sqcup \sigma_2)(x) = \sigma_1(x) \sqcup_I \sigma_2(x)$$

Similarly to other case studies in this chapter, we assume in the following that the program of interest is represented as a control flow graph.

$[x := a]^n$	$f_n(\sigma)$	$=$	$\sigma[x \mapsto \mathcal{A}_I[\![a]\!]\sigma]$
$[\mathbf{skip}]^n$	$f_n(\sigma)$	$=$	σ
$[b]^n$	$f_n(\sigma)$	$=$	σ

Table 7.4: Transfer functions for Interval Analysis.

In interval analysis, we require the least solution of the equations and we use the forward edges in the flow graph. The data flow equations are defined as follows

$$A(n) = \begin{cases} \iota & \text{if } n = n_{entry} \\ \bigsqcup\{f_n(A(n')) \mid (n', n) \in E\} & \text{otherwise} \end{cases}$$

where $A(n)$ represents analysis information at the exit from the node n. The initial analysis information ι is defined as $\lambda x.\top$, hence it initializes all variables with top value. The mapping of nodes to transfer functions is given in Table 7.4. The definition of transfer functions uses function $\mathcal{A}_I$ for evaluating expression, which is defined in Table 7.5. In the case of a variable x the function $\mathcal{A}_I$ simply returns the corresponding interval indicated by the environment σ. When evaluating constant c, a possible over-approximation of an interval $[c, c]$ is returned. This is because we need to make sure that both ends of the interval belong to $\mathbf{Z}$. We achieve that by using functions $\inf'$ and $\sup'$. Notice that in the case $c \in \mathbf{Z}$, the exact interval $[c, c]$ is returned. The case of arithmetic expressions is handled by first recursively evaluating the sub-expressions, followed by performing the corresponding arithmetic operation on returned intervals, denoted by $\star_I$. Finally, in order to make sure that the ends of the returned interval are in $\mathbf{Z}$, we again make use of functions $\inf'$ and $\sup'$.

Now, we shortly explain the definition of transfer functions from Table 7.4. In the case of assignment the transfer function returns an environment identical to σ, except that mapping for variable x is updated with the interval being the result of evaluating the arithmetical expression a. For two other kinds of actions the transfer functions are simply identities.

Now let us give an LLFP specification of interval analysis. The analysis is defined by the predicate A; the underlying universe $\mathcal{U}$ is a set of all variables $x \in \mathit{Var}$ and constants $z \in \mathbf{Z}$ appearing in the program, as well as the set of nodes in the control flow graph. The analysis is defined over the lattice $(\mathit{Interval}, \sqsubseteq_I)$, defined at the beginning of this section. The representation function β maps each constant $z \in \mathbf{Z}$ into an interval $[z, z]$, whereas all other elements of the

$$\begin{array}{lcl}\mathcal{A}_I[\![x]\!]\sigma & = & \sigma(x) \\ \mathcal{A}_I[\![c]\!]\sigma & = & [\sup'(\{z' \in \mathbf{Z} \mid z' \leq c\}), \inf'(\{z' \in \mathbf{Z} \mid c \leq z'\})] \\ \mathcal{A}_I[\![a_1 \star a_2]\!]\sigma & = & \big[\sup'(\{z' \in \mathbf{Z} \mid z' \leq z_1\}), \inf'(\{z' \in \mathbf{Z} \mid z_2 \leq z'\})\big] \\ & & \text{where } [z_1, z_2] = \mathcal{A}_I[\![a_1]\!]\sigma \star_I \mathcal{A}_I[\![a_2]\!]\sigma\end{array}$$

Table 7.5: Function for analyzing expressions.

universe are mapped to $\perp$. Formally we have

$$\beta(a) = \begin{cases} [a, a] & \text{if } a \in \mathbf{Z} \\ \perp & \text{otherwise} \end{cases}$$

The first case in the above definition is obvious i.e. each constant $z \in \mathbf{Z}$ is mapped into an interval $[z, z]$. The second case is defined only so that β is defined for all elements in the universe. As we shall see in the ananlysis specification, β is only applied to constants $z \in \mathbf{Z}$.

The specification consists of two kinds of clauses corresponding to two cases in the data flow equations. The following clause schema corresponds to the first equation, and an instance of it is generated for each variable $x \in \mathit{Var}$ appearing in the program

$$A(n_{entry}, x; \top)$$

Intuitively, it captures the fact that at the beginning all variables may have all possible values, here denoted by interval $\top$. For the second equation schema we distinguish three cases depending on the corresponding action. Whenever we have an edge $(s, t) \in E$ where $[x := y \star z]^t$ in the control flow graph we generate the clauses

$$\begin{array}{l}\forall i_y : \forall i_z : A(s, y; i_y) \land A(s, z; i_z) \Rightarrow A(t, x; f_\star(i_y, i_z)) \\ \forall w : \forall i : w \neq x \land A(s, w; i) \Rightarrow A(t, w; i)\end{array}$$

where function $f_\star : \mathit{Interval} \times \mathit{Interval} \to \mathit{Interval}$ evaluates the arithmetic operation $\star$. The first clause expresses that if at the exit from node s variables y and z are mapped to intervals i_y and i_z, respectively, then at the exit from node t variable x gets mapped to $f_\star(i_y, i_z)$. The second clause propagates the analysis information for all variables except x, which is assigned at node t. Similarly, for assignments of the form $[x := y \star c]^t$ and $[x := c]^t$ the corresponding LLFP clauses are

$$\begin{array}{l}\forall i_y : A(s, y; i_y) \Rightarrow A(t, x; f_\star(i_y, [c])) \\ \forall w : \forall i : w \neq x \land A(s, w; i) \Rightarrow A(t, w; i)\end{array}$$

$[x := 3]^{n_1}$;
if $[x = y]^{n_2}$ **then** $[y := x + 2]^{n_3}$; **else** $[y := x - 2]^{n_4}$;
$[\textbf{skip}]^{n_5}$;

Figure 7.5: Example program.

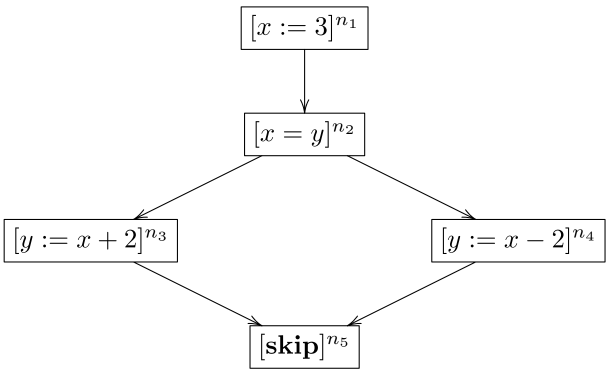


Figure 7.6: Control flow graph corresponding to program from Figure 7.5.

and

$$A(t, x; [c]) \land (\forall w : \forall i : w \neq x \land A(s, w; i) \Rightarrow A(t, w; i))$$

respectively. Notice, that in the case the expressions in the control flow graph contain constants, we make use of the lattice terms $[u]$ in the resulting LLFP clauses. We do that in order to map constants from the universe into the corresponding lattice values. Moreover, whenever we have an edge $(s, t) \in E$ where $[b]^t$ or $[\textbf{skip}]^t$ in the control flow graph we generate the clause

$$\forall w : \forall i : A(s, w; i) \Rightarrow A(t, w; i)$$

The clause simply propagates the analysis information without altering it.

In order to present interval analysis in action, consider the simple program in Figure 7.5. The corresponding control flow graph is depicted in Figure 7.6.

The LLFP specification of the interval analysis for the program is as follows. First we have two clauses initializing the entry node

$$A(n_{entry}, x; \top) \land A(n_{entry}, y; \top)$$

The assignment in node n_1 gives rise to the following clause

$$A(n_1, x; [3]) \wedge (\forall w : \forall v : w \neq x \wedge A(n_{entry}, w; v) \Rightarrow A(n_1, w; v))$$

The condition in node n_2 simply propagates the analysis information as follows

$$\forall w : \forall i : A(n_1, w; i) \Rightarrow A(n_2, w; i)$$

Two assignments in nodes n_3 and n_4 give rise to

$$\begin{array}{l} \forall i_x : A(n_2, x; i_x) \Rightarrow A(n_3, y; \mathrm{sum}(i_x, [2])) \wedge \\ \forall w : \forall i : w \neq y \wedge A(n_2, w; i) \Rightarrow A(n_3, w; i) \wedge \end{array}$$

and

$$\begin{array}{l} \forall i_x : A(n_2, x; i_x) \Rightarrow A(n_4, y; \mathrm{sum}(i_x, [-2])) \wedge \\ \forall w : \forall i : w \neq y \wedge A(n_2, w; i) \Rightarrow A(n_4, w; i) \wedge \end{array}$$

where function sum : *Interval* $\times$ *Interval* $\rightarrow$ *Interval* is defined as

$$\mathrm{sum}(int_1, int_2) = [\inf(int_1) + \inf(int_2), \sup(int_1) + \sup(int_2)]$$

Finally, for node n_5 we have two clauses propagating the analysis information from two branches of the *if* statement

$$\begin{array}{l} \forall w : \forall i : A(n_3, w; i) \Rightarrow A(n_5, w; i) \wedge \\ \forall w : \forall i : A(n_4, w; i) \Rightarrow A(n_5, w; i) \end{array}$$

By evaluating the above clauses we obtain the analysis result presented in Table 7.6. The assignment at node n_1 results in variable x being mapped to interval $[3, 3]$. The assignments in nodes n_3 and n_4 cause variable y to be mapped to intervals $[5, 5]$ and $[1, 1]$, respectively. Finally, both mappings for variable y are joined at node n_5 resulting in the mapping to the interval $[1, 5]$, as expected.

	x	y
n_{entry}	$\top$	$\top$
n_1	$[3,3]$	$\top$
n_2	$[3,3]$	$\top$
n_3	$[3,3]$	$[5,5]$
n_4	$[3,3]$	$[1,1]$
n_5	$[3,3]$	$[1,5]$

Table 7.6: Analysis result for the program in Figure 7.5.

Chapter 8

Case study: Model Checking

In this chapter we present applications of the LFP logic to specify model checking problems for two modal logics: Computation Tree Logic (CTL) [17] and Action Computation Tree Logic (ACTL). By doing so, we show that LFP may be used to specify a prototype model checker. We believe that this chapter enhances our understanding of the interplay between static analysis and model checking by showing that model checking can be seen as static analysis of modal logic formulae. This chapter builds on the developments of Steffen and Schmidt [53, 52] on one hand, and on work by Nielson and Nielson [42] on the other.

8.1 CTL Model Checking

This section is concerned with the application of the LFP logic to the CTL model checking problem [4]. In particular we show how LFP can be used to specify a prototype model checker for a special purpose modal logic of interest. Here we illustrate the approach on the familiar case of Computation Tree Logic (CTL) [17]. Throughout this section, we assume that the transition system, defined in Section 2.2, is finite and has no terminal states.

CTL distinguishes between state formulae and path formulae. CTL state formulae Φ over the set AP of atomic propositions and CTL path formulae φ are formed according to the following grammar

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbf{E}\varphi \mid \mathbf{A}\varphi$$
$$\varphi ::= \mathbf{X}\Phi \mid \Phi_1\mathbf{U}\Phi_2 \mid \mathbf{G}\Phi$$

where $a \in AP$. The satisfaction relation $\models$ is defined for state formula by

$$\begin{array}{lll} s \models \mathbf{true} & \underline{\texttt{iff}} & true \\ s \models a & \underline{\texttt{iff}} & a \in L(s) \\ s \models \neg\Phi & \underline{\texttt{iff}} & \text{not } s \models \Phi \\ s \models \Phi_1 \wedge \Phi_2 & \underline{\texttt{iff}} & s \models \Phi_1 \text{ and } s \models \Phi_2 \\ s \models \mathbf{E}\varphi & \underline{\texttt{iff}} & \pi \models \varphi \text{ for some } \pi \in Paths(s) \\ s \models \mathbf{A}\varphi & \underline{\texttt{iff}} & \pi \models \varphi \text{ for all } \pi \in Paths(s) \end{array}$$

where $Paths(s)$, defined in Section 2.2, denote the set of maximal path fragments π starting in s. The maximal path fragment is an infinite path fragment (since we assumed that the are no terminal states) that cannot be prolonged. The satisfaction relation $\models$ for path formulae is defined by

$$\begin{array}{lll} \pi \models \mathbf{X}\Phi & \underline{\texttt{iff}} & \pi[1] \models \Phi \\ \pi \models \Phi_1\mathbf{U}\Phi_2 & \underline{\texttt{iff}} & \exists j \geq 0 : (\pi[j] \models \Phi_2 \wedge (\forall 0 \leq k < j : \pi[k] \models \Phi_1)) \\ \pi \models \mathbf{G}\Phi & \underline{\texttt{iff}} & \forall j \geq 0 : \pi[j] \models \Phi \end{array}$$

where for path $\pi = s_0 s_1 \ldots$ and an integer $i \geq 0$, $\pi[i]$ denotes the i-th state of π, i.e. $\pi[i] = s_i$.

Now, let us briefly explain the sematnics of CTL. We begin with the state formulas. The boolean value **true** is satisfied by all states. An atomic proposition a holds in a state s if and only if state s is labelled with a by the labelling function L. A state s satisfies the formula $\neg\Phi$ if and only if s does not satisfy Φ. The formula $\mathbf{E}\varphi$ is valid in state s if and only if there exists a path starting in s that satisfies φ. Finally, $\mathbf{A}\varphi$ is valid in state s if and only if all paths starting in s satisfy φ. Now, let us turn into the path formulae. The path formula $\mathbf{X}\Phi$ is valid for a path π if and only if Φ is valid in the first state of that path i.e. state $\pi[1]$. The formula $\Phi_1\mathbf{U}\Phi_2$ is valid for a path π if and only if π has an initial finite prefix such that Φ_2 holds in the last state of that prefix and Φ_1 holds in all the other states of that prefix. Finally, the path formula $\mathbf{G}\Phi$ is valid for path π if and only if for each state on that path the formula Φ holds.

As an example let us consider a transition system depicted in Figure 8.1. The set of atomic propositions is $AP = \{a, b\}$. The labeling function is defined as follows

$$L(s_1) = \{a, b\}, L(s_2) = \{a\}, L(s_3) = \{b\}$$

Table 8.1 contains example CTL formulae with the corresponding sets of states satisfying them. The formula $\mathbf{EX}(a \wedge b)$ is valid in the state s_2 since there is

Φ	$\{s \mid s \models \Phi\}$
$\mathbf{EX}(a \wedge b)$	$\{s_2\}$
$\mathbf{AX}b$	$\{s_2, s_3\}$
$\mathbf{E}\neg a\mathbf{U}b$	$\{s_1, s_3\}$
$\mathbf{A}(a \wedge b)\mathbf{U}(a \wedge \neg b)$	$\{s_1\}$
$\mathbf{EG}a$	$\{s_1, s_2\}$
$\mathbf{AG}\neg a$	$\{s_3\}$

Table 8.1: Example CTL formulae.

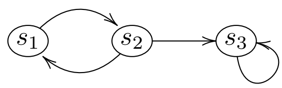


Figure 8.1: Example for the semantics of CTL.

a transition from s_2 to s_1 in which both a and b hold. The formula $\mathbf{AX}b$ is valid in states s_2 and s_3 since all outgoing transitions from both states lead to states satisfying b. The formula $\mathbf{E}\neg a\mathbf{U}b$ is valid in states s_1 and s_3 since b holds in both of them. The formula $\mathbf{A}(a \wedge b)\mathbf{U}(a \wedge \neg b)$ is valid only in state s_1 because both a and b hold in s_1 and the only outgoing transition leads to state s_2 where a hold and b does not hold. The formula $\mathbf{EG}a$ is valid in the states s_1 and s_2 since there is an infinite path $s_1 s_2 s_1 s_2 \ldots$ along which a holds globally. Finally, the formula $\mathbf{AG}\neg a$ holds in state s_3 since the only path starting in s_3 is $s_3 s_3 s_3 \ldots$ along which a does not hold.

CTL model checking amounts to a recursive computation of the set $Sat(\Phi)$ of all states satisfying Φ, which is sometimes referred to as *global* model checking. The algorithm boils down to a bottom-up traversal of the abstract syntax tree of the CTL formula Φ. The nodes of the abstract syntax tree correspond to the sub-formulae of Φ, and leaves are either a constant *true* or an atomic proposition $a \in AP$.

Now let us consider the corresponding LFP specification, where for each formula Φ we define a relation $Sat_\Phi \subseteq S$ characterizing states where Φ hold. The specification is defined in Table 8.2. Notice that there is only a finite number of subformulae; hence there are only finitely many relations Sat_Φ to be defined. The clause for *true* is straightforward and says that *true* holds in all states. The clause for an atomic proposition a expresses that a state satisfies a whenever it is in L_a, where we assume that we have a predicate $L_a \subseteq S$ for each $a \in AP$. The clause for $\Phi_1 \wedge \Phi_2$ captures that a state satisfies $\Phi_1 \wedge \Phi_2$ whenever it satisfies both Φ_1 and Φ_2. Similarly a state satisfies $\neg\Phi$ if it does not satisfy Φ. The formula

$$define(\forall s : Sat_{true}(s))$$
$$define(\forall s : L_a(s) \Rightarrow Sat_a(s))$$
$$define(\forall s : Sat_{\Phi_1}(s) \wedge Sat_{\Phi_2}(s) \Rightarrow Sat_{\Phi_1 \wedge \Phi_2}(s))$$
$$define(\forall s : \neg Sat_{\Phi}(s) \Rightarrow Sat_{\neg\Phi}(s))$$

$$define(\forall s : (\exists s' : T(s, s') \wedge Sat_{\Phi}(s')) \Rightarrow Sat_{\mathbf{EX}\Phi}(s))$$

$$define(\forall s : (\forall s' : \neg T(s, s') \vee Sat_{\Phi}(s')) \Rightarrow Sat_{\mathbf{AX}\Phi}(s))$$

$$define\left(\begin{array}{l} (\forall s : Sat_{\Phi_2}(s) \Rightarrow Sat_{\mathbf{E}[\Phi_1 \mathbf{U} \Phi_2]}(s)) \wedge \\ (\forall s : Sat_{\Phi_1}(s) \wedge (\exists s' : T(s, s') \wedge Sat_{\mathbf{E}[\Phi_1 \mathbf{U} \Phi_2]}(s')) \Rightarrow Sat_{\mathbf{E}[\Phi_1 \mathbf{U} \Phi_2]}(s)) \end{array} \right)$$

$$define\left(\begin{array}{l} (\forall s : Sat_{\Phi_2}(s) \Rightarrow Sat_{\mathbf{A}[\Phi_1 \mathbf{U} \Phi_2]}(s)) \wedge \\ (\forall s : Sat_{\Phi_1}(s) \wedge (\forall s' : \neg T(s, s') \vee Sat_{\mathbf{A}[\Phi_1 \mathbf{U} \Phi_2]}(s')) \Rightarrow Sat_{\mathbf{A}[\Phi_1 \mathbf{U} \Phi_2]}(s)) \end{array} \right)$$

$$constrain\left(\begin{array}{l} (\forall s : Sat_{\mathbf{EG}\Phi}(s) \Rightarrow Sat_{\Phi}(s)) \wedge \\ (\forall s : Sat_{\mathbf{EG}\Phi}(s) \Rightarrow (\exists s' : T(s, s') \wedge Sat_{\mathbf{EG}\Phi}(s'))) \end{array} \right)$$

$$constrain\left(\begin{array}{l} (\forall s : Sat_{\mathbf{AG}\Phi}(s) \Rightarrow Sat_{\Phi}(s)) \wedge \\ (\forall s : Sat_{\mathbf{AG}\Phi}(s) \Rightarrow (\forall s' : \neg T(s, s') \vee Sat_{\mathbf{AG}\Phi}(s'))) \end{array} \right)$$

Table 8.2: LFP specification of satisfaction sets.

for $\mathbf{EX}\Phi$ captures that a state s satisfies $\mathbf{EX}\Phi$, if there is a transition to state s' such that s' satisfies Φ. The formula for $\mathbf{AX}\Phi$ expresses that a state s satisfies $\mathbf{AX}\Phi$ if for all states s': either there is no transition from s to s', or otherwise s' satisfies Φ. The formula for $\mathbf{E}[\Phi_1\mathbf{U}\Phi_2]$ captures two possibilities. If a state satisfies Φ_2 then it also satisfies $\mathbf{E}[\Phi_1\mathbf{U}\Phi_2]$. Alternatively if the state s satisfies Φ_1 and there is a transition to a state satisfying $\mathbf{E}[\Phi_1\mathbf{U}\Phi_2]$ then s also satisfies $\mathbf{E}[\Phi_1\mathbf{U}\Phi_2]$. The formula $\mathbf{A}[\Phi_1\mathbf{U}\Phi_2]$ also captures two cases. If a state satisfies Φ_2 then it also satisfies $\mathbf{A}[\Phi_1\mathbf{U}\Phi_2]$. Alternatively state s satisfies $\mathbf{A}[\Phi_1\mathbf{U}\Phi_2]$ if it satisfies Φ_1 and for all states s' either there is no transition from s to s' or $\mathbf{A}[\Phi_1\mathbf{U}\Phi_2]$ is valid in s'. Let us now consider the formula for $\mathbf{EG}\Phi$. Since the set of states satisfying $\mathbf{EG}\Phi$ is defined as a largest set satisfying the semantics of $\mathbf{EG}\Phi$, the property is defined by means of a *constrain* clause. The first conjunct expresses that whenever a state satisfies $\mathbf{EG}\Phi$ it also satisfies Φ. The second conjunct says that if a state satisfies $\mathbf{EG}\Phi$ then there exists a transition to a state s' such that s' satisfies $\mathbf{EG}\Phi$. Finally let us consider the formula for $\mathbf{AG}\Phi$, which is also defined in terms of constrain clause and distinguishes between two cases. In the first one whenever a state satisfies $\mathbf{AG}\Phi$, it also satisfies Φ. Alternatively, if a state s satisfies $\mathbf{AG}\Phi$ then for all states s': either there is no transition from s to s' or otherwise s' satisfies $\mathbf{AG}\Phi$.

Generating clauses for Sat_Φ is performed by postorder traversal of Φ; hence the clauses defining sub-formulas of Φ are defined in the lower layers. More precisely, the relations corresponding to the sub-formulae of Φ have lower ranks, and hence are asserted before the relation Sat_Φ corresponding to the formula Φ. It can also be shown that the LFP clauses constructed for a CTL formula are both *closed* and *stratified*. It can be accomplished analogously to Section 4 in [42]. Moreover, it is important to note that the specification in Table 8.2 is both correct and precise. By correctness we mean that whenever we have $s \models \Phi$, then $s \in \varrho(Sat_\Phi)$, where ϱ is the least model of the corresponding LFP clauses. In addition to correctness, the LFP specification is precise, which is not usually the case in static analysis where we usually have an over-approximate result. More precisely, whenever $s \in \varrho(Sat_\Phi)$ in the least model ϱ of the LFP clauses, then $s \models \Phi$. It follows that an implementation of the given specification of CTL by means of the LFP solver constitutes a model checker for CTL.

We may estimate the worst-case time complexity of model checking performed using LFP. Consider a CTL formula Φ of size $|\Phi|$; it is immediate that the LFP clause has size $\mathcal{O}(|\Phi|)$, and the nesting depth is at most 2. According to Proposition 4.6 the worst case time complexity of the LFP specification is $\mathcal{O}(|S|+|S|^2|\Phi|)$, where $|S|$ is the number of states in the transition system. Using a more refined reasoning than that of Proposition 4.6 we obtain $\mathcal{O}(|S|+|T||\Phi|)$, where $|T|$ is the number of transitions in the transition system. It is due to the fact that the "double quantifications" over states in Table 8.2 really correspond to traversing all possible transitions rather than all pairs of states. Thus our

LFP model checking algorithm has the same worst case complexity as classical model checking algorithms [4].

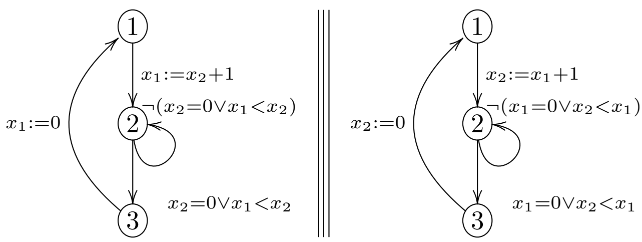
As an example let us consider the Bakery mutual exclusion algorithm [35]. Although the algorithm is designed for an arbitrary number of processes, we consider the simpler setting with two processes. Let P_1 and P_2 be the two processes, and x_1 and x_2 be two shared variables both initialized to 0. We can represent the algorithm as an interleaving of two program graphs [4], which are directed graphs where actions label the edges rather than the nodes. The algorithm is as follows

	‖	

while *true* **do**
  $x_1 := x_2 + 1;$
  while $\neg(x_2 = 0 \vee x_1 < x_2)$ **do**
    skip;
  od
  $c := \cdots ;$
  $x_1 := 0;$
od
	‖	**while** *true* **do**
		  $x_2 := x_1 + 1;$
		  while $\neg(x_1 = 0 \vee x_2 < x_1)$ **do**
		    skip;
		  od
		  $c := \cdots ;$
		  $x_2 := 0;$
		od

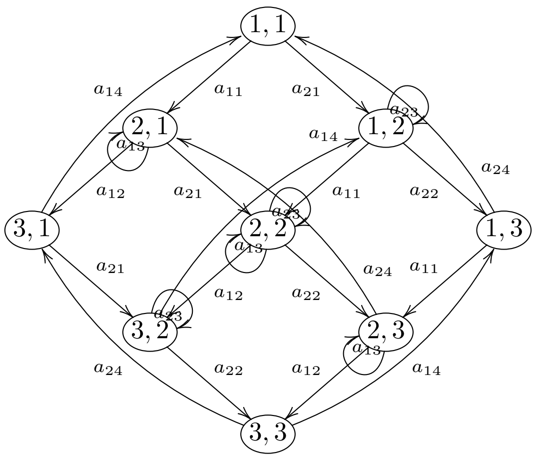
The variables x_1 and x_2 are used to resolve the conflict when both processes want to enter the critical section. When x_i is equal to zero, the process P_i is not in the critical section and does not attempt to enter it — the other one can safely proceed to the critical section. Otherwise, if both shared variables are non-zero, the process with smaller "ticket" (i.e. value of the corresponding variable) can enter the critical section. This reasoning is captured by the conditions of busy-waiting loops. When a process wants to enter the critical section, it simply takes the next "ticket" hence giving priority to the other process.

The corresponding representation of the two processes as program graphs is given by



From the algorithm above, we can obtain a program graph corresponding to the interleaving of the two processes, which is depicted in Figure 8.2.

The CTL formulation of the mutual exclusion property is $AG\neg(crit_1 \wedge crit_2)$, which states that along all paths globally it is never the case that $crit_1$ and $crit_2$ hold at the same time.



$$
\begin{array}{ll}
a_{11}: & x_1 := x_2 + 1 \\
a_{12}: & (x_2 = 0) \vee (x_1 < x_2) \\
a_{13}: & \neg((x_2 = 0) \vee (x_1 < x_2)) \\
a_{14}: & x_1 := 0 \\
a_{21}: & x_2 := x_1 + 1 \\
a_{22}: & (x_1 = 0) \vee (x_2 < x_1) \\
a_{23}: & \neg((x_1 = 0) \vee (x_2 < x_1)) \\
a_{24}: & x_2 := 0
\end{array}
$$

Figure 8.2: Interleaved program graph.

As already mentioned, in order to specify the problem we proceed bottom up by specifying formulae for the sub problems. After simplification we obtain the following LFP clauses

$$
\begin{array}{l}
\textit{define}(\forall s : L_{crit_1}(s) \wedge L_{crit_2}(s) \Rightarrow Sat_{crit}(s)), \\
\textit{constrain}\left(\begin{array}{l} (\forall s : Sat_{AG(\neg crit)}(s) \Rightarrow \neg Sat_{crit}(s)) \wedge \\ (\forall s : Sat_{AG(\neg crit)}(s) \Rightarrow (\forall s' : \neg T(s, s') \vee Sat_{AG(\neg crit)}(s'))) \end{array}\right)
\end{array}
$$

where relation L_{crit_1} (respectively L_{crit_1}) characterizes states in the interleaved program graph that correspond to process P_1 (respectively P_2) being in the critical section. Furthermore, the AG modality is defined by means of a constrain clause. The first conjunct expresses that whenever a state satisfies a mutual exclusion property $AG(\neg crit)$ it does not satisfy $crit$. The second one states that if a state satisfies a mutual exclusion property then all successors do as well, i.e. for an arbitrary state, it is either not a successor or else satisfies the mutual exclusion property.

8.2 ACTL Model Checking

In this section we present the application of LFP to the *Action Computation Tree Logic* (ACTL) model checking [40]. The developments in this section follows the work presented in [42], where ALFP logic was used. The advantage of using LFP is the ability of directly expressing formulas characterized by the greatest fixpoints, which was not directly possible using ALFP.

The prerequisite for model checking is a model of the system under consideration. Here, we assume that the underlying model is given by a labelled transition system (LTS) as defined in Section 2.2. We also assume that the underlying labelled transition system is finite and has no terminal states.

First let us present syntax and semantics of ACTL. Similarly to the syntax of CTL, we distinguish between state and path formulae. ACTL state formulae over the set of atomic propositions AP are formed according to the following grammar

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \mathbf{E}\varphi \mid \mathbf{A}\varphi$$

where $a \in AP$ and φ is a path formula. ACTL path formulae are formed according to the following grammar

$$\varphi ::= \mathbf{X}_\Omega\Phi \mid \Phi_1{}_{\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2 \mid \mathbf{G}_\Omega\Phi$$

where Φ, Φ_1 and Φ_2 are state formulae, and Ω, Ω_1 and Ω_2 are subsets of Act. The satisfaction relation $\models$ is defined for state formula by

$$
\begin{array}{lll}
s \models \mathbf{true} & \underline{\texttt{iff}} & true \\
s \models a & \underline{\texttt{iff}} & a \in L(s) \\
s \models \neg\Phi & \underline{\texttt{iff}} & \text{not } s \models \Phi \\
s \models \Phi_1 \wedge \Phi_2 & \underline{\texttt{iff}} & s \models \Phi_1 \text{ and } s \models \Phi_2 \\
s \models \mathbf{E}\varphi & \underline{\texttt{iff}} & \rho \models \varphi \text{ for some } \rho \in \mathit{Execs}(s) \\
s \models \mathbf{A}\varphi & \underline{\texttt{iff}} & \rho \models \varphi \text{ for all } \rho \in \mathit{Execs}(s)
\end{array}
$$

where $\mathit{Execs}(s)$, defined in Section 2.2 denote the set of maximal execution fragments ρ starting in s. The satisfaction relation $\models$ for path formulae is defined by

$$
\begin{array}{lll}
\rho \models \mathbf{X}_\Omega\Phi & \underline{\texttt{iff}} & \rho_{Act}[0] \in \Omega \wedge \rho_S[1] \models \Phi \\
\rho \models \Phi_1{}_{\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2 & \underline{\texttt{iff}} & \exists j \geq 0 : (\rho_{Act}[j] \in \Omega_2 \wedge \rho_S[j+1] \models \Phi_2 \wedge \\
 & & (\forall 0 \leq k < j : \rho_{Act}[k] \in \Omega_1 \wedge \rho_S[k+1] \models \Phi_1)) \\
\rho \models \mathbf{G}_\Omega\Phi & \underline{\texttt{iff}} & \forall j \geq 0 : \rho_{Act}[j] \in \Omega \Rightarrow \rho_S[j+1] \models \Phi
\end{array}
$$

where for execution $\rho = s_0\alpha_0 s_1\alpha_1 \ldots$ and $i \geq 0$, $\rho_S[i]$ and $\rho_{Act}[i]$ denote the i-th state and action of ρ, respectively; i.e. $\rho_S[i] = s_i$ and $\rho_{Act}[i] = \alpha_i$.

As an example, let us now consider the labelled transition system $(S, Act, \rightarrow, I, AP, L)$, where $S = \{1, 2, 3\}$, $Act = \{\alpha, \beta, \gamma, \delta\}$, $AP = \{err\}$, and $L(3) = \{err\}$, $L(1) = L(2) = \emptyset$. The transition relation is given in Figure 8.3. Let $Sat(\Phi) \subseteq S$ denote the set of states satisfying modal formula Φ, and consider formulas and corresponding satisfaction sets in Table 8.3. The sets of states satisfying given formulas are fairly straightforward; hence let us focus on the last formula only, namely $\mathbf{AG}_{\{\gamma\}}err$. Notice that the only action that the formula mentions is γ. By looking at the transition relation in Figure 8.3 we

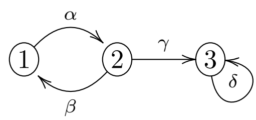


Figure 8.3: Example labelled transition system.

Φ	$Sat(\Phi)$
$\mathbf{EX}_{Act}\, err$	$\{2,3\}$
$\mathbf{AX}_{Act}\, err$	$\{3\}$
$\mathbf{E}(\mathbf{true}_{Act}\mathbf{U}_{Act}\, err)$	$\{1,2,3\}$
$\mathbf{A}(\mathbf{true}_{Act}\mathbf{U}_{Act}\, err)$	$\{3\}$
$\mathbf{EG}_{Act}\, err$	$\{2,3\}$
$\mathbf{AG}_{Act}\, err$	$\{3\}$
$\mathbf{AG}_{\{\gamma\}}\, err$	$\{1,2,3\}$

Table 8.3: Example ACTL formulae and corresponding satisfaction sets.

see that the only transition labelled γ leads to state satisfying predicate *err*, namely state 3. Thus the transition $2 \xrightarrow{\gamma} 3$ satisfies the semantics' condition for property $\mathbf{AG}_{\{\gamma\}}\, err$. For all the other transitions the condition holds trivially since the left hand site of the implication is false. As a result the formula is satisfied by all three states.

Notice that we do not get the equivalent CTL formulae by merely setting Ω to *Act*. For example, consider again the above transition system and transition relation depicted in Figure 8.3. The set of states satisfying the CTL formula $\mathbf{EG}\, err$ according to the semantics of CTL from Section 8.1 is $\{3\}$, whereas according to Table 8.3 the satisfaction set for an ACTL formula $\mathbf{EG}_{Act}\, err$ is $\{2,3\}$. Our choice of the semantics aims to illustrate how LFP can be used to specify a prototype model checker for a special purpose modal logic of interest. Other choices are also possible and they would follow similar pattern as the one presented here.

For a given ACTL formula, model checking aims at computing the set $Sat(\Phi) \subseteq S$ of states that satisfy the modal formula Φ. Similarly to the case of CTL, the algorithm usually proceeds in a syntax directed manner on the formula Φ. More precisely, the algorithm traverses the abstract syntax tree of the ACTL formula Φ in a bottom-up manner. The nodes of the abstract syntax tree correspond to the sub-formulae of Φ, and leaves are either the constant *true* or an atomic

$$
\begin{array}{l}
\mathit{define}(\forall s : Sat_{true}(s)) \\
\mathit{define}(\forall s : L_a(s) \Rightarrow Sat_a(s)) \\
\mathit{define}(\forall s : Sat_{\Phi_1}(s) \wedge Sat_{\Phi_2}(s) \Rightarrow Sat_{\Phi_1 \wedge \Phi_2}(s)) \\
\mathit{define}(\forall s : \neg Sat_{\Phi}(s) \Rightarrow Sat_{\neg\Phi}(s))
\end{array}
$$

$$
\mathit{define}(\forall s : (\exists a : \exists s' : T(s,a,s') \wedge \Omega(a) \wedge Sat_{\Phi}(s')) \Rightarrow Sat_{\mathbf{EX}_{\Omega}\Phi}(s))
$$

$$
\mathit{define}(\forall s : (\forall a : \forall s' : \neg T(s,a,s') \vee (\Omega(a) \wedge Sat_{\Phi}(s'))) \Rightarrow Sat_{\mathbf{AX}_{\Omega}\Phi}(s))
$$

$$
\mathit{define}\left(\begin{array}{l}
(\forall s : (\exists a : \exists s' : T(s,a,s') \wedge \Omega_2(a) \wedge Sat_{\Phi_2}(s')) \Rightarrow \\
\qquad\qquad Sat_{\mathbf{E}[\Phi_1{}_{\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]}(s)) \wedge \\
(\forall s : (\exists a : \exists s' : T(s,a,s') \wedge \Omega_1(a) \wedge Sat_{\Phi_1}(s') \wedge \\
\qquad\qquad Sat_{\mathbf{E}[\Phi_1{}_{\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]}(s')) \Rightarrow Sat_{\mathbf{E}[\Phi_1{}_{\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]}(s))
\end{array}\right)
$$

$$
\mathit{define}\left(\begin{array}{l}
(\forall s : (\forall a : \forall s' : \neg T(s,a,s') \vee (\Omega_2(a) \wedge Sat_{\Phi_2}(s')) \vee \\
\qquad\qquad (\Omega_1(a) \wedge Sat_{\Phi_1}(s') \wedge Sat_{\mathbf{A}[\Phi_1{}_{\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]}(s'))) \Rightarrow \\
\qquad\qquad\qquad Sat_{\mathbf{A}[\Phi_1{}_{\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]}(s))
\end{array}\right)
$$

$$
\mathit{constrain}\left(\begin{array}{l}
\forall s : Sat_{\mathbf{EG}_{\Omega}\Phi}(s) \Rightarrow (\exists a : \exists s' : T(s,a,s') \wedge \\
\qquad\qquad (\neg\Omega(a) \vee Sat_{\Phi}(s')) \wedge Sat_{\mathbf{EG}_{\Omega}\Phi}(s'))
\end{array}\right)
$$

$$
\mathit{constrain}\left(\begin{array}{l}
(\forall s : Sat_{\mathbf{AG}_{\Omega}\Phi}(s) \Rightarrow \\
\qquad\qquad (\forall a : \forall s' : \neg T(s,a,s') \vee \neg\Omega(a) \vee Sat_{\Phi}(s'))) \wedge \\
(\forall s : Sat_{\mathbf{AG}_{\Omega}\Phi}(s) \Rightarrow \\
\qquad\qquad (\forall a : \forall s' : \neg T(s,a,s') \vee Sat_{\mathbf{AG}_{\Omega}\Phi}(s')))
\end{array}\right)
$$

Table 8.4: LFP specification of satisfaction sets.

proposition $a \in AP$.

The LFP specification follows the similar pattern as in the CTL case, namely for each formula Φ we define a relation $Sat_{\Phi} \subseteq S$ characterizing states where Φ hold. The specification is defined in Table 8.4. The clause for *true* is straightforward and says that *true* holds in all states. The clause for an atomic proposition a expresses that a state satisfies a whenever it is in L_a, where we assume that we have a predicate $L_a \subseteq S$ for each $a \in AP$. The clause for $\Phi_1 \wedge \Phi_2$ captures that a state satisfies $\Phi_1 \wedge \Phi_2$ whenever it satisfies both Φ_1 and Φ_2. Similarly a state satisfies $\neg\Phi$ if it does not satisfy Φ. Notice, that all the propositional operators are given by means of *define* clauses, since they represent the least sets satisfying the specifications.

Now, let us focus on the modal operators. In the following we assume that for all subsets Ω of *Act* we have a corresponding relation Ω on actions, i.e. $\Omega \subseteq Act$. The formula for $\mathbf{EX}_\Omega\Phi$ captures that a state s satisfies $\mathbf{EX}_\Omega\Phi$, if there is a transition labelled a, where $\Omega(a)$ holds, to state s' such that s' satisfies Φ. The formula for $\mathbf{AX}_\Omega\Phi$ expresses that a state s satisfies $\mathbf{AX}_\Omega\Phi$ if for all actions a and states s': either there is no transition labelled a from s to s', or otherwise $\Omega(a)$ holds and s' satisfies Φ. The formula for $\mathbf{E}[\Phi_{1\,\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]$ captures two possibilities. The first one states that a state s satisfies $\mathbf{E}[\Phi_{1\,\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]$, if there is a transition labelled a, where $\Omega(a)$ holds, to state s' satisfying Φ_2. Alternatively if there is a transition from s labelled a, where $\Omega(a)$ holds, to a state satisfying both Φ_1 and $\mathbf{E}[\Phi_{1\,\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]$ then s also satisfies $\mathbf{E}[\Phi_{1\,\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]$. The formula for $\mathbf{A}[\Phi_{1\,\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]$ expresses that a state s satisfies the modal formula $\mathbf{A}[\Phi_{1\,\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]$ if for all states s' and actions a either there is no transition from s to s' labelled with a, or $\Omega_2(a)$ holds and Φ_2 is valid in s', or alternatively $\Omega_1(a)$ holds and both Φ_1 and $\mathbf{A}[\Phi_{1\,\Omega_1}\mathbf{U}_{\Omega_2}\Phi_2]$ are valid in s'. Notice, that since all the above LFP formulas represent the smallest sets of states satisfying corresponding modal formulas, they are given by means of a *define* clause. Let us now consider the formula for $\mathbf{EG}_\Omega\Phi$. Since the set of states satisfying $\mathbf{EG}_\Omega\Phi$ is defined as a largest set satisfying the semantics of $\mathbf{EG}_\Omega\Phi$, the property is defined by means of *constrain* clause. The clause expresses that whenever a state satisfies $\mathbf{EG}_\Omega\Phi$ then there exist a state s' and an action a such that there is a transition labelled a from s to s' and either $\Omega(a)$ does not hold or s' satisfies Φ, and furthermore s' satisfies $\mathbf{EG}_\Omega\Phi$. Finally let us consider the formula for $\mathbf{AG}_\Omega\Phi$, which is also defined in terms of *constrain* clause and distinguishes between two cases. In the first one whenever a state satisfies $\mathbf{AG}_\Omega\Phi$, then it must be the case that for all actions a and states s' either there is no transition labelled a from s to s', or otherwise either $\Omega(a)$ does not hold or s' satisfies Φ. Alternatively, if a state s satisfies $\mathbf{AG}_\Omega\Phi$ then for all actions a and states s': either there is no transition labelled a from s to s' or otherwise s' satisfies $\mathbf{AG}_\Omega\Phi$.

Now, let us make an analysis of the worst case time complexity of ACTL model checking performed by means of LFP. Thus, let us assume that the state space S has size $|S|$, whereas the size of the transition relation $\rightarrow$ is $|T|$. Notice also that for the ACTL formula Φ of size $|\Phi|$, the corresponding LFP clause has size $\mathcal{O}(|\Phi|)$, and the quantifier nesting depth is at most 3. According to Proposition 4.6 the worst case time complexity of the LFP specification is $\mathcal{O}(|S| + |S|^3|\Phi|)$. In the above we assumed that the number of atomic propositions is bounded by a constant. If we additionally assume that the number of action labels is also bounded by some constant, then the worst case time complexity of the LFP specification become $\mathcal{O}(|S| + |S|^2|\Phi|)$. The reason for that is the fact that the quantification over actions can be ignored and hence the maximal nesting depth of quantification becomes 2. Using a more refined reasoning than that of Proposition 4.6 we obtain $\mathcal{O}(|S|+|T||\Phi|)$. It is due to the fact that the "double quantifications" over states in Table 8.4 really correspond to traversing all possible transitions rather than all pairs of states. As a consequence, our LFP model checking algorithm has the same worst-case complexity as classical model checking algorithms [4].

Chapter 9

Conclusions and future work

In this dissertation we presented a framework for succinctly expressing static analysis and model checking problems. The framework facilitates rapid prototyping and consists of variants of ALFP logic and associated solvers. Since analysis specifications are usually written in a declarative style, logical formulations are convenient for creating their executable specifications. We also believe that the logical specifications of analysis problems are clearer and simpler to analyse for complexity and correctness than their imperative counterparts. Moreover, they give a clear distinction between specification of the analysis, and the computation of the best analysis result.

The great advantage of the framework is its applicability to various problems arising in both static analysis and model checking. For that reason we believe that this dissertation enhances our understanding of the interplay between static analysis and model checking - to the extent that they can be seen as essentially solving the same problem.

The main ingredients of the framework are as follows:

- ALFP logic developed by Nielson et al. [44], and the associated solving algorithms for computing the least model of a given ALFP formula. Currently there are two algorithms available; a differential algorithm developed by Nielson et al. [44] as well as a BDD-based one.

- LLFP logic that allows interpretations over complete lattices satisfying Ascending Chain Condition. We established a Moore Family result for LLFP that guarantees that there always is single best solution for a problem under consideration. We also developed a solving algorithm, that computes the least solution guaranteed by the Moore Family result. The key features of the algorithm is the use of prefix trees and its combination of continuation-passing style with propagation of differences.

- LFP logic, which has direct support for both inductive computations of behaviors as well as co-inductive specifications of properties. Two main theoretical contributions are a Moore Family result and a parametrized worst-case time complexity result. We also presented a BDD-based solving algorithm, which computes the least solution guaranteed by the Moore Family result with worst-case time complexity as given by the complexity result.

We showed that the logics and the associated solvers can be used for rapid prototyping. We illustrated that by a variety of case studies from static analysis and model checking.

As a future work we would like to implement a front-end for automatically extracting analysis specifications from program source code. This would allow to conduct a performance evaluation of the presented solving algoritms on the real-world systems. In order to enhance the efficiency of the algorithms it would be beneficial to extend the existing logic into a many-sorted ones. This would mean that the underlying universe would be partitioned into domains, and then each relation would be defined over specific domains. It would also be interesting to investigate the applicability of the magic set transformation described in Chapter 6 to other logics of this thesis. In particular, it is not entirely clear how it could be applied to LFP logic in the case of co-inductive specifications. Another direction for future work would be to lift the Ascending Chain Condition for the complete lattice over which the LLFP formulae are defined and use e.g. widening operator [20, 21] in order to ensure termination of the fixed point computation. The expressivity of the LLFP logic could also be extended by adding universal quantification of variables in preconditions. This would allow to express modal logic formulae that universally quantify over the paths e.g. $\mathbf{AX}\varphi$ or $\mathbf{AF}\varphi$. Moreover, we would like to enhance the framework by adding more logics, e.g. multi valued logics, that are more expressive and can handle problems that currently are beyond the current capabilities of the framework.

Appendix A

Proofs

A.1 Proof of Lemma 3.2

PROOF.

Reflexivity $\forall \varrho \in \Delta : \varrho \preceq \varrho$.

To show that $\varrho \preceq \varrho$ let us take $j = s$. If $\mathrm{rank}(R) < j$ then $\varrho(R) = \varrho(R)$ as required. Otherwise if $\mathrm{rank}(R) = j$ then from $\varrho(R) = \varrho(R)$ we get $\varrho(R) \sqsubseteq \varrho(R)$. Thus we get the required $\varrho \preceq \varrho$.

Transitivity $\forall \varrho_1, \varrho_2, \varrho_3 \in \Delta : \varrho_1 \preceq \varrho_2 \wedge \varrho_2 \preceq \varrho_3 \Rightarrow \varrho_1 \preceq \varrho_3$.

Let us assume that $\varrho_1 \preceq \varrho_2 \wedge \varrho_2 \preceq \varrho_3$. From $\varrho_i \preceq \varrho_{i+1}$ we have j_i such that conditions (a)–(c) are fulfilled for $i = 1, 2$. Let us take j to be the minimum of j_1 and j_2. Now we need to verify that conditions (a)–(c) hold for j. If $\mathrm{rank}(R) < j$ we have $\varrho_1(R) = \varrho_2(R)$ and $\varrho_2(R) = \varrho_3(R)$. It follows that $\varrho_1(R) = \varrho_3(R)$, hence (a) holds. Now let us assume that $\mathrm{rank}(R) = j$. We have $\varrho_1(R) \sqsubseteq \varrho_2(R)$ and $\varrho_2(R) \sqsubseteq \varrho_3(R)$ and from transitivity of $\sqsubseteq$ we get $\varrho_1(R) \sqsubseteq \varrho_3(R)$, which gives (b). Let us now assume that $j \neq s$, hence $\varrho_i(R) \sqsubset \varrho_{i+1}(R)$ for some $R \in \mathcal{R}$ and $i = 1, 2$. Without loss of generality let us assume that $\varrho_1(R) \sqsubset \varrho_2(R)$. We have $\varrho_1(R) \sqsubset \varrho_2(R)$ and $\varrho_2(R) \sqsubseteq \varrho_3(R)$, hence $\varrho_1(R) \sqsubset \varrho_3(R)$, and (c) holds.

Anti-symmetry $\forall \varrho_1, \varrho_2 \in \Delta : \varrho_1 \preceq \varrho_2 \wedge \varrho_2 \preceq \varrho_1 \Rightarrow \varrho_1 = \varrho_2$.

Let us assume $\varrho_1 \preceq \varrho_2$ and $\varrho_2 \preceq \varrho_1$. Let j be minimal such that $\mathrm{rank}(R) = j$ and $\varrho_1(R) \neq \varrho_2(R)$ for some $R \in \mathcal{R}$. Then, since $\mathrm{rank}(R) = j$, we have $\varrho_1(R) \sqsubseteq \varrho_2(R)$ and $\varrho_2(R) \sqsubseteq \varrho_1(R)$. Hence $\varrho_1(R) = \varrho_2(R)$ which is a contradiction. Thus it must be the case that $\varrho_1(R) = \varrho_2(R)$ for all $R \in \mathcal{R}$. □

A.2 Proof of Lemma 3.3

Proof. First we prove that $\bigsqcap_{\triangle} M$ is a lower bound of M; that is $\bigsqcap_{\triangle} M \preceq \varrho$ for all $\varrho \in M$. Let j be maximum such that $\varrho \in M_j$; since $M = M_0$ and $M_j \supseteq M_{j+1}$ clearly such j exists. From definition of M_j it follows that $(\bigsqcap_{\triangle} M)(R) = \varrho(R)$ for all R with $\mathrm{rank}(R) < j$; hence (a) holds. If $\mathrm{rank}(R) = j$ we have $(\bigsqcap_{\triangle} M)(R) = \lambda\vec{a}.\bigsqcap\{\varrho'(R)(\vec{a}) \mid \varrho' \in M_j\} \sqsubseteq \varrho(R)$ showing that (b) holds. Finally let us assume that $j \neq s$; we need to show that there is some R with $\mathrm{rank}(R) = j$ such that $(\bigsqcap_{\triangle} M)(R) \sqsubset \varrho(R)$. Since we know that j is maximum such that $\varrho \in M_j$, it follows that $\varrho \notin M_{j+1}$, hence there is a relation R with $\mathrm{rank}(R) = j$ such that $(\bigsqcap_{\triangle} M)(R) \sqsubset \varrho(R)$; thus (c) holds.

Now we need to show that $\bigsqcap_{\triangle} M$ is the greatest lower bound. Let us assume that $\varrho' \preceq \varrho$ for all $\varrho \in M$, and let us show that $\varrho' \preceq \bigsqcap_{\triangle} M$. If $\varrho' = \bigsqcap_{\triangle} M$ the result holds vacuously, hence let us assume $\varrho' \neq \bigsqcap_{\triangle} M$. Then there exists a minimal j such that $(\bigsqcap_{\triangle} M)(R) \neq \varrho'(R)$ for some R with $\mathrm{rank}(R) = j$. Let us first consider R such that $\mathrm{rank}(R) < j$. By our choice of j we have $(\bigsqcap_{\triangle} M)(R) = \varrho'(R)$ hence (a) holds. Next assume that $\mathrm{rank}(R) = j$. Since we assumed that $\varrho' \preceq \varrho$ for all $\varrho \in M$ and $M_j \subseteq M$, it follows that $\varrho'(R) \sqsubseteq \varrho(R)$ for all $\varrho \in M_j$. Thus we have $\varrho'(R) \sqsubseteq \lambda\vec{a}.\bigsqcap\{\varrho(R)(\vec{a}) \mid \varrho \in M_j\}$. Since $(\bigsqcap_{\triangle} M)(R) = \lambda\vec{a}.\bigsqcap\{\varrho(R)(\vec{a}) \mid \varrho \in M_j\}$, we have $\varrho'(R) \sqsubseteq (\bigsqcap_{\triangle} M)(R)$ which proves (b). Finally since we assumed that $\varrho'(R) \neq (\bigsqcap_{\triangle} M)(R)$ for some R with $\mathrm{rank}(R) = j$, it follows that (c) holds. Thus we proved that $\varrho' \preceq \bigsqcap_{\triangle} M$. □

A.3 Proof of Proposition 3.4

In order to prove Proposition 3.4 we first state and prove two auxiliary lemmas.

Lemma A.1 *If* $\varrho = \bigsqcap_\Delta M$, *pre occurs in* cl_j *and* $(\varrho, \varsigma) \models_\beta$ *pre then also* $(\varrho', \varsigma) \models_\beta$ *pre for all* $\varrho' \in M_j$.

Proof. We proceed by induction on j and in each case perform a structural induction on the form of the precondition *pre* occurring in cl_j.
Case: $pre = R(\vec{u}; V)$
Let us take $\varrho = \bigsqcap_\Delta M$ and assume that

$$(\varrho, \varsigma) \models_\beta R(\vec{u}; V)$$

From Table 3.1 we have:

$$\varrho(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(V)$$

Depending on the rank of R we have two cases. If $\text{rank}(R) = j$ then $\varrho(R) = \lambda\vec{a}.\bigsqcap\{\varrho'(R)(\vec{a}) \mid \varrho' \in M_j\}$ and hence we have

$$\bigsqcap\{\varrho'(R)(\varsigma(\vec{u})) \mid \varrho' \in M_j\} \sqsupseteq \varsigma(V)$$

It follows that for all $\varrho' \in M_j$

$$\varrho'(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(V)$$

Now if $\text{rank}(R) < j$ then $\varrho(R) = \varrho'(R)$ for all $\varrho' \in M_j$ hence we have that for all $\varrho' \in M_j$

$$\varrho'(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(V)$$

which according to Table 3.1 is equivalent to

$$\forall \varrho' \in M_j : (\varrho', \varsigma) \models_\beta R(\vec{u}; V)$$

which was required and finishes the case.
Case: $pre = Y(u)$
Let us take $\varrho = \bigsqcap_\Delta M$ and assume that

$$(\varrho, \varsigma) \models_\beta Y(u)$$

According to the semantics of LLFP in Table 3.1 we have

$$\beta(\varsigma(u)) \sqsubseteq \varsigma(Y)$$

It follows that

$$\forall \varrho' \in M_j : \beta(\varsigma(u)) \sqsubseteq \varsigma(Y)$$

which according to the semantics of LLFP in Table 3.1 is equivalent to

$$\forall \varrho' \in M_j : (\varrho', \varsigma) \models_\beta Y(u)$$

which was required and finishes the case.
Case: $pre = \neg R(\vec{u}; V)$
Let us take $\varrho = \bigsqcap_\Delta M$ and assume that

$$(\varrho, \varsigma) \models_\beta \neg R(\vec{u}; V)$$

From Table 3.1 we have:

$$\complement(\varrho(R)(\varsigma(\vec{u}))) \sqsupseteq \varsigma(V)$$

Since $\text{rank}(R) < j$ then we know that $\varrho(R) = \varrho'(R)$ for all $\varrho' \in M_j$ hence we have that

$$\forall \varrho' \in M_j : \complement(\varrho(R)(\varsigma(\vec{u}))) \sqsupseteq \varsigma(V)$$

Which according to Table 3.1 is equivalent to

$$\forall \varrho' \in M_j : (\varrho', \varsigma) \models_\beta \neg R(\vec{u}; V)$$

which was required and finishes the case.
Case: $pre = pre_1 \wedge pre_2$
Let us take $\varrho = \bigsqcap_\Delta M$ and assume that

$$(\varrho, \varsigma) \models_\beta pre_1 \wedge pre_2$$

According to Table 3.1 we have

$$(\varrho, \varsigma) \models_\beta pre_1$$

and

$$(\varrho, \varsigma) \models_\beta pre_2$$

From the induction hypothesis we get that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_\beta pre_1$$

and

$$(\varrho', \varsigma) \models_\beta pre_2$$

It follows that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_\beta pre_1 \wedge pre_2$$

which was required and finishes the case.
Case: $pre = pre_1 \vee pre_2$
Let us take $\varrho = \bigsqcap_\Delta M$ and assume that

$$(\varrho, \varsigma) \models_\beta pre_1 \vee pre_2$$

According to Table 3.1 we have

$$(\varrho, \varsigma) \models_\beta pre_1$$

or

$$(\varrho, \varsigma) \models_\beta pre_2$$

From the induction hypothesis we get that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_\beta pre_1$$

or

$$(\varrho', \varsigma) \models_\beta pre_2$$

It follows that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_\beta pre_1 \vee pre_2$$

which was required and finishes the case.
Case: $pre = \exists x : pre'$
Let us take $\varrho = \bigsqcap_\Delta M$ and assume that

$$(\varrho, \varsigma) \models_\beta \exists x : pre'$$

According to Table 3.1 we have

$$\exists a \in \mathcal{U} : (\varrho, \varsigma[x \mapsto a]) \models_\beta pre'$$

From the induction hypothesis we get that for all $\varrho' \in M_j$

$$\exists a \in \mathcal{U} : (\varrho', \varsigma[x \mapsto a]) \models_\beta pre'$$

It follows from Table 3.1 that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_\beta \exists x : pre'$$

which was required and finishes the case.
Case: $pre = \exists Y : pre'$
Let us take $\varrho = \bigsqcap_\Delta M$ and assume that

$$(\varrho, \varsigma) \models_\beta \exists Y : pre'$$

According to Table 3.1 we have

$$\exists l \in \mathcal{L}_{\neq\perp} : (\varrho, \varsigma[Y \mapsto l]) \models_\beta pre'$$

From the induction hypothesis we get that for all $\varrho' \in M_j$

$$\exists l \in \mathcal{L}_{\neq\perp} : (\varrho', \varsigma[Y \mapsto l]) \models_\beta pre'$$

It follows from Table 3.1 that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_\beta \exists Y : pre'$$

which was required and finishes the case. □

Lemma A.2 *If* $\varrho = \bigsqcap_{\Delta} M$ *and* $(\varrho', \varsigma) \models_\beta cl_j$ *for all* $\varrho' \in M$ *then* $(\varrho, \varsigma) \models_\beta cl_j$*.*

Proof. We proceed by induction on j and in each case perform a structural induction on the form of the clause occurring in cl_j.
Case: $cl_j = R(\vec{u}; V)$
Assume that for all $\varrho' \in M$

$$(\varrho', \varsigma) \models_\beta R(\vec{u}; V)$$

From the semantics of LLFP we have that for all $\varrho' \in M$

$$\varrho'(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(V)$$

It follows that:

$$\bigsqcap\{\varrho'(R)(\varsigma(\vec{u})) \mid \varrho' \in M\} \sqsupseteq \varsigma(V)$$

Since $M_j \subseteq M$, we have:

$$\bigsqcap\{\varrho'(R)(\varsigma(\vec{u})) \mid \varrho' \in M_j\} \sqsupseteq \varsigma(V)$$

We know that $\text{rank}(R) = j$; hence $\varrho(R) = \lambda\vec{a}.\bigsqcap\{\varrho'(R)(\vec{a}) \mid \varrho' \in M_j\}$; thus

$$\varrho(R)(\varsigma(\vec{u})) = \bigsqcap\{\varrho'(R)(\varsigma(\vec{u})) \mid \varrho' \in M_j\} \sqsupseteq \varsigma(V)$$

Which according to Table 3.1 is equivalent to

$$(\varrho, \varsigma) \models_\beta R(\vec{u}; V)$$

Case: $cl_j = cl_1 \wedge cl_2$
Assume that for all $\varrho' \in M$:

$$(\varrho', \varsigma) \models_\beta cl_1 \wedge cl_2$$

From Table 3.1 it is equivalent to

$$(\varrho', \varsigma) \models_\beta cl_1 \text{ and } (\varrho', \varsigma) \models_\beta cl_2$$

The induction hypothesis gives that

$$(\varrho, \varsigma) \models_\beta cl_1 \text{ and } (\varrho, \varsigma) \models_\beta cl_2$$

Which according to Table 3.1 is equivalent to

$$(\varrho, \varsigma) \models_\beta cl_1 \wedge cl_2$$

and finishes the case.
Case: $cl_j = pre \Rightarrow cl$
Assume that for all $\varrho' \in M$:

$$(\varrho', \varsigma) \models_\beta pre \Rightarrow cl \tag{A.1}$$

We have two cases. In the first one $(\varrho, \varsigma) \models_\beta pre$ is $false$, hence $(\varrho, \varsigma) \models_\beta pre \Rightarrow cl$ holds trivially. In the second case let us assume:

$$(\varrho, \varsigma) \models_\beta pre \tag{A.2}$$

Lemma A.1 gives that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_\beta pre$$

From (A.1) we have that for all $\varrho' \in M_j$

$$(\varrho', \varsigma) \models_\beta cl$$

and the induction hypothesis gives:

$$(\varrho, \varsigma) \models_\beta cl$$

Hence from (A.2) we get:

$$(\varrho, \varsigma) \models_\beta pre \Rightarrow cl$$

which was required and finishes the case.
Case: $cl_j = \forall x : cl$
Assume that for all $\varrho' \in M$

$$(\varrho', \varsigma) \models_\beta \forall x : cl$$

From Table 3.1 we have that for all $\varrho' \in M$ and for all $a \in \mathcal{U}$

$$(\varrho', \varsigma[x \mapsto a]) \models_\beta cl$$

Thus from the induction hypothesis we get that for all $a \in \mathcal{U}$

$$(\varrho, \varsigma[x \mapsto a]) \models_\beta cl$$

According to Table 3.1 it is equivalent to

$$(\varrho, \varsigma) \models_\beta \forall x : cl$$

which was required and finishes the case.
Case: $cl = \forall Y : cl$
Assume that for all $\varrho' \in M$

$$(\varrho', \varsigma) \models_\beta \forall Y : cl$$

From Table 3.1 we have that $\varrho' \in M$

$$\forall l \in \mathcal{L}_{\neq \bot} : (\varrho', \varsigma[Y \mapsto l]) \models_\beta cl$$

Thus from the induction hypothesis we get that

$$\forall l \in \mathcal{L}_{\neq\perp} : (\varrho, \varsigma[Y \mapsto l]) \models_\beta cl$$

According to Table 3.1 it is equivalent to

$$(\varrho, \varsigma) \models_\beta \forall Y : cl$$

which was required and finishes the case. □

Proposition 3.4. Assume cls is a stratified LLFP clause sequence, and let ς_0 be an interpretation of free variables in cls. Furthermore, ϱ_0 is an interpretation of all relations of rank 0. Then

$$\{\varrho \mid (\varrho, \varsigma_0) \models_\beta cls \wedge \forall R : \mathrm{rank}(R) = 0 \Rightarrow \varrho_0(R) \sqsubseteq \varrho(R)\}$$

is a Moore family.

Proof. The result follows from Lemma A.2. □

A.4 Proof of Proposition 3.5

Proposition 3.5: If ϕ is a well formed LLFP formula (a precondition, clause or a clause sequence), the underlying complete lattice is $\mathcal{P}(\mathcal{U})$ and $\beta : \mathcal{U} \to \mathcal{L}$ is defined as $\beta(a) = \{a\}$ for all $a \in \mathcal{U}$, then

$$(\varrho, \varsigma) \models_\beta \phi \quad \Leftrightarrow \quad \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\phi)$$

Proof. We proceed by structural induction on ϕ.

Positive query. For a positive query $R(\vec{u}; V)$ we have to prove: $(\varrho, \varsigma) \models_\beta R(\vec{u}; V) \Leftrightarrow \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(R(\vec{u}; V))$. We have three sub-cases:

Case: $R(\vec{u}; Y)$
We have:

$$(\varrho, \varsigma) \models_\beta R(\vec{u}; Y)$$

Which according to the Table 3.1 gives:

$$\varrho(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(Y) \tag{A.3}$$

On the other hand we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(R(\vec{u}; Y))$$

Hence, from Table 3.2 we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models R(\vec{u}, x_Y)$$

Then according to Table 2.1 we have:

$$\forall \sigma \in f(\varsigma) : (\sigma(\vec{u}), \sigma(x_Y)) \in f(\varrho)(R)$$

Which according to (3.1) gives:

$$\forall \sigma \in f(\varsigma) : \beta(\sigma(x_Y)) \sqsubseteq \varrho(R)(\sigma(\vec{u}))$$

Which equals to:

$$\forall \sigma \in f(\varsigma) : \beta(\sigma(x_Y)) \sqsubseteq \varrho(R)(\varsigma(\vec{u})) \tag{A.4}$$

(A.3)$\Rightarrow$(A.4): For any $\sigma \in f(\varsigma)$ we have $\beta(\sigma(x_Y)) \sqsubseteq \varsigma(Y)$ according to (3.2); from (A.3) we have $\varsigma(Y) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$ and this gives (A.4).

(A.4)⇒(A.3): Let $\beta(a) = \varsigma(Y)$; it then follows from (3.2) that there is $\sigma_a \in f(\varsigma)$ such that: $\sigma_a(x_Y) = a$. Then from (A.4) we get: $\beta(a) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$ and since we assumed that $\beta(a) = \varsigma(Y)$ this proves (A.3).

Case: $R(\vec{u}; [v])$
We have:

$$(\varrho, \varsigma) \models_\beta R(\vec{u}; [v])$$

Which according to the Table 3.1 gives:

$$\varsigma([v]) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

On the other hand we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(R(\vec{u}; [v]))$$

Hence, from Table 3.2 we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models R(\vec{u}, v)$$

Then according to Table 2.1 we have:

$$\forall \sigma \in f(\varsigma) : (\sigma(\vec{u}), \sigma(v)) \in f(\varrho)(R)$$

From (3.2) it follows that:

$$(\varsigma(\vec{u}), \varsigma(v)) \in f(\varrho)(R)$$

Which is equivalent to:

$$\beta(\varsigma(v)) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

and finishes the case since $\varsigma([v]) = \beta(\varsigma(v))$.

Case: $Y(u)$
We have:

$$(\varrho, \varsigma) \models_\beta Y(u)$$

Which according to the Table 3.1 gives:

$$\beta(\varsigma(u)) \sqsubseteq \varsigma(Y)$$

On the other hand we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(Y(u))$$

Hence, from Table 3.2 we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models x_Y = u$$

Then according to Table 2.1 we have:

$$\forall \sigma \in f(\varsigma) : \sigma(x_Y) = \sigma(u)$$

From (3.2) it follows that:

$$\forall \sigma \in f(\varsigma) : \beta(\sigma(u)) \sqsubseteq \varsigma(Y)$$

Since $\varsigma(u) = \sigma(u)$ we have:

$$\beta(\varsigma(u)) \sqsubseteq \varsigma(Y)$$

quod erat demonstrandum.

Negative query. For a negative query $\neg R(\vec{u}; V)$ we have to prove: $(\varrho, \varsigma) \models_\beta \neg R(\vec{u}; V) \Leftrightarrow \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\neg R(\vec{u}; V))$. We have two sub-cases:

Case: $\neg R(\vec{u}; Y)$
We have:

$$(\varrho, \varsigma) \models_\beta \neg R(\vec{u}; Y)$$

Which according to the Table 3.1 gives:

$$\complement \varrho(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(Y) \tag{A.5}$$

On the other hand we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\neg R(\vec{u}; Y))$$

Hence, from Table 3.2 we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models \neg R(\vec{u}, x_Y)$$

Then according to Table 2.1 we have:

$$\forall \sigma \in f(\varsigma) : (\sigma(\vec{u}), \sigma(x_Y)) \notin f(\varrho)(R)$$

Which according to (3.1) gives:

$$\forall \sigma \in f(\varsigma) : \beta(\sigma(x_Y)) \not\sqsubseteq \varrho(R)(\sigma(\vec{u}))$$

Which equals to:

$$\forall \sigma \in f(\varsigma) : \beta(\sigma(x_Y)) \not\sqsubseteq \varrho(R)(\varsigma(\vec{u})) \tag{A.6}$$

(A.5)⇒(A.6): Assume that: $\beta(\sigma(x_Y)) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$, that is, $\beta(\sigma(x_Y)) \not\sqsubseteq \complement \varrho(R)(\varsigma(\vec{u}))$. Then from (A.5) it follows that: $\beta(\sigma(x_Y)) \not\sqsubseteq \varsigma(Y)$, which is a contradiction and (A.6) follows.

(A.6)⇒(A.5): Assume that $\beta(a) \sqsubseteq \varsigma(Y)$. Then (3.2) gives that there is some $\sigma_a \in f(\varsigma)$ such that $\sigma_a(x_Y) = a$. Then it follows from (A.6) that: $\beta(a) \not\sqsubseteq \varrho(R)(\varsigma(\vec{u}))$; hence we get that: $\beta(a) \sqsubseteq \complement\varrho(R)(\varsigma(\vec{u}))$. Since we assumed that $\beta(a) \sqsubseteq \varsigma(Y)$ we get (A.5), which was required.

Case: $\neg R(\vec{u}; [v])$
We have:

$$(\varrho, \varsigma) \models_\beta \neg R(\vec{u}; [v])$$

Which according to the Table 3.1 gives:

$$\varsigma([v]) \sqsubseteq \complement\varrho(R)(\varsigma(\vec{u}))$$

On the other hand we have:

$$\forall\sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\neg R(\vec{u}; [v]))$$

Hence, from Table 3.2 we have:

$$\forall\sigma \in f(\varsigma) : (f(\varrho), \sigma) \models \neg R(\vec{u}, v)$$

Then according to Table 2.1 we have:

$$\forall\sigma \in f(\varsigma) : (\sigma(\vec{u}), \sigma(v)) \notin f(\varrho)(R)$$

From (3.2) it follows that:

$$(\varsigma(\vec{u}), \varsigma(v)) \notin f(\varrho)(R)$$

Which gives:

$$\beta(\varsigma(v)) \not\sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Which is equivalent to:

$$\beta(\varsigma(v)) \sqsubseteq \complement\varrho(R)(\varsigma(\vec{u}))$$

Since $\varsigma([v]) = \beta(\varsigma(v))$, we have:

$$\varsigma([v]) \sqsubseteq \complement\varrho(R)(\varsigma(\vec{u}))$$

quod erat demonstrandum.

The cases for $pre_1 \wedge pre_2$ and $pre_1 \vee pre_2$ follows directly from the induction hypothesis.

Case: $\exists x : pre$. For a precondition $\exists x : pre$ we have to prove: $(\varrho, \varsigma) \models_\beta \exists x : pre \Leftrightarrow \forall\sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\exists x : pre)$. The induction hypothesis says that for all $a \in \mathcal{U} : (\varrho, \varsigma[x \mapsto a]) \models_\beta pre \Leftrightarrow \forall\sigma' \in f(\varsigma[x \mapsto a]) : (f(\varrho), \sigma') \models f(pre)$.

We have:

$$(\varrho, \varsigma) \models_\beta \exists x : pre$$

Which according to the Table 3.1 gives:

$$\exists a \in \mathcal{U} : (\varrho, \varsigma[x \mapsto a]) \models_\beta pre$$

Then, from the induction hypothesis we get:

$$\exists a \in \mathcal{U} : \forall \sigma' \in f(\varsigma[x \mapsto a]) : (f(\varrho), \sigma') \models f(pre)$$

which from (3.2) gives:

$$\exists a \in \mathcal{U} : \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma[x \mapsto a]) \models f(pre)$$

Which equals:

$$\forall \sigma \in f(\varsigma) : \exists a \in \mathcal{U} : (f(\varrho), \sigma[x \mapsto a]) \models f(pre)$$

Then from Table 2.1 it follows that:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models \exists x : f(pre)$$

Hence from Table 3.2 we get the required:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\exists x : pre)$$

Case: $\exists Y : pre$. For a precondition $\exists Y : pre$ we have to prove: $(\varrho, \varsigma) \models_\beta \exists Y : pre \Leftrightarrow \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\exists Y : pre)$. The induction hypothesis says that for all $l \in \mathcal{L}_{\neq\bot} : (\varrho, \varsigma[Y \mapsto l]) \models_\beta pre \Leftrightarrow \forall \sigma' \in f(\varsigma[Y \mapsto l]) : (f(\varrho), \sigma') \models f(pre)$.

We have:

$$(\varrho, \varsigma) \models_\beta \exists Y : pre$$

Which according to the Table 3.1 gives:

$$\exists l \in \mathcal{L}_{\neq\bot} : (\varrho, \varsigma[Y \mapsto l]) \models_\beta pre$$

Then, from the induction hypothesis we get:

$$\exists l \in \mathcal{L}_{\neq\bot} : \forall \sigma' \in f(\varsigma[Y \mapsto l]) : (f(\varrho), \sigma') \models f(pre)$$

which from (3.2) gives:

$$\exists l \in \mathcal{L}_{\neq\bot} : \forall a : \beta(a) \sqsubseteq l : \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma[x_Y \mapsto a]) \models f(pre)$$

Which equals:

$$\forall \sigma \in f(\varsigma) : \exists l \in \mathcal{L}_{\neq\perp} : \forall a : \beta(a) \sqsubseteq l : (f(\varrho), \sigma[x_Y \mapsto a]) \models f(pre)$$

Then from Table 2.1 we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models \exists Y : f(pre)$$

Hence from Table 3.2 we get the required:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\exists Y : pre)$$

Assertion. For an assertion $R(\vec{u}; V)$ we have to prove: $(\varrho, \varsigma) \models_\beta R(\vec{u}; V) \Leftrightarrow \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(R(\vec{u}; V))$. We have two sub-cases:

Case: $R(\vec{u}; Y)$
We have:

$$(\varrho, \varsigma) \models_\beta R(\vec{u}; Y)$$

Which according to the Table 3.1 gives:

$$\varrho(R)(\varsigma(\vec{u})) \sqsupseteq \varsigma(Y) \tag{A.7}$$

On the other hand we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(R(\vec{u}; Y))$$

Hence, from Table 3.2 we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models R(\vec{u}, x_Y)$$

Then according to Table 2.1 we have:

$$\forall \sigma \in f(\varsigma) : (\sigma(\vec{u}), \sigma(x_Y)) \in f(\varrho)(R)$$

Which according to (3.2) gives:

$$\forall \sigma \in f(\varsigma) : \beta(\sigma(x_Y)) \sqsubseteq \varrho(R)(\sigma(\vec{u}))$$

Since $\sigma(\vec{u}) = \varsigma(\vec{u})$ it equals to:

$$\forall \sigma \in f(\varsigma) : \beta(\sigma(x_Y)) \sqsubseteq \varrho(R)(\varsigma(\vec{u})) \tag{A.8}$$

(A.7)⇒(A.8) For any $\sigma \in f(\varsigma)$ we have: $\beta(\sigma(x_Y)) \sqsubseteq \varsigma(Y)$. From (A.7) we have $\varsigma(Y) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$, which gives the required.

(A.8)⇒(A.7) Let $\beta(a) = \sigma_a(Y)$; from (3.2) it follows that there is $\sigma_a \in f(\varsigma)$ such that: $\sigma_a(x_Y) = a$. Then from (A.8) we get: $\beta(a) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$, and since we assumed that $\beta(a) = \sigma_a(Y)$ we get (A.7), which was required.

Case: $R(\vec{u};[v])$
We have:

$$(\varrho,\varsigma) \models_\beta R(\vec{u};[v])$$

Which according to the Table 3.1 gives:

$$\varsigma([v]) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

On the other hand we have:

$$\forall\sigma \in f(\varsigma) : (f(\varrho),\sigma) \models f(R(\vec{u};[v]))$$

Hence, from Table 3.2 we have:

$$\forall\sigma \in f(\varsigma) : (f(\varrho),\sigma) \models R(\vec{u},v)$$

Then according to Table 2.1 we have:

$$\forall\sigma \in f(\varsigma) : (\sigma(\vec{u}),\sigma(v)) \in f(\varrho)(R)$$

From (3.2) it follows that:

$$(\varsigma(\vec{u}),\varsigma(v)) \in f(\varrho)(R)$$

Which is equivalent to:

$$\beta(\varsigma(v)) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Since $\beta(\varsigma(v)) = \varsigma([v])$ we have:

$$\varsigma([v]) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

quod erat demonstrandum.

The case of **1** follows directly from the definition.

The cases of $cl_1 \wedge cl_2$ and $pre \Rightarrow cl$ follow from the induction hypothesis.

Case: $\forall Y : cl$. For a clause $\forall Y : cl$ we have to prove: $(\varrho,\varsigma) \models_\beta \forall Y : cl \Leftrightarrow \forall\sigma \in f(\varsigma) : (f(\varrho),\sigma) \models f(\forall Y : cl)$. The induction hypothesis says that for all $l \in \mathcal{L}_{\neq\perp} : (\varrho,\varsigma[Y \mapsto l]) \models_\beta cl \Leftrightarrow \forall\sigma' \in f(\varsigma[Y \mapsto l]) : (f(\varrho),\sigma') \models f(cl)$.

We have:

$$(\varrho,\varsigma) \models_\beta \forall Y : cl$$

Which according to the Table 3.1 gives:

$$\forall l \in \mathcal{L}_{\neq\bot} : (\varrho, \varsigma[Y \mapsto l]) \models_\beta cl$$

Then, from the induction hypothesis we get:

$$\forall l \in \mathcal{L}_{\neq\bot} : \forall \sigma' \in f(\varsigma[Y \mapsto l]) : (f(\varrho), \sigma') \models f(cl)$$

which from Table 3.2 gives:

$$\forall l \in \mathcal{L}_{\neq\bot} : \forall a : \beta(a) \sqsubseteq l : \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma[x_Y \mapsto a]) \models f(cl)$$

Which is equivalent to:

$$\forall a \in \mathcal{U} : \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma[x_Y \mapsto a]) \models f(cl)$$

Which equals:

$$\forall \sigma \in f(\varsigma) : \forall a \in \mathcal{U} : (f(\varrho), \sigma[x_Y \mapsto a]) \models f(cl)$$

Then from Table 2.1 we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models \forall x_Y : f(cl)$$

Hence from Table 3.2 we get the required:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\forall Y : cl)$$

Case: $\forall x : cl$. For a clause $\forall x : cl$ we have to prove: $(\varrho, \varsigma) \models_\beta \forall x : cl \Leftrightarrow \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\forall x : cl)$. The induction hypothesis says that for all $a \in \mathcal{U} : (\varrho, \varsigma[x \mapsto a]) \models_\beta cl \Leftrightarrow \forall \sigma' \in f(\varsigma[x \mapsto a]) : (f(\varrho), \sigma') \models f(cl)$.

We have:

$$(\varrho, \varsigma) \models_\beta \forall x : cl$$

Which according to the Table 3.1 gives:

$$\forall a \in \mathcal{U} : (\varrho, \varsigma[x \mapsto a]) \models_\beta cl$$

Then, from the induction hypothesis we get:

$$\forall a \in \mathcal{U} : \forall \sigma' \in f(\varsigma[x \mapsto a]) : (f(\varrho), \sigma') \models f(cl)$$

which from (3.2) gives:

$$\forall a \in \mathcal{U} : \forall \sigma \in f(\varsigma) : (f(\varrho), \sigma[x \mapsto a]) \models f(cl)$$

Which equals:

$$\forall \sigma \in f(\varsigma) : \forall a \in \mathcal{U} : (f(\varrho), \sigma[x \mapsto a]) \models f(cl)$$

Then from Table 2.1 we have:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models \forall x : f(cl)$$

Hence from Table 3.2 we get the required:

$$\forall \sigma \in f(\varsigma) : (f(\varrho), \sigma) \models f(\forall x : cl)$$

A.5 Proof of Lemma 3.7

Lemma 3.7: If cl is an LLFP clause, then $h(cl)$ is in Horn format and:

$$(\varrho, \varsigma) \models_\beta cl \quad \Leftrightarrow \quad (\varrho, \varsigma) \models_\beta h(cl)$$

PROOF. We conduct the proof by showing that each step of the transformation is semantics preserving.
Step 1: Renaming of variables is semantics preserving.
Step 2: Assume that

$$(\varrho, \varsigma) \models_\beta (\exists x : pre) \Rightarrow cl$$

From Table 3.1 we have

$$(\varrho, \varsigma) \not\models_\beta (\exists x : pre) \vee (\varrho, \varsigma) \models_\beta cl$$

Using Table 3.1 it follows that for all $a \in \mathcal{U}$

$$(\varrho, \varsigma[x \mapsto a]) \not\models_\beta pre \vee (\varrho, \varsigma) \models_\beta cl$$

Since $x \notin fv(cl)$ we have that for all $a \in \mathcal{U}$

$$(\varrho, \varsigma[x \mapsto a]) \not\models_\beta pre \vee (\varrho, \varsigma[x \mapsto a]) \models_\beta cl$$

Which is equivalent to

$$\forall a \in \mathcal{U} : (\varrho, \varsigma[x \mapsto a]) \models_\beta pre \Rightarrow cl$$

From Table 3.1 we get

$$(\varrho, \varsigma) \models_\beta \forall x : pre \Rightarrow cl$$

which was required and finishes the case.
Step 3: Transformation into DNF is semantics preserving.
Step 4: We consider a simplified case for two disjuncts only. Assume that

$$(\varrho, \varsigma) \models_\beta (pre_1 \vee pre_2) \Rightarrow cl$$

From Table 3.1 we have

$$(\varrho, \varsigma) \not\models_\beta (pre_1 \vee pre_2) \vee (\varrho, \varsigma) \models_\beta cl$$

From De Morgan's laws we get

$$((\varrho, \varsigma) \not\models_\beta pre_1 \wedge (\varrho, \varsigma) \not\models_\beta pre_2) \vee (\varrho, \varsigma) \models_\beta cl$$

Which is equivalent to

$$((\varrho, \varsigma) \not\models_\beta pre_1 \vee (\varrho, \varsigma) \models_\beta cl) \wedge ((\varrho, \varsigma) \not\models_\beta pre_2 \vee (\varrho, \varsigma) \models_\beta cl)$$

From which it follows that

$$((\varrho,\varsigma) \models_\beta pre_1 \Rightarrow cl) \wedge ((\varrho,\varsigma) \models_\beta pre_2 \Rightarrow cl)$$

Hence from Table 3.1 we have

$$(\varrho,\varsigma) \models_\beta (pre_1 \Rightarrow cl) \wedge (pre_2 \Rightarrow cl)$$

which was required and finishes the case.
Step 5: Assume

$$(\varrho,\varsigma) \models_\beta pre \Rightarrow \forall x : cl$$

From Table 3.1 we have

$$((\varrho,\varsigma) \not\models_\beta pre) \vee ((\varrho,\varsigma) \models_\beta \forall x : cl)$$

Using Table 3.1 we get

$$((\varrho,\varsigma) \not\models_\beta pre) \vee (\forall a \in \mathcal{U} : (\varrho,\varsigma[x \mapsto a]) \models_\beta cl)$$

Since $x \notin fv(pre)$ the above is equivalent to

$$\forall a \in \mathcal{U} : ((\varrho,\varsigma[x \mapsto a]) \not\models_\beta pre) \vee ((\varrho,\varsigma[x \mapsto a]) \models_\beta cl)$$

It follows that

$$\forall a \in \mathcal{U} : (\varrho,\varsigma[x \mapsto a]) \models_\beta pre \Rightarrow cl$$

Which according to Table 3.1 is equivalent to

$$(\varrho,\varsigma) \models_\beta \forall x : (pre \Rightarrow cl)$$

which was required and finishes the case.
Step 6: Assume

$$(\varrho,\varsigma) \models_\beta \forall x : (pre \Rightarrow cl_1 \wedge cl_2)$$

From Table 3.1 we have that for all $a \in \mathcal{U}$

$$(\varrho,\varsigma[x \mapsto a]) \models_\beta (pre \Rightarrow cl_1 \wedge cl_2)$$

Which is equivalent to

$$((\varrho,\varsigma[x \mapsto a]) \not\models_\beta pre) \vee ((\varrho,\varsigma[x \mapsto a]) \models_\beta cl_1 \wedge (\varrho,\varsigma[x \mapsto a]) \models_\beta cl_2)$$

It follows that

$$((\varrho,\varsigma[x \mapsto a]) \not\models_\beta pre \vee (\varrho,\varsigma[x \mapsto a]) \models_\beta cl_1)$$

and

$$((\varrho,\varsigma[x \mapsto a]) \not\models_\beta pre \vee (\varrho,\varsigma[x \mapsto a]) \models_\beta cl_2)$$

Hence we have that for all $a \in \mathcal{U}$

$$((\varrho, \varsigma[x \mapsto a]) \models_\beta pre \Rightarrow cl_1) \wedge ((\varrho, \varsigma[x \mapsto a]) \models_\beta pre \Rightarrow cl_2)$$

According to Table 3.1 we have

$$((\varrho, \varsigma) \models_\beta \forall x : pre \Rightarrow cl_1) \wedge ((\varrho, \varsigma) \models_\beta \forall x : pre \Rightarrow cl_2)$$

which was required and finishes the sub-case.
Now let us assume

$$(\varrho, \varsigma) \models_\beta \forall x : (pre' \Rightarrow (pre'' \Rightarrow cl))$$

From Table 3.1 we have that for all $a \in \mathcal{U}$

$$(\varrho, \varsigma[x \mapsto a]) \models_\beta pre' \Rightarrow (pre'' \Rightarrow cl)$$

Which is equivalent to

$$(\varrho, \varsigma[x \mapsto a]) \not\models_\beta pre' \vee (\varrho, \varsigma[x \mapsto a]) \not\models_\beta pre' \vee (\varrho, \varsigma[x \mapsto a]) \models_\beta cl$$

Using De Margan's laws we get

$$(\varrho, \varsigma[x \mapsto a]) \not\models_\beta (pre' \wedge pre'') \vee (\varrho, \varsigma[x \mapsto a]) \models_\beta cl$$

Which is equivalent to

$$(\varrho, \varsigma[x \mapsto a]) \models_\beta (pre' \wedge pre'') \Rightarrow cl$$

According to Table 3.1 we have

$$(\varrho, \varsigma) \models_\beta \forall x : (pre' \wedge pre'') \Rightarrow cl$$

Which was required and finishes the case.

A.6 Proof of Lemma 3.9

Lemma 3.9: Assume that *pre* contains k defining occurrences of Y and that

$$\begin{array}{l} (\varrho, \varsigma[Y \mapsto l]) \models_\beta pre \\ (\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_\beta g(pre) \end{array}$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. Then

$$\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k](W^Y_{pre}) = l$$

If Y does not occur in a defining position in pre then $W_{pre}^{Y} = L$.

Proof. We proceed by the induction on the structure of pre.
Case $pre = R'(\vec{u}; Y)$
Assume:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} R'(\vec{u}; Y), l \neq \bot$$

Then from Table 3.1 we have:

$$l \sqsubseteq \varrho(R')(\varsigma(\vec{u})) \tag{A.9}$$

Let's also assume that:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models_{\beta}^{\#} g(R'(\vec{u}; Y)),\ l_i \neq \bot,\ (1 \leq i \leq k)$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. Since $W_{pre}^{Y} = Y_i$ and $g(R'(\vec{u}; Y)) = R'(\vec{u}; Y_i)$, we have:

$$(\varrho, \varsigma[Y_i \mapsto l_i]) \models_{\beta}^{\#} R'(\vec{u}; Y_i),\ l_i \neq \bot$$

From Table 3.3 it follows that:

$$l_i = \varrho(R')(\varsigma(\vec{u})) \tag{A.10}$$

We have to show:

$$\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k](Y_i) = l$$

From (A.9) and (A.10) we have:

$$l_i = \varrho(R')(\varsigma(\vec{u})) \sqsupseteq l$$

We also know that $l_i \sqsubseteq l$ for $1 \leq i \leq k$. Thus it follows that:

$$l_i = l = \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k](Y_i)$$

which finishes the case.

Case $pre = R'(\vec{u}; [v])$
Assume:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} R'(\vec{u}; [v]), l \neq \bot$$

Then from Table 3.1 we have:

$$\beta(\varsigma(v)) \sqsubseteq \varrho(R')(\varsigma(\vec{u})) \tag{A.11}$$

Let's also assume that:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models_{\beta}^{\#} g(R'(\vec{u}; [v])),\ l_i \neq \bot,\ (1 \leq i \leq k)$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. Since $W^Y_{pre} = L$ and $g(R'(\vec{u};[v])) = R'(\vec{u};[v])$, we have:

$$(\varrho, \varsigma[Y_i \mapsto l_i]) \models^{\#}_{\beta} R'(\vec{u};[v]),\ l_i \neq \bot$$

From Table 3.3 it follows that:

$$\beta(\varsigma(v)) \sqsubseteq \varrho(R')(\varsigma(\vec{u})) \tag{A.12}$$

We have to show:

$$\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k](W^Y_{pre}) = L = l$$

which holds trivially and finishes the case.

Case $pre = Y(u)$
Assume:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} Y(u), l \neq \bot$$

Then from Table 3.1 we have:

$$\beta(\varsigma(\vec{u})) \sqsubseteq \varsigma[Y \mapsto l](Y)$$

Let's also assume that:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(Y(u)),\ l_i \neq \bot,\ (1 \leq i \leq k)$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. We have $g(Y(u)) = W^Y_{pre}(u)$, and $W^Y_{pre} = L$, thus we get:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} W^Y_{pre}(u),\ l_i \neq \bot,\ (1 \leq i \leq k)$$

We have to show:

$$\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k](W^Y_{pre}) = L = l$$

which holds trivially and finishes the case.

Case $pre = \neg R'(\vec{u}; Y)$
Assume:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} \neg R'(\vec{u}; Y), l \neq \bot$$

Then from Table 3.1 we have:

$$l \sqsubseteq \complement\varrho(R')(\varsigma(\vec{u})) \tag{A.13}$$

Let's also assume that:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(\neg R'(\vec{u}; Y)),\ l_i \neq \bot,\ (1 \leq i \leq k)$$

where $l_i \subseteq l$ for $1 \leq i \leq k$. Since $g(\neg R'(\vec{u}; Y)) = R'(\vec{u}; Y_i)$ we have:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} R'(\vec{u}; Y_i),\ l_i \neq \bot,\ (1 \leq i \leq k)$$

From Table 3.3 it follows that:

$$l_i = \varrho(R')(\varsigma(\vec{u}))$$

We have $W^Y_{pre} = \complement Y_i$, and

$$\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k](\complement Y_i) = \complement l_i = \complement \varrho(R')(\varsigma(\vec{u})) \sqsupseteq l$$

Hence we have that $l_i \sqsubseteq l$ and $\complement l_i \sqsupseteq l$, which means that $l_i = \bot$. The assumption is false; thus the case holds trivially.

Case $pre = \neg R'(\vec{u}; [v])$
Assume:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} \neg R'(\vec{u}; [v]), l \neq \bot$$

Then from Table 3.1 we have:

$$\beta(\varsigma(v)) \sqsubseteq \complement \varrho(R')(\varsigma(\vec{u})) \tag{A.14}$$

Let's also assume that:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(\neg R'(\vec{u}; [v])),\ l_i \neq \bot,\ (1 \leq i \leq k)$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. Since $W^Y_{pre} = L$ and $g(\neg R'(\vec{u}; [v])) = \neg R'(\vec{u}; [v])$, we have:

$$(\varrho, \varsigma[Y_i \mapsto l_i]) \models^{\#}_{\beta} \neg R'(\vec{u}; [v]),\ l_i \neq \bot$$

From Table 3.3 it follows that:

$$\beta(\varsigma(v)) \sqsubseteq \complement \varrho(R')(\varsigma(\vec{u})) \tag{A.15}$$

We have to show:

$$\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k](W^Y_{pre}) = L = l$$

which holds trivially and finishes the case.

Case $pre = pre_1 \wedge pre_2$
Assume:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre_1 \wedge pre_2$$

Then from Table 3.1 we have:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre_1 \text{ and } (\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre_2$$

Let's also assume:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(pre_1 \wedge pre_2)$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. From Table 3.3 we have:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_{k_1} \mapsto l_{k_1}]) \models^{\#}_{\beta} g(pre_1)$$

and

$$(\varrho, \varsigma[Y_{k_1+1} \mapsto l_{k_1+1}] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(pre_2)$$

The induction hypothesis gives:

$$\varsigma[Y_1 \mapsto l_1] \cdots [Y_{k_1} \mapsto l_{k_1}](W^Y_{pre_1}) = l \tag{A.16}$$

and

$$\varsigma[Y_{k_1+1} \mapsto l_{k_1+1}] \cdots [Y_k \mapsto l_k](W^Y_{pre_2}) = l \tag{A.17}$$

Since the definition of function g ensures linearity; meaning that variables Y_i $(1 \leq i \leq k)$ occurring in pre_1 and pre_2 are pairwise disjoint, and from $W^Y_{pre} = W^Y_{pre_1} \sqcap W^Y_{pre_2}$, it follows that:

$$\varsigma[Y_1 \mapsto l_1] \cdots [Y_{k_1} \mapsto l_{k_1}] \cdots [Y_k \mapsto l_k](W^Y_{pre}) = l$$

which was required and finishes the case.

Lemma 3.10: Assume that $(\varrho, \varsigma) \models^{\#}_{\beta} g(cl)$. Then $(\varrho, \varsigma) \models_{\beta} cl$.

PROOF. **Case:** $\forall Y : pre \Rightarrow R(\vec{u}; Y)$.
Assume:

$$(\varrho, \varsigma) \models^{\#}_{\beta} \forall Y_1 \cdots \forall Y_k : g(pre) \Rightarrow R(\vec{u}; W^Y_{pre})$$

That is for all $l_1, \cdots, l_k$, $(l_i \neq \bot,\ 1 \leq i \leq k)$:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(pre) \Rightarrow R(\vec{u}; W^Y_{pre}) \tag{A.18}$$

We want to show that for all l, $l \neq \bot$:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre \Rightarrow R(\vec{u}; Y)$$

Let's assume:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre$$

Hence we need to prove:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} R(\vec{u}; Y)$$

Which according to Table 3.1 is equivalent to:

$$l \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Let's also assume:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(pre),\ \ l_i \neq \bot\ (1 \leq i \leq k) \tag{A.19}$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. Then Lemma 3.9 gives:

$$(\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])(W^Y_{pre}) = l \tag{A.20}$$

Then from (A.18) and (A.19) we have:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} R(\vec{u}; W^Y_{pre})$$

Hence:

$$(\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])(W^Y_{pre}) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Then from (A.20) we get the required:

$$l = (\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])(W^Y_{pre}) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Case: $\forall Y : pre \Rightarrow R(\vec{u}; [v])$.
Assume:

$$(\varrho, \varsigma) \models^{\#}_{\beta} \forall Y_1 \cdots \forall Y_k : g(pre) \Rightarrow R(\vec{u}; W^Y_{pre})$$

That is for all $l_1, \cdots, l_k$, ($l_i \neq \bot$, $1 \leq i \leq k$):

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(pre) \Rightarrow R(\vec{u}; W^Y_{pre}) \tag{A.21}$$

We want to show that for all l, $l \neq \bot$:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre \Rightarrow R(\vec{u}; [v])$$

Let's assume:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre$$

Hence we need to prove:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} R(\vec{u}; [v])$$

Which according to Table 3.1 is equivalent to:

$$\beta(\varsigma(v)) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Let's also assume:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(pre),\ \ l_i \neq \bot\ (1 \leq i \leq k) \tag{A.22}$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. Then Lemma 3.9 gives:

$$(\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])(W^Y_{pre}) = l \tag{A.23}$$

Then from (A.21) and (A.22) we have:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} R(\vec{u}; W^Y_{pre})$$

Hence:

$$(\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])(W^Y_{pre}) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Since $W^Y_{pre} = [v]$, we have:

$$(\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])([v]) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

It follows that:

$$\beta(\varsigma(v)) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

quod erat demonstrandum.

Lemma 3.11: Assume that $(\varrho, \varsigma) \models_{\beta} cl$. Then $(\varrho, \varsigma) \models^{\#}_{\beta} g(cl)$.

PROOF. **Case** $\forall Y : pre \Rightarrow R(\vec{u}; Y)$
Assume:

$$(\varrho, \varsigma) \models_{\beta} \forall Y : pre \Rightarrow R(\vec{u}; Y)$$

Then from Table 3.1 we know that:

$$\forall l \in \mathcal{L}_{\neq \bot} : (\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre \Rightarrow R(\vec{u}; Y) \tag{A.24}$$

We want to show that:

$$(\varrho, \varsigma) \models^{\#}_{\beta} \forall Y_1 \cdots \forall Y_k : g(pre) \Rightarrow R(\vec{u}; W^Y_{pre})$$

That is for all $l_1, \cdots, l_k$, $l_i \neq \bot, (1 \leq i \leq k)$:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(pre) \Rightarrow R(\vec{u}; W^Y_{pre})$$

Assume:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} g(pre)$$

Then, we need to show:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^{\#}_{\beta} R(\vec{u}; W^Y_{pre})$$

Assume also:

$$(\varrho, \varsigma[Y \mapsto l]) \models_{\beta} pre \tag{A.25}$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. Then from Lemma 3.9 we have:

$$l = (\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])(W^Y_{pre})$$

Then from (A.24) and (A.25) we know:

$$(\varrho, \varsigma[Y \mapsto l]) \models_\beta R(\vec{u}; Y)$$

Hence, from Table 3.1 we know:

$$\varsigma[Y \mapsto l](Y) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

which equals to:

$$l \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Since $l = (\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])(W^Y_{pre})$ we have:

$$l = (\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])(W^Y_{pre}) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Thus from Table 3.3 we get:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^\#_\beta R(\vec{u}; W^Y_{pre})$$

quod erat demonstrandum.

Case $\forall Y : pre \Rightarrow R(\vec{u}; [v])$
Assume:

$$(\varrho, \varsigma) \models_\beta \forall Y : pre \Rightarrow R(\vec{u}; [v])$$

Then from Table 3.1 we know that:

$$\forall l \in \mathcal{L}_{\neq\perp} : (\varrho, \varsigma[Y \mapsto l]) \models_\beta pre \Rightarrow R(\vec{u}; [v]) \quad \text{(A.26)}$$

We want to show that:

$$(\varrho, \varsigma) \models^\#_\beta \forall Y_1 \cdots \forall Y_k : g(pre) \Rightarrow R(\vec{u}; W^Y_{pre})$$

That is for all $l_1, \cdots, l_k$, $l_i \neq \perp, (1 \leq i \leq k)$:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^\#_\beta g(pre) \Rightarrow R(\vec{u}; W^Y_{pre})$$

Assume:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^\#_\beta g(pre)$$

Then, we need to show:

$$(\varrho, \varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k]) \models^\#_\beta R(\vec{u}; W^Y_{pre})$$

Since $W^Y_{pre} = [v]$, it follows form Table 3.3 that we need to show:

$$\beta(\varsigma(v)) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

Assume also:

$$(\varrho, \varsigma[Y \mapsto l]) \models_\beta pre \quad \text{(A.27)}$$

where $l_i \sqsubseteq l$ for $1 \leq i \leq k$. Then from Lemma 3.9 we have:

$$l = (\varsigma[Y_1 \mapsto l_1] \cdots [Y_k \mapsto l_k])(W^Y_{pre})$$

From (A.26) and (A.27) we know:

$$(\varrho, \varsigma[Y \mapsto l]) \models R(\vec{u}; [v])$$

Hence, from Table 3.1 we know:

$$\beta(\varsigma[Y \mapsto A](v)) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

which equals to:

$$\beta(\varsigma(v)) \sqsubseteq \varrho(R)(\varsigma(\vec{u}))$$

quod erat demonstrandum.

A.7 Proof of Proposition 3.13

In order to prove Proposition 3.13 we first state and prove an auxiliary lemma.

Lemma A.3 *If* $\varrho = \bigsqcap_{\Delta} M$ *and* $(\varrho', \zeta, \varsigma) \models_\beta cl_j$ *for all* $\varrho' \in M$ *then* $(\varrho, \zeta, \varsigma) \models_\beta cl_j$.

Proof. We proceed by induction on j and in each case perform a structural induction on the form of the clause occurring in cl_j. We only consider cases involving function terms in the lattice component. For other cases, see Appendix A.3.

Case: $cl_j = R(\vec{u}; f(\vec{V'}))$

Assume that for all $\varrho' \in M$

$$(\varrho', \varsigma, \zeta) \models_\beta R(\vec{u}; f(\vec{V'}))$$

From the semantics of LLFP we have that for all $\varrho' \in M$

$$\varrho'(R)(\varsigma(\vec{u})) \sqsupseteq [\![f(\vec{V'})]\!](\zeta, \varsigma) = \zeta(f)[\![\vec{V'}]\!](\zeta, \varsigma)$$

It follows that:

$$\bigsqcap\{\varrho'(R)(\varsigma(\vec{u})) \mid \varrho' \in M\} \sqsupseteq [\![f(\vec{V'})]\!](\zeta, \varsigma) = \zeta(f)[\![\vec{V'}]\!](\zeta, \varsigma)$$

Since $M_j \subseteq M$, we have:

$$\bigsqcap\{\varrho'(R)(\varsigma(\vec{u})) \mid \varrho' \in M_j\} \sqsupseteq [\![f(\vec{V'})]\!](\zeta, \varsigma) = \zeta(f)[\![\vec{V'}]\!](\zeta, \varsigma)$$

We know that $\text{rank}(R) = j$; hence $\varrho(R) = \lambda\vec{a}.\bigsqcap\{\varrho'(R)(\vec{a}) \mid \varrho' \in M_j\}$; thus

$$\varrho(R)(\varsigma(\vec{u})) = \bigsqcap\{\varrho'(R)(\varsigma(\vec{u})) \mid \varrho' \in M_j\} \sqsupseteq [\![f(\vec{V'})]\!](\zeta, \varsigma)$$

Which according to Table 3.1 is equivalent to

$$(\varrho, \varsigma, \zeta) \models_\beta R(\vec{u}; f(\vec{V'}))$$

and finishes the case, and the proof. □

Proposition 3.13. Assume cls is a stratified LLFP clause sequence, ς_0 and ζ_0 are interpretations of free variables and function symbols in cls, respectively. Furthermore, ϱ_0 is an interpretation of all relations of rank 0. Then

$$\{\varrho \mid (\varrho, \zeta_0, \varsigma_0) \models_\beta cls \wedge \forall R : \text{rank}(R) = 0 \Rightarrow \varrho_0(R) \sqsubseteq \varrho(R)\}$$

is a Moore family.

Proof. The result follows from Lemma A.3. □

A.8 Proof of Lemma 4.3

Proof.

Reflexivity $\forall \varrho \in \Delta : \varrho \sqsubseteq \varrho$.

To show that $\varrho \sqsubseteq \varrho$ let us take $j = s$. If $\text{rank}(R) < j$ then $\varrho(R) = \varrho(R)$ as required. Otherwise if $\text{rank}(R) = j$ and either R is a defined relation or $j = 0$, then form $\varrho(R) = \varrho(R)$ we get $\varrho(R) \subseteq \varrho(R)$. The last case is when $\text{rank}(R) = j$ and R is a constrained relation. Then from $\varrho(R) = \varrho(R)$ we get $\varrho(R) \supseteq \varrho(R)$. Thus we get the required $\varrho \sqsubseteq \varrho$.

Transitivity $\forall \varrho_1, \varrho_2, \varrho_3 \in \Delta : \varrho_1 \sqsubseteq \varrho_2 \wedge \varrho_2 \sqsubseteq \varrho_3 \Rightarrow \varrho_1 \sqsubseteq \varrho_3$.

Let us assume that $\varrho_1 \sqsubseteq \varrho_2 \wedge \varrho_2 \sqsubseteq \varrho_3$. From $\varrho_i \sqsubseteq \varrho_{i+1}$ we have j_i such that conditions (a)–(d) are fulfilled for $i = 1, 2$. Let us take j to be the minimum of j_1 and j_2. Now we need to verify that conditions (a)–(d) hold for j. If $\text{rank}(R) < j$ we have $\varrho_1(R) = \varrho_2(R)$ and $\varrho_2(R) = \varrho_3(R)$. It follows that $\varrho_1(R) = \varrho_3(R)$, hence (a) holds. Now let us assume that $\text{rank}(R) = j$ and either R is a defined relation or $j = 0$. We have $\varrho_1(R) \subseteq \varrho_2(R)$ and $\varrho_2(R) \subseteq \varrho_3(R)$ and from transitivity of $\subseteq$ we get $\varrho_1(R) \subseteq \varrho_3(R)$, which gives (b). Alternatively $\text{rank}(R) = j$ and R is a constrained relation. We have $\varrho_1(R) \supseteq \varrho_2(R)$ and $\varrho_2(R) \supseteq \varrho_3(R)$ and from transitivity of $\supseteq$ we get $\varrho_1(R) \supseteq \varrho_3(R)$, thus (c) holds. Let us now assume that $j \neq s$, hence $\varrho_i(R) \neq \varrho_{i+1}(R)$ for some $R \in \mathcal{R}$ and $i = 1, 2$. Without loss of generality let us assume that $\varrho_1(R) \neq \varrho_2(R)$. In case R is a defined relation we have $\varrho_1(R) \subsetneq \varrho_2(R)$ and $\varrho_2(R) \subseteq \varrho_3(R)$, hence $\varrho_1(R) \neq \varrho_3(R)$. Similarly in case R is a constrained relation we have $\varrho_1(R) \supsetneq \varrho_2(R)$ and $\varrho_2(R) \supseteq \varrho_3(R)$. Hence $\varrho_1(R) \neq \varrho_3(R)$, and (d) holds.

Anti-symmetry $\forall \varrho_1, \varrho_2 \in \Delta : \varrho_1 \sqsubseteq \varrho_2 \wedge \varrho_2 \sqsubseteq \varrho_1 \Rightarrow \varrho_1 = \varrho_2$.

Let us assume $\varrho_1 \sqsubseteq \varrho_2$ and $\varrho_2 \sqsubseteq \varrho_1$. Let j be minimal such that $\text{rank}(R) = j$ and $\varrho_1(R) \neq \varrho_2(R)$ for some $R \in \mathcal{R}$. If $j = 0$ or R is a defined relation, then we have $\varrho_1(R) \subseteq \varrho_2(R)$ and $\varrho_2(R) \subseteq \varrho_1(R)$. Hence $\varrho_1(R) = \varrho_2(R)$ which is a contradiction. Similarly if R is a constrained relation we have $\varrho_1(R) \supseteq \varrho_2(R)$ and $\varrho_2(R) \supseteq \varrho_1(R)$. It follows that $\varrho_1(R) = \varrho_2(R)$, which again is a contradiction. Thus it must be the case that $\varrho_1(R) = \varrho_2(R)$ for all $R \in \mathcal{R}$. □

A.9 Proof of Lemma 4.4

PROOF. First we prove that $\bigsqcap M$ is a lower bound of M; that is $\bigsqcap M \sqsubseteq \varrho$ for all $\varrho \in M$. Let j be maximum such that $\varrho \in M_j$; since $M = M_0$ and $M_j \supseteq M_{j+1}$ clearly such j exists. From definition of M_j it follows that $(\bigsqcap M)(R) = \varrho(R)$ for all R with $\text{rank}(R) < j$; hence (a) holds.

If $\text{rank}(R) = j$ and either R is a defined relation or $j = 0$ we have $(\bigsqcap M)(R) = \bigcap\{\varrho'(R) \mid \varrho' \in M_j\} \subseteq \varrho(R)$ showing that (b) holds.

Similarly, if R is a constrained relation with $\text{rank}(R) = j$ we have $(\bigsqcap M)(R) = \bigcup\{\varrho'(R) \mid \varrho' \in M_j\} \supseteq \varrho(R)$ showing that (c) holds.

Finally let us assume that $j \neq s$; we need to show that there is some R with $\text{rank}(R) = j$ such that $(\bigsqcap M)(R) \neq \varrho(R)$. Since we know that j is maximum such that $\varrho \in M_j$, it follows that $\varrho \notin M_{j+1}$, hence there is a relation R with $\text{rank}(R) = j$ such that $(\bigsqcap M)(R) \neq \varrho(R)$; thus (d) holds.

Now we need to show that $\bigsqcap M$ is the greatest lower bound. Let us assume that $\varrho' \sqsubseteq \varrho$ for all $\varrho \in M$, and let us show that $\varrho' \sqsubseteq \bigsqcap M$. If $\varrho' = \bigsqcap M$ the result holds vacuously, hence let us assume $\varrho' \neq \bigsqcap M$. Then there exists a minimal j such that $(\bigsqcap M)(R) \neq \varrho'(R)$ for some R with $\text{rank}(R) = j$. Let us first consider R such that $\text{rank}(R) < j$. By our choice of j we have $(\bigsqcap M)(R) = \varrho'(R)$ hence (a) holds.

Next assume that $\text{rank}(R) = j$ and either R is a defined relation of $j = 0$. Then $\varrho' \sqsubseteq \varrho$ for all $\varrho \in M_j$. It follows that $\varrho'(R) \subseteq \varrho(R)$ for all $\varrho \in M_j$. Thus we have $\varrho'(R) \subseteq \bigcap\{\varrho(R) \mid \varrho \in M_j\}$. Since $(\bigsqcap M)(R) = \bigcap\{\varrho(R) \mid \varrho \in M_j\}$, we have $\varrho'(R) \subseteq (\bigsqcap M)(R)$ which proves (b).

Now assume $\text{rank}(R) = j$ and R is a constrained relation. We have that $\varrho' \sqsubseteq \varrho$ for all $\varrho \in M_j$. Since R is a constrained relation it follows that $\varrho'(R) \supseteq \varrho(R)$ for all $\varrho \in M_j$. Thus we have $\varrho'(R) \supseteq \bigcup\{\varrho(R) \mid \varrho \in M_j\}$. Since $(\bigsqcap M)(R) = \bigcup\{\varrho(R) \mid \varrho \in M_j\}$, we have $\varrho'(R) \supseteq (\bigsqcap M)(R)$ which proves (c).

Finally since we assumed that $(\bigsqcap M)(R) \neq \varrho'(R)$ for some R with $\text{rank}(R) = j$, it follows that (d) holds. Thus we proved that $\varrho' \sqsubseteq \bigsqcap M$. □

A.10 Proof of Proposition 4.5

In order to prove Proposition 4.5 we first state and prove two auxiliary lemmas.

Definition A.4 We introduce an ordering $\subseteq_{/j}$ defined by $\varrho_1 \subseteq_{/j} \varrho_2$ if and only if

- $\forall R : \text{rank}(R) < j \Rightarrow \varrho_1(R) = \varrho_2(R)$
- $\forall R : \text{rank}(R) = j \Rightarrow \varrho_1(R) \subseteq \varrho_2(R)$

Lemma A.5 *Assume a condition cond occurs in* cl_j*, and let* ς *be a valuation of free variables in cond. If* $\varrho_1 \subseteq_{/j} \varrho_2$ *and* $(\varrho_1, \varsigma) \models cond$ *then* $(\varrho_2, \varsigma) \models cond$.

Proof. We proceed by induction on j and in each case perform a structural induction on the form of the condition *cond* occurring in cl_j.
Case: $cond = R(\vec{x})$
Assume $\varrho_1 \subseteq_{/j} \varrho_2$ and

$$(\varrho_1, \varsigma) \models R(\vec{x})$$

From Table 4.1 it follows that

$$[\![\vec{x}]\!]([\,], \varsigma) \in \varrho_1(R)$$

Depending of the rank of R we have two sub-cases.
(1) Let $\text{rank}(R) < j$, then from Definition A.4 we know that $\varrho_1(R) = \varrho_2(R)$ and hence

$$[\![\vec{x}]\!]([\,], \varsigma) \in \varrho_2(R)$$

Which according to Table 4.1 is equivalent to

$$(\varrho_2, \varsigma) \models R(\vec{x})$$

(2) Let us now assume $\text{rank}(R) = j$, then from Definition A.4 we know that $\varrho_1(R) \subseteq \varrho_2(R)$ and hence

$$[\![\vec{x}]\!]([\,], \varsigma) \in \varrho_2(R)$$

which is equivalent to

$$(\varrho_2, \varsigma) \models R(\vec{x})$$

and finishes the case.
Case: $cond = \neg R(\vec{x})$
Assume $\varrho_1 \subseteq_{/j} \varrho_2$ and

$$(\varrho_1, \varsigma) \models \neg R(\vec{x})$$

From Table 4.1 it follows that

$$[\![\vec{x}]\!]([\,],\varsigma) \notin \varrho_1(R)$$

Since $\text{rank}(R) < j$, then from Definition A.4 we have $\varrho_1(R) = \varrho_2(R)$ and hence

$$[\![\vec{x}]\!]([\,],\varsigma) \notin \varrho_2(R)$$

Which according to Table 4.1 is equivalent to

$$(\varrho_2, \varsigma) \models \neg R(\vec{x})$$

Case: $\mathit{cond} = \mathit{cond}_1 \wedge \mathit{cond}_2$

Assume $\varrho_1 \subseteq_{/j} \varrho_2$ and

$$(\varrho_1, \varsigma) \models \mathit{cond}_1 \wedge \mathit{cond}_2$$

From Table 4.1 it follows that

$$(\varrho_1, \varsigma) \models \mathit{cond}_1 \text{ and } (\varrho_1, \varsigma) \models \mathit{cond}_2$$

The induction hypothesis gives

$$(\varrho_2, \varsigma) \models \mathit{cond}_1 \text{ and } (\varrho_2, \varsigma) \models \mathit{cond}_2$$

Hence we have

$$(\varrho_2, \varsigma) \models \mathit{cond}_1 \wedge \mathit{cond}_2$$

Case: $\mathit{cond} = \mathit{cond}_1 \vee \mathit{cond}_2$

Assume $\varrho_1 \subseteq_{/j} \varrho_2$ and

$$(\varrho_1, \varsigma) \models \mathit{cond}_1 \vee \mathit{cond}_2$$

From Table 4.1 it follows that

$$(\varrho_1, \varsigma) \models \mathit{cond}_1 \text{ or } (\varrho_1, \varsigma) \models \mathit{cond}_2$$

The induction hypothesis gives

$$(\varrho_2, \varsigma) \models \mathit{cond}_1 \text{ or } (\varrho_2, \varsigma) \models \mathit{cond}_2$$

Hence we have

$$(\varrho_2, \varsigma) \models \mathit{cond}_1 \vee \mathit{cond}_2$$

Case: $\mathit{cond} = \exists x : \mathit{cond}'$

Assume $\varrho_1 \subseteq_{/j} \varrho_2$ and

$$(\varrho_1, \varsigma) \models \exists x : \mathit{cond'}$$

From Table 4.1 it follows that

$$\exists a \in \mathcal{U} : (\varrho_1, \varsigma[x \mapsto a]) \models \mathit{cond'}$$

The induction hypothesis gives

$$\exists a \in \mathcal{U} : (\varrho_2, \varsigma[x \mapsto a]) \models \mathit{cond'}$$

Hence from Table 4.1 we have

$$(\varrho_2, \varsigma) \models \exists x : \mathit{cond'}$$

Case: $\mathit{cond} = \forall x : \mathit{cond'}$

Assume $\varrho_1 \subseteq_{/j} \varrho_2$ and

$$(\varrho_1, \varsigma) \models \forall x : \mathit{cond'}$$

From Table 4.1 it follows that

$$\forall a \in \mathcal{U} : (\varrho_1, \varsigma[x \mapsto a]) \models \mathit{cond'}$$

The induction hypothesis gives

$$\forall a \in \mathcal{U} : (\varrho_2, \varsigma[x \mapsto a]) \models \mathit{cond'}$$

Hence from Table 4.1 we have

$$(\varrho_2, \varsigma) \models \forall x : \mathit{cond'}$$

□

Lemma A.6 *If* $\varrho = \bigsqcap M$ *and* $(\varrho', \zeta, \varsigma) \models cl_j$ *for all* $\varrho' \in M$ *then* $(\varrho, \zeta, \varsigma) \models cl_j$.

Proof. We proceed by induction on j and in each case perform a structural induction on the form of the clause cl occurring in cl_j.

Case: $cl_j = \mathit{define}(\mathit{cond} \Rightarrow R(\vec{u}))$

Assume

$$\forall \varrho' \in M : (\varrho', \zeta, \varsigma) \models \mathit{cond} \Rightarrow R(\vec{u}) \tag{A.28}$$

Let us also assume

$$(\varrho, \varsigma) \models \mathit{cond}$$

Since $\varrho = \bigsqcap M$ we know that

$$\forall \varrho' \in M : \varrho \sqsubseteq \varrho' \tag{A.29}$$

Let R' occur in *cond*. We have two possibilities; either $\mathrm{rank}(R') = j$ and R' is a defined relation, then from (A.29) if follows that $\varrho(R') \subseteq \varrho'(R')$. Alternatively $\mathrm{rank}(R') < j$ and from (A.29) it follows that $\varrho(R') = \varrho'(R')$. Hence from Definition A.4 we have that $\varrho \subseteq_{/j} \varrho'$. Thus from Lemma A.5 it follows that

$$\forall \varrho' \in M : (\varrho', \varsigma) \models cond$$

Hence from (A.28) we have

$$\forall \varrho' \in M : (\varrho', \zeta, \varsigma) \models R(\vec{u})$$

Which from Table 4.1 is equivalent to

$$\forall \varrho' \in M : [\![\vec{u}]\!](\zeta, \varsigma) \in \varrho'(R)$$

It follows that

$$[\![\vec{u}]\!](\zeta, \varsigma) \in \bigcap\{\varrho'(R) \mid \varrho' \in M\} = \varrho(R)$$

Which from Table 4.1 is equivalent to

$$(\varrho, \zeta, \varsigma) \models R(\vec{u})$$

and finishes the case.

Case: $cl_j = \mathit{define}(\mathit{def}_1 \wedge \mathit{def}_2)$

Assume

$$\forall \varrho' \in M : (\varrho', \zeta, \varsigma) \models \mathit{def}_1 \wedge \mathit{def}_2$$

From Table 4.1 we have that for all $\varrho' \in M$

$$(\varrho', \zeta, \varsigma) \models \mathit{def}_1 \text{ and } (\varrho', \zeta, \varsigma) \models \mathit{def}_2$$

The induction hypothesis gives

$$(\varrho, \zeta, \varsigma) \models \mathit{def}_1 \text{ and } (\varrho, \zeta, \varsigma) \models \mathit{def}_2$$

Hence from Table 4.1 we have

$$(\varrho, \zeta, \varsigma) \models \mathit{def}_1 \wedge \mathit{def}_2$$

Case: $cl_j = \mathit{define}(\forall x : \mathit{def})$

Assume

$$\forall \varrho' \in M : (\varrho', \zeta, \varsigma) \models \forall x : def \quad (A.30)$$

From Table 4.1 we have that

$$\varrho' \in M : \forall a \in \mathcal{U} : (\varrho', \zeta, \varsigma[x \mapsto a]) \models def$$

Thus

$$\forall a \in \mathcal{U} : \varrho' \in M : (\varrho', \zeta, \varsigma[x \mapsto a]) \models def$$

The induction hypothesis gives

$$\forall a \in \mathcal{U} : (\varrho, \zeta, \varsigma[x \mapsto a]) \models def$$

Hence from Table 4.1 we have

$$(\varrho, \zeta, \varsigma) \models \forall x : def$$

Case: $cl_j = constrain(R(\vec{u}) \Rightarrow cond)$

Assume

$$\forall \varrho' \in M : (\varrho', \zeta, \varsigma) \models R(\vec{u}) \Rightarrow cond \quad (A.31)$$

Let us also assume

$$(\varrho, \zeta, \varsigma) \models R(\vec{u})$$

From Table 4.1 it follows that

$$[\![\vec{u}]\!](\zeta, \varsigma) \in \bigcup \{\varrho'(R) \mid \varrho' \in M\}$$

Thus there is some $\varrho' \in M$ such that

$$[\![\vec{u}]\!](\zeta, \varsigma) \in \varrho'(R)$$

From (A.31) it follows that

$$(\varrho', \varsigma) \models cond$$

Since $\varrho = \bigsqcap M$ we know that

$$\forall \varrho' \in M : \varrho \sqsubseteq \varrho' \quad (A.32)$$

Let R' occur in *cond*. We have two possibilities; either $\text{rank}(R') = j$ and R' is a constrained relation, then from (A.32) if follows that $\varrho(R') \supseteq \varrho'(R')$. Alternatively $\text{rank}(R') < j$ and from (A.32) it follows that $\varrho(R') = \varrho'(R')$. Hence from Definition A.4 we have that $\varrho' \subseteq_{/j} \varrho$. Thus from Lemma A.5 it follows that

$$(\varrho, \varsigma) \models cond$$

which finishes the case.

Case: $cl_j = constrain(con_1 \land con_2)$

Assume
$$\forall \varrho' \in M : (\varrho', \zeta, \varsigma) \models con_1 \land con_2$$
From Table 4.1 we have that for all $\varrho' \in M$
$$(\varrho', \zeta, \varsigma) \models con_1 \;\; and \; (\varrho', \zeta, \varsigma) \models con_2$$
The induction hypothesis gives
$$(\varrho, \zeta, \varsigma) \models con_1 \;\; and \; (\varrho, \zeta, \varsigma) \models con_2$$
Hence from Table 4.1 we have
$$(\varrho, \zeta, \varsigma) \models con_1 \land con_2$$

Case: $cl_j = constrain(\forall x : con)$

Assume
$$\forall \varrho' \in M : (\varrho', \zeta, \varsigma) \models \forall x : con \tag{A.33}$$
From Table 4.1 we have that
$$\varrho' \in M : \forall a \in \mathcal{U} : (\varrho', \zeta, \varsigma[x \mapsto a]) \models con$$
Thus
$$\forall a \in \mathcal{U} : \varrho' \in M : (\varrho', \zeta, \varsigma[x \mapsto a]) \models con$$
The induction hypothesis gives
$$\forall a \in \mathcal{U} : (\varrho, \zeta, \varsigma[x \mapsto a]) \models con$$
Hence from Table 4.1 we have
$$(\varrho, \zeta, \varsigma) \models \forall x : con$$

□

Proposition 4.5: Assume *cls* is a stratified LFP formula, ς_0 and ζ_0 are interpretations of the free variables and function symbols in *cls*, respectively. Furthermore, ϱ_0 is an interpretation of all relations of rank 0. Then $\{\varrho \mid (\varrho, \zeta_0, \varsigma_0) \models cls \land \forall R : \text{rank}(R) = 0 \Rightarrow \varrho(R) \supseteq \varrho_0(R)\}$ is a Moore family. Proof. The result follows from Lemma A.6. □

A.11 Proof of Proposition 4.6

Proposition 4.6: For a finite universe $\mathcal{U}$, the best solution ϱ such that $\varrho_0 \sqsubseteq \varrho$ of a LFP formula $cl_1, \ldots, cl_s$ (w.r.t. an interpretation of the constant symbols) can be computed in time

$$\mathcal{O}(|\varrho_0| + \sum_{1 \leq i \leq s} |cl_i||\mathcal{U}|^{k_i})$$

where k_i is the maximal nesting depth of quantifiers in the cl_i and $|\varrho_0|$ is the sum of cardinalities of predicates $\varrho_0(R)$ of rank 0. We also assume unit time hash table operations (as in [39]). PROOF. Let cl_i be a clause corresponding to the i-th layer. Since cl_i can be either a define clause, or a constrain clause, we have two cases.

Let us first assume that $cl_i = \textit{define}(\textit{def})$; the proof proceed in three phases. First we transform *def* to *def'* by replacing every universal quantification $\forall x : \textit{def}_{cl}$ by the conjunction of all $|\mathcal{U}|$ possible instantiations of $\textit{def}_{cl}$, every existential quantification $\exists x : \textit{cond}$ by the disjunction of all $|\mathcal{U}|$ possible instantiations of *cond* and every universal quantification $\forall x : \textit{cond}$ by the conjunction of all $|\mathcal{U}|$ possible instantiations of *cond*. The resulting clause *def'* is logically equivalent to *def* and has size

$$\mathcal{O}(|\mathcal{U}|^k|\textit{def}|) \tag{A.34}$$

where k is the maximal nesting depth of quantifiers in *def*. Furthermore, *def'* is *boolean*, which means that there are no variables or quantifiers and all literals are viewed as nullary predicates.

In the second phase we transform the formula *def'*, being the result of the first phase, into a sequence of formulas $\textit{def}'' = \textit{def}'_1, \ldots, \textit{def}'_l$ as follows. We first replace all top-level conjunctions in *def'* with ",". Then we successively replace each formula by a sequence of simpler ones using the following rewrite rule

$$\textit{cond}_1 \vee \textit{cond}_2 \Rightarrow R(\vec{u}) \mapsto \textit{cond}_1 \Rightarrow Q_{new}, \textit{cond}_2 \Rightarrow Q_{new}, Q_{new} \Rightarrow R(\vec{u})$$

where Q_{new} is a fresh nullary predicate that is generated for each application of the rule. The transformation is completed as soon as no replacement can be done. The conjunction of the resulting define clauses is logically equivalent to *def'*.

To show that this process terminates and that the size of *def''* is at most a constant times the size of the input formula *def'* , we assign a cost to the formulae. Let us define the cost of a sequence of clauses as the sum of costs of all occurrences of predicate symbols and operators (excluding ","). In general, the cost of a symbol or operator is 1 except disjunction that counts 6. Then the above rule decreases the cost from $k+7$ to $k+6$, for suitable value of k. Since the cost of the initial sequence is at most 6 times the size of *def*, only a linear number of rewrite steps can be performed. Since each step increases the size at most by a constant, we conclude that the *def"* has increased just by a constant factor. Consequently, when applying this transformation to *def'*, we obtain a boolean formula without sharing of size as in (A.34).

The third phase solves the system that is a result of phase two, which can be done in linear time by the classical techniques of e.g. [25].

Let us now assume that the $cl_i = constrain(con)$. We begin by transforming *con* into a logically equivalent (modulo fresh predicates) *define* clause. The transformation is done by function f_i defined as

$$
\begin{array}{lcl}
f_i(constrain(con)) & = & define(g(con)), define(h_i(con)) \\
\\
g(\forall x : con) & = & \forall x : g(con) \\
g(con_1 \wedge con_2) & = & g(con_1) \wedge g(con_2) \\
g(R(\vec{u}) \Rightarrow cond) & = & (\neg cond[R^{\mathsf{C}}(\vec{u})/\neg R(\vec{u})] \Rightarrow R^{\mathsf{C}}(\vec{u})) \\
\\
h_i(\forall x : con) & = & \forall x : h_i(con) \\
h_i(con_1 \wedge con_2) & = & h_i(con_1) \wedge h_i(con_2) \\
h_i(R(\vec{u}) \Rightarrow cond) & = & let\ cond' = cond[true/(R'(\vec{v}) \mid \mathrm{rank}(R') = i)]\ in \\
 & & cond' \wedge \neg R^{\mathsf{C}}(\vec{u}) \Rightarrow R(\vec{u})
\end{array}
$$

where R^{C} is a new predicate corresponding to the complement of R. The size of the formula increases by a number of constraint predicates; hence the size of the input formula is increased by a constant factor. Then the proof proceeds as in case of *define* clause.

The three phases of the transformation result in the sequence of define clauses of size

$$\mathcal{O}(\sum_{1 \le i \le s} |cl_i||\mathcal{U}|^{k_i})$$

which can then be solved in linear time. We also need to take into account the size of the initial knowledge i.e. the cardinality of all predicates of rank 0; thus the overall worst case complexity is

$$\mathcal{O}(|\varrho_0| + \sum_{1 \le i \le s} |cl_i||\mathcal{U}|^{k_i})$$

□

A.12 Proof of Lemma 6.2

Proof. We conduct the proof by showing that each step of the transformation is semantics preserving.
Step 1: Renaming of variables is semantics preserving.
Step 2: We need to show that provided that

$$(\rho, \sigma) \models \forall x : pre \Rightarrow P(\vec{y}) \tag{A.35}$$

the following holds

$$(\rho, \sigma) \models (\forall x : pre) \Leftrightarrow (\rho, \sigma) \models (\forall x : P(\vec{y}))$$

Assume that $x \in fv(pre)$ and hence x appears in $\vec{y}$, since the other case holds trivially.
($\Rightarrow$) Assume that

$$(\rho, \sigma) \models (\forall x : pre)$$

From Table 2.1 we have

$$\forall a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre$$

Using (A.35) it follows that

$$\forall a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models P(\vec{y})$$

Which according to Table 2.1 is equivalent to

$$(\rho, \sigma) \models (\forall x : P(\vec{y}))$$

which was required and finishes this direction.
($\Leftarrow$) Assume that

$$(\rho, \sigma) \models (\forall x : P(\vec{y}))$$

From Table 2.1 we have

$$\forall a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models P(\vec{y})$$

Since the clause (A.35) is the only one asserting predicate P it must be the case that

$$\forall a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre$$

According to Table 2.1 it is equivalent to

$$(\rho, \sigma) \models (\forall x : pre)$$

which was required and finishes the case.
Step 3: We need to prove that

$$(\rho, \sigma) \models (\exists x : pre_1 \vee pre_2) \Leftrightarrow (\rho, \sigma) \models (\exists x : pre_1) \vee (\exists x : pre_2)$$

Assume that

$$(\rho, \sigma) \models (\exists x : pre_1 \vee pre_2)$$

According to Table 2.1 it is equivalent to

$$\exists a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre_1 \vee pre_2$$

It follows that

$$\exists a \in \mathcal{U} : ((\rho, \sigma[x \mapsto a]) \models pre_1 \vee (\rho, \sigma[x \mapsto a]) \models pre_1)$$

This is equivalent to

$$(\exists a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre_1) \vee (\exists a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre_1)$$

Which according to Table 2.1 is equivalent to

$$(\rho, \sigma) \models (\exists x : pre_1) \vee (\exists x : pre_2)$$

and finishes the case.
Step 4: Transformation into DNF is semantics preserving.
Step 5: We consider a simplified case for two disjuncts only. Assume that

$$(\rho, \sigma) \models (pre_1 \vee pre_2) \Rightarrow cl$$

From Table 2.1 we have

$$(\rho, \sigma) \not\models (pre_1 \vee pre_2) \vee (\rho, \sigma) \models cl$$

From De Morgan's laws we get

$$((\rho, \sigma) \not\models pre_1 \wedge (\rho, \sigma) \not\models pre_2) \vee (\rho, \sigma) \models cl$$

Which is equivalent to

$$((\rho, \sigma) \not\models pre_1 \vee (\rho, \sigma) \models cl) \wedge ((\rho, \sigma) \not\models pre_2 \vee (\rho, \sigma) \models cl)$$

From which it follows that

$$((\rho, \sigma) \models pre_1 \Rightarrow cl) \wedge ((\rho, \sigma) \models pre_2 \Rightarrow cl)$$

Hence from Table 2.1 we have

$$(\rho, \sigma) \models (pre_1 \Rightarrow cl) \wedge (pre_2 \Rightarrow cl)$$

which was required and finishes the case.
Step 6: Assume

$$(\rho, \sigma) \models pre \Rightarrow \forall x : cl$$

From Table 2.1 we have

$$((\rho, \sigma) \not\models pre) \vee ((\rho, \sigma) \models \forall x : cl)$$

Using Table 2.1 we get

$$((\rho, \sigma) \not\models pre) \vee (\forall a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models cl)$$

Since $x \notin fv(pre)$ the above is equivalent to

$$\forall a \in \mathcal{U} : ((\rho, \sigma[x \mapsto a]) \not\models pre) \vee ((\rho, \sigma[x \mapsto a]) \models cl)$$

It follows that

$$\forall a \in \mathcal{U} : (\rho, \sigma[x \mapsto a]) \models pre \Rightarrow cl$$

Which according to Table 2.1 is equivalent to

$$(\rho, \sigma) \models \forall x : (pre \Rightarrow cl)$$

which was required and finishes the case.
Step 7: Assume

$$(\rho, \sigma) \models \forall x : (pre \Rightarrow cl_1 \wedge cl_2)$$

From Table 2.1 we have that for all $a \in \mathcal{U}$

$$(\rho, \sigma[x \mapsto a]) \models (pre \Rightarrow cl_1 \wedge cl_2)$$

Which is equivalent to

$$((\rho, \sigma[x \mapsto a]) \not\models pre) \vee ((\rho, \sigma[x \mapsto a]) \models cl_1 \wedge (\rho, \sigma[x \mapsto a]) \models cl_2)$$

It follows that

$$((\rho, \sigma[x \mapsto a]) \not\models pre \vee (\rho, \sigma[x \mapsto a]) \models cl_1)$$

and

$$((\rho, \sigma[x \mapsto a]) \not\models pre \vee (\rho, \sigma[x \mapsto a]) \models cl_2)$$

Hence we have that for all $a \in \mathcal{U}$

$$((\rho, \sigma[x \mapsto a]) \models pre \Rightarrow cl_1) \wedge ((\rho, \sigma[x \mapsto a]) \models pre \Rightarrow cl_2)$$

According to Table 2.1 we have

$$((\rho, \sigma) \models \forall x : pre \Rightarrow cl_1) \wedge ((\rho, \sigma) \models \forall x : pre \Rightarrow cl_2)$$

which was required and finishes the sub-case.
Now let us assume

$$(\rho, \sigma) \models \forall x : (pre' \Rightarrow (pre'' \Rightarrow cl))$$

From Table 2.1 we have that for all $a \in \mathcal{U}$

$$(\rho, \sigma[x \mapsto a]) \models pre' \Rightarrow (pre'' \Rightarrow cl)$$

Which is equivalent to

$$(\rho, \sigma[x \mapsto a]) \not\models pre' \vee (\rho, \sigma[x \mapsto a]) \not\models pre' \vee (\rho, \sigma[x \mapsto a]) \models cl$$

Using De Margan's laws we get

$$(\rho, \sigma[x \mapsto a]) \not\models (pre' \wedge pre'') \vee (\rho, \sigma[x \mapsto a]) \models cl$$

Which is equivalent to

$$(\rho, \sigma[x \mapsto a]) \models (pre' \wedge pre'') \Rightarrow cl$$

According to Table 2.1 we have

$$(\rho, \sigma) \models \forall x : (pre' \wedge pre'') \Rightarrow cl$$

Which was required and finishes the case. □

A.13 Proof of Proposition 6.7

PROOF. The proof is based on the proof of Theorem 3.1 by Beeri and Ramakrishnan [8], and it uses the fact that for each clause in cls' if the adornment is dropped, we obtain a clause in cls.

First, we show that we can obtain a derivation of a fact in cls from a derivation of that fact in cls'. In order to do that simply notice that an unadorned version of the fact can be obtained by dropping the adornments and hence using unadorned version of the clauses and unadorned facts.

Now we want to show that we can obtain a derivation of a fact in cls' from a derivation of that fact in cls. We note an invariant in a bottom-up computation of cls and cls': namely that at each iteration the adorned versions of predicates in cls' contain the same tuples as the corresponding predicate in cls. □

A.14 Proof of Proposition 6.8

The proof is based on the proof of Proposition 3 by Balbin et al. [6]. In order to prove the Proposition 6.8 we state necessary definitions first.

Definition A.7 A ground instance cl_g of an ALFP clause cl is a clause constructed from cl by applying some substitution θ of constants from the universe $\mathcal{U}$ to all variables in cl.

Definition A.8 The set of ground clauses $g(cls)$ of a clause sequence cls is a subset of ground instances of clauses from cls such that for each $cl_g \in g(cls)$ the positive literals in preconditions are in the least model of cls.

Definition A.9 The set of relevant ground clauses for a query $R(\vec{u})$ on cls is a subset $g^{R(\vec{u})}(cls) \subseteq g(cls)$ defined as follows

- Initialize $g^{R(\vec{u})}(cls)$ with each clause in $g(cls)$ whose asserted literal is an instance of $R(\vec{u})$,
- Recursively, $g^{R(\vec{u})}(cls)$ contains each clause in $g(cls)$ whose asserted literal appeared as a positive literal in precondition of some clause in $g^{R(\vec{u})}(cls)$.

Definition A.10 The relevant tuples for a query $R(\vec{u})$ on cls is a set of tuples corresponding to the set of asserted literals in $g^{R(\vec{u})}(cls)$.

Definition A.11 Let $R(\vec{a})$ denote a ground atom appearing in the $g(cls)$, and let cls_g denote a ground clause in $g(cls)$. Define

$$\begin{array}{lcl} \text{height}(R(\vec{a})) & = & 1 + \min(\text{height}(cl_g) \mid cl_g \text{ asserts } R(\vec{a})) \\ \text{height}(cl_g) & = & \begin{cases} 0 & \text{if } cl_g \text{ is a fact} \\ \max(\text{height}(R_i(\vec{a}_i)) \mid R_i(\vec{a}_i) \in \text{Pre}(cl_g)) & \text{otherwise} \end{cases} \end{array}$$

Intuitively, the height of a ground atom in the least model of cls is the number of iterations required for that atom to appear as an asserted ground literal in a ground instance of a clause in $g(cls)$.

Proposition 6.8 Let cls be a closed and stratified adorned ALFP$^{\text{s}}$ formula and $R^\alpha(\vec{v})$ be a query on cls. Let cls' be the result of the magic set transformation. Then $cls \equiv^{R^\alpha(\vec{v})}_{R^\alpha(\vec{v})} cls'$.

PROOF.

Let ρ_1 and ρ_2 be two least models such that $(\rho_1, [\,]) \models cls$ and $(\rho_2, [\,]) \models cls'$.
($\Rightarrow$) First we prove that

$$\forall \vec{a} \in \varsigma_{\vec{v}}(\mathcal{U}) : \vec{a} \in \rho_1(R^\alpha) \Rightarrow \vec{a} \in \rho_2(R^\alpha)$$

which is equivalent to

$$\{\vec{a} \in \varsigma_{\vec{v}}(\mathcal{U}) \mid \vec{a} \in \rho_1(R^\alpha)\} \subseteq \{\vec{a} \in \varsigma_{\vec{v}}(\mathcal{U}) \mid \vec{a} \in \rho_2(R^\alpha)\}$$

Hence we show that the set of instances of $R^\alpha(\vec{v})$ in ρ_1 is a subset of the set of instances of $R^\alpha(\vec{v})$ in ρ_2. Let $g^{R^\alpha(\vec{v})}(cls)$ be a set of ground relevant clauses corresponding to the query $R^\alpha(\vec{v})$. We also define $\rho : \mathcal{R} \to \bigcup_k \mathcal{P}(\mathcal{U}^k)$ as an interpretation of ground atoms asserted in $g^{R^\alpha(\vec{v})}(cls)$. Formally

$$\rho(P) = \vec{a} \Leftrightarrow P(\vec{a}) \text{ is a ground atom asserted in } g^{R^\alpha(\vec{v})}(cls)$$

We know that

$$\{\vec{a} \in \varsigma_{\vec{v}}(\mathcal{U}) \mid \vec{a} \in \rho_1(R^\alpha)\} \subseteq \{\vec{a} \in \varsigma_{\vec{v}}(\mathcal{U}) \mid \vec{a} \in \rho(R^\alpha)\}$$

and we show that for all predicates P in $\mathcal{R}$ we have

$$\rho(P) \subseteq \rho_2(P)$$

In order to show the above, it is sufficient to prove that for all predicates P in $\mathcal{R}$

$$\vec{a}_P \in \rho(P) \Rightarrow \vec{a}_P \in \rho_2^{+P(\vec{a}_P)}(P)$$

where

$$\rho_2^{+P(\vec{a}_P)} = \lambda.R \begin{cases} \vec{a}_R & \text{if mkMagic}(P(\vec{a}_P)) = R(\vec{a}_R) \\ \rho_2(R) & \text{otherwise} \end{cases}$$

Hence essentially the interpretation $\rho_2^{+P(\vec{a}_P)}$ is exactly as ρ_2 except that it is extended with a fact mkMagic($P(\vec{a}_P)$) which acts as a seed for a fact $P(\vec{a}_P)$ that we are trying to derive. We conduct the proof by induction on the height of each literal $P(\vec{a})$ such that $\vec{a} \in \rho(P)$. In the following we write $P(\vec{a}) \in \rho$ if and only if $\vec{a} \in \rho(P)$.

Base case: For each $P(\vec{a}) \in \rho$ such that height$(P(\vec{a})) = 1$, $P(\vec{a})$ is a fact in cls and hence it is also a fact in cls'. It follows that also $P(\vec{a}) \in \rho_2^{+P(\vec{a})}$.

Inductive case: The induction hypothesis gives that for all $i < n$ if $P(\vec{a}) \in \rho$ and height$(P(\vec{a})) = i$, then $P(\vec{a}) \in \rho_2^{+P(\vec{a})}$. We need to show that if a ground atom $P_0(\vec{a}_0) \in \rho$ such that height$(P_0(\vec{a}_0)) = n$, then also $P_0(\vec{a}_0) \in \rho_2^{+P_0(\vec{a}_0)}$. Let cl be a clause in cls asserting P_0; thus cl is of the form

$$\forall \vec{x} : pre \Rightarrow P_0(\vec{v}_0)$$

We know that such clause exists since $\text{height}(P_0(\vec{a}_0)) = n$, where $n > 1$. The magic set transformation transforms the clause cl into cl' by inserting a literal $\text{mkMagic}(P_0(\vec{v}_0))$ into the precondition of cl. It follows that cl' is of the form

$$\forall \vec{x} : \text{mkMagic}(P_0(\vec{v}_0)) \wedge pre \Rightarrow P_0(\vec{v}_0)$$

In order for $P_0(\vec{a}_0)$ to be in ρ_2, there must be a ground instance cl'_g of that clause in $g^{R^\alpha(\vec{v})}(cls')$. The ground instance would be of the form

$$\text{mkMagic}(P_0(\vec{a}_0)) \wedge pre_g \Rightarrow P_0(\vec{a}_0)$$

Since by the definition of $\rho_2^{+P_0(\vec{a}_0)}$ we know that $\text{mkMagic}(P_0(\vec{a}_0))$ is in $\rho_2^{+P_0(\vec{a}_0)}$, we need to show that each ground instance $P_i(\vec{a}_i) \in \text{Pre}(cl'_g)$ is in $\rho_2^{+P_0(\vec{a}_0)}$. Recall that the function Pre returns all literals appearing in the given clause; it was formally defined in Section 6.2. According to the induction hypothesis, each $P_i(\vec{a}_i)$ is in ρ, since by definition of the height function, $\text{height}(P_i(\vec{a}_i)) = i < n$. Hence we have to show that $\text{mkMagic}(P_i(\vec{a}_i))$ is in $\rho_2^{+P_0(\vec{a}_0)}$. To show that $\text{mkMagic}(P_i(\vec{a}_i)) \in \rho_2^{+P_0(\vec{a}_0)}$ we consider each derived literal in the precondition of cl'. Let $P_1(\vec{v}_1)$ be the first derived literal in the precondition. By definition of SIPS, we know that there is a tuple $(V_1, \mathcal{W}_1, P_1(\vec{v}_1))$, where V_1 consists of base literals and possibly $P_0(\vec{v}_0)$. Thus there is a magic clause defining $\text{mkMagic}(P_1(\vec{v}_1))$, and the precondition consisting of the literals from V_1 and possibly $\text{mkMagic}(P_0(\vec{v}_0))$. Since the facts defining the base predicates in cls are also in cls', and $\text{mkMagic}(P_0(\vec{a}_0))$ is in $\rho_2^{+P_0(\vec{a}_0)}$, then $\text{mkMagic}(P_1(\vec{a}_1))$ is in $\rho_2^{+P_0(\vec{a}_0)}$. By the induction hypothesis it follows that $P_1(\vec{a}_1) \in \rho_2^{+P_0(\vec{a}_0)}$. Now, let us consider the next derived literal in the precondition, $P_2(\vec{v}_2)$, in the total order imposed by SIPS. Let the tuple in the SIPS be $(V_2, \mathcal{W}_2, P_2(\vec{v}_2))$. From the definition of the SIPS, we know that V_2 may contain $P_1(\vec{v}_1)$ and the corresponding magic clause would then contain $P_1(\vec{v}_1)$ in the precondition. Since we showed that $P_1(\vec{a}_1) \in \rho_2^{+P_0(\vec{a}_0)}$ we can use the analogous arguments to show that both $\text{mkMagic}(P_2(\vec{a}_2))$ and $P_2(\vec{a}_2)$ are in $\rho_2^{+P_0(\vec{a}_0)}$. Repeating this for all derived literals in the precondition, we conclude that also $P_0(\vec{a}_0)$ is in $\rho_2^{+P_0(\vec{a}_0)}$. By induction, the hypothesis holds for all literals $P(\vec{a})$ such that $\vec{a} \in \rho(P)$ and thus it completes the proof in this direction.

($\Leftarrow$) Now we need to prove that

$$\forall \vec{a} \in \varsigma_{\vec{v}}(\mathcal{U}) : \vec{a} \in \rho_2(R^\alpha) \Rightarrow \vec{a} \in \rho_1(R^\alpha)$$

which is equivalent to

$$\{\vec{a} \in \varsigma_{\vec{v}}(\mathcal{U}) \mid \vec{a} \in \rho_2(R^\alpha)\} \subseteq \{\vec{a} \in \varsigma_{\vec{v}}(\mathcal{U}) \mid \vec{a} \in \rho_1(R^\alpha)\}$$

The proof in this direction simply follows from the fact that clauses in cls' are more restrictive than these in cls. This is because the magic set transformation modifies the clauses in *cls* by inserting additional literal (corresponding to the magic predicate) into the preconditions of the clauses. □

Bibliography

[1] http://en.wikipedia.org/wiki/Pac-Man#Split-screen.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007.

[3] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, 1988.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[5] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3):259–262, 1987.

[6] Isaac Balbin, Graeme S. Port, Kotagiri Ramamohanarao, and Krishnamurthy Meenakshi. Efficient bottom-up computation of queries on stratified databases. *J. Log. Program.*, 11(3&4):295–344, 1991.

[7] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15, 1986.

[8] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *J. Log. Program.*, 10(3&4):255–299, 1991.

[9] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI*, pages 103–114, 2003.

[10] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[11] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

[12] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[13] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: $10^{2}0$ states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[14] Ashok K. Chandra and David Harel. Computable queries for relational data bases (preliminary report). In *STOC*, pages 309–318, 1979.

[15] Witold Charatonik and Andreas Podelski. Set-based analysis of reactive infinite-state systems. In *TACAS*, pages 358–375, 1998.

[16] B. Le Charlier and P. Van Hentenryck. A universal top-down fixpoint algorithm. Technical report, CS-92-25, Brown University, 1992.

[17] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[18] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[19] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[20] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.

[21] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, pages 269–295, 1992.

[22] Steven Dawson, C. R. Ramakrishnan, and David Scott Warren. Practical program analysis using general purpose logic programming systems - a case study. In *PLDI*, pages 117–126, 1996.

[23] Giorgio Delzanno and Andreas Podelski. Model checking in clp. In *TACAS*, pages 223–239, 1999.

[24] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[25] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.

[26] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nordic Journal of Computing*, 5(4):304–329, 1998.

[27] Piotr Filipiuk, Flemming Nielson, and Hanne Riis Nielson. Lattice based Least Fixed Point Logic. *CoRR*, abs/1207.5384, 2012.

[28] Piotr Filipiuk, Flemming Nielson, and Hanne Riis Nielson. Layered fixed point logic. In *PPDP*, 2012.

[29] Piotr Filipiuk, Hanne Riis Nielson, and Flemming Nielson. Explicit versus symbolic algorithms for solving ALFP constraints. *Electr. Notes Theor. Comput. Sci.*, 267(2):15–28, 2010.

[30] Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. *ACM Comput. Surv.*, 16(2):153–185, 1984.

[31] E. M. Clarke (Jr.), O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[32] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[33] Gary A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206, 1973.

[34] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *PODS*, pages 1–12, 2005.

[35] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[36] Ondrej Lhoták and Laurie J. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), 2008.

[37] J. Lind-Nielsen. Buddy, a binary decision diagram package.

[38] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[39] David A. McAllester. On the complexity analysis of static analyses. *J. ACM*, 49(4):512–537, 2002.

[40] Rocco De Nicola and Frits W. Vaandrager. Action versus state based logics for transition systems. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer, 1990.

[41] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[42] Flemming Nielson and Hanne Riis Nielson. Model checking *is* static analysis of modal logic. In *FOSSACS*, pages 191–205, 2010.

[43] Flemming Nielson, Hanne Riis Nielson, Hongyan Sun, Mikael Buchholtz, René Rydhof Hansen, Henrik Pilegaard, and Helmut Seidl. The succinct solver suite. In *TACAS*, pages 251–265, 2004.

[44] Flemming Nielson, Helmut Seidl, and Hanne Riis Nielson. A succinct solver for alfp. *Nord. J. Comput.*, 9(4):335–372, 2002.

[45] R. Paige. Symbolic finite differencing - part i. In *ESOP'90*, volume 432 of *LNCS*, pages 36–56. Springer, 1990.

[46] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David Scott Warren. Efficient model checking using tabled resolution. In *CAV*, pages 143–154, 1997.

[47] C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Yifei Dong, Xiaoqun Du, Abhik Roychoudhury, and V. N. Venkatakrishnan. Xmc: A logic-programming-based verification toolset. In *CAV*, pages 576–580, 2000.

[48] Thomas W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases (Book), ILPS*, pages 163–196, 1993.

[49] Thomas W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.

[50] J. Rohmer, R. Lescoeur, and Jean-Marc Kerisit. The alexander method - a technique for the processing of recursive axioms in deductive databases. *New Generation Comput.*, 4(3):273–285, 1986.

[51] Domenico Saccà and Carlo Zaniolo. Implementation of recursive queries for a data language based on pure horn logic. In *ICLP*, pages 104–135, 1987.

[52] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *POPL*, pages 38–48, 1998.

[53] David A. Schmidt and Bernhard Steffen. Program analysis *as* model checking of abstract interpretations. In Giorgio Levi, editor, *SAS*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer, 1998.

[54] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 59–76. ACM, 2005.

[55] Jeffrey D. Ullman. Implementation of logical query languages for databases. *ACM Trans. Database Syst.*, 10(3):289–321, 1985.

[56] Jeffrey D. Ullman. Bottom-up beats top-down for datalog. In *PODS*, pages 140–149, 1989.

[57] Richard J. Wallace. Why AC-3 is almost always better than AC4 for establishing arc consistency in csps. In *IJCAI*, pages 239–247, 1993.

[58] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *APLAS*, pages 97–118, 2005.

[59] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

[60] Yuanlin Zhang and Roland H. C. Yap. Making AC-3 an optimal algorithm. In *IJCAI*, pages 316–321, 2001.