Universitat Politècnica de València
Departamento de Sistemas Informáticos y Computación
Doctorado en Informática

Ph.D. Thesis

Logic-based techniques for program analysis and specification synthesis

Candidate:
Marco Antonio Feliú Gabaldón

Supervisors:
María Alpuente Frasnedo
Alicia Villanueva García

September 2013

Work partially supported by the EU (FEDER) and the Spanish MEC, under grants TIN 2004-7943-C04-02, TIN 2007-68118-C02, TIN 2010-21062-C02-02 and FPU AP2008-00608; by the Generalitat Valenciana, under grants PROMETEO/2011/052 and GV/2009/24; and by the Universitat Politècnica de València, under grant TACPAS/PAID 2007.

Author's address:

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera, s/n
46022 Valencia
España

To life.

Abstract

Program analysis techniques have many uses in the real world. However, there are aspects that can help improving its widespread adoption. This thesis deals with techniques aimed at improving two limiting aspects: the lack of customizability and the burdensome learning process.

The lack of customizability of program analysis comes from the high complexity of program analysis algorithms that prevents untrained developers from creating specific analysis that help to improve the quality of their software. Declarative program analysis aims at reducing the effort for implementing analyses by incrementing the level of abstraction of the specification language, and by offering an efficient execution of the analysis specification comparable with traditional implementations.

In this thesis, we improve declarative program analysis of JAVA programs based on the logic specification language Datalog in two aspects. On one hand, we translate Datalog specifications into *Boolean Equation Systems* for easing the distribution of the analysis computation and speeding it up. *Boolean Equation Systems* provide efficient and distributed algorithms for their evaluation and industrial tools that implement them. On the other hand, we translate Datalog specifications into *Rewriting Logic* theories to support the extension of the specification language Datalog for expressing more sophisticated analysis, for example, those involving the use of *reflection*.

Another contribution of this thesis is relative to the automated inference of specifications to assist techniques for improving software quality (program analysis, verification, debugging, documentation, ...). In particular, specifications are at the core of program analysis, since every program analysis checks the conformation of the program of interest with respect to a specification. Untrained developers may not be capable of formulating program specifications that could be used as input of other tools like static analyzers, testing or program verifiers, or just for human inspection and documentation. Automated inference of specifications aims at reducing the effort for creating software specifications by producing approximated ones without human intervention. We improve automated specification inference for the multiparadigm language CURRY and for object-oriented programs in general by proposing two new approaches. On one hand, we present a technique to infer algebraic specifications for CURRY programs by classifying expressions built from their

signature according to their abstract semantics. Contrary to other existing approaches, this technique allows to discriminate between *correct* and *possibly correct* parts of the specification. On the other hand, we present a technique to infer high-level specifications in the form of pre/post-conditions for object-oriented languages. This technique is formalized in the *Matching Logic* verification setting to allow the verification of the inferred specifications.

Resumen

Las técnicas de análisis de programas tienen una gran cantidad de aplicaciones en el mundo actual. Sin embargo, existen aún aspectos a mejorar dentro de las mismas que pueden ayudar a difundir más su uso. Esta tesis está dedicada a la mejora de dos aspectos del análisis de programas: la rigidez de sus técnicas y su complejo proceso de aprendizaje.

La rigidez de las técnicas de análisis de programas es debida a la gran complejidad de los algoritmos de análisis, que provocan que, sin un costoso entrenamiento y aprendizaje previos, los desarrolladores no puedan crear sus propios análisis para mejorar la calidad de sus programas. El análisis de programas declarativo tiene como objetivo reducir el esfuerzo en el diseño de implementación de análisis gracias al aumento del nivel de abstracción del lenguaje de especificación usado, siempre sin renunciar a ofrecer un método de ejecución del análisis comparable en términos de eficiencia a las implementaciones más tradicionales.

En esta tesis se mejora en dos aspectos la aproximación de análisis de programas JAVA basada en el lenguaje lógico de especificación Datalog. En primer lugar, se traducen especificaciones Datalog a *Sistemas de Ecuaciones Booleanas* con el fin de distribuir el cómputo de los análisis mejorando así sus tiempos de ejecución. Los *Sistemas de Ecuaciones Booleanas* están equipados con algoritmos eficientes y distribuidos para su evaluación y existen herramientas industriales que los implementan. En segundo lugar, se traducen especificaciones Datalog a teorías de *Lógica de Reescritura* con el fin de soportar la extensión del lenguaje de especificación Datalog de forma que puedan expresarse análisis más sofisticados, por ejemplo análisis que tengan en cuenta el uso de reflexión en los programas.

Otra contribución de esta tesis está relacionada con la automatización de la inferencia de especificaciones como mecanismo de apoyo a las técnicas que mejoran la calidad de los programas (análisis de programas, verificación, depuración, documentación, etc.). Podemos decir que las especificaciones son la base del análisis de programas en particular (y del resto de técnicas mencionadas en general) ya que cualquier análisis comprueba si el comportamiento del programa que está siendo analizado se corresponde con el dado por una especificación. Sin un entrenamiento previo, los desarrolladores pueden no ser capaces de formular especificaciones adecuadas que puedan ser usadas como

entrada de analizadores estáticos, herramientas de testing o verificadores de programas, o incluso como documentación o para ser estudiadas y analizadas de forma manual. La inferencia automática de especificaciones tiene como objetivo final reducir el esfuerzo necesario para escribir especificaciones de programas. Para ello computa especificaciones aproximadas sin necesidad de intervención del desarrollador. En esta tesis se mejora la inferencia automática de especificaciones para el lenguaje multiparadigma Curry y para programas orientados a objetos en general proponiendo dos nuevas aproximaciones. Por un lado, se presenta una técnica para inferir especificaciones algebraicas para programas Curry construyendo expresiones a partir de la signatura del programa y clasifiándolas en función de la semántica asociada a dichas expresiones. En contraste con las aproximaciones existentes en la literatura, esta técnica permite distinguir entre partes de la especificacion *correctas* y partes *possiblemente correctas*. Por otro lado, se presenta una técnica para inferir especificaciones de alto nivel presentadas en forma de pre/post-condiciones para lenguajes orientados a objetos. Esta técnica se formaliza en el contexto del marco de verificación de *Matching Logic* de forma que se habilita la posibilidad de verificar la especificación inferida.

Resum

Les tècniques d'anàlisi de programes tenen una gran quantitat d'aplicacions en el món actual. No obstant això, existixen encara aspectes a millorar dins de les mateixes que poden ajudar a difondre més el seu ús. Esta tesi està dedicada a la millora de dos aspectes de l'anàlisi de programes: la rigidesa de les seues tècniques i el seu complex procés d'aprenentatge.

La rigidesa de les tècniques d'anàlisi de programes és deguda a la gran complexitat dels algoritmes d'anàlisi, que provoquen que, sense un costós entrenament i aprenentatge previs, els desenvolupadors no puguen crear els seus pròpis anàlisis per a millorar la qualitat dels seus programes. L'anàlisi de programes declaratiu té com a objectiu reduir l'esforç en el disseny d'implementació d'anàlisis gràcies a l'augment del nivell d'abstracció del llenguatge d'especificació usat, sempre sense renunciar a oferir un mètode d'execució de l'anàlisi comparable en termes d'eficiència a les implementacions més tradicionals.

En esta tesi es millora en dos aspectes l'aproximació d'anàlisi de programes Java basada en el llenguatge lògic d'especificació Datalog. En primer lloc, es traduïxen especificacions Datalog a *Sistemes d'Equacions Booleanes* a fi de distribuir el còmput dels anàlisis millorant així els seus temps d'execució. Els Sistemes d'Equacions Booleanes estan equipats amb algoritmes eficients i distribuïts per a la seua avaluació i hi ha ferramentes industrials que els implementen. En segon lloc, es traduïxen especificacions Datalog a teories de Lògica de Reescriptura a fi de suportar l'extensió del llenguatge d'especificació Datalog de manera que puguen expressar-se anàlisi més sofisticats, per exemple anàlisi que tinguen en compte l'ús de reflexió en els programes."

Una altra contribució d'esta tesi està relacionada amb l'automatització de la inferència d'especificacions com a mecanisme de suport a les tècniques que milloren la qualitat dels programes (anàlisi de programes, verificació, depuració, documentació, etc.). Podem dir que les especificacions són la base de l'anàlisi de programes en particular (i de la resta de tècniques mencionades en general) ja que qualsevol anàlisi comprova si el comportament del programa que està sent analitzat es correspon amb el dau per una especificació. Sense un entrenament previ, els desenvolupadors poden no ser capaços de formular especificacions adequades que puguen ser usades com a entrada d'analitzadors estàtics, ferramentes de testing o verificadors de programes, o inclús com

a documentació o per a ser estudiades i analitzades de forma manual. La inferència automàtica d'especificacions té com a objectiu final reduir l'esforç necessari per a escriure especificacions de programes. Per a això computa especificacions aproximades sense necessitat d'intervenció del desenvolupador. En esta tesi es millora la inferència automàtica d'especificacions per al llenguatge multiparadigma Curry i per a programes orientats a objectes en general proposant dos noves aproximacions. D'una banda, es presenta una tècnica per a inferir especificacions algebraiques per a programes Curry construint expressions a partir de la signatura del programa i classificant-les en funció de la semàntica associada a les dites expressions. En contrast amb les aproximacions existents en la literatura, esta tècnica permet distingir entre parts de l'especificació *correctes* i parts *posiblement correctes*. D'altra banda, es presenta una tècnica per a inferir especificacions d'alt nivell presentades en forma de pre/post- condicions per a llenguatges orientats a objectes. Esta tècnica es formalitza en el context del marc de verificació de *Matching Logic* de manera que s'habilita la possibilitat de verificar l'especificació inferida.

Acknowledgments

I would like to thank my two advisors, María Alpuente and Alicia Villanueva, for giving me the opportunity to do my PhD. María is an example of hard work and dedication to her research group, the ELP, and I thank her for discovering me and having taken care of all of us. Alicia is an example of silent hard work, she is always looking after others, and I thank her for becoming the sister I never had. I also thank my advisors for having accepted me as I am and having made these years so pleasant.

I want to thank the current and past members of the ELP and MiST groups that I have had the pleasure to know for being so kind and making so easy to be willing to go to work everyday. In particular, I would like to thank Christophe Joubert for his help during the first years of my research career. I also want to thank Germán Vidal for being *himself* and being so funny at the same time; Santiago Escobar, for your humility, advice and sense of humor; Francisco Frechina, Fran, for your generosity and your friendship; Salvador Tamarit, Tama, for Utrecht and our endless bike rides; Sonia Santiago, for your kindness and conversation; Javier Espert, for your insightful discussions about anything; Javier and David Insa, for being my most reliable *merienda* companions and making me play *fútbol* again; Fernando Martínez, Nando, for your sympathy and your constancy; Julia Sapiña, for your company and for being the last to arrive and our heir; Josep Silva, for being so special and amusing; Salvador Lucas, Salva, for your work and passion for science, and for Pat Metheny; Marisa Llorens, for your company and care; Carlos Herrero, for your wall-trespassing guffaws; Alexei Lescaylle, for your smile and your affection; Daniel Omar Romero, Dani, for your jokes and happiness; Michele Baggi, for your goodness; José Iborra, Pepe, for your long rants about functional programming or whatever; César Tomás, for you humility and help; Beatriz Alarcón, for your support when I started this adventure; Javier Oliver, for inadvertently helping me realise that what I really loved doing was computer science; and Demis Ballis, María José Ramírez, César Ferri, José Orallo, Antonio González, Antonio Bella, Fernando Tarín, Mauricio Alba, Gustavo Arroyo, Arístides Dasso and Ana Funes for being so nice.

I am also grateful to the Departamento de Sistemas Informáticos y Computación (DSIC), my department, and to the Universitat Politècnica de València (UPV), my university, for providing me with a nice environment to fulfil my

thesis. In particular, I want to thank the people at the administration section of the DSIC for their help during these years: Susana Membrilla, Maite Valmaña, Toni Alfani, Amparo Ferrer, Rosa Martínez, Carolina Argente and specially Aída Gil who helped me finish the paperwork to present this thesis on time, even if I am always late. I also would like to thank the Centro de Transferencia de Tecnología of the UPV for helping me with the bureaucracy of my doctoral grant, specially Ana Belén Rubio, who was often of great help. I also want to thank the rest of members of my department for contributing to having a quiet and nice atmosphere. Specially I would like to thank some colleagues for their sympathy: Ignacio Mansanet (Nacho, *The Boss*), Ismael Torres (Isma), Míriam Gil and María Gómez (do you remember that laugh we had at the library trying to bind some books?), Clara Ayora, Mario Cervera, and Marcelino Campos. Finally, thanks to the people that keeps our workplace tidy, specially to Consuelo Casabán and Sergio Martínez (your emails about your lost trolley were so fun!).

I want to thank many people at the Università degli Studi di Udine where I made a research stay. Thank you Marco Comini for accepting me as a friend and share with me your knowledge and sympathy. Thanks to Valeria Patrizio, Marinella Mattu, Matteo Cicuttin, Giovanni Bacci, Giorgio Bacci, Luca Torella, Paolo Baiti, Francesca Nadalin, Marco Paviotti and Andrea Baruzzo for sharing your sympathy with me. Also in Italy, I would like to thank Alberto Pettorossi for his advice and affection. I also want to thank some people at the University of Illinois at Urbana-Champaign. Thanks to Grigore Roşu for accepting me for a research stay at your group and allowing me to learn by sharing the day to day work with all of you. Thanks to Chucky Ellison and Traian Florin Şerbănuţă for your sympathy. Thanks to Ralf Sasse, Nana Arizumi and Alexandre Duchateau (Lex) for being my friends at Urbana-Champaign. Special thanks to Raúl Gutiérrez for being like a brother during those months we lived together.

I would also like to thank my teachers at primary and secondary school, who taught me the pleasure of learning and knowing. Special thanks to Arturo Morales, my teacher of mathematics at school, for motivating me, for showing me the beauty of knowledge, and for introducing me to Dick Feynman. Also thanks to Luis Arana for becoming more than a teacher, a friend, and showing me the beauty of the Castilian (Spanish) language. I would also like to thank my teachers during the year I spent at Montreal when I was a child, Madame Louise Labonté-Costa, Madame Lucilla González and Monsieur Paquin for making me feel special.

However, I would not be finishing my doctoral studies without many people who have loved me and supported me for many years. I thank my strongest

condicio sine qua non, my mother, Marta Angélica Feliú, for doing everything that was necessary for her son, for me. I thank my uncle Nelson Feliú and aunt Ada Flores for actually being my second parents and loving me like their son. I thank my grandparents Lala and Tata for being an example of kindness and integrity that is impossible to erase from my mind even if I only shortly enjoyed their presence when I was a little child. I miss you. I thank Laura Titolo, my *rompi*, my *miñu*, for sharing her life with me and for so many years keeping on knowing and loving each other.

I like to think that who I am right now is also a consequence of all the good people I have met during my life. That is why I am also thankful...

to George (Jorge Castellanos) for always being there with your arms wide open for so many years and for still remembering my old self;

to Michelangelo (Miguel Ángel Ayuso) for being my oldest friend and an example of self-improvement;

to the Fail Club: The Rat, Pollo, Nekro, Perku, Zom, etc. for those wonderful times we spent together learning to enjoy life; specially thanks to The Rat and Nekro for sharing a dream even if it only lasted for a couple of months;

to Raúl Martínez, alias *Jimmy*, for making me constant on two things: music and our friendship;

to Héctor Galán, for being an inspiration and a good friend;

to Soncio (Luis Arana Jr) for always wanting to share your happiness with the others;

to the *Homeless Software Team* for being a part of you;

to Pepe Mena and Mari Carmen Espasa, for loving me and giving me my first PC;

to the García Soler family: Pepe, Mari, Laura and Isabel, for having been like a family to me for many years;

to Álex Eduardo Valdés and Simona Miceviciute, for making me company for a long time, and for keeping at it with their business;

to Lorella Daniel and Luigi Titolo, for your affection and those peaceful and happy stays in Italy;

to Luis Arana and Amparo Chicote, for your affection and generosity;

to Alicia and Raúl again, for being there and giving life to Alba;

to Leti (Leticia Pascual) and Ghada El Khamlichi, for your affection and support;

to María Garijo, a friend of my grandmother, for being there when my grandmother was gone, and to her sister Antonia Garijo for showing me Barcelona and the Camp Nou;

to *la abuelita* Alicia Aguirre for your love;

to Gloria Feliú, Katherine Laborde, Mirta and Fernando Alvarado, Teresa Feliú, Cindy Alvarado and the rest of my family around the world in Chile, Australia, Norway, Sweden and Canada, for loving and recalling me even in the distance;

to Peter, Valentín, Eugenia, María José, Alicia, Cayetana, Mireia and "los mafiosos", the O_2 salsa group, Salva, Xisco, Itziar, Boris, Inés, Xiomara, Melissa, Carla, Ana, David, Jesús, Jim, Laura, Ana Isabel, Noemí, Nuria, Vicky, Silvia, Frank, Antonio, Javier, Pepe, Juanito, Xordi, Antonio José, Jordi, Josep, Dani, Andreu, José Luis, Julio, José Vicente, Miguel, Nel·lo, Paquito, Quique, Mascarós, Ignacio, Chris, Isabel, Mar, Jenny, Abel, Rocío, Víctor, Carlos, Tais, María Jesús, Juanje, Fernando, Vicent, Vicent Andreu, Edwin, Nando, Daniel, Carla, Ilia, Gleb, Ahmad Wally, Ahmadullah, TJ, Mohsen, Ruosi... so many people and so many good moments;

to Florencia Pérez, Sor Flor, who recently passed away, for being so gentle and loving so much everyone around you;

to María Juliana Malerba for fighting for your life without losing your smile and tenderness.

I also would like to thank my country, Spain. Even If I am angry with it for so many things; it also has many other good points, like having given me the opportunity to study and do research.

Thank you all. Even if the useless stress of life does not allow me to think about you more often, I remember you and I wish all the best to you. And, by the way, sorry for my English ;-).

September 2013,

Marco A. Feliú

Contents

Introduction vii
I.1 Challenges . . . ix
I.1.1 Declarative and effective program analysis . . . ix
I.1.2 Automated Inference of Specifications . . . x
I.2 The Proposed Approach . . . xii
I.3 Contributions of the Thesis . . . xiv
I.4 Plan of the Thesis . . . xv

1 Preliminaries 1
1.1 Rewriting Logic . . . 1
1.2 Maude . . . 2

I Datalog-based Declarative Program Analysis 9

2 Datalog and Boolean Equations Systems 15
2.1 Datalog . . . 15
2.2 Datalog-based analysis . . . 18
2.3 Parameterised Boolean Equation Systems . . . 20

3 From Datalog to Boolean Equations Systems 23
3.1 From Datalog to BES. . . . 24
3.2 A complete Datalog to BES transformation . . . 25
3.2.1 Instantiation to parameterless BES . . . 28
3.2.2 Optimizations to the basic BES resolution technique . . . 32
3.2.3 Solution extraction . . . 36
3.3 The prototype DATALOG_SOLVE . . . 36
3.4 Experimental results . . . 39
3.4.1 Further Improvements . . . 40
3.5 Conclusions and Related Work . . . 43

4 From Datalog to *Rewriting Logic* 45
4.1 From Datalog to RWL. . . . 46
4.2 A complete Datalog to RWL transformation . . . 51

4.3 Dealing with JAVA reflection . . . 71
4.4 The prototype DATALAUDE . . . 78
4.5 Experimental results . . . 80
4.5.1 Comparison w.r.t. a previous rewriting-based implementation . . . 80
4.5.2 Comparison w.r.t. other Datalog solvers . . . 81
4.6 Conclusions and Related Work . . . 82

II Automated Inference of Specifications 83

5 CURRY, the K Framework and *Matching Logic* 87
5.1 CURRY . . . 87
5.2 The **K** Framework . . . 91
5.3 *Matching Logic* . . . 92

6 Inference of Specifications from CURRY Programs 95
6.1 Specifications in the functional-logic paradigm . . . 96
6.2 Formalization of equivalence notions . . . 99
6.3 Deriving Specifications from Programs . . . 104
6.3.1 Pragmatical considerations . . . 111
6.4 Case Studies . . . 112
6.4.1 ABSSPEC: The prototype . . . 115
6.5 Conclusions and Related Work . . . 117

7 Inference of Specifications from C Programs 119
7.1 Specification Discovery . . . 121
7.2 Extending the **ML** Symbolic Machine . . . 125
7.2.1 The MATCHC Extension . . . 128
7.2.2 The Pattern Extraction . . . 129
7.3 Inference process . . . 130
7.3.1 Refining the inference process . . . 134
7.4 A case study of specification inference . . . 136
7.5 Conclusions and Related Work . . . 142

Conclusion and Future Work 145

Bibliography 147

List of Figures

2.1 Datalog specification of a *flow-insensitive*, *context-insensitive* *points-to* analysis with a precomputed call-graph. 19

3.1 Simplified Datalog points-to analysis. 24
3.2 PBES representing the points-to analysis in Figure 3.1. 24
3.3 JAVA program analysis using the DATALOG_SOLVE tool. 37
3.4 DATALOG_SOLVE input file specifying Andersen's points-to analysis. 38
3.5 JAVA program analysis using the DATALOG_SOLVE 2.0 tool. . . 41
3.6 DACAPO analysis times (sec.) for various Datalog implementations. 42
3.7 DACAPO memory usage (MB.) for various Datalog implementations. 43

4.1 JAVA reflection example. 72
4.2 The structure of the reflective analysis. 77
4.3 JAVA program analysis using DATALAUDE. 79
4.4 Average resolution times of four Datalog solvers (logarithmic time). 81

6.1 A general view of the inference process. 105

7.1 KERNELC implementation of a set with linked lists. 123
7.2 Inferred specification for the `add` function. 124
7.3 Symbolic execution rule for accessing a value in the memory. . 128
7.4 Paths computed by SE($\texttt{add}(\texttt{s},\texttt{x})[\cdot]$). 137
7.5 Paths resulting from SE($\mathit{contains}(\texttt{s},\texttt{x})[\mathit{call\_pattern}(p_2)]$) . . . 140

List of Tables

3.1 Description of the JAVA projects used as benchmarks. 39
3.2 Time (in seconds) and peak memory usage (in megabytes) for the context-insensitive pointer analysis of each benchmark. . . . 40

4.1 Number of initial facts (`a/2` and `vP0/2`) and computed answers (`vP/2`), and resolution time (in seconds) for the three implementations. 81

6.1 Inference process of example programs 116

7.1 Experimental results 136

Introduction

"We sail within a vast sphere, ever drifting in uncertainty, driven from end to end."

Blaise Pascal

Today, software is everywhere. You know what I mean. Almost everything you can imagine may have attached a small device with a program in it that controls it, identifies it, or whatever. And if it does not have it, probably one day it will.

Letting aside dogs with remote control, software is everywhere and it affects us. Today you lose the metro because the automatic ticket vending machine does not work, tomorrow your cell phone is unexpectedly dying, the day after somebody steals all the money in your bank account by sending it somewhere in the Cayman Islands— well, maybe it was a caiman playing with a computer. Fortunately, reality is not so bad. Normally, the metro arrives late and you can enjoy yourself rushing to another ticket machine; your phone performs much better after you reset it; and, who does not want to help animals enjoying themselves?

Software rules many devices that have different levels of importance with respect to economy and human lives. The world would be a better place if all software worked perfectly with respect to their intent, but it does not [oCP13, Cen13]. Down to earth, we have to content ourselves with trying to assure the correct behavior of programs with a clear, high impact (in economic terms or human lives), which are normally called *critical systems*. This does not mean that programs in general are not somehow *checked* for correctness, but not with the exhaustiveness necessary to assure it. Today, the main impact of software errors is economical. For example, it is well known that software bugs cost the U.S. economy around 59.5 billion per year [Tas02]. However, there are other aspects in which the society is vulnerable to unpredictable software errors like ecology, medicine or quality of life, and this perspective should not be underestimated [ZC09].

In order to gain confidence on the *correct* behavior of programs we have to analyze them, and *program analysis* is the discipline of computer science that deals with that. In program analysis, there are, overall, two different approaches: dynamic analysis and static analysis. Dynamic analyses execute

programs to analyze their behavior, while static analyses do not. In any case, prior to analyze a program we should have an ex-ante idea of what a *correct* behavior is. We can express that idea in the form of specific values, formulas or models that constitute what we call a *specification*.

Dynamic analyses generally select, more or less appropriately, some inputs (or contexts) to execute the program while observing its outputs or its intermediate states (i.e., the execution trace). The dynamic analysis then compares its observations against the specification in order to inform of any discrepancy. It should be clear that typical programs cannot be exhaustively tested with all their possible inputs under every possible environment because the quantity of cases to consider is enormous with respect to the computing resources available nowadays. Therefore, even if dynamic analysis is of great help in finding software defects, it cannot guarantee their absence.

On the contrary, static analyses do not execute the code. Static analyses consist of compile-time techniques for predicting *safe* and *computable approximations* of the behavior of a program [NNH99]. The key word is *approximation*. Any interesting property of programs is undecidable and therefore we cannot hope to compute it accurately. That is why static analyses *compute approximations*. These approximations have to be *safe* (or correct) with respect to the semantics of the language. By *safe* we mean that the approximation should consider at least (but not necessarily only) all the possible behaviors of the program as implied by its semantics. With these safe approximations, static analysis can guarantee the absence of errors but it may also introduce *false positives*, i.e., errors that exist in the program approximation, but do not in the actual program. Either way, static analysis is the most prominent approach to ensure quality in *critical systems*.

Program analysis techniques in general have many uses. For instance, they can *automatically* catch bugs early in the software development process reducing the cost of corrections, which increases as development time passes. They can also aid code reuse by helping documenting code that has no specification or comments. *Static analysis*, in particular, can (truly) prove the absence of particular bugs, and it can even improve other quality assurance techniques like testing. Moreover, these direct benefits provide some additional indirect ones: the reduction of the number of bugs induces more productivity as it speeds up the development; reduces the cost of maintenance; and exchanges human time for computational power, which is a fair trade-off given the machines available nowadays.

Research has shown that *static analysis* can detect up to 87 percent of common coding errors [Jon09], and that 60 percent of software faults that were found in released software products could have been detected with static

analysis tools [PCJL95]. Moreover, it is estimated that more than a third of the cost of software errors (in the case of the U.S. economy, $22.2 billion) could be eliminated improving the infrastructure of development [Tas02] to one in which static analysis plays an important role [Jon09]. In summary, software quality has an important economic value and so does static analysis [Jon09].

I.1 Challenges in Declarative Static Analysis and Automated Synthesis of Specifications

Despite the benefits of using static analysis, software reliability has not substantially improved in the real world [oCP13, Cen13]. Many reasons have been given as an attempt to explain why the industry has not embraced static analysis: market pressures, inertia, legacy code, lack of practical tools, imprecision, etc. The market pressure can force developers to move to production an unfinished (or not tested enough) system because of marketing reasons. Inertia is not a meaningful reason *per se*, but the lack of economic perspective regarding software quality seems more than common; it is actually a constant. The presence of legacy code is wrongly used as an excuse because it should be a motivation for using static analysis methods, as they can give insight into the unknown code. The lack of practical tools should not be an excuse, either; there are not static analysis tools for every programming language or development environment, but there exist many tools to assist real world development for real world languages like Java or C such as Astrée [CCF+05], FindBugs [CHH+06], Coverity [BBC+10] or the Clang Static Analyzer [Sou13]. Imprecision is an inherent attribute of static analysis that has been used as an excuse for not using it; however, imprecision of static analysis is constantly reducing as research and computational power increase. There are two reasons for not embracing static analysis that are directly related with this thesis: the lack of fully customizable static analysis frameworks and the lack of trained developers.

Declarative specifications of program analyses and inference of specifications, which are the two main topics addressed in this thesis, can help improving customizability and under-training of programmers, as we shall see in the following.

I.1.1 Declarative and effective program analysis

Programs contain many application-specific errors. Until recently, program analysis tools did not allow users to specify their own specific properties to

check [BPS00]. In any case, the *ad-hoc* implementation of a *custom* program analysis is a complex task that mostly only experts in program analysis and programming languages can successfully perform. This makes very hard for program analysis to be widely customized and applied to application-specific problems.

Nonetheless, the logic language Datalog [Ull85] has been recently proposed as a mean to declaratively specify, customize and, thus, experiment with static analyses [WACL05, Ull89].

The key point for the practical application of the declarative specification of program analyses is their efficient execution with respect to other specifically tuned standard implementations of the analysis. Efficiency is important because it affects the precision of the analysis, which is crucial for ensuring its usefulness. Another aspect that affects precision in the case of declarative program analysis is the expressiveness of the specification language. For example, Datalog is not a *Turing complete* language so it cannot specify every possible static analysis that we may conceive in order to increase the precision.

In [WACL05], a translator from Datalog programs to *binary decision diagrams* (BDDs) is proposed for exploiting BDDs' redundancy summarizing capabilities. BDDs are a data structure used to represent, in a compressed way, boolean functions and, at a more abstract level, sets or relations. The use of BDDs in this context allows for exhaustive and efficient computation of the analysis by means of highly-optimized BDD-libraries. By exhaustive computation we mean that the process is not guided by any goal; thus, it potentially performs unnecessary computation. On the contrary, [BS09] extends Datalog to include indexing information that increases the efficiency of the analysis execution on a commercial Datalog engine developed by LogicBlox Inc. There are two tasks that none of these Datalog engines is suitable for: the distribution of computations (to make use of different machines for improving the performance) and the extension of the specification language (i.e., Datalog) for encoding (more) sophisticated analysis in a natural and uniform way. We consider these two aspects crucial. Distributed computation is how the industry is dealing with the technological limits that nature imposes to the speed of uniprocessor systems, whereas expressiveness is as important as the complexity of the program analysis to be specified is.

I.1.2 Automated Inference of Specifications

Specifications are intrinsic to program analysis. Every time we apply a certain static analysis we are checking if the (approximation of the semantics of the) program satisfies some property encoded in the analysis specification. For

example, if we develop a static analysis that checks for dereferences of null pointers, we are implicitly specifying that our program should not dereference a null pointer.

Given a catalog of generic static analyses, a software engineer does not need special training, except for the management of *false positives*. If this software engineer wanted to write a custom analysis for some property specific to his application, he should have been trained in order for the analysis to be "as safe as possible". However, maybe the application satisfies other properties he is not considering that are important for the development of that software. What if a *special* static analysis for a program could provide all the different properties that might be successfully verified on it? This is how static inference of specifications works. Inference of specifications provides the developer with properties he may not be aware of, and that can be extremely useful for verification and documentation of the code.

There are many approaches to the inference of specifications. Many aspects vary from one proposal to another: the kind of specifications that are computed (e.g., model-oriented vs. property-oriented specifications), the kind of programs considered, the correctness or completeness of the method, etc. [ABL02, CSH10, HRD07, GMM09, TCS06]. When we studied the related literature, we observed that two techniques were specially simple and intuitive. One of these techniques is the one implemented by QuickSpec [CSH10], which infers, by means of *random testing*, equations for functional programs as the ones used by the automatic testing tool QuickCheck [CH00]—which automatically generates test cases for the functional language Haskell. The other technique is used by Axiom Meister [TCS06], which obtains high-level specifications in the form of pre/post-conditions for imperative programs by means of *symbolic execution*. Unfortunately, neither QuickSpec nor Axiom Meister can ensure the correctness of the specifications they infer. QuickSpec can never guarantee correctness because it is based on *testing*, whereas Axiom Meister cannot either because of the fundamental *incompleteness* of plain *symbolic execution*. Moreover, they neither support the (automatic or manual) verification of the specifications they infer. We consider the ability of a technique to deal with the correction of inferred specifications crucial for its practical use, either by guaranteeing correction by construction or by subsequently allowing its verification.

I.2 The Proposed Approach

Certainly, the improved opportunities for program analysis derived from this dissertation can increase the impact of static analysis on the software development process. We have detected and addressed several shortages of static program analysis as we summarize in this section.

Regarding the deficiencies of the Datalog-based program analysis, the problem was approached by focusing on transforming Datalog into appropriate formalisms that may support more sophisticated and efficient analyses.

First, we defined a translation of Datalog programs into **Boolean Equation Systems (**BES**)** [And94a] for easing the distribution of Datalog query evaluation. BES are sets of equations defining boolean variables that can be resolved with linear-time complexity. In particular, parameterised Boolean Equation Systems (PBES) provide a more compact representation for BESs by extending them with typed parameters. PBESs have been successfully used to encode several hard verification problems [MT08, CPvW07, Mat98]. We encoded Datalog program analyses into PBESs which are then solved by expanding them into plain BESs. By encoding Datalog into BESs, we are able to use the verification framework CADP [GMLS07] to perform the translation and resolution of PBES in an on-the-fly and demand-driven way. In addition, using the PBES formalism potentially allows the distributed algorithm for BES resolution of [JM06] to be applied into our program analysis setting.

We defined a translation of Datalog to ***Rewriting Logic*** (RWL) [Mes92] to support the extension of the specification language Datalog for expressing more sophisticated analysis. RWL [Mes92] is a very general *logical* and *semantical framework* based on equational logic, in which several logics and languages can be formally described and mechanized. Due to the reflective nature of RWL, in the sense that it can perfectly represent itself, RWL naturally supports unmatched extensibility because it can execute an extension of itself within itself. Moreover, RWL is efficiently implemented in the high-level executable specification language Maude [CDE+07]. More specifically, we translated Datalog program analyses into Maude *Rewriting Logic* theories. We have taken advantage of many Maude features for the translation, for instance by using its efficient representations for the data, its memoization and reflection capabilities, and its algebraic properties that lead to more concise theories. We have also implemented an extension of Datalog program analysis by means of RWL's reflection.

With respect to the lack of assistance for ensuring the correctness of the inferred specifications, we developed and implemented a declarative framework that allows us to *discover* and to (some extent) *check* program specifications.

We have used abstract interpretation for creating a static analysis methodology for the *functional-logic* language Curry [Han06] that infers equational properties satisfied by the functions defined in the program. **Abstract Interpretation** is a formalism to develop program abstractions for *safe* static analysis [CC77, CC79]. Abstractions are defined with respect to a *concrete* program semantics such that they model the interesting aspects for the properties to be analyzed. To this aim, it defines *safe* mappings between *concrete* and *abstract* versions of the values and operators in the language of interest. In our case, we built up our technique by abstracting an existing *condensed* and *goal-independent* semantics for the first order fragment of Curry [BC10, BC11]. Our technique uses abstract versions of the program semantics to infer equations between terms built from the signature of the program. With this novel approach, our technique can guarantee, under ascertained conditions, the correctness of parts of the specification, whereas QuickSpec can only infer *possibly* correct approximations. In addition, the specifications inferred have also been extended in order to represent the functional-logic features that are intrinsic to Curry and not present in Haskell, like logical variables and nondeterminism. As a result, our approach makes available the inference of QuickCheck-like specifications to a wider class of programs, the multi-paradigm *functional-logic* programs.

Matching Logic (**ML**) is a logic for the verification of programs that brings together operational and axiomatic semantics [Rc12]. We have used *Matching Logic* as a formal framework for developing an inference method such that the inferred specifications can further be verified by means of **ML** verification systems. Similarly to Rwl, **ML** is a rule-based framework. However, **ML** extends the standard representation of data used in Rwl and other formalisms (i.e., the notion of *term*), and in this regard, **ML** can be considered an extension of Rwl. Despite its generality, only small (yet not simple) verification tasks have been reported to date using a prototype, called MatchC, that allows reasoning with a subset of the C language [Rc11]. We have formalized a technique for inferring specifications with *Matching Logic* to leverage the reasoning capabilities of MatchC. Our technique infers pre/post-condition specifications of C-like programs that use the *functions* defined in a C program to simplify the representation of program states by means of *observational abstraction*. As a result, we automatically obtain concise high-level specifications of C programs.

I.3 Contributions of the Thesis

This dissertation makes the following contributions in the two main areas mentioned above: program analysis and automated synthesis of specifications.

- **Program Analysis**
  - **A semantics-preserving transformation from Datalog to Boolean Equation Systems.**
    We have developed a transformation from Datalog programs to BES that respects the program semantics and is aimed for static analysis of imperative programs. We introduce a prototype called DATALOG_SOLVE that takes advantage of the algorithms available for BES resolutions in the verification framework CADP and shows good efficiency. These results have been published in [AFJV09d, AFJV08, AFJV09b, AFJV09a, AFJV11].
  - **A semantics-preserving transformation from Datalog to *Rewriting Logic***
    We have developed a transformation from Datalog to *Rewriting Logic* that is proven to be correct and achieves extensibility of declarative program analysis. We introduce a prototype called DATALAUDE that translates Datalog into *Rewriting Logic* theories aimed at reducing the size of the transformed program and that increases the speed of the resolution. We show that the efficiency of DATALAUDE is comparable with other *interpreted* Datalog solvers. Finally, by using *Rewriting Logic*'s reflective capabilities, we exemplify a particular extension of a Datalog program analysis by developing a *points-to analysis involving reflection*. These results have been originally published in [AFJV09c, AFJV10, AFJV11].
- **Synthesis of Specifications**
  - **A technique for inferring Equational Specifications from CURRY programs.**
    We present a technique for inferring Equational Specifications for CURRY programs. We show its feasibility and the quality of the inferred specifications by introducing the prototype ABSSPEC, which implements the technique and is based on an abstract interpretation of a precise and compact semantics for CURRY. These results have been published in [BCFV11, BCFV12a, BCFV12b].

– **A technique for inferring Pre/Post Specifications for C-like programs.**
We present a technique for inferring high level Pre/Post Specifications for C-like programs within the *Matching Logic* framework. We demonstrate the practicality of the proposed technique by introducing and evaluating the prototype KindSpec that extends the *Matching Logic* verifier MatchC for supporting full *symbolic execution (deduction)* of *Matching Logic* formulas, and infers specifications for programs written in a subset of the C language. These results have been published in [AFV13].

I.4 Plan of the Thesis

This dissertation is divided in two parts that are organized as follows. Part I deals with Datalog-based declarative static analysis and Part II is concerned with the automatic inference of formal specifications from imperative programs and multiparadigm programs. Chapter 1 provides some preliminaries that are needed for the two parts of the thesis. Part I begins in Chapter 2 where some background about Datalog, Bes and Datalog-based program analysis is provided. Chapter 3 presents the transformation from Datalog to Boolean Equation Systems and the prototype Datalog_Solve that implements this technique. Chapter 4 presents the transformation from Datalog to *Rewriting Logic*, the prototype Datalaude that implements this transformation, and a declarative extension of the analysis. Part II begins in Chapter 5 by providing some background on Curry, the **K** framework and *Matching Logic*. Chapter 6 presents our technique for inferring formal algebraic specifications from Curry programs by using abstract interpretation, and describes AbsSpec, the prototype that implements this methodology. Chapter 7 describes our method to infer axiomatic specifications for C-like programs developed in the framework of *Matching Logic*; it also presents KindSpec, the prototype that implements the proposed inference technique, concluding Part II. Finally, the last chapter is devoted to *conclusions* and summarizes the contributions, concluding this thesis.

1 Preliminaries

1.1 Rewriting Logic

Rewriting logic is a powerful logical framework that allows us to formally represent a wide range of systems [Mes92], including models of concurrency, distributed algorithms, network protocols, semantics of programming languages, and models of cell biology, just to mention a few. Rewriting logic is also an expressive universal logic, i.e., a flexible logical framework in which many different logics and inference systems can be represented and mechanized. The *Rewriting Logic* framework is efficiently implemented in the high-performance language Maude [CDE+07].

A rewrite theory is a tuple $R = (\Sigma, E, R)$, with:

- (Σ, E) an equational theory with function symbols Σ and equations E; and
- R a set of labeled rewrite rules of the general form

$$r : t \longrightarrow t'$$

with t, t' Σ-terms which may contain variables in a countable set X of variables.

Intuitively, R specifies a concurrent system whose states are elements of the initial algebra $T_{\Sigma/E}$ specified by (Σ, E), and whose concurrent transitions are specified by the rules R. The equations E may be decomposed as the union $E = E_0 \cup A$, where A is a (possibly empty) set of structural axioms (such as associativity, commutativity, and identity axioms).

Rewriting logic expresses an equivalence between logic and computation in a particularly simple way. Namely, system states are in bijective correspondence with formulas (modulo whatever structural axioms are satisfied by

these formulas: for example, modulo the associativity and commutativity of a certain operator) and concurrent computations in a system are in bijective correspondence with proofs (modulo appropriate notions of equivalence among computations and proofs).

Given this equivalence between computation and logic, a rewriting logic axiom of the form:

$$t \rightarrow t'$$

has two readings. Computationally, it means that a fragment of a system's state that is an instance of the pattern t can change to the corresponding instance of t' concurrently with any other state change; that is, the computational reading is that of a local concurrent transition. Logically, it just means that we can derive the expression t' from the expression t; that is, the logical reading is that of an inference rule.

Rewriting logic is entirely neutral about the structure and properties of the expressions/states t. They are entirely *user definable* as an algebraic data type satisfying certain equational axioms, so that rewriting deduction takes place modulo such axioms. Because of this neutrality, rewriting logic has good properties: as a logical framework, many other logics can be naturally represented in it; as a semantic framework, many different system styles, models of concurrent computation, and languages can be naturally expressed.

1.2 Maude

Maude[1] [CDE+07] is a very efficient implementation of *Rewriting Logic*, as we mentioned above. As it is presented in this section, Maude is a programming language that uses rewriting rules, similarly to the so-called functional languages like HASKELL, ML, SCHEME, or LISP. In the following, we briefly present some of the features of this language that have been used in our work.

A Maude program is made up of different *modules*. Each module can include:

- *sort* (or *type*) declarations;
- *variable* declarations;
- *operator* declarations;
- *rules* and/or *equations* describing the behavior of the system operators, i.e., the *functions*.

[1] http://maude.cs.uiuc.edu/

Maude mainly distinguishes two kinds of modules depending on the constructions they define and on their expected behavior. Functional modules do not contain rules and the behavior of their equations is expected to be confluent and terminating. On the contrary, system modules can contain both equations and rules and, though the behavior of their equations is also expected to be confluent and terminating, the behavior of its rules may be non-confluent and non-terminating. A functional module is limited by the reserved keywords `fmod` and `endfm`, whereas a system module is defined in between `mod` and `endm`.

Sorts. A sort declaration looks like

```
sort T .
```

where `T` is the identifier of the newly introduced sort `T`. Maude *identifiers* are sequences of ASCII characters without white spaces, nor the special characters '{','}','(',')','[', and ']' unless they are escaped with the back-quote character '`'. If we want to introduce many sorts `T1 T2 ... Tn` at the same time, we write:

```
sorts T1 T2 ... Tn .
```

After having declared the sorts, we can define operators on them.

Operators. Operators are declared as follows:

```
op C : T1 T2 ... Tn -> T .
```

where `T1 T2 ... Tn` are the sorts of the arguments for operator `C`, and `T` is the resulting sort for the operator. We can also declare at the same time many operators `C1 C2 ... Cn` with the same signature (i.e., sort of arguments and resulting sort) at the same time:

```
op C1 C2 ... Cn : T1 T2 ... Tm -> T .
```

Operators can represent two kinds of objects: *constructors* and *defined symbols*. Constructors constitute the *ground terms* or *data* associated to a sort, whereas defined symbols represent functions whose behavior will be specified by means of equations or rules. The rewriting engine of Maude does not distinguish between constructors or defined symbols, so there is no real syntactic difference between them. However, for documentation (and debugging) purposes, operators that are used as constructors can be labeled with the attribute `ctor`.

Operator attributes. Operator attributes are labels that can be associated to an operator in order to provide additional information (either syntactic or semantic) about the operator. All such attributes are declared within a single pair of enclosing square brackets "`[`" and "`]`":

```
op C1 C2 ... Cn : T1 T2 ... Tm -> T [A1 ... Ao] .
```

where the `Ai` are attribute identifiers. The set of operator attributes includes among others: `ctor`, `assoc`, `comm`, `id`, `ditto`, etc., that are described below.

Mix-fix notation. Another interesting feature of operators in Maude is *mix-fix* notation. Every operator defined as above is declared in *prefix* notation, that is, its arguments are separated by commas, and enclosed in parenthesis, following the operator symbol, as in:

```
C(t1, t2, ... , tn)
```

where `C` is an operator symbol, and `t1`, `t2`,... `tn` are, respectively, terms of sorts `T1`, `T2`,... `Tn`. Nevertheless, Maude provides a powerful and tunable syntax analyzer that allows us to declare operators composed of different identifiers separated by its arguments. Arguments can be set in any position, in any order, and even separated by white spaces. Mix-fix operators are identified by the sequence of its component identifiers, with characters '_' inserted in the place each argument is expected to be, as in:

```
op if_then_else_fi : Bool Exp Exp -> Exp .
op __ : Element List -> List .
```

The first line above defines an if-then-else operator, while the second one defines `List`s of juxtaposed (i.e., separated by white spaces) `Element`s. A term built with the `if_then_else_fi` operator looks like

```
if b1 then e1 else e2 fi
```

where the tokens `if`, `then`, `else` and `fi` represent the mixfix operator, `b1` represents a term of sort `Bool`, and finally `e1` and `e2` represent terms of sort `Exp`. A term built with the `__` operator looks like

```
e1 e2
```

where `e1` is a term of sort `Element`, `l1` is a term of sort `List`, and the space separating them represents the juxtaposition operator `__`.

Sort orders. Sorts can be organized into *hierarchies* with `subsort` declarations. In

```
subsort T1 < T2 .
```

we state that each element in `T1` is also in `T2`. For example, we can define natural numbers by considering their classification as positives or as the zero number in this way:

```
sorts Nat Zero NonZeroNat .
subsort Zero < Nat .
subsort NonZeroNat < Nat .
op 0 : -> Zero [ctor] .
op s : Nat -> NonZeroNat [ctor].
```

Maude also provides operator *overloading*. For example, if we add:

```
sort Binary .
op 0 : -> Binary [ctor] .
op 1 : -> Binary [ctor] .
```

to the previous declarations, the operator `0` is used to construct values both for the `Nat` and for the `Binary` sorts.

Structural axioms. The language allows the specification of structural axioms over operators, i.e., certain algebraic properties like *Associativity*, *Commutativity* and *Identity element* that operators may satisfy. In the following, we write ACI to refer to the three previous algebraic properties. Structural axioms serve to perform the computation on equivalence classes of expressions, instead of on simple expressions. In order to carry out computations on equivalence classes, Maude chooses a canonical representative of each class and uses it for the computation. Thanks to the structural information given as operator attributes, Maude can also choose specific data structures for an efficient low-level representation of expressions.

For example, let us define a list of natural numbers separated by colons:

```
sorts NatList EmptyNatList NonEmptyNatList .
subsort EmptyNatList < NatList .
subsort Nat < NonEmptyNatList .
subsort NzNat < NatList .
op nil : -> EmptyNatList [ctor] .
op _:_ : Nat NatList -> NonEmptyNatList [assoc] .
```

The operator "_:_" is declared as *associative* by means of its attribute `assoc`. Associativity means that the value of an expression is not dependent on the subexpression grouping considered, that is, the places where the parenthesis are inserted. Thus, if "_:_" is associative Maude considers the following expressions as equivalent:

```
s(0) : s(s(0)) : nil
(s(0) : s(s(0))) : nil
s(0) : (s(s(0)) : nil)
```

As another example, let us define an associative list with `nil` as its identity element:

```
sort NatList .
subsort Nat < NatList .
op nil : -> NatList [ctor] .
op _:_ : NatList NatList -> NatList [assoc id: nil] .
```

The operator "_:_" is declared as having `nil` as its *identity element* by means of its attribute `id: nil`. Having an identity element e means that the value of an expression is not dependent on the presence of e's as subexpressions, that is, it is possible to insert e's without changing the meaning of the expression. Thus, in our example Maude considers the following expressions (and an infinite number of similar ones) as equivalent:

```
s(0) : s(s(0))
nil : s(0) : s(s(0))
s(0) : nil : s(s(0))
s(0) : s(s(0)) : nil
nil : s(0) : nil : s(s(0)) : nil
⋮
```

For that reason, Maude omits `nil` in the canonical representative, unless it appears alone as an expression. Now, let us introduce how we define a multi-set, that is, an associative and commutative list with `nil` as its identity element:

```
sort NatMultiSet .
subsort Nat < NatMultiSet .
op nil : -> NatMultiSet [ctor] .
op _:_ : NatList NatList -> NatList [assoc comm id: nil] .
```

In this example, the operator "_:_" is declared to be *commutative* by means of the attribute `comm`. Commutativity means that the value of an expression

is not dependent on the order of its subexpressions, that is, it is possible to change the order of subexpressions without changing the meaning of the expression. Thus, if "_:_" is a commutative and associative operator, Maude considers the following expressions equivalent:

```
s(0) : s(s(0)) : s(s(0))
s(s(0)) : s(0) : s(s(0))
s(s(0)) : s(s(0)) : s(0)
```

The structural properties are efficiently built in Maude. Additional structural properties can be defined by means of equations, as we discuss below.

Rules and equations. In Maude, *rules* or *equations* characterize the behavior of the *defined symbols*. Both language constructions have a similar structure:

```
rl l => r .
eq l = r .
```

`l` and `r` are `terms`, i.e., expressions recursively built by nesting correctly typed operators and variables. `l` is called the left-hand side of a rule or equation, whereas `r` is its right-hand side. Variables can be declared when they are used in an expression by using the structure *name*: *sort*, or also in a general variable declaration:

```
var N1 N2 ... Nm : S .
```

where `N1`, `N2`, ..., and `Nm` are variable names, and `S` is a sort. Terms form patterns which may represent many *ground terms* (terms without variables). The *pattern* nature of terms allows them to be *matched* with other terms. The *pattern-matching* between a (pattern) term p and another term t consists in finding the *substitution* θ from variables in p to terms such that $p\theta = t$, i.e., substitution θ applied to p makes the instantiated term equal to t.

Definition 1 (Rewriting semantics) *Let P be a* Maude *program. Given a term t,* rewriting *t into a new term t' with respect to P, written $t \rightarrow_P t'$, consists in*

1. *finding a rule or equation of the form* `rl l => r .` *or* `eq l = r .` *whose left-hand side* `l` *matches some subterm t_{sub} of t with substitution θ (i.e., $\mathtt{l}\theta = t_{sub}$)*
2. *obtaining t' by replacing in t the subterm t_{sub} by $\mathtt{r}\theta$*

If t does not match the left-hand side of any rule or equation, then it is a canonical form.

In Maude, it can be specified that an equation should only be used for rewriting if none of the rest can. To do that, we label (with the same syntax of operators) the equation of interest with the reserved keyword `owise`.

I

Datalog-based Declarative Program Analysis

Data-flow Analysis

Data-flow analysis is a set of techniques that derive information about the *flow of data* during the execution of a program. The program execution can be viewed as a series of transformations of the program state, which consists of all the variables in the program, active stack frames, heap, etc. An *execution path* consists of the sequence of program states that results from executing a valid sequence of instructions of a program from a certain initial state.

In order to correctly analyze the behavior of a program, all possible execution paths must be considered. Then, we extract from all the program states forming an execution path the information needed for the particular analysis problem we want to solve. Unfortunately, in the general case, there is an infinite number of possible execution paths for a given program and the length of each path can be unbounded. This is why data-flow analyses summarize or abstract all the program states with different finite representations. However, in general, no possible abstraction made by an analysis describes perfectly the original state space.

Pointer analysis. The family of static analyses that is focused on answering the question "which elements can point to which elements?" is called *pointer analysis*. In object-oriented programming languages like JAVA, the "elements" that are able to point to something are *variables* and *fields* inside objects, whereas the "elements" that can be pointed to are *objects* located in the heap. In this case, pointer analysis approximates all possible flows of object references through variables and object fields.

Now, we introduce as a leading example a version of Andersen's *flow-insensitive* points-to analysis [And94b]. This analysis computes an approximation of all the possible objects a variable of the program can refer to during the execution of the program. For instance, given the following piece of code

```
q = new SomeClass(); /* o1 */
s = p;
p = q;
```

we can analyze its pointer information prior to its execution. Let us assume that all variables initially point to `null`, and that the execution runs naturally from the instruction at the top to the one at the bottom, i.e., the analysis is *flow-sensitive*. First, the initial instruction consists of creating the object `o1` by means of the expression `new SomeClass()` and assigning it to variable `q`; thus, after its execution, `q` points to `o1`. Then, the second instruction assigns the reference contained in `p` to the variable `s`; hence, since `p` points to `null`, `s`

points to `null` after this instruction is executed. Finally, after the execution of the last instruction, `p` points to whatever `q` points to, in this case `o1`.

The notion of a *flow-sensitive* analysis is inherently related with the notion of *program-point specific* versus *summary* analysis. An analysis is *program-point specific* if it computes information (in this case, points-to information) for each program point; in contrast, an analysis that computes a summary of information (in this case, all the objects that a variable can point to) that is valid for all program points is called a *summary* analysis. A *flow-sensitive* analysis like the previous one is straightforward and precise but, in general, computationally very expensive. On the contrary, *flow-insensitive* analysis are less precise but much more efficient.

In our example, we can conceive a *flow-insensitive* pointer analysis as one that does not take *control-flow* into account, i.e., one in which the instructions can be shuffled as in:

```
q = new SomeClass(); /* o1 */
p = q;
s = p;
```

where we have exchanged the position of the second and third instructions. For this new sequence of instructions, if we apply a *flow-sensitive* points-to analysis, we obtain that `p`, `q` and `s` point to `o1`. *Flow-insensitive* analyses perform the equivalent to a *flow-sensitive* analysis for every possible shuffle of instructions, but they do it efficiently. They extract the relevant information from every instruction ignoring its location, and they calculate with these data the points-to information. For instance, in the original example:

```
q = new SomeClass(); /* o1 */
s = p;
p = q;
```

a *flow-insensitive* analysis would compute that `p`, `q` and `s` could all point to `null` or `o1` during execution. The results of *flow-insensitive* analyses clearly overapproximate the *flow-sensitive* ones, and that is why they must be considered safe.

Recent proposals for data-flow analysis specify the analyses of interest by using declarative formalisms, such as the declarative language Datalog. This approach partially overcomes the problem of the high complexity of static analysis implementations. However, in order for the approach to be competitive, highly efficient solvers for the formalism must exist. Hence the optimization for Datalog solvers is a hot topic of research [AFJV11].

In this first part of the thesis, we investigate new techniques for evaluating Datalog programs in the context of declarative program analysis.

2

Datalog and Boolean Equations Systems

This chapter introduces the background knowledge that is necessary to understand the work presented in this Part I of the thesis. We present concepts related to the Datalog logic language and *Boolean Equation Systems* (BESs), as well as how to encode static analyses as Datalog programs.

2.1 Datalog

Datalog [Ull85] is a relational language that uses declarative *clauses* to both describe and query a deductive database. It is a language that uses a PROLOG-like notation, but whose semantics is far simpler than that of PROLOG.

Predicates, atoms and literals. Let $\mathcal{P}$ be a set of *predicate* symbols, $\mathcal{V}$ be a finite set of *variable* symbols, and $\mathcal{C}$ a set of *constant* symbols. The basic elements of Datalog are *atoms* of the form $p(a_1, a_2, \ldots, a_n)$ where $p \in \mathcal{P}$ and $a_i \in \mathcal{V} \cup \mathcal{C}$. Predicate symbols represent assertions concerning the arguments given between parenthesis at its right. The elements of $\mathcal{V} \cup \mathcal{C}$ (i.e., variables or constants) are called terms $a_1, a_2, \ldots, a_n$.

A *ground atom* is an atom with only constants as arguments. Every ground atom asserts a particular fact, and its value is either true or false. A *predicate* is a *relation* that can be represented, for example, as a table of its true ground atoms. Each ground atom is represented by a single row, or tuple, of the relation. The columns of the relation are its attributes, and each tuple has a value for each attribute. The attributes correspond to the argument positions of the predicate represented by the considered relation. Any ground atom in the relation is true and we call it a *fact*, whereas ground atoms not in

the relation are false. From now on, we use the terminology 'relation p' and 'predicate p' interchangeably.

A *literal* is either an atom or a negated atom. We indicate negation with the word NOT in front of the atom. Thus, NOT $p(a_1, a_2, \ldots, a_n)$ is an assertion stating that $p(a_1, a_2, \ldots, a_n)$ is false, i.e., $a_1, a_2, \ldots, a_n$ is not a row of the relation p.

Rules. Rules (also called clauses) are a way of expressing logical inferences, and suggest how the computation of the true facts should be carried out. Let $\mathcal{P}$ be a set of *predicate* symbols, $\mathcal{V}$ be a finite set of *variable* symbols, and $\mathcal{C}$ a set of *constant* symbols. A Datalog *rule* r defined over a finite alphabet $P \subseteq \mathcal{P}$ and arguments from $V \cup C$, $V \subseteq \mathcal{V}$, $C \subseteq \mathcal{C}$, has the following syntax:

$$p_0(a_{0,1}, \ldots, a_{0,n_0}) \text{ :- } p_1(a_{1,1}, \ldots, a_{1,n_1}), \ldots, p_m(a_{m,1}, \ldots, a_{m,n_m}).$$

where each p_i is a predicate symbol of arity n_i with arguments $a_{i,j} \in V \cup C$ ($j \in [1..n_i]$).

We call $p_0(a_{0,1}, \ldots, a_{0,n_0})$ the *head* of a clause, and $p_1(a_{1,1}, \ldots, a_{1,n_1}), \ldots, p_m(a_{m,1}, \ldots, a_{m,n_m})$ the *body* of the clause. Each of the elements in the body is called a *subgoal* or *hypothesis* of the rule. We should read the :- symbol as "if"; the "," operators separating each subgoal of the body is a logical *and* operator. Thus, the meaning of a rule is "the head is true if the body is true". More precisely, a rule is *applied* to a given set of ground atoms as follows. Consider all possible substitutions of constants for the variables of the rule. If a certain substitution makes every subgoal of the body true, i.e., each subgoal is in the given set of ground atoms, then we can infer that the instantiated head is a true fact.

Programs. A *Datalog program* is a collection of rules together with the starting "data", in the form of an initial set of facts for some of the predicates. The semantics of the program is the set of ground atoms inferred by using the facts and applying the rules until no more inferences can be made. The initial set of facts of a Datalog program is called the *extensional database*, whereas the set of facts inferred by means of clauses with non-empty bodies is called the *intensional database*. In this way, a predicate defined in the extensional or intensional databases is called *extensional* or *intensional* predicate, respectively.

Fixpoint semantics of programs. Now, let us formalize the semantics of Datalog in an appropriate way for our purposes. The *Herbrand Universe* of a

Datalog program R defined over P, V and C, denoted U_R, is the finite set of all ground arguments, i.e., constants of C. The *Herbrand Base* of R, denoted B_R, is the finite set of all ground atoms that can be built by assigning elements of U_R to the predicate symbols in P. A *Herbrand Interpretation* of R, denoted I (from a set $\mathcal{I}$ of Herbrand interpretations, $\mathcal{I} \subseteq B_R$), is a set of ground atoms.

Definition 2 (Fixpoint semantics) *Let R be a Datalog program. The least* Herbrand model *of R is a Herbrand interpretation I of R defined as the least fixpoint of a monotonic, continuous operator $T_R : \mathcal{I} \to \mathcal{I}$ known as the* immediate consequences operator *and defined by:*

$$T_R(I) = \{h \in B_R \mid h :\!- b_1, ..., b_m \text{ is a ground instance of a rule in } R, \text{ with } b_i \in I, i = 1..m, m \geq 0\}$$

The number of Herbrand models being finite for a Datalog program R, there always exists a least fixpoint for T_R, denoted μT_R, which is the least Herbrand model of R. In practice, one is generally interested in the computation of some specific atoms, called *queries*, and not in the whole database of atoms. Hence, queries may be used to prevent the computation of facts that are irrelevant for the atoms of interest, i.e., facts that are not derived from the query.

There are many approaches for the evaluation of Datalog queries. The two basic ones are the *top-down* and the *bottom-up* strategies. The *top-down* approach solves queries by reasoning backwards, whereas the *bottom-up* approach blindly infers all the program facts and then checks if the query has been previously inferred. In this thesis, we use the top-down approach.

Queries. In a *demand-driven* context, that is, under the assumption that we are not interested in everything that can be inferred from a Datalog program, queries allow us to restrict the information to be computed. This restriction improves the execution in terms of (execution) time and (memory) space.

A *Datalog goal* has this form:

$$:\!- \ p_1(a_{1,1}, \ldots, a_{1,n_1}), \ \ldots \ , \ p_m(a_{m,1}, \ldots, a_{m,n_m}). \tag{2.1}$$

The structure of a goal is analogous to the one of a rule body. We can read a query as a question "there exists a substitution of the variables used in the goal that makes true all the subgoals?".

Definition 3 (Query Evaluation) *A* Datalog query *q is defined as a pair $\langle G, R \rangle$ such that*

- *R is a Datalog program defined over P, V and C,*
- *G is a set of* goals.

Given a query q, its evaluation *consists in computing μT_Q, Q being the extension of the Datalog program R with the Datalog rules in G.*

The evaluation of a Datalog program augmented with a set of goals deduces all the different constant combinations that, when assigned to the variables in the goals, can make one of the goal clauses true, i.e., all atoms b_i in its body are satisfied.

2.2 Datalog-based analysis

In [WACL05, Ull89], the authors propose Datalog as a *more natural* language for specifying program analysis. In this approach, a program analysis is encoded as the evaluation of a query over a Datalog program. On one hand, the *extensional database* represents the information extracted from the program that is relevant for the analysis, and it can be extracted by means of a compiler. On the other hand, the *intensional database* encodes the analysis *logic*, i.e., the algorithm that computes the analysis. Finally, the *queries* represent a part of the analysis results that we are interested in, supporting the computation of only the necessary parts to satisfy them, which is the idea behind *demand-driven* techniques.

In order to adapt a program analysis to this setting, each program statement is decomposed into *basic operations* (e.g., assigning one variable to another, storing some value in an object, ...) performed over basic program elements which are grouped in their respective *domains* (e.g., variables, types, code locations, ...). Each kind of basic operation is described by a predicate relating program elements conforming the *extensional database*. The information of interest resulting from the execution of the analysis is also represented by predicates relating program elements. However, these predicates are defined by clauses, i.e., they constitute the *intensional database*, and its computation represents the execution of the analysis. By considering only finite program elements domains, and applying standard loop-checking techniques, Datalog program execution is ensured to terminate.

Next, we present an example of Datalog-based program analysis.

Example 4

Figure 2.1 presents a version of Andersen's [And94b] points-to analysis borrowed from [WACL05]. Technically, it is a Datalog-based flow-insensitive,

```
public A foo {
  ...
  p = new SomeClass(); /* o1 */
  q = new SomeClass(); /* o2 */
  r = q;
  w = r;
  w = q.f;
  q.f = p;
  ...
  }
```

```
vP0(p,o1).
vP0(q,o2).
assign(r,q).
assign(w,r).
load(q,f,w).
store(q,f,p).
```

```
vP(V1,H1)  :- vP0(V1,H1).
vP(V1,H1)  :- assign(V1,V2),  vP(V2,H1).
hP(H1,F,H2):- store(V1,F,V2), vP(V1,H1), vP(V2,H2).
vP(V1,H1)  :- load(V2,F,V1),  vP(V2,H2), hP(H2,F,H1).
```

Figure 2.1: Datalog specification of a *flow-insensitive*, *context-insensitive points-to* analysis with a precomputed call-graph.

context-insensitive pointer analysis that uses a precomputed call-graph. The reason for not considering calling contexts at method invocation sites and using a precomputed call-graph falls outside the scope of this thesis but it allows a very simple analysis specification.

The upper left side of the figure shows a simple JAVA program where `o1` and `o2` are the addresses of the heap allocation instructions (extracted by a JAVA compiler from the corresponding bytecode). The upper right side contains the facts that can be extracted from each line of code, which represent the relevant information needed for this analysis. These are the predicates that form the *extensional database* for this particular analysis:

`vP0(V,H)`: A new object `H` is created and is assigned to the variable `V`. Here, `H` is an *allocation site*, i.e., the position in the code where a constructor for some class of objects is called; the allocation site is used as a *heap abstraction* that represents all the possible objects allocated at that position.

`assign(V1,V2)`: The contents of the variable `V2` are assigned to the variable `V1` .

`load(V1,F,V2)`: The value of the field `F` of an object referenced by variable

`V1` is assigned to variable `V2`.

`store(V1,F,V2)`: The value of variable `V2` is assigned to the field `F` of an object referenced by variable `V1`.

Using these extracted facts, the analysis deduces further pointer-related information, like points-to relations from local variables to heap objects (e.g., `vP(V1,H1)` in Figure 2.1) as well as points-to relations between heap objects through field identifiers (e.g., `hP(H0,F,H2)` in Figure 2.1).

A Datalog query triggers the computation of the analysis. For instance, the query `:- vP(X,Y)` computes the complete set of program variables (instantiating `X`) that may point to any heap object (instantiating `Y`) during program execution. In the example above, the query computes the following answers: {`X/p,Y/o1`}, {`X/q,Y/o2`}, {`X/r,Y/o2`}, {`X/w,Y/o1`} and {`X/w,Y/o2`}. For example, the answer {`X/p,Y/o1`} for the previous query states that `vP(p,o1)` is `true`; for this pointer analysis, `vP(p,o1)` means that, during the execution of the program, variable `p` can contain a reference to an object allocated at `o1`. This answer was obtained by applying the first rule in the program to the first fact.

The Datalog-based approach to program analysis partially overcomes the problem of the high complexity of static analysis implementations. However, highly efficient solvers for the formalism must exist in order for the approach to be competitive, hence the interest in new optimizations for Datalog solvers.

2.3 Parameterised Boolean Equation Systems

Parameterised Boolean Equation Systems (PBESs) are a low-level formalism that has been largely studied in the context of formal verification. There exist very efficient tools for solving PBES in an industrial setting [GMLS07]. In the following, we present the basic notions for working with PBES.

Definition 5 (Parameterised Boolean Equation System [Mat98]) *Let $\mathcal{X}$ be a set of boolean variables and $\mathcal{D}$ a set of data terms, a* Parameterised Boolean Equation System *$B = (x_0, M_1, ..., M_n)$ is a set of n blocks M_i, each one containing $p_i \in \mathbb{N}$ fixpoint equations of the form*

$$x_{i,j}(\vec{d}_{i,j} : \vec{D}_{i,j}) \overset{\sigma_i}{=} \phi_{i,j}$$

with $j \in [1..p_i]$ and $\sigma_i \in \{\mu, \nu\}$, also called sign *of equation i, the least (μ) or greatest (ν) fixpoint operator. Each $x_{i,j}$ is a boolean variable from $\mathcal{X}$ that*

binds zero or more data terms $d_{i,j}$ of type $D_{i,j}$[1] *which may occur in the* boolean formula $\phi_{i,j}$ *(from a set Φ of boolean formulae). $x_0 \in \mathcal{X}$, defined in block M_1, is a boolean variable whose value is of interest in the context of the local resolution methodology.*

Boolean formulae $\phi_{i,j}$ are formally defined as follows.

Definition 6 (Boolean Formula [Mat98]) *A* boolean formula ϕ*, defined over an alphabet of (parameterised) boolean variables $X \subseteq \mathcal{X}$ and data terms $D \subseteq \mathcal{D}$, has the following syntax given in positive form:*

$$\phi ::= \mathsf{true} \mid \mathsf{false} \mid \phi \wedge \phi \mid \phi \vee \phi \mid X(e) \mid \forall d \in D.\ \phi \mid \exists d \in D.\ \phi$$

where boolean constants and operators have their usual definition, e is a data term (constant or variable of type D), $X(e)$ denotes the call *of a boolean variable X with parameter e, and d is a term of type D.*

A *boolean environment* $\delta \in \Delta$ is a partial function $\mathcal{X} \to (D \to \mathbb{B})$ mapping each (parameterised) boolean variable $x(d : D)$ to a predicate , with $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$. *Boolean constants* true and false abbreviate the empty conjunction $\wedge\varnothing$ and the empty disjunction $\vee\varnothing$ respectively. A *data environment* $\varepsilon \in \mathcal{E}$ is a partial function $D \to D$ mapping each data term e of type D to a value , which forms the so-called support of ε, noted $supp(\varepsilon)$. Note that $\varepsilon(e) = e$ when e is a constant data term. The *overriding* of ε_1 by ε_2 is defined as $(\varepsilon_1 \oslash \varepsilon_2)(x) = \mathit{if}\ x\ \in supp(\varepsilon_2)\ \mathit{then}\ \varepsilon_2(x)\ \mathit{else}\ \varepsilon_1(x)$. The *interpretation function* $[\![\phi]\!]\delta\varepsilon$, where $[\![.]\!] : \Phi \to \Delta \to \mathcal{E} \to \mathbb{B}$, gives the truth value of boolean formula ϕ in the context of δ and ε, where all free boolean variables x are evaluated by $\delta(x)$, and all free data terms d are evaluated by $\mathcal{E}(d)$.

Definition 7 (Semantics of Boolean Formula [Mat98]) *Let ε be a data environment and δ be a boolean environment. The semantics of a boolean formula ϕ is inductively defined by the following interpretation function:*

$$\begin{aligned}
[\![\mathsf{true}]\!]\delta\varepsilon &= \mathsf{true} \\
[\![\mathsf{false}]\!]\delta\varepsilon &= \mathsf{false} \\
[\![\phi_1 \wedge \phi_2]\!]\delta\varepsilon &= [\![\phi_1]\!]\delta\varepsilon \wedge [\![\phi_2]\!]\delta\varepsilon \\
[\![\phi_1 \vee \phi_2]\!]\delta\varepsilon &= [\![\phi_1]\!]\delta\varepsilon \vee [\![\phi_2]\!]\delta\varepsilon \\
[\![x(e)]\!]\delta\varepsilon &= (\delta(x))(\varepsilon(e)) \\
[\![\forall d \in D.\ \phi]\!]\delta\varepsilon &= \forall\, v \in D,\ [\![\phi]\!]\delta(\varepsilon \oslash [v/d]) \\
[\![\exists d \in D.\ \phi]\!]\delta\varepsilon &= \exists\, v \in D,\ [\![\phi]\!]\delta(\varepsilon \oslash [v/d])
\end{aligned}$$

[1] To simplify our description, in the rest of this chapter we intentionally restrict to one the maximum number of data term parameters, i.e., we simply assume $d : D$.

Definition 8 (Semantics of Equation Block [Mat98]) *Given a* PBES *of the form* $B = (x_0, M_1, ..., M_n)$ *and a boolean environment* δ*, the solution* $[\![M_i]\!]\delta$ *to a block* $M_i = \{x_{i,j}(d_{i,j} : D_{i,j}) \stackrel{\sigma_i}{=} \phi_{i,j}\}_{j \in [1,p_i]}$ *(*$i \in [1..n]$*) is defined as follows:*

$$[\![\{x_{i,j}(d_{i,j} : D_{i,j}) \stackrel{\sigma_i}{=} \phi_{i,j}\}_{j \in [1,p_i]}]\!]\delta = \sigma_i \Psi_{i\delta}$$

where $\Psi_{i\delta} : (D_{i,1} \to \mathbb{B}) \times \ldots \times (D_{i,p_i} \to \mathbb{B}) \to (D_{i,1} \to \mathbb{B}) \times \ldots \times (D_{i,p_i} \to \mathbb{B})$ *is a vectorial functional defined as*

$$\Psi_{i\delta}(g_1, ..., g_{p_i}) = (\lambda v_{i,j} : D_{i,j}.[\![\phi_{i,j}]\!](\delta \oslash [g_1/x_{i,1}, ..., g_{p_i}/x_{i,p_i}])[v_{i,j}/d_{i,j}])_{j \in [1,p_i]}$$

where $g_i : D_i \to \mathbb{B}$*,* $i \in [1..p_i]$*.*

A PBES is *alternation-free* if there are no mutual recursion between boolean variables defined by least ($\sigma_i = \mu$) and greatest ($\sigma_i = \nu$) fixpoint boolean equations. In this case, equation blocks can be sorted topologically such that the resolution of a block M_i only depends upon variables defined in a block M_k with $i < k$. A block M_i is *closed* when the resolution of all its boolean formulae $\phi_{i,j}$ only depends upon boolean variables $x_{i,k}$ from M_i.

Definition 9 (Semantics of alternation-free PBES [Mat98]) *Given an alternation-free* PBES $B = (x_0, M_1, ..., M_n)$ *and a boolean environment* δ*, the semantics* $[\![B]\!]\delta$ *of* B *is the value of its main variable* x_0 *given by the semantics of* M_1*, i.e.,* $\delta_1(x_0)$*, where the contexts* δ_i *are calculated as follows:*

$$\begin{array}{rcl} \delta_n & = & [\![M_n]\!][] \text{ (the context is empty because } M_n \text{ is closed)} \\ \delta_i & = & ([\![M_i]\!]\delta_{i+1}) \oslash \delta_{i+1} \text{ for } i \in [1, n-1] \end{array}$$

where each block M_i *is interpreted in the context of all blocks.*

3

From Datalog to Boolean Equations Systems

This chapter summarizes how Datalog queries can be solved by means of Boolean Equation System (BES) resolution [And94a]. The key idea of this approach is to translate the Datalog specification that represents a specific analysis into an implicit BES, whose resolution corresponds to the execution of the analysis [AFJV09d]. This technique has been implemented in the Datalog solver DATALOG_SOLVE[1] [AFJV09a] that is based on the well-established verification toolbox CADP [GMLS07], which provides a generic library for local BES resolution.

A Boolean Equation System is a set of equations defining boolean variables that can be solved with linear-time complexity. Parameterised Boolean Equation System [Mat98] (PBES) are defined as BES with typed parameters. Since PBES are a more compact representation than BESs for a system, we first present an elegant and natural intermediate representation of a Datalog program as a PBES. Then, we establish a precise correspondence between Datalog query evaluation and PBES resolution, which is formalized as a linear-time transformation from Datalog to PBES, and vice-versa.

In the rest of the chapter, we first informally illustrate in Section 3.1 how a PBES can be obtained from a Datalog program in an automatic way. Section 3.2 describes the transformation in depth. Section 3.3 presents the prototype developed to evaluate Datalog queries by means of our transformation. Section 3.4 evaluates the efficiency of our prototype by performing some analyses on real programs. Finally, Section 3.5 discusses related work and concludes.

[1] http://www.dsic.upv.es/users/elp/datalog_solve

3.1 From Datalog to BES.

In Figure 3.1 we introduce a simplified version of the Andersen points-to analysis, previously given in Figure 2.1, that contains four facts and the first two clauses that define the predicate `vP`. Given the query `:- vP(V,o2).` and the

```
vP0(p,o1).
vP0(q,o2).
assign(r,q).
assign(w,r).
vP(V,H) :- vP0(V,H).
vP(V,H) :- assign(V,V2), vP(V2,H).
```

Figure 3.1: Simplified Datalog points-to analysis.

Datalog program shown in Figure 3.1, our transformation constructs the PBES shown in Figure 3.2, in which the boolean variable x_0 and three parameterised boolean variables ($x_{\texttt{vP0}}$, $x_{\texttt{assign}}$ and $x_{\texttt{vP}}$) are defined. Parameters of these boolean variables are defined on a specific domain and may be either variables or constants. The domains in the example are the heap domain ($D_{\texttt{H}} = \{\texttt{o1}, \texttt{o2}\}$) and the source program variable domain ($D_{\texttt{V}} = \{\texttt{p}, \texttt{q}, \texttt{r}, \texttt{w}\}$). PBES are evaluated by a least fixpoint computation (μ) that sets the variable x_0 to `true` if there exists a value for V that makes the parameterised variable $x_{\texttt{vP}}(V, \texttt{o2})$ `true`. Logical connectives are interpreted as usual.

$$
\begin{array}{lcl}
x_0 & \overset{\mu}{=} & \exists V \in D_{\texttt{V}} \,.\, x_{\texttt{vP}}(V, \texttt{o2}) \\
x_{\texttt{vP0}}(\texttt{p}, \texttt{o1}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\texttt{vP0}}(\texttt{q}, \texttt{o2}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\texttt{assign}}(\texttt{r}, \texttt{q}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\texttt{assign}}(\texttt{w}, \texttt{r}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\texttt{vP}}(V : D_{\texttt{V}}, H : D_{\texttt{H}}) & \overset{\mu}{=} & x_{\texttt{vP0}}(V, H) \vee \\
 & & \exists V_2 \in D_{\texttt{V}} . (x_{\texttt{assign}}(V, V_2) \wedge x_{\texttt{vP}}(V_2, H))
\end{array}
$$

Figure 3.2: PBES representing the points-to analysis in Figure 3.1.

Intuitively, the Datalog query is transformed into the *relevant* variable x_0, i.e., the variable that guides the PBES resolution. Each Datalog fact is transformed into an *instantiated* parameterised boolean variable (no variables appear in the parameters), whereas each predicate symbol defined by Datalog

clauses (different from facts) is transformed into a parameterised boolean variable (in the example $x_{\mathtt{vP}}(V\colon D_{\mathtt{V}}, H\colon D_{\mathtt{H}})$). This parameterised boolean variable is defined as the disjunction of the boolean variables that represent the bodies of the corresponding Datalog clauses. Variables that do not appear in the parameters of the boolean variables are existentially quantified on the specific domain. In the example, $\exists V \in D_{\mathtt{V}}$ quantifies the free variable V occurring in the equation defining x_0, and $\exists V_2 \in D_{\mathtt{V}}$ quantifies the variable V_2 occurring in the equation defining $x_{\mathtt{vP}}(V : D_{\mathtt{V}}, H : D_{\mathtt{H}})$.

Among the different known techniques for solving a PBES (see [vDPW08] and the references therein), we consider the resolution method based on transforming the PBES into an alternation-free parameterless boolean equation system (BES) that can be solved by linear time and memory algorithms when data domains are finite [Mat98]. Actually, we do not explicitly construct neither the PBES nor the BES. Instead, an implicit representation of the transformed parameterless BES is defined. This implicit representation is then used by the CADP toolbox to generate the explicit parameterless BES on-the-fly. Intuitively, the construction of the BES can be seen as the resolution of the analysis. Nonetheless, the direct PBES to BES transformation is not efficient enough for our purposes. Hence, in Section 3.2.2, we also present an optimized variation of the transformation to improve the efficiency of the BES resolution.

3.2 A complete Datalog to BES transformation

We propose a transformation of the Datalog query into a related logical query, naturally expressed as a parameterised boolean variable of interest and a PBES, which is subsequently evaluated using traditional PBES evaluation techniques. To simplify our description, in the rest of this chapter we intentionally restrict to one both the arity of predicate symbols and the maximum number of data term parameters that a boolean variable $x \in \mathcal{X}$ can have.

Proposition 10 *Let $q = \langle G, R\rangle$ be a Datalog query, defined over alphabet P, variables set V and constants set C, and let $B_q = (x_0, M_1)$, with $\sigma_1 = \mu$, be a* PBES *defined over a set $\mathcal{X}$ of boolean variables x_p (in one-to-one correspondence with predicate symbols p of P plus a special variable x_0), a set $\mathcal{D}$ of data terms (in one-to-one correspondence with variable and constant symbols of $V \cup C$), and M_1 the block containing exactly the following equations, where fresh variables are existentially quantified after the transformation:*

$$x_0 \overset{\mu}{=} \bigvee_{:-\ q_1(d_1),\ \ldots,\ q_m(d_m).\ \in G} \bigwedge_{i:=1}^{m} x_{q_i}(d_i) \tag{3.1}$$

$$\{x_p(d:D) \stackrel{\mu}{=} \bigvee_{p(d) \text{ :- } p_1(d_1),\ldots\ p_m(d_m).\ \in R} \bigwedge_{i:=1}^{m} x_{p_i}(d_i) \mid p \in P\} \qquad (3.2)$$

Then, q is satisfiable if and only if $[\![B_q]\!]\delta(x_0) = \mathsf{true}$.

The result follows immediately by construction since satisfaction of the predicates is preserved in the transformation: each predicate is transformed into a disjunction of the conjunction of the boolean variables that represent the literals in the body of each clause.

Roughly speaking, the boolean variable x_0 encodes the set of Datalog goals G, whereas the (parameterized) boolean variables $x_p(d:D)$ represent the set of Datalog rules R modulo renaming.

In our framework, the reverse direction of reducibility consists in the transformation of a parameterised boolean variable of interest, defined in a Pbes, into a related relation of interest expressed as a Datalog query, which could be evaluated using traditional Datalog evaluation techniques.

Proposition 11 *Let $B = (x_0, M_1)$, with $\sigma_1 = \mu$, be a* Pbes *defined over a set $\mathcal{X}$ of boolean variables and a set $\mathcal{D}$ of data terms, and $q_B = \langle G, R\rangle$ be a Datalog query defined over a set P of predicate symbols p (in one-to-one correspondence with boolean variables x_p of $\mathcal{X} \setminus \{x_0\}$), a set $V \cup C$ of variable and constant symbols (in one-to-one correspondence with data terms of D), and $\langle G, R\rangle$ containing exactly the following Datalog rules:*

$$G = \left\{ \begin{array}{c} \text{:-}\ \ q_{1,1}(d_{1,1}),\ \ldots\ , q_{1,n_1}(d_{1,n_1}).,\\ \vdots \\ \text{:-}\ q_{m_0,1}(d_{m_0,1}),\ \ldots\ , q_{m_0,n_{m_0}}(d_{m_0,n_{m_0}}). \end{array} \middle| \ x_0 \stackrel{\mu}{=} \bigvee_{i=1}^{m_0} \bigwedge_{j=1}^{n_i} x_{q_{i,j}}(d_{i,j}) \in M_1 \right\}$$

$$R = \left\{ \begin{array}{c} p(d) \text{:-}\ \ p_{1,1}(d_{1,1}),\ \ldots\ , p_{1,n_1}(d_{1,n_1}).,\\ \vdots \\ p(d) \text{:-}\ p_{m_p,1}(d_{m_p,1}),\ \ldots\ , p_{m_p,n_{m_p}}(d_{m_p,n_{m_p}}). \end{array} \middle| \ x_p(d) \stackrel{\mu}{=} \bigvee_{i=1}^{m_p} \bigwedge_{j=1}^{n_i} x_{p_{i,j}}(d_{i,j}) \in M_1 \right\}$$

Then $[\![B]\!]\delta(x_0) = \mathsf{true}$ *if and only if $q_B = \langle G, R\rangle$ is satisfiable.*

Symmetrically, the result follows immediately by construction since satisfaction of the boolean variables is preserved in the transformation: each boolean variable which is defined as a disjunction of conjunctions is transformed into a predicate defined by a set of clauses with the same head and whose bodies consist of the predicates that represent the boolean variables in the each conjunctions.

Example 12

This example illustrates the reduction method from Datalog to PBES by means of the pointer analysis introduced previously in the Example 4 of Chapter 2. Let $q = \langle G, R \rangle$ be the following Datalog query with domains $D_{\mathtt{H}} = \{o1, o2\}$) and $D_{\mathtt{V}} = \{p, q, r, w\}$:

```
:- vP (V, o2).
vP0(p,o1).
vP0(q,o2).
assign(r,q).
assign(w,r).
load(q,f,w).
store(q,f,p).
vP(V,H) :- vP0(V,H).
vP(V,H) :- assign(V,V2), vP(V2,H).
vP(V,H) :- load(V2,F,V), vP(V2,H2), hP(H2,F,H).
hP(H1,F,H2) :- store(V1,F,V2), vP(V1,H1), vP(V2,H2).
```

By using Proposition 10, we obtain the following PBES:

$$
\begin{array}{lcl}
x_0 & \overset{\mu}{=} & \exists V \in D_{\mathtt{V}} \,.\, x_{\mathtt{vP}}(V, \mathtt{o2}) \\
x_{\mathtt{vP0}}(\mathtt{p}, \mathtt{o1}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\mathtt{vP0}}(\mathtt{q}, \mathtt{o2}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\mathtt{assign}}(\mathtt{r}, \mathtt{q}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\mathtt{assign}}(\mathtt{w}, \mathtt{r}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\mathtt{load}}(\mathtt{q}, \mathtt{f}, \mathtt{w}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\mathtt{store}}(\mathtt{q}, \mathtt{f}, \mathtt{p}) & \overset{\mu}{=} & \mathsf{true} \\
x_{\mathtt{vP}}(V : D_{\mathtt{V}}, H : D_{\mathtt{H}}) & \overset{\mu}{=} & x_{\mathtt{vP0}}(V, H) \vee \\
 & & \exists V_2 \in D_{\mathtt{V}}. \ (\\
 & & \quad x_{\mathtt{assign}}(V, V_2) \wedge \\
 & & \quad x_{\mathtt{vP}}(V_2, H) \\
 & &) \vee \\
 & & \exists F \in D_{\mathtt{F}}. \ \exists H_2 \in D_{\mathtt{H}}. \ \exists V_2 \in D_{\mathtt{V}}. \ (\\
 & & \quad x_{\mathtt{load}}(V_2, F, V) \wedge \\
 & & \quad x_{\mathtt{vP}}(V_2, H_2) \wedge \\
 & & \quad x_{\mathtt{hP}}(H_2, F, H) \\
 & &)
\end{array}
$$

$$x_{\texttt{hP}}(H_1 : D_{\texttt{H}}, F : D_{\texttt{F}}, H_2 : D_{\texttt{H}}) \overset{\mu}{=} \exists V_1 \in D_{\texttt{V}}.\exists V_2 \in D_{\texttt{V}}.\ (x_{\texttt{store}}(V_1, F, V_2) \wedge x_{\texttt{vP}}(V_2, H_2) \wedge x_{\texttt{vP}}(V_1, H_1))$$

In the rest of this section, we further develop our methodology for solving Datalog queries by using PBESs. Given that BES solvers usually work at the plain BES level, as opposed to the PBES level, we have to transform our generated PBES into a plain BES for our approach to effectively take advantage of the existing BES technology.

3.2.1 Instantiation to parameterless BES

Among the different known techniques for solving a PBES [vDPW08], such as Gauss elimination with symbolic approximation, and the use of patterns, under/over approximations, or invariants, we consider the resolution method that is based on transforming the PBES into an alternation-free parameterless boolean equation system (BES) that can be solved by linear time and memory algorithms [Mat98, vDPW08] when data domains are finite.

Definition 13 (Boolean Equation System) *A* Boolean Equation System *(*BES*)* $B = (x_0, M_1, ..., M_n)$ *is a* PBES *where data domains are removed and boolean variables, being independent from data parameters, are considered to be* propositional*.*

To obtain a direct transformation into a parameterless BES, we first describe the PBES in a simpler format. This simplification step consists in introducing new variables, such that each formula at the right-hand side of a boolean equation only contains at most one operator. Hence, boolean formulae are restricted to pure disjunctive or conjunctive formulae.

Given a Datalog query $q = \langle G, R \rangle$, by applying this simplification to the PBES of Proposition 10, we obtain the following PBES:

$$x_0 \overset{\mu}{=} \bigvee_{:- q_1(d_1),...,q_m(d_m). \in G} g_{q_1(d_1),...,q_m(d_m)}$$

$$g_{q_1(d_1),...,q_m(d_m)} \overset{\mu}{=} \bigwedge_{i:=1}^{m} x_{q_i}(d_i)$$

$$x_p(d : D) \overset{\mu}{=} \bigvee_{p(d) :- p_1(d_1),...,p_m(d_m). \in R} r_{p(d) :- p_1(d_1),...,p_m(d_m)}$$

$$r_{p(d) :- p_1(d_1),\ldots,p_m(d_m)} \overset{\mu}{=} \bigwedge_{i:=1}^{m} x_{p_i}(d_i)$$

Each $g_{\ldots}$ and $r_{\ldots}$ boolean variables respectively represents a Datalog *goal* and *rule*. Consequently, they are both defined as the conjunction of the boolean variables $x_{\ldots}$ representing their respective subgoals. Notice that facts are considered to be rules with an empty body, which is equivalent to a body consisting of just the formula true. By encoding a predicate symbol `p` as a corresponding parameterised boolean variable $x_{\mathtt{p}}$, a fact `p(d)` is represented by means of a (parameterless) boolean variable $r_{\mathtt{p(d)}}$, which represents a rule with an empty body and the fact `p(d)` as its head. Since the conjunction of an empty set is true, this boolean variable is defined as:

$$r_{\mathtt{p(d)}} \overset{\mu}{=} \mathsf{true}$$

By applying the instantiation algorithm of Mateescu [Mat98], we eventually obtain a parameterless BES, where all possible values of each typed data terms have been enumerated over their corresponding finite data domains.

The resulting implicit parameterless BES is defined as follows, where $\preceq$ is the standard preorder of relative generality (instantiation ordering).

$$x_0 \overset{\mu}{=} \bigvee_{:- \ q_1(d_1),\ldots,q_m(d_m). \ \in G} g_{q_1(d_1),\ldots,q_m(d_m)} \tag{3.3}$$

$$g_{q_1(d_1),\ldots,q_m(d_m)} \overset{\mu}{=} \bigvee_{1\leq i\leq m,\, e_i\in D_i \wedge d_i \preceq e_i} g^i_{q_1(e_1),\ldots,q_m(e_m)} \tag{3.4}$$

$$g^i_{q_1(e_1),\ldots,q_m(e_m)} \overset{\mu}{=} \bigwedge_{i:=1}^{m} x_{q_i(e_i)} \tag{3.5}$$

$$x_{p(d)} \overset{\mu}{=} \bigvee_{p(d) :- p_1(d_1),\ldots,p_m(d_m). \ \in R} r_{p(d) :- p_1(d_1),\ldots,p_m(d_m)} \tag{3.6}$$

$$r_{p_0(d_0) :- p_1(d_1),\ldots,p_m(d_m)} \overset{\mu}{=} \bigvee_{0\leq i\leq m,\, e_i\in D_i \wedge d_i \preceq e_i} r^i_{p_0(e_0) :- p_1(e_1),\ldots,p_m(e_m)} \tag{3.7}$$

$$r^i_{p_0(e_0) :- p_1(e_1),\ldots,p_m(e_m)} \overset{\mu}{=} \bigwedge_{i:=1}^{m} x_{p_i(e_i)} \tag{3.8}$$

Observe that Equation 3.1 is transformed into a set of parameterless equations (3.3, 3.4, 3.5). First, Equation 3.3 describes the set of parameterised goals $g_{q_1(d_1),\ldots,q_m(d_m)}$ of the query. Then, Equation 3.4 represents the instantiation of each variable parameter d_i to the possible values e_i from the domain. Finally, Equation 3.5 states that each instantiated goal $g^i_{q_1(e_1),\ldots,q_m(e_m)}$ is satisfied whenever the values e_i make all predicates q_i of the goal true. Similarly,

Equation 3.2 (describing Datalog rules) is encoded into a set of parameterless equations (3.6, 3.7, 3.8).

Example 14

Let us instantiate the PBES we obtained in Example 12. By applying Mateescu instantiation algorithm, we obtain the following BES:

$$
\begin{aligned}
x_0 &\overset{\mu}{=} g_{\texttt{vP(V,o2)}} \\
g_{\texttt{vP(V,o2)}} &\overset{\mu}{=} g^i_{\texttt{vP(p,o2)}} \vee g^i_{\texttt{vP(q,o2)}} \\
&\quad \vee g^i_{\texttt{vP(r,o2)}} \vee g^i_{\texttt{vP(w,o2)}} && (3.9)\\
g^i_{\texttt{vP(p,o2)}} &\overset{\mu}{=} x_{\texttt{vP(p,o2)}} && (3.10)\\
x_{\texttt{vP(p,o2)}} &\overset{\mu}{=} r_{\texttt{vP(p,o2) :- vP0(p,o2)}} && (3.11)\\
&\quad \vee r_{\texttt{vP(p,o2) :- assign(p,V2),vP(V2,o2)}} \\
&\quad \vee r_{\texttt{vP(p,o2) :- load(V2,F,p),vP(V2,H2),}\\
&\qquad\qquad \texttt{hP(H2,F,o2)}} \\
r_{\texttt{vP(p,o2) :- vP0(p,o2)}} &\overset{\mu}{=} r^i_{\texttt{vP(p,o2) :- vP0(p,o2)}} && (3.12)\\
r^i_{\texttt{vP(p,o2) :- vP0(p,o2)}} &\overset{\mu}{=} x_{\texttt{vP0(p,o2)}} \\
x_{\texttt{vP0(p,o2)}} &\overset{\mu}{=} \mathsf{false} \\
r_{\texttt{vP(p,o2):-assign(p,V2),vP(V2,o2)}} &\overset{\mu}{=} r^i_{\texttt{vP(p,o2) :- assign(p,p),vP(p,o2)}} && (3.13)\\
&\quad \vee r^i_{\texttt{vP(p,o2) :- assign(p,q),vP(q,o2)}} \\
&\quad \vee r^i_{\texttt{vP(p,o2) :- assign(p,r),vP(r,o2)}} \\
&\quad \vee r^i_{\texttt{vP(p,o2) :- assign(p,w),vP(w,o2)}} \\
r^i_{\texttt{vP(p,o2) :- assign(p,p),vP(p,o2)}} &\overset{\mu}{=} x_{\texttt{assign(p,p)}} \wedge x_{\texttt{vP(p,o2)}} \\
x_{\texttt{assign(p,p)}} &\overset{\mu}{=} \mathsf{false} \\
r^i_{\texttt{vP(p,o2) :- assign(p,q),vP(q,o2)}} &\overset{\mu}{=} x_{\texttt{assign(p,q)}} \wedge x_{\texttt{vP(q,o2)}} \\
x_{\texttt{assign(p,q)}} &\overset{\mu}{=} \mathsf{false} \\
r^i_{\texttt{vP(p,o2) :- assign(p,r),vP(r,o2)}} &\overset{\mu}{=} x_{\texttt{assign(p,r)}} \wedge x_{\texttt{vP(r,o2)}} \\
x_{\texttt{assign(p,r)}} &\overset{\mu}{=} \mathsf{false} \\
r^i_{\texttt{vP(p,o2) :- assign(p,w),vP(w,o2)}} &\overset{\mu}{=} x_{\texttt{assign(p,w)}} \wedge x_{\texttt{vP(w,o2)}} \\
x_{\texttt{assign(p,w)}} &\overset{\mu}{=} \mathsf{false}
\end{aligned}
$$

$$r_{\texttt{vP(p,o2) :- load(V2,F,p)...}} \overset{\mu}{=} r^i_{\texttt{vP(p,o2) :- load(p,f,p),vP(p,o1), hP(o1,f,o2)}} \vee r^i_{\texttt{vP(p,o2) :- load(p,f,p),vP(p,o2), hP(o2,f,o2)}} \vee r^i_{\texttt{vP(p,o2) :- load(q,f,p),vP(q,o1), hP(o1,f,o2)}} \vee \cdots \tag{3.14}$$

$$g^i_{\texttt{vP(q,o2)}} \overset{\mu}{=} x_{\texttt{vP(q,o2)}} \tag{3.15}$$

$$x_{\texttt{vP(q,o2)}} \overset{\mu}{=} r_{\texttt{vP(q,o2) :- vP0(q,o2)}} \vee r_{\texttt{vP(q,o2) :- assign(q,V2),vP(V2,o2)}} \vee r_{\texttt{vP(q,o2) :- load(V2,F,q),vP(V2,H2), hP(H2,F,o2)}}$$

$$r_{\texttt{vP(q,o2) :- vP0(q,o2)}} \overset{\mu}{=} r^i_{\texttt{vP(q,o2) :- vP0(q,o2)}}$$

$$r^i_{\texttt{vP(q,o2) :- vP0(q,o2)}} \overset{\mu}{=} x_{\texttt{vP0(q,o2)}}$$

$$x_{\texttt{vP0(q,o2)}} \overset{\mu}{=} r_{\texttt{vP0(q,o2)}}$$

$$r_{\texttt{vP0(q,o2)}} \overset{\mu}{=} \mathsf{true}$$

$$g^i_{\texttt{vP(r,o2)}} \overset{\mu}{=} x_{\texttt{vP(r,o2)}} \tag{3.16}$$

$$x_{\texttt{vP(r,o2)}} \overset{\mu}{=} r_{\texttt{vP(r,o2) :- vP0(r,o2)}} \vee r_{\texttt{vP(r,o2) :- assign(r,V2),vP(V2,o2)}} \vee r_{\texttt{vP(r,o2) :- load(V2,F,r),vP(V2,H2), hP(H2,F,o2)}}$$

$$r_{\texttt{vP(r,o2) :- vP0(r,o2)}} \overset{\mu}{=} r^i_{\texttt{vP(r,o2) :- vP0(r,o2)}}$$

$$r^i_{\texttt{vP(r,o2) :- vP0(r,o2)}} \overset{\mu}{=} x_{\texttt{vP0(r,o2)}}$$

$$x_{\texttt{vP0(r,o2)}} \overset{\mu}{=} \mathsf{false}$$

$$r_{\texttt{vP(r,o2) :- assign(r,V2),vP(V2,o2)}} \overset{\mu}{=} r^i_{\texttt{vP(r,o2) :- assign(r,p),vP(p,o2)}} \vee r^i_{\texttt{vP(r,o2) :- assign(r,q),vP(q,o2)}} \vee r^i_{\texttt{vP(r,o2) :- assign(r,r),vP(r,o2)}} \vee r^i_{\texttt{vP(r,o2) :- assign(r,w),vP(w,o2)}}$$

$$r^i_{\texttt{vP(r,o2) :- assign(r,p),vP(p,o2)}} \overset{\mu}{=} x_{\texttt{assign(r,p)}} \wedge x_{\texttt{vP(p,o2)}}$$

$$x_{\texttt{assign(r,p)}} \overset{\mu}{=} \mathsf{false}$$

$$
\begin{aligned}
r^{i}_{\texttt{vP(r,o2) :- assign(r,q),vP(q,o2)}} &\overset{\mu}{=} x_{\texttt{assign(r,q)}} \wedge x_{\texttt{vP(q,o2)}} \\
x_{\texttt{assign(r,q)}} &\overset{\mu}{=} r_{\texttt{assign(r,q)}} \\
r_{\texttt{assign(r,q)}} &\overset{\mu}{=} \mathsf{true} \\
&\vdots \\
g^{i}_{\texttt{vP(w,o2)}} &\overset{\mu}{=} x_{\texttt{vP(w,o2)}} \\
&\vdots
\end{aligned}
\tag{3.17}
$$

We have omitted some parts of the transformation since they do not contribute further to its understanding. In order to improve readability, we have also shortened the names of some boolean variables.

The boolean variable $g_{\texttt{vP(V,o2)}}$, which represents the query `:- vP(V,o2)`, is instantiated in Equation 3.9, producing the new boolean variables defined in Equations 3.10, 3.15, 3.16 and 3.17. The boolean variables that represent a predicate (e.g., $x_{\texttt{vP(p,o2)}}$) are defined by means of other boolean variables representing partial instances of rules that have the represented predicate at its head (e.g., $r_{\texttt{vP(p,o2) :- vP0(p,o2)}}$) as in Equation 3.11. These boolean variables representing partial instances of rules are further instantiated producing variables that represent fully instantiated rules (e.g., $r^{i}_{\texttt{vP(p,o2) :- assign(p,p),vP(p,o2)}}$) as in Equations 3.12, 3.13 and 3.14.

3.2.2 Optimizations to the basic BES resolution technique

The parameterless BES described above is rather inefficient since it adopts a brute-force approach that, at the very first steps of the computation (Equation 3.4), enumerates all possible tuples of the query (see Equation 3.9 in Example 14). It is well-known that a general Datalog program runs in $O(n^k)$ time, where k is the largest number of variables in any single rule, and n is the number of constants in the facts and rules [TL10]. Similarly, for a simple query like `:- vP(V,H).`, with `V` and `H` respectively being elements of domains D_v and D_h, each one of size 10 000, Equation 3.4 generates D^2, i.e., 10^8, boolean variables representing all possible combinations of values `V` and `H` in the relation `vP`. Usually, for each atom in a Datalog program, the number of facts that are given or inferred by the Datalog rules is much lower than the product of the *domain's sizes* of its corresponding arguments. Ideally, the Datalog query evaluation should enumerate (given or inferred) facts only *on-demand*.

Among the existing optimizations for top-down evaluation of queries in Datalog, the so-called *Query-Sub-Query* [Vie86] technique consists in minimizing the number of tuples derived by a rewriting of the program based

on the propagation of bindings. Basically, the method aims at keeping the bindings of variables between atoms $p(a)$ in a rule.

In our Datalog evaluation technique based on BES, we adopt a similar approach: two boolean Equations 3.19 and 3.25, which are modified versions of Equations 3.4 and 3.7 of the basic BES instantiation, only enumerate the values of variable arguments that appear more than once in the body of the corresponding Datalog rule; otherwise, arguments are kept unchanged. Moreover, if the atom $p(a)$ is part of the extensional database, the only possible values of its variable arguments are values that reproduce a given fact of the Datalog program. We denote as D_i^p the subdomain of D that contains all possible values of the i^{th} variable argument of p if p is in the extensional database, otherwise $D_i^p = D$. Hence, the resulting BES resolution is likely to process fewer facts and be more efficient than the brute-force approach.

Following this optimization technique, a parameterless BES can directly be derived from the previous BES representation as follows:

$$x_0 \overset{\mu}{=} \bigvee_{:-\ q_1(d_1),\ldots,q_m(d_m).\ \in G} g_{q_1(d_1),\ldots,q_m(d_m)} \tag{3.18}$$

$$g_{q_1(d_1),\ldots,q_m(d_m)} \overset{\mu}{=} \bigvee_{\substack{\{a_1,\ \ldots,\ a_m\} \in \{V \cup D_1^{q_1}\} \times \cdots \times \{V \cup D_1^{q_m}\} \\ \textit{if}\ (\exists\ j \in [1..m],\ j \neq i \mid d_i = d_j \wedge d_i \in V) \\ \quad \textit{then}\ a_i \in D_1^{q_i}\ \wedge\ (\forall\ j \in [1..m],\ d_i = d_j \mid a_j := a_i) \\ \quad \textit{else}\ a_i := d_i}} g^{pi}_{q_1(a_1),\ldots,q_m(a_m)} \tag{3.19}$$

$$g^{pi}_{q_1(a_1),\ldots,q_m(a_m)} \overset{\mu}{=} \bigwedge_{i:=1}^{m} x_{q_i(a_i)} \tag{3.20}$$

$$x_{q(a)} \overset{\mu}{=} x^f_{q(a)} \vee x^c_{q(a)} \tag{3.21}$$

$$x^f_{q(a)} \overset{\mu}{=} \bigvee_{(e:=a\ \wedge\ a \in C)\ \vee\ (e \in D_1^q\ \wedge\ a \in V)\ \mid q(e).\in R} x^i_{q(e)} \tag{3.22}$$

$$x^i_{q(e)} \overset{\mu}{=} \mathsf{true} \tag{3.23}$$

$$x^c_{p(a)} \overset{\mu}{=} \bigvee_{p(a):-p_1(d_1),\ldots,p_m(d_m).\ \in R} r_{p(a):-p_1(d_1),\ldots,p_m(d_m)} \tag{3.24}$$

$$r_{p_0(d_0):-p_1(d_1),\ldots,p_m(d_m)} \overset{\mu}{=} \bigvee_{\substack{\{a_0,\ldots,a_m\} \in \{V \cup D_1^{p_0}\} \times \cdots \times \{V \cup D_1^{p_m}\} \\ \textit{s.t. if}\ (\exists\ j \in [1..m],\ j \neq i \mid d_i = d_j \wedge d_i \in V) \\ \quad \textit{then}\ a_i \in D_1^{p_i}\ \wedge\ (\forall\ j \in [1..m],\ d_i = d_j \mid a_j := a_i) \\ \quad \textit{else}\ a_i := d_i}} r^{pi}_{p_0(d_0):-p_1(a_1),\ldots,p_m(a_m)} \tag{3.25}$$

$$r^{pi}_{p_0(d_0):-p_1(a_1),\ldots,p_m(a_m)} \overset{\mu}{=} \bigwedge_{i:=1}^{m} x_{p_i(a_i)} \tag{3.26}$$

Boolean variables whose name starts with x are those that correspond to the goal and subgoals of the original program and we call them *original* variables, whereas boolean variables starting with r or g are auxiliary variables that are defined during unfolding and instantiation of (sub)goals. Observe that Equations 3.18, 3.20, 3.24 and 3.26 respectively correspond to Equations 3.3, 3.5, 3.6 and 3.8 of the previous BES definition with only a slight renaming of the generated boolean variables. The important novelty here is that, instead of enumerating all possible values of the domain, as it is done in Equation 3.4, the corresponding new Equation 3.19 only enumerates the values of variable arguments that are repeated in the body of a rule; otherwise, variable arguments are kept unchanged i.e., $a_i := d_i$. Actually, the generated boolean variables $g^{pi}_{q_1(a_1),\ldots,q_m(a_m)}$, where pi stands for *partially instantiated*, may still refer to atoms containing variable arguments. Thus, the combinatorial explosion of possible tuples is avoided at this point and delayed to future steps. Equation 3.21 generates two boolean successors for variable $x_{q(a)}$: $x^f_{q(a)}$ and $x^c_{q(a)}$. When q is part of the extensional database, $x^f_{q(a)}$ represents the possible solutions to $x_{q(a)}$ whereas $x^c_{q(a)}$ is false. On the contrary, if q is part of the intensional database, $x^c_{q(a)}$ computes solutions to $x_{q(a)}$ while $x^f_{q(a)}$ is false. In Equation 3.22, each value of a variable or constant that leads to a given fact $q(e)$ of the program generates a new boolean variable $x^i_{q(e)}$, where i stands for (fully) *instantiated*, that is true by definition of a fact. Equation 3.24 mimics the evaluation of Datalog rules whose head is $p(a)$. Note that Equations 3.19, 3.22, and 3.25 enumerate only possible values of the subdomains $D_1^{p_i}$ instead of the full domain D. For the Datalog program described in Figure 3.1, this restriction would consist in using four new subdomains $D_{v1}^{\texttt{vP0}} = \{p, q\}$, $D_{h2}^{\texttt{vP0}} = \{o1, o2\}$, $D_{v1}^{\texttt{assign}} = \{r, w\}$, and $D_{v2}^{\texttt{assign}} = \{q, r\}$, instead of full domains D_h and D_v for the values of each variable argument in relations `vP0` and `assign`.

Example 15

To illustrate the idea behind this optimized version of the generated BES, we show (a part of) the BES that results from applying the preceding optimization to our running example.

$$
\begin{aligned}
x_0 &\overset{\mu}{=} g_{\texttt{vP(V,o2)}} \\
g_{\texttt{vP(V,o2)}} &\overset{\mu}{=} g^{pi}_{\texttt{vP(V,o2)}} \\
g^{pi}_{\texttt{vP(V,o2)}} &\overset{\mu}{=} x_{\texttt{vP(V,o2)}} \\
x_{\texttt{vP(V,o2)}} &\overset{\mu}{=} x^{f}_{\texttt{vP(V,o2)}} \vee x^{c}_{\texttt{vP(V,o2)}} \\
x^{f}_{\texttt{vP(V,o2)}} &\overset{\mu}{=} \mathsf{false} \\
x^{c}_{\texttt{vP(V,o2)}} &\overset{\mu}{=} \bigvee r_{\texttt{vP(V,o2) :- vP0(V,o2)}} \\
&\qquad \vee r_{\texttt{vP(V,o2) :- assign(V,V2),vP(V2,o2)}} \\
r_{\texttt{vP(V,o2) :- vP0(V,o2)}} &\overset{\mu}{=} r^{pi}_{\texttt{vP(V,o2) :- vP0(V,o2)}} \\
r^{pi}_{\texttt{vP0(V,o2)}} &\overset{\mu}{=} x_{\texttt{vP0(V,o2)}} \\
x_{\texttt{vP0(V,o2)}} &\overset{\mu}{=} x^{f}_{\texttt{vP0(V,o2)}} \vee x^{c}_{\texttt{vP0(V,o2)}} \\
x^{f}_{\texttt{vP0(V,o2)}} &\overset{\mu}{=} x^{i}_{\texttt{vP0(q,o2)}} \\
x^{i}_{\texttt{vP0(q,o2)}} &\overset{\mu}{=} \mathsf{true} \\
r_{\texttt{vP(V,o2) :- assign(V,V2),vP(V2,o2)}} &\overset{\mu}{=} \bigvee r^{pi}_{\texttt{vP(V,o2) :- assign(V,q),vP(q,o2)}} \\
&\qquad \vee r^{pi}_{\texttt{vP(V,o2) :- assign(V,r),vP(r,o2)}} \\
r^{pi}_{\texttt{vP(V,o2) :- assign(V,q),vP(q,o2)}} &\overset{\mu}{=} x_{\texttt{assign(V,q)}} \wedge x_{\texttt{vP(q,o2)}} \\
x_{\texttt{assign(V,q)}} &\overset{\mu}{=} x^{f}_{\texttt{assign(V,q)}} \vee x^{c}_{\texttt{assign(V,q)}} \\
x^{f}_{\texttt{assign(V,q)}} &\overset{\mu}{=} x^{i}_{\texttt{assign(r,q)}} \\
x^{i}_{\texttt{assign(r,q)}} &\overset{\mu}{=} \mathsf{true} \\
&\vdots
\end{aligned}
$$

Each *original* variable is defined as the disjunction of r (or g) boolean variables that represent the body of one of the clauses that define the corresponding predicate (see equation for variable $x^{c}_{\texttt{vP(V,o2)}}$). Then, each r (or g) variable is defined as the disjunction of the different possible instantiations of the query on the shared variables (see equation for variable $r_{\texttt{vP(V,o2) :- assign(V,V2),vP(V2,o2)}}$). These *partial instantiations* are represented by r^{pi} (or g^{pi}) boolean variables. The r^{pi} variables are defined as the conjunction of the subgoals, which are represented by (*original*) x variables. Finally, the original variables x are defined as the disjunction of the boolean variables that correspond to querying the *facts* x^{f} and querying the *clauses* x^{c} (see equation for $x_{\texttt{assign(V,q)}}$).

As stated above, when the r^{pi} variables are generated, only variables that are shared by two or more subgoals in the body of the Datalog program are instantiated, and only values that appear in the corresponding parameters of the program facts are used. In other words, we do not generate spurious variables, such as $r^{pi}_{\texttt{vP(V,H) :- assign(V,w),vP(w,H)}}$, which can never be true.

3.2.3 Solution extraction

Considering the optimized parameterless BES defined above, the query satisfiability problem is reduced to the local resolution of boolean variable x_0. The value (true or false) computed for x_0 indicates whether there exists at least one satisfiable goal in G. We can remark that the BES representing the evaluation of a Datalog query is only composed of one equation block that contains alternating dependencies between disjunctive and conjunctive variables. Hence, it can be solved by optimized depth-first search (DFS) for such a type of equation block. However, since the DFS strategy can only conclude the existence of a solution to the query by computing a minimal number of boolean variables, it is necessary to use a breadth-first search (BFS) strategy to compute all the different solutions to a Datalog query. Such a strategy will "force" the resolution of all boolean variables that have been put in the BFS queue, even if the satisfiability of the query has been computed in the meantime. Consequently, the solver will compute all possible boolean variables $x^i_{q(e)}$, which represent potential solutions for the query. Upon termination of the BES resolution (ensured by finite data domains and table-based exploration), query solutions, i.e., combinations of variable values $\{e_1, \ldots, e_m\}$, one for each atom of the query that lead to a satisfied query, are extracted from all boolean variables $x^i_{q(e)}$ that are reachable from the boolean variable x_0 through a path of true boolean variables.

3.3 The prototype DATALOG_SOLVE

We implemented the Datalog query transformation to BES in a powerful, fully automated Datalog solver tool, called DATALOG_SOLVE, that we developed within the CADP verification toolbox. Without loss of generality, in this section, we describe the DATALOG_SOLVE tool focusing on JAVA program analysis. Other source languages and classes of problems can be specified in Datalog and solved by our tool as well.

DATALOG_SOLVE takes three different inputs (see Figure 3.3): the (optional) domain definitions (`.map`), the Datalog constraints or *facts* (`.tuples`), and a Datalog query $q = \langle G, R \rangle$ (`.datalog`, e.g. `pa.datalog` in Figure 3.4). The domain definitions state the possible values for each predicate's argument of the query. These are meaningful names for the numerical values that are used to efficiently described the Datalog constraints. For example, in the context of pointer analyses, variable names (`var.map`) and heap locations (`heap.map`) are two domains of interest. Each line of a `.map` file represents a different domain element. For efficiency reasons, a domain element is identi-

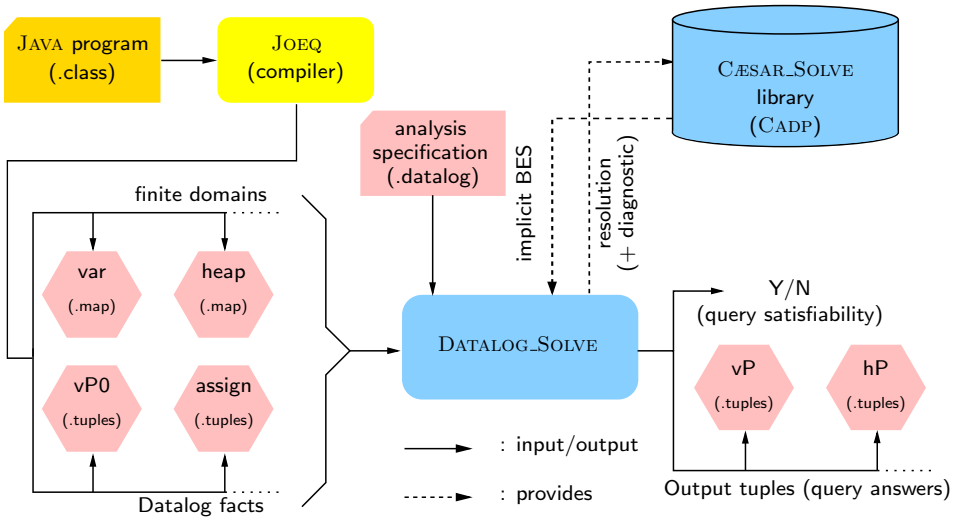


Figure 3.3: JAVA program analysis using the DATALOG_SOLVE tool.

fied by its line number, thus its human-readable description is provided by the content of its `.map` file's associated line. The Datalog constraints represent information relevant for the analysis. For instance, `vP0.tuples` gives all direct references from variables to heap objects in a given program. These combinations are described by numerical values in the range $0..(\mathit{domain\ size} - 1)$, which represents domain elements identifiers accordingly to the `.map` files.

Both, domain definitions and facts are specified in the `.datalog` input file (see Figure 3.4 for an example) and they are automatically extracted from program source code by using the JOEQ compiler framework [Wha05] that we slightly modified to generate *tuple-based* instead of BDD-*based* input relations (extensional database). The `.datalog` input file has three sections separated by its corresponding headers:

Domains Declares a domain on each line by means of three consecutive fields: the domain identifier, the domain size, and the domain `.map` file.

Relations Declares the predicate symbols that are used in the program by means of their identifiers, the association of their arguments to previously declared domains, and stating whether they are part of the extensional (`inputtuples`) or the intensional (`outputtuples`) databases. If a predicate p is declared as extensional, a file named p`.tuples` will be used to load the facts associated with p.

```
### Domains
V 262144 variable.map
H 65536 heap.map
F 16384 field.map
### Relations
vP_0      (variable :  V, heap :  H)            inputtuples
store     (base :  V, field :  F, source :  V)  inputtuples
load      (base :  V, field :  F, dest :  V)    inputtuples
assign    (dest :  V, source :  V)              inputtuples
vP        (variable :  V, heap :  H)            outputtuples
hP        (base :  H, field :  F, target :  H)  outputtuples
### Rules
vP (V1, H1)     :- vP_0(V1, H1).
vP (V1, H1)     :- assign(V1, V2), vP(V2, H2).
hP (H1, F1, H2) :- store(V1, F1, V2), vP(V1, H1), vP(V2, H2).
vP (V2, H2)     :- load (V1, F1, V2), vP(V1, H1), hP(H1, F1, H2).
```

Figure 3.4: DATALOG_SOLVE input file specifying Andersen's points-to analysis.

Rules Contains the rules which specify the analysis to be performed.

The core of the tool is DATALOG_SOLVE 1.0 (120 lines of LEX, 380 lines of BISON and 3 500 lines of C code) which proceeds in two steps:

1. The front-end of DATALOG_SOLVE constructs the optimized implicit BES representation given by Equations 3.18-3.26 from the inputs.

2. The back-end of our tool carries out the demand-driven generation, resolution and interpretation of the BES by means of a generic library of CADP called CÆSAR_SOLVE, which is devised for local BES resolution and diagnostic generation.

This architecture clearly separates the implementation of Datalog-based static analyses from the resolution engine, which can be extended and optimized independently. We further discuss some optimizations in Section 3.4.

DATALOG_SOLVE returns both the query's satisfiability and the computed answers represented numerically in various output files (`.tuples` files) upon termination, which is ensured by safe input Datalog programs. The tool takes as a default query the computation of the least set of facts that contains all the facts that can be inferred using the given rules. This represents the worst

case of a demand-driven evaluation and computes all the information derivable from the considered Datalog program.

3.4 Experimental results

We have applied the DATALOG_SOLVE tool to a number of JAVA programs by computing the context-insensitive pointer analysis described in Figure 3.4.

In order to test the scalability and applicability of the transformation, we applied our technique to four of the most popular 100% JAVA projects on Sourceforge that could compile directly as standalone applications and were previously used as benchmarks for the BDDBDDB tool [WACL05]. They are all real applications with tens of thousands of users each. Projects vary in the number of classes, methods, variables, and heap allocations. The information details, shown on Table 3.1, are calculated on the basis of a context-insensitive callgraph precomputed by the JOEQ[2] compiler. All experiments were conducted using JAVA JRE 1.5, JOEQ version 20030812, on a Intel Core 2 T5500 1.66GHz with 3 Gigabytes of RAM, running Linux Kubuntu 8.04.

Table 3.1: Description of the JAVA projects used as benchmarks.

freetts (1.2.1): speech synthesis system			
Classes:	215	Variables:	8K
Methods:	723	Allocations:	3K
nfcchat (1.1.0): scalable, distributed chat client			
Classes:	283	Variables:	11K
Methods:	993	Allocations:	3K
jetty (6.1.10): server and servlet container			
Classes:	309	Variables:	12K
Methods:	1160	Allocations:	3K
joone (2.0.0): Java neural net framework			
Classes:	375	Variables:	17K
Methods:	1531	Allocations:	4K

[2] http://joeq.sourceforge.net/

Table 3.2: Time (in seconds) and peak memory usage (in megabytes) for the context-insensitive pointer analysis of each benchmark.

Name	time (sec.)	memory (Mb.)
freetts (1.2.1)	10	61
nfcchat (1.1.0)	8	59
jetty (6.1.10)	73	70
joone (2.0.0)	4	58

The analysis time and memory usage of our context-insensitive pointer analysis, shown in Table 3.2, illustrate the scalability of our BES resolution and validate our theoretical results on real examples. DATALOG_SOLVE solves the (default) query for all benchmarks in a few seconds. The computed results were additionally verified by comparing them with the solutions computed by the BDDBDDB tool on the same benchmark of JAVA programs and analysis.

3.4.1 Further Improvements

A new BES-based approach for the resolution of Datalog programs was developed by the author, in joint work with Christophe Joubert and Fernando Tarín [FJT10b, FJT10a]. Our contribution is a novel bottom-up evaluation strategy specially tailored for Datalog-based program analysis. Our work is based on the evaluation strategy presented by Liu and Stoller in [LS03], and further detailed in an extended version of the work [LS09]. Their strategy is a generalization of the systematic algorithm development method of Paige et al. [PK82], which transforms extensive set computations like set union, intersection, and difference into incremental operations. Incremental operations are supported by sophisticated data structures with constant access time. They derive an imperative resolution algorithm, which computes a fixpoint over all (preformatted) rules from an input Datalog program by first considering input predicates, then considering rules with one subgoal, and finally considering rules with two subgoals.

Our novel proposal is to enhance this evaluation strategy by means of the following:

1. A declarative description of the bottom-up resolution strategy that is separate from the fixpoint computation. This is achieved by transforming Datalog programs into *Boolean Equation Systems* (BESs) and evaluating the resulting BESs by standard solvers.

2. A predicate order that is employed to simplify the BES by removing various set operations. This order is determined by the dependency between predicate symbols and the number of times that rules are fired.

3. A sophisticated data-structure with quicker access time and lower memory consumption. This efficient data-structure is based on a complex representation of a *trie*. *Tries*, also called prefix trees, are ordered tree data structures where each node position in the tree is the key that is associated to this node. This structure has faster look-up times than binary search trees and imperfect hash tables.

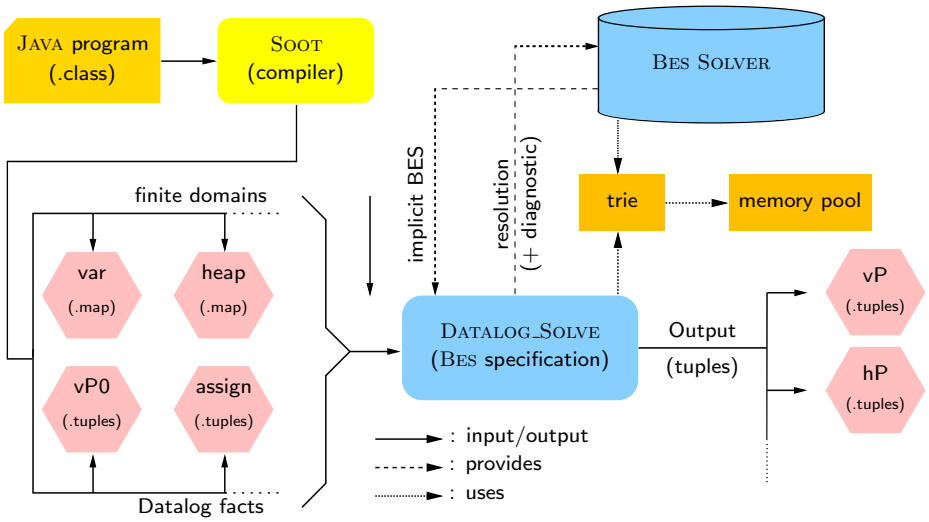


Figure 3.5: JAVA program analysis using the DATALOG_SOLVE 2.0 tool.

We endowed the DATALOG_SOLVE prototype with the new evaluation strategy applied to the evaluation of Andersen's points-to analysis encoded as a BES. DATALOG_SOLVE 2.0 (see Figure 3.5) does not depend on CADP, and uses a simple and fast specific BES solver. Facts are extracted by an extended version of SOOT [3] from the JAVA programs of the DACAPO [4] benchmark with JDK 1.6. We tested the efficiency and feasibility of our implementation by comparing it to two state-of-the-art Datalog solvers XSB `3.2`[5] and the proto-

[3] http://www.sable.mcgill.ca/soot

[4] http://voxel.dl.sourceforge.net/sourceforge/dacapobench/dacapo-2006-10-MR2-xdeps.zip

[5] http://xsb.sourceforge.net

type of Liu and Stoller[6], which in the rest of the chapter we call Ls.

In Figures 3.6 and 3.7, performance results are presented in terms of user time and peak memory consumption evaluation. All experiments were performed on an Intel Core 2 duo E4500 2.2 GHz, with 2048 KB cache, 4 GB of RAM, and running Linux Ubuntu 10.04. Datalog_Solve and Xsb solver were compiled using gcc 4.4.1. Python 2.6.4 was used for the Ls solver. The measures do not include the time needed by Xsb and Ls to precompile the facts. The analysis results were verified by comparing the outputs of all solvers. Datalog_Solve 2.0 evaluates the whole benchmark in only 3 seconds with a

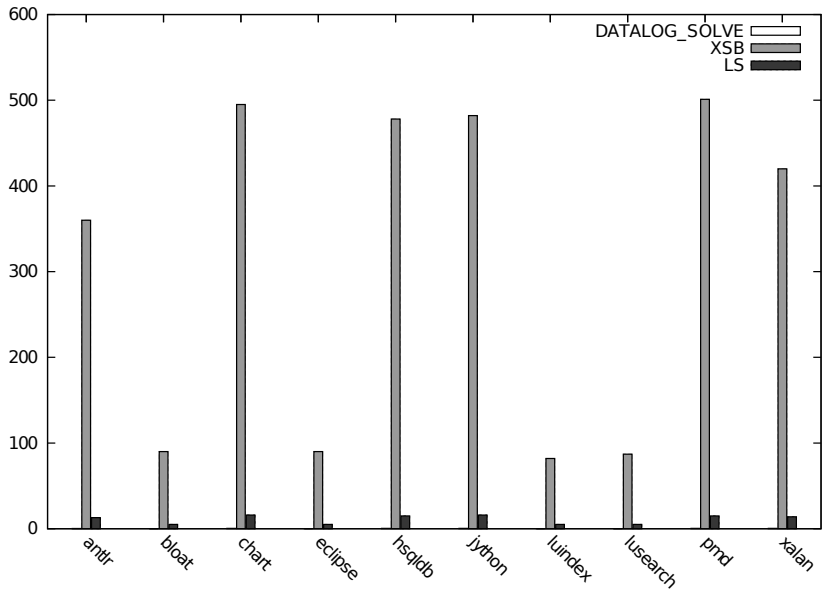


Figure 3.6: Dacapo analysis times (sec.) for various Datalog implementations.

mean-time of 0.3 seconds per program. This explains why the time measures for Datalog_Solve are hardly visible in Figure 3.6. Our experiments demonstrate that Xsb is much slower than Ls, which is in turn an order of magnitude slower than Datalog_Solve. For example, for the benchmark program *pmd*, which is a multi-language source code analyzer, Xsb evaluated the points-to analysis in 501 seconds, Ls solved it in 15 seconds, and Datalog_Solve took 0.391 seconds to solve it. With respect to memory consumption, Ls consumes significantly more than Xsb and Datalog_Solve. For the *pmd* ex-

[6]Provided by the authors of [LS09]

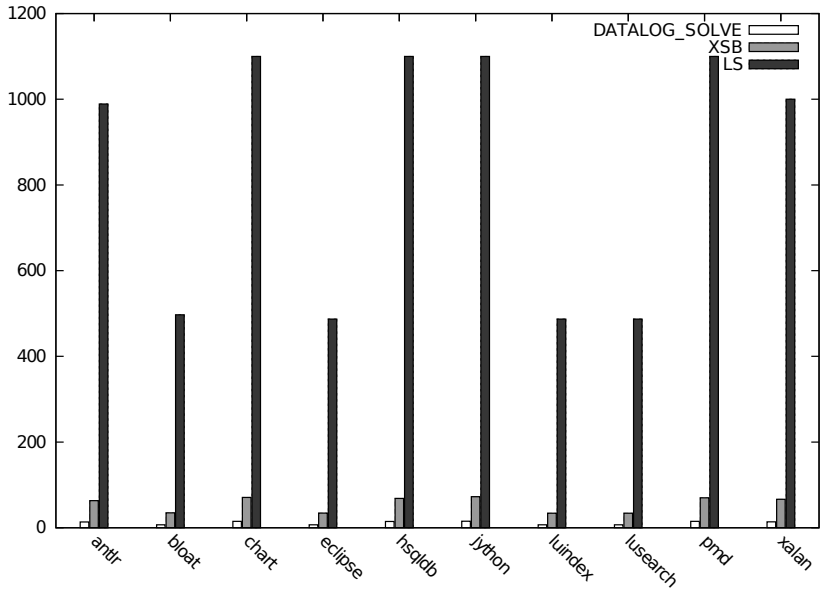


Figure 3.7: DACAPO memory usage (MB.) for various Datalog implementations.

ample, LS required 1.1 GB of memory, while XSB consumed 70 MB, and DATALOG_SOLVE consumed 15 MB. These performance results show that the BES-based evaluation strategy together with an optimized data-structure scales really well for very large programs regarding Andersen's points-to analysis.

3.5 Conclusions and Related Work

We have presented a transformation from Datalog to BES that preserves the program semantics and is aimed at Datalog-based static analysis. The transformation carries Datalog programs to the powerful framework of BESs, which have been widely used for verification of industrial critical systems, and for which many efficient resolution algorithms exist. We have also presented some experimental results which show that the presented transformation pays off in practice. Additionally, we have briefly discussed how we have improved this transformation, showing the progress reached by our approximation.

Recently, BESs with typed parameters [Mat98], called PBES, have been successfully used to encode several hard verification problems such as the first-

order value-based modal μ-calculus model-checking problem [MT08], and the equivalence checking of various bisimulations [CPvW07] on (possibly infinite) labeled transition systems. However, until our work in [AFJV09d], PBESs had not been previously used to compute complex interprocedural program analyses involving dynamically created objects. The work that is most closely related to our BES-based analysis approach is [LS98], where Dependency Graphs (DGs) are used to represent satisfaction problems, including propositional Horn Clause satisfaction and BES resolution. A linear time algorithm for propositional Horn Clause satisfiability is described in terms of the least solution of a DG equation system. This corresponds to an alternation-free BES, which can only deal with propositional logic problems. The extension of Liu and Smolka's work [LS98] to Datalog query evaluation is not straightforward. This is testified by the encoding of data-based temporal logics in equation systems with parameters in [MT08], where each boolean variable may depend on multiple data terms. DGs are not sufficiently expressive to represent such data dependencies on each vertex. Hence, it is necessary to work at a higher level, on the PBES representation.

A very efficient Datalog program analysis technique based on binary decision diagrams (BDDs) is available in the BDDBDDB system [WACL05], which scales to large programs and is competitive w.r.t. the traditional (imperative) approach. The computation is achieved by a fixpoint computation starting from the everywhere false predicate (or some initial approximation based on Datalog facts). Datalog rules are then applied in a bottom-up manner until saturation is reached so that all the solutions that satisfy each relation of a Datalog program are exhaustively computed. These sets of solutions are then used to answer complex formulas. In contrast, our approach focuses on demand-driven techniques to solve the considered query with no *a priori* computation of the derivable atoms. In the context of program analysis, note that all program updates, like pointer updates, might potentially be inter-related, leading to an exhaustive computation of all results. Therefore, improvements to top-down evaluation are particularly important for program analysis applications. Recently, Zheng and Rugina [ZR08] showed that demand-driven CFL-reachability with worklist algorithm compares favorably with an exhaustive solution. Our technique to solve Datalog programs based on local BES resolution goes in the same direction and provides a novel approach to demand-driven program analyses almost for free.

4

From Datalog to Rewriting Logic

With the aim to achieve higher expressiveness for static-analysis specification, in this chapter we present a translation of Datalog into a powerful and highly extensible framework, namely, the *Rewriting Logic* (Rwl), which is efficiently implemented in the high-performance language Maude [CDE+07]. The expressiveness of a specification language is important in order to naturally model analyses involving advanced features of programming languages that would otherwise need complex encodings or *ad-hoc* extensions. One example of a feature that complicates the specification of analyses is *reflection*, i.e., the ability to manipulate running program code at execution time. We benefit from the fact that *Rewriting Logic* is a reflective logic to express analyses with *reflective* features in a natural way.

Since the operational principles of logic programming (Datalog) and functional programming (*Rewriting Logic*), namely, *resolution* and *term rewriting*, share some similarities [Han94], many proposals exist for transforming logic programs into term rewriting systems [Mar94, Red84, SKGST07]. However, no transformation in the literature tackles the problem of efficiently encoding logic (Datalog) programs containing a huge amount of facts in a rewriting-based infrastructure such as Maude. As a result, we developed a new transformation suited to our needs. Because efficiency does matter in the context of Datalog-based program analysis, our proposed transformation is the result of an iterative process that is aimed at optimizing the running time of the transformed program. The basic idea of the translation is to automatically compile Datalog clauses into deterministic equations, having queries and answers consistently represented as terms so that the query is evaluated by reducing its term representation into a *constraint set* that represents the answers.

This chapter summarizes how Datalog queries can be solved with *Rewriting*

Logic by means of a *non-moded* transformation that preserves the computed answers behavior and supports reflective features. In Section 4.1, we first informally illustrate how a RWL *term rewriting system* can be obtained from a Datalog program in an automatic way. Section 4.2 describes the transformation in depth. Section 4.3 illustrates how our transformation can implement an analysis involving *reflection* in a declarative way. Section 4.4 presents the prototype developed to evaluate Datalog queries by means of our transformation, called DATALAUDE. Section 4.5 evaluates the efficiency of our prototype by comparing it to other Datalog solvers. Finally, Section 4.6 discusses the related work and concludes the chapter.

4.1 From Datalog to RWL.

Let us recall the simplified version of the Andersen points-to analysis that we introduced in Figure 3.1 (Chapter 3):

```
vP0(p,o1).
vP0(q,o2).
assign(r,q).
assign(w,r).
vP(V,H) :- vP0(V,H).
vP(V,H) :- assign(V,V2), vP(V2,H).
```

We use this example to illustrate the transformation. We first show how values, variables and answers are represented in Maude. Then, the resulting Maude program is presented by showing how each Datalog clause in the example program is transformed into RWL rules.

Datalog answers are expressed as equational *constraints* that relate the variables of the queries to values of the problem domain. Values are represented as *ground terms* of sort `Constant` that are constructed by means of Maude *Quoted Identifiers* (`Qid`s). Since logical variables cannot be represented with rewriting rule variables because of their dual input-output nature, we give a representation for them as ground terms of sort `Variable` by means of the overloaded `vrbl` constructor. A `Term` is either a `Constant` or a `Variable`. These elements are represented in Maude as follows:

```
sorts Variable Constant Term .
subsort Variable Constant < Term .
subsort Qid < Constant .

op vrbl :  Term -> Variable [ctor] .
```

In our formulation, answers are recorded within the term that represents the ongoing partial computation of the Maude program. Thus, we represent a

(partial) answer for the original Datalog query as a conjunction of equational constraints (called answer constraints) that represent the substitution of (logical) variables by (logical) constants that are incrementally computed during the program execution. We define the sort `Constraint` whose values represent single answers for a Datalog query as follows:

```
sort Constraint .

op _=_ :  Term       Constant   -> Constraint .
op T   :                        -> Constraint .
op F   :                        -> Constraint .
op _,_ :  Constraint Constraint -> Constraint [assoc comm id:  T] .

eq F, C:Constraint = F .                       --- Zero element
```

Constraints are constructed by the conjunction (`_,_`) of solved equations of the form `T:Term=C:Constant`, the *false* constraint `F`, or the *true* constraint `T`.[1] Note that the conjunction operator `_,_` obeys the laws of associativity and commutativity. `T` is defined as the identity of `_,_`, and `F` is used as the zero element. Unification of expressions is performed by combining the corresponding answer constraints and checking the satisfiability of the compound. Simplification equations are introduced in order to simplify trivial constraints by reducing them to `T`, or to detect inconsistencies (unification failure) so that the whole conjunction can be drastically replaced by `F`, as shown in the following code excerpt:

```
var Cst Cst1 Cst2 :  Constant .
var V : Variable .

eq (V = Cst)  , (V = Cst)  = (V = Cst) , T . --- Idempotence
eq (V = Cst1) , (V = Cst2) = F [owise] .     --- Unsatisfiability
```

In our setting, a failing computation occurs when a query is reduced to `F`. If a query is reduced to `T`, then the original (ground) query is proved to be satisfiable. On the contrary, if the query is reduced to a set of solved equations, then the computed answer is given by a substitution $\{x_1/t_1, \ldots, x_n/t_n\}$ that is expressed as an equation set in solved form by the computed normal form x_1 `=` t_1 `,` `...` `,` x_n `=` t_n.

Since equations in Maude are run deterministically, all the non-determinism of the original Datalog program has to be embedded into the term under reduction. This means that we need to carry all the possible (partial) answers at a given execution point. To this end, we introduce the notion of *set of answer constraints*, and we define a new sort called `ConstraintSet` as follows:

[1] The actual transformation defines a more complex hierarchy of sorts in order to obtain simpler equations and improve performance which we discuss in Section 4.2.

```
sorts ConstraintSet .
subsort Constraint < ConstraintSet .

op _;_ : ConstraintSet ConstraintSet -> ConstraintSet [assoc comm id: F] .
```

The set of constraints is constructed as the (possibly empty) disjunction `_;_` of accumulated constraints. The disjunction operator `_;_` obeys the laws of associativity and commutativity and is also given the identity element `F`.

Now we are ready to show how the predicates are transformed. Predicates are naturally expressed as functions (with the same arity) whose codomain is the `ConstraintSet` sort. They will be reduced to the set of constraints that represent the satisfiable instantiations of the original query. The functions that represent the three predicate of our running example are declared in Maude by providing the following signature:

```
ops vP vP0 assign :  Term Term -> ConstraintSet .
```

In order to incrementally add new constraints throughout the program execution, we define the composition operator `x` for constraint sets as follows:

```
op _x_ :  ConstraintSet ConstraintSet -> ConstraintSet [assoc] .
```

The composition operator `x` allows us to combine (partial) solutions of the subgoals in a clause body.

Let us now exemplify the transformation by evaluating different types of queries in our running example. Example 16 illustrates how *extensional* predicates are transformed, while Example 17 does so for the case of the *intensional* ones.

Example 16

The execution of the Datalog query `:- vP0(p,Y)` on the program in Figure 3.1, yields the solution {`Y/o1`}. Here, `vP0` is a predicate defined only by facts, so the answers to the query represent the variable instantiations as given by the existing facts. Thus, we would expect the query's RWL representation `vP0('p, vrbl('Y))` to be reduced to the `ConstraintSet` (with just one constraint) `vrbl('Y) = 'o1`. This is accomplished by representing facts according to the following equation pattern:

```
var T0 T1 :  Term .

eq vP0(T0,T1) = (T0 = 'p , T1 = 'o1) ; (T0 = 'q , T1 = 'o2) .
eq assign(T0,T1) = (T0 = 'r , T1 = 'q) ; (T0 = 'w , T1 = 'r) .
```

The right-hand side of the RWL equation that is used to represent the facts that define a given predicate (in the example `vP0` and `assign`) consists of

the set of constraints that express the satisfiable instantiations of the original predicate. As it can be observed, arguments are propagated to the constraints, thus allowing the already mentioned equational unification and simplification process on the constraints to happen.

For the considered goal, the reduction of the transformed Datalog query `vP0('p, vrbl('Y))` proceeds as follows:

```
vP0('p,vrbl('Y))
  → ('p = 'p , vrbl('Y) = 'o1) ; ('p = 'q , vrbl('Y) = 'o2)
  →* (T , vrbl('Y) = 'o1) ; (F , vrbl('Y) = 'o2)
  →* vrbl('Y) = 'o1 ; F
  → vrbl('Y) = 'o1
```

Let us proceed with the case of transforming *intensional* predicates.

Example 17

Let us consider the Datalog query `:- vP(V,o2)`, whose execution in the Andersen program delivers the solutions {`V/q,V/r,V/w`}. Thus, we expect the term `vP(vrbl('V),'o2)` to be reduced to the set of constraints `(vrbl('V) = 'q) ; (vrbl('V) = 'r) ; (vrbl('V) = 'w)`. In this case, `vP` is a predicate defined by clauses, so the answers to the query are the disjunction of the answers provided by all the clauses that define it. This is represented in RWL by introducing auxiliary functions to separately compute the answers for each clause, and the equation to join them, which can be expressed as follows:

```
op vP-clause-1 vP-clause-2 :  Term Term -> ConstraintSet .
var V H : Term .
eq vP(V,H) = vP-clause-1(V , H) ; vP-clause-2(V , H) .
```

In order to compute the answers delivered by a clause, we look for the satisfiable instantiations of its body's subgoals. In our translation, we explore the possible instantiations from the leftmost subgoal to the rightmost one. For example, in:

```
vP(V,H) :- vP0(V,H).
```

we would explore the instantiations of the only subgoal `vP0`. This can directly be done by reducing the term representing the head of the clause `vP(V,H)` to the term representing the subgoal `vP0(V,H)`. On the contrary, in:

```
vP(V,H) :- assign(V,V2), vP(V2,H).
```

we would first explore the instantiations for `assign` and then those for `vP`. In order to do that, we create a different (auxiliary) unraveling function for each subgoal to impose this left-to-right exploration.

As shown in the following code excerpt, the first Datalog clause of the leading program example can be transformed without using unraveling functions. For the second Datalog clause (with two subgoals in its body) one unraveling function `unrav` is needed in order to force the reduction of the first subgoal.

```
op vrbl-V2 :  Term          Term     -> Variable .
op unrav   :  ConstraintSet TermList -> ConstraintSet .

eq vP-clause-1(V,H) = vP0(V,H) .
eq vP-clause-2(V,H) = unrav( assign(V, vrbl-V2(V,H)) , V H ) .
```

Each unravelling function computes the partial answer derived from the subgoal it represents and the previous ones, and propagates that partial answer to the subsequent unraveling function. Additionally, existentially quantified variables that occur only in the body of original Datalog clauses, e.g., `V2`, are introduced by using a ground representation that is parameterised with the corresponding call pattern in order to generate fresh variables, e.g., `vrbl-V2(V,H)` in the example.

The `unrav` function has two arguments: a `ConstraintSet`, which is the first (reduced) subgoal (the original subgoal `assign(V,V2)` in this case); and the `V H` call pattern. This function is defined as follows:

```
var Cnt :  Constant .
var TS  :  TermList .
var C   :  Constraint .
var CS  :  ConstraintSet .

eq unrav( ( (vrbl-V2(V,H) = Cnt , C) ; CS ) , V H ) =
    ( vP(Cnt,H) x (vrbl-V2(V,H) = Cnt , C) ) ; unrav( CS , V H ) .
eq unrav( F , TS ) = F .
```

This `unrav` function takes a set of partial answers as its first argument. It requires the partial answers to be in equational solved form, which is enforced by the (default) *innermost* reduction strategy used by Maude, thus ensuring the left-to-right execution of the goals. The second argument is the call pattern of the translated clause and serves to reference the introduced existentially quantified variables. The propagated call pattern is represented as a `TermList`, that is, a juxtaposition (__ operator) of `Terms`. The two `unrav` equations (recursively) combine each (partial) answer obtained from the first subgoal with every (partial) answer computed from the (instantiated) subsequent subgoal.

As an example, consider again the Datalog query `:- vP(V,o2)`. We undertake each possible query reduction by using the equations above. Given the size of the execution trace, we use the following abbreviations: V stands for `vrbl('V)`, `vPci` for `vP-clause-i`, and `V2-V-H` for `vrbl-V2(V,H)`.

```
vP(V,'o2 )
  → vPc1(V,'o2) ; vPc2(V,'o2)
  →* vP0(V,'o2) ; unrav( assign(V,V2-V-o2) , V 'o2 )
  →* ( (V = 'p , 'o2 = 'o1) ; (V = 'q , 'o2 = 'o2) )
     ; unrav( ((V = 'r , V2-V-o2 = 'q) ; (V = 'w , V2-V-o2 = 'r)) ,
              V 'o2 )
  →* (F ; (V = 'q , T))
     ; ( vP('q,'o2) x (V = 'r , V2-V-o2 = 'q) )
     ; unrav( (V = 'w , V2-V-o2 = 'r) , V 'o2 )
  →* (V = 'q)
     ; ( (vPc1('q,'o2) ; vPc2('q,'o2)) x (V = 'r , V2-V-o2 = 'q) )
     ; ( vP('r,'o2) x (V = 'w , V2-V-o2 = 'r) ) ; unrav( F , V 'o2 )
  ⋮
  →* (V = 'q) ; (V = 'r) ; (V = 'w)
```

As it can be seen, the evaluation of a Datalog query is naturally transformed to the process of reducing the equational version of the query into its normal form solutions.

4.2 A complete Datalog to RWL transformation

As explained above, we are interested in computing all answers for a given query by means of term rewriting. A naïve approach is to translate the Datalog clauses into Maude rules, and then use the `search`[2] order to mimic all possible executions of the original Datalog program. However, in the context of program analysis with a huge number of facts, this approach results in poor performance [AFJV09c]. This is because *rules* are handled non-deterministically in Maude whereas *equations* are applied deterministically [CDE+07].

In the following, given a Datalog program $\mathcal{R}$ and a query q, we assume a top-down approach and use SLD-resolution to compute the set of answers of q in $\mathcal{R}$. Given the successful derivation $\mathcal{D} \equiv q \Rightarrow^{\theta_1}_{SLD} q_1 \Rightarrow^{\theta_2}_{SLD} \ldots \Rightarrow^{\theta_n}_{SLD} \square$, the answer computed by $\mathcal{D}$ is the composed substitution $\theta_1\theta_2\ldots\theta_n$ restricted to the variables occurring in q.

[2] Intuitively, `search` $t \rightarrow t'$ explores the whole rewriting space from the term t to any other terms that match t' [CDE+07].

In this section, we formulate a complete representation in Maude of the Datalog computed answers, and then, we give a formal description of our equation-based transformation together with proofs of its correctness and completeness.

Answer representation. Let us first introduce our representation of variables and constants of a Datalog program as *ground terms* of a given sort in Maude. We define the sorts `Variable` and `Constant` to specifically represent the variables and constants of the original Datalog program in Maude, whereas the sort `Term` (resp. `TermList`) represents Datalog terms (resp. lists of terms that are built by simple juxtaposition):

```
sorts Variable Constant Term TermList .
subsort Variable Constant < Term .
subsort Term < TermList .

op __ :  TermList TermList -> TermList [assoc] .
op nil :  -> TermList .
```

For instance, `T1 T2` represents the list of terms `T1` and `T2`. In order to construct the elements of the `Variable` and `Constant` sorts, we introduce two constructor symbols: Datalog constants are represented as Maude *Quoted Identifiers* (`Qid`s), whereas logical variables are encoded in Maude by means of the constructor symbol `vrbl`. These constructor symbols are specified in Maude as follows:

```
subsort Qid < Constant .              --- Every Qid is a Constant

op vrbl :  Qid -> Variable [ctor] .
op vrbl :  Term Term -> Variable [ctor] .
```

The last line of the above code excerpt allows us to build variable terms of the form `vrbl(T1,T2)` where both `T1` and `T2` are `Term`s. This is used to ensure that the ground representation in Maude for existentially quantified variables that appear in the body of Datalog clauses is unique to the whole Maude program.

By using ground terms to represent variables, we still lack a way to collect the answers for an output variable. In our formulation, answers are stored within the term that represents the ongoing partial computation of the Maude program. Thus, we represent a (partial) answer for the original Datalog query as a sequence of equations (called answer constraint) that expresses the substitution of (logical) variables by (logical) constants that is performed during the program execution. We define the sort `Constraint` to represent a single answer for a Datalog query, but we also define a hierarchy of subsorts (e.g.,

the sort `FConstraint` at the bottom of the hierarchy represents inconsistent solutions) that allows us to identify the *inconsistent* as well as the *trivial* constraints (e.g., `Cte = Cte`) whenever possible. This hierarchy allows us to simplify constraints as soon as possible and to improve performance. The Maude code that implements this constraint-based infrastructure is as follows:

```
sorts Constraint EmptyConstraint NonEmptyConstraint .
subsort EmptyConstraint NonEmptyConstraint < Constraint .
sorts TConstraint FConstraint .
subsort TConstraint FConstraint < EmptyConstraint .

op T : -> TConstraint .
op F : -> FConstraint .
op _=_ :  Term Constant -> NonEmptyConstraint .
op _,_ :  Constraint Constraint -> Constraint   [assoc comm id:  T] .
op _,_ :  FConstraint Constraint
          -> FConstraint                                [ditto] .
op _,_ :  TConstraint TConstraint
          -> TConstraint                                [ditto] .
op _,_ :  NonEmptyConstraint TConstraint
          -> NonEmptyConstraint                         [ditto] .
op _,_ :  NonEmptyConstraint FConstraint
          -> FConstraint                                [ditto] .
op _,_ :  NonEmptyConstraint NonEmptyConstraint
          -> NonEmptyConstraint                         [ditto] .
```

As we have said before, a query reduced to `T` represents a successful computation, whereas a failing computation is represented by a final `F` term. Note that the conjunction operator `_,_` has identity element `T` and obeys the laws of associativity and commutativity. The properties of associativity, commutativity and identity element can be easily expressed by using `ACU` attributes in Maude, thus simplifying the equational specification and achieving better efficiency. Other properties of the constraint-builder operators must be expressed with equations: for example, we express the idempotency property of the operator `_,_` by a specific equation on variables from the `NonEmptyConstraint` subsort `NEC`. Moreover, in order to keep information consistent and without redundancy, additional simplification equations are automatically applied. These equations make every inconsistent constraint collapse into an `F` value, and simplify every redundant or trivial constraint. The Maude code that implements these reductions is:

```
var Cte Cte1 Cte2 :  Constant .
var NEC : NonEmptyConstraint .
var V : Variable .
eq (Cte = Cte) = T .                    --- Simplification
eq (Cte1 = Cte2) = F [owise] .          --- Unsatisfiability
eq NEC,NEC = NEC .                      --- Idempotence
eq F,NEC = F .                          --- Zero element
eq F,F = F .                            --- Simplification
eq (V = Cte1),(V = Cte2) = F [owise] .--- Unsatisfiability
```

In order to embed the non-determinism of the original Datalog program into the deterministic execution of equations in Maude, we introduce the notion of *set of answer constraints*. This construction allows us to carry on all the possible (partial) answers at a given execution point. The sort `ConstraintSet` represents *sets of answer constraints* and is implemented as follows:

```
sorts ConstraintSet EmptyConstraintSet NonEmptyConstraintSet .
subsort EmptyConstraintSet NonEmptyConstraintSet < ConstraintSet .
subsort NonEmptyConstraint TConstraint < NonEmptyConstraintSet .
subsort FConstraint < EmptyConstraintSet .

op _;_ :  ConstraintSet ConstraintSet
          -> ConstraintSet                    [assoc comm id:  F] .
op _;_ :  NonEmptyConstraintSet ConstraintSet
          -> NonEmptyConstraintSet                          [ditto] .

var NECS : NonEmptyConstraintSet .

eq NECS ; NECS = NECS .      --- Idempotence
```

It is easy to grasp the intuition behind the different sorts and the subsort hierarchies in the above fragment of Maude code. The operator `_;_` represents the disjunction of constraints. It is an associative and commutative operator that has `F` as its identity element. We express the idempotency property of the operator `_;_` by a specific equation on variables from the `NonEmptyConstraintSet` subsort.

In order to incrementally add new constraints throughout the program execution, we define the composition operator `x` as follows:

```
op _x_ :  ConstraintSet ConstraintSet -> ConstraintSet [assoc] .

var CS            :  ConstraintSet .
var NECS1 NECS2   :  NonEmptyConstraintSet .
var NEC NEC1 NEC2 :  NonEmptyConstraint .

eq F x CS = F .                                  --- L-Zero element
eq CS x F = F .                                  --- R-Zero element
eq F x F = F .                                   --- Double-Zero
eq NEC1 x (NEC2 ; CS)
          = (NEC1 , NEC2) ; (NEC1 x CS) .        --- L-Distributive
eq (NEC ; NECS1) x NECS2
          = (NEC x NECS2) ; (NECS1 x NECS2) .    --- R-Distributive
```

The transformation of clauses. Let P be a Datalog program defining predicate symbols $p_1 \dots p_n$ where we denote by ar_i the arity of p_i. Before describing the transformation process, we introduce some auxiliary notations. $|p|$ is the number of facts or clauses defining the predicate symbol p. Following the Datalog standard, without loss of generality we assume that a predicate p is defined only by facts, or only by clauses [vL90].

Let us start by describing the case when predicates are defined by facts. We transform the whole set of facts that define a given predicate symbol p into a single equation by means of a disjunction of answer constraints. Formally, for each p_i with $1 \leq i \leq n$ that is defined in the Datalog program only by facts, we write the following snippet of Maude code, where the symbol $c_{i,j,k}$ is the k-th argument of the j-th fact defining the predicate symbol p_i:

```
var T_{i,1} ... T_{i,ar_i} :  Term .
eq p_i(T_{i,1}, ... ,T_{i,ar_i}) = (T_{i,1} = c_{i,1,1}, ... , T_{i,ar_i} = c_{i,1,ar_i}) ; ...
                                 ; (T_{i,1} = c_{i,|p_i|,1}, ... , T_{i,ar_i} = c_{i,|p_i|,ar_i}) .
```

Let us now consider the transformation for Datalog clauses with non-empty body. Similarly to the previous case, the proposed transformation combines in a single equation the disjunction of the calls to all functions representing the different clauses for the considered predicate symbol p. For each p_i with $1 \leq i \leq n$ defined only by clauses with non empty body, we have the following piece of code:

```
var T_{i,1} ... T_{i,ar_i} :  Term .
eq p_i(T_{i,1}, ... ,T_{i,ar_i}) = p_{i,1}(T_{i,1}, ... ,T_{i,ar_i}) ; ...
                                 ; p_{i,|p_i|}(T_{i,1},...,T_{i,ar_i}) .
```

Each call to a function $\mathtt{p}_{i,j}$ with $1 \leq j \leq |p_i|$ produces the answers that are computed by the j-th clause of the considered predicate symbol. Now we

proceed to define how each of these clauses is transformed. Notation $\tau^a_{i,j,s,k}$ denotes the name of the variable or constant symbol that appears in the k-th argument of the s-th subgoal in the j-th clause defining the i-th predicate of the original Datalog program. When $s = 0$, then the function refers to the arguments in the head of the clause.

Let us start by considering the case when we just have one subgoal in the body of the clause. We define the function $\tau^p_{i,j,s}$, which returns the predicate symbol that appears in the s-th subgoal of the j-th clause that defines the i-th predicate in the Datalog program. For each clause having just one subgoal, we formulate the following transformation:

$$\texttt{eq } \texttt{p}_{i,j}(\tau^a_{i,j,0,1},\ldots,\tau^a_{i,j,0,ar_i}) = \tau^p_{i,j,1}(\tau^a_{i,j,1,1},\ldots,\tau^a_{i,j,1,r}) \ .$$

In our formalization, r is a shorthand that denotes an arity. When r is used to index the arguments of a certain predicate p it denotes the arity of p, e.g., in the example above, r is the arity of $\tau^p_{i,j,1}$.

In the case when more than one subgoal appears in the body of the clause, we want to impose a left-to-right evaluation strategy. We use auxiliary functions that are defined with specific patterns to force such an execution order[3]. Specifically, we impose that a subgoal cannot be invoked until the subgoal variables that also occur in previous subgoals have been instantiated. We call these variables *linked* variables. Let us first formalize the auxiliary notions that we need for expressing our transformation.

Definition 18 (linked variable) *A variable is called* linked variable *if and only if it occurs in two or more subgoals of the clause's body.*

Definition 19 (function linked) *Let C be a Datalog clause. Then the function* linked*(C) is the function that returns the list of pairs containing a linked variable in the first component, and the list of positions where such a variable occurs in the body of the clause in the second component*[4]*.*

Example 20

For example, given the Datalog clause

C = `p(X1,X2) :- p1(X1,X3), p2(X1,X3,X4), p3(X4,X2).`

we have that

linked(C) = `[(X1,[1.1,2.1]),(X3,[1.2,2.2]),(X4,[2.3,3.1])]`

[3]Conditional equations could also be used to impose left-to-right evaluation, but in practice they suffer from poor performance as revealed by our experiments.

[4]Positions extend to goals in the natural way.

Given a goal g, we define the notion of *relevant* linked variables for a given subgoal of g as the linked variables of the subgoal that also appear in a preceding subgoal in g.

Definition 21 (Relevant linked variables) *Given a clause C and an integer number n, we define the function* relevant *that returns the variables that are shared by the n-th subgoal and some preceding subgoal:*

$$\mathit{relevant}(n, C) = \{X \mid (X, LX) \in \mathsf{linked}(C), \exists i, j, m \,/\, n.i \in LX, m.j \in LX, m < n\}$$

Similarly to [SKGST07], note that we are not marking the input/output positions of the predicates, as opposed to more traditional transformations. We are just identifying the variables whose values must be propagated in order to evaluate the subsequent subgoals following the evaluation strategy.

Now we are ready to address the problem of transforming a clause with more than one subgoal (and maybe existentially quantified variables) into a set of equations. Intuitively, the main function initially calls to an auxiliary function that undertakes the execution of the first subgoal. We have as many auxiliary functions as subgoals in the original clause minus one. Also, in the right-hand side (*rhs*s) of the auxiliary function definitions, the execution order of the successive subgoals is implicitly controlled by passing the results of each subgoal as a parameter to the subsequent function call.

Let the function $\mathtt{p}_{i,j}$ generate the solutions calculated by the j-th clause of the predicate symbol p_i. We state that $\mathtt{ps}_{i,j,s}$ represents the auxiliary function that corresponds to the s-th subgoal of the j-th clause defining the predicate p_i. Then, for each clause, we have the following translation, where the variables $\mathtt{X}_1 \ldots \mathtt{X}_N$ of each equation are calculated by the function $\mathsf{relevant}(s, \mathit{clause}(i,j))$[5] and transformed into the corresponding Maude terms.

The equation for $\mathtt{p}_{i,j}$ below calls the first auxiliary function ($\mathtt{ps}_{i,j,2}$) that computes the (partial) answers for the second subgoal. This is done by first computing the answers for the first subgoal $\tau^p_{i,j,1}$ in its first argument. The second argument of the call to $\mathtt{ps}_{i,j,2}$ represents the list of terms in the initial predicate call that, together with the information retrieved from Definitions 19 and 21, allow us to correctly build the patterns and function calls during the transformation.

$$\begin{aligned}
&\texttt{eq } \mathtt{p}_{i,j}(\tau^a_{i,j,0,1}, \ldots, \tau^a_{i,j,0,ar_i}) \texttt{ = } \mathtt{ps}_{i,j,2}(\\
&\qquad\qquad \tau^p_{i,j,1}(\tau^a_{i,j,1,1}, \ldots, \tau^a_{i,j,1,r}), \\
&\qquad\qquad \tau^a_{i,j,0,1} \ \cdots \ \tau^a_{i,j,0,ar_i} \\
&\qquad) \ .
\end{aligned}$$

[5] The notation $\mathit{clause}(i,j)$ represents the j-th Datalog clause defining the predicate symbol p_i.

Then, for each auxiliary (unraveling) function, we declare as many constants as relevant variables are in the corresponding subgoal.

```
var C1 ... CN : Constant .
var NECS : NonEmptyConstraintSet .
```

$$\texttt{eq ps}_{i,j,s}\texttt{(NECS, T}_1\texttt{...T}_{ar_i}\texttt{) =} \quad (4.1)$$
$$\texttt{ps}_{i,j,s+1}\texttt{(ps'}_{i,j,s}\texttt{(NECS, T}_1\texttt{...T}_{ar_i}\texttt{), T}_1\texttt{...T}_{ar_i}\texttt{) .}$$

$$\texttt{eq ps}_{i,j,s}\texttt{(F, LL) = F .} \quad (4.2)$$

$$\texttt{eq ps'}_{i,j,s}\texttt{(((X}_1\texttt{=C}_1\texttt{,...,X}_N\texttt{=C}_N\texttt{, C) ; CS), T}_1\texttt{...}T_{ar_i}\texttt{) =} \quad (4.3)$$
$$\texttt{(}\ (\tau^p_{i,j,s}(\tau^v_{i,j,s,1},\ldots,\tau^v_{i,j,s,r})\texttt{[X}_1\texttt{\textbackslash C}_1\texttt{,...,X}_N\texttt{\textbackslash C}_N\texttt{]) x}$$
$$\texttt{(X}_1\texttt{=C}_1\texttt{,...,X}_N\texttt{=C}_N\texttt{, C)}$$
$$\texttt{) ; ps'}_{i,j,s}\texttt{(CS, T}_1\texttt{...T}_{ar_i}\texttt{) .}$$

$$\texttt{eq ps'}_{i,j,s}\texttt{((T ; CS), T}_1\texttt{...T}_{ar_i}\texttt{) =} \quad (4.4)$$
$$\tau^p_{i,j,s}(\tau^v_{i,j,s,1},\ldots,\tau^v_{i,j,s,r})\texttt{ ; ps'}_{i,j,s}\texttt{(CS, T}_1\texttt{...T}_{ar_i}\texttt{) .}$$

$$\texttt{eq ps'}_{i,j,s}\texttt{(F , LL) = F .} \quad (4.5)$$

The left hand side of the Equation (4.3) for this auxiliary function is defined by using patterns that adjust the relevant variables to the values already computed by the execution of a preceding subgoal. Note that we may have more assignments in the constraint, which is represented by `C`, and that we may have more possible solutions in `CS`. The Equations (4.3), (4.4) and (4.5) for $\texttt{ps'}_{i,j,s}$ take each possible (partial) solution and combine it with the solutions given by the s-th subgoal in the clause (whose predicate symbol is $\tau^p_{i,j,s}$). Note that we propagate the instantiation of the relevant variables by means of a suitable substitution.

The equation for the last subgoal in the clause is slightly different, since we do not need to recursively invoke the auxiliary equation $\texttt{ps'}_{i,j,s}$. Assuming that g denotes the number of subgoals in a clause, we define

```
eq ps_{i,j,g}(((X1=C1,...,XN=CN, C) ; CS) , T1...T_{ar_i}) =
   ((τ^p_{i,j,g}(τ^v_{i,j,g,1},...,τ^v_{i,j,g,r})[X1\C1,...,XN\CN]) x (X1=C1,...,XN=CN, C))
   ; ps_{i,j,g}(CS , T1...T_{ar_i}) .
eq ps_{i,j,g}((T ; CS) , T1...T_{ar_i}) =
   τ^p_{i,j,g}(τ^v_{i,j,g,1},...,τ^v_{i,j,g,r}) ; ps_{i,j,g}(CS , T1...T_{ar_i}) .
eq ps_{i,j,g}(F , LL) = F .
```

Query representation. Finally, we define the transformation for a Datalog query $q(X_1,\ldots,X_n)$ (where X_i, $1{\leq}i{\leq}n$ are Datalog variables or constants) as the Maude function $\texttt{q}(\tau^q_1,\ldots,\tau^q_n)$, where τ^q_i, $1{\leq}i{\leq}n$ is the result of applying the proposed transformation to the corresponding X_i.

Correctness of the transformation.

The transformation from Datalog programs into Maude programs defined so far satisfies that the normal form computed for the term that stands for a Datalog query represents the set of computed answers for a query of the original Datalog program. Below we demonstrate that the transformation is sound and complete w.r.t. the observable of computed answers.

We first introduce some auxiliary notation. Let `CS` be a `ConstraintSet` of the form $\texttt{C}_1$ `;` $\texttt{C}_2$ `; ...;` $\texttt{C}_n$ where each $\texttt{C}_i$, $i \geq 1$ is a `Constraint` in normal form ($\texttt{C}_1$ `=` $\texttt{Cte}_1$`,...,`$\texttt{C}_m$ `=` $\texttt{Cte}_m$), and let V be a list of variables. We write $\texttt{C}_i|_V$ to the restriction of the constraint $\texttt{C}_i$ to the variables in V. We extend the notion to sets of constraints in the natural way, and denote it as $\texttt{CS}|_V$. Given two terms t and t', we write $t \rightarrow_S^* t'$ when there exists a rewriting sequence from t to t' in the Maude program S. Also, $var(t)$ is the set of variables occurring in t.

Now we define a suitable notion of *(rewriting) answer constraint*:

Definition 22 (Answer Constraint Set) *Given a* Maude *program S that is the result of our transformation and an input term t, we say that the* answer constraint set *computed by $t \rightarrow_S^* \texttt{CS}$ is $\texttt{CS}|_{var(t)}$.*

There is a natural isomorphism between the equational constraint C and an idempotent substitution $\theta = \{X_1/C_1, X_2/C_2, \ldots, X_n/C_n\}$, which is given by the following: `C` is equivalent to θ *iff* $(\texttt{C} \Leftrightarrow \hat{\theta})$, where $\hat{\theta}$ is the equational representation of θ. By abuse, given a disjunction `CS` of equational constraints and a set of idempotent substitutions $(\Theta = \cup_{i=1}^n \theta_i)$, we define $\Theta \equiv$ `CS` *iff* $\texttt{CS} \Leftrightarrow \bigvee_{i=1}^n \hat{\theta}_i$.

Next, we prove that, for a given query in a Datalog program P, each answer constraint set computed for the corresponding input term in the transformed Maude program is equivalent to the set of computed answers for P.

Theorem 23 (Correctness and completeness) *Consider a Datalog program P together with a query q. Let $\mathcal{T}(P)$ be the corresponding transformed* Maude *program, and let $\mathcal{T}_g(q)$ be the corresponding transformed input term. Let Θ be the set of computed answers of P for the query q, and let $\texttt{CS}|_{var(\mathcal{T}_g(q))}$ be the answer constraint set computed by $\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)}^* \texttt{CS}$. Then,*

$$\Theta \equiv \texttt{CS}|_{var(\mathcal{T}_g(q))}.$$

Proof of Theorem 23

($\Leftarrow$) We proceed by induction on both the structure of the clauses and the length of the computations.

In order to demonstrate the claim, we must prove that, if $\mathcal{T}_g(q) \rightarrow^{!}_{\mathcal{T}(P)} \mathtt{CS}$, then for every $\mathtt{C}$ in the answer constraint set $\mathtt{CS}$, there exists a computed answer θ for q and P such that $\mathtt{C}|_{var(\mathcal{T}_g(q))} \equiv \theta$.

Let us first consider the case when q is defined only by facts.

By the definition of our transformation, when the predicate symbol (of arity m) of the query q is defined by facts[6] , there exists an equation in $\mathcal{T}(P)$, whose left hand side has the form $\mathtt{q}(T_1, \ldots, T_m)$, that rewrites to an answer constraint set which contains as many answer constraints as the number of facts that define the predicate in the Datalog program. Also by definition, each answer constraint corresponds to one (ground) fact in the Datalog program that instantiates each argument of the predicate to the appropriate constant.

In this case, the rewriting sequence for the initial term $\mathcal{T}_g(q)$ is

$$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathtt{C}_1 \ ; \ \ldots \ ; \mathtt{C}_n \rightarrow^{!}_{\mathcal{T}(P)} \mathtt{C}_v \ ; \ \ldots \ ; \mathtt{C}_w$$

where n is the number of facts defining the Datalog predicate and $v, \ldots, w \in \{1, \ldots, n\}$. Each answer constraint occurring in $\mathtt{C}_1 \ ; \ \ldots \ ; \mathtt{C}_n$ comes up from one Datalog fact. The second part of the sequence is the simplification for the union operator `;` and for the constraint constructors. The simplification consists in removing duplicate elements and collapsing inconsistent constraints to `F`. The inconsistent constraints appear when a single variable is equaled to two different values or when two different constants are equaled. This case may occur when a query is partially (or totally) instantiated and/or when it has a variable that appears multiple times. In this case, all the answer constraints that are incompatible with the passed value are collapsed into `F`. In the Datalog setting, this corresponds with failing of the attempt to unify the query with the facts that generate these answers. It is easy to observe that the Datalog resolution is able to compute each of these consistent solutions.

Now we consider the case when q is defined by n clauses with non-empty bodies. By definition of our transformation, the initial term rewrites as follows:

$$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathtt{q_1}(T_1, \ldots, T_m) \ ; \ \ldots \ ; \mathtt{q_n}(T_1, \ldots, T_m).$$

Also by definition, each function $\mathtt{q}_i$ can be defined in our transformation in two different ways, depending on the number of subgoals in the clause represented by $\mathtt{q}_i$.

Let us consider the case of a clause having a single subgoal. Let the equation defining the function symbol $\mathtt{q}_i$ be

$$\mathtt{eq}\ \mathtt{q}_i(U_1, \ldots, U_m) = \mathtt{p}(V_1, \ldots, V_z)$$

[6] Remember that, in Datalog, predicates are defined by facts or by clauses but not by facts and clauses altogether.

where $U_1,\ldots,U_m$ and $V_1,\ldots,V_z$ are the terms in the Datalog clause. Therefore, many of them may coincide, and the set of variables in $V_1,\ldots,V_z$ subsumes the set of variables in $U_1,\ldots,U_m$ (remember that we are considering *safe* Datalog programs).

Hence, the rewriting sequence given by the equation shown above is as follows:

$$\mathtt{q}_i(T_1,\ldots,T_m) \rightarrow_{\mathcal{T}(P)} \mathtt{p}(W_1,\ldots,W_z)$$

Notice that p is a predicate symbol in the Datalog program that is also transformed. By inductive hypothesis, $\mathtt{p}(W_1,\ldots,W_z)$ rewrites to the set of its correct answer constraints $\mathtt{C'}_1 \ ; \ \ldots \ ; \ \mathtt{C'}w|_{var(\mathtt{p}(W_1,\ldots,W_z))}$. Since we are considering safe Datalog programs, we know that all the variables $T_1,\ldots,T_m$ occur in the arguments of the body subgoals and are thus in the set of variables $\{W_1,\ldots,W_z\}$. Therefore, the correct answer constraint for the query is $\mathtt{C'}_1 \ ; \ \ldots \ ; \ \mathtt{C'}_w|_{var(\mathtt{q}(T_1,\ldots,T_m))}$.

Let us now proceed with the general case when the clause body contains more than one subgoal. In this case, the rewriting sequence starts by rewriting to an auxiliary function s_2 which represents the execution of the second subgoal, after having reduced the first one (on the first argument of s_2). This is ensured by the operational semantics of Maude and the patterns in the definition of the auxiliary functions.

The second part of the sequence below corresponds to the computation of the first subgoal:

$$\begin{aligned} \mathtt{q}_i(T_1,\ldots,T_m) &\rightarrow_{\mathcal{T}(P)} s_2(\ \mathtt{p}(W_1,\ldots,W_z),\ T_1\ldots T_m\ ) \\ &\rightarrow^*_{\mathcal{T}(P)} s_2(\ \mathtt{C}_1 \ ; \ \ldots \ ; \ \mathtt{C}_w,\ T_1\ldots T_m\ ) \end{aligned}$$

By inductive hypothesis, the set $\mathtt{C}_1 \ ; \ \ldots \ ; \ \mathtt{C}_w$ contains correct answer constraints for $\mathtt{p}(W_1,\ldots,W_z)$. At this execution point, following the definition of our transformation, there are two possibilities depending on whether or not there are more subgoals. In the following, (Case 1) will refer to the case in which there are more subgoals, and (Case 2) to the case in which there are not. Let us assume without loss of generality that we are dealing with the i-th subgoal (function symbol s_i).

Case 1 In this case, the computation may proceed in two different ways:

1. There is no solution for $\mathtt{p}(W_1,\ldots,W_z)$; thus, the answer constraint set is F. In this case, the rewriting sequence is:

$$s_i(\ \mathtt{C}_1 \ ; \ \ldots \ ; \ \mathtt{C}_w,\ T_1\ldots T_m\ ) \rightarrow_{\mathcal{T}(P)} \mathtt{F}$$

Therefore, there exists no solution for the first subgoal and the computation of the query trivially fails, which corresponds with the Datalog resolution of $\mathtt{p}(W_1,\ldots,W_z)$.

2. Consider the case when there are w different answer constraints for $\mathtt{p}(W_1,\ldots,W_z)$. Following the definition of our transformation, the rewriting sequence is:

$$\begin{aligned} & s_i(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ) \\ & \quad \rightarrow_{\mathcal{T}(P)}\ s_{i+1}(\ s'_i(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ),\ T_1\ldots T_m\ ) \end{aligned}$$

Note that, in order to compute the answer constraints for the third subgoal (s_{i+1}), we first need to rewrite the second subgoal by reducing the redex s'_i that contains the partially accumulated answer constraint set. Depending on the form of this constraint set, we have three possible rewritings:

(a) The first answer constraint ($\mathtt{C}_1$) is `T` (which is a term of the `EmptyConstraint` sort); thus, the execution of the preceding subgoal (which is ground) computed no variable substitution:

$$\begin{aligned} & s_{i+1}(\ s'_i(\ \mathtt{T}\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ),\ T_1\ldots T_m\ ) \\ \rightarrow_{\mathcal{T}(P)}\ & s_{i+1}(\ \mathtt{q}(Q_1,\ldots,Q_z)\ ;\ s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ), \\ & \qquad T_1\ldots T_m\ ) \\ \rightarrow^*_{\mathcal{T}(P)}\ & s_{i+1}(\ \mathtt{C}'_1\ ;\ \ldots\ ;\ \mathtt{C}'_{w'}\ ;\ s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ),\ T_1\ldots T_m\ ) \end{aligned}$$

By inductive hypothesis, $\mathtt{C}'_1,\ldots,\mathtt{C}'_{w'}$ are correct answer constraints for the i-th subgoal (whose function symbol is `q`, given by the function τ^p of our transformation). Intuitively, this rewriting step represents the propagation of variable assignments to the subsequent subgoals. The recursive call in s'_i propagates not only the information from the set of answer constraints for the first subgoal, but also the call pattern. We will come back to this point of the proof after discussing the remaining cases.

(b) The first answer constraint is not `T`, generating the following rewriting sequence:

$$\begin{aligned} & s_{i+1}(\ s'_i(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ),\ T_1\ldots T_m\ ) \\ \rightarrow_{\mathcal{T}(P)}\ & s_{i+1}(\ \mathtt{q}(Q_1,\ldots,Q_z)[Q_j\backslash X_j,\ldots,Q_k\backslash X_k])\ \mathtt{x}\ \mathtt{C}_1\ ; \\ & \qquad s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ), \\ & \qquad T_1\ldots T_m\ ) \\ \rightarrow^*_{\mathcal{T}(P)}\ & s_{i+1}(\ (\mathtt{C}'_1\ ;\ \ldots\ ;\ \mathtt{C}'_{w'})\ \mathtt{x}\ \mathtt{C}_1\ ; \\ & \qquad s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ), \end{aligned}$$

$T_1 \ldots T_m$)

Note that, the substitution $\mathtt{q}(Q_1,\ldots,Q_z)[Q_j \backslash X_j,\ldots,Q_k \backslash X_k]$ is defined to replace each relevant variable X_j of $\mathtt{q}$ by its computed value, captured in the pattern of the *lhs* of the corresponding transformation equation. The constraints for these values are also in the computed answer $\mathtt{C}_1$. By inductive hypothesis, $\mathtt{C'}_1,\ldots,\mathtt{C'}_{w'}$ are correct answer constraints for the term $\mathtt{q}(Q_1,\ldots,Q_z)[Q_j \backslash X_j,\ldots,Q_k \backslash X_k]$. The operator $\mathtt{x}$ combines each solution of the second subgoal with the information in $\mathtt{C}_1$. Since we have propagated the shared information with the applied substitution before the subsequent reduction step, we know that the shared variables have the same value; thus, the new combined solutions are consistent for the conjunction of the two (or more) subgoals. Note that the only case when inconsistencies may arise (and be simplified) by the $\mathtt{x}$ operator is when both sets of answers contain an output variable and each one computes a different value for it. This inconsistent case is reduced to false, so no inconsistent answer constraint is carried on.

(c) There are no answer constraints to proceed with; thus, the first argument is $\mathtt{F}$ and the rewriting sequence is:

$$s_{i+1}(\ s'_i(\mathtt{F},\ T_1 \ldots T_m\ )),\ T_1 \ldots T_m\ ) \rightarrow_{\mathcal{T}(P)} s_{i+1}(\ \mathtt{F},\ T_1 \ldots T_m\ )$$

This last case is the base case for the recursion appearing in the two previous ones. By induction on the number of elements in the answer constraint set $\mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w$, we can see that the subterm $s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ )$ in the cases (a) and (b) is a smaller recursive call.

Hence, we are at the point in which we have computed all the accumulated answer constraints up to the i-th subgoal:

$$s_{i+1}(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_n,\ T_1 \ldots T_m\ )$$

Case 2 In this case, s_i is the last subgoal, so no propagation of information is performed. Let us recall the term that had to be reduced:

$$s_i(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ )$$

Also in this case, there are three possible paths:

1. The first answer constraint $\mathtt{C}_1$ is `T` (which is an `EmptyConstraint`), thus the computation of the previous subgoal (which is ground) performed no substitution of variables:

$$\begin{array}{ll} & s_i(\ \mathtt{T}\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \\ \rightarrow_{\mathcal{T}(P)} & \mathtt{q}(Q_1,\ldots,Q_n)\ ;\ s_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \\ \rightarrow^*_{\mathcal{T}(P)} & \mathtt{C'}_1\ ;\ \ldots\ ;\ \mathtt{C'}_{w'}\ ;\ s_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \end{array}$$

   By inductive hypothesis, $\mathtt{C'}_1\ ;\ \ldots\ ;\ \mathtt{C'}_{w'}$ are the correct answer constraints of $\mathtt{q}(Q_1,\ldots,Q_n)$. For the recursive call, the proof is perfectly analogous to the corresponding one for the other cases discussed above.

2. The first answer constraint is not `T`, generating the following rewriting sequence:

$$\begin{array}{ll} & s_i(\ \mathtt{C_1}\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \\ \rightarrow_{\mathcal{T}(P)} & \mathtt{q}(Q_1,\ldots,Q_z)[Q_j \backslash X_j,\ldots,Q_k \backslash X_k])\ \mathtt{x}\ \mathtt{C}_1\ ; \\ & s_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \\ \rightarrow^*_{\mathcal{T}(P)} & (\mathtt{C'_1}\ ;\ \ldots\ ;\ \mathtt{C'_{w'}})\ \mathtt{x}\ \mathtt{C}_1\ ;\ s_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \end{array}$$

   Similarly to Case (1.2.b) above, the X_j are the linked variables that have already been instantiated, and their value is propagated to the corresponding Q_j. The X_j variables are computed in $\mathtt{C}_1$. By inductive hypothesis, $\mathtt{C'}_1,\ldots,\mathtt{C'}_{w'}$ are correct answer constraints for the term $\mathtt{q}(Q_1,\ldots,Q_z)[Q_j \backslash X_j,\ldots,Q_k \backslash X_k]$. Then, the `x` operator combines each solution of the second subgoal with the information in $\mathtt{C}_1$. Since we have passed the shared information with the substitution before reduction, we know that the shared variables have the same value; thus, no inconsistency comes up due to these variables. The only case when inconsistencies may arise (and be simplified) by the `x` operator is when both sets of answers contain an output variable and each one computes a different value for it. This inconsistent case is reduced to false, so no inconsistent answer constraint is carried on.

   As in the previous case, we will consider the recursive call after having presented the three cases.

3. There are no answer constraints to proceed with; thus, the first argument is `F` and the rewriting sequence is:

$$s_i(\ \mathtt{F},\ T_1 \ldots T_m\ ) \rightarrow_{\mathcal{T}(P)} \mathtt{F}$$

This last case is the base case for the recursion appearing in the two previous ones. By induction on the cardinality of the set of answer constraints $\mathtt{C}_1$; ... ; $\mathtt{C}_w$, we can see that the subterm

$$s_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ )$$

is a smaller recursive call, thus at some point the base case will be reached.

Hence, we are at the point in which we have computed all the accumulated answer constraints up to the last i-th subgoal:

$$\mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_n$$

($\Rightarrow$) We proceed by induction on both the structure of the clauses and the length of the computations.

We must prove that, for each computed answer θ for q and P, after the reduction $\mathcal{T}_g(q) \rightarrow^{!}_{\mathcal{T}(P)} \mathtt{CS}$, there exists a $\mathtt{C}$ in the answer constraint set $\mathtt{CS}$ such that $\mathtt{C}|_{var(\mathcal{T}_g(q))} \equiv \theta$.

Let us first consider the case when q is just defined by facts. For each fact defining the predicate of the query in the Datalog program, there are two cases:

1. It is possible to unify the query with the fact, getting a computed answer given by the substitution θ.

2. The query does not unify with the fact, so there is no computed answer for this execution branch.

The second case may occur (1) when a query is partially (or totally) instantiated and the given values do not coincide with those in the corresponding facts; or (2) when a query has a variable that appears multiple times in its arguments and a single fact assigns two different values to such variable at the same time.

By definition, our transformation generates an answer constraint for each fact. Assume that the query has the form $\mathtt{q}(A_1,\ldots,A_m)$ where each A_i, $1 \leq i \leq m$ is a variable or a constant. Given a fact $\mathtt{q}(t_1,\ldots,t_m)$, by definition of our transformation, there exists a $\mathtt{C}$ in $\mathtt{CS}$ of the form, $\bigwedge_{1\leq i\leq m} A_i = t_i$. For the first case above, clearly θ is equal to $\mathtt{C}$ in normal form (i.e., after having simplified the constraints of the form `Cte = Cte` when some argument in the query is instantiated). Now consider the second case above; then there exists an equality constraint `Cte = Cte'` for two different constants, or two

equality constraints `V = Cte , V = Cte'` with `Cte` $\neq$ `Cte'`; therefore, after normalization, the answer constraint reduces to `F` (correctness).

The rewriting sequence for the initial term $\mathcal{T}_g(q)$ is

$$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathtt{C}_1 \ ; \ \ldots \ ; \mathtt{C}_n \rightarrow^{!}_{\mathcal{T}(P)} \mathtt{C}_v \ ; \ \ldots \ ; \mathtt{C}_w$$

where n is the number of facts defining the Datalog predicate and $v, \ldots, w \in \{1, \ldots, n\}$. Each answer constraint in $\mathtt{C}_1 \ ; \ \ldots \ ; \mathtt{C}_n$ comes up from one Datalog fact. The second part of the sequence is the simplification for the union operator and constraint constructors.

Now we consider the case when q is defined by n clauses with non-empty bodies. We must ensure that each of these solutions is included in the set of answer constraints `CS`. By definition of our transformation, the set of answer constraints for q is the disjunction of the sets of answer constraints generated for each clause. Let us consider the solutions computed by each clause independently.

We recall the first step of the initial Maude term rewriting sequence:

$$\mathcal{T}_g(q) \rightarrow_{\mathcal{T}(P)} \mathtt{q}_1(T_1, \ldots, T_m) \ ; \ \ldots \ ; \mathtt{q}_n(T_1, \ldots, T_m).$$

Next we prove that the solutions computed from the i-th clause are included in the set of answer constraints computed by reducing $\mathtt{q}_i(\mathtt{T}_1, \ldots, T_m)$. By definition, each function $\mathtt{q}_i$ can be defined in our transformation in two different ways, depending on the number of subgoals in the clause represented by $\mathtt{q}_i$.

Let us consider the case of a clause having a single subgoal. Assume that the term at the *rhs* of that clause is a predicate call with predicate symbol `p` and z arguments: $\mathtt{p}(V_1, \ldots, V_z)$. By definition of our transformation, the equation for this clause is the following one, where `p` is now a defined function symbol:

$$\mathtt{eq}\ \mathtt{q}_i(U_1, \ldots, U_m) = \mathtt{p}(V_1, \ldots, V_z)$$

where $U_1, \ldots, U_m$ and $V_1, \ldots, V_z$ are the terms in the Datalog clause. Therefore, many of them may coincide, and the set of variables in $V_1, \ldots, V_z$ subsumes the set of variables in $U_1, \ldots, U_m$ (recall we are considering *safe* Datalog programs).

The rewriting sequence given by the equation shown above is as follows:

$$\mathtt{q}_i(T_1, \ldots, T_m) \rightarrow_{\mathcal{T}(P)} \mathtt{p}(W_1, \ldots, W_z) \rightarrow^{!}_{\mathcal{T}(P)} \mathtt{C'}_1 \ ; \ \ldots \ ; \mathtt{C'}_w$$

By inductive hypothesis, for every computed answer θ of the Datalog query $\mathtt{p}(W_1, \ldots, W_z)$, there exists an answer constraint $\mathtt{C'}_i$, $1 \leq i \leq w$ such that $\theta \equiv \mathtt{C'_i}$. Since the names of arguments in the Datalog program are preserved in the

Maude code, the computed answers restricted to the variables of the initial query form the answers for the Maude query. It is clear that, if the same restriction is applied to the answer constraint, the Datalog answers are still equivalent to the restricted answer constraint.

Let us now proceed with the general case when the clause body contains more than one subgoal. In this case, the chosen top-down left-to-right Datalog strategy states that, in order to compute the answers for the query, the answers for the first subgoal must be computed first. Then, the rest of the body with the corresponding substitutions (from the resolution of the first subgoal) must be resolved. As in the above case, we prove that each computed answer for this specific clause has an equivalent answer constraint computed by the corresponding q_i function.

Following our transformation, the rewriting sequence starts by rewriting to an auxiliary function s_2. This function represents the execution of the second subgoal after having reduced the first subgoal (on the first argument of s_2). This is ensured by the operational semantics of Maude, the definition of linked and relevant variables, and the patterns in the definition of the auxiliary functions.

The second part of the sequence below corresponds to the computation of that first subgoal:

$$\begin{aligned} & \mathtt{q}_i(T_1,\ldots,T_m) \\ \rightarrow_{\mathcal{T}(P)} \; & s_2(\, \mathtt{p}(W_1,\ldots,W_z),\, T_1 \ldots T_m \,) \\ \rightarrow^*_{\mathcal{T}(P)} \; & s_2(\, C_1 \,;\, \ldots \,;\, C_w,\, T_1 \ldots T_m \,) \end{aligned}$$

By inductive hypothesis, for each computed answer θ of the Datalog query $\mathtt{p}(W_1,\ldots,W_z)$, there exists an answer constraint C_i in the set $C_1 \,;\, \ldots \,;\, C_w$ such that $\theta \equiv C_i$. At this execution point, following the definition of our transformation, there are two possibilities depending on whether or not there are more subgoals. In the following, (Case 1) will refer to the case in which there are more subgoals, and (Case 2) to the case in which there are not.

Let us assume that we are dealing with the i-th subgoal, which has the function symbol s_i.

Case 1 In this case, the computation may proceed in two different ways:

1. There is no solution for $\mathtt{p}(W_1,\ldots,W_z)$. Therefore, the answer constraint set is of the form `F`. The rewriting sequence in this case is:

   $$s_i(\, \mathtt{C}_1 \,;\, \ldots \,;\, \mathtt{C}_w,\, T_1 \ldots T_m \,) \rightarrow_{\mathcal{T}(P)} \mathtt{F}$$

   This means that there is no solution for the first subgoal, so this case is trivially proved.

2. Consider the case when there are w different answer constraints for $\mathtt{p}(W_1,\ldots,W_z)$ (that by induction hypothesis include the equivalent answer constraints for each Datalog computed answer). The rewriting sequence following the definition of our transformation is:

$$s_i(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ) \rightarrow_{\mathcal{T}(P)} s_{i+1}(\ s'_i(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ),\ T_1\ldots T_m\ )$$

Note that, in order to compute the answer constraints for the subgoal s_{i+1}, we first have to rewrite the previous one by reducing the redex s'_i that contains the partially accumulated answer constraint set. Depending on the form of this constraint set, we have three possible rewritings:

(a) The first answer constraint for the previous subgoal ($\mathtt{C}_1$) is $\mathtt{T}$ (which is an `EmptyConstraint`). Therefore, the computation of the previous subgoal (which is ground) performed no substitution of variables:

$$\begin{aligned} & s_{i+1}(\ s'_i(\ \mathtt{T}\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ))\ ,\ T_1\ldots T_m\ ) \\ \rightarrow_{\mathcal{T}(P)}\ & s_{i+1}(\ \mathtt{q}(Q_1,\ldots,Q_z)\ ;\ s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m\ ),\ T_1\ldots T_m\ ) \\ \rightarrow^*_{\mathcal{T}(P)}\ & s_{i+1}(\ \mathtt{C'}_1\ ;\ \ldots\ ;\ \mathtt{C'}_{w'}\ ;\ s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1\ldots T_m),\ T_1\ldots T_m\ ) \end{aligned}$$

By inductive hypothesis, for each computed answer θ for the call $\mathtt{q}(Q_1,\ldots,Q_z)$, there exists an answer constraint $\mathtt{C}_i$ in the set $\mathtt{C'}_1\ ;\ \ldots\ ;\ \mathtt{C'}_{w'}$ such that $\theta \equiv \mathtt{C}_i$. These are all answers for the i-th subgoal (whose function symbol is $\mathtt{q}$, given by the function τ^p of our transformation). Intuitively, this rewriting step represents the propagation of variable assignments to the subsequent subgoals. It can be seen that, since no substitution needed to be propagated, all the answer constraints are also answer constraints for the query consisting of the conjunction of the previous subgoal(s) and the present one. Therefore, no solution is lost.

The recursive call of s'_i propagates not only the information from the first answer constraint, but also the information that is needed to proceed with the computation of the rest of the solutions. We will come back to this point of the proof after introducing the rest of the cases in order to prove that answers are also preserved for them.

(b) The first answer constraint is not $\mathtt{T}$ but a set $\mathtt{C}_1$; ... ; $\mathtt{C}_w$, which by hypothesis includes the equivalent answer constraints for the computed answers of the i-th subgoal. The rewriting sequence is:

$$\begin{aligned}
& s_{i+1}(\ s'_i(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m),\ T_1 \ldots T_m\ ) \\
\rightarrow_{\mathcal{T}(P)}\ & s_{i+1}(\ \mathtt{q}(Q_1,\ldots,Q_z)[Q_j \backslash X_j,\ldots,Q_k \backslash X_k])\ \mathtt{x}\ \mathtt{C}_1\ ; \\
& \qquad s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w, T_1 \ldots T_m\ ), \\
& \qquad T_1 \ldots T_m\ ) \\
\rightarrow^*_{\mathcal{T}(P)}\ & s_{i+1}(\ (\mathtt{C'}_1\ ;\ \ldots\ ;\ \mathtt{C'}_{w'})\ \mathtt{x}\ \mathtt{C}_1\ ; \\
& \qquad s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ), \\
& \qquad T_1 \ldots T_m\ )
\end{aligned}$$

where the X_j are the linked variables that have already been instantiated, and their value is propagated to the corresponding Q_j. The X_j variables are computed in $\mathtt{C}_1$. By inductive hypothesis, for each computed answer θ produced by the query $\mathtt{q}(Q_1,\ldots,Q_z)[Q_j \backslash X_j,\ldots,Q_k \backslash X_k]$, there exists a $\mathtt{C'}_i$ in $\mathtt{C'}_1$; ... ; $\mathtt{C'}_{w'}$ such that $\theta \equiv \mathtt{C'}_i$. Then, the $\mathtt{x}$ operator combines each solution of the second subgoal with the information in $\mathtt{C}_1$. Since we have passed the shared information with the applied substitution before the subsequent reduction step, we know that the shared variables have the same value. Therefore, the new combined solutions are consistent for the conjunction of the two (or more) subgoals. Note that the only case when inconsistencies may arise (and be simplified) by the $\mathtt{x}$ operator is when both sets of answers contain an output variable and each one computes a different value for it. This inconsistent case is reduced to false, so no consistent answer constraint is deleted.

(c) There are no answer constraints to proceed with, so the first argument is $\mathtt{F}$ and the rewriting sequence is:

$$s_{i+1}(\ s'_i(\ \mathtt{F},\ T_1 \ldots T_m\ ),\ T_1 \ldots T_m\ ) \rightarrow_{\mathcal{T}(P)} s_{i+1}(\ \mathtt{F},\ T_1 \ldots T_m\ )$$

This last case is the base case for the recursion of the two previous ones. By induction on the number of elements in the answer constraint set $\mathtt{C}_1$; ... ; $\mathtt{C}_w$, it can be observed that the subterm $s'_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ )$ in the cases (a) and (b) is a smaller recursive call. Therefore, at some point the sequence will reach the base case.

Hence, we are at the point in which we have computed all the

accumulated answer constraints up to the i-th subgoal and they include the equivalent answer constraints to the computed answers of the Datalog query $s_{i+1}(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_n,\ T_1 \ldots T_m\ )$.

Case 2 In this case, s_i is the last subgoal, so no propagation of information must be performed. We now also prove that, in this case, for each computed answer to the query, there exists an equivalent answer constraint as the result of the rewriting until normalization of the corresponding transformed query.

Recall that the term that had to be reduced at this point and that should generate the answer constraints for the considered Datalog clause is

$$s_i(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ )$$

where $\mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w$ include the equivalent answer constraints for the computed answers of $\mathtt{p}(W_1,\ldots,W_z)$. Similarly to Case 1 above, in this case there are also three possible rewriting sequences:

1. The first answer constraint for the previous subgoal $\mathtt{C}_1$ is `T` (which is an `EmptyConstraint`); thus, the computation of the previous subgoal (which is ground) performed no substitution of variables:

   $$\begin{array}{ll} & s_i(\ \mathtt{T}\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \\ \rightarrow_{\mathcal{T}(P)} & \mathtt{q}(Q_1,\ldots,Q_n)\ ;\ s_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \\ \rightarrow^*_{\mathcal{T}(P)} & \mathtt{C'}_1\ ;\ \ldots\ ;\ \mathtt{C'}_{w'}\ ;\ s_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \end{array}$$

   By inductive hypothesis, for each computed answer θ for the call $\mathtt{q}(Q_1,\ldots,Q_z)$, there exists in the set $\mathtt{C'_1}\ ;\ \ldots\ ;\ \mathtt{C'}_{\mathtt{w'}}$ an answer constraint $\mathtt{C}_i$ such that $\theta \equiv \mathtt{C}_i$. These are all answers for the i-th subgoal (whose function symbol is `q`, given by the function τ^p of our transformation).

   For the recursive call, we come back to this point of the proof after introducing the rest of the cases to prove that answers are also preserved for them.

2. The first answer constraint is not `T` but a set $\mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w$ that by hypothesis includes the equivalent answer constraints for the computed answers of the i-th subgoal. The rewriting sequence is:

   $$\begin{array}{ll} & s_i(\ \mathtt{C}_1\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \\ \rightarrow_{\mathcal{T}(P)} & \mathtt{q}(Q_1,\ldots,Q_z)[Q_j \backslash X_j,\ldots,Q_k \backslash X_k]\ \mathtt{x}\ \mathtt{C}_1\ ; \\ & s_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \\ \rightarrow^*_{\mathcal{T}(P)} & (\mathtt{C'}_1\ ;\ \ldots\ ;\ \mathtt{C'}_{w'})\ \mathtt{x}\ \mathtt{C}_1\ ;\ s_i(\ \mathtt{C}_2\ ;\ \ldots\ ;\ \mathtt{C}_w,\ T_1 \ldots T_m\ ) \end{array}$$

Similarly to Case (1.2.b) above, the X_j are the linked variables that have already been instantiated, and their value is propagated to the corresponding Q_j. The X_j variables are computed in $\mathtt{C}_1$. By inductive hypothesis, for each computed answer θ produced by the query $\mathtt{q}(Q_1,\ldots,Q_z)[Q_j\backslash X_j,\ldots,Q_k\backslash X_k])$, there exists a $\mathtt{C'}_i$ in $\mathtt{C'}_1$; ... ; $\mathtt{C'}_{w'}$ such that $\theta \equiv \mathtt{C'}_i$. Then, the $\mathtt{x}$ operator combines each solution of the second subgoal with the information in $\mathtt{C}_1$. Since we have passed the shared information with the applied substitution before the subsequent reduction step, we know that the shared variables have the same value, thus the new combined solutions are consistent for the conjunction of the two (or more) subgoals. We note that the only case when inconsistencies may arise (and be simplified) by the $\mathtt{x}$ operator is when both sets of answers contain an output variable and each one computes a different value for it. This inconsistent case is reduced to false, so no consistent answer constraint is deleted.

As in the previous case, we consider the recursive call after having presented the three cases.

3. There are no answer constraints to proceed with, thus the first argument is $\mathtt{F}$ and the rewriting sequence is:

$$s_i(\ \mathtt{F},\ T_1 \ldots T_m\ ) \rightarrow_{\mathcal{T}(P)} \mathtt{F}$$

This last case is the base case for the recursion of the two previous ones. By induction on the cardinality of the answer constraint set $\mathtt{C}_1$; ... ; $\mathtt{C}_w$, it can be observed that the subterm $s'_i(\ \mathtt{C}_2$; ... ; $\mathtt{C}_w,\ T_1 \ldots T_m\ )$ in the cases (a) and (b) is a smaller recursive call. Therefore, at some point of the sequence the base case will be reached.

Finally, we are at the point in which we have computed all the accumulated answer constraints up to the (last) i-th subgoal and they include the equivalent answer constraints to the computed answers of the Datalog query: $\mathtt{C}_1$; ... ; $\mathtt{C}_n$.

This concludes the proof.

4.3 Dealing with JAVA reflection

Addressing reflection is considered a difficult problem in the static analysis of JAVA programs, which is generally handled in an unsound or ad-hoc manner [LWL05]. Reflection in JAVA is a powerful technique that is used when a

program needs to examine or modify the runtime behavior of applications running in the JAVA virtual machine. For example, by using reflection, it is possible to write to object fields and invoke methods that are not known at compile time. JAVA provides a set of methods to handle reflection. These methods are found in the package java.lang.reflect.

In Figure 4.1 we show a simple example of reflection in JAVA. We define a class `P0` with two fields: `c1` and `c2`. In the `Main` class, an object `u` of class `P0` is created by using the constructor method `new`, which assigns the empty string to the two fields of `u`. Then, `r` is defined as a field of a class, specifically, as the field `c1` of an object of class `P0` since `v` stores the value `"c1"`. The sentence `r.set(u, w)` states that `r` is the field object `c1` of `u`, and its value is that of `w`, i.e., `"c2"`. Finally, the last instruction sets the new value of `v` to the value of `u.c1`, i.e., `"c2"`.

```
class P0 {
      P0 (String c1, String c2) {
         this.c1 = c1;
         this.c2 = c2;
      }
      public String c1;
      public String c2;
}

public class Main {
       public static void main(String[] args) {
       P0 u = new P0("","");
       String v = "c1";
       String w = "c2";
       java.lang.reflect.Field r = P0.class.getField(v);
       r.set(u, w);
       v = u.c1;
       } }
```

Figure 4.1: JAVA reflection example.

A pointer flow-insensitive analysis of this program would tell us that `r` may point not only to the field object `u.c1`, but also `u.c2` since the variable `v` appearing in the argument of the reflective method `getField` may be assigned both strings, `"c1"` and `"c2"`.

The key point for the reflective analysis is the fact that we do not have all the basic information for the points-to analysis at the beginning of the computation. In fact, the variables that occur in the methods handling reflection may generate new basic information. A sound proposal for handling JAVA reflection is proposed in [LWL05], which is essentially achieved by first annotating the Datalog program so that it is subsequently transformed by means

of an external (to Datalog) engine. As in [LWL05], we assume we know the name of the methods and objects that may be used in the invocations. In our approach, we use the Maude reflection capability to automatically generate the rules that represent new deduced information without resorting to any ad-hoc notation or external artifact.

Let us start by showing which pointer-analysis information JOEQ would extract from our example. We enforce the fact that we work at the *bytecode level*, so some JAVA instructions are converted into more than one bytecode instructions and some new auxiliary variables —in the example `$0`— are introduced.

Java Code	Extracted Information
`PO u = new PO("","");`	`vPO(u,0).` `vT(u,PO).`
`String v = "c1";`	`vPO(v,12).` `vT(v,string).`
`String w = "c2";`	`vPO(w,15).` `vT(w,string).`

`java.lang.reflect.Field r`
`  = PO.class.getField(v);`
	`vPO($0,18).`
	`vT($0,ClassPO).`
	`vT(r,field).`
	`mI(main,21,getField).`
	`iRet(21,r).`
	`actual(21,0,$0).`
	`actual(21,1,v).`

`r.set(u, w);`	`mI(main,30,set).`
	`actual(30,0,r).`
	`actual(30,1,u).`
	`actual(30,2,w).`

`v = u.c1;`	`l(u,c1,v).`

The following predicates state properties or actions performed to references and heap objects.

`vPO(V,H)`: A new object `H` is created —where `H` is the position of the call to the object's constructor in the code— and is referenced by the variable `V`.

`vT(V,T)`: The declared type of variable `V` is `T`.

`hT(H,T)`: The object `H` has type `T`.

`actual(I,N,V)`: The variable `V` is used as the actual parameter number `N` at the *invocation point*[7] `I`.

`mI(M,I,N)`: At invocation point `I` of method `M` there is a method call to be resolved with the name `N`.

`iRet(I,V)`: The variable `V` will receive the return value of the invocation at point `I`.

`l(V1,F,V2)`: The value of the field `F` of variable `V1` is assigned to variable `V2`.

`s(V1,F,V2)`: The value of variable `V2` is assigned to the field `F` of variable `V1`.

With this kind of information, it is easy to specify a non-reflective pointer analysis by means of Datalog clauses as in [WACL05]. The analysis would then mimic any possible flow of pointers in the code. Nevertheless, the analysis would be missing some hidden flow of pointers related to the use of reflection. Following the code execution with the semantics of the reflection API of JAVA in mind, `v` is the name of the field represented by the reflective object `r`. Then, the instruction `r.set(u,w)` stores the value of `w` in the field `c1` (represented by `r`) of the object pointed by `u`, and this would be resumed in the Datalog fact `s(u,c1,w)`. However, this behavior is *dynamic* because it depends on the runtime values of the variable `v`, and so we have no way to know what objects `v` can point to at compile time. For example, if `v` points to the string `"c1"`, as it does in the example, a new *reflective* object which represents a `"c1"` field of objects of class `PO` would be created and assigned to the variable `r`. Any call to the method `set` on the previous object would store within the field `"c1"` of the first parameter the content of the second parameter. Because `v` could potentially point to many other strings representing fields, `r` could point to many reflective objects representing the correspondent fields, and so calls to method `set` on `r` could mean many different kinds of stores `s(V1,F,V2)`.

The reflective analysis proposed by [LWL05] uses additional information (extracted by the JOEQ compiler) regarding which calls are done to the reflective API. This enriches the analysis allowing us to deduce new "on-the-fly" (at analysis time) facts that in the basic, non-reflective analysis were considered static information. For example, store facts `s(V1,F,V2)` can also be deduced by the clause:

```
s(V1,F,V2) :- iE(I,'Field.set') , actual(I,0,V) , vP(V,H) ,
              fieldObject(H,F)  , actual(I,1,V1), actual(I,2,V2) .
```

[7] An *invocation point* is either a method call, a static call or a special call at the bytecode level.

Let us present the new predicates that appear in this rule:

`iE(I,M)`: There is a call to the *resolved method*[8] `M` at the invocation point `I`. This predicate represents an approximation to the program's *call-graph*.

`fieldObject(H,F)`: The object `H` is a reflective object representing the field `F`.

These predicates are also derived from other facts. The meaning of the clause is straightforward: we state that `V2` is stored in the field `F` of `V1` if there is call to `Field.set` over a reflective object representing field `F` (`fieldObject(H,F)`) and the first and second parameters of that call are `V1` and `V2`, respectively.

In our reflective setting, we have followed the direct approach of translating Datalog clauses into Maude rules as in [AFJV09c] in order to ease the manipulation of modules at the metalevel. In this approach, each Datalog clause is translated into a Maude conditional rule. Therefore, checking that the clause body is satisfiable essentially boils down to checking if the condition of that rule holds. Following this idea, facts are translated into non-conditional rules in one-to-one correspondence. Consequently, deducing information amounts to rewriting queries into assignments to its arguments. The rule above is translated into the following Maude rule:

```
crl s(V1,F,V2) => V1 -> CteV1, F -> CteF, V2 -> CteV2
    if iE(I,'Field.set) => I -> CteI, 'Field.set -> 'Field.set
        /\ isConsistent I -> CteI
    /\ actual(CteI,'0,V) => CteI -> CteI, '0 -> '0 ,V -> CteV
        /\ isConsistent V -> CteV
    /\ vP(CteV,H) => CteV -> CteV , H -> CteH
        /\ isConsistent H -> CteH
    /\ fieldObject(CteH,F) => CteH -> CteH, F -> CteF
        /\ isConsistent F -> CteF
    /\ actual(CteI,'1,V1) => CteI -> CteI, '1 -> '1, V1 -> CteV1
        /\ isConsistent V1 -> CteV1
    /\ actual(CteI,'2,V2) => CteI -> CteI, '2 -> '2, V2 -> CteV2
        /\ isConsistent V2 -> CteV2 .
```

With this transformation, it can be seen that the structure of the resulting Maude code is very close to the original Datalog program. The novelty in the reflective analysis is in the need for new information to support the analysis, such as identifiers of reflective methods and string constants representing

[8] A *resolved method* refers to specific code from a certain class.

names of reflective objects. In our proof-of-concept prototype, we have considered field-reflection analysis. This implies that JOEQ must recover facts for the following two predicates:

`stringToField(H,F):` The object `H` is a string representation of field `F`.

`getField(M):` The method `M` is a reflective method which returns a reflective object representing a field.

The following field-reflection information would be extracted from our example:

```
stringToField(12,c1).
stringToField(15,c2).
getField(Class.getField).
```

Adding these extra information to the basic, non-reflective analysis we can deduce new reflective information which enriches the basic analysis. Then, the enriched basic analysis allows us to deduce new reflective information starting an iterative process until a fixpoint is reached.

Rewriting logic is reflective in a precise mathematical way: there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that we can represent (as data) any finitely presented rewrite theory $\mathcal{R}$ in $\mathcal{U}$ (including $\mathcal{U}$ itself), and then mimic the behavior of $\mathcal{R}$ in $\mathcal{U}$. The fact that rewriting logic is a reflective logic and the fact that Maude effectively supports reflective rewriting logic computation make reflective design (in which theories become data at the metalevel) ideally suited for manipulation tasks in Maude.

Maude's reflection is systematically exploited in our tool. On one hand, we can easily define new rules to be included in the specification by manipulating term meta-representations of rules and modules. On the other hand, by virtue of our reflective design, our metatheory of program analysis (which includes a common fixpoint infrastructure) is made accessible to the user who writes a particular analysis in a clear and principled way.

We have endowed our prototype implementation with the capability to carrying on reflection analysis for JAVA. The extension essentially consists of a module at the Maude meta–level that implements a generic infrastructure to deal with reflection. Figure 4.2 shows the structure of a typical reflection analysis running in our tool.

The static analysis is specified in two object-level modules, a *basic module* and a *reflective module*, that can be written in either Datalog or Maude, since Datalog analyses are automatically compiled into Maude code. The *basic program analysis (PA)* module contains the rules for the classical analysis (that

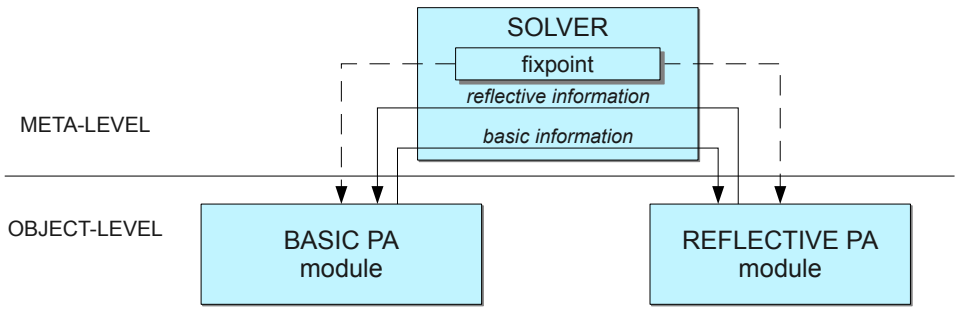


Figure 4.2: The structure of the reflective analysis.

neglects reflection) whereas the *reflective program analysis* module contains the part of the analysis dealing with the reflective components of the considered JAVA program. For example, the rule representing the reflective clause `s(V1,F,V2)` would be included in the reflective program analysis module.

The module called *solver* deals with the program analysis modules at the meta-level. It consists of a generic fixpoint algorithm that feds the reflective module with the information that can be inferred by the basic analysis and vice versa. Our implementation of the fixpoint is the following:

```
op fixpoint :  Module Module -> Module .

var M1 M2 M3 :  Module .

ceq fixpoint(M1,M2) = fixpoint(M3,M1)
    if M3 := closure(M1,M2)
    /\ M3 =/= M2 .
 eq fixpoint(M1,M2) = closure(M1,M2) [owise] .
```

The `closure` function infers all the information from the module given as its first parameter and adds it to the module given as its second parameter, returning the resulting updated module. In order to do that, `closure` queries the first module, translates the solutions into rules, and finally adds them to the second module.

For the points-to analysis with field reflection, the reflective and basic modules contain 11 rules each, whereas the generic solver is written in just 50 rules (including those that generate rules from the new computed information). The fact of separating the specification of the analysis into several modules enhances its comprehension and allows us to easily compose analysis

on demand.

4.4 The prototype DATALAUDE

DATALAUDE is a Haskell program that implements the Datalog transformation to RWL we have presented in this chapter. DATALAUDE is accessible via a web interface in `http://www.dsic.upv.es/users/elp/datalaude`.

As can be seen in Figure 4.3, DATALAUDE takes the same input files as DATALOG_SOLVE (as explained in Section 3.3), but its output is a Maude program. This program can subsequently be used by the Maude interpreter to reduce Datalog queries into sets of constraints representing the corresponding solutions to the original Datalog query. To do so, first the user should load the `.maude` file obtained from DATALAUDE into the interpreter, and then ask Maude to reduce the necessary queries.

Example 24

If DATALAUDE is fed with the classical Andersen points-to analysis, we obtain a file called `andersen.maude`. From the Maude interpreter we should load the transformation with the command:

```
load andersen.maude .
```

To execute the query `:- vP(V,o2).`, which is naturally written in Maude as `vP(vrbl('V),'o2)`, we would write the following:

```
reduce vP(vrbl('V),'o2) .
```

The output of Maude is shown below. The first part specifies the term that has been reduced (first line). The second part shows the number of rewrites and the execution time that Maude invested to perform the reduction (second line). The last part, which is written in several lines for the sake of readability, shows the result of the reduction (i.e., the set of answer constraints) together with its sort.

```
reduce in ANALYSIS : vP(vrbl('v), 'o2) .
rewrites:  39 in 0ms cpu5 (0ms real) (  rewrites/second)
result NonEmptyConstraintSet:
  vrbl('v) = 'q ;
  vrbl('v) = 'r , vrbl(vrbl('v) , 'o2) = 'q ;
  vrbl('v) = 'v1, vrbl(vrbl('v) , 'o2) = 'q , vrbl('q , 'o2) = 'r)
```

Note that the constraints obtained reference not only the variables occurring in the query, but also the existentially quantified variables used to infer the solutions.

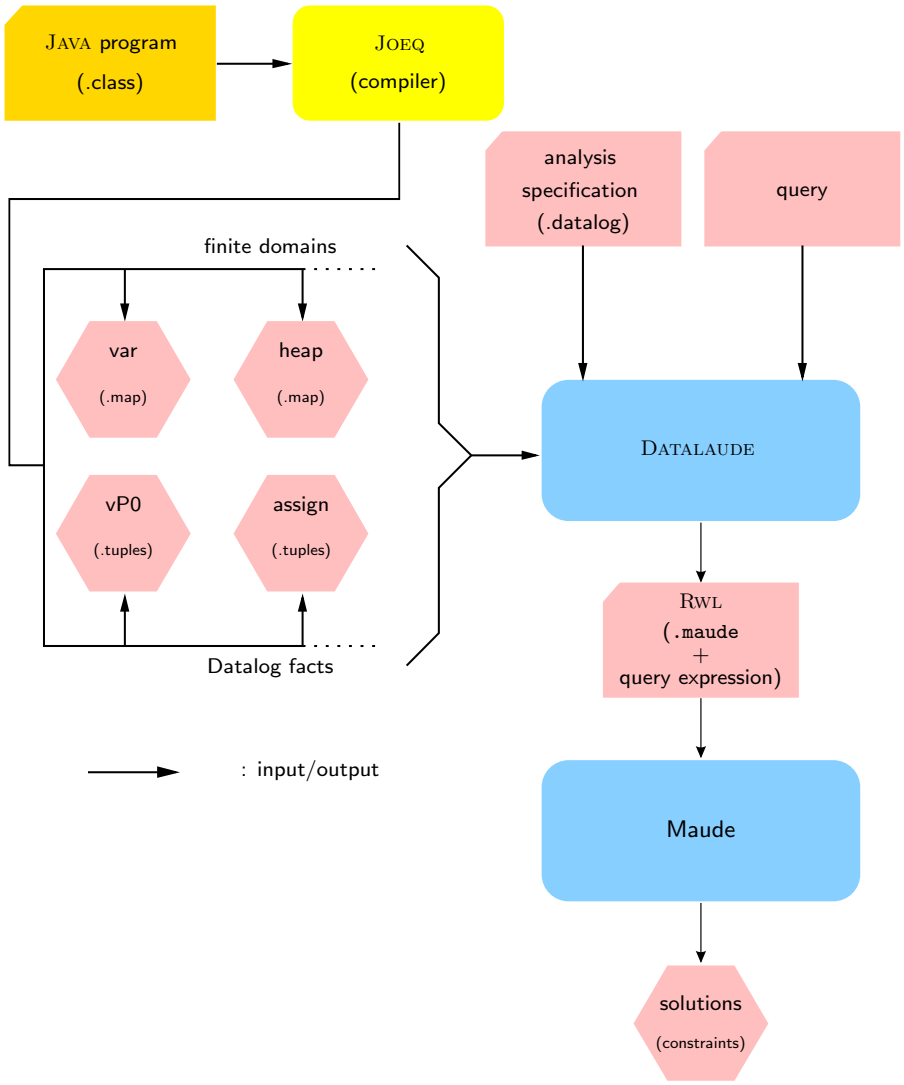


Figure 4.3: JAVA program analysis using DATALAUDE.

4.5 Experimental results

This section reports on the performance of our prototype, DATALAUDE, implementing the proposed transformation. First, we compare the efficiency of our implementation with respect to a naïve transformation to rewriting logic documented in [AFJV09c] and shown in Section 4.3; then, we evaluate the performance of our prototype by comparing it to three public Datalog solvers. All the experiments were conducted using JAVA JRE 1.6.0, JOEQ version 20030812, on a Mobile AMD Athlon XP2000+ (1.66GHz) with 700 Megabytes of RAM, running Ubuntu Linux 8.04.

4.5.1 Comparison w.r.t. a previous rewriting-based implementation

We implemented several transformations from Datalog programs to Maude programs before developing the transformation procedure presented in this thesis [AFJV09c]. The first attempt consisted of a one-to-one mapping from Datalog rules into Maude conditional rules. Then, in order to get rid of all the non-determinism caused by conditional equations and rules in Maude, we restricted our transformation to produce only unconditional equations as defined in the previous section.

In the following, we present the results obtained by using the rule-based approach, the equational-based approach, and the equational-based approach improved by using the *memoization* capability of Maude [CDE+07]. Maude is able to store each call to a given function (in the running example `vP(V,H)`) together with its normal form. Thus, when Maude finds a memoized call it does not reduce it but it just replaces it with its normal form, saving a great number of rewrites.

Table 4.1 shows the resolution times of the three selected versions. The sets of initial Datalog facts (`a/2` and `vP0/2`) are extracted by the JOEQ compiler from a JAVA program (with 374 lines of code) implementing a tree visitor. The Datalog clauses are those of our running example: the Andersen points-to analysis. The evaluated query is `?- vP(Var,Heap).`, i.e., all possible answers that satisfy the predicate `vP/2`.

The results obtained with the equational implementation are an order of magnitude better than those obtained by the naïve transformation based on rules. These results are due to the fact that the backtracking associated to the non-deterministic evaluation penalizes the naïve version. It can also be observed that using memoization allows us to gain another order of magnitude in execution time with respect to the basic equational implementation.

Table 4.1: Number of initial facts (`a/2` and `vP0/2`) and computed answers (`vP/2`), and resolution time (in seconds) for the three implementations.

a/2	vP0/2	VP/2	rule-based	equational	eq.+memozation
100	100	144	6.00	0.67	0.02
150	150	222	20.59	2.23	0.04
200	200	297	48.48	6.11	0.10
403	399	602	382.16	77.33	0.47
807	1669	2042	4715.77	1098.64	3.52

4.5.2 Comparison w.r.t. other Datalog solvers

We have used the same sets of initial facts to compare our prototype (the equational-based version with memoization) with three Datalog solvers: XSB 3.2 [9], Datalog 1.4 [10], and IRIS 0.58 [11]. Average resolution times of three runs for each solver are shown in Figure 4.4.

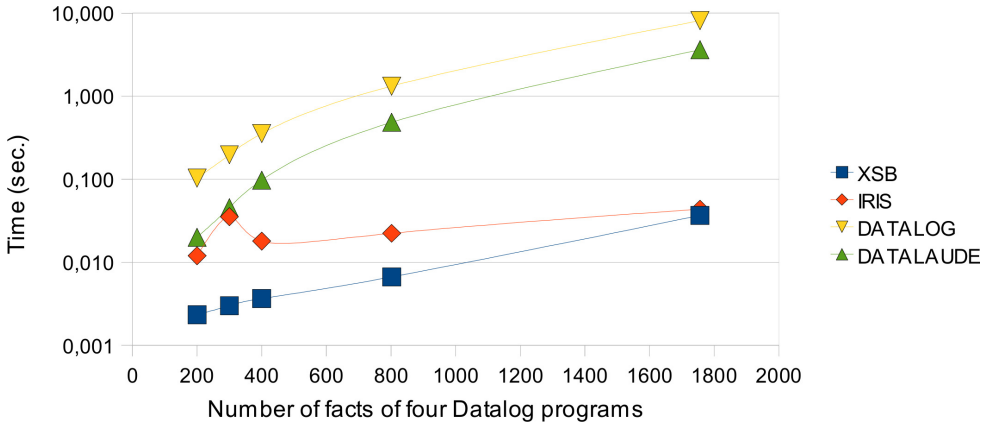


Figure 4.4: Average resolution times of four Datalog solvers (logarithmic time).

In order to evaluate the performance of our implementation with respect to the other Datalog solvers, only resolution times are presented in Figure 4.4 since the compared implementations are quite different in nature. This means that initialization operations, like loading and compilation, are not taken into account in the results. Our experiments conclude that DATALAUDE performs

[9] `http://xsb.sourceforge.net`
[10] `http://datalog.sourceforge.net`
[11] `http://iris-reasoner.sourceforge.net`

similarly to optimized deductive database systems like Datalog 1.4, which is implemented in C, although it is slower than XSB or IRIS. These results confirm that the equational implementation fits our program analysis purposes better, and provides a versatile and competitive Datalog solver as compared to other implementations of Datalog.

4.6 Conclusions and Related Work

We have presented a sound transformation from Datalog to RWL that is aimed towards Datalog-based static analysis. The transformation carries Datalog programs to the empowered framework of RWL while preserving its declarative nature. Reflection is a key capability of RWL that is specially suited to implement the evolution of systems. We have applied reflection to formalize static analyses that deal with JAVA reflection in a declarative way.

We have also presented some experimental results which show that, under a suitable transformation scheme (such as the proposed equation-based transformation extended with memoization), Maude can process a large number of equations extracted from statically analyzed, real JAVA programs. Our purpose has not been to produce the faster Datalog solver ever, but to provide a tool that supports sophisticated analyses with reasonable performance in a purely declarative way.

In the literature, many approaches have been proposed for transforming logic programs into term rewriting systems [Mar94, Red84, SKGST07]. These transformations aim at reusing the term rewriting infrastructure to run the (transformed) logic program while preserving the intended observable behavior (e.g., termination, success set, computed answers, etc.). Traditionally, translations of logic programs into functional programs are based on imposing an input/output relation (mode) on the parameters of the original program [Red84]. However, one distinguished feature of Datalog programs that burdens the transformation is that predicate arguments are not *moded*, meaning that they can be used both as input or output parameters. In particular, a novel transformation that does not impose modes on parameters was presented in [SKGST07]. The authors defined a transformation from definite logic programs into (infinitary) term rewriting for the termination analysis of logic programs. Contrary to our approach, the transformation of [SKGST07] is not concerned with preserving the computed answers, but only the termination behavior. Moreover, [SKGST07] does not tackle the problem of efficiently encoding logic (Datalog) programs containing a huge amount of facts in a rewriting-based infrastructure such as Maude

II

Automated Inference of Specifications

Specification Inference

In the context of computer programs, a specification defines what a computer program is expected to do. Specifications have been widely used for several purposes: they can be used to aid (formal) verification, validation or testing, to instrument software development, as summaries in program understanding, as documentation of programs, and to discover components in libraries or services in a network context, just to mention a few [ABL02, RMY+09, CSH10, HRD07, GM10, YKZZ08, NLV09, GMM09]. Depending on the context and the use of specifications, they can be defined before coding (e.g. for validation purposes), during the program coding (e.g. for testing or understanding purposes), or after the code has been written (for verification or documentation). Unfortunately, formal specifications are notoriously hard to write and debug, causing many programs to lack appropriate documentation. Specification inference can help to mitigate these problems and is also useful for legacy program understanding and malware deobfuscation, where the challenge is to understand what the malicious code is doing [CJK07]. We can find several proposals for (automatic) inference of high-level specifications (from an executable or from the source code) of a system, like [ABL02, CSH10, HRD07, GMM09], which have been proved to be very helpful.

In the literature, specification formalisms have been classified through some common characteristics [KU10]. It is frequent to distinguish between *property-oriented* specifications and *model-oriented* or *functional* specifications. It can be said that property-oriented specifications are at a higher description level than other kinds of specifications: they consist in an indirect definition of the system's behavior by means of stating a set of properties, usually in the form of axioms, that the system must satisfy [Win90, vV93]. In other words, a *property-oriented* specification does not represent the functionality of the program (the output of the system) but its properties in terms of relations among the operations that can be invoked in the program (i.e., it identifies different calls that have the same behavior when executed). This kind of specifications is particularly well suited for program understanding: the user can realize non-evident information about the behavior of a given function by observing its relation to other functions. Moreover, the inferred properties can manifest potential symptoms of program errors which can be used as input for (formal) validation and verification purposes.

Clearly, the task of automatically inferring program specifications is in general undecidable and, given the complexity of the problem, there exists a large number of different proposals which impose several restrictions. Many aspects vary from one solution to another: the kind of specifications that are

computed (e.g., model-oriented vs. property-oriented specifications), the kind of programs considered, the correctness or completeness of the method, etc.

We can also identify two mainstream approaches to perform the inference of specifications: glass-box and black-box. The glass-box approach [ABL02, CSH10] assumes that the source code of the program is available. In this context, the goal of inferring a specification is mainly applied to document the code, or to understand it [CSH10]. Therefore, the specification must be more succinct and comprehensible than the source code itself. The inferred specification can also be used to automatize the testing process of the program [CSH10] or to verify that a given property holds [ABL02]. The black-box approach [HRD07, GMM09] works only by running the executable. This means that the only information that is used during the inference process is the input-output behavior of the program. In this setting, the inferred specification is often used to discover the functionality of the system (or services in a network) [GMM09]. Although black-box approaches work without any restriction on the considered language –which is rarely the case in a glass-box approach– in general, they cannot *guarantee* the correctness of the results (whereas indeed semantics-based glass-box approaches can).

In this second part of the thesis, we investigate white-box techniques for synthesizing property-based specifications in two programming paradigms: functional-logic programming —as exemplified in the multiparadigm declarative language Curry— and object-oriented programming —exemplified in C.

5

Curry, the K Framework and Matching Logic

This chapter introduces the background knowledge that is necessary to understand the work presented in this Part II of the thesis. We introduce some fundamentals related to the multi-paradigm programming language Curry, the rule-based executable semantic framework **K**, and *Matching Logic*, which is a logic for the verification of systems.

5.1 Curry

Curry is a universal programming language aiming at the amalgamation of the most important declarative programming paradigms; namely, functional programming and logic programming. Curry combines features like *nested expressions*, *lazy evaluation* and *higher-order functions* from functional programming; and *logical variables*, *nondeterministic operations* and *built-in search* from logic programming.

A Curry program consists of *data* declarations and *function* definitions. For instance, the following data declaration:

```
data List a = Empty | Cons a (List a)
```

recursively defines a new polymorphic *datatype* `List a` whose values can be either `Empty` or have the form `Cons x xs`, where `x` and `xs` stand for values of types `a` and `List a`, respectively. A function definition is composed of (possibly conditional) oriented equations $l = r$ that define a transformation of an expression e matching the *pattern* expression on its left-hand side l, to the expression on its right-hand side r. For example:

```
append :: List a -> List a -> List a
append Empty ys = ys
```

```
append (Cons x xs) ys = Cons x (append xs ys)
```

recursively defines a function `append` that concatenates the two lists passed as arguments. The first line declares the type of `append`, which is a function that takes two arguments of type `List a` (the first two `List a` separated by a `->`) and returns a value of type `List a`. The next lines specify how an `append` expression is transformed. There are two equations, one for each possible form of the first argument. If the first argument is `Empty`, the first equation states that the result of the expression is the second argument; if the argument has the form `Cons x xs` (where `x` and `xs` can be any expression of type `List a`), the second equation specifies that the result is the expression `Cons x (append xs ys)`.

Executing a CURRY program means to simplify an expression until a value (or a solution) is computed. For example, assume that we want to compute the concatenation of two lists, one of them consisting of just the element `1`, and the other one with just the element `2`. Then, we write the expression to evaluate:

```
append (Cons 1 Empty) (Cons 2 Empty)
```

The CURRY interpreter, then, yields the following output:

```
Result: Cons 1 (Cons 2 Empty)
No more solutions.
```

where the first line states that a *result* `Cons 1 (Cons 2 Empty)` was found. This is the expected result, because it represents a list of two elements, being the first `1` and the second `2`. The second line is more enigmatic. Since CURRY is nondeterministic, as we mentioned earlier, it tries to reduce expressions in all possible ways, and *each possible reduction way* together with its respective *final value* constitute a *solution*. In this example, CURRY failed to reduce the goal/input expression in more than one way; this is why it outputs the message "`No more solutions.`".

One way of introducing nondeterminism in CURRY programs is by defining equations whose left-hand sides overlap. When CURRY has to evaluate an expression that matches the left-hand side of two or more equations, it nondeterministically chooses one of them to perform the reduction; that is, it tries both separately. Nondeterminism in CURRY is implemented with *backtracking*, as in *logic programming*. In the presence of a *nondeterministic* selection to make, backtracking takes one decision and *tracks* the rest of options together with the evaluation context. In this way, the evaluation context allows to recreate the state of the execution in which the decision was made for later exploration of the remaining alternatives.

In other words, when CURRY finds nondeterminism during evaluation, it makes decisions that determine a *possible* reduction path. After that reduction, if another solution is required, CURRY recovers the evaluation context and the alternatives of the last nondeterministic situation and takes a different decision, thus, exploring a different reduction path. For example, the following definition of a nondeterministic constant (0-ary function):

```
coin = 0
coin = 1
```

nondeterministically models the fact that the evaluation of an expression `coin` can have as a result both values `0` and `1`. If we feed the expression `coin` to the CURRY interpreter we obtain:

```
Result: 0
Result: 1
No more solutions.
```

In this case, since there are only two equations defining `coin`, there exist only two ways in which `coin` can be reduced.

CURRY equations can also be conditional, having the form $l \mid c = r$, where c is an expression of the predefined type `Success` and represents a condition or *constraint*. In CURRY, expressions with return type `Success` represent the logic programming concept of *goal*; thus, a function with return type `Success` can be considered as the counterpart of a logic programming *predicate*. `Success` expressions are special because CURRY only checks them for satisfiability, which is related to the fact that `Success` type has only one possible value: "`success`". `Success` expressions are normally used in the conditions of equations to check if the respective equation can be applied, but they can also be used standalone in order to ask CURRY if an expression is satisfiable (i.e., if it is reducible to `success`). For example, suppose we have the counterpart of a logic programming *database*:

```
data Person = Mike | Hanna | George | John

parent :: Person -> Person -> Success
parent Mike  Hanna  = success
parent Hanna George = success
parent Hanna John   = success
```

The first line defines a new *datatype* `Person` which has four possible values: `Mike`, `Hanna`, `George` and `John`. The next program statement declares that `parent` is a *function* with two arguments that returns a value of type `Success`,

that is, a *predicate*. The following lines are equations that specify for which values the function `parent` is defined, that is, for which values that *predicate* holds. If we execute the following expression in the previous program:

```
parent Mike John
```

the execution yields "`No more solutions.`" stating that CURRY cannot find a way to solve the goal expression. This behavior is consistent with the fact that the function `parent` is not defined for the provided arguments `Mike` and `John`. On the other hand, the execution of this similar expression:

```
parent Hanna John
```

yields the following output

```
Result: success.
No more solutions.
```

This output states that CURRY succeeded once in reducing the expression to `success`. This is the expected behavior because of the third program equation.

CURRY also allows *logical variables* in expressions by declaring them with the keyword `free`. Logical variables are another way of producing nondeterminism in CURRY. When CURRY needs to reduce an expression containing variables, it tries to unify it with the left-hand sides of the equations in the program. In this way, each successful unification creates a *possible* reduction, thus producing nondeterminism. Furthermore, every successful unification yields a *unifier* which contains the (partial) instantiation made to the logical variable in order to reduce the initial expression with the unifying equation. The *unifier*s produced along a reduction sequence are accumulated and their composition is returned by the interpreter together with the *final value*. For example, if we want CURRY to find the children of `Hanna`, we can evaluate the expression:

```
parent Hanna x
  where x free
```

in which the keyword `where` introduces local declarations (in this case, the declaration of `x` as a *logical variable*). The evaluation of this expression produces the following output:

```
Result: success
Bindings:
x=George
Result: success
```

```
Bindings:
x=John
No more solutions.
```

stating that CURRY found two solutions with the same final value `success`, but with different instantiations for the `free` variable `x`: `George` in the first case, and `John` in the second.

The combination of these features make CURRY a very powerful programming language. Moreover, the academic interest in CURRY is bringing new practical features for the language and new compilers to support them in real software.

5.2 The K Framework

K is an executable semantic framework in which programming languages, calculi, as well as type systems and formal analysis tools can be defined making use of configurations, computations, and rules. The most complete formal program semantics in the literature for Scheme, Java 1.4, Verilog, and C are currently available in **K**. **K** semantics are compiled into Maude [CDE+07] for execution, debugging, and model checking.

Program configurations are represented in **K** as potentially nested structures of labeled cells that represent the state of the program. **K** cells are containers that can act as lists, maps, (multi)sets of computations, or as a multiset of other cells. We represent **K** cells by surrounding their *contents* with square brackets and a subindex with the name of the cell. For example, the cell heap that represents a heap in which only the memory position 1 is initialized with the value 0 is represented by

$$\left[\ 1 \mapsto 0\ \right]_{\mathsf{heap}},$$

where the symbol $\mapsto$ represents a mapping between the objects at its sides. **K** computations carry "computational meaning" and they are represented as special nested list structures that sequentialize computational tasks, such as fragments of a program. Rules in **K** state how configurations (terms) can evolve during computation.

In Chapter 7, we use the **K** semantics for a C-like imperative language called KERNELC. The part of the **K** configuration structure for the KERNELC semantics that is relevant for understanding the techniques we discuss later is shown below.

$$\left[\ \left[\ \mathrm{K}\ \right]_{\mathsf{k}}\left[\ \mathrm{Map}\ \right]_{\mathsf{env}}\left[\ \mathrm{List}\ \right]_{\mathsf{stack}}\left[\ \mathrm{Map}\ \right]_{\mathsf{heap}}\ \right]_{\mathsf{cfg}}$$

Containers (or cells) in a configuration represent pieces of the program state, including a computation stack or continuation (named k), environments (env, heap), and a call stack (stack), among others.

Rules in **K** are graphically represented in two levels and state how configurations change. Changes in the current configuration (which is shown in the upper level) are explicitly represented by underlying the part of the configuration that changes. The new value that substitutes the one that changes is written below the underlined part.

As an example, we show the KERNELC rule for assigning a value V of type T to the variable X. This rule uses two cells, k and env. The env cell is a mapping of variables to their values, whereas the k cell represents a stack of computations waiting to be run, with the left-most (i.e., top) element of the stack being the next computation to be undertaken.

$$\left[\ \underline{X = \mathrm{tv}(T,V)}\ \cdots\ \right]_{\mathsf{k}} \left[\ \cdots\ X \mapsto \underline{\_}\ \cdots\ \right]_{\mathsf{env}}$$
$$\mathrm{tv}(T,V) \qquad\qquad V$$

The rule states that, if the next pending computation (which may be a part of the evaluation of a bigger expression) consists of an assignment $X = \mathrm{tv}(T,V)$, then we look for X in the environment $(X \mapsto \_)$ and we update the associated mapping with the new value V. The typed value $\mathrm{tv}(T,V)$ is kept at the top of the stack (it might be used in the evaluation of the bigger expression). The rest of the cell's content in the rule does not undergo any modification (this is represented by the $\cdots$ symbol).

This example rule reveals a feature of **K**: «rules only need to mention the minimum part of the configuration that is relevant for their operation». That is, only the cells read or changed by the rule have to be specified, and, within a cell, it is possible to omit parts of it by simply writing "$\cdots$". For example, the rule above emphasizes the interest in: the instruction $X = \mathrm{tv}(T,V)$ only at the beginning of the k cell, and the mapping from variable X to any value "$\_$" at any position in the env cell.

5.3 *Matching Logic*

Matching Logic is a logic for the verification of programs. We use the **ML** formalization given in [RES11, RS09]. Formulas in **ML** are called *patterns* and they represent sets of concrete program configurations. Patterns can be formalized as first-order logic (FOL) formulas whose atomic propositions include the constructs for representing program configurations. Intuitively, patterns

introduce logical variables into program configurations and also introduce formulas that constrain those logical variables. Pattern variables are typed.

Starting from the operational semantics of KernelC specified in **K** that we mentioned above, a handy axiomatic semantics that is based on **ML** patterns has been systematically defined for verification purposes within the **ML** framework [RES11]. The **ML** patterns of the KernelC axiomatic semantics consist of **K** configuration cells that are additionally constrained with FOL formulas. We use the sugared notation of [RES11, RS09] that embeds the FOL formula within a special cell called ϕ and substitutes explicit existential logical quantifiers with a special mark ? at the beginning of the existentially quantified variable's name. For example, the set of KernelC configurations where the specific program variable x holds a value that is greater than 5 can be represented with the following **ML** pattern, where free variable E matches any other information in the env cell, and C matches the rest of the cells in the configuration:

$$\left[\ [\mathtt{x} \mapsto ?\mathsf{x}, \mathsf{E}]_{\mathsf{env}} [?\mathsf{x} > 5]_{\phi}\ \mathsf{C}\ \right]_{\mathsf{cfg}} \tag{5.1}$$

Program variables[1] in **ML** formulas such as x are written in *teletype* font and logical variables such as x are written in sans font. Metavariables in **ML** such as C are written in capital letters.

The **ML** axiomatic semantics uses inference rules to derive sequents like $\mathrm{P}_1 \Downarrow \mathrm{P}_2$, called *correctness pairs*, which relate patterns before and after the execution of a program fragment. The meaning of a sequent $\mathrm{P}_1 \Downarrow \mathrm{P}_2$ is that the execution of a concrete configuration matching P_1 yields a configuration matching P_2. The basic axiom for final symbolic configurations that indicate the end of a successful (axiomatic semantics) derivation is as follows:

$$\frac{}{[\Gamma]_{\mathsf{cfg}} \Downarrow [\Gamma]_{\mathsf{cfg}}}$$

with Γ being a final pattern, i.e., a pattern whose k cell is empty or consists of just a value. As an example of **ML** inference rule, let us describe the rule that corresponds to the assignment of a program variable.

$$\frac{[[\mathsf{K}]_{\mathsf{k}}\mathsf{C}]_{\mathsf{cfg}} \Downarrow [[\mathsf{I}]_{\mathsf{k}} [\rho]_{\mathsf{env}} \mathsf{C}']_{\mathsf{cfg}}}{[[\mathsf{X} = \mathsf{K};]_{\mathsf{k}}\mathsf{C}]_{\mathsf{cfg}} \Downarrow [[]_{\mathsf{k}} [\rho[\mathsf{X} \mapsto \mathsf{I}]]_{\mathsf{env}} \mathsf{C}']_{\mathsf{cfg}}}$$

In the rule above, K matches a computation expression, I an integer value, X a program variable, ρ a map, and C and C′ a bag of configuration cells

[1] Note that program variables are constants in **ML** formulas.

(including the ϕ cell). The rule hypothesis can be read as: «starting from any configuration with K as the only remaining computation in the k cell, assume that a configuration that matches $[[\mathsf{I}]_{\mathsf{k}}[\rho]_{\mathsf{env}}\mathsf{C}']_{\mathsf{cfg}}$ is obtained, with I being the result of evaluating the expression K». Then, the conclusion can be read as: «starting from a configuration that consists of C plus the assignment of expression K to the variable identified by X as the only remaining computation, its execution yields a configuration consisting of C', an empty k cell (i.e., no pending computations remain), and an updated environment $\rho[\mathsf{X} \mapsto \mathsf{I}]$ in which I is assigned to the variable X ».

The following example illustrates the application of the assignment inference rule.

Example 25

Let us consider the following (axiomatic semantics) symbolic configuration, which is an instance of the **ML** pattern of Equation (5.1) and contains the assignment instruction `x=3` in the k cell:

$$\left[\ [\mathtt{x} = 3;]_{\mathsf{k}}[\mathtt{x} \mapsto ?\mathsf{x}, \mathsf{E}]_{\mathsf{env}}[?\mathsf{x} > 5]_{\phi}\mathsf{C}''\ \right]_{\mathsf{cfg}}$$

This symbolic configuration is also a pattern that matches the left-hand pattern of the correctness pair in the consequent of the inference rule. Moreover, the antecedent of the rule holds since

$$[[3]_{\mathsf{k}}\mathsf{C}]_{\mathsf{cfg}} \Downarrow [[3]_{\mathsf{k}}[\mathtt{x} \mapsto ?\mathsf{x}, \mathsf{E}]_{\mathsf{env}}\mathsf{C}']_{\mathsf{cfg}}$$

with $\mathsf{C} = [\mathtt{x} \mapsto ?\mathsf{x}, \mathsf{E}]_{\mathsf{env}}\ \mathsf{C}'$ and $\mathsf{C}' = [?\mathsf{x} > 5]_{\phi}\mathsf{C}''$. This is because, according to the **K** type system, 3 is both an integer value and a program expression, thus it respectively matches the I and the K variables in the rule hypothesis. Therefore, the derivation computed by the application of the rule is

$$\left[[\mathtt{x} = 3;]_{\mathsf{k}}\mathsf{C}\right]_{\mathsf{cfg}} \Downarrow \left[[]_{\mathsf{k}}[\mathtt{x} \mapsto 3, \mathsf{E}]_{\mathsf{env}}\mathsf{C}'\right]_{\mathsf{cfg}}$$

Note that **ML** does not need logical "separation" (meaning that the heap can be split into two disjoint parts where the separate formulas hold [Rey02]) because it achieves it at the structural level. That is, any pair of subterms in a pattern configuration that are not related by the containment order are considered to be distinct and disjoint; and, if a pattern matches two terms in a multiset, the two terms have to be distinct. For example, in the pattern of Equation (5.1), the binding $\mathtt{x} \mapsto ?\mathsf{x}$ is matched with the current configuration separately from the rest of the environment E, and, thus, no overlapping of bindings can occur.

6

Inference of Specifications from Curry Programs

This chapter investigates how equational program specifications can be automatically inferred from programs that are written in the multiparadigm declarative language Curry. To do this work, we took inspiration from QuickSpec [CSH10], which is an (almost) black-box inference approach for Haskell programs [PJ03] based on testing. QuickSpec automatically infers program specifications as sets of equations of the form $e_1 = e_2$, where e_1, e_2 are generic program expressions that appear to have the same computational behavior. This approach has two key properties:

it is completely automatic as it needs only the program to run, plus some indications regarding target functions and generic values to be employed in the synthesized equations, and

the outcomes are very intuitive since they are expressed *only* in terms of the program components, so the user does not need any kind of extra knowledge to interpret the results.

However, our proposal ended up being radically different from QuickSpec:

- First, we aim to infer *correct* (algebraic) property-oriented specifications. To this end, instead of a testing-based approach, we propose a glass-box *semantic-based* approach.

- Second, we consider the functional logic language Curry defined in [Han97, Han06]. Curry is a multi-paradigm programming language that combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions) and logic programming (logic variables, partial data structures, built-in search).

Due to lazy evaluation in presence of (free) logical variables, the problem of inferring specifications for this kind of language poses several additional problems w.r.t. other programming paradigms. We discuss these issues in Section 6.1.

In the rest of the chapter, we first introduce the problem of generating useful specifications for the functional logic paradigm by discussing a simple, illustrative example. In Section 6.2, we define our notion of specification, which is composed of equations of different kinds. Thereafter, in Section 6.3, we explain in detail how the specifications are computed. In Section 6.4 we discuss some examples of specifications computed by the prototype AbsSpec that implements the technique. Finally, Section 6.5 discusses the main related work and concludes.

6.1 Specifications in the functional-logic paradigm

Curry is a *lazy* functional *logic* language that admits free (logical) variables in expressions and whose program rules are evaluated non-deterministically. Differently from the functional case, i.e., languages without logic variables that may be instantiated during the program execution, an equation $e_1 = e_2$ can be interpreted in many different ways. Let us discuss the key points of the interpretation problem by means of a (very simple) illustrative example.

Example 26 (Boolean logic example)
Consider the standard definition of the boolean data type with values `True` and `False` and operations defining the logic connectives `and`, `or`, `not` and `imp`:

```
and True  x = x
and False _ = False
or True  _ = True
or False x = x
not True  = False
not False = True
imp False x = True
imp True  x = x
```

This is a pretty standard "short-cut" definition of boolean connectives. For example, the definition of `and` states that whenever the first argument is equal to `False`, the function returns the value `False`, regardless of the value of the second argument. Since the language is lazy, in this case the second argument will not be evaluated.

For this example, one could expect a (property-oriented) specification with equations like[1]

```
imp x y = or (not x) y                                  (6.1)
not (or x y) = and (not x) (not y)                      (6.2)
not (and x y) = or (not x) (not y)                      (6.3)
not (not x) = x                                         (6.4)
and x (and y z) = and (and x y) z                       (6.5)
and x y = and y x                                       (6.6)
```

which are well-known laws among the (theoretical) boolean operators. This comprehensible specification aids the user to learn the properties of the program. In addition, the specification can be useful to detect bugs in the program by observing both, properties (equations) that occur in the specification but were not expected, and expected equations that are eventually missing. These equations, of the form $e_2 = e_2$, can be read as

all possible results for e_1 *are also results for* e_2,
and vice versa. (6.7)

In the following, we call this notion of equivalence *computed result equivalence* and we denote it by $=_{CR}$.

Actually, Equations (6.1), (6.2), (6.3) and (6.5) are *literally* valid in this sense since, in Curry, free variables are admitted in expressions, and the mentioned equations are valid *as they are*. Note that this is not true for Equations (6.4) and (6.6) since the expressions "`not (not x)`" and "`and x y`" do not compute the same results as "`x`" and "`and y x`", respectively, in the considered program. This interpretation of equations is quite different from the pure functional case where equations *have to be interpreted* as properties that hold for any *ground* instance of the variables occurring in the equation.

Let us first introduce the notation for evaluations by means of an example. The expression $\{\texttt{x/True}\} \cdot \texttt{True}$ denotes that the normal form `True` (at the right of the $\cdot$ symbol) has been reached with computed answer substitution $\{\texttt{x/True}\}$ (at the left of the $\cdot$ symbol). Now, the goal on the left hand side of the equation `not (not x)` evaluates to two normal forms: $\{\texttt{x/True}\} \cdot \texttt{True}$ and $\{\texttt{x/False}\} \cdot \texttt{False}$, whereas the right hand side of the equation `x` evaluates just to $\{\} \cdot \texttt{x}$. Note however that any ground instance of the two goals evaluates to

[1] In this section, our main goal is to give the intuition of the specification computed by our approach, thus we show just a subset of the equations satisfied by the program.

the same results, namely both `True` and `not (not True)` evaluate to $\{\} \cdot$ `True`, and both `False` and `not (not False)` evaluate to $\{\} \cdot$ `False`.

This fact motivates the use of an additional notion of equivalence, called *ground equivalence*, which can be helpful for the user since the equations that hold under this equivalence represent, in general, interesting properties of the program. We denote it by $=_G$. This notion coincides with the (only possible) equivalence notion used in the pure functional paradigm: two terms are ground equivalent if, for all ground instances, the computed results for both terms coincide.

Because of the presence of logic variables, there is another very relevant difference w.r.t. the pure functional case that is concerned with *contextual equivalence*: given a valid equation $e_1 = e_2$, is it true that, for any context C, the equation $C[e_1] = C[e_2]$ still holds? Due to the use of *narrowing*, Curry is not referentially transparent[2] w.r.t. its operational behavior, i.e., an expression can produce different computed values when it is embedded in a context that promotes different bindings for its free variables (as shown in the following example), which makes the answer to the question posed above not straightforward.

Example 27

Given a program with the following rules

```
g x = C (h x)
g' A = C A
h A = A
f (C x) B = B
```

the expressions `g x` and `g' x` compute the same result, namely $\{$`x/A`$\} \cdot$ `C A`. However, the expression `f (g x) x` computes one result, namely $\{$`x/B`$\} \cdot$ `B`, while expression `f (g' x) x` computes none.

Thus, in the Curry case, it can be of interest a stronger equivalence notion that requires the computed results of two terms to be equal even when the two terms are embedded within any context. We call this equivalence *contextual equivalence* and we denote it by $=_C$. As we show later, Equations (6.1), (6.2), (6.3) and (6.5) are valid w.r.t. this equivalence notion.

[2] The concept of referential transparency of a language can be stated in terms of a formal semantics as: the semantics equivalence of two expressions e, e' implies the semantics equivalence of e and e' when used within any context $C[\cdot]$. Namely, $\forall e, e', C.\ [\![e]\!] = [\![e']\!] \Rightarrow [\![C[e]]\!] = [\![C[e']]\!]$.

We can see that $=_C$ is (obviously) stronger than $=_{CR}$, which is in turn stronger than $=_G$ (in symbols, $=_C \subseteq =_{CR} \subseteq =_G$). As a conclusion, for our example we would get the following (partial) specification.[3]

```
imp x y =_C or (not x) y
not (or x y) =_C and (not x) (not y)
not (and x y) =_C or (not x) (not y)
not (not x) =_G x
and x (and y z) =_C and (and x y) z
and x y =_G and y x
```

This example has shown, first, the kind of property-oriented specifications that we want to compute from the given program, and second, the need to consider different kinds of equalities between terms in order to get a useful specification. It is worth noticing that adopting *only* a notion of equivalence based on the referentially transparent semantics (the $=_C$ equivalence) can be too restrictive: we may lose important properties. However, by using the weaker notions we cannot know if two equivalent expression are also equivalent within any context.

The need for determining $=_C$ equalities is the reason why we believe that, in the case of Curry, the use of a semantics-based approach can be more suited than testing-based approaches. In a test-based approach expressions should have to be nested within some outer context in order to establish their $=_C$ equivalence. Since the number of needed terms to be evaluated grows exponentially w.r.t. the depth of nestings, the addition of a further outer context would dramatically worsen the performance. Moreover, if we try to mitigate this problem by reducing the number of terms/tests to be checked, the quality of the produced equations degrades sensibly. On the contrary, a semantics-based approach can achieve the $=_C$ equivalence by construction as shown below.

6.2 Formalization of equivalence notions

In this section, we formally present all the kinds of term equivalence notions that are used to compute equations of the inferred specification. We need first to introduce some basic formal notions that are used in the rest of the chapter.

[3] As discussed later, our technique computes a complete specification for a specific *size* of terms in equations.

We say that a first order CURRY program is a set of rules P built over a signature Σ which is partitioned in $\mathcal{C}$, the *constructor* symbols, and $\mathcal{D}$, the *defined* symbols, being $\mathcal{C} \cap \mathcal{D} = \varnothing$. $\mathcal{V}$ denotes a (fixed) countably infinite set of variables and $\mathcal{T}(\Sigma, \mathcal{V})$ denotes the terms built over signature Σ and variables $\mathcal{V}$. A *fresh* variable is a variable that appears nowhere else.

In order to state formally the equivalences $=_C$ and $=_{CR}$ described in the previous section, we need two semantic evaluation functions $\mathcal{S}^C[\![\cdot]\!]$ and $\mathcal{S}^{CR}[\![\cdot]\!]$ which enjoy some specific properties.

Definition 28 (Computed Results Semantics) *Given a program P, we say that a semantic evaluation function $\mathcal{S}[\![\cdot]\!]$ is fully abstract w.r.t. to the behavior of* computed results *if, given two terms t_1 and t_2, the semantics of both terms are identical ($\mathcal{S}[\![t_1]\!] = \mathcal{S}[\![t_2]\!]$) if and only if the evaluations of t_1 and t_2 compute the same results.*

We refer to a semantic evaluation function which satisfies this property as a computed results semantics *and we represent it as $\mathcal{S}^{CR}[\![\cdot]\!]_P$.*

Definition 29 (Contextual Semantics) *Given a program P, we say that a semantic evaluation function $\mathcal{S}[\![\cdot]\!]$ is fully abstract w.r.t. to the behavior of* computed results under any context *if, given two terms t_1 and t_2, the semantics of both terms are identical ($\mathcal{S}[\![t_1]\!] = \mathcal{S}[\![t_2]\!]$) if and only if, for any context $C[\cdot]$, the evaluations of $C[t_1]$ and $C[t_2]$ compute the same results. We say that such a semantics fulfills referential transparency.*

We refer to a semantic evaluation function which satisfies this property as a contextual semantics *and we represent it as $\mathcal{S}^C[\![\cdot]\!]_P$.*

Now we are ready to introduce our notion of specification.

The specification. Formally, an algebraic specification is a set of *sequences of equations* of the form $t_1 =_K t_2 =_K \ldots =_K t_n$, with $K \in \{C, CR, G\}$ and $t_1, t_2, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$. Actually, we call *sequence of equations* to the set of equations that can be build by taking any two terms t_i and t_j from the sequence $t_1 =_K \ldots =_K t_n$. K distinguishes the kinds of computational equalities that we previously informally discussed, and which we now present formally.

Contextual Equivalence $=_C$. This equivalence states that two terms t_1 and t_2 are equivalent if $C[t_1]$ and $C[t_2]$ have the same behavior for any context $C[\cdot]$. Given that $\mathcal{S}^C[\![\cdot]\!]$ is fully abstract w.r.t. the behavior of computed results under any context, two terms t_1 and t_2 are related by the contextual relation $=_C$ if and only if their semantics coincide; namely,

$$t_1 =_C t_2 \iff \mathcal{S}^C[\![t_1]\!]_P = \mathcal{S}^C[\![t_2]\!]_P$$

This is the more restrictive notion and the most difficult equivalence to be established by testing approaches.

Computed-result equivalence $=_{CR}$. This equivalence states that two terms are equivalent when the computed results of their evaluation are the same; namely,

$$t_1 =_{CR} t_2 \iff \mathcal{S}^{CR}[\![t_1]\!]_P = \mathcal{S}^{CR}[\![t_2]\!]_P.$$

Therefore, the computed-result equivalence abstracts from the way in which the results are produced during computation. The $=_{CR}$ equivalence is coarser than $=_C$ ($=_C \subseteq =_{CR}$) as shown by Example 27.

Theoretically, it is possible to use just a correct semantics as $\mathcal{S}^{CR}$, but clearly in such case we will not have all equivalences which are valid w.r.t. a fully abstract semantics.

Ground Equivalence $=_G$. This equivalence states that two terms are equivalent if all their possible ground instances have the same computed results; namely

$$t_1 =_G t_2 \iff \forall \theta \,\text{grounding}.\ \mathcal{S}^{CR}[\![t_1\theta]\!]_P = \mathcal{S}^{CR}[\![t_2\theta]\!]_P.$$

By definition, the $=_G$ equivalence is coarser than $=_{CR}$ ($=_{CR} \subseteq =_G$).

User Defined Equivalence $=_{Ueq}$. This equivalence depends upon a notion of equality defined by the user. When dealing with a user-defined data type, the user may have defined a specific notion of equivalence by means of an "equality" function. Let us call $equal(t_1, t_2)$ such user-defined function. Then, we state that

$$\begin{array}{rcl} t_1 =_{Ueq} t_2 & \iff & \mathcal{S}^{CR}[\![equal(t_1, t_2)]\!]_P = \mathcal{S}^{CR}[\![\texttt{True}]\!]_P \\ & \iff & equal(t_1, t_2) =_{CR} \texttt{True} \end{array}$$

Clearly, we do not have necessarily any relation between $=_{Ueq}$ and the others, as the user function *equal* may have nothing to do with equality. However, in typical situations such a function is defined to preserve at least $=_G$, meaning that $t_1 =_G t_2$ implies $t_1 =_{Ueq} t_2$.

These equations can provide the user significant information about the structure and behavior of the program and a pragmatical tool should reasonably present a sequence

$$\texttt{True} =_{CR} equal(t_1, t_2) =_{CR} \ldots =_{CR} equal(t_{n-1}, t_n)$$

as

$$t_1 =_{Ueq} \ldots =_{Ueq} t_n$$

for readability purposes.

In any case, it is clear from the definition that this is technically just a particular instance of $=_{CR}$, so it does not need to be considered by itself and in the following we do not consider it explicitly.

To summarize, we have $=_C \subseteq =_{CR} \subseteq =_G$ and only $=_C$ is referentially transparent (i.e., a congruence w.r.t. contextual embedding).

We have to instantiate now the generic semantics notions $\mathcal{S}^C[\![\cdot]\!]_P$ and $\mathcal{S}^{CR}[\![\cdot]\!]_P$ with specific semantics. In the following we briefly introduce the **WERS**-semantics of [Bac12, BC12] that we use in order to generate this kind of specifications.

The semantics. We evaluate first order CURRY programs on the **WERS** semantics, which is a condensed, goal-independent semantics recently defined in [Bac12, BC12] for functional logic programs, and we represent it by $\mathcal{E}[\![\cdot]\!]_{\mathcal{F}[\![P]\!]}$. We preferred this semantics instead of the well-established (small-step) operational [Han06] and I/O semantics [AHH+05] because these previous semantics do not fulfill referential transparency, whereas the **WERS** semantics does. This fact makes the (more elaborated) semantics of [Bac12, BC12] an appropriate contextual semantics ($\mathcal{S}^C[\![\cdot]\!]_P$) for computing equations w.r.t. $=_C$. Intuitively, due to the definition of the **WERS** semantics, $t_1 =_C t_2$ means that *all the different ways* in which these two terms reach their computed results coincide. We could have used the operational and I/O semantics as computed results semantics but, as we show later, we can easily obtain $\mathcal{S}^{CR}[\![\cdot]\!]$ from $\mathcal{E}[\![\cdot]\!]_{\mathcal{F}[\![P]\!]}$. Moreover, the semantics that we use has another property which is very important from a pragmatical point of view: it is condensed, meaning that denotations are the smallest possible (between all those semantics which are fully abstract). This is an almost essential feature in order to develop a semantic-based tool which has to *compute* (an approximation of) the semantics. In particular, with this semantics it is reasonable to compute a finite number of iterations of the program's denotation itself, while for the other mentioned semantics it is not the case.

The denotation $\mathcal{F}[\![P]\!]$ of a program P is the least fixpoint of an immediate consequence operator $\mathcal{P}[\![P]\!]$. This operator is based on a term evaluation function $\mathcal{E}[\![t]\!]$ which, for any term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ and any interpretation $\mathcal{I}$, gives the semantics of t as $\mathcal{E}[\![t]\!]_{\mathcal{I}}$. Intuitively, the evaluation $\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$ computes, w.r.t. program P, a tree-like structure collecting the "relevant history" of the

computation of all (partially) computed results of t, abstracting from function calls and focusing only on the way in which the result is built. In particular, every leaf of the tree represents a normal form of the initial term. Nodes are pairs of the form $\sigma \cdot s$, where σ is a substitution (binding variables of the initial expression with linear constructor terms), and s is a partially computed value, that is, a term in $\mathcal{T}(\mathcal{C}, \mathcal{V} \cup \mathcal{V}_\varrho)$ that may contain special variables ϱ_0, ϱ_1, $\ldots \in \mathcal{V}_\varrho$ (a set disjoint from $\mathcal{V}$) indicating an unevaluated subterm. Leaves with no occurrences of special variables represent computed results. We denote by $cr(T)$ the set of computed results of the semantic tree T.

Full-abstraction w.r.t. the computed-result behavior and referential transparency of the semantics are proven in [Bac12].

Theorem 30 ([Bac12]) *Let P be a first-order* Curry *program, e, e' terms in $\mathcal{T}(\Sigma, \mathcal{V})$, and $C[\cdot]$ be a context. If $\mathcal{E}[\![e]\!]_{\mathcal{F}[\![P]\!]} = \mathcal{E}[\![e']\!]_{\mathcal{F}[\![P]\!]}$, then it holds $\mathcal{E}[\![C[e]]\!]_{\mathcal{F}[\![P]\!]} = \mathcal{E}[\![C[e']]\!]_{\mathcal{F}[\![P]\!]}$.*

Thus, $\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$ is an appropriate instantiation for $\mathcal{S}^C[\![t]\!]$. Moreover, the following theorem shows that $\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]}$ can be used to obtain $\mathcal{S}^{CR}[\![t]\!]$, thus avoiding an unnecessary recomputation of the semantics.

Theorem 31 ([Bac12]) *Let P be a first-order* Curry *program and t be a term in $\mathcal{T}(\Sigma, \mathcal{V})$. Then, $cr(\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]})$ corresponds to the set of computed results of t using P.*

Thus, $cr(\mathcal{E}[\![t]\!]_{\mathcal{F}[\![P]\!]})$ is a suitable instantiation for $\mathcal{S}^{CR}[\![t]\!]$.

Let us show some examples of programs' semantics, which are represented by families of semantic trees indexed by most general calls (a function symbol applied to distinct variables). Edges in the semantic tree are labeled with the special variable that is instantiated with an expression (that may contain another special variable).

Example 32 (Example 27 continued)

The fixpoint semantics for the program P in Example 27 is the following:

$$\mathcal{F}[\![P]\!] = \begin{cases} \texttt{g x} \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \varepsilon \cdot \texttt{C}\varrho_1 \xrightarrow{\varrho_1} \{\texttt{x/A}\} \cdot \texttt{C A} \\ \texttt{g' x} \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{\texttt{x/A}\} \cdot \texttt{C A} \\ \texttt{h x} \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{\texttt{x/A}\} \cdot \texttt{A} \\ \texttt{f x y} \mapsto \varepsilon \cdot \varrho \xrightarrow{\varrho} \{\texttt{x/C x'},\texttt{y/B}\} \cdot \texttt{B} \end{cases}$$

In this case all the semantic trees have only one path. The symbol ε denotes the empty substitution. Since $\mathcal{F}[\![P]\!](\texttt{g x}) \neq \mathcal{F}[\![P]\!](\texttt{g' x})$ we know that it

exists a context which manifests different computed results. Indeed, within context `f` $[\cdot]$ x the two expressions have different computed results. Namely,

$$\mathcal{E}[\![\texttt{f (g x) x}]\!]_{\mathcal{F}[\![P]\!]} = \varepsilon \cdot \varrho \xrightarrow{\varrho} \{\texttt{x/B}\} \cdot \texttt{B}$$
$$\mathcal{E}[\![\texttt{f (g' x) x}]\!]_{\mathcal{F}[\![P]\!]} = \varepsilon \cdot \varrho$$

Note that $\varepsilon \cdot \varrho$ is not a computed result due to the occurrence of the ϱ variable.

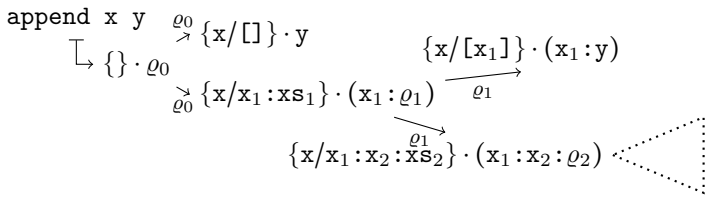
Even if this semantics is condensed (i.e., it does not contain redundant information), the trees in denotations can be infinite both in depth and in width. The following example shows a program whose semantics is infinite in depth.

Example 33

Consider the classical append function:

```
append [] y = y
append (x:xs) y = x:(append xs y)
```

The semantics of the function `append` in $\mathcal{F}[\![P]\!]$ is as follows



The dotted triangle in the figure denotes that the semantics has not been completely computed.

In order to define an effective method, we need to use an approximated (abstract) semantics (obtained by abstract interpretation [CC77, CC79]) for the computation of our inference method (in Section 6.3). A discussion about effectiveness and precision regarding the inference of specifications is given in Section 6.3.1.

6.3 Deriving Specifications from Programs

Now we are ready to describe the process of inferring specifications. The input of the process consists of the Curry program to be analyzed and two additional parameters: a *relevant* API, denoted Σ^r, and a maximum term size, *max_size*. The *relevant* API allows the user to choose the operations in the program that will be present in the inferred specification, whereas the

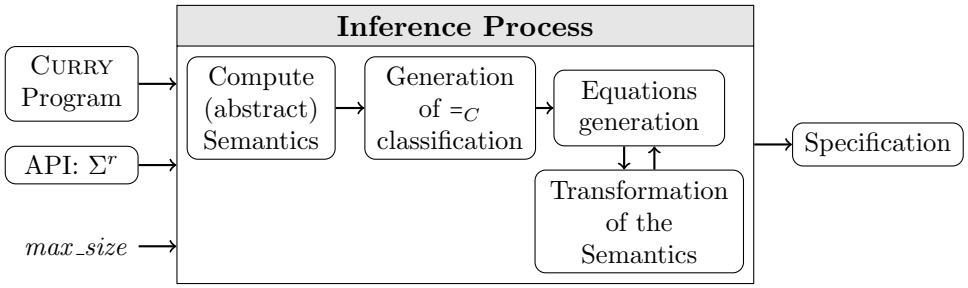


Figure 6.1: A general view of the inference process.

maximum term size limits the size of the terms in the specification. As a consequence, these two parameters tune the granularity of the specification, both making the process terminating and allowing the user to keep the specification concise and easy to understand. The output consists of a set of equations represented by equivalence classes of terms. Note that inferred equations may differ for the same program depending on the considered API and on the maximum term size. Similarly to other property-oriented approaches like [CSH10], the computed specification is complete up to terms of size max_size, i.e., it includes all the properties (relations) that hold between the operations in the relevant API and that are expressible by terms of size less than or equal to max_size.

The inference process consists of three phases, as depicted in Figure 6.1. First, (an approximation of) the semantics of the input program is computed. Second, a partition of terms, formed with functions from the relevant API of size less than or equal to the provided maximum size is computed. In our implementation, the size of a term is determined by its depth; however, the inference process is parametric w.r.t. the *size* function, thus other notions for term size are allowed (e.g., number of parameters, length, etc.). Each equivalence class of the partition contains terms that are equivalent w.r.t. the contextual equivalence $=_C$ defined in Section 6.2. Finally, the equations of the specification are generated: first, the equations of the contextual partition are computed, and then, by transforming the semantics, the equations corresponding to the other two notions of equivalence are computed as well.

In the following, we explain in detail the phases of the computation process by referring to the pseudo-code given in Algorithm 1. For the sake of comprehension, we present an untyped version of the algorithm. The actual one is a straightforward modification conformant w.r.t. types.

Let us start by presenting the data structure that represents the classifi-

cation of terms. A *partition* of terms consists of a set of *equivalence classes* (*ec*) formed by

- *sem*(*ec*): the semantics of (all) the terms in that class
- *rep*(*ec*): the *representative term* of the class, which is defined as the smallest term in the class (w.r.t. the function *size*), and
- *terms*(*ec*): the set of terms belonging to that equivalence class.

We represent each equivalence class by a triple $\langle sem(ec), rep(ec), terms(ec)\rangle$. The *representative term* is used in order to avoid much redundancy in the generation of equations. As we will see, the generation process is iterative, thus we generate first equations of smaller size, and then we increment the size until the size limit is reached. Instead of using every term of an equivalence class to build new terms of greater size, we just use the representative term. Now we are ready to describe the process.

Computation of the abstract semantics (and initial classification). The first phase of the algorithm, Lines 1 to 2 (in Algorithm 1), is the computation of the initial classification that is needed to compute the classification w.r.t. $=_C$. In order to make the method effective, it is based on the computation of an approximation of the semantics of the program P (the abstract semantics $\mathcal{F}^\alpha[\![P]\!]$). More details about the use of an abstract semantics can be found in Section 6.3.1.

More specifically, the *initial_part* function (Line 2) builds the initial classification which contains:

- one class for a free (logic) variable $\langle \mathcal{E}^\alpha[\![x]\!]_{\mathcal{F}^\alpha[\![P]\!]}, x, \{x\}\rangle$;[4]
- the classes for any built-in or user-defined constructor.

During the definition of the initial classification, it might occur that two terms, for instance $t_1 := f(x_1, \ldots, x_n)$ and $t_2 := g(y_1, \ldots, y_n)$, have the same semantics. If this happens, we do not generate two different classes, one for each term, but the second generated term is added to the class of the first one. For this particular example, we would have $\langle \mathcal{E}^\alpha[\![t_1]\!]_{\mathcal{F}^\alpha[\![P]\!]}, t_1, \{t_1, t_2\}\rangle$.

[4]The typed version of the inference method uses one variable for each type.

Algorithm 1 Inference of an algebraic specification

Require: Program P;
Program's *relevant* API Σ^r;
Maximum term size *max_size*

1. Compute $\mathcal{F}^\alpha[\![P]\!]$: the (abstract) semantics of P
2. $part \leftarrow initial\_part(\mathcal{F}^\alpha[\![P]\!])$
3. **repeat**
4. $part' \leftarrow part$
5. **for all** $f/n \in \Sigma^r$ **do**
6. **for all** $ec_1, \ldots, ec_n \in part$ such that at least one ec_i has been introduced in the previous iteration **do**
7. $t \leftarrow f(rep(ec_1), \ldots, rep(ec_n))$ where the $rep(ec_i)$ are renamed apart
8. **if** $t \notin part$ **and** $size(t) \leq max\_size$ **then**
9. $s \leftarrow \mathcal{E}^\alpha[\![t]\!]_{\mathcal{F}^\alpha[\![P]\!]}$: Compute the (abstract) semantics of term t
10. $add\_to\_partition(t, s, part')$
11. **end if**
12. **end for**
13. **end for**
14. **until** $part' = part$
15. $specification \leftarrow \varnothing$
16. $add\_equations(specification, part)$
17. **for all** $kind \in [CR, G]$ **do**
18. $part \leftarrow transform\_semantics(kind, part)$
19. $add\_equations(specification, part)$
20. **end for**
21. **return** *specification*

Generation of $=_C$ classification. The second phase of the algorithm, Lines 3 to 14, is the (iterative) computation of the classification of terms w.r.t. $=_C$. As mentioned before, this classification is also the basis for the generation of the other categories of equivalence classes.

We iteratively select all symbols f/n of the relevant API Σ^r (Line 5) [5] and n equivalence classes $ec_1, \ldots, ec_n$ from the current partition (Line 6) such that at least one ec_i was newly produced in the previous iteration. We build the term $t := f(t_1, \ldots, t_n)$, where each t_i is the representative term of ec_i, i.e., $t_i = rep(ec_i)$. In this way, by construction, the term t has surely not been

[5] Following the standard notation f/n denotes a function f of arity n.

considered yet; Then, we compute the semantics $s = \mathcal{E}^{\alpha}[\![t]\!]_{\mathcal{F}^{\alpha}[\![P]\!]}$ and update the current partition $part'$ by using the method $add\_to\_partition(t, s, part')$ (Lines 7 to 11). Here, the compositionality of the semantics makes possible to compute the semantics of terms (Line 9) efficaciously: since the semantics $s_i = sem(ec_i)$ for each term t_i is already stored in ec_i, then the computation of the semantics of t can be done in an efficient way just by *nesting* the semantics s_i into the semantics of $f(x_1, \ldots, x_n)$. This semantics nesting operation is the core of the $\mathcal{E}$ operation.[6]

The $add\_to\_partition(t, s, part)$ function looks for an equivalence class ec in the current classification $part$ whose semantics coincides with s. If it is found, then the term t is added to the set of terms in ec. Otherwise, a new equivalence class $\langle s, t, \{t\}\rangle$ is created.

If the partition suffers any modification during the current iteration (i.e., any term is added to the partition), then the algorithm iterates. This phase terminates eventually because at each iteration we consider, by construction, terms which are different from those already existing in the partition and whose size is strictly greater than the size of its subterms (but the size is bounded by max_size).

The following example illustrates how the iterative process works:

Example 34

Consider again the program of Example 26 and choose as relevant API the functions `and`, `or` and `not`. The following are the terms considered during the first iteration:

$$t_{1.1} := \texttt{not x}$$
$$t_{1.2} := \texttt{and x y}$$
$$t_{1.3} := \texttt{or x y}$$

Since the semantics of all these terms are different, three new classes are added to the initial partition. Thus, the partition at the end of the first iteration consists of four equivalence classes: the three ones corresponding to terms $t_{1.1}$, $t_{1.2}$ and $t_{1.3}$ and the equivalence class for the boolean free variable.

Then, during the second iteration, the following two terms (among others) are built

$$t_{2.1} := \texttt{and (not x) (not x')}$$
$$t_{2.2} := \texttt{not (or x y)}$$

[6] The interested reader can see [Bac12] for the technical details about the semantic operators.

More specifically, the term $t_{2.1}$ is built as the instantiation of $t_{1.2}$ with $t_{1.1}$ (in both arguments), and the term $t_{2.2}$ is the instantiation of $t_{1.1}$ with $t_{1.3}$. The semantics of these two terms is the same, but it is different from the semantics of the existing equivalence classes. Therefore, during this iteration (at least) a new class ec_1 for this new semantics is added, having as representative the term $t_{2.2}$ (i.e., $rep(ec_1) = t_{2.2}$).

From this point on, only the representative of the class will be used for constructing new terms. This means that terms like `not (and (not x) (not x'))`, which is the instantiation of $t_{1.1}$ with $t_{2.1}$, are never built since only $t_{2.1}$ can be used.

We recall here that, thanks to the closedness w.r.t. context of the semantics, this strategy for generating terms is *safe*. In other words, when we avoid to build a term, it is because it is not able to produce a behavior different from the behaviors associated to the existing terms, thus we are not losing completeness.

Generation of the specification. The third phase of the algorithm (Lines 15 to 20) constructs the specification for the provided CURRY program. First, Line 16 computes the $=_C$ equations from the current partition. Since we have avoided much redundancy thanks to the strategy used to generate the equivalence classes, the *add_equations* function needs only to take each equivalence class with more than one term and generate equations for these terms.

This function generates also a side effect on the equivalence classes that is needed in the successive steps. Namely, it modifies the third component of the classes so that it replaces the (non-singleton) set of terms with a singleton set containing just the representative term.

Then, Lines 17 to 20 compute the equations that correspond to the remaining equivalence notions defined in Section 6.2. Let us explain in detail the case for the computed result equations (kind CR). As already noted, from the (tree) semantics T in the equivalence classes computed during the second phase of the algorithm, it is possible to construct (by losing the tree internal structure and collecting just the computed result leaves $cr(T)$) the semantics that models the computed result behavior. Therefore, we apply this transformation to the semantic values of each equivalence class. After the transformation, some of the equivalence classes which had different semantic values may now collapse into the same class. This transformation and reclassification is performed by the *transform_semantics* function. The resulting (coarser) partition is then used to produce the $=_{CR}$ equations by an application of *add_equations*.

Thanks to the fact that *add_equations* ends with a partition made of just singleton term sets, we cannot generate (again) equations $t_1 =_{CR} t_2$ when an

equation $t_1 =_C t_2$ had been already issued.

Let us clarify this third phase by an example.

Example 35

Assume we have a partition consisting of three equivalence classes with semantics s_1, s_2 and s_3 and representative terms t_{11}, t_{22} and t_{31}:

$$ec_1 = \langle s_1, t_{11}, \{t_{11}, t_{12}, t_{13}\}\rangle$$
$$ec_2 = \langle s_2, t_{22}, \{t_{21}, t_{22}\}\rangle$$
$$ec_3 = \langle s_3, t_{31}, \{t_{31}\}\rangle$$

The *add_equations* procedure generates the equations

$$\{\ t_{11} =_C t_{12} =_C t_{13},\quad t_{21} =_C t_{22}\ \}$$

and, as a side effect, the partition becomes

$$ec_1 = \langle s_1, t_{11}, \{t_{11}\}\rangle$$
$$ec_2 = \langle s_2, t_{22}, \{t_{22}\}\rangle$$
$$ec_3 = \langle s_3, t_{31}, \{t_{31}\}\rangle$$

Now, assume that $cr(s_1) = x_0$ and $cr(s_2) = cr(s_3) = x_1$. Then, after applying *transform_semantics*, we obtain the new partition

$$ec_4 = \langle x_0,\ t_{11},\ \{t_{11}\}\rangle$$
$$ec_5 = \langle x_1,\ t_{22},\ \{t_{22}, t_{31}\}\rangle$$

Hence, the only new equation is $t_{22} =_{CR} t_{31}$. Indeed, equation $t_{11} =_{CR} t_{12}$ is uninteresting, since we already know $t_{11} =_C t_{12}$ and equation $t_{21} =_{CR} t_{31}$ is redundant (because $t_{21} =_C t_{22}$ and $t_{22} =_{CR} t_{31}$).

In summary, if $t_1 =_C t_2$ holds, then $t_1 =_{\{CR,G\}} t_2$ are not included in the computed specification.

The same strategy can be used to generate also the $=_G$ equations. Conceptually, this could be done with a semantic transformation which replaces each free variable in every computed result with all its ground instances. In practice, we can use a completely dual approach, where we use a variable to represent all its possible ground instances. The transformation which corresponds to this representation is as follows:

- it retains only the most general instances of the original semantics (removing computed results which are instances of others) and,
- it replaces a set of computed results R with its least general anti-instance r when appropriate. This happens when the set of all ground instances of r is the same as that of the union of all ground instances of all elements in R. This can be implemented by checking if we have a set with all the constructors of a given type (applied to free variables), then, we replace the set of constructors by a free variable and then we repeat the process until we reach a fix point.

In this way the semantics is transformed by removing further internal structure and again classes may collapse and new equations (w.r.t. $=_G$) are generated.

6.3.1 Pragmatical considerations

In a semantic-based approach, one of the main problems to be tackled is effectiveness. The semantics of a program is in general infinite and thus we use abstract interpretation [CC77] in order to have a terminating method. More specifically, in this work we use a correct abstraction of the semantics of [Bac12, BC11] over the $depth(k)$ abstract domain (also introduced in [Bac12]). In the $depth(k)$ abstraction, terms (occurring in the nodes of the semantic trees) are "cut" at depth k by replacing them with *cut variables*, distinct from program variables. Hence, for a given signature Σ, the universe of abstract semantic trees is finite (although it increases exponentially w.r.t. k). Therefore, the finite convergence of the computation of the abstract semantics is guaranteed.

The presence of cut variables in the nodes of the abstract semantics denotes that the (partial) computed result has been abstracted. However, if no cut variable occurs in a node, we know that it coincides with a node in the concrete semantics. Thanks to this structure, $depth(k)$ semantics is technically an over approximation of the semantics, but simultaneously it can be very precise (concrete) when computed results show up without "cuts".

Therefore, equations coming from equivalence classes whose $depth(k)$ semantics does not contain cut variables are *correct* equations, while for the others we do not know (yet). If we use a bigger k, the latter can definitively become valid or not. Thus, equations involving approximation are equations that *have not been falsified up to that point*, analogously to what happens in the testing-based approach. We call these equations *unfalsified* equations. When showing the specification, we mark unfalsified equations with a special equivalence symbol $=^{\alpha}_{C}$. Unfalsified equations are the only kind of equations that testing-based approaches can compute in general.

The main advantage of our proposal w.r.t. the testing-based approaches is the fact that we are able to distinguish when an equation *certainly* holds, and when it just *can* hold. Moreover, we can deal with non terminating programs.

Since the overall construction is (almost) independent of the actual structure of the abstract semantics, it would be possible in the future to use other abstract domains to reach a better trade-off between efficiency of the computation and accuracy of the specifications.

6.4 Case Studies

Let us illustrate our methodology with a more elaborated example. The following CURRY program implements a two-sided queue where it is possible to insert or delete elements on both left and right sides:

```
data Queue a = Q [a] [a]

new = Q [] []

inl x (Q xs ys) = Q (x:xs) ys
inr x (Q xs ys) = Q xs (x:ys)

outl (Q [] ys) = Q (tail (reverse ys)) []
outl (Q (_:xs) ys) = Q xs ys

outr (Q xs []) = Q [] (tail (reverse xs))
outr (Q xs (_:ys)) = Q xs ys

null (Q [] []) = True
null (Q (_:_) _) = False
null (Q [] (_:_)) = False

eqQ (Q xs ys) (Q xs' ys') =
  (xs++reverse ys) =:= (xs'++reverse ys')
```

The queue is implemented by means of two lists where the first list corresponds to the first part of the queue and the second list is the second part of the queue reversed. The `inl` function adds the new element to the head of the first list, whereas the `inr` function adds the new element to the head of the second list (the last element of the queue). The `outl` (`outr`) function drops one element from the left (right) list, unless the list is empty, in which case it reverses the other list and then swaps the two lists before removal. If we include all the

functions in the API and by assuming $k \geq 3$ for the abstraction, an excerpt of the inferred specification for this program is the following one:

$$\texttt{null new} =_C \texttt{True} \tag{6.8}$$

$$\texttt{new} =_C \texttt{outl (inl x new)} =_C \texttt{outr (inr x new)} \tag{6.9}$$

$$\texttt{outl (inl x q)} =_C \texttt{outr (inr x q)} \tag{6.10}$$

$$\texttt{outr (inl x new)} =_C \texttt{outl (inr x new)} \tag{6.11}$$

$$\texttt{inr x (inl y q)} =_C \texttt{inl y (inr x q)} \tag{6.12}$$

$$\texttt{inl x (outl (inl y q))} =_G^{\alpha} \texttt{outr (inl x (inr y q))} \tag{6.13}$$

$$\texttt{outl (inl x (outl q))} =_G^{\alpha} \texttt{outl (outl (inl x q))} \tag{6.14}$$

$$\texttt{outr (outl (inl x q))} =_C \texttt{outl (inl x (outr q))} \tag{6.15}$$

$$\texttt{null (inl x new)} =_C \texttt{null (inr x new)} =_C \texttt{False} \tag{6.16}$$

$$\texttt{eqQ (inr x new) y} =_C \texttt{eqQ (inl x new) y} \tag{6.17}$$

We can see different kinds of equations in the specifications. The asymmetry in the definition of the queue makes that Equation (6.13) holds only for ground instances. Moreover, the semantics for terms in Equations (6.13) and (6.14) is abstracted (in fact, the semantic tree is infinite for these cases). Equations (6.9), (6.10) and (6.11) state that adding and removing one element produces always the same result independently from the side in which we add and remove it. Equations (6.12), (6.13), (6.14) and (6.15) show a sort of *restricted* commutativity between functions. Finally, Equation (6.17) shows that, w.r.t. the user defined predicate `eqQ`, that identifies queues which contain the same elements, `inr x new` is equivalent to `inl x new`, although the internal structure of the queue differs.

It is worth noticing that the unfalsified equations for the Queue example (Equations (6.13) and (6.14) above), represent properties that actually hold for the program. However, it might occur that unfalsified equations correspond to *false* properties of the program, as the following example shows.

Consider the following program that computes the double of natural numbers in Peano notation:

```
data Nat = Z | S Nat

double, double' :: Nat -> Nat
double Z = Z
double (S x) = S (S (double x))
```

```
double' x = plus x x

plus :: Nat -> Nat -> Nat
plus Z y = y
plus (S x) y = S (plus x y)
```

Some of the inferred equations for the program are:

$$\texttt{double' (double' (double x))} \\ \quad =^{\alpha}_{CR} \texttt{double' (double (double' x))} \tag{6.18}$$

$$\texttt{double x} =^{\alpha}_{CR} \texttt{double' x} \tag{6.19}$$

$$\texttt{double' (double' x)} \\ \quad =^{\alpha}_{CR} \texttt{double' (double x)} \\ \quad =^{\alpha}_{CR} \texttt{double (double' x)} \\ \quad =^{\alpha}_{CR} \texttt{double (double x)} \tag{6.20}$$

$$\texttt{(S (double x))} =^{\alpha}_{CR} \texttt{double (S x)} \tag{6.21}$$

$$\texttt{plus (double x) y} =^{\alpha}_{CR} \texttt{plus (double' x) y} \tag{6.22}$$

We can observe that, in this case, all the equations are unfalsified due to the nature of the example. Moreover, all equations hold with the $=_{CR}$ relation. This is due to the asymmetry in the definition of the two versions of double: although the computed results of both versions are the same, there exist contexts in which the terms behave differently. This characteristic of the program is not easy to realize by just looking at the code, thus this is an example of the usefulness of having different notions of equivalence.

Finally, Equation (6.21) is an unfalsified equation that states a property which is false in the program. This is due to the approximation of the abstraction. It is worth noting that we would need to completely compute the (infinite) concrete semantics in order to discard the equation from the specification.

We do not remove unfalsified equations from the specifications since they have their own interest. Although it might be unfeasible to guarantee correctness of some equations (as in the example above), unfalsified equations may nevertheless show behaviors of the program which are actually correct. As mentioned on page 111, this is the only possibility in testing-based approaches, where all the equations must be considered unfalsified since it is impossible to distinguish them from correct equations. In any case, we can try to prove correctness of these equations by using a complementary verification or validation technique.

6.4.1 AbsSpec: The prototype

We have implemented the basic functionality of this methodology in the prototype AbsSpec that is written in Haskell. The core of the AbsSpec prototype[7] consists of about 800 lines of code implementing the tasks of generating and classifying terms. The inference core of AbsSpec is generic w.r.t. the abstract domain, i.e., the operations implementing the abstract domain are passed to the generic inference process. Note that the AbsSpec prototype invokes the semantics' prototype implementation, which consists of about 7500 additional lines of code. On top of the core part of the prototype, the interface module implements some functions that allow the user both to check if a specific set of equations hold, or to get the whole specification. It is worth noting that, although in this chapter we consider as input Curry programs, the prototype also accepts programs written in (the first order fragment of) Haskell (which are automatically converted by orthogonalization into Curry equivalent programs).

Unfortunately, we do not know of sets of benchmarks in the literature to be used to evaluate the prototype. Hence, we wrote some examples as a proof of concept in order to get some impressions on the efficacy of our proposal. Since the prototype does not handle built-in arithmetic operators yet, we tested it on both Curry and Haskell programs which do not involve arithmetics (mainly implementation of abstract data structures like queues, binary trees, arrays, heaps, *etc.*).

The experiments were conducted on an machine with an Intel Core2 Quad CPU Q9300 (2.50GHz) and 6 Gigabytes of RAM. AbsSpec was compiled with version 6.12.3 of the Glorious Glasgow HASKELL Compilation System (GHC). Table 7.1 shows the number of computed equivalence classes together with the execution times for the inference of each program example with some additional information.

Column *Program* shows the name of the example program. The first three cases correspond to the examples shown in this chapter. The fourth example is a more elaborated logic example where a data structure representing formulas is defined; column *Rules* shows the number of rules defining the program; column *API size* shows the number of operations included in the *relevant* API for the experiment. For the Boolean example, the experiment includes the operator for the logic implication defined explicitly (not in terms of the other boolean operators); column *Terms* shows the number of terms generated (thus, whose semantics is computed) during the inference process; columns $\#_{=_{\mathbf{C}}}$, and $\#_{=_{CR}}$ show, respectively, the number of $=_C$, and $=_{CR}$ equivalence classes

[7] Available at http://safe-tools.dsic.upv.es/absspec.

Program	Rules	API size	Term size	Terms	$\#_{=_C}$	$\#_{=_{CR}}$	Time
Boolean	9	5	2	121	65	1	0.13s
			3	2549	410	1	6.55s
			4	6399	1378	1	42.32s
Queue	11	6	2	34	1	0	0.08s
			3	132	12	1	0.18s
			4	473	56	8	0.58s
Double	5	3	2	25	0	5	0.11s
			3	676	55	157	19.86s
			4	2638	410	344	43.71s
Formula	8	4	2	42	0	0	m 53.98s
			3	1648	2	0	9m 0.14s
			4	-	-	-	-

Table 6.1: Inference process of example programs

with more than one term that have been generated.[8]

Our preliminary experiments show that many interesting properties hold over the $depth(k)$ domain with low k values (we run the prototype with depth 7 by default). Also, many interesting properties show up with $max\_size = 3$. For example, we can see that for the Queue example, with $max\_size = 2$, only one equivalence class is defined whereas for $max\_size = 3$, 13 (sequences of) equations belong to the specification. We have used also $max\_size = 4$, but specifications tend to be less comprehensible for the user (64 equivalence classes for the same Queue example). Hence, increasing this value should be done only when bigger terms make sense, being at the same time very careful in choosing a sufficiently small API. The Double example illustrates the usefulness of the $=_{CR}$ equations: with $max\_size = 2$ we have already five of these equations.

The last example illustrates the fact that, for a data structure with a complex semantics, the high number of generated terms penalizes the inference process. Not that the increase of the number of generated terms depends, not only on the number of elements included in the relevant API (and on the number of arguments of the functions in the API), but also on the semantics of the program. Intuitively, if we have a program in which many terms share the same semantics, then fewer terms will be generated.

[8]The prototype does not compute the $=_G$ equations yet.

6.5 Conclusions and Related Work

This chapter presents a method to automatically infer high-level, property-oriented (algebraic) specifications in a functional logic setting. A specification represents relations that hold between operations (nested calls) in the program.

The method computes a concise and clear specification of program properties from the source code of the program. These specifications are useful for the programmer in order to detect possible errors, or to check that the program corresponds to the intended behavior.

The computed specification is particularly well suited for program understanding since it allows to discover non-evident behaviors, and also to be combined with testing. In the context of (formal) verification, the specification can be used to ease the verification tasks, for example by using the correct equations as annotations, or unfalsified equations as candidate axioms to be proved.

Our inference approach relies on computing an approximation of the program semantics. Therefore, to achieve effectiveness and good performance results, we use a suitable abstract semantics instead of the concrete one. This means that we may not guarantee correctness of all the equations in the specification, but we can nevertheless effectively infer correct equations thanks to a good compromise between correctness and efficiency.

We have developed a prototype that implements the basic functionality of the approach. We are working on the inclusion of all the functionality described in this chapter.

To the best of our knowledge, our methodology is the first proposal for specification synthesis in the functional logic setting. There is a testing tool, called EasyCheck [CF08], in which specifications are *used* as the input for the testing process. Given the target properties, EasyCheck executes ground tests in order to check whether each property holds. This tool could be used as a companion tool of ours in order to check if the unfalsified $=_G$ equations can be actually falsified. However EasyCheck is not capable of checking the $=_C$ and $=_{CR}$ equations because it is based only on the execution of ground tests.

QuickSpec [CSH10] computes an algebraic specification for Haskell programs by means of (almost black-box) testing. Like our approach, its inferred specifications are complete up to a certain depth of the analyzed terms because of its exhaustiveness. However, all the equations in their specification are unfalsified, i.e., the specification may be incorrect due to the use of testing for the equation generation. In contrast, we follow a (glass-box) semantic-based approach that allows us to compute specifications as complete as those of QuickSpec, but with correctness guarantees on a part of them (depending

on the abstraction). The performance of QuickSpec is generally better than that of our prototype for similar programs. However, we have to recall that our purpose is more ambitious since, for the case of functional-logic languages, using just the ground equivalence is not enough: important behaviors regarding the loss of referential transparency would not show up, as shown by the `double` example. Moreover, our method is the only one that deals with non terminating programs.

7

Inference of Specifications from C Programs

Symbolic execution (SE) is a well-known program analysis technique that allows the program to be executed using *symbolic* input values instead of actual (concrete) data so that it executes the program by manipulating program expressions involving the symbolic values [Kin76, PV09]. Intuitively, symbolic execution means that each data structure field and program variable initially hold a symbolic value. Then, each program statement execution can update the configuration cells (such as *env*, *heap* and *cfg* in the examples of Section 5.2) by mapping fields and variables to (symbolic) values represented as relational expressions. Recently, SE has found renewed interest due in part to the advances in new algorithmic developments and decision procedures.

The technique presented in this chapter relies on SE to automatically infer high-level, formal specifications for C-like, heap-manipulating code by using the notation of *Matching Logic* (**ML**), which is a novel program verification foundation that is built upon operational semantics [Rc11].

The key idea behind our method is as follows. In order to infer the specification for a given procedure, we use the **ML** machinery to symbolically execute it and to collect its pre/post-conditions. These pre/post-conditions represent the value of each program variable and field of the post-state in terms of the values of program variables and fields of the pre-state. We apply our technique to the KernelC language [RSŞ09], which is a non-trivial fragment of C that includes functions, structures, pointers and I/O primitives. The symbolic execution of KernelC programs is supported in **ML** by using the MatchC verifier, which has several applications including the verification of functional correctness and static detection of runtime errors [Rc11]. MatchC is implemented using the **K** semantic framework [RŞ10], which compiles into the high-performance programming language Maude [CDE+07].

The proposed inference technique relies on the classification scheme developed in [LG86] for data abstractions in general, where a function (method) may be either a *constructor*, *modifier* or *observer*. A constructor returns a new object of the class from scratch (i.e., without taking the object as an input parameter). A modifier alters an existing class instance (i.e., it changes the state of one or more of the data attributes in the instance). An observer inspects the object and returns a value characterizing one or more of its state attributes. We do not assume the traditional premise of the original classification in [LG86] that states that observer functions do not cause side effects on the state. This is because we want to apply our technique to any program, written by third-party software producers that may not follow the observer purity discipline.

Our symbolic analysis of KernelC programs allows us to explain the execution of a *modifier* function m by using other (observer) routines in the program. Starting from an initial symbolic state s^0, we first evaluate symbolically m on s_0 obtaining as a result a set of pairs (s_i^0, s_i^f) of refined initial and final symbolic states, respectively. In order to compute suitable explanations for the routine m, we symbolically evaluate the observer methods on each state s_i^0 and s_i^f so that when the observer returns the same value at the end of each of its branches, then we can conclude that the observer is a (partial) observational abstraction or explanation of the constraints in the state. For each pair of refined initial and final states, a pre/post statement is synthesized where the precondition is expressed in terms of the observers that *explain* the initial state s_i^0, whereas the postcondition contains the observers that *explain* the final state s_i^f. Then, the synthesized pre/post axioms that abstract from any implementation details are further simplified (to be given a more compact representation), and are eventually presented in a more friendly sugared form.

The main contributions of this inference technique are as follows:

- A new lightweight approach to extract high-level specifications from heap-manipulating code that consists of a symbolic analysis that explores and summarizes the behavior of a *modifier* program routine by using other available routines in the program, called *observers*.
- Correctness conditions for our specification discovery technique;
- A practical demonstration that the technique is capable of extracting accurate specifications from nontrivial procedures and functions;
- An implementation of the framework that targets KernelC programs and uses **K** as an intermediate language to translate KernelC to **ML** constraints.

Specification inference takes advantage of the unsat core generated by the SAT solver CVC3 [BT07] that is coupled to **ML**.

This technique improves existing approaches in the literature in several ways. On one hand, our technique is the first approach that can automatically ensure delivery of correct specifications, which is done by using the same MatchC verifier that we use for the specification discovering. On the other hand, since our approach relies on the **K** semantics specification of KernelC, the methodology developed in this work can be easily extended to cope with any language for which a **K** semantics is given, like Java 1.4, Scheme and Verilog [RŞ10]. There is an executable formal semantics for C that describes the semantics of the whole C99 standard, and it will be possible to use it in our framework as soon it is coupled into the MatchC verifier.

In the rest of the chapter, we first introduce in Section 7.1 the running example that we use throughout this chapter to illustrate the technique. The running example also allows us to outline the major research problems addressed. Section 7.2 describes how we extended the symbolic machinery of the MatchC verifier in order to support our inference approach. Section 7.3 introduces two algorithms that mechanize the inference technique and provides some experimental results. Section 7.4 illustrates the two algorithms previously introduced by applying them to the inference of a specification for the running example. Finally, Section 7.5 discusses the related work and concludes.

7.1 Specification Discovery

A logic specification is a logical relation between inputs and outputs of a program. Specification discovery is the task of inferring high-level specifications that closely describe the program behavior. Obviously, these specifications can only be correct with respect to user intent if the original program is correct itself. But even if it is not correct, the ascertained specification can still be very helpful in several important scenarios such as improving program understanding, synthesizing test units, and helping the programmer to debug the code.

Given a program `P`, the specification discovery problem for `P` is typically described as the problem of inferring a likely specification for every function `m` in `P` that uses I/O primitives and/or modifies the state of encapsulated, dynamic data structures defined in the program. Following the standard terminology, any such function `m` is called a *modifier*. The specification for `m` is to be cleanly expressed by using any combination of the non-modifier functions of

P, i.e., functions, called *observers*, that inspect the program state and return values expressing some information about the encapsulated data. However, because the C language does not enforce data encapsulation, we cannot assume purity of any function: every function in the program can potentially change the execution state, including the *heap* component of the state. In other words, any function can potentially be a *modifier*. As a consequence, we simply define an *observer* as any function whose return type is different from `void`, hence, potentially expressing an observed property regarding the value of the function arguments or the contents in the *heap*.

The following example introduces the case that we use as a running example throughout this chapter.

Example 36

The program in Figure 7.1 implements an abstract datatype for representing sets. A set is internally represented in KernelC as a data structure (`struct set`) that contains a pointer (`struct lnode *`) to a list of elements (field `elems`), together with the number of elements in the set (field `size`) and the maximum number of elements that may contain (field `capacity`).

The `new` function allocates memory for storing a `struct set` data structure with initial size `0`, the capacity given by the input value for the `capacity` parameter, and the `NULL` value for the pointer that references the list of elements in the set. Upon completion, it returns the address of the allocated structure.

A call `add(s,x)` to the `add` function proceeds as follows: it first checks that the pointer `s` to the set is different from `NULL`; next, it checks that the size of `s` is lower than its capacity; and then, it checks that `x` is not an element of `s` yet. Provided all these conditions hold, it allocates a new list node (`struct lnode`) `*new_node` whose first element is `x` and that is followed by the list of elements representing the original set; finally, it increases the size of the set by 1. If the insertion operation `add` succeeds, the call returns `1` once the new element has been added to the list; otherwise, it returns `0` (standing for unsuccessful insertion).

Function `isfull` returns `1` if the size of the set argument `s` is greater than or equal to its capacity; otherwise, it returns `0`. Similarly, `isnull` returns `1` if the address of the set argument is `NULL`; it returns `0` otherwise. Finally, the execution of `contains(s,x)` returns `1` if the argument element `x` belongs to `s` and returns `0` otherwise.

Technically, the inferred specification for a given function consists of a set of implication formulas of the form $t_1 \Rightarrow t_2$ where t_1 and t_2 are conjunctions of equations of the form $l = r$. Each left-hand side l can be either

```
#include <stdlib.h>

struct lnode {
 int value;
 struct lnode *next;
};

struct set {
 int capacity;
 int size;
 struct lnode *elems;
};

struct set* new(int capacity)
{
 struct set *new_set;

 new_set =
  (struct set*) malloc(sizeof(
      struct set));
 if (new_set == NULL)
  return NULL; /* no memory
      left */

 new_set->capacity = capacity;
 new_set->size = 0;
 new_set->elems = NULL;

 return new_set;
}

int add(struct set *s,int x)
{
 struct lnode *new_node;
 struct lnode *end_node;
 struct lnode *n;

 if (s == NULL)
  return 0; /* NULL set */

 if (s->size >= s->capacity)
  return 0; /* no space left */

 n = end_node = s->elems;
 while (n != NULL) {
  if (n->value == x)
   return 0; /* element already
       added */
  end_node = n;
  n = n->next;
 }

 /* Initialize new node */
 new_node =
  (struct lnode*) malloc(sizeof
      (struct lnode));
 if (new_node == NULL)
  return 0; /* no memory left
      */
 new_node->value = x;
 new_node->next = s->elems;

 /* Link new node */
 s->elems = new_node;

 /* Update set info */
 s->size += 1;

 return 1; /* element added */
}

int isfull(struct set *s)
{
 if (s == NULL)
  return 0; /* NULL set
      provided */
 if (s->size >= s->capacity)
  return 1; /* is full */
 return 0; /* is not full */
}

int isnull(struct set *s)
{
 if (s == NULL)
  return 1;
 return 0;
}

int contains(struct set *s,int
    x)
{
 struct lnode *n;

 if (s == NULL)
  return 0; /* NULL set */

 n = s->elems;
 while (n != NULL) {
  if (n->value == x)
   return 1; /* element found
       */
  n = n->next;
 }

 return 0; /* element NOT found
      */
}
```

Figure 7.1: KernelC implementation of a set with linked lists.

```
isnull(s) = 1          ⇒   ret = 0 ∧ isnull(s') = 1

isfull(s) = 1          ⇒   ret = 0 ∧
                           contains(s,x) = contains(s',x)

contains(s,x) = 1      ⇒   ret = 0 ∧ contains(s',x)= 1

isnull(s) = 0 ∧            ret = 1 ∧
isfull(s) = 0 ∧        ⇒   isnull(s') = 0 ∧
contains(s,x) = 0          contains(s',x) = 1
```

Figure 7.2: Inferred specification for the `add` function.

- a call to an observer function and then r represents the return value of that call;
- the label `ret`, and then r represents the value returned by the modifier function being observed.

Informally, the statements at the left-hand and right-hand sides of the symbol ⇒ are respectively satisfied before and after the execution of the function call. We adopt the standard primed notation for representing variable values before and after execution. For instance, given a variable `s` that stands for the value of the parameter `s` before the function is executed, the primed version `s'` stands for the value after the execution.

Example 37

Consider again the program of Example 36. The specification for the (modifier) function `add(s,x)` (that inserts an element `x` in the set `s`) is shown in Figure 7.2.

The specification consists of four implications stating the conditions that are satisfied before and after the execution of the considered function. For instance, the first formula can be read as follows: if the result of running `isnull(s)` is equal to `1` before executing `add(s,x)`, then the value returned by the call `add(s,x)` is `0`, and, after its execution, the outcome of `isnull(s')` is also `1`.

Even though the observers `isnull` and `isfull` behave as boolean functions (predicates) in this example, we prefer not to write them in sugared relational form (i.e., `isfull(s)` instead of `isfull(s)=1`) since a specific datatype for Boolean numbers does not exist in C. Hence, even when we can detect that the observer function only returns two scalar values, say `0` and `1` as in the example, we cannot give it the semantics of a logical predicate.

Note that any implication formula in the inferred specification may contain multiple facts (in the pre or post-condition) that refer to function calls that are assumed to be run independently under the same initial conditions. This avoids making assumptions about the function purity or side-effects.

Our technique for inferring specifications relies on the symbolic execution engine of the *Matching Logic* verifier MatchC. MatchC works in a forward manner by symbolically executing an **ML** pattern that is provided as the program precondition, and non-deterministically obtaining a set of final patterns that are then used to discharge the postcondition. This is an instance of a general strategy to calculate the strongest postcondition of a predicate transformation semantics as explained in [GC10]. However, MatchC is incomplete for the purpose of general symbolic execution in the sense that its symbolic machinery does not support incremental assumptions regarding the initial structure of the program memory; it can only assume the structure that is implicitly imposed by the initial pattern. For the inference purposes of this chapter, we cannot assume any *ex-ante* condition for the initial program state; on the contrary, we need to incrementally collect all the assumptions that allow each symbolic execution path to be successfully executed. In the following section, we explain how we extended MatchC to support collecting assumptions on-the-fly within the symbolic configurations as needed.

7.2 Extending the ML Symbolic Machine

Symbolic execution typically proceeds like standard execution except that, when a function or routine is called, symbolic values are assigned to its actual parameters and computed values become symbolic expressions that record the operations applied to them. When symbolic execution reaches a conditional control flow statement, every possible execution path from this statement must be explored. In order to keep track of the explored execution paths, symbolic execution also records the assumed (symbolic) conditions on the program inputs that determine each execution path in the so-called *path constraints* (one per possible branch), which are empty at the beginning of the execution. A path constraint consists of the set of constraints that the arguments of a given function must satisfy in order for a concrete execution of the function to follow the considered path. Without loss of generality, we assume that the symbolically executed functions access no global variables; they could be easily modeled by passing them as additional function arguments.

Example 38

Consider again the `add` function of Example 36. Assume that the input values for the actual parameters `s` and `x` are the symbolic values `s` and `x`, respectively. Then, when the symbolic execution reaches the first `if` statement in the code, it explores the two paths arising from considering both, the satisfaction and non satisfaction of the guard in the conditional statement. The path constraint of the first branch is updated with the constraint `s = NULL`, whereas `s ≠ NULL` is added to the path constraint of the second branch.

To summarize, symbolic execution can be represented as a tree-like structure where each branch corresponds to a possible execution path and has an associated path constraint. When the path constraint is satisfiable, the *successful* path ends in a final (symbolic) configuration that typically stores a (symbolic) computed result.

For the symbolic execution of C programs, we must pay attention to pointer dereference and initialization. In C, a structured datatype (`struct`) is an aggregate type that describes a nonempty set of sequentially allocated member objects[1], called fields, each of which has a name and a type. When a `struct` value is created, C uses the address of its first field to refer to the whole structure. In order to access a specific field `f` of the given structure type, C computes `f`'s address by adding an offset (the sum of the sizes of each preceding field in the definition) to the address of the whole structure. In our symbolic setting, all the pointer arithmetic is done by means of symbolic address expressions that may appear in (the domain of) heap cells of MatchC patterns.

Example 39

(Example 38 continued) Consider the second `if` statement of the `add` function given in Example 36. The evaluation of the guard of the conditional statement requires accessing both `p->size` and `p->capacity`. Since `capacity` is the first field in the `struct set` definition, its location coincides with the (base) address `p`. However, in order to access `p->size`, its address must be computed by adding an offset[2] of `1` to the (base) address `p` (i.e., if we assume that the symbolic address held by the variable `p` is `p`, then the computed address for the field `size` is `p+1`.)

[1] An object in C is a region of data storage in the execution environment.

[2] We assume that the memory is indexed by words and that a value of type `int` has the size of a word.

Another critical point is the *undefinedness* problem that occurs in C programs when accessing uninitialized memory addresses. The KernelC semantics that we use preserves the concrete *well-definedness* behavior of pointer-based program functions of C while still detecting the *undefinedness* cases in a way similar to the C operational semantics of [ER12]. However, in the discovery setting of our approach, we have no a priori information regarding the memory (in particular, information about the (un)initialized memory addresses). Therefore, when symbolic execution accesses (potentially uninitialized) memory positions, two cases must be considered: the case in which the memory is actually initialized, and the case in which it is not. In the second case, the symbolic execution gets stuck, thus identifying *undefined* behavior as in [ER12]. For the case in which the memory positions are actually initialized and execution should proceed, a strategy to reconstruct the original object in memory is needed. We adapt to our setting the lazy initialization of fields of [KPV03]: when a symbolic address (or address expression) is accessed for the first time, SE initializes the memory object that is located at that address with a new symbolic value. This means that the mapping in the heap cell is updated by assigning a new free variable to the symbolic address of the accessed field so that from that point on, accesses to that field can only succeed. In contrast, in the case of failure, an `undefined` computation is pushed onto the k cell, which stops the execution.

Example 40

(Example 39 continued) Before executing the second `if` statement, assume that the heap cell is empty, which means that nothing is known about the structure of the heap cell. After symbolically executing the guard of the `if` statement (which refers to the `capacity` field and `size` field of the structured type in `s`), the heap cell has the form:

$$[\mathsf{s} \mapsto \mathsf{s.capacity}, \mathsf{s}+1 \mapsto \mathsf{s.size}]_{\mathsf{heap}}$$

In other words, new symbolic bindings regarding the actual parameters are added, which represent the assumptions we made over the corresponding data structures.

In the following, we augment MatchC symbolic configurations (MatchC patterns) with new cells and naturally extend its symbolic execution machinery to work with the augmented patterns.

$$\langle \frac{\mathrm{load}(T, I)}{\mathrm{tv}(T, \mathsf{NewFreeVar})} \cdots \rangle_{\mathsf{k}} \langle \mathit{Heap} \frac{}{I \mapsto \mathsf{NewFreeVar}} \rangle_{\mathsf{heap}} \langle \cdots \frac{}{I \mapsto \mathsf{NewFreeVar}} \cdots \rangle_{\mathsf{iheap}}$$

$$\text{if } I \notin \mathrm{keys}(\mathit{Heap})$$

Figure 7.3: Symbolic execution rule for accessing a value in the memory.

7.2.1 The MatchC Extension

The heap cell of MatchC patterns cannot be used to keep track of the assumptions made by the lazy initialization described above since the subsequent assignment statements that may occur during the symbolic execution typically overwrite the heap cell values. Therefore, we have extended MatchC patterns by introducing two additional cells: the iheap and ik cells. The iheap cell monotonically stores all the (structural) assumptions that are dynamically made for the *initial* heap. In this way, when symbolic execution finishes, the iheap cell together with the ϕ cell contain the path constraint for the symbolic parameters that point to the dynamic data structures that were accessed along the branch. The ik cell stores the contents of the k cell when the symbolic execution starts. In our case, this cell always contains a symbolic (initial) function call.

The following example illustrates how the iheap cell is used. It also illustrates how the KernelC rules have been conservatively augmented to manipulate extended configurations thanks to the modularity and underlying type structure of the **K** framework.

Example 41

The rule in Figure 7.3 states that, whenever the symbolic interpreter accesses an uninitialized piece of memory (condition $I \notin \mathrm{keys}(\mathit{Heap})$[3]), a new symbolic variable NewFreeVar is introduced for that position in the heap cell, and also in the iheap cell, thus making that assumption persistent independently of the effect of subsequent assignments on the heap.

As already mentioned, the exhaustive symbolic execution of all paths cannot always be achieved in practice because an unbounded number of paths can arise in the presence of loops or recursion. We follow the standard approach to avoid the exponential blowup inherent in path enumeration by exploring loops

[3]In C, the function keys(M) returns the domain of the mapping M. Thus, function call keys(Heap) represents the set of initialized memory positions in the *heap*.

and recursion up to a specified number of unfoldings. This ensures that SE ends for all explored paths, thus representing a subset of the program behavior [DKW08]. Obviously, not all the potential execution paths are feasible. We use the automatic theorem prover CVC3 [BT07] to check the satisfiability of *path constraints*, to simplify path conditions and to eliminate unfeasible symbolic computations whenever the corresponding path constraint is unsatisfiable.

In order to facilitate the specification inference, in the following section we define two types of patterns, called *observation* patterns (the *call-pattern* and the *return-pattern*), that we extract from the symbolic final configurations at the end of the symbolic execution paths.

7.2.2 The Pattern Extraction

We define a *call-pattern* as a pattern whose k cell consists of just a function call with (possibly symbolic) arguments. A *return-pattern* is a pattern that only has either a `return` instruction with the corresponding value or an `undefined` computation at the top of its k cell. The *call-patterns* and *return-patterns* (called observation patterns) respectively represent the *observable* state of a program before and after a specific function call is executed.

We note that all the information needed to extract the observation patterns is accumulated in the final MatchC symbolic configurations. In order to reuse the MatchC verification machinery, we formalize the two extracted patterns in terms of traditional **ML** patterns in the following way:

- The call-pattern is defined by filling the heap cell with the content of the iheap cell, and the k cell with the content of the ik cell, and then discarding the iheap and ik cells.
- The return-pattern is obtained by simply deleting the iheap and ik cells.

In the following, we call *initial extended pattern* or *initial symbolic configuration* to the pattern that starts the symbolic execution of a function.

Example 42

To symbolically execute the `int add(struct set *s,int x)` function by using the extended MatchC verifier, we start from the initial symbolic configuration[4]

$$\overline{p} = [\cdots\ [\mathtt{add(s,x)}]_{\mathsf{k}}[\mathtt{add(s,x)}]_{\mathsf{ik}}[\,]_{\mathsf{heap}}[\,]_{\mathsf{iheap}}[\,]_{\phi}\ \cdots]_{\mathsf{cfg}}\ ,$$

[4] We only write those cells that we need to consider for the inference.

and from this extended pattern, we obtain for each branch of the symbolic execution tree a final extended pattern $\overline{p_i}$ with the form

$$\overline{p_i} = [\cdots[\texttt{return}\,\mathsf{Value}_i]_{\mathsf{k}}[\texttt{add}(\mathsf{s},\mathsf{x})]_{\mathsf{ik}}[\mathsf{Heap}_i]_{\mathsf{heap}}[\mathsf{IHeap}_i]_{\mathsf{iheap}}[\Phi_i]_{\phi}\cdots]_{\mathsf{cfg}}$$

The call patterns and return patterns are extracted from $\overline{p_1}\ldots\overline{p_n}$:

$$\mathit{call\_pattern}(\overline{p_i}) = [\cdots\,[\texttt{add}(\mathsf{s},\mathsf{x})]_{\mathsf{k}}[\mathsf{IHeap}_i]_{\mathsf{heap}}[\Phi_i]_{\phi}\,\cdots]_{\mathsf{cfg}}\,,$$
$$\mathit{return\_pattern}(\overline{p_i}) = [\cdots\,[\texttt{return}\,\mathsf{Value}_i]_{\mathsf{k}}[\mathsf{Heap}_i]_{\mathsf{heap}}[\Phi_i]_{\phi}\,\cdots]_{\mathsf{cfg}}.$$

Formally, the extracted observation patterns are also **ML** patterns and satisfy

$$\mathit{call\_pattern}(\overline{p_i}) \Downarrow \mathit{return\_pattern}(\overline{p_i}).$$

This is because, by construction, $\mathit{call\_pattern}(\overline{p_i})$ records all the assumptions needed to ensure that $\mathit{return\_pattern}(\overline{p_i})$ holds at the end of the symbolic execution branch following the **ML** proof system implemented in MatchC. This allows us to ascertain the conditions for the completeness of our inference technique:

If the disjunction of the extracted call patterns is logically equivalent to the **ML** pattern that is obtained by removing the iheap and ik cells from the initial symbolic configuration (the extended pattern $\overline{p}$), then the set

$$\{\mathit{call\_pattern}(\overline{p_i}) \Downarrow \mathit{return\_pattern}(\overline{p_i})\}_{i\in\{1\ldots n\}}$$

of correctness pairs fully describes the input/output behavior of the considered program function.

In the following section, we formulate an algorithm that symbolically executes the program and automatically extracts and combines the call and return patterns in order to infer the pursued logical specifications.

7.3 Inference process

Let us introduce the basic notions that we use in our formalization. Given an input program, let $\mathcal{F}$ be the set of functions in the program. We distinguish the set of observers $\mathcal{O}$ and the set of modifiers $\mathcal{M}$. A function can be considered to be an *observer* if it explicitly returns a value, whereas any method can be considered to be a *modifier*. Thus, the set $\mathcal{O} \cap \mathcal{M}$ is non empty.

We denote with the symbol $\cdot$ the *universal* **ML** pattern that represents every possible program state, i.e., it imposes no constraint to the state. Given a function $f \in \mathcal{F}$, we represent the call to f with a list of arguments *args* as

$f(args)$. Then, $f(args)[p]$ is the extended pattern built by first adding both the ik and iheap cells to the pattern p, next inserting the call $f(args)$ into the ik and k cells, and then copying to the iheap cell the contents of the heap cell of p. The intuition is that $f(args)[p]$ propagates the information in p to the execution of $f(args)$. $f(args)[\cdot]$ stands for the extended pattern that represents the execution of f with arguments $args$ under a state without constraints, i.e., with empty information brought in by the universal pattern. Given an initial extended pattern $\overline{p}$, we denote as $\text{SE}(\overline{p})$ the set of final extended patterns $\{\overline{p_i}\}_{i>0}$ resulting from the symbolic execution of $\overline{p}$ in our MatchC system, i.e., the leaves of the symbolic execution tree for $\overline{p}$. Each $\overline{p_i}$ has associated a correctness pair $call\_pattern(\overline{p_i}) \Downarrow return\_pattern(\overline{p_i})$. Given a return pattern q, $q|_{ret}$ is the projection of q to its `return` value or `undefined` computation, which are in the k cell.

Our specification inference technique is formalized in Algorithm 2. First,

Algorithm 2 Inferring specifications.

Require: $m \in \mathcal{M}$ of arity n;
1. $S = \text{SE}(m(\mathtt{a}_1, \ldots, \mathtt{a}_n)[\cdot])$
2. $axiomSet := \varnothing$;
3. **for all** $\overline{p_i} \in S$ **do**
4. $eqs_{pre} := explains(call\_pattern(\overline{p_i}), [\mathtt{a}_1, \ldots, \mathtt{a}_n])$;
5. $eqs_{post} := explains(return\_pattern(\overline{p_i}), [\mathtt{a}_1, \ldots, \mathtt{a}_n])$;
6. $eq_{ret} := ret = return\_pattern(\overline{p_i})|_{ret}$;
7. $axiomSet := axiomSet \cup \{eqs_{pre} \Rightarrow (eqs_{post} \cup eq_{ret})\}$;
8. **end for**
9. spec := $simplify(axiomSet)$
10. **return** spec

the *modifier* method of interest is symbolically executed with fresh symbolic variables $\mathtt{a}_1, \ldots, \mathtt{a}_n$ as arguments. As a result, the set of final extended patterns S is computed. Then, by extracting and processing the call and return patterns of each $\overline{p_i} \in S$, a set of axioms is obtained that defines the behavior of the program. This is done by means of the function $explains(p, as)$ given in Algorithm 3. The computed axioms are implications of the form $l_i \Rightarrow r_i$. The function *simplify* implements a post-processing which consists on (1) disjoin the preconditions l_i that have the same postcondition r_i and simplify the resulting precondition, and (2) conjoin the postconditions r_i that share the same precondition and simplify the resulting postcondition.

Let us show an example of the application of the algorithm.

Example 43

Assume that we want to infer a specification for the `add` *modifier* function of Example 36. Following the algorithm, we first compute SE(`add(s,x)`[·]), with `s` and `x` (free) symbolic variables. Since there are not initial assumptions for the initial symbolic configuration, the execution covers all possible initial concrete configurations. The symbolic execution computes five final extended patterns[5]. The following extended pattern e represents the path that ends in the body of the second `if` statement:

$$\left[\begin{array}{ll} \left[\begin{array}{c} ret \mapsto 0, \\ \cdots\ \mathsf{s} \mapsto \mathsf{s},\ \cdots \\ \mathsf{x} \mapsto \mathsf{x} \end{array}\right]_{\mathsf{env}} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0 \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size} \end{array}\right]_{\phi} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size} \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}}$$

The execution of this path returns the value `0`; note that the fields `s->size` and `s->capacity` are accessed after checking that `s` is not `NULL` (i.e., `0`). In the extended pattern e, the return value `0` is represented by the binding $ret \mapsto 0$ in the env cell. The failed check of `s == NULL` adds the constraint $\mathsf{s} \neq 0$ to the ϕ cell. Given our assumption that any access to a field through a non-`NULL` pointer does succeed, the symbolic fields s.capacity and s.size are generated in the iheap cell. The successful check of `s->size >= s->capacity` adds an analogous constraint to the ϕ cell. Note that, since during the execution of this path the *heap* is not modified, the heap and iheap cells are identical, i.e., the initial and the final *heaps* are the same.

Next, for each final extended pattern the algorithm explains its *call-* and *return-patterns* by using the function $explains(p, as)$, which delivers suitable sets of equations. For the extended pattern e, the equations `isnull(s) = 0` and `isfull(s) = 1` are generated for both the *call-* and *return-patterns*. Additionally, the equation `ret = 0` is generated with the return value of e. Finally, by combining these equations we generate the following axiom:

`isnull(s) = 0` $\wedge$ `isfull(s) = 1` $\Rightarrow$
`ret = 0` $\wedge$ `isnull(s) = 0` $\wedge$ `isfull(s) = 1`

which is the computed explanation for e.

Let us now describe Algorithm 3 that defines the function $explains(p, as)$. Given an **ML** pattern p and a list of symbolic variables as, this function computes a set of equations as the description of p. These equations are composed of calls to observer functions and *built-in* functions that are bound

[5] For simplicity, we set the number of loop unrollings to one.

to the (symbolic) values that are returned by the calls. In the algorithm, $args \sqsubseteq as$ states that the list of elements $args$ is a permutation of some (or all) elements in as.

Algorithm 3 Computing explanations: $explains(p, as)$

Require: p : the pattern to be explained
Require: as : a list of symbolic variables
1. $\mathcal{C}$: the universe of observer calls;
2. $eqSet := \varnothing$;
3. **for all** $o(args) \in \mathcal{C}$ and $args \sqsubseteq as$ **do**
4. $S = \mathrm{SE}(o(args)[p])$
5. **if** $\nexists\ \overline{p_1}, \overline{p_2} \in S$ s.t. $return\_pattern(\overline{p_1})|_{ret} \neq return\_pattern(\overline{p_2})|_{ret}$ **then**
6. $eqSet := eqSet \cup (t = return\_pattern(\overline{p_1})|_{ret})$
7. **end if**
8. **end for**
9. **return** $eqSet$

Roughly speaking, given a pattern p, $explains(p, as)$ first generates the universe of observer function calls $\mathcal{C}$, which consists of all the function calls $o(args)$ that satisfy that:

- o belongs to $\mathcal{O}$ or to the set of (predefined) built-in functions,
- $args$ in the call $o(args)$ is a suitable selection of variables from the symbolic variable list as that is received as argument, respecting the type and arity of o.

Then, for each call $o(args) \in \mathcal{C}$, it checks whether all the final symbolic configurations (leaves) resulting from the execution of $o(args)$ on a state that satisfies the constraints in p have the same return value. For the calls that satisfy this condition, an equation is generated (line 6 in Algorithm 3). The intuition of this step is that, if we symbolically execute the observer at a given initial state and for all its execution branches we get the same value, then the observer together with the return value (partially) characterize the considered state. The last step of the algorithm returns the set of all the generated equations.

Example 44 (Example 43 continued)
Let us show how we compute the explanation for the *return-pattern* of Example 43 given the symbolic variables considered in the example.

Given the observer functions `isfull`, `isnull` and `contains`, and the symbolic variables `s` and `x`, the universe of observer calls is `isfull(s)`, `isnull(s)` and `contains(s,x)`. Let us show in detail the case for the observer `isnull(s)`.

When we symbolically execute `isnull(s)` over this *return-pattern*, we only obtain the extended pattern:

$$\left[\begin{array}{cc} \left[\cdots\ ret \mapsto 0\ \cdots\right]_{\mathsf{env}} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0 \ \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size} \end{array}\right]_{\phi} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size} \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}}$$

Since there are no observer paths returning different values because there is only one path (the one for computed extended pattern) and its associated return value is `0`, the equation `isnull(s) = 0` can be used as a (partial) explanation for the pattern under consideration. Then, this equation is added to the set of equations *eqSet* that will be returned by Algorithm 3.

Due to bounded loop unrolling, we cannot ensure completeness of the inferred specifications as we do not cover all possible execution paths. Also due to generalization, we cannot ensure that all inferred axioms are sound. Nevertheless, when we consider loops that are characterized by a known invariant, then we could guarantee both the soundness and completeness of our technique.

7.3.1 Refining the inference process

There are some cases in which explanations cannot be achieved due to the lack of sufficiently precise observers. In order to mitigate this problem, we present a refinement process that allows more accurate specifications to be computed and that can be applied automatically. The idea is to split the pattern that could not be explained (because there were different computed results in the leaves of the symbolic execution of observers) into multiple refined patterns that the observer functions are then able to explain.

Algorithm 4 describes how to refine a pattern p by using the observer call c. First, we compute $\mathrm{SE}(c[p])$, which obtains a set S of final extended patterns. Note that the *call-pattern* of each of these final patterns $\overline{p_i} \in S$ contains additional constraints that are imposed by the symbolic execution of the observer call. In other words, the set consisting of the new extracted call patterns forms a refinement of p.

Running the observer c under the new refined patterns delivers a single return value for each run, which brings the corresponding explanation using c.

An interesting open question is to be able to determine when we have enough observers, that is, to determine which observers are missing.

Algorithm 4 Computation of refined explanations for p given an observer c: $refined\_explains(p, c)$

Require: p : the pattern to be explained
Require: c : the observer call that will explain p by refinement
1. $expl := \varnothing$;
2. $S = \mathrm{SE}(c[p])$
3. **for all** $\overline{p_i} \in S$ **do**
4. $\quad expl := expl \cup \{p \mapsto explains(call\_pattern(\overline{p_i}))\}$;
5. **end for**
6. **return** $expl$

We have developed a prototype called KindSpec[6] that implements the inference methodology described in this chapter. By using KindSpec, we applied our methodology onto KernelC implementations of *collection* libraries for manipulating linked lists in C such as those provided in the GDSL generic data structure library [Tea13]. We evaluated the accuracy of the specifications inferred by our technique by comparing them with the original specifications written by hand, and we found out that only in a few cases there was a significant loss of information. Table 7.1 summarizes the obtained results for a set of selected benchmark programs. For each program example, the first three columns show the number of modifiers, observers and lines of code of the corresponding program, respectively. Column # corresponds to the number of explored paths as determined by the imposed termination criteria. The last two columns indicate the number of generated axioms and the number of inferred axioms that are sound, respectively. This last measurement is also related to the imposed termination criteria: the greater the depth of unrolling, the highest number of correct axioms is obtained. With respect to the time cost of the inference, it ranges from 1s. to 10s., which is quite promising and comparable to the performance of similar tools, e.g. [TCS06].

Of course, our synthesis system KindSpec inherits the current limitations of the underlying MatchC verifier. To effectively synthesize highly accurate specifications for larger programs that involve more complicated reasoning, more efficient verifiers are needed that are actually forthcoming.

[6] Available at http://safe-tools.dsic.upv.es/kindspec.

Table 7.1: Experimental results

Module	Modifiers	Obs.	LOC	#	Axioms	Sound
Set List	`add`	4	70	87	4	4
	`remove`	5	90	54	2	2
Double-Linked Lists	`append`	9	180	43	4	3
	`remove`	9	180	66	2	2
Double-ended Queues	`push_head`	4	100	106	3	3
	`push_tail`	4	100	142	3	3
	`pop_head`	4	100	16	2	2
	`pop_tail`	4	100	24	2	2
Stack	`push`	3	25	14	2	2
	`pop`	2	25	15	2	2

7.4 A case study of specification inference

Let us illustrate how we can use Algorithm 2 to discover a likely specification for the `add` *modifier* function of Example 36. First, we compute SE(`add(s,x)`[·]) with `s` and `x` (free) symbolic variables. Since there are not initial assumptions for the initial symbolic configuration, the execution covers all possible initial concrete configurations. In this case, we get five final extended patterns (shown in Figure 7.4) that correspond to five symbolic execution paths.[7] Pattern p_1 results from the execution of the path that terminates in the body of the first `if` statement. Thus, its return value is `0` and the pointer `s` is 0, i.e., `NULL`. Pattern p_2 represents the path ending in the body of the second `if` statement: its return value is also `0`, but the pointer `s` is assumed to be different from `NULL` since the guard in the previous `if` statement was not satisfied. Additionally, this path successfully accesses the `s->size` field and `s->capacity` field since the pointer `s` is not `NULL` (this can be deduced from the contents of the iheap cell). The remaining paths jump over the first two `if` statements; hence, their associated patterns impose that the pointer `s` is not `NULL` and that there is space enough in the set to add new elements (`s->size` < `s->capacity`). Additionally, pattern p_3 jumps over the `while` statement (`s->elems` is `NULL`), allocates memory address `n` for adding the new value, initializes it (represented by the assignments to addresses `n` and `n`+1 in the heap cell, but not in the iheap) and links it to the set object (represented

[7] To simplify the description, in this example we approximate the symbolic loop unrolling to just one unfolding.

$$p_1.\ \Big[\ \big[\cdots\ ret \mapsto 0,\ \mathtt{s} \mapsto \mathsf{s},\ \mathtt{x} \mapsto \mathsf{x}\ \cdots\big]_{\mathsf{env}} \big[\ \cdot\ \big]_{\mathsf{heap}} \big[\ \cdot\ \big]_{\mathsf{iheap}} \big[\ \mathsf{s} = 0\ \big]_{\phi}\ \Big]_{\mathsf{cfg}},$$

$$p_2.\ \left[\begin{array}{l} \big[\cdots\ ret \mapsto 0,\ \mathtt{s} \mapsto \mathsf{s},\ \mathtt{x} \mapsto \mathsf{x}\ \cdots\big]_{\mathsf{env}} \\ \big[\ \mathsf{s} \neq 0\ \wedge \mathsf{s.capacity} \leq \mathsf{s.size}\ \big]_{\phi} \end{array} \quad \begin{array}{l} \big[\ \mathsf{s} \mapsto \mathsf{s.capacity},\ \mathsf{s}+1 \mapsto \mathsf{s.size}\ \big]_{\mathsf{heap}} \\ \big[\ \mathsf{s} \mapsto \mathsf{s.capacity},\ \mathsf{s} \mapsto \mathsf{s.size}\ \big]_{\mathsf{iheap}} \end{array} \right]_{\mathsf{cfg}},$$

$$p_3.\ \left[\begin{array}{ll} \left[\begin{array}{c} ret \mapsto 1, \\ \cdots\ \mathtt{s} \mapsto \mathsf{s},\ \cdots \\ \mathtt{x} \mapsto \mathsf{x} \end{array}\right]_{\mathsf{env}} & \left[\begin{array}{cc} \mathsf{n} \mapsto \mathsf{x}, & \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{n}+1 \mapsto \mathsf{s.elems}, & \mathsf{s}+1 \mapsto \mathsf{s.size}+1, \\ & \mathsf{s}+2 \mapsto \mathsf{n} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0\ \wedge \\ \mathsf{s.capacity} > \mathsf{s.size}\ \wedge \\ \mathsf{s.elems} = 0 \end{array}\right]_{\phi} & \left[\begin{array}{l} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems} \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}},$$

$$p_4.\ \left[\begin{array}{ll} \left[\begin{array}{c} ret \mapsto 0, \\ \cdots\ \mathtt{s} \mapsto \mathsf{s},\ \cdots \\ \mathtt{x} \mapsto \mathsf{x} \end{array}\right]_{\mathsf{env}} & \left[\begin{array}{l} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0\ \wedge \\ \mathsf{s.capacity} > \mathsf{s.size}\ \wedge \\ \mathsf{s.elems} \neq 0\ \wedge \\ \mathsf{s.elems.value} = \mathsf{x} \end{array}\right]_{\phi} & \left[\begin{array}{l} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value} \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}},$$

$$p_5.\ \left[\begin{array}{ll} \left[\begin{array}{c} ret \mapsto 1, \\ \cdots\ \mathtt{s} \mapsto \mathsf{s},\ \cdots \\ \mathtt{x} \mapsto \mathsf{x} \end{array}\right]_{\mathsf{env}} & \left[\begin{array}{l} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}+1, \\ \mathsf{s}+2 \mapsto \mathsf{n}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value}, \\ \mathsf{s.elems}+1 \mapsto \mathsf{s.elems.next}, \\ \mathsf{n} \mapsto \mathsf{x}, \\ \mathsf{n}+1 \mapsto \mathsf{s.elems} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0\ \wedge \\ \mathsf{s.capacity} > \mathsf{s.size}\ \wedge \\ \mathsf{s.elems} \neq 0\ \wedge \\ \mathsf{s.elems.value} \neq \mathsf{x} \end{array}\right]_{\phi} & \left[\begin{array}{l} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value}, \\ \mathsf{s.elems}+1 \mapsto \mathsf{s.elems.next}, \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}}$$

Figure 7.4: Paths computed by SE(add(s,x)[·]).

by $\mathsf{s}+2 \mapsto \mathsf{n}$ in the heap cell). Finally, it returns the value `1`, which represents the successful addition of the value `x`. Pattern p_4 enters the `while` loop once and ends after completing the execution of the body of its inner `if` statement. In this case, the first element of the list that represents the set coincides with the value to be inserted (represented by the formula $\mathsf{s.elems.value} = \mathsf{x}$), so `x` is not added to the set and the value `0` is returned. Finally, pattern p_5 executes the `while` loop once but it does not enter its inner `if` statement, which means that the initial set has just one element that is different from the one to be inserted (represented by the formula $\mathsf{s.elems.value} \neq \mathsf{x}$). Afterwards, it allocates, initializes and links a new memory object representing the new value of the set `x`. An interesting difference between this path and path p_3 is that the new created object is directly linked by the set `s` (represented by $\mathsf{s}+2 \mapsto \mathsf{n}$ in the heap cell) while, initially, the only element `s` in `s` was linked within the iheap cell, which was represented by $\mathsf{s}+2 \mapsto \mathsf{s.elems}$.

At this point, we have symbolically executed function `add` which has yield five symbolic paths (in terms of final symbolic configurations). The next step of the inference Algorithm 2 is to explain each pair of extracted *call-* and *return-patterns* in terms of the observer functions. According to Algorithm 3, an observer call is chosen as an explanation of an **ML** pattern p if, when we symbolically execute it starting from an initial symbolic configuration that includes the information in p, all its paths return the same value. In the following, we show how our approach can explain the call patterns extracted from p_2 by using the observers `isnull`, `isfull` and `contains`.

The first observer to be considered is `isnull`. The symbolic execution of this observer over pattern p_2, represented by $\mathrm{SE}(isnull(s)[call\_pattern(p_2)])$, returns the only extended pattern:

$$\left[\begin{array}{cc} \left[\cdots\ ret \mapsto 0\ \cdots\right]_{\mathsf{env}} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0\ \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size} \end{array}\right]_{\phi} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size} \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}}$$

Since there is only one path and its associated return value is `0`, then the equation `isnull(s) = 0` can be used as a (partial) explanation of p_2. In particular, the equation states that, from a concrete configuration satisfying (i.e., matching) p_2, the execution of `isnull(s)` always returns 0.

Similarly, we check whether `isfull(s)` can be part of the explanation of pattern p_2 by computing $\mathrm{SE}(isfull(\mathsf{s})[call\_pattern(p_2)])$, which yields only

one path whose result is `1`:

$$\left[\begin{array}{l} \left[\cdots\ ret \mapsto 1\ \cdots\right]_{\mathsf{env}} \quad \left[\begin{array}{r} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0 \ \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size} \end{array}\right]_{\phi} \quad \left[\begin{array}{r} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size} \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}}$$

Therefore, equation `isfull(s)` = `1` is generated.

In contrast, for the observer `isfull(s)`, three different paths are obtained when executing SE($contains(\mathsf{s},\mathsf{x})[call\_pattern(p_2)]$) (see Figure 7.5). The first and the third extended patterns return `0`, i.e., $return\_pattern(p_{2,1})$ = $return\_pattern(p_{2,3})$ = 0, whereas the second extended pattern returns `1`, i.e., $return\_pattern(p_{2,2})$ = 1. Therefore, not all the final symbolic configurations have the same return value. This means that we cannot define an equation that represents all the paths to (partially) explain the symbolic configuration p_2 but we have basically two options: 1) refine the original pattern p_2 with the added constraints and then describe the refined patterns with the new observer, or 2) discard this observer call as it does not lead to a plausible explanation. Algorithm 3 adopts the second solution: it discards the observer calls that do not always return the same value. Nonetheless, if Algorithm 3 fails to explain the pattern with all the generated observer calls, an explanation that uses a given specific observer can still be attempted as follows.

If we refine $call\_patern(P_2)$ with respect to the observer `contains(s,x)`, we obtain three new (refined) patterns:

1. $$\left[\begin{array}{l} \left[\cdots\ ret \mapsto 0\ \cdots\right]_{\mathsf{env}} \quad \left[\begin{array}{r} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0 \ \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size} \ \wedge \\ \mathsf{s.elems} = 0 \end{array}\right]_{\phi} \quad \left[\begin{array}{r} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems} \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}}$$

2. $$\left[\begin{array}{l} \left[\cdots\ ret \mapsto 1\ \cdots\right]_{\mathsf{env}} \quad \left[\begin{array}{r} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0 \ \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size} \ \wedge \\ \mathsf{s.elems} \neq 0 \ \wedge \\ \mathsf{s.elems.value} = \mathsf{x} \end{array}\right]_{\phi} \quad \left[\begin{array}{r} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value} \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}}$$

$$
p_{2,1}.\ \left[\begin{array}{ll}
\left[\cdots\ ret \mapsto 0\ \cdots\right]_{\mathsf{env}} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems} \end{array}\right]_{\mathsf{heap}} \\
\left[\begin{array}{c} \mathsf{s} \neq 0\ \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size}\ \wedge \\ \mathsf{s.elems} = 0 \end{array}\right]_{\phi} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems} \end{array}\right]_{\mathsf{iheap}}
\end{array}\right]_{\mathsf{cfg}}
$$

$$
p_{2,2}.\ \left[\begin{array}{ll}
\left[\cdots\ ret \mapsto 1\ \cdots\right]_{\mathsf{env}} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value} \end{array}\right]_{\mathsf{heap}} \\
\left[\begin{array}{c} \mathsf{s} \neq 0\ \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size}\ \wedge \\ \mathsf{s.elems} \neq 0\ \wedge \\ \mathsf{s.elems.value} = \mathsf{x} \end{array}\right]_{\phi} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value} \end{array}\right]_{\mathsf{iheap}}
\end{array}\right]_{\mathsf{cfg}}
$$

$$
p_{2,3}.\ \left[\begin{array}{ll}
\left[\cdots\ ret \mapsto 0\ \cdots\right]_{\mathsf{env}} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value}, \\ \mathsf{s.elems}+1 \mapsto \mathsf{s.elems.next} \end{array}\right]_{\mathsf{heap}} \\
\left[\begin{array}{c} \mathsf{s} \neq 0\ \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size}\ \wedge \\ \mathsf{s.elems} \neq 0\ \wedge \\ \mathsf{s.elems.value} \neq \mathsf{x} \end{array}\right]_{\phi} & \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value}, \\ \mathsf{s.elems}+1 \mapsto \mathsf{s.elems.next} \end{array}\right]_{\mathsf{iheap}}
\end{array}\right]_{\mathsf{cfg}}
$$

Figure 7.5: Paths resulting from SE($contains(\mathsf{s},\mathsf{x})[call\_pattern(p_2)]$)

3. $$\left[\begin{array}{l} \left[\cdots\ ret \mapsto 0\ \cdots\right]_{\mathsf{env}} \quad \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value}, \\ \mathsf{s.elems}+1 \mapsto \mathsf{s.elems.next} \end{array}\right]_{\mathsf{heap}} \\ \left[\begin{array}{c} \mathsf{s} \neq 0 \ \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size} \ \wedge \\ \mathsf{s.elems} \neq 0 \ \wedge \\ \mathsf{s.elems.value} \neq \mathsf{x} \end{array}\right]_{\phi} \quad \left[\begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size}, \\ \mathsf{s}+2 \mapsto \mathsf{s.elems}, \\ \mathsf{s.elems} \mapsto \mathsf{s.elems.value}, \\ \mathsf{s.elems}+1 \mapsto \mathsf{s.elems.next} \end{array}\right]_{\mathsf{iheap}} \end{array}\right]_{\mathsf{cfg}}$$

In other words, these three patterns represent three initial symbolic configurations under which the execution of the observer function returns a single value. Thus, for each of these three patterns, we generate an equation with a call to the `contains` observer that explains it. Specifically, the explanations of the first and the third pattern delivers the equation $contains(s, x) = 0$, while for the second pattern we have $contains(s, x) = 1$. For each of the refined patterns we obtain, respectively, the following pre/post-conditions for the refined pattern:

1. `isnull(s) = 0 ∧ isfull(s) = 1 ∧ contains(s,x) = 0` ⇒ `ret = 0 ∧ isnull(s') = 0 ∧ isfull(s') = 1 ∧ contains(s',x) = 0`
2. `isnull(s) = 0 ∧ isfull(s) = 1 ∧ contains(s,x) = 1` ⇒ `ret = 0 ∧ isnull(s') = 0 ∧ isfull(s') = 1 ∧ contains(s',x) = 1`
3. `isnull(s) = 0 ∧ isfull(s) = 1 ∧ contains(s,x) = 0` ⇒ `ret = 0 ∧ isnull(s') = 0 ∧ isfull(s') = 1 ∧ contains(s',x) = 0`

Note that the first and third formulas are equivalent and can thus be simplified,

obtaining the two equivalent formulas:

```
isnull(s) = 0 ∧                ret = 0 ∧
isfull(s) = 1 ∧          ⇒     isnull(s') = 0 ∧
contains(s,x) = 0              isfull(s') = 1 ∧
                               contains(s',x) = 0

isnull(s) = 0 ∧                ret = 0 ∧
isfull(s) = 1 ∧          ⇒     isnull(s') = 0 ∧
contains(s,x) = 1              isfull(s') = 1 ∧
                               contains(s',x) = 1
```

7.5 Conclusions and Related Work

We have presented a technique for automatically synthesizing formal specifications for heap-manipulating programs together with some practical correctness conditions. The whole approach is formalized in the *Matching Logic* setting. We have illustrated the technique by applying it to the KernelC language. Finally, we have implemented a prototype of the method called KindSpec in order to demonstrate the practicality of the technique.

Specification extraction is itself not new. The automatic generation of likely specifications (either in the form of contracts, interfaces, summaries, assumptions, invariants, properties, component abstractions, process models, rules, graphs, automatas, etc.) from program code has received increasing attention. Specifications can be property oriented (i.e., described by pre/post-conditions or functional code); stateful (i.e., described by some form of state machines); or intensional (i.e., described by axioms). Here we only try to cover those lines of research that have influenced our work most.

Unlike our symbolic specification inference method, Daikon [EPG+07] and DIDUCE [HL02] detect program invariants by extensive test runs. Also, Henkel and Diwan [HD03] built a tool that dynamically discovers specifications for interfaces of Java classes by first generating, using the class signature, many test cases that consist of terms representing sequences of method invocations, and then generalizing the results of these tests to algebraic specifications. QuickSpec [CSH10] is another inference tool that is based on testing and can be used to generate laws that a Haskell program satisfies. Whereas Daikon discovers invariants that hold at existing program points, QuickSpec discovers equations between arbitrary terms constructed using an API, similarly to [HD03]. Also, they use a similar overall approach that is based on testing: they generate terms and evaluate them, then dynamically identify

terms that are equal, and finally generate equations, filtering away redundant ones. AbsSpec [BCFV12a], as presented in Chapter 6, is a semantic-based inference method that relies on abstract interpretation and generates laws for Curry programs in the style of QuickSpec. A different abstract interpretation approach to infer approximate specifications is [TJ07]. A combination of symbolic execution with dynamic testing is used in Dysy [CTS08]. Ghezzi *et al.* [GMM09] infer specifications of container-like classes as finite state automatas combined with graph transformation rules. All these proposals observe that conditional equations would be useful, but neither tool generates them nor include associative or commutative operators, which are naturally supported in our approach thanks to the handling of Maude's (hence **K**'s) equational attributes [CDE+07]. By supporting the modular combination of associative, commutative, idempotent and unity equational attributes for function symbols (which makes these combinations transparent to the developer), the **K** framework naturally conveys enough expressive power to reason about typed data structures such as lists (list concatenation is associative with unity element *nil*), multisets (insertion is associative-commutative with unity $\varnothing$), or sets (insertion is associative-commutative-idempotent with unity $\varnothing$). By using equational attributes to declare such properties, we can avoid non-termination problems and achieve much more efficient evaluation of terms containing such operators. We take advantage of these capabilities at three levels:

- for the definition of the extended language semantics, where the heap structures and pointer handling are represented as appropriate data structures and their associated operations,
- for the mechanization of the inference process: we efficiently handle (eventually huge) sets of axioms, paths and constraints
- most importantly, for computing the sugared version of the specification, where, by simply imposing an order relation on terms, we get a first simplification of axioms almost for free, and we infer specifications where the function symbols can be given equational attributes as well.

An alternative approach to software specification discovery is based on inductive matching learning: rather than using test cases to validate a tentative specification, they are used to *induce* the specification. Much of the work on specification mining is targeted at inferring API protocols dynamically. For instance, Whaley *et al.* [WML02] describe a system to extract component interfaces as finite state machines from execution traces. The work in [CTS08] offers a thorough revision of data mining approaches for inferring different kinds

of specifications, typically from traces or observed program runs (e.g., models, summaries, regular invocation patterns or state machines). An algorithm for interface generation of software components using learning techniques is presented in [GP09] and implemented in the JavaPathfinder model-checking framework.

Our approach differs from most of the above because we do not infer abstract properties by observations of (concrete) program runs. Our axiomatic representation of functions and of their effects is inspired by [TCS06]. However, our approach does not rely on a model checker for symbolic execution, as opposed to [TCS06]. Also, we do not generate the output as parameterized unit tests or Spec# specifications; we have simpler and more accurate formulas that avoid reasoning with the global heap but rather separate the different pieces of the heap that are reachable from the function argument addresses. Moreover, we can refine the observers by means of Algorithm 4 so that we are able to get more accurate specifications, although more experiments have yet to be done to compare all these inference methods in larger code.

As a further advantage w.r.t. [TCS06], in our framework, correctness of the inferred axioms can be checked automatically by using the very same MatchC verifier. Also, our methodology can be easily applied to any language which is given a semantics in the **K** framework.

Conclusion and Future Work

In the Part I of this thesis we have presented two different Datalog query answering techniques that are specially-tailored to object-oriented program analysis. These techniques essentially consist in transforming the original Datalog program into a suitable set of rules which are then executed under an optimized top-down strategy that caches and reuses "rewrites" in the target language.

More specifically, we have formalized the transformation of any given set of definite Datalog clauses into two efficient implementation frameworks, namely Boolean Equation Systems [And94a] and Rewriting Logic [Mes92].

In the BES-based program analysis methodology, the Datalog clauses that encode a particular analysis, together with a set of Datalog facts that are automatically extracted from the program source code, are dynamically transformed into a BES whose local resolution corresponds to the demand-driven evaluation of the program analysis. This approach has allowed existing efficient general purpose analysis and verification toolboxes such as CADP to be reused. We have implemented this technique into a prototype called DATALOG_SOLVE that shows good performance. As future work, we envisage two directions. First, it would be interesting to optimize the transformation by using more sophisticated data structures and evaluation strategies, as we have already done in [FJT10b]. The other direction consists in distributing the resolution of the BES between different machines. The distribution of the resolution could be done at the BES level [JM06] or at the Datalog level [AU10].

One of our motivations for developing the RWL-based query answering technique for Datalog was to provide purely declarative yet efficient program analyses that overcome the difficulty of handling meta-programming features such as reflection in traditional analysis frameworks [LWL05]. By transforming Datalog programs into Maude programs, we take advantage of the flexibility and versatility of Maude in order to achieve meta-programming capabilities, and we make significant progress towards scalability without losing the declarative nature of specifying complex program analyses in Datalog. We have implemented this technique into a prototype called DATALAUDE, and we have concluded that it is competitive w.r.t. other optimized deductive database systems. Actually, the provided tool supports sophisticated analyses with reasonable performance in a clean way. As future work, we envisage two di-

rections for this line of research. On one hand, the transformation can be further refined by investigating on program refactorings for Maude that aim to improve program efficiency [Cue13]. On the other hand, the use of the Maude meta-level facilities endows us with a fine-grained control of the Rwl execution, thus making possible the implementation of resolution strategies with cost guarantees [LS09], or even compositional reasoning over Datalog programs [BJ03].

In the Part II of this thesis we have presented two different techniques for synthesizing specifications for two different programming paradigms. These techniques infer succint and comprehensible specifications which are specially-tailored to program understanding and documentation.

For the multiparadigm programming language Curry, different types of equations are generated by classifying expressions built from (a subset of) the program signature according to its abstract semantics. Our method achieves a good compromise between correctness and efficiency by using a suitable abstract semantics that allows one to discriminate between *correct* and *possibly correct* parts of the specification. We have implemented this technique into a prototype called AbsSpec and proved its efficacy. As future work, we envisage two directions. First, we plan to experiment with other abstract domains and semantics in order to be able to guarantee the inference of a greater number of correct equations. Second, we think that there is an opportunity to further simplify the inferred equations in order to obtain even more concise specifications by using equational resoning.

The second technique for specification inference was designed to obtain high-level specifications from object-oriented languages and we formulated it for a subset of C. Our method infers for each function in a program a set of high-level pre/post-condition pairs, whose low-level details are *observationally abstracted* by means of other functions in the same program. We have formalized the technique in the *Matching Logic* verification setting in order to be able to verify the correctness of the inferred specifications. We have implemented this technique into a prototype called KindSpec that we applied to infer specifications for *collection* libraries. As future work, we are interested in increasing the conciseness of the generated specifications. In particular, we think that the application of generalization techniques can further improve the simplicity of the inferred specifications.

As a conclusion, in this thesis we have presented different techniques aimed at improving different aspects of the existing approaches for declarative static analysis and automated synthesis of specifications. Hopefully, our work will set the grounds for further investigations in these areas.

Bibliography

[ABL02] G. Ammons, R. Bodík, and J. R. Larus. Mining Specifications. In J. Launchbury and J. C. Mitchell, editors, *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 4–16, New York, NY, USA, 2002. ACM Press.

[AFJV08] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Static Analysis of JAVA Programs in a Rule-Based Framework. In J. Almendros and M. Suárez-Cabal, editors, *Pre-Proceedings of the VIII Jornadas sobre Programación y Lenguajes (PROLE 2008)*, ISBN: 978-84-612-5819-2. Gráficas Rigel, 2008.

[AFJV09a] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. DATALOG_SOLVE: A Datalog-Based Demand-Driven Program Analyzer. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 248:57–66, 2009.

[AFJV09b] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Defining Datalog in Rewriting Logic. Technical Report DSIC-II/07/09, Universitat Politècnica de València, 2009.

[AFJV09c] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Implementing Datalog in Maude. In R. Peña, editor, *Proceedings of the IX Jornadas sobre Programación y Lenguajes (PROLE'09) and I Taller de Programación Funcional (TPF 2009)*, pages 15–22, 2009. ISBN 978-84-692-4600-9.

[AFJV09d] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Using Datalog and Boolean Equation Systems for Program Analysis. In D. Cofer and A. Fantechi, editors, *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, volume 5596 of *Lecture Notes in Computer Science*, pages 215–231, Berlin, Heidelberg, 2009. Springer-Verlag.

[AFJV10] M. Alpuente, M.A. Feliú, C. Joubert, and A. Villanueva. Defining Datalog in Rewriting Logic. In D. Schreye, editor, *Proceedings of the 19th International Conference on Logic-Based Program*

Synthesis and Transformation (LOPSTR 2009), volume 6037 of *Lecture Notes in Computer Science*, pages 188–204, Berlin, Heidelberg, 2010. Springer-Verlag.

[AFJV11] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva. Datalog-Based Program Analysis with BES and RWL. In O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, editors, *Selected Papers of the 1st International Workshop Datalog Reloaded (Datalog 2010)*, Lecture Notes in Computer Science, pages 1–20, Berlin, Heidelberg, 2011. Springer-Verlag.

[AFV13] M. Alpuente, M. A. Feliú, and A. Villanueva. Automatic Inference of Specifications using Matching Logic. In E. Albert and S.-C. Mu, editors, *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM 2013)*, pages 127–136, New York, NY, USA, 2013. ACM Press.

[AHH+05] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[And94a] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.

[And94b] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[AU10] F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-Reduce Environment. In I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan, editors, *Proceedings of the 13th International Conference on Extending Database Technology (EDBT 2010)*, volume 426 of *ACM International Conference Proceeding Series*, pages 99–110. ACM Press, 2010.

[Bac12] G. Bacci. *An Abstract Interpretation Framework for Semantics and Diagnosis of Lazy Functional-Logic Languages*. Ph.D., Università di Udine, 2012.

[BBC+10] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs

in the Real World. *Communications of the ACM*, 53(2):66–75, 2010.

[BC10] G. Bacci and M. Comini. A compact goal-independent bottom-up fixpoint modeling of the behaviour of first order curry. Technical Report 6/2010, Università di Udine, 2010.

[BC11] G. Bacci and M. Comini. Abstract Diagnosis of First Order Functional Logic Programs. In M. Alpuente, editor, *Selected Papers of the 20th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2010)*, volume 6564 of *Lecture Notes in Computer Science*, pages 215–233, Berlin, Heidelberg, 2011. Springer-Verlag.

[BC12] G. Bacci and M. Comini. A Fully-Abstract Condensed Goal-Independent Bottom-Up Fixpoint Modeling of the Behaviour of First Order Curry. Submitted for Publication. Available at http://www.dimi.uniud.it/comini/Papers/, 2012.

[BCFV11] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. Automatic Synthesis of Specifications for Curry Programs. In G. Vidal, editor, *Pre-Proceedings of the 21th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2011)*, pages 143–152, Odense, Denmark, 2011.

[BCFV12a] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. Automatic Synthesis of Specifications for First Order Curry Programs. In D. De Schreye, G. Janssens, and A. King, editors, *Proceedings of the 14th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2012)*, pages 25–34, New York, NY, USA, 2012. ACM Press.

[BCFV12b] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. The additional difficulties for the automatic synthesis of specifications posed by logic features in functional-logic languages. In A. Dovier and V. Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, volume 17 of *LIPIcs*, pages 144–153. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[BJ03] F. Besson and T. P. Jensen. Modular Class Analysis with DATALOG. In Cousot R., editor, *Proceedings of the 10th International Static Analysis Symposium (SAS 2003)*, volume 2694 of

Lecture Notes in Computer Science, pages 19–36, Berlin, Heidelberg, 2003. Springer-Verlag.

[BPS00] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software -— Practice & Experience*, 30(7):775–802, 2000.

[BS09] M. Bravenboer and Y. Smaragdakis. Strictly Declarative Specification of Sophisticated Points-to Analyses. In S. Arora and G. T. Leavens, editors, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, pages 243–262, New York, NY, USA, 2009. ACM Press.

[BT07] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, Berlin, Heidelberg, 2007. Springer-Verlag.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In A. V. Aho, S. N. Zilles, and B. K. Rosen, editors, *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1979)*, pages 269–282, New York, NY, USA, 1979. ACM Press.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ Analyzer. In M. Sagiv, editor, *Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming (ESOP 2005)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, Berlin, Heidelberg, 2005.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude – A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 2007.

[Cen13] CERT Coordination Center. CERT/CC Statistics of vulnerabilities reported (Historical) 1988-2008, Last accesed: September 2013. http://www.cert.org/stats/.

[CF08] J. Christiansen and S. Fischer. EasyCheck – Test Data for Free. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, volume 4989 of *Lecture Notes in Computer Science*, pages 322–336, Berlin, Heidelberg, 2008. Springer-Verlag.

[CH00] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[CHH+06] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving Your Software Using Static Analysis to Find Bugs. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 673–674, New York, NY, USA, 2006. ACM Press.

[CJK07] M. Christodorescu, S. Jha, and C. Kruegel. Mining Specifications of Malicious Behavior. In I. Crnkovic and A. Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE 2007)*, pages 5–14. ACM Press, 2007.

[CPvW07] T. Chen, B. Ploeger, J. van de Pol, and T. A. C. Willemse. Equivalence Checking for Infinite Systems Using Parameterized Boolean Equation Systems. In L. Caires and V. T. Vasconcelos, editors, *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR 2007)*, volume 4703 of *Lecture Notes in Computer Science*, pages 120–135, Berlin, Heidelberg, 2007. Springer-Verlag.

[CSH10] K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In G. Fraser and A. Gargantini, editors, *Proceedings of the 4th International Conference on Tests and Proofs (TAP 2010)*, volume 6143 of *Lecture Notes in Computer Science*, pages 6–21. Springer-Verlag, Berlin, Heidelberg, 2010.

[CTS08] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic Symbolic Execution for Invariant Inference. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 281–290. ACM Press, 2008.

[Cue13] Á. Cuenca. Técnicas de Refactorización para Maude. MSc Thesis, Universitàt Politècnica de València, 2013.

[DKW08] V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[EPG+07] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[ER12] C. Ellison and G. Roşu. An Executable Formal Semantics of C with Applications. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 533–544. ACM Press, 2012.

[FJT10a] M. A. Feliú, C. Joubert, and F. Tarín. Efficient BES-based Bottom-Up Evaluation of Datalog Programs. In V. Gulías, J. Silva, and A. Villanueva, editors, *Proceedings of the X Jornadas sobre Programación y Lenguajes (PROLE 2010)*, pages 165–176, 2010.

[FJT10b] M. A. Feliú, C. Joubert, and F. Tarín. Evaluation Strategies for Datalog-based Points-To Analysis. *ECEASST*, 35, 2010.

[GC10] M. Gordon and H. Collavizza. Forward with hoare. In A. W. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the*

Work of C.A.R. Hoare, History of Computing, pages 101–121. Springer-Verlag, Berlin, Heidelberg, 2010.

[GM10] C. Ghezzi and A. Mocci. Behavior Model Based Component Search: An Initial Assessment. In *Proceedings of the 2nd International Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation (SUITE 2010)*, pages 9–12, New York, NY, USA, 2010. ACM Press.

[GMLS07] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 158–163, Berlin, Heidelberg, 2007. Springer-Verlag.

[GMM09] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing Intensional Behavior Models by Graph Transformation. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 430–440. IEEE, 2009.

[GP09] D. Giannakopoulou and C. S. Păsăreanu. Interface Generation and Compositional Verification in JavaPathfinder. In M. Chechik and M. Wirsing, editors, *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009)*, volume 5503 of *Lecture Notes in Computer Science*, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal on Logic Programming*, 19-20, Supplement 1(0):583–628, 1994.

[Han97] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*, pages 80–93, New York, NY, USA, 1997. ACM Press.

[Han06] M. Hanus. Curry: An integrated functional logic language (vers. 0.8.2), 2006. Last accesed: September 2013. http://www.informatik.uni-kiel.de/~curry.

[HD03] J. Henkel and A. Diwan. Discovering Algebraic Specifications from Java Classes. In L. Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 431–456, Berlin, Heidelberg, 2003. Springer-Verlag.

[HL02] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In W. Tracz, M. Young, and J. Magee, editors, *Proceedings of the 22rd International Conference on Software Engineering (ICSE 2002)*, pages 291–301. ACM Press, 2002.

[HRD07] J. Henkel, C. Reichenbach, and A. Diwan. Discovering Documentation for Java Container Classes. *IEEE Transactions on Software Engineering*, 33(8):526–542, 2007.

[JM06] C. Joubert and R. Mateescu. Distributed On-the-Fly Model Checking and Test Case Generation. In A. Valmari, editor, *Proceedings of the 13th International SPIN Workshop on Model Checking Software (SPIN 2006)*, volume 3925 of *Lecture Notes in Computer Science*, pages 126–145, Berlin, Heidelberg, 2006. Springer-Verlag.

[Jon09] C. Jones. *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*. AccessEngineering. McGraw-Hill Education, 2009.

[Kin76] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KPV03] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In H. Garavel and J. Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568, Berlin, Heidelberg, 2003. Springer-Verlag.

[KU10] A. A. Khwaja and J. E. Urban. A property based specification formalism classification. *Journal of Systems and Software*, 83(11):2344–2362, 2010.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science. MIT Press, Cambridge, MA, USA, 1986.

[LS98] X. Liu and S. A. Smolka. Simple Linear-Time Algorithms for Minimal Fixed Points. In K. Larsen Guldstrand, S. Skyum, and G. Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP 1998)*, volume 1443 of *Lecture Notes in Computer Science*, pages 53–66, London, UK, 1998. Springer-Verlag.

[LS03] Y. A. Liu and S. D. Stoller. From Datalog Rules to Efficient Programs with Time and Space Guarantees. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2003)*, pages 172–183, New York, NY, USA, 2003. ACM Press.

[LS09] Y. A. Liu and S. D. Stoller. From Datalog Rules to Efficient Programs with Time and Space Guarantees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6), 2009.

[LWL05] V. B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In K. Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS 2005)*, volume 3780 of *Lecture Notes in Computer Science*, pages 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

[Mar94] M. Marchiori. Logic Programs as Term Rewriting Systems. In G. Levi and M. Rodríguez-Artalejo, editors, *Proceedings of the 4th International Conference on Algebraic and Logic Programming (ALP 1994)*, volume 850 of *Lecture Notes in Computer Science*, pages 223–241, Berlin, Heidelberg, 1994. Springer-Verlag.

[Mat98] R. Mateescu. Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation (VMCAI 1998)*, 1998.

[Mes92] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MT08] R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In J. Cuéllar, T. S. E.

Maibaum, and K. Sere, editors, *Proceedings of the 15th International Symposium on Formal Methods (FM 2008)*, volume 5014 of *Lecture Notes in Computer Science*, pages 148–164, Berlin, Heidelberg, 2008. Springer-Verlag.

[NLV09] I. Nunes, A. Lopes, and V. Vasconcelos. Bridging the Gap between Algebraic Specification and Object-Oriented Generic Programming. In S. Bensalem and D. Peled, editors, *Selected Papers of the 9th International Workshop on Runtime Verification (RV 2009)*, volume 5779 of *Lecture Notes in Computer Science*, pages 115–131, Berlin, Heidelberg, 2009. Springer-Verlag.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Secaucus, NJ, USA, 1999.

[oCP13] ACM Committee on Computers and Public Policy. Forum On Risks To The Public In Computers And Related Systems, Last accesed: September 2013. http://catless.ncl.ac.uk/Risks/.

[PCJL95] C. Potter, T. Cory, T. Jowitt, and ButlerBloor Ltd. *CAST Tools: An Evaluation and Comparison*. ButlerBloor Limited, 1995.

[PJ03] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003. Available at http://www.haskell.org/definition/.

[PK82] R. Paige and S. Koenig. Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):402–454, 1982.

[PV09] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):339–353, 2009.

[Rc11] G. Roşu and A. Ştefănescu. Matching Logic: A New Program Verification Approach (NIER Track). In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, pages 868–871. ACM Press, 2011.

[Rc12] G. Roşu and A. Ştefănescu. Towards a unified theory of operational and axiomatic semantics. In A. Czumaj, K. Mehlhorn,

A. M. Pitts, and R. Wattenhofer, editors, *Part II of the Proceedings of the 39th International Colloquium on Automata, Languages, and Programming (ICALP 2012)*, volume 7392 of *Lecture Notes in Computer Science*, pages 351–363, Berlin, Heidelberg, 2012. Springer-Verlag.

[Red84] U. S. Reddy. Transformation of Logic Programs into Functional Programs. In *Proceedings of the 1st International Symposium on Logic Programming (SLP 1984)*, pages 187–197. IEEE Computer Society Press, 1984.

[RES11] G. Roşu, C. Ellison, and W. Schulte. Matching Logic: An Alternative to Hoare/Floyd Logic. In M. Johnson and D. Pavlovic, editors, *Selected Papers of the 13th International Conference on Algebraic Methodology and Software Technology (AMAST 2010)*, volume 6486 of *Lecture Notes in Computer Science*, pages 142–162, Berlin, Heidelberg, 2011. Springer-Verlag.

[Rey02] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[RMY+09] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile Specifications. In S. Arora and G. T. Leavens, editors, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, pages 999–1006. ACM Press, 2009.

[RS09] G. Roşu and W. Schulte. Matching logic – extended report. Technical Report Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign, 2009.

[RŞ10] G. Roşu and T. F. Şerbănuţă. An Overview of the **K** Semantic Framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[RSŞ09] G. Roşu, W. Schulte, and T. F. Şerbănuţă. Runtime Verification of C Memory Safety. In S. Bensalem and D. Peled, editors, *Selected Papers of the 9th International Workshop on Runtime Verification (RV 2009)*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–152, Berlin, Heidelberg, 2009. Springer-Verlag.

[SKGST07] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs by Term Rewriting. In G. Puebla, editor, *Selected Papers of the 16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2006)*, volume 4407 of *Lecture Notes in Computer Science*, pages 177–193, Berlin, Heidelberg, 2007. Springer-Verlag.

[Sou13] Open Source. LLVM/Clang Static Analyzer, Last accessed: September 2013. http://clang-analyzer.llvm.org.

[Tas02] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, USA, 2002.

[TCS06] N. Tillmann, F. Chen, and W. Schulte. Discovering Likely Method Specifications. In Z. Liu and Jifeng He, editors, *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *Lecture Notes in Computer Science*, pages 717–736, Berlin, Heidelberg, 2006. Springer-Verlag.

[Tea13] The GDSL Team. The Generic Data Structures Library GDSL, Last accesed: September 2013. http://home.gna.org/gdsl/.

[TJ07] M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *Automated Software Engineering*, 14(1):87–121, 2007.

[TL10] K. T. Tekle and Y. A. Liu. Precise Complexity Analysis for Efficient Datalog Queries. In T. Kutsia, W. Schreiner, and M. Fernández, editors, *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2010)*, pages 35–44, New York, NY, USA, 2010. ACM Press.

[Ull85] J. D. Ullman. Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems (TODS)*, 10(3):289–321, 1985.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems – Volume II: The New Technologies.* Computer Science Press, Rockville, MD, USA, 1989.

[vDPW08] A. van Dam, B. Ploeger, and T. A. C. Willemse. Instantiation for Parameterised Boolean Equation Systems. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, editors, *Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing (ICTAC 2008)*, volume 5160 of *Lecture Notes in Computer Science*, pages 440–454, Berlin, Heidelberg, 2008. Springer-Verlag.

[Vie86] L. Vieille. Recursive Axioms in Deductive Databases: The Query/Subquery Approach. In L. Kerschberg, editor, *Proceedings of the 1st International Conference on Expert Database Systems (EDS 1986)*, pages 253–267, 1986.

[vL90] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science: Formal Models and Semantics*, volume B. Elsevier, The MIT Press, Cambridge, MA, USA, 1990.

[vV93] J. C. van Vliet. *Software Engineering – Principles and Practice*. John Wiley & Sons, Inc., 1993.

[WACL05] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In K. Yi, editor, *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer-Verlag, Berlin, Heidelberg, 2005.

[Wha05] J. Whaley. Joeq: A virtual machine and compiler infrastructure. *Science of Computer Programming – Special Issue on Advances in Interpreters, Virtual Machines and Emulators*, 57(3):339–356, 2005.

[Win90] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, 1990.

[WML02] J. Whaley, M. C. Martin, and M. S. Lam. Automatic Extraction of Object-Oriented Component Interfaces. *SIGSOFT Software Engineering Notes*, 27(4):218–228, 2002.

[YKZZ08] B. Yu, L. Kong, Y. Zhang, and H. Zhu. Testing Java Components based on Algebraic Specifications. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 190–199. IEEE Computer Society, 2008.

[ZC09] M. Zhivich and R. K. Cunningham. The Real Cost of Software Errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

[ZR08] X. Zheng and R. Rugina. Demand-Driven Alias Analysis for C. In G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 197–208, New York, NY, USA, 2008. ACM Press.