KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Offline Partial Deduction for the Analysis and Compilation of Coroutines

Vincent Nys

Supervisors:
Prof. dr. Danny De Schreye
Prof. dr. ir. Tom Schrijvers

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

September 2019

Offline Partial Deduction for the Analysis and Compilation of Coroutines

Vincent NYS

Examination committee:
Prof. dr. ir. Hugo Hens, chair
Prof. dr. Danny De Schreye, supervisor
Prof. dr. ir. Tom Schrijvers, supervisor
Prof. dr. ir. Gerda Janssens
Prof. dr. ir. Bart Jacobs
Prof. dr. John Gallagher
(Roskilde University)
Prof. dr. Michael Leuschel
(Universität Düsseldorf)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

September 2019

© 2019 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Vincent Nys, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

The work before you deals with the compilation of coroutines in a logic programming context. It investigates different ways in which a program using a non-standard control flow mechanism can transformed into a program using a standard control flow mechanism. The final product is published under my name, but I could not have achieved anything near this on my own.

I owe a great debt of gratitude to my supervisor, Danny De Schreye. I look up to you as a researcher and as an educator. I am honored to call you a friend.

I am grateful to the members of the jury. You have shown an interest in the work I have spent a sizeable portion of my life on. You have often provided new perspectives and spurred me on with insightful questions.

I am thankful to the Research Foundation - Flanders (FWO) and the department of Computer Science for making this work possible.

I am thankful for and to the students I have advised. You have demonstrated great creativity and diligence. Your enthusiasm was often infectious and encouraged me to put in an extra effort more often than you may believe.

I am thankful to my former co-workers. I have enjoyed our time together and I deeply admire the work you have done and continue to do.

Special thanks go to my mother and sister, for too much to list.

Abstract

Coroutines are a powerful, but underappreciated tool for managing program control flow. They are generalizations of subroutines whose execution can be suspended and resumed. Unlike the relationship between subroutines, the relationship between coroutines is not hierarchical, but symmetrical. Coroutines are especially suited to non-preemptive multitasking and are present in some form in most modern programming languages, allowing for concurrency without parallelism, which is often simpler than using program threads.

In logic programming languages, coroutines are goals which may give up control before they are run to completion. Non-standard selection rules often implement coroutines in some form. Such selection rules epitomize Kowalski's well-known equation "Algorithm = Logic + Control", which means that a program's logic should be specified separately from its control mechanism. A major drawback of these mechanisms is that they complicate program analysis. Furthermore, there is no generally accepted coroutining mechanism for logic programming languages, so program analysis techniques often disregard the use of coroutines entirely.

The current work proposes a way to circumvent this problem through the use of program transformation. It shows that it is possible to compile a declarative program specification and a specific control flow mechanism, resulting in a program which is highly procedural in nature, but which is easy to analyze automatically. The technique we use is based on an earlier ad-hoc technique known as compiling control, but it has a stronger formal foundation. This work also contributes an implementation of the transformation, programs which pose problems not considered in the original work on compiling control, solutions to these problems and new, simpler program synthesis procedures. Finally, it demonstrates that the transformation does, indeed, facilitate static analysis of coroutining logic programs by applying termination analysis techniques to transformed programs.

Beknopte samenvatting

Coroutines zijn een krachtig, maar ondergewaardeerd mechanisme om de controle van een programma te sturen. Ze veralgemenen subroutines, in die zin dat hun uitvoering gepauzeerd en hervat kan worden. In tegenstelling tot subroutines zijn coroutines niet hiërarchisch geordend, maar staan verschillende coroutines op een gelijk niveau. Coroutines zijn bijzonder geschikt voor coöperatief *multitasken* en zijn, weliswaar vaak in beperkte vorm, terug te vinden in de meeste hedendaagse programmeertalen. Ze staan gelijktijdigheid zonder parallelisme toe, wat vaak eenvoudiger is dan het gebruik van *threads*.

In logische programmeertalen zijn coroutines doelen die niet noodzakelijk volledig worden afgewerkt vooraleer de controle wordt overgedragen aan een ander doel. Alternatieve selectiemechanismen implementeren vaak een vorm van coroutines. Dergelijke selectiemechanismen illustreren Kowalski's vergelijking "Algoritme = Logica + Controle", wat betekent dat de kennis vervat in een programma afzonderlijk van het uitvoeringsmechanisme gespecifieerd zou moeten worden. Een belangrijk nadeel van alternatieve selectiemechanismen is dat ze de analyse van programma's bemoeilijken. Bovendien is er geen algemeen aanvaarde manier om coroutines te implementeren in logische programmeertalen. Het gevolg hiervan is dat technieken voor de analyse van programma's het gebruik van coroutines vaak negeren.

Dit werk stelt een aanpak voor om dit probleem te omzeilen door middel van programmatransformatie. Het toont dat het mogelijk is een declaratief programma en een controlemechanisme voor coroutines te compileren tot een programma dat eerder op een procedurele manier gestructureerd is, maar dat wel onderworpen kan worden aan een automatische analyse. De aanpak is gebaseerd op een oudere techniek, *compiling control*, maar de formele basis is verder uitgewerkt. Deze thesis omvat ook een implementatie van de programmatransformatie, een studie van problemen die niet aan bod kwamen in de oorspronkelijke publicaties rond *compiling control* en nieuwe, simpelere manieren om een getransformeerd programma te bekomen. Ten slotte toont

dit werk ook dat de transformatie het makkelijker maakt analyses te doen, door technieken voor terminatieanalyse toe te passen op getransformeerde programma's.

List of Symbols

$=_r$	syntactic identity up to reordering
α	the abstraction function in abstract interpretation
α_1	a fixed pre-abstraction function
γ	the concretization function in abstract interpretation
$\longmapsto$	transition relation in CHR
$\mathcal{A}$	the set of analyzed atoms in partial deduction; the set of analyzed conjunctions in conjunctive partial deduction; the set of analyzed abstract conjunctions in abstract conjunctive partial deduction
$\mathcal{C}$	the set of all conjunctions of atoms over a given alphabet
$\mathcal{M}(A)$	the set of all multisets composed of elements of the set A
$AAtom_P$	the set of all abstract atoms in the analysis of P
$AAtomMulti_P$	the set of all abstract atoms and *multi* abstractions in the language underlying P
$ACon_P$	the set of all abstract constants in the analysis of P
$AConAtom_P$	the set of all abstract conjunctions constructed from elements of $AAtom_P$
$AConAtomMulti_P$	the set of all conjunctions of elements in $AAtomMulti_P$
$ADom_P$	shorthand for $ATerm_P \cup AAtom_P \cup AConAtom_P$
$ADomMulti_P$	the abstract-multi domain
$ATerm_P$	the set of all abstract terms in the analysis of P

$AVar_P$	the set of all abstract variables in the analysis of P
$ctxt(SC, AC)$	context of subconjunction SC in conjunction AC
ν	an atomic renaming
$\overline{t_1} \leq \overline{t_2}$	$\overline{t_1}$ is an equivalence class with a smaller concretization than the equivalence class $\overline{t_2}$
$\| \cdot \|$	symbolic norm function
σ	an abstract-concrete substitution
Σ,Θ,...	an abstract substitution
$\sigma' \succcurlyeq \sigma$	σ' is an a-g-extension of σ
$a \xrightarrow{p} b$	a takes strict precedence over b for selection
a_i	abstract variable representing the set of all concrete terms, where $i \in \mathbb{N}_0$
$c(acon)$	the concrete conjunction in the root for the abstract conjunction $acon$
Dom_σ	the domain of an abstract-concrete substitution σ
f_k	abstract variable representing the set of all free concrete variables, where $k \in \mathbb{N}_0$
g_j	abstract variable representing the set of all ground concrete terms, where $j \in \mathbb{N}_0$
P	the source program which is compiled under a certain selection rule for a certain top-level goal
p	a partitioning function
P'	the program produced by compiling P under a certain selection rule for a certain top-level goal
$t_1 \approx t_2$	t_1 and t_2 are equivalent elements of $ADom_P$
$t_1 \succcurlyeq t_2$	t_1 is a more general abstract term or atom than t_2
$++$	concatenation operator
$Atom_P$	the set of all atoms which can be constructed from $Pred_P$ and $Term_P$

Con_P	the set of all constants in the language underlying P
$ConAtom_P$	the set of all conjunctions of atoms from $Atom_P$
Dom_P	shorthand for $Term_P \cup Atom_P \cup ConAtom_P$
Fun_P	the set of all functors in the language underlying P
$Pred_P$	the set of all predicate symbols in the language underlying P
$Term_P$	the set of all terms constructable from Con_P, Var_P and Fun_P
Var_P	the set of all variables in the language underlying P

Contents

Abstract iii

Beknopte samenvatting v

List of Symbols ix

Contents xi

List of Figures xvii

List of Listings xxi

List of Tables xxiii

1 Introduction 1

1.1 Coroutines and logic programs 1

1.1.1 Logic programming 1

1.1.2 Coroutines 2

1.2 Research goal 3

1.3 Motivation 4

1.3.1 Coroutining mechanisms 4

1.3.2 Related mechanisms 5

1.4 Related work 6
1.5 Overview 7

2 Preliminaries **9**

2.1 Conventions 9
2.2 Compiling Control 9
2.3 Partial Deduction 13
2.4 Conjunctive Partial Deduction 17
2.5 Abstract Interpretation 20
2.6 Abstract Conjunctive Partial Deduction 21

3 Abstract Conjunctive Partial Deduction **23**

3.1 A simple example 23
3.1.1 Basic concepts 23
3.2 A More Formal Account 31
3.2.1 Abstract Unification 32
3.2.2 Abstract resolve 40
3.2.3 Correctness of the approach 44
3.2.4 Control and widening 47
3.3 A more complex example, introducing the *multi* abstraction . . 51
3.4 A final example, illustrating more aspects 65
3.5 Remarks 66

4 Limitations **69**

4.1 Scope 69
4.2 Lowest common denominator control 69
4.2.1 Correspondence between abstract and concrete selection 70
4.2.2 Speculative compiling control 73

4.3 Expressivity of *multi* 78
4.4 Beyond depth-k abstraction 82

5 CHR synthesis 87

5.1 Motivation 87
5.2 The refined operational semantics of CHR 88
5.2.1 Basic concepts 88
5.2.2 Operational semantics 89
5.2.3 Refined operational semantics 90
5.3 Permutation sort in CHR 91
5.4 Incorporating the *multi* abstraction: confused queens 92
5.5 A more refined CHR synthesis 99
5.6 Conclusions 102

6 Compiling Control as Offline Partial Deduction 109

6.1 Introduction 109
6.2 Preliminaries 110
6.2.1 The first Futamura projection 110
6.2.2 Offline vs. online partial deduction 111
6.3 A suitable meta-interpreter 112
6.3.1 Instantiation of the first Futamura projection 113
6.4 Programs requiring the *multi* abstraction 115
6.4.1 An extended meta-interpreter 116
6.5 Specialization using Logen 118
6.5.1 Simple meta-interpreter 119
6.5.2 Extended meta-interpreter 120
6.6 Equivalence with the interpreter-free approach 121
6.7 Discussion 122

7 Implementation **123**

7.1 General design 123

7.2 Representation of the abstract domain 124

7.2.1 $ADom_P$ 124

7.2.2 $ADomMulti_P$ 124

7.2.3 Analyzer input 125

7.3 Representing the analysis 127

7.4 Abstract unification 129

7.5 The selection rule 130

7.6 Program synthesis 131

7.7 Performance 131

8 Application: termination analysis **137**

8.1 Meta-interpreter 137

8.2 Utilized termination analyzers 140

8.3 Analysis 140

8.3.1 Summary of TerminWeb 140

8.4 Remarks 146

9 Conclusion **147**

9.1 Overview of the contributions 147

9.2 Future work 148

9.2.1 Leveraging abstract interpretation 148

9.2.2 Application to alternate interpreters 148

9.2.3 Efficient, full automation 148

9.2.4 Depth-k abstraction 149

9.2.5 Translating `block` declarations 149

A Example programs **151**

A.1 Prime sieve (Peano encoding) 151

A.2 Lucky numbers sieve 152

A.3 N queens 153

B CHR performance **155**

C Meta-interpreter for specialization **159**

D Meta-interpreter for termination analysis **165**

E How to reproduce results **175**

E.1 Analysis 175

E.2 Synthesis 176

F Extra syntheses **177**

F.1 Speculative permutation sort 177

F.2 Alternating graph coloring 180

F.3 Continuation passing graph coloring 184

F.4 Complete Prolog synthesis of confused queens 186

F.5 Procedural permutation sort 188

Bibliography **191**

Curriculum vitae **201**

List of Figures

2.1 Symbolic trace tree for permutation sort with compiling control 13

2.2 Symbolic trace graph for permutation sort with compiling control 14

2.3 Reduced trace graph for permutation sort with compiling control 14

2.4 Derivation tree for partial deduction example 16

2.5 SLD-tree τ_1 for example 1 18

2.6 SLD-tree τ_2 for example 1 19

3.1 Abstract tree for $permsort(g_1, a_1)$ 28

3.2 Abstract tree for $perm(g_1, a_1), ord([g_2|a_1])$ 29

3.3 Concrete tree corresponding to figure 3.1 30

3.4 Concrete tree corresponding to figure 3.2 30

3.5 Use of the *multi* abstraction in the prime sieve 62

3.6 A *multi* abstraction is unfolded "from the left" during the analysis of sameleaves 67

3.7 A *multi* abstraction is unfolded "from the right" during the analysis of sameleaves 68

4.1 Foundation for alternative $ATerm_P$ 74

4.2 Speculative compiling control analysis of permutation sort . . . 76

4.3 Adverse effect of depth-k abstraction in graph coloring variant 84

4.4 Generalizing with *complex*. For readability, the counterparts of the sets of constraints in *multi* are not included. 85

4.5 Atomic term is unfolded from *complex*. 85

4.6 Complex term is unfolded from *complex*. 85

5.1 Operational semantics of CHR 90

5.2 First abstract tree for confused queens 104

5.3 Second abstract tree for confused queens 105

5.4 Third abstract tree for confused queens. To reduce the size of the figure, constraints are only shown if they are not clear from parent nodes. 106

5.5 First concrete tree for confused queens 107

5.6 Second concrete tree for confused queens 107

5.7 Third concrete tree for confused queens. 108

7.1 The numbers in the leaf nodes of the parse tree represent their parents, the parameterized abstract variables. 125

7.2 Representation of the analysis when the top-level goal has not yet been unfolded. 128

7.3 Representation of the analysis immediately after the first abstract resolution step. Note that `4`, in the bottom fact, represents the clause defining `permsort/2`. 128

7.4 Representation of the analysis immediately after the second abstract resolution step with an applicable clause. 128

7.5 Representation of the analysis immediately after introduction of the first loop. 133

7.6 Representation of the analysis immediately after the first full evaluation step. 134

7.7 Representation of generalization during the analysis. 135

8.1 State machine encoded by permutation sort synthesis 141

8.2 State machine encoded by permutation sort synthesis, with a key constraint in the abstract binary unfoldings semantics 142

8.3 State machine encoded by graph coloring synthesis 146

List of Listings

2.1 Permutation sort 10
2.2 Permutation sort in SICStus 10
3.1 Synthesized permutation sort 31
3.2 Prime sieve 51
3.3 Prime sieve in SICStus 51
3.4 Sameleaves 65
3.5 Sameleaves in SICStus 65
4.2 Graph coloring 78
4.3 Alternating graph coloring variant 79
4.4 Continuation passing graph coloring variant 82
5.1 Permutation sort in CHR 92
5.2 Confused queens 93
5.3 Partial Prolog synthesis of confused queens 95
5.4 CHR synthesis of confused queens, with pitfall 98
5.5 CHR synthesis of confused queens, without pitfall 101
6.1 Simple meta-interpreter for specialization 112
6.2 Permutation sort as an object program 114
6.3 Specialized version of meta-interpreter running permutation sort 115
8.2 Graph coloring synthesis after structure filtering 144

A.1 Prime sieve with filters using Peano encoding 151
A.2 Lucky numbers sieve 152
A.3 N queens 153
C.1 Extended meta-interpreter for specialization 159
D.1 Meta-interpreter with concrete selection by an abstract instantiation-based selection rule 165
F.1 Speculative permutation sort 177
F.2 Alternating graph coloring synthesis 180
F.3 Continuation passing graph coloring synthesis 184
F.4 Complete Prolog synthesis of confused queens 186

List of Tables

7.1 Program transformation timings, in seconds 132

B.1 Performance comparison for permutation sort 156

B.2 Performance comparison for confused queens 157

Chapter 1

Introduction

1.1 Coroutines and logic programs

1.1.1 Logic programming

Logic programming is a programming paradigm which first came into being in the early 1970s. It was born from earlier work in logic, especially in automated theorem proving, which took place in the 1960s. A key contribution was made in 1965, when Robinson published the resolution mechanism [73] based on syntactical unification. Resolution is a complete mechanism for deduction in first-order logic, based on proof by refutation. This means that, if a statement is false given some prior knowledge base, resolution can always deduce a contradiction. Furthermore, resolution is highly systematic. Both properties together entail that resolution is well-suited for use in an automated tool.

The key insight in implementing such a tool was that large subsets of first-order logic do not only have a declarative reading[1], but also possess a procedural[2] reading. Based on this insight, Colmerauer and his students developed the first — and still most widely used — logic programming language, Prolog [15]. Prolog uses a variant of resolution known as SLDNF[3]-resolution which can be implemented very efficiently and which is sound and complete for the (substantial) subset of first order logic based on Horn clauses. Kowalski [44]

[1] i.e. representing knowledge
[2] i.e. representing instructions
[3] Selective Linear Definite-clause with Negation as Failure

showed that Horn clauses are at the foundation of Prolog and that Prolog could be used as a general-purpose programming language.[4]

The definitive introduction to the logic programming paradigm was written by Lloyd [53]. We will adopt terminology from this work and direct the reader to it for an explanation of general logic programming concepts. These concepts transfer over to Prolog, but Prolog adds the conventions of depth-first traversal of a search space, left-to-right selection of atoms and negation as failure.

Note that a distinction is often made between "pure" logic programs and "impure" logic programs. The former rely only on SLD-resolution and Horn clauses. The latter use SLD-resolution and Horn clauses as a foundation, but make use of other mechanisms as well. Commonly used impure mechanisms included in Prolog are negation as failure, the "cut" operator for reducing search space size, side-effectful computations such as input and output procedures, etc. These are sometimes invaluable to programmers, but they also undermine the simplicity which makes logic programs highly readable and suited to static analysis techniques. The current work mostly focuses on pure logic programs, but will occasionally make use of impure features. Any deviations from the pure logic programming paradigm will be justified explicitly.

1.1.2 Coroutines

Coroutines were first described in the context of the design of a modular compiler by Conway [16]. He first outlined the concept in fairly abstract terms as follows:

> [...] each module may be made into a *coroutine*; that is, it may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program. There is no bound placed by this definition on the number of inputs and outputs a coroutine may have.

The key idea put forward by Conway is the use of coroutines in so-called "separable" programs. In common shell programming terminology, these are programs constructed using pipes. These are also often referred to as "generate-

[4] Colmerauer had originally intended it to be a tool for natural language processing.

and-test" programs. The coroutine concept can be applied to programs which process (or "test") input and generate output asynchronously.[5]

The concept of coroutines was studied in depth by Marlin, who developed the simple language ACL[6] specifically for this purpose [55]. Marlin pointed out the fundamental characteristics of any coroutine:

- the values of data local to a coroutine persist between successive calls
- the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage

In the context of logic programming, coroutines are predicates whose evaluation may be suspended before they are run to completion and which can be resumed. In other words, they do not adhere to the depth-first control flow mechanism found in Prolog. The generate-and-test approach is common in logic programming, therefore coroutines can preserve the declarative nature of generate-and-test programs while improving their often poor performance.[7]

1.2 Research goal

The current research aims to develop a transformation which retains the flow of data introduced into a logic program through the use of coroutines, while eliminating the use of an explicit mechanism for coroutines. In this sense, the goal is to reverse the motivating transformation used by Conway [16], which used the coroutine abstraction to eliminate other control flow constructs and obtain a more human-readable program, albeit in COBOL. Similar work was performed in the late 1980s [8] [9] [19] and was known as compiling control. Unfortunately, compiling control left several questions open. This work aims to deal with these questions by framing compiling control as a combination of more thoroughly studied techniques.

[5] Input is processed and output is produced asynchronously in UNIX pipes, hence their components can be regarded as coroutines. This is not the case for MS-DOS pipes, hence their components can be regarded as standard subroutines.

[6] A Coroutining Language

[7] Anecdotally, the authors of the first ten Prolog programming contests [23] state that many problems have a declarative solution which uses the generate-and-test approach.

1.3 Motivation

The original motivation behind compiling control was to mitigate performance issues introduced by coroutining mechanisms. For modern logic programming languages, the performance of these mechanisms is less of an issue, though the technique can still yield performance improvements. Our goal is to transform away the control flow abstraction to simplify the total set of mechanisms needed to execute a program. Doing so makes the program more amenable to static analysis techniques such as termination analysis and type checking. Such techniques are typically developed under the assumption of Prolog's standard selection rule. When they are not (e.g. in [76]), they apply their own ad hoc transformation which is sound w.r.t. to the analyzed property. An additional advantage is that the transformation makes Prolog programs more portable, as different Prolog systems support different mechanisms for coroutines. By automating the technique for a realistic logic programming language, we hope to make it accessible for real-world use cases.

1.3.1 Coroutining mechanisms

The oldest mechanisms for coroutines in a logic programming language are the *dif*/2 and *freeze*/2 predicates [14]. The *dif*/2 predicate expresses disequality of two terms. It succeeds immediately if its two arguments are guaranteed to be different terms and fails if they are identical, but it is blocked if its two arguments could unify at some later point. It is reactivated once the identity of the two arguments can be decided upon. The *freeze*/2 predicate delays the execution of a goal (its second argument) until its first argument is no longer a free variable. A somewhat more general predicate which is also implemented in several major Prolog systems is *when*/2. Like *freeze*/2, it executes its second argument when a certain condition specified by its first argument is met. However, more flexibility with regard to this condition is allowed: disequality and a lack of freeness, as well as groundedness can be checked as atomic conditions, and conditions can be combined using a disjunction and conjunction operator.

Another popular mechanism for coroutines is found in SICStus Prolog, which uses so called "block declarations" [11]. These are inserted into a program using the special syntax `:- block :BlockSpec, ..., :BlockSpec.`, where each `BlockSpec` has the form of an atom whose arguments are either `-` or `?`. These arguments specify conditions for delaying the evaluation of any atom with the corresponding predicate symbol. Specifically, a block declaration evaluates to true if and only if all arguments specified as `-` are free variables. An atom is delayed if any corresponding block declaration evaluates to true. Note that,

unlike the aforementioned predicates, block declarations operate at the predicate level. As such, they implement a clearer separation between program *logic* and *control*, the two basic components of an algorithm, according to Kowalski. On the other hand, specifying control at the predicate level is not as fine-grained as specifying it at the atom level, so neither approach makes the other obsolete.

An entirely different approach to coroutining is to write a meta-interpreter which encodes a non-standard selection rule. More precisely, let $\leftarrow \ldots, A, \ldots, B, \ldots$ be a goal encountered during the evaluation of some program P by the meta-interpreter. If this meta-interpreter first resolves A (or B) and does not reduce said atom to the empty goal before resolving $B\theta$ (or, conversely, $A\sigma$), where θ and σ are substitutions, it is a coroutining meta-interpreter. One such meta-interpreter is provided in appendix D.[8]

1.3.2 Related mechanisms

Coroutines are not the only mechanisms for rescheduling evaluation. A direct descendant of the logic programming paradigm is the constraint (logic) programming paradigm. A constraint is a logical relation between variables which take values from a given domain. In a constraint logic programming language, constraints take the form of predicates and are integrated with a logic language. For instance, the widely used ECLiPSe constraint programming language has a Prolog subset. Indeed, the *dif*/2 predicate mentioned earlier expresses a constraint. Ideally, constraints are rarely satisfied through domain search and are instead selected dynamically, based on whether they can be satisfied through the propagation of known values. This is similar to the use of coroutines for dynamic goal selection, but the granularity is more coarse: the constraint solver schedules constraints for evaluation and the programmer only defines a certain encoding of a problem (which can allow more or less effective inference) and overall heuristics. While not a direct implementation of a constraint programming language, CHR [28] should be mentioned in this regard. CHR is a guest language originally developed for building constraint solvers. It typically uses Prolog as a host language and also enables suspending and resuming the evaluation of Prolog predicates. Strictly speaking, this means that it can also be used to implement coroutines in Prolog, though it is more general.

Delimited continuations and effect handlers [26] [72] are other mechanisms for manipulating control. These were recently introduced in Prolog [75] [74]

[8] This meta-interpreter implements an "instantiation-based selection rule", a concept which is introduced in section 3.2.4.

and are particularly useful for the definition of new language features at the program-level, rather than the meta-program level (e.g. tabling as a library [24]).

Concurrent logic programming languages, such as PARLOG and concurrent Prolog [78] also incorporate a flow of control which differs from that found in general-purpose Prologs. In these languages, a goal is regarded as consisting of *concurrent* processes and variables shared between processes enable inter-process communication. Through the use of guards, rules are applied when the obtained data permits it. This is similar to how coroutines are typically initiated or resumed only when they have enough input data to perform a useful computation. Unlike standard logic programming languages with coroutining facilities, however, concurrent logic programming languages emphasize don't-care nondeterminism, committed choice selection of rules and AND-parallelism.

1.4 Related work

In this section, we only provide an overview of work with a similar high-level goal as ours. Work related only to specific aspects of our approach is discussed in the appropriate chapters. The current work is a direct continuation of the work performed on compiling control [8] [9] [19]. It applies the same basic approach, but it uses established frameworks and explores new examples.

A first reformulation of the compiling control transformation was proposed in the context of the "programs-as-proofs" paradigm, in [81]. It was shown that compiling control transformations, to a limited extent, could be formalized in a proof-theoretic program synthesis context.

In [5], the compiling control transformation was revisited on the basis of a combination of abstract interpretation and constraint processing. This improved the formalization of the technique, but it did not clarify the relation with partial deduction.

Other work in the same area [56] also applies static analysis techniques to coroutining programs, but the end goal is to check whether certain optimizations can be applied and not to remove coroutining mechanisms from the program entirely.

The use of abstract interpretation and transformation of coroutining programs to obtain a non-coroutining program is not entirely unique. In [37], a program which does not use coroutining mechanisms *is* produced. However, said program does not behave identically for a top-level call pattern. Instead, it is a result of abstract compilation [4]. It can be regarded as a specialized program analyzer for that particular program. Running this analyzer (which can be done under a

left-to-right selection rule) for a particular call pattern yields a set of generated call patterns linked to a particular abstract domain. These provide useful information, but the resulting abstract program cannot be analyzed directly using an existing analysis tool.

Work on properties such as termination [76] in coroutining programs also exists, but such work includes a property-preserving transformation that does not carry over to other techniques.

A transformation from CHR to Prolog has also been applied to prove properties of CHR programs, notably termination [71] and type analysis [70].

To the best of our knowledge, the original work on compiling control is the only work which directly allows existing, coroutining-agnostic analyzers to be reused.

1.5 Overview

The remainder of this thesis is structured as follows. Chapter 2 provides an introduction to compiling control, abstract interpretation and partial deduction (and some of its variants). Chapter 3 details the notion of an "instantiation-based selection rule", i.e. a type of selection rule which is amenable to compilation using compiling control. The same chapter casts compiling control as an instance of abstract conjunctive partial deduction, a framework which integrates abstract interpretation and partial deduction and details an additional abstraction, *multi*, which, when added to the domain used for abstract conjunctive partial deduction, can be used to compile a more meaningful set of programs. Chapter 4 shows that the *multi* abstraction is still insufficient to analyze all programs with an instantiation-based selection rule and presents further, more experimental, abstractions to remedy this. It also provides an example of a program which cannot be analyzed because the abstract domain cannot accurately capture all aspects and another experimental solution to this issue. Chapter 5 discusses an alternative approach to synthesis, in which the target language is CHR, rather than Prolog. Chapter 6 confirms the long-standing conjecture [32] that compiling control can also be realized as an instance of the first Futamura projection, simplifying the program synthesis. In chapter 7, the applicability and key components of the current implementation are discussed. Chapter 8 shows how the promise of compiling control can be realized by applying a static technique for termination analysis to programs for which an instantiation-based selection rule is simulated through meta-interpretation and through program transformation. Chapter 9 wraps up with an overview of the contributions made in this thesis and with avenues for future work.

Chapter 2

Preliminaries

2.1 Conventions

It is assumed that the reader has a basic familiarity with logic programs and Prolog. For an introduction to logic programming, see [53]. For an introduction to Prolog, any textbook will do (e.g. [6]). Variable names adhere to the Prolog variable syntax. Names of constants (except for numbers) and functors adhere to the Prolog syntax for atoms. In the special case of lists, we use the SWI-Prolog convention with `'[|]'` instead of `'.'`. We also use the widespread Prolog syntactic sugar for lists, e.g. `[1,2,3]` is syntactic sugar for `'[|]'(1,'[|]'(2,'[|]'(3,[])))`. With regard to terminology, we will use the same as in [53]. This is important, as general logic programming terminology and Prolog terminology sometimes conflict (e.g. "atom", which is how Prolog refers to an atomic term and which is how general logic programming refers to an atomic formula).

2.2 Compiling Control

Compiling control [8] [9] is the precursor to the current research. It is a transformation technique which compiles the runtime behavior of logic programs executed under a non-standard selection rule to logic programs which are totally equivalent under the standard left-to-right execution rule of Prolog. The main motivation behind compiling control was to avoid the "substantial overhead" [19] introduced by separate control mechanisms. The technique is designed to work

in two phases. In the first, a part of the computation flow of the source program, executed under the non-standard rule, is analyzed for a specific query pattern. This is done by simulating its execution under the assumption of certain groundness and aliasing characteristics of variables. This results in a part of a "symbolic trace tree" that represent the program flow for said query pattern. A complete symbolic trace tree is typically infinite and each node can be thought of as a program state. The path between two nodes represents the computation that occurs when moving between these states.

This is best illustrated with an example program. The prototypical example is permutation sort. This is an implementation of a sorting algorithm which works by permuting a list and checking whether it is ordered. It is a clear-cut example of a generate-and-test program. It can be run under the standard left-to-right selection rule of Prolog, but it can also be run under a selection rule which applies an ordering check whenever a pair of adjacent elements has been computed. Executable Prolog code for this program is given in listing 2.1.

```
perm([],[]).
perm([X|Y],[U|V]) :- select(U,[X|Y],W), perm(W,V).
ord([]).
ord([X]).
ord([X,Y|Z]) :- X =< Y, ord([Y|Z]).
```

Listing 2.1: Permutation sort

The SICStus Prolog code in listing 2.2 interleaves these calls in the same way for the intended call pattern, but it requires some changes to the source program and uses block declarations, which are not portable across Prolog implementations.

```
:- use_module(library(lists),[select/3]).
:- block ord(-).
:- block ord_(?,-,?).
permsort(X,Y) :- ord(Y), perm(X,Y).
perm([],[]).
perm([X|Y],[U|V]) :- select(U,[X|Y],W), perm(W,V).
ord([]).
ord([X]).
ord([X,Y|Z]) :- ord_(X,Y,Z).
ord_(X,Y,Z) :- X =< Y, ord([Y|Z]).
```

Listing 2.2: Permutation sort in SICStus

The top part of the symbolic trace tree for the expected top-level query is given in figure 2.1. In this figure, the following notational conventions hold:[1]

- A bold variable stands for a value which will be ground at runtime.
- An overlined variable stands for a free variable at runtime.
- An atom underlined once is selected for resolution with clauses whose heads unify with said atom.
- An atom underlined twice is not analyzed, but is evaluated as if it was a builtin.
- The equalities on the edges indicate the substitutions that are applied during resolution.

We postpone a formal definition of the type of selection rule that can be dealt with. We assume it is clear from the example that the atom selections indeed implement the desired behavior.

From the symbolic trace tree, an infinite program with the desired behavior can be obtained by introducing a new predicate for each state, encoding each state transition as a Horn clause and applying the substitutions which are part of the transition. For instance, the transition from state 2 to state 3 can be encoded as the following clause: $s2(perm([],[]),ord([])) \leftarrow s3(ord([]))$. If a goal is to be fully evaluated (treated as a builtin), it is added to the body. For instance, because the evaluation of $select/3$ is not interleaved with any predicate defined in listing 2.1: $s4(select(V1,[U1|K1],W1),perm(W1,L1),ord([V1|L1])) \leftarrow select(V1,[U1|K1],W1),s5(perm(W1,L1),ord([V1|L1]))$.

To obtain a runnable, finite program, the infinite symbolic trace tree is encoded as a finite "symbolic trace graph". This is done by finding a finite number of generalized representations for the program states and encoding transitions between these generalized representations as Horn clauses. For permutation sort, this is easy, as there is an infinite number of program states, but only a finite number of program states after variable renaming. Indeed, every state from state 10 onward is a renaming of an earlier state. Therefore, the clause $s9(perm(W2,L2),V1 \leq V2,ord([V2|L2])) \leftarrow V1 \leq V2,s10(perm(W2,L2),ord([V2|L2]))$ can be replaced with the clause $s9(perm(W2,L2),V1 \leq V2,ord([V2|L2])) \leftarrow V1 \leq V2,s5(perm(W2,L2),ord([V2|L2]))$ and no further clauses are needed to

[1] The notational conventions used for this example are somewhat different from those in the original work on compiling control to improve consistency with the rest of this text.

describe the transitions starting in $s10, s11, \ldots$ This is visualized in figure 2.2.

A working program can be synthesized in the above manner, but it is easy to apply inline expansion to states which can only be resolved using a single clause. This gives rise to a "reduced trace graph". The reduced tree graph for permutation sort is shown in figure 2.3. A more concise program can therefore be obtained, either through inline expansion of a program synthesized from the program trace graph, or by encoding each transition in the reduced trace graph directly. In this case, all the equalities that must hold for the grouped transitions to apply are applied and all fully evaluated atoms are added, in their respective order, to the head of the clause. For instance, the cycle in the reduced trace graph is encoded as $s5(perm([U2|K2], [V2|L2]), ord([V1, V2|L2])) \leftarrow select(V2, [U2|K2], W2), V1 \leq V2, s5(perm(W2, L2), ord([V2|L2]))$.

Compiling control was formalized and proven correct under certain technical conditions, but it possesses certain drawbacks. Most importantly, it is an ad hoc solution. That is, it does not leverage correctness results for existing analysis frameworks. As a result, showing that the analysis phase of the information is complete for a specific program requires a manual proof by induction. In the case of permutation sort, "similar" states are simply renamed instances of each other, but other examples of similar states require more creativity on the part of the individual performing the analysis. The original work [9] on the subject does state that the technique was automated, but it makes several assumptions. First, it assumes assumes that a "sufficiently large" segment of the trace tree is provided as input without stating how large this segment is required to be. Indeed, the original work explicitly considers this an open question. Second, it assumes a non-automatic feature, i.e. that a user can specify that the instantiation patterns of some parameters are irrelevant to the computation rule. Third, the notion of similarity of states was based on ad hoc heuristics. While the original implementation is no longer available, the published work states that it is "very hard to decide whether or not we have gathered sufficiently many heuristics to cover all possible input programs and computation rules".

Finally, the original work on compiling control also conjectures that any program can be transformed when its computation rule "is finite and its decisions are based solely on the instantiation patterns of the subgoals in the goal statement." In this work, we revisit these open questions and build towards a more complete analysis.

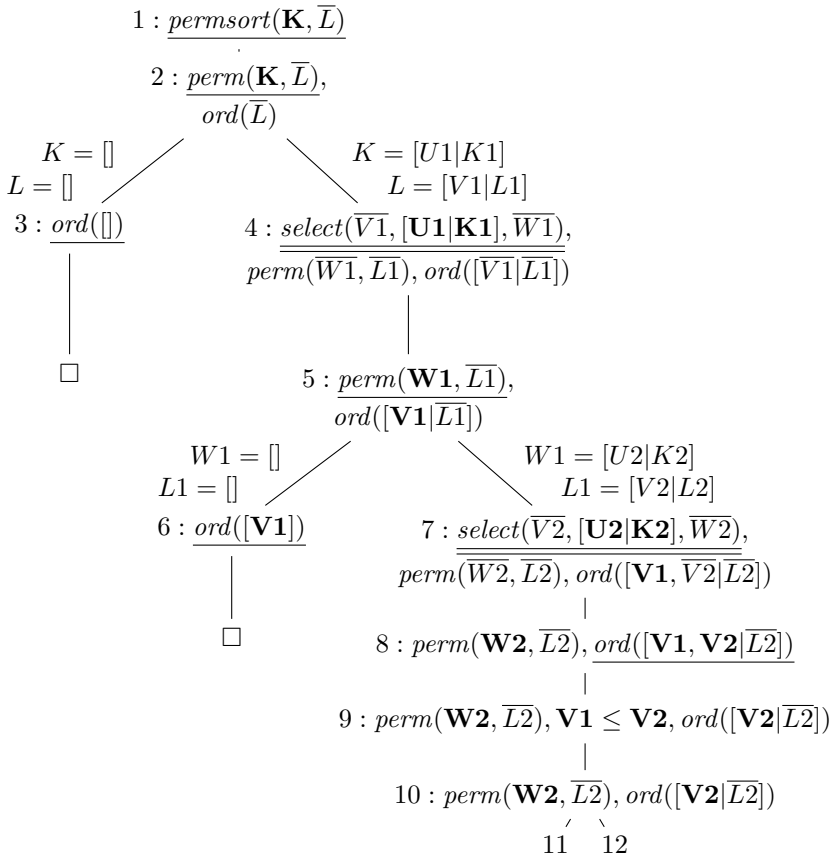


Figure 2.1: Symbolic trace tree for permutation sort with compiling control

2.3 Partial Deduction

Partial deduction is a technique for logic program specialization originally introduced in [43] and formalized in [54]. Intuitively, the idea behind partial deduction is to compute a set of derivation trees for some top-level goal A such that all expected queries instantiate A. The computations represented by the branches of these trees can then be collected into logical implications, also referred to as "resultants". These resultants are then encoded as logic program clauses, so that a logic program is obtained which is totally equivalent to the original program for all queries which instantiate A, but not necessarily for other queries. The derivation trees are SLDNF-trees, therefore the results

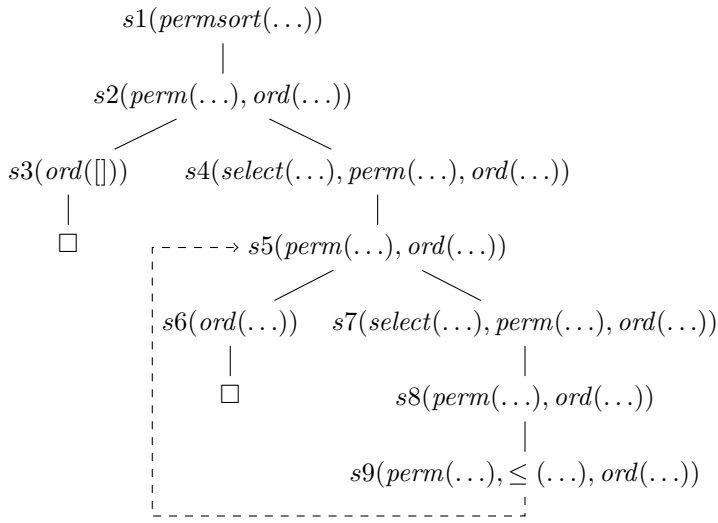


Figure 2.2: Symbolic trace graph for permutation sort with compiling control

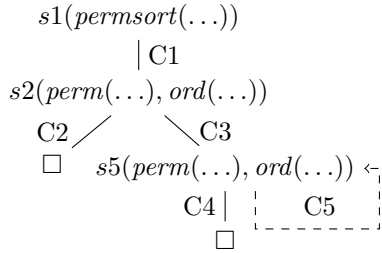


Figure 2.3: Reduced trace graph for permutation sort with compiling control

for partial deduction do not just apply to definite programs, but to normal programs. Here, we reproduce the key definitions from [54].

Definition 1 (Resultant):
A resultant is a first-order formula of the form $Q_1 \leftarrow Q_2$ where Q_i is either absent or a conjunction of literals. Any variables in Q_1 or Q_2 are assumed to be universally quantified at the front of the resultant.

Definition 2 (Resultant of a derivation):
Let P be a normal program, G a normal goal $\leftarrow Q$ and $G_0 = G, G_1, \ldots, G_n$ an SLDNF-derivation of $P \cup \{G\}$, where the sequence of substitutions is $\theta_1, \ldots, \theta_n$ and G_n is $\leftarrow Q_n$. Let θ be the restriction of $\theta_1, \ldots, \theta_n$ to the variables in G.

Then we say the derivation has length n with computed answer θ and resultant $Q\theta \leftarrow Q_n$. (If $n = 0$, the resultant is $Q \leftarrow Q$.)

With the notion of a resultant of a derivation, the notion of a partial deduction can be defined.[2]

Definition 3 (Partial deduction of a normal program with respect to a set of atoms):
Let P be a normal program, A an atom and T an SLDNF-tree for $P \cup \{\leftarrow A\}$. Let $\leftarrow G_1, \ldots, \leftarrow G_r$ be (nonroot) goals in T chosen so that each nonfailing branch of T contains exactly one of them. Let $R_i (i = 1, \ldots, r)$ be the resultant of the derivation from $\leftarrow A$ down to $\leftarrow G_i$ given by the branch leading to $\leftarrow G_i$. Then the set of clauses $R_1, \ldots, R_r$ is called a partial deduction of A in P. If $\mathcal{A} = \{A_1, \ldots, A_s\}$ is a finite set of atoms, then a partial deduction of $\mathcal{A}$ in P is the union of the partial deductions of $A_1, \ldots, A_s$ in P. A partial deduction of P with respect to $\mathcal{A}$ is a normal program obtained from P by replacing the set of clauses in P whose head contains one of the predicate symbols appearing in $\mathcal{A}$ (called the partially evaluated predicates) with a partial deduction of $\mathcal{A}$ in P.

We also reproduce an example from [54]. Consider the definite program given by:

$$grandmother(X, Y) \leftarrow mother(X, Z), parent(Z, Y)$$

$$grandfather(X, Y) \leftarrow father(X, Z), parent(Z, Y)$$

$$parent(X, Y) \leftarrow mother(X, Y)$$

$$parent(X, Y) \leftarrow father(X, Y)$$

$$mother(sue, simon) \leftarrow true$$

$$mother(sue, patrick) \leftarrow true$$

$$mother(monica, sue) \leftarrow true$$

together with a definition for $father/2$. Let G be $\leftarrow grandmother(X, simon)$. Then a partial evaluation of P with respect to $\{grandmother(X, simon)\}$ (shown in figure 2.4) produces the following definition for $grandmother/2$, where the atom selected by the selection rule was always the leftmost atom, unless the predicate for the atom was $father/2$, in which case the derivation was terminated:

$$grandmother(sue, simon) \leftarrow father(simon, simon)$$

[2]The definition is reproduced from [54], though there it is referred to as the "partial evaluation of a logic program".

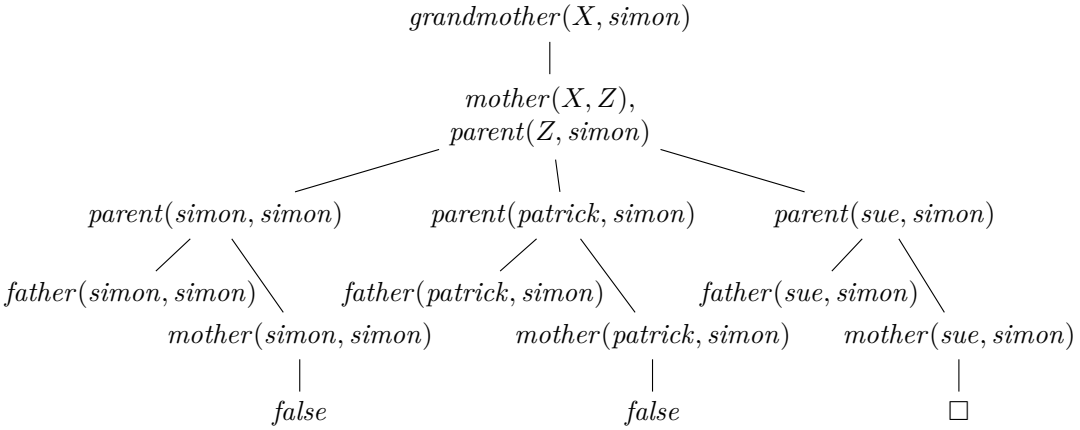


Figure 2.4: Derivation tree for partial deduction example

$$grandmother(sue, simon) \leftarrow father(patrick, simon)$$
$$grandmother(monica, simon) \leftarrow true$$
$$grandmother(monica, simon) \leftarrow father(sue, simon)$$

This program, referred to as P', together with the definition of $father/2$ can be used to answer goals of the form $grandmother(X, simon)$. We refer to $father/2$ as a "residual" predicate, meaning that it is not specialized, but that we retain its definition in the specialized program.

Finally, we reproduce the notions of "closedness" and "independence".

Intuitively, under closedness, all atoms occurring in a partial deduction are instances of an atom in the root of some derivation tree. The implication is that a goal is never reduced to a goal that has not been specialized, so that completeness is ensured.

Definition 4 (Closedness):
Let S be a set of first-order formulae and $\mathcal{A}$ a finite set of atoms. We say S is $\mathcal{A}$-closed if each atom in S with a predicate symbol occurring in an atom in $\mathcal{A}$ is an instance of an atom in $\mathcal{A}$.

The concept of independence is used to impose a technical condition which is needed for the partially evaluated program to be equivalent to the original program.

Definition 5 (Independence):
Let $\mathcal{A}$ be a finite set of atoms. We say $\mathcal{A}$ is independent if no two atoms in $\mathcal{A}$ have a common instance.

The set $\mathcal{A}$ consists of atoms for which resultants are generated. Therefore, if $\mathcal{A}$ is not independent, a common instance of two distinct atoms may be resolved with the resultants of both atoms. This means that additional answers are produced and that procedural equivalence is lost. Luckily, independence can always be guaranteed through the use of a "renaming transformation" [3]. Such a transformation introduces new defining predicates for atoms in $\mathcal{A}$ which are not independent. These new predicates have distinct symbols, therefore their instances are always independent. The transformation then replaces the dependent elements of $\mathcal{A}$ with instances of these new predicates, guaranteeing that $\mathcal{A}$ is independent.

2.4 Conjunctive Partial Deduction

Conjunctive partial deduction [21] is a conceptually straightforward extension of partial deduction. For standard partial deduction, the elements of $\mathcal{A}$ are atoms. If a leaf node of a derivation tree contains a conjunction, said conjunction must be split into atoms, which are analyzed separately. This precludes certain optimizations which can be performed if conjunctions are analyzed as a whole.

The following example of conjunctive partial deduction is taken from [21]. To avoid confusion in the notation for a set of conjunctions, we will denote a conjunction of atoms, $A_1, \ldots, A_n$, as $\wedge(A_1, \ldots, A_n)$, where we assume the existence of $\wedge/n$ functors for all possible arities n.

Example 1 (Conjunctive partial deduction):
Let P be the following program, which computes both the maximum M and the length L of a list X:

(C_1) $max_length(X, M, L) \leftarrow max(X, M), length(X, L)$

(C_2) $max(X, M) \leftarrow max1(X, 0, M)$

(C_3) $max1([], M, M) \leftarrow true$

(C_4) $max1([H|T], N, M) \leftarrow H \leq N, max1(T, N, M)$

(C_5) $max1([H|T], N, M) \leftarrow H > N, max1(T, H, M)$

(C_6) $length([], 0) \leftarrow true$

(C_7) $length([H|T], L) \leftarrow length(T, K), L \text{ is } K + 1$

$$\underline{max_length(X, M, L)}$$
$$\mid C_1$$
$$\underline{max(X, M)}, length(X, L)$$
$$\mid C_2$$
$$max1(X, 0, M), length(X, L)$$

Figure 2.5: SLD-tree τ_1 for example 1

Running P as given, if X is instantiated to a list, requires X to be traversed twice. Let $\mathcal{A} = \{max_length(X, M, L), \wedge(max1(X, N, M), length(X, L))\}$. Consider the finite SLD trees τ_1 and τ_2 in figures 2.5 and 2.6, where arcs have been labelled with clause numbers used for the derivation steps and where the root of each tree corresponds to an element of $\mathcal{A}$. The associated conjunctive partial deductions are $resultants(\tau_1) = \{R_{1,1}\}$ and $resultants(\tau_2) = \{R_{2,1}, R_{2,2}, R_{2,3}\}$, which are as follows:

$(R_{1,1})$ $max_length(X, M, L) \leftarrow max1(X, 0, M), length(X, L)$

$(R_{2,1})$ $max1([], N, N), length([], 0) \leftarrow true$

$(R_{2,2})$ $max1([H|T], N, M), length([H|T], L) \leftarrow$

$$H \leq N, max1(T, N, M), length(T, K), L \text{ is } K + 1$$

$(R_{2,3})$ $max1([H|T], N, M), length([H|T], L) \leftarrow$

$$H > N, max1(T, H, M), length(T, K), L \text{ is } K + 1$$

The concepts from partial deduction given above are easily generalized to use conjunctions rather than atoms. To encode resultants as logic program clauses, conjunctions are renamed, i.e. a fresh predicate symbol for that conjunction is introduced. In order to decide which conjunctions in the resultant are grouped and define a renamed predicate, a "partitioning function" is used.

Definition 6 (Partitioning function):
Let $=_r$ denote syntactic identity up to reordering on conjunctions. Let $\mathcal{M}(A)$ denote all multisets composed of elements of the set A. Let $\mathcal{C}$ denote the set of all conjunctions of atoms over the given alphabet. A partitioning function is a mapping $p : \mathcal{C} \rightarrow \mathcal{M}(\mathcal{C})$ such that for any $C \in \mathcal{C} : C =_r \wedge_{Q \in p(C)} Q$.

Definition 7 (Atomic renaming):
An atomic renaming ν for a given set of conjunctions $\mathcal{A}$ is a mapping from $\mathcal{A}$ to atoms, such that:

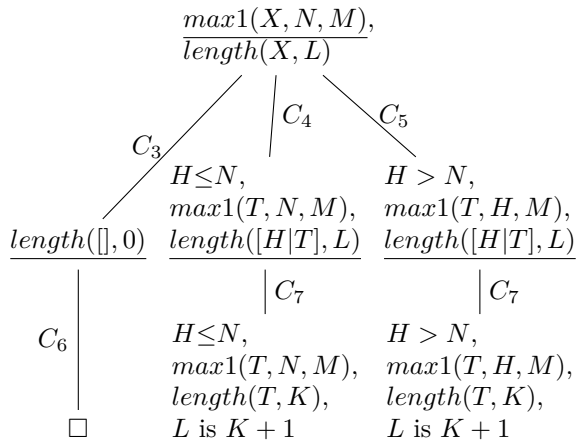


Figure 2.6: SLD-tree τ_2 for example 1

- for each $A \in \mathcal{A} : vars(\nu(A)) = vars(A)$
- for $A, A' \in \mathcal{A}$ such that $A \neq A'$: the predicate symbols of $\nu(A)$ and $\nu(A')$ are distinct

Example 2 (Atomic renaming):
A possible atomic renaming for $\mathcal{A}$ as in example 1 is:

- $\nu(max_length(X, M, L)) = q1(X, M, L)$
- $\nu(max1(X, N, M) \wedge length(X, L)) = q2(X, N, M, L)$

Note that ν also renames atoms in $\mathcal{A}$. This is beneficial, as it immediately satisfies the independence condition (definition 5).

Definition 8 (Renaming function):
Let ν be an atomic renaming for $\mathcal{A}$ and p a partitioning function. A renaming function $\rho_{\nu,p}$ for $\mathcal{A}$ (based on ν and p) is a mapping from conjunctions to conjunctions such that:

$$\rho_{\nu,p}(B) =_r \wedge_{C_i \in p(B)} \nu(A_i)\theta_i \text{ where each } C_i = A_i\theta_i \text{ for some } A_i \in \mathcal{A}$$

Definition 9 (Conjunctive partial deduction):
Let P be a program, $\mathcal{A} = \{Q_1, \ldots, Q_n\}$ be a finite set of conjunctions and $\rho_{\nu,p}$ be a renaming for $\mathcal{A}$ based on the atomic renaming ν and the partitioning

function p. For each $i \in \{1, \ldots, n\}$, let R_i be a conjunctive partial deduction of Q_i in P and let $P_{\mathcal{A}} = \{R_i | i \in \{1, \ldots, n\}\}$. Then the program

$$\{\nu(Q_i)\theta \leftarrow \rho_{\nu,p}(B) | Q_i\theta \leftarrow B \in R_i \wedge 1 \leq i \leq n \wedge \rho_{\nu,p}(B) \text{ is defined}\}$$

is called the conjunctive partial deduction of P w.r.t. $\mathcal{A}$,$P_{\mathcal{A}}$ and $\rho_{\nu,p}$.

Let P' be the following conjunctive partial deduction of P as given in example 1:

$$q1(X, M, L) \leftarrow q2(X, 0, M, L)$$

$$q2([], N, N, 0) \leftarrow$$

$$q2([H|T], N, M, L) \leftarrow H \leq N, q2(T, N, M, K), L \text{ is } K + 1$$

$$q2([H|T], N, M, L) \leftarrow H > N, q2(T, H, M, K), L \text{ is } K + 1$$

Note that the list bound to X will only be traversed once. This optimization is the direct result of grouping (or "tupling") the $max1/3$ and $length/2$ atoms.

2.5 Abstract Interpretation

Abstract interpretation is a framework for the static analysis of programs. It was introduced by Cousot and Cousot [17] in the late 1970s. It generalized over several pre-existing static analysis methods by providing generic conditions to obtain a complete and consistent set of program properties, a so-called "safe approximation" of program semantics. Key ideas in abstract interpretation are the following:

- an "abstract domain" which mirrors the concrete domain, but whose values represents sets of concrete values[3]
- an organization of both the concrete and abstract domain as a complete semilattice
- a monotone "abstraction function" α which computes an abstraction over its argument and a monotone "concretization function" γ which computes the denotation of its argument

[3]Different texts on abstract interpretation formulate this in different ways. In Cousot and Cousot's original work, the fact that a single abstract value represents a set of concrete values is somewhat implicit and is handled through extra concepts and notation.

- the formulation of a solution of an abstract interpretation as a fixpoint whose existence can be guaranteed

Cousot and Cousot's seminal work was language-independent, though it was only illustrated by the authors using imperative programs. The approach was first applied to logic programming in [60] and a framework for the abstract interpretation of logic programs was introduced in [7]. The latter defined the notion of a correct logic program analysis and showed that global correctness follows from local correctness of a handful of application-dependent operations. In this way, users could obtain a safe program analysis simply by defining these operations and proving their local correctness.

In the remainder of the text, we will sometimes refer to the original work on compiling control in [9]. When discussing the analysis phase of this work, we will refer to "symbolic" values rather than "abstract" values to highlight the fact that the compiling control analysis was not originally formulated as a form of abstract interpretation.

2.6 Abstract Conjunctive Partial Deduction

Abstract conjunctive partial deduction is a framework, described in [45], which integrates abstract interpretation and (conjunctive) partial deduction. In [45], the program analysis aspect of partial deduction is reformulated as a form of abstract interpretation. The result of this abstract interpretation is the set $\mathcal{A}$ described above. A key observation made while reframing partial deduction as an abstract interpretation is that partial deduction does not distinguish between the concrete and abstract domain: concrete conjunctions function as abstract conjunctions. Based on this insight, Leuschel shows how several specialized variants of partial deduction can be regarded as forms of partial deduction with an alternative abstract domain, making them instances of the general abstract conjunctive partial deduction framework. In this framework, as well, correctness of instances of the framework follows from local correctness conditions. These conditions apply to two application-dependent functions, *aunfold* and *aresolve*. In chapter 3, we demonstrate how simple instances of compiling control can be regarded as instances of this framework and in 4 we point out why abstract conjunctive partial deduction cannot be used to establish correctness of all instances of compiling control.

Chapter 3

Abstract Conjunctive Partial Deduction

In the current chapter, we show how compiling control can be seen as an instance of Leuschel's abstract conjunctive partial deduction framework in some cases. By doing so, we inherit correctness properties from the abstract conjunctive partial deduction framework. We also point out that the assumptions made by the abstract conjunctive partial deduction framework are too strict to encompass every application of compiling control and propose an abstraction that can alleviate the issue. The current chapter is based on previously published work [22] [65], but it introduces a formal definition of the selection rule, presents the sameleaves program in more detail and uses notation which is consistent with later work.

3.1 A simple example

In this section, we illustrate the foundation of our approach, again through the use of the permutation sort example.

3.1.1 Basic concepts

First, we introduce the concrete and the abstract domain.

Given a program P, Con_P, Var_P, Fun_P and $Pred_P$ respectively denote the sets of all constants, variables, functors and predicate symbols in the language underlying P. $Term_P$ will denote the set of all terms constructable from Con_P, Var_P and Fun_P. $Atom_P$ denotes the set of all atoms which can be constructed from $Pred_P$ and $Term_P$. We will often need to refer to conjunctions of atoms of $Atom_P$ and we denote the set of all such conjunctions as $ConAtom_P$. Dom_P will denote $Term_P \cup Atom_P \cup ConAtom_P$.

The abstract domain is built upon two types of abstract variables: a_i and g_j, $i, j \in \mathbb{N}_0$. The symbols g_j denote any ground term in the concrete language. The basic intuition for the symbols a_i is that they represent variables of the concrete domain. However, we want the concretization of abstract terms to be closed under substitution (if an abstract term denotes some concrete term — in this case, a free variable — then it should also denote all of its instances). This is a requirement to make use of the abstract conjunctive partial deduction framework. Therefore, an abstract variable a_i will actually represent any term of the concrete language. This limits the granularity with which control can be compiled somewhat, but still allows us to compile all the examples from earlier work on compiling control. We discuss the limitations imposed by this choice in 4. The set of all abstract variables is denoted $AVar_P$.

Based on Con_P, there is a set of abstract constants, $ACon_P$, consisting of the same elements, viewed as elements of the abstract domain. Based on $AVar_P$, there is a corresponding set of abstract terms, $ATerm_P$, which consists of the terms that can be constructed from $AVar_P$, $ACon_P$ and Fun_P. $AAtom_P$ denotes the set of abstract atoms, i.e. the atoms which can be constructed from $ATerm_P$ and $Pred_P$. $AConAtom_P$ denotes the set of conjunctions of elements of $AAtom_P$. Finally, $ADom_P$ denotes $ATerm_P \cup AAtom_P \cup AConAtom_P$.

The subscripts i and j in terms a_i or g_j are used to represent aliasing. If an abstract term, abstract atom or abstract conjunction of atoms contains a_i several times (with the same subscript), the denoted concrete terms, atoms or conjunctions of atoms contain the *same* term in all positions corresponding to those occupied by a_i. Similarly, if a domain element contains g_j several times, the denoted concrete values contain the *same* ground term in all positions occupied by g_j. Note that if a_i and g_j share an index, i.e. $i = j$, this has no particular meaning and the terms are not assumed to be aliased.

As an example of the aliasing concept, $perm(g_1, a_1), ord(a_1)$ denotes the set of concrete conjunctions $\{perm(t_1, t_2), ord(t_2) \mid t_1, t_2 \in Term_P \text{ and } t_1 \text{ is ground}\}$.

Definition 10 (Abstract domain):
The abstract domain, $ADom_P$, is the union of:

- $AVar_P = \{a_i \mid i \in \mathbb{N}_0\} \cup \{g_j \mid j \in \mathbb{N}_0\}$

- $ATerm_P$, $AAtom_P$ and $AConAtom_P$, defined as the sets of the terms, atoms and conjunctions of atoms constructable from $AVar_P$, $ACon_P$, Fun_P and $Pred_P$

Next, we define the semantics of the abstract domain, through a concretization function γ. The definition of γ will be based on functions that map abstract variables to concrete terms, called abstract-concrete substitutions.

Definition 11 (Abstract-concrete substitution):
An abstract-concrete substitution σ is a function from Dom_σ, a subset of $AVar_P$, into $Term_P$.

Note that, in definition 11, we do not require that g_j symbols are mapped to ground terms. Of course, we will impose this extra requirement in the concretization function. However, we also need the more general notion of definition 11, further on, for definition 14.

Example 3 (Abstract-concrete substitution):
As an example, $\sigma = \{a_1/f(x), a_2/h(y, A), g_2/h(f(A), A)\}$.

For any $t \in ADom_P$, by $AVar(t)$ we denote the set of all abstract variables a_i and g_j occurring in t.

Definition 12 (Application of an abstract-concrete substitution):
Let $t \in ADom_P$ and σ an abstract-concrete substitution, such that $AVar(t) \subseteq Dom_\sigma$. The application of σ on t, denoted $t\sigma$, is an element of Dom_P obtained by replacing all occurrences of abstract variables a_i and g_j in t by their corresponding concrete term in σ.

Example 4 (Application of an abstract-concrete substitution):
Let $t = perm(g_1, a_1), ord(a_1)$ and $\sigma = \{a_1/X, g_1/[s(0), s(s(0))]\}$. Then $t\sigma = perm([s(0), s(s(0))], X), ord(X)$.

Note that, due to the condition $AVar(t) \subseteq Dom_\sigma$, $t\sigma$ is always concrete.

Definition 13 (Concretization function):
The concretization function $\gamma : ADom_P \to 2^{Dom_P}$ is defined as, for $t \in ADom_P$: $\gamma(t) = \{t\sigma \mid \sigma = \{a_{i_1}/t_1, \ldots, a_{i_n}/t_n\} \cup \{g_{j_1}/s_1, \ldots, g_{j_m}/s_m\}$ with σ an abstract-concrete substitution such that $AVar(t) \subseteq Dom_\sigma$ and for all $l = 1, \ldots, m : s_l$ is ground $\}$.

Example 5 (Concretization function):
$\gamma(p(f(a_2, g_1), a_1, a_2, q(h(a_1)))) = \{p(f(t_1, t_2), t_3, t_1, q(h(t_3))) \mid t_1, t_3 \in Term_P,$ t_2 is a ground term of $Term_P\}$.

The abstract domain introduced above is infinitely large. There are two causes for this. Terms can be nested to any depth, therefore infinitely many different terms exist. In addition, there are infinitely many $a_i, i \in \mathbb{N}_0$ and $g_j, j \in \mathbb{N}_0$, symbols.

If so desired, the abstract domain can be refined, so that it becomes finite. This can be done by using depth-k abstraction and by defining an equivalence relation on $\{a_i \mid i \in \mathbb{N}_0\}$ and on $\{g_j \mid j \in \mathbb{N}_0\}$. In section 3.2 we define this equivalence relation and in chapter 4 we discuss depth-k abstraction. However, for the purpose of this example, the infinite size of the abstract domain is not a problem.

Again, we use the source code for permutation sort on page 10. Assume that instantiation-based delay declarations are used to interleave calls. We will formalize these declarations in section 3.2.4.

We now show how abstract conjunctive partial deduction can be used to compile the control. Abstract conjunctive partial deduction requires a top-level abstract atom (or conjunction) to start the transformation. Let $permsort(g_1, a_1)$ be this atom.

Central to abstract conjunctive partial deduction is the $\mathcal{A}$-coveredness condition, which is a generalization of the closedness condition in definition 4. Stated somewhat informally, $\mathcal{A}$-coveredness means that, given a finite set of finite abstract derivation trees, obtained for queries from a set $\mathcal{A}$ of abstract conjunctions, the conjunctions in the leaves of all these trees can be split into new abstract conjunctions, $\wedge_{i=1,\ldots,n} C_i$, such that, for every C_i, there exists an $A_i \in \mathcal{A}$ with $\gamma(C_i) \subseteq \gamma(A_i)$. For our purposes, splitting is not desirable, however. The reason for this is that we aim to analyze the *interaction* between coroutining predicates. Coroutines in logic programs share information through shared variables and, as in example 1 on page 17, splitting a conjunction means that information regarding shared variables is lost. Therefore, we assume a stricter version of the condition, which is closer to the $\mathcal{A}$-closedness condition of partial deduction. To avoid any confusion, we will refer to this as AC-$\mathcal{A}$-closedness. This condition states that the conjunctions in the leaves of the abstract derivation trees are *variants* of elements of $\mathcal{A}$. That is, they are representatives of an equivalence class in $\mathcal{A}$. The reason why they must be variants and not mere instances will be discussed along with the concept of depth-k abstraction in chapter 4.

Here, our initial set $\mathcal{A}$ is $\{permsort(g_1, a_1)\}$. Below, we construct a finite number of finite, abstract partial deduction derivation trees for abstract (conjunctions of) atoms starting from this initial abstract conjunction. The construction of these trees assumes an "abstract unification" and an "abstract resolve" operation. To simplify the description of cycle detection in the analysis, we assume the trees are built using a depth-first strategy. Their formal definitions can be found in section 3.2. For now, we only show their effects in abstract partial derivation trees.

Partial deduction poses two problems known as the "local control" and "global control" problem. These are sometimes assumed to be handled by an "oracle". This means that there is a solution to these problems, but it can be application-dependent and is assumed to be provided. This oracle typically has two functions:

- to decide which atom of the current goal should be selected for resolving (local control)
- to decide whether an obtained goal should be unfolded further, or whether it should be kept residual (to be added to $\mathcal{A}$) (global control)

Compiling control also assumes that these questions can be answered and it places a third responsibility upon the oracle: to decide whether an atom should be "fully evaluated", i.e. whether it is not involved in the coroutining computation.

Our specific requirements allow us to specify how these two functions are performed. With regard to the first function, a goal G other than the top-level goal is added to $\mathcal{A}$ if both of the following conditions hold:

- it contains no atoms that can be fully evaluated
- if further unfolding (of G or one of its ancestors) produces a variant of G

The second function is based on an "abstract instantiation-based selection rule". Such a rule is a partial order on abstract atoms, expressing which abstract atoms take precedence over other abstract atoms. We formally define this in section 3.2.4.

The third type of decision is not commonly supported in partial deduction. What it means is that we decide not to transform a certain predicate of the original program, but merely keep its original definition in the transformed program. In partial deduction, this can be done by never selecting these atoms,

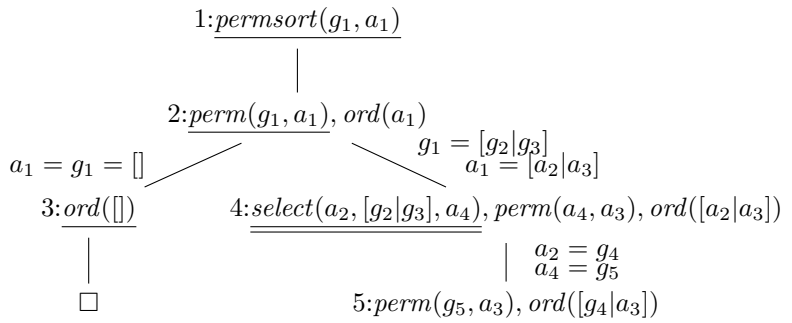


Figure 3.1: Abstract tree for $permsort(g_1, a_1)$

including them in $\mathcal{A}$ and including their original definition in the transformed program.

In our setting, however, we want to know the effect that solving the atom has on the remainder of the goal. Therefore, we will assume that a full abstract interpretation over our abstract domain computes the abstract bindings that solving the atom results in, if any. If solving the atom is possible, these bindings are applied to the remainder of the goal. If the abstract atom cannot be solved, the branch on which it occurs is terminated. Note that full evaluation cannot easily be performed in standard partial deduction, as it may not terminate. In [79], a similar functionality is integrated in a hybrid approach to conjunctive partial deduction and in [7] the same idea is used for the abstract evaluation of builtins. We will fully evaluate all predicates whose evaluation is not interleaved with that of other predicates, regardless of whether they are builtins.

Figures 3.1 and 3.2 show the abstract partial derivation trees that our approach based on abstract conjunctive partial deduction builds for permutation sort and top level $\mathcal{A} = \{permsort(g_1, a_1)\}$.

In these figures, in each goal, the atom selected for abstract resolution is underlined. If an atom is underlined twice, this expresses that the atom was selected for full evaluation.

Both resolving and full evaluation may create abstract bindings. Our abstract unification collects abstract bindings made on the a_i and g_j variables.[1]

[1]We assume that the relevant bindings created by fully evaluating an atom are known. These bindings are typically documented. They could also be obtained using existing systems for abstract interpretation. Furthermore, if the programmer of the coroutining program cannot provide them, he or she has an incomplete mental model of the control flow. Therefore this assumption does not impose any real restrictions.

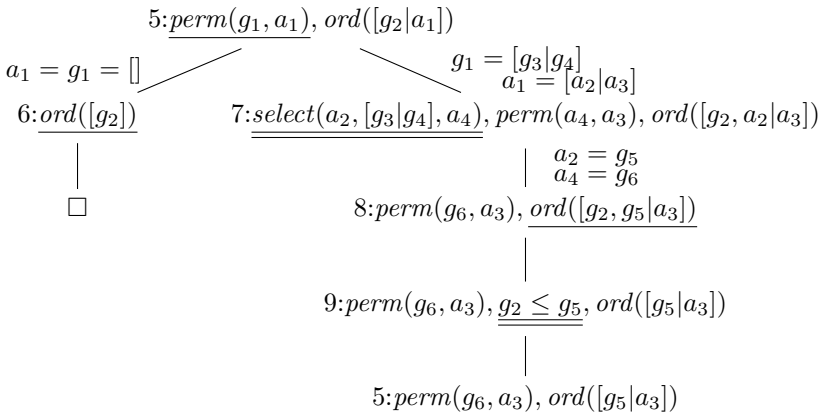


Figure 3.2: Abstract tree for $perm(g_1, a_1), ord([g_2|a_1])$

A goal with no underlined atom indicates that the oracle selects no atom and decides to keep the conjunction residual. After the construction of the tree in figure 3.1, our approach adds the abstract conjunction $perm(g_5, a_3), ord([g_4|a_3])$ to $\mathcal{A}$, as part of its aim to eventually reach an AC-$\mathcal{A}$-closed set $\mathcal{A}$. The approach starts a new tree for this atom. A renamed instance of this tree is shown in figure 3.2. In this case, it is easy to see that the tree does not need to be elaborated further, as its leaf nodes are the empty goal and a renaming of the root, which is in $\mathcal{A}$.

Returning to the example, AC-$\mathcal{A}$-closedness now holds for $\mathcal{A} = \{permsort(g_1, a_1), \wedge(perm(g_1, a_1), ord([g_2|a_1]))\}$. In standard (concrete) conjunctive partial deduction, the analysis phase would now be completed. In this context, however, we need an additional step before code generation can be performed. In the abstract derivation trees, we have not collected the concrete bindings that unfolding would produce. These are required to generate the resolvents. Therefore, we need to essentially construct the same two trees again, but now using concrete terms and concrete unification.

These trees are shown in figures 3.3 and 3.4, respectively. We define the root of a concrete derivation tree corresponding to an abstract tree as follows.

Definition 14 (Concrete conjunction in the root):
Let $acon \in \mathcal{A}$. Let $\sigma = \{a_{i_1}/t_1, \ldots, a_{i_n}/t_n\} \cup \{g_{j_1}/s_1, \ldots, g_{j_m}/s_m\}$ be an abstract-concrete substitution with $AVar(acon) \subseteq Dom_\sigma$, such that t_k and s_l, for $k = 1, \ldots, n$ and $l = 1, \ldots, n$, are distinct variables of Var_P. The conjunction in the root of the corresponding concrete tree for $acon$, denoted $c(acon)$, is $acon\ \sigma$.

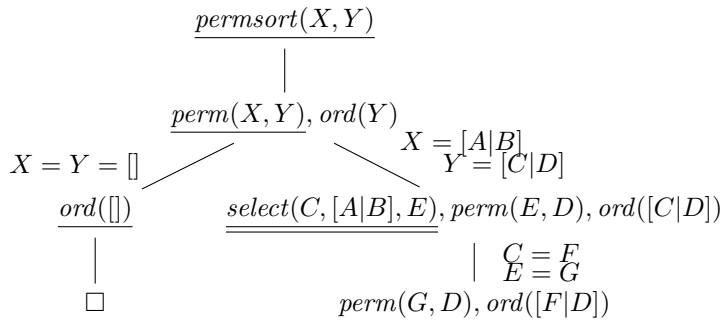


Figure 3.3: Concrete tree corresponding to figure 3.1

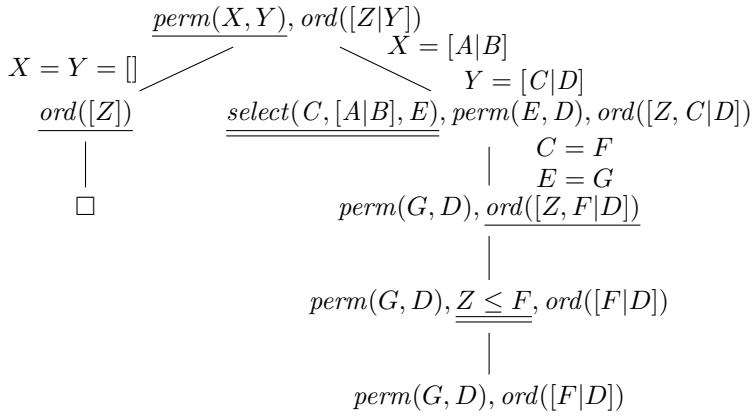


Figure 3.4: Concrete tree corresponding to figure 3.2

Note that $c(acon)$ is uniquely defined, up to variable renaming. When unfolding the concrete tree, every abstract resolution of the abstract tree is mimicked, using the same clauses, over the concrete domain. For some of these unfoldings, the concrete unification may fail: the abstract resolution is only a safe approximation of the concrete one. As a result, some branches of the abstract tree may be aborted in the concrete one.

The step of full abstract interpretation of the $\leq /2$ and $select/3$ abstract atoms means that these atoms are residual and that their definitions are made available to the transformed program. Since these particular predicates are builtins, no action is required. If they were user-defined predicates, their definitions would be included in the transformed program.

As a next step, we derive resultants from the concrete counterparts to the abstract trees. We obtain the following resultants from the tree in figure 3.3:

$$permsort([], []) \leftarrow true$$

$$permsort([A|B], [F|D]) \leftarrow select(F, [A|B], G), perm(G, D), ord([F|D])$$

From the concrete counterpart of the tree in figure 3.2, we get:

$$perm([], []), ord([Z]) \leftarrow true$$

$$perm([A|B], [F|D]), ord([Z, F|D]) \leftarrow$$

$$select(F, [A|B], G), Z \leq F, perm(G, D), ord([F|D])$$

Finally, we rename and filter[2] $perm(G, D), ord([F|D])$ to $p1(G, D, F)$. This translates directly to the Prolog program in listing 3.1.

```
permsort([],[]).
permsort([A|B],[F|D]) :- select(F,[A|B],G), p1(G,D,F).
p1([],[],_Z).
p1([A|B],[F|D],Z) :- select(F,[A|B],G), Z =< F, p1(G,D,F).
```

Listing 3.1: Synthesized permutation sort

This program achieves our initial aim. When executed under the standard left-to-right selection rule with a top-level call to *permsort*/2 such that the first argument is ground and the second argument is free, it interleaves the stepwise generation of a permutation with the check for ordering. As a result, the generation of a permutation is aborted as soon as it contains two elements which are out of order.

3.2 A More Formal Account

Section 3.1 only provides the intuitions of the approach, based on an example. In the current section, we formally present the key concepts. This includes the central operations Abstract unify and Abstract resolve.[3] Abstract resolve requires the ability to generate abstract versions of the program clauses. We

[2] In this context, filtering means removing an occurrence of a variable which occurs twice at corresponding positions in all calls [34].

[3] We use the upper case "A" to avoid confusion with Leuschel's functions *aunfold* and *aresolve*, which in our case rely on these operations.

therefore also introduce an order on the abstract domain and an abstraction function, α, which allows us to compute these abstract clauses.

Then, we discuss correctness of the transformed programs. We show that the local correctness requirements of abstract conjunctive partial deduction are fulfilled, so that the main correctness theorem of [45] applies. To conclude this section, we discuss a formal representation of control and the effects of widening.

3.2.1 Abstract Unification

We adapt the standard unification algorithm from [57] to $ADom_P$. In algorithm 1, both the symbols a_i and g_j take the role of the variables in the Martelli-Montanari algorithm. For the a_i symbols, this seems obvious, as a_i represents any concrete term (including a variable). For the g_j symbols, as g_j represents any ground term, the abstract unification algorithm needs to be opportunistic in their treatment in the unification. The abstract unification $g_j = t$ opportunistically always succeeds, assuming that g_j represents a ground instance of t. As stated above, concrete unifications mimicking this unification may fail.

As a result of the use of abstract variables, the majority of the rewrite steps in algorithm 1 are the direct counterparts of the rewrites in the Martelli-Montanari algorithm, reformulated both for the a_i's and the g_j's.

In addition, we need to deal with the fact that our variables are colored: with color either "a" or "g". The "g" color is dominant. Any "a"-colored variable becomes "g"-colored, after unification with a "g"-colored variable. This requires two additional rewrite steps with respect to the Martelli-Montanari algorithm.

Algorithm 1 (Abstract unification):
Let $s1, s2 \in ATerm_P \cup AAtom_P$. Let $mgu = \{s1 = s2\}$. Repeatedly select an equality e and perform any of the following transformations on mgu. If no transformation applies, stop with success.

Step 1: remove any equality of the form $t = t$
Step 2: select any equality $t = s$, with s of the form a_i or g_j, with t not of the form a_k or g_k (where k represents a natural number which may or may not be equal to that represented by i or j) and replace it by $s = t$
Step 3: select any equality $s = t$, with s of the form a_i or g_j, where t contains s as a subterm and stop with failure
Step 4: select any equality $g_j = a_i$, replace it by $a_i = g_j$ and replace all other occurrences of a_i in mgu by g_j
Step 5: select any equality $g_j = t$, where t contains $a_{i_1}, \ldots, a_{i_n}$, with $n > 0$, and where t is not an a_j and do the following:

- introduce $g_{k_1}, \ldots, g_{k_n}$, with $k_1, \ldots, k_n$ fresh indices from $\mathbb{N}_0$
- replace all occurrences of a_{i_j} in *mgu* by g_{k_j}, for all $j = 1, \ldots, n$
- add equalities $a_{i_j} = g_{k_j}$, for all $j = 1, \ldots, n$, to *mgu*

Step 6: select any equality $s = t$, with s of the form a_i or g_j, where s occurs elsewhere in *mgu* and replace all other occurrences of s in *mgu* by t
Step 7: select any equality of the type $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$ and stop with failure if $f \neq g$ or $n \neq m$; replace $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$ in *mgu* by $t_1 = s_1, \ldots, t_n = s_n$ otherwise

In algorithm 1, five out of seven transformations for *mgu* are the direct counterparts of the rewrites in the Martelli-Montanari algorithm. They only differ from the concrete versions in that a_i and g_j symbols take on the role of the variables. The two additional rewrites, steps 4 and 5, deal with the "g"-coloring of a_i symbols. Both these rewrites make sure that, if an a_i symbol is unified with a g_j symbol, it is replaced by a (possibly fresh) g_k symbol. Due to the similarity with the Martelli-Montanari algorithm, the proofs for termination and correctness are easily adapted from [57].

Proposition 1 (Termination of Abstract unify):
Algorithm 1 terminates.

Proof. We introduce the following numbers:

- n_a is the number of a_i symbols in *mgu* that occur more than once **or** do not occur as the left-hand side of an equality
- n_g is the number of g_j symbols in *mgu* that occur more than once **or** do not occur as the left-hand side of an equality
- n_{fun} is the total number of occurrences of function symbols in *mgu*
- n_{equ} is the number of equalities in *mgu* of the type $s = s$ or $t = a_i$, with t not of the form a_j or g_k, or $t = g_j$, with t not of the form a_i or g_k

It suffices to prove that the lexicographic order on 4-tuples $(n_a, n_g, n_{fun}, n_{equ})$ strictly decreases with every application of a rewrite on *mgu*. As the lexicographic order is well-founded, there can only be a finite number of consecutive strict decreases.

We will not discuss our counterparts of the five rewrites in the concrete unification algorithm. The arguments for these rewrites are completely similar to those in [57].

For step 4, n_a decreases by one. Therefore $(n_a, n_g, n_{fun}, n_{equ})$ decreases. For step 5, the term t contains at least one a_i. Therefore, again, n_a decreases by at least one and the lexicographic order decreases.

Note that these two steps are the reason why we need two separate variables n_a and n_g in the order, instead of just one for the union of all a_i and g_j (as it is the case in [57]). For step 5, n_g increases with the number of fresh variables that are introduced. Also step 4 may increase n_g. □

To show the correctness of algorithm 1, we need the ability to "apply" the outcome of Abstract unify to abstract atoms and terms. We need the following definitions and properties.

Definition 15 (Solved form):
A set AE of equalities of elements of $ATerm_P$ is in solved form if it is of the form $\{a_{i_1} = t_1, \ldots, a_{i_n} = t_n, g_{j_1} = s_1, \ldots, g_{j_m} = s_m\}$, where:

- all $i_k, k = 1, \ldots, n$ are distinct and all $j_l, l = 1, \ldots, m$, are distinct
- no a_{i_k} occurs in $t_1, \ldots, t_n$, nor in $s_1, \ldots, s_m$
- no g_{j_l} occurs in $t_1, \ldots, t_n$, nor in $s_1, \ldots, s_m$
- no a_i variable occurs in $s_1, \ldots, s_m$

Due to the conditions above, we can associate an abstract substitution with any set of equalities AE in solved form.

Definition 16 (Abstract substitution):
An abstract substitution Θ_A is a finite set of ordered pairs in $AVar_P \times ATerm_P$, such that there exists a set AE of equalities of elements of $ATerm_P$ in solved form, with $\Theta_A = \{x/y \mid x = y \in AE\}$.

Using, again, the conditions in definition 15, it is clear that abstract substitutions can be applied to elements of $ADom_P$.

Definition 17 (Application of an abstract substitution):
Let $s \in ADom_P$ and let Θ_A be an abstract substitution. Θ_A can be applied to s, resulting in $s\Theta_A \in ADom_P$, obtained by replacing all a_i and g_j in s occurring on the left-hand side of a pair in Θ_A, by the corresponding right-hand side.

Example 6 (Application of an abstract substitution):
Let $s = perm(g_1, a_1)$ and $\Theta_A = \{a_1/[a_2|a_3], g_1/[g_2|g_3]\}$. Then $s\Theta_A = perm([g_2|g_3], [a_2|a_3])$.

Note that we can also apply an abstract substitution Θ_A to a set of equalities AE between elements of $ATerm_P$. This is done by applying the abstract substitution to every term in AE. The result is denoted $AE\Theta_A$.

We can now define an abstract unifier.

Definition 18 (Abstract unifier):
Let $s_1, s_2 \in ADom_P$. An abstract substitution, Θ_A, is an abstract unifier for (s_1, s_2), if $s_1\Theta_A = s_2\Theta_A$.

Let $AE = \{s_{i_1} = s_{i_2} \mid i = 1, \ldots, n\}$ be a set of equalities between elements of $ATerm_P$. An abstract substitution Θ_A is an abstract unifier for AE if Θ_A is an abstract unifier for every pair (s_{i_1}, s_{i_2}).

Observe that an abstract substitution Θ_A is an abstract unifier for AE if and only if $AE\Theta_A$ is a set of syntactic identity equalities.

In order to prove that the final set *mgu* computed by algorithm 1 corresponds to an abstract unifier for the initial abstract terms or atoms s_1 and s_2, we still need the following proposition and definitions.

Proposition 2 (Solved form in algorithm 1):
If algorithm 1 terminates with success, then the final set *mgu* is in solved form.

Proof. It is easy to check that, in case any of the conditions of definition 15 is not met, one of the rewrites of algorithm 1 is still applicable. This contradicts successful termination of the algorithm. □

As a result of proposition 2, we can associate with the final set *mgu* produced by algorithm 1, in the case that the algorithm terminates successfully, an abstract substitution which we will denote as Θ_{mgu}. If we want to be explicit concerning the $s_1, s_2 \in ADom_P$ on which we applied the algorithm, we denote it as $\Theta_{mgu}(s_1, s_2)$.

Proposition 3 (Correctness of Abstract unify (1)):
Let $s_1, s_2 \in ATerm_P \cup AAtom_P$ and let $\Theta_{mgu}(s_1, s_2)$ be the abstract substitution associated with the final set *mgu* obtained by successfully terminating algorithm 1 on (s_1, s_2). Then, Θ_{mgu} is an abstract unifier for (s_1, s_2).

Proof. The proof is similar to the one in [57]. First note that for any set of equalities in solved form AE, the associated abstract substitution Θ_A is an abstract unifier for AE. This is easily verified: the application of Θ_A on the equalities of AE reduces them to syntactic identity equalities. Thus, Θ_{mgu} is an abstract unifier for the last set *mgu* in algorithm 1, by proposition 2.

It now suffices to prove that every abstract unifier of the *mgu* set obtained after any rewrite step of algorithm 1, is also an abstract unifier of the *mgu* set before that rewrite step.

We do not discuss the argument for the five rewrite rules which are direct counterparts of the rules in [57]. The argument is identical to that in [57].

Consider step 4 in algorithm 1. Let $AE \cup \{g_j = a_i\}$ be the set *mgu* before step 4. Let $\Theta = \{a_i/g_j\}$. Then the next set *mgu* is $AE\Theta \cup \{a_i = g_j\}$.

Let Θ_A be any abstract unifier for $AE\Theta \cup \{a_i = g_j\}$. Then, $AE\Theta\Theta_A$ consists of syntactic identity equalities and $a_i\Theta_A = g_j\Theta_A$ is a syntactic identity equality. Now consider $AE\Theta_A$. AE only differs from $AE\Theta$ in that some occurrences of g_j of $AE\Theta$ are replaced by occurrences of a_i in AE. But, as $a_i\Theta_A$ and $g_j\Theta_A$ are syntactically equal, $AE\Theta_A = AE\Theta\Theta_A$ and therefore it only consists of syntactic identity equalities. Thus, Θ_A is a unifier for $AE \cup \{g_j = a_i\}$.

Next, consider step 5 in algorithm 1. Let $AE \cup \{g_j = t\}$, where t contains $a_{i_1}, \ldots, a_{i_n}, n > 0$, be the set *mgu* before step 5. Let $\Theta = \{a_{i_1}/g_{k_1}, \ldots, a_{i_n}/g_{k_n}\}$, with $k_1, \ldots, k_n$ fresh indices from $\mathbb{N}_0$. Step 5 transforms *mgu* into $AE' = AE\Theta \cup \{g_j = t\}\Theta \cup \{a_{i_1} = g_{k_1}, \ldots, a_{i_n} = g_{k_n}\}$.

Let Θ_A be an abstract unifier for AE'. Then, $AE\Theta\Theta_A$ consists of syntactic identity equalities and $g_j\Theta\Theta_A = t\Theta\Theta_A$, $a_{i_1}\Theta_A = g_{k_1}\Theta_A, \ldots, a_{i_n}\Theta_A = g_{k_n}\Theta_A$ are syntactic identity equalities. $AE \cup \{g_j = t\}$ only differs from $AE\Theta \cup \{g_j = t\}\Theta$ in that all occurrences of $g_{k_1}, \ldots, g_{k_n}$ in $AE\Theta \cup \{g_j = t\}\Theta$ are replaced by corresponding $a_{i_1}, \ldots, a_{i_n}$ in $AE \cup \{g_j = t\}$. Thus, as $a_{i_1}\Theta_A = g_{k_1}\Theta_A, \ldots, a_{i_n}\Theta_A = g_{k_n}\Theta_A$ are identities, $AE\Theta_A \cup \{g_j = t\}\Theta_A = AE\Theta\Theta_A \cup \{g_j = t\}\Theta\Theta_A$ and therefore it consists only of syntactic identity equalities. Thus, Θ_A is an abstract unifier for $AE \cup \{g_j = t\}$. □

Note that there is a second correctness concern related to Abstract unify. Namely, the fact that the abstract unification correctly mimics the concrete unification. In order to prove this second correctness result, we need the following concept and lemma.

Definition 19 (a-g-extension):
Let σ be an abstract-concrete substitution. An abstract-concrete substitution σ' is an a-g-extension of σ, denoted $\sigma' \succcurlyeq \sigma$, if there exists a set $\{a_{i_1}, \ldots, a_{i_n}\} \subseteq Dom_\sigma$ and a set $\{g_{k_1}, \ldots, g_{k_n}\}$ of abstract variables not occurring in Dom_σ, such that:

1. $Dom_{\sigma'} = Dom_\sigma \cup \{g_{k_1}, \ldots, g_{k_n}\}$
2. $\sigma'(t) = \sigma(t)$, for $t \in Dom_\sigma$

3. $\sigma'(g_{k_j}) = \sigma(a_{i_j})$, for $j = 1, \ldots, n$

Example 7 (a-g-extension):
Let $\sigma = \{g_1/f(X,Y), a_1/X, a_2/Y\}$. Then, $\sigma' = \{g_1/f(X,Y), a_1/X, a_2/Y, g_2/X, g_3/Y\}$ is an a-g-extension of σ.

Lemma 1 (Transitivity of $\succcurlyeq$):
The relation $\succcurlyeq$ is transitive.

Proof. Let $\sigma'' \succcurlyeq \sigma'$ and let $\sigma' \succcurlyeq \sigma$. In order to prove transitivity of $\succcurlyeq$, we need to establish that conditions 1, 2 and 3 are maintained between σ'' and σ. For condition 1, we have $Dom_{\sigma''} = Dom_{\sigma'} \cup \{g_{l_1}, \ldots, g_{l_m}\}$ by definition. As $Dom_{\sigma'} = Dom_{\sigma} \cup \{g_{k_1}, \ldots, g_{k_n}\}$, condition 1 is indeed maintained.

For condition 2, $\sigma''(t) = \sigma'(t)$, for $t \in Dom_{\sigma'}$ by definition. As $\sigma'(t) = \sigma(t)$, for $t \in Dom_{\sigma}$, condition 2 is similarly maintained.

For condition 3, first consider the variables $\{g_{l_1}, \ldots, g_{l_m}\}$ introduced in the a-g-extension $\sigma'' o f \sigma'$. We have $\sigma''(g_{l_j}) = \sigma'(a_{i_j})$, for $j = 1, \ldots, m$, for some $a_{i_j} \in Dom_{\sigma'}$. By condition 1, if $a_{i_j} \in Dom_{\sigma'}$, then $a_{i_j} \in Dom_{\sigma}$. By condition 2, $\sigma'(a_{i_j}) = \sigma(a_{i_j})$. Thus, $\sigma''(g_{l_j}) = \sigma(a_{i_j})$. For the variables $\{g_{k_1}, \ldots, g_{k_n}\}$ introduced in the a-g-extension σ' of σ, $\sigma''(g_{k_j}) = \sigma'(g_{k_j})$, because condition 2 holds for σ'' and σ'. The result then follows from condition 3 on $\sigma' \succcurlyeq \sigma$. □

Example 8 (Motivation for a-g-extensions):
Let $s_1 = g_1$ and let $s_2 = f(a_1, a_2)$. Then, an abstract-concrete substitution σ which makes $(s_1 = s_2)$ a syntactic equality is $\sigma = \{g_1/f(X,Y), a_1/X, a_2/Y\}$. A set mgu for s_1 and s_2 is $mgu = \{g_1 = f(g_2, g_3), a_1 = g_2, a_2 = g_3\}$. Applying σ to mgu does not provide us with a set of syntactic equalities.

Let $\sigma' = \{g_1/f(X,Y), a_1/X, a_2/Y, g_2/X, g_3/Y\}$. Applying σ' to mgu does provide a set of syntactic equalities. Moreover, σ' is an a-g-extension of σ.

For the following lemma, we are assuming that algorithm 1 successfully terminates, given a starting equality $s_1 = s_2, s_1, s_2 \in ATerm_P \cup AAtom_P$. With $mgu(s_1, s_2)$, we denote the final mgu set. We also denote by $AVar(s_1, s_2)$ the abstract variables occurring in s_1 or s_2.

Lemma 2:
Let σ be an abstract-concrete substitution, with $Dom_{\sigma} = AVar(s_1, s_2)$, such that $\sigma(s_1 = s_2)$ is a syntactic identity equality and such that, for any $g_j \in Dom_{\sigma}$, $\sigma(g_j)$ is a ground term. Then, there exists an a-g-extension σ' of σ, such that $\sigma'(mgu(s_1, s_2))$ is a set of syntactic identity equalities and such that, for any $g_j \in Dom_{\sigma'}$, $\sigma'(g_j)$ is a ground term.

Proof. Consider any rewrite step in algorithm 1. We denote the set mgu before the step as mgu_1 and the set mgu after the step as mgu_2. We will prove that, if σ is an abstract-concrete substitution such that $\sigma(mgu_1)$ is a set of syntactic identity equalities and such that, for any $g_j \in Dom_\sigma$, $\sigma(g_j)$ is a ground term, then there exists an a-g-extension σ' of σ, such that $\sigma'(mgu_2)$ is a set of syntactic identity equalities and such that, for any $g_j \in Dom_{\sigma'}$, $\sigma'(g_j)$ is a ground term.

As the relation $\succcurlyeq$ is transitive by lemma 1, lemma 2 will follow directly from this proof.

So, consider step 1 in algorithm 1. Here, $mgu_2 = mgu_1 \backslash \{t = t\}$. Clearly, with $\sigma' = \sigma$, if $\sigma(mgu_1)$ is a set of syntactic identity equalities, then so is $\sigma'(mgu_2)$.

For step 2, mgu_1 contains either $t = a_i$ or $t = g_j$, while in mgu_2, this equality is replaced by $a_i = t$, respectively $g_j = t$. Obviously, with $\sigma' = \sigma$, the result holds.

Step 3 does not need to be considered, as we are assuming that algorithm 1 successfully terminates.

For step 4, mgu_1 contains $g_j = a_i$ and mgu_2 is obtained from mgu_1 by replacing $g_j = a_i$ by $a_i = g_j$ and by replacing all other occurrences of a_i in mgu_1 by g_j. Again, let $\sigma' = \sigma$. As $\sigma(g_j = a_i)$ is a syntactic identity equality, so is $\sigma'(a_i = g_j)$. Moreover, all other equalities in mgu_2 only differ from those in mgu_1 by replacements of a_i by g_j. As $\sigma(a_i) = \sigma'(g_j)$, $\sigma(mgu_1)$ being syntactic identity equalities implies the same for $\sigma'(mgu_2)$.

Only step 5 requires the notion of an a-g-extension. Here, mgu_1 contains an equality $g_j = t$, where t contains $a_{i_1}, \ldots, a_{i_n}$, with $n > 0$. The step introduces new variables $g_{k_1}, \ldots, g_{k_n}$ and mgu_2 is obtained from mgu_1 by replacing all occurrences of a_{i_j} by $g_{k_j}, j = 1, \ldots, n$, and adding the equalities $a_{i_j} = g_{k_j}, j = 1, \ldots, n$.

Let σ' be the a-g-extension of σ based on the sets $\{a_{i_1}, \ldots, a_{i_n}\}$ and $\{g_{k_1}, \ldots, g_{k_n}\}$. By definition 19, $\sigma'(g_{k_j}) = \sigma(a_{i_j})$ and $\sigma'(a_{i_j}) = \sigma(a_{i_j})$, for $j = 1, \ldots, n$. Thus, $\sigma'(a_{i_j} = g_{k_j})$ are syntactic identity equalities, for $j = 1, \ldots, n$. As all other equalities in mgu_2 are obtained by replacing a_{i_j}'s by g_{k_j}'s in the equalities in mgu_1, $\sigma(mgu_1)$ are syntactic identity equalities and $\sigma'(g_{k_j})$'s are identical to $\sigma(a_{i_j})$'s, $\sigma'(mgu_2)$ is also a set of syntactic identity equalities.

For this step, we have the additional proof obligation on the groundness of σ' on newly introduced variables g_{k_j}. It is given that mgu_1 contains the equality $g_j = t$ and that $\sigma(g_j)$ is ground. Thus, $\sigma(t)$ is ground. As $a_{i_1}, \ldots, a_{i_n}$ are subterms of t, $\sigma(a_{i_1}), \ldots, \sigma(a_{i_n})$ are ground. Therefore, by $\sigma'(g_{k_j}) = \sigma(a_{i_j})$,

$j = 1, \ldots, n$, $\sigma'(g_{k_j})$ is ground.

For step 6, mgu_1 contains an equality $a_i = t$ or $g_j = t$. Here, mgu_2 is obtained from mgu_1 by replacing all other occurrences of a_i, respectively g_j, by t. Let $\sigma' = \sigma$. As $\sigma(a_i = t)$, respectively $\sigma(g_j = t)$, is a syntactic identity equality, $\sigma'(mgu_2) = \sigma(mgu_1)$, so the result holds.

Finally, for step 7, mgu_1 contains an equality $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$. As we assume that algorithm 1 terminates successfully, $f = g$, $n = m$ and mgu_2 is obtained from mgu_1 by replacing the equality above by $t_1 = s_1, \ldots, t_n = s_n$.

Let $\sigma' = \sigma$. As $\sigma(f(t_1, \ldots, t_n) = g(s_1, \ldots, s_n))$ is a syntactic identity equality, so are $\sigma'(t_1 = s_1), \ldots, \sigma'(t_n = s_n)$. All other equalities of $\sigma'(mgu_2)$ are in $\sigma(mgu_1)$, so the result holds. □

Proposition 4 (Correctness of Abstract unify (2)):
Let $s_1, s_2 \in ATerm_P \cup AAtom_P$, such that $AVar(s_1) \cap AVar(s_2) = \emptyset$ and let $\Theta_{mgu}(s_1, s_2)$ be the abstract unifier associated with the final set mgu obtained by successfully terminating algorithm 1 on (s_1, s_2). For any $cs_1 \in \gamma(s_1)$ and $cs_2 \in \gamma(s_2)$, if cs_1 and cs_2 unify with unifier δ_{mgu}, then $cs_1\delta_{mgu} = cs_2\delta_{mgu} \in \gamma(s_1\Theta_{mgu}(s_1, s_2))$.

Proof. As $cs_1 \in \gamma(s_1)$ and $cs_2 \in \gamma(s_2)$, there exist abstract-concrete substitutions σ_1 and σ_2, such that $cs_1 = \sigma_1(s_1)$, $cs_2 = \sigma_2(s_2)$ and such that for all $g_{j_k} \in AVar(s_1)$, $\sigma_1(g_{j_k})$ is ground and for all $g_{l_k} \in AVar(s_2)$, $\sigma_2(g_{j_k})$ is ground.

As δ_{mgu} exists, $cs_1\delta_{mgu} = cs_2\delta_{mgu} = \sigma_1(s_1)\delta_{mgu} = \sigma_2(s_2)\delta_{mgu}$. Considering δ_{mgu} as a function, we define ψ_1 to be the composition of σ_1 and δ_{mgu} and ψ_2 to be the composition of σ_2 and δ_{mgu}. Thus, we have:

$$cs_1\delta_{mgu} = cs_2\delta_{mgu} = \psi_1(s_1) = \psi_2(s_2) \tag{3.1}$$

As $AVar(s_1) \cap AVar(s_2) = \emptyset$, next we define an abstract-concrete substitution σ on $AVar(s_1) \cup AVar(s_2)$:

$\sigma(t) = \psi_1(t)$, for $t \in AVar(s_1)$

$\sigma(t) = \psi_2(t)$, for $t \in AVar(s_2)$

From Equation (3.1), we get:

$$cs_1\delta_{mgu} = cs_2\delta_{mgu} = \sigma(s_1) = \sigma(s_2) \tag{3.2}$$

As $\sigma(s_1) = \sigma(s_2)$, $\sigma(s_1 = s_2)$ is a syntactic identity equality. Moreover, as $\sigma_1(g_{j_k})$ is ground on all $g_{j_k} \in AVar(s_1)$ and $\sigma_2(g_{l_k})$ is ground on all $g_{l_k} \in$

$AVar(s_2)$, so is σ. Thus, by lemma 2 there exists an *a-g*-extension σ' of σ, such that $\sigma'(mgu(s_1, s_2))$ is a set of syntactic identity equalities and such that, for any $g_i \in Dom_{\sigma'}, \sigma'(g_i)$ is ground.

Now, consider $\sigma'(s_1\Theta_{mgu(s_1,s_2)})$. As $\sigma'(mgu(s_1, s_2))$ is a set of syntactic identity equalities, for every pair s/t in $mgu(s_1, s_2)$, $\sigma'(s)$ is identical to $\sigma'(t)$. As $\Theta_{mgu(s_1,s_2)}$ only replaces such s's by t's in s_1, $\sigma'(s_1\Theta_{mgu(s_1,s_2)}) = \sigma'(s_1)$.

By definition, an *a-g*-extension σ' of an abstract-concrete substitution σ satisfies $\sigma'|_{Dom_\sigma} = \sigma$. Thus, $\sigma'(s_1\Theta_{mgu(s_1,s_2)}) = \sigma(s_1)$.

Then, from Equation (3.2), we obtain $cs_1\delta_{mgu} = cs_2\delta_{mgu} = \sigma'(s_1\Theta_{mgu(s_1,s_2)})$.

As σ' is ground for any $g_i \in Dom_\sigma$, we can conclude that $cs_1\delta_{mgu} = cs_2\delta_{mgu} \in \gamma(s_1\Theta_{mgu(s_1,s_2)})$. □

Note that the condition that $AVar(s_1) \cap AVar(s_2) = \emptyset$ is necessary for the proposition to hold. This is shown in the following example.

Example 9 (Incorrect usage of Abstract unify):
Let $s_1 = f(a_1, g_1)$ and $s_2 = f(g_1, a_1)$. Algorithm 1 terminates successfully for $s_1 = s_2$ and $mgu(s_1, s_2) = \{a_1/g_1\}$. We have that $s_1\Theta_{mgu(s_1,s_2)} = s_2\Theta_{mgu(s_1,s_2)} = f(g_1, g_1)$.

Take $cs_1 = f(X, p)$ and $cs_2 = f(q, Y)$. Clearly, $cs_1 \in \gamma(s_1)$ and $cs_2 \in \gamma(s_2)$. Also, cs_1 and cs_2 unify, with $\delta_{mgu} = \{X/q, Y/p\}$ and resulting term $cs_1\delta_{mgu} = f(p, q)$. However, $f(p, q) \notin \gamma(f(g_1, g_1))$.

3.2.2 Abstract resolve

Next, we need a more formal account of operation used to resolve abstract conjunctions and program clauses, Abstract resolve. To do this, it is useful to have a definition of the abstraction function, so that we can map concrete clauses to their abstract counterparts. To define the abstraction function, in turn, it is useful to first introduce an order relation on $ADom_P$.

Definition 20 (Pre-ordering on $ADom_P$):
Let $t_1, t_2 \in ADom_P$.
$t_1 \succcurlyeq t_2$ if and only if: $\exists\Theta : t_1\Theta = t_2$, where Θ is an abstract substitution.

Example 10 (Pre-ordering on $ADom_P$):

- $a_4 \succcurlyeq f(a_2, g_1)$
- $g_1 \succcurlyeq f(g_2, g_3)$

- $f(h(a_1), g_1, a_1) \succcurlyeq f(h(a_3), h(g_2), a_3)$
- $f(h(a_1), g_1, a_1) \succcurlyeq f(h(k(a_2)), g_2, k(a_2))$

Proposition 5 (Pre-ordering in $ADom_P$):
$\succcurlyeq$ is a pre-ordering on $ADom_P$.

Proof.
Reflexivity of $\succcurlyeq$:
Obviously, for $t \in ADom_P$ and $\Theta_{Id} = \{a_i/a_i \mid a_i \in AVar(t)\} \cup \{g_j/g_j \mid g_j \in AVar(t)\}$, $t\Theta_{Id} = t$.

Transitivity of $\succcurlyeq$:
For two abstract substitutions, Θ_1 and Θ_2, their composition, $\Theta_2 \circ \Theta_1$, can be defined in the same way as composition of concrete substitutions. For each $t \in ADom_P$, it holds that $t(\Theta_2 \circ \Theta_1) = (t\Theta_1)\Theta_2$. Then, it is clear that, if $t_2 = t_1\Theta_1$ and $t_3 = t_2\Theta_2$, then $t_3 = (t_1\Theta_1)\Theta_2 = t_1(\Theta_2 \circ \Theta_1)$. □

Note that for any $t \in ADom_P$ and Θ an abstract substitution: $\gamma(t\Theta) \subseteq \gamma(t)$, because the abstract substitution only replaces abstract variables by potentially more instantiated abstract terms. Therefore, if $t_1 \succcurlyeq t_2$, then $\gamma(t_1) \supseteq \gamma(t_2)$.

Obviously, $\succcurlyeq$ is not anti-symmetric (e.g. both $a_1 \succcurlyeq a_2$ and $a_2 \succcurlyeq a_1$ hold) and therefore not a partial order. We use the standard approach for associating an equivalence relation and a partial order to the pre-order (see e.g. [2]).

Definition 21 (Equivalence and partial order on $ADom_P$):
Let $t_1, t_2 \in ADom_P$. t_1 and t_2 are equivalent, denoted $t_1 \approx t_2$, if $t_1 \succcurlyeq t_2$ and $t_2 \succcurlyeq t_1$.

The relation $\approx$ defines an equivalence relation on $ADom_P$. We denote the quotient set as $ADom_{P/\approx}$. For any $t \in ADom_P$, we denote by $\overline{t}$ its equivalence class in $ADom_{P/\approx}$. Let $\overline{t_1}, \overline{t_2} \in ADom_{P/\approx}$. $\overline{t_1} \leq \overline{t_2}$ if $t_1 \preccurlyeq t_2$, for some representatives t_1 of $\overline{t_1}$ and t_2 of $\overline{t_2}$. $\leq$ is well-defined and defines a partial order on $ADom_{P/\approx}$.

Note that $ADom_{P/\approx}$ merely formalizes the intuition that we have been using all along, that indices of a_i and g_j symbols in elements of $ADom_P$ are only relevant up to index renaming.

Next, we define how elements of the concrete domain are abstracted. We will not yet define the abstraction function in general, for any subset of Dom_P, but only on individual elements (singleton subsets) of Dom_P. We therefore refer to it as a pre-abstraction function.

Definition 22 (Pre-abstraction function):
A pre-abstraction function, $\alpha_1 : Dom_P \to ADom_P$, is a one-to-one function, mapping elements of Con_P to their counterparts in $ACon_P$ and mapping elements of Var_P to $\{a_i \mid i \in \mathbb{N}_0\}$ and such that $\forall t = f(t_1, \ldots, t_n) \in Dom_P : \alpha_1(t) = f(\alpha_1(t_1), \ldots, \alpha_1(t_n))$.

In what follows, by α_1, we denote a fixed pre-abstraction function. We also define $\overline{\alpha_1} : Dom_P \to ADom_{P/\approx}$ as $\overline{\alpha_1}(t) = \overline{(\alpha_1(t))}$.

$\overline{\alpha_1}$ may be seen as a "cleaner" formalization of the abstraction function, as it is independent of the selected α_1. Nevertheless, we will continue our formalization on the basis of representatives of equivalence classes, unless there is cause for confusion.

We now move towards the formalization of Abstract resolve. Every Horn clause, $h \leftarrow b_1, \ldots, b_m$, $m \in \mathbb{N}$, in P will be represented as $\texttt{:-}(h, b_1, \ldots, b_m)$.

To every Horn clause $C = \texttt{:-}(h, b_1, \ldots, b_m)$ in P, we can associate an abstract version of C: $AC = \alpha_1(\texttt{:-}(h_1, b_1, \ldots, b_n))$. Furthermore, we assume the availability of a renaming function on indices. More precisely, for each $t \in ADom_P$, there is a function $Rename(t) : ADom_P \to ADom_P$, such that for each $s \in ADom_P$, $Rename(t)(s)$ is an index renaming of all a_i and g_j occurring in s, such that all the abstract variables in $Rename(t)(s)$ are distinct from those in t.

Definition 23 (Abstract resolve):
Let $ConAt = \wedge(A_1, \ldots, A_i, \ldots, A_n) \in AConAtom_P$ and $AC = \texttt{:-}(H, B_1, \ldots, B_m)$ be an abstraction of Horn clause C of P, such that A_i and H have the same predicate symbol and arity. Let $Rename(ConAt)(\texttt{:-}(H, B_1, \ldots, B_m))$ be $\texttt{:-}(H', B'_1, \ldots, B'_m)$. If Abstract unify for (A_i, H') terminates successfully and if the final mgu corresponds to $\Theta_{mgu}(A_i, H')$, then the abstract resolvent of $ConAt$ and AC with selected literal A_i, $Aresolve(ConAt, AC, A_i)$, exists and is equal to $\wedge(A_1, \ldots, B'_1, \ldots, B'_m, \ldots, A_n)\Theta_{mgu}(A_i, H')$

Example 11 (Application of Abstract resolve):
We return to the construction of the second derivation tree for permutation sort in figure 3.2. Let us focus on the first Abstract resolve step, on the right-hand side of the tree. $ConAt = \wedge(perm(g_1, a_1), ord([g_2|a_1]))$. The applied Horn clause is $perm([X|Y], [U|V]) \leftarrow select(U, [X|Y], W), perm(W, V)$. The abstract clause associated to it, using the fixed α_1, AC, is $\texttt{:-}(perm([a_1|a_2], [a_3, a_4]), select(a_3, [a_1|a_2], a_5), perm(a_5, a_4))$. A possible renaming with respect to $ConAt$ could be $\texttt{:-}(perm([a_6|a_2], [a_3|a_4]), select(a_3, [a_6|a_2], a_5), perm(a_5, a_4)$. The selected atom in $ConAt$ is $perm(g_1, a_1)$. Then Abstract unify is applied to $perm(g_1, a_1)$ and

$perm([a_6|a_2],[a_3|a_4])$. Algorithm 1 terminates successfully and with final set $mgu = \{a_6 = g_3, a_2 = g_4, a_1 = [a_3|a_4], g_1 = [g_3|g_4]\}$.

Thus $\Theta_{mgu} = \{a_6/g_3, a_2/g_4, a_1/[a_3|a_4], g_1/[g_3|g_4]\}$.

Then, $Aresolve(ConAt, AC, perm(g_1, a_1)) = \wedge(select(a_3, [g_3|g_4], a_5), perm(a_5, a_4), ord([g_2, a_3|a_4])$, which is a renaming of the abstract conjunction of the next node of figure 3.2.

Proposition 6 (Correctness of Abstract resolve):
Let $ConAt \in AConAtom_P$, A_i an abstract atom in $ConAt$ and AC an abstraction of a Horn Clause C of P, such that $Aresolve(ConAt, AC, A_i)$ exists. Let $conat \in \gamma(ConAt)$ and cA_i the atom in $conat$ corresponding to A_i. Let the concrete resolvent of $conat$ and C, with selected literal cA_i, be $cresolve(conat, C, cA_i) \in ConAtom_P$. If $cresolve(conat, C, cA_i)$ exists, then $cresolve(conat, C, cA_i) \in \gamma(Aresolve(ConAt, AC, A_i))$.

Proof. Let $ConAt = \wedge(A_1, \ldots, A_i, \ldots, A_n)$ and $Rename(ConAt)(:\text{-}(H, B_1, \ldots, B_m)) = :\text{-}(H', B'_1, \ldots, B'_m)$, so that $Aresolve(ConAt, AC, A_i) = \wedge(A_1, \ldots, B'_1, \ldots, B'_m, \ldots, A_n)\Theta_{mgu(A_i,H')}$. Let $conat = \wedge(cA_1, \ldots, cA_i, \ldots, cA_n)$ and the renamed clause of C be $:\text{-}(h', b'_1, \ldots, b'_m)$, so that $cresolve(conat, C, cA_i) = \wedge(cA_1, \ldots, b'_1, \ldots, b'_m, \ldots, cA_n)\delta_{mgu(cA_i,h')}$.

We need to prove: $cA_j\delta_{mgu(cA_i,h')} \in \gamma(A_j\Theta_{mgu(A_i,H')})$, for all $j = 1, \ldots, i-1, i+1, \ldots, n$, and $b'_j\delta_{mgu(cA_i,h')} \in \gamma(B'_j\Theta_{mgu(A_i,H')})$, for all $j = 1, \ldots, m$. It is given that $cA_j \in \gamma(A_j)$ and $b'_j \in \gamma(B'_j)$ for all relevant j. Therefore, the term structures of cA_j and A_j (and b'_j and B'_j) are the same: they differ only in the subterms of A_j (and B'_j) that are abstract variables.

Therefore, it suffices to prove for each abstract variable a_k (and g_l) occurring in some A_j (or B'_j), with corresponding concrete term ca_k (or cg_l) in cA_j (or b'_j) that:

$$ca_k\delta_{mgu(cA_i,h')} \in \gamma(a_k\Theta_{mgu(A_i,H')})$$

$$cg_l\delta_{mgu(cA_i,h')} \in \gamma(g_l\Theta_{mgu(A_i,H')})$$

There are two cases to consider. In the first case, a_k (or g_l) occurs in (A_i, H'). Then, the result follows directly from proposition 4. In the second case, in which a_k (or g_l) does not occur in (A_i, H'), then $a_k\Theta_{mgu(A_i,H')} = a_k$ (and $g_l\Theta_{mgu(A_i,H')} = g_l$). Then, the result follows from the closedness under substitution of our abstract domain. □

3.2.3 Correctness of the approach

We are now ready to position our approach with respect to the main correctness theorem of the framework of Leuschel.

First, [45] imposes two important conditions on the abstract domain: *downward closedness* and *the existence of concrete dominators*. Downward closedness means that:

$$\forall A \in ADom_P, \forall t \in \gamma(A) : \{t\theta \mid \theta \text{ a substitution}\} \subseteq \gamma(A)$$

Our $ADom_P$ satisfies this property, by construction. This is because concrete variables are only represented by "a"-type abstract variables. Existence of a concrete dominator means that:

$$\forall A \in ADom_P, \exists t \in Dom_P : \gamma(A) \subseteq \{t\theta \mid \theta \text{ a substitution}\}$$

So, for any abstract conjunction, A, there exists a concrete conjunction, t, so that all the concrete conjunctions represented by A are instances of this t. As an example, the condition excludes an abstract domain in which some $\gamma(A)$ contains both $p(x)$ and $\wedge(p(x), p(f(x))$: the length of all concrete conjunctions represented by A needs to be the same. As another example, the condition excludes an abstract domain in which some $\gamma(A)$ contains both $p(x)$ and $q(x)$: the concrete conjunctions represented by A must all consist of fixed sequence of atoms, having the same sequence of predicate symbols.

The reason for imposing this condition is that it ensures that the usual way of generating resolvents, borrowed from concrete partial deduction, can still be applied in the abstract case. Again, the existence of a concrete dominator is clearly satisfied in our $ADom_P$.

Next, the framework introduces direct generalizations of the key correctness notions of $\mathcal{A}$-*closedness* and *independence* from the (concrete) partial deduction framework of [54]. The generalization of the former is referred to as $\mathcal{A}$-coveredness. Our approach implements our own, more strict version, *AC*-$\mathcal{A}$-closedness, which clearly includes Leuschel's definition of coveredness given earlier.

Independence means that for every A_i and $A_j \in \mathcal{A}$, $\gamma(A_i) \cap \gamma(A_j) = \emptyset$. Independence can always be achieved by appropriate renaming of predicates. Abstract conjunctive partial deduction requires that these renamings respect some basic correctness properties and refers to such renamings as those producing *admissible renamed variants*. The renaming of conjunctions to atoms that we apply in our approach, such as the one illustrated in section 3.1, produces admissible renamed variants.

Next, we recall the definition of an abstract partial deduction of P with respect to $\mathcal{A}$. The definition is based on two functions, *aunfold*/2 and *aresolve*/2.

Informally, for a program P and an abstract conjunction A, $aunfold(P, A)$ is a finite set of concrete resultants computed from P, such that these resultants are "totally correct" for all conjunctions in $\gamma(A)$: no computed answers are lost or added with respect to P. Definition 5.3 from [45] provides some fairly technical conditions, formalizing the above. We will not restate these conditions here, because proposition 5.6 in [45] provides and proves a simple sufficient condition under which the conditions on *aunfold* are fulfilled. The latter condition will be sufficient for our purpose.

In our approach, for $A \in \mathcal{A}$, $aunfold(P, A)$ is the set of all concrete resultants computed in the second phase of our analysis, for the concrete conjunction $c(A)$, using the same derivation steps as in the abstract tree for A. The correctness of our *aunfold*/2 is ensured by the following proposition:

Proposition 7 ([45], proposition 5.6):
Let $\mathbf{Q}$ be an abstract conjunction and let Q be a concrete dominator for $\mathbf{Q}$. Let τ be an SLD-tree for $P \cup \{\leftarrow Q\}$ and let $R \subseteq resultants(\tau)$ be a set of resultants such that for all resultants $Q\theta \leftarrow B \in (resultants(\tau) \backslash R)$ we have that no instance of $Q\theta$ is in $\gamma(\mathbf{Q})$. Then, $aunfold(P, \mathbf{Q}) = R$ satisfies definition 5.3 from [45].

In our approach, for $A \in \mathcal{A}$, $c(A)$ is a concrete dominator for A. In fact, $c(A)$ is a maximally specific concrete dominator for A. By construction, the derivations considered for $c(A)$ in our second step mimic the derivations in the abstract tree for A. Some of the derivations in the considered concrete tree for $c(A)$ could fail and not give rise to a resultant C_i. This is not a problem: the abstract derivations are only a safe approximation. On the other hand, additional concrete derivations could exist for $c(A)$, which we are not considering in the concrete tree because they have no counterpart in the abstract tree. They correspond to $resultants(\tau) \backslash R$ in proposition 7. This is not a problem either: by proposition 6, *Aresolve*/3 safely approximates all the concrete derivations for conjunctions in $\gamma(A)$. If additional ones exist for $c(A)$, say with resolvent $c(A)\theta \leftarrow B$, then $c(A)\theta \notin \gamma(A)$. Thus, proposition 7 holds for our approach.

The second function introduced in [45], *aresolve*/2, maps each abstract conjunction $A \in \mathcal{A}$ and resultant $C_i \in aunfold(P, A)$ to an abstract conjunction $aresolve(A, C_i)$, which safely approximates all possible resolvent goals that can occur after resolving an element of $\gamma(A)$ with C_i.

In our approach, by construction, for every $C_i \in aunfold(P, A)$, there is a branch in the corresponding abstract tree for A, consisting of a finite sequence of *Aresolve*/3 applications, serving as the template for the concrete derivation

from which C_i was obtained. By proposition 6, *Aresolve*/3 safely approximates every concrete derivation step. Thus, the abstract conjunction in the leaf of that branch safely approximates all possible concrete resolvent goals for elements of $\gamma(A)$ with C_i. Therefore, our approach also respects the condition on *aresolve*/2 of [45], definition 5.4.

We can now recall the definition of an abstract partial deduction.

Definition 24 (Abstract partial deduction, [45], definition 7.6):
Let $\mathcal{A}$ be an $\mathcal{A}$-covered set of abstract conjunctions. We then define an abstract partial deduction of P with respect to $\mathcal{A}$ to be the set of clauses:

$\{\rho_{\mathbf{A}}(H) \leftarrow \rho_{\mathcal{A},\mathbf{A}'}(B) \mid H \leftarrow B \in \mathit{aunfold}(P, \mathbf{A}) \wedge \mathbf{A}' = \mathit{aresolve}(\mathbf{A}, H \leftarrow B) \wedge \mathbf{A} \in \mathcal{A}\}$

The functions $\rho_{\mathbf{A}}$ and $\rho_{\mathcal{A},\mathbf{A}'}$ are renamings producing admissible renamed variants ([45], definitions 7.4, 7.5 and 8.1). Our program transformation, then, is defined as definition 24, using our own functions, corresponding to *aunfold*/2 and *aresolve*/2.

Finally, as our approach satisfies all the conditions of [45], the following correctness result holds for it:

Theorem 1 (*Correctness of the abstract partial deduction, [45], theorem 8.2*):
Let P' be an abstract partial deduction of P wrt an $\mathcal{A}$-covered set of abstract conjunctions $\mathcal{A}$ and let Q' be an admissible renamed variant of Q wrt $\mathcal{A}$.
Then:

1. If $P \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ then $P' \cup \{\leftarrow Q'\}$ has an SLD-refutation with computed answer θ' such that $Q\theta$ is a variant of $Q'\theta'$.
2. If $P' \cup \{\leftarrow Q'\}$ has an SLD-refutation with computed answer θ' then $P \cup \{\leftarrow Q\}$ has an SLD-refutation with computed answer θ such that $Q\theta$ is a variant of $Q'\theta'$.
3. If $P' \cup \{\leftarrow Q'\}$ has a finitely-failed SLD-tree then so does $P \cup \{\leftarrow Q\}$.

As a concluding observation, we want to motivate why we have not used the functions *aunfold*/2 and *aresolve*/2 from the start to introduce our instance of the framework. These functions are well-suited to formalize and reason about the correctness abstract partial deduction, but they are less suited to develop a new instance of the framework. In particular, *aresolve*/2 is defined in terms of *aunfold*/2, which suggests that an application would first generate concrete

resultants and then develop abstract derivation trees associated with these resultants. It is more natural — certainly in our application — to first compute the abstract derivation trees and then produce the resultants on the basis of these.

3.2.4 Control and widening

Apart from the correctness issues, abstract conjunctive partial deduction requires control. In [45], control issues are related to the vast amount of work done on the topic in partial deduction.

In our context, control is at the heart of the problem and must be specified explicitly. A first concern relates to the decisions regarding the selection rule.

One way to (indirectly) specify the selection rule is to provide delay declarations (e.g. block declarations in SICStus) and to order tests before generators in programs. This is how coroutines are typically implemented in Prolog programs, including the program in listing 2.2. The delay declarations express when a test is sufficiently instantiated to be unfolded. Using such declarations, we can ensure that a system takes the right decisions on which subgoal to unfold next. Unfortunately, such a system is not entirely orthogonal to the program logic, as evidenced by the introduction of *ord_*/3 in listing 2.2. Furthermore, the information in these block declarations does not translate to our abstract domain for all instances of the top-level query. For instance, the synthesis in listing 3.1 encodes the same computations as the version with block declarations in listing 2.2 for a top-level query $permsort([4,9,2,7,3,6],X)$ (which is a concrete instance of $permsort(g_1,a_1)$) but not for $permsort([4,9,2,7,3,6],[2,3,4,9,7,6])$. The latter is also an instance of $permsort(g_1,a_1)$, but our synthesis will still generate the first elements of the permutation (which is already given) whereas the SICStus version derives that 9 and 7 are out of order before generating any elements of the permutation.[4]

Therefore, we formalize a selection mechanism for *abstract* conjunctions. This mechanism relies on a strict partial order $\xrightarrow{p}$ on abstract atoms which expresses that an abstract atom takes strict precedence over another abstract atom. As a strict partial order, $\xrightarrow{p}$ is transitive, antisymmetric and irreflexive. The least element according to this order will be selected and syntactic ordering will be used to guarantee that atom selection takes place when there is no least element.

[4]This is not a regression with respect to the original compiling control. There, the synthesized program was always in accordance with its block declaration-based counterpart, but free variables in the symbolic top-level query could technically not be instantiated further.

Definition 25 (Abstract instantiation-based selection rule):
An abstract instantiation-based selection rule for P is a strict partial order $\xrightarrow{p}$ on $AAtom_{P/\approx}$, for which $\gamma(s_1) \subseteq \gamma(s_2)$ implies $s_2 \not\xrightarrow{p} s_1$.

Definition 26 (Selection by an abstract instantiation-based selection rule):
Let $A \in AConAtom_{P/\approx}$. The abstract atom selected by $\xrightarrow{p}$ is the leftmost atom b for which $\nexists c \in A : c \xrightarrow{p} b$.

For convenience, we introduce an assumption which is not necessary but which allows us to describe the control flow of our examples with less notational overhead: All fully evaluated abstract atoms atoms encountered during the analysis of P are unordered with respect to each other and for any fully evaluated abstract atom a and any non-fully evaluated abstract atom $b : a \xrightarrow{p} b$.

With this assumption and with the transitivity property in mind, we will represent $\xrightarrow{p}$ by a generating set of pairs, *Preprior*. The selection rule itself is inferred from *Preprior*.

Example 12:
For permutation sort, the control can be specified concisely as $Preprior = \{(ord([g_1, g_2|a_1]), perm(g_1, a_1))\}$.

For the second functionality of the oracle, the full abstract interpretation of abstract atoms, a user of our approach and system needs to specify which abstract atoms are not involved in the coroutine and, occasionally, which abstract atoms will always lead to failure. The *select*/3 predicate of section 3.1 is not involved in the coroutine. Note that it is convenient to be able to specify this on the level of abstract atoms, rather than predicates. We will show more intricacies of this oracle function when we cover the sameleaves example.

In addition to correctness and control, termination of the approach is also an issue. Ensuring a terminating analysis for partial deduction has been the focus of much research in the past. Quite a number of proposals have been made, including e.g. [51], [58], [50]. In any case, especially because our abstract domain is infinitely large, a widening operator is required to ensure that a finite covered set $\mathcal{A}$ can always be reached. We provide a function to compute the widening of abstract terms and atoms.

In the definition of a widening function below, $+_b$ denotes modulo-2 addition.

Definition 27 (Widening):
A widening function $\vee : ATerm_P \times ATerm_P \cup AAtom_P \times AAtom_P \rightarrow ATerm_P \cup AAtom_P$ is a one-to-one function, defined as:
for $A_0, A_1 \in ATerm_P \cup AAtom_P$:

- If A_i, for some $i \in \{0,1\}$, is of the form a_j, then $A_0 \vee A_1 \in \{a_k \mid k \in \mathbb{N}_0\}$.

- Else, if A_i, for some $i \in \{0,1\}$, is of the form g_j:
  - If $A_{i+_b 1}$ does not contain any symbol a_l, $A_0 \vee A_1 \in \{g_k \mid k \in \mathbb{N}_0\}$.
  - Otherwise, $A_0 \vee A_1 \in \{a_k \mid k \in \mathbb{N}_0\}$.
- Else, if $A_i = f(t_1, \ldots, t_n)$ and $A_{i+_b 1} = h(s_1, \ldots, s_m)$, for some $i \in \{0,1\}$:
  - If $f \neq h$ or $n \neq m$
    - If no A_i, $i \in \{0,1\}$, contains any symbol a_l: $A_0 \vee A_1 \in \{g_k \mid k \in \mathbb{N}_0\}$.
    - Otherwise $A_0 \vee A_1 \in \{a_k \mid k \in \mathbb{N}_0\}$.
  - Otherwise, $A_0 \vee A_1 = f(t_1 \vee s_1, \ldots, t_n \vee s_n)$

We can extend a widening function $\vee$ to $A_0, A_1 \in AConAtom_P$ by considering a conjunction as a special case of the third case above, with $f = g = \wedge$.

In what follows, by $\vee$, we denote a fixed widening function.

Example 13 (Widening):
We provide an illustration for each of the cases expressed in definition 27.

$$\vee(a_2, p(f(g_1), a_1)) = a_3$$
$$\vee(g_1, h(g_2, f(g_3))) = g_4$$
$$\vee(g_2, f(a_1)) = a_2$$
$$\vee(f(g_1), h(g_2, g_3)) = g_5$$
$$\vee(f(g_1, g_2), h(a_1)) = a_4$$
$$\vee(f(a_1, g_1, h(g_1, a_1)), f(a_2, g_3, h(g_4, a_2))) = f(a_5, g_6, h(g_7, a_5))$$

Given two abstract terms or atoms, it is tedious, but not hard to verify, that the widening operator, $\vee/2$, computes a minimally larger element of $ATerm_P$ or $AAtom_P$, with respect to $\succcurlyeq /2$. All these minimally larger elements are equivalent under $\approx /2$.

Similarly, given two abstract terms or atoms, abstract unify computes a maximally smaller element of $ATerm_P$ or $AAtom_P$, with respect to $\succcurlyeq /2$. Again, these are equivalent under $\approx /2$. As such, we establish that $(ATerm_{P/\approx}, \geq)$ and $(AAtom_{P/\approx}, \geq)$ are lattices.

Using the widening operator $\vee/2$, we can now extend definition 22 to an abstraction function on 2^{Dom_P}.

Definition 28 (Abstraction function):
Let $S \subseteq Term_P$ or $S \subseteq Atom_P$ or $S \subseteq AConAtom_P$.
Then the abstraction function on S is defined as: $\alpha(S) = \bigvee_{s \in S} \alpha_1(s)$.

A specific form of widening is "depth-k abstraction". In our setting, depth-k abstraction is somewhat more specific than in most logic programming techniques. Usually, depth-k abstraction replaces terms with concrete variables. In our case, it replaces abstract terms which only represent ground values with g-type abstract variables and other terms with a-type abstract variables.

To avoid any confusion, we use the following definition of depth-k abstraction.

Definition 29 (Depth-k abstraction):
Let t be an abstract term. The depth-k abstraction of t, denoted $d_k(t)$, is obtained by replacing any non-constant subterm of t at depth k with a newly created abstract variable. If the concretization $\gamma(st)$ of a replaced subterm st consists of only ground terms, st is replaced with a newly created variable of type g. Otherwise, it is replaced with a newly created variable of type a.

Example 14 (Depth-k abstraction):

- $d_0(foo(1, g_1)) = g_2$
- $d_1(foo(1, g_1)) = foo(1, g_1)$
- $d_1(foo(1, bar(g_1))) = foo(1, g_2)$

Widening carries the risk of overgeneralizing. In our case, this means that information which is relevant to the program control flow may be lost during the analysis. Therefore, an incorrect selection rule may be compiled if widening is used. This is shown in the following example:

Example 15:
Consider figure 3.2 on page 29. If the depth of predicate arguments is restricted to be at most 1, $select(a_2, [g_3|g_4], a_4), perm(a_4, a_3), ord([g_2, a_2|a_3])$ becomes $select(a_2, [g_3|g_4], a_4), perm(a_4, a_3), ord([g_2|a_5])$. The aliasing between the *perm/2* and *ord/1* atoms is lost and the program that can be synthesized from the completed analysis is incorrect. In fact, the incorrectly compiled selection rule triggers instantiation errors during some program executions.

3.3 A more complex example, introducing the *multi* abstraction

We have shown that abstract conjunctive partial deduction is indeed sufficient to formally revisit compiling control for a simple example. However, for more complex examples, abstract conjunctive partial deduction is not directly applicable. Consider the prime number generator in listing 3.2.

```
primes(N,P) :- integers(2,I),sift(I,P),len(P,N).
integers(N,[]).
integers(N,[N|I]) :- M is N+1, integers(M,I).
sift([N|Is],[N|Ps]) :- filter(N,Is,F), sift(F,Ps).
sift([],[]).
divides(N,M) :- 0 is M mod N.
does_not_divide(N,M) :- M mod N > 0.
filter(N,[M|I],F) :- divides(N,M), filter(N,I,F).
filter(N,[M|I],[M|F]) :- does_not_divide(N,M), filter(N,I,F).
filter(N,[],[]).
len([],0).
len([H|T],N) :- M is N - 1, len(T,M).
```

Listing 3.2: Prime sieve

The program is intended to be called with a goal $primes(N, P)$, with N a positive integer and P a free variable. The *integers*/2 predicate generates growing lists of consecutive integer numbers. The *filter*/3 predicate represents the removal of all multiples of a single integer N from a list. The *sift*/2 predicate recursively filters out multiples of an initial list element which is prime. Here, the desired abstract instantiation-based selection rule meets the assumptions needed for a concise specification. Rather than list *Preprior*, we describe it in terms of a set $KA = \{integers(g_1, a_1), sift(a_1, a_2), filter(g_1, a_1, a_2), len(a_1, g_2)\}$. The following rule provides sufficient information to complete the analysis phase: $\forall x \in AAtom_{P/\approx} : (\exists y \in KA : \gamma(x) \subset \gamma(y)) \Rightarrow \forall y \in KA : x \xrightarrow{p} y$ (where $\subset$ denotes the strict subset relation).

The SICStus program in listing 3.3 provides a concrete implementation of this control flow for the intended type of top-level query $primes(N, P)$, where N is a natural number and P is a free variable.

```
primes(N,P) :- len(P,N), sift(I,P), integers(2,I).
```

```
integers(N,[]).
integers(N,[N|I]) :- M is N+1, integers(M,I).

:- block sift(-,?).
sift([],[]).
sift([N|Is],[N|Ps]) :- sift(F,Ps), filter(N,Is,F).
divides(N,M) :- 0 is M mod N.
does_not_divide(N,M) :- X is M mod N, X > 0.

:- block filter(?,-,?).
filter(N,[],[]).
filter(N,[M|I],F) :- divides(N,M), filter(N,I,F).
filter(N,[M|I],[M|F]) :- does_not_divide(N,M), filter(N,I,F).

:- block len(-,?).
len([],0).
len([H|T],N) :- M is N - 1, len(T,M).
```

Listing 3.3: Prime sieve in SICStus

We only present the most relevant parts of the abstract partial deduction, which is significantly longer than that of permutation sort. The top level goal for the abstract analysis is $\mathit{primes}(g_1, a_1)$, so that the initial set $\mathcal{A}$ is $\{\mathit{primes}(g_1, a_1)\}$. A first abstract derivation tree describes the initialization for the computation. Its rightmost branch leads to the leaf: $\wedge(\mathit{integers}(g_2, a_3), \mathit{filter}(g_2, a_3, a_6), \mathit{sift}(a_6, a_5), \mathit{len}(a_5, g_3))$. Assume that this is added to $\mathcal{A}$.[5][6]

Next, we construct an abstract derivation tree for the latter conjunction. For the case where, in a concrete execution, a number passes through the filter, we add the following conjunction to $\mathcal{A}$:$\wedge(\mathit{integers}(g_5, a_{10}), \mathit{filter}(g_2, a_{10}, a_7), \mathit{filter}(g_3, a_7, a_8), \mathit{sift}(a_8, a_9), \mathit{len}(a_9, g_6))$.

At this point it becomes clear that an analysis following only the steps shown in section 3.1 will not terminate. The two abstract conjunctions, most recently added to $\mathcal{A}$, are identical – up to renaming of a_i's and g_j's – except that the latter conjunction contains two atoms $\mathit{filter}(g_k, a_i, a_j)$, instead of just one. A further analysis, building additional derivation trees, will result in the construction of continuously growing conjunctions, with continuously increasing numbers of $\mathit{filter}/3$ atoms.

[5] Actually, this conjunction is slightly more general than the one which is actually produced for the sake of presentation. The actual conjunction is more specific because we have included $ACon_P$ in the abstract domain, which makes the analysis more precise but longer.

[6] This abstract conjunction would not be added to $\mathcal{A}$ based on the criteria given earlier. Again, we do this for the sake of presentation and adding an additional abstract conjunction to $\mathcal{A}$ only leads to a more general analysis and does not affect correctness.

We could solve this by cutting the goal into two smaller conjunctions and adding these to $\mathcal{A}$. However, all these atoms are coroutines which communicate through shared variables. By splitting the conjunction, we would no longer be able to analyze the coroutining behavior. We therefore avoid splitting conjunctions if the evaluation of their atoms is interleaved.

Recall that abstract conjunctive partial deduction requires that, for any abstract conjunction of atoms, $acon \in AConAtom_P$, there exists a concrete dominator, $con \in ConAtom_P$, such that: for all $con_i \in \gamma(acon)$: con_i is an instance of con. In practice, this means that an abstract conjunction is not allowed to represent a set of concrete conjunctions whose elements have a distinct number of conjuncts. However, in order to solve the problem observed in our example, we need the ability to represent a set of conjunctions, with a growing number of atoms, by an abstract atom. Therefore, we need to extend abstract conjunctive partial deduction.[7]

We will extend our abstract domain and introduce a new abstraction, *multi*, which makes it possible to represent growing conjunctions, with a number of copies of a single abstract conjunction.

As the new abstraction function is fairly complex, let us first introduce the intuition. Assume that we have a number of abstract conjunctions in which there is one abstract atom with an increasing number of occurrences, while all other abstract atoms have a fixed number of occurrences. We can rename the indices of a_i and g_i variables occurring in the atoms whose frequency increases, such that the indices in abstract atoms **not** involved in the growing subconjunction are fixed.

For listing 3.2, the following conjunctions are renamed variants of abstract conjunctions we aim to generalize:

- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_5), sift(a_5, a_4), len(a_4, g_4))$
- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_6), filter(g_2, a_6, a_5), sift(a_5, a_4), len(a_4, g_4))$
- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_6), filter(g_2, a_6, a_7), filter(g_5, a_7, a_5), sift(a_5, a_4), len(a_4, g_4))$

We wish to introduce a new syntactic construct to represent the growing subconjunction of $filter(g_i, a_j, a_k)$ atoms. This construct should specify the abstract atom repeated in the subconjunction ($filter(g_i, a_j, a_k)$). It should also

[7] As will become apparent in chapter 4, there is an alternative approach which does not violate the concrete dominator assumption. However, it would require an additional program transformation and changes to the selection mechanism.

allow us to reconstruct all the aliasing on abstract variables, both between abstract atoms inside the subconjunction and between abstract atoms of the subconjunction with abstract atoms outside the subconjunction.

In our example, the new abstraction that we propose to represent the subconjunction is: $multi(filter(g_{Id(A),\mu,1}, a_{Id(A),\mu,3}, a_{Id(A),\mu,6}), \wedge(a_{Id(A),1,3} = a_3), \wedge(a_{Id(A),\mu+1,3} = a_{Id(A),\mu,6}), \wedge(a_{Id(A),\mathcal{L},6} = a_5))$ Here, $A = filter(g_1, a_3, a_6)$.

The first argument in the abstraction represents the abstract conjunction $(\overline{filter(g_1, a_3, a_6)})$ which is repeated in the grouped subconjunction, but also introduces a new naming scheme for the indices of a_i and g_j variables. This naming scheme has three components:

- $Id(A)$ is an identifier for the generalized subconjunction.
- The second component is a symbolic index. It can have four possible values: 1, $\mathcal{L}$, μ and $\mu + 1$. 1 and $\mathcal{L}$ respectively refer to the first and last atom of the conjunction. μ and $\mu + 1$ refer to a pair of consecutive atoms of the conjunction.

  Most often, we will use the symbolic index to represent the shared pattern of the abstracted atoms. We will then use the Greek letter μ. For instance, $filter(g_{Id(A),\mu,1}, a_{Id(A),\mu,3}, a_{Id(A),\mu,6})$ represents the pattern of the abstract atoms in the subconjunction. We can use the symbol μ to refer to a_i and g_j symbols of consecutive $filter(g_1, a_3, a_6)$ atoms in the sequence, like in an equation $a_{Id(A),\mu+1,3} = a_{Id(A),\mu,6}$.

  We also allow instantiating the symbolic index to a number or a symbol representing a specific number. For instance, with $a_{Id(A),1,3}$, we can refer to the a_3 variable in the **first** $filter(g_1, a_3, a_6)$ atom in the sequence. With $a_{Id(A),\mathcal{L},6}$, we can refer to the a_6 variable in the **last** $filter(g_1, a_3, a_6)$ atom in the sequence. Here, $\mathcal{L}$ is still a symbol, but it represents one specific (unknown) number related to the sequence.
- The last argument is an index used to identify a particular a_i or g_j in the $filter(g_1, a_3, a_6)$ abstracted conjunction.

The second, third and fourth arguments of the abstraction are conjunctions of constraints on the (newly named) a_i's. In our example, each of these conjunctions contains only one constraint. The second argument, $\wedge(a_{Id(A),1,3} = a_3)$, expresses the aliasing between the first $filter(g_1, a_3, a_6)$ atom in the sequence and the abstract atoms not contained in the subconjunction. The third argument, $\wedge(a_{Id(A),\mu+1,3} = a_{Id(A),\mu,6})$, expresses the aliasing between two consecutive $filter(g_1, a_3, a_6)$ atoms in the sequence. The fourth argument, $\wedge(a_{Id(A),\mathcal{L},6} = a_6)$,

expresses the aliasing between the last $filter(g_1, a_3, a_6)$ in the sequence and the abstract atoms not contained in the subconjunction.

We formally introduce the parameterized naming scheme for a_i and g_j variables and apply this to abstract atoms.

Definition 30 (Parameterized naming and parameterized abstract atom):
Let $A \in AConAtom_P$. By $Id(A)$, we denote a unique identifier associated with A.

Let $a_i \in AVar_P, i \in \mathbb{N}_0$, such that a_i occurs in A, then the parameterized naming of a_i is the symbol $a_{Id(A),\mu,i}$.

Let $g_j \in AVar_P, j \in \mathbb{N}_0$, such that g_j occurs in A, then the parameterized naming of g_j is the symbol $g_{Id(A),\mu,j}$.

The parameterized conjunction for A, $p(A)$, is obtained by replacing every a_i and g_j occurring in A by their parameterized namings, $a_{Id(A),\mu,i}$ and $g_{Id(A),\mu,j}$.

The new abstraction *multi* will depend on the abstract conjunction in which it is embedded, which we will define shortly as its *context*. This conjunction may contain abstract variables, a_i and g_j. It may also contain parameterized namings of abstract variables, $a_{Id(A),\mu,i}$ and $g_{Id(A),\mu,j}$. This is due to the fact that a *multi* abstraction will typically contain parameterized namings and that an abstract conjunction can contain multiple *multi* abstractions.

Definition 31 (Context of a subconjunction):
Let SC be a subconjunction of an abstract conjunction AC, such that SC is composed of the abstract atoms from (1-based) index position i through index position j in SC. Then, the context of SC in AC, denoted $ctxt(SC, AC)$, is the subconjunction composed of the abstract atoms from index position 1 through $i-1$ and from $j+1$ through the last index in AC.

Given an abstract conjunction C, we denote $a(C) = \{a_i \in AVar_P \mid a_i \text{ occurs in } C\}$. We also denote $g(C) = \{g_j \in AVar_P \mid g_j \text{ occurs in } C\}$. Furthermore, we denote $pa(C) = \{a_{Id(A),1,i} \mid a_{Id(A),1,i} \text{ occurs in } C\} \cup \{a_{Id(A),\mathcal{L},i} \mid a_{Id(A),\mathcal{L},i} \text{ occurs in } C\}$. We also denote $pg(C) = \{g_{Id(A),1,j} \mid g_{Id(A),1,j} \text{ occurs in } C\} \cup \{g_{Id(A),\mathcal{L},j} \mid g_{Id(A),\mathcal{L},j} \text{ occurs in } C\}$.

Example 16 (Context of a *multi* abstraction):
For the repeating atom in primes, referring to the three conjunctions AC we aim to represent (page 53), the context for subconjunctions SC of $filter(g_1, a_3, a_6)$ atoms is $ctxt(SC, AC) = \wedge(integers(g_3, a_3), sift(a_5, a_4), len(a_4, g_4))$. Note that $a(ctxt(SC, AC)) = \{a_3, a_4, a_5\}$, $g(ctxt(SC, AC)) = \{g_3, g_4\}$, $pa(ctxt(SC, AC)) = pg(ctxt(SC, AC)) = \emptyset$.

Definition 32 (*multi* abstraction):
Let AC be an abstract conjunction. A *multi* abstraction is a conjunct in AC of the form $multi(p(A), First, Consecutive, Last)$, where, if $multi(p(A), First, Consecutive, Last) = SC$:

- $p(A)$ is the parameterized conjunction for some $A \in AConAtom_P$.
- *First* is a conjunction of equalities of the form $a_{Id(A),1,i} = b_i$, where $b_i \in a(ctxt(SC, AC)) \cup pa(ctxt(SC, AC)) \cup g(ctxt(SC, AC)) \cup pg(ctxt(SC, AC))$ and all left-hand sides of the equalities are distinct, and equalities $g_{Id(A),1,j} = b_j$, where $b_j \in g(ctxt(SC, AC)) \cup pg(ctxt(SC, AC))$ and all left-hand sides of the equalities are distinct.
- *Consecutive* is a conjunction of equalities of the form $a_{Id(A),\mu+1,i} = a_{Id(A),\mu,i'}$, where $i, i' \in \mathbb{N}_0$ and all left-hand sides of the equalities are distinct, and equalities $g_{Id(A),\mu+1,j} = g_{Id(A),\mu,j'}$, where $k, k' \in \mathbb{N}_0$ and all left-hand sides of the equalities are distinct.
- *Last* is a conjunction of equalities of the form $a_{Id(A),\mathcal{L},i} = b_i$, where $b_i \in a(ctxt(SC, AC)) \cup pa(ctxt(SC, AC)) \cup g(ctxt(SC, AC)) \cup pg(ctxt(SC, AC))$ and all left-hand sides of the equalities are distinct, and equalities $g_{Id(A),\mathcal{L},j} = b_j$, where $b_j \in g(ctxt(SC, AC)) \cup pg(ctxt(SC, AC))$ and all left-hand sides of the equalities are distinct.

Example 17 (*multi* abstraction):
We return to the primes example and consider the three abstract conjunctions:

- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_5), sift(a_5, a_4), len(a_4, g_4))$
- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_6), filter(g_2, a_6, a_5), sift(a_5, a_4), len(a_4, g_4))$
- $\wedge(integers(g_3, a_3), filter(g_1, a_3, a_6), filter(g_2, a_6, a_7), filter(g_5, a_7, a_5), sift(a_5, a_4), len(a_4, g_4))$

Let $A = filter(g_1, a_3, a_6)$. Let any sequence of $filter/3$ atoms above be SC and let $ctxt(SC, AC) = \wedge(integers(g_3, a_3), sift(a_5, a_4), len(a_4, g_4))$. Then, each one of the three abstract conjunctions can be generalized, using the *multi* abstraction, to $\wedge(integers(g_3, a_3), multi(filter(g_{Id(A),\mu,1}, a_{Id(A),\mu,3}, a_{Id(A),\mu,6}), \wedge(a_{Id(A),1,3} = a_3), \wedge(a_{Id(A),\mu+1,3} = a_{Id(A),\mu,6}), \wedge(a_{Id(A),\mathcal{L},6} = a_5)), sift(a_5, a_4), len(a_4, g_4))$.

We can now define a new abstract domain, which includes the abstraction.

Definition 33 (Abstract-multi domain):
Let $AAtomMulti_P = AAtom_P \cup \{multi(p(A), First, Consecutive, Last) \mid A \in$

$AAtom_P$ and *First*, *Consecutive* and *Last* are conjunctions of equalities $a_{Id(A),\mu,i} = b_i$ or $g_{Id(A),\mu,j} = b_j$, with b_i, b_j abstract variables or parameterized abstract variables }.

Let $AConAtomMulti_P$ be the set of all conjunctions AC of elements of $AAtomMulti_P$, such that each of its $multi(p(A), First, Consecutive, Last)$ conjuncts, denoted SC, respects the conditions of definition 32.

The Abstract-multi domain, $ADomMulti_P$, is $AVar_P \cup ATerm_P \cup AAtom_P \cup AConAtomMulti_P$.

Abstract conjunctions containing *multi* abstractions represent infinitely many abstract conjunctions without the *multi* abstraction. In the example, these contain either one or multiple $filter(g_j, a_i, a_k)$ atoms.

In what follows, we will omit the $Id(A)$ subscript in the parameterized namings $a_{Id(A),\mu,j}$ and $g_{Id(A),\mu,k}$ and just refer to $a_{\mu,j}$ and $g_{\mu,k}$ instead. The $Id(A)$ subscript is only relevant for abstract conjunctions containing multiple *multi* abstractions, a case which we will not consider for the moment.

In order to describe the abstract conjunctions represented by an abstract conjunction containing a *multi* abstraction, we need the ability to map parameterized namings back to ordinary a_j and g_k variables. This requires the following concepts.

Definition 34 (Concrete index assignment mapping):
Let $n \in \mathbb{N}_0$. The concrete index assignment mapping, $R(\mu, n)$, is a mapping defined on any syntactic construct, S, containing parameterized namings $a_{\mu,i}$, $a_{\mu+1,i}$, $g_{\mu,j}$ or $g_{\mu+1,j}$. $R(\mu, n)$ replaces every occurrence of a parameterized naming $a_{\mu,i}$, $a_{\mu+1,i}$, $g_{\mu,j}$ and $g_{\mu+1,j}$ in S by the parameterized naming $a_{n,i}$, $a_{n+1,i}$, $g_{n,j}$ and $g_{n+1,j}$, respectively.

Note that, while $\mu + 1$ is a symbol, $n + 1$ is a number.

Example 18 (Concrete index assignment mapping):
$R(\mu, 1)(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6})) = filter(g_{1,1}, a_{1,3}, a_{1,6})$.

Definition 35 (Double-index mapping):
Let s be any symbolic index. The double-index mapping, ψ, is a mapping defined on any syntactic construct, S, containing parameterized namings $a_{s,i}$ or $g_{s,k}$. ψ replaces every occurrence of a parameterized naming $a_{s,i}$ and $g_{s,j}$ in S by a_{s_i}, respectively g_{s_j} where s_i and s_j denote fresh elements of $\mathbb{N}_0$, not occurring as the index of any abstract variable yet.

Example 19 (Double-index mapping):
$\psi(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6})) = filter(g_{\mu_1}, a_{\mu_3}, a_{\mu_6})$, with μ_1, μ_3, μ_6 fresh elements of $\mathbb{N}_0$.

Definition 36 (Substitution corresponding to equality constraints): Let s be any symbolic index. Let *Constraint* be a conjunction of equality constraints of the form $a_{s,i} = b_i$ and $g_{s,j} = b_j$, with $a_{s,i}$ and $g_{s,j}$ parameterized namings, and such that all left-hand sides of equalities are mutually distinct. The substitution corresponding to *Constraint* is the substitution $\Theta_{Constraint} = \{\psi(a_{s,i})/\psi(b_i) \mid a_{s,i} = b_i \in Constraint\} \cup \{\psi(g_{s,j})/\psi(b_j) \mid g_{s,j} = b_j \in Constraint\}$.

Note that this definition is meant to deal with the conjunctions of equalities in the *First*, *Consecutive* and *Last* arguments of the *multi* abstraction. The *Last* argument requires some extra attention. Sometimes, during analysis, we will actually know the value of $\mathcal{L}$. To avoid confusion, we will use the symbol l when this value is known.

Example 20 (Substitutions corresponding to equality constraints): For the conjunctions of equality constraints within the *multi* abstraction in example 17, the corresponding substitutions are: $\Theta_{First} = \{a_{1_3}/a_3\}$, $\Theta_{Consecutive} = \{a_{(\mu+1)_3}/a_{\mu_6}\}$, $\Theta_{Last} = \{a_{\mathcal{L}_6}/a_5\}$.

Note that, while $\Theta_{Consecutive}$ and Θ_{Last} cannot be applied directly to abstract conjunctions, $\Theta_{R(\mu,n)(Consecutive)}$ and $\Theta_{R(\mathcal{L},n)(Last)}$ can be applied if n is a concrete index.

With these notions, we can now describe the abstract conjunctions represented by a *multi* abstraction.

Definition 37 (Abstract conjunctions represented by *multi*): Let $l \in \mathbb{N}_0$. The abstract conjunctions represented by $multi(p(A), First, Consecutive, Last)$ are:

- $\psi(R(\mu,1)(p(A)))\Theta_{First} \circ \Theta_{R(\mathcal{L},1)(Last)}$
- $\psi(R(\mu,1)(p(A)))\Theta_{First} \wedge \psi(R(\mu,2)(p(A)))\Theta_{R(\mu,1)(Consecutive)} \wedge \ldots \wedge \psi(R(\mu,l)(p(A)))\Theta_{R(\mu,l-1)(Consecutive)} \circ \Theta_{R(\mathcal{L},l)(Last)}$, with $l > 1$.

Example 21 (Abstract conjunctions represented by *multi*): For the *multi* abstraction in example 17, $multi(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6}), \wedge(a_{1,3} = a_3), \wedge(a_{\mu+1,3} = a_{\mu,6}), \wedge(a_{\mathcal{L},6} = a_5))$, the represented abstract conjunctions are:

- $filter(g_{1_1}, a_3, a_5)$
- $filter(g_{1_1}, a_3, a_{1_6}) \wedge filter(g_{2_1}, a_{1_6}, a_{2_3}) \wedge \ldots \wedge filter(g_{l_1}, a_{(l-1)_6}, a_5), l > 1.$

Note that, by applying definition 37 and by using the concretization function γ on the resulting elements of $ADom_P$, we have now defined the concretization of all elements of $ADomMulti_P$.

Next, we need to define the abstract unfolding of a *multi* abstraction. Unfolding a *multi* abstraction makes a case split. Either the *multi* abstraction represents only one abstract conjunction, or it represents more than one. When the *multi* represents more than one conjunction, we extract an instance of the conjunction and a new *multi* abstraction. Whether the instance of the conjunction appears to the left or to the right of the new *multi* abstraction is determined by the oracle: in practice, this is dependent upon the selection rule. In all cases the abstract bindings with the surrounding abstract conjunction and, in case the abstraction represents several conjunctions, the abstract bindings between consecutive conjunctions need to be respected.

Definition 38 (Abstract unfold of *multi*):
Abstract unfold of *multi* produces a branching in the abstract derivation tree. Let $\bigwedge$ represent the application of $\wedge$ to a set of arguments. An abstract atom $multi(p(A), First, Consecutive, Last)$ is replaced in one branch by $\psi(R(\mu,1)(p(A)))\Theta_{First} \circ \Theta_{R(\mathcal{L},1)(Last)}$ and in a second branch by $\psi(R(\mu,1)(p(A)))\Theta_{First} \wedge multi(p(A), NewFirst, Consecutive, Last)$, where $NewFirst = \bigwedge\{a_{1,i} = a_{1_{i'}} \mid a_{(\mu+1),i} = a_{\mu,i'} \in Consecutive\} \wedge \bigwedge\{g_{1,j} = g_{1_{j'}} \mid g_{(\mu+1),j} = g_{\mu,j'} \in Consecutive\}$ or $multi(p(A), First, Consecutive, NewLast) \wedge \psi(R(\mu,l)p(A))$, where l is a concrete index and $NewLast = \bigwedge\{a_{\mathcal{L},i} = a_{\mathcal{L}_{i'}} \mid a_{(\mu+1),i'} = a_{\mu,i} \in Consecutive\} \wedge \bigwedge\{g_{\mathcal{L},j} = g_{\mathcal{L}_{j'}} \mid g_{(\mu+1),j'} = g_{\mu,j} \in Consecutive\}$

When a set *NewFirst* is introduced, we will say that the *multi* abstraction is unfolded "from the left" and when a set *NewLast* is introduced, we will say that the *multi* abstraction is unfolded "from the right".

Example 22 (Abstract unfold of *multi*):
Again returning to example 17, abstract unfold of $multi(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6}), \wedge(a_{1,3} = a_3), \wedge(a_{\mu+1,3} = a_{\mu,6}), \wedge(a_{\mathcal{L},6} = a_5))$ produces both $filter(g_{1_1}, a_3, a_5)$ and $filter(g_{1_1}, a_3, a_{1_6}) \wedge multi(filter(g_{\mu,1}, a_{\mu,3}, a_{\mu,6}), \wedge(a_{1,3} = a_{1_6}), \wedge(a_{\mu+1,3} = a_{\mu,6}), \wedge(a_{\mathcal{L},6} = a_5))$. This assumes the abstraction is unfolded from the left.

A few comments on this definition are in order. First, the definition of *NewFirst* and *NewLast* may seem strange, because both sides of the equalities have a "1" index. However, note that on the left-hand side of the equality, it is in a parameterized naming, e.g. $a_{1,i}$, referring to the first atom represented by the *multi*, while on the right-hand side, it is in an abstract variable $a_{1_{i'}}$, referring to an atom that was just moved outside of the *multi*. Second, it is important to remember that the abstract variables a_{1_i} and g_{1_j} are produced by $\psi(a_{1,i})$ and

$\psi(g_{1,j})$ calls and that their indices 1_i and 1_j need to be fresh indices, not yet occurring with any a or g variables, respectively. This is particularly important in cases where we perform several abstract unfoldings of *multi* in sequence. At each unfold, new fresh subscripts need to be introduced.

Finally, we need to define abstract generalization with *multi*, allowing us to replace conjunctions of identically instantiated and similarly aliased abstract atoms by a *multi* construct.

Definition 39 (Abstract generalization with *multi*):
Let $A \in AConAtom_P$. Let $A_1, \ldots, A_n \in AConAtom_P$ and let $\bigwedge_{m=1,\ldots,n} A_m$ occur in an abstract conjunction Con. Let $C = Con - (\wedge_{m=1,\ldots,n} A_m)$. Let $a(C)$, $g(C)$ and $pg(C)$, $pa(C)$ respectively be the abstract variables and the parameterized namings occurring in C. Let $r_m, m = 1, \ldots, n$, be renamings of A, such that $r_m(A) = A_m$.

$Gen(\bigwedge_{m=1,\ldots,n} A_m) = multi(p(A), First, Consecutive, Last)$ is the abstract generalization with *multi* of $\bigwedge_{m=1,\ldots,n} A_m$ in C if:

- for any $b_i \in a(C) \cup pa(C), a_{1,i} = b_i \in First$ if and only if $r_1(a_i) = b_i$
- for any $b_j \in g(C) \cup pg(C), g_{1,j} = b_j \in First$ if and only if $r_1(g_j) = b_j$
- $a_{\mu+1,i} = a_{\mu,i'} \in Consecutive$ if and only if for all $p = 1, \ldots, n-1$: $r_{p+1}(a_i) = r_p(a_{i'})$
- $g_{\mu+1,j} = g_{\mu,j'} \in Consecutive$ if and only if for all $p = 1, \ldots, n-1$: $r_{p+1}(g_j) = r_p(g_{j'})$
- for any $b_i \in a(C) \cup pa(C)$, $a_{\mathcal{L},i} = b_i \in Last$ if and only if $r_n(a_i) = b_i$
- for any $b_j \in g(C) \cup pg(C)$, $g_{\mathcal{L},j} = b_j \in Last$ if and only if $r_n(g_j) = b_j$

We can extend definition 39 to a generalization $Gen(\bigwedge_{m=1,\ldots,k} A_m \wedge multi(p(A), First, Consecutive, Last)) = multi(p(A), First', Consecutive, Last)$ and to similar generalizations which combine an existing *multi* abstraction with an abstract conjunction. We omit the details for these generalizations.

We will illustrate abstract generalization with *multi* in the prime numbers example. Observing the growing number of *filter*/3 atoms in the second abstract conjunction of example 17 (w.r.t. the conjunction already present in $\mathcal{A}$), we perform the generalization. Recall that $C = \wedge(integers(g_3, a_3), sift(a_5, a_4), len(a_4, g_4))$, so that $a(C) = \{a_3, a_4, a_5\}$ and $g(C) = \{g_3, g_4\}$. $Gen(\wedge(filter(g_1, a_3, a_6), filter(g_2, a_6, a_5))) = multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,1} = a_3), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5))$. Here, we include the $Id(A)$ again, because we will have multiple *multi* abstractions. We arbitrarily select $Id(A)$ to be 1.

Then we construct a new abstract derivation tree for this conjunction, including – among others – an abstract unfold of *multi* and abstract generalizations with *multi*. In figure 3.5, we show this abstract tree.

After abstract unfolding of $integers(g_3, a_3)$, the tree contains an abstract unfolding of $multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = [g_3|a_7]), \wedge(a_{1,\mu+1,6} = a_{1,\mu,3}), \wedge(a_{1,\mathcal{L},6} = a_5))$. This unfolding can lead to one instance of *filter*/3 or several. A full evaluation of $does_not_divide(g_6, g_3)$ leads to a new generalization which produces a renaming of the root of this tree.

Eventually, the analysis ends up with a final set $\mathcal{A}$:

$$\{ \wedge (primes(g_3, a_3)),$$

$$\wedge (integers(g_3, a_3), multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = a_3),$$

$$\wedge (a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5)), sift(a_5, a_4), len(a_4, g_4)),$$

$$\wedge (multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = g_5), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}),$$

$$\wedge (a_{1,\mathcal{L},6} = a_5)), sift(a_5, a_4), len(a_4, g_4)),$$

$$\wedge (integers(g_5, a_7), multi(filter(g_{2,\mu,1}, a_{2,\mu,3}, a_{2,\mu,6}), \wedge(a_{2,1,3} = a_7),$$

$$\wedge (a_{2,\mu+1,3} = a_{2,\mu,6}), \wedge(a_{2,\mathcal{L},6} = a_8)), multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}),$$

$$\wedge (a_{1,1,3} = [g_6|a_{2,\mathcal{L},6}]), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5)), sift(a_5, a_4),$$

$$len(a_4, g_4))\}$$

All non-empty leaves in the abstract derivation trees for these atoms are (renamings of) elements of $\mathcal{A}$. This shows AC-$\mathcal{A}$-closedness and the abstract phase of the analysis terminates.

Similar to what was observed for permutation sort, we still need an extra analysis to collect the concrete bindings, so that the resultants can be generated. Special care is required for the *multi* abstraction. There are three issues: how to represent *multi* in the concrete domain, how to deal with the concrete counterparts of abstract generalization with *multi* and abstract unfolding of *multi*.

Definition 14, on page 29, defined the concrete counterparts of the conjunctions in $\mathcal{A}$. We extend it to $multi(A)$:

Definition 40 (Concrete conjunction for $multi(A, First, Consecutive, Last)$**):**

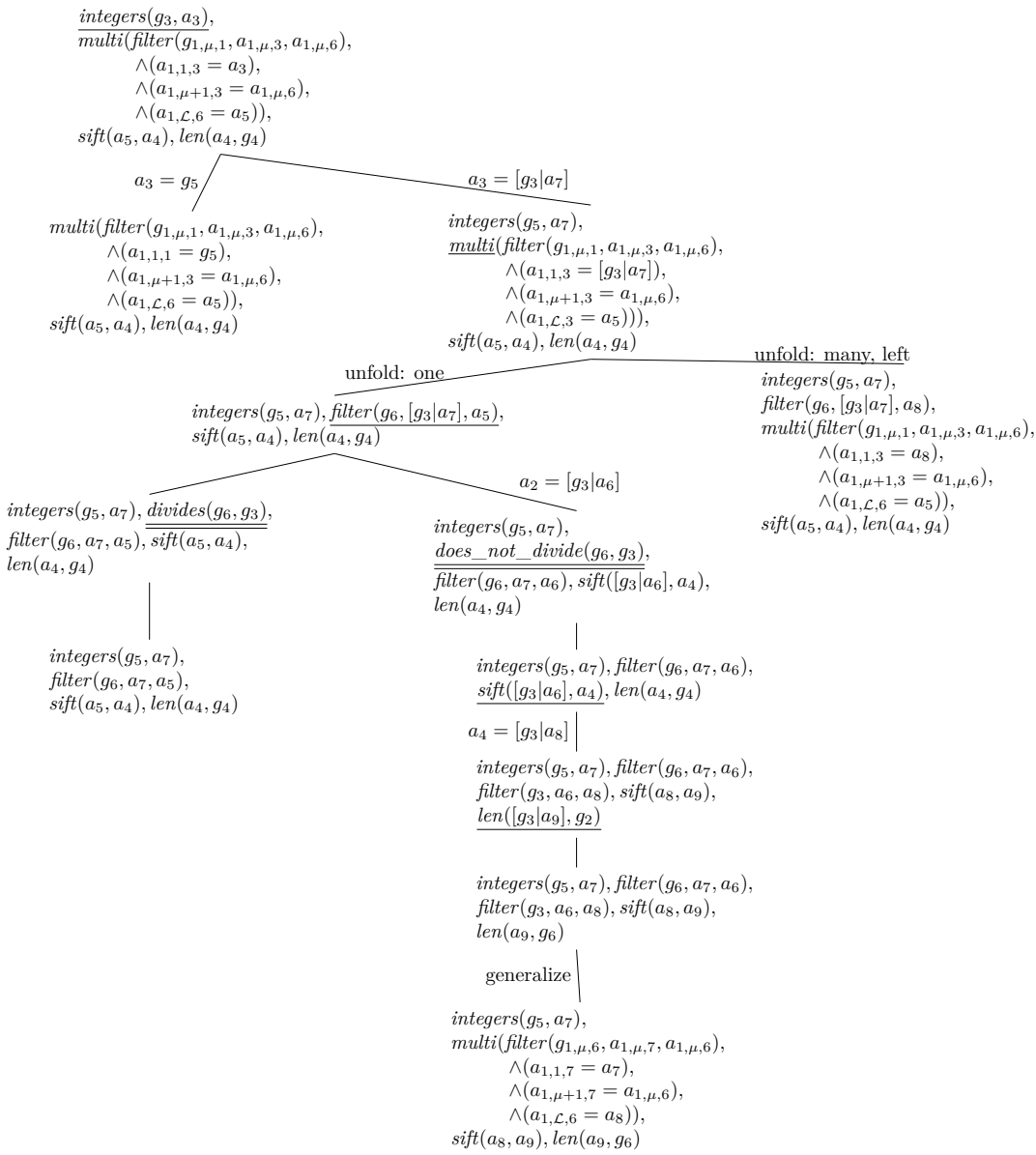


Figure 3.5: Use of the *multi* abstraction in the prime sieve

Let $A \in AAtom_P$, then $c(multi(p(A), First, Consecutive, Last)) = multi([c(A)|T])$, with T a fresh variable.

It may seem strange that in the concrete analysis phase we omit the three arguments *First*, *Consecutive* and *Last*. These arguments are needed in the abstract analysis to correctly capture the data flow and to correctly model the unfolding under the coroutining selection rule. In the concrete analysis phase, as we are completely mimicking the unfolding in the corresponding abstract trees, we are still performing the correct selection. Moreover, the only point of the concrete analysis phase is to collect the concrete bindings produced by unfolding the concrete clauses. The extra arguments are not needed for this purpose.

Example 23 (Concrete conjunction for $multi(A, First, Consecutive, Last)$):
$c(multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = a_3), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5))) = multi([filter(X, I1, F1)|T])$

For the abstract generalization with *multi*, we define the concrete counterpart as follows.

Definition 41 (Concrete generalization):
Let $A \in AAtom$.

- If the abstract generalization with *multi* is of the type $Gen(\bigwedge_{i=1,\ldots,n} A) = multi(A, First, Consecutive, Last)$, then the corresponding node in the concrete derivation contains $c(\bigwedge_{i=1,\ldots,n} A)$. The concrete generalization is defined as $ConGen(c(\bigwedge_{i=1,\ldots,n} A)) = multi(c([A, \ldots, A]))$, with n conjunctions in the list.
- If the generalization with *multi* is of the type $Gen((\bigwedge_{i=1,\ldots,n} A) \wedge multi(A, First, Consecutive, Last)) = multi(A, First', Consecutive, Last)$, then the corresponding node in the concrete derivation contains $c(\bigwedge_{i=1,\ldots,n} A) \wedge multi(List)$, where *List* is a list of at least one $c(A)$. The concrete generalization is $ConGen(c(\bigwedge_{i=1,\ldots,n} A) \wedge multi(List)) = multi([c(A), \ldots, c(A)|List])$ with n new members prefixed to *List*.
- The third case, $Gen(multi(A, First, Consecutive, Last) \wedge (\bigwedge_{i=1,\ldots,n} A)) = multi(A, First, Consecutive, Last')$, is treated similarly to the previous case, but the concrete conjunctions are appended to the existing list.
- The final case is combines two *multi* abstractions into a single abstraction. In this case, $ConGen(multi(List1) \wedge multi(List2)) = multi(List3)$, where $List3$ is the concatenation of $List1$ and $List2$.

Example 24 (Concrete generalization):
Let $integers(A, B), filter(C, B, D), filter(E, D, F), sift(F, G), len(G, H)$ occur in a concrete conjunction in a concrete derivation tree, where abstract generalization with *multi* is performed on the corresponding abstract conjunction. Then, as a next step in the concrete derivation tree, this conjunction is replaced by $integers(A, B), multi([filter(C, B, D), filter(E, D, F)]), sift(F, G), len(G, H)$.

Note that this "generalization" actually does not generalize anything. It only gives the information a form that can be represented more generally. The actual generalization happens implicitly in the move to the construction of the next concrete derivation tree. If our conjunction is a leaf of the concrete derivation tree, then the corresponding abstract conjunction is added to the set $\mathcal{A}$. Let $\wedge(integers(g_3, a_3), multi(filter(g_{1,\mu,1}, a_{1,\mu,3}, a_{1,\mu,6}), \wedge(a_{1,1,3} = a_3), \wedge(a_{1,\mu+1,3} = a_{1,\mu,6}), \wedge(a_{1,\mathcal{L},6} = a_5)), sift(a_5, a_4), len(a_4, g_4))$, for instance, be the corresponding abstract conjunction that is added to $\mathcal{A}$. Then, a new concrete tree is built for a concrete conjunction corresponding to this abstract one.

In this example, the root of that concrete tree is $\wedge(integers(A, B), multi([filter(C, B, D)|T]), sift(E, F), len(F, G))$

Finally, we still need to define the counterpart of abstract unfold of *multi* in the concrete tree. To do this, we add the following definition of $multi/1$ to the original program P for the synthesis step.

```
multi([H]) :- H.
multi([H|T]) :- H, multi(T).
multi([H|T]) :- last([H|T],L,R), multi(R), L.
```

Here, *last/3* extracts the last element from its first argument and unifies the remainder of the list with the third argument. Concrete unfolding of concrete $multi/1$ atoms with the above definition for $multi/1$ gives us the desired counterpart of the case split performed in abstract unfold of $multi/1$ if we apply concrete bindings analogous to abstract bindings used in the abstract unfold.

With the concepts above, we construct a concrete derivation tree, mimicking the steps in the abstract derivation tree – but over the concrete domain – for every conjunction in the set $\mathcal{A}$. Collecting all the resultants from these concrete trees, we get the transformed program. A working Prolog program is provided in the electronic appendix, along with syntheses for several other examples.[8]

[8] `www.programming-emu.com/phd-appendix.zip`

3.4 A final example, illustrating more aspects

Consider the program in listing 3.4. Its purpose is to check whether its two arguments, which are assumed to be trees, have an identical sequence of leaf nodes. Assuming a top-level query of the form $sameleaves(g_1, g_2)$, this program can also be executed more efficiently by using a non-standard selection rule.

```
sameleaves(T1,T2) :-
  collect(T1,T1L), collect(T2,T2L), eq(T1L,T2L).
eq([],[]).
eq([H|T1],[H|T2]) :- eq(T1,T2).
collect(node(X),[X]).
collect(tree(L,R),C) :-
  collect(L,CL), collect(R,CR), append(CL,CR,C).
append([],L,L).
append([H|T],L,[H|TR]) :- append(T,L,TR).
```

Listing 3.4: Sameleaves

A variant for SICStus Prolog again uses block declarations to implement the desired control flow:

```
sameleaves(T1,T2) :-
  eq(T1L,T2L),
  collect(T1,T1L),
  collect(T2,T2L).

:- block eq(-,-).
eq([],[]).
eq([H|T1],[H|T2]) :- eq(T1,T2).

collect(node(X),[X]).
collect(tree(L,R),C) :-
  my_append(CL,CR,C), collect(L,CL), collect(R,CR).

:- block my_append(-,?,-).
my_append([],L,L).
my_append([H|T],L,[H|TR]) :- my_append(T,L,TR).
```

Listing 3.5: Sameleaves in SICStus

The following abstract conjunctions occur in the analysis phase. The underlined subconjunctions form a repeating pattern.

- $collect(g_1, a_1)$, $\underline{collect(g_2, a_2), append(a_1, a_2, a_3)}$, $collect(g_3, a_4)$, $eq(a_3, a_4)$
- $collect(g_1, a_1)$, $\underline{collect(g_2, a_2), append(a_1, a_2, a_3), collect(g_3, a_4), append(a_3, a_4, a_5)}$, $collect(g_4, a_6)$, $eq(a_5, a_6)$
- $collect(g_1, a_1)$, $\underline{collect(g_2, a_2), append(a_1, a_2, a_3), collect(g_3, a_4), append(a_3, a_4, a_5), collect(g_4, a_6), append(a_5, a_6, a_7)}$, $collect(g_5, a_8)$, $eq(a_7, a_8)$

There is the subconjunction $collect(g_2, a_2), append(a_1, a_2, a_3)$ which has a growing number of occurrences in a fixed sequential pattern $collect(g_i, a_i)$, $append(a_j, a_i, a_k)$. This is the reason why the repeated elements in a *multi* abstraction belong to $AConAtom_P$, rather than $AAtom_P$.

This example also showcases unfolding of *multi* "from the right". The reason for this is that a node value is first obtained from the first tree, then assumed to be equal to the corresponding value, then "pushed down" the second tree. The result is that a computed leaf node value sometimes "bubbles" through the multi abstraction from left to right and sometimes from right to left. Figure 3.6 shows a small segment of the analysis of the sameleaves program which illustrates the first scenario. Figure 3.7 illustrates the second scenario.

Note that no additional information is required in order to choose whether to unfold from the left or from the right: Unfolding the multi from the left in this conjunction produces a subconjunction which is less instantiated than the subconjunction which can be unfolded from the right. Therefore, the selection rule determines whether a *multi* abstraction is unfolded from the left or from the right.

While it is significantly longer than the analysis of the prime sieve, the analysis of sameleaves requires no additional techniques other than the aforementioned two. The analysis and synthesis can be reproduced according to the instructions given in appendix E.

3.5 Remarks

In this chapter, we have shown that compiling control can be recast as an instance of abstract conjunctive partial deduction in the case of "simple" programs. These "simple" coroutines essentially correspond to the *strongly regular* logic programs of [80], based on [40].

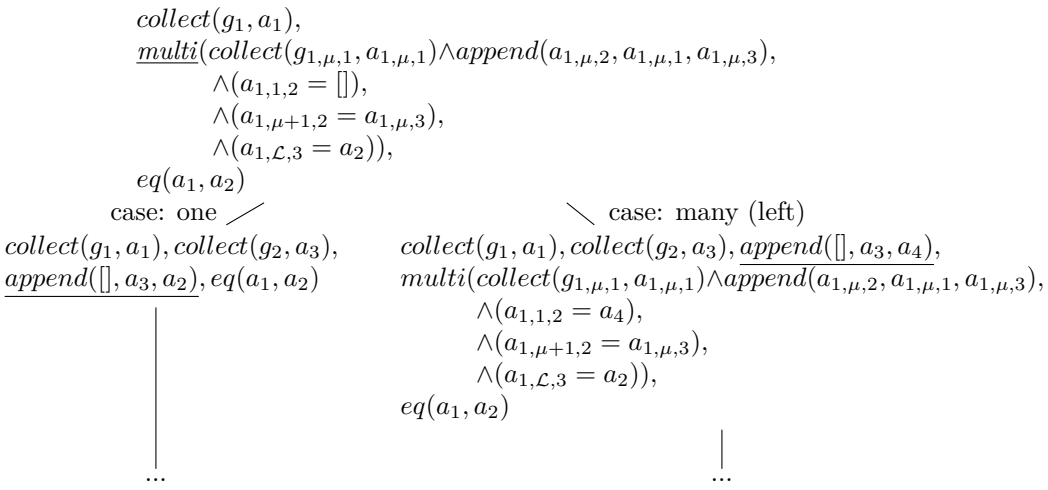


Figure 3.6: A *multi* abstraction is unfolded "from the left" during the analysis of sameleaves

To achieve this, we defined an expressive abstract domain, capturing modes, term structure and aliasing. We provide a full formalization of the approach, including Abstract unify, Abstract resolve, the order on the abstract domain, widening and the abstraction function.

Because the approach – for simple coroutines – fits fully within the abstract conjunctive partial deduction framework, it inherits the correctness results from the framework. In particular, $\mathcal{A}$-coveredness in the form of *AC*-$\mathcal{A}$-closedness and independence guarantee the completeness and correctness of the analysis. In addition, the transformation preserves all computed answers (in both directions) and finite failure of the transformed program implies finite failure of the original.

We have proposed an extension to our abstract domain: the *multi*-abstraction. A *multi* can represent one or more concrete conjunctions. We have defined abstract unfold and abstract generalization operations for this abstraction. We have shown, in an example, that this abstraction and these operations allow us to extend ACPD, enabling it to perform a complete analysis, and to compile the more complex coroutines. However, the *multi* abstraction does not fit within the framework, as it explicitly violates Leuschel's requirement that, for every abstract conjunction, there is a concrete dominator. An alternative perspective which does not call for changes to an existing framework is offered in chapter 6.

With $ADomMulti_P$, several well-known generate-and-test programs can be compiled. Examples to which the approach can be applied are:

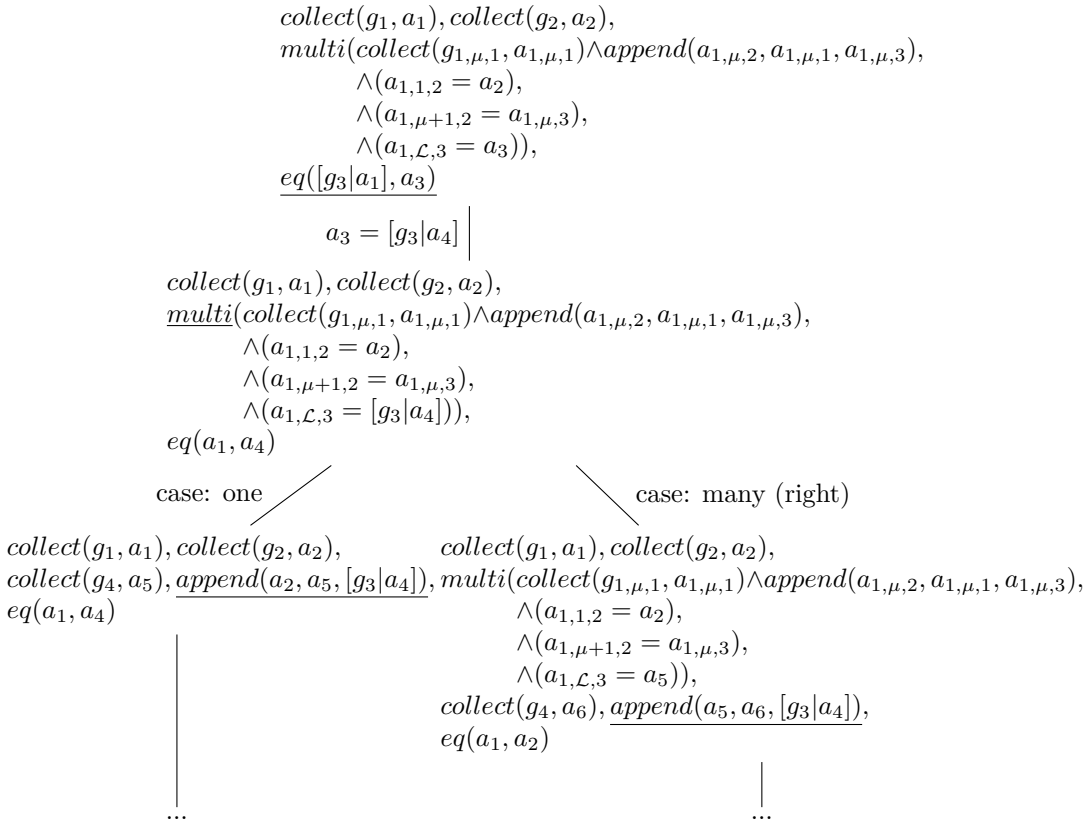


Figure 3.7: A *multi* abstraction is unfolded "from the right" during the analysis of sameleaves

- permutation sort
- graph coloring
- primes (and similar sieves, such as one for "lucky" numbers)
- N queens (and N confused queens)
- sameleaves

Chapter 4

Limitations

4.1 Scope

The previous chapter ended with an overview of some representative programs which can be compiled using published techniques. These programs provide a base line, but they do not cover the full spectrum of programs with a finite, instantiation-based selection rule. In the current chapter, we present several new techniques which allow a strictly greater set of programs to be transformed. The examples provided will be somewhat artificial, because even simple programs which require these techniques have large analysis trees and because the techniques have not yet been automated. However, we believe that they can be used for the analysis of larger, more practically useful coroutining programs. The work in this chapter has not previously been published.

4.2 Lowest common denominator control

The approach described in chapter 3 encodes a selection rule which the programmer considers optimal based on limited information regarding instantiation. Consider again listing 2.1 on page 10. Assuming that optimality equates to a minimal number of inferences required, the encoded selection rule can be considered "optimal" for queries of the form $permsort(X, Y)$, where X is ground and Y is a free variable. It cannot be considered optimal for more specific instances of the same query. For instance, under an optimal selection rule, the query $\leftarrow permsort([2,7,6,5],[2,5,7,6])$ can be answered without ever

resolving a *perm/2* atom, because $ord([2,5,7,6])$ must fail. The version with block declarations does, indeed, infer that $7 \not\leq 6$ without permuting. The program synthesis, however, encodes a selection rule which *interleaves* $perm/2$ and $ord/1$.[1] In this sense, the encoded selection rule is an optimal "lowest common denominator": It is the best that can be done with the information regarding instantiation and aliasing that is provided in and can be inferred from the abstract top-level query. We first provide a more formal explanation of why and when this discrepancy between compiled and non-compiled control appears.

4.2.1 Correspondence between abstract and concrete selection

Definition 26 on page 48 detailed how the selection of *abstract* atoms occurs during the analysis phase. The discrepancy between compiled and non-compiled control arises when the selection of an abstract atom does not correspond to the selection of a concrete atom by a runtime mechanism, such as one based on block declarations. To bridge the gap between these two types of mechanisms, we introduce a new concept, i.e. the selection of a *concrete* atom by an abstract instantiation-based selection rule.

Definition 42 (Concrete selection by an abstract instantiation-based selection rule):
Let A be a conjunction of atoms and let $\xrightarrow{p}$ be an abstract instantiation-based selection rule. Let PA be $\{\mathbf{a} | \exists \mathbf{b} : \mathbf{a} \xrightarrow{p} \mathbf{b} \vee \mathbf{b} \xrightarrow{p} \mathbf{a}\} \cup \{pred(t_1,\ldots,t_n) | pred(t_1,\ldots,t_n)$ is an upper bound w.r.t. $\preccurlyeq$ for all instances of $pred/n$ encountered during the analysis of $P\}$. Let $\mathbf{A}$ be conjunction of abstract atoms, for which all of the following hold:

- $A \in \gamma(\mathbf{A})$
- each abstract atom $\mathbf{a}$ in $\mathbf{A}$ occurs in PA
- the abstract atom $\mathbf{a}$ corresponding to any concrete atom c is minimal (i.e. "not preceded") w.r.t. $\xrightarrow{p}$ among all abstract atoms $\mathbf{a}'$ in PA for which $c \in \gamma(\mathbf{a}')$

Then, the concrete atom selected from A is the one corresponding to the abstract atom selected from $\mathbf{A}$.

[1] The version with block declarations is also not "optimal" for all instances, e.g. $permsort([2,7,6,5],[A,B,7,6])$, but the point is that it allows more flexibility.

Example 25 (Concrete selection by an abstract instantiation-based selection rule (1)):
Let $A = perm([4, 9, 7, 3, 6], A), ord([2|A])$. Let the abstract instantiation-based selection rule be as in example 12 on page 48. Assume the upper bound from definition 42 is a *least* upper bound.[2] Then, the most specific abstraction of the first atom in PA is $perm(g_1, a_1)$ and the most specific abstraction of the second atom is $ord(a_1)$. Because neither abstraction is ordered with respect to the other, the selected concrete atom is the leftmost one, $perm([4, 9, 7, 3, 6], A)$.

Example 26 (Concrete selection by an abstract instantiation-based selection rule (2)):
Let $A = perm([4, 9, 7, 6], A), ord([2, 3|A])$. Let the abstract instantiation-based selection rule be as in the previous example. Then, the most specific abstraction of the first atom in PA is $perm(g_1, a_1)$ and the most specific abstraction of the second atom is $ord([g_1, g_2|a_1])$. Because $ord([g_1, g_2|a_1]) \xrightarrow{p} perm(g_1, a_1)$, the selected concrete atom is $ord([2, 3|A])$.

We can guarantee that abstract selection corresponds to concrete selection for a single conjunction, i.e. that the compiled selection rule implements the desired runtime selections, under specific conditions.

Proposition 8 (Local concrete-abstract selection correspondence):
Let A be a concrete conjunction of atoms and let $\mathbf{A}$ be an abstraction of A, such that $\mathbf{A}$ consists only of atoms occurring in the selection rule or in the abstract analysis of P for the top-level query. Furthermore, let $\mathbf{A}$ be maximally specific (i.e. minimal w.r.t. $\preccurlyeq$, see page 40) within these parameters. Then, concrete selection by an abstract-instantiation based selection rule always selects the concrete atom matching the abstract atom selected through abstract selection by the same selection rule.

Proof. Each abstract atom $\mathbf{a}$ in $\mathbf{A}$ abstracting some atom a is minimal w.r.t. $\preccurlyeq$ among the atoms in the selection rule and the analysis by definition. Because $\gamma(\mathbf{a}) \subseteq \gamma(\mathbf{a}')$ implies $\mathbf{a}' \not\xrightarrow{p} \mathbf{a}$, each abstract atom $\mathbf{a}$ is also minimal w.r.t. $\xrightarrow{p}$ among a's potential abstractions in PA (see definition 42). Then, $\mathbf{A}$ meets the requirements of $\mathbf{A}$ in definition 42, so the concrete atom selected from A matches the abstract atom selected from $\mathbf{A}$. □

Proposition 8 does not mean that the concrete selections by an abstract instantiation-based selection rule match the abstract selections performed during a complete analysis. It remains to be shown that concrete resolution cannot

[2] A more general upper bound would also work, but could require that we disambiguate the selection rule in a way which has no bearing on the analysis.

introduce information relevant to the control flow which is not captured by the abstract analysis. If this were possible, an analysis starting from a maximally specific abstraction could lead to the introduction of abstractions which are not maximally specific, so that proposition 8 would only be safely applicable to the very first abstract and concrete conjunction. An additional proposition guarantees that relevant information is not introduced under specific conditions.

Proposition 9 (Global concrete-abstract selection correspondence): Assume depth-k abstraction is never applied. Let $A_1, \ldots, A_n$ be a conjunction of concrete atoms. Let $\mathbf{A_1}, \ldots, \mathbf{A_n}$ be a maximally specific abstraction of $A_1, \ldots, A_n$, such that $\mathbf{A_1}, \ldots, \mathbf{A_n}$ all occur in the selection rule or in the abstract analysis of P. Let $\mathbf{H} \leftarrow \mathbf{B_1}, \ldots, \mathbf{B_m}$ be $\alpha(H \leftarrow B_1, \ldots, B_m)$ (where $H \leftarrow B_1, \ldots, B_m$ is some program clause in P), let θ be the unifier for H and some A_i and let Θ be the abstract unifier for $\mathbf{H}$ and $\mathbf{A_i}$. Then, there cannot be an abstract substitution Σ such that $(A_1, \ldots, B_1, \ldots, B_m, \ldots, A_n)\theta \in \gamma((\mathbf{A_1}, \ldots, \mathbf{B_1}, \ldots, \mathbf{B_m}, \ldots, \mathbf{A_n})\Theta\Sigma)$ and such that $\mathbf{C_j}\Theta \not\preccurlyeq \mathbf{C_k}$ and $\mathbf{C_j}\Theta\Sigma \preccurlyeq \mathbf{C_k}$, where $\mathbf{C_j}$ is an abstract atom in $\mathbf{A_1}, \ldots, \mathbf{B_1}, \ldots, \mathbf{B_m}, \ldots, \mathbf{A_n}$ and $\mathbf{C_k}$ is some abstract atom which occurs in the selection rule or in the abstract analysis.

A corollary of this is that proposition 8 can be applied to every concrete conjunction and its abstraction, if the abstraction of the top-level conjunction is maximally precise within the parameters of proposition 8.

Proof of proposition 9. Assume an abstract substitution Σ which restricts an abstract atom to a more specific abstract atom in the analysis or selection rule as in proposition 9 does exist. That is, for at least one atom C_j, $C_j\theta \in \gamma(\mathbf{C_j}\Theta\Sigma)$, which implies $C_j\theta \in \gamma(\mathbf{C_j}\Theta)$. This can only be the case if an application of Σ aliases abstract variables v_1 and v_2 in $\mathbf{C_j}\Theta$, or if it instantiates at least one abstract variable v to a more specific abstract term (or a combination of both).

Suppose Σ aliases two abstract variables v_1 and v_2 in $\mathbf{C_j}\Theta$. Then, θ aliases the corresponding concrete terms if they were not already aliased in C_j. This happens if and only if the concrete terms are unified with aliased concrete terms t_1 and t_2 in H. By definition of the abstraction of a program clause, the abstractions of t_1 and t_2 are also aliased in $\mathbf{H}$. Therefore, v_1 and v_2 must be aliased by Θ and cannot be aliased by Σ.

Next, suppose Σ further instantiates some abstract variable v in $\mathbf{C_j}\Theta$. Then, θ further instantiates the corresponding concrete term c from C_j if c was not already in $\gamma(v\Theta\Sigma)$ prior to unification. This happens if and only if c is unified with a concrete term c' in H, such that all their common instances are in $\gamma(v\Theta\Sigma)$. This means that c' is not equally general as or more general than c. By definition of the abstraction of a program clause, any term structure of c' is

retained in its abstraction in $\mathbf{H}$. Therefore, v is sufficiently instantiated by Θ and is not further instantiated by Σ.

As a result, Σ specifies an abstract term by neither aliasing nor further instantiating any subterms, which is contradictory. □

Proposition 9 explains why the atom selections performed during the abstract analysis are guaranteed to match the concrete atom selections performed for a top-level query such as $permsort([4,9,2,7,3,6],X)$, but not for a top-level query such as $permsort([4,9,2,7,3,6],[2,3|X])$: $permsort(g_1,a_1)$ is a maximally specific abstraction of the former top-level query among the abstract conjunctions in the selection rule and analysis, but it is not a maximally specific abstraction of the latter. A maximally specific abstraction of the latter would be $permsort(g_1,[g_2,g_3|a_1])$.

4.2.2 Speculative compiling control

In some cases, proposition 9 is too restrictive. It may not be practical to encode all information related to instantiation and aliasing in the abstraction of the top-level query. A workaround within the abstract conjunctive partial deduction framework seems out of place. The reason for this is that abstract conjunctive partial deduction requires that the image of γ is downwards-closed. That is, if $\mathbf{A}$ is an abstract conjunction: $\forall Q \in \gamma(\mathbf{A}) : \{Q\theta | \theta \text{ is a (concrete) substitution}\} \subseteq \gamma(\mathbf{A})$. This is the reason why the approach in chapter 3 uses abstract variables a_i which denote any concrete value, rather than abstract variables f_i which denote free variables. All analyses would — if the *multi* abstraction were integrated into the framework — be considered sound, yet one would still perform several analyses and select the synthesis made for the most specific abstraction of the top-level query. In addition, there can be many degrees of freedom and there may not be a good synthesis which covers all use cases.

The cause of this limitation is the fact that atoms must be selected for unfolding and full evaluation based on information which is available during the abstract analysis phase. If certain additional runtime checks are permissible, it *is* possible to encode a selection rule which does not only emulate block declarations for queries of the form $permsort(X,Y)$, where X is ground and Y is a free variable, but for all queries of the form $permsort(X,Y)$, where X is ground.

Under the assumption that no other constants than [] and numbers are supplied in the top-level query, permutation sort can be analyzed using an alternative abstract domain which includes a more expressive version of $ATerm_P$. The foundation of this alternative $ATerm_P$ is shown in figure 4.1. We do not provide

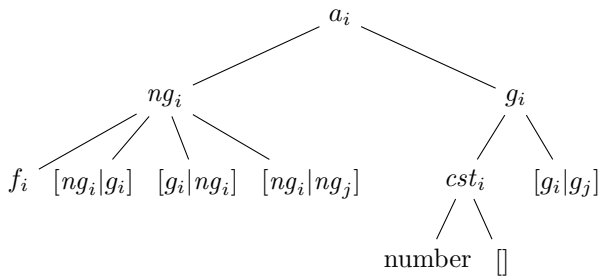


Figure 4.1: Foundation for alternative $ATerm_P$

a fully formalized definition of this set of abstract terms, as it is still quite experimental, but we will state the intended denotation of new elements. In figure 4.1, ng_k stands for any non-ground term. The use of a subscript is analogous to that in the abstract variables a_i and g_j from chapter 3. Another abstract variable is f_m, which stands for any free variable. A symbol cst_n stands for any constant, which in the case of permutation sort is either a number or the empty list. Note that all types represented on a single level of the diagram are disjoint. This allows case splits based on these types to be performed easily. These case splits can be mimicked during a concrete execution with builtins and they partition the concretization of some abstract domain element into the disjoint concretizations of a set of more specific abstract domain elements. For instance, if an abstract term a_1 matches a more specific abstract term f_1, and if X is a concrete term corresponding to a_1 in a concrete execution, $var(X)$ holds.

The complete alternative $ATerm_P$ used for the analysis of permutation sort also includes expressions using an operator ++. These are not shown on the diagram and represent the concatenation of the left and right operand, which are both lists. For example, if the only program constants are the empty list and the numbers 1 through 3, $\gamma(g_1{+}{+}g_1)$ starts with $\{[], [1,1], [2,2], [3,3], [1,1,1,1], [1,2,1,2], [1,3,1,3], [2,1,2,1], \ldots\}$. To give a different example, $\gamma([1,2,3]{+}{+}a_1)$ would be the set of all instances of the concrete term $[1,2,3|X]$. They are not shown for two reasons: they complicate the presentation and, more importantly, they are not disjoint from the types that would be positioned on the same level in figure 4.1. This overlap is not an issue if a case split based on whether a term is the concatenation of two abstract terms is never applied.

The fact that the resulting abstract domain is not downwards closed is not an issue: downwards closedness is a requirement of the abstract conjunctive partial deduction framework, but not of abstract interpretation of logic programs in general.

A representative fragment of the analysis (which is considerably longer than that of permutation sort with least common denominator control) is provided in figure 4.2 on page 76. The dotted lines starting from the root node represent the first case split: either the term represented by a_1 is also an instance of $[g_j|a_i]$, or it is not. If it is, abstract resolution and full evaluation are applied until a state is reached to which a "regrouping" operation can be applied. This is a form of widening of an expression with ++ which retains the right operand. The resulting conjunction is a variant of the root conjunction and the branch is closed. If a_1 is not an instance of $[g_j|a_i]$, it must be an instance of some other pattern. All of these patterns are enumerated, using the partitioned structure of $ATerm_P$ to create disjoint analyses.

This leads to more detailed case splits: either a_1 is an instance of ng_k, or it is an instance of g_j. If a_1 is an instance of ng_k, several more case splits are required. Figure 4.2 does not show all cases. It only shows the case in which a_1 is an instance of f_m *and* g_2 is non-empty. After the calls to *perm*/2 and to *select*/3, a variant of a case of the top node is obtained, so another cycle is introduced. Returning to the second case split that was performed, if a_1 is not an instance of ng_k, but an instance of g_j and not an instance of $[g_j|a_i]$, it *must* also be an instance of the abstract empty list, []. Otherwise, $ord([g_3|a_1])$ will never succeed. This information can be obtained from an oracle. Realistically, this would be a separate call set analysis. This means that the resulting node, for the only potentially non-failing branch where a_1 is also an instance of g_j and not an instance of $[g_j|a_i]$, can be simplified to $perm(g_1, g_2), ord([g_3])$. In this case, the residual definitions of *ord*/1 and *perm* can be used because interleaving is not useful.

Note that the analysis does not distinguish between failure and program errors. For instance, the query $permsort([2, 7, 6, 5], [2, 5, [A], []])$, in the SICStus implementation, raises an error. In the synthesis corresponding to the analysis in figure 4.2, it fails. This is only done for simplicity and additional runtime checks could be used to emulate the behavior induced by the block declarations more precisely.

A synthesis based on figure 4.2 is provided in listing 4.1. A complete synthesis is provided in appendix F. We call this extension of compiling control "speculative compiling control".

```
s1(G1,G2,A1,G3) :-
  nonvar(A1),
  A1 =.. ['[|]',G4,A2],
  ground(G4),
  G3 =< G4,
```

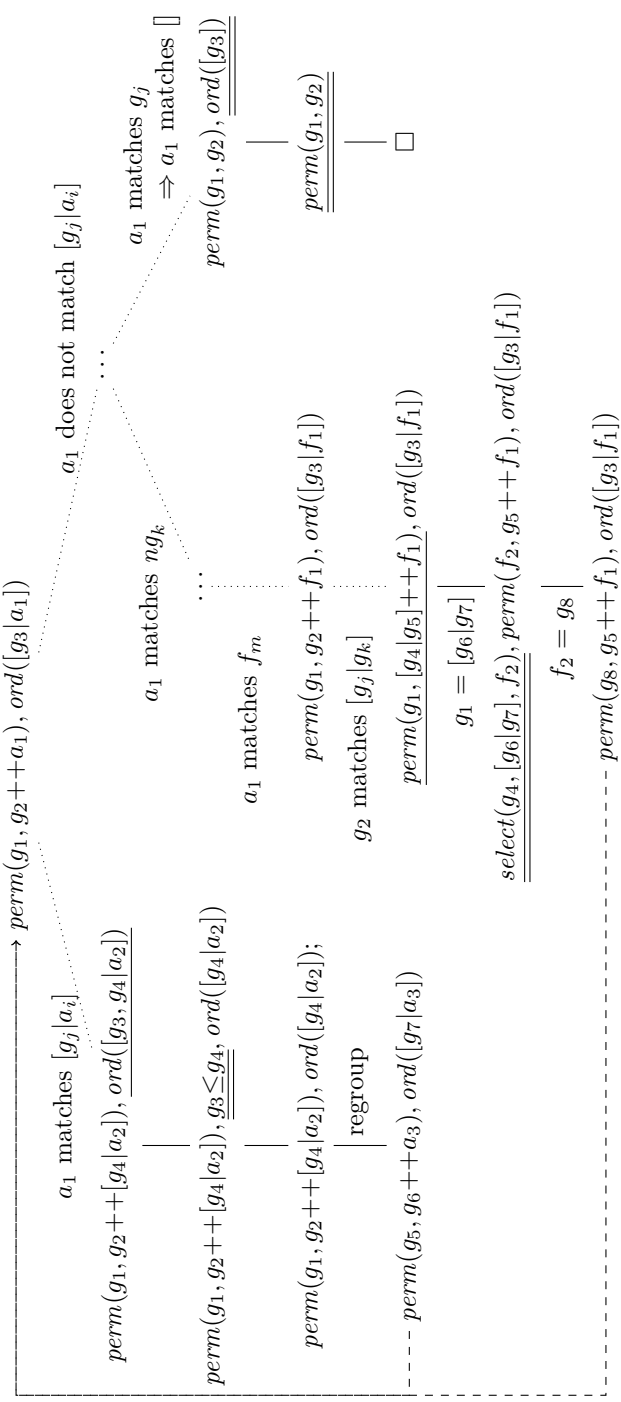


Figure 4.2: Speculative compiling control analysis of permutation sort

```
  append(G2,[G4],G6),
  s1(G1,G6,A2,G4).
s1([G6|G7],[G4|G5],A1,G3) :-
  var(A1),
  select(G4,[G6|G7],F2),
  s1(F2,G5,A1,G3).
s1(G1,G2,A1,_G3) :-
  A1 == [],
  perm(G1,G2).

perm([],[]).
perm([X|Y],[U|V]) :-
  select(U,[X|Y],W), perm(W,V).
```

Listing 4.1: Partial permutation sort with runtime checks

Added runtime checks using predicates such as $var/1$ partition the concretization of some abstract domain element into the disjoint concretizations of a set of more specific abstract domain elements. For instance, if $var(a_1)$ succeeds, a_1 is restricted to match f_1, whose concretization is the set of all free variables. While the added runtime checks lead to a more complicated synthesis involving non-logical builtins, this synthesis still frees analysis tools from the burden of handling multiple control mechanisms.

Unfortunately, speculative compiling control is not currently supported by the implementation for several reasons: First, the abstract domain is tailored to known examples and a less ad hoc abstract domain would be preferable — the ++ operator, especially, is tailored to this specific problem. Second, even permutation sort requires a non-trivial regrouping step, which is based on insight into the program and which displays a conceptual similarity with the *multi* abstraction. Third, the analysis uses information which would be obtained from an oracle or, more realistically, an additional tool for abstract interpretation. For instance, a branch of the analysis tree is aborted due to the presence of and $ord([[ng|a]|a])$ atom, based on the assumption that $ord/1$ fails (or produces an error) when called with anything other than a flat list of numbers.[3] A general approach to speculative compiling control may be possible, but for the remainder of this text we will consider only lowest common denominator control.

[3]In Prolog, $ord/1$ can actually be called with a list of evaluables, but we are free to use a restricted version which makes assumptions about Dom_P.

4.3 Expressivity of *multi*

The *multi* abstraction, as described in chapter 3, is not expressive enough to deal with certain variations on the example programs referenced in the preceding chapters. Consider the graph coloring Prolog program given in listing 4.2.

```
test(N,C,N1,C1) :-
  bi_edge(N,N1),
  C \== C1 ;
  N \== N1,
  bi_noedge(N,N1).

safe_coloring(Ns,Cs) :-
  coloring(Cs),
  safe(Ns,Cs).

coloring([],[]).
coloring([C|Cs]) :-
  color(C),
  coloring(Cs).

safe([],[]).
safe([_],[_]).
safe([N1,N2|Ns],[C1,C2|Cs]) :-
  allsafe(N1,C1,[N2|Ns],[C2|Cs]),
  safe([N2|Ns],[C2|Cs]).

allsafe(_,_,[],[]).
allsafe(N1,C1,[N2|Ns],[C2|Cs]) :-
  test(N1,C1,N2,C2),
  allsafe(N1,C1,Ns,Cs).
```

Listing 4.2: Graph coloring

Assume a top-level query of the form $safe_coloring(g_1, a_1)$ and assume a selection rule which tests whether a coloring is safe whenever said coloring contains at least two colors, i.e. whenever the list argument of *coloring*/1 begins with two ground values. Under the assumption explained on page 48, such a selection rule can be represented by a set *Preprior* which contains at least the following information:

- $allsafe(g_1, g_2, g_3, [g_4|a_1]) \xrightarrow{p} safe([g_1|g_2], [g_3, g_4|a_1])$
- $allsafe(g_1, g_2, [g_3|g_4], [g_5|a_1]) \xrightarrow{p} safe([g_1, g_2|g_3], [g_4, g_5|a_1])$

- $allsafe(g_1, g_2, g_3, []) \xrightarrow{p} safe([g_1|g_2], [g_3])$
- $allsafe(g_1, g_2, [], []) \xrightarrow{p} safe([g_1], [g_2])$
- $coloring(a_1) \xrightarrow{p} safe([g_1|g_2], [g_3|a_1])$
- $coloring(a_1) \xrightarrow{p} allsafe(g_1, g_2, g_3, a_1)$
- $safe(g_1, [g_2, g_3|a_1]) \xrightarrow{p} coloring(a_1)$

An analysis of this program is quite simple, though it does require the use of the *multi* abstraction. Such an analysis can be generated using the implementation by following the guidelines in appendix E. Next, consider the variant of the graph coloring program in listing 4.3. Again, assume a selection rule which tests whether a coloring is safe whenever said coloring contains at least two colors. Such a selection rule consists of pairs indicating precedence like the ones just given, but using either *allsafeA* or *allsafeB* as a symbol wherever *allsafe* was used. Abstract atoms *allsafeA* and *allsafeB* identical levels of instantiation are not ordered with respect to each other, so a sequence composed of both types of atoms is traversed from left to right. Then, both programs produce the same answers for the same concrete top-level query (though the second produces most answers multiple times) and display the same finite failure behavior.

```
test(N,C,N1,C1) :-
  bi_edge(N,N1),
  C \== C1 ;
  N \== N1,
  bi_noedge(N,N1).

safe_coloring(Ns,Cs) :-
  coloring(Cs),
  safe(Ns,Cs).

coloring([]),[].
coloring([C|Cs]) :-
  color(C),
  coloring(Cs).

safe([],[]).
safe([_],[_]).
safe([N1,N2|Ns],[C1,C2|Cs]) :-
  allsafeA(N1,C1,[N2|Ns],[C2|Cs]),
```

```
  safe([N2|Ns],[C2|Cs]).
safe([N1,N2|Ns],[C1,C2|Cs]) :-
  allsafeB(N1,C1,[N2|Ns],[C2|Cs]),
  safe([N2|Ns],[C2|Cs]).

allsafeA(_,_,[],[]).
allsafeA(N1,C1,[N2|Ns],[C2|Cs]) :-
  test(N1,C1,N2,C2),
  allsafeA(N1,C1,Ns,Cs).

allsafeB(_,_,[],[]).
allsafeB(N1,C1,[N2|Ns],[C2|Cs]) :-
  test(N1,C1,N2,C2),
  allsafeB(N1,C1,Ns,Cs).
```

Listing 4.3: Alternating graph coloring variant

Nonetheless, the *multi* abstraction cannot fully capture the analysis of the second program. The reason for this is that, on most branches, a sequence of arbitrary length composed of *allsafeA* and *allsafeB* abstract atoms appears and subconjunctions of this sequence do not unify with any single pattern of bounded length. To illustrate, the following conjunction could be produced during the abstract analysis of the original graph coloring program:

$$coloring(a_1), allsafe(g_2, g_3, g_1, a_1), allsafe(g_4, g_5, g_1, a_1), allsafe(g_6, g_7, g_1, a_1),$$

$$allsafe(g_8, g_9, [g_{10}|g_1], [g_{11}|a_1]), safe([g_{10}|g_1], [g_{11}|a_1]).$$

This can be generalized to:

$$coloring(a_1), multi(allsafe(g_{1,\mu,2}, g_{1,\mu,3}, g_{1,\mu,1}, a_{1,\mu,1}), \wedge(a_{1,1,1} = a_1),$$

$$\wedge(g_{1,\mu+1,1} = g_{1,\mu,1}, a_{1,\mu+1,1} = a_{1,\mu,1}), \wedge()),$$

$$allsafe(g_8, g_9, [g_{10}|g_1], [g_{11}|a_1]), safe([g_{10}|g_1], [g_{11}|a_1]).$$

The variant may produce all of the following conjunctions:

- $coloring(a_1), allsafeA(g_2, g_3, g_1, a_1), allsafeA(g_4, g_5, g_1, a_1),$ $allsafeA(g_6, g_7, g_1, a_1), allsafeA(g_8, g_9, [g_{10}|g_1], [g_{11}|a_1]),$ $safe([g_{10}|g_1], [g_{11}|a_1]).$
- $coloring(a_1), allsafeB(g_2, g_3, g_1, a_1), allsafeB(g_4, g_5, g_1, a_1),$ $allsafeA(g_6, g_7, g_1, a_1), allsafeA(g_8, g_9, [g_{10}|g_1], [g_{11}|a_1]),$ $safe([g_{10}|g_1], [g_{11}|a_1]).$

- $coloring(a_1), allsafeA(g_2, g_3, g_1, a_1), allsafeB(g_4, g_5, g_1, a_1),$
  $allsafeA(g_6, g_7, g_1, a_1), allsafeA(g_8, g_9, [g_{10}|g_1], [g_{11}|a_1]),$
  $safe([g_{10}|g_1], [g_{11}|a_1]).$

There is no generalization which uses the previously described *multi* abstraction which covers all of these. However, atomic subconjunctions of this sequence do unify with *one of two* patterns. Hence, the sequence can be represented using an extension of the *multi* abstraction, which we will refer to as $multi^{\vee}$. In this case, the abstraction would take the form:

$coloring(a_1),$

$multi^{\vee}(A : allsafeA(g_{1,\mu,2}, g_{1,\mu,3}, g_{1,\mu,1}, a_{1,\mu,1})$

$\vee\ B : allsafeB(g_{1,\mu,2}, g_{1,\mu,3}, g_{1,\mu,1}, a_{1,\mu,1}),$

$FirstA,$

$FirstB,$

$ConsecutiveAA,$

$ConsecutiveAB,$

$ConsecutiveBA,$

$ConsecutiveBB,$

$LastA,$

$LastB),$

$allsafeA(g_8, g_9, [g_{10}|g_1], [g_{11}|a_1]),$

$safe([g_{10}|g_1], [g_{11}|a_1]).$

Here, *FirstA* represents the bindings between the first represented subconjunction and the context, if the first represented subconjunction matches the first pattern. *FirstB* represents the bindings between the first represented subconjunction and the context, if the first represented subconjunction matches the second pattern. The conjunctions of equalities *LastA* and *LastB* are analogous. The different conjunctions of constraints *ConsecutiveXY* detail how variables of a subconjunction matching pattern X are aliased with those of the following subconjunction, which matches pattern Y. When unfolding is applied to $multi^{\vee}$, both the possibility that the unfolded conjunction of atoms

is of type A and the possibility that it is of type B must be considered, which leads to very long analyses. Therefore, we have only performed a full analysis of the graph coloring variant with this abstraction. A complete program synthesis is provided in appendix F.

4.4 Beyond depth-k abstraction

In some cases, depth-k abstraction is a useful technique which allows a non-terminating analysis to be terminated without losing the desired control flow. For instance, consider a variant of the primes program from page 51, but using Peano arithmetic instead of built-in arithmetic and where zero is included in the abstract domain. Such a variant is provided in appendix A. Then, without widening, all *filter* atoms would have a different first argument and the *multi* abstraction would be unable to represent them. Depth-k abstraction would be an obvious solution in this case, because the degree of instantiation of the filters' first argument does not affect the control flow. There are programs, however, where a more refined approach is needed to restrict the depth of terms. Consider yet another variant of graph coloring, given in listing 4.4. This program produces the same answers as the original graph coloring program and uses the same declarative logic, but it avoids generating arbitrarily long conjunctions of atoms. Therefore, a *multi* abstraction is not needed for the analysis. However, it does this by generating arbitrarily deeply nested terms, which can be regarded as the opposite side of the same coin as the *multi* abstraction in the original graph coloring program. Therefore, some form of generalization over these arbitrarily deep terms is necessary.

```
test(N,C,N1,C1) :-
  bi_edge(N,N1), C \== C1 ; N \== N1, bi_noedge(N,N1).

safe_coloring(Ns,Cs) :-
  coloring(Cs),
  safe(Ns,Cs).

coloring([]).
coloring([C|Cs]) :-
  color(C),
  coloring(Cs).
safe(Ns,Cs) :-
  safe_(Ns,Cs,done,_).
```

```
% K1: continuation before processing
% K2: continuation after processing
safe_([],[],K1,K2) :- process(K1,done,K2).
safe_([_],[C1],K1,K2) :- process(K1,done,K2).
safe_([N1,N2|Ns],[C1,C2|Cs],K1,K2) :-
  process(frame(N1,C1,[N2|Ns],[C2|Cs],K1),done,K2),
  safe_([N2|Ns],[C2|Cs],K2,_).

process(done,Acc,Acc).
process(frame(N1,C1,[],[],K),Acc,Res) :-
  process(K,frame(N1,C1,[],[],Acc),Res).
process(frame(N1,C1,[N2|Ns],[C2|Cs],K),Acc,Res) :-
  test(N1,C1,N2,C2),
  process(K,frame(N1,C1,Ns,Cs,Acc),Res).
```

Listing 4.4: Continuation passing graph coloring variant

First, consider the effect of applying depth-k abstraction, as shown in figure 4.3. Here, the abstract analysis is blocked, because it is unclear whether the *process*/3 atom in the leaf can be unfolded further. Applying the third clause, for instance, would introduce a *test*/4 call which cannot be safely fully evaluated. Indeed, this call cannot become sufficiently instantiated, because aliasing with the generator predicate has been lost. Therefore, the analysis cannot be completed. More generally, depth-k abstraction is simply too crude for this problem.

A solution to this problem is to use a construct similar to *multi* which does not represent arbitrarily long conjunctions with a fixed structure, but arbitrarily deeply nested terms with a fixed structure. This is shown in figure 4.4. This construct is called *complex*. Instead of a description of the sequence of atoms which it represents, it contains a grammatical description of the terms which it represents. The idea of case splitting is retained, but the case split is based on term depth rather than conjunction length. This is shown in figures 4.5 and 4.6.

Given the use case for the $multi^{\vee}$ abstraction, it is easy to imagine an abstraction $complex^{\vee}$, as well. More generally, a grammar representing abstract conjunctions of arbitrary length and containing terms of arbitrary depth would abstract all the abstractions proposed here. Viewing the *multi* and related abstractions as grammars also opens up new perspectives for applying established results. As an example, techniques from the field of grammar induction could theoretically be applied to automatically discover applicable abstractions.

Figure 4.3: Adverse effect of depth-k abstraction in graph coloring variant

$$
\begin{array}{l}
\quad\quad\quad

Chapter 5

CHR synthesis

In the current chapter, we focus on the synthesis of the transformed program. Instead of synthesizing a new logic program, we synthesize a CHR(Prolog) program which mimics the coroutining program. The synthesis to CHR yields programs containing only simplification rules, which are amenable to certain static analysis techniques. Moreover, this work can lead to more portable programs, as CHR implementations can be embedded in other languages than Prolog. The programs shown here can be ported to those, as well, providing counterparts to any built-ins used are available. This work was previously described in [66], but the presentation has been streamlined and a different running example is used.

5.1 Motivation

The synthesis step described in chapter 3 has some disadvantages. First, the analysis phase analyzes the computational behavior of conjunctions of atoms. Our instance of abstract conjunctive partial deduction builds abstract derivation trees with conjunctions of (abstract) atoms in their roots and leaves. To synthesize the branches of such trees into new clauses, renaming schemes are necessary to convert the conjunctions into atoms, so that they are then allowed to occur in the head of the Prolog clause. This complexity could be avoided by compiling to a language with multi-headed rules. CHR [28] is one such language.

Second, the synthesis of a branch in the derivation tree compiles how some

state in the computation is resolved to produce a new state. As Prolog does not have a notion of "state", we need to encode entire states, both root states and those represented in the leaves, into new predicates. This produces very complex predicates which obfuscate the actual computations encoded by the state transitions. Their use could be avoided if we compiled to a language that keeps track of the current state for us, and only requires us to express what changes with respect to the root state. Again, CHR has a store that supports precisely this.

In this chapter, we show how a synthesis to CHR instead of Prolog is possible. The alternative synthesis has a number of advantages, including added insight into the computational process and more readable transformed code. While we will assume the compilation target to be CHR embedded in SWI-Prolog for the sake of illustration, this technique also facilitates porting logic programs to CHR systems in other languages, such as JCHR for Java or HaskellCHR for Haskell.

5.2 The refined operational semantics of CHR

While CHR has various interesting semantics, implementations of CHR almost exclusively implement the "refined operational semantics" [25]. In this section, we provide a brief tutorial-style introduction to the refined operational semantics summarizing parts of [27], so that the synthesized programs can be understood. For a more complete explanation, we refer to [25] or [27].

5.2.1 Basic concepts

CHR is an embedded, committed-choice constraint programmming language. A CHR program consists of *rules* which operate on a *store*, which is a multiset of *constraints*. These constraints are atomic formulae whose interpretation depends on the program. The store can be regarded as a program state and the program terminates when no further rules can be applied. In CHR, there are two basic types of rules: simplification rules and propagation rules.[1] Simplification rules are meant to replace existing constraints with simpler constraints, while preserving logical equivalence. They take the form $H \Leftrightarrow C|B$, where H, the *head*, is a conjunction of constraints, C is a *guard*, i.e. a condition required for the rule to be applicable and B, the *body*, is also a conjunction of constraints. For instance, a program can encode the antisymmetry property

[1] We disregard simpagation rules, which are essentially syntactic sugar for certain simplification rules.

with the simplification rule `X >= Y, Y >= X <=> X = Y`. Propagation rules add additional constraints, which are logically redundant, but which can directly express information sought after by the user or allow further simplification. For instance, the transitivity property can be encoded as `X >= Y, Y >= Z ==> X >= Z`. These rules are a very simple, but very expressive mechanism and allow a very high-level specification of algorithms.

Constraints are divided into two disjoint sets: *built-in* constraints and *user-defined* constraints. In general terms, the semantics of built-in constraints is defined by a consistent first-order constraint theory CT. In a CHR program embedded in a Prolog environment (also known as a CHR(Prolog)) program, built-in constraints are Prolog predicates. These need not be built-in predicates, but can also be user-defined.[2] For instance, $=/2$ and $format/1$ in `start <=> X = "hello world", format(X)` operate as a Prolog programmer would expect, provided that the constraint $start/0$ is placed in the constraint store. User-defined constraints are defined by the rules in the CHR program.

5.2.2 Operational semantics

In order for a CHR program to run, it must be supplied with some initial state, i.e. one or several constraints. Rules will then be applied until no more rules are applicable or until failure occurs.

The operational semantics is given by a transition system, represented in figure 5.1. Let P be a CHR program. The transition relation $\longmapsto$ is defined using one computation step or transition for each type of rule. States (or "stores") are goals, i.e. conjunctions of built-in and user-defined constraints. In the figure, upper case letters stand for conjunctions of constraints, CT is the constraint theory for built-in constraints, and G_{bi} denotes the built-in constraints of G, which is part of the current state.

Starting from the initial state, rules are applied exhaustively until a fixpoint is reached. Trivial non-termination of the "Propagate" step is avoided by applying a propagation rule at most once to the same set of constraints. A rule is applicable if its head constraints are matched by constraints in the current goal and if, under the matching, the guard is implied by the built-in constraints in the goal. Once a rule is applied, its application cannot be undone. That is, CHR is a *committed-choice* language which does not backtrack.[3]

[2] Admittedly, this makes the terminology a bit confusing.

[3] It is worth pointing out that *Prolog* builtins can still be backtracked over. There is no backtracking *over the application of CHR rules*.

Simplify

If $(H \Leftrightarrow C \mid B)$ is a fresh variant with variables $\bar{x}$ of a rule in P
and $CT \models \forall\,(G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then $(H' \wedge G) \mapsto_P^{\mathbf{Simplify}} (B \wedge G \wedge H{=}H' \wedge C)$

Propagate

If $(H \Rightarrow C \mid B)$ is a fresh variant with variables $\bar{x}$ of a rule in P
and $CT \models \forall\,(G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then $(H' \wedge G) \mapsto_P^{\mathbf{Propagate}} (H' \wedge B \wedge G \wedge H{=}H' \wedge C)$

Figure 5.1: Operational semantics of CHR

More specifically, a fresh variant of a rule is applicable to a state $H' \wedge G$ if H' matches the rule's head H and if the rule's guard C is implied by the builtin constraints appearing in G. Matching, also known as *one-way unification*, succeeds if H' is an instance of H. A computation of a goal G is a sequence $S_0, S_1, \ldots$ of states with $S_i \longmapsto S_{i+1}$, beginning with initial state $S_0 = G$ and ending in a final state or not terminating. A final state is one in which no rules are applicable or where the built-in constraints are inconsistent.

5.2.3 Refined operational semantics

The *refined* operational semantics is a concretization of the operational semantics which removes two important sources of non-determinism: the order in which constraints in a query are processed and the order in which rules are applied. Under the refined operational semantics, as in Prolog, queries are executed from left to right and rules are applied in textual order. With these conventions, a constraint in a query can be understood as a procedure which traverses the rules of a program in textual order. A single constraint triggers this rule traversal and it is called the *active* constraint. This constraint can be seen as the most recently added user-defined constraint. When it is added, it is pushed to the top of an *execution stack*, i.e. a stack of constraints for which rules can still be tried. When the active constraint matches a head constraint of a rule, *partner constraints* are located in the *constraint store*, which contains *inactive* constraints. Once these constraints are found, the guard (if any) of the rule in question is checked. If the active constraint has not been removed (due to simplification) after trying all rules, it is popped off the execution stack and suspended in the constraint store.

A constraint from the constraint store is reconsidered when a built-in constraint causes its variables to become more instantiated. This is because new rules may

become applicable.

Note that the refined operational semantics is not entirely free of non-determinism: the order in which constraints from the store are woken is not fixed. This means that, if two constraints share a variable and this variable becomes more instantiated, either constraint may become active first and affect which rule is applied next. This will not prove an issue for our technique, as a precaution — the *lock*/0 constraint — is taken to guarantee the correct control flow.

5.3 Permutation sort in CHR

We will again illustrate this technique using permutation sort, as defined in listing 2.1 on page 10 with the same abstract instantiation-based selection rule as before. The technique generates a CHR program which, under the refined operational semantics of CHR produces equivalent answers to the original program. It also evaluates constraints in the same order in which the abstract analysis evaluates corresponding abstract atoms.

The procedure for generating the CHR program is the following: First, the non-built-in predicates (in the terminology of logic programming) of the initial program are partitioned into two sets: those that will be CHR constraints in the synthesized program and those that will be built-in constraints. The former will be encoded using CHR rules, the latter will remain Prolog predicates, which may or may not be user-defined. The partitioning criterion is easy: Predicates that are fully abstractly evaluated in the abstract trees are built-in constraints, those that are abstractly unfolded in the trees become CHR constraints. We require the abstract analysis to be consistent in the choice between these two options. That is, this procedure requires either all instances of a predicate to be fully evaluated in the analysis, or none at all. For permutation sort, all three user-defined predicates, *permsort*/2, *perm*/2 and *ord*/1, become CHR constraints.

The synthesis of the CHR rules is based on the same concrete derivation trees as before. For permutation sort, these start on page 30. For every non-failing branch in a derivation tree, we synthesize one (possibly guarded) simplification rule. The head of this rule consists of all atoms in the root of the tree that are unfolded in the branch. The guard consists of "passive constraints" *as referred to in constraint logic programming literature* [1]. That is, restrictions on the values of variables which can be evaluated (and thus fail or succeed) but whose evaluation does not restrict the domain of the involved variables. The body of the rule consists of residual atoms and unification steps along the branch which

cannot be expressed in the guard, as well as all the atoms in the leaf of the branch which are newly introduced with respect to the root. In many cases, passive constraints expressed in the guard can be moved into the head, as we have done in our syntheses. In permutation sort, this gives the rules in listing 5.1. For this particular program, all orderings of the rules lead to an equivalent program.

```
permsort([],X) <=> X = [].
permsort([A|B],Y) <=>
  Y = [C|D],
  select(C,[A|B],E),
  perm(E,D),
  ord([C|D]).
perm([],D), ord([C|D]) <=> D = [].
perm([F|G],D), ord([C|D]) <=>
  D = [H|I],
  select(H,[F|G],J),
  C =< H,
  perm(J,I),
  ord([H|I]).
```

Listing 5.1: Permutation sort in CHR

Comparing listing 3.1 (page 31) with listing 5.1, permutation sort already allows us to draw some conclusions when comparing the two synthesis approaches. First, there is no more need to rename conjunctions of atoms to a new predicate, in order to obtain the clause format.

The second improvement is that, where the Prolog synthesis needs to represent transitions of a complete goal (conjunction) in the root of a tree to the complete goal in the leaf of the tree, the CHR synthesis only needs to focus on the unfolded atoms of the root. For permutation sort, however, the Prolog and CHR syntheses are very similar, because all constraints are rewritten simultaneously. Our second example will highlight both advantages.

5.4 Incorporating the *multi* abstraction: confused queens

In this section, we study the transformation to CHR for a more representative example which requires the *multi* abstraction: the confused queens problem.

We use this problem because it is well-suited to highlighting a particular pitfall for the transformation technique. We will deal with that pitfall in section 5.5.

The confused queens problem is a variation on the well-known N-queens problem. Like in N-queens, the goal is to arrange N queens on a chess board of size $N \times N$. Unlike in the N-queens problem, for confused queens, every pair of queens should be on the same row or on the same diagonal. Each piece is still placed in a different column, which is implicit in the use of a list data structure. Prolog code for the program (using a standard computation rule) is shown in Listing 5.2.

```
cqueens(N,D) :-
  genlist(N,L),
  draw(N,L,D),
  confused(D).

genlist(N,L) :-
  N >= 1,
  genlist_acc(N,[],L).
genlist_acc(N,Acc,L) :-
  N > 1,
  Nmin is N-1,
  genlist_acc(Nmin,[N|Acc],L).
genlist_acc(1,Acc,[1|Acc]).

draw(0,_,[]).
draw(N,L,[E|R]) :-
  N > 0,
  Nmin is N - 1,
  member(E,L),
  draw(Nmin,L,R).
confused([]).
confused([_X]).
confused([A,B|C]) :-
  attack_all(A,1,[B|C]),
  confused([B|C]).

attack_all(_,_,[]).
attack_all(A,Off,[B|C]) :-
  Offplus is Off + 1,
  attack(A,Off,B),
  attack_all(A,Offplus,C).
```

```
attack(A,_,A).
attack(A,Off,B) :-
  Diff is A - B,
  abs(Diff,Off).
```

Listing 5.2: Confused queens

We analyze the program for the abstract goal $cqueens(g_1, a_1)$. A sufficiently detailed generating set *Preprior* for the rule is:

- $draw(g_1, g_2, a_1) \xrightarrow{p} confused([g_1|a_1])$
- $draw(g_1, g_2, a_1) \xrightarrow{p} attack_all(g_1, g_2, a_1)$
- $attack_all(g_1, g_2, []) \xrightarrow{p} confused([g_1])$
- $attack_all(g_1, g_2, [g_3|a_1]) \xrightarrow{p} confused([g_1, g_2|a_1])$
- $confused([g_1, g_2|a_1]) \xrightarrow{p} draw(g_1, g_2, a_1)$

The trees in figures 5.2, 5.3 and 5.4 (starting on page 104) show the first, second and third abstract derivation trees built during the analysis for $cqueens(g_1, a_1)$.[4] The trees in figures 5.5, 5.6 and 5.7 show their concrete counterparts.

The remaining abstract trees for an $\mathcal{A}$-closed analysis for $cqueens(g_1, a_1)$ and the corresponding concrete trees can be produced by our implementation. After generating the concrete trees, resultants for the branches can be computed. The resultants (i.e. a logic program, not a CHR program) for figures 5.5 through 5.7 are as follows:

- $cqueens(0, [])$.
- $cqueens(A, [D|E]) \leftarrow$
  $genlist(A, C), A > 0, F \text{ is } A - 1, member(D, C),$
  $draw(F, C, E), confused([D|E])$.
- $draw(0, X, []), confused([Y])$.
- $draw(A, B, [E|F]), confused([D, E|F]) \leftarrow$
  $A \geq 0, G \text{ is } A - 1, member(E, B), draw(G, B, F),$
  $multi([attack_all(D, 1, [E|F])]), confused([E|F])$.

[4] The sets of constraints of *multi* abstractions have been omitted where they are immediately obvious.

- $draw(A, B, C), multi([attack_all(D, E, [F|C])]), confused([F|C]) \leftarrow$
  H is $E+1, attack(D, E, F), draw(A, B, C), multi([attack_all(D, H, C)|I]),$
  $confused([F|C])$
- $draw(A, B, C), multi([attack_all(D, E, [F|C]), attack_all(H, I, [F|C])|J)]),$
  $confused([F|C]) \leftarrow$
  K is $E+1, attack(D, E, F), draw(A, B, C), multi([attack_all(D, K, C)|L]),$
  $multi([attack_all(H, I, [F|C])|J]), confused([F|C])$

Finally, when all resultants of all trees have been computed, renaming transformations are applied to obtain Prolog clauses. The resulting clauses corresponding to the resultants for the first three trees are shown in Listing 5.3.

```
cqueens(0,[]).
cqueens(A,[D|E]) :-
   genlist(A,C),
   A > 0,
   F is A - 1,
   member(D,C),
   a(draw(F,C,E),
     confused([D|E])).
a(draw(0,_,[]),confused([_])).
a(draw(A,B,[E|F]),
  confused([D,E|F])) :-
  A >= 0,
  G is A - 1,
  member(E,B),
  b(draw(G,B,F),
    multi([attack_all(D,1,[E|F])]),
    confused([E|F])).
b(draw(A,B,C),
  multi([attack_all(D,E,[F|C])]),
  confused([F|C])) :-
  H is E + 1,
  attack(D,E,F),
  c(draw(A,B,C),
    multi([attack_all(D,H,C)]),
    confused([F|C])).
b(draw(A,B,C),
  multi([attack_all(D,E,[F|C]),
  attack_all(H,I,[F|C])|J]),
  confused([F|C])) :-
```

```
  K is E + 1,
  attack(D,E,F),
  d(draw(A,B,C),
    multi([attack_all(D,K,C)]),
    multi([attack_all(H,I,[F|C])|J]),
    confused([F|C])).
```

Listing 5.3: Partial Prolog synthesis of confused queens

Note that these clauses become rather complex. They explicitly describe every possible transition from one concrete goal corresponding to an element of $\mathcal{A}$ to another. This obscures the actual logic behind the transition, i.e. the *change* that occurs between two states. We are now in a position to show that the synthesis to CHR expresses this logic more clearly.

From the first branch in figure 5.2, we generate:

```
cqueens(0,B) <=> genlist(0,C), B = [], lock.                    (1)
```

From the second branch:

```
cqueens(A, B) <=> A > 0 | B = [E|F], genlist(A,C), D is A-1,    (2)
  member(E,C), draw(D,C,F), confused([E|F]), lock.
```

The purpose of the *lock*/0 constraint is to prevent a transition from firing at certain points during program execution. The refined operational semantics of CHR dictates that atoms are evaluated and constraints are added to the store from left to right. This means that a rule could fire *while* another rule is being applied, raising an issue for our transformation: rules represent transitions between instances of abstract states in $\mathcal{A}$. Thus, they should only fire when the contents of the store are an instance of an abstract state in $\mathcal{A}$. The *lock*/0 ensures that new rules only fire when an entire new state is reached, because it is always required for a state transition and because it is always added last.

The left branch of the tree in figure 5.3 yields:

```
draw(0,B,C), confused([D|C]), lock <=> C = [], lock.            (3)
```

The right branch yields:

```
draw(A,B,C), confused([D|C]), lock <=> A > 0 | C = [E|F],       (4)
  G is A-1, member(E,B), draw(G,B,F), attack_all(D,1,[E|F]),
  confused([E|F]), lock.
```

Note that, unlike in the Prolog synthesis, we do not introduce a representation of *multi* in the body of the CHR rule. We rely on the CHR store to accumulate multiple *attack_all*/3 constraints.

From the left branch of the tree in figure 5.4, we generate the following rule:

```
attack_all(D,E,[F|C]), lock <=>                                    (5)
  H is E + 1, attack(D,E,F), attack_all(D,H,C), lock.
```

Again, we do not synthesize the unfolding of *multi*. We only consider the unfolding of an $attack_all/3$ constraint which is present in the store.

For the right branch of the tree in figure 5.4, the generated rule is identical. There are two leaf nodes in the abstract tree, but this is only due to the case split on *multi*. The unfoldings and full evaluations of abstract atoms required to go from the root to a leaf node are identical for both branches. In the CHR synthesis, there is no case split: a specific number of concrete instances of $attack_all(g_i, g_j, [g_4|a_1])$ is in the constraint store. That is, the two branches of the concrete counterpart to figure 5.4 are mutually exclusive. Therefore, it is not necessary to add an additional rule to the synthesized program. In the Prolog synthesis, we do get a different synthesized clause for each of these branches, because the context of the derivation is different in each branch and the Prolog synthesis encodes the context in its new predicates. Because rules do not need to be repeated and because they only express the change between states, the alternative synthesis can be significantly more parsimonious than the Prolog synthesis. In fact, several more branches in the full program analysis generate rule (5).

As a result, only the following new rule is synthesized from the remaining trees:

```
attack_all(X,Y,[]), lock <=> lock.                                 (6)
```

Finally, a housekeeping rule is required:

```
lock <=> true.                                                     (7)
```

We always add rule (7) at the end of a synthesized program so that the $lock/0$ constraint is discarded in an otherwise empty state.

After synthesizing all the rules, we order rules in any pair (r_1, r_2) with respect to each other by constructing a partial order $\xrightarrow{r}$ which dictates the textual order:

1. The housekeeping rule is preceded by every synthesized rule, as the former removes the mechanism for ensuring valid state transitions.
2. If there is no root of an analysis tree containing abstractions of the heads of both r_1 and r_2 (ignoring the housekeeping constraint), $r_1 \not\xrightarrow{r} r_2$ and $r_2 \not\xrightarrow{r} r_1$. This is because at most one of the two rules can fire in any given state.

3. If an abstract atom in the root of the tree from which r_1 was synthesized takes precedence over the abstract atoms instantiated in the head of r_2, $r_1 \xrightarrow{r} r_2$. Note that precedence need not be due to $\xrightarrow{p}$: it is also possible that the abstract atom in the root of the first tree always precedes those instantiated in r_2 syntactically.

The synthesis is successful and the program works as intended for the expected queries. It is shown in listing 5.4. The complete Prolog program, which is provided in appendix A, contains nearly twice as many lines, and the lines themselves are significantly longer.

```
:- use_module(library(chr)).
:- chr_constraint cqueens/2, draw/3, confused/1,
                  attack_all/3, lock/0.
genlist(N,L) :- N >= 1, genlist_acc(N,[],L).
genlist_acc(N,Acc,L) :-
  N > 1, Nmin is N-1, genlist_acc(Nmin,[N|Acc],L).
genlist_acc(N,Acc,[1|Acc]) :- N is 1.

attack(A,_,A).
attack(A,Offset,B) :- Diff is A - B, abs(Diff,Offset).

cqueens(0, B) <=> genlist(0,C), B = [].
cqueens(A, B) <=>
  A > 0 |
  B = [E|F],
  genlist(A,C),
  D is A-1,
  member(E,C),
  draw(D,C,F),
  confused([E|F]),
  lock.

attack_all(A,Of,[B|C]), lock <=>
  Of1 is Of + 1, attack(A,Of,B), attack_all(A, Of1, C), lock.
attack_all(X,Y,[]), lock <=> lock.

draw(0,B,C), confused([D|C]), lock <=> C = [], lock.
draw(A,B,C), confused([D|C]), lock <=>
  A > 0 |
  C = [E|F],
```

```
  G is A-1,
  member(E,B),
  draw(G,B,F),
  attack_all(D,1,[E|F]),
  confused([E|F]),
  lock.

lock <=> true.
```

Listing 5.4: CHR synthesis of confused queens, with pitfall

Note that the guard conditions `A > 0` can be moved into the rule bodies, though this further restricts the rule order. Also note that it is possible to write a program for which the rules dictating $\xrightarrow{r}$ do not apply, e.g. by producing permuted variants of a single abstract conjunction without strictly specifying $\xrightarrow{p}$. If this poses a problem for a program, the *lock* constraint can be parameterized with a state identifier, though the resulting program will be less compact.

As a final note, because the resulting synthesis expresses only the changes between consecutive states, it can be said to be more procedural in nature than the Prolog synthesis. This suggests that similar syntheses for imperative languages are feasible. Appendix F contains a proof-of-concept.

5.5 A more refined CHR synthesis

The CHR program synthesized in section 5.4 for confused queens compiles the coroutines correctly for all top level queries $cqueens(X, Y)$, where X is an integer and Y a free variable. Unfortunately, there are instances of the abstract atom $cqueens(g_1, a_1)$ which show that the transformation is unsound. Furthermore, while our synthesis of permutation sort is sound, a different — but valid — abstract analysis would also have led to an unsound synthesis.

The lack of soundness can be demonstrated as follows: for queries such as $cqueens(4, [X, Y, Z, T])$, where we partially instantiate the output, the CHR program ends with a run-time error. The reason is that we rely on the CHR matching mechanism to identify whether a constraint $attack_all/3$ has a sufficiently instantiated third argument to activate rule (5). If the output argument in *cqueens* is a variable, then a sufficiently instantiated third argument will have a ground value for its first list member. But if the output argument is given the required list structure from the start, then CHR is unable to distinguish between *attack_all* atoms that have a value ready for use, and those that have the structure, but not the value of the first member.

A trivial way to solve this is to add a guard condition $ground(B)$ to rule (5). This will provide a correct synthesis for all intended queries. However, this defeats the purpose of the transformation, as we are then explicitly encoding a kind of "delay" in the CHR program, while the aim is to eliminate the delays or freezes from the Prolog program.

Fortunately, the abstract analysis provides us with the knowledge of when an *attack_all/3* atom's first list element will be instantiated. This information can be encoded into the constraints, so that it is available to the matching mechanism. In the conjunctions in the analysis, there are two *multi*-constructs. One with *attack_all/3* atoms that are sufficiently instantiated, the other with such atoms that are not yet sufficiently instantiated. In our synthesis, we can take advantage of this information and add guarantees about the instantiation level of the final argument to each *attack_all*/3, making it an *attack_all/4* constraint. More generally, for every CHR constraint (except for the *lock*/0), we add an additional list argument. For every element in the original list of arguments, we add a list element representing the guaranteed instantiation of that element to the new argument. For instance, if an *attack_all* constraint is sufficiently instantiated for rewriting, its added fourth argument will be $[g, g, [g|a]]$.

This mainly affects the rules (4) and (5), although it also requires trivial modifications in other rules, as the arity of coroutining constraints increases everywhere and as the instantiation argument must be kept consistent with the abstract analysis. In rule (5), the newly generated *attack_all* is not sufficiently instantiated. So the rule becomes:

```
lock, attack_all(D,E,[F|C],[g,g,[g|a]]) <=>
  H is E + 1, attack(D,E,F), attack_all(D,E,C,[g,g,a]), lock.
```

In rule (4) the newly generated *attack_all* is sufficiently instantiated. The rule becomes:

```
draw(A,B,C), confused([D|C],[g|a]), lock <=> A > 0 | C = [E|F],
  G is A-1, member(E,B), draw(G,B,F,[g,g,a]),
  attack_all(D,1,[E|F],[g,g,[g|a]]), confused([E|F],[g|a]), lock.
```

However, in the abstract analysis we see that in the branches corresponding to this rule, all *attack_all*/3 atoms that were not sufficiently instantiated become sufficiently instantiated. If a single specific constraint was modified, it could be rewritten directly. Due to the use of *multi*, however, the instantiation information cannot be updated for all affected constraints directly. Therefore, we introduce a new constraint, *rename*/0, in the body of the rule, resulting in:

```
draw(A,B,C), confused([D|C],[g|a]), lock <=> A > 0 | C = [E|F],
  G is A-1, member(E,B), draw(G,B,F,[g,g,a]), rename,
  attack_all(D,1,[E|F],[g,g,[g|a]]), confused([E|F],[g|a]), lock.
```

In addition, we add the following rules for *rename/0*:

```
lock, rename, attack_all(A,B,C,[g,g,a]) <=>
  attack_all(A,B,C,[g,g,[g|a]]), rename,lock.
lock, rename <=> lock.
```

This will transform all *attack_all/4* constraints labelled as insufficiently instantiated into constraints labelled as sufficiently instantiated. The resulting program is in given in Listing 5.5.

```
:- use_module(library(chr)).
:- chr_constraint cqueens/3, draw/4, confused/2, attack_all/4.
:- chr_constraint lock/0, rename/0.
genlist(N,L) :- N >= 1, genlist_acc(N,[],L).
genlist_acc(N,Acc,L) :-
  N > 1,
  Nmin is N-1,
  genlist_acc(Nmin,[N|Acc],L).
genlist_acc(N,Acc,[1|Acc]) :- N is 1.

attack(A,_,A).
attack(A,Offset,B) :- Diff is A - B, abs(Diff,Offset).

rename, attack_all(A,B,C,[g,g,a]), lock <=>
  attack_all(A,B,C,[g,g,[g|a]]), rename, lock.
rename, lock <=> lock.

cqueens(0,B,[g,a]) <=> genlist(0,C), B = [].
cqueens(A,B,[g,a]) <=>
  A > 0 |
  B = [E|F],
  genlist(A,C),
  D is A-1,
  member(E,C),
  draw(D,C,F,[g,g,a]),
  confused([E|F],[g|a]),
  lock.
```

```
attack_all(A,Off,[B|C],[g,g,[g|a]]), lock <=>
  Off1 is Off + 1, attack(A,Off,B),
  attack_all(A,Off1,C,[g,g,a]), lock.
attack_all(X,Y,[],_), lock <=> lock.

draw(0,B,C,[g,g,a]), confused([D|C],[g|a]), lock <=> C = [], lock.
draw(A,B,C,[g,g,a]), confused([D|C],[g|a]), lock <=>
  A > 0 |
  C = [E|F],
  G is A-1,
  member(E,B),
  draw(G,B,F,[g,g,a]),
  rename,
  attack_all(D,1,[E|F],[g,g,[g|a]]),
  confused([E|F],[g|a]),
  lock.

lock <=> true.
```

Listing 5.5: CHR synthesis of confused queens, without pitfall

This program correctly compiles the coroutines for any query in the concretization of $cqueens(g_1, a_1)$. The syntheses of this and related programs contain only simplification rules, which is a desirable property for the application of ranking-based techniques [29] [30] for analysis of CHR programs.

5.6 Conclusions

We have presented a new application of compiling control, which allows coroutining logic programs to be compiled to concise CHR programs with a simple structure. One could argue that a Prolog program with delays or *freeze* predicates is also a concise representation. However, the CHR rules explicitly describe the procedural transitions, which are much harder to see in a Prolog program with delays. Due to the straightforward implementation of these transitions using simplification rules, the synthesized CHR programs provide a suitable target for further program analysis based on tight rankings [29].

The resulting CHR programs are slower than their Prolog counterparts — benchmarks are provided in appendix B — but they are far more readable: They express only the changes that take place between two states in $\mathcal{A}$, they

do not require conjunctions to be renamed using meta-predicates, and they do not require a concrete representation of the *multi* abstraction. The procedure can also serve as a stepping stone for syntheses in various other languages with better runtime performance.

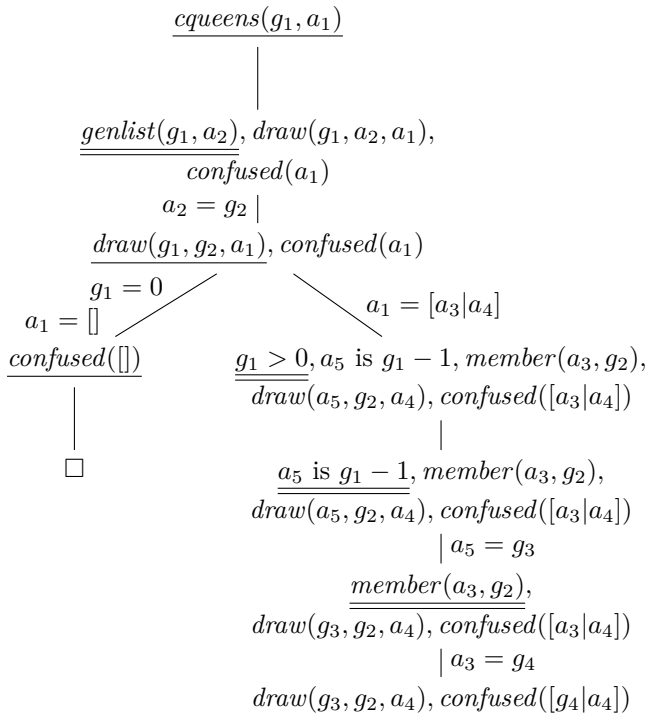


Figure 5.2: First abstract tree for confused queens

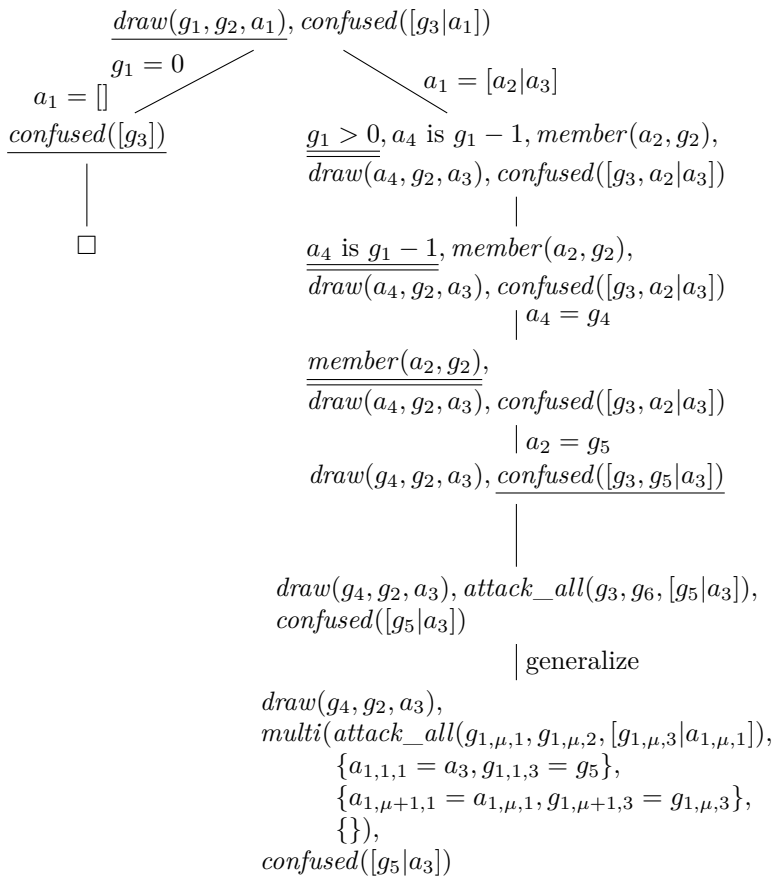


Figure 5.3: Second abstract tree for confused queens

Figure 5.4: Third abstract tree for confused queens. To reduce the size of the figure, constraints are only shown if they are not clear from parent nodes.

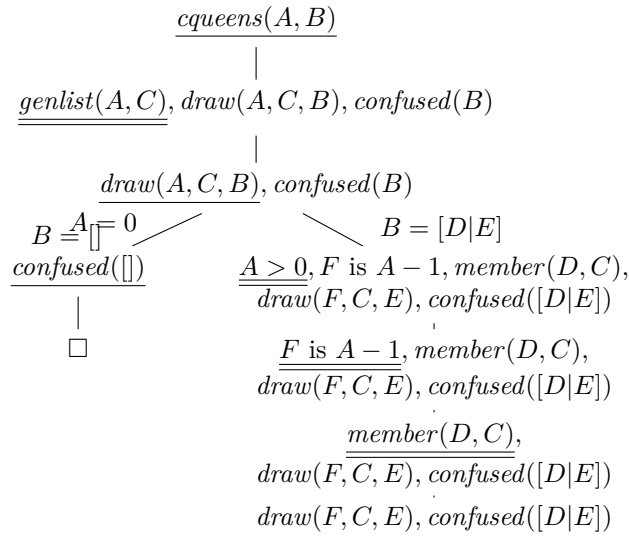


Figure 5.5: First concrete tree for confused queens

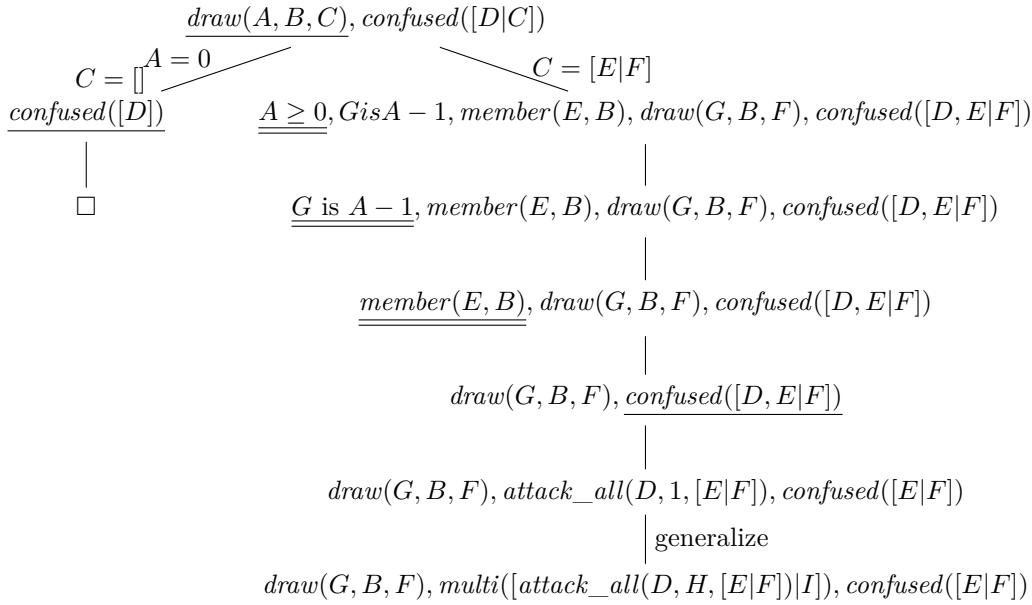


Figure 5.6: Second concrete tree for confused queens

Figure 5.7: Third concrete tree for confused queens.

Chapter 6

Compiling Control as Offline Partial Deduction

6.1 Introduction

In the current chapter, we propose yet another perspective on the synthesis phase of compiling control. We show that a synthesis highly similar to the one obtained using the approach discussed in chapter 3 can be obtained by applying the first Futamura projection, in which program specialization is applied to an interpreter and an input program. Such an approach to compiling control has been speculated upon in the past. The idea of implementing compiling control as an instance of the first Futamura projection was first put forward in [32]. The concept is illustrated in [32] through the partial evaluation of a coroutining meta-interpreter for the sameleaves program, but an automatic procedure is not provided. Here, we provide the tools to reproduce the results and to apply them to new examples.

This synthesis technique is not an instance of the abstract conjunctive partial deduction framework, but rather a standard offline partial deduction [46]. This is encouraging because it implies that no changes to the very general and rather complex abstract conjunctive partial deduction framework are needed to accommodate compiling control. It also opens up a path towards abstract domains which are not closed under substitution, which would allow more challenging coroutining programs and more general queries to be analyzed. Based on our experiments, the approach does not incur any loss of flexibility with respect to the "interpreter-free" approach detailed in chapter 3, so more general

abstract domains would strictly increase the expressive power of compiling control.

The idea of applying the first Futamura projection to obtain a more structured representation of control flow can also be found in [36], where Java bytecode is decompiled to Prolog. The notion of modelling and analyzing the execution of a (PIC) program as a logic program and partially evaluating that can be found in [38]. This is akin to what we do, though we are able to abstractly analyze the program to be executed itself, which is already a logic program, and partially evaluate the interpreter. Other examples of specialization of logic program interpreters, also using the Logen system, are provided in [47].

We will first give a brief introduction to the first Futamura projection and to offline partial deduction in section 6.2. We will then show a simple meta-interpreter which, using information obtained from the analysis, can be configured to implement the desired abstract instantiation-based selection rule for permutation sort. Once the basic idea has been illustrated, we will show the extensions to the meta-interpreter required to run programs requiring a *multi* abstraction, notably the prime sieve, in a satisfactory way in section 6.4. In section 6.5, we will show how the meta-interpreter is specialized using the Logen system for offline partial deduction. Finally, in section 6.6, we will show that the obtained specialization is indeed equivalent to the "interpreter-free" synthesis using empirical evidence. This chapter is a slightly modified version of work which was previously published [67].

6.2 Preliminaries

6.2.1 The first Futamura projection

In [31], Futamura showed that partially evaluating an interpreter for a language l_1 (written in language l_2) for a "source program" (in l_1) yields an "object program" (in l_2) with the semantics of the source program, as run by the interpreter. That is, partially evaluating an interpreter for a source program is an act of compiling.

This is expressed by the following equation: $int(s, r) = \beta(int, s)(r)$. Here, int is the function encoded by the interpreter. Its arguments s and r are the "static" and "runtime" inputs, i.e. inputs which remain constant and which may vary between executions, respectively. In the setting of partial evaluation of an interpreter, the "static" input is the program to be interpreted, whereas the "runtime" input is the input to the interpreted program. The function β is the partial evaluation function: it specializes its first argument for the static

input, which is supplied as the second argument. In this setting, this produces a program which takes the runtime inputs, so r, and behaves as the interpreter would when also given s. Therefore $\beta(int, s)$ is the compiled version of the source program which Futamura calls the "object program". The observation that a compiled program can be obtained through specialization of an interpreter is known as the first Futamura projection [42].

The first Futamura projection can be applied in various contexts. Notably, [33] and [39] point out the possibility of partially evaluating non-standard meta-interpreters and explicitly refer to compiling control as a closely related application. As the first Futamura projection is language-independent, applications to non-standard interpreters outside of logic programming also exist. For instance, [41] shows how a memoizing meta-interpreter can be specialized to achieve superlinear program speedups for a minimal imperative language, but also points out that finding a suitable interpreter for specialization is non-trivial.

6.2.2 Offline vs. online partial deduction

In chapter 3, we relegated the problem of global control to the oracle and explained that, for our particular application, the elements of $\mathcal{A}$ can simply be the top-level goal and any abstract goal which occurs multiple times, modulo renaming, in the analysis. We also stated that there is no harm in adding *every* abstract conjunction to $\mathcal{A}$, as long as a finite analysis can be obtained. In general, however, control poses a challenge which is approached in one of two ways, namely "online" and "offline" (conjunctive) partial deduction [46]. In online partial deduction, control is tackled during the specialization phase itself. That is, the construction of SLDNF-trees is monitored and unfolding continues as long as there is evidence that interesting computations are performed. Offline partial deduction differs from online partial deduction in that control decisions are taken before the actual specialization phase. These are then cast in the form of annotations for the specialized program, which are understood by the specializer. During the actual specialization phase, unfolding proceeds in a left-to-right fashion. Broadly speaking, depending on its annotation, a call may be unfolded, it may be treated as an instance of potentially more general specialized call or it may be kept residual. Note that this is precisely the type of information that is obtained during the analysis phase in chapter 3, though an interpreter is not explicitly considered there. Given the right annotations, specialization is straightforward, as no decisions regarding control need to be taken. Annotations can be written manually by a programmer with sufficient knowledge of both the program and offline partial deduction, but they can also be derived automatically in a separate analysis phase. This analysis phase is

referred to as a "binding-time analysis" and is performed *before* the actual static input is available.

From a theoretical point of view, online and offline partial deduction are equally powerful [12], but this information should be nuanced. On the one hand online partial evaluation is often more accurate in practice because the static values are available for making control decisions. On the other, hybrid approaches are possible: [79] explicitly describes a "hybrid" approach and the Logen system for offline partial evaluation also incorporates certain techniques from online partial evaluation. Since we have already established a procedure which provides control decisions, we will recast compiling control as a form of offline partial deduction.

6.3 A suitable meta-interpreter

Consider again the analysis trees for permutation sort on page 28. If each abstract conjunction is assigned the number in front of it as an identifier and the empty goal is assigned the (Prolog) atom "empty", a simple Prolog meta-interpreter can run permutation sort under the desired abstract instantiation-based selection rule. This interpreter is shown in listing 6.1. Any missing definitions are either builtins or are supplied with the object program.

```
compute(Gs) :- mi(Gs,1).
mi([],_).
mi([G0|Gs0],State) :-
  selected_index(State,Idx), % supplied
  divide_goals([G0|Gs0],Idx,Before,Selected,After),
  mi_clause(Selected,Body,RuleIdx), % supplied
  state_transition(State,NewState,RuleIdx), % supplied
  append(Before,Body,NewGA),
  append(NewGA,After,NewG),
  mi(NewG,NewState).
mi([G0|Gs0],State) :-
  selected_index(State,Idx),
  divide_goals([G0|Gs0],Idx,Before,Selected,After),
  mi_full_eval(Selected,FullAIIdx), % supplied
  call(Selected),
  state_transition(State,NewState,FullAIIdx),
  append(Before,After,NewG),
  mi(NewG,NewState).
divide_goals(Goals,Idx,Before,Selected,After) :-
  length(Before,Idx),
```

```
append(Before,[Selected|After],Goals).
```

Listing 6.1: Simple meta-interpreter for specialization

Here, $mi/2$ is the meta-interpretation predicate. It takes a concrete conjunction, represented as a list, and a state identifier. The initial call is $compute([permsort(G, A)])$ where G is instantiated to a ground value and A can be any value. The $mi_clause/3$ predicate encodes clauses as a head, a list of body atoms and a unique identifier for the clause. The $mi_full_eval/2$ predicate identifies fully evaluated goals, e.g. $mi_full_eval(select(X, Y, Z), fullai1)$ to remove an element X from a list Y, which yields Z. The $selected_index/2$ predicate supplies the index of the atom to be selected in a particular state. The meta-interpreter itself does not inspect groundness or aliasing characteristics of conjunctions: the selection rule is an *abstract* instantiation-based selection rule, hence these properties are inferred before interpretation and "least common denominator" control is applied. The $state_transition/3$ predicate expresses which states are directly reachable from which other states and which rules cause the transitions.

6.3.1 Instantiation of the first Futamura projection

Using Logen, the interpreter can be specialized for $mi([permsort(X, Y)], 1)$. We cannot express to Logen that X will be instantiated to a ground value, but this is not needed: the control flow is already encoded in $selected_index/2$ and $state_transition/3$. Therefore, the result of the specialization will still be a compiled program with the desired control flow.

With regard to the equation $int(s, r) = \beta(int, s)(r)$, the vector s is described in [31] as "a source program and information needed for syntax analysis and semantic analysis". As such, the selection rule is a component of s in addition to the source program, P. Because the program should work for multiple input queries, a program's top-level goal is considered a runtime input. However, a (concrete) top-level goal instantiated by all runtime top-level goals is a static input. Hence, such a goal is a component of s, even if P is not intended to be run with this goal directly. This observation is what allows logic programs to be specialized for certain top-level calls. Groundness characteristics of the query pattern can also be considered as static information. This is not done in partial deduction of pure logic programs, but we have to take it into account for our approach. We do not do this directly, but by encoding the compiling control analysis as a component of s. Because the analysis provides all required information about the static inputs to the meta-interpreter, there is no need to provide those as direct inputs to the evaluator as well. However, the analysis

does *not* necessarily contain enough information about predicates not involved in the coroutining control flow. That is, fully evaluated predicates may not necessarily be builtins and can be part of the source program, so the source program is also a component of s. The analysis completely specifies the local and global control for a partial deduction of P under the semantics implemented by *int*. In light of the first Futamura projection, an offline partial evaluation of the interpreter must then produce a program which is totally equivalent to that produced by the synthesis step in chapter 3.[1] In other words, the "interpreter-free" synthesis step can be seen as a problem-specific shortcut to the outcome of the first Futamura projection.

Indeed, assume the encoding of permutation sort given in listing 6.2 is used:

```
mi_clause(permsort(X,Y),[perm(X,Y), ord(Y)],rule1).

mi_clause(perm([],[]),[],rule2).
mi_clause(perm([X|Y],[U|V]),[select(U,[X|Y],W),perm(W,V)],rule3).

mi_clause(ord([]),[],rule4).
mi_clause(ord([_]),[],rule5).
mi_clause(ord([X,Y|Z]),[=<(X,Y), ord([Y|Z])],rule6).

mi_full_eval(select(_X,_Y,_Z),fullai1).
mi_full_eval(=<(_X,_Y),fullai2).

selected_index(1,0).
selected_index(2,0).
selected_index(3,0).
selected_index(4,0).
selected_index(5,0).
selected_index(6,0).
selected_index(7,0).
selected_index(8,1).
selected_index(9,1).
state_transition(1,2,rule1).
state_transition(2,3,rule2).
state_transition(2,4,rule3).
state_transition(3,empty,rule4).
state_transition(4,5,fullai1).
state_transition(5,6,rule2).
state_transition(5,7,rule3).
```

[1] We use the term "partial evaluation" rather than "partial deduction" because *int* is not implemented using purely logical constructs.

```
state_transition(6,empty,rule5).
state_transition(7,8,fullai1).
state_transition(8,9,rule6).
state_transition(9,5,fullai2).
```

Listing 6.2: Permutation sort as an object program

When this program is specialized using techniques outlined below, the code in listing 6.3 is obtained.

```
top(A,B) :-
    top__1(A,B).
top__1([],[]).
top__1([A|B],[C|D]) :-
    select(C,[A|B],E),
    mi__5__2(D,E,C).
mi__5__2([],[],A).
mi__5__2([A|B],[C|D],E) :-
    select(A,[C|D],F),
    E =< A,
    mi__5__2(B,F,A).
```

Listing 6.3: Specialized version of meta-interpreter running permutation sort

After inlining the *top*/2 predicate, the above program becomes a syntactic variant of that in listing 3.1 on page 31.

6.4 Programs requiring the *multi* abstraction

A specification of the control flow using *selected_index*/2 is no longer possible if the *multi* abstraction is required to complete the analysis. In two different concrete instances of an abstract conjunction containing *multi*, the concrete atoms selected for resolution can be at different index positions. This is due to the fact that an abstract conjunction containing *multi* represents an infinite number of concrete conjunctions of arbitrary length. The solution, then, is to transform the program so that this is no longer the case. Transforming the interpreted program P into a different purely logical program which can be dealt with — at least through a simple transformation which does not introduce new problems for the analysis — is not an option. It is, however, possible to generalize the SLD-based execution mechanism implemented by the meta-interpreter in a way which still allows the results of partial deduction to be applied. The idea behind this is to introduce an additional operation, *grouping*,

which transforms atoms into arguments of an atom with a special predicate symbol, *cmulti*. This operation is a counterpart to the concrete generalization operation in chapter 3 and does not affect the results of the program in any way, but strengthens the correspondence between the abstract analysis and a concrete execution. In this way, the length of *concrete* conjunctions represented in the meta-interpreter also becomes bounded. To show how this works in more detail, we first detail the operation of the meta-interpreter.

6.4.1 An extended meta-interpreter

The key modifications to the meta-interpreter are shown in the clauses in listing 6.4. The complete source code is provided in appendix C.

```
mi([G|Gs],State) :-
  state_instantiation(State,[G|Gs]),
  selected_index(State,Idx),
  state_transition(State,NewState,1), % case: one
  divide_goals([G|Gs],Idx,Before,cmulti([BBPatt]),After),
  BBPatt =.. [building_block,Patt],
  append([Before,Patt,After],NewGs),
  mi(NewGs,NewState).
mi([G|Gs],State) :-
  state_instantiation(State,[G|Gs]),
  selected_index(State,Idx),
  state_transition(State,NewState,2), % case: many (from left)
  divide_goals([G|Gs],Idx,Before,
               cmulti([building_block(BB),BB2|BBs]),After),
  append([Before,BB,[cmulti([BB2|BBs])],After],NewGs),
  mi(NewGs,NewState).
mi([G|Gs],State) :-
  state_instantiation(State,[G|Gs]),
  selected_index(State,Idx),
  state_transition(State,NewState,3), % case: many (from right)
  divide_goals([G|Gs],Idx,Before,cmulti([BB1,BB2|BBs]),After),
  length([BB1,BB2|BBs],LL),
  LLM is LL - 1,
  take(LLM,[BB1,BB2|BBs],IBBs),
  last([BB2|BBs],LBB),
  LBB = building_block([LBBH|LBBT]),
  append([Before,[cmulti(IBBs)],[LBBH|LBBT],After],NewGs),
```

```
  mi(NewGs,NewState).
mi([G|Gs],State) :-
  state_instantiation(State,[G|Gs]),
  grouping(State,NextState,Groupings),
  apply_groupings([G|Gs],Groupings,NewGs),
  mi(NewGs,NextState).
```

Listing 6.4: Extended meta-interpreter for specialization

First, a few general remarks are necessary: In the extended meta-interpreter, every *mi*/2 clause begins with a call to *state_instantiation*. This call has no effect on executions of the meta-interpreter, but it will provide Logen with useful information later on. Also, each instance of the pattern of a *multi* is wrapped inside a *building_block* structure to easily extract a single instance of the pattern from the *cmulti* at a later point in the program. This is useful for programs like sameleaves (page 65).

The first three new clauses extract atoms from a concrete counterpart to a *multi* abstraction. Specifically, the first extracts a single conjunction from a concrete *multi* abstraction when the latter does not contain any more conjunctions. It strips away the *building_block* functor from the instance of the *multi* abstraction's pattern and then inserts the stripped pattern instance in the next goal to be interpreted.

The second is similar, but it extracts a conjunction at the beginning of a concrete *multi* abstraction when the latter contains multiple conjunctions. It uses *append*/2 to insert both the extracted pattern instance and the remaining concrete *multi* abstraction.

The third extracts a conjunction at the end of a concrete *multi* abstraction when the latter contains multiple conjunctions. Recall that this was necessary for the sameleaves program. Figure 3.7 on page 68 illustrates when and how this happens.

The fourth additional clause one is the one which introduces the concrete counterpart to the *multi* abstraction during interpretation. It uses *apply_groupings*/3 to group certain concrete conjunctions in a concrete *cmulti* structure. The code for this operation is quite long, but it is sufficient to know that the second argument to *apply_groupings*/3 specifies, for a given state, which subconjunctions of the overall goal should be considered instantiations of a *multi* abstraction. This information is available from the analysis phase and is encoded using the *grouping*/3 predicate, which states which atom indices are grouped during the transition from one state to the next.

For example, $grouping(54, 55, [[(2, 3), (3, 4)]])$ expresses that, in a concrete

instance of the transition from state 54 to state 55, the atoms at index positions 2 (that is, from index position 2 to right before index 3) and 3 (that is, from index position 3 to right before index position 4) instantiate a *multi* and can be grouped together.

In the following sections, we show how a specialization of the meta-interpreter can be obtained using Logen and that it is equivalent to the "interpreter-free" synthesis.

6.5 Specialization using Logen

As a system for *offline* partial deduction, Logen is driven by annotations. These take the form of "filter declarations" and "clause annotations" [49] [18]. Filter declarations associate arguments of predicates with so-called "binding types". A binding type can be a binding *time*, e.g. "static" or "dynamic", but it can also restrict the structure of the argument. Our filter declarations require the following binding types: "static", "dynamic", "nonvar", "struct" and "list". The first two are common binding *times*, expressing whether or not an argument will be known at specialization time. The third expresses that an argument is ground or has some structure during specialization, but provides no further information. The fourth can specify the structure of an argument, i.e. the functor and the binding types of its arguments. The last binding type is for *closed* lists, i.e. lists whose length is bounded. A binding type can also be a disjunction of binding types, in which case an argument is considered to be an instance of the first applicable disjunct. Intuitively, the purpose of a binding type is this: before a call to a predicate with certain binding types is specialized, the call is generalized and is then used as the root of a derivation tree. The binding types determine to which extent the call is generalized before construction of the tree begins. A static argument is not generalized, whereas a dynamic one is replaced with a variable. The more specific the binding types, the more information can be used during specialization.

Clause annotations indicate how every call in a clause body should be treated during unfolding. All clause annotations for the current work can be written using the following strategies: "unfold", "call", "rescall" and "memo". The first means that an atom should simply be resolved and that Logen should continue building a derivation tree. The second means that a call should be executed at specialization time, while the third means that it should be executed at runtime. The last one means that it should not be unfolded further, but that it should be used as the root for a new derivation tree (after generalization, as specified by the filter declarations).

6.5.1 Simple meta-interpreter

For the simple meta-interpreter, we start from the naive program annotation performed by Logen. This is built into the system and annotates every predicate call as "unfold" and every builtin call as "call". We change the recursive call to $mi/2$ from "unfold" to "memo", so that it may be specialized separately. This can be seen as starting a new tree in partial deduction and adding the root of this tree to the set of analyzed conjunctions $\mathcal{A}$ for every newly derived conjunction. If the default option, "unfold", is used, new derivation trees are not started and the set of analyzed conjunctions $\mathcal{A}$ is not closed. Furthermore, we annotate the call to $call/1$ as "rescall" instead of "call". Calls to $call/1$ in the meta-interpreter apply to predicates which, in the abstract analysis, are underlined twice and which we consider residual. The "rescall" annotation expresses precisely the idea that a call should simply be executed when the program is run. The "call" annotation should not be used, as this executes the call during specialization time instead of during program execution. The filters are as follows:

- `compute(struct(permsort,[dynamic,dynamic]))`

  We specify as much information about the top-level call as possible. We cannot specify that the arguments to the call will have certain groundness characteristics, but this is not necessary as the $selected_index/2$ and $state_transition/3$ predicates encode this information.

- `mi(type(list(nonvar)),static)`

  Neither the goal, which is represented as a list, nor its elements should be abstracted to variables in the root of a derivation tree. The goal can be abstracted to a bounded list, because the length of each concrete conjunction is known when the state identifier is known. We also keep track of said identifier and do not generalize it. This is what allows the partial deduction procedure to keep track of which abstract conjunction corresponds to the concrete call currently being specialized.

The arguments of other filters can be left dynamic: some values *will*, in fact, be known at specialization time, but if the calls to an annotated predicate are never used as the root of a derivation tree (and they are not, because they never have the "memo" annotation), annotating these known arguments as "static" is not harmful, but has no impact on the partial deduction.

6.5.2 Extended meta-interpreter

For the extended meta-interpreter, the reasoning behind the clause annotations remains mostly the same. However, a small but crucial change must be applied to deal with the case split on the *multi* abstraction during specialization. During partial evaluation, information regarding the structure of the *multi* abstraction is necessarily generalized away. To the best of our knowledge, we cannot use filter declarations to preserve crucial info from the abstract analysis about the internal structure of a concrete *multi and* properly generalize calls so that the set $\mathcal{A}$ is closed. The problem lies in the fact that the specialization procedure must generalize, in the concrete domain, over lists of arbitrary length. This can only be done using a variable to represent the tail of the list, which means that information regarding the contents of the tail is lost. Specifically, it becomes impossible to use append operations to build a conjunction at specialization time if some of the appended elements are free variables. However, we can re-encode the required information into the call annotations. This is why the *state_instantiation*/2 calls were added to the *mi*/2 clauses above. An example of a *state_instantiation* clause for the sameleaves program, for instance, is:

```
state_instantiation(19,
  [collect(_,A),
   collect(_,[_|_]),
   cmulti([building_block([collect(_,B),
                           append(_,B,_)])|_]),
   eq(A,_)]).
```

With these remarks in mind, the important annotations are shown in listing 6.5.

```
logen(mi/2,mi([B|C],A)) :-
       logen(unfold,state_instantiation(A,[B|C])),
       logen(unfold,selected_index(A,D)),
       logen(unfold,state_transition(A,E,1)),
       logen(unfold,divide_goals([B|C],D,F,cmulti([H]),G)),
       logen(call,H=..[building_block,I]),
       logen(call,append(F,I,J)),
       logen(call,append(J,G,K)),
       logen(memo,mi(K,E)).
logen(mi/2,mi([B|C],A)) :-
       logen(unfold,state_instantiation(A,[B|C])),
       logen(unfold,selected_index(A,D)),
```

```
        logen(unfold,state_transition(A,E,2)),
        logen(unfold,
              divide_goals([B|C],D,F,
                           cmulti([building_block(H),I|J]),G)),
        logen(call,append(F,H,K)),
        logen(call,append(K,[cmulti([I|J])],L)),
        logen(call,append(L,G,M)),
        logen(memo,mi(M,E)).
logen(mi/2,mi([B|C],A)) :-
        logen(unfold,state_instantiation(A,[B|C])),
        logen(unfold,selected_index(A,D)),
        logen(unfold,state_transition(A,E,3)),
        logen(unfold,divide_goals([B|C],D,F,cmulti([H,I|J]),G)),
        logen(rescall,length([H,I|J],K)),
        logen(rescall,L is K-1),
        logen(rescall,take(L,[H,I|J],M)),
        logen(rescall,last([I|J],N)),
        logen(rescall,N=building_block([O|P])),
        logen(rescall,append(F,[cmulti(M)],Q)),
        logen(rescall,append(Q,[O|P],R)),
        logen(rescall,append(R,G,S)),
        logen(memo,mi(S,E)).
```

Listing 6.5: Annotated version of the extended meta-interpreter

For unfolding *multi* from the left, calls to *append* can be performed at specialization time, as indicated by the *call* annotation. For unfolding *multi* from the right, the *rescall* annotation is used instead. This is because the calls with this annotation extract the *last* conjunction inside a *multi* abstraction. This is not possible at specialization time, because the list position of this last conjunction is not fixed.

Due to the use of *state_instantiation*/2, which is unfolded at specialization time, type annotation for *mi*/2 is simplified to `mi(dynamic,static)`.

6.6 Equivalence with the interpreter-free approach

The programs produced by Logen appear quite different from the syntheses obtained for the same programs using the interpreter-free approach. However, the resulting programs produce identical answer sets and display identical finite failure behavior. Furthermore, despite surface differences, they produce their answers after a similar number of inferences. This demonstrates that both

versions implement the same control flow. For instance, the sum total of the number of inferences required to find all solutions to the 10-queens problem with one approach deviates less than 22% from the number of inferences required with the other approach. The difference can be explained by the presence of redundant code, which could be filtered out with additional post-processing.[2] More exhaustive benchmarks can be found in the electronic appendix and they all lead to similar observations. Therefore, although we do not have a formal proof, empirical evidence strongly points to both approaches being equivalent.

6.7 Discussion

We have shown that compiling control can be regarded as a conjunctive partial deduction of a program run by a meta-interpreter or, equivalently, as a specialization of the meta-interpreter for said program. This is an application of the first Futamura projection, which answers a question which has long remained open. We have also provided a software pipeline which applies this idea in practice. The possibility of recasting the ad hoc synthesis step of compiling control as a partial deduction raises the question whether the analysis step can be similarly recast as an abstract interpretation of a meta-interpreter and of what this would entail. This would be a meta-interpreter which performs concrete selection based on an abstract instantiation-based selection rule, as per definition 42 on page 70. This is not as obvious as it may seem at first sight, because, for most of this research, we have assumed "lowest common denominator" control, as discussed in chapter 4. Therefore, a different abstract domain which is not closed under substitution seems inevitable to perform such an analysis.

[2] In a previous version of this text, we used the Ecce system for partial deduction to achieve this. However, the literature [48] points out that Ecce does not support programs which depend on left-to-right selection and this was likely the cause of errant results in the specialized version of the prime generator.

Chapter 7

Implementation

7.1 General design

The current implementation of our work is an interpreter for logic programs (written in SWI-Prolog), which uses our abstract semantics. It is semi-automated: It performs abstract atom selection given an instantiation-based selection rule, applies abstract resolution, full evaluation or unfolding of *multi* and tracks the program analysis tree. It can also extract an instantiation-based selection rule from user input, which is convenient compared to completely specifying an instantiation-based selection rule. The implementation does not automatically apply the *multi* abstraction, nor does it support any other abstractions than the linear *multi*.

The user-facing side of the implementation is an interactive loop. For conjunctions which are handled in a non-trivial way (which excludes empty conjunctions and loops), two options are presented:

- Unfolding or full evaluation: An abstract conjunct will be selected for unfolding. If the abstract instantiation-based selection rule is only partially specified, the user is first asked to select an abstract conjunct to which an operation will be applied. If the operation is full evaluation, abstract resolution will not be applied, even if there are clauses whose head matches.
- Generalization: The user is asked to specify a new abstract conjunction containing a *multi* abstraction which represents, among others, the current abstract conjunction.

A third form of user interaction is supported, but it is not part of the interactive loop: users can state that branches containing certain abstract atoms should be aborted. This is useful when it is clear that a branch of the abstract analysis tree has no concrete instances which can lead to success. As an example, in the sameleaves program, abstract conjunctions containing abstract atoms of the form $collect(g_1, [])$ appear, but the branches on which they appear should be aborted: Because every tree has at least one leaf, the presence of this abstract atom in an abstract conjunction implies branch failure. If branch failure is not inferred in this way, the computation becomes significantly more complex and is no longer inside the scope of our implementation. Inferring branch failure from such "useless" abstract atoms can be seen as a particular case of full evaluation.

Instructions for using the implementation and data files for reproducing our results are provided in appendix E.

7.2 Representation of the abstract domain

7.2.1 $ADom_P$

As stated in section 3.1.1, a_i and g_j are abstract variables. Indeed, during the analysis phase, they behave as concrete variables whose guaranteed level of instantiation is tracked. Therefore, they are represented using concrete variables, associated through the `-/2` functor with a constant indicating the level of instantiation. For instance, every occurrence of an abstract variable a_1 in the implementation is represented using a Prolog term `A1-a` (or some consistently renamed variant). An abstract variable g_1 is represented using a Prolog term `G1-g` (or some consistently renamed variant). Abstract terms and atoms are represented as Prolog compounds in which no concrete variables appear, unless those variables are associated with a constant indicating their level of instantiation. Abstract conjunctions are represented using lists of abstract atoms.

7.2.2 $ADomMulti_P$

The *multi* abstraction is represented using a `multi/4` term. The first argument is an open-ended list, representing a number of instances of the pattern of the abstraction. Parameterized variables inside the pattern are represented using numbers indicating those variables' first appearance within a parse tree of the pattern's representation, if this tree is traversed in a depth-first manner. Figure 7.1 demonstrates how this numbering scheme works

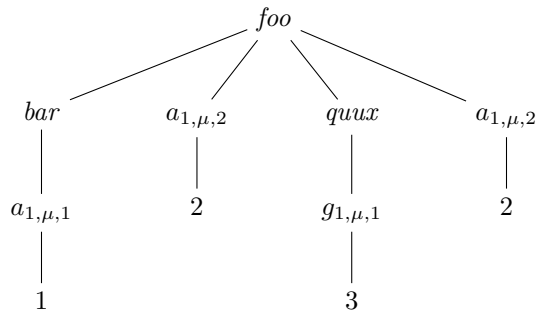


Figure 7.1: The numbers in the leaf nodes of the parse tree represent their parents, the parameterized abstract variables.

using a relatively complex pattern, $foo(bar(a_{1,\mu,1}), a_{1,\mu,2}, quux(g_{1,\mu,1}), a_{1,\mu,2})$. To see how this works in an abstract conjunction, consider the conjunction $integers(g_1, a_1), multi(filter(g_{1,\mu,1}, a_{1,\mu,1}, a_{1,\mu,2}), \{a_{1,1,1} = a_1\}, \{a_{1,\mu+1,1} = a_{1,\mu,2}\}, \{a_{1,\mathcal{L},2} = a_2\}), sift(a_2, a_3), length(a_3, g_2)$ from the prime sieve. Here, $a_{1,\mu,1}$ is the second abstract variable in the pattern and $a_{1,\mu,2}$ is the third. If a_1 is represented as `A1-a` and a_2 is represented as `A2-a`, the abstraction as a whole is `multi([building_block([filter(_-g,_-a,_-a)])|_],[(2,A1-a)],[(2,3)],[(3,A2-a)])`.

Note that the pattern is wrapped inside a `building_block/1` functor. This makes it easy to unfold a single instance from the pattern, even if the pattern consists of more than one atom (as is the case in, e.g., sameleaves).

7.2.3 Analyzer input

Programs to be analyzed are represented using four predicates.[1] These are as follows:

- `prog_abstract_conjunction(Idx,ACon)`: This predicate provides the implementation with two abstract conjunctions known at the start of the analysis: the empty conjunction and the top-level query. For example, the input for the analysis of permutation sort initially contains the following two facts for `prog_abstract_conjunction/2`:

  ```
  prog_abstract_conjunction(0,[]).
  prog_abstract_conjunction(u(1),[permsort(_-g,_-a)]).
  ```

[1] The examples export a fifth predicate, `prog_case_clause`, but this is specific to more experimental examples which were deemed out of scope.

The index for the top-level query is wrapped to indicate that the conjunction has not yet been analyzed. In other programs, only `permsort(_-g,_-a)` is replaced with a different top-level query. The remainder of this predicate's definition stays the same.

- `prog_clause(Head,Body)`: These facts are a representation of the program to be analyzed. `Head` is the same as the head of a corresponding clause in a native program. `Body` is a list of mostly concrete atoms. However, variables that only occur in the body must be annotated explicitly as an "any"-type variable. For example, the file for permutation sort contains the following clause representation, in which `W` only occurs inside the clause body:

  ```
  prog_clause(perm([X|Y],[U|V]),
    [select(U,[X|Y],W-a),perm(W-a,V)]).
  ```

  This additional requirement is due to some provisions made to later support the "speculative" analysis.

- `prog_full_eval(Pattern,Bindings)`: `Pattern` is an abstract atom. Any abstract atoms equivalent to or more specific than `Pattern` can be fully evaluated. Doing so applies the bindings specified in `Bindings`, which are supplied by the user. Every binding is written as a tuple consisting of the variable part of an abstract variable in `Pattern` and an abstract term to which the variable will become bound upon full evaluation.

  For permutation sort, the following fact illustrates this:

  ```
  prog_full_eval(select(A-a,[_-g|_-g],B-a),[(A,_-g),(B,_-g)]).
  ```

- `prog_precedence_pair(ACon1,ACon2)`: A fact of this type represents `ACon1` $\xrightarrow{p}$ `ACon2`, as defined in section 3.2.4. This predicate need not be fully defined by the user. It is possible to perform atom selections interactively and to let a definition for this predicate be generated by the implementation. This is explained in section 7.5.

- `prog_failure(Pattern)`: A fact of this type indicates that abstract atoms equivalent to or more specific than `Pattern` guarantee branch failure. For example, in sameleaves, `prog_failure(collect(_-g,[]))` indicates that collecting the leaves of a tree will never yield an empty list.

7.3 Representing the analysis

To represent the analysis, the implementation uses dynamic predicates whose definition is altered as the analysis is expanded. Figures 7.2 through 7.6 show how a graph representing a (partial) abstract analysis is translated to Prolog facts using the first four predicates. Figure 7.7 shows how a small segment of a different abstract analysis (in which a *multi* abstraction is used) is represented.

The predicates are as follows:

- `abstract_conjunction(Idx,ACon)`: The analysis tree node labeled `Idx` contains an abstract conjunction from the equivalence class of abstract conjunctions `ACon`. The label can be "symbolic" or nonsymbolic. A symbolic label is used for nodes to which an operation (represented using one of the four following dynamic predicates) has not yet been applied, while a nonsymbolic one is used for nodes to which an operation has been applied. The nonsymbolic indexes make up a contiguous sequence of natural numbers when the tree nodes are traversed in depth-first order, with the root having index `1`. When all indexes are nonsymbolic, the analysis is complete.[2]

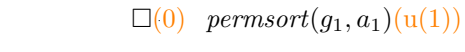
- `op_full_eval(Parent,Sel,EvalIdx,OutIdx)`: The abstract atom at index position `Sel` in the abstract conjunction corresponding to tree node `Parent` was fully evaluated. The logic for full evaluation is identified by the index `EvalIdx`. The result of applying full evaluation is the abstract conjunction identified by the index `OutIdx`.

- `op_resolve(Parent,Sel,RuleIdx,OutIdx)`: Similar to `op_full_eval/4`, but represents that an abstract atom was unfolded using a clause identified by `RuleIdx`, rather than being fully evaluated.

- `op_cycle(From,To)`: The abstract conjunction identified by the index `From` is a renamed instance of that identified by the index `To`.

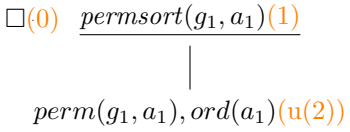
- `op_generalize(From,Ranges,To)`: The abstract conjunction identified by index `From` was generalized (using a *multi* abstraction) to the abstract conjunction identified by `To`. The abstract atoms and pre-existing *multi* abstractions (if any) grouped into the new *multi* abstraction are identified by `Ranges`. Specifically, each range represents a contiguous string of instances of the pattern of the newly created *multi*. This information

[2]This representation using a dynamic predicate masks aliasing between different nodes. This is different from the visualizations, but it does not impact our ability to complete an analysis or perform a synthesis.



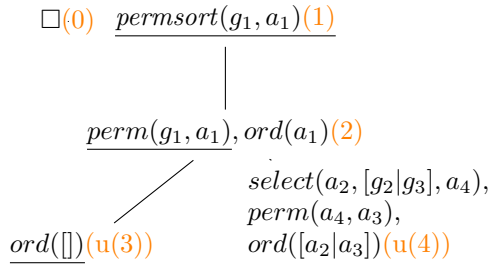

```
abstract_conjunction(0,[]).
abstract_conjunction(u(1),[permsort(_-g,_-a)]).
```

Figure 7.2: Representation of the analysis when the top-level goal has not yet been unfolded.




```
abstract_conjunction(0,[]).
abstract_conjunction(1,[permsort(_-g,_-a)]).
abstract_conjunction(u(2),[perm(_-g,A1-a),ord(A1-a)]).
op_resolve(1,1,4,u(2)).
```

Figure 7.3: Representation of the analysis immediately after the first abstract resolution step. Note that 4, in the bottom fact, represents the clause defining `permsort/2`.




```
abstract_conjunction(0,[]).
abstract_conjunction(1,[permsort(_-g,_-a)]).
abstract_conjunction(2,[perm(_-g,A-a),ord(A-a)]).
abstract_conjunction(u(3),[ord([])]).
abstract_conjunction(u(4),
  [select(A-a,[_-g|_-g],B-a),perm(B-a,C-a),ord([A-a|C-a])]).
op_resolve(1,1,4,2).
op_resolve(2,1,5,u(3)).
op_resolve(2,1,6,u(4)).
```

Figure 7.4: Representation of the analysis immediately after the second abstract resolution step with an applicable clause.

is used to generate the `grouping/3` facts used to implement the "group" discussed in chapter 6.

7.4 Abstract unification

Most of the techniques used in the implementation are common in Prolog meta-interpreters. The implementation of abstract unification uses a technique which is less obvious. As with algorithm 1 in chapter 3, the key idea is that abstract unification involves the same logic as concrete unification, but must also maintain consistent annotations for variables.

The implementation is given in listing 7.1. Because abstract unification involves the same steps as concrete unification, plus additional steps to guarantee correct annotations, the first part of listing 7.1 (lines 2-4) strips the annotations from variables and applies concrete unification. Specifically, `Stripped1` and `Stripped2` are nearly identical to `AT1` and `AT2`, except that they contain only the concrete parts of abstract variables. The third argument of `strip_var_types/3` associates each variable with its type *before* unification. For instance, if `AT1` contains a subterm `foo(A-a)`, then `Assoc` is a list containing `A-a`. Therefore, after unification, `Assoc` can contain redundant entries. For example, if a_1 (represented by `A1-a`) is unified with g_1 (`G1-g`), `Assoc` will contain `G1-a` *and* `G1-g`. Only the strictest of these entries, which is `G1-g` in the example, contains information which will be used by the analysis. The job of `simplify_assoc/2` on line 6 is to sift out all but the strictest information about each variable. The predicate on the last line, `associate_vars_with_types/1`, unifies each variable with a fresh variable, annotated with the strictest annotation. Due to variable aliasing, this means that every occurrence of the variable now carries the correct annotation.

```
1 abstract_unify(AT1,AT2) :-
2   strip_var_types(AT1,Stripped1,Assoc1),
3   strip_var_types(AT2,Stripped2,Assoc2),
4   unify_with_occurs_check(Stripped1,Stripped2),
5   append(Assoc1,Assoc2,Assoc),
6   simplify_assoc(Assoc,SimplifiedAssoc),
7   associate_vars_with_types(SimplifiedAssoc).
```

Listing 7.1: Implementation of abstract unification

The approach does come with a caveat: It can lead to the introduction of multiple annotations, with the annotation closest to the concrete variable being more

specific. For instance, `foo(A-a)`, after unification with `foo(G-g)`, is instantiated to `foo(_-g-a)`. The implementation solves this by running a cleanup predicate, `clean_redundant_annotations/2`, on unified terms as well as other terms which may share variables with them after abstract unification. This only retains the most specific annotation for the abstract variables contained in its input argument.

7.5 The selection rule

As indicated above, the user need not fully specify the selection rule using `prog_precedence_pair/2` facts. This would be rather unpractical. Instead, the implementation asks the user to select an abstract atom from a set of options when there is insufficient information about $\xrightarrow{p}$. However, this raises the question of when there is insufficient information. By definition 26, if no elements are ordered by $\xrightarrow{p}$, left-to-right selection is applied. Therefore, even with no information about $\xrightarrow{p}$, an atom selection can be made.

A practical solution is not to use the strict partial order $\xrightarrow{p}$ on abstract atoms, but a strict partial order $\xrightarrow{ip}$ on classes of abstract atoms. Both the pairs that make up $\xrightarrow{ip}$ and the contents of the classes of abstract atoms can be unspecified at the start of the analysis. For an abstract conjunction $A_1, \ldots, A_n$, the abstract atom selected is the leftmost one belonging to a class $\bar{a}$ such that, for all other classes $\bar{b}$ of $A_1, \ldots, A_n$, $\bar{a} \xrightarrow{ip} \bar{b}$ is guaranteed by the known information about $\xrightarrow{ip}$ (including the properties of a strict partial order). If there is no such class, the user must select an abstract atom A_i. This further specifies $\xrightarrow{ip}$ by establishing $\bar{A_i} \xrightarrow{ip} \bar{A_1}, \ldots, \bar{A_i} \xrightarrow{ip} \bar{A_{i-1}}$. It does not imply $\bar{A_i} \xrightarrow{ip} \bar{A_{i+1}}, \ldots, \bar{A_i} \xrightarrow{ip} \bar{A_n}$, as $\bar{A_{i+1}}$ and $\bar{A_n}$ could still be identical to $\bar{A_i}$.

Given a complete program analysis, a sufficiently detailed specification of $\xrightarrow{p}$ can be derived from $\xrightarrow{ip}$: if two abstract atoms a and b belong to the same class, neither is ordered with respect to the other by $\xrightarrow{p}$. If there are two classes $\bar{a}$ and $\bar{b}$ such that $\bar{a} \xrightarrow{ip} \bar{b}$, for every representative a of $\bar{a}$ and every representative b of $\bar{b}$, $a \xrightarrow{p} b$ holds.

The implementation makes the assumption that each class of abstract atoms is a singleton set. This means that two non-equivalent abstract atoms are never equivalent with regard to precedence. However, this is strictly for ease of implementation, because none of the examples in scope required the additional flexibility.

7.6 Program synthesis

Due to the representation of the program analysis outlined in 7.3, it is straightforward to generate a meta-interpreter based on the extended meta interpreter in chapter 6 for the analyzed program. The synthesis then becomes the generation of this meta-interpreter, followed by its specialization using Logen. The original approach to specialization is no longer supported, as it entails more work and does not offer any additional functionality. Compilation targets besides Prolog are also not currently supported.

Generating the meta-interpreter only requires bundling the code for the extended meta-interpreter from chapter 6 with the following predicates:

- `mi_clause/3`: This is the program logic. Definitions of fully evaluated, non-built-in predicates must also be provided in this way (or added to the generated program manually).
- `state_transition/3`: This tracks how state indexes change based on which program clauses are used for resolution. This is extracted from the `op_resolve` and `op_full_eval` facts.
- `selected_index/2`: This is generated by determining the index position of the abstract atom whose class precedes that of those of all other abstract atoms, as explained in section 7.5.
- `grouping/3`: This is directly specified by the user.
- `state_instantiation/2`: This follows from `abstract_conjunction/2` and guarantees that Logen has vital information which it would otherwise abstract away.

An explanation of how precisely to annotate this meta-interpreter with type declarations and filter annotations is provided in appendix E.

7.7 Performance

It is difficult to measure the performance of our control compiler, as it relies on user input. To eliminate this from the equation, we have written input files (supplied in the electronic appendix) which can be redirected to the implementation. Of course, eliminating user input from the equation provides a significant speed boost, particularly because the input files tell the implementation which *multi* abstractions should be introduced. Therefore, we

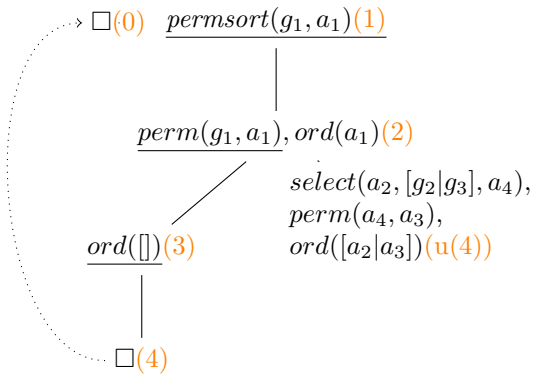
program	**analysis**	**synthesis**	**abstractions**	**total**
permutation sort	0.029	0.019	0.037	0.085
graph coloring	4.772	4.590	176.385	185.747
N queens	85.601	85.877	173.944	345.422
primes	6.474	6.432	190.203	203.109
sameleaves	39.839	29.384	79.238	148.461

Table 7.1: Program transformation timings, in seconds

have also included the time needed to detect which *multi* abstractions are useful using an experimental tool, which is provided in the electronic appendix and which detects usable abstractions for all our examples, though its completeness properties are not yet fully established. The benchmarks in table 7.1 were performed on commodity hardware: a laptop with an Intel Core i7-4800MQ CPU running Ubuntu 18.04.

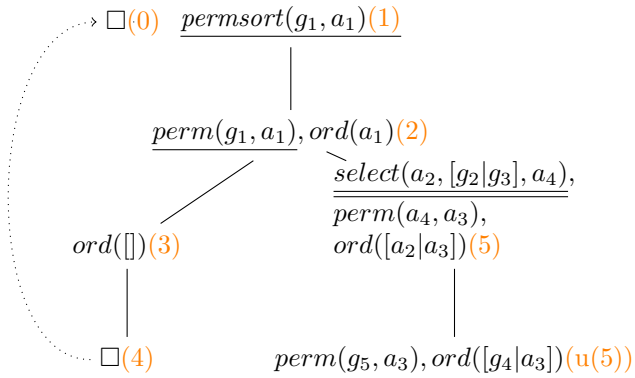
As expected, the implementation discussed here generally takes longer for programs which go through more states. Surprisingly, the N queens problem, which does not exhibit any major differences with the prime sieve as far as coroutining behavior is concerned, is a significant outlier. It is currently unclear which aspect of the program acts as a performance bottleneck. The experimental detection of useful *multi* abstractions follows a different pattern. It is mainly affected by how quickly repeating patterns begin to occur, if at all. Indeed, this happens much sooner during the analysis of the sameleaves program than during that of arguably simpler programs.

The overall image is that the transformation is not instantaneous, but that it can feasibly be applied to the programs which are currently in scope. Furthermore, the implementation has not yet been optimized for high performance.




```
abstract_conjunction(0,[]).
abstract_conjunction(1,[permsort(_-g,_-a)]).
abstract_conjunction(2,[perm(_-g,A-a),ord(A-a)]).
abstract_conjunction(u(4),
  [select(A-a,[_-g|_-g],B-a),perm(B-a,C-a),ord([A-a|C-a])]).
abstract_conjunction(3,[ord([])]).
abstract_conjunction(4,[]).
op_resolve(1,1,4,2).
op_resolve(2,1,6,u(4)).
op_resolve(2,1,5,3).
op_resolve(3,1,7,4).
op_cycle(4,0).
```

Figure 7.5: Representation of the analysis immediately after introduction of the first loop.




```
abstract_conjunction(0,[]).
abstract_conjunction(1,[permsort(_-g,_-a)]).
abstract_conjunction(2,[perm(_-g,A-a),ord(A-a)]).
abstract_conjunction(3,[ord([])]).
abstract_conjunction(4,[]).
abstract_conjunction(5,
  [select(A-a,[_-g|_-g],B-a),perm(B-a,C-a),ord([A-a|C-a])]).
abstract_conjunction(u(5),[perm(_-g,A-a),ord([_-g|A-a])]).
op_resolve(1,1,4,2).
op_resolve(2,1,6,5).
op_resolve(2,1,5,3).
op_resolve(3,1,7,4).
op_cycle(4,0).
op_full_eval(5,1,11,u(5)).
```

Figure 7.6: Representation of the analysis immediately after the first full evaluation step.

$$
\begin{array}{l}
\quad\quad\quad

Chapter 8

Application: termination analysis

The compiling control transformation removes advanced control mechanisms and produces a program which is hard to read, but which uses simpler mechanisms. An advantage of using simple mechanisms, even if the program becomes harder to read, is that further automated static analyses, such as termination analysis, become easier. This has been demonstrated in practice in the past by applying termination analysis to partially evaluated programs [77]. In what follows, we show that our transformation also has the benefit of facilitating termination analysis. This chapter presents findings which have not previously been published.

8.1 Meta-interpreter

To the best of our knowledge, freely available termination analyzers supporting Prolog, e.g. TermiLog [52], TerminWeb [13], cTI [62], PolyTool [63] and APRoVe [35], operate under the assumption of a left-to-right selection rule. This entails that these termination analyzers cannot be applied directly to any of the source programs considered in this work under their intended selection rule. However, they can be applied to the source programs as executed by a meta-interpreter which implements an instantiation-based selection rule, if said meta-interpreter is implemented using a left-to-right selection rule. If such a hypothetical meta-interpreter can be found and if a termination analysis of this meta-interpreter yields weaker results than a termination analysis of a synthesized program,

this makes a strong case for our transformation.[1] The meta-interpreters from chapter 6 operate under left-to-right selection, but they utilize information obtained during the analysis phase, which is the most challenging part of the transformation. Therefore, they are not suitable candidates for a comparison with the synthesized programs.

A meta-interpreter which uses *concrete* selection by an abstract instantiation-based selection rule (definition 42, page 70) does not need the abstract analysis and proposition 9 guarantees that the same selections executed by such a meta-interpreter are also encoded in the program synthesis for the top-level queries under consideration. The most essential parts of such a meta-interpreter are given in listing 8.1 and a complete version is supplied in appendix D. Careful readers may notice that the meta-interpreter in listing 8.1 includes code specific to the interpreted program, permutation sort. This code is only included so that termination analyzers which expect a single input file or listing can be used. The interpreter also makes the assumption of fully evaluated atoms being evaluated first. This could be removed, but the additional cost would be a more verbose selection rule. This meta-interpreter is restricted to ISO-Prolog for compatibility with existing termination analyzers.

```
% interpreted program logic
mi_clause(permsort(X,Y), [perm(X,Y), ord(Y)]).
... % nothing new
mi_clause(ord([X,Y|Z]), [=<(X,Y), ord([Y|Z])]).

% interpreted program control
precedence(perm(_-g,_-a),
           [ord(_-a),
            ord([_-g|_-a])]).
precedence(ord([_-g,_-g|_-a]),
           [perm(_-a,_-a),
            perm(_-g,_-a)]).

% Goal is atomic
compute(Goal) :-
  adjacency_matrix(M), % based on precedence/2
  tc(M,TC), % transitive closure because relation is transitive
  uniq_aatoms(UAA), % abstract atoms act as row/column headings
  mi([Goal],TC,UAA). % begin actual meta-interpretation
```

[1] Admittedly, we will not be able to demonstrate that we have found *the* meta-interpreter which is most amenable to termination analysis, but we can show that the transformation solves a non-trivial problem.

```
mi([],_,_). % empty goal
mi([G0|Gs0],TC,AAtomsOrig) :-
  select_conjunct([G0|Gs0],TC,AAtomsOrig,Selected,GBefore,GAfter),
  \+((full_eval(FullyEvaluatedPattern),
      abstracts(FullyEvaluatedPattern,Selected))),
  mi_clause(Selected, Body),
  my_append([GBefore,Body,GAfter],NewG), % append/2 is not ISO
  mi(NewG,TC,AAtomsOrig).
mi([G0|Gs0],TC,AAtomsOrig) :-
  select_conjunct([G0|Gs0],TC,AAtomsOrig,Selected,GBefore,GAfter),
  full_eval(FullyEvaluatedPattern),
  abstracts(FullyEvaluatedPattern,Selected),
  call(Selected),
  my_append(GBefore,GAfter,NewG),
  mi(NewG,TC,AAtomsOrig).

... % boilerplate code

select_conjunct([G|Gs],_,_,Selected,GBefore,GAfter) :-
  select(Selected,[G|Gs],_Unselected),
  full_eval(FE),
  abstracts(FE,Selected),
  !,
  index_of(Idx,[G|Gs],Selected),
  IdxMin is Idx - 1,
  take(IdxMin,[G|Gs],GBefore),
  drop(Idx,[G|Gs],GAfter).
select_conjunct([G|Gs],TC,AAtomsOrig,Selected,GBefore,GAfter) :-
  select(Selected,[G|Gs],Unselected),
  precedes_all(Selected,Unselected,TC,AAtomsOrig),
  !,
  index_of(Idx,[G|Gs],Selected),
  IdxMin is Idx - 1,
  take(IdxMin,[G|Gs],GBefore),
  drop(Idx,[G|Gs],GAfter).
```

Listing 8.1: Meta-interpreter for termination analysis

8.2 Utilized termination analyzers

Not all of the aforementioned termination analyzers can easily be run on a modern system. Therefore, TermiLog [52] and PolyTool [63] were not considered. The online interfaces were used for cTI [61] and AProVE [35]. For TerminWeb [13], minor changes were applied for compatibility with recent versions of SICStus Prolog. A patch containing these changes is in the electronic appendix. The patched version of TerminWeb was used with SICStus Prolog 4.3.2. All of the considered termination analyzers support builtins to some extent, including the cut and negation.

8.3 Analysis

In practice, the meta-interpreter from listing 8.1 is outside the scope of existing termination analyzers. None were able to prove termination for the meta-interpreter running the permutation sort program with a ground first argument.[2] In fact, none were able to prove termination for even simpler programs consisting only of facts, indicating that the meta-interpreter itself poses a problem for termination analysis.

The essence of the object programs, however, is quite simple. The transformed version of permutation sort can be regarded as an encoding of a simple state machine. This state machine is shown in figure 8.1. Off-the-shelf termination analyzers can indeed guarantee termination for the synthesis, given the top-level query. TerminWeb, cTI and AProVE all guarantee termination. In what follows, we will only discuss the functioning of TerminWeb. This is because its approach is well-suited to our examples and because our main concern is not termination analysis itself, but whether the transformation simplifies further static analysis.

8.3.1 Summary of TerminWeb

TerminWeb operates by approximating the "binary unfoldings semantics" of the source program, which has the same termination characteristics as the operational semantics. For a given program, it defines a set of "binary unfoldings". These binary unfoldings are clauses whose body consists of a single atom or the value *true*. Intuitively, a binary clause $a \leftarrow b$ means that a call to a at any point in a computation implies an eventual subsequent call

[2] To be precise, none were able to prove termination using their default settings. Whether a specific set of parameters could enable a tool to analyze the interpreter is a question left up to the developers of said tools.

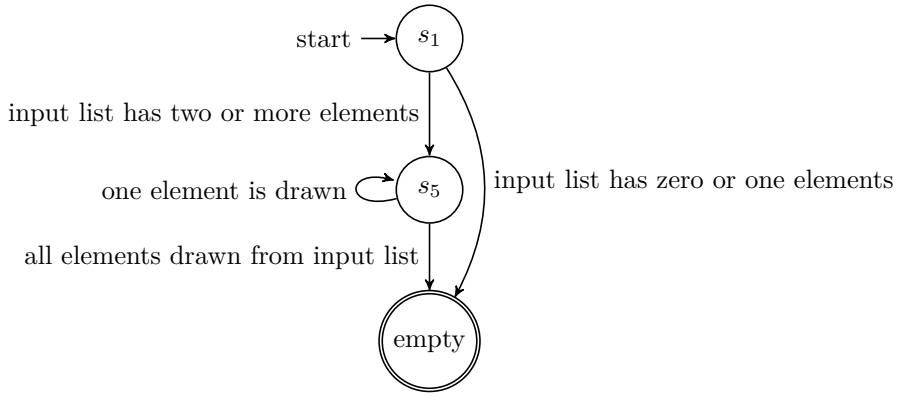


Figure 8.1: State machine encoded by permutation sort synthesis

to b [13]. The binary unfoldings semantics itself is not directly usable for termination analysis, as computing it provides the exact same information about termination behavior as simply running the program. However, by abstracting the program, defining an abstract binary unfoldings semantics and choosing a suitable widening operator, termination characteristics can be safely approximated. The first step is to abstract every term in the program by replacing it by its size according to some "symbolic norm" [52].

Definition 43 (Symbolic norm):
Let $\mathbf{T}(\Sigma, \mathcal{V})$ denote the set of terms constructed using symbols from a vocabulary of function symbols Σ and a set of variables $\mathcal{V}$. Let $\Sigma' = \mathbb{N} \cup \{+/2\}$. A symbolic norm is a function $\| \cdot \|: \mathbf{T}(\Sigma, \mathcal{V}) \to \mathbf{T}(\Sigma', \mathcal{V})$ such that: $\| t \| = c + \Sigma_{i=0}^{n} a_i \| t_i \|$ if $t = f(t_1, \ldots, t_n)$ and $\| t \| = t$ if t is a variable, where c and $a_1, \ldots, a_n$ are non-negative integer constraints depending only on f/n.

For the purpose of analyzing the examples shown up until now, the list length symbolic norm is useful.

Definition 44 (List length symbolic norm):
$\| t \|_{list\ length} = 1 + \| Xs \|_{listlength}$ if $t = [X|Xs]$, $\| t \|_{list\ length} = t$ if t is a variable, $\| t \|_{list\ length} = 0$ otherwise

It is worth pointing out that the definition of the list length symbolic norm given here differs from the common, non-symbolic list length norm given in, e.g., Definition 3.3 in [20] in that variables are mapped to themselves rather than to 0.

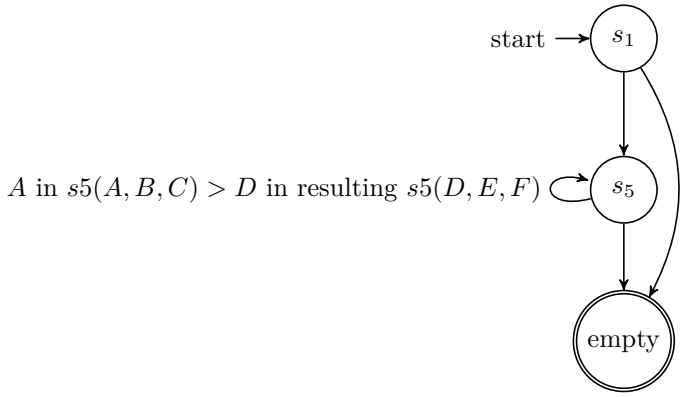


Figure 8.2: State machine encoded by permutation sort synthesis, with a key constraint in the abstract binary unfoldings semantics

The abstract binary unfoldings semantics differs from the binary unfoldings semantics in that it expresses how the size according to the norm of arguments changes between calls. It does this through the use of *constrained* binary clauses, which can contain constraints related to the size of arguments according to the norm. Computing this semantics is still not guaranteed to terminate. Hence, a widening operator is needed. A standard approximation from the abstract interpretation literature can be used for this.

Indeed, from figure 8.2 it is immediately apparent that the list length symbolic norm can show that the synthesis of permutation sort is terminating: the only strongly connected component, i.e. the subgraph in which every vertex can reach every other vertex, has $s5$ as its only vertex and the associated loop reduces a term's size according to the symbolic norm. Because it is impossible for any term's size to become negative, the loop must eventually end and the program must terminate.

More advanced programs which use the *multi* abstraction cannot be handled *directly* by existing termination analyzers using default settings. However, termination can be guaranteed for several of our examples by TerminWeb once structure filtering — a common post-processing step in program transformation — has been applied. These are:

- permutation sort
- graph coloring (including the variant which showcases $multi^{\vee}$)

- N (confused) queens (if only the coroutining part of the program is considered, see section 8.4)

The prime sieve is not terminating. For the continuation passing variant of graph coloring, termination cannot be proven using the out-of-the-box settings. Tweaks to the norm and postprocessing should be enough to remedy this, however, as the variant encodes the same logic as the original. For speculative permutation sort and sameleaves, termination could not be proven using an existing tool with out-of-the-box settings.

As an illustration of what the transitions and constraints considered by TerminWeb look like for a synthesis which relies on *multi*, the diagram in figure 8.3 shows the state machine encoded by a post-processed synthesis of the graph coloring program from listing 4.2 — the simplest example of a synthesis which requires the *multi* abstraction. Note that only the edges of the unique maximal strongly connected component are numbered: non-terminating loops always traverse a cyclical path in a strongly connected component, so only these edges have to be considered. This synthesis is provided in listing 8.2. It is not intended to be read in full, but to be used as a reference for figure 8.3. The reasoning about the constraints involved is more complex, but the key point is that constrained binary clauses do demonstrate that no path can be traversed an infinite number of times. The following constraints match the numbered edges in figure 8.3. The state $s2$ is the state in which a color must be assigned for further testing to take place. The state $s3$ is that in which the most recently assigned color has been tested against some, but not all nations which already have a color. We only show the most informative (in)equalities.:

1. $\|F\| = \|I\|, \|J\| = \|M\|, \|J\| < \|F\|, \|L\| < \|H\|$ for initial state $s2(F, G, H, I)$ and resulting state $s2(J, K, L, M)$. Here, H in the initial state and L in the resulting state (i.e. the third argument) is the list of nations which have not yet been assigned a color. F and J in these two states are the lists of unassigned colors. The transition assigns a color to a nation, so the two inequalities are as expected. The two equalities are between identical terms, so they are also not surprising.

2. $\|J\| < \|F\|, \|L\| < \|G\|$ for initial state $s2(F, G, H, I)$ and resulting state $s3(J, K, L, M, N)$. The list of unassigned colors shrinks, as does the list of nations tested for the most recently assigned color.

3. $\|I\| < \|M\|$ for initial state $s3(G, H, I, J, K)$ and resulting state $s2(L, M, N, O)$. This transition means that the final color is tested. $I < M$ then means that the list of nations ready for testing (whose actual size in this case is 1) is shorter than the list of nations which will be made ready for testing when a color is assigned.

4. $\|H\| < \|M\|, \|N\| < \|I\|$ for initial state $s3(G, H, I, J, K)$ and resulting state $s3(L, M, N, O, P)$. This transition represents the shift of an *allsafe/4* atom from the sequence of immediately callable *allsafe/4* atoms to the suspended ones. H and M correspond the list of non-immediately callable *allsafe*/4 atoms. I and N correspond to the list of immediately callable *allsafe*/4 atoms. In other words, the constraint expresses that one list shrinks, while another grows.

The full output of TerminWeb analyzes paths rather than single edges. Therefore, the above analysis is incomplete, but the purpose of this example is simply to indicate how the analysis translates to observations about the transitions between program states.

```
... % color/1, edge/2 and noedge/2 facts
bi_edge(X,Y) :- edge(X,Y).
bi_edge(Y,X) :- edge(X,Y).
bi_noedge(X,Y) :- noedge(X,Y).
bi_noedge(Y,X) :- noedge(X,Y).

s1([],[]).
s1([N],[C]) :- color(C).
s1([N1,N2|NT],[C1,C2|CT]) :- color(C1),
                             s2([C2|CT],
                                [N1,C1,[N2|NT],[C2|CT]],
                                [N2|NT],[C2|CT]).

s2([C1],
   [N0,C0,[N1],[C1]],
   [N1],[C1]) :- color(C1),
                 test(N0,C0,N1,C1).
s2([C1,C2|C3s],
   [N0,C0,[N1,N2|N3s],[C1,C2|C3s]],
   [N1,N2|N3s],[C1,C2|C3s]) :- color(C1),
                               test(N0,C0,N1,C1),
                               s2([C2|C3s],
                                  [N0,C0,[N2|N3s],[C2|C3s],
                                   N1,C1,[N2|N3s],[C2|C3s]],
                                  [N2|N3s], [C2|C3s]).
s2([C1|C2s],
   [N0,C0,[N1|N2s],[C1|C2s],N01,C01,[N1|N2s],[C1|C2s]|T],
   [N1|N2s],[C1|C2s]) :- color(C1),
```

```
                        test(N0,C0,N1,C1),
                        s3(C2s,
                           [N0,C0,N2s,C2s],
                           [N01,C01,[N1|N2s],[C1|C2s]|T],
                           [N1|N2s],[C1|C2s]).

s3([],
   [N0,C0,[],[]|AST1],
   [N01,C01,[N1],[C1]],
   [N1],[C1]) :- test(N01,C01,N1,C1).
s3([C2|C3s],
   [N0,C0,[N2|N3s],[C2|C3s]|AST1],
   [N01,C01,[N1,N2|N3s],[C1,C2|C3s]],
   [N1,N2|N3s],[C1,C2|C3s]) :- test(N01,C01,N1,C1),
                               s2([C2|C3s],
                                  [N0,C0,[N2|N3s],[C2|C3s],
                                   N1,C1,[N2|N3s],[C2|C3s],
                                   N01,C01,[N2|N3s],[C2|C3s]|AST1],
                                  [N2|N3s],[C2|C3s]).
s3(C2s,
   [N0,C0,N2s,C2s|AST1],
   [N01,C01,[N1|N2s],[C1|C2s],N02,C02,[N1|N2s],[C1|C2s]|AST2],
   [N1|N2s],[C1|C2s]) :- test(N01,C01,N1,C1),
                        my_append(AST1,[N01,C01,N2s,C2s],Appended),
                        s3(C2s,
                           [N0,C0,N2s,C2s|Appended],
                           [N02,C02,[N1|N2s],[C1|C2s]|AST2],
                           [N1|N2s],[C1|C2s]).

test(N,C,N1,C1) :- bi_edge(N,N1), C \== C1.
test(N,C,N1,C1) :- N \== N1, bi_noedge(N,N1).

my_append([],X,X).
my_append([H|T],X,[H|Z]) :-
  my_append(T,X,Z).
```

Listing 8.2: Graph coloring synthesis after structure filtering

The reason why structure filtering is crucial to proving termination of the synthesis of graph coloring is the use of the list length symbolic norm. Without structure filtering, the list length symbolic norm is applied to a concrete multi structure used as an argument of predicates which represent program states. By definition, the list length of any non-list structure is 0. Structure filtering

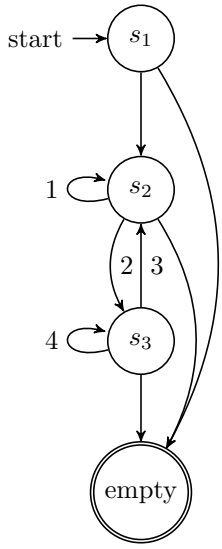


Figure 8.3: State machine encoded by graph coloring synthesis

strips away the redundant concrete functor and enables TerminWeb to discover how atoms are extracted from and added to the multi abstraction as states are traversed. An alternative way to achieve the result could be through the automatic discovery of suitable size norms, as in [10].

8.4 Remarks

The synthesized code for the (confused) queens problem, of which part was given in listing 5.3 (page 95), cannot immediately be analyzed successfully after post-processing by TerminWeb. However, the issue lies with a residual predicate `genlist/2`, which is evaluated once at the start of the program, before any strongly connected component of the state diagram is reached. Termination of the coroutining computation can still be proven by providing a top-level goal which assumes `genlist/2` has successfully been evaluated. Here, too, moving away from a hard-coded norm could help to achieve this result without user intervention.

Chapter 9

Conclusion

9.1 Overview of the contributions

With this work, we have provided several contributions:

- We have recast compiling control as an instance of the abstract conjunctive partial deduction framework. In doing so, we have formally established an abstract domain and fully specified the key operations on this domain.
- We have introduced and formalized the *multi* abstraction.
- We have provided several new examples which demonstrate that the original work on compiling control cannot handle all instantiation-based selection mechanisms.
- We have made several proposals for further abstractions which extend the applicability of compiling control, including variants of the *multi* abstraction and a "speculative" variant of compiling control.
- We have demonstrated how other languages than Prolog, notably CHR, can be a compilation target.
- We have recast the synthesis step of compiling control as an instance of the first Futamura projection.
- We have shown that compiling control facilitates the work of existing analysis tools, notably termination analyzers.

9.2 Future work

9.2.1 Leveraging abstract interpretation

There are examples (see chapter 4) to suggest that a wider range of abstract domains would be useful: a domain expressing freeness, or a universal abstraction representing a grammar of conjunctions have been mentioned. Furthermore, our analysis abstracts the operation of a meta-interpreter, but it is not based on an abstract interpretation of the interpreter itself. To stretch the boundaries of our techniques, a closer integration with the general field of abstract interpretation would be both useful and interesting.

9.2.2 Application to alternate interpreters

We showed that the non-standard semantics of a coroutining meta-interpreter can be compiled in effectively, but somewhat indirectly. It is possible that the non-standard semantics of other meta-interpreters can be compiled in using a similar approach. For instance, a parallelizing interpreter (among others) is put forward as a challenge for program specialization in [69].

9.2.3 Efficient, full automation

The implementation discussed here requires the user to specify how to generalize using *multi*. A prototype which detects applicable linear *multi* abstractions automatically exists and is supplied in the electronic appendix. It was not discussed in the text due to its experimental nature, but it handles the example programs used in this text. It works by analyzing certain recursive calls, but it has its shortcomings: It is hard to prove that it terminates for programs that can indeed be analyzed; For performance reasons, its scope is limited to small programs, with the analysis of our example programs taking several minutes on commodity hardware.

Here, too, it would be useful to generalize from the *multi* abstraction to grammars of conjunctions of arbitrary complexity. In theory, such grammars could be induced using techniques from the field of grammar induction, given sufficient conjunctions produced by the analysis. This would pose several major challenges: *multi* abstractions contain constraints that bind them to their surrounding context, meaning these grammars would be context-sensitive; they would each have to be produced from the right subset of conjunctions; grammar induction is a subfield of machine learning, so it would be difficult to

place upper bounds on the time in which these grammars could be found, if at all. Nonetheless, the prospect of a universal abstraction which can be inferred automatically warrants further investigation.

Another aspect of automation is our approach to full evaluation of abstract atoms. We have assumed that the resulting bindings (or guaranteed failure) are given. This places a burden upon the user. By integrating our implementation with an existing system for abstract interpretation, this burden could be lifted.

This would leave the user with the sole responsibility of selecting atoms, making our technique as accessible as established runtime coroutining mechanisms.

9.2.4 Depth-k abstraction

In this work, we have not focused on depth-k abstraction. We did show how it *can* affect the correctness of the compiled selection rule, but this is not to say that all applications of depth-k abstraction lead to incorrect programs. Specifically, depth-k abstraction which cannot abstract away any information used to perform atom selection is acceptable. We have not examined to what extent harmful applications of depth-k abstraction can be distinguished from harmless ones.

9.2.5 Translating `block` declarations

SICStus Prolog's coroutining mechanism with `block` declarations is a popular one. Therefore, our technique could be widely applied if there was a systematic procedure for translating block declarations into an abstract instantiation-based selection rule. Our current mechanism for concrete selection by an abstract instantiation-based selection rule is not equivalent[1], but variations are possible. This is a specific case in which a more expressive abstract domain would be needed. The abstract domain would, at minimum, allow freeness to be expressed, as `block` declarations distinguish between free and nonfree terms.

[1] This is easy to see: Our mechanism always selects some atom, so it cannot be used for floundering programs.

Appendix A

Example programs

A.1 Prime sieve (Peano encoding)

The following program demonstrates that depth-k abstraction remains inevitable in some cases, but also that it does not always affect control flow.

```
primes(N,P) :- integers(s(s(0)),I),sift(I,P),len(P,N).

integers(N,[]).
integers(N,[N|I]) :- integers(s(N),I).

sift([N|Is],[N|Ps]) :- filter(N,Is,F), sift(F,Ps).
sift([],[]).

divides(N,M) :-
  peano_to_standard(N,N1),
  peano_to_standard(M,M1),
  0 is M1 mod N1.

does_not_divide(N,M) :-
  peano_to_standard(N,N1),
  peano_to_standard(M,M1),
  M1 mod N1 > 0.

filter(N,[M|I],F) :- divides(N,M), filter(N,I,F).
filter(N,[M|I],[M|F]) :- does_not_divide(N,M), filter(N,I,F).
filter(N,[],[]).
```

```
len([],0).
len([H|T],N) :- M is N - 1, len(T,M).

peano_to_standard(0,0).
peano_to_standard(s(X),Res) :-
  peano_to_standard(X,Min),
  Res is Min+1.
```

Listing A.1: Prime sieve with filters using Peano encoding

A.2 Lucky numbers sieve

```
% start with a list of all odd integers from 3 to inf
% sift out every third occurrence, then every seventh
% (7 is non-sifted number after 3), and so on
% when done sifting, check the length of the result
lucky(N,L) :-
  odd_integers(3,I),
  sift(I,3,L),
  len(L,N).

% generate list of integers with two step increment, start with N
odd_integers(N,[]).
odd_integers(N,[N|I]) :-
  N1 is N + 2,
  odd_integers(N1,I).

% repeatedly filter out unlucky occurrences
sift([],_,[]).
sift([L|I],P,[L|S]) :-
  filter(I,L,P,R),
  P1 is P+1,
  sift(R,P1,S).

% filter "unlucky" (i.e. L-th) occurrences of numbers
filter([],_L,_P,[]).
% unlucky occurrence: drop N, next occurrence is 1
filter([N|I],L,L,R) :-
  filter(I,L,1,R).
% lucky occurrence: keep counting up
```

```
filter([N|I],L,P,[N|R]) :-
  L =\= P,
  P1 is P+1,
  filter(I,L,P1,R).

len([],0).
len([H|T],N) :-
  M is N - 1,
  len(T,M).
```

Listing A.2: Lucky numbers sieve

A.3 N queens

```
queens(N,Q) :-
  genlist(N,L),
  perm(L,Q),
  safe(Q).

perm([],[]).
perm([X|Y],[U|V]) :-
  select(U,[X|Y],W),
  perm(W,V).

safe([]).
safe([_X]).
safe([X,Y|Z]) :-
  ndiag(X,1,[Y|Z]),
  safe([Y|Z]).

ndiag(_X,_T,[]).
ndiag(X,T,[Y|Z]) :-
  test(X,T,Y),
  S is T+1,
  ndiag(X,S,Z).

genlist(N,L) :-
  N >= 1,
  genlist_acc(N,[],L).
genlist_acc(N,Acc,L) :-
  N > 1,
  Nmin is N-1,
```

```
  genlist_acc(Nmin,[N|Acc],L).
genlist_acc(N,Acc,[1|Acc]) :-
  N is 1.

test(X,T,Y) :-
  Diff is X-Y,
  abs(Diff,AbsDiff),
  T =\= AbsDiff.
```

Listing A.3: N queens

Appendix B

CHR performance

Table B.1 shows a performance comparison between the (SWI-)Prolog synthesis and the CHR synthesis of permutation sort. For each list size from 1 to 20, we generated 10 lists of random elements between 1 and 10 and computed the average time required to generate all solutions. The fifth and sixth columns shows the relative increase in the number of inferences from the problem size on the next line with respect to the problem size on the current line. Larger problem sizes could not be analyzed on commodity hardware, but the data strongly suggest both syntheses have the same (exponential) asymptotic complexity, with the CHR version being slower by a factor of less than 4. A slowdown is to be expected, as CHR uses Prolog as a host language.

Table B.2 shows a performance comparison between the syntheses of confused queens. Here, there is no need to determine the average number of inferences, as the solutions depend solely on the number of queens. The data suggest both syntheses have the same (exponential) asymptotic complexity, though the Prolog ratio moves towards the asymptotic ratio slightly more quickly. To illustrate this, we have added a column, rratio, which shows how the increase over the previous CHR reading relates to that over the previous Prolog reading. This number seems to be moving towards 1.00. Because it goes down slowly, however, the CHR synthesis has a much larger constant factor.

list size	Inferences (Prolog)	inferences(CHR)	CHR / Prolog	ratio Prolog	ratio CHR
1	21.30	57.50	2.70	1.13	2.75
2	24.00	158.00	6.58	2.11	2.66
3	50.60	420.80	8.32	2.18	2.11
4	110.20	889.80	8.07	2.42	2.28
5	266.20	2,032.30	7.63	2.45	2.27
6	652.00	4,604.70	7.06	2.57	2.40
7	1,676.60	11,032.40	6.58	2.45	2.28
8	4,112.40	25,124.80	6.11	3.01	2.91
9	12,380.20	73,165.20	5.91	2.85	2.74
10	35,319.60	200,775.20	5.68	3.94	4.02
11	139,226.20	806,299.20	5.79	1.62	1.42
12	225,767.60	1,145,682.60	5.07	2.74	2.61
13	618,544.20	2,985,476.80	4.83	3.33	3.31
14	2,057,996.00	9,875,311.00	4.80	4.77	5.05
15	9,818,654.20	49,859,217.20	5.08	2.75	2.55
16	27,045,593.20	127,279,040.20	4.71	1.92	1.79
17	51,887,553.80	228,101,140.80	4.40	3.79	3.73
18	196,899,172.00	850,395,879.00	4.32	1.26	1.17
19	247,625,162.20	992,932,189.20	4.01	4.08	4.02
20	1,010,709,846.00	3,996,547,817.00	3.95		

Table B.1: Performance comparison for permutation sort

Queens	Inferences (Prolog)	Inferences (CHR)	CHR/Prolog	ratio Prolog	ratio CHR	rratio
5	15705	55057	3.51	1.80	12.41	6.90
10	28261	683233	24.18	4.00	4.79	1.20
15	113081	3275297	28.96	2.75	3.17	1.16
20	310581	10390703	33.46	2.22	2.51	1.13
25	690251	26037492	37.72	1.94	2.15	1.11
30	1337851	55953873	41.82	1.76	1.93	1.10
35	2355371	107889137	45.81	1.64	1.78	1.08
40	3861071	191885243	49.70	1.55	1.67	1.08
45	5989441	320557732	53.52	1.48	1.59	1.07
50	8891241	509377313	57.29	1.43	1.53	1.07
55	12733461	776950777	61.02	1.39	1.47	1.06
60	17699361	1145302583	64.71	1.36	1.43	1.06
65	23988431	1640155772	68.37	1.33	1.40	1.05
70	31816431	2291213553	72.01	1.30	1.37	1.05
75	41415351	3132440217	75.63	1.28	1.34	1.05
80	53033451	4202342723	79.24	1.26	1.32	1.05
85	66935221	5544251612	82.83	1.25	1.30	1.04
90	83401421	7206602593	86.41	1.23	1.28	1.04
95	102729041	9243217457	89.98	1.22	1.27	1.04
100	125231341	11713585663	93.54	1.21	1.25	1.04
105	151237811	14683145252	97.09	1.20	1.24	1.04
110	181094211	18223564433	100.63	1.19	1.23	1.04
115	215162531	22413022497	104.17	1.18	1.22	1.03
120	253821031	27336491403	107.70	1.17	1.21	1.03
125	297464201	33086016692	111.23	1.16	1.20	1.03
130	346502801	39760999073	114.75	1.16	1.19	1.03
135	401363821	47468475337	118.27	1.15	1.19	1.03
140	462490521	56323399943	121.78	1.15	1.18	1.03
145	530342391	66448925932	125.29	1.14	1.17	1.03
150	605408035	77976686518	128.80			

Table B.2: Performance comparison for confused queens

Appendix C

Meta-interpreter for specialization

Listing C.1 contains a meta-interpreter which is suited for the specialization of programs requiring a *multi* abstraction. Unlike most programs described in this text, it is not written for SWI-Prolog, but for Ciao Prolog. This is because Logen does not support SWI-Prolog. To make the interpreter (before specialization) run under SWI-Prolog, rename `nth/3` to `nth1/3`.

```
% If any non-builtins are treated as builtins, define them here
% e.g.
% plus(X,Y,Z) :- Z is X + Y.
% minus(X,Y,Z) :- Z is X - Y.
% divides(X,Y) :- 0 is Y mod X.
% does_not_divide(X,Y) :- Z is Y mod X, Z \== 0.

mi([],_).
mi([G|Gs],State) :-
  state_instantiation(State,[G|Gs]),
  selected_index(State,Idx),
  divide_goals([G|Gs],Idx,Before,Selected,After),
  mi_clause(Selected,Body,RuleIdx),
  state_transition(State,NewState,RuleIdx),
  append(Before,Body,NewGsA),
  append(NewGsA,After,NewGs),
  mi(NewGs,NewState).
mi([G|Gs],State) :-
```

```
  state_instantiation(State,[G|Gs]),
  selected_index(State,Idx),
  divide_goals([G|Gs],Idx,Before,Selected,After),
  mi_full_eval(Selected,FullAIIdx),
  call(Selected),
  state_transition(State,NewState,FullAIIdx),
  append(Before,After,NewGs),
  mi(NewGs,NewState).
mi([G|Gs],State) :-
  state_instantiation(State,[G|Gs]),
  selected_index(State,Idx),
  state_transition(State,NewState,1), % 1 is hardcoded to case:one
  divide_goals([G|Gs],Idx,Before,cmulti([BBPatt]),After),
  BBPatt =.. [building_block,Patt],
  append(Before,Patt,NewGsA),
  append(NewGsA,After,NewGs),
  mi(NewGs,NewState).
mi([G|Gs],State) :-
  state_instantiation(State,[G|Gs]),
  selected_index(State,Idx),
  state_transition(State,NewState,2), % 2 is case:many (left)
  divide_goals([G|Gs],Idx,Before,
              cmulti([building_block(BB),BB2|BBs]),After),
  append(Before,BB,NewGsA),
  append(NewGsA,[cmulti([BB2|BBs])],NewGsB),
  append(NewGsB,After,NewGs),
  mi(NewGs,NewState).
mi([G|Gs],State) :-
  state_instantiation(State,[G|Gs]),
  selected_index(State,Idx),
  state_transition(State,NewState,3), % 3 is case:many (right)
  divide_goals([G|Gs],Idx,Before,cmulti([BB1,BB2|BBs]),After),
  length([BB1,BB2|BBs],LL),
  LLM is LL - 1,
  take(LLM,[BB1,BB2|BBs],IBBs),
  last([BB2|BBs],LBB),
  LBB = building_block([LBBH|LBBT]),
  append(Before,[cmulti(IBBs)],NewGsA),
  append(NewGsA,[LBBH|LBBT],NewGsB),
  append(NewGsB,After,NewGs),
  mi(NewGs,NewState).
mi([G|Gs],State) :-
```

```
  state_instantiation(State,[G|Gs]),
  grouping(State,NextState,Groupings),
  apply_groupings([G|Gs],Groupings,NewGs),
  mi(NewGs,NextState).

apply_groupings(Goal,[Grouping|Groupings],NewGoal) :-
  groupings_2_bbs(Goal,[Grouping|Groupings],
                  [ListOfBBs|ListsOfBBs]),
  flatten_groupings([Grouping|Groupings],FlattenedIRs),
  insert_cmultis(Goal,FlattenedIRs,
                 [ListOfBBs|ListsOfBBs],0,NewGoal).

insert_cmultis(Goal,[],[],_,Goal).
insert_cmultis(Goal,[(IRStart,IREndBefore)|IRs],
               [ListOfBBs|ListsOfBBs],Offset,NewGoal) :-
  IRStartPlusOffset is IRStart + Offset,
  IREndBeforePlusOffset is IREndBefore + Offset,
  drop(IREndBeforePlusOffset,Goal,After),
  take(IRStartPlusOffset,Goal,Before),
  append(Before,[cmulti(ListOfBBs)],Intermediate),
  append(Intermediate,After,Replaced),
  NewOffset is Offset + (IREndBefore - IRStart - 1),
  insert_cmultis(Replaced,IRs,ListsOfBBs,NewOffset,NewGoal).

flatten_groupings([],[]).
flatten_groupings([Grouping|Groupings],[FlatIR|FlatIRs]) :-
  flatten_grouping(Grouping,FlatIR),
  flatten_groupings(Groupings,FlatIRs).
flatten_grouping([(IRStart,IREndBefore1)|IRs],
                 (IRStart,IREndBefore)) :-
  last([(IRStart,IREndBefore1)|IRs],(_,IREndBefore)).

groupings_2_bbs(_Goal,[],[]).
groupings_2_bbs(Goal,[Grouping|Groupings],
                [ListOfBBs|ListsOfBBs]) :-
  grouping_2_bbs(Goal,Grouping,ListOfBBs),
  groupings_2_bbs(Goal,Groupings,ListsOfBBs).

grouping_2_bbs(_Goal,[],[]).

grouping_2_bbs(Goal,[(IRStart,IREndBefore)],MultiContents) :-
  IREndBefore is IRStart + 1,
```

```
  IRStartP1 is IRStart + 1,
  nth(IRStartP1,Goal,cmulti(MultiContents)). % nth1 in SWI
grouping_2_bbs(Goal,
               [(IRStart1,IREndBefore1),
                (IRStart2,IREndBefore2)|IRs],
               Appended) :-
  IREndBefore1 is IRStart1 + 1,
  IRStartP1 is IRStart1 + 1,
  nth(IRStartP1,Goal,cmulti(MultiContents)), % nth1 in SWI
  grouping_2_bbs(Goal,[(IRStart2,IREndBefore2)|IRs],Intermediate),
  append(MultiContents,Intermediate,Appended).
grouping_2_bbs(Goal,[(IRStart,IREndBefore)|IRs],Appended) :-
  IREndBefore is IRStart + 1,
  IRStartP1 is IRStart + 1,
  nth(IRStartP1,Goal,Conjunct), % nth1 in SWI
  Conjunct =.. [Functor|_Rest],
  Functor \== cmulti,
  grouping_2_bbs(Goal,IRs,Intermediate),
  append([building_block([Conjunct])],Intermediate,Appended).
grouping_2_bbs(Goal,[(IRStart,IREndBefore)|IRs],Appended) :-
  IRStartP1 is IRStart + 1,
  IREndBefore > IRStartP1,
  take(IREndBefore,Goal,CutOffEnd),
  drop(IRStart,CutOffEnd,BBConjunction),
  grouping_2_bbs(Goal,IRs,Intermediate),
  append([building_block(BBConjunction)],Intermediate,Appended).

divide_goals(Goals,Idx,Before,Selected,After) :-
  length(Before,Idx),
  append(Before,[Selected|After],Goals).

take(0,_,[]).
take(N,[H|T],[H|TakenTail]) :-
  N > 0,
  NMin is N - 1,
  take(NMin,T,TakenTail).

drop(0,Lst,Lst).
drop(N,[_|T],Rest) :-
  N > 0,
  NMin is N - 1,
```

```
drop(NMin,T,Rest).
```

Listing C.1: Extended meta-interpreter for specialization

Appendix D

Meta-interpreter for termination analysis

Listing D.1 shows a meta-interpreter which applies concrete selection by an abstract instantiation-based selection rule (70). It encodes the additional assumption that fully evaluated abstract atoms are always handled before other abstract atoms, but this assumption could easily be lifted. Doing so would only make the selection rule more cumbersome. The listing includes code specific to permutation sort in a clearly demarcated section. The reason for this is twofold: permutation sort as an object program already shows that proving termination is non-trivial for existing analyzers and these analyzers expect single-file programs. The interpreter is flexible enough to run other examples, however.

```
%%%%%%%%%%
%% LOGIC %%
%%%%%%%%%%
mi_clause(permsort(X,Y), [perm(X,Y), ord(Y)]).
mi_clause(perm([],[]), []).
mi_clause(perm([X|Y],[U|V]), [select(U,[X|Y],W), perm(W,V)]).
mi_clause(ord([]),[]).
mi_clause(ord([_]),[]).
mi_clause(ord([X,Y|Z]), [=<(X,Y), ord([Y|Z])]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% CONTROL - must be complete and consistent %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
precedence(perm(_-g,_-a),
           [ord(_-a),
            ord([_-g|_-a])]).
precedence(ord([_-g,_-g|_-a]),
           % first entry is due to definition concrete selection
           [perm(_-a,_-a),
            perm(_-g,_-a)]).

%%%
```

```
mi([G0|Gs0],TC,AAtomsOrig) :-
  select_conjunct([G0|Gs0],TC,AAtomsOrig,Selected,GBefore,GAfter),
  \+((full_eval(FullyEvaluatedPattern),
      abstracts(FullyEvaluatedPattern,Selected))),
  mi_clause(Selected, Body),
  my_append([GBefore,Body,GAfter],NewG),
  mi(NewG,TC,AAtomsOrig).
mi([G0|Gs0],TC,AAtomsOrig) :-
  select_conjunct([G0|Gs0],TC,AAtomsOrig,Selected,GBefore,GAfter),
  full_eval(FullyEvaluatedPattern),
  abstracts(FullyEvaluatedPattern,Selected),
  call(Selected),
  my_append(GBefore,GAfter,NewG),
  mi(NewG,TC,AAtomsOrig).

adjacency_matrix(M) :-
  uniq_aatoms(AAtoms),
  length(AAtoms,Dim),
  length(M,Dim),
  populate_adj_rows(M,AAtoms,AAtoms),
  add_zeroes(M).

populate_adj_rows([],[],_).
populate_adj_rows([Row|Rows],[AAtom|AAtoms],AAtomsOrig) :-
  precedence(AAtom,Precedes0),
  my_append([AAtom],Precedes0,Precedes),
  length(AAtomsOrig,Length),
  length(Row,Length),
  populate_adj_row(Row,Precedes,AAtomsOrig),
  populate_adj_rows(Rows,AAtoms,AAtomsOrig).

populate_adj_rows([Row|Rows],[AAtom|AAtoms],AAtomsOrig) :-
  \+precedence(AAtom,_),
  Precedes = [AAtom],
  length(AAtomsOrig,Length),
  length(Row,Length),
  populate_adj_row(Row,Precedes,AAtomsOrig),
  populate_adj_rows(Rows,AAtoms,AAtomsOrig).

populate_adj_row(_,[],_).
populate_adj_row(R,[P|Ps],AAtomsOrig) :-
  index_of(Idx,AAtomsOrig,P),
```

```
  list_ref(Idx,R,1),
  populate_adj_row(R,Ps,AAtomsOrig).

add_zeroes([]).
add_zeroes([R|Rs]) :-
  add_zeroes_row(R),
  add_zeroes(Rs).

add_zeroes_row([]).
add_zeroes_row([H|T]) :-
  ground(H),
  add_zeroes_row(T).
add_zeroes_row([H|T]) :-
  var(H),
  H = 0,
  add_zeroes_row(T).

construct_abstract_atom((F,N),AA) :-
  length(Args,N),
  maplist([Var,_]>>(Var=_-a),Args,_),
  AA =.. [F|Args].

% constructs the set PA
uniq_aatoms(AAtoms) :-
  findall((F,N),(mi_clause(H,_),H=..[F|Args],length(Args,N)),FNs1),
  findall((F,N),(full_eval(FE),FE=..[F|Args],length(Args,N)),FNs2),
  my_append(FNs1,FNs2,FNs),
  maplist([(F,N),AA]>>construct_abstract_atom((F,N),AA),
          FNs,
          AAtoms0),
  findall(Fst,precedence(Fst,_),AAtoms1),
  findall(Snd,(precedence(_,Lst),member(Snd,Lst)),AAtoms2),
  my_append([AAtoms0,AAtoms1,AAtoms2],AAtoms3),
  filter_uniq(AAtoms3,AAtoms).

filter_uniq(In,Out) :- filter_uniq_aux(In,[],Out).

filter_uniq_aux([],Acc,Acc).
filter_uniq_aux([AAt1|AAts],AccIn,AccOut) :-
  member(OAAt,AccIn),
  copy_term(OAAt,AAt1),
  filter_uniq_aux(AAts,AccIn,AccOut).
```

```
filter_uniq_aux([AAt1|AAts],AccIn,AccOut) :-
  \+((member(OAAt,AccIn),copy_term(OAAt,AAt1))),
  filter_uniq_aux(AAts,[AAt1|AccIn],AccOut).

mmult(M1,M2,M3) :-
  transpose(M2,M2T),
  length(M1,NumRows),
  length(M2T,NumCols),
  length(M3,NumRows),
  lengths(M3,NumCols),
  row_col_multiplications(M1,1,NumRows,M2T,1,NumCols,M3).

tc(M,M) :-
  mmult(M,M,MM),
  normalize_matrix(MM,M).
tc(M,TC) :-
  mmult(M,M,MM),
  normalize_matrix(MM,MMN),
  MMN \== M,
  tc(MMN,TC).
normalize_matrix([],[]) :- !.
normalize_matrix([R|Rs],[NR|NRs]) :-
  normalize_row(R,NR),
  normalize_matrix(Rs,NRs),
  !.
normalize_row([],[]).
normalize_row([0|T],[0|NT]) :-
  normalize_row(T,NT).
normalize_row([N|T],[1|NT]) :-
  N > 0,
  normalize_row(T,NT).

row_col_multiplications(_,M1Idx,NumRows,_,_,_,_) :-
  M1Idx > NumRows.
row_col_multiplications(M1,M1Idx,NumRows,M2T,M2Idx,NumCols,M3) :-
  M2Idx > NumCols,
  M1Idx1 is M1Idx + 1,
  row_col_multiplications(M1,M1Idx1,NumRows,M2T,1,NumCols,M3).
row_col_multiplications(M1,M1Idx,NumRows,M2T,M2Idx,NumCols,M3) :-
  list_ref(M1Idx,M1,Row),
  list_ref(M2Idx,M2T,Col),
  scalar_product(Row,Col,Elem),
```

```
  list_ref(M1Idx,M3,M3Row),
  list_ref(M2Idx,M3Row,Elem),
  M2Idx1 is M2Idx + 1,
  row_col_multiplications(M1,M1Idx,NumRows,M2T,M2Idx1,NumCols,M3).

scalar_product([H1|T1],[H2|T2],P) :-
  Component is H1 * H2,
  scalar_product(T1,T2,Components),
  P is Component + Components.
scalar_product([],[],0).

normalized_equal([],[]) :- !.
normalized_equal([H1|T1],[H2|T2]) :-
  normalized_equal_row(H1,H2),
  normalized_equal(T1,T2),
  !.
normalized_equal_row([],[]).
normalized_equal_row([0|T1],[0|T2]) :-
  normalized_equal_row(T1,T2).
normalized_equal_row([N|T1],[1|T2]) :-
  N > 0, normalized_equal_row(T1,T2).

lengths([],_).
lengths([H|T],L) :-
  length(H,L),
  lengths(T,L).

append_map_unwrap([building_block(Conjunction)],Conjunction).
append_map_unwrap([building_block(Conjunction1),
                   H2|Rest],
                  Unwrapped) :-
  append_map_unwrap([H2|Rest],UnwrappedTail),
  my_append(Conjunction1,UnwrappedTail,Unwrapped).

select_conjunct([G|Gs],_,_,Selected,GBefore,GAfter) :-
  select(Selected,[G|Gs],_Unselected),
  full_eval(FE),
  abstracts(FE,Selected),
  !,
  index_of(Idx,[G|Gs],Selected),
  IdxMin is Idx - 1,
  take(IdxMin,[G|Gs],GBefore),
```

```
  drop(Idx,[G|Gs],GAfter).

select_conjunct([G|Gs],TC,AAtomsOrig,Selected,GBefore,GAfter) :-
  select(Selected,[G|Gs],Unselected),
  precedes_all(Selected,Unselected,TC,AAtomsOrig),
  !,
  index_of(Idx,[G|Gs],Selected),
  IdxMin is Idx - 1,
  take(IdxMin,[G|Gs],GBefore),
  drop(Idx,[G|Gs],GAfter).

take(0,_,[]).
take(N,[H|T],[H|TakenTail]) :-
  N > 0,
  NMin is N - 1,
  take(NMin,T,TakenTail).

drop(0,Lst,Lst).
drop(N,[_|T],Rest) :-
  N > 0,
  NMin is N - 1,
  drop(NMin,T,Rest).

abstracts(Abstr,Concr) :-
  ignore_groundness(Abstr,ConcrAbstr),
  copy_term(ConcrAbstr,ConcrAbstrCopy),
  copy_term(Concr,ConcrCopy),
  ConcrAbstrCopy = ConcrCopy,
  pure_variant(ConcrAbstrCopy, Concr),
  check_groundness(Abstr,Concr).

precedes_all(Selected,Unselected,TC,AAtomsOrig) :-
  member(ASelected,AAtomsOrig),
  abstracts(ASelected,Selected),
  map_to_abstractions(Unselected,UnselectedAbstractions,AAtomsOrig),
  a_precedes_all(ASelected,UnselectedAbstractions,TC,AAtomsOrig).

ignore_groundness(Var-_,Var) :- !.
ignore_groundness(Term,MappedTerm) :-
  Term =.. [C1,C2|Cs],
  !,
  ignore_groundness_lst([C1,C2|Cs],ComponentsWithoutGroundness),
```

```
  MappedTerm =.. ComponentsWithoutGroundness.
ignore_groundness(Term,Term) :-
  Term =.. [Term].

ignore_groundness_lst([],[]).
ignore_groundness_lst([Elem|Elems],[MappedElem|MappedElems]) :-
  ignore_groundness(Elem,MappedElem),
  ignore_groundness_lst(Elems,MappedElems).

% assumes abstraction is at least as general
% as concrete term if groundness is ignored
check_groundness(_-g,Concr) :-
  ground(Concr),
  !.
check_groundness(_-a,_) :- !.
check_groundness(Abstr,Concr) :-
  Abstr =.. [AC1,AC2|ACs],
  nonvar(Concr),
  Concr =.. [C1,C2|Cs],
  check_groundness_all([AC1,AC2|ACs],[C1,C2|Cs]).
check_groundness(Abstr,Concr) :-
  Abstr =.. [Abstr],
  Concr =.. [Abstr].

check_groundness_all([],[]).
check_groundness_all([AT|ATs],[T|Ts]) :-
  check_groundness(AT,T),
  check_groundness_all(ATs,Ts).

abstractions(Conjunct,AAtomsOrig,Abstractions) :-
  findall(Abstr,
          (member(Abstr,AAtomsOrig),abstracts(Abstr,Conjunct)),
          Abstractions).

map_to_abstractions([],[],_).
map_to_abstractions([H|T],[AH|AT],AAtomsOrig) :-
  abstractions(H,AAtomsOrig,AH),
  map_to_abstractions(T,AT,AAtomsOrig).

a_precedes_all(_,[],_,_).
a_precedes_all(ASelected,[UA|UAs],TC,AAtomsOrig) :-
  a_precedes_possibilities(ASelected,UA,TC,AAtomsOrig),
```

```
  a_precedes_all(ASelected,UAs,TC,AAtomsOrig).

a_precedes_possibilities(_,[],_,_).
a_precedes_possibilities(ASel,[P|Ps],TC,AAtomsOrig) :-
  ac_precedes_ac(ASel,P,TC,AAtomsOrig),
  a_precedes_possibilities(ASel,Ps,TC,AAtomsOrig).

ac_precedes_ac(A1,A2,TC,AAtomsOrig) :-
  index_of(Idx1,AAtomsOrig,A1),
  index_of(Idx2,AAtomsOrig,A2),
  list_ref(Idx1,TC,Row),
  list_ref(Idx2,Row,1).

% in the absence of attributed variables, this is correct
% the interpreted program's variables do not use attributes
pure_variant(A,B) :-
  copy_term(A, Ac),
  copy_term(B, Bc),
  numbervars(Ac, 0, N),
  numbervars(Bc, 0, N),
  Ac == Bc.

% makes some assumptions, but these hold for the CC examples
transpose(M,T) :-
  transpose_aux(M,1,[],T).
transpose_aux([R|_],CI,T,T) :-
  length(R,RL),
  CI > RL.
transpose_aux([R|Rs],CI,PT,T) :-
  CIP is CI + 1,
  index_to_col([R|Rs],CI,Col),
  my_append(PT,[Col],NextPT),
  transpose_aux([R|Rs],CIP,NextPT,T).
index_to_col([],_,[]).
index_to_col([R|Rs],Idx,[ElemN|ElemsN]) :-
  list_ref(Idx,R,ElemN),
  index_to_col(Rs,Idx,ElemsN).

% instead of nth1 because the latter is not in ISO
index_of(1,[E|_],E) :- !.
index_of(Idx,[_|T],E):-
  index_of(IdxMin,T,E),
```

```
  !,
  Idx is IdxMin+1.

list_ref(1,[E|_],E).
list_ref(N,[_|T],E) :-
  N > 1,
  NMin is N - 1,
  list_ref(NMin,T,E).
```

Listing D.1: Meta-interpreter with concrete selection by an abstract instantiation-based selection rule

Appendix E

How to reproduce results

E.1 Analysis

For our examples involving only linear *multi* abstractions, results can be reproduced using the implementation, available from `www.programming-emu.com/phd-appendix.zip`. The implementation is written in SWI-Prolog and requires that Prolog environment to be installed. It also requires some additional information from the user (to apply branch failure or generalization), so it is bundled with example programs and input files, e.g. `inputsameleaves.txt`. The latter contain input exactly as it was supplied to the implementation and can be passed to the implementation using the `cat` UNIX program. For example, in the Bash shell, `cat inputsameleaves.txt - | swipl analysis.pl &> outputsameleaves.txt` reproduces our full analysis of sameleaves and stores it in `outputsameleaves.txt`.

Analyses can be stored using `cat` in conjunction with `tee`. For example, `cat - | tee inputsameleaves.txt | swipl analysis.pl` is the command which was used to create our input file for sameleaves.

When an analysis is complete, the implementation will state that there are no conjunctions left to handle.

E.2 Synthesis

Synthesis makes use of Logen, which can be found at `https://github.com/leuschel/logen`. Logen must be built before it can be used. Instructions on how to do so are given in `README.md`. We have made use of Ciao Prolog as its backend language.

After the analysis, run `futamura_interpreter` from inside the SWI-Prolog shell to print the source code for a specializable meta-interpreter (as described in chapter 6). This meta-interpreter can be run before specialization, but the generated code is intended to be compatible with Ciao Prolog, which we assume to be the backend language for Logen. In order to run the meta-interpreter under SWI-Prolog, calls to `nth/3` should be replaced with calls to `nth1/3`. Assuming this interpreter is stored in the main Logen directory as `generated_mi.pl`, generate initial annotations based on a naive binding-time analysis by running `bin/logen -simple_bta unfold generated_mi.pl generated_mi.pl.ann`. The file `generated_mi.pl.ann` contains the annotated version of the program and needs a few changes:

1. For the clause annotations, recursive calls to `mi/2` should be annotated as `memo` rather than unfold.
2. The filter annotation for `mi/2` should be changed to `mi(dynamic,static)`.
3. For the calls in the `mi/2` clause associated with a state transition via clause 3 (unfolding *multi* from the right), starting with `length/2` up to the append operation just before the next call to `mi/2`, annotate as `rescall`.
4. Annotate the final call (to `append/3`) in the third `grouping_2_bbs` clause as `rescall`.
5. Attempt to specialize using `bin/logen generated_mi.pl "mi(Goal,1)"`, where `Goal` is the same as the top-level goal for the analysis, but lacks information about groundness. For example, `bin/logen generated_mi.pl "mi([sameleaves(A,B)],1)"`.
6. Specialization should be successful after performing the steps above. If not, we have provided annotated example files in the electronic appendix. Users can compare their own files with our example files using a program such as `diff` to pinpoint any mistakes. Note that definitions for non-builtin predicates annotated as `rescall` must be added to the resulting program.

Appendix F

Extra syntheses

F.1 Speculative permutation sort

```
permsort(G1,A1) :-
  nonvar(A1),
  A1 =.. ['[|]',G2,G3A2],
  nonvar(G3A2),
  G3A2 =.. ['[|]',G3,A2],
  ground(G2),
  ground(G3),
  G2 =< G3,
  s1(G1,[G2,G3],A2,G3).

permsort([],A1) :-
  A1 == [].
permsort([G],A1) :-
  ground(A1),
  A1 = [G].
permsort(G1,A1) :-
  var(A1),
  q0(G1,A1).
permsort([G3|G4],A1) :-
  nonvar(A1),
  A1 =.. ['[|]',F1,G2],
  var(F1),
  ground(G2),
```

```
  select(F1,[G3|G4],F2),
  ord([F1|G2]), % residual, could unfold further but little point
  perm(F2,G2). % residual, could unfold further but little point
permsort([G2|G3],A1) :-
  nonvar(A1),
  A1 =.. ['[|]',F1,NG1],
  var(F1),
  \+ground(NG1),
  select(F1,[G2|G3],F2),
  s1(F2,[],NG1,F1).
permsort([G2|G3],A1) :-
  nonvar(A1),
  A1 =.. ['[|]',G4,NG1],
  ground(G4),
  \+ground(NG1),
  select(G4,[G2|G3],F1),
  s1(F1,[],NG1,G4).

s1(G1,G2,A1,G3) :-
  nonvar(A1),
  A1 =.. ['[|]',G4,A2],
  ground(G4),
  G3 =< G4,
  append(G2,[G4],G6),
  s1(G1,G6,A2,G4).

s1([G4|G5],[G6|G2s],A1,G3) :-
  var(A1),
  select(G6,[G4|G5],F4),
  s1(F4,G2s,A1,G3).

s1(G1,[],A1,G3) :-
  var(A1),
  q1(G1,A1,G3).

% error potentially caused by =< (for non-numbers)
% is transformed into failure
% could do more checks and generate the error if desired
s1([G5|G6],[G7|G2s],A1,G4) :-
  nonvar(A1),
  A1 =.. ['[|]',NG1,G3],
  var(NG1),
```

```
  ground(G3),
  select(G7,[G5|G6],F4),
  s1(F4,G2s,[NG1|G3],G4).

s1([G5|G6],[],A1,G4) :-
  nonvar(A1),
  A1 =.. ['[|]',F1,G3],
  var(F1),
  ground(G3),
  select(F1,[G5|G6],F3),
  ord([G4,F1|G3]),
  perm(F3,G3).

s1([G4|G5],[G6|G2s],A1,G3) :-
  nonvar(A1),
  A1 =.. ['[|]',F1,NG1],
  var(F1),
  \+ground(NG1),
  select(G6,[G4|G5],F4),
  s1(F4,G2s,[F1|NG1],G3).

s1([G4|G5],[],A1,G3) :-
  nonvar(A1),
  A1 =.. ['[|]',F1,NG1],
  var(F1),
  \+ground(NG1),
  select(F1,[G4|G5],F4),
  G3 =< F1,
  s1(F4,[],NG1,F1).

s1(G1,G2,A1,_G3) :-
  A1 == [],
  perm(G1,G2).

% "CLASSICAL" perm sort with extra state for easy entry
% can use this when LCD control is appropriate
% states begin with a "q" by convention, as opposed to "s"
perm([],[]).
perm([X|Y],[U|V]) :-
  select(U,[X|Y],W), perm(W,V).

ord([]) .
```

```
ord([_]) .
ord([X,Y|Z]) :- X =< Y, ord([Y|Z]).

q0([],[]).
q0([XH|Xs],[YH|Ys]) :-
  select(YH,[XH|Xs],XD),
  q1(XD,Ys,YH).

q1([],[],_Z).
q1([XH|Xs],[YH|Ys],Z) :-
  select(YH,[XH|Xs],XD),
  q2(XD,Ys,Z,YH).

q2(W,X,Y,Z) :-
  Y =< Z,
  q1(W,X,Z).
```

Listing F.1: Speculative permutation sort

F.2 Alternating graph coloring

```
edge(belgium, france).
edge(belgium, germany).
edge(belgium, luxembourg).
edge(belgium, netherlands).
edge(czechrepublic, germany).
edge(denmark, germany).
edge(france, germany).
edge(france, luxembourg).
edge(france, switzerland).
edge(germany, luxembourg).
edge(germany, netherlands).
edge(germany, switzerland).
noedge(belgium, czechrepublic).
noedge(belgium, denmark).
noedge(belgium, switzerland).
noedge(czechrepublic, denmark).
noedge(czechrepublic, france).
noedge(czechrepublic, luxembourg).
noedge(czechrepublic, netherlands).
noedge(czechrepublic, switzerland).
noedge(denmark, france).
```

```
noedge(denmark, luxembourg).
noedge(denmark, netherlands).
noedge(denmark, switzerland).
noedge(france, netherlands).
noedge(luxembourg, netherlands).
noedge(luxembourg, switzerland).
noedge(netherlands, switzerland).

bi_edge(X,Y) :- edge(X,Y).
bi_edge(Y,X) :- edge(X,Y).
bi_noedge(X,Y) :- noedge(X,Y).
bi_noedge(Y,X) :- noedge(X,Y).

color(red).
color(green).
color(blue).
color(orange).

test(N,C,N1,C1) :-
  bi_edge(N,N1), C \== C1 ;
  N \== N1, bi_noedge(N,N1).

qS(safe_coloring([],[])).
qS(safe_coloring([_],[G2])) :-
  color(G2).
qS(safe_coloring([G4,G5|G6],[G2,G3|A5])) :-
  color(G2),
  color(G3),
  test(G4,G2,G5,G3),
  (qB(coloring(A5),
      multior([building_block([allsafeA(G4,G2,G6,A5)],1)]),
      safe([G5|G6],[G3|A5])) ;
   qB(coloring(A5),
      multior([building_block([allsafeB(G4,G2,G6,A5)],2)]),
      safe([G5|G6],[G3|A5]))).

qB(coloring([]),multior(BBs),safe([G5|G6],[G3])) :-
  qC(multior(BBs),safe([G5|G6],[G3])).
qB(coloring([G7|A8]),
   multior([building_block([allsafeA(G8,G9,[G10|G11],[G7|A8])],1)]),
   safe([G5,G10|G11],[G3,G7|A8])) :-
  color(G7),
```

```
  test(G8,G9,G10,G7),
  qD(coloring(A8),
     multior([building_block([allsafeA(G8,G9,G11,A8)],1)]),
     safe([G5,G10|G11],[G3,G7|A8])).
qB(coloring([G7|A8]),
   multior([building_block([allsafeB(G8,G9,[G10|G11],[G7|A8])],2)]),
   safe([G5,G10|G11],[G3,G7|A8])) :-
  color(G7),
  test(G8,G9,G10,G7),
  qD(coloring(A8),
     multior([building_block([allsafeB(G8,G9,G11,A8)],2)]),
     safe([G5,G10|G11],[G3,G7|A8])).
qB(coloring([G7|A8]),
   multior([building_block([allsafeA(G8,G9,[G10|G11],[G7|A8])],1),
            building_block(BB,BBT)|BBs]),
   safe([G5,G10|G11],[G3,G7|A8])) :-
  color(G7),
  test(G8,G9,G10,G7),
  qE(coloring(A8),
     multior([building_block([allsafeA(G8,G9,G11,A8)],1)]),
     multior([building_block(BB,BBT)|BBs]),
     safe([G5,G10|G11],[G3,G7|A8])).
qB(coloring([G7|A8]),
   multior([building_block([allsafeB(G8,G9,[G10|G11],[G7|A8])],2),
            building_block(BB,BBT)|BBs]),
   safe([G5,G10|G11],[G3,G7|A8])) :-
  color(G7),
  test(G8,G9,G10,G7),
  qE(coloring(A8),
     multior([building_block([allsafeB(G8,G9,G11,A8)],2)]),
     multior([building_block(BB,BBT)|BBs]),
     safe([G5,G10|G11],[G3,G7|A8])).

qC(multior([building_block([allsafeA(_,_,[],[])],1)]),safe([_],[_])).
qC(multior([building_block([allsafeB(_,_,[],[])],2)]),safe([_],[_])).
qC(multior([building_block([allsafeA(_,_,[],[])],1),
            building_block(BB,BBT)|BBs]),safe([G5],[G3])) :-
  qC(multior([building_block(BB,BBT)|BBs]),safe([G5],[G3])).
qC(multior([building_block([allsafeB(_,_,[],[])],2),
            building_block(BB,BBT)|BBs]),safe([G5],[G3])) :-
  qC(multior([building_block(BB,BBT)|BBs]),safe([G5],[G3])).
```

```
qD(coloring(A8),multior(BBs),safe([G5,G10|G11],[G3,G7|A8])) :-
  test(G5,G3,G10,G7),
  append(BBs,[building_block([allsafeA(G5,G3,G11,A8)],1)],NewBBs),
  qB(coloring(A8),
     multior(NewBBs),
     safe([G10|G11],[G7|A8])).
qD(coloring(A8),multior(BBs),safe([G5,G10|G11],[G3,G7|A8])) :-
  test(G5,G3,G10,G7),
  append(BBs,[building_block([allsafeB(G5,G3,G11,A8)],2)],NewBBs),
  qB(coloring(A8),
     multior(NewBBs),
     safe([G10|G11],[G7|A8])).

qE(coloring(A8),
   multior(BBs1),
   multior([building_block([allsafeA(G8,G9,[G10|G11],[G7|A8])],1)]),
   safe([G5,G10|G11],[G3,G7|A8])) :-
  test(G8,G9,G10,G7),
  append(BBs1,[building_block([allsafeA(G8,G9,G11,A8)],1)],NewBBs),
  qD(coloring(A8),multior(NewBBs),safe([G5,G10|G11],[G3,G7|A8])).
qE(coloring(A8),
   multior(BBs1),
   multior([building_block([allsafeB(G8,G9,[G10|G11],[G7|A8])],2)]),
   safe([G5,G10|G11],[G3,G7|A8])) :-
  test(G8,G9,G10,G7),
  append(BBs1,[building_block([allsafeB(G8,G9,G11,A8)],2)],NewBBs),
  qD(coloring(A8),multior(NewBBs),safe([G5,G10|G11],[G3,G7|A8])).
qE(coloring(A8),
   multior(BBs1),
   multior([building_block([allsafeA(G8,G9,[G10|G11],[G7|A8])],1),
            building_block(BB,BBT)|BBs2]),
   safe([G5,G10|G11],[G3,G7|A8])) :-
  test(G8,G9,G10,G7),
  append(BBs1,[building_block([allsafeA(G8,G9,G11,A8)],1)],NewBBs),
  qE(coloring(A8),
     multior(NewBBs),
     multior([building_block(BB,BBT)|BBs2]),
     safe([G5,G10|G11],[G3,G7|A8])).
qE(coloring(A8),
   multior(BBs1),
   multior([building_block([allsafeB(G8,G9,[G10|G11],[G7|A8])],2),
            building_block(BB,BBT)|BBs2]),
```

```
   safe([G5,G10|G11],[G3,G7|A8])) :-
  test(G8,G9,G10,G7),
  append(BBs1,[building_block([allsafeB(G8,G9,G11,A8)],2)],NewBBs),
  qE(coloring(A8),
     multior(NewBBs),
     multior([building_block(BB,BBT)|BBs2]),
     safe([G5,G10|G11],[G3,G7|A8])).
```

Listing F.2: Alternating graph coloring synthesis

F.3 Continuation passing graph coloring

```
edge(belgium, france).
edge(belgium, germany).
edge(belgium, luxembourg).
edge(belgium, netherlands).
edge(czechrepublic, germany).
edge(denmark, germany).
edge(france, germany).
edge(france, luxembourg).
edge(france, switzerland).
edge(germany, luxembourg).
edge(germany, netherlands).
edge(germany, switzerland).
noedge(belgium, czechrepublic).
noedge(belgium, denmark).
noedge(belgium, switzerland).
noedge(czechrepublic, denmark).
noedge(czechrepublic, france).
noedge(czechrepublic, luxembourg).
noedge(czechrepublic, netherlands).
noedge(czechrepublic, switzerland).
noedge(denmark, france).
noedge(denmark, luxembourg).
noedge(denmark, netherlands).
noedge(denmark, switzerland).
noedge(france, netherlands).
noedge(luxembourg, netherlands).
noedge(luxembourg, switzerland).
noedge(netherlands, switzerland).

bi_edge(X,Y) :- edge(X,Y).
```

```
bi_edge(Y,X) :- edge(X,Y).
bi_noedge(X,Y) :- noedge(X,Y).
bi_noedge(Y,X) :- noedge(X,Y).

color(red).
color(green).
color(blue).
color(orange).

test(N,C,N1,C1) :-
  bi_edge(N,N1), C \== C1 ;
  N \== N1, bi_noedge(N,N1).

q0(safe_coloring([],[])).
q0(safe_coloring([_],[C1])) :-
  color(C1).
q0(safe_coloring([G4,G5|G6],[G2,G3|A5])) :-
  color(G2),
  color(G3),
  q1(coloring(A5),
     safe_([G4,G5|G6],[G2,G3|A5],done,_A6)).

q1(coloring(A5),
   safe_([G4,G5|G6],[G2,G3|A5],done,A6)) :-
  q2(coloring(A5),
     process(frame(G4,G2,[G5|G6],[G3|A5],done),done,A6),
     safe_([G5|G6],[G3|A5],A6,_A7)).
q1(coloring(A5),
   safe_([G4,G5|G6],
         [G2,G3|A5],
         frame(G7,G8,[G5|G6],[G3|A5],Complex1),A6)) :-
   q2(coloring(A5),
      process(frame(G4,G2,[G5|G6],[G3|A5],
                    frame(G7,G8,[G5|G6],[G3|A5],Complex1)),
              done,A6),
      safe_([G5|G6],[G3|A5],A6,_A7)).

q2(coloring([G7|A9]),
   process(done,done,done),
   safe_([G5,G8|G9],[G3,G7|A9],done,A7)) :-
  color(G7),
  q1(coloring(A9),
```

```
    safe_([G5,G8|G9],[G3,G7|A9],done,A7)).
q2(coloring([]),
   process(done,done,done),
   safe_([_G5],[_G3],done,done)).
q2(coloring([G9|A9]),
   process(done,
          frame(G7,G8,[G10|G11],[G9|A9],Complex3),
          frame(G7,G8,[G10|G11],[G9|A9],Complex3)),
   safe_([G5,G10|G11],[G3,G9|A9],
        frame(G7,G8,[G10|G11],[G9|A9],Complex3),A7)) :-
  color(G9),
  q1(coloring(A9),
     safe_([G5,G10|G11],[G3,G9|A9],
          frame(G7,G8,[G10|G11],[G9|A9],Complex3),A7)).
q2(coloring([]),
   process(done,
          frame(G7,G8,[],[],Complex),
          frame(G7,G8,[],[],Complex)),
   safe_([_G5],[_G3],frame(G7,G8,[],[],Complex),_A7)).
q2(coloring(A5),
   process(frame(G7,G8,[G5|G6],[G3|A5],Complex3),done,A6),
   safe_([G5|G6],[G3|A5],A6,A7)) :-
  test(G7,G8,G5,G3),
  q2(coloring(A5),
     process(Complex3,frame(G7,G8,G6,A5,done),A6),
     safe_([G5|G6],[G3|A5],A6,A7)).
q2(coloring(A5),
   process(frame(G7,G8,[G5|G6],[G3|A5],Complex3),
          frame(G9,G10,G6,A5,Complex4),A6),
   safe_([G5|G6],[G3|A5],A6,A7)) :-
  test(G7,G8,G5,G3),
  q2(coloring(A5),
     process(Complex3,
            frame(G7,G8,G6,A5,frame(G9,G10,G6,A5,Complex4)),
            A6),
     safe_([G5|G6],[G3|A5],A6,A7)).
```

Listing F.3: Continuation passing graph coloring synthesis

F.4 Complete Prolog synthesis of confused queens

```
genlist(N,L) :- N >= 1, genlist_acc(N,[],L).
```

```
genlist_acc(N,Acc,L) :- N > 1, Nmin is N-1, genlist_acc(Nmin,[N|Acc],L).
genlist_acc(N,Acc,[1|Acc]) :- N is 1.

attack(A,_,A).
attack(A,Offset,B) :-
  Diff is A - B,
  abs(Diff,Offset).

queens(0,[]).
queens(A,[D|E]) :-
  genlist(A,C),
  A > 0,
  F is A - 1,
  member(D,C),
  a(draw(F,C,E),confused([D|E])).

a(draw(0,_,[]),confused([_])).
a(draw(A,B,[E|F]),confused([D,E|F])) :-
  A >= 0,
  G is A - 1,
  member(E,B),
  b(draw(G,B,F),
    multi([attack_all(D,1,[E|F])]),
    confused([E|F])).

b(draw(A,B,C),multi([attack_all(D,E,[F|C])]),confused([F|C])) :-
  H is E + 1,
  attack(D,E,F),
  c(draw(A,B,C),
    multi([attack_all(D,H,C)]),
    confused([F|C])).
b(draw(A,B,C),multi([attack_all(D,E,[F|C]),
  attack_all(H,I,[F|C])|J]),confused([F|C])) :-
  K is E + 1,
  attack(D,E,F),
  d(draw(A,B,C),multi([attack_all(D,K,C)]),
    multi([attack_all(H,I,[F|C])|J]),confused([F|C])).

d(draw(A,B,C),multi([attack_all(D,E,C)|F]),
  multi([attack_all(G,H,[I|C])]),confused([I|C])) :-
  K is H + 1,
  attack(G,H,I),
```

```
  append([attack_all(D,E,C)|F],[attack_all(G,K,C)],
         Appended),
  c(draw(A,B,C),multi(Appended),confused([I|C])).
d(draw(A,B,C),multi([attack_all(D,E,C)|F]),
  multi([attack_all(G,H,[I|C]),attack_all(K,L,[I|C])|M]),
  confused([I|C])) :-
  N is H + 1,
  attack(G,H,I),
  append([attack_all(D,E,C)|F],[attack_all(G,N,C)],
         Appended),
  d(draw(A,B,C),multi(Appended),
    multi([attack_all(K,L,[I|C])|M]),confused([I|C])).

c(draw(0,B,[]),multi([attack_all(D,E,[])|F]),confused([G])) :-
  e(multi([attack_all(D,E,[])|F]),confused([G])).
c(draw(A,B,[H|I]),multi([attack_all(D,E,[H|I])|F]),confused([G,H|I])) :-
  A > 0,
  J is A - 1,
  member(H,B),
  append([attack_all(D,E,[H|I])|F],
         [attack_all(G,1,[H|I])],Appended),
  b(draw(J,B,I),multi(Appended),confused([H|I])).

e(multi([attack_all(A,B,[])]),confused([Z])).
e(multi([attack_all(A,B,[]),
  attack_all(C,D,[])|E]),confused([Z])) :-
  e(multi([attack_all(C,D,[])|E]),confused([Z])).
```

Listing F.4: Complete Prolog synthesis of confused queens

F.5 Procedural permutation sort

The following program is an implementation of permutation sort in Python 3, obtained using the principles behind the CHR synthesis. Using Pypy3, its runtime performance is roughly 2.5 times better than that of the Prolog synthesis run using SWI-Prolog. However, the comparison could be biased towards either language, as this synthesis provides its own counterpart to Prolog's *select*/3 predicate.

```
def permsort(in_lst):
    for outcome in transition_to_success(
```

```
    # the None values correspond to free variables
        (1,in_lst,None,None,None,None)):
        # the first None will be replaced with the result
        box = outcome[2]
        res = []
        while box:
            if box.head:
                res.append(box.head)
            box = box.remaining_permutation
        print(res)

def transition_to_success(in_state):
    r"""
    Generator function which yields all success states.
    """
    if in_state[0] == "done":
        yield in_state
    elif in_state[0] == "fail":
        return
    else:
        for successor in get_successor_states(in_state):
            yield from transition_to_success(successor)

def select(in_lst):
    r"""
    Emulates the select/3 Prolog builtin.
    Not entirely faithful in case there are duplicates in the list,
    but irrelevant for this program.
    """
    for elem in enumerate(in_lst):
        lst_copy = list(in_lst)
        del(lst_copy[elem[0]])
        yield(elem[1],lst_copy)

def get_successor_states(state):
    (state_num,
     r1_2_arg1,
     r1_2_arg2,
     r3_4_arg1,
     r3_4_arg2,
     r3_4_arg3) = state
    if state_num == 1:
```

```
        if r1_2_arg1 == []:
            yield ("done",r1_2_arg1,Box(None),None,None,None)
        else:
           for (selected,remaining) in select(r1_2_arg1):
               box = Box(selected)
               yield (5, # next state number
                      r1_2_arg1, # never changes
                      box, # wrapper as "free variable"
                      remaining, # still to be permuted
                      box, # tail should be assigned
                      box) # first two elements should be compared
    elif state_num == 5:
        if r3_4_arg1 == []:
            r3_4_arg2.remaining_permutation = Box(None)
            yield ("done",
                   r1_2_arg1,
                   r1_2_arg2,
                   r3_4_arg1,
                   r3_4_arg2,
                   r3_4_arg3)
        else:
            for (selected,remaining) in select(r3_4_arg1):
                box2 = Box(selected)
                if r3_4_arg2.head > selected:
                    continue # won't lead to success state
                r3_4_arg2.remaining_permutation = box2
                yield(5,r1_2_arg1,r1_2_arg2,remaining,box2,box2)

class Box:
    r"""A wrapper class to emulate a pointer to a list."""
    def __init__(self,head):
        self.head = head
        self.remaining_permutation = None
```

Bibliography

[1] Apt, K. R., and Wallace, M. *Constraint logic programming using Eclipse*. Cambridge University Press, 2007.

[2] Baader, F., and Nipkow, T. *Term rewriting and all that*. Cambridge University Press, 1998.

[3] Benkerimi, K., and Hill, P. M. Supporting transformations for the partial evaluation of logic programs. *J. Log. Comput. 3*, 5 (1993), 469–486.

[4] Boucher, D., and Feeley, M. Abstract compilation: A new implementation paradigm for static analysis. In *Compiler Construction, 6th International Conference, CC'96, Linköping, Sweden, April 24-26, 1996, Proceedings* (1996), T. Gyimóthy, Ed., vol. 1060 of *Lecture Notes in Computer Science*, Springer, pp. 192–207.

[5] Boulanger, D., Bruynooghe, M., and De Schreye, D. Compiling control revisited: A new approach based upon abstract interpretation for constraint logic programs. In *Proceedings of the 5th Workshop on Logic Programming Environments (LPE 1993), October 29-30, 1993, In conjunction with ILPS 1993, Vancouver, British Columbia, Canada* (1993), pp. 39–51.

[6] Bratko, I. *Prolog Programming for Artificial Intelligence, 4th Edition*. Addison-Wesley, 2012.

[7] Bruynooghe, M. A practical framework for the abstract interpretation of logic programs. *J. Log. Program. 10*, 2 (1991), 91–124.

[8] Bruynooghe, M., De Schreye, D., and Krekels, B. Compiling control. In *Proceedings of the 1986 Symposium on Logic Programming, Salt Lake City, Utah, USA, September 22-25, 1986* (1986), IEEE-CS, pp. 70–77.

[9] Bruynooghe, M., De Schreye, D., and Krekels, B. Compiling control. *J. Log. Program. 6*, 1&2 (1989), 135–162.

[10] Bruynooghe, M., Gallagher, J., and Van Humbeeck, W. Inference of well-typings for logic programs with application to termination analysis. In *International Static Analysis Symposium* (2005), Springer, pp. 35–51.

[11] Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., and Sjöland, T. *SICStus Prolog user's manual*, vol. 3. Swedish Institute of Computer Science Kista, Sweden, 1988.

[12] Christensen, N. H., and Glück, R. Offline partial evaluation can be as accurate as online partial evaluation. *ACM Trans. Program. Lang. Syst. 26*, 1 (2004), 191–220.

[13] Codish, M., and Taboch, C. A semantic basis for the termination analysis of logic programs. *J. Log. Program. 41*, 1 (1999), 103–123.

[14] Colmerauer, A., et al. Prolog ii: Reference manual and theoretical model. *Groupe D'intelligence Artificielle, Faculté Des Sciences De Luminy, Marseille* (1982).

[15] Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. Un systeme de communication homme-machine en francais. Tech. rep., Technical report, Groupe de Intelligence Artificielle Universite de Aix-Marseille II, 1973.

[16] Conway, M. E. Design of a separable transition-diagram compiler. *Commun. ACM 6*, 7 (1963), 396–408.

[17] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977* (1977), R. M. Graham, M. A. Harrison, and R. Sethi, Eds., ACM, pp. 238–252.

[18] Craig, S., Gallagher, J. P., Leuschel, M., and Henriksen, K. S. Fully automatic binding-time analysis for Prolog. In *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers* (2004), S. Etalle, Ed., vol. 3573 of *Lecture Notes in Computer Science*, Springer, pp. 53–68.

[19] De Schreye, D., and Bruynooghe, M. On the transformation of logic programs with instantiation based computation rules. *J. Symb. Comput. 7*, 2 (1989), 125–154.

[20] De Schreye, D., and Decorte, S. Termination of logic programs: The never-ending story. *J. Log. Program. 19/20* (1994), 199–260.

[21] De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., and Sørensen, M. H. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *J. Log. Program. 41*, 2-3 (1999), 231–277.

[22] De Schreye, D., Nys, V., and Nicholson, C. J. Analysing and Compiling Coroutines with Abstract Conjunctive Partial Deduction. In *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers* (Canterbury, UK, 2014), Lecture Notes in Computer Science, Springer, pp. 21–38.

[23] Demoen, B., Nguyen, P.-L., Schrijvers, T., and Tronçon, R. *The first 10 Prolog programming contests.* Bart Demoen, 2005.

[24] Desouter, B., van Dooren, M., Schrijvers, T., and Vandenbroucke, A. Tabling as a library with delimited control. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016* (2016), S. Kambhampati, Ed., IJCAI/AAAI Press, pp. 4145–4149.

[25] Duck, G. J., Stuckey, P. J., de la Banda, M. J. G., and Holzbaur, C. The refined operational semantics of Constraint Handling Rules. In *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings* (2004), B. Demoen and V. Lifschitz, Eds., vol. 3132 of *Lecture Notes in Computer Science*, Springer, pp. 90–104.

[26] Felleisen, M. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988* (1988), J. Ferrante and P. Mager, Eds., ACM Press, pp. 180–190.

[27] Frühwirth, T. Programming in constraint handling rules.

[28] Frühwirth, T. *Constraint Handling Rules.* Aug. 2009.

[29] Frühwirth, T. W. Proving termination of constraint solver programs. In *New Trends in Contraints, Joint ERCIM/Compulog Net Workshop, Paphos, Cyprus, October 25-27, 1999, Selected Papers* (1999), K. R. Apt, A. C. Kakas, E. Monfroy, and F. Rossi, Eds., vol. 1865 of *Lecture Notes in Computer Science*, Springer, pp. 298–317.

[30] Frühwirth, T. W. As time goes by: Automatic complexity analysis of simplified rules. In *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02),*

Toulouse, France, April 22-25, 2002 (2002), D. Fensel, F. Giunchiglia, D. L. McGuinness, and M. Williams, Eds., Morgan Kaufmann, pp. 547–557.

[31] Futamura, Y. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation 12*, 4 (Dec 1999), 381–391.

[32] Gallagher, J. P. Transforming logic programs by specialising interpreters. In *ECAI* (1986), pp. 313–326.

[33] Gallagher, J. P. Tutorial on specialisation of logic programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993* (1993), D. A. Schmidt, Ed., ACM, pp. 88–98.

[34] Gallagher, J. P., and Bruynooghe, M. Some low-level source transformations of logic programs. In *Proceedings of the Second Workshop on Meta-programming in Logic* (1990), pp. 229–244.

[35] Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., and Thiemann, R. Analyzing program termination and complexity automatically with aprove. *J. Autom. Reasoning 58*, 1 (2017), 3–31.

[36] Gómez-Zamalloa, M., Albert, E., and Puebla, G. Decompilation of Java bytecode to Prolog by partial evaluation. *Information & Software Technology 51*, 10 (2009), 1409–1427.

[37] Heaton, A., Hill, P. M., and King, A. Analysis of logic programs with delay. In *Logic Programming Synthesis and Transformation, 7th International Workshop, LOPSTR'97, Leuven, Belgium, July 10-12, 1997, Proceedings* (1997), N. E. Fuchs, Ed., vol. 1463 of *Lecture Notes in Computer Science*, Springer, pp. 148–167.

[38] Henriksen, K. S., and Gallagher, J. P. Abstract interpretation of PIC programs through logic programming. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006), 27-29 September 2006, Philadelphia, Pennsylvania, USA* (2006), IEEE Computer Society, pp. 184–196.

[39] Hill, P. M., and Gallagher, J. P. *Handbook of Logic in Artificial Intelligence, volume 5*. 1998, ch. Meta-Programming in Logic Programming, pp. 421–498.

[40] Hruza, J., and Stepánek, P. Speedup of logic programs by binarization and partial deduction. *TPLP 4*, 3 (2004), 355–380.

[41] Jones, N. D. Transformation by interpreter specialisation. *Science of Computer Programming 52*, 1-3 (2004), 307–339.

[42] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial evaluation and automatic program generation.* Prentice Hall international series in computer science. Prentice Hall, 1993.

[43] Komorowski, H. J. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982* (1982), R. A. DeMillo, Ed., ACM Press, pp. 255–267.

[44] Kowalski, R. A. Predicate logic as programming language. In *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974.* (1974), J. L. Rosenfeld, Ed., North-Holland, pp. 569–574.

[45] Leuschel, M. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst. 26*, 3 (2004), 413–463.

[46] Leuschel, M., and Bruynooghe, M. Logic program specialisation through partial deduction: Control issues. *TPLP 2*, 4-5 (2002), 461–515.

[47] Leuschel, M., Craig, S., Bruynooghe, M., and Vanhoof, W. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic: A Decade of Research Advances in Logic-Based Program Development*, M. Bruynooghe and K. Lau, Eds., vol. 3049 of *Lecture Notes in Computer Science*. Springer, 2004, pp. 340–375.

[48] Leuschel, M., Elphick, D., Varea, M., Craig, S., and Fontaine, M. The ecce and logen partial evaluators and their web interfaces. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006* (2006), J. Hatcliff and F. Tip, Eds., ACM, pp. 88–94.

[49] Leuschel, M., Jørgensen, J., Vanhoof, W., and Bruynooghe, M. Offline specialisation in Prolog using a hand-written compiler generator. *CoRR cs.PL/0208009* (2002).

[50] Leuschel, M., and Martens, B. Global control for partial deduction through characteristic atoms and global trees. In *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers* (1996), pp. 263–283.

[51] Leuschel, M., Martens, B., and De Schreye, D. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Trans. Program. Lang. Syst. 20*, 1 (1998), 208–258.

[52] Lindenstrauss, N., and Sagiv, Y. Automatic termination analysis of logic programs. In *Logic Programming, Proceedings of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, July 8-11, 1997* (1997), L. Naish, Ed., MIT Press, pp. 63–77.

[53] Lloyd, J. W. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[54] Lloyd, J. W., and Shepherdson, J. C. Partial evaluation in logic programming. *J. Log. Program. 11*, 3&4 (1991), 217–242.

[55] Marlin, C. D. *Coroutines: A Programming Methodology, a Language Design and an Implementation*, vol. 95 of *Lecture Notes in Computer Science*. Springer, 1980.

[56] Marriott, K., de la Banda, M. J. G., and Hermenegildo, M. V. Analyzing logic programs with dynamic scheduling. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994* (1994), H. Boehm, B. Lang, and D. M. Yellin, Eds., ACM Press, pp. 240–253.

[57] Martelli, A., and Montanari, U. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2 (1982), 258–282.

[58] Martens, B., De Schreye, D., and Horváth, T. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science 122*, 1 (1994), 97–117.

[59] Meert, W., Nys, V., Theunis, R., Fannes, T., Ingels, M., Guns, T., Van den Broeck, G., Driessens, K., De Schreye, D., and Verhelst, M. Innovation lab@ KU Leuven: Education, engineering and artificial intelligence.

[60] Mellish, C. Abstract interpretation of prolog programs. In *Third International Conference on Logic Programming, Imperial College of Science and Technology, London, United Kingdom, July 14-18, 1986,*

Proceedings (1986), E. Shapiro, Ed., vol. 225 of *Lecture Notes in Computer Science*, Springer, pp. 463–474.

[61] Mesnard, F., and Bagnara, R. cti: A constraint-based termination inference tool for iso-prolog. *Theory and Practice of Logic Programming 5*, 1-2 (2005), 243–257.

[62] Mesnard, F., and Neumerkel, U. Applying static analysis techniques for inferring termination conditions of logic programs. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings* (2001), P. Cousot, Ed., vol. 2126 of *Lecture Notes in Computer Science*, Springer, pp. 93–110.

[63] Nguyen, M. T., and De Schreye, D. Polytool: Proving termination automatically based on polynomial interpretations. In *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers* (2006), G. Puebla, Ed., vol. 4407 of *Lecture Notes in Computer Science*, Springer, pp. 210–218.

[64] Nys, V. Advances in analyzing coroutines by abstract conjunctive partial deduction. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015.* (2015), M. D. Vos, T. Eiter, Y. Lierler, and F. Toni, Eds., vol. 1433 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[65] Nys, V., and De Schreye, D. Abstract conjunctive partial deduction for the analysis and compilation of coroutines. *Formal Aspects of Computing 29*, 1 (Jan 2017), 125–153.

[66] Nys, V., and De Schreye, D. Transforming coroutining logic programs into equivalent CHR programs. In *Proceedings Fifth International Workshop on Verification and Program Transformation, VPT@ETAPS 2017, Uppsala, Sweden, 29th April 2017.* (2017), A. Lisitsa, A. P. Nemytykh, and M. Proietti, Eds., vol. 253 of *EPTCS*, pp. 9–35.

[67] Nys, V., and De Schreye, D. Compiling control as offline partial deduction. In *Logic-Based Program Synthesis and Transformation - 28th International Symposium, LOPSTR 2018, Frankfurt/Main, Germany, September 4-6, 2018, Revised Selected Papers* (2018), F. Mesnard and P. J. Stuckey, Eds., vol. 11408 of *Lecture Notes in Computer Science*, Springer, pp. 115–131.

[68] Nys, V., Sneyers, J., and De Schreye, D. Automatic music generation: a logic-probabilistic étude generator. In *BNAIC 2015: Proceedings of the 27th Benelux Conference on Artificial Intelligence* (2015).

[69] P. Gallagher, J. Challenges in the specialisation of smart horn clause interpreters: Extended abstract at the sixth workshop on horn clauses for verification and synthesis and third workshop on program equivalence and relational reasoning. *Electronic Proceedings in Theoretical Computer Science 296* (07 2019), 3–5.

[70] Pilozzi, P., Schrijvers, T., and Bruynooghe, M. A transformational approach for proving properties of the CHR constraint store. De Schreye, Danny, pp. 1–15.

[71] Pilozzi, P., Schrijvers, T., and De Schreye, D. Proving termination of CHR in Prolog: A transformational approach. Hofbauer, D, pp. 30–33.

[72] Plotkin, G. D., and Pretnar, M. Handlers of algebraic effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings* (2009), G. Castagna, Ed., vol. 5502 of *Lecture Notes in Computer Science*, Springer, pp. 80–94.

[73] Robinson, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM) 12*, 1 (1965), 23–41.

[74] Saleh, A. H., and Schrijvers, T. Efficient algebraic effect handlers for Prolog. *TPLP 16*, 5-6 (2016), 884–898.

[75] Schrijvers, T., Demoen, B., Desouter, B., and Wielemaker, J. Delimited continuations for Prolog. *TPLP 13*, 4-5 (2013), 533–546.

[76] Smaus, J., Hill, P. M., and King, A. Verifying termination and error-freedom of logic programs with `block` declarations. *TPLP 1*, 4 (2001), 447–486.

[77] Tamary, L., and Codish, M. Abstract partial evaluation for termination analysis.

[78] Tick, E. The deevolution of concurrent logic programming languages. *J. Log. Program. 23*, 2 (1995), 89–123.

[79] Vidal, G. A hybrid approach to conjunctive partial evaluation of logic programs. In *Logic-Based Program Synthesis and Transformation - 20th International Symposium, LOPSTR 2010, Hagenberg, Austria, July 23-25,*

2010, Revised Selected Papers (2010), M. Alpuente, Ed., vol. 6564 of *Lecture Notes in Computer Science*, Springer, pp. 200–214.

[80] Vidal, G. A hybrid approach to conjunctive partial evaluation of logic programs. In *Logic-Based Program Synthesis and Transformation*. Springer-Verlag Berlin Heidelberg, 2011, pp. 200–214.

[81] Wiggins, G. A. *The improvement of Prolog program efficiency by compiling control: A proof-theoretic view.* Department of Artificial Intelligence, University of Edinburgh, 1990.

[82] Winters, T., Nys, V., and De Schreye, D. Automatic joke generation: Learning humor from examples. In *Distributed, Ambient and Pervasive Interactions: Technologies and Contexts - 6th International Conference, DAPI 2018, Held as Part of HCI International 2018, Las Vegas, NV, USA, July 15-20, 2018, Proceedings, Part II* (2018), N. Streitz and S. Konomi, Eds., vol. 10922 of *Lecture Notes in Computer Science*, Springer, pp. 360–377.

Curriculum vitae

Work experience

Present September 2018	Lecturer at CVO Crescendo, Mechelen *Continuing education in ICT* Taught courses on programming and computer networking. Currently teaching courses on programming, operating systems and databases.
Present March 2014	Doctoral researcher at KU Leuven, Leuven *Compiling coroutines* Formal analysis of logic programs. Presented at international conferences and published in an international journal. Taught exercises in artificial intelligence, authored and taught exercises in scripting languages, taught science communication, worked on promoting STEM programs as a part of *InnovationLab*.
February 2014 September 2013	Junior consultant at HealthConnect, Vilvoorde *Software for health professionals* Developed tools for secure communication between health professionals. Built electronic forms for insurance agencies.

Education

Present	Doctor of Engineering Science in Computer Science,
March 2014	KU Leuven, Leuven
January 2019	Certificate in Education,
September 2015	CVO Crescendo, Mechelen
September 2013	Master of Engineering Science in Computer Science,
September 2011	KU Leuven, Leuven
	magna cum laude
June 2011	Bachelor of Science in Computer Science,
September 2008	KU Leuven, Leuven
	cum laude
June 2008	Bachelor of Arts in Linguistics and Literature,
September 2005	KU Leuven, Leuven
	cum laude

Research interests

Programming languages:	Prolog, Racket, Python
Applications:	compilation and domain specific languages, music generation, web programming

List of publications

Directly related to compiling control

- De Schreye, D., Nys, V., and Nicholson, C. J. Analysing and Compiling Coroutines with Abstract Conjunctive Partial Deduction. In *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers* (Canterbury, UK, 2014), Lecture Notes in Computer Science, Springer, pp. 21–38

- Nys, V. Advances in analyzing coroutines by abstract conjunctive partial deduction. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015.* (2015), M. D. Vos, T. Eiter, Y. Lierler, and F. Toni, Eds., vol. 1433 of *CEUR Workshop Proceedings*, CEUR-WS.org

- Nys, V., and De Schreye, D. Abstract conjunctive partial deduction for the analysis and compilation of coroutines. *Formal Aspects of Computing 29*, 1 (Jan 2017), 125–153

- Nys, V., and De Schreye, D. Transforming coroutining logic programs into equivalent CHR programs. In *Proceedings Fifth International Workshop on Verification and Program Transformation, VPT@ETAPS 2017, Uppsala, Sweden, 29th April 2017.* (2017), A. Lisitsa, A. P. Nemytykh, and M. Proietti, Eds., vol. 253 of *EPTCS*, pp. 9–35

- Nys, V., and De Schreye, D. Compiling control as offline partial deduction. In *Logic-Based Program Synthesis and Transformation - 28th International Symposium, LOPSTR 2018, Frankfurt/Main, Germany, September 4-6, 2018, Revised Selected Papers* (2018), F. Mesnard and P. J. Stuckey, Eds., vol. 11408 of *Lecture Notes in Computer Science*, Springer, pp. 115–131

Additional publications

- Meert, W., Nys, V., Theunis, R., Fannes, T., Ingels, M., Guns, T., Van den Broeck, G., Driessens, K., De Schreye, D., and Verhelst, M. Innovation lab@ KU Leuven: Education, engineering and artificial intelligence

- Nys, V., Sneyers, J., and De Schreye, D. Automatic music generation: a logic-probabilistic étude generator. In *BNAIC 2015: Proceedings of the 27th Benelux Conference on Artificial Intelligence* (2015) **(best paper award)**

- Winters, T., Nys, V., and De Schreye, D. Automatic joke generation: Learning humor from examples. In *Distributed, Ambient and Pervasive Interactions: Technologies and Contexts - 6th International Conference, DAPI 2018, Held as Part of HCI International 2018, Las Vegas, NV, USA, July 15-20, 2018, Proceedings, Part II* (2018), N. Streitz and S. Konomi, Eds., vol. 10922 of *Lecture Notes in Computer Science*, Springer, pp. 360–377



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DECLARATIVE LANGUAGES AND SYSTEMS GROUP
Celestijnenlaan 200A box 2402
B-3001 Leuven