BANE: A Library for Scalable Constraint-Based Program Analysis

by

Manuel Alfred Fähndrich

B.E. (Ecole Polytechnique Fédérale de Lausanne) 1993
M.S. (University of California at Berkeley) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Alexander Aiken, Chair
Professor Susan Graham
Professor Hendrik Lenstra

1999

The dissertation of Manuel Alfred Fähndrich is approved:

Chair Date

Date

Date

University of California at Berkeley

1999

BANE: A Library for Scalable Constraint-Based Program Analysis

Abstract

BANE: A Library for Scalable Constraint-Based Program Analysis

by

Manuel Alfred Fähndrich
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Alexander Aiken, Chair

Program analysis is an important aspect of modern program development. Compilers use program analysis to prove the correctness of optimizing program transformations. Static error detection tools use program analysis to alert the programmer to the presence of potential errors. This dissertation focuses on the expressiveness and implementation of constraint-based program analyses, *i.e.*, analyses that are expressed as solutions to a system of constraints. We show that structuring the implementation of program analyses around a library of generic constraint solvers promotes reuse, gives control over precision-efficiency tradeoffs, and enables optimizations that yield orders of magnitude speedups over standard implementations.

The first part of the dissertation develops the formalism of mixed constraints which provides a combination of several constraint formalisms with distinct precision-efficiency tradeoffs. We provide a semantics for constraints and constraint resolution algorithms. The second part of the dissertation describes an implementation of mixed constraints and a number of novel techniques to support the practical resolution of large constraint systems. We give empirical results supporting the claims of scalability, reuse, and choice of precision-efficiency tradeoffs provided by the mixed constraint framework.

Professor Alexander Aiken
Dissertation Committee Chair

To my wife Barbara,

for the patience and support.

Contents

List of Figures vii

List of Tables ix

1 Introduction 1

2 Background 6

2.1 An Ideal Model for Types 6

2.2 Set Constraints 9

2.2.1 Term-Set Model 9

2.2.2 An Ideal Model for Set Expressions 9

I Mixed Constraints 13

3 Mixed Constraints 15

3.1 A Motivating Example 16

3.2 Syntax 17

3.3 Sorts 18

4 Semantics 22

4.1 Value Domains 23

4.2 Interface Paths 26

4.3 Type Collections 29

4.4 Semantic Relations 36

4.5 Solutions 39

4.6 Discussion and Related Work 39

5 Constraint Resolution 45

5.1 Inductive Constraints 45

5.2 Set Constraint Resolution 46

5.2.1 Upward-Closure and Negation 46

5.2.2 L-Intersection Simplification 52

5.2.3 Set Resolution Rules 55

5.3 Term and FlowTerm Resolution 55

5.4 Row Resolution 55
5.4.1 Domain Constraints 58
5.4.2 General Row Resolution 60
5.4.3 Splitting of Closed Rows 62
5.4.4 Splitting of Minimal and Maximal Rows 62
5.5 Inductive Systems 66
5.5.1 Level Semantics 71
5.5.2 Contractive Systems of Equations 75
5.5.3 Contractive Term-Equations 77
5.5.4 Generating Set and FlowTerm-Equations 78
5.5.5 Solutions For Row-Constraints 85

6 Practical Aspects of Constraint Resolution 87
6.1 Conditional Constraints 87
6.2 Graph Formulation 92
6.2.1 Transitive Lower Bound 99
6.2.2 Condition resolution 101
6.3 Complexity 103
6.4 Discussion and Related Work 103

II BANE 107

7 BANE: An Implementation of Mixed Constraints 109
7.1 Constraint Graph Representation 109
7.1.1 Nodes 110
7.1.2 Edges 111
7.1.3 Alias Edges 112
7.1.4 Expression Hashing and Bound Representation 113
7.1.5 Online Cycle Detection and Elimination 114
7.1.6 General Constraint Resolution Algorithm 120
7.2 Set Sort 122
7.2.1 Projections 122
7.2.2 Projection Merging 123
7.3 Term Sort 125
7.4 FlowTerm Sort 126
7.5 Row Sorts 128
7.6 Polymorphic Analysis 130
7.6.1 Multiple Constraint Graphs 131
7.6.2 Quantification and Instantiation 132
7.6.3 Simplification 132

8 Example Analyses **136**
8.1 Points-to Analysis for C . . . 136
8.1.1 Re-Formulation using Standard Set Constraints . . . 138
8.1.2 Constraint Generation . . . 139
8.1.3 A Complete Points-to Example . . . 142
8.2 ML Exception Inference . . . 145
8.2.1 Type and Constraint Generation . . . 149
8.2.2 An Example Inference . . . 153
8.2.3 Exception Inference for Full SML . . . 155
8.2.4 Precision-Efficiency Variations . . . 157
8.2.5 Related Exception Work . . . 158

9 Experiments **159**
9.1 Measurement Methodology . . . 159
9.2 Online Cycle Elimination . . . 160
9.2.1 Measurements . . . 161
9.3 Projection Merging . . . 165
9.4 Standard Form v.s. Inductive Form . . . 168
9.4.1 Standard Form . . . 169
9.4.2 Inductive Form . . . 170
9.4.3 Measurements . . . 171
9.4.4 TLB on demand . . . 175
9.5 Precision-Efficiency Tradeoffs . . . 177

10 Related Work **184**
10.1 The Cubic-Time Bottleneck . . . 184
10.2 Sub-Cubic Time Formalisms . . . 185
10.3 Program Analysis Frameworks . . . 186

11 Conclusions **188**

Bibliography **190**

List of Figures

1.1 Constraint-based approach to program analysis 2

3.1 Set operations in Π^{Set} 19
3.2 Operations $\Pi^{\mathsf{Row}(s)}$ for sort $\mathsf{Row}(s)$ 20

4.1 Development of lattices $(\mathsf{S}^s, \subseteq_s)$ 23

5.1 L-intersection simplification 55
5.2 Resolution rules for Set-constraints. 56
5.3 Resolution of FlowTerm-constraints. 56
5.4 Resolution of Term-constraints. 57
5.5 Resolution of Row-constraints (simple cases). 67
5.6 Resolution of Row-constraints (complex cases). 68
5.7 Simplification of domain constraints (complete) 69

6.1 Transforming constraints with conditional expressions into conditional constraints 91
6.2 Structural rewrite rule for conditional constraints 91
6.3 Structural rewrite rule for inconsistent constraint 91
6.4 Unnecessary edges in full graph closure 97

7.1 Example frequency of bound sizes 113
7.2 Cycle detection in an example graph 116
7.3 The two kinds of cycles detected 117
7.4 Possible 3-cycles 118
7.5 Transitive edges to projection patterns 123
7.6 Specialized Term resolution 127
7.7 Specialized FlowTerm-resolution 129

8.1 Example points-to graph 137
8.2 Constraint generation for Andersen's analysis 140
8.3 More complex C example 142
8.4 Points-to graph of program in Figure 8.3 142
8.5 Type and exception inference rules for expressions 148
8.6 Type and exception inference rules for patterns 149

8.7 Example Mini-ML program 152

9.1 IF without cycle elimination 162
9.2 Analysis times with cycle detection and oracle 162
9.3 Speedups through online cycle detection 163
9.4 Graph Schema 165
9.5 Paths from sources to $\mathcal{Y}$ to sinks 165
9.6 Paths added through projection merging 166
9.7 Speedups through projection merging 168
9.8 Example constraints in SF and IF 170
9.9 SF and IF without cycle elimination 173
9.10 Relative execution times of Shapiro and Horwitz's SF implementation of C points-to analysis (SH) over SF-Plain 173
9.11 Analysis times with cycle detection and oracle 174
9.12 Speedups through online cycle detection 174
9.13 Speedups through inductive form 175
9.14 Fraction of variables on cycles found online 175
9.15 Final graph sizes of SF and IF 175
9.16 Comparison of the Base experiments 183

List of Tables

9.1 Benchmark data common to all experiments 160
9.2 Experiments 161
9.3 Benchmark data for IF-Plain and IF-Oracle 162
9.4 Benchmark data for IF-Online 163
9.5 Cycle detection statistics 164
9.6 Benchmark data for IF-PM 167
9.7 Points-to Experiments 172
9.8 Benchmark data for SF-Plain and SF-Oracle 172
9.9 Benchmark data for SF-Online 174
9.10 Benchmark data for on demand TLB 176
9.11 Precision-efficiency variations for exception analysis 177
9.12 ML Benchmarks for exception analysis 179
9.13 Experiments for exception analysis 179
9.14 Benchmark data of all implementations and experiments 180
9.15 Benchmark data for Term-Set Base experiment with Cartesian-closed constructors. 182

Acknowledgements

My deepest thanks go to my advisor Alex Aiken, without whose support and encouragement this dissertation would have never seen the light of day. His never wavering optimism and cheerfulness kept me going through those unavoidable lows of graduate school and I hope I was able to take on some of these qualities myself.

I'm also deeply indebted to my Berkeley fellow graduates Jeff Foster and Zhendong Su for their role as early adopters of BANE. Their work, discussions, and feedback helped me focus on the important aspects of BANE and spurned me on to find yet another factor of ten. Other Berkeley people I'd like to thank for generating ideas through various discussions are David Gay, Raph Levien, Ben Liblit, Boris Vaysman, Remzi Arpaci, and Andrew Begel.

Finally, I'd like to thank my other committee members Sue Graham and Hendrik Lenstra for their time and useful feedback.

Chapter 1

Introduction

This dissertation supports the thesis that structuring the implementation of program analyses around a library of generic constraint solvers promotes reuse, gives control over precision-efficiency tradeoffs, and enables optimizations that yield orders of magnitude speedups over standard implementations.

The work described here is part of an ongoing research project on program analysis at the University of California at Berkeley under the supervision of Professor Alexander Aiken. The goals of this project are to advance the state of the art of program analysis by 1) developing novel sophisticated program analyses for compiler optimizations or error detection, and 2) finding novel ways of implementing program analyses (new and old) so as to improve their scaling behavior. My involvement in this research has resulted in BANE, a library for constraint-based program analysis. BANE addresses the need of our research group to rapidly develop prototypes of program analysis ideas and perform experimentation, without spending a huge effort on implementation. Without respect to efficiency and scalability, prototypes could in principle be developed reasonably cheaply. However, such naive implementations of prototypes do not answer interesting questions like whether in practice an analysis scales. Furthermore, poor prototype efficiency substantially slows down experimentation since the measurement cycle becomes intolerably long.

Experimentation is important, since paper designs of program analyses often do not prove practical for two reasons. First, the computed information may be too approximate on real programs to be useful. Second, the running time and especially the scaling behavior of the analysis may make it impractical. (Analysis designers sometimes ignore the asymptotic complexity of their algorithms, hoping that the worst case bound is not met in practice.) Shortcomings in precision and efficiency are usually only found after a substantial effort has been invested in writing and tuning an implementation. At that point, it may be difficult to change the precision and or efficiency of the analysis, since tuning often results in monolithic, obscure code that is hard to change. For the same reason, such implementations are also hard to maintain and reuse.

Constraint-based implementations of program analyses need not fall into this trap. Figure 1.1 shows graphically the organization of a constraint-based program analysis. The goal of any static program analysis is to infer information about a program given in some source representation. In constraint-based program analysis, this information is expressed

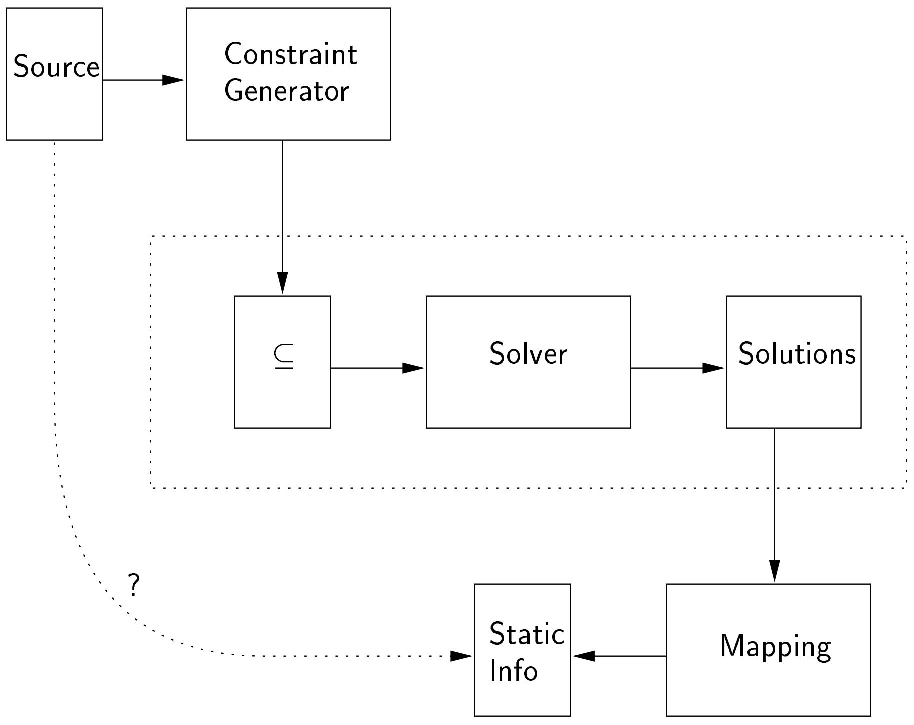

Figure 1.1: Constraint-based approach to program analysis

as the solutions to a system of constraints derived from the program source. Constraint-based program analysis thus proceeds in three steps: 1) constraints are generated according to a set of rules from the source, 2) constraints are solved, and 3) the static information is extracted from the solutions. A constraint-based program analysis is uniquely defined by the constraint generation rules and the mapping from solutions to the desired static information. The analysis is therefore independent of the constraint resolution which is the bulk of the implementation. The constraint resolution can be thought of as a black box, where constraints are entered on one side, and solutions come out on the other side. The black box interface is well defined, namely by the constraint language and the meaning of the constraints. As a result, the black box can be implemented completely independently of any particular program analysis, as long as the implementation produces the solutions to the given constraints. It is this clean separation of constraint generation and constraint resolution that naturally supports reuse and non-intrusive tuning. Constraint resolution engines can be factored into libraries and their code tuned independently of any particular constraint generation code. The narrow interface to the constraint engines further enables more aggressive tuning than in monolithic implementations. For example, the representation of constraints and the resolution algorithm can be completely changed without affecting any client analyses.

BANE is based on a novel constraint formalism called *mixed constraints.* Mixed constraints provides inclusion constraints between *mixed expressions.* Mixed expressions are a combination of expressions from a number of base constraint formalisms. Mixed constraints then synthesizes constraint relations out of the constraint relations of the base formalisms. The synthesis of the mixed constraint relations is parameterized by *constructor signatures* that specify how expressions of the base formalisms are combined into mixed expressions. If the base formalisms have distinct precision-efficiency characteristics, then it possible to choose the precision-efficiency tradeoffs of mixed constraints explicitly via a suitable set of constructor signatures.

One of the base constraint formalisms provided in BANE is set-constraints. Many well-known program analyses can be expressed in this formalism. Resolution of set constraints in the simplest case has cubic worst case time complexity. Implementations in the past have shown that analyses in this constraint class are feasible on small programs. Unfortunately, the resolution of set-constraint problems for larger programs quickly becomes intractable on today's hardware. This scaling problem has resulted in a shift towards less precise but computationally cheaper algorithms for program analyses. One contribution of this dissertation is to provide a transition path from precise but expensive analyses to coarser but faster analysis through the mixed constraint formalism.

Another major contribution of the work presented here is a number of implementation techniques that substantially improve practical performance of set-constraint resolution. Even though these techniques do not lower the worst-case computational complexity, they do improve the "common" case scaling behavior of set constraints. Using these techniques, we can solve set-constraint problems that are orders of magnitude larger than the largest previously reported problems. Our hope is that by showing how to structure implementations for better scaling, more sophisticated program analyses become feasible for very large programs.

BANE has been used in three large analysis applications with several variations produced by our research group. Furthermore, it has been used by a number of graduate student groups to realize projects in a class on programming language semantics taught at Berkeley. This dissertation describes the architecture of BANE, motivating engineering tradeoffs and new implementation techniques with a series of experiments.

Consumers of Program Analysis

Program analysis computes information about a program's runtime behavior. Program analysis can be either *static*, *dynamic*, or a combination of the two. In static program analysis, information about all executions of a program are inferred without actually running the program. Dynamic program analysis takes the form of trace gathering during particular executions of programs. In this dissertation program analysis always refers to static program analysis.

Program analysis is an important component of software development tools. There are two main uses of program analysis. Traditionally, program analyses is an integral component of optimizing compilers. Program optimizations are semantics preserving code transformations. Before a compiler can perform any particular optimization, it must prove that the transformation is semantics preserving in the particular context. For all but trivial transformations, this proof is provided by a program analysis. For example, a procedure call with late binding semantics may be transformed into a static direct call whenever the receiver of the call can be uniquely identified through a closure or receiver class analysis. Unfortunately, most properties of programs are undecidable in general. Program analysis is thus always concerned with computing approximations to the actual program properties. One often hears that an analysis is *conservative*, meaning that the analysis errs on the side of safety w.r.t. the use of the inferred properties. For example, if receiver class analysis is used to replace late bindings with static calls, the analysis must over-estimate the potential receivers, or the program transformation may not be safe.

The use of program analysis in compilers started with the earliest optimizing Fortran compilers. Since then, the use of program analysis has become ubiquitous in compilers across the whole language spectrum. As new programming paradigms develop, new optimization opportunities arise that result in novel program analyses. Examples that come to mind are *strictness analysis* for lazy functional languages, *receiver class analysis* for object oriented languages, *array use analysis* for functional languages. Although program analyses in compilers have become more sophisticated since the Fortran era, the main criteria for such analyses is efficiency, both absolute speed and scaling behavior. Many sophisticated program analyses found in the literature have not made their way into commercial compilers (or even into research compilers) due to bad scaling behavior.

The second use of program analysis is *error detection*. This application has become progressively more important in the last decade and, at least in the author's view, will overshadow the optimization use of program analysis by far in the future. The most common manifestation of this use is in type checkers and type inference systems. In this scenario, program analysis guarantees that a program produces no runtime errors arising from applying primitive operations to unsuitable arguments. In contrast to its use in optimizations where program analyses always have to be conservative by under-estimating properties, program analyses in error detection may also err on the other side, *i.e.*, over-estimate properties, giving rise to two camps of error detection. In the first camp, error detection proves the *absence* of errors. In the second camp, it proves the *existence* of errors. The difference between the two camps arises from the approximation inherent in program analyses. Error detection can be viewed as verifying a *partial program specification*. If a conservative program analysis proves that a particular program satisfies a partial specification, then it proves the absence of errors w.r.t. the specification. However, if the conservative analysis fails to prove the specification, it is not known whether the program actually behaves in unspecified ways, or whether the proof failed due to the approximations of the analysis. A program analysis trying to prove the existence of an error also verifies whether a program satisfies a particular specification. However, if the analysis finds an unspecified behavior w.r.t. the specification, then the error is a guaranteed error. On the other hand, if no errors are found, it is not known whether the program satisfies the specification, or whether the absence of errors is a result of over-estimating program properties.

Program analysis in error detection can also be distinguished from program analysis for optimizations in terms of the efficiency requirements. Scaling is still an important issue, but absolute speed is not. For example, it may be well worth running a sophisticated analysis of a large program overnight if the analysis detects bugs or proves their absence. Compilation cannot tolerate long analysis times because it disrupts the edit-compile-test cycle and thus programmer productivity.

Outline of the Dissertation

The dissertation is organized into two parts. The first part describes the mixed constraint formalism and its constraint resolution and is more theoretical in nature. The second part describes implementation techniques used in BANE and experiments to validate our claims of improved scaling. The next chapter introduces background information needed in the first part of the dissertation.

Chapter 2

Background

This chapter provides background information on the ideal model for types and set constraints.

2.1 An Ideal Model for Types

The ideal model of types was proposed by MacQueen, Plotkin, and Sethi [56, 57]. This section summarizes the results of that work used in the development of the mixed constraint formalism.

Types are used in programming languages to characterize certain subsets of all runtime values. If a program expression e has a type τ, then it is understood that any evaluation of e results in a value v, where v is an element of the set of values denoted by τ. To make these notions precise, the set of runtime values $\mathbf{V}$, and the subsets of $\mathbf{V}$ that form the denotations of types must be characterized. This approach is referred to as the denotational model of program semantics. Other models, such as operational models or axiomatic models, have different notions of types.

The set of values $\mathbf{V}$ is usually characterized by an equation of the form

$$\mathbf{V} = \mathbf{B} \cup \{\mathit{wrong}, \bot\} \cup (\mathbf{V} \rightarrow \mathbf{V}) \cup (\mathbf{V} \times \mathbf{V}) \cup (\mathbf{V} + \mathbf{V}) \tag{2.1}$$

where $\mathbf{B}$ stands for a set of base values, for example the natural numbers or the truth values. The distinguished value *wrong* is used to give meaning to evaluations that result in an error, for example applying a function to a value outside its domain. The special value bottom $\bot$ is the value of a non-terminating computation. The set $\mathbf{V} \rightarrow \mathbf{V}$ is the set of functions from $\mathbf{V}$ to $\mathbf{V}$. The set of $\mathbf{V} \times \mathbf{V}$ is the set of pairs of values from $\mathbf{V}$, and finally the coalesced sum $\mathbf{V} + \mathbf{V}$ is the set of pairs $\langle i, v \rangle$, where $v \in \mathbf{V} - \{\bot\}$, and $i = 1, 2$. The index[1] i specifies from which set in the sum the value v is taken.

Sets $\mathbf{V}$ that satisfy such equations have been shown to exist by Scott (see for example Gunter and Scott [36]) and are commonly referred to as *domains*. Domains are

[1]The index set is disjoint from $\mathbf{V}$.

essentially complete partial orders (cpo), where the order is given by

$$\begin{aligned}\bot &\leq v \\ f \leq f' &\iff \forall x \in \mathbf{V}.f\ x \leq f'\ x \\ \langle v_1, v_2\rangle \leq \langle v_1', v_2'\rangle &\iff v_1 \leq v_1' \ \wedge\ v_2 \leq v_2' \\ \langle i, v\rangle \leq \langle i, v'\rangle &\iff v \leq v'\end{aligned}$$

Domains D are constructed as the limit of a series. In the case of $\mathbf{V}$, the series is $\mathbf{V}_0 = \{\bot\}$ and

$$\mathbf{V}_{n+1} = \mathbf{B} \cup \{\mathit{wrong}, \bot\} \cup (\mathbf{V}_n \to \mathbf{V}_n) \cup (\mathbf{V}_n \times \mathbf{V}_n) \cup (\mathbf{V}_n + \mathbf{V}_n) \tag{2.2}$$

Elements of domains are either finite or infinite. An element $v \in \mathbf{V}$ is finite, if whenever v is less than the lub (least upper bound) of a directed set A (a set containing upper-bounds for all finite subsets), v is less than some element $v' \in A$. Infinite elements are the least upper bounds of increasing sequences.

A subset I of D is an *ideal*, iff

1. $I \neq \emptyset$
2. $\forall v \in I.\forall w \in D.w \leq v \implies w \in I$
3. for all increasing sequences $\langle v_i\rangle$ in I, $\bigsqcup v_i \in I$

Ideals are the non-empty, downward-closed subsets of D that are closed under lubs of increasing sequences. The collection of ideals of a domain D is written $\mathcal{I}(D)$. If a set satisfies only the first two requirements, then it is called an *order ideal*. Remarkably, when restricting the domain D to its finite elements D°, the collection of ideals $\mathcal{I}(D^\circ)$ is isomorphic to $\mathcal{I}(D)$. It is thus possible to consider only the order ideals and the finite elements of D. The collection of ideals $\mathcal{I}(D)$ forms a complete lattice under set-inclusion, where the meet is given by set-theoretic intersection, and the lub is defined by MacQueen et al. [57] as

$$\left(\bigsqcup_\lambda I_\lambda\right)^\circ = \bigcup_\lambda I_\lambda^\circ$$

The subsets of a domain D that give meaning to type expressions are the ideals of D that do not contain *wrong*. Let E be a language of type expressions formed by the following grammar

$$E ::= \mathbf{B} \mid t \mid E \to E \mid E \times E \mid E + E \mid E \cap E \mid E \cup E \mid \forall t.E \mid \exists t.E$$

where t are type variables and $\mathbf{B}$ is a set of base types. A type expression E is said to be *contractive* in t, if

- E is b for some base type $b \in B$
- $E = t'$ with $t \neq t'$

- E is of the form $E_1 \to E_2$, $E_1 \times E_2$, or $E_1 + E_2$
- E is of the form $E_1 \cap E_2$ or $E_1 \cup E_2$ and both E_1 and E_2 are contractive in t
- E is of the form $\forall t'.E$ or $\forall t'.E$, and either $t = t'$ or E is contractive in t

MacQueen, Plotkin, and Sethi's main result [56] then states that a system of type equations of the form

$$\begin{aligned} t_1 &= E_1 \\ &\vdots \\ t_n &= E_n \end{aligned}$$

has unique solutions in $\mathcal{I}(\mathbf{V})$ as long as the expressions E_i are contractive in $t_1..t_n$. This result is derived in five steps.

- A function $\gamma(I,J)$ is defined that measures the "closeness" of two ideals I, J from $\mathcal{I}(D)$. The closeness function is parameterized by a *rank* function $r(I)$ that associates a natural number with each finite element in D.

$$\gamma(I,J) = \begin{cases} \infty & \text{if } I = J \\ \min\{r(v) \mid v \in (I \cup J - I \cap J)\} & \text{otherwise} \end{cases} \tag{2.3}$$

- The collection of ideals $\mathcal{I}(D)$ is shown to form a complete metric space under the distance metric $\delta(I,J) = 2^{-\gamma(I,J)}$. Series called *Cauchy* sequences converge in complete metric spaces.

  Definition 2.1 *A sequence of ideals $\langle I_i \rangle$ is a* Cauchy *sequence, if given any real $\epsilon > 0$, there exists an n, such that for all $n_1, n_2 \geq n$, the distance $\delta(I_{n_1}, I_{n_2}) < \epsilon$.*

- The rank function $r(v)$ is defined for elements in $v \in \mathbf{V}$ as the minimal i, such that $v \in \mathbf{V}_i$ in the series (2.2).

- The semantic counter-parts of the type constructors $\to$, $+$, and $\times$ are shown to be contractive in the metric δ and the operations $\cup$ and $\cap$ are shown to be *non-expansive* in the following sense.

  Definition 2.2 *Expressed in terms of the closeness function γ, an n-ary function f over ideals is contractive if*

$$\gamma(f(I_1,\ldots,I_n), f(J_1,\ldots,J_n)) > \min_i \gamma(I_i,J_i)$$

  and f is non-expansive if

$$\gamma(f(I_1,\ldots,I_n), f(J_1,\ldots,J_n)) \geq \min_i \gamma(I_i,J_i)$$

- Each system of contractive type equations is then a contractive map over $\mathcal{I}(D)$. The main result follows from the Banach fix-point theorem which states that contractive maps over complete metric spaces have unique solutions.

2.2 Set Constraints

Set constraints express inclusion relations between set expressions. Set expressions in turn denote sets of elements in some underlying domain. Set expressions E are formed by the grammar below. A set expression is either a set-variable $\mathcal{X}$ from the collection V, the empty set 0, the universal set 1, a *constructor* c from a collection of uninterpreted function symbols Σ applied to n set expressions (where n is the arity of c), a union, an intersection, or a negation.

$$E := \mathcal{X} \mid 0 \mid 1 \mid c(E_1, \ldots, E_n) \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid \neg E$$

Constants are just a special case of constructors with arity 0. The arity of a constructor $c \in \Sigma$ is written $a(c)$.

An inclusion constraint $E_1 \subseteq E_2$ between set expressions E_1 and E_2 expresses that the set of elements denoted by E_1 must be a set-theoretic subset of the set of elements denoted by E_2. The next two subsections describe two models for interpreting set expressions and characterizing their solutions.

2.2.1 Term-Set Model

The simplest interpretation of set expressions is as sets of elements in a Herbrand universe H. This model is commonly known as the *term-set* model [52, 51]. The universe $\mathbf{H}$ is formed by applying constructors from the same set Σ that is used to build constructed set expressions.

$$H = \{c(v_1, \ldots, v_{a(c)}) \mid v_i \in H, c \in \Sigma\}$$

In this model, each set expression E denotes a subset of H. A *variable assignment* σ is a map from variables to subsets of H. The interpretation or denotation of set expressions in the term-set model under a given variable assignment σ is then given by μ:

$$\begin{array}{rcl}
\mu[\![\mathcal{X}]\!]\sigma & = & \sigma(\mathcal{X}) \\
\mu[\![0]\!]\sigma & = & \emptyset \\
\mu[\![1]\!]\sigma & = & H \\
\mu[\![c(E_1, \ldots, E_n)]\!]\sigma & = & \{c(v_1, \ldots, v_n) \mid v_i \in \mu[\![E_i]\!]\sigma\} \\
\mu[\![E_1 \cup E_2]\!] & = & \mu[\![E_1]\!]\sigma \cup \mu[\![E_2]\!]\sigma \\
\mu[\![E_1 \cap E_2]\!] & = & \mu[\![E_1]\!]\sigma \cap \mu[\![E_2]\!]\sigma \\
\mu[\![\neg E]\!]\sigma & = & H - \mu[\![E]\!]\sigma
\end{array}$$

A solution of a system of constraints $\{E_1 \subseteq E_1', \ldots, E_n \subseteq E_n'\}$ is a variable assignment σ, such that the inclusions $\mu[\![E_i]\!]\sigma \subseteq \mu[\![E_i']\!]\sigma$ hold in the model for $i = 1..n$. The complexity and algorithms for solving systems set constraints (and extensions with projections) have been widely studied [74, 48, 41, 10, 5, 6, 13, 14].

2.2.2 An Ideal Model for Set Expressions

If the set constraint language is extended with expressions $E_1 \rightarrow E_2$ (standing for a set of functions taking arguments from the set E_1 and producing results in the set E_2), then the meaning of set expressions can no longer be adequately captured by the term-set model.[2] Instead, a model for set constraints can be built based on the ideal model of types discussed in the previous section. The semantic domain $\mathbf{V}$ is defined in terms of the equation

$$\mathbf{V} = \{\bot\} \cup (\mathbf{V} \to \mathbf{V}) \cup \bigcup_{c \in \Sigma} c(\mathbf{V} - \{\bot\}, \ldots, \mathbf{V} - \{\bot\}) \cup \{\mathit{wrong}\}$$

where $\bigcup_{c\in\Sigma} c(\mathbf{V} - \{\bot\}, \ldots, \mathbf{V} - \{\bot\})$ can be expressed as the coalesced sum of smash products [36]. More precisely, each value $c(v_1, \ldots, v_n)$ of a n-ary constructor c is an element of the coalesced sum

$$\underbrace{\mathbf{V} + \cdots + \mathbf{V}}_{|\Sigma| \text{ times}}$$

where the index set used in the coalesced sum are the constructors $c \in \Sigma$. Thus, the value $c(v_1, \ldots, v_n)$ has the form $\langle c, v\rangle$, where v is an element of the smash product

$$\underbrace{\mathbf{V} \otimes \cdots \otimes \mathbf{V}}_{n \text{ times}}$$

The smash product differs from ordinary product in that elements $\langle v_1, \ldots, v_n\rangle$, where $v_i = \bot$ for some i, are identified with $\bot$. This property is commonly referred to as being *strict* in $\bot$. Constructors in Σ are then said to be strict.

Note that $\bot$ and the element $\lambda x.\bot$ of $\mathbf{V} \to \mathbf{V}$ are distinct values. The former denotes a non-terminating computation, the latter denotes a function that is non-terminating when applied.

The domain $\mathbf{T}$ is the domain $\mathbf{V}$ without *wrong*. Set expressions are then interpreted as ideals of $\mathbf{T}$, *i.e.*, elements of $\mathcal{I}(\mathbf{T})$. Let σ be a variable assignment mapping set variables to elements of $\mathcal{I}(\mathbf{T})$. The meaning function μ for set expressions in the ideal model is then

$$\begin{array}{rcl}
\mu[\![\mathcal{X}]\!]\sigma & = & \sigma(\mathcal{X}) \\
\mu[\![0]\!]\sigma & = & \emptyset \\
\mu[\![1]\!]\sigma & = & \mathbf{T} \\
\mu[\![E_1 \to E_2]\!]\sigma & = & \{f \mid x \in \mu[\![E_1]\!]\sigma \implies f\ x \in \mu[\![E_2]\!]\sigma\} \\
\mu[\![c(E_1, \ldots, E_n)]\!]\sigma & = & \{c(v_1, \ldots, v_n) \mid v_i \in \mu[\![E_i]\!]\sigma - \{\bot\}\} \\
\mu[\![E_1 \cup E_2]\!] & = & \mu[\![E_1]\!]\sigma \cup \mu[\![E_2]\!]\sigma \\
\mu[\![E_1 \cap E_2]\!] & = & \mu[\![E_1]\!]\sigma \cap \mu[\![E_2]\!]\sigma \\
\mu[\![\neg E]\!]\sigma & = & \mathbf{T} - \mu[\![E]\!]\sigma \cup \{\bot\}
\end{array}$$

Complement expressions $\neg E$ pose a problem in this model since the set complement of an ideal in $\mathbf{T}$ is not an ideal. Even when adding $\{\bot\}$ to the complement, the result may not be an ideal, unless $\mu[\![E]\!]\sigma$ is *upward-closed.* An ideal I is upward-closed in $\mathbf{V}$ if $\forall v \in I - \{\bot\}.\forall w \in \mathbf{V}.w \geq v \implies w \in I$. This problem leads to restrictions on well-formed set expressions and the form of set constraints that can be solved. These restrictions guarantee that negations only appear on upward-closed expressions. Since these same restrictions apply to mixed constraints, we will discuss them in detail in Section 5.2.1.

[2]An alternative approach to the ideal model has been studied by Flemming Damm [19].

The main result of Aiken and Wimmers [3] then states that every set of constraints S satisfying the restrictions due to negations is equivalent to a set of *inductive systems* and that each inductive system has solutions. In order to define inductive systems, we assume an arbitrary total order on the set-variables appearing in the constraints. The index i in this order is written as a subscript to the variable, as in $\mathcal{X}_i$. To define solutions inductively, we need the notion of a top-level variable.

Definition 2.3 *The set of top-level variables* $\mathsf{TLV}(E)$ *of an expression* E *is defined by*

$$\begin{aligned} \mathsf{TLV}(0) &= \{\} & \mathsf{TLV}(1) &= \{\} \\ \mathsf{TLV}(\mathcal{X}) &= \{\mathcal{X}\} & \mathsf{TLV}(c(\ldots)) &= \{\} \\ \mathsf{TLV}(E_1 \cup E_2) &= \mathsf{TLV}(E_1) \cup \mathsf{TLV}(E_2) \\ \mathsf{TLV}(E_1 \cap E_2) &= \mathsf{TLV}(E_1) \cup \mathsf{TLV}(E_2) \end{aligned}$$

A constraint set S is an inductive system, iff

1. S has the form $L_1 \subseteq \mathcal{X}_1 \subseteq U_1, \ldots L_n \subseteq \mathcal{X}_n \subseteq U_n$ where L_i and U_i are set expressions without negations.
2. The top-level variables appearing in L_i and U_i have index strictly less than i, *i.e.*, $\mathsf{TLV}(L_i) \cup \mathsf{TLV}(U_i) \subseteq \{\mathcal{X}_1, \ldots, \mathcal{X}_{i-1}\}$ for all i.
3. The system S is closed under transitive constraints $L_i \subseteq U_i$. More formally, the following property holds for all solutions σ of S and for all i, j:

   $$\begin{array}{l} \forall k < i . \mu_j[\![L_k]\!]\sigma \subseteq \mu_j[\![\mathcal{X}_k]\!]\sigma \subseteq \mu_j[\![U_k]\!]\sigma \wedge \\ \forall k \geq i . \mu_{j-1}[\![L_k]\!]\sigma \subseteq \mu_{j-1}[\![\mathcal{X}_k]\!]\sigma \subseteq \mu_{j-1}[\![U_k]\!]\sigma \end{array} \implies \mu_j[\![L_i]\!]\sigma \subseteq \mu_j[\![U_i]\!]\sigma$$

   where μ_j is the meaning function up to level j defined by

   $$\mu_j[\![E]\!]\sigma = \mu[\![E]\!]\sigma \cap \mathbf{T}'_j$$

   and $\mathbf{T}'_j$ is the downward-closure of the finite domain approximation $\mathbf{T}_j$ of the sequence $\mathbf{T}_0, \mathbf{T}_1, \ldots$ used to define the domain $\mathbf{T}$.

The third condition of inductive systems states that if σ satisfies the constraints up to level j for all variables with index less than i, and σ satisfies all constraints up to level $j-1$ for all indices greater or equal to i, then the transitive constraint $L_i \subseteq U_i$ is also satisfied up to level j.

An inductive system $L_i \subseteq \mathcal{X}_i \subseteq U_i$ is equivalent to the set of equations

$$\begin{aligned} \mathcal{X}_1 &= L_1 \cup \mathcal{Y}_1 \cap U_i \\ &\vdots \\ \mathcal{X}_n &= L_n \cup \mathcal{Y}_n \cap U_n \end{aligned}$$

where the variables $\mathcal{Y}_i$ are fresh. This system in turn is equivalent to the following system of equations $\mathcal{E}$

$$\begin{aligned} \mathcal{X}_1 &= L'_1 \cup \mathcal{Y}_1 \cap U'_i \\ &\vdots \\ \mathcal{X}_n &= L'_n \cup \mathcal{Y}_n \cap U'_n \end{aligned}$$

where

- L'_i is equal to L_i where all top-level variables $\mathcal{X}_j$ of L_i are replaced with $L'_j \cup \mathcal{Y}_j \cap U'_j$, and
- U'_i is equal to U_i where all top-level variables $\mathcal{X}_j$ of U_i are replaced with $L'_j \cup \mathcal{Y}_j \cap U'_j$

Note that the above transformation is well-defined, since the top-level variables appearing in L_i and U_i have index strictly less than i due to Condition 2 of inductive systems. The transformation can thus be performed starting with L_1 and U_1 which have no top-level variables, and proceeding through L_n, U_n in order. The resulting system of equations $\mathcal{E}$ has no top-level variables from $\{\mathcal{X}_1, \dots, \mathcal{X}_n\}$. Therefore $\mathcal{E}$ is contractive in $\mathcal{X}_1..\mathcal{X}_n$ in the sense of Section 2.1. Every arbitrary assignment σ mapping the variables $\mathcal{Y}_i$ to elements of $\mathcal{I}(\mathbf{T})$, induces a unique solution of the equations $\mathcal{E}$.

Part I

Mixed Constraints

Chapter 3

Mixed Constraints

This part introduces the formalism used to specify and solve constraints in BANE. We define the syntax of mixed expressions and constraints. Chapter 4 defines their semantics using a denotational model and characterizes when constraints have solutions. Chapter 5 gives an algorithm for solving the constraints. Chapter 6 discusses practical aspects of constraint resolution.

Building an analysis framework on a single constraint formalism leads to a tension between the generality and the efficiency of the chosen formalism. Generality is needed for the framework to be useful in more than a couple of specialized cases, whereas efficiency is crucial for the framework to be useful in practice. The constraint formalism at the base of BANE—set constraints—satisfies the generality requirement. Indeed, many program analyses can be expressed within set constraints, including but not limited to *closure analysis*, *pointer analysis*, *receiver class analysis*, and *subtype inference*.

The directionality of set constraints allows expressing analyses with subtyping. Thus set constraints cover a much larger class of analyses than, for example, equality constraints. On the efficiency side, set constraints have not fared well in the past. Published accounts of implementations report practical performance on small problems, but report bad performance on large problems or don't report performance on large problems at all [68, 3, 4, 39, 24, 31, 58, 30]. Scaling is the main problem of such analyses. The scaling problem of set-constraints has led to less precise program analyses based on equality constraints solvable using unification [75] in nearly linear time. Mixed constraints addresses the tension between precision and scalability by integrating a number of specialized constraint formalisms with distinct precision-efficiency tradeoffs.

To motivate the development, consider using set constraints for specifying and implementing the resolution of equality constraints arising in Hindley-Milner type inference [60]. A standard set constraint solver can do this task. However, the efficiency will fall short of the standard unification-based implementation. In this case it is clear that a unification-based implementation is preferable. There are two important aspects that make the equality constraints for Hindley-Milner type inference simpler to solve than general set constraints. The first aspect is that equalities induce equivalence classes which are more efficient to represent and merge than the partial orders that arise from inclusions. The second aspect is the solution domain of expressions appearing in the equality constraints. Each type expression denotes a single tree of the form $c(\dots)$, with a unique head constructor c. Set expressions on the other hand denote sets of terms with a variety of head constructors. The general observation is that if the shape or domain of solution is restricted in certain ways, a specialized and more efficient algorithm should exist.

Given an analysis problem, it is natural to select the most specialized class of constraints in which the problem is expressible. This idea can be pursued by building a library implementing a variety of specialized constraint solvers. However, mixed constraints goes beyond this idea. Analysis problems may not be uniform in their precision requirements. It can well be that part of an analysis requires general sets for precision, and other parts may only need restricted sets built from a single head constructor. With a library of independent, specialized solvers such a problem must be expressed uniformly in a set constraint formalism, without taking advantage of the single head constructor restrictions. The idea of mixed constraints is to combine several constraint formalisms into a common framework that enables the construction of analyses where precision-efficiency tradeoffs are made explicit.

3.1 A Motivating Example

This section motivates mixed constraints with a concrete example analysis: Uncaught exception inference for a subset of the ML language. Exception inference is an interesting problem for mixed constraints because it can be expressed as a minimal refinement of standard Hindley-Milner type inference, while still making essential use of set expressions.

We begin by illustrating the problem of types that are more general than needed if the exception inference is uniformly expressed in a set constraint formalism. In ML, the type of an exception value v is simply exn—no indication is given of the possible exception constructors of v. Consider a refinement of the ML type system that models exception types with an explicit annotation of the set of exception constructors. For example, the type of the exception constructor `Subscript` is modeled as $\mathsf{exn}(\texttt{Subscript})$. A possible inference rule for `if`-expressions based on inclusion constraints is

$$\frac{\begin{array}{ll} A \vdash p : \mathrm{bool} & \\ A \vdash e_1 : T_1 & \\ A \vdash e_2 : T_2 & \\ T_1 \subseteq \mathcal{X} & \mathcal{X} \text{ fresh} \\ T_2 \subseteq \mathcal{X} & \end{array}}{A \vdash \texttt{if } p \texttt{ then } e_1 \texttt{ else } e_2 : \mathcal{X}} \quad [\mathrm{IF}]$$

The rule says that the result type must contain the types of both branches. The conditional expression

```
if p then Subscript else x
```

returns either the exception value `Subscript` (exceptions are first-class), or the value of the program variable `x`. Assuming `x` has type $\mathcal{Y}$, applying the inference rule to this expression gives the type $\mathcal{X}$ along with two lower bounds, written

$$\mathcal{X} \texttt{ where } \quad \mathsf{exn}(\texttt{Subscript}) \subseteq \mathcal{X} \ \wedge \ \mathcal{Y} \subseteq \mathcal{X}$$

There are many solutions for $\mathcal{X}$ and $\mathcal{Y}$ satisfying these constraints. One possible solution is

$$\begin{aligned}\mathcal{Y} &\mapsto \mathsf{int}\\ \mathcal{X} &\mapsto \mathsf{exn}(\mathtt{Subscript}) \cup \mathsf{int}\end{aligned}$$

For many programming languages (and in particular for ML), this solution is uninteresting, because the union of an integer and an exception cannot be used anywhere. We are really only interested in solutions where the type of the `else` branch is also an exception. However, we cannot simply require both branches to have the same type as in a standard ML type system, because the `else` branch may contribute an exception other than `Subscript`. For example, if the `else` branch returns $\mathsf{exn}(\mathtt{Match})$, we would like to infer that the entire `if` returns $\mathsf{exn}(\mathtt{Subscript})$ or $\mathsf{exn}(\mathtt{Match})$. Thus we have two conflicting goals: On one hand we need the generality of inclusion constraints to allow different exception constructors in the branches of the conditional, and on the other hand we do not want the full generality of inclusion constraints, since they admit many uninteresting solutions. In the example, the interesting solutions all have the form

$$\begin{aligned}\mathcal{Y} &= \mathsf{exn}(\mathcal{Z})\\ \mathcal{X} &= \mathsf{exn}(\mathtt{Subscript} \cup \mathcal{Z})\end{aligned}$$

which clarifies that the `if`-expression and both branches return exceptions and that the set of exception constructors of the result includes the `Subscript` exception and any exceptions contributed by the `else`-branch. In summary, the example illustrates two points:

- For particular analyses, inclusion constraints may admit more solutions than required.
- Set types are needed to express sets of values with more than one head constructor (e.g. $\mathtt{Subscript} \cup \mathcal{Z}$).

3.2 Syntax

Mixed constraints are a combination of constraints in several specialized constraint formalisms, which we refer to as *sorts*. Each sort in a collection of sorts S is characterized by a constraint language, a constraint relation, a solution domain, and a resolution algorithm. Expressions for a sort s are built from a collection of s-variables V^s, s-constructors Σ^s, and s-operations Π^s. Each constructor $c \in \Sigma^s$ and operation $op \in \Pi^s$ is equipped with a signature. A signature

$$c : \iota_1 \cdots \iota_k \to s$$

specifies the constructor sort s, the number of arguments k, the sort of arguments ι_j, and their variance (ι_j is t or $\overline{t}$ for some $t \in S$, overlined sorts mark contravariant arguments, the rest are covariant). Signatures also specify the strictness of the constructor in each argument. To keep the notation readable, we omit the strictness annotations. The signatures for operations are fixed, whereas the signatures for constructors parameterize the language. We refer to Σ^s for the set of s-constructors along with their signatures. The arity of a constructor c is written $a(c)$.

Let $\mathsf{V} = \bigcup_{s \in S} \mathsf{V}^s$ be the set of variables of all sorts. The language L^s of expressions of sort s is formed by the term algebra $\mathsf{T}_{\Sigma^s \cup \Pi^s}(\mathsf{V})$.

Definition 3.1 *An expression $E \in \mathsf{L}^s$ is well-formed if it is*

- *a variable of sort s ($E \in \mathsf{V}^s$),*
- *of the form $c(E_1, \ldots, E_k)$, where $c \in \Sigma^s$ has signature $\iota_1 \cdots \iota_k \to s$, and expression E_j is either an expression of sort s (if $\iota_j = s$), or a variable from V^{ι_j}*[1]
- *of the form $op(E_1, \ldots, E_k)$, where $op \in \Pi^s$ has signature $\iota_1 \cdots \iota_k \to s$, and E_j is either an expression of sort s (if $\iota_j = s$), or a variable from V^{ι_j}.*

Depending on the signature, some arguments to a constructor c of sort s may be of sort t distinct from s. An s-constructor c is called *mixed* if at least one argument of c is of a sort other than s. Otherwise the constructor is called *pure*. The constraint relation for sort s is written $\subseteq_s$. Constraints observe sorts, *i.e.*, a constraint $E_1 \subseteq_s E_2$ is well-formed if both E_1 and E_2 are s-expressions. Expressions and constraints are assumed to be well-formed unless otherwise mentioned.

We use the following terminology: Expressions on the left of a constraint $\subseteq_s$ are said to occur in an *L-context*, and expressions on the right occur in an *R-context*. Sub-expressions occur in the same context (L- or R-) as their immediately enclosing expression, unless the sub-expression is the ith argument to a constructor that is contravariant in i. In that case its context is inverse w.r.t. the enclosing expression. We call an expression occurring in an L-context (R-context) an *L-expression* (*R-expression*).

For example, let Term be a sort of equality constraints between tree expressions, formed by a set of constructors Σ^{Term} and variables $\mathsf{V}^{\mathsf{Term}}$ and no additional operations. Pure Term-expressions are defined by giving each constructor $c \in \Sigma^{\mathsf{Term}}$ the signature

$$c : \underbrace{\mathsf{Term} \cdots \mathsf{Term}}_{a(c)} \to \mathsf{Term}$$

3.3 Sorts

For concreteness, the development of mixed constraints focuses on the set of sorts provided in BANE. There are three base sorts: Set, Term, and FlowTerm, abbreviated by **s**,**t**, and **ft**.

The Set-sort is a language of set expressions with operations Π^{s} given in Figure 3.1 including joins $\sqcup$, meets $\sqcap$, the empty set (written 0), and negation. The universal set, which we refer to as 1, is simply shorthand for $\neg\{\}$. Note that we write unions and intersections of Set-expressions with $\sqcup$ and $\sqcap$ instead of the usual set-theoretic union and intersections $\cup$ and $\cap$. We make this distinction because the semantics of mixed expressions we give in Chapter 4 shows that the meet of two Set-expressions is not exactly set-theoretic intersection, but may in fact be smaller. Similarly, the join of two Set-expressions may be larger than set-theoretic union. Constraints $E_1 \subseteq_{\mathsf{Set}} E_2$ are inclusions between set expressions E_1 and E_2. The resolution rules are given in full detail in Chapter 5. Here we only show the rule

[1]When we introduce the operations of each sort, we also allow 0^t (the smallest expression of sort t) and 1^t (the largest expression of sort t) to appear as arguments of mixed constructors.

$$\begin{array}{rcll} \sqcup & : & \mathsf{Set}\ \mathsf{Set} \rightarrow \mathsf{Set} & \\ \sqcap & : & \mathsf{Set}\ \mathsf{Set} \rightarrow \mathsf{Set} & \\ 0 & : & \mathsf{Set} & \\ \neg\{c_1, \ldots, c_n\} & : & \mathsf{Set} & \text{for any set of } \mathsf{Set}\text{-constructors } c_i \in \Sigma^{\mathsf{Set}} \end{array}$$

Figure 3.1: Set operations in Π^{Set}

involving constructors to give insight into how constraints involving mixed constructors connect constraints of different sorts

$$S \cup \{c(E_1, \ldots, E_k) \subseteq_{\mathbf{s}} c(E'_1, \ldots, E'_k)\} \iff S \cup \{E_j \subseteq_{\iota_j} E'_j\}$$
$$\text{where } c : \iota_1 \cdots \iota_k \rightarrow \mathbf{s}$$

Depending on the signature $c : \iota_1 \cdots \iota_k \rightarrow \mathbf{s}$, the constraints $E_j \subseteq_{\iota_j} E'_j$ may be on expressions of sort ι_j distinct from s. For contravariant arguments, the constraint relation is flipped, *i.e.*, $E_j \subseteq_{\overline{s}} E'_j \equiv E'_j \subseteq_s E_j$.

The Term sort provides inclusion constraints between restricted sets. There are two constant operations in Π^{Term}: 0 stands for the empty set, and 1 for the set of all Term denotations. Sets are restricted in that they are built from a single head constructor (besides 0 and 1). On single head constructor expressions, the relation $\subseteq_t$ is actually equality. Thus, the resolution rule for constructors is symmetric in the constraints on constructor arguments

$$S \cup \{c(E_1, \ldots, E_k) \subseteq_{\mathbf{t}} c(E'_1, \ldots, E'_k)\} \iff S \cup \{E_j \subseteq_{\iota_j} E'_j, E'_j \subseteq_{\iota_j} E_j\}$$
$$\text{where } c : \iota_1 \cdots \iota_k \rightarrow \mathbf{t}$$

The third base sort, FlowTerm, also provides inclusion constraints over restricted sets. It has the same language as Term, *i.e.*, Π^{FlowTerm} contains only 0 and 1. This sort distinguishes itself from the Term sort in that the constraint relation is still directed for single head constructor expressions. The constructor rule is therefore not symmetric.

$$S \cup \{c(E_1, \ldots, E_k) \subseteq_{\mathbf{ft}} c(E'_1, \ldots, E'_k)\} \iff S \cup \{E_j \subseteq_{\iota_j} E'_j\}$$
$$\text{where } c : \iota_1 \cdots \iota_k \rightarrow \mathbf{ft}$$

The FlowTerm sort can be thought of as a combination of the Set and Term sort, inheriting the directionality from Set-constraints and the solution domain from the Term sort.

For each base sort s there is a corresponding row-sort Row(s), abbreviated by $\mathbf{r}(s)$. Expressions of sort Row(s) model record types of labeled fields of sort s. Such records denote sets of partial functions from an infinite set of labels $\mathbf{L}$ to elements of sort s. There are no constructors for row expressions. Instead all row expressions are formed from the operations in Figure 3.2. The constants 1 and 0 denote the set of all records and the least record. The syntax $\langle l : E_l \rangle_{l \in A}$ stands for a sequence $\langle l_1 : E_{l_1}, \ldots, l_n : E_{l_n} \rangle$ where $A = \{l_1, \ldots, l_n\}$. Row-expressions can be composed with $\circ$. Composition is restricted to the case where the left side is an explicit finite enumeration of labels. Furthermore, we are principally interested in strict extensions, *i.e.*, the domain of the Row-expression on the right does not contain

$$\begin{array}{rcl} 1 & : & \mathsf{Row}(s) \\ 0 & : & \mathsf{Row}(s) \\ \langle\rangle & : & \mathsf{Row}(s) \\ \\ \langle l : \cdot\rangle_{l\in A} \circ \cdot & : & \underbrace{s \cdots s}_{|A| \text{ times}} \ \mathsf{Row}(s) \to \mathsf{Row}(s) \\ & & \text{for any finite set } A \subseteq \mathbf{L} \end{array}$$

Figure 3.2: Operations $\Pi^{\mathsf{Row}(s)}$ for sort $\mathsf{Row}(s)$

any labels from A. We will make this notion precise in Chapter 5 when we address the resolution of the constraints. Besides associativity, a number of equivalences hold for row expressions. The empty record $\langle\rangle$ is a neutral element for composition.

$$E_1 \circ (E_2 \circ E_3) = (E_1 \circ E_2) \circ E_3 \tag{3.1}$$

$$\langle\rangle \circ E_1 = E_1 \circ \langle\rangle = E_1 \tag{3.2}$$

$$\langle l : E_l\rangle_{l\in A} \circ \langle l : E_l\rangle_{l\in A'} = \langle l : E_l\rangle_{l\in A \uplus A'} \tag{3.3}$$

Row expressions obtained by permuting the order of label-expression pairs are also considered equivalent. Using the above equivalences, every Row-expression is equivalent to a Row-expression of the form $\langle l : E_l\rangle_{l\in A} \circ E$, where $E = \langle\rangle$, $E = \mathcal{X}$, $E = 1$, or $E = 0$. Such Row-expressions are *normalized* and we assume that we work only with normalized Row-expressions. Where necessary, we disambiguate the field sort s by adding the sort as a superscript to a Row-expression as in $\langle l : E_l\rangle^s_{l\in A}$.

Our definition of mixed expressions requires that arguments of sort t to constructors and operations of sort s, where $t \neq s$ are variables from V^t. For convenience, we relax this restriction now for the constants 0 and 1 (of sort t), which may also appear as arguments in these cases.

Coming back to our motivating example on exception inference, we can now give signatures to the constructors for exceptions, and exception names:

$$\begin{array}{c} \mathsf{exn} : \mathbf{s} \to \mathbf{ft} \\ \texttt{Subscript} : \mathbf{s} \\ \cdots \end{array}$$

Types are modeled by FlowTerm expressions, *i.e.*, expressions having a single head constructor. To distinguish exceptions by name, the exception constructor exn is refined to a unary constructor, embedding sets of exception names into the FlowTerm sort. Exception names are constants of sort Set. Some exceptions constructors in ML carry a value. Such exception constructors c can be given the signature

$$c : \mathbf{ft} \to \mathbf{s}$$

making clear that the value is of sort FlowTerm and the resulting exception is of sort Set.

The set of sorts S is now fixed $\{\mathsf{s}, \mathsf{ft}, \mathsf{t}, \mathsf{r}(\mathsf{s}), \mathsf{r}(\mathsf{ft}), \mathsf{r}(\mathsf{t})\}$. The constants 0 and 1 are overloaded, but their sort should always be apparent from the context. Every choice of signatures Σ^{s}, Σ^{t}, and Σ^{ft} generates the languages for mixed expressions L^{s}, L^{t}, L^{ft}. We refer to Σ^{s}, Σ^{t}, and Σ^{ft} as fixed, but unknown signature sets.

Chapter 4

Semantics

This chapter formalizes the semantics of mixed expressions and constraints by defining semantic domains for the different sorts, the semantic constraint relations, and the notion of solutions.

We provide a denotational semantics, *i.e.*, a semantics where each expression denotes a subset of values from an underlying value domain. Constraints between expressions are then interpreted as inclusion constraints between these values sets. The novelty of mixed constraints is that these inclusions are actually stronger for some sorts than set-theoretic inclusion. The motivation for a denotational semantics stems from the desire to make the constraints suitable for quick prototyping of program analyses. The model in mind is that given a programming language and an analysis problem, mixed expressions are used to conservatively approximate the sets of values of each program expression. Constraints are used to conservatively model the value flow in the program, *i.e.*, if values from a program expression e_1 may flow to an expression e_2 where the values of e_1 are abstracted by a mixed expression E_1 and those of e_2 by E_2, then there should be a constraint $E_1 \subseteq_s E_2$. If all value-flows in a program are conservatively approximated by constraints, then one obtains a sound analysis. An analysis is sound if for any program expression e and possible evaluation of e to v, the denotation of the mixed expression E approximating e contains v.

In the above naive formulation, we have ignored the evaluation context. If different approximations of an expression are used according to the context, then the soundness argument must be refined. The idea here is to give an intuitive approach to modeling value flows at a range of abstractions. Without a denotational model and constraints that imply set inclusion, it becomes harder to reason about the appropriateness of an analysis.

We quickly summarize the development of the mixed constraints semantics. We use the ideal model of types as outlined in Chapter 2 to assign semantics to mixed expressions. The semantics is parameterized in two ways. The first parameterization is a collection of semantic constructors which are used to construct a semantic domain $\mathbf{V}$. The second parameterization of our semantics is given by the syntactic constructors appearing in mixed constraints, and their signatures. Each n-ary constructor comes with an interpretation function that maps sequences of n ideals to ideals. We use these constructor interpretations to partition the domain $\mathbf{V}$ into sub-domains $\mathbf{V}^s$ for each sort s. This partitioning induces a partitioning of the collection of ideals $\mathcal{I}(\mathbf{V})$ into collections of ideals $\mathcal{I}(\mathbf{V}^s)$ for each sort s.

$$\begin{tikzcd}
 & & \mathbf{V} \arrow[dll, dash] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] \arrow[drrr, dash] & & & \\
\mathbf{V}^{\mathsf{s}} \arrow[d, dash] & \mathbf{V}^{\mathsf{t}} \arrow[d, dash] & \mathbf{V}^{\mathsf{ft}} \arrow[d, dash] & \mathbf{V}^{\mathsf{r(s)}} \arrow[d, dash] & \mathbf{V}^{\mathsf{r(t)}} \arrow[d, dash] & \mathbf{V}^{\mathsf{r(ft)}} \arrow[d, dash] \\
(\mathcal{I}(\mathbf{V}^{\mathsf{s}}), \subseteq) \arrow[d, dash] & (\mathcal{I}(\mathbf{V}^{\mathsf{t}}), \subseteq) \arrow[d, dash] & (\mathcal{I}(\mathbf{V}^{\mathsf{ft}}), \subseteq) \arrow[d, dash] & (\mathcal{I}(\mathbf{V}^{\mathsf{r(s)}}), \subseteq) \arrow[d, dash] & (\mathcal{I}(\mathbf{V}^{\mathsf{r(ft)}}), \subseteq) \arrow[d, dash] & (\mathcal{I}(\mathbf{V}^{\mathsf{r(t)}}), \subseteq) \arrow[d, dash] \\
(\mathsf{S}^{\mathsf{s}}, \subseteq_{\mathsf{s}}) & (\mathsf{S}^{\mathsf{t}}, \subseteq_{\mathsf{t}}) & (\mathsf{S}^{\mathsf{ft}}, \subseteq_{\mathsf{ft}}) & (\mathsf{S}^{\mathsf{r(s)}}, \subseteq_{\mathsf{r(s)}}) & (\mathsf{S}^{\mathsf{r(t)}}, \subseteq_{\mathsf{r(t)}}) & (\mathsf{S}^{\mathsf{r(ft)}}, \subseteq_{\mathsf{r(ft)}})
\end{tikzcd}$$

Figure 4.1: Development of lattices $(\mathsf{S}^s, \subseteq_s)$

We then select only a subset S^s of each collection $\mathcal{I}(\mathbf{V}^s)$ which we call the *type-collection* S^s of sort s. The ordering on ideals $\mathcal{I}(\mathbf{V}^s)$ is set-theoretic inclusion. The relations expressed in mixed constraints are in general stronger than set-theoretic inclusion. We define the order relations $\subseteq_s$ on elements of S^s and show that $(\mathsf{S}^s, \subseteq_s)$ forms a lattice. Figure 4.1 summarizes the development in graphical form.

We use the following notational convention. The symbols $\cup$ and $\cap$ refer to set-theoretic union and intersection, or to joins and meets of ideals in the complete lattice of ideals $\mathcal{I}(\mathbf{V})$ (Section 2.1). Symbols $\sqcap$ and $\sqcup$ are used for meet and join operations of ideals w.r.t. the type collections S^s constructed in this chapter.

4.1 Value Domains

The semantics of mixed expressions is given in a common domain $\mathbf{V}$ similar to the ideal domain described in Chapter 2. The main differences from the semantics of set constraints given there are

- The semantic constructors from which the domain is built are disjoint from the syntactic constructors used in the constraints.

- The relation between syntactic and semantic constructors is captured by a collection of constructor interpretations $\phi_c : \mathcal{I}(\mathbf{V}) \cdots \mathcal{I}(\mathbf{V}) \to \mathcal{I}(\mathbf{V})$ that parameterize the semantics.

- The domain $\mathbf{V}$ contains total functions over an infinite label set $\mathbf{L}$ for modeling records. There is a special value *abs* in the range of these functions to also model partial functions which correspond more closely to actual record values.

- Not all ideals $\mathcal{I}(\mathbf{V})$ are denotations of mixed expressions. The constructor signatures and their interpretations impose more structure.

To distinguish semantic constructors from syntactic constructors, we refer to the set of semantic constructors with the letter $\mathcal{C}$ and to individual semantic constructors with the letter κ. The semantic domain $\mathbf{V}$ is the solution of the domain equation

$$\mathbf{V} = \{\bot\} \ \cup \ \bigcup_{\kappa \in \mathcal{C}} \kappa(\mathbf{V} - \{\bot\}, \dots, \mathbf{V} - \{\bot\}) \ \cup$$
$$\mathbf{V} \to (\mathbf{V} \cup \{\mathit{wrong}\}) \ \cup \ \mathbf{L} \to (\mathbf{V} \cup \{\mathit{abs}\})$$

The domain of values $\mathbf{V} \to (\mathbf{V} \cup \{\mathit{wrong}\})$ are the strict continuous functions from $\mathbf{V}$ to $\mathbf{V} \cup \{\mathit{wrong}\}$. The definition contains functions returning *wrong*. This provision is different from the development by Aiken and Wimmers [3], where functions are restricted to $\mathbf{V} \to \mathbf{V}$. Including functions that return *wrong* models function implementations more realistically (applying a function to an argument outside its domain may generate an error) and has the advantage of getting rid of the following equivalence (assuming $I \neq \{\bot\}$)

$$I \to \mathbf{V} = \{\bot\} \to \mathbf{V}$$

If functions may return *wrong*, this equivalence does not hold since the right side contains functions that return *wrong* for $x \in I$, whereas the left side does not.

The set of values $\mathbf{L} \to (\mathbf{V} \cup \{\mathit{abs}\})$ is the set of functions from labels to elements in $\mathbf{V}$ or *abs*. The idea of *abs* is to model partial functions. A record with finite domain $A \subset \mathbf{L}$ is modeled as returning *abs* on all labels not in A. The domain $\mathsf{dom}(f)$ of a function f is thus the set

$$\mathsf{dom}(f) = \{l \mid f\ l \neq \mathit{abs}\}$$

We say that an ideal is a Row-ideal if it is an element of $\mathcal{I}(\mathbf{L} \to (\mathbf{V} \cup \{\mathit{abs}\}))$. The domain of a Row-ideal I written $\mathsf{dom}(I)$ is the intersection of the domains of all its elements

$$\mathsf{dom}(I) = \bigcap_{f \in I - \{\bot\}} \mathsf{dom}(f)$$

Furthermore, we write $I(l)$ for the set

$$I(l) = \bigcup_{f \in I - \{\bot\}} f\ l$$

Given a finite subset of labels $A \overset{\mathsf{fin}}{\subseteq} \mathbf{L}$, the composition of two functions f and g, where $A \subseteq \mathsf{dom}(f)$, is written $f \circ_A g$ and corresponds to the function

$$f \circ_A g = \lambda l.\texttt{if}\ \ l \in A\ \ \texttt{then}\ \ f\ l\ \ \texttt{else}\ \ g\ l$$

Given two Row-ideals I and J such that $\mathsf{dom}(I)$ is finite, the composition $I \circ J$ is the set

$$I \circ J = \{f \circ_{\mathsf{dom}(I)} g \mid f \in I - \{\bot\} \wedge g \in J - \{\bot\}\} \ \cup \ \{\bot\} \qquad (4.1)$$

We now motivate the parameterization of constructor interpretations and then proceed by partitioning $\mathbf{V}$ into sub-domains $\mathbf{V}^s$ for each sort s. Most published accounts of set constraints interpret constructor expressions $c(E_1, \ldots, E_n)$ as sets of strict tuples labeled by c, where the syntactic constructors and the semantic constructors stand in a one-to-one relationship

$$\mu[\![c(E_1, \ldots, E_n)]\!] = \{c(t_1, \ldots, t_n) \mid t_i \in \mu[\![E_i]\!] - \{\bot\}\} \cup \{\bot\}$$

(μ is the function mapping expressions to their denotation). The arrow constructor for sets of functions cannot be modeled this way and is treated specially. Expressions $E_1 \to E_2$ have the interpretation

$$\{f \mid x \in \mu[\![E_1]\!] \implies f\ x \in \mu[\![E_2]\!]\} \cup \{\bot\}$$

Other standard type constructors, e.g. list also need to be modeled as special cases in such an interpretation. For example, the standard interpretation for a type $\mathsf{list}(E)$ is

$$\begin{array}{l}\mu[\![\mathsf{list}(E)]\!] = X \quad \text{where } X \text{ is defined by the equation} \\ X = \{\mathit{nil}, \bot\} \cup \{\mathit{cons}(v, w) \mid v \in \mu[\![E]\!] - \{\bot\} \wedge w \in X\}\end{array}$$

which is rather different from a set of tuples labeled by list, and has the additional property of being non-strict in E.

Having numerous special cases is impractical and interpreting all constructors as labeled tuples severely limits the applicability of the formalism developed here. Our approach is to assume as little as possible about the interpretation of constructors. We assume only that there exists an interpretation function $\phi_c : \mathcal{I}(\mathbf{V})^n \to \mathcal{I}(\mathbf{V})$ for each n-ary constructor c in any given signature set Σ^s. Each ϕ_c gives meaning to expressions with head constructor c

$$\mu[\![c(E_1, \ldots, E_n)]\!] = \phi_c(\mu[\![E_1]\!], \ldots, \mu[\![E_n]\!])$$

The mappings ϕ_c must satisfy a number of axioms. These properties are used in the remainder of the development.

Axiom 4.1 The mapping ϕ_c for constructor c must satisfy the following conditions.

i. [Ideals] ϕ_c is a mapping of ideals to ideals, *i.e.*, $\phi_c : \mathcal{I}(\mathbf{V})^n \to \mathcal{I}(\mathbf{V})$

ii. [Contractive] ϕ_c must be contractive according to the Definition 2.2.

iii. [Variance] ϕ_c must observe the variance of the declared signature $c : \iota_1 \cdots \iota_n \to s$. Formally,

$$I_j \subseteq J \implies \left\{ \begin{array}{ll} \phi_c(I_1, \ldots, I_n) \subseteq \phi_c(I_1..I_{j-1}, J, I_{j+1}..I_n) & \iota_j = t \in \mathsf{S} \\ \phi_c(I_1..I_{j-1}, J, I_{j+1}..I_n) \subseteq \phi_c(I_1, \ldots, I_n) & \iota_j = \bar{t} \in \mathsf{S} \end{array} \right.$$

iv. [Disjoint] The interpretations of any two distinct constructors c, d of some sort s must be disjoint, *i.e.*, $\phi_c(I_1, \ldots, I_{a(c)}) \cap \phi_d(J_1, \ldots, J_{a(d)}) = \{\bot\}$, for any sets I_i and J_j.

v. [Non-empty] The interpretation of any constructor c is non-empty (besides $\perp$) when applied to non-empty arguments, *i.e.*, $\phi_c(I_1, \ldots, I_{a(c)}) \supset \{\perp\}$ for all sets $I_i \supset \{\perp\}$.

vi. [Strictness] Every constructor interpretation is either strict in all arguments, or non-strict in all arguments.

vii. [Injective] The interpretation ϕ_c of non-strict constructors must be injective.

To simplify the development in the rest of the dissertation, we require all Term and FlowTerm constructors to be non-strict. We believe that this requirement can be relaxed.

The following equations define the domain of values for each sort, splitting up the common domain $\mathbf{V}$.

$$\begin{aligned}
\mathbf{V}^{\mathsf{t}} &= \bigcup\{\phi_c(I_1, \ldots, I_{a(c)}) \mid c : \iota_1 \cdots \iota_{a(c)} \rightarrow \mathsf{t} \;\wedge\; I_j \in \mathcal{I}(\mathbf{V}^{\iota_j}), j = 1..a(c)\} \\
\mathbf{V}^{\mathsf{ft}} &= \bigcup\{\phi_c(I_1, \ldots, I_{a(c)}) \mid c : \iota_1 \cdots \iota_{a(c)} \rightarrow \mathsf{ft} \;\wedge\; I_j \in \mathcal{I}(\mathbf{V}^{\iota_j}), j = 1..a(c)\} \\
\mathbf{V}^{\mathsf{s}} &= \bigcup\{\phi_c(I_1, \ldots, I_{a(c)}) \mid c : \iota_1 \cdots \iota_{a(c)} \rightarrow \mathsf{s} \;\wedge\; I_j \in \mathcal{I}(\mathbf{V}^{\iota_j}), j = 1..a(c)\}) \\
\mathbf{V}^{\mathsf{r}(s)} &= \{\perp\} \cup \mathbf{L} \rightarrow (\mathbf{V}^{s} \cup \{abs\})
\end{aligned}$$

Since the mappings ϕ_c are contractive, the above equations are also contractive and thus have unique solutions.

Mixed expressions allow Set-expressions to include e.g., embedded Term-expressions. Thus, inclusion on Set-ideals may actually be a stronger relation than set-theoretic inclusion, because the embedded Term-ideals require equality between constructor arguments and not inclusions. To make precise what inclusion relations on mixed expressions mean, we introduce the notion of *interface paths* and interface-path projections.

4.2 Interface Paths

Let an *access path* be a sequence formed by constructor-index pairs, *i.e.*, $P = (\Sigma \times \mathbb{N})^*$ where $\Sigma = \Sigma^{\mathsf{s}} \cup \Sigma^{\mathsf{t}} \cup \Sigma^{\mathsf{ft}}$. An access path $p \in P$ is *well-formed of sort s* if

1. $p = \epsilon$ (p is empty), or

2. $p = (c, j)p'$, $0 \leq j < a(c)$, $c : \iota_1 \cdots \iota_{a(c)} \rightarrow s$, and p' is well-formed and of sort ι_j.

Definition 4.2 *The following definitions characterize useful properties of paths.*

1. *The sort of a non-empty path p is $sort(p)$. Where convenient, we write p_s when p is non-empty and $sort(p) = s$.*

2. *A path p is* uniform of sort s *iff every prefix path of p is of sort s.*

3. *The* interface $\mathsf{if}(p)$ *of a non-empty, well-formed path p is t, if $p = p'(c, j)$, $c : \iota_1 \cdots \iota_{a(c)} \rightarrow s$, and $\iota_j = t$ or $\iota_j = \overline{t}$ for some $t \in \mathsf{S}$. Additionally, we say that the interface is covariant if $\iota_j = t$ and contravariant if $\iota_j = \overline{t}$. Where convenient, we write p^t whenever p is non-empty, well-formed, and $\mathsf{if}(p) = t$.*

4. *A path is* non-strict *if all constructors in the path are non-strict, otherwise the path is* strict.

5. *A non-empty path p is called an* interface path *if p is uniform of sort s, and* $\mathsf{if}(p) \neq s$.

6. *Finally, a path p is called* even *(*odd*), iff it the number of distinct prefix paths of p with contravariant interface is even (odd).*

Unless stated explicitly, paths are always well-formed and finite, and interface paths are always non-strict.

We now define what it means for a path to be *present* in a an ideal $\mathcal{I}(\mathbf{V})$.

Definition 4.3 *A path p is present in I, written $present(p, I)$, iff $p = \epsilon$ or $p = (c, j)q$ with $c : \iota_1 \cdots \iota_{a(c)} \to s$ and $\exists I_1 \in \mathcal{I}(\mathbf{V}^{\iota_1})..I_{a(c)} \in \mathcal{I}(\mathbf{V}^{\iota_{a(c)}})$ such that $\{\perp\} \subset \phi_c(I_1, \ldots, I_{a(c)}) \subseteq I$ and $present(q, I_j)$.*

Definition 4.4 *The projection $p^{-1}(I)$ of an ideal I w.r.t. to a path p is defined as follows:*

$$p^{-1}(I) = \begin{cases} I & \text{if } p = \epsilon \\ q^{-1}(\phi_c^{-1}(I, j)) & \text{if } p = (c, j)q \end{cases}$$

where $c : \iota_1 \cdots \iota_{a(c)} \to s$ and

$$\phi_c^{-1}(I, j) = \begin{cases} \bigcup\{I_j \mid \{\perp\} \subset \phi_c(I_1, \ldots, I_{a(c)}) \subseteq I\} & \iota_j = t \text{ for some } t \in \mathsf{S} \\ \bigcap\{I_j \mid \{\perp\} \subset \phi_c(I_1, \ldots, I_{a(c)}) \subseteq I\} & \iota_j = \bar{t} \text{ for some } t \in \mathsf{S} \end{cases}$$

with $I_k \in \mathcal{I}(\mathbf{V}^{\iota_k})$ for $k = 1..a(c)$, $\bigcup\{\} = \{\perp\}$, and $\bigcap\{\} = \mathbf{V}^t$, if $\iota_j = \bar{t}$.

The projection of non-present even paths is $\{\perp\}$ and the projection of non-present odd paths of interface t is $\mathbf{V}^t$. (Note that $\mathcal{I}(\mathbf{V}^s)$ is a complete lattice and that the above intersections and unions are defined for infinite collections.)

We now prove that projections of non-strict paths distribute over unions and intersections. We'll need the following lemmas.

Lemma 4.5 *For any ideals I, J, constructor c and index $1 \leq j \leq a(c)$, such that c is non-strict and covariant in j, we have*

$$\phi_c^{-1}(I, j) \cap \phi_c^{-1}(J, j) = \phi_c^{-1}(I \cap J, j)$$

Proof: First consider the case $\supseteq$. Since c is covariant in j, we have

$$\begin{array}{ll} \phi_c^{-1}(I \cap J, j) \subseteq \phi_c^{-1}(I, j) & \text{and also} \\ \phi_c^{-1}(I \cap J, j) \subseteq \phi_c^{-1}(J, j) & \text{thus} \\ \phi_c^{-1}(I \cap J, j) \subseteq \phi_c^{-1}(I, j) \cap \phi_c^{-1}(J, j) & \end{array}$$

Consider the case $\subseteq$. Suppose $t \in \phi_c^{-1}(I, j) \cap \phi_c^{-1}(J, j)$. Then

$$\begin{array}{ll} t \in \bigcup\{Y_j \mid \{\perp\} \subset \phi_c(Y_1, \ldots, Y_{a(c)}) \subseteq I\} & \text{and also} \\ t \in \bigcup\{Y_j' \mid \{\perp\} \subset \phi_c(Y_1', \ldots, Y_{a(c)}') \subseteq J\} & \end{array}$$

Thus $\exists Y_1..Y_{a(c)}$ and $\exists Y_1'..Y_{a(c)}'$ such that

$$\begin{array}{l} \{\perp\} \subset \phi_c(Y_1, \ldots, Y_{a(c)}) \subseteq I \;\wedge\; t \in Y_j \\ \{\perp\} \subset \phi_c(Y_1', \ldots, Y_{a(c)}') \subseteq J \;\wedge\; t \in Y_j' \end{array}$$

Since ϕ_c observes variance and is non-strict, $\{\perp\} \subset \phi_c(Y_1 \mathbin{\overline{\cap}} Y_1', \ldots, Y_{a(c)} \mathbin{\overline{\cap}} Y_{a(c)}') \subseteq I \cap J$, where

$$\overline{\cap} = \left\{ \begin{array}{ll} \cap & \text{for covariant arguments} \\ \cup & \text{for contravariant arguments} \end{array} \right.$$

Since $t \in Y_j \mathbin{\overline{\cap}} Y_j'$, we have $t \in \phi_c^{-1}(I \cap J, j)$. □

The next lemma states the dual case.

Lemma 4.6 *For any sets I, J, constructor c and index $j \leq a(c)$, such that c is non-strict and contravariant in j, we have*

$$\phi_c^{-1}(I, j) \cup \phi_c^{-1}(J, j) = \phi_c^{-1}(I \cap J, j)$$

The proof is symmetric to the proof of Lemma 4.5. In similar spirit one can prove the following

Lemma 4.7 *For any sets I, J, non-strict constructor c and index $j \leq a(c)$ we have*

$$\phi_c^{-1}(I \cup J, j) = \left\{ \begin{array}{ll} \phi_c^{-1}(I, j) \cup \phi_c^{-1}(J, j) & c \textit{ covariant in } j \\ \phi_c^{-1}(I, j) \cap \phi_c^{-1}(J, j) & c \textit{ contravariant in } j \end{array} \right.$$

The next lemma states that intersections and unions distribute over projections of non-strict paths.

Lemma 4.8 *For any sets I, J, and non-strict path p,*

$$p^{-1}(I) \cap p^{-1}(J) = \left\{ \begin{array}{ll} p^{-1}(I \cap J) & p \textit{ even} \\ p^{-1}(I \cup J) & p \textit{ odd} \end{array} \right.$$

and

$$p^{-1}(I) \cup p^{-1}(J) = \left\{ \begin{array}{ll} p^{-1}(I \cup J) & p \textit{ even} \\ p^{-1}(I \cap J) & p \textit{ odd} \end{array} \right.$$

Proof: By induction on the length of p. If $p = \epsilon$, the lemma holds. We prove the induction step for intersections. The case for unions is analogous. Suppose $p = (c, j)q$. By definition and the induction hypothesis we have

$$\begin{aligned} p^{-1}(I) \cap p^{-1}(J) &= q^{-1}\left(\phi_c^{-1}(I, j)\right) \cap q^{-1}\left(\phi_c^{-1}(J, j)\right) \\ &= \left\{ \begin{array}{ll} q^{-1}\left(\phi_c^{-1}(I, j) \cap \phi_c^{-1}(J, j)\right) & q \text{ even} \\ q^{-1}\left(\phi_c^{-1}(I, j) \cup \phi_c^{-1}(J, j)\right) & q \text{ odd} \end{array} \right. \end{aligned}$$

Note that since p is non-strict c is non-strict. Suppose that p is even. There are two cases. If q is even, then c is covariant in j. By Lemma 4.5 we have $\phi_c^{-1}(I, j) \cap \phi_c^{-1}(J, j) = \phi_c^{-1}(I \cap J, j)$, and thus $q^{-1}(\phi_c^{-1}(I,j) \cap \phi_c^{-1}(J,j)) = q^{-1}(\phi_c^{-1}(I \cap J,j))$. Now suppose that q is odd. Then c is contravariant in j and by Lemma 4.6 we have $\phi_c^{-1}(I,j) \cup \phi_c^{-1}(J,j) = \phi_c^{-1}(I \cap J,j)$, and thus $q^{-1}(\phi_c^{-1}(I,j) \cup \phi_c^{-1}(J,j)) = q^{-1}(\phi_c^{-1}(I \cap J,j))$. In either case $q^{-1}(\phi_c^{-1}(I \cap J,j)) = p^{-1}(I \cap J)$.

Now assume that p is odd. We have again two cases. If q is even, then c is contravariant in j. By Lemma 4.7 $\phi_c^{-1}(I,j) \cap \phi_c^{-1}(J,j) = \phi_c^{-1}(I \cup J,j)$, and thus $q^{-1}(\phi_c^{-1}(I,j) \cap \phi_c^{-1}(J,j)) = q^{-1}(\phi_c^{-1}(I \cup J,j))$. Now suppose that q is odd, then c is covariant in j. By Lemma 4.7 $\phi_c^{-1}(I,j) \cup \phi_c^{-1}(J,j) = \phi_c^{-1}(I \cup J,j)$, and thus $q^{-1}(\phi_c^{-1}(I,j) \cup \phi_c^{-1}(J,j)) = q^{-1}(\phi_c^{-1}(I \cup J,j))$. In either case $q^{-1}(\phi_c^{-1}(I \cup J,j)) = p^{-1}(I \cup J)$. □

4.3 Type Collections

This section defines for each sort s the collections $\mathsf{S}^s \subseteq \mathcal{I}(\mathbf{V}^s)$ of ideals denoted by mixed expressions of sort s (see also Figure 4.1). We call these collections *type collections.* Whereas in the work of Aiken and Wimmers [3] every ideal of $\mathcal{I}(\mathbf{V})$ is a possible denotation of a set-expression, in mixed constraints we impose more structure on the denotations that we consider. For example, Term and FlowTerm expressions only denote ideals of the domain obtained by applying a single head constructor. The interpretation functions ϕ_c make it convenient to specify the collections of ideals S^s.

Before we define the type collections S^s, we characterize three kinds of Row-ideals denoted by Row-expressions.

Definition 4.9 *Each Row-ideal I is fully characterized by the ranges $I(l)$ for all labels $l \in \mathbf{L}$. Thus, for all labels l, if $f \in I$, then for each $v \in I(l)$, there exists $f' \in I$, such that $f'(l) = v$, and $f'(l') = f(l)$ for all labels $l' \neq l$. Furthermore, a Row-ideal I is called*

1. [Maximal] *iff* $\mathsf{dom}(I)$ *is finite and* $\forall l \notin \mathsf{dom}(I).I(l) = \mathbf{V}^s \cup \{abs\}$.
2. [Closed] *iff* $\mathsf{dom}(I)$ *is finite and* $\forall l \notin \mathsf{dom}(I).I(l) = \{\perp, abs\}$.
3. [Minimal] *iff* $\mathsf{dom}(I) = \mathbf{L}$ *and there are finitely many* $l \in \mathbf{L}$ *such that* $I(l) \neq \{\perp\}$.

The three kinds of Row-denotations correspond to the three Row-expressions that can appear as the right-most expression of a Row-composition: 1, $\langle\rangle$, and 0. Maximal Rows are denoted by expressions of the form $\langle l : E_l \rangle_A \circ 1$, closed Rows by expressions of the form $\langle l : E_l \rangle_A = \langle l : E_l \rangle_A \circ \langle\rangle$, and minimal Rows by expressions of the form $\langle l : E_l \rangle_A \circ 0$. We define the *minimal* domain $\mathsf{dom}_\perp(I)$ of any Row-ideal I to be

$$\mathsf{dom}_\perp(I) = \begin{cases} \{l \mid I(l) \neq \{\perp\}\} & \text{if } I \text{ minimal} \\ \mathsf{dom}(I) & \text{otherwise} \end{cases}$$

Definition 4.9 implies that $\mathsf{dom}_\perp(I)$ is finite for all Row-ideals we consider. Even though we won't discuss any applications of Row-expressions until Section 4.6 and Chapter 8 we briefly give some intuition about the three different kinds of Row-ideals in our model. Width-subtyping is common in type systems and refers to the ability to forget the presence of certain fields in record types, or add new fields in variant types. Width-subtyping is reflected in the relation of the minimal domains of two Row-ideals. The semantic relation

$I \subseteq_{\mathsf{r}(s)} J$ on Row-ideals that we define later in Section 4.4 captures three distinct width-subtyping relations, depending on the kinds of Row-ideals I and J. If I and J are minimal Row-ideals, then $I \subseteq_{\mathsf{r}(s)} J$ only if $\mathsf{dom}_\perp(I) \subseteq \mathsf{dom}_\perp(J)$. If I and J are closed Row-ideals, then $I \subseteq_{\mathsf{r}(s)} J$ only if $\mathsf{dom}_\perp(I) = \mathsf{dom}_\perp(J)$. Finally, if I and J are maximal Row-ideals, then $I \subseteq_{\mathsf{r}(s)} J$ only if $\mathsf{dom}_\perp(I) \supseteq \mathsf{dom}_\perp(J)$.

The following invariant specifies the properties of the type collections S^s that make them distinct from $\mathcal{I}(\mathbf{V}^s)$.

Invariant 4.10 (Type Collections) *The collections S^s must satisfy the following conditions.*

1. *All elements (besides $\perp$ and $\top$, defined below) of $\mathsf{S}^{\mathbf{t}}_n$ and $\mathsf{S}^{\mathbf{ft}}_n$ are of the form $\phi_c(I_1, \ldots, I_n)$ for some constructor $c : \iota_1 \cdots \iota_k \to \{\mathbf{t}, \mathbf{ft}\}$ and elements $I_j \in \mathsf{S}^{\iota_j}_{n-1}$.*
2. *For every finite non-strict interface path $p^t_{\mathbf{s}}$ and every element $I \in \mathsf{S}^{\mathbf{s}}_n$ the projection $p^{-1}(I)$ is an element of $\mathsf{S}^t_{n-|p|}$. (By convention $\mathsf{S}^s_i = \mathsf{S}^s_0$ for negative i).*
3. *For every $I \in \mathsf{S}^{\mathsf{r}(s)}_i$ and every label l in the domain of I, the set $I(l)$ is an element of S^s_{i-1}, and I is either closed, maximal, or minimal according to Definition 4.9.*

Condition 1 is motivated by the desire to restrict the ideals of Term and FlowTerm-expressions to the form $\phi_c(\ldots)$, *i.e.*, ideals built from a unique head constructor c. Condition 2 guarantees that for any Set-ideal I in S^{s}, the projection $p^{-1}(I)$ of any non-strict interface path p^t is an element of S^t. We use this property in Section 4.4 to define the semantic relation $I \subseteq_{\mathsf{s}} J$ to be stronger than set-theoretic inclusion $I \subseteq J$ by also requiring that for any non-strict interface path p^t, $p^{-1}(I) \subseteq_t p^{-1}(J)$. Condition 3 is similar to Condition 2, but for Row-ideals. Note that Row-composition defined by (4.1) preserves Condition 3.

The collections $\mathsf{S}^s \subseteq \mathcal{I}(\mathbf{V}^s)$ are now defined inductively for each sort, along with binary operations $\sqcap_s$ and $\sqcup_s$. These operations will serve as the meet and join when we define the semantic relations on the type collections. The meet and join also serve as the generalized intersection and union operations for mixed Set-expressions. We use two abbreviations $\perp_s$ and $\top_s$ to denote the smallest and the largest ideal for sort s. For sorts $s \in \{\mathbf{s}, \mathbf{ft}, \mathbf{t}\}$, the smallest set is $\perp_s = \{\perp\}$. For Row-sorts, the smallest ideal is $\perp_{\mathsf{r}(s)} = \{\perp, \lambda x.\perp\}$. The largest ideal $\top_s$ for sort s is simply $\mathbf{V}^s$. The type-collections S^s are defined inductively as the smallest fix-point of a series $\mathsf{S}^s_0, \mathsf{S}^s_1, \mathsf{S}^s_2, \ldots$. For notational convenience, we say that $\top_{\overline{s}} = \perp_s$ and $\perp_{\overline{s}} = \top_s$. Where the sort s is understood, we omit it. In the base case, type collections merely contain $\perp_s$ and $\top_s$.

$$\begin{aligned}
\mathsf{S}^{\mathbf{t}}_0 &= \{\perp_{\mathbf{t}}, \top_{\mathbf{t}}\} \\
\mathsf{S}^{\mathbf{ft}}_0 &= \{\perp_{\mathbf{ft}}, \top_{\mathbf{ft}}\} \\
\mathsf{S}^{\mathsf{r}(s)}_0 &= \{\perp_{\mathsf{r}(s)}, \top_{\mathsf{r}(s)}\} \\
\mathsf{S}^{\mathbf{s}}_0 &= \{\perp_{\mathbf{s}}, \top_{\mathbf{s}}\}
\end{aligned}$$

At level n of the induction, we define S^s_n in terms of two simpler collections T^s_n and O^s_n. T^s_n is the collection of ideals built by applying constructors of sort s to elements of

S^t_{n-1}, and O^s_n is built from T^s_n by adding ideals built using operations of sort s (besides meet and join operations). Finally, S^s_n is built from O^s_n by closing under finite meets and joins (defined below). The following equality defines how T^s_n is built from the collections S^t_{n-1}. (Note that for Row-sorts, this collection is empty since Row-sorts have no constructors).

$$\mathsf{T}^s_n = \{\phi_c(I_1, \ldots, I_k) \mid c : \iota_1 \cdots \iota_k \to s, I_j \in \mathsf{S}^{\iota_j}_{n-1}\} \tag{4.2}$$

O^s_n is obtained from T^s_n by adding the denotations of constants 0 and 1 for all sorts. Additionally, denotations for Row-constant $\langle\rangle$ and Row-composition are added to sorts $\mathbf{r}(s)$, and negations of single constructors are added to the Set-sort.

$$\mathsf{O}^s_n = \mathsf{T}^s_n \cup \{\bot_s, \top_s\} \cup \begin{cases} \emptyset & s \in \{\mathbf{t}, \mathbf{ft}\} \\ \{\rho_A(g) \circ K \mid A \overset{\mathsf{fin}}{\subseteq} \mathbf{L} \ \wedge \ g \in A \to \mathsf{S}^t_{n-1}\} & s = \mathbf{r}(t) \\ \{\top_s - \phi_c(\top_{\iota_1}, \ldots, \top_{\iota_{a(c)}}) \mid c : \iota_1 \cdots \iota_{a(c)} \to \mathbf{s}\} & s = \mathbf{s} \end{cases} \tag{4.3}$$

where K is $\bot_{\mathbf{r}(s)}$, $\mathsf{abs}_{\mathbf{r}(s)}$, or $\top_{\mathbf{r}(s)}$, and

$$\begin{aligned} \rho_A(g) =& \{f \in \mathbf{L} \to (\mathbf{V}^s \cup \{abs\}) \mid l \in A \implies f\ l \in g\ l \ \wedge \ l \notin A \implies f\ l \in \{\bot, abs\}\} \\ & \cup \{\bot\} \end{aligned} \tag{4.4}$$

and

$$\begin{aligned} \mathsf{abs}_{\mathbf{r}(s)} &= \{f \in \mathbf{L} \to (\mathbf{V}^s \cup \{abs\}) \mid f\ l \in \{\bot, abs\}\} \cup \{\bot\} \\ &= \mathbf{L} \to \{\bot, abs\} \end{aligned}$$

Note that $\mathsf{abs}_{\mathbf{r}(s)}$ is in $\mathsf{O}^{\mathbf{r}(s)}_n$ since $\rho_\emptyset(g) \circ \mathsf{abs}_{\mathbf{r}(s)} = \mathsf{abs}_{\mathbf{r}(s)}$.

Finally, collections S^s_n are obtained from O^s_n by closing under finite meets and joins ($\sqcap^n_s$, $\sqcup^n_s$ defined below)

$$\mathsf{S}^s_n = \min \mathsf{X} \supseteq \mathsf{O}^s_n \text{ such that for all non-empty finite subsets } \mathsf{Y} \text{ of } \mathsf{X}$$
$$\bigsqcap{}^n_s \mathsf{Y} \in \mathsf{X} \ \wedge \ \bigsqcup{}^n_s \mathsf{Y} \in \mathsf{X}$$

We now show that the inductive definition of the type collections S^s satisfy Invariant 4.10. Conditions 1–3 of Invariant 4.10 hold for the base collections S^s_0. Assuming they hold for S^t_{n-1}, then by equations (4.2) and (4.3) they hold for T^s_n and also for O^s_n. We now inductively define the two binary operations $\sqcap^n_s$ and $\sqcup^n_s$ that perform meets and joins of elements of S^s_n and show that the invariants also hold for S^s_n provided they hold for S^s_{n-1}. At the same time we show that for all sorts s and all induction levels n,

$$I \sqcap^n_s J \subseteq I \cap J \tag{4.5}$$
$$I \sqcup^n_s J \supseteq I \cup J \tag{4.6}$$

i.e., meets are smaller than set-theoretic intersections and joins are larger than set-theoretic unions.

For the Term-sort the meet and join operations are particularly simple. The join of two non-bottom elements of $\mathsf{O}^{\mathbf{t}}_n$ is $\top$ unless the elements are equal, or one is $\bot$. Similarly, the meet of two non-top elements is $\bot$ unless they are equal, or one is $\top$.

$$I \sqcap_{\mathbf{t}} J = \begin{cases} I & \text{if } I = J \\ I & \text{if } J = \top \\ J & \text{if } I = \top \\ \bot & \text{otherwise} \end{cases} \qquad I \sqcup_{\mathbf{t}} J = \begin{cases} I & \text{if } I = J \\ I & \text{if } J = \bot \\ J & \text{if } I = \bot \\ \top & \text{otherwise} \end{cases} \tag{4.7}$$

In the first three cases $I \sqcap_{\mathbf{t}} J = I \cap J$ and $I \sqcup_{\mathbf{t}} J = I \cup J$. In the last case, $\bot_{\mathbf{t}} = I \sqcap_{\mathbf{t}} J \subseteq I \cap J$ and $\top_{\mathbf{t}} = I \sqcup_{\mathbf{t}} J \supseteq I \cup J$. Thus (4.5) and (4.6) are satisfied for all n. The definition directly implies that for all $I, J \in \mathsf{O}^{\mathbf{t}}_n$ we have $I \sqcap_{\mathbf{t}} J \in \mathsf{O}^{\mathbf{t}}_n$ and $I \sqcup_{\mathbf{t}} J \in \mathsf{O}^{\mathbf{t}}_n$. Thus $\mathsf{S}^{\mathbf{t}}_n = \mathsf{O}^{\mathbf{t}}_n$ and Condition 1 of Invariant 4.10 is satisfied.

For notational convenience we define $\sqcap_{\overline{s}} = \sqcup_s$ and $\sqcup_{\overline{s}} = \sqcap_s$.

$$I \sqcap^n_{\mathbf{ft}} J = \begin{cases} \phi_c(I_1 \sqcap^{n-1}_{\iota_1} J_1, \ldots, I_k \sqcap^{n-1}_{\iota_k} J_k) & c : \iota_1 \cdots \iota_k \to \mathbf{ft} \\ \qquad I = \phi_c(I_1, \ldots, I_k) \ \wedge\ J = \phi_c(J_1, \ldots, J_k) & \\ I & \text{if } J = \top \\ J & \text{if } I = \top \\ \bot & \text{otherwise} \end{cases} \tag{4.8}$$

$$I \sqcup^n_{\mathbf{ft}} J = \begin{cases} \phi_c(I_1 \sqcup^{n-1}_{\iota_1} J_1, \ldots, I_k \sqcup^{n-1}_{\iota_k} J_k) & c : \iota_1 \cdots \iota_k \to \mathbf{ft} \\ \qquad I = \phi_c(I_1, \ldots, I_k) \ \wedge\ J = \phi_c(J_1, \ldots, J_k) & \\ I & \text{if } J = \bot \\ J & \text{if } I = \bot \\ \top & \text{otherwise} \end{cases} \tag{4.9}$$

The meet and join for $\mathsf{FlowTerm}$-ideals are defined in terms of the meets and joins of constructor arguments. The base cases ($n = 0$) degenerate to the cases for Term-ideals. Note that $I_j, J_j \in \mathsf{S}^{\iota_j}_{n-1}$. Since $\mathsf{S}^{\iota_j}_{n-1}$ is closed under meets and joins, we have $I_j \sqcup^{n-1}_{\iota_j} J_j \in \mathsf{S}^{\iota_j}_{n-1}$. Therefore for all $I, J \in \mathsf{O}^{\mathbf{ft}}_n$ we have $I \sqcap^n_{\mathbf{ft}} J \in \mathsf{O}^{\mathbf{ft}}_n$ and $I \sqcup^n_{\mathbf{ft}} J \in \mathsf{O}^{\mathbf{ft}}_n$. Thus $\mathsf{S}^{\mathbf{ft}}_n = \mathsf{O}^{\mathbf{ft}}_n$ and Condition 1 is satisfied. By induction on (4.5) and (4.6) and the fact that ϕ_c observes variance, we conclude that $I \sqcap^n_{\mathbf{ft}} J \subseteq I \cap J$ and $I \sqcup^n_{\mathbf{ft}} J \supseteq I \cup J$.

To simplify the cases for the meet and join of Row-ideals, we use the following definition.

Definition 4.11 *The* kind *of a Row-ideal $I \in \mathsf{S}^{\mathbf{r}(s)}_n$—written $k(I)$—is*

$$k(I) = \begin{cases} \top_{\mathbf{r}(s)} & \textit{if } I \textit{ is maximal} \\ \bot_{\mathbf{r}(s)} & \textit{if } I \textit{ is minimal} \\ \mathsf{abs}_{\mathbf{r}(s)} & \textit{if } I \textit{ is closed} \end{cases} \tag{4.10}$$

Note that $\bot_{\mathbf{r}(s)} \subset \mathsf{abs}_{\mathbf{r}(s)} \subseteq \top_{\mathbf{r}(s)}$ and thus these kinds are closed under union and intersection. The meet and join for elements in row type-collections is defined in terms of meets and joins of the range of each label and the meet and join of the kinds.

$$I \sqcap^n_{\mathbf{r}(s)} J = \rho_{\mathsf{dom}(I) \cup \mathsf{dom}(J)}(\lambda l. I(l) \overset{*}{\sqcap}{}^{n-1}_s J(l)) \circ k_\sqcap(I, J) \tag{4.11}$$

$$I \sqcup^n_{\mathbf{r}(s)} J = \rho_{\mathsf{dom}(I) \cap \mathsf{dom}(J)}(\lambda l. I(l) \sqcup^{n-1}_s J(l)) \circ k_\sqcup(I, J) \tag{4.12}$$

where

$$X \overset{*}{\sqcap}{}_s^{n-1} Y = (X - \{abs\}) \sqcap_s^{n-1} (Y - \{abs\})$$

and

$$k_\sqcap(I, J) = \begin{cases} \bot_{\mathsf{r}(s)} & \text{if } k(I) = k(I) \cap k(J) = \mathsf{abs}_{\mathsf{r}(s)} \\ & \text{and } \mathsf{dom}(J) \not\subseteq \mathsf{dom}(I) \\ \bot_{\mathsf{r}(s)} & \text{if } k(J) = k(I) \cap k(J) = \mathsf{abs}_{\mathsf{r}(s)} \\ & \text{and } \mathsf{dom}(I) \not\subseteq \mathsf{dom}(J) \\ k(I) \cap k(J) & \text{otherwise} \end{cases}$$

$$k_\sqcup(I, J) = \begin{cases} \top_{\mathsf{r}(s)} & \text{if } k(I) = k(I) \cup k(J) = \mathsf{abs}_{\mathsf{r}(s)} \\ & \text{and } \mathsf{dom}_\bot(J) \not\subseteq \mathsf{dom}(I) \\ \top_{\mathsf{r}(s)} & \text{if } k(J) = k(I) \cup k(J) = \mathsf{abs}_{\mathsf{r}(s)} \\ & \text{and } \mathsf{dom}_\bot(I) \not\subseteq \mathsf{dom}(J) \\ k(I) \cup k(J) & \text{otherwise} \end{cases}$$

In order for these operations to be meets and joins of the constraint relation $\subseteq_{\mathsf{r}(s)}$ which we will define shortly, we require that if the meet of I and J is closed, then its domain must be equal to the domain of the closed arguments (I, J, or I and J), and similarly for the join. The extra cases for $k_\sqcap$ and $k_\sqcup$ take care of this by making the meet minimal (join maximal), if these domain constraints are not satisfied. The special cases for the meet are furthermore needed to guarantee (4.6). Consider for example two closed Row-ideals I and J, such that $l \in \mathsf{dom}(I)$, $I(l) \neq \{\bot\}$, and $l \notin \mathsf{dom}(J)$. Then there exists an element f of I, such that $f\ l \notin \{\bot, abs\}$. Note that $l \notin \mathsf{dom}(I \sqcup J)$, thus $(I \sqcup J)(l) = \{\bot, abs\}$ if $k(I \sqcup J) = \mathsf{abs}_{\mathsf{r}(s)}$. But then $f \notin I \sqcup J$. The definition makes sure that in this case the kind of $I \sqcup J$ becomes $\top_{\mathsf{r}(s)}$ to guarantee that $f \in I \sqcup J$.

We show that for all $I, J \in \mathsf{O}_n^{\mathsf{r}(s)}$, $I \sqcup_{\mathsf{r}(s)}^n J$ is an element of $\mathsf{O}_n^{\mathsf{r}(s)}$. In the base case ($n = 0$) the (minimum) domains of I and J are empty and the meet and join reduce to intersections and unions of kinds. Otherwise, $I(l), J(l) \in \mathsf{S}_{n-1}^s$ for $l \in \mathsf{dom}(I) \cap \mathsf{dom}(J)$. By induction, $I(l) \sqcup_s^{n-1} J(l) \in \mathsf{S}_{n-1}^s$. Suppose $\mathsf{dom}(I) \cap \mathsf{dom}(J)$ is finite. Then, $\rho_{\mathsf{dom}(I) \cap \mathsf{dom}(J)}(\lambda l.I(l) \sqcup_s^{n-1} J(l)) \circ k_\sqcup(I, J)$ is an element of $\mathsf{O}_n^{\mathsf{r}(s)}$ by definition. Otherwise, I and J are minimal. In this case, $\mathsf{dom}_\bot(I) \cup \mathsf{dom}_\bot(J) \subset \mathsf{dom}(I) \cap \mathsf{dom}(J)$ is finite and thus $I \sqcup J$ is equivalent to

$$\rho_{\mathsf{dom}_\bot(I) \cup \mathsf{dom}_\bot(J)}(\lambda l.I(l) \sqcup_s^{n-1} J(l)) \circ \bot_{\mathsf{r}(s)}$$

which is in $\mathsf{O}_n^{\mathsf{r}(s)}$. For $\sqcap_{\mathsf{r}(s)}^n$ the argument is similar with the additional observation that if $abs \in I(l)$ for some $I \in \mathsf{S}_n^{\mathsf{r}(s)}$, then $I(l) - \{abs\} \in \mathsf{S}_{n-1}^s$. Thus $\mathsf{S}_n^{\mathsf{r}(s)} = \mathsf{O}_n^{\mathsf{r}(s)}$ and Condition 3 is satisfied. For (4.5) note that if $f \in I \sqcap_{\mathsf{r}(s)}^n J$, then

$$\begin{aligned} l \in \mathsf{dom}(I) \cup \mathsf{dom}(J) &\implies f\ l \in I(l) \overset{*}{\sqcap}{}_s^{n-1} J(l) \\ \wedge\ l \notin \mathsf{dom}(I) \cup \mathsf{dom}(J) &\implies f\ l \in (k(I) \cap k(J))(l) \end{aligned}$$

By induction, $I(l) \overset{*}{\sqcap}{}_s^{n-1} J(l) \subseteq I(l) \cap J(l)$ and thus $f \in I$ and $f \in J$. The argument for (4.6) is given above.

The crux of the definition comes with the Set-sort. If we use set-theoretic intersection and union for the meet and join operations, we cannot guarantee Condition 2, since projections of non-strict interface paths p^t (of length $|p| \leq n$) yield arbitrary unions and intersections of elements in $\mathsf{S}^t_{n-|p|}$ which themselves are not necessarily in $\mathsf{S}^t_{n-|p|}$. We therefore adjust the set-theoretic union and intersection to establish the invariant.

$$\begin{aligned} I \sqcap^n_{\mathsf{s}} J = \ & \max X \in \mathcal{I}(\mathbf{V}^{\mathsf{s}}) \text{ st. } X \subseteq I \cap J \\ & \text{and for all finite non-strict } p^t_{\mathsf{s}} \\ & \begin{cases} p^{-1}(X) = p^{-1}(I) \sqcap^k_t p^{-1}(J) & \text{if } p \text{ even, } k = n - |p| \\ p^{-1}(X) = p^{-1}(I) \sqcup^k_t p^{-1}(J) & \text{if } p \text{ odd, } k = n - |p| \end{cases} \end{aligned} \tag{4.13}$$

$$\begin{aligned} I \sqcup^n_{\mathsf{s}} J = \ & \min X \in \mathcal{I}(\mathbf{V}^{\mathsf{s}}) \text{ st. } X \supseteq I \cup J \\ & \text{and for all finite non-strict } p^t_{\mathsf{s}} \\ & \begin{cases} p^{-1}(X) = p^{-1}(I) \sqcup^k_t p^{-1}(J) & \text{if } p \text{ even, } k = n - |p| \\ p^{-1}(X) = p^{-1}(I) \sqcap^k_t p^{-1}(J) & \text{if } p \text{ odd, } k = n - |p| \end{cases} \end{aligned} \tag{4.14}$$

To see that the operations $\sqcap^n_{\mathsf{s}}$ and $\sqcup^n_{\mathsf{s}}$ are well-defined we give an explicit construction of the maximal (minimal) X. We will use the following auxiliary definitions. For any finite non-strict path p, and element Y of $\mathsf{S}^s_{i-|p|}$, define

$$\min_p(Y) = \begin{cases} Y & \text{if } p = \epsilon \\ \phi_c(\bot_{\iota_1} .. \bot_{\iota_{j-1}}, \min_q(Y), \bot_{\iota_{j+1}} .. \bot_{\iota_{a(c)}}) & \text{if } p = (c,j)q \\ & \text{covariant in } j \\ \phi_c(\bot_{\iota_1} .. \bot_{\iota_{j-1}}, \max_q(Y), \bot_{\iota_{j+1}} .. \bot_{\iota_{a(c)}}) & \text{if } p = (c,j)q \\ & \text{contravariant in } j \end{cases} \tag{4.15}$$

$$\max_p(Y) = \begin{cases} Y & \text{if } p = \epsilon \\ \top_s - \phi_c(\top_{\iota_1} .. \top_{\iota_{a(c)}}) \cup \phi_c(\top_{\iota_1} .. \top_{\iota_{j-1}}, \max_q(Y), \top_{\iota_{j+1}} .. \top_{\iota_{a(c)}}) & \text{if } p = (c,j)q \\ & \text{covariant in } j \\ \top_s - \phi_c(\top_{\iota_1} .. \top_{\iota_{a(c)}}) \cup \phi_c(\top_{\iota_1} .. \top_{\iota_{j-1}}, \min_q(Y), \top_{\iota_{j+1}} .. \top_{\iota_{a(c)}}) & \text{if } p = (c,j)q \\ & \text{contravariant in } j \end{cases} \tag{4.16}$$

Note that $\max_p(Y) \in \mathsf{S}_n$. Using these min and max ideals, we can define $\sqcap^n_{\mathsf{s}}$ and $\sqcup^n_{\mathsf{s}}$ explicitly (p ranges over non-strict paths present in I and J where $k = n - |p| \geq 0$).

$$\begin{aligned} I \sqcap^n_{\mathsf{s}} J = \ & I \cap J \\ & \cap \bigcap_{\text{even } p^t} \max_p \left(p^{-1}(I) \sqcap^k_t p^{-1}(J)\right) \\ & \cap \bigcap_{\text{odd } p^t} \max_p \left(p^{-1}(I) \sqcup^k_t p^{-1}(J)\right) \end{aligned} \tag{4.17}$$

$$\begin{aligned} I \sqcup^n_{\mathsf{s}} J = \ & I \cup J \\ & \cup \bigcup_{\text{even } p^t} \min_p \left(p^{-1}(I) \sqcup^k_t p^{-1}(J)\right) \\ & \cup \bigcup_{\text{odd } p^t} \min_p \left(p^{-1}(I) \sqcap^k_t p^{-1}(J)\right) \end{aligned} \tag{4.18}$$

We illustrate these definitions with an example.

Example 4.12 Consider two constant Term-constructors $\mathsf{bool} : \mathbf{t}$ and $\mathsf{int} : \mathbf{t}$ which denote boolean and integer values, and a covariant unary mixed Set-constructor $\mathsf{list} : \mathbf{t} \to \mathbf{s}$ for lists. Assuming semantic constructors *nil* and *cons*, the meaning function ϕ_{list} is defined by

$$\phi_{\mathsf{list}}(A) = X \text{ where } X = \{\mathit{nil}\} \cup \{\mathit{cons}(h,t) \mid h \in A - \{\bot\} \wedge t \in X\}$$

Suppose we form the join of two Set-expressions

$$\mathsf{list}(\mathsf{bool}) \sqcup \mathsf{list}(\mathsf{int})$$

and consider the natural denotation I of the above expression given by

$$I = \phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \sqcup_{\mathsf{s}} \phi_{\mathsf{list}}(\phi_{\mathsf{int}})$$

If $\sqcup_{\mathsf{s}}$ were simply Set-theoretic union, then the projection $p^{-1}(I)$ of interface path $p = (\mathsf{list}, 1)$ results in the ideal $J = \phi_{\mathsf{bool}} \cup \phi_{\mathsf{int}}$. However, $J \notin \mathsf{S}^{\mathbf{t}}$, since it is not of the form $\phi_c(\dots)$, *i.e.*, there is no unique head constructor c for J. Our definition of $\sqcup_{\mathsf{s}}$ avoids this problem by raising the above join, such that $p^{-1}(J) = \phi_{\mathsf{bool}} \sqcup_{\mathbf{t}} \phi_{\mathsf{int}} = \top_{\mathbf{t}}$. In particular,

$$\begin{aligned}
\phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \sqcup_{\mathsf{s}} \phi_{\mathsf{list}}(\phi_{\mathsf{int}}) &= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \cup \phi_{\mathsf{list}}(\phi_{\mathsf{int}}) \cup \bigcup_{\text{even } p^t} \min_p \left(p^{-1}(I) \sqcup_t p^{-1}(J)\right) \\
&= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \cup \phi_{\mathsf{list}}(\phi_{\mathsf{int}}) \cup \min_{(\mathsf{list},1)} (\phi_{\mathsf{bool}} \sqcup_{\mathbf{t}} \phi_{\mathsf{int}}) \\
&= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \cup \phi_{\mathsf{list}}(\phi_{\mathsf{int}}) \cup \phi_{\mathsf{list}}(\top_{\mathbf{t}}) \\
&= \phi_{\mathsf{list}}(\top_{\mathbf{t}})
\end{aligned}$$

where the second to last step follows from the definition of $\sqcup_{\mathbf{t}}$ and $\min_p$, and the last step follows from covariance of ϕ_{list}. The projection $p^{-1}(I)$ is thus $\top_{\mathbf{t}}$, which is the only Term-ideal larger than both ϕ_{bool} and ϕ_{int}.
■

To see that construction (4.17) for $I \sqcap_{\mathsf{s}}^n J$ satisfies requirement (4.13) in general, note that we have for all finite even interface paths p^t,

$$\begin{aligned}
p^{-1}(I \sqcap_{\mathsf{s}}^n J) &= p^{-1}(I) \cap p^{-1}(J) \cap p^{-1}\left(\bigcap_{\text{even } q^t} \max_q^n \left(q^{-1}(I) \sqcap_t^k q^{-1}(J)\right)\right) \\
&= p^{-1}(I) \cap p^{-1}(J) \cap p^{-1}\left(\max_p^n \left(p^{-1}(I) \sqcap_t^k p^{-1}(J)\right)\right) \\
&= p^{-1}(I) \cap p^{-1}(J) \cap p^{-1}(I) \sqcap_t^k p^{-1}(J) \\
&= p^{-1}(I) \sqcap_t^k p^{-1}(J)
\end{aligned}$$

The first equality follows from Lemma 4.8, the second simply chooses the path $q = p$ from the intersection $\bigcap_{\text{even } q}$, and the last equality follows from the inductive hypothesis that $I \sqcap_s^k J \subseteq I \cap J$ for all sorts s and $k = n - |p| < n$. The case for odd paths is analogous, and similarly for $\sqcup_s^n$.

Note that by Conditions iii, vi and vii of Axiom 4.1, $\min_p(Y)$ is by construction the smallest set X such that $p^{-1}(X) = Y$. Assume that Z is another set satisfying (4.14), then $\min_p(p^{-1}(I) \sqcup_t p^{-1}(J)) \subseteq Z$ for all finite even non-strict interface paths p^t and $\min_p(p^{-1}(I) \sqcap_t p^{-1}(J)) \subseteq Z$ for all finite odd non-strict interface paths p^t. Thus $Z \supseteq I \sqcup_{\mathsf{s}} J$ and $I \sqcup_{\mathsf{s}} J$ is minimal. The argument for the maximality of $I \sqcap_{\mathsf{s}} J$ is analogous. By construction we have $I \sqcap^n_{\mathsf{s}} J \subseteq I \cap J$ and $I \sqcup^n_{\mathsf{s}} J \supseteq I \cup J$. Condition 2 is satisfied by the definition.

The type-collections S^s are then defined as the limits of the series $\mathsf{S}^s_0, \mathsf{S}^s_1, \ldots$.

Theorem 4.13 *The limits* S^s *exist.*

Proof: Each series $\mathsf{S}^s_0, \mathsf{S}^s_1, \ldots$ is monotonically increasing in the powerset lattice of ideals $\mathcal{P}(\mathcal{I}(\mathbf{V}^s))$. This lattice is complete, and thus the series $\mathsf{S}^s_0, \mathsf{S}^s_1, \ldots$ has a limit, which is given by the least upper bound of the series. $\square$

4.4 Semantic Relations

Having defined our type-collections, we are ready to define the semantic relations $\subseteq_s$ and show that they form lattices over S^s with meet and join being $\sqcap_s$ and $\sqcup_s$ respectively. A difficulty in defining the semantic relations for mixed constraints not present in non-mixed formalisms is that the constraint relations are interdependent. For example, the constraint relation for mixed Set-constraints is not simply set-theoretic inclusion. The relation depends on the constraint relations of other sorts that appear in mixed Set-expressions.

We now define the semantic relation $\subseteq_s$ for each sort s and show that $I \subseteq_s J \implies I \subseteq_{\mathbf{V}^s} J$, *i.e.*, the semantic relations $\subseteq_s$ imply set-theoretic inclusion in the underlying value domain $\mathbf{V}^s$. For notational convenience, we state that $I \subseteq_{\overline{s}} J \iff J \subseteq_s I$.

The semantic relation for the Term-sort is

$$\begin{aligned} \bot &\subseteq_{\mathsf{t}} I \\ I &\subseteq_{\mathsf{t}} \top \\ I &\subseteq_{\mathsf{t}} I \end{aligned} \tag{4.19}$$

Note that $I \subseteq_{\mathsf{t}} J$ implies $I \subseteq J$. For the FlowTerm-sort, the semantic relation is

$$\begin{aligned} \bot &\subseteq_{\mathsf{ft}} I \\ I &\subseteq_{\mathsf{ft}} \top \\ \phi_c(I_1, \ldots, I_n) &\subseteq_{\mathsf{ft}} \phi_c(J_1, \ldots, J_n) \iff I_j \subseteq_{\iota_j} J_j \end{aligned} \tag{4.20}$$

Note that $I \subseteq_{\mathsf{ft}} J$ implies $I \subseteq J$. The first two cases are trivial. For the last case note that ϕ_c observes the declared variance of constructor c (Axiom 4.1, Condition iii). For rows, the semantic relation is

$$\begin{aligned} I \subseteq_{\mathsf{r}(s)} J \iff & I \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} J \ \wedge \\ & I \text{ and } J \text{ closed} \implies \mathsf{dom}(I) = \mathsf{dom}(J) \ \wedge \\ & \text{forall } l \in \mathsf{dom}(J) \ I(l) \subseteq_s J(l) \end{aligned} \tag{4.21}$$

Two Row-ideals I and J stand in I $\subseteq_{\mathsf{r}(s)}$ relation iff I is a set-theoretic subset of J (written $I \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} J$), the domains of I and J agree if both are closed, and for all labels l in the domain of J, the projection $I(l)$ is related to $J(l)$ according to $\subseteq_s$. Note that $I \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} J$ implies $\mathsf{dom}(I) \supseteq \mathsf{dom}(J)$.

The semantic relation $\subseteq_{\mathbf{s}}$ has two parts, a standard set-theoretic inclusion on the domain elements (written $\subseteq_{\mathbf{V}^{\mathsf{s}}}$ for clarity), and a set of *interface constraints* using interface paths. Our definition of S^{s} guarantees that for any interface path p^t and any ideal $I \in \mathsf{S}^{\mathsf{s}}$, $p^{-1}(I) \in \mathsf{S}^t$ and thus the interface constraints are well-defined.

$$I \subseteq_{\mathbf{s}} J \iff I \subseteq_{\mathbf{V}^{\mathsf{s}}} J \;\wedge\; \text{forall finite non-strict interface paths } p^t \\ \begin{cases} p^{-1}(I) \subseteq_t p^{-1}(J) & p \text{ even} \\ p^{-1}(J) \subseteq_t p^{-1}(I) & p \text{ odd} \end{cases} \tag{4.22}$$

Note how the relations are stronger than the set-theoretic inclusions in the underlying domain. This fact is not surprising, since the major goal of mixed constraints is to narrow the set of possible solutions such that more specialized and efficient algorithms can be applied.

Example 4.14 Reconsider the covariant mixed Set-constructor $\mathsf{list} : \mathbf{t} \to \mathbf{s}$ from Example 4.12. Suppose I and J are two Term-ideals from $\mathsf{S}^{\mathbf{t}}$ and consider the set-theoretic relation

$$\phi_{\mathsf{list}}(I) \subseteq \phi_{\mathsf{list}}(J) \tag{4.23}$$

By covariance of ϕ_{list}, this relation is satisfied iff $I \subseteq J$. Now consider the relation $\subseteq_{\mathbf{s}}$

$$\phi_{\mathsf{list}}(I) \subseteq \phi_{\mathsf{list}}(J)$$

which requires (4.23) and in addition

$$I \subseteq_{\mathbf{t}} J \tag{4.24}$$

From the definition of $\subseteq_{\mathbf{t}}$, the relation (4.24) is satisfied if $I = J$, unless $I = \bot_{\mathbf{t}}$ or $J = \top_{\mathbf{t}}$. The equality constraints arising through the use of Term-expressions can be represented and solved more efficiently than inclusion constraints (Section 7.3).
■

Lemma 4.15 *The operations $\sqcap_s$ and $\sqcup_s$ are the meet and join operations of $\subseteq_s$.*

Proof: By induction on S^s_i. For S^s_0 the lemma holds, since meet and join degenerate to intersection and union in that case.

For $I \sqcup_{\mathbf{t}} J$ we have $I \subseteq_{\mathbf{t}} I \sqcup_{\mathbf{t}} J$ and $J \subseteq_{\mathbf{t}} I \sqcup_{\mathbf{t}} J$ from the definition. Suppose there exists another upper-bound Z. If Z is $\bot_{\mathbf{t}}$ or $\top_{\mathbf{t}}$ we are done. Otherwise, $Z = I = J = I \sqcup_{\mathbf{t}} J$ for all n. The case for $\sqcap_{\mathbf{t}}$ is similar.

For $I \sqcup_{\mathbf{ft}} J$ we have by induction $I_j \subseteq_{\mathbf{ft}} I_j \sqcup^{n-1}_{\iota_j} J_j$ and $J_j \subseteq_{\mathbf{ft}} I_j \sqcup^{n-1}_{\iota_j} J_j$. Thus by variance of ϕ_c, we have $I \subseteq_{\mathbf{ft}} I \sqcup^n_{\mathbf{ft}} J$ and $J \subseteq_{\mathbf{ft}} I \sqcup^n_{\mathbf{ft}} J$. For the inductive step of minimality, suppose there exists another upper-bound $Z \in \mathsf{S}^{\mathbf{ft}}$ of I and J. If Z is $\bot_{\mathbf{ft}}$ or

$\top_{\mathsf{ft}}$ we are done. Otherwise, since constructor interpretations are disjoint, Z must be of the form $\phi_c(Z_1, \dots, Z_k)$ with $I_j \subseteq_{\iota_j} Z_j$ and $J_j \subseteq_{\iota_j} Z_j$. By induction, $I_j \sqcup_{\iota_j}^{n-1} J_j \subseteq_{\iota_j} Z_j$, and thus by variance of ϕ_c, $I \sqcup_{\mathsf{ft}}^n J \subseteq_{\mathsf{ft}} Z$. The case for $\sqcap_{\mathsf{ft}}$ is similar.

For $I \sqcup_{\mathsf{r}(s)} J$ we need to show that $I \subseteq_{\mathsf{r}(s)} I \sqcup_{\mathsf{r}(s)} J$, *i.e.*, 1) $I \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} I \sqcup_{\mathsf{r}(s)} J$, 2) if I, $I \sqcup_{\mathsf{r}(s)} J$ are closed then $\mathsf{dom}(I) = \mathsf{dom}(I \sqcup_{\mathsf{r}(s)} J)$, and 3) that for all $l \in \mathsf{dom}(I \sqcup_{\mathsf{r}(s)} J)$ $I(l) \subseteq_s (I \sqcup_{\mathsf{r}(s)} J)(l)$. We have $I \sqcup_{\mathsf{r}(s)} J = \rho_{\mathsf{dom}(I) \cap \mathsf{dom}(J)}(\lambda l. I(l) \sqcup_s^{n-1} J(l)) \circ k_\sqcup(I, J)$. Since $I \sqcup_{\mathsf{r}(s)} J \supseteq I \cup J \supseteq I$, we have the first part. For the second part, suppose I and $I \sqcup_{\mathsf{r}(s)} J$ are closed. Then by definition of $\sqcup_{\mathsf{r}(s)}$ we have that J is not maximal, and that $\mathsf{dom}_\perp(J) \subseteq \mathsf{dom}(I)$. If J is minimal, then $\mathsf{dom}(I \sqcup_{\mathsf{r}(s)} J) = \mathsf{dom}(I) \cap \mathsf{dom}(J) = \mathsf{dom}(I)$. If J is closed, then we have $\mathsf{dom}(J) \subseteq \mathsf{dom}(I)$ and symmetrically, $\mathsf{dom}(I) \subseteq \mathsf{dom}(J)$. For the third part we have for all $l \in \mathsf{dom}(I \sqcup_{\mathsf{r}(s)} J) = \mathsf{dom}(I) \cap \mathsf{dom}(J)$ that $(I \sqcup_{\mathsf{r}(s)} J)(l) = I(l) \sqcup_s^{n-1} J(l)$. By induction, $I(l) \subseteq_s I(l) \sqcup_s^{n-1} J(l)$ and thus for all $l \in \mathsf{dom}(I \sqcup_{\mathsf{r}(s)}^n J)$ we have $I(l) \subseteq_s (I \sqcup_{\mathsf{r}(s)}^n J)(l)$ as desired. The case for $\sqcap_{\mathsf{r}(s)}^n$ is analogous.

To show minimality, suppose there exists another upper-bound $Z \in \mathsf{S}^{\mathsf{r}(s)}$ of I and J. We have $I \cup J \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} Z$ implying that $\mathsf{dom}(Z) \subseteq \mathsf{dom}(I \cup J) = \mathsf{dom}(I) \cap \mathsf{dom}(J) = \mathsf{dom}(I \sqcup_{\mathsf{r}(s)} J)$. Suppose $\mathsf{dom}(Z) = \mathsf{dom}(I \sqcup_{\mathsf{r}(s)} J)$, otherwise we are done. Clearly, $k(Z) \supseteq k_\sqcup(I, J)$, otherwise Z is not an upper-bound. Assume $k(Z) = k_\sqcup(I, J)$, otherwise we are done. Then for all $l \in \mathsf{dom}(Z)$ we have $I(l) \subseteq_s Z(l)$ and $J(l) \subseteq_s Z(l)$. By induction, $I(l) \sqcup_s J(l) \subseteq_s Z(l)$ and thus $(I \sqcup_{\mathsf{r}(s)} J)(l) \subseteq_s Z(l)$. The case for $\sqcap_{\mathsf{r}(s)}$ is similar.

Finally, for $I \sqcup_{\mathsf{s}} J$ we have by definition that $I \subseteq_{\mathbf{V}^{\mathsf{s}}} I \sqcup_{\mathsf{s}} J$ and $J \subseteq_{\mathbf{V}^{\mathsf{s}}} I \sqcup_{\mathsf{s}} J$. Also by definition, for all even interface paths p^t, $p^{-1}(I) \subseteq_t p^{-1}(I \sqcup_{\mathsf{s}}^n J) = p^{-1}(I) \sqcup_t^k p^{-1}(J)$ and for all odd interface paths p^t, $p^{-1}(I \sqcup_{\mathsf{s}}^n J) = p^{-1}(I) \sqcap_t^k p^{-1}(J) \subseteq_t p^{-1}(I)$. Similarly for J. Thus by induction ($k < n$), we have $I \subseteq_{\mathsf{s}} I \sqcup_{\mathsf{s}}^n J$ and $J \subseteq_{\mathsf{s}} I \sqcup_{\mathsf{s}}^n J$. Suppose there exists another upper-bound Z. Then $Z \supseteq I \cup J$ and $p^{-1}(I) \subseteq_t p^{-1}(Z)$ and $p^{-1}(J) \subseteq_t p^{-1}(Z)$ for even non-strict interface paths p^t, and $p^{-1}(Z) \subseteq_t p^{-1}(I)$ and $p^{-1}(Z) \subseteq_t p^{-1}(J)$ for odd non-strict interface paths p^t. By induction, $p^{-1}(I) \sqcup_t p^{-1}(J) \subseteq_t p^{-1}(Z)$ for even p and $p^{-1}(Z) \subseteq_t p^{-1}(I) \sqcap_t p^{-1}(J)$ for odd p. Thus by variance of constructor interpretations, $I \sqcup_{\mathsf{s}} J \subseteq_{\mathsf{s}} Z$. The case for $\sqcap_{\mathsf{s}}$ is analogous. □

Corollary 4.16 *For all n, the collections S_n^s with relations $\subseteq_s$ form lattices with meets and joins defined as above; thus for all s*

$$I \sqcup_s J \subseteq_s Z \iff I \subseteq_s Z \ \wedge \ J \subseteq_s Z$$
$$I \subseteq_s J \sqcap_s Z \iff I \subseteq_s J \ \wedge \ I \subseteq_s Z$$

The motivation for our construction of the lattices $(\mathsf{S}^s, \subseteq_s)$ was to exclude certain undesired ideals in $\mathcal{I}(\mathbf{V}^s)$ from S^s, and to make the relations $\subseteq_s$ stronger than set-theoretic inclusion. These two points where illustrated by Example 4.12 and Example 4.14. Another reason for the non-standard definition of S^{s} is to rule out sets that cannot be built using constructor application, union, and intersection. Reconsider the standard interpretation of a list constructor given in Example 4.12

$$\phi_{\mathsf{list}}(A) = X \text{ where } X = \{\mathit{nil}\} \cup \{\mathit{cons}(h, t) \mid h \in A - \{\perp\} \wedge t \in X\}$$

The set of lists of length greater than 0 (not containing *nil*) is an ideal of $\mathbf{V}^s$, but it cannot be constructed by applying ϕ_{list}. Note that a list constructor is *not* the only way to refer to ideals for modeling lists. More precise ideals for lists can be denoted by using two distinct constructors nil and cons. Using these constructors it is possible to distinguish non-empty lists from all lists, a distinction that cannot be captured with the list constructor above.

4.5 Solutions

To give meaning to mixed constraints we define a meaning function, mapping expressions to types, and then say that the solutions to a set of syntactic constraints are the solutions of their semantic counterparts.

Variable assignments σ are maps from variables to elements of S^s. A variable assignment σ is *well-sorted*, if for all sorts s and all $\mathcal{X} \in \mathsf{V}^s$ we have $\mathcal{X} \in \mathsf{dom}(\sigma) \implies \sigma(\mathcal{X}) \in \mathsf{S}^s$. We consider only well-sorted variable assignments. Given a variable assignment σ, we define the meaning μ of mixed expressions by

$$\begin{array}{rcll}
\mu[\![\mathcal{X}]\!]\sigma & = & \sigma(\mathcal{X}) & \\
\mu[\![0^s]\!]\sigma & = & \bot_s & \\
\mu[\![1^s]\!]\sigma & = & \top_s & \\
\mu[\![c(E_1,\dots,E_n)]\!]\sigma & = & \phi_c(\mu[\![E_1]\!]\sigma,\dots,\mu[\![E_n]\!]\sigma) & \\
\mu[\![E_1 \sqcap E_2]\!]\sigma & = & \mu[\![E_1]\!]\sigma \sqcap_{\mathsf{s}} \mu[\![E_2]\!]\sigma & \\
\mu[\![E_1 \sqcup E_2]\!]\sigma & = & \mu[\![E_1]\!]\sigma \sqcup_{\mathsf{s}} \mu[\![E_2]\!]\sigma & \\
\mu[\![\neg\{c_1,\dots,c_n\}]\!]\sigma & = & \top_{\mathsf{s}} - \bigcup\limits_{c\in\{c_1,\dots,c_n\}} \phi_c(\top_{\iota_1},\dots,\top_{\iota_{a(c)}}) & c : \iota_1 \cdots \iota_{a(c)} \to \mathsf{s} \\
\mu[\![\langle\rangle^s]\!]\sigma & = & \mathsf{abs}_{\mathsf{r}(s)} & \\
\mu[\![\langle l : E_l\rangle_{\{l\}} \circ E]\!]\sigma & = & \rho_{\{l\}}(\lambda l.\mu[\![E_l]\!]\sigma) \circ \mu[\![E]\!]\sigma &
\end{array}$$

Since S^{s} is closed under meet and join, and $\mathsf{S}^{\mathsf{r}(s)}$ closed under composition, we have that μ is well-sorted under any well-sorted variable assignment σ, *i.e.*, $E \in \mathsf{L}^s$ implies $\mu[\![E]\!]\sigma \in \mathsf{S}^s$.

We can now define the meaning of mixed constraints. Under a given set of constructor signatures Σ, a mixed constraint problem S is the conjunction of constraint problems S^s, where each S^s is a collection of s-constraints $\{E_1 \subseteq_s E_1',\dots,E_n \subseteq_s E_n'\}$. A solution to such a system S is a well-sorted variable assignment σ, such that

$$\mu[\![E_i]\!]\sigma \subseteq_s \mu[\![E_i']\!]\sigma \quad \text{forall } E_i \subseteq_s E_i' \in S$$

4.6 Discussion and Related Work

We start our discussion of the semantics of mixed constraints with a number of examples that illustrate how the choice of constructor signatures influences the precision of mixed constraints. Assume list is the unary constructor for lists with the semantic interpretation ϕ_{list} given in Example 4.12, and bool and int are constant constructors denoting the set of boolean values and the set of integers. Consider the two constraints

$$\begin{array}{r}
\mathsf{list}(\mathsf{bool}) \subseteq_s \mathcal{X} \\
\mathsf{list}(\mathsf{int}) \subseteq_s \mathcal{X}
\end{array} \qquad (4.25)$$

We now explore the possible minimal solution for $\mathcal{X}$ under a number of constructor signatures for $\mathsf{list}, \mathsf{bool}$ and int. First consider the signatures

$$\begin{aligned}\mathsf{list} &: \mathbf{s} \to \mathbf{s}\\ \mathsf{bool} &: \mathbf{s}\\ \mathsf{int} &: \mathbf{s}\end{aligned}$$

defining all constructors as Set-constructors. Then the constraints (4.25) have the minimal solution

$$\begin{aligned}\mathcal{X} &= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \sqcup_{\mathbf{s}} \phi_{\mathsf{list}}(\phi_{\mathsf{int}})\\ &= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \cup \phi_{\mathsf{list}}(\phi_{\mathsf{int}})\end{aligned}$$

expressing that $\mathcal{X}$ is the set of integer and boolean lists, where each list contains either only booleans, or only integers. Next consider the variation on the signatures where list is made a $\mathsf{FlowTerm}$-constructor.

$$\begin{aligned}\mathsf{list} &: \mathbf{s} \to \mathbf{ft}\\ \mathsf{bool} &: \mathbf{s}\\ \mathsf{int} &: \mathbf{s}\end{aligned}$$

In this case the minimal solution of constraints (4.25) is

$$\begin{aligned}\mathcal{X} &= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \sqcup_{\mathbf{ft}} \phi_{\mathsf{list}}(\phi_{\mathsf{int}})\\ &= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}} \sqcup_{\mathbf{s}} \phi_{\mathsf{int}})\\ &= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}} \cup \phi_{\mathsf{int}})\end{aligned}$$

expressing that $\mathcal{X}$ is the set of lists where each element is either a boolean or an integer. This set is strictly larger than the minimal solution under the previous signatures. Another coarsening can be obtained by choosing all constructors to be of sort $\mathsf{FlowTerm}$.

$$\begin{aligned}\mathsf{list} &: \mathbf{ft} \to \mathbf{ft}\\ \mathsf{bool} &: \mathbf{ft}\\ \mathsf{int} &: \mathbf{ft}\end{aligned}$$

In this case the minimal solution of constraints (4.25) is

$$\begin{aligned}\mathcal{X} &= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \sqcup_{\mathbf{ft}} \phi_{\mathsf{list}}(\phi_{\mathsf{int}})\\ &= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}} \sqcup_{\mathbf{ft}} \phi_{\mathsf{int}})\\ &= \phi_{\mathsf{list}}(\top_{\mathbf{ft}})\end{aligned}$$

expressing that $\mathcal{X}$ is the set of all lists. A final coarsening can be obtained by making list a Term-constructor.

$$\begin{aligned}\mathsf{list} &: \mathbf{ft} \to \mathbf{t}\\ \mathsf{bool} &: \mathbf{ft}\\ \mathsf{int} &: \mathbf{ft}\end{aligned}$$

In this case the minimal solution of constraints (4.25) is

$$\begin{aligned}\mathcal{X} &= \phi_{\mathsf{list}}(\phi_{\mathsf{bool}}) \sqcup_{\mathbf{t}} \phi_{\mathsf{list}}(\phi_{\mathsf{int}}) \\ &= \top_{\mathbf{t}}\end{aligned}$$

expressing that $\mathcal{X}$ is the set of all values.

Condition 2 of Invariant 4.10 only applies to non-strict interface paths. To illustrate why, we show that the definition of meet and join for Set-expressions does not extend to strict interface paths. Consider the following signatures.

$$\begin{array}{lcl} a & : & \mathbf{t} \\ b & : & \mathbf{t} \\ c & : & \mathbf{t}\,\mathbf{t} \to \mathbf{s} \end{array}$$

Let $X = \mu[\![c(a,b) \sqcup c(b,a)]\!]$ be the join of $c(a,b)$ and $c(b,a)$. If c is non-strict, the construction of the join adds the sets $\min_{(c,1)}(a \sqcup_{\mathbf{t}} b)$ and $\min_{(c,2)}(b \sqcup_{\mathbf{t}} a)$ so that the projections $(c,1)^{-1}(X)$ and $(c,2)^{-2}(X)$ yield elements of $\mathsf{S}^{\mathbf{t}}$. We relied on the minimum and maximum sets $\min_p$ and $\max_p$ to force the projection of interface paths of the join and meet to be elements of the appropriate type collection. The existence of such minimal and maximal sets is guaranteed by Axiom 4.1 (Conditions iii,vi,vii). If c is strict however, then the meaning function ϕ_c is not injective and the construction $\min_{(c,1)}(X)$ is always $\{\bot\}$ for strict constructors with more than one argument. Thus $(c,1)^{-1}(\min_{(c,1)}(X)) \neq X$ for all X strictly larger than $\{\bot\}$.

Most common constructor interpretations satisfy Axiom 4.1. Unfortunately, the standard function constructor interpretation does not satisfy Condition vii of Axiom 4.1 since it is non-injective if the first argument is $\{\bot\}$. Instead of the standard function interpretation, we can use a non-standard function interpretation that is injective. Consider the interpretation,

$$\phi_{\to}(X,Y) = \{(f,r) \mid f \in (\top_{\mathbf{t}} \to \top_{\mathbf{t}} \cup \{\mathit{wrong}\}) \cup \bot, r \in Y \ \wedge \ x \in X \implies f\ x \in Y\} \cup \{\bot\}$$

which pairs each function with a witness r from its range. Now for $A \subset \top_{\mathbf{t}}$ the set $\phi_{\to}(\bot_{\mathbf{t}}, A)$ is distinct from $\phi_{\to}(\bot_{\mathbf{t}}, \top_{\mathbf{t}})$ since the first set only contains pairs (f,r) where f is any function, and $r \in A$. Such an interpretation is still valid for the purposes of modeling sets of functions in programs. It merely provides distinctions that are only needed for the technical development and not observable by any actual denotations of program expressions.

Note that it is not simply our choice of the construction for the set $\max_p$ that causes this problem. For the standard function constructor, there exists no maximal set $\max_{(\to,2)}(X)$ such that $(\to,2)^{-1}(\max_{(\to,2)}(X)) = X$. To see that this set does not exist, note that our semantic domain $\mathbf{V}$ contains only functions strict in $\bot$. Thus the set of functions that map $\bot$ to an element in X is the set of all functions. To guarantee that the projection of the range is X, we would need to add at least one element v to the domain, restricting the set of functions to map v to an element of X. There are infinitely many choices for v and no minimal set exists.

Applicability of the Model

The design goal for the model of mixed constraints is to provide an intuitive meaning to mixed constraints that allows analysis designers to reason about the correctness of a constraint-based program analysis. This goal motivates the choice of a denotational model where mixed expressions denote certain sets of values, and where a mixed constraint $E_1 \subseteq_s E_2$ implies that the denotation of E_1 is a set-theoretic subset of E_2. If an analysis models all flows of values using mixed constraints, then in principle, one obtains a sound over-approximation of the possible runtime values of any expression.

Consider an expression E used to over-approximate a set of values V. We say that in this case E occurs in an L-context (Chapter 3). Note that if E contains any sub-expressions in R-contexts, then those occur in argument positions of contravariant constructors. Clearly, if E is to be an over-approximation of V, then the projection of any odd path p present in E must *under*-approximate the projection $p^{-1}(V)$. In general, expressions in L-contexts must over-approximate sets, and expressions in R-contexts must under-approximate sets. Our choice of L- and R-contexts coincides with the occurrence of such expressions in mixed constraints. If expressions E_1 and E_2 appear in a mixed constraint $E_1 \subseteq_s E_2$, then we say that E_1 appears in an L-context and E_2 appears in an R-context. The fact that E_1 over-approximates a set V_1 and E_2 under-approximates a set V_2 makes the constraint harder to satisfy than the set-theoretic inclusion $V_1 \subseteq V_2$. Thus, if the mixed constraint $E_1 \subseteq_s E_2$ is satisfied in a solution, then so is the constraint $V_1 \subseteq V_2$.

Our definitions of meet and join are such that meets are no larger than set-theoretic intersection and joins are not smaller than set-theoretic unions. If two expressions E_1 and E_2 over-approximate the set of values at a given program point, then the join also over-approximates this set. Similarly, if E_1 and E_2 under-approximate a set, then so does their meet. On the other hand, the meet of two expressions E_1 and E_2 that over-approximate a set V does not necessarily over-approximate V, and dually for the join. As a result, meets may be used in L-contexts only if they are guaranteed to coincide with set-theoretic intersection, and joins may be used in R-contexts only if they are guaranteed to coincide with set-theoretic union. This principle serves as a guide when designing an analysis.

Abstract Interpretation

The general theory of program analysis has been set on well-defined mathematical grounds in the work on *abstract interpretation* originated by Cousot and Cousot [17]. In the view of abstract interpretation, a program analysis is an evaluation of a program over an abstract semantics. The abstract semantics of a programming language conservatively approximates its concrete semantics. The conservative approximation is formalized by a Galois connection (α, γ) between the values of the concrete semantics ($\mathbf{V}$) and the values of the abstract semantics, drawn from an abstract domain $\mathbf{A}$. The function $\alpha : \mathbf{V} \to \mathbf{A}$ maps each concrete value v to the abstract value a that "best" approximates it. The function $\gamma : \mathbf{A} \to 2^{\mathbf{V}}$ maps abstract values a to the set of concrete values v that it approximates. The functions α and γ form a Galois connection, if for all $v \in \mathbf{V}$, $v \in \gamma(\alpha(v))$, and for all $a \in A$, $a = \bigsqcup\{\alpha(v) \mid v \in \gamma(a)\}$. The abstract domain $\mathbf{A}$ must form a join semi-lattice.

It is conjectured that any program analysis can be seen as an abstract interpretation. In certain cases it may be necessary to consider a concrete semantics that is much richer than the standard semantics of a programming language. Such cases arise when an analysis infers information about a program execution that is not captured by the standard concrete semantics, such as for example its memory allocation strategy [22].

Abstract interpretation is sometimes mistaken to stand only for the particular iterative fix-point approach proposed by Cousot and Cousot [17] to compute the solution of an abstract interpretation. That algorithm is only one possible implementation strategy. In general, abstract interpretations can be viewed as minimal solutions to systems of constraints between expressions denoting abstract values. These expressions involve meets and joins and other monotone operations on abstract values.

The type-collections S^s for mixed expressions can thus be seen as abstract domains for the concrete value domain $\mathbf{V}$. Mixed expressions then provide syntax to denote abstract values in these abstract domains, and mixed constraints provide lattice constraints between these abstract values. The constructors and their signatures appearing in mixed expressions synthesize the abstract domains. This synthesis has been recognized previously in the context of pure set-expressions by Heintze [38] and been described by Cousot and Cousot for set-expressions [18] and for type-expressions [16].

The Row Model

The row part of the model is to the best of our knowledge novel in its ability to combine three distinct width-subtype relations for rows with depth-subtyping. Given two row types $r_1 = \langle l : \tau_l \rangle_{l \in A}$ and $r_2 = \langle l : \tau'_l \rangle_{l \in B}$, width-subtyping refers to the relation between the domains of A and B of the rows, whereas depth subtyping refers to the subtype relationship between the types τ_l and τ'_l at certain labels l.

If the rows model record types (finite functions from labels to types), width-subtyping allows the right-side r_2 to have a smaller domain than r_1, and depth-subtyping allows each type τ_l to be less than τ'_l for $l \in B$. This notion of record subtyping is most common in the literature [86, 12, 78]. Constraints between maximal rows of sort $\mathsf{r}(s)$ provide the width-subtyping of record types. Depth-subtyping is provided to the degree that sort s provides subtyping. If $s = \mathsf{t}$, only depth-subtyping involving $\top$ and $\perp$ is present.

The constraint relation between closed rows has only depth-subtyping, but no width-subtyping, since the domains on either side of the constraint must agree. This row relation is not common in the literature. Such a relation is useful however for an analysis that extends an ML-like type system with subtyping, but where subtyping on records is to be limited to depth subtyping to match the absence of width-subtyping in ML.

Labeled variants are the dual of record types. A labeled variant is a label paired with a value $\langle l, v \rangle$. A variant type $[l : \tau_l]_{l \in A}$ then denotes the set of pairs $\langle l, v \rangle$, where $l \in A$ and $v \in \tau_l$. Variant types are treated in the literature as rows with a different subtype relation than rows used for records. The width-subtyping relation is reversed, *i.e.*, $[l : \tau_l]_{l \in A}$ is a subtype of $[l : \tau'_l]_{l \in B}$ iff $A \subseteq B$ and also $\tau_l \leq \tau'_l$ for $l \in A$. The mixed constraint relation between minimal rows matches the constraint relation used for variant types. The denotation given in terms of finite functions does not match up with the intuitive meaning of variants as unions. But there is a simple isomorphism between a minimal Row-ideal J

and the set-interpretation of variants:

$$J \equiv \{\langle l, v\rangle \mid l \in \mathbf{L} \ \wedge \ v \in J(l) - \{\perp\}\}$$

There are models for record types and subtype relations that are not based on a denotational model. For example, Bruce and Longo describe models for subtyping of record types (and other types) based on partial equivalence relations [11].

Chapter 5

Constraint Resolution

Constraint resolution is the process of deciding whether there exist any solutions to a given system of constraints, and making the solutions explicit. Here we adapt techniques developed by Aiken and Wimmers [3]. Constraints are solved by a collection of *resolution rules* that transform the constraints into a canonical form.

An application of a resolution rule can be thought of as a rewrite step $S \Rightarrow S'$, transforming the constraints S into the constraints S'. A resolution rule is sound, if every solution σ of S' is also a solution of S, and a resolution rule is complete, if every solution σ of S is also a solution of S'. If resolution rules may introduce fresh variables, the above statement can be relaxed to say that a rule is sound, if for every solution σ' of S', there exists a solution σ of S, such that σ' and σ agree on the variables occurring in S, and similarly for completeness.

All the resolution rules we consider preserve sorts, *i.e.*, given a constraint $E_1 \subseteq_s E_2$ of sort s where E_1 and E_2 are expressions of sort s, the resolution rules only generate well-formed constraints $E \subseteq_t E'$ where the sorts of E and E' agree with the constraint sort t. This fact follows directly from the well-formedness of mixed expressions.

The chapter first defines the notion of *inductive constraints*. The subsequent sections show how to transform constraints of all sorts into inductive constraints through the use of resolution rules. Section 5.5 defines the notion of an *inductive system* of constraints and shows how to transform a system of constraints into an collection of inductive systems. Inductive systems consist of inductive constraints and are closed under transitivity. Inductive systems contain no inconsistent constraints (constraints that cannot be satisfied in any solution).

5.1 Inductive Constraints

It is necessary to distinguish variables occuring in the constraints at *top-level* (e.g. $\mathcal{X} \subseteq_s \mathcal{Y}$) from variables occurring inside constructors. The following definition (an extension of Definition 2.3 to Row-expressions) makes the notion of top-level variables precise.

Definition 5.1 *The set of top-level variables* $\mathsf{TLV}(E)$ *of an expression* E *is defined by*

$$\begin{aligned}
\mathsf{TLV}(0) &= \{\} & \mathsf{TLV}(1) &= \{\}\\
\mathsf{TLV}(\mathcal{X}) &= \{\mathcal{X}\} & \mathsf{TLV}(\langle\rangle) &= \{\}\\
\mathsf{TLV}(c(\ldots)) &= \{\} & \mathsf{TLV}(\langle l : E_l\rangle \circ E) &= \mathsf{TLV}(E)\\
\mathsf{TLV}(E_1 \sqcup E_2) &= \mathsf{TLV}(E_1) \cup \mathsf{TLV}(E_2) & \mathsf{TLV}(\neg\{c_1,\ldots,c_n\}) &= \{\}\\
\mathsf{TLV}(E_1 \sqcap E_2) &= \mathsf{TLV}(E_1) \cup \mathsf{TLV}(E_2)
\end{aligned}$$

Note that the right side of a Row-composition expression appears at top-level.

In Section 5.5 we will show how to build solutions of the constraints using a double induction over the level of the series $\mathsf{S}_0^s, \mathsf{S}_1^s, \ldots$ and an arbitrary, but fixed, sequence of the variables. For this purpose, we assume a total ordering on variables characterized by an injective mapping $o : V \to \mathbb{N}$. Where convenient, we write the index $o(\mathcal{X}) = i$ as a subscript on the variable, as in $\mathcal{X}_i$. We now define when a constraint is inductive.

Definition 5.2 (Inductive Constraint) *A constraint* $E \subseteq_s \mathcal{X}_i$ *or* $\mathcal{X}_i \subseteq_s E$ *is* inductive*, iff* $\mathsf{TLV}(E) \subseteq \{\mathcal{X}_1, \ldots, \mathcal{X}_{i-1}\}$.

5.2 Set Constraint Resolution

This section describes the resolution of Set-constraints, *i.e.*, the systematic transformation of a system of constraints into inductive constraints. Before giving all the resolution rules we discuss the difficult cases in the following subsections.

5.2.1 Upward-Closure and Negation

There are two forms of Set-constraints that are difficult to decompose during resolution: $E_1 \sqcap E_2 \subseteq E_3$ and $E_1 \subseteq E_2 \sqcup E_3$. In pure set theory, the constraint $E_1 \subseteq E_2 \cup E_3$ is equivalent to $E_1 \cap \neg E_2 \subseteq E_3$ (we use $\cap$ and $\cup$ when talking about intersections and unions of pure Set-expressions). However, if Set-expressions are interpreted as ideals (downward-closed sets of values) as we do, the complement of a type is not necessarily a downward-closed set, and thus not an ideal. For example, the complement of the function type $1 \to 0$ contains every function except the least function $\lambda x.\bot$, but the least function is part of every function type. Even though general negation is not closed in the semantic domain, L-intersections (intersections in L-contexts) and R-unions (unions in R-contexts) need not be dropped entirely since there is a subset of the domain for which negation is closed, namely the upward- and downward-closed subsets. A set X is upward-closed, if for all $x \in X - \{\bot\}, y \in D$ we have $y \geq x \implies y \in X$.

Aiken and Wimmers identified simple restrictions on L-intersections and R-unions so that negations only arise on upward-closed sets during the constraint resolution. The two restrictions are

- R-unions $E_1 \cup E_2$ must be disjoint, *i.e.*, $\mu[\![E_1]\!]\sigma \cap \mu[\![E_2]\!]\sigma = \{\bot\}$ in all solutions σ.

- L-intersections must be of the form $E \cap M$, where E is an L-expression subject to the same restrictions and M is a ground expression (no variables) denoting an upward-closed set. We call such expressions *M-expressions.*

More formally, we define when expressions are *compatible* with L-contexts (*L-compatible*) or R-contexts (*R-compatible*). An expression that is both L- and R-compatible is said to be LR-compatible.

- 0, 1, and null-ary constructors are LR-compatible.
- A variable $\mathcal{X}$ is LR-compatible.
- A constructed expression $c(E_1, \ldots, E_n)$ is
  - L-compatible if covariant arguments are L-compatible and contravariant arguments are R-compatible.
  - R-compatible if covariant arguments are R-compatible and contravariant arguments are L-compatible.
- An intersection $E_1 \sqcap E_2$ is
  - L-compatible if E_1 is L-compatible and E_2 is an M-expression (or vice versa).
  - R-compatible if E_1 and E_2 are R-compatible.
- A union $E_1 \sqcup E_2$ is
  - L-compatible if E_1 and E_2 are L-compatible.
  - R-compatible if E_1 and E_2 are disjoint in all interpretations.

In well-formed constraints, all expressions appearing in L-contexts are L-compatible, expressions appearing in R-contexts are R-compatible. Since we are only interested in well-formed constraints, we can refer to L-compatible (R-compatible) expressions as L-expressions (R-expressions).

Given only well-formed pure Set-constraints, Aiken-Wimmers showed that the problematic forms can be rewritten using the following two rules:

$$E_1 \subseteq E_2 \cup E_3 \;\Leftrightarrow\; E_1 \cap \neg\overline{E_2} \subseteq E_3 \;\wedge\; E_1 \cap \neg\overline{E_3} \subseteq E_2 \tag{5.1}$$

$$E_1 \cap M \subseteq E_2 \;\Leftrightarrow\; E_1 \subseteq (E_2 \cap M) \cup \neg M \tag{5.2}$$

Expression $\overline{E}$ is the smallest M-expressions s.t. $\mu[\![\overline{E}]\!]\sigma \supseteq \mu[\![E]\!]\sigma$ for all variable assignments σ. The type expression $\neg\overline{E}$ denotes $(\mathbf{V} - \mu[\![\overline{E}]\!]\sigma) \cup \{\bot\}$, which is the complement of $\overline{E}$. (To see this, note that $\mu[\![\overline{E}]\!]\sigma \cup \mu[\![\neg\overline{E}]\!]\sigma = \mathbf{V}$ and $\mu[\![\overline{E}]\!]\sigma \cap \mu[\![\neg\overline{E}]\!]\sigma = \{\bot\}$.)

However, note that these rules don't make any progress in simplifying the constraints unless there is an additional requirement: L-intersections $E_1 \cap \neg\overline{E_2}$ and $E_1 \cap \neg\overline{E_3}$ arising in the first rule must be simplified to L-expressions containing only intersections on variables of the form $\mathcal{X} \cap M$. To do so, it must be possible to move intersections under constructors, *i.e.*,

$$c(E_1, \ldots, E_n) \cap c(M_1, \ldots, M_n) = c(E_1', \ldots, E_n')$$

for some expressions E_i'. Thus, M-expressions in L-intersections must be explicit, *i.e.*, a union of constructor expressions. Consequently, the ability to compute the two type expressions $\overline{E}$ and $\neg\overline{E}$ *explicitly* for any expression E is required.

In order to generalize this approach to mixed constraints we must address the following questions.

- Are the above rewrite rules sound and complete for $\subseteq_s$? Given that we use $\sqcap_{\mathbf{s}}$ and $\sqcup_{\mathbf{s}}$ instead of $\cap$ and $\cup$, this is not obvious.
- Given that constructor interpretations are abstract, can we compute explicit forms of upward-closure and negation?
- What are the restrictions on constructor interpretations so that we can simplify L-intersections syntactically?

We proceed as follows. We first review the approach of Aiken-Wimmers for computing upward-closure and negation expressions and show that it is unsuitable for mixed constraints. We then introduce an abbreviation expression and new resolution rules that serve our purpose and show that the resolution rules are sound and complete for $\subseteq_{\mathbf{s}}$. The problem of simplifying intersections is deferred to the following section.

The algorithms for computing $\overline{E}$ and $\neg\overline{E}$ given by Aiken and Wimmers [3] are syntax-directed and depend crucially on the interpretation of constructors as labeled tuples. The table below reproduces these rules. (For simplicity, we avoided the extra cases to keep the resulting unions of M-expressions disjoint.)

$$\begin{aligned} \overline{\mathcal{X}} &= 1 & \overline{0} &= 0 \\ \overline{c(E_1,\dots,E_n)} &= c(\overline{E_1},\dots,\overline{E_n}) & \overline{1} &= 1 \\ \overline{E_1 \cup E_2} &= \overline{E_1} \cup \overline{E_2} & \overline{E_1 \cap E_2} &= \overline{E_1} \cap \overline{E_2} \\ \overline{\neg\{c_1,\dots,c_n\}} &= \neg\{c_1,\dots,c_n\} \end{aligned}$$

$$\begin{aligned} \neg 0 = 1 \qquad & \neg 1 = 0 \\ \neg c(M_1,\dots,M_n) &= \neg\{c\} \cup \bigcup_{j=1..n} c(1,\dots,1,\neg M_j,1,\dots,1) \\ \neg(M_1 \cup M_2) &= \neg M_1 \cap \neg M_2 \\ \neg(\neg\{c_1,\dots,c_n\}) &= \bigcup_{j=1..n} c_j(1,\dots,1) \end{aligned}$$

For example, the negation of an M-expression $c(a)$ is $\neg\{c\} \cup c(\neg\{a\})$. The rules for $\neg M$ assume that constructors are strict and that

$$c(M_1,\dots,M_n) \cup \bigcup_{j=1..n} c(1,\dots,1,\neg M_j,1,\dots,1) = c(1,\dots,1)$$

Having given a semantics to mixed expressions based on ideals $\mathcal{I}(\mathbf{V})$, we face the same limitations as Aiken-Wimmers concerning L-intersections and R-unions. However, there are additional complications in the context of mixed constraints due to the fact that we have abstracted the constructor interpretations in the semantics. Without knowing the interpretation of a constructor c, an expression $c(E_1, \ldots, E_{a(c)})$ cannot automatically be transformed into its upward-closure or negation. To see this, consider a standard function constructor $\cdot \rightarrow \cdot$ with the interpretation

$$\phi_{\rightarrow}(X, Y) = \{f \mid x \in X \implies f\ x \in Y\} \cup \{\bot\}$$

Since the least function $\lambda x.\bot$ is less than any other function according to the domain ordering $\leq$ it is an element of every downward-closed set containing functions. As a result, the upward-closure of any downward-closed set containing some function contains all functions. In other words, the expression $\overline{E_1 \rightarrow E_2}$ is $0 \rightarrow 1$ (the set of all functions[1]) for any E_1 and E_2.

Even if we are given an explicit M-expression its negation can still not be computed as above because the second assumption used by the negation algorithm does not hold in general. As an example, reconsider the interpretation of a list constructor given in Example 4.12 and suppose true denotes the set $\{\mathit{true}, \bot\}$ containing the value true. Observe that the expression $\mathsf{list}(\mathsf{true})$ is upward-closed, (the semantic constructor *cons* used by the interpretation is strict, and there are no functions involved). However, $\mathsf{list}(\mathsf{true}) \cup \mathsf{list}(\neg\{\mathsf{true}\})$ is not equal to $\mathsf{list}(1)$. To see this note that the 2-element list $[\mathit{true}, \mathit{false}]$ has one element from true and one element not in true and is therefore neither part of $\mathsf{list}(\mathsf{true})$ nor $\mathsf{list}(\neg\{\mathsf{true}\})$. This problem does *not* arise because ϕ_{list} is non-strict in its argument. Consider the unary constructor pair and its interpretation given by

$$\phi_{\mathsf{pair}}(A) = \{p(t_1, t_2) \mid t_1, t_2 \in A - \{\bot\}\} \cup \{\bot\}$$

Even though ϕ_{pair} is strict, the union $\mathsf{pair}(\mathsf{true}) \cup \mathsf{pair}(\neg\{\mathsf{true}\})$ is not equal to $\mathsf{pair}(1)$. To see this, note that the element $p(\mathit{true}, \mathit{false}) \in \mathsf{pair}(1)$ is neither part of $\mathsf{pair}(\mathsf{true})$ nor of $\mathsf{pair}(\neg\{\mathsf{true}\})$.

Given that the semantics of constructors in mixed constraints is a priori not known, we cannot hope to compute the upward-closure and negation of expressions automatically. Fortunately, inspection of the two Rules 5.1 and 5.2 shows that the set of M-expressions required during resolution is fixed by the initial constraints. Therefore, we can circumvent the problem by putting the constraints in a form that makes all required M-expressions explicit in the initial system of constraints. We define an abbreviation Pat as follows.[2]

$$\mathsf{Pat}[E, M] = (E \sqcap M) \sqcup \neg M$$

where M is an M-expression. With Pat we reformulate the resolution rules for L-intersections as follows.

[1] $0 \rightarrow 1$ is the set of all functions in $\mathbf{V}$ since $\mathbf{V}$ contains only strict functions, *i.e.*, mapping $\bot$ to $\bot$. If $\mathbf{V}$ also contains non-strict functions, then the set $\{f \mid f\ \bot = \mathit{wrong}\}$ is not in $0 \rightarrow 1$. Strictness however has the undesired property of inducing a set of equivalences: $0 \rightarrow X \equiv 0 \rightarrow 1$ for all downward-closed sets X.

[2] Pat stands for *pattern*, since the expressions it abbreviates are used most frequently in constraints generated for pattern matching.

Theorem 5.3 *The following rule is sound and complete.*

$$E_1 \sqcap M \subseteq_{\mathsf{s}} E_2 \;\Leftrightarrow\; E_1 \subseteq_{\mathsf{s}} (E_2 \sqcap M) \sqcup \neg M \tag{5.3}$$
$$\equiv\; E_1 \subseteq_{\mathsf{s}} \mathsf{Pat}[E_2, M] \tag{5.4}$$

Rule 5.3 reduces to rule 5.2 if Σ^{s} contains only pure constructors. We now prove its soundness and completeness under any Σ^{s}. We need the following observation and lemma.

Observation 5.4 *The definition of mixed expressions guarantees that M-expressions do not contain subexpressions of the form $c(\dots)$, where c is a mixed constructor. This follows from the requirement that M-expressions are ground, and that other-sorted arguments to mixed constructors must be variables, 0, or 1 (Definition 3.1).*

Lemma 5.5 *Given an M-expression M of sort s, we have $p^{-1}(M) = \top_t$ for all even interface paths p^t present in M and $p^{-1}(M) = \bot_t$ for all odd interface paths present in M.*

Proof: By Observation 5.4, the only subexpressions of M whose denotations contain interface paths are of the form $E = \neg\{c_1, \dots, c_n\}$. Furthermore, since M is upward-closed, such subexpressions occur in covariant contexts. Thus for all interface paths p^t present in M, there exists a sub-expression $\neg\{c_1, \dots, c_n\}$ of M and a path q such that $p = qr$. Since the sub-expression appears in a covariant context, q must be even. Thus r has the same variance as p. We can thus restrict ourselves to the case where $M = \neg\{c_1, \dots, c_n\}$. We proceed by induction on the length of p. Let p^t be an interface path present in M. Base case: $p = (c, j)$, where c is a Set-constructor not in $\{c_1, \dots, c_n\}$. If c is covariant in j then p is even and $p^{-1}(M) = \top_t$. Otherwise p is odd and $p^{-1}(M) = \bot_t$. Induction: $p = (c, j)r$ with $r \neq \epsilon$. Then c's *jth* argument is of sort s. If c is covariant in j then $(c, j)^{-1}(M) = \top_{\mathsf{s}}$. If p is even, then r is even and by induction, $p^{-1}(M) = r^{-1}(\top_{\mathsf{s}}) = r^{-1}(\mu[\![\neg\{\}]\!]) = \top_t$. If p is odd, then r is odd and $p^{-1}(M) = \bot_t$ by the same reasoning. If c is contravariant in j, then $(c, j)^{-1}(M) = \bot_{\mathsf{s}}$, but since $r \neq \epsilon$, r cannot be present in $\bot_{\mathsf{s}}$ and thus there is no such p. □

From Lemma 5.5 we immediately obtain the following

Corollary 5.6 *For all expressions E, and variable assignments σ,*

$$\mu[\![E \sqcap M]\!]\sigma = \mu[\![E]\!]\sigma \sqcap_{\mathsf{s}} \mu[\![M]\!]\sigma = \mu[\![E]\!]\sigma \cap \mu[\![M]\!]\sigma$$

and also

$$\mu[\![E \sqcup M]\!]\sigma = \mu[\![E]\!]\sigma \sqcup_{\mathsf{s}} \mu[\![M]\!]\sigma = \mu[\![E]\!]\sigma \cup \mu[\![M]\!]\sigma$$

By the above corollary we are free to write $E \cap M$ and $E \cup M$ instead of $E \sqcap M$ and $E \sqcup M$ without causing confusion as to the operation being performed. We now prove Theorem 5.3.
Proof: of Theorem 5.3 First we deal with the inclusion on the underlying domain $\mathbf{V}^{\mathsf{s}}$. To simplify the notation, we write E to mean $\mu[\![E]\!]\sigma$ for some fixed σ. By Corollary 5.6 we have $E_1 \sqcap_{\mathsf{s}} M = E_1 \cap M \subseteq_{\mathbf{V}^{\mathsf{s}}} E_2$, and thus by set theory $E_1 \subseteq_{\mathbf{V}^{\mathsf{s}}} E_2 \cap M \cup \neg M$. Applying Corollary 5.6 again, we obtain $E_1 \subseteq_{\mathbf{V}^{\mathsf{s}}} E_2 \sqcap_{\mathsf{s}} M \sqcup_{\mathsf{s}} \neg M$.

We now prove that the interface constraints hold as well. Direction $\Longrightarrow$: Assume $E_1 \cap M \subseteq_{\mathsf{s}} E_2$. Let p^t_{s} be a non-strict interface path present in E_1. Assume p is even. There are two cases. First assume p is present in M. By assumption we have $p^{-1}(E_1 \cap M) \subseteq_t p^{-1}(E_2)$. By Lemma 4.8 we have $p^{-1}(E_1) \cap p^{-1}(M) \subseteq_t p^{-1}(E_2)$. By Lemma 5.5 we have $p^{-1}(M) = \mathbf{V}^t$. Since M and $\neg M$ are disjoint and p non-strict it follows that p is not present in $\neg M$ and $p^{-1}(\neg M) = \{\bot\}$. Thus $p^{-1}(E_1) \subseteq_t p^{-1}(E_2) \cap p^{-1}(M) \cup p^{-1}(\neg M)$. Applying Lemma 4.8 once more, we obtain $p^{-1}(E_1) \subseteq_t p^{-1}(E_2 \cap M \cup \neg M)$ and by Corollary 5.6 $p^{-1}(E_1) \subseteq_t p^{-1}(E_2 \sqcap_{\mathsf{s}} M \sqcup_{\mathsf{s}} \neg M)$. We still need to show the above for all even paths p present in E_1, but not in M. Since M and $\neg M$ are disjoint, $\mathit{present}(p, \neg M)$. Then $p^{-1}(\neg M) = \mathbf{V}^t$ and $p^{-1}(E_2 \cap M \cup \neg M) = \mathbf{V}^t$. Thus $p^{-1}(E_1) \subseteq_t p^{-1}(E_2 \cap M \cup \neg M)$ holds for all even p present in E_1.

Now assume p is odd. There are again two cases. Assume p is present in M. By assumption we have $p^{-1}(E_2) \subseteq_t p^{-1}(E_1 \cap M)$. By Lemma 5.5 we have $p^{-1}(M) = \{\bot\}$. Also, since p is non-strict and M disjoint from $\neg M$, p is not present in $\neg M$ and thus $p^{-1}(\neg M) = \mathbf{V}^t$. By Corollary 5.6 and Lemma 4.8 we have $\big(p^{-1}(E_2) \cup p^{-1}(M)\big) \cap p^{-1}(\neg M) \subseteq_t p^{-1}(E_1 \cap M)$. Applying Lemma 4.8 and canceling $p^{-1}(M)$ on the right, we obtain $p^{-1}(E_2 \cap M \cup \neg M) \subseteq_t p^{-1}(E_1)$. A final application of Corollary 5.6 yields $p^{-1}(E_2 \sqcap_{\mathsf{s}} M \sqcup_{\mathsf{s}} \neg M) \subseteq_t p^{-1}(E_1)$. We still need to show the above for all odd paths p present in E_1, but not in M. Since M and $\neg M$ are disjoint, $\mathit{present}(p, \neg M)$. Then $p^{-1}(\neg M) = \{\bot\}$ and $p^{-1}(E_2 \cap M \cup \neg M) = \{\bot\}$. Thus $p^{-1}(E_2 \sqcap_{\mathsf{s}} M \sqcup_{\mathsf{s}} \neg M) \subseteq_t p^{-1}(E_1)$ holds for all odd p present in E_1.

Now we prove $\Leftarrow$: Assume p non-strict and present in $E_1 \cap M$. Thus p present in E_1. Assume first that p is even. Then by assumption, we have $p^{-1}(E_1) \subseteq_t p^{-1}(E_2 \cap M \cup \neg M)$. Applying Corollary 5.6 and Lemma 4.8 we obtain $p^{-1}(E_1) \subseteq_t p^{-1}(E_2) \cap p^{-1}(M) \cup p^{-1}(\neg M)$. Since p is even and present in M we have $p^{-1}(M) = \mathbf{V}^t$ by Lemma 5.5. Since p is non-strict, it is thus not present in $\neg M$ and thus $p^{-1}(\neg M) = \{\bot\}$. We thus have $p^{-1}(E_1) \cap p^{-1}(M) \subseteq_t p^{-1}(E_2)$. Applying Lemma 4.8 and Corollary 5.6 once more, we obtain the desired conclusion $p^{-1}(E_1 \sqcap_{\mathsf{s}} M) \subseteq_t p^{-1}(E_2)$. Now assume that p is odd. Then by assumption we have $p^{-1}(E_2 \sqcap_{\mathsf{s}} M \sqcup_{\mathsf{s}} \neg M) \subseteq_t p^{-1}(E_1)$. Applying Corollary 5.6 and Lemma 4.8 we obtain $(p^{-1}(E_2) \cup p^{-1}(M)) \cap p^{-1}(\neg M) \subseteq_t p^{-1}(E_1)$. Since p is odd and present in M we have $p^{-1}(M) = \{\bot\}$ by Lemma 5.5. Since p is non-strict, it is thus not present in $\neg M$ and thus $p^{-1}(\neg M) = \mathbf{V}^t$. We thus have $p^{-1}(E_2) \subseteq_t p^{-1}(E_1) \cup p^{-1}(M)$. Applying Lemma 4.8 and Corollary 5.6 once more, we obtain the desired conclusion $p^{-1}(E_2) \subseteq_t p^{-1}(E_1 \sqcap_{\mathsf{s}} M)$. $\square$

Using Rule 5.4 the negation $\neg M$ no longer appears in the abbreviation and there is thus no need to compute it. Representing arbitrary disjoint R-unions is more complex.

Lemma 5.7 *If $A, B \in \mathsf{S}^{\mathsf{s}}$ such that for all even paths p, either $p^{-1}(A) = \top$ or $p^{-1}(B) = \top$ and for all odd paths p either $p^{-1}(A) = \bot$ or $p^{-1}(B) = \bot$, then*

$$A \sqcap_{\mathsf{s}} B = A \cap B$$

Similarly, if $A, B \in \mathsf{S}^{\mathsf{s}}$ such that for all even paths p, either $p^{-1}(A) = \bot$ or $p^{-1}(B) = \bot$ and for all odd paths p either $p^{-1}(A) = \top$ or $p^{-1}(B) = \top$, then

$$A \sqcup_{\mathsf{s}} B = A \cup B$$

Proof: Simply note that in the first case, for all even p^t, $p^{-1}(A \cap B) = p^{-1}(A) \cap p^{-1}(B) = p^{-1}(A) \sqcap_t p^{-1}(B)$. For odd p^t, $p^{-1}(A \cap B) = p^{-1}(A) \cup p^{-1}(B) = p^{-1}(A) \sqcup_t p^{-1}(B)$. The proof for $\sqcup_\mathsf{s}$ is analogous. □

Let E_1 and E_2 denote disjoint types. Observe that by applying Corollary 5.6 numerous times

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathsf{Pat}[E_1, \overline{E_1}] \sqcap_\mathsf{s} \mathsf{Pat}[E_2, \overline{E_2}] \sqcap_\mathsf{s} \mathsf{Pat}[0, \neg(\overline{E_1} \cup \overline{E_2})]} & \\
= & (E_1 \cap \overline{E_1} \cup \neg\overline{E_1}) \sqcap_\mathsf{s} (E_2 \cap \overline{E_2} \cup \neg\overline{E_2}) \sqcap_\mathsf{s} (\overline{E_1} \cup \overline{E_2}) & \text{def. of } \mathsf{Pat} \\
= & (E_1 \cup \neg\overline{E_1}) \sqcap_\mathsf{s} (E_2 \cup \neg\overline{E_2}) \sqcap_\mathsf{s} (\overline{E_1} \cup \overline{E_2}) & E \cap \overline{E} = E \\
= & \big((E_1 \cup \neg\overline{E_1}) \sqcap_\mathsf{s} (\overline{E_1} \cup \overline{E_2})\big) \sqcap_\mathsf{s} \big((E_2 \cup \neg\overline{E_2}) \sqcap_\mathsf{s} (\overline{E_1} \cup \overline{E_2})\big) & \text{associativity} \\
= & \big((E_1 \cup \neg\overline{E_1}) \cap (\overline{E_1} \cup \overline{E_2})\big) \sqcap_\mathsf{s} \big((E_2 \cup \neg\overline{E_2}) \cap (\overline{E_1} \cup \overline{E_2})\big) & \text{Corollary 5.6} \\
= & \big(E_1 \cap (\overline{E_1} \cup \overline{E_2}) \cup \neg\overline{E_1} \cap (\overline{E_1} \cup \overline{E_2})\big) \sqcap_\mathsf{s} & \\
 & \big(E_2 \cap (\overline{E_1} \cup \overline{E_2}) \cup \neg\overline{E_2} \cap (\overline{E_1} \cup \overline{E_2})\big) & \text{distribute} \\
= & (E_1 \cup \overline{E_2}) \sqcap_\mathsf{s} (E_2 \cup \overline{E_1}) & E \cap \overline{E} = E, E_1 \sqcap_\mathsf{s} E_2 = 0 \\
= & (E_1 \cup \overline{E_2}) \cap (E_2 \cup \overline{E_1}) & \text{Lemma 5.7} \\
= & E_1 \ \cup \ E_2 & \overline{E_1} \cap \overline{E_2} = 0
\end{array}$$

To represent an R-union $E_1 \sqcup E_2$ we therefore need the M-expressions for the upward-closures $\overline{E_1}$ and $\overline{E_2}$, as well as the M-expression for the complement $\neg(\overline{E_1} \cup \overline{E_2})$. As long as these M-expressions exist, the R-union can be represented. Below, we show the resolution of a constraint involving $E_1 \sqcup E_2$:

$$\begin{array}{rcl}
E \subseteq E_1 \sqcup E_2 & = & E \subseteq \mathsf{Pat}[E_1, \overline{E_1}] \sqcap \mathsf{Pat}[E_2, \overline{E_2}] \sqcap \mathsf{Pat}[0, \neg(\overline{E_1} \cup \overline{E_2})] \\
 & \Leftrightarrow & E \subseteq \mathsf{Pat}[E_1, \overline{E_1}] \ \wedge \ E \subseteq \mathsf{Pat}[E_2, \overline{E_2}] \ \wedge \ E \subseteq \mathsf{Pat}[0, \neg(\overline{E_1} \cup \overline{E_2})] \\
 & \Leftrightarrow & E \cap \overline{E_1} \subseteq E_1 \ \wedge \ E \cap \overline{E_2} \subseteq E_2 \ \wedge \ E \cap \neg(\overline{E_1} \cup \overline{E_2}) \subseteq 0
\end{array}$$

Only R-unions of M-expressions remain, and these are themselves M-expressions only appearing in second positions of Pat where they need never be decomposed.

To summarize, the two rules (5.2) and (5.1) for rewriting L-intersections and R-unions are replaced with the new rule (5.4) using the abbreviation Pat. The implicit negation present in Pat avoids the need to form negations during resolution.

5.2.2 L-Intersection Simplification

Constraint simplification as presented by Aiken-Wimmers requires simplifying intersections between L-expressions and M-expressions. The basic axiom that enables this simplification in their model is

$$c(X_1, \ldots, X_n) \cap c(Y_1, \ldots, Y_n) = c(X_1 \cap Y_1, \ldots, X_n \cap Y_n)$$

for any constructor c. In mixed constraints, constructors may be contravariant in certain arguments. The natural generalization of the above axiom is thus

$$c(X_1, \ldots, X_n) \cap c(Y_1, \ldots, Y_n) = c(X_1 \,\overline{\cap}\, Y_1, \ldots, X_n \,\overline{\cap}\, Y_n)$$

where

$$\overline{\cap} = \begin{cases} \cap & \text{for covariant arguments} \\ \cup & \text{for contravariant arguments} \end{cases}$$

Unfortunately, this generalization is incorrect even for standard function constructors. Consider the type expression

$$\mathsf{true} \to \mathsf{false} \cap \mathsf{false} \to \mathsf{true}$$

The boolean negation function is certainly an element of this type. However, under the axiom, the above type should be equivalent to $(\mathsf{true} \cup \mathsf{false}) \to 0$, which does not contain the boolean negation function.

We can weaken the axiom since we know that $c(Y_1, \ldots, Y_n)$ must be an M-expression $c(M_1, \ldots, M_n)$. Suppose c is contravariant in argument j. Thus M_j co-varies with the domain of a function set in the interpretation. For that set to be upward-closed, the domain must be $\{\perp\}$, and thus $c(M_1, \ldots, M_n) = c(M_1, \ldots, M_{j-1}, 0, M_{j+1}, \ldots, M_n)$. This motivates the following weakening.

Axiom 5.8 (LI-Simplification) *For any Set-constructor c, L-expression $c(E_1, \ldots, E_{a(c)})$ and any M-expression $c(M_1, \ldots, M_{a(c)})$, interpretation ϕ_c must satisfy*

$$\phi_c(E_1, \ldots, E_{a(c)}) \cap \phi_c(M_1, \ldots, M_{a(c)}) = \phi_c(E'_1, \ldots, E'_{a(c)})$$

where

$$E'_j = \begin{cases} E_j \cap M_j & c \textit{ covariant in } j \\ E_j & c \textit{ contravariant in } j \end{cases}$$

By Corollary 5.6 we immediately have

$$\phi_c(E_1, \ldots, E_{a(c)}) \sqcap_{\mathsf{s}} \phi_c(M_1, \ldots, M_{a(c)}) = \phi_c(E'_1, \ldots, E'_{a(c)})$$

where

$$E'_j = \begin{cases} E_j \sqcap_{\mathsf{s}} M_j & c \text{ covariant in } j \\ E_j & c \text{ contravariant in } j \end{cases}$$

Under this weakening, standard function interpretation satisfies the axiom, since the only upward-closed set of functions is $\phi_{\to}(\{\perp\}, \mathbf{V}^{\mathsf{s}})$ (all functions):

$$\begin{aligned} \phi_{\to}(X, Y) \sqcap_{\mathsf{s}} \phi_{\to}(\{\perp\}, \mathbf{V}^{\mathsf{s}}) &= \phi_{\to}(X, Y) \\ &= \phi_{\to}(X, Y \sqcap_{\mathsf{s}} \mathbf{V}^{\mathsf{s}}) \end{aligned}$$

The abstraction of constructor interpretations in mixed constraints leaves enough freedom to violate even this weaker axiom. As an example, consider the following interpretation

$$\phi_{\mathsf{c}}(A, B) = \{c(t) \mid t \in (A \cup B) - \{\perp\}\} \cup \{\perp\}$$

Note that $\mathsf{c}(\mathsf{false}, \mathsf{true})$ is an M-expression. Yet,

$$\begin{aligned} \mathsf{c}(\mathsf{true}, \mathsf{false}) \cap \mathsf{c}(\mathsf{false}, \mathsf{true}) &= \{c(\mathit{true}), c(\mathit{false}), \perp\} \cap \{c(\mathit{true}), c(\mathit{false}), \perp\} \\ &= \{c(\mathit{true}), c(\mathit{false}), \perp\} \\ &\neq \{\perp\} \\ &= \mathsf{c}(0, 0) \\ &= \mathsf{c}(\mathsf{true} \cap \mathsf{false}, \mathsf{false} \cap \mathsf{true}) \end{aligned}$$

Constructor interpretations that violate Axiom 5.8 cannot be used in the mixed constraint system we present, and it is up to the analysis designer to ensure that no such constructor interpretations are used. We have not found this restriction to be significant in practice—standard interpretations satisfy Axiom 5.8.

Given that M-expressions are pure Set-expressions, driving intersection below constructors only produces non-trivial intersections on Set-expressions. We need the following Lemma for the correctness of our simplification rules.

Lemma 5.9 *For any M-expression M, and any expressions E_1 and E_2,*

$$(E_1 \sqcup E_2) \sqcap M = E_1 \sqcap M \sqcup E_2 \sqcap M$$

Proof: We again write E for the denotation $\mu[\![E]\!]\sigma$ for a fixed σ. By Corollary 5.6 $(E_1 \sqcup_{\mathsf{s}} E_2) \sqcap_{\mathsf{s}} M = (E_1 \sqcup_{\mathsf{s}} E_2) \cap M$. By the definition of $\sqcup_{\mathsf{s}}$, this is the smallest set X greater than $(E_1 \cup E_2) \cap M$, such that for all non-strict interface paths p^t

$$\begin{cases} p^{-1}(X) = (p^{-1}(E_1) \sqcup_t p^{-1}(E_2)) \cap p^{-1}(M) & p \text{ even} \\ p^{-1}(X) = (p^{-1}(E_1) \sqcap_t p^{-1}(E_2)) \cup p^{-1}(M) & p \text{ odd} \end{cases}$$

There are two cases, first assume p present in M, then since $p^{-1}(M) = \top_t$ for even p and $p^{-1}(M) = \bot_t$ for odd p, we have

$$\begin{cases} p^{-1}(X) = p^{-1}(E_1) \cap p^{-1}(M) \sqcup_t p^{-1}(E_2) \cap p^{-1}(M) & p \text{ even} \\ p^{-1}(X) = (p^{-1}(E_1) \cup p^{-1}(M)) \sqcap_t (p^{-1}(E_2) \cup p^{-1}(M)) & p \text{ odd} \end{cases}$$

But this is exactly the condition on $(E_1 \cap M) \sqcup_{\mathsf{s}} (E_2 \cap M)$. Now suppose p is not present in M. Since $p^{-1}(M) = \bot_t$ for even p and $p^{-1}(M) = \top_t$ for odd p, we have

$$\begin{cases} p^{-1}(X) = \bot_t & p \text{ even} \\ p^{-1}(X) = \top_t & p \text{ odd} \end{cases}$$

which is equivalent to

$$\begin{cases} p^{-1}(X) = p^{-1}(E_1) \cap p^{-1}(M) \sqcup_t p^{-1}(E_2) \cap p^{-1}(M) & p \text{ even} \\ p^{-1}(X) = (p^{-1}(E_1) \cup p^{-1}(M)) \sqcap_t (p^{-1}(E_2) \cup p^{-1}(M)) & p \text{ odd} \end{cases}$$

Thus in either case, the smallest set X greater than $(E_1 \cup E_2) \cap M$ is equal to the smallest set greater than $E_1 \cap M \cup E_2 \cap M$ satisfying the above, which is $E_1 \cap M \sqcup_{\mathsf{s}} E_2 \cap M$. By Corollary 5.6 we conclude $(E_1 \sqcup_{\mathsf{s}} E_2) \sqcap_{\mathsf{s}} M = E_1 \sqcap_{\mathsf{s}} M \sqcup_{\mathsf{s}} E_2 \sqcap_{\mathsf{s}} M$. □

We now give the rules for simplifying L-intersections. L-expressions and M-expressions adhere to the productions

$$\begin{array}{lcl} L & ::= & 0 \mid \mathcal{X} \mid \mathcal{X} \cap M \mid L_1 \cup L_2 \mid c(E_1, \ldots, E_{a(c)}) \mid \neg\{c_1, \ldots, c_n\} \\ M & ::= & 0 \mid M_1 \cup M_2 \mid c(M_1, \ldots, M_{a(c)}) \mid \neg\{c_1, \ldots, c_n\} \end{array}$$

Note that M-expressions are a subset of L-expressions. Figure 5.1 shows the equations governing the simplification of L-intersections and M-intersections. Symmetric cases and the case covered by Axiom 5.8 are omitted. The rules are correct by Corollary 5.6 and Lemma 5.9.

$$\begin{array}{rcl}
0 \cap M & = & 0 \\
L \cap 0 & = & 0 \\
(\mathcal{X} \cap M_1) \cap M_2 & = & \mathcal{X} \cap (M_1 \cap M_2) \\
(L_1 \cup L_2) \cap M & = & L_1 \cap M \cup L_2 \cap M \\
L \cap (M_1 \cup M_2) & = & L \cap M_1 \cup L \cap M_2 \\
c(\dots) \cap d(\dots) & = & 0 \qquad c \neq d \\
c(E_1, \dots, E_{a(c)}) \cap \neg\{c_1, \dots, c_n\} & = & \begin{cases} 0 & \text{if } c \in \{c_1, \dots, c_n\} \\ c(E_1, \dots, E_{a(c)}) & \text{otherwise} \end{cases} \\
\neg\{c_1, \dots, c_n\} \cap \neg\{d_1, \dots, d_m\} & = & \neg\{c_1, \dots, c_n, d_1, \dots, d_m\}
\end{array}$$

Figure 5.1: L-intersection simplification

5.2.3 Set Resolution Rules

Figure 5.2 shows the complete set of resolution rules for Set-constraints. The rules transform a system of constraint sets into a simpler system of constraint sets. More than a single constraint set is required when constraints can be decomposed in different ways. For example, Rule 5.7, which simplifies constraints between strict constructor expressions, produces two constraint systems. The first corresponds to the case where the expression on the left is assumed to be non-zero. The second constraints set captures the solutions when the expression on the left is zero.

The rules should be read as left-to-right rewrite rules. Rules 5.5–5.12 are from Aiken-Wimmers. The correctness of Rules 5.9 and 5.10 follows from the fact that $\sqcap_s$ and $\sqcup_s$ are meet and join operations for $\subseteq_s$. Correctness of Rule 5.12 follows from Corollary 5.6. Rule 5.7 is sound but not complete in the presence of strict interface paths.

The rules below the horizontal line in Figure 5.2 are new rules. Rule 5.13 and Rule 5.14 are proven correct by Theorem 5.3. The rules involving negations are straightforward.

5.3 Term and FlowTerm Resolution

The rules in Figure 5.3 for FlowTerm resolution are a subset of the rules for the Set-sort. The rules in Figure 5.4 for Term constraints are similar, except that Rule 5.27 for constructor resolution is symmetric in the constructor arguments. The soundness and completeness argument is analogous to the one for Set-constraints.

5.4 Row Resolution

Recall from Section 4.3 that every Row-expression denotes one of three kinds of Row-ideals: a maximal, closed, or minimal Row-ideal. This section gives complete resolution rules for Row-constraints involving all three kinds of Row-expressions. However, the implementation described in the second part of this dissertation only implements a fraction of the resolution rules, namely all rules involving Row-expressions without Row-variables, and all rules

$$\Gamma, S \cup \{0 \subseteq_{\mathsf{s}} E\} \equiv \Gamma, S \tag{5.5}$$

$$\Gamma, S \cup \{c(E_1..E_n) \subseteq_{\mathsf{s}} c(E'_1..E'_n)\} \equiv \Gamma, S \cup \{E_j \subseteq_{\iota_j} E'_j \mid c : \iota_1 \cdots \iota_n \to \mathsf{s}\} \quad c \text{ non-strict} \tag{5.6}$$

$$\Gamma, S \cup \{c(E_1..E_n) \subseteq_{\mathsf{s}} c(E'_1..E'_n)\} \equiv \Gamma, S \cup \{E_j \subseteq_{\iota_j} E'_j \mid c : \iota_1 \cdots \iota_n \to \mathsf{s}\}, \; S \cup \{c(E_1..E_n) \subseteq_{\mathsf{s}} 0\} \quad c \text{ strict} \tag{5.7}$$

$$\Gamma, S \cup \{c(E_1..E_n)\} \subseteq_{\mathsf{s}} E \equiv \begin{cases} \Gamma & c : \iota_1 \cdots \iota_n \to \mathsf{s} \text{ non-strict} \\ \Gamma, S \cup \{E_1 \subseteq_{\iota_1} 0\}, \ldots, S \cup \{E_n \subseteq_{\iota_n} 0\} & c \text{ strict} \end{cases} \quad \text{if } E \text{ is } 0 \text{ or } d(..) \text{ where } c \neq d \tag{5.8}$$

$$\Gamma, S \cup \{E_1 \cup E_2 \subseteq_{\mathsf{s}} E\} \equiv \Gamma, S \cup \{E_1 \subseteq_{\mathsf{s}} E, E_2 \subseteq_{\mathsf{s}} E\} \tag{5.9}$$

$$\Gamma, S \cup \{E \subseteq_{\mathsf{s}} E_1 \cap E_2\} \equiv \Gamma, S \cup \{E \subseteq_{\mathsf{s}} E_1, E \subseteq_{\mathsf{s}} E_2\} \tag{5.10}$$

$$\Gamma, S \cup \{\mathcal{X} \subseteq_{\mathsf{s}} \mathcal{X}\} \equiv \Gamma, S \tag{5.11}$$

$$\Gamma, S \cup \{\mathcal{X} \cap M \subseteq_{\mathsf{s}} \mathcal{X}\} \equiv \Gamma, S \tag{5.12}$$

$$\Gamma, S \cup \{E_1 \subseteq_{\mathsf{s}} \mathsf{Pat}[E_2, M]\} \equiv \Gamma, S \cup \{E_1 \cap M \subseteq_{\mathsf{s}} E_2\} \tag{5.13}$$

$$\Gamma, S \cup \{\mathcal{X} \cap M \subseteq_{\mathsf{s}} E\} \equiv \Gamma, S \cup \{\mathcal{X} \subseteq_{\mathsf{s}} \mathsf{Pat}[E, M]\} \tag{5.14}$$

$$\Gamma, S \cup \{\neg\{c_1..c_n\} \subseteq_{\mathsf{s}} E\} \equiv \begin{cases} \Gamma, S & \text{if } E = \neg\{d_1..d_m\} \text{ and } \\ & \{d_1..d_m\} \subseteq \{c_1..c_n\} \\ \Gamma & \text{otherwise} \end{cases} \tag{5.15}$$

$$\Gamma, S \cup \{c(E_1..E_n)\} \subseteq_{\mathsf{s}} \neg\{c_1..c_k\} \equiv \begin{cases} \Gamma & c \text{ non-strict} \\ \Gamma, S \cup \{c(E_1..E_n) \subseteq_{\mathsf{s}} 0\} & c \text{ strict} \end{cases} \quad \text{if } c \in \{c_1..c_k\} \tag{5.16}$$

$$\Gamma, S \cup \{c(..) \subseteq_{\mathsf{s}} \neg\{d_1..d_m\}\} \equiv \Gamma, S \quad \text{if } c \notin \{d_1..d_m\} \tag{5.17}$$

$$\Gamma, S \cup \{E_1 \subseteq_{\bar{\mathsf{s}}} E_2\} \equiv \Gamma, S \cup \{E_2 \subseteq_{\mathsf{s}} E_1\} \tag{5.18}$$

Figure 5.2: Resolution rules for Set-constraints.

$$\Gamma, S \cup \{0 \subseteq_{\mathsf{ft}} E\} \equiv \Gamma, S \tag{5.19}$$

$$\Gamma, S \cup \{E \subseteq_{\mathsf{ft}} 1\} \equiv \Gamma, S \tag{5.20}$$

$$\Gamma, S \cup \{c(E_1..E_n) \subseteq_{\mathsf{ft}} c(E'_1..E'_n)\} \equiv \Gamma, S \cup \{E_j \subseteq_{\iota_j} E'_j \mid c : \iota_1 \cdots \iota_n \to \mathbf{ft}\} \tag{5.21}$$

$$\Gamma, S \cup \{c(E_1..E_n)\} \subseteq_{\mathsf{ft}} E \equiv \Gamma \quad \text{if } E \text{ is } 0 \text{ or } d(..) \text{ where } c \neq d \tag{5.22}$$

$$\Gamma, S \cup \{\mathcal{X} \subseteq_{\mathsf{ft}} \mathcal{X}\} \equiv \Gamma, S \tag{5.23}$$

$$\Gamma, S \cup \{E_1 \subseteq_{\overline{\mathsf{ft}}} E_2\} \equiv \Gamma, S \cup \{E_2 \subseteq_{\mathsf{ft}} E_1\} \tag{5.24}$$

Figure 5.3: Resolution of FlowTerm-constraints.

$$\begin{aligned}
\Gamma, S \cup \{0 \subseteq_{\mathsf{t}} E\} &\equiv \Gamma, S && (5.25)\\
\Gamma, S \cup \{E \subseteq_{\mathsf{t}} 1\} &\equiv \Gamma, S && (5.26)\\
\Gamma, S \cup \{c(E_1..E_n) \subseteq_{\mathsf{t}} c(E'_1..E'_n)\} &\equiv \Gamma, S \cup \{E_j \subseteq_{\iota_j} E'_j, E'_j \subseteq_{\iota_j} E_j \mid c : \iota_1 \cdots \iota_n \to \mathbf{t}\} && (5.27)\\
\Gamma, S \cup \{c(E_1..E_n)\} \subseteq_{\mathsf{t}} E &\equiv \Gamma \quad \text{if } E \text{ is } 0 \text{ or } d(..) \text{ where } c \neq d && (5.28)\\
\Gamma, S \cup \{\mathcal{X} \subseteq_{\mathsf{t}} \mathcal{X}\} &\equiv \Gamma, S && (5.29)\\
\Gamma, S \cup \{E_1 \subseteq_{\bar{\mathsf{t}}} E_2\} &\equiv \Gamma, S \cup \{E_2 \subseteq_{\mathsf{t}} E_1\} && (5.30)
\end{aligned}$$

Figure 5.4: Resolution of Term-constraints.

involving Row-variables that denote closed Row-ideals. This section may safely be skipped on a first reading.

Before we develop the core resolution rules for Row-constraints, we make a simplifying assumption. We assume that the kind of each Row-variable in a system of constraints S is fixed a priori.[3] Recall from Definition 4.11 that the kind $k(X)$ of a Row-ideal is either $\top_{\mathsf{r}(s)}$, $\perp_{\mathsf{r}(s)}$ or $\mathsf{abs}_{\mathsf{r}(s)}$. In other words, we associate a unique kind with each Row-variable. We make this kind explicit where needed by superscripts on variables. Thus $\mathcal{X}^{\mathsf{c}}$ is a variable which denotes only closed Row-ideals in all solutions. We simply say that $\mathcal{X}$ is closed. Minimal Row-variables are written $\mathcal{X}^{\perp}$ and maximal Row-variables are written $\mathcal{X}^{\top}$. Where convenient, we use the same superscripts on expressions. Given the explicit kinds of Row-variables, we consider only solutions σ that agree with the kind of each variable, formally

$$\begin{aligned}
k(\sigma(\mathcal{X}^{\perp})) &= \perp_{\mathsf{r}(s)}\\
k(\sigma(\mathcal{X}^{\mathsf{c}})) &= \mathsf{abs}_{\mathsf{r}(s)}\\
k(\sigma(\mathcal{X}^{\top})) &= \top_{\mathsf{r}(s)}
\end{aligned}$$

Note that the kind annotation on variables fixes the kind of every Row-expression E.

The main problem faced in the resolution of Row-constraints is that the right side of a Row-composition expression appears at top-level. In order for a constraint $\langle l : E_l\rangle_A \circ \mathcal{X} \subseteq_{\mathsf{r}(s)} \mathcal{Y}$ to be inductive, the index of $\mathcal{X}$ must be less than the index of $\mathcal{Y}$, *i.e.*, $o(\mathcal{X}) < o(\mathcal{Y})$. If that is not the case, we need a way to break up the composition. We proceed as follows. The next section introduces *domain constraints* which are used to justify a number of resolution rules. Domain constraints are furthermore used in the following section to argue the termination of the resolution of constraints when both sides are closed Rows. Finally we show how to split Row-constraints in the remaining cases.

[3]The resolution under fixed kinds is easier to formulate. In principle, complete resolution of Row-constraints without fixed kinds is possible by solving the constraints under each kind assignment.

5.4.1 Domain Constraints

Consider the following resolution rule for transforming a constraint of the form

$$\langle l : E_l \rangle_{l \in A} \circ E \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_{l \in B} \circ E'$$

where $A \cap B \neq \emptyset$, into a set of simpler constraints.

$$\langle l : E_l \rangle_{l \in A} \circ E \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_{l \in B} \circ E' \iff \bigwedge_{l \in A \cap B} E_l \subseteq_s E'_l \ \wedge \\ \langle l : E_l \rangle_{l \in A-B} \circ E \subseteq_s \langle l : E'_l \rangle_{l \in B-A} \circ E' \quad (5.31)$$

The rule is sound, *i.e.*, if the constraints on the right of $\iff$ have a solution, then that solution also satisfies the original constraint. However, for the simplification to be complete, *i.e.*, a solution to the first constraint is also a solution to the second set of constraints, we need to know how E and E' behave on the range of labels in $A \cap B$. If $E(l)$ and $E'(l)$ can be arbitrary for $l \in A \cap B$ and $l \in \mathsf{dom}(E')$, then the constraints $E(l) \subseteq_s E'(l)$ may not be satisfied, even though the original constraint is. We thus establish an invariant that specifies what is known about E, if E appears on the right side of a composition expression $\langle l : E_l \rangle_A \circ E$.

Invariant 5.10 (Row Composition) *For any Row-expression of the form $\langle l : E_l \rangle_{l \in A} \circ E$, the (minimum) domain of E does not contain any labels from A in all solutions. Formally,*

$$E(l) = \mathsf{rng}(k(E)) \qquad \forall l \in A \tag{5.32}$$

Note that the range of $k(X)$ is the unique set Y, such that there exist infinitely many $l' \in L$ for which $X(l') = Y$. Precisely,

$$\mathsf{rng}(k(X)) = \begin{cases} \{\bot\} & \text{if } X \text{ is minimal} \\ \{\bot, abs\} & \text{if } X \text{ is closed} \\ \top_s \cup \{abs\} & \text{if } X \text{ is maximal} \end{cases}$$

The invariant restricts the denotation of E such that for each label $l \in A$ and each function $f \in E$, it holds that $f\ l \in \mathsf{rng}(k(E))$. Furthermore, for each $l \in A$ and each value $v \in \mathsf{rng}(k(E))$, there exists a function $f \in E$, such that $f\ l = v$.

Invariant 5.10 guarantees that Rule 5.31 is complete, *i.e.*, direction $\Longrightarrow$ holds. However, the rule is no longer sound, since not all solutions for the right-hand side establish the above invariant for the original constraint. In order to obtain a rule that is sound and complete, we need to make the restrictions on the domain of Row-expressions explicit through a new form of constraints called *domain constraints.*

Definition 5.11 *The domain-complement of a Row-ideal I, written $\alpha(I)$ is the set of labels not in the minimum domain of I. Formally,*

$$\alpha(I) = L - \mathsf{dom}_\bot(I)$$

A *domain variable* $\alpha_{\mathcal{X}}$ stands for the set of labels that must be "absent" from the minimum domain of Row-variable $\mathcal{X}$ in all solutions. For any solution σ of the constraints, the solution $\sigma(\alpha_{\mathcal{X}})$ of the domain variable and the solution $\sigma(\mathcal{X})$ of the associated Row-variable are related by.

$$\begin{array}{ll} \sigma(\alpha_{\mathcal{X}}) = \alpha(\sigma(\mathcal{X}^{\mathsf{c}})) & \text{if } \mathcal{X} \text{ closed} \\ \sigma(\alpha_{\mathcal{X}}) \subseteq \alpha(\sigma(\mathcal{X})) & \text{otherwise} \end{array} \tag{5.33}$$

Note that if $\mathcal{X}$ is closed, then $\alpha_{\mathcal{X}}$ is exactly the domain complement of $\mathcal{X}$. Otherwise, $\alpha_{\mathcal{X}}$ is only a lower bound of the domain-complement of $\mathcal{X}$. In general domain-complement expressions have the form

$$\begin{aligned} \delta &::= N \mid \alpha_{\mathcal{X}} \mid \alpha(E) \mid \delta \cup N \mid \delta \cap N \\ N &::= \neg A \mid A \end{aligned}$$

where N is either a finite or co-finite set of labels. The domain-complement $\alpha(E)$ of a Row-expression E is a lower-bound (equal if E is closed) of the domain-complement of E in all solutions σ, *i.e.*,

$$\begin{aligned} \mu[\![\alpha(E^{\mathsf{c}})]\!]\sigma &= \alpha(\mu[\![E^{\mathsf{c}}]\!])\sigma \\ \mu[\![\alpha(E)]\!]\sigma &\subseteq \alpha(\mu[\![E]\!])\sigma \end{aligned}$$

Variable assignments σ are extended to map domain variables to subsets of $\mathbf{L}$, and μ is extended over domain-complement expressions as follows.

$$\begin{array}{rcl} \mu[\![N]\!]\sigma &=& N \\ \mu[\![\alpha_{\mathcal{X}}]\!]\sigma &=& \sigma(\alpha_{\mathcal{X}}) \\ \mu[\![\alpha(\langle l : E_l\rangle_A \circ E^{\perp})]\!]\sigma &=& (\mu[\![\alpha(E^{\perp})]\!]\sigma - \{l \mid \mu[\![E_l]\!]\sigma \neq \perp_s\}) \\ \mu[\![\alpha(\langle l : E_l\rangle_A \circ E)]\!]\sigma &=& (\mu[\![\alpha(E)]\!]\sigma - A) \quad E \neq E^{\perp} \\ \mu[\![\alpha(0)]\!]\sigma &=& \mathbf{L} \\ \mu[\![\alpha(\langle\rangle)]\!]\sigma &=& \mathbf{L} \\ \mu[\![\alpha(1)]\!]\sigma &=& \mathbf{L} \\ \mu[\![\delta \cup N]\!]\sigma &=& \mu[\![\delta]\!]\sigma \cup N \\ \mu[\![\delta \cap N]\!]\sigma &=& \mu[\![\delta]\!]\sigma \cap N \end{array}$$

Domain constraints now have the form

$$\begin{aligned} N &\subseteq_{\mathsf{d}} \delta \\ \alpha_{\mathcal{X}} &\subseteq_{\mathsf{d}} \delta \\ \alpha(E^{\mathsf{c}}) &\subseteq_{\mathsf{d}} \delta \end{aligned}$$

Domain constraints are a special form of set constraints. The notions of top-level variables and inductive constraints are easily adapted to domain constraints. Using the simplification rules given in Figure 5.7, domain constraints can be transformed into inductive constraints of the form

$$\begin{aligned} N &\subseteq_{\mathsf{d}} \alpha_{\mathcal{X}} \\ \alpha_{\mathcal{X}} \cap N &\subseteq_{\mathsf{d}} \alpha_{\mathcal{Y}} \\ \alpha_{\mathcal{X}} &\subseteq_{\mathsf{d}} \alpha_{\mathcal{Y}} \cup N \\ \alpha_{\mathcal{X}} &\subseteq_{\mathsf{d}} N \\ \alpha_{\mathcal{X}} &\subseteq_{\mathsf{d}} \alpha(\langle l : E_l \rangle_A \circ 0) \end{aligned}$$

Domain constraints enable us to restrict the minimum domain of a Row-variable to exclude certain labels. A domain constraint has the form $A \subseteq_{\mathsf{d}} \alpha_{\mathcal{X}}$, where A is a set of labels. We extend domain constraints to Row-expressions E of sort $\mathbf{r}(s)$ by writing $A \subseteq_{\mathsf{d}} \alpha(E)$. The constraint states that the minimum domain of E must not contain any labels from A. Invariant 5.10 is now made explicit using a domain constraint.

Invariant 5.10 (Row Composition Revised) *If a system of constraints S contains any sub-expression of the form $\langle l : E_l \rangle_{l \in A} \circ E$, then there are domain constraints in S implying that $A \subseteq_{\mathsf{d}} \alpha(E)$ (written $S \models A \subseteq_{\mathsf{d}} \alpha(E)$).*

Furthermore, we can now state a sound and complete version of Rule 5.31.

Resolution Rule 5.12 (Common Labels)

$$\begin{aligned} & S \cup \{\langle l : E_l \rangle_{l \in A} \circ E \subseteq_{\mathbf{r}(s)} \langle l : E'_l \rangle_{l \in B} \circ E'\} \iff \\ & \qquad S \cup \{E_l \subseteq_s E'_l \mid l \in A \cap B\} \cup \\ & \qquad \{\langle l : E_l \rangle_{l \in A-B} \circ E \subseteq_s \langle l : E'_l \rangle_{l \in B-A} \circ E'\} \\ & \qquad\qquad \textit{where } S \models A \subseteq_{\mathsf{d}} \alpha(E) \ \wedge \ B \subseteq_{\mathsf{d}} \alpha(E') \end{aligned}$$

The above rule is sound because the constraints $A \subseteq_{\mathsf{d}} \alpha(E)$ and $B \subseteq_{\mathsf{d}} \alpha(E')$ implied by S on the right establish Invariant 5.10 on the left. Similarly, the rule is complete, because Invariant 5.10 on the left implies the domain constraints explicit on the right.

5.4.2 General Row Resolution

Given the three kinds of Row-expressions in our language, there are nine possible combinations for the kinds of E_1 and E_2 in a constraint $E_1 \subseteq_{\mathbf{r}(s)} E_2$. Since maximal rows are never a subset of closed rows, and closed rows are never a subset of minimal rows, three cases are easily identified as having no solutions.

$$\begin{aligned} \Gamma, S \cup \{E^{\top} \subseteq_{\mathbf{r}(s)} E^{\mathsf{c}}\} &\equiv \Gamma \\ \Gamma, S \cup \{E^{\top} \subseteq_{\mathbf{r}(s)} E^{\perp}\} &\equiv \Gamma \\ \Gamma, S \cup \{E^{\mathsf{c}} \subseteq_{\mathbf{r}(s)} E^{\perp}\} &\equiv \Gamma \end{aligned}$$

Next we consider the base cases, where the sides are equal, the left side is 0, or the right side is 1.

$$\begin{aligned}\Gamma, S \cup \{E \subseteq_{\mathsf{r}(s)} E\} &\equiv \Gamma, S\\ \Gamma, S \cup \{0 \subseteq_{\mathsf{r}(s)} E\} &\equiv \Gamma, S\\ \Gamma, S \cup \{E \subseteq_{\mathsf{r}(s)} 1\} &\equiv \Gamma, S\end{aligned}$$

Now we consider the cases where one side does not contain a variable. The following cases assume that $A \cap B = \emptyset$, $A \cap C = \emptyset$, $B \cap C = \emptyset$, and neither A nor B is empty but C may be empty. Note that Rule 5.12 handles the cases when the intersection $A \cap B$ is non-empty.

$$\begin{aligned}\Gamma, S \cup \{\langle l : E_l\rangle_A \circ 0 \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_B \circ E\} &\equiv \Gamma, S \cup \{\langle l : E_l\rangle_A \circ 0 \subseteq_{\mathsf{r}(s)} E\} && (5.34)\\ \Gamma, S \cup \{\langle l : E_l\rangle_A \circ E \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_C \circ 0\} &\equiv \Gamma, S \cup \{E_l \subseteq_s 0 \mid l \in A\} \cup \\ &\qquad \{E \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_C \circ 0\}\\ \Gamma, S \cup \{\langle l : E_l\rangle_C \circ \langle\rangle \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_B \circ E\} &\equiv \Gamma && (5.35)\\ \Gamma, S \cup \{\langle l : E_l\rangle_A \circ E \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_C \circ \langle\rangle\} &\equiv \Gamma, S \cup \{E_l \subseteq_s 0 \mid l \in A\} \cup \\ &\qquad \{E \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_C \circ 0\}\\ \Gamma, S \cup \{\langle l : E_l\rangle_C \circ 1 \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_B \circ E\} &\equiv \Gamma\\ \Gamma, S \cup \{\langle l : E_l\rangle_A \circ E \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_C \circ 1\} &\equiv \Gamma, S \cup \{E \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_C \circ 1\} && (5.36)\end{aligned}$$

We illustrate the correctness of equivalence (5.35), the other cases are argued similarly. Note that B is non-empty and disjoint from C. Thus, in all solutions σ, there exists $l' \in B$, such that $(\mu[\![\langle l : E_l\rangle_C \circ \langle\rangle]\!]\sigma)(l') = \{\bot, abs\}$. But, the right-side of the constraint maps l' to $(\mu[\![\langle l : E'_l\rangle_{l \in B} \circ E]\!]\sigma)(l') = \mu[\![E'_{l'}]\!]\sigma \in \mathsf{S}^s$, and abs is not an element of any ideal in S^s for any sort s (abs only appears in the range of functions modeling Row-ideals).

We are left with cases where both sides contain variables. Again, $A \cap B = \emptyset$ and at least one of A or B is non-empty.

$$\langle l : E_l\rangle_A \circ \mathcal{X} \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_B \circ \mathcal{Y}$$

There are three cases. Either $\mathcal{X} = \mathcal{Y}$, or A is non-empty and $o(\mathcal{X}) > o(\mathcal{Y})$, or B is non-empty and $o(\mathcal{Y}) > o(\mathcal{X})$. Otherwise the constraint is inductive. The first case ($\mathcal{X} = \mathcal{Y}$) splits into three cases again, depending on the kind of $\mathcal{X}$. If $\mathcal{X}$ is maximal or closed, the constraint is inconsistent. To see why, note that by Invariant 5.10, B must be absent from the domain of $\mathcal{X}$. Thus the left side contains functions returning abs for some $l \in B$, whereas the right side contains no such functions. If $\mathcal{X}$ is minimal, $\mathcal{X}$ maps $l \in A \cup B$ to $\bot$ by Invariant 5.10. Thus the constraint is satisfied as long as $E_l \subseteq_s 0$ for $l \in A$.

$$\begin{aligned}\Gamma, S \cup \langle l : E_l\rangle_A \circ \mathcal{X}^{\mathsf{c}} \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_B \circ \mathcal{X}^{\mathsf{c}} &\equiv \Gamma\\ \Gamma, S \cup \langle l : E_l\rangle_A \circ \mathcal{X}^{\top} \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_B \circ \mathcal{X}^{\top} &\equiv \Gamma && (5.37)\\ \Gamma, S \cup \langle l : E_l\rangle_A \circ \mathcal{X}^{\bot} \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_B \circ \mathcal{X}^{\bot} &\equiv \Gamma, S \cup \{E_l \subseteq_s 0 \mid l \in A\} && (5.38)\end{aligned}$$

The remaining cases are harder and dealt with in the following subsections. First we consider the case where both sides are closed.

5.4.3 Splitting of Closed Rows

Consider the constraint between closed rows $\langle l : E_l \rangle_A \circ \mathcal{X} \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ are distinct. Since both sides are closed, the constraint is only satisfied if the domains of both sides are equal. Suppose B is non-empty and $o(\mathcal{Y}) > o(\mathcal{X})$ (the case where A is non-empty and $o(\mathcal{X}) > o(\mathcal{Y})$ is symmetric). In all solutions, $\mathcal{X}$ must be of the form $\langle l : \mathcal{X}_l \rangle_B \circ \mathcal{X}'$ where $\mathcal{X}_l \subseteq_s E'_l$ for $l \in B$ and $\langle l : E_l \rangle_A \circ \mathcal{X}' \subseteq_{\mathsf{r}(s)} \mathcal{Y}$. This can be captured by the rule

Resolution Rule 5.13 (Closed Row Split)

$$\begin{aligned} S \cup \{\langle l : E_l \rangle_A \circ \mathcal{X} \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ \mathcal{Y}\} \equiv S \cup \{B \subseteq_{\mathsf{d}} \alpha_{\mathcal{X}'}\} \cup \\ \{\langle l : \mathcal{X}_l \rangle_B \circ \mathcal{X}' \subseteq_{\mathsf{r}(s)} \mathcal{X}, \mathcal{X} \subseteq_{\mathsf{r}(s)} \langle l : \mathcal{X}_l \rangle_B \circ \mathcal{X}'\} \cup \\ \{\mathcal{X}_l \subseteq_s E'_l \mid l \in B\} \cup \{\langle l : E_l \rangle_A \circ \mathcal{X}' \subseteq_{\mathsf{r}(s)} \mathcal{Y}\} \end{aligned}$$

where $\mathcal{X}'$ is a fresh closed Row-variable of sort $\mathsf{r}(s)$ and $\mathcal{X}_l$ with $l \in B$ are fresh variables of sort s. If we choose the index of $\mathcal{X}'$ less than the index of $\mathcal{X}$, the resulting constraints on $\mathcal{X}$ and $\mathcal{Y}$ are inductive, since $o(\mathcal{X}') < o(\mathcal{X}) < o(\mathcal{Y})$.

Since this rule introduces fresh variables, we need to argue that the rule is only applied finitely many times given a finite initial system of constraints. The argument assumes that no other sorts introduce fresh variables. Call the variable $\mathcal{X}'$ a child of $\mathcal{X}$. Note the domain constraint $B \subseteq_{\mathsf{d}} \alpha_{\mathcal{X}'}$ associated with the fresh Row-variable $\mathcal{X}'$. The domain of $\mathcal{X}'$ is strictly smaller than the domain of $\mathcal{X}$, since everything absent in $\mathcal{X}$ must be absent in $\mathcal{X}'$, and additionally B is absent from $\mathcal{X}'$, whereas B is contained in the domain of $\mathcal{X}$. Since there are only finitely many distinct labels in the constraints, $\mathcal{X}'$ the number of generations generated from $\mathcal{X}'$ is finite. Furthermore, since the index of $\mathcal{X}'$ and all of its descendants have smaller index than $\mathcal{X}$, they cannot cause the same rule to trigger another split of $\mathcal{X}$. Thus Rule 5.13 can be applied only finitely many times.

5.4.4 Splitting of Minimal and Maximal Rows

The splitting-approach of closed Row-expressions does not work for minimal and maximal rows because the domains of the two sides need not be equal in this case. Instead we transform Row-constraints involving a minimal or maximal row using the following two basic transformations.

Resolution Rule 5.14 (Split-Left)

$$\begin{aligned} \langle l : E_l \rangle_A \circ \mathcal{X} \subseteq_{\mathsf{r}(s)} E \iff \langle l : E_l \rangle_A \circ 0 \subseteq_{\mathsf{r}(s)} E \ \wedge \\ \langle l : 0 \rangle_{l \in A} \circ \mathcal{X} \subseteq_{\mathsf{r}(s)} E \end{aligned}$$

The correctness of this rule follows from the fact that

$$\langle l : E_l \rangle_A \circ \mathcal{X} = \langle l : E_l \rangle_A \circ 0 \ \sqcup_{\mathsf{r}(s)} \ \langle l : 0 \rangle_A \circ \mathcal{X}$$

Note that we depend crucially on the definition of $\sqcup_{\mathsf{r}(s)}$. If $\sqcup_{\mathsf{r}(s)}$ were set-theoretic union, the equivalence would not hold. Similarly, the following rule splits rows on the right side of the inclusion.

Resolution Rule 5.15 (Split-Right)

$$\begin{aligned} E \subseteq_{\mathsf{r}(s)} \langle l : E_l\rangle_A \circ \mathcal{X} \iff & E \subseteq_{\mathsf{r}(s)} \langle l : E_l\rangle_A \circ 1 \;\wedge \\ & E \subseteq_{\mathsf{r}(s)} \langle l : 1\rangle_{l\in A} \circ \mathcal{X} \end{aligned}$$

The correctness of this rule follows from the fact that

$$\langle l : E_l\rangle_A \circ \mathcal{X} = \langle l : E_l\rangle_A \circ 1 \sqcap_{\mathsf{r}(s)} \langle l : 1\rangle_A \circ \mathcal{X}$$

These rules are correct for all Row-kinds. Note that we only apply these rules if E contains a top-level variable $\mathcal{Y}$ such that $o(\mathcal{X}) > o(\mathcal{Y})$. Otherwise, one of the rules in Section 5.4.2 can be applied.

Rules 5.14 and 5.15 are only the first step in obtaining inductive constraints. Consider the constraint $\langle l : 0\rangle_{l\in A}\circ\mathcal{X} \subseteq_{\mathsf{r}(s)} E$ obtained by applying Rule 5.14. This constraint is still not inductive. Intuitively, we need an expression E' that is equivalent to E, except that the set A is masked out from the domain, and such that $\langle l : 0\rangle_A\circ\mathcal{X} \subseteq_{\mathsf{r}(s)} E$ is equivalent to $\mathcal{X} \subseteq_{\mathsf{r}(s)} E'$. The constraint $\mathcal{X} \subseteq_{\mathsf{r}(s)} E'$ is then inductive.

In general, we need to extend our Row-expression language with two forms of expressions.

Definition 5.16 *Let A be a finite subset of the set of labels $\mathbf{L}$. If E is a well-formed L-compatible minimal (maximal) $\mathsf{r}(s)$-expression, then $E \wedge_A 0$ is also a well-formed L-compatible minimal (maximal) $\mathsf{r}(s)$-expression, with the meaning*

$$E \wedge_A 0 = \{f \mid l \in A \implies f\, l \in \bot_s \wedge l \notin A \implies f\, l \in (\mu[\![E]\!]\sigma)(l)\} \tag{5.39}$$

Furthermore, if E is a well-formed minimal (maximal) $\mathsf{r}(s)$-expression, then $E \vee_A 1$ is also a well-formed minimal (maximal) $\mathsf{r}(s)$-expression. The meaning of $E \vee_A 1$ is

$$\mu[\![E \vee_A 1]\!]\sigma = \begin{cases} \{f \mid l \in A \implies f\, l \in \top_s \wedge l \notin A \implies f\, l \in (\mu[\![E]\!]\sigma)(l)\} \\ \hfill \textit{if } E \textit{ is minimal} \\ \{f \mid l \in A \implies f\, l \in \top_s \cup \{abs\} \wedge l \notin A \implies f\, l \in (\mu[\![E]\!]\sigma)(l)\} \\ \hfill \textit{if } E \textit{ is maximal} \end{cases} \tag{5.40}$$

We call expressions of the form $E \wedge_A 0$ L-masks and expressions of the form $E \vee_A 1$ R-masks. Intuitively, $E \wedge_A 0$ is equivalent to E, except that the rows in $E \wedge_A 0$ map all labels in A to $\bot$. Similarly, $E \vee_A 1$ is equivalent to E, except that for $l \in A$, no assumptions on the rows of $E \vee_A 1$ are made. This last requirement also motivates why we do not define $E \vee_A 1$ when E is closed. The denotation of $E \vee_A 1$ when E is closed must be the same as the one when E is maximal. However, that denotation is not a closed Row-ideal, since $(E \vee_A 1)(l) = \top_s \cup \{abs\}$ for $l \in A$. For minimal and maximal Rows the denotations of the new expression are already present in our type-collections.

The introduction of L-masks and R-masks is governed by the following two rules.

Resolution Rule 5.17 *If E' is not closed, then the following equivalence holds.*

$$\langle l : 0\rangle_A \circ E \subseteq_{\mathsf{r}(s)} E' \;\wedge\; A \subseteq_{\mathsf{d}} \alpha(E) \iff E \subseteq_{\mathsf{r}(s)} E' \vee_A 1 \;\wedge\; A \subseteq_{\mathsf{d}} \alpha(E)$$

Proof: Note that $A \subseteq_{\mathsf{d}} \alpha(E)$ implies by (5.16) that $\mu[\![E]\!]\sigma = \mu[\![E \vee_A 1]\!]\sigma$ for all σ. To simplify the notation, we drop the $\mu[\![\cdot]\!]\sigma$ notation in the proof. Furthermore to simplify the proof, we assume that $I \vee_A 1$ is defined for closed row-ideals as for maximal row-ideals (5.16). Note that for any row-ideal I, $I \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} I \vee_A 1$, and for any row-ideal I, $\langle l : 0\rangle_A \circ E \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} E$. Also note that $\cdot \vee_A 1$ as a unary operation on ideals (5.16) is monotone it its argument, and so is row-composition in its right argument. Direction $\Longrightarrow$: We first show that $E \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} E' \vee_A 1$, and then show that the interface constraints are satisfied. We have that $E \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} E \vee_A 1 = (\langle l : 0\rangle_A \circ E) \vee_A 1 \subseteq_{\mathsf{r}(s)} E' \vee_A 1$, where the last step follows from monotonicity of $\cdot \vee_A 1$. For the interface constraints, consider $l \in \mathsf{dom}(E') - A$. Clearly $l \in \mathsf{dom}(E')$, and thus $E(l) = ((\langle l : 0\rangle_A \circ E)(l) \subseteq_s E'(l) = E' \vee_A 1(l)$. Direction $\Longleftarrow$: Note that $\langle l : 0\rangle_A \circ (E' \vee_A 1) \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} E'$. By monotonicity of $\circ$, we have $\langle l : 0\rangle_A \circ E \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} \langle l : 0\rangle_A \circ (E' \vee_A 1)$ and thus $\langle l : 0\rangle_A \circ E \subseteq_{\mathbf{V}^{\mathsf{r}(s)}} E'$. For the interface constraints, consider $l \in A$. Clearly $((\langle l : 0\rangle_A \circ E)(l) \subseteq_s \bot_s \subseteq_s E'(l)$. Otherwise $l \in \mathsf{dom}(E') - A$. We have $((\langle l : 0\rangle_A \circ E)(l) = E(l) \subseteq_s (E' \vee_A 1)(l) \subseteq_s E'(l)$. □

Resolution Rule 5.18 *If E is not closed, then the following equivalence holds.*

$$E \subseteq_{\mathsf{r}(s)} \langle l : 1\rangle_A \circ E' \ \wedge\ A \subseteq_{\mathsf{d}} \alpha(E') \iff E \wedge_A 0 \subseteq_{\mathsf{r}(s)} E' \ \wedge\ A \subseteq_{\mathsf{d}} \alpha(E')$$

The proof is similar to the previous resolution rule.

Finally, we need one more rule that allows us to move the mask from the left of an inclusion to the right and vice-versa.

Resolution Rule 5.19 *If E and E' are not closed, then the following equivalence holds.*

$$E \wedge_A 0 \subseteq_{\mathsf{r}(s)} E' \iff E \subseteq_{\mathsf{r}(s)} E' \vee_A 1$$

Proof: Direction $\Longrightarrow$: We have $\forall f. f \in E \wedge_A 0 \implies f \in E'$ we proof $\forall f. f \in E \implies f \in E' \vee_A 1$. Suppose $f \in E$. If $f \in E \wedge_A 0$ we have $f \in E'$ and thus $f \in E' \vee_A 1$. Otherwise, there exits $f' \in E \wedge_A 0$, such that for all $l \notin A$ we have $f'(l) = f(l)$. Since $f' \in E'$, we have $f \in E' \vee_A 1$. For the interface constraints, first assume E' is maximal. Then $\mathsf{dom}(E' \vee_A 1) = \mathsf{dom}(E') - A$, and for all $l \in \mathsf{dom}(E') - A$, we have $E(l) = (E \wedge_A 0)(l) \subseteq_s E'(l) = (E \vee_A 1)(l)$. If E' is minimal, then the above argument also holds. In addition, for all $l \in A$, we have $E(l) \subseteq_s (E' \vee_A 1)(l) = \top_s$. Direction $\Leftarrow$: We have $\forall f. f \in E \implies f \in E' \vee_A 1$ we proof $\forall f. f \in E \wedge_A 0 \implies f \in E'$. Suppose $f \in E \wedge_A 0$. Then $f \in E$ and thus $f \in E' \vee_A 1$. By Definition 4.9 and (5.16), we have that $f \in E'$, since for all $l \in A$, $f(l) = \bot$, and $\bot \in E'(l)$ for all labels l. For the interface constraints, suppose $l \in \mathsf{dom}(E')$. If $l \in A$, then $(E \wedge_A 0)(l) = \bot_s \subseteq_s E'(l)$. Otherwise, $(E \wedge_A 0)(l) = E(l) \subseteq_s (E' \vee_A 1)(l) = E'(l)$. □

Before we show how to apply the above rules to transform the remaining constraints into inductive form, we state a number of equivalences involving $\wedge_A 0$ and $\vee_A 1$.

$$\begin{aligned} 1 \wedge_A 0 &= \langle l : 0\rangle_A \circ 1 \\ 0 \wedge_A 0 &= 0 \\ (E \wedge_B 0) \wedge_A 0 &= E \wedge_{A \cup B} 0 \\ (\langle l : E_l\rangle_{l \in B} \circ E) \wedge_A 0 &= \langle l : E_l\rangle_{l \in B - A} \circ (E \wedge_A 0) \end{aligned}$$

$$\begin{aligned}
1 \vee_A 1 &= 1 \\
0 \vee_A 1 &= \langle l : 1 \rangle_A \circ 0 \\
(E \vee_B 1) \vee_A 1 &= E \vee_{A \cup B} 1 \\
(\langle l : E_l \rangle_{l \in B} \circ E) \vee_A 1 &= \langle l : E_l \rangle_{l \in B-A} \circ (E \vee_A 1)
\end{aligned}$$

These equivalences suggest that Row-expressions involving a Row-variable $\mathcal{X}$ can always be normalized to the form

$$\langle l : E_l \rangle_A \circ [\mathcal{X}]_B$$

where $[\mathcal{X}]_B$ is $\mathcal{X}$ if $B = \emptyset$, or

$$[\mathcal{X}]_B = \begin{cases} \mathcal{X} \wedge_B 0 & \text{if } [\mathcal{X}] \text{ appears to the left of } \subseteq_{\mathsf{r}(s)} \\ \mathcal{X} \vee_B 1 & \text{if } [\mathcal{X}] \text{ appears to the right of } \subseteq_{\mathsf{r}(s)} \end{cases}$$

Where convenient, we drop the label set subscript. The remaining non-inductive constraints are then of the form

$$\langle l : E_l \rangle_A \circ [\mathcal{X}] \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ [\mathcal{Y}]$$

where $A \cap B = \emptyset$, at least one of A and B is non-empty, and not both $\mathcal{X}$ and $\mathcal{Y}$ are closed. By combining Rule 5.14 with Rule 5.17, we obtain

Resolution Rule 5.20 *If $A \cap B = \emptyset$, A is non-empty, $o(\mathcal{X}) > o(\mathcal{Y})$, and $\mathcal{Y}$ is not closed, then the following equivalence holds.*

$$\begin{aligned}
&S \cup \{\langle l : E_l \rangle_{l \in A} \circ [\mathcal{X}] \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_{l \in B} \circ [\mathcal{Y}]\} \iff \\
&\qquad S \cup \{\langle l : E_l \rangle_A \circ 0 \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ [\mathcal{Y}]\} \cup \\
&\qquad \{[\mathcal{X}] \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_{l \in B-A} \circ ([\mathcal{Y}] \vee_A 1)\} \\
&\qquad\qquad\qquad \textit{where } S \models A \subseteq_{\mathsf{d}} \alpha([\mathcal{X}])
\end{aligned}$$

The resulting constraint involving $\mathcal{X}$ and $\mathcal{Y}$ is either inductive ($[\mathcal{X}] = \mathcal{X}$), or inductive after applying Rule 5.19. The constraint involving only $\mathcal{Y}$ can be transformed into an inductive constraint using (5.34) and Rule 5.19.

The analogous rule in the case where $o(\mathcal{Y}) > o(\mathcal{X})$ is obtained by combining Rule 5.15 and Rule 5.18.

Resolution Rule 5.21 *If $A \cap B = \emptyset$, B is non-empty, $o(\mathcal{Y}) > o(\mathcal{X})$, and $\mathcal{X}$ is not closed, then the following equivalence holds.*

$$\begin{aligned}
&S \cup \{\langle l : E_l \rangle_{l \in A} \circ [\mathcal{X}] \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_{l \in B} \circ [\mathcal{Y}]\} \iff \\
&\qquad S \cup \{\langle l : E_l \rangle_A \circ [\mathcal{X}] \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ 1\} \cup \\
&\qquad \{\langle l : E_l \rangle_{A-B} \circ ([\mathcal{X}] \wedge_B 0) \subseteq_{\mathsf{r}(s)} [\mathcal{Y}]\} \\
&\qquad\qquad\qquad \textit{where } S \models B \subseteq_{\mathsf{d}} \alpha([\mathcal{Y}])
\end{aligned}$$

The resulting constraint involving $\mathcal{X}$ and $\mathcal{Y}$ is inductive ($[\mathcal{Y}] = \mathcal{Y}$), or inductive after applying Rule 5.19. The constraint involving only $\mathcal{X}$ can be transformed into and inductive constraint using (5.36) and Rule 5.19.

Since we needed to restrict the above rules to the cases where the lower indexed variable is not closed, we introduce the following invariant.

Invariant 5.22 *The index of any closed* Row*-variable* $\mathcal{X}^{\mathsf{c}}$ *is larger than the index of any minimal or maximal* Row*-variable* $\mathcal{Y}$.

Since it is always possible to choose a variable order satisfying the invariant, we do not restrict the family of constraints that can be solved.

Due to the introduction of masks, there are still some constraints that we haven't dealt with. The equivalences (5.37) and (5.38) need to be extended to the cases

$$S \cup \{\langle l : E_l \rangle_A \circ [\mathcal{X}^{\top}]_B \subseteq_{\mathbf{r}(s)} \langle l : E_l' \rangle_C \circ [\mathcal{X}^{\top}]_D\}$$
$$S \cup \{\langle l : E_l \rangle_A \circ [\mathcal{X}^{\perp}]_B \subseteq_{\mathbf{r}(s)} \langle l : E_l' \rangle_C \circ [\mathcal{X}^{\perp}]_D\}$$

where $A \cap B$. Furthermore, since we never form new composition expressions involving masks, we know by Invariant 5.10 that S implies the domain constraints $A \subseteq_{\mathbf{d}} \alpha_{\mathcal{X}}$ and $C \subseteq_{\mathbf{d}} \alpha_{\mathcal{X}}$. Applying Rule 5.14 followed by (5.34), Rule 5.17, and Rule 5.19 we obtain the equivalent constraints

$$S \cup \{\langle l : E_l \rangle_{A-D} \circ 0 \subseteq_{\mathbf{r}(s)} \mathcal{X}, \mathcal{X} \subseteq_{\mathbf{r}(s)} \langle l : E_l' \rangle_{C-B} \circ (\mathcal{X} \vee_{A \cup B \cup D} 1)\}$$
$$\text{where } S \models A \cup C \subseteq_{\mathbf{d}} \alpha_{\mathcal{X}}$$

If $\mathcal{X}$ is minimal, then the domain constraints imply that $\mathcal{X}(l) = \{\perp\}$ in all solutions for all $l \in A \cup C$. Thus the second constraint is always satisfied and the first is equivalent to $\{E_l \subseteq_s 0 \mid l \in A - D\}$.

If $\mathcal{X}$ is maximal, then the domain constraints imply that $\mathcal{X}(l) = \top_s \cup \{abs\}$ in all solutions for all $l \in A \cup C$. Thus the first constraint is always satisfied and the second constraint is satisfied if and only if $C \subseteq B$. The complete set of rules is summarized in Figures 5.5–5.7.

5.5 Inductive Systems

The resolution rules presented so far allow us to transform an arbitrary system of constraints into an equivalent collection of constraint systems consisting only of inductive constraints. This section defines *inductive systems*, which are systems of inductive constraints closed under transitivity. We show how to transform an original system of constraints into an equivalent collection of inductive systems. If this collection is empty, then the original constraints have no solutions. Otherwise, the original constraints satisfy the constraints up to any finite level of the type collections $\mathsf{S}_0^s, \mathsf{S}_1^s, \mathsf{S}_2^s, \ldots$ (Section 4.3). We conjecture that inductive systems always have solutions.

To simplify the presentation, we initially leave out Row-sorts, *i.e.*, we assume that the constraints contain no Row-variables. We say that a constraint $E \subseteq_s \mathcal{X}$ is *L-inductive*,

$$\Gamma, S \cup \{E^{\top} \subseteq_{\mathsf{r}(s)} E^{\mathsf{c}}\} \equiv \Gamma \tag{5.41}$$

$$\Gamma, S \cup \{E^{\top} \subseteq_{\mathsf{r}(s)} E^{\perp}\} \equiv \Gamma \tag{5.42}$$

$$\Gamma, S \cup \{E^{\mathsf{c}} \subseteq_{\mathsf{r}(s)} E^{\perp}\} \equiv \Gamma \tag{5.43}$$

$$\Gamma, S \cup \{E \subseteq_{\mathsf{r}(s)} E\} \equiv \Gamma, S \tag{5.44}$$

$$\Gamma, S \cup \{0 \subseteq_{\mathsf{r}(s)} E\} \equiv \Gamma, S \tag{5.45}$$

$$\Gamma, S \cup \{E \subseteq_{\mathsf{r}(s)} 1\} \equiv \Gamma, S \tag{5.46}$$

$$\Gamma, S \cup \{\mathcal{X} \subseteq_{\mathsf{r}(s)} \mathcal{X}\} \equiv \Gamma, S \tag{5.47}$$

$$\Gamma, S \cup \{E_1 \subseteq_{\overline{\mathsf{r}(s)}} E_2\} \equiv \Gamma, S \cup \{E_2 \subseteq_{\mathsf{r}(s)} E_1\} \tag{5.48}$$

The rules below assume that $A \cap B = \emptyset, A \cap C = \emptyset, B \cap C = \emptyset$, A and B are non-empty, and C may be empty.

$$\Gamma, S \cup \{\langle l : E_l \rangle_A \circ 0 \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ E\} \equiv \Gamma, S \cup \{\langle l : E_l \rangle_A \circ 0 \subseteq_{\mathsf{r}(s)} E\} \tag{5.49}$$

$$\begin{aligned}\Gamma, S \cup \{\langle l : E_l \rangle_A \circ E \subseteq_{\mathsf{r}(s)} n\langle l : E'_l \rangle_C \circ 0\} \equiv{}& \Gamma, S \cup \{E_l \subseteq_s 0 \mid l \in A\} \cup \\ & \{E \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_C \circ 0\}\end{aligned} \tag{5.50}$$

$$\Gamma, S \cup \{\langle l : E_l \rangle_C \circ \langle\rangle \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ E\} \equiv \Gamma \tag{5.51}$$

$$\begin{aligned}\Gamma, S \cup \{\langle l : E_l \rangle_A \circ E \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_C \circ \langle\rangle\} \equiv{}& \Gamma S \cup \{E_l \subseteq_s 0 \mid l \in A\} \cup \\ & \{E \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_C \circ 0\}\end{aligned} \tag{5.52}$$

$$\Gamma, S \cup \{\langle l : E_l \rangle_C \circ 1 \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ E\} \equiv \Gamma \tag{5.53}$$

$$\Gamma, S \cup \{\langle l : E_l \rangle_A \circ E \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_C \circ 1\} \equiv \Gamma, S \cup \{E \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_C \circ 1\} \tag{5.54}$$

$$\Gamma, S \cup \langle l : E_l \rangle_A \circ \mathcal{X}^{\mathsf{c}} \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ \mathcal{X}^{\mathsf{c}} \equiv \Gamma \tag{5.55}$$

$$\Gamma, S \cup \{\langle l : E_l \rangle_A \circ [\mathcal{X}^{\top}]_B \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_C \circ [\mathcal{X}^{\top}]_D\} \equiv \Gamma, S \qquad C \subseteq B \tag{5.56}$$

$$\Gamma, S \cup \{\langle l : E_l \rangle_A \circ [\mathcal{X}^{\top}]_B \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_C \circ [\mathcal{X}^{\top}]_D\} \equiv \Gamma \qquad C \nsubseteq B \tag{5.57}$$

$$\Gamma, S \cup \{\langle l : E_l \rangle_A \circ [\mathcal{X}^{\perp}]_C \subseteq_{\mathsf{r}(s)} \langle l : E'_l \rangle_B \circ [\mathcal{X}^{\perp}]_D\} \equiv \Gamma, S \cup \{E_l \subseteq_s 0 \mid l \in A - D\} \tag{5.58}$$

Figure 5.5: Resolution of Row-constraints (simple cases).

The rule below assumes $A \cap B \neq \emptyset$ and that $S \models A \subseteq_{\mathsf{d}} \alpha(E) \ \wedge\ B \subseteq_{\mathsf{d}} \alpha(E')$

$$\begin{aligned} \Gamma, S \cup \{\langle l : E_l\rangle_{l\in A} \circ E \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_{l\in B} \circ E'\} \equiv{}& \Gamma, S \cup \{E_l \subseteq_s E'_l \mid l \in A \cap B\} \cup \\ & \{\langle l : E_l\rangle_{l\in A-B} \circ E \subseteq_s \langle l : E'_l\rangle_{l\in B-A} \circ E'\} \end{aligned} \tag{5.59}$$

The two rules below assume $o(\mathcal{X}^{\mathsf{c}}) > o(\mathcal{Y}^{\mathsf{c}})$, $A \cap B = \emptyset$, A non-empty, and that $S \models A \subseteq_{\mathsf{d}} \alpha_{\mathcal{X}} \ \wedge\ B \subseteq_{\mathsf{d}} \alpha_{\mathcal{Y}}$. Variable $\mathcal{X}'$ is a fresh closed variable, $\mathcal{X}_l$ for $l \in B$ are fresh variables of sort s.

$$\begin{aligned} \Gamma, S \cup \{\langle l : E_l\rangle_A \circ \mathcal{X}^{\mathsf{c}} \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_B \circ \mathcal{Y}^{\mathsf{c}}\} \equiv{}& \Gamma, S \cup \{B \subseteq_{\mathsf{d}} \alpha_{\mathcal{X}'}\} \cup \\ & \{\langle l : \mathcal{X}_l\rangle_B \circ \mathcal{X}' \subseteq_{\mathsf{r}(s)} \mathcal{X}^{\mathsf{c}}\} \cup \\ & \{\mathcal{X}^{\mathsf{c}} \subseteq_{\mathsf{r}(s)} \langle l : \mathcal{X}_l\rangle_B \circ \mathcal{X}'\} \cup \\ & \{\mathcal{X}_l \subseteq_s E'_l \mid l \in B\} \cup \\ & \{\langle l : E_l\rangle_A \circ \mathcal{X}' \subseteq_{\mathsf{r}(s)} \mathcal{Y}^{\mathsf{c}}\} \end{aligned} \tag{5.60}$$

$$\begin{aligned} \Gamma, S \cup \{\langle l : E_l\rangle_B \circ \mathcal{Y}^{\mathsf{c}} \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_A \circ \mathcal{X}^{\mathsf{c}}\} \equiv{}& \Gamma, S \cup \{B \subseteq_{\mathsf{d}} \alpha_{\mathcal{X}'}\} \cup \\ & \{\langle l : \mathcal{X}_l\rangle_B \circ \mathcal{X}' \subseteq_{\mathsf{r}(s)} \mathcal{X}^{\mathsf{c}}\} \cup \\ & \{\mathcal{X}^{\mathsf{c}} \subseteq_{\mathsf{r}(s)} \langle l : \mathcal{X}_l\rangle_B \circ \mathcal{X}'\} \cup \\ & \{E_l \subseteq_s \mathcal{X}_l \mid l \in B\} \cup \\ & \{\mathcal{Y}^{\mathsf{c}} \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_A \circ \mathcal{X}'\} \end{aligned} \tag{5.61}$$

The following rules assume $o(\mathcal{X}) > o(\mathcal{Y})$, $\mathcal{Y}$ is not closed, $A \cap B = \emptyset$, and A non-empty. $S \models A \subseteq_{\mathsf{d}} \alpha([\mathcal{X}])$

$$\begin{aligned} \Gamma, S \cup \{\langle l : E_l\rangle_{l\in A} \circ [\mathcal{X}] \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_{l\in B} \circ [\mathcal{Y}]\} \equiv{}& \Gamma, S \cup \{\langle l : E_l\rangle_A \circ 0 \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_B \circ [\mathcal{Y}]\} \cup \\ & \{[\mathcal{X}] \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_{B-A} \circ ([\mathcal{Y}] \vee_A 1)\} \end{aligned} \tag{5.62}$$

$$\begin{aligned} \Gamma, S \cup \{\langle l : E_l\rangle_{l\in B} \circ [\mathcal{Y}] \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_{l\in A} \circ [\mathcal{X}]\} \equiv{}& \Gamma, S \cup \{\langle l : E_l\rangle_B \circ [\mathcal{Y}] \subseteq_{\mathsf{r}(s)} \langle l : E'_l\rangle_A \circ 1\} \cup \\ & \{\langle l : E_l\rangle_{B-A} \circ ([\mathcal{Y}] \wedge_A 0) \subseteq_{\mathsf{r}(s)} [\mathcal{X}]\} \end{aligned} \tag{5.63}$$

$$\Gamma, S \cup \{\mathcal{X} \wedge_A 0 \subseteq_{\mathsf{r}(s)} E\} \equiv \Gamma, S \cup \{\mathcal{X} \subseteq_{\mathsf{r}(s)} E \vee_A 1\} \tag{5.64}$$

$$\Gamma, S \cup \{E \subseteq_{\mathsf{r}(s)} \mathcal{X} \vee_A 1\} \equiv \Gamma, S \cup \{E \wedge_A 0 \subseteq_{\mathsf{r}(s)} \mathcal{X}\} \tag{5.65}$$

Figure 5.6: Resolution of Row-constraints (complex cases).

$$\Gamma, S \cup \{\emptyset \subseteq_{\mathsf{d}} \delta\} \equiv \Gamma, S \tag{5.66}$$

$$\Gamma, S \cup \{\delta_1 \subseteq_{\mathsf{d}} \delta_2 \cup \neg\emptyset\} \equiv \Gamma, S \tag{5.67}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(0) \cup N\} \equiv \Gamma, S \tag{5.68}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(\langle\rangle \cup N)\} \equiv \Gamma, S \tag{5.69}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(1) \cup N\} \equiv \Gamma, S \tag{5.70}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(\mathcal{X}) \cup N\} \equiv \Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha_{\mathcal{X}} \cup N\} \tag{5.71}$$

$$\Gamma, S \cup \{\alpha_{\mathcal{X}} \subseteq_{\mathsf{d}} \alpha_{\mathcal{Y}} \cup N\} \equiv \Gamma, S \cup \{\alpha_{\mathcal{X}} \cap N \subseteq_{\mathsf{d}} \alpha_{\mathcal{Y}}\} \qquad o(\mathcal{X}) < o(\mathcal{Y}) \tag{5.72}$$

$$\Gamma, S \cup \{\alpha(\langle\rangle) \subseteq_{\mathsf{d}} \delta\} \equiv \Gamma, S \cup \{\neg\emptyset \subseteq_{\mathsf{d}} \delta\} \tag{5.73}$$

$$\Gamma, S \cup \{\alpha(\langle l : E_l\rangle_A \circ E^{\mathsf{c}}) \subseteq_{\mathsf{d}} \delta\} \equiv \Gamma, S \cup \{\alpha(E^{\mathsf{c}}) \subseteq_{\mathsf{d}} \delta \cup \neg A\} \tag{5.74}$$

$$\Gamma, S \cup \{N_1 \subseteq_{\mathsf{d}} N_2\} \equiv \Gamma, S \qquad N_1 \subseteq N_2 \tag{5.75}$$

$$\Gamma, S \cup \{N_1 \subseteq_{\mathsf{d}} N_2\} \equiv \Gamma \qquad N_1 \not\subseteq N_2 \tag{5.76}$$

$$\Gamma, S \cup \{N_1 \subseteq_{\mathsf{d}} \delta \cup N_2\} \equiv \Gamma, S \cup \{N_1 - N_2 \subseteq_{\mathsf{d}} \delta\} \tag{5.77}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(\langle l : E_l\rangle_A \circ E^{\perp}) \cup N\} \equiv \Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(\langle l : E_l\rangle_{A-N} \circ 0), \delta \subseteq_{\mathsf{d}} \alpha(E^{\perp}) \cup N\} \tag{5.78}$$

$$\Gamma, S \cup \{N \subseteq_{\mathsf{d}} \alpha(\langle l : E_l\rangle_A \circ 0)\} \equiv \Gamma, S \cup \{E_l \subseteq_s 0 \mid l \in A \cap N\} \tag{5.79}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(\langle l : E_l\rangle_A \circ E) \cup N\} \equiv \Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \neg A \cup N, \delta \subseteq_{\mathsf{d}} \alpha(E) \cup N\} \quad E \neq E^{\perp} \tag{5.80}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(E^{\top} \vee_A 1) \cup N\} \equiv \Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(E^{\top}) \cup (N \cup A)\} \tag{5.81}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(E^{\perp} \vee_A 1)\} \equiv \Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \neg A \cup N, \delta \subseteq_{\mathsf{d}} \alpha(E^{\perp}) \cup N\} \tag{5.82}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(E^{\top} \wedge_A 0)\} \equiv \Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \neg A \cup N, \delta \subseteq_{\mathsf{d}} \alpha(E^{\top}) \cup N\} \tag{5.83}$$

$$\Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(E^{\perp} \wedge_A 0) \cup N\} \equiv \Gamma, S \cup \{\delta \subseteq_{\mathsf{d}} \alpha(E^{\perp}) \cup (N \cup A)\} \tag{5.84}$$

Figure 5.7: Simplification of domain constraints (complete)

if it is inductive and $o(\mathcal{X}) > o(\mathcal{Y})$ for any $\mathcal{Y} \in \mathsf{TLV}(E)$. Similarly, a constraint $\mathcal{X} \subseteq_s E$ is *R-inductive*, if it is inductive and $o(\mathcal{X}) > o(\mathcal{Y})$ for any $\mathcal{Y} \in \mathsf{TLV}(E)$. The following definitions characterize inductive systems.

Definition 5.23 *A system of inductive constraints S is equivalent to a system $S \cup \{E_1 \subseteq_s E_2\}$, if applying the resolution rules of Figures 5.2–5.7 presented in the previous sections to the system $S \cup \{E_1 \subseteq_s E_2\}$ yields the collection of systems $S_1, S_2, \ldots, S_n$ with $n \geq 1$, and $S = S_i$ for some $i = 1..n$.*

Definition 5.24 (Inductive System) *A system S of constraints is inductive, if every constraint $E \subseteq_s E'$ in S is inductive, and furthermore for each pair of L-inductive constraint $E_1 \subseteq_s \mathcal{X}$ and R-inductive constraint $\mathcal{X} \subseteq_s E_2$ in S, S is equivalent to $S \cup \{E_1 \subseteq_s E_2\}$ according to Definition 5.23.*

The algorithm below transforms an arbitrary system S of constraints into a collection of inductive systems.

Algorithm 5.25 Let $\Gamma = S$. Repeat the following steps until no new constraint is added.

1. If Γ contains any system with a constraint that is not inductive, apply the earliest resolution rule appearing in one of the Figures 5.2–5.7 and let Γ be the new collection of constraint systems.

2. If $S \in \Gamma$ contains an L-inductive constraint $E_1 \subseteq_s \mathcal{X}$ and an R-inductive constraint $\mathcal{X} \subseteq_s E_2$, add the transitive constraint $E_1 \subseteq_s E_2$ to S, unless it has been added before.

The algorithm terminates, since the number of distinct inductive constraints is bounded by the size of the original constraints S. Inductive systems have a pleasing property that each constraint forms either a lower or an upper-bound on a variable. Note that a constraint $\mathcal{X} \subseteq_s \mathcal{Y}$ between two variables can either act as a lower-bound on $\mathcal{Y}$, or as an upper-bound on $\mathcal{X}$. We disambiguate these cases using the ordering $o(\cdot)$ on variables by stating that all L-inductive constraints are lower-bounds, and all R-inductive constraints are upper bounds. Thus a constraint $\mathcal{X} \subseteq_s \mathcal{Y}$ is a lower-bound on $\mathcal{Y}$ if $o(\mathcal{X}) < o(\mathcal{Y})$, otherwise it is an upper-bound on $\mathcal{X}$. An inductive system S then has the form

$$\begin{array}{c}
L_{1,1} \sqcup L_{1,2} \sqcup \cdots \sqcup L_{1,l_1} \subseteq_s \mathcal{X}_1 \subseteq_s U_{1,1} \sqcap U_{1,2} \sqcup \cdots \sqcap U_{1,u_1} \\
L_{2,1} \sqcup L_{2,2} \sqcup \cdots \sqcup L_{2,l_2} \subseteq_s \mathcal{X}_2 \subseteq_s U_{2,1} \sqcap U_{2,2} \sqcup \cdots \sqcap U_{2,u_2} \\
\vdots \\
L_{n,1} \sqcup L_{n,2} \sqcup \cdots \sqcup L_{n,l_n} \subseteq_s \mathcal{X}_n \subseteq_s U_{n,1} \sqcap U_{n,2} \sqcup \cdots \sqcap U_{n,u_n}
\end{array} \tag{5.85}$$

where each constraint $L_{i,j} \subseteq_s \mathcal{X}_i$ is L-inductive ($j = 1..l_i$) and each constraint $\mathcal{X}_i \subseteq_s U_{i,j}$ is R-inductive ($j = 1..u_i$). Note the use of joins and meets to combine the lower and upper-bounds on each variable. Even though not all sorts have syntax for these operations, they are semantically well defined for all sorts, and two lower-bounds $L_1 \subseteq_s \mathcal{X}$, $L_2 \subseteq_s \mathcal{X}$ have the effect of constraining $\mathcal{X}$ to be larger than $L_1 \sqcup L_2$ in all solutions, and similarly for meets of upper-bounds. We thus use $\sqcap$ and $\sqcup$ as syntax for meets and joins of all sorts in this section. The meaning function μ is extended naturally by adding the cases

$$\mu[\![E_1 \sqcap_s E_2]\!]\sigma = \mu[\![E_1]\!]\sigma \sqcap_s \mu[\![E_2]\!]\sigma$$
$$\mu[\![E_1 \sqcup_s E_2]\!]\sigma = \mu[\![E_1]\!]\sigma \sqcup_s \mu[\![E_2]\!]\sigma$$

Let $\mathcal{L}_i$ be the set of lower-bounds of variable $\mathcal{X}_i$ and $\mathcal{U}_i$ be the set of upper-bounds of $\mathcal{X}_i$. Without loss of generality, we can assume that for all Set-variables $\mathcal{X}_i$, no $L \in \mathcal{L}_i$ contains any top-level unions and no $U \in \mathcal{U}_i$ contains any top-level intersections (a top-level union $L_1 \cup L_2$ in $\mathcal{L}_i$ can be split into separate expressions L_1 and L_2 of $\mathcal{L}_i$, and similarly for intersections in $\mathcal{U}_i$).

We briefly review the technique of Aiken-Wimmers used to show the existence of solutions for Set-constraints alone (Section 2.2.2). Consider the family of equations obtained from (5.85) by equating each variable $\mathcal{X}_i$ with the join of its lower-bounds, joined with the meets of its upper-bounds and a fresh *auxiliary variable* $\mathcal{Y}_i$ (meets have precedence over joins)

$$\begin{aligned}
\mathcal{X}_1 &= L_{1,1} \sqcup L_{1,2} \sqcup \cdots \sqcup L_{1,l_1} \sqcup \mathcal{Y}_1 \sqcap U_{1,1} \sqcap U_{1,2} \sqcap \cdots \sqcap U_{1,u_1} \\
\mathcal{X}_2 &= L_{2,1} \sqcup L_{2,2} \sqcup \cdots \sqcup L_{2,l_2} \sqcup \mathcal{Y}_2 \sqcap U_{2,1} \sqcap U_{2,2} \sqcap \cdots \sqcap U_{2,u_2} \\
&\vdots \\
\mathcal{X}_n &= L_{n,1} \sqcup L_{n,2} \sqcup \cdots \sqcup L_{n,l_n} \sqcup \mathcal{Y}_n \sqcap U_{n,1} \sqcap U_{n,2} \sqcap \cdots \sqcap U_{n,u_n}
\end{aligned} \tag{5.86}$$

Each assignment for the auxiliary variables $\mathcal{Y}_1..\mathcal{Y}_n$ selects an assignment for the variables $\mathcal{X}_1..\mathcal{X}_n$ in between their lower and upper-bounds. To show that inductive systems have solutions, it is sufficient to show that there exist assignments for the auxiliary variables $\mathcal{Y}_1..\mathcal{Y}_n$ such that the equations (5.86) have solutions for $\mathcal{X}_1..\mathcal{X}_n$, and that these solutions satisfy the constraints of the inductive system S from which the equations were constructed.

We first show that for every choice of $\mathcal{Y}_1..\mathcal{Y}_n$, the induced assignments for $\mathcal{X}_i$ satisfy the constraints up to any finite level j. Section 5.5.2 then discusses contractive systems of equations and shows that—unlike for pure Set-constraints (Section 2.2.2)—not all assignments of $\mathcal{Y}_1..\mathcal{Y}_n$ induce contractive equations for inductive systems of mixed constraints. We form a conjecture that there always exist assignments for the auxiliary variables $\mathcal{Y}_1..\mathcal{Y}_n$ such that the equations induced by (5.86) have solutions. We provide support for this conjecture in Section 5.5.3 and Section 5.5.4. Finally, Section 5.5.5 briefly sketches how the results of Section 5.5.1 can be extended to Row-sorts.

5.5.1 Level Semantics

The following definitions formalize what we mean by satisfying the constraints "up to level j".

Definition 5.26 (Assignment Sequence) *A variable assignment sequence is a sequence of variable assignments $\langle\sigma_j\rangle$, such that the domains of the σ_j agree, and for each variable $\mathcal{X}_i$ of sort s in that domain, σ_j maps $\mathcal{X}_i$ to an element of* S^s_j, i.e., $\sigma_j(\mathcal{X}_i) \in \mathsf{S}^s_j$.

Definition 5.27 *For every level j, assignment sequence $\langle\sigma_i\rangle$ defined up to level j, and mixed expression E of sort s, the meaning function μ_j of the family of meaning functions defined below, assigns to E an element of S_j^s,* i.e., $\mu_j[\![E]\!]\langle\sigma_i\rangle \in \mathsf{S}_j^s$.

$$
\begin{array}{rcll}
\mu_{j\le 0}[\![E^s]\!]\langle\sigma_i\rangle & = & \bot_s & \\
\mu_j[\![1^s]\!]\langle\sigma_i\rangle & = & \top_s & \\
\mu_j[\![0^s]\!]\langle\sigma_i\rangle & = & \bot_s & \\
\mu_j[\![\mathcal{X}^s]\!]\langle\sigma_i\rangle & = & \sigma_j(\mathcal{X}) & \\
\mu_j[\![c(E_1,\dots,E_n)]\!]\langle\sigma_i\rangle & = & \phi_c(\mu_{j-1}[\![E_1]\!]\langle\sigma_i\rangle,\dots,\mu_{j-1}[\![E_n]\!]\langle\sigma_i\rangle) & \\
\mu_j[\![E_1 \sqcap E_2]\!]\langle\sigma_i\rangle & = & \mu_j[\![E_1]\!]\langle\sigma_i\rangle \sqcap_s \mu_j[\![E_2]\!]\langle\sigma_i\rangle & \\
\mu_j[\![E_1 \sqcup E_2]\!]\langle\sigma_i\rangle & = & \mu_j[\![E_1]\!]\langle\sigma_i\rangle \sqcup_s \mu_j[\![E_2]\!]\langle\sigma_i\rangle & \\
\mu_j[\![\neg\{c_1,\dots,c_n\}]\!]\langle\sigma_i\rangle & = & \top_{\mathsf{s}} - \bigcup_{c\in\{c_1,\dots,c_n\}} \phi_c(\top_{\iota_1},\dots,\top_{\iota_{a(c)}}) & c:\iota_1\cdots\iota_{a(c)}\to \mathsf{s} \\
\mu_j[\![\langle\rangle^s]\!]\langle\sigma_i\rangle & = & \mathsf{abs}_{\mathsf{r}(s)} & \\
\mu_j[\![\langle l : E_l\rangle_{\{l\}} \circ E]\!]\langle\sigma_i\rangle & = & \rho_{\{l\}}(\lambda l.\mu_{j-1}[\![E_l]\!]\langle\sigma_i\rangle) \circ \mu_j[\![E]\!]\langle\sigma_i\rangle &
\end{array}
$$

Note that our definition of $\mu_j[\![E]\!]$ is careful in assigning meaning to each sub-expression E' of E according to the nesting depth of E' within E.

Definition 5.28 (Inductive Assignment Sequence) *Let S be an inductive system over a set of variables $\mathcal{X}_1..\mathcal{X}_n$. Consider the system of equations* (5.86) *obtained from S by introducing auxiliary variables $\mathcal{Y}_1..\mathcal{Y}_n$ and let $\langle\sigma_i\rangle$ be an assignment sequence for $\mathcal{Y}_1..\mathcal{Y}_n$.*

The inductive assignment sequence $\mathsf{IAS}(S,\langle\sigma_i\rangle)$ of S with respect to $\langle\sigma_i\rangle$ is the sequence $\langle\sigma'_i\rangle$ defined inductively by

$$
\begin{array}{ll}
\sigma'_0(\mathcal{X}_j) = \bot_s & \mathcal{X}_j \textit{ of sort } s \\
\sigma'_0(\mathcal{Y}_j) = \sigma_0(\mathcal{Y}_j) &
\end{array}
\tag{5.87}
$$

and

$$
\begin{array}{ll}
\sigma'_i(\mathcal{Y}_j) = \sigma_i(\mathcal{Y}_j) & \\
\sigma'_i(\mathcal{X}_j) = \mu_i[\![\bigsqcup_s \mathcal{L}_i \sqcup_s \mathcal{Y}_i \sqcap_s \bigsqcap_s \mathcal{U}_i]\!]\langle\sigma'_i\rangle & \mathcal{X}_j \textit{ has sort } s
\end{array}
\tag{5.88}
$$

Note that equation (5.88) in the definition of $\mathsf{IAS}(S,\langle\sigma_i\rangle)$ is non-recursive since the top-level variables appearing in $\mathcal{L}_j$ and $\mathcal{U}_j$ have indices smaller than j and appear already in σ_i.

Definition 5.29 *An assignment sequence $\langle\sigma_i\rangle$ satisfies a system of inductive constraints S up to level j, if for each $k \le j$, and each constraint $E_1 \subseteq_s E_2$ in S*

$$\mu_k[\![E_1]\!]\langle\sigma_i\rangle \subseteq_s \mu_k[\![E_2]\!]\langle\sigma_i\rangle$$

Theorem 5.30 *If S is inductive, then the inductive variable assignment $\mathsf{IAS}(S,\langle\sigma_i\rangle)$ satisfies the constraints of S up to any finite level j for any auxiliary assignment sequence $\langle\sigma_i\rangle$.*

Proof: By induction on the level j and the variable order $o(\cdot)$. Let $\langle\sigma_i'\rangle = \mathsf{IAS}(S, \langle\sigma_i\rangle)$. For $j = 0$, $\mu_j[\![E]\!]\langle\sigma'\rangle = \bot_s$ for any expression E of sort s, thus the constraints are satisfied at level 0. Suppose the constraints are satisfied up to level $j-1$. To make the non-recursive nature of $\langle\sigma_i'\rangle$ in Definition 5.28 explicit, consider the assignment $\sigma_{j,i}'$ defined as follows.

$$\sigma_{j,i}'(\mathcal{Y}_k) = \sigma_j(\mathcal{Y}_k)$$
$$\sigma_{j,i}'(\mathcal{X}_k) = \begin{cases} \mu_j[\![\bigsqcup \mathcal{L}_k \sqcup \mathcal{Y}_k \sqcap \bigsqcap \mathcal{U}_k]\!]\langle\sigma_0, \dots, \sigma_{j-1}, \sigma_{j,i-1}\rangle & \text{if } k \le i \\ \text{undefined} & \text{otherwise} \end{cases}$$

At the boundary case, let $\sigma_{j,0}$ be undefined for all $\mathcal{X}_k$. To simplify the notation, let $\langle\sigma_{j,i}\rangle$ denote the sequence $\langle\sigma_0, \dots, \sigma_{j-1}, \sigma_{j,i}\rangle$. Consider variable $\mathcal{X}_1$ and $\sigma_{j,1}'(\mathcal{X}_1)$ defined by

$$\sigma_{j,1}'(\mathcal{X}_1) = \mu_j[\![\bigsqcup \mathcal{L}_1 \sqcup \mathcal{Y}_1 \sqcap \bigsqcap \mathcal{U}_1]\!]\langle\sigma_{j,0}\rangle$$

The definition is well-formed since no expression in $\mathcal{L}_1$ or $\mathcal{U}_1$ contains any top-level variables and thus no lookup on $\sigma_{j,0}$ is performed. We need to show that

1. the lower-bound constraints are satisfied at level j, *i.e.*,
$$\mu_j[\![\bigsqcup \mathcal{L}_1]\!]\langle\sigma_{j,1}'\rangle \subseteq_s \sigma_{j,1}'(\mathcal{X}_1)$$
and that

2. the upper-bound constraints are satisfied at level j, *i.e.*,
$$\sigma_{j,1}'(\mathcal{X}_1) \subseteq_s \mu_j[\![\bigsqcap \mathcal{U}_1]\!]\langle\sigma_{j,1}'\rangle$$

The lower-bound constraints are satisfied, since

$$\begin{aligned} \mu_j[\![\bigsqcup \mathcal{L}_1]\!]\langle\sigma_{j,1}'\rangle &= \mu_j[\![\bigsqcup \mathcal{L}_1]\!]\langle\sigma_{j,0}'\rangle \\ &\subseteq_s \mu_j[\![\bigsqcup \mathcal{L}_1]\!]\langle\sigma_{j,0}\rangle \sqcup \mu_j[\![\mathcal{Y}_1 \sqcap \bigsqcap \mathcal{U}_1]\!]\langle\sigma_{j,0}\rangle \\ &= \mu_j[\![\bigsqcup \mathcal{L}_1 \sqcup \mathcal{Y}_1 \sqcap \bigsqcap \mathcal{U}_1]\!]\langle\sigma_{j,0}\rangle \\ &= \sigma_{j,1}'(\mathcal{X}_1) \end{aligned}$$

For the upper-bounds, we have that

$$\begin{aligned} \sigma_{j,1}'(\mathcal{X}_1) &= \mu_j[\![\bigsqcup \mathcal{L}_1 \sqcup \mathcal{Y}_1 \sqcap \bigsqcap \mathcal{U}_1]\!]\langle\sigma_{j,0}\rangle \\ &= \mu_j[\![\bigsqcup \mathcal{L}_1]\!]\langle\sigma_{j,0}\rangle \sqcup \mu_j[\![\mathcal{Y}_1 \sqcap \bigsqcap \mathcal{U}_1]\!]\langle\sigma_{j,0}\rangle \end{aligned}$$

Since $I \sqcup J \subseteq_s K \iff I \subseteq_s K \;\wedge\; K \subseteq_s K$, we can break up the relation into two relations

$$\mu_j[\![\bigsqcup \mathcal{L}_1]\!]\langle\sigma_{j,0}\rangle \subseteq_s \mu_j[\![\bigsqcap \mathcal{U}_1]\!]\langle\sigma_{j,1}'\rangle$$
$$\mu_j[\![\mathcal{Y}_1 \sqcap \bigsqcap \mathcal{U}_1]\!]\langle\sigma_{j,0}\rangle \subseteq_s \mu_j[\![\bigsqcap \mathcal{U}_1]\!]\langle\sigma_{j,1}\rangle$$

Since $\mu_j[\![\bigsqcap\mathcal{U}_1]\!]\langle\sigma'_{j,1}\rangle = \mu_j[\![\bigsqcap\mathcal{U}_1]\!]\langle\sigma'_{j,0}\rangle$, the second relation is trivially satisfied. For the first relation, we need to show that for all $E \in \mathcal{L}_1$ and $E' \in \mathcal{U}_1$,

$$\mu_j[\![E]\!]\langle\sigma'_{j,0}\rangle \subseteq_s \mu_j[\![E']\!]\langle\sigma'_{j,0}\rangle$$

Since S is inductive, S is equivalent to $S \cup \{E \subseteq_s E'\}$ according to Definition 5.23. Thus if $E = 1$, then either $\mathcal{U}_1$ is empty, or $E' = 1$. Similarly if $E' = 0$. If $E = 0$ or $E' = 1$, the constraint is trivially satisfied. We split up the remaining cases according to the sort of $\mathcal{X}_1$.

- Suppose $\mathcal{X}_1$ is of Term-sort. Then $E = c(E_1, \dots, E_n)$ and $E' = c'(E'_1, \dots, E'_m)$. Since S is inductive, S is equivalent to $S \cup \{c(E_1, \dots, E_n) \subseteq_{\mathsf{t}} c'(E'_1, \dots, E'_m)\}$. Thus $c = c'$ and S contains constraints equivalent to $E_k \subseteq_{\iota_k} E'_k$ and $E'_k \subseteq_{\iota_k} E_k$ for all $k = 1..n$ (by Rule 5.27). Since the constraints are satisfied at level $j-1$, we know that $\mu_{j-1}[\![E_k]\!]\langle\sigma'_{j,0}\rangle = \mu_{j-1}[\![E'_k]\!]\langle\sigma'_{j,0}\rangle$ for $k = 1..n$, and thus $\mu_j[\![c(E_1, \dots, E_n)]\!]\langle\sigma'_{j,0}\rangle = \mu_j[\![c(E'_1, \dots, E'_n)]\!]\langle\sigma'_{j,0}\rangle$, which is equivalent to $\mu_j[\![E]\!]\langle\sigma'_{j,0}\rangle \subseteq_{\mathsf{t}} \mu_j[\![E']\!]\langle\sigma'_{j,0}\rangle$ (Definition of $\subseteq_{\mathsf{t}}$ in Section 4.4).

- If $\mathcal{X}_1$ is of $\mathsf{FlowTerm}$-sort, then $E = c(E_1, \dots, E_n)$ and $E' = c'(E'_1, \dots, E'_m)$. Since S is inductive, S is equivalent to $S \cup \{c(E_1, \dots, E_n) \subseteq_{\mathsf{t}} c'(E'_1, \dots, E'_m)\}$. Thus $c = c'$ and S contains constraints equivalent to $E_k \subseteq_{\iota_k} E'_k$ for all $k = 1..n$ (by Rule 5.21). Since the constraints are satisfied at level $j-1$, we know that $\mu_{j-1}[\![E_k]\!]\langle\sigma'_{j,0}\rangle \subseteq_{\iota_k} \mu_{j-1}[\![E'_k]\!]\langle\sigma'_{j,0}\rangle$ for $k = 1..n$, and thus $\mu_j[\![c(E_1, \dots, E_n)]\!]\langle\sigma'_{j,0}\rangle \subseteq_{\mathsf{ft}} \mu_j[\![c(E'_1, \dots, E'_n)]\!]\langle\sigma'_{j,0}\rangle$ (Definition of $\subseteq_{\mathsf{ft}}$ in Section 4.4).

- If $\mathcal{X}_1$ is of Set-sort, then there are a number of cases for the forms of E and E'. We can focus on the cases where E or E' contains variables, since S is equivalent to $S \cup \{E \subseteq_{\mathsf{s}} E'\}$ and constraints between ground expressions are either satisfied in all solutions or in none. We focus on the most interesting case $E = c(E_1, \dots, E_n)$ and $E' = \mathsf{Pat}[c'(E'_1, \dots, E'_m), M,]$. This constraint is equivalent to $E \cap M \subseteq_{\mathsf{s}} E'$ by Rule 5.4. There are two cases. If the intersection $c(\top_{\iota_1}, \dots, \top_{\iota_n}) \cap M$ is empty, then the constraint is trivially satisfied. Otherwise the intersection is equal to $c(M_1, \dots, M_n)$, and by Axiom 5.8 the constraint is equivalent to

$$c(E_1 \mathbin{\overline{\cap}} M_1, \dots, E_n \mathbin{\overline{\cap}} M_n) \subseteq_{\mathsf{s}} c'(E'_1, \dots, E'_m)$$

  If $c \neq c'$, then by Rule 5.8, c must be strict and S contains constraints equivalent to $E_k \cap M_k \subseteq_{\iota_k} 0$ for some k. Since the constraints are satisfied at level $j-1$, we have $\mu_{j-1}[\![E_k \cap M_k]\!]\langle\sigma'_{j,0}\rangle = \bot_{\iota_k}$ and $\mu_j[\![c(E_1 \mathbin{\overline{\cap}} M_1, \dots, E_n \mathbin{\overline{\cap}} M_n)]\!]\langle\sigma'_{j,0}\rangle = \bot_{\mathsf{s}}$. Thus the constraint is satisfied at level j. Otherwise, we have $c = c'$ and by Rule 5.8, we are either in the above case, or S contains constraints equivalent to $E_k \mathbin{\overline{\cap}} M_k \subseteq_{\iota_k} E'_k$ for $k = 1..n$. Since the constraints are satisfied at level $j-1$, we have $\mu_{j-1}[\![E_k \mathbin{\overline{\cap}} M_k]\!]\langle\sigma'_{j,0}\rangle \subseteq_{\iota_k} \mu_{j-1}[\![E'_k]\!]\langle\sigma'_{j,0}\rangle$ and thus by Axiom 4.1 (c's interpretation observes the declared variance) $\mu_j[\![c(E_1 \mathbin{\overline{\cap}} M_1, \dots, E_n \mathbin{\overline{\cap}} M_n)]\!]\langle\sigma'_{j,0}\rangle \subseteq_{\mathsf{V^s}} \mu_j[\![c(E'_1, \dots, E'_n)]\!]\langle\sigma'_{j,0}\rangle$. If c is non-strict, then for any interface path $p = (c,k)q^t$, if $q = \epsilon$, then the interface constraint $(c,k)^{-1}(\mu_{j-1}[\![E]\!]\langle\sigma'_{j,0}\rangle) \subseteq_{\iota_k} (c,k)^{-1}(\mu_{j-1}[\![E']\!]\langle\sigma'_{j,0}\rangle)$ is equivalent to $\mu_{j-1}[\![E_k \mathbin{\overline{\cap}} M_k]\!]\langle\sigma'_{j,0}\rangle \subseteq_{\iota_k} \mu_{j-1}[\![E'_k]\!]\langle\sigma'_{j,0}\rangle$ and is satisfied. Otherwise, we know that the interface constraint q is satisfied by $\mu_{j-1}[\![E_k \mathbin{\overline{\cap}} M_k]\!]\langle\sigma'_{j,0}\rangle \subseteq_{\iota_k} \mu_{j-1}[\![E'_k]\!]\langle\sigma'_{j,0}\rangle$ and thus the interface constraint on path p is satisfied.

We proceed by induction on the variables $\mathcal{X}_2..\mathcal{X}_n$. Assume that $\sigma'_{j,i-1}$ satisfies the constraints up to level $j-1$ for all variables, and up to level j for variables $\mathcal{X}_1..\mathcal{X}_{i-1}$. Consider the case for variable $\mathcal{X}_i$. The only additional case not covered above for variable $\mathcal{X}_1$ is when E or E' contain a top-level variable. Suppose that $E = \mathcal{X}_k$. Then since S is inductive, a constraint equivalent to $\mathcal{X}_k \subseteq_s E'$ is in S. There are two cases, if the index of $\mathcal{X}_k$ is larger than the index of any top-level variables of E', then S contains the inductive constraint $\mathcal{X}_k \subseteq_s E'$. Since $k < i$ and $\langle\sigma'_{j,i-1}\rangle$ satisfies the constraints up to level j for all variables $\mathcal{X}_k$ with index less than i, the constraint is satisfied at level j. Otherwise, E' contains a top-level variable $\mathcal{X}_{k'}$ with index higher than $\mathcal{X}_k$. Then the constraint is equivalent to an inductive constraint $E'' \subseteq_s \mathcal{X}_{k'}$ and since $k' < i$, the constraint is also satisfied at level j. For Set-sorts, E can be of the form $\mathcal{X}_k \cap M$ and E' can be of the form $\mathsf{Pat}[\mathcal{X}_{k'}, M]$, but the basic argument does not change. □

Definition 5.31 *A series of assignments $\langle\sigma_i\rangle$ is Cauchy, if for every variable $\mathcal{X}$ in the domain of the series, the sequence $\sigma_0(\mathcal{X}), \sigma_1(\mathcal{X}), \sigma_2(\mathcal{X}), \ldots$ is Cauchy with respect to Definition 2.1 and the metric on ideals given in Section 2.1.*

Theorem 5.32 (Solutions) *If $\mathsf{IAS}(S, \langle\sigma_i\rangle)$ is Cauchy, then the limit of $\mathsf{IAS}(S, \langle\sigma_i\rangle)$ is a solution of S.*

Proof: Note that the expressions in S have a maximal height, say k. By definition of μ_j, $\mu_j[\![E]\!]\langle\sigma_i\rangle$ only requires assignments $\langle\sigma_{j-k}, \ldots, \sigma_j\rangle$ (assuming $j > k$). If $\mathsf{IAS}(S, \langle\sigma_i\rangle)$ converges to $\overline{\sigma}$, then $\mu_{k+1}[\![E]\!]\langle\underbrace{\overline{\sigma}, \ldots, \overline{\sigma}}_{k+1 \text{ times}}\rangle$ is equal to $\mu[\![E]\!]\overline{\sigma}$ (Section 4.5), and thus $\overline{\sigma}$ is a solution of the constraints S. □

5.5.2 Contractive Systems of Equations

Aiken-Wimmers prove the existence of solutions for pure Set-constraints by showing that the solutions of an inductive system of Set-constraints are equivalent to the solutions of a family of contractive equations. We describe why this proof cannot be adapted to mixed constraints and then provide support for a conjecture that inductive systems of mixed constraints nevertheless have solutions.

Consider again the family of equations (5.86) obtained from an inductive system S. Each assignment of the auxiliary variables $\mathcal{Y}_1..\mathcal{Y}_n$ selects a system of *cascading equations*

$$\{\mathcal{X}_1 = \mathsf{RHS}_1, \ldots, \mathcal{X}_n = \mathsf{RHS}_n\}$$

from (5.86). An equation $\mathcal{X}_i = \mathsf{RHS}_i$ is cascading, if every top-level variable $\mathcal{X}_k$ in RHS_i has lower index than $\mathcal{X}_i$ ($o(\mathcal{X}_k) < o(\mathcal{X}_i)$). The equations are cascading since they were obtained from the inductive constraints of S and thus the top-level variables in $L_{i,k}$ and $U_{i,k}$ have indices strictly lower than $o(X_i)$. Due to this property, we can transform cascading equations into equations without top-level variables. To see this, order the equations according to the variable order $o(\cdot)$, *i.e.*, let the indices of the variables match the order $o(\cdot)$. Then, clearly

RHS_1 has no top-level variables. Assume that we have transformed all RHS_k into RHS'_k where RHS'_k has no top-level variables for $k = 1..i-1$. Then RHS_i can be transformed into RHS'_i without top-level variables by replacing each top-level variable $\mathcal{X}_k$ in RHS_i by RHS'_k. Since the equation $\mathcal{X}_i = \mathsf{RHS}_i$ is cascading, each top-level variable $\mathcal{X}_k$ in RHS_i has index lower than $\mathcal{X}_i$ and thus RHS'_k has already been computed.

The only operations in RHS'_i are joins, meets, and constructor applications. If all expressions denote pure Set-ideals, the joins and meets are set-theoretic union and intersection. These operations are non-expansive (Section 2.1). Since all variables appear inside constructors in RHS'_i and constructor interpretations are contractive (Axiom 4.1), the system of equations $\mathcal{X}_i = \mathsf{RHS}'_i$ is contractive and thus has a unique solution (Section 2.1).

Unfortunately, in the case of mixed constraints, the meet and join operations are not always non-expansive, *i.e.*, for some arguments, the meet and join operations can be expansive. The resulting equations are thus not contractive. The culprits are the meet and join operations for Term-ideals $I \sqcap_{\mathbf{t}} J$ and $I \sqcup_{\mathbf{t}} J$ which are expansive if the two operands are of the form $I = \phi_c(I_1, \ldots, I_n)$, and $J = \phi_c(J_1, \ldots, J_n)$, and I_k differs from J_k for some k. In this case, $I \sqcup_{\mathbf{t}} J = \top_{\mathbf{t}}$ and $I \sqcap_{\mathbf{t}} J = \bot_{\mathbf{t}}$. These operations are expansive in I and J, meaning that if I and I' have distance d, then the distance between between the join $I \sqcup_{\mathbf{t}} J$ and the join $I' \sqcup_{\mathbf{t}} J$ may be larger than d, and similarly for the meet and the second argument (Section 2.1). The expansiveness of the meet and join for Term-ideals then induces expansiveness for the meet and join of other sorts. As an example, consider the inductive system S

$$\begin{aligned} c(\mathcal{X}_1) \sqcup_{\mathbf{t}} c(\mathcal{X}_2) &\subseteq_{\mathbf{t}} \mathcal{X}_1 \\ \top_{\mathbf{t}} &\subseteq_{\mathbf{t}} \mathcal{X}_2 \end{aligned} \tag{5.89}$$

where $c : \mathbf{t} \to \mathbf{t}$ is a unary Term-constructor. The family of equations associated with S are

$$\begin{aligned} \mathcal{X}_1 &= c(\mathcal{X}_1) \sqcup_{\mathbf{t}} c(\mathcal{X}_2) \sqcup_{\mathbf{t}} \mathcal{Y}_1 \\ \mathcal{X}_2 &= \top_{\mathbf{t}} \sqcup_{\mathbf{t}} \mathcal{Y}_2 \end{aligned} \tag{5.90}$$

The cascading equations obtained from the above family through the auxiliary variable assignment $\mathcal{Y}_1 = \bot_{\mathbf{t}}$ and $\mathcal{Y}_2 = \bot_{\mathbf{t}}$ is

$$\begin{aligned} \mathcal{X}_1 &= c(\mathcal{X}_1) \sqcup_{\mathbf{t}} c(\mathcal{X}_2) \\ \mathcal{X}_2 &= \top_{\mathbf{t}} \end{aligned}$$

This system is not contractive, since the join $c(\mathcal{X}_1) \sqcup_{\mathbf{t}} c(\mathcal{X}_2)$ is expansive for some assignments of $\mathcal{X}_1$ and $\mathcal{X}_2$. To illustrate the problem with non-contractive equations, consider the function

$$F(\sigma)(\mathcal{X}_i) = \mu[\![\mathsf{RHS}'_i]\!]\sigma \tag{5.91}$$

transforming any variable assignment σ into a new variable assignment σ' through the equations. A fix-point of F is a solution of the equations. Contractiveness guarantees that F has a fix-point, and that the fix-point is reached by the sequence of variable assignments $\sigma_0, \sigma_1 = F(\sigma_0), \sigma_2 = F(F(\sigma_0)), \ldots, \sigma_i = F^i(\sigma_0), \ldots$ starting from an arbitrary assignment σ_0. The sequence of variable assignments obtained in our example starting with $[\mathcal{X}_1 \mapsto \bot, \mathcal{X}_2 \mapsto \bot]$ is

Assignment	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5	
$\mathcal{X}_1$	$\bot$	$c(\bot)$	$\top$	$c(\top)$	$\top$	$c(\top)$	$\cdots$
$\mathcal{X}_2$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\cdots$

which doesn't converge. However, there are different choices for the auxiliary variable assignment of $\mathcal{Y}_1$ so that the equations obtained from (5.90) are contractive and thus have solutions. For example, choosing $\mathcal{Y}_1 = \top$, results in the equations

$$\begin{aligned}\mathcal{X}_1 &= \top_{\mathbf{t}}\\ \mathcal{X}_2 &= \top_{\mathbf{t}}\end{aligned}$$

which clearly define a solution of the inductive system (5.89). Thus, in the case of mixed constraints, not all auxiliary variable assignments $\mathcal{Y}_i$ induce contractive equations and solutions. However, in order for the series $\langle F^i(\sigma_0)\rangle$ to converge, contractiveness is not necessary. It is sufficient if the series $\langle F^i(\sigma_0)\rangle$ is Cauchy according to Definition 5.31.

We conjecture that there always exist choices for $\mathcal{Y}_i$ such that the equations induced by the family (5.86), generate a series of assignments that is Cauchy.

Conjecture 5.33 *If S is an inductive system of constraints, then there exists an auxiliary variable assignment for $\mathcal{Y}_i$, such that the equations induced by the family* (5.86) *generate a series of assignments that is Cauchy.*

To support this conjecture, we proceed as follows. Section 5.5.3 shows how to eliminate all joins and meets appearing directly in bounds of Term-variables in an inductive system S by generating contractive equations for Term-variables. We show that these equations together with the equations (5.86) for the FlowTerm and Set variables satisfy the constraints up to any finite level for any auxiliary variable assignment. Section 5.5.4 then proposes a construction of auxiliary variable assignments $\mathcal{Y}_1..\mathcal{Y}_n$ for each FlowTerm and Set-variable $\mathcal{X}_i$ such that the induced equations (5.86) combined with the equations on Term-variables should generate a series of variable assignments for $\mathcal{X}_i$ that is Cauchy and thus converges towards a solution of the constraints.

5.5.3 Contractive Term-Equations

This subsection shows how to eliminate meets and joins appearing directly in bound of Term-variables. We proceed by induction on the order of the Term-variables, creating a contractive equation $\mathcal{X}_i = \mathsf{RHS}'_i$ from the upper and lower bounds $\mathcal{U}_i$ and $\mathcal{L}_i$, such that $\bigsqcup \mathcal{L}_i \subseteq_{\mathbf{t}} E_i \subseteq_{\mathbf{t}} \bigsqcap \mathcal{U}_i$. Consider the Term-variable $\mathcal{X}_1$ of minimum index with lower bounds $\mathcal{L}_1$ and upper bounds $\mathcal{U}_1$ (Without loss of generality, we can assume that Term-variables have indices $1..m-1$). Recall that no expression L in $\mathcal{L}_1$ or U in $\mathcal{U}_1$ contains any top-level variables. Thus each L is of the form 0, 1, or $c(\dots)$ and similarly for each U in $\mathcal{U}_1$. Let $\mathcal{U}'_1$ be $\mathcal{U}_1 - \{1\}$, and let $\mathcal{L}'_1 = \mathcal{L}_1 - \{0\}$. The removed expressions 1 from $\mathcal{U}_1$ and removed expressions 0 from $\mathcal{L}_1$ do not constrain the solutions for $\mathcal{X}_1$. Now, if $\mathcal{L}'_1 = \emptyset$, we set $\mathsf{RHS}'_1 = \bot$. Clearly, $\bot \subseteq_{\mathbf{t}} \bigsqcap \mathcal{U}_1$, and also $\bigsqcup \mathcal{L}_1 \subseteq_{\mathbf{t}} \bot$. Analogously, if $\mathcal{U}'_1 = \emptyset$, we set $\mathsf{RHS}'_1 = \top$. Otherwise, $\mathcal{L}'_1$ and $\mathcal{U}'_1$ contain only constructed expressions (if $1 \in \mathcal{L}_1$, then by transitivity, $\mathcal{U}'_1 = \emptyset$, and similarly, if $0 \in \mathcal{U}_1$). Thus, consider the expressions $c_i(E_{i,1}, \dots, E_{i,a(c_i)})$ in $\mathcal{L}'_1$ for some set of indices $i \in I$, and the expressions $c'_j(E'_{j,1}, \dots, E'_{j,a(c'_j)})$ in $\mathcal{U}'_1$ for some indices $j \in J$. Since our system S is inductive, it is closed under transitivity, and we can conclude that $c_i = c'_j$ for all i and j. Then by Rule 5.27 for Term-constructors, S contains constraints equivalent to $E_{i,k} \subseteq_{s_k} E'_{j,k}$ and $E'_{j,k} \subseteq_{s_k} E_{i,k}$ for all $k = 1..a(c_i)$, $i \in I$, and $j \in J$. Suppose we have an assignment sequence $\langle \sigma'_{j-1} \rangle$ that satisfies the constraints up to level $j-1$. Then we have $\mu_{j-1}[\![E_{i,k}]\!]\langle \sigma'_{j-1} \rangle = \mu_{j-1}[\![E'_{j,k}]\!]\langle \sigma'_{j-1} \rangle$ for all $k = 1..a(c_i)$, $i \in I$, and $j \in J$. Thus, expressions $c_i(\mathcal{Y}_{i,1}, \dots, \mathcal{Y}_{i,a(c_i)})$ and $c'_j(\mathcal{Y}'_{j,1}, \dots, \mathcal{Y}'_{j,a(c'_j)})$ are equal at level j. We can thus set $\mathsf{RHS}'_1 = c_i(E_{i,1}, \dots, E_{i,a(c')})$ for an arbitrary $i \in I$, and $\mathcal{X}_1$ will satisfy the constraints at any level j, provided all other variables satisfy the constraints up to level $j-1$. Now the same construction can be performed for the next Term-variable $\mathcal{X}_2$ in the order, using RHS'_1 to expand any appearance of $\mathcal{X}_1$ at top-level in $\mathcal{L}_2$ or $\mathcal{U}_2$. Proceeding in order on all Term-variables, we obtain a system of Term-equations

$$\begin{aligned} \mathcal{X}_1 &=_{\mathbf{t}} \mathsf{RHS}'_1 \\ \mathcal{X}_2 &=_{\mathbf{t}} \mathsf{RHS}'_2 \\ &\vdots \\ \mathcal{X}_{m-1} &=_{\mathbf{t}} \mathsf{RHS}'_{m-1} \end{aligned} \tag{5.92}$$

where all RHS'_i are of the form 0, 1, or $c_i(\dots)$ for some constructor c_i. Since these equations contain no meets or joins and all variables appear inside constructors, the equations are contractive. Similarly to $\mathcal{X}_1$, the assignment of $\mathcal{X}_i$ for $i = 2..m-1$ satisfy the constraints at any level j, provided that all variables satisfy the constraints up to level $j-1$.

5.5.4 Generating Set and FlowTerm-Equations

Unfortunately, even if we eliminate all meets and joins in bounds of Term-variables as outlined in the previous subsection, we still don't have contractive equations for all variables. Joins and meets on Term-ideals may appear in joins and meets of FlowTerm and Set ideals. As an example, consider the inductive system

$$\begin{aligned} c(\mathcal{X}_3) \subseteq_{\mathbf{t}} \mathcal{X}_1 &\subseteq_{\mathbf{t}} c(\mathcal{X}_3) \\ c(\mathcal{X}_4) \subseteq_{\mathbf{t}} \mathcal{X}_2 &\subseteq_{\mathbf{t}} c(\mathcal{X}_4) \\ d(\mathcal{X}_1) \cup d(\mathcal{X}_2) &\subseteq_{\mathbf{s}} \mathcal{X}_3 \\ d(1) &\subseteq_{\mathbf{s}} \mathcal{X}_4 \end{aligned}$$

where $\mathcal{X}_1$ and $\mathcal{X}_2$ are Term-variables, $\mathcal{X}_3$ and $\mathcal{X}_4$ are Set-variables, c is a Term-constructor, and d a Set-constructor with signatures

$$\begin{aligned} c &: \mathbf{s} \to \mathbf{t} \\ d &: \mathbf{t} \to \mathbf{s} \end{aligned}$$

Using the construction above, we transform the constraints on the Term-variables $\mathcal{X}_1$ and $\mathcal{X}_2$ into the contractive equations

$$\begin{aligned} \mathcal{X}_1 &= c(\mathcal{X}_3) \\ \mathcal{X}_2 &= c(\mathcal{X}_4) \end{aligned}$$

Choosing the auxiliary variable assignment $[\mathcal{Y}_3 \mapsto \bot, \mathcal{Y}_4 \mapsto \bot]$ we obtain the system of equations

$$\begin{aligned} \mathcal{X}_1 &= c(\mathcal{X}_3) \\ \mathcal{X}_2 &= c(\mathcal{X}_4) \\ \mathcal{X}_3 &= d(\mathcal{X}_1) \sqcup_{\mathsf{s}} d(\mathcal{X}_2) \\ \mathcal{X}_4 &= d(1) \end{aligned} \tag{5.93}$$

This system is not contractive, since the join $d(\mathcal{X}_1) \sqcup_{\mathsf{s}} d(\mathcal{X}_2)$ involves the join $\mathcal{X}_1 \sqcup_{\mathsf{t}} \mathcal{X}_2$ which is expansive for some assignments of $\mathcal{X}_1$ and $\mathcal{X}_2$. The sequence of variable assignments obtained starting with $[\mathcal{X}_i \mapsto \bot]$ for $i = 1..4$ is

	σ_0	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8	
$\mathcal{X}_1$	$\bot$	$c(\bot)$	$c(d(\bot))$	$c(d(c(\bot)))$	$c(d(\top))$	$c(d(\top))$	$c(d(c(d(\top))))$	$c(d(c(d(\top))))$	$c(d(\top))$	$\cdots$
$\mathcal{X}_2$	$\bot$	$c(\bot)$	$c(d(\top))$	$c(d(\top))$	$c(d(\top))$	$c(d(\top))$	$c(d(\top))$	$c(d(\top))$	$c(d(\top))$	$\cdots$
$\mathcal{X}_3$	$\bot$	$d(\bot)$	$d(c(\bot))$	$d(\top)$	$d(\top)$	$d(c(d(\top)))$	$d(c(d(\top)))$	$d(\top)$	$d(\top)$	$\cdots$
$\mathcal{X}_4$	$\bot$	$d(\top)$	$d(\top)$	$d(\top)$	$d(\top)$	$d(\top)$	$d(\top)$	$d(\top)$	$d(\top)$	$\cdots$

which clearly doesn't converge, since assignments 4–7 are repeated ad infinitum.

Observe that we can obtain a system of equations generating a converging series of assignments in the example of the previous subsection, if we choose a different auxiliary variable assignment where $\mathcal{Y}_3 = d(\top)$. In that case, the induced equations are

$$\begin{aligned} \mathcal{X}_1 &= c(\mathcal{Y}_1) \\ \mathcal{X}_2 &= c(\mathcal{Y}_2) \\ \mathcal{X}_3 &= d(\mathcal{X}_1) \sqcup_{\mathsf{s}} d(\mathcal{X}_2) \sqcup_{\mathsf{s}} d(\top_{\mathsf{t}}) \\ \mathcal{X}_4 &= d(1) \end{aligned}$$

The right-side of equation $\mathcal{X}_3$ is now contractive, since it results in $d(\top_{\mathsf{t}})$ for any assignment of $\mathcal{X}_1$ and $\mathcal{X}_2$.

In this section we construct auxiliary variable assignments for each $\mathcal{Y}_i$ where $\mathcal{X}_i$ is a FlowTerm or Set-variable. The assignments for $\mathcal{Y}_i$ are defined through their own system of equations. The basic idea is that if we choose $\mathcal{Y}_1..\mathcal{Y}_n$ to contain only $\top_{\mathsf{t}}$ and $\bot_{\mathsf{t}}$ for all projections of Term-interface paths p, then all problematic meets and joins of Term-ideals arise from Term-variables syntactically present in the upper and lower-bounds of FlowTerm and Set-variables. This follows from Definition 3.1 which restricts the Term-arguments to FlowTerm and Set-constructors to be either 1^{t}, 0^{t}, or a Term-variable $\mathcal{X}$. Since Term-meets and Term-joins involving 1 and 0 are non-expansive, the potentially expansive meets and joins involve the denotation of a Term-variable.

Our construction is based on the following idea. For every FlowTerm or Set-variable $\mathcal{X}_i$, we choose $\mathcal{Y}_i$ such that the projection $p^{-1}(\mathcal{Y}_i)$ is $\top_{\mathsf{t}}$ for every even Term-path p where $p^{-1}(\bigsqcup_{L \in \mathcal{L}_i} L)$ is determined in part by a Term-variable $\mathcal{X}$, and $\mathcal{X}$ is defined by an equation $\mathcal{X} = c(E_1, \ldots, E_{a(c)})$, *i.e.*, $\mathcal{X}$ is non-empty in all solutions (recall from Axiom 4.1 that Term-constructors are non-strict). If $\mathcal{X}_i$ is defined by

$$\mathcal{X}_i = \bigsqcup_{L \in \mathcal{L}_i} L \sqcup_s \mathcal{Y}_i \sqcap_s \prod_{U \in \mathcal{U}_i} U$$

then for any even path p with interface Term, the projection $p^{-1}(\mathcal{X}_i)$ depends on $p^{-1}(\bigsqcap_{U\in\mathcal{U}_i} U)$. If $p^{-1}(\bigsqcap_{U\in\mathcal{U}_i} U)$ is $\top_{\mathsf{t}}$ in all solutions, *i.e.*, the upper-bounds of $\mathcal{X}_i$ do not constrain the solution of $\mathcal{X}_i$ at interface p, then

$$\begin{aligned} p^{-1}(\mathcal{X}_i) &= p^{-1}(\bigsqcup_{L\in\mathcal{L}_i} L) \sqcup_{\mathsf{t}} p^{-1}(\mathcal{Y}_i) \sqcap_{\mathsf{t}} p^{-1}(\bigsqcap_{U\in\mathcal{U}_i} U) \\ &= p^{-1}(\bigsqcup_{L\in\mathcal{L}_i} L) \sqcup_{\mathsf{t}} \top_{\mathsf{t}} \sqcap_{\mathsf{t}} \top_{\mathsf{t}} \\ &= \top_{\mathsf{t}} \end{aligned}$$

Otherwise, the upper-bounds $\mathcal{U}_i$ constrain the solution of $\mathcal{X}_i$ at interface p to be less than some Term-variable $\mathcal{X}'$, and since $p^{-1}(\bigsqcup_{L\in\mathcal{L}_i} L) \subseteq_{\mathsf{t}} p^{-1}(\bigsqcap_{U\in\mathcal{U}_i} U)$ (transitivity of inductive systems), $\mathcal{X}'$ is defined by an equation $\mathcal{X}' = c(E'_1, \ldots, E'_{a(c)})$ involving the same constructor as the equation of $\mathcal{X}$ which constrains $\mathcal{X}_i$ at interface p from below. Furthermore, the inductive system S constrains the arguments of the constructed expressions $c(E_1, \ldots, E_{a(c)})$ and $c(E'_1, \ldots, E'_{a(c)})$ to be equal ($E_k = E'_k$ for $k = 1..a(c)$).[4] Since this argument holds for all pairs of such variables $\mathcal{X}$ and $\mathcal{X}'$, we know that these variables denote equal Term-ideals I in all solutions and their meet or join is thus non-expansive. The projection of $\mathcal{X}_i$ at interface p is then

$$\begin{aligned} p^{-1}(\mathcal{X}_i) &= p^{-1}(\bigsqcup_{L\in\mathcal{L}_i} L) \sqcup_{\mathsf{t}} p^{-1}(\mathcal{Y}_i) \sqcap_{\mathsf{t}} p^{-1}(\bigsqcap_{U\in\mathcal{U}_i} U) \\ &= I \sqcup_{\mathsf{t}} \top_{\mathsf{t}} \sqcap_{\mathsf{t}} I \\ &= I \end{aligned}$$

The idea for odd paths is similar. To simplify the presentation, we assume that the original constraints are in *normal form*, where the arguments E_i of each constructed expression $c(E_1, \ldots, E_n)$ are variables, unless the entire expression is an M-expression M. Since the only new constructor expressions introduced by the resolution rules of the previous sections arise in simplifications of L-intersections $c(E_1, \ldots, E_n) \cap M$ to $c(E_1 \cap M_1, \ldots, E_n \cap M_n)$, we can assume that each argument E_i of a constructed expression $c(E_1, \ldots, E_n)$ is either a variables $\mathcal{X}$, or a variable intersected with an M-expression $\mathcal{X} \cap M$. The latter case only arises for Set-variables.

Definition 5.34 *A non-strict* Term*-path p is a path with interface* $\mathbf{t}$ *(Section 4.2). We say that p is syntactically present in variable $\mathcal{X}_i$ of an inductive system S and leads to* Term*-variable $\mathcal{X}_j$ (written $present_{\mathsf{S}}(\mathcal{X}_j, p, \mathcal{X}_i)$ when S is understood), if one of the following conditions hold.*

1. *$p = \epsilon$ and $\mathcal{X}_i$ is a* Term*-variable, thus $present_{\mathsf{S}}(\mathcal{X}_i, \epsilon, \mathcal{X}_i)$.*
2. *$p = (c, k)q$ and $\mathcal{X}_i$ is a* $\mathsf{FlowTerm}$ *or* Set *variable, and $\mathcal{L}_i$ contains an expression E such that one of the following cases applies:*

[4]Note that $\mathcal{X}'$ cannot be 0, for otherwise the transitive constraints through $\mathcal{X}_i$ lead to the inconsistent constraint $c(E_1, \ldots, E_{a(c)}) \subseteq_{\mathsf{t}} 0$ and inductive systems do not contain any inconsistent constraints.

(a) $E = c(E_1, \dots, E_{a(c)})$, *where* c *is non-strict,* $E_k = \mathcal{X}_{i'}$, *and present*$_{\mathsf{S}}(\mathcal{X}_j, q, \mathcal{X}_{i'})$.

(b) $E = c(E_1, \dots, E_{a(c)})$, *where* c *is non-strict,* $E_k = \mathcal{X}_{i'} \cap M$, $q^{-1}(M) = \top_{\mathsf{t}}$, *and present*$_{\mathsf{S}}(\mathcal{X}_j, q, \mathcal{X}_{i'})$.

(c) $E = \mathcal{X}_{i'}$ *and present*$_{\mathsf{S}}(\mathcal{X}_j, p, \mathcal{X}_{i'})$.

(d) $E = \mathcal{X}_{i'} \cap M$, $p^{-1}(M) = \top_{\mathsf{t}}$, *and present*$_{\mathsf{S}}(\mathcal{X}_j, p, \mathcal{X}_{i'})$.

If a path p is syntactically present in $\mathcal{X}_i$ of inductive system S and leads to Term-variable $\mathcal{X}_j$, then the projection $p^{-1}(\sigma(\mathcal{X}_i))$ is determined in part by $\sigma(\mathcal{X}_j)$ for any solution σ

Definition 5.35 *The expansive interface variables of $\mathcal{X}_i$ w.r.t. non-strict* Term*-path p is the set*

$$\mathsf{EIF}(\mathcal{X}_i, p) = \{\mathcal{X}_j \mid present_{\mathsf{S}}(\mathcal{X}_j, p, \mathcal{X}_i) \wedge \mathsf{RHS}'_j = c(\dots)\}$$

i.e., *the set of* Term*-variables $\mathcal{X}_j$ syntactically present in $\mathcal{X}_i$ w.r.t. path p, and for which the equation $\mathcal{X}_j = \mathsf{RHS}'_j$ derived in Section 5.5.3 is of the form $\mathcal{X}_j = c(\dots)$ for some constructor c.*

The expansive interface variables $\mathsf{EIF}(\mathcal{X}_i, p)$ are the Term-variables whose join (if p is even) or meet (if p is odd) may be expansive. In our example equations (5.93) at the beginning of this subsection, the syntactic interface variables for $\mathcal{X}_3$ and path $(d, 1)$ are

$$\mathsf{EIF}(\mathcal{X}, (d, 1)) = \{\mathcal{X}_1, \mathcal{X}_2\}$$

which are defined by the equations

$$\mathcal{X}_1 = c(\mathcal{X}_3)$$
$$\mathcal{X}_2 = c(\mathcal{X}_4)$$

and it is the join of $c(\mathcal{X}_3) \sqcup_{\mathsf{t}} c(\mathcal{X}_4)$ which is expansive for some assignments of $\mathcal{X}_3$ and $\mathcal{X}_4$. Note that Term-variables $\mathcal{X}_j$ such that $\mathsf{RHS}'_j = 1$ or $\mathsf{RHS}'_j = 0$ are not considered expansive, since meets and joins involving 0 and 1 are never expansive.

Our goal is now to construct the auxiliary assignments $\mathcal{Y}_i$, such that for every path p for which $\mathsf{EIF}(\mathcal{X}_i, p)$ is non-empty, $p^{-1}(\mathcal{Y}_i) = \top_{\mathsf{t}}$ if p is even, and $p^{-1}(\mathcal{Y}_i) = \bot_{\mathsf{t}}$ if p is odd. In essence, the projection of every even Term-path p is raised to $\top_{\mathsf{t}}$ and the projection of every odd Term-path p is lowered to $\bot_{\mathsf{t}}$ through the assignment $\mathcal{Y}_i$. As a result, the combined join $\bigsqcup \mathsf{EIF}(\mathcal{X}_i, p) \sqcup p^{-1}(\mathcal{Y}_i) = \bigsqcup \mathsf{EIF}(\mathcal{X}_i, p) \sqcup \top_{\mathsf{t}} = \top_{\mathsf{t}}$ is non-expansive (for odd paths p the meet is $\bot_{\mathsf{t}}$). The complete argument needs to take the upper-bounds $\mathcal{U}_i$ of $\mathcal{X}_i$ into account, since $\mathcal{X}_i$ is defined by

$$\mathcal{X}_i = \bigsqcup \mathcal{L}_i \sqcup \mathcal{Y}_i \sqcap \bigsqcap \mathcal{U}_i$$

and the meet $\mathcal{Y}_i \sqcap \bigsqcap \mathcal{U}_i$ may eliminate some raised or lowered projections $p^{-1}(\mathcal{Y}_i)$. However, if $p^{-1}(\bigsqcap \mathcal{U}_i) \neq \top_{\mathsf{t}}$ for some even path p, then either $p^{-1}(\mathcal{Y}_i) = \bot_{\mathsf{t}}$, or the denotation of variables $\mathsf{EIF}(\mathcal{X}_i, p)$ are all equal. In either case, the meet or join of these variables is non-expansive, and similarly for odd paths p.

Before we construct the auxiliary variable assignments $\mathcal{Y}_i$, we need to know which $\mathsf{FlowTerm}$-variables $\mathcal{X}_k$ are $\top_{\mathbf{ft}}$ in all solutions. This information is easily extracted from the inductive system S by creating a map $\mathsf{Min}_{\mathbf{ft}}$, associating with each $\mathsf{FlowTerm}$-variable a unique constructor c, 0, or 1. Start with the $\mathsf{FlowTerm}$-variable $\mathcal{X}_{f_1}$ of lowest index. $\mathcal{L}_{f_1}$ has no top-level variables, so $\mathsf{Min}_{\mathbf{ft}}(\mathcal{X}_{f_1}) = 0$ if $\mathcal{L}_{f_1}$ contains only 0. If $\mathcal{L}_{f_1}$ contains only 0 and expressions $c(\dots)$, then $\mathsf{Min}_{\mathbf{ft}}(\mathcal{X}_{f_1}) = c$. Otherwise, $\mathcal{L}_{f_1}$ contains 1 or two expressions $c(\dots)$ and $c'(\dots)$ where $c \neq c'$. In that case $\mathsf{Min}_{\mathbf{ft}}(\mathcal{X}_{f_1}) = 1$. Proceed according to the ordering $o(\cdot)$ for all $\mathsf{FlowTerm}$-variables. Assume we have constructed $\mathsf{Min}_{\mathbf{ft}}(\mathcal{X}_k)$ for all $\mathsf{FlowTerm}$-variables $\mathcal{X}_k$, such that $o(\mathcal{X}_k) < o(\mathcal{X}_i)$. Construct $\mathsf{Min}_{\mathbf{ft}}(\mathcal{X}_i)$ as for $\mathcal{X}_{f_1}$, using the mapping $\mathsf{Min}_{\mathbf{ft}}$ to replace all top-level variables $\mathcal{X}_k$ appearing in $\mathcal{L}_i$ with 1 if $\mathsf{Min}_{\mathbf{ft}}(\mathcal{X}_k) = 1$, with 0 if $\mathsf{Min}_{\mathbf{ft}}(\mathcal{X}_k) = 0$, and with $c(1, \dots, 1)$ if $\mathsf{Min}_{\mathbf{ft}}(\mathcal{X}_k) = c$.

We now construct equations for $\mathcal{Y}_i$. We require an extra set of auxiliary variables $\mathcal{Y}_i^-$ to provide distinct projections for odd paths.

Definition 5.36 *Each auxiliary variable $\mathcal{Y}_i$ associated with a $\mathsf{FlowTerm}$ or Set-variable $\mathcal{X}_i$ is defined by the equation*

$$\mathcal{Y}_i = \bigsqcup_{E \in \mathcal{L}_i} \mathsf{Tl}^+(E) \tag{5.94}$$

where $\mathsf{Tl}^+(E)$ is defined on expressions of all sorts s as follows:

$$\begin{aligned}
\mathsf{Tl}^+(\mathcal{X}_j) &= \begin{cases} 1^{\mathbf{t}} & \mathcal{X}_j \textit{ is a Term-variable with equation } \mathcal{X}_j =_{\mathbf{t}} d(\dots) \\ 0^{\mathbf{t}} & \mathcal{X}_j \textit{ is a Term-variable with equation } \mathcal{X}_j =_{\mathbf{t}} 0 \textit{ or } \mathcal{X}_j =_{\mathbf{t}} 1 \\ \mathcal{Y}_j & \mathcal{X}_j \textit{ is a FlowTerm or Set-variable} \end{cases} \\
\mathsf{Tl}^+(\mathcal{X}_j \cap M) &= \mathcal{Y}_j \cap M \\
\mathsf{Tl}^+(c(E_1, \dots, E_{a(c)})) &= \begin{cases} 0^s & \textit{if } c \textit{ strict or } a(c) = 0 \\ c(\mathsf{Tl}^{z_1}(E_1), \dots, \mathsf{Tl}^{z_{a(c)}}(E_{a(c)})) & \end{cases} \\
\mathsf{Tl}^+(E) &= 0^s \qquad \textit{otherwise}
\end{aligned}$$

where $z_k = +$ if c is covariant in k, and $z_k = -$ if c contravariant in k.

Similarly, we define $\mathcal{Y}_i^-$ by the equations

$$\mathcal{Y}_i^- = \begin{cases} 1^{\mathbf{ft}} & \textit{if } \mathcal{X}_i \textit{ is of sort FlowTerm and } \mathsf{Min}_{\mathbf{ft}}(\mathcal{X}_i) = 1 \\ \bigsqcap_{E \in \mathcal{L}_i} \mathsf{Tl}^-(E) & \textit{otherwise} \end{cases} \tag{5.95}$$

where $\mathsf{Tl}^-(E)$ is defined on expressions of all sorts s as follows:

$$\begin{aligned}
\mathsf{Tl}^-(\mathcal{X}_j) &= \begin{cases} 0^{\mathbf{t}} & \mathcal{X}_j \textit{ is a Term-variable with equation } \mathcal{X}_j =_{\mathbf{t}} d(\dots) \\ 1^{\mathbf{t}} & \mathcal{X}_j \textit{ is a Term-variable with equation } \mathcal{X}_j =_{\mathbf{t}} 0 \textit{ or } \mathcal{X}_j =_{\mathbf{t}} 1 \\ \mathcal{Y}_j^- & \mathcal{X}_j \textit{ is a FlowTerm or Set-variable} \end{cases} \\
\mathsf{Tl}^-(\mathcal{X}_j \cap M) &= \mathsf{Pat}[\mathcal{Y}_j^-, M] \\
\mathsf{Tl}^-(c(E_1, \dots, E_{a(c)})) &= \begin{cases} 1^s & \textit{if } c \textit{ strict or } a(c) = 0 \\ \mathsf{Pat}[E', c(1^{\iota_1}, \dots, 1^{\iota_{a(c)}})] & \textit{if } c : \iota_1 \cdots \iota_{a(c)} \to \mathbf{s} \\ E' & \textit{if } c : \iota_1 \cdots \iota_{a(c)} \to \mathbf{ft} \end{cases} \\
\mathsf{Tl}^-(E) &= 1^s \qquad \textit{otherwise}
\end{aligned}$$

where $E' = c(\mathsf{TI}^{z_1}(E_1), \ldots, \mathsf{TI}^{z_{a(c)}}(E_{a(c)}))$ *and* $z_k = -$ *if* c *is covariant in* k*, and* $z_k = +$ *if* c *contravariant in* k*.*

The equations for $\mathcal{Y}_i$ and $\mathcal{Y}_i^-$ of Definition 5.36 define an assignment sequence $\langle \sigma_j \rangle$ constructed analogously to the inductive assignment sequence built in Definition 5.28.

Now consider the system of equations below where we assume that Term-variables have indices $1..m-1$ and FlowTerm and Set-variables have indices $m..n$, and that the indices match the variable order $o(\cdot)$ of $\mathcal{X}_1..\mathcal{X}_n$. The equations for Term-variables are the equations described in the previous subsection.

$$\begin{aligned}
\mathcal{X}_1 &=_{\mathbf{t}} \mathsf{RHS}'_1 \\
&\vdots \\
\mathcal{X}_{m-1} &=_{\mathbf{t}} \mathsf{RHS}'_{m-1} \\
\mathcal{Y}_m &= \bigsqcup_{E \in \mathcal{L}_m} \mathsf{TI}^+(E) \\
&\vdots \\
\mathcal{Y}_n &= \bigsqcup_{E \in \mathcal{L}_n} \mathsf{TI}^+(E) \\
\mathcal{Y}_m^- &= \bigsqcap_{E \in \mathcal{L}_m} \mathsf{TI}^-(E) \\
&\vdots \\
\mathcal{Y}_n^- &= \bigsqcap_{E \in \mathcal{L}_n} \mathsf{TI}^-(E) \\
\mathcal{X}_m &= \bigsqcup \mathcal{L}_m \sqcup \mathcal{Y}_m \sqcap \bigsqcap \mathcal{U}_m \\
&\vdots \\
\mathcal{X}_n &= \bigsqcup \mathcal{L}_n \sqcup \mathcal{Y}_n \sqcap \bigsqcap \mathcal{U}_n
\end{aligned} \tag{5.96}$$

Let the relative order of variables $\mathcal{Y}_i$ be given by their indices and similarly for variables $\mathcal{Y}_i^-$. Note that each top-level variable in the right-hand side of an equation $\mathcal{Y}_i \bigsqcup_{E \in \mathcal{L}_i} \mathsf{TI}^+(E)$ contains only top-level variables $\mathcal{Y}_j$ with $j < i$, and similarly for $\mathcal{Y}_i^-$. This follows from the the fact that any top-level variable $\mathcal{X}_j$ in $\mathcal{L}_i$ has index $j < i$ and that the indices of variables $\mathcal{Y}_j$ and $\mathcal{Y}_j^-$ in the construction in Definition 5.36 mirror the indices of $\mathcal{X}_j$. Thus the right-hand side of equation $\mathcal{Y}_m$ has no top-level variables. Let $E_{\mathcal{Y}_m} = \bigsqcup_{E \in \mathcal{L}_m} \mathsf{TI}^+(E)$. Now proceed in the order $\mathcal{Y}_m..\mathcal{Y}_n$. Assume we have constructed expressions $E_{\mathcal{Y}_k}$ equivalent to $\bigsqcup_{E \in \mathcal{L}_m} \mathsf{TI}^+(E)$ but without top-level variables for $k = m..j-1$. Now let $E_{\mathcal{Y}_j} =$

$\bigsqcup_{E \in \mathcal{L}_j} \mathsf{TI}^{+}(\mathsf{RHS}(E))$, where

$$\begin{aligned}
\mathsf{RHS}(\mathcal{Y}_k) &= E_{\mathcal{Y}_k} \\
\mathsf{RHS}(\mathcal{Y}_k \cap M) &= E_{\mathcal{Y}_k} \cap M \\
\mathsf{RHS}(\mathsf{Pat}[\mathcal{Y}_k, M]) &= \mathsf{Pat}[E_{\mathcal{Y}_k}, M] \\
\mathsf{RHS}(E) &= E \qquad \text{otherwise}
\end{aligned}$$

Since each top-level variable $\mathcal{Y}_k$ of $E \in \mathcal{L}_j$ has index lower than $\mathcal{Y}_j$, we have already constructed $E_{\mathcal{Y}_k}$ and the expansion $\mathsf{RHS}(E)$ is well-defined.

Similarly, the right-hand side of equation $\mathcal{Y}_m^-$ has no top-level variables and we can construct $E_{\mathcal{Y}_k^-}$ without top-level variables for $k = m..n$. Finally, the right-hand sides of $\mathcal{X}_m..\mathcal{X}_n$ can be transformed into $E_{\mathcal{X}_k}$ without top-level variables by noting that $\bigsqcup \mathcal{L}_m \sqcup \mathcal{Y}_m \sqcap \bigsqcap \mathcal{U}_m$ contains only $\mathcal{Y}_m$ as a top-level variable. Since we have computed $E_{\mathcal{Y}_m}$, we can replace $\mathcal{Y}_m$ by $E_{\mathcal{Y}_m}$ in the right-hand side of $\mathcal{X}_m$, thus obtaining $E_{\mathcal{X}_m}$ without top-level variables. In a similarly fashion, we can obtain $E_{\mathcal{X}_{m+1}}..E_{\mathcal{X}_n}$.

We have thus transformed the set of equations 5.96 into the equivalent system below, where no top-level variables appear in the right-hand sides.

$$\begin{aligned}
\mathcal{X}_1 &=_{\mathbf{t}} \mathsf{RHS}'_1 \\
&\vdots \\
\mathcal{X}_{m-1} &=_{\mathbf{t}} \mathsf{RHS}'_{m-1} \\
\mathcal{Y}_m &= E_{\mathcal{Y}_m} \\
&\vdots \\
\mathcal{Y}_n &= E_{\mathcal{Y}_n} \\
\mathcal{Y}_m^- &= E_{\mathcal{Y}_m^-} \\
&\vdots \\
\mathcal{Y}_n^- &= E_{\mathcal{Y}_n^-} \\
\mathcal{X}_m &= E_{\mathcal{X}_m} \\
&\vdots \\
\mathcal{X}_n &= E_{\mathcal{X}_n}
\end{aligned} \tag{5.97}$$

We conjecture that the assignment series generated by these equations is Cauchy and thus defines a solution of the constraints. Note that by Definition 5.36, the expressions $E_{\mathcal{Y}_m}..E_{\mathcal{Y}_n}$ and $E_{\mathcal{Y}_m^-}..E_{\mathcal{Y}_n^-}$ contain no variables besides $\mathcal{Y}_m..\mathcal{Y}_n$ and $\mathcal{Y}_m^-..\mathcal{Y}_n^-$. Since the auxiliary variables are of sorts $\mathsf{FlowTerm}$ and Set, we know that for any Term interface path p, the projections $p^{-1}(\mathcal{Y}_k)$ and $p^{-1}(\mathcal{Y}_k^-)$ result in meets and joins involving only $\top_{\mathbf{t}}$ and $\bot_{\mathbf{t}}$ and are thus non-expansive. Thus the equations for the auxiliary variables $\mathcal{Y}_i$ and $\mathcal{Y}_i^-$ are contractive.

5.5.5 Solutions For Row-Constraints

We briefly sketch how the result of Section 5.5.1 adapts to constraint systems with Row-constraints.

In light of our addition of domain constraints and the assumption that each Row-variable has a fixed kind, we refine our definition of a well-sorted variable assignment given in Section 4.5, to *well-kinded* assignments.

Definition 5.37 *A well-sorted variable assignment σ is* well-kinded, *iff for every Row-variable $\mathcal{X}$ of kind K, $\sigma(\mathcal{X})$ is of kind K, and furthermore,*

$$\begin{aligned} \mathsf{dom}(\mu[\![\mathcal{X}]\!]\sigma) &= \mathbf{L} - \sigma(\alpha_{\mathcal{X}}) \qquad \textit{if } \mathcal{X} \textit{ is closed} \\ \mathsf{dom}_{\perp}(\mu[\![\mathcal{X}]\!]\sigma) &\subseteq \mathbf{L} - \sigma(\alpha_{\mathcal{X}}) \end{aligned}$$

A well-sorted and well-kinded variable assignment σ is a solution of a system of constraints S, if for every constraint $E_1 \subseteq_s E_2$ in S, the relation $\mu[\![E_1]\!]\sigma \subseteq_s \mu[\![E_2]\!]\sigma$ is satisfied (the semantic relation for $\subseteq_{\mathsf{d}}$ is simply set-theoretic inclusion) The notion of an inductive system is extended to guarantee that well-kinded assignments for Row-variables exist.

Definition 5.38 (Inductive System with Row-Constraints) *A system S of constraints is inductive, if every constraint $E \subseteq_s E'$ in S is inductive, and furthermore for each pair of L-inductive constraint $E_1 \subseteq_s \mathcal{X}$ and R-inductive constraint $\mathcal{X} \subseteq_s E_2$ in S, S is equivalent to $S \cup \{E_1 \subseteq_s E_2\}$ according to Definition 5.23. Furthermore, S is equivalent to each of the following systems:*

1. *$S \cup \{\alpha(\mathcal{X}) \subseteq_{\mathsf{d}} \alpha(E_1)\}$ if $\mathcal{X}$ is a minimal or closed Row-variable.*
2. *$S \cup \{\alpha(\mathcal{X}) \subseteq_{\mathsf{d}} \alpha(E_2)\}$ if $\mathcal{X}$ is a closed or maximal Row-variable.*
3. *$S \cup \{\alpha(E_1) \subseteq_{\mathsf{d}} \alpha(\mathcal{X})\}$ if $\mathcal{X}$ is a closed Row-variable and E_1 is a closed Row-expression.*
4. *$S \cup \{\alpha(E_2) \subseteq_{\mathsf{d}} \alpha(\mathcal{X})\}$ if $\mathcal{X}$ is a closed Row-variable and E_2 is a closed Row-expression.*

For minimal Row-variables, Condition 1 guarantees that the labels $\alpha_{\mathcal{X}}$ are absent from the minimal domain of all lower-bounds E_1 on $\mathcal{X}$. This condition is necessary since $\mu[\![E_1]\!]\sigma \subseteq_{\mathsf{r}(s)} \sigma(\mathcal{X})$ in all solutions σ, implying that $\mathsf{dom}_{\perp}(\mu[\![E_1]\!]\sigma) \subseteq \mathsf{dom}_{\perp}(\sigma(\mathcal{X}))$. Similarly, for maximal Row-variables, Condition 2 guarantees that the labels $\alpha_{\mathcal{X}}$ are absent from the domain of all upper-bounds E_2 on $\mathcal{X}$. This condition is necessary since $\sigma(\mathcal{X}) \subseteq_{\mathsf{r}(s)} \mu[\![E_2]\!]\sigma$ in all solutions σ, implying that $\mathsf{dom}(\mu[\![E_2]\!]\sigma) \subseteq \mathsf{dom}_{\perp}(\sigma(\mathcal{X}))$. Similarly, Conditions 1–4 guarantee that the domain constraints for closed Row-variables are satisfiable.

If we choose the auxiliary assignments $\mathcal{Y}_i$ such that for all Row-variables $\mathcal{X}_i$, the minimal domain of $\mathcal{Y}_i$ satisfies the domain constraints on $\mathcal{X}_i$, *i.e.*,

$$\begin{aligned} \mathsf{dom}(\mu[\![\mathcal{Y}_i]\!]\sigma) &= \mathbf{L} - \sigma(\alpha_{\mathcal{X}_i}) \qquad \text{if } \mathcal{X}_i \text{ is closed} \\ \mathsf{dom}_{\perp}(\mu[\![\mathcal{Y}_i]\!]\sigma) &\subseteq \mathbf{L} - \sigma(\alpha_{\mathcal{X}_i}) \end{aligned}$$

we can show that the equations induced by $\mathcal{Y}_i$ on the family (5.86) satisfy the constraints up to any finite level j, analogously to the development in Section 5.5.1. Definition 5.27 is easily extended over domain-complement expressions as follows.

$$\begin{array}{rcl}
\mu_{i\leq 0}[\![\delta]\!]\langle\sigma_j\rangle & = & \mathbf{L} \\
\mu_i[\![\alpha_{\mathcal{X}}]\!]\langle\sigma_j\rangle & = & \sigma_i(\alpha_{\mathcal{X}}) \\
\mu_i[\![\alpha(\langle l : E_l\rangle_A \circ E^{\perp})]\!]\langle\sigma_j\rangle & = & (\mu_i[\![\alpha(E^{\perp})]\!]\langle\sigma_j\rangle - \{l \mid \mu_{i-1}[\![E_l]\!]\langle\sigma_j\rangle \neq \perp_s\}) \\
\mu_i[\![\alpha(\langle l : E_l\rangle_A \circ E)]\!]\langle\sigma_j\rangle & = & (\mu_i[\![\alpha(E)]\!]\langle\sigma_j\rangle - A) \quad E \neq E^{\perp} \\
\mu_i[\![\alpha(0)]\!]\langle\sigma_j\rangle & = & \mathbf{L} \\
\mu_i[\![\alpha(\langle\rangle)]\!]\langle\sigma_j\rangle & = & \mathbf{L} \\
\mu_i[\![\alpha(1)]\!]\langle\sigma_j\rangle & = & \mathbf{L} \\
\mu_i[\![\delta \cup N]\!]\langle\sigma_j\rangle & = & \mu_i[\![\delta]\!]\langle\sigma_j\rangle \cup N \\
\mu_i[\![\delta \cap N]\!]\langle\sigma_j\rangle & = & \mu_i[\![\delta]\!]\langle\sigma_j\rangle \cap N
\end{array}$$

Chapter 6

Practical Aspects of Constraint Resolution

The transformation of constraint systems into a collection of inductive systems as presented in the previous chapter involves splitting of constraint systems. Such an approach is completely impractical for an implementation since the split systems share most constraints and a large fraction of their resolution is duplicated. Furthermore, the rewrite formulation of constraint resolution serves mainly a didactic purpose but is not practical as an implementation strategy. This chapter addresses these two issues. Section 6.1 introduces conditional constraints as a way to delay splitting of constraint systems during resolution. Section 6.2 presents constraint resolution as the construction of a closed constraint-graph and characterizes when constraint-graphs are consistent. The chapter concludes with a discussion of constraint resolution and related work.

6.1 Conditional Constraints

To avoid splitting constraint systems during resolution, we encode splits using *conditional expressions.* Conditional set-expressions were originally introduced by Reynolds [74]. Aiken, Wimmers, and Lakshman [4] use conditional set-expressions to capture certain aspects of control flow in a soft-typing system. A conditional expression has the form $C \overset{l}{\Longrightarrow} E$, where E is an L-compatible conditional expression, and C is a condition. The conditions used by Aiken et al. [4] are simply L-compatible expressions. Given a solution σ, the meaning function μ is extended to conditional expressions as follows:

$$\mu[\![C \overset{l}{\Longrightarrow} E]\!]\sigma = \begin{cases} \mu[\![E]\!]\sigma & \text{if } \mu[\![C]\!]\sigma \neq \{\bot\} \\ \{\bot\} & \text{otherwise} \end{cases}$$

i.e., the meaning of $C \overset{l}{\Longrightarrow} E$ is the meaning of E if C is non-empty, and $\{\bot\}$ if C is empty. The resolution rule proposed by Aiken et al. [4] for constraints involving conditional expressions is

$$\Gamma, S \cup \{C \overset{l}{\Longrightarrow} E_1 \subseteq_s E_2\} \equiv \Gamma, S \cup \{C \subseteq_s 0\}, S \cup \{E_1 \subseteq_s E_2\} \qquad (6.1)$$

The system $S \cup \{C \overset{l}{\Longrightarrow} E_1 \subseteq_s E_2\}$ is split into two systems, one where the condition is false $S \cup \{C \subseteq_s 0\}$, and one where the inclusion holds $S \cup \{E_1 \subseteq_s E_2\}$.

Some of the resolution rules we have presented in the previous sections introduce splitting. We can reformulate these rules without splitting by introducing conditional expressions. The resolution rule for resolving constraints between strict constructor expressions (Rule 5.7)

$$\begin{aligned}\Gamma, S \cup \{c(E_1..E_n) \subseteq_s c(E'_1..E'_n)\} \equiv & \Gamma, S \cup \{E_j \subseteq_{\iota_j} E'_j \mid c : \iota_1 \cdots \iota_n \to s\},\\ & S \cup \{c(E_1..E_n) \subseteq_s 0\} \quad c \text{ strict}\end{aligned}$$

is reformulated as

$$\Gamma, S \cup \{c(E_1..E_n) \subseteq_s c(E'_1..E'_n)\} \equiv \quad \Gamma, S \cup \{c(E_1..E_n) \overset{l}{\Longrightarrow} c'(E_1..E_n) \subseteq_s c'(E'_1..E'_n)\}$$

where c' is an auxiliary constructor with the same signature as c, but non-strict in all arguments. This rule in conjunction with the rule for splitting conditional expressions (6.1) and the resolution for non-strict constructors (Rules 5.6, 5.21, and 5.27) has the same behavior as the original rule for strict constructors: either the left-side is empty (the condition is false), or the inclusions between the corresponding constructor arguments hold.

The only other rule that splits constraint systems is Rule 5.8 which has the form

$$\Gamma, S \cup \{c(E_1..E_n) \subseteq_s E\} \quad \equiv \quad \begin{cases} \Gamma & c : \iota_1 \cdots \iota_n \to s \text{ non-strict} \\ \Gamma, S \cup \{E_1 \subseteq_{\iota_1} 0\}, \ldots, S \cup \{E_n \subseteq_{\iota_n} 0\} & c \text{ strict} \end{cases}$$
$$\text{where } E \text{ is } 0 \text{ or } d(..) \text{ where } c \neq d$$

If c is strict, then one of $E_1, \ldots, E_n$ must be 0. This choice can be encoded by the following constraint

$$(E_1 \wedge E_2 \wedge \cdots \wedge E_{n-1}) \overset{l}{\Longrightarrow} E_n \subseteq_s 0$$

where the conjunction $E_1 \wedge E_2 \cdots E_{n-1}$ is a condition that is true if all of $E_1, \ldots, E_{n-1}$ are true. Note that a conjunction of two conditions $E_1 \wedge E_2$ is equivalent to the condition $p(E_1, E_2)$, where p is an arbitrary strict constructor. Conjunctions in this context are thus merely a syntactic convenience and we make free use of them. We reformulate Rule 5.8 as follows

$$\Gamma, S \cup \{c(E_1..E_n) \subseteq_s E\} \equiv \begin{cases} \Gamma & c : \iota_1 \cdots \iota_n \to s \text{ non-strict} \\ \Gamma, S \cup \{(E_1 \wedge \cdots \wedge E_{n-1}) \overset{l}{\Longrightarrow} E_n \subseteq_{\iota_n} 0\} & c \text{ strict} \end{cases} \tag{6.2}$$
$$\text{where } E \text{ is } 0 \text{ or } d(..) \text{ where } c \neq d$$

Applying Rule 6.1 n times to the system

$$S \cup \{(E_1 \wedge E_2 \wedge \cdots \wedge E_{n-1}) \overset{l}{\Longrightarrow} E_n \subseteq_s 0\}$$

produces the same collection of constraint systems as the original rules

$$S \cup \{E_1 \subseteq_{\iota_1} 0\}, \ldots, S \cup \{E_n \subseteq_{\iota_n} 0\}$$

Now the only rule that splits constraint systems is Rule 6.1 for conditional expressions.

Before proceeding, we introduce a normal form for conditions. Conditions are conjuncts of *atomic conditions* which take the form $\mathcal{X} \cap M$, i.e. a variable intersected with an M-expression. An atomic condition $\mathcal{X} \cap M$ is true for a solution σ, if $\sigma(\mathcal{X}) \cap \mu[\![M]\!]\sigma$ is *non-empty.* By non-empty, we mean a set distinct from $\{\bot\}$, since every denotation contains $\bot$. Thus, no atomic conditions involving Row-variables need to be formed, since Row-variables are always non-empty. Furthermore, if $\mathcal{X}$ is a Term or FlowTerm variable, then $M = 1$. An empty list of conjuncts represents the trivial condition *true.* A conjunct of conditions $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ is true if each individual condition is true. Note that this form of conditions is more restrictive than allowing any L-compatible expression to be a condition. In particular, L-compatible expressions as conditions can directly express disjuncts: for example assuming c is a strict constructor, $c(E_1 \cup E_2)$ is true if E_1 or E_2 is non-empty. However, by normalizing $c(E_1 \cup E_2)$ to $c(\mathcal{X})$, where $\mathcal{X}$ is a fresh variable with the associated constraint $E_1 \cup E_2 \subseteq_{\mathsf{s}} \mathcal{X}$ the same condition can still be expressed. Moreover, if $L \overset{l}{\Longrightarrow} E$ is a conditional expression and L an arbitrary L-compatible expression, then we can form an equivalent union $\bigcup_i (C_i \overset{l}{\Longrightarrow} E)$ of conditional constraints, where the C_i adhere to our restriction and without the need to introduce fresh variables. The conditions C_i are simply the disjuncts of the condition L.

Conditions are introduced every time one of our reformulated resolution rules is applied to a strict constructor. Instead of generating the conditions implicitly in these resolution rules, we can add the normalized conditions directly to the expressions involving strict constructors in the original constraints. Replace all expressions $c(E_1, \ldots, E_n)$ appearing in an L-context of the original constraints and where c is strict, by the conditional expression $(E_1 \wedge \cdots \wedge E_n) \overset{l}{\Longrightarrow} c'(E_1, \ldots, E_n)$, where c' is again a constructor with the same signature as c, but non-strict in all arguments. Replace all remaining occurrences of c by c' in the constraints. If we perform this step for all strict constructors in the original constraints, and the resolution does not produce new constructor expressions, then we obtain the same inductive systems. However, L-intersection simplification may introduce new constructor expressions, *i.e.*, when simplifying $c'(E_1, \ldots, E_n) \cap M$ to $c'(E_1 \cap M_1, \ldots, E_n \cap M_n)$ and c was originally strict, the simplified L-intersection should be transformed into the conditional expression $(C_1 \wedge \cdots \wedge C_n) \overset{l}{\Longrightarrow} c'(E_1 \cap M_1, \ldots, E_n \cap M_n)$, where C_i is the normalized condition for $E_i \cap M_i$. Note that this process terminates, since there are finitely many atomic conditions $\mathcal{X} \cap M$ that can be formed. As a final step, we can remove the cases for strict constructors from the resolution rules above and modify the rule for splitting constraints with conditional expressions to deal with conjuncts directly:

$$\begin{aligned}\Gamma, S \cup \{(C_1 \wedge \cdots \wedge C_n) \overset{l}{\Longrightarrow} E_1 \subseteq_s E_2\} \equiv & \Gamma, S \cup \{C_1 \subseteq_s 0\}, \\ & S \cup \{C_2 \wedge \cdots \wedge C_n) \overset{l}{\Longrightarrow} E_1 \subseteq_s E_2\}\end{aligned} \tag{6.3}$$

Since C_1 is atomic, the constraint $C_1 \subseteq_s 0$ does not require the use of Rule 6.2. Thus, by putting conditions into normal form and adding conditions explicitly to the original constraints, we can avoid the introduction of conditional expressions during resolution (except when intersections of strict constructors are simplified), and the only rule that splits constraint systems is the rule for conditional expressions. Special resolution rules for strict constructors are no longer needed.

We now show how to avoid splitting constraint systems by delaying the use of

Rule 6.3. The basic idea for avoiding the splitting is to transform constraints with conditional expressions into *conditional constraints* and vice-versa. Before proceeding, we extend our conditional expressions to R-compatible expressions. If R is an R-compatible expression, then $C \overset{r}{\Longrightarrow} R$ is a conditional R-compatible expression with the meaning

$$\mu[\![C \overset{r}{\Longrightarrow} E]\!]\sigma = \begin{cases} \mu[\![E]\!]\sigma & \text{if } \mu[\![C]\!]\sigma \text{ is true} \\ \top & \text{otherwise} \end{cases}$$

Conditional constraints now have the form $C \Longrightarrow (L \subseteq_s R)$. A solution σ satisfies such a conditional constraint if either $\mu[\![C]\!]\sigma$ is false, or if $\mu[\![L]\!]\sigma \subseteq_{\mathsf{s}} \mu[\![R]\!]\sigma$. The basic idea is then to use the equivalences

$$\begin{aligned} (C \overset{l}{\Longrightarrow} L) \subseteq_s R &\equiv C \Longrightarrow (L \subseteq_s R) \\ &\equiv L \subseteq_s (C \overset{r}{\Longrightarrow} R) \end{aligned}$$

to transform constraints with conditional expressions into conditional constraints, and vice versa. The above equivalences are trivial: suppose C is true, then all three forms are equivalent to $L \subseteq_s R$. If C is false, then we have

$$\begin{aligned} (C \overset{l}{\Longrightarrow} L) \subseteq_s R &\equiv 0 \subseteq_s R \\ &\equiv \text{“no constraint”} \\ &\equiv L \subseteq_s 1 \\ &\equiv L \subseteq_s (C \overset{r}{\Longrightarrow} R) \end{aligned}$$

Thus a conditional constraint $\mathsf{true} \Longrightarrow (L \subseteq_s R)$ is equivalent to $L \subseteq_s R$. These equivalences motivate the replacement of Rule 6.1 with the two transformation rules depicted in Figure 6.1.

The new transformations introduce conditional constraints and we need the structural rewrite rule of Figure 6.2 to deal with them. Intuitively, given the constraint $C \Longrightarrow (L \subseteq_s R)$, we strip off condition C and solve the non-conditional constraint in the standard way. If this constraint is equivalent to $\{C_1 \Longrightarrow (L_1 \subseteq_{s_1} R_n), \ldots, C_n \Longrightarrow (L_n \subseteq_{s_n} R_n)\}$, then we add the condition C back onto each individual constraint, yielding $\{C \wedge C_1 \Longrightarrow (L_1 \subseteq_{s_1} R_1), \ldots, C \wedge C_n \Longrightarrow (L_n \subseteq_{s_n} R_n)\}$. It is immediate that this rule is sound, but not complete in the case where $L \subseteq_s R$ is inconsistent. We will address this case shortly.

The algorithm for transforming constraints into inductive form can now be adapted to constraints with conditional expressions. Instead of transforming constraints into inductive constraints, we transform them to *conditional inductive constraints*, i.e. constraints of the form $C \Longrightarrow (L \subseteq_s R)$ where $L \subseteq_s R$ is inductive. To solve conditional constraints, repeat the following two steps until either an inconsistency is found, or all constraints are conditional inductive.

- For any constraint that is not conditional inductive, apply one of the equivalences in Figure 6.1, 6.2, or the standard resolution rules.
- For any pair of conditional inductive constraints $C_1 \Longrightarrow (L \subseteq_s \mathcal{X})$ and $C_2 \Longrightarrow (\mathcal{X} \subseteq_s R)$, add the transitive constraint $C_1 \wedge C_2 \Longrightarrow (L \subseteq_s R)$.

$$\begin{array}{rcl} (C \overset{l}{\Longrightarrow} L) \subseteq_s R) & \equiv & C \Longrightarrow (L \subseteq_s R) \\ L \subseteq_s (C \overset{r}{\Longrightarrow} R) & \equiv & C \Longrightarrow (L \subseteq_s R) \end{array}$$

Figure 6.1: Transforming constraints with conditional expressions into conditional constraints

$$\frac{\{L \subseteq_s R\} \equiv \{C_1 \Longrightarrow (L_1 \subseteq_{s_1} R_1), \ldots, C_n \Longrightarrow (L_n \subseteq_{s_n} R_n)\}}{S \cup \{C \Longrightarrow (L \subseteq_s R)\} \equiv S \cup \{C \wedge C_1 \Longrightarrow (L_1 \subseteq_{s_1} R_1), \ldots, C \wedge C_n \Longrightarrow (L_n \subseteq_{s_n} R_n)\}}$$

Figure 6.2: Structural rewrite rule for conditional constraints

$$\frac{\{L \subseteq_s R\} \equiv \text{Inconsistent}}{S \cup \{(\bigwedge_{i=0\ldots n} C_i) \Longrightarrow (L \subseteq_s R)\} \equiv S \cup \{(\bigwedge_{i=1..n} C_i) \Longrightarrow (C_0 \subseteq_s 0)\}}$$

Figure 6.3: Structural rewrite rule for inconsistent constraint

The transitive constraints combine the conditions of the two constraints involved. The justification behind this rule is as follows: If both conditions are true, then we have an ordinary transitive constraint. Otherwise, the transitive constraint need not be satisfied.

As in the original algorithm, the final form of a solved system of constraints can be expressed as lower and upper bounds on variables $\mathcal{L}_i \subseteq_s \mathcal{X}_i \subseteq_s \mathcal{U}_i$, where $\mathcal{L}_i$ and $\mathcal{U}_i$ are

$$\begin{array}{l} \mathcal{L}_i = \{C \overset{l}{\Longrightarrow} L \mid C \Longrightarrow (L \subseteq_s \mathcal{X}_i) \text{ is L-inductive}\} \\ \mathcal{U}_i = \{C \overset{r}{\Longrightarrow} R \mid C \Longrightarrow (\mathcal{X}_i \subseteq_s R) \text{ is R-inductive}\} \end{array}$$

Thus in the final result, no conditional constraints remain, only conditional upper and lower bounds on variables.

As described, the algorithm gives up as soon as an inconsistent constraint is found, no matter whether the inconsistent constraint was conditional or not. E.g. any system containing the constraint $C \Longrightarrow (c \subseteq_s d)$ is inconsistent under this algorithm, provided constructor c is distinct from d. This approach does not find all solutions, for in the above example, there could still be solutions where C is *false* and the constraint satisfied.

The form of conditions is simple enough so that we can encode the fact that a condition is false with appropriate set-constraints. If C is of the form $\bigwedge_{i=1\ldots n} \mathcal{X}_i \cap M_i$, then we can express the fact that C is false with the constraint $C' \Longrightarrow (\mathcal{X}_j \cap M_j \subseteq_s 0)$, where $\mathcal{X}_j \cap M_j$ is an arbitrary atomic condition of C, and $C' = \bigwedge_{i=1\ldots j-1,j+1\ldots n} \mathcal{X}_i \cap M_i$. In other words, either $\mathcal{X}_j \cap M_j$ is empty implying that C is false, or C' is false. If C is *true*, then the constraints have indeed no solution. Figure 6.3 contains the structural rewrite rule to handle inconsistent systems.

The conditional inductive system is obtained without applying Rule 6.1 for splitting systems with conditional expressions. As a result, we can transform a constraint system into a single conditional inductive constraint system. If the original inductive systems are desired, the splitting rule can be applied to this system. Thus we essentially delay the splitting until the constraints are almost in inductive form. Applying the splitting rule only introduces additional constraints of the form $E \subseteq_s 0$.

6.2 Graph Formulation

This section describes a graph-based representation of constraints and an algorithm to transform a constraint system into a fully-closed *constraint graph*. We also characterize when fully-closed constraint graphs are consistent, in which case, they correspond to a conditional inductive system. Inconsistent constraint graphs and their associated constraints have no solutions.

Even though the previous section presented constraint resolution as a rewrite system, constraint resolution is usually presented as a graph closure. The graph formulation presented here makes it convenient to study different constraint resolution algorithms based on graph closure.

In principle, given a set of mixed constraints S, the fully-closed constraint graph corresponding to S is a collection of disjoint graphs G^s, one for each sort s. To simplify the presentation here, we only show the graph construction for Set-constraints. A restriction of this construction can be applied to the Term and FlowTerm-constraints. For Row-constraints, the graph formulation is similar but sets of labels are used in place of M-expressions.

Definition 6.1 (Set-Constraint Graph) *A constraint graph $G = (V, A)$ consists of a set of nodes V which are Set-expressions E with the restrictions that E is of the form*

$$E ::= \mathcal{X} \mid c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \mid 1 \mid 0 \mid \neg\{c_1, \ldots, c_n\}$$

i.e., *V contains no unions or intersections, and the arguments to constructors are variables. An edge (or arrow) of A is a quadruple of $V \times V \times M \times C$, where M is an M-expression, and C a normalized condition. We write $E_1 \xrightarrow{M,C} E_2$ for the edge (E_1, E_2, M, C). A non-variable node is called a* source *if it occurs to the left of an arrow, and is called a* sink*, if it occurs to the right of an arrow.*

Definition 6.2 (Graph Path) *We say that there exists a path of length n in G from E to E' under M-expression M and condition C, if there exists a sequence of edges in G*

$$E \xrightarrow{M_1,C_1} \mathcal{X}_1 \xrightarrow{M_2,C_2} \mathcal{X}_2 \cdots \mathcal{X}_{n-1} \xrightarrow{M_n,C_n} E'$$

and

$$C = \bigwedge_{i=1..n} C_i$$
$$M = \bigcap_{i=1..n} M_i$$

If the intermediate nodes are not of interest, we refer to such a path using the notation $E \xrightarrow{M,C}{}^{*} E'$.

We now show how to transform a system of constraints S into its *initial* constraint graph. The first step is to normalize constructor expressions $c(E_1, \dots, E_n)$ where some E_i are not variables. Replace each expression $c(E_1, \dots, E_n)$ in the constraints with $c(\mathcal{X}_1, \dots, \mathcal{X}_n)$ where $\mathcal{X}_1..\mathcal{X}_n$ are fresh variables, and add the constraints $E_i \subseteq_{\iota_i} \mathcal{X}_i$ to S if the occurrence $c(E_1, \dots, E_n)$ being replaced appears in an L-context in the constraint. Otherwise (the occurrence appears in an R-context), add the constraints $\mathcal{X}_i \subseteq_{\iota_i} E_i$ to S.

Constraints $E_1 \subseteq_s E_2$ of S where E_1 and E_2 adhere to the node-restrictions in the definition above can be directly added to the graph, by adding E_1 and E_2 and their sub-expressions to V, and adding the edge $E_1 \xrightarrow{\mathsf{1,true}} E_2$ to A. The remaining constraints are of the forms

$$\begin{aligned}
E_1 \sqcup E_2 &\subseteq_s E_3 \\
E_1 &\subseteq_s E_2 \sqcap E_3 \\
\mathcal{X} \cap M &\subseteq_s E \\
E_1 &\subseteq_s \mathsf{Pat}[E_2, M] \\
C \overset{l}{\Longrightarrow} E_1 &\subseteq_s E_2 \\
E_1 &\subseteq_s C \overset{r}{\Longrightarrow} E_2
\end{aligned}$$

We show how to normalize each of these in turn:

- [L-union] If $E_1 \sqcup E_2 \subseteq_s E_3$ is in S, remove it and add $E_1 \subseteq_s E_3$ and $E_2 \subseteq_s E_3$ to S.

- [R-inter] If $E_1 \subseteq_s E_2 \sqcap E_3$ is in S, remove it and add $E_1 \subseteq_s E_2$ and $E_1 \subseteq_s E_3$ to S.

- [L-inter] If $\mathcal{X} \cap M \subseteq_s E$ is in S, and E adheres to the node-restriction, then the constraint is normalized. Otherwise, apply one of the other rules matching E.

- [R-Pat] If $E_1 \subseteq_s \mathsf{Pat}[E_2, M]$ is in S, and E_1 and E_2 adhere to the node-restriction, then the constraint is normalized.

  Otherwise, apply the other rules to normalize E_1. If E_1 adheres, but E_2 does not adhere to the node-restriction, remove the constraint from S and add the constraint $E_1 \cap M \subseteq_s E_2$ to the constraints, normalizing the intersection $E_1 \cap M$ if necessary.

- [L-cond] If $C \overset{l}{\Longrightarrow} E_1 \subseteq_s E_2$ is in S and E_1 or E_2 do not adhere to the node-restriction, remove the constraint from S and add $E_1 \subseteq_s E_2$ to a new system S' and normalize it. Then for each normal constraint $C_i \implies (E_i \subseteq_s E_i')$ in S', add the conditional constraint $C \wedge C_i \implies (E_i \subseteq_s E_i')$ back to S.

- [R-cond] If $E_1 \subseteq_s C \overset{r}{\Longrightarrow} E_2$ is in S and E_1 or E_2 do not adhere to the node-restriction, remove the constraint from S and add $E_1 \subseteq_s E_2$ to a new system S' and normalize it. Then for each normal constraint $C_i \implies (E_i \subseteq_s E_i')$ in S', add the conditional constraint $C \wedge C_i \implies (E_i \subseteq_s E_i')$ back to S.

After normalization, S contains constraints of the form

$$\begin{aligned} C &\implies (\mathcal{X} \cap M \subseteq_s E_2) \\ C &\implies (E_1 \subseteq_s \mathsf{Pat}[E_2, M]) \\ C &\implies (E_1 \subseteq_s E_2) \end{aligned}$$

where each E_1 and E_2 adhere to the node-restriction, and C is a conjunction of atomic conditions. (Recall that the empty conjunct is equivalent to true). These constraints are then added to G using the following three rules

- [L-inter] If $C \implies (\mathcal{X} \cap M \subseteq_s E)$ is in S, remove it and add $\mathcal{X}$, E, and their sub-expressions to V, and add the edge $\mathcal{X} \xrightarrow{M,C} E$ to A.
- [R-Pat] If $C \implies (E_1 \subseteq_s \mathsf{Pat}[E_2, M])$ is in S, remove it and add E_1, E_2, and their sub-expressions to V and add the edge $E_1 \xrightarrow{M,C} E_2$ to A.
- [Cond] If $C \implies (E_1 \subseteq_s E_2)$ is in S and E_1 and E_2 adhere to the node-restriction, remove the constraint and add E_1, E_2, and their sub-expressions to V, and add the edge $E_1 \xrightarrow{1,C} E_2$ to A.

We now present the rules for closing a constraint graph under transitivity and structural constraints.

Transitive Closure Rule If $E_1 \xrightarrow{M_1,C_1} \mathcal{X}$ and $\mathcal{X} \xrightarrow{M_2,C_2} E_2$ are edges in G, add the transitive edge $E_1 \xrightarrow{M_1 \cap M_2, C_1 \wedge C_2} E_2$ to the edges of G.

The structural closure rules cover all combinations of source-sink pairs.

Structural Closure Rules If $E_1 \xrightarrow{M,C} E_2$ is an edge of G between a source E_1 and a sink E_2, apply the closure rule that applies to E_1, E_2 below.

- $c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \xrightarrow{M,C} c(\mathcal{Y}_1, \ldots, \mathcal{Y}_n)$ and $M \cap c(\top_{\iota_1}, \ldots, \top_{\iota_n}) = \bigcup_j c(M_{j1}, \ldots, M_{jn})$ where each $c(M_{j1}, \ldots, M_{jn}) \neq 0$, then add edges $\mathcal{X}_i \xrightarrow{M_{ji}, C \wedge C'_j} \mathcal{Y}_i$ for all j, where

$$C'_j = \begin{cases} \bigwedge_i \mathcal{X}_i \cap M_{ji} & \text{if } c \text{ strict} \\ \mathsf{true} & \text{otherwise} \end{cases}$$

- $c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \xrightarrow{M,C} \neg\{c_1, \ldots, c_n\}$ and $c \in \{c_1, \ldots, c_n\}$, then add the edge

$$c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \xrightarrow{M,C} 0$$

- $c(\mathcal{X}_1, \ldots, \mathcal{X}_n) \xrightarrow{M,C} E_2$ where $E_2 = 0$, or $E_2 = d(\ldots)$ with $c \neq d$.

1. if $n > 0$, c strict, and $M \cap c(\top_{\iota 1}, \dots, \top_{\iota n}) = \bigcup_j c(M_{j1}, \dots, M_{jn})$, where each $c(M_{j1}, \dots, M_{jn}) \neq 0$, then add edges $1 \xrightarrow{1, C \wedge C'_j} 0$ for all j, where

$$C'_j = \bigwedge_i \mathcal{X}_i \cap M_{ji}$$

2. otherwise add the edge $1 \xrightarrow{M \cap c(\top_{\iota 1}, \dots, \top_{\iota n}), C} 0$.

- $\neg\{c_1, \dots, c_n\} \xrightarrow{M,C} E_2$, and E_2 is $c(\dots)$, 0, or $\neg\{d_1, \dots, d_n\}$, then add the edge

$$1 \xrightarrow{\neg\{c_1, \dots, c_n\} \cap M, C} E_2$$

- $1 \xrightarrow{M,C} c(\mathcal{X}_1, \dots, \mathcal{X}_n)$ add the edge $1 \xrightarrow{M \cap \neg\{c\}, C} 0$, and if $M \cap c(\top_{\iota 1}, \dots, \top_{\iota n}) = \bigcup_j c(M_{j1}, \dots, M_{jn})$ where each $c(M_{j1}, \dots, M_{jn}) \neq 0$, then also add edges

$$1 \xrightarrow{M_{ji}, C} \mathcal{X}_i$$

  for all j.

- $1 \xrightarrow{M,C} \neg\{c_1, \dots, c_n\}$, and $n \neq 0$, add the edge $1 \xrightarrow{M \cap M', C} 0$, where

$$M' = \bigcup_{c \in \{c_1, \dots, c_n\}} c(\top_{\iota 1}, \dots, \top_{\iota n})$$

- $1 \xrightarrow{M,C} 0$, $M \neq 0$, and $C = C_1 \wedge \dots \wedge C_n$, $n > 0$, then add the edge

$$\mathcal{X}_1 \xrightarrow{M_1, C'} 0$$

  where

$$C_1 = \mathcal{X}_1 \cap M_1$$
$$C' = C_2 \wedge \dots \wedge C_n$$

Definition 6.3 (Graph Consistency) *A constraint graph is consistent, if its closure does not contain any edge of the form* $1 \xrightarrow{M, \mathsf{true}} 0$*, where* $M \neq 0$.

If a graph is consistent, then there exists an equivalent conditional inductive system S, obtained as follows:

- if $\mathcal{X} \xrightarrow{M,C} E$ is in G and E is a sink, then add the inductive constraint $\mathcal{X} \subseteq_s C \overset{r}{\Longrightarrow} \mathsf{Pat}[E, M,]$ to S.

- if $E \xrightarrow{M,C} \mathcal{X}$ is in G and E is a source, then add the inductive constraint $C \overset{l}{\Longrightarrow} (E \cap M) \subseteq_s \mathcal{X}$ to S.

- if $\mathcal{X} \xrightarrow{M,C} \mathcal{Y}$ is in G and $o(\mathcal{X}) > o(\mathcal{Y})$, then add the inductive constraint $\mathcal{X} \subseteq_s C \overset{r}{\Longrightarrow} \mathsf{Pat}[\mathcal{Y}, M,]$ to S.
- if $\mathcal{X} \xrightarrow{M,C} \mathcal{Y}$ is in G and $o(\mathcal{Y}) > o(\mathcal{X})$, then add the inductive constraint $C \overset{l}{\Longrightarrow} (\mathcal{X} \cap M) \subseteq_s \mathcal{Y}$ to S.

The resulting system S is inductive, since all constraints above are inductive, and the transitive closure rule of the graph adds all edges added by Algorithm 5.25.

On the other hand, if the closure of the constraint graph G corresponding to the normalized constraint set S is inconsistent, then applying Algorithm 5.25 to S also results in no inductive systems. To see this, observe that the structural graph closure rules correspond to the resolution rules given earlier, and that Algorithm 5.25 has the following property.

Property 6.4 *If E and E' are expressions without top-level variables and there is a path*

$$E \xrightarrow{M_1,C_1} \mathcal{X}_1 \xrightarrow{M_2,C_2} \mathcal{X}_2 \cdots \mathcal{X}_{n-1} \xrightarrow{M_n,C_n} E'$$

in the initial graph G corresponding to the normalized constraint system S, then Algorithm 5.25 applied to S generates the constraint $C \Longrightarrow (E \cap M) \subseteq_s E'$, where

$$C = \bigwedge_{i=1..n} C_i$$
$$M = \bigcap_{i=1..n} M_i$$

Proof: If $n = 1$ the result is immediate. Note by edge $E \xrightarrow{M_1,C_1} \mathcal{X}_1$ we have that S contains the conditional inductive constraint constraint $C_1 \Longrightarrow E \cap M_1 \subseteq_s \mathcal{X}_1$. Call this constraint the *source constraint.* Similarly by edge $\mathcal{X}_{n-1} \xrightarrow{M_n,C_n} E'$ we have that S contains the conditional inductive constraint $C_n \Longrightarrow \mathcal{X}_{n-1} \subseteq_s \mathsf{Pat}[E', M_n]$. Call this edge the *sink constraint.* If $n = 2$, then $\mathcal{X}_1 = \mathcal{X}_{n-1}$, and Algorithm 5.25 adds the transitive constraint $C_1 \wedge C_2 \Longrightarrow E \cap M_1 \subseteq_s \mathsf{Pat}[E', M_2]$, which is equivalent to the sought constraint. We now proceed by induction on the length of the path. Assume the property holds for paths of length k. We show that it also holds for paths of length $k + 1$. Suppose $n = k + 1$. There are three cases to consider.

1. If the index $o(\mathcal{X}_1)$ is greater than the index $o(\mathcal{X}_2)$, then by edge $\mathcal{X}_1 \xrightarrow{M_2,C_2} \mathcal{X}_2$ we know that S contains the conditional inductive constraint $C_2 \Longrightarrow \mathcal{X}_1 \subseteq_s \mathsf{Pat}[\mathcal{X}_2, M_2]$. This constraint together with the source constraint produces the transitive constraint $C_1 \wedge C_2 \Longrightarrow E \cap M_1 \subseteq_s \mathsf{Pat}[\mathcal{X}_2, M_2]$, which is transformed to the conditional inductive constraint $C_1 \wedge C_2 \Longrightarrow E \cap (M_1 \cap M_2) \subseteq_s \mathcal{X}_2$. But now there exists a path $E \xrightarrow{M,C}{}^* E'$ of length k and by induction the constraint $C \Longrightarrow E \cap M \subseteq_s E'$ is added.
2. If the index $o(\mathcal{X}_{n-1})$ is greater than the index $o(\mathcal{X}_{n-2})$, then by edge $\mathcal{X}_{n-2} \xrightarrow{M_{n-1},C_{n-1}} \mathcal{X}_{n-1}$ we know that S contains the conditional inductive constraint $C_{n-1} \Longrightarrow \mathcal{X}_{n-2} \cap M_{n-1} \subseteq_s \mathcal{X}_{n-1}$. This constraint together with the sink constraint produces the transitive constraint $C_{n-1} \wedge C_n \Longrightarrow \mathcal{X}_{n-2} \cap M_{n-1} \subseteq_s \mathsf{Pat}[E', M_n]$, which is transformed

$$\begin{tikzcd} \mathsf{list} \arrow[r] \arrow[d, dash] & \mathsf{list} \arrow[d, dash] \\ \mathcal{X} \arrow[r, dashed] & \mathcal{Y} \end{tikzcd}$$

Figure 6.4: Unnecessary edges in full graph closure

to the conditional inductive constraint $C_{n-1} \wedge C_n \implies \mathcal{X} \subseteq_s \mathsf{Pat}[E', M_1 \cap M_2]$. But now there exists a path $E \overset{M,C}{\longrightarrow}{}^* E'$ of length k and by induction the constraint $C \implies E \cap M \subseteq_s E'$ is added.

3. Otherwise $o(\mathcal{X}_2) > o(\mathcal{X}_1)$ and $o(\mathcal{X}_{n-2}) > o(\mathcal{X}_{n-1})$. Thus there exists a variable with maximum index among $\mathcal{X}_2..\mathcal{X}_{n-2}$. Assume this variable is $\mathcal{X}_j$. Then the edges $\mathcal{X}_{j-1} \xrightarrow{M_j,C_j} \mathcal{X}_j$ and $\mathcal{X}_j \xrightarrow{M_{j+1},C_{j+1}} \mathcal{X}_{j+1}$ correspond to the initial conditional inductive constraints $C_j \overset{l}{\implies} \mathcal{X}_{j-1} \cap M_j \subseteq_s \mathcal{X}_j$ and $\mathcal{X}_j \subseteq_s C_{j+1} \overset{r}{\implies} \mathsf{Pat}[\mathcal{X}_{j+1}, M_{j+1}]$. Thus Algorithm 5.25 adds the transitive constraint $(C_1 \wedge C_2) \implies \mathcal{X}_{j-1} \cap M_{j-1} \subseteq_s \mathsf{Pat}[\mathcal{X}_{j+1}, M_{j+1}]$, which is transformed into the conditional inductive constraint $(C_1 \wedge C_2) \implies (\mathcal{X}_{j-1} \cap (M_{j-1} \cap M_{j+1}) \subseteq_s \mathcal{X}_{j+1})$ if $o(\mathcal{X}_{j+1}) > o(\mathcal{X}_{j-1})$, or into the conditional inductive constraint $(C_1 \wedge C_2) \implies \mathcal{X}_{j-1} \subseteq_s \mathsf{Pat}[\mathcal{X}_{j+1}, M_{j-1} \cap M_{j+1}]$ if $o(\mathcal{X}_{j-1}) > o(\mathcal{X}_{j+1})$. In either case there is now a path $E \overset{M,C}{\longrightarrow}{}^* E'$ of length k and by induction the constraint $C \implies E \cap M \subseteq_s E'$ is added.

□

The full closure of the constraint graph contains many edges that are not really necessary to decide whether the graph is consistent or not. Edges between sources and sinks that are not inconsistent and for which the corresponding structural closure rule has been applied can be removed from the graph. Consider the example in Figure 6.4. The graph represents the closure of the constraint $\mathsf{list}(\mathcal{X}) \subseteq_{\mathsf{s}} \mathsf{list}(\mathcal{Y})$, where the vertical edges relate constructors with their arguments and horizontal edges are inclusion edges. Applying the structural rule to to the inclusion edge between the two list constructor nodes adds the dashed inclusion edge between $\mathcal{X}$ and $\mathcal{Y}$. After this step, the inclusion edge between the two list constructors is never needed again and can be deleted.

This optimization is easy to implement. Simply apply the structural rules immediately whenever a new source-sink edge is to be added. Furthermore, the transitive closure rule can be restricted so as to add only edges between a source E_1 and a sink E_2, if there is a path $E_1 \overset{M,C}{\longrightarrow}{}^* E_2$ in the graph. In fact the transitive edge needs only be added, if the M is non-empty and the condition C is true in all solutions. Transitive edges where one end-point is a variable can be avoided since they do not trigger any structural rules and do not make the graph inconsistent. One can thus formulate a minimal transitive closure rule:

Minimal Transitive Closure Rule If there exists a path $E \overset{M,C}{\longrightarrow}{}^* E'$ between a source E and a sink E' in G, and $M \neq 0$, and C is true in all solutions of the constraints, then add the edge $E \xrightarrow{M,C} E'$ to G.

The minimal transitive closure rule does not yield a practical algorithm however, since whenever a new edge is added to the graph by the structural closure rules, all paths would need to be reexamined. Furthermore, deciding whether a condition is true in all solutions requires information about paths from sources to variables, and this information may be expensive to compute and maintain in itself.

The distinguishing feature between different algorithms for computing constraint graph consistency is thus the transitive closure rule. Practical algorithms usually use a *local* rule, where only a small constant number of edges need to be considered, instead of paths of arbitrary length. We have already seen one alternative transitive closure rule that is local, namely the one applied by the algorithm computing inductive systems.

Inductive Transitive Closure Rule If G contains two edges $E_1 \xrightarrow{M_1,C_1} \mathcal{X}$ and $\mathcal{X} \xrightarrow{M_2,C_2} E_2$ such that any top-level variables of E_1 and E_2 have indices lower than $o(\mathcal{X})$, then add the transitive edge $E_1 \xrightarrow{M_1 \cap M_2, C_1 \wedge C_2} E_2$ to the graph.

It is instructive to examine where exactly sources and sinks meet on a path $E \xrightarrow{M,C}{}^{*} E'$ using the above rule. From the proof of Property 6.4 we know that the closure rule adds transitive edges between variables so as to guarantee that there exists a path

$$E \xrightarrow{M_1,C_1} \mathcal{X}_1 \xrightarrow{M_2,C_2} \mathcal{X}_2 \cdots \mathcal{X}_{n-1} \xrightarrow{M_n,C_n} E'$$

where no variable $\mathcal{X}_j$ among $\mathcal{X}_2..\mathcal{X}_{n-2}$ has index larger than its neighbors, *i.e.*, $o(\mathcal{X}_j) \not> o(\mathcal{X}_{j-1})$ and $o(\mathcal{X}_j) \not> o(\mathcal{X}_{j+1})$. Consider thus the sequence of indices $o(\mathcal{X}_1), o(\mathcal{X}_2), \ldots, o(\mathcal{X}_n)$ and the variable $\mathcal{X}_j$ with minimal index. Then the sequence $o(\mathcal{X}_1), \ldots, o(\mathcal{X}_j)$ is strictly decreasing, and the sequence $o(\mathcal{X}_j), \ldots, o(\mathcal{X}_n)$ is strictly increasing. As a result, the edges $\mathcal{X}_{i-1} \xrightarrow{M_i,C_i} \mathcal{X}_i$ with $i \leq j$ are R-inductive and the source E is propagated forward along this path to each $\mathcal{X}_i$ up to the minimal indexed variable $\mathcal{X}_j$. Similarly, the edges $\mathcal{X}_i \xrightarrow{M_{i+1},C_{i+1}} \mathcal{X}_{i+1}$ with $i \geq j$ are R-inductive and the sink E' is propagated backward along this path to each $\mathcal{X}_i$ down to the minimal indexed variable $\mathcal{X}_j$. Thus the inductive transitive closure rule produces the two inductive constraints

$$\begin{aligned}(C_1 \wedge \cdots \wedge C_j) &\implies E \cap (M_1 \cap \cdots \cap M_j) \subseteq_s \mathcal{X}_j \\ (C_{j+1} \wedge \cdots \wedge C_n) &\implies \mathcal{X}_j \subseteq_s \mathsf{Pat}[E', M_{j+1} \cap \cdots \cap M_n]\end{aligned}$$

from which a final application of the rule produces the edge $E \xrightarrow{M,C} E'$.

The inductive closure rule is rather non-standard in that sources move forwards and sinks backwards along certain, but not all paths. Most published algorithms for this kind of graph closure use something akin to the following rule, which we thus call the *standard* transitive closure rule.

Standard Transitive Closure Rule If G contains two edges $E_1 \xrightarrow{M_1,C_1} \mathcal{X}$ and $\mathcal{X} \xrightarrow{M_2,C_2} E_2$ such that E_1 is a source, then add the transitive edge $E_1 \xrightarrow{M_1 \cap M_2, C_1 \wedge C_2} E_2$ to the graph.

This standard closure works rather differently from the inductive closure rule. The rule uniquely propagates sources forward along paths until they meet up with a sink. One could imagine a dual rule that propagates sinks backwards until they meet a source. However, the standard rule has the advantage of computing an explicit form of the *transitive lower bound* for each variable which we discuss in the following section. Empirical results on the practical tradeoffs of using the inductive transitive closure rule over the standard transitive closure rule are studied in more detail in Chapter 9.

6.2.1 Transitive Lower Bound

In many analyses, it is often desirable to answer queries such as, "does $\mathcal{X}$ contain an ideal $\phi_c(I_1, \dots, I_n) \neq \{\bot\}$ for some I_j in all solutions of the constraints?" To answer such queries easily, the *transitive lower bound* of $\mathcal{X}$ (written $\mathsf{TLB}(\mathcal{X})$) can be computed.

Definition 6.5 (Transitive Lower Bound) *The transitive lower bound of $\mathcal{X}$, consists of the union of all non-empty expressions in the intersection $E \cap M$ where $E \xrightarrow{M,C}{}^{*} \mathcal{X}$ is a path in G, and C is true in all solutions. Formally,*

$$\mathsf{TLB}(\mathcal{X}) = \bigcup \{\mathsf{TLB}(C \overset{l}{\Longrightarrow} E \cap M) \mid E \xrightarrow{M,C}{}^{*} \mathcal{X}\}$$

where $E = c(\mathcal{X}_1, \dots, \mathcal{X}_n)$, $M \cap c(\top_{\iota_1}, \dots, \top_{\iota_n}) = \bigcup_j c(M_{j1}, \dots, M_{jn})$, and

$$\mathsf{TLB}(C \overset{l}{\Longrightarrow} E \cap M) = \{c(\mathcal{X}_1 \cap M_{j1}, .., \mathcal{X}_n \cap M_{jn}) \mid C \wedge C_j \textit{ is true in all solutions}\}$$

$$C_j = \begin{cases} \mathsf{true} & c \textit{ non-strict} \\ \mathcal{X}_1 \cap M_{j1} \wedge \cdots \wedge \mathcal{X}_n \cap M_{jn} & \textit{otherwise} \end{cases}$$

Computing $\mathsf{TLB}(\mathcal{X})$ thus involves deciding which conditions are true in all solutions. An atomic condition $\mathcal{Y} \cap M$ is true in all solutions, if $\mathsf{TLB}(\mathcal{Y}) \cap M$ is non-empty. Thus, computing the transitive lower bound and resolving conditions that are true in all solutions depend on each other. We tackle both problems by first considering the *conditional transitive lower bound* of $\mathcal{X}$ ($\mathsf{CTLB}(\mathcal{X})$) which essentially consists of all conditional expressions $C \overset{l}{\Longrightarrow} E$ where E is a candidate for $\mathsf{TLB}(\mathcal{X})$ if C is true in all solutions. The transitive lower bound $\mathsf{TLB}(\mathcal{X})$ is related to $\mathsf{CTLB}(\mathcal{X})$ by

$$\begin{aligned} c(E_1, \dots, E_n) \in \mathsf{TLB}(\mathcal{X}) \iff\ & C \overset{l}{\Longrightarrow} c(E_1, \dots, E_n) \in \mathsf{CTLB}(\mathcal{X}) \ \wedge \\ & C \text{ true in all solutions} \end{aligned}$$

Definition 6.6 (Conditional Transitive Lower Bound) *The conditional transitive lower-bound of $\mathcal{X}$ consists of the union of all conditional expressions in $\mathsf{CTLB}(C \overset{l}{\Longrightarrow} E \cap M)$ where E is a source and there is a path $E \xrightarrow{M,C}{}^{*} \mathcal{X}$ in G. Formally,*

$$\mathsf{CTLB}(\mathcal{X}) = \bigcup \{\mathsf{CTLB}(C \overset{l}{\Longrightarrow} E \cap M) \mid E \xrightarrow{M,C}{}^{*} \mathcal{X}\}$$

$$\mathsf{CTLB}(C \overset{l}{\Longrightarrow} E \cap M) = \bigcup_j \left(C \wedge C_j \overset{l}{\Longrightarrow} c(\mathcal{X}_1 \cap M_{j1}, .., \mathcal{X}_n \cap M_{jn})\right)$$

where $E = c(\mathcal{X}_1, \dots, \mathcal{X}_n)$, $M \cap c(\top_{\iota_1}, \dots, \top_{\iota_n}) = \bigcup_j c(M_{j1}, \dots, M_{jn})$, and

$$C_j = \begin{cases} \mathsf{true} & c \textit{ non-strict} \\ \mathcal{X}_1 \cap M_{j1} \wedge \cdots \wedge \mathcal{X}_n \cap M_{jn} & \textit{otherwise} \end{cases}$$

The next section describes how to identify conditions that are true in all solutions of the constraints using the conditional transitive lower bounds. In the remainder of this section we show how to compute the conditional transitive lower bounds when constraint graphs are closed under the standard transitive closure rule, and when constraint graphs are closed under the inductive transitive closure rule.

Suppose G is closed under the standard transitive closure rule (STCR). Recall that STCR propagates sources forward along all paths. If there is a path $E \xrightarrow{M,C}{}^* \mathcal{X}$ in G, then G also contains the edge $E \xrightarrow{M,C} \mathcal{X}$. Therefore, to compute $\mathsf{CTLB}(\mathcal{X})$ for a variable $\mathcal{X}$ from G it suffices to compute $\mathsf{CTLB}(C \implies E \cap M)$ for all edges $E \xrightarrow{M,C} \mathcal{X}$ in G.

Now suppose G is closed under the inductive transitive closure rule (ITCR). Recall from above that if G is closed under ITCR and $E \xrightarrow{M,C}{}^* \mathcal{X}$ is a path from a source E to $\mathcal{X}$, then G contains a path

$$E \xrightarrow{M_1,C_1} \mathcal{X}_1 \xrightarrow{M_2,C_2} \mathcal{X}_2 \cdots \mathcal{X}_{n-1} \xrightarrow{M_n,C_n} \mathcal{X}$$

where the sequence of indices $o(\mathcal{X}_1), o(\mathcal{X}_2), \ldots, o(\mathcal{X}_{n-1}), o(\mathcal{X})$ are strictly increasing. This suggests an inductive strategy to compute $\mathsf{CTLB}(\mathcal{X})$:

- Compute $\mathsf{CTLB}(\mathcal{Y})$ for all $\mathcal{Y}$ with index lower than $\mathcal{X}$.
- Let

$$\begin{aligned}\mathsf{CTLB}(\mathcal{X}) = & \bigcup\{\mathsf{CTLB}(C \overset{l}{\implies} E \cap M) \mid E \xrightarrow{M,C} \mathcal{X} \in G, E \text{ a source}\} \\ & \cup \bigcup\{\mathsf{CTLB}(C \wedge C' \overset{l}{\implies} E \cap M) \mid \mathcal{Y} \xrightarrow{M,C} \mathcal{X} \ \wedge\ o(\mathcal{Y}) < o(\mathcal{X}) \ \wedge \\ & \qquad C' \overset{l}{\implies} E \in \mathsf{CTLB}(\mathcal{Y})\}\end{aligned}$$

The strategy is well founded since if $\mathcal{X}$ is the variable with minimal index, then for every path $E \xrightarrow{M,C}{}^* \mathcal{X}$ in G where E is a source, there is an edge $E \xrightarrow{M,C} \mathcal{X}$ in G, and $\mathsf{CTLB}(\mathcal{X})$ does not depend on the CTLB of any other variable.

The CTLB computation on a graph closed under ITCR is similar to computing the transitive closure of an acyclic graph. Consider the simple case where all conditions in G are true, all M-expressions are 1, and all constructors are non-strict. Then computing $\mathsf{CTLB}(\mathcal{X})$ corresponds exactly to computing the transitive closure of the acyclic subgraph G' formed by all variables and sources of G, and containing all edges $E \xrightarrow{1,\mathsf{true}} \mathcal{X}$ of G where E is a source, or $E = \mathcal{Y}$ and $o(\mathcal{Y}) < o(\mathcal{X})$, *i.e.*, $\mathsf{CTLB}(\mathcal{X}) = \{E \mid E \xrightarrow{1,\mathsf{true}}{}^* \mathcal{X} \in G'\}$. There exist efficient algorithms for computing the transitive closure of an acyclic graph with worst-case time complexity $O(ne^-)$, where n is the number of nodes of G', and e^- is the number of edges in the transitive reduction of G [84]. We now present an algorithm for the general case that has this complexity in the above restricted case.

The algorithm computes a set $R(\mathcal{X}_j)$ for each variable $\mathcal{X}_j$, which is a set of triples (E, M, C) such that there is a path $E \xrightarrow{M,C}{}^* \mathcal{X}_j$ in G'. The conditional transitive lower bound $\mathsf{CTLB}(\mathcal{X}_j)$ can then be computed from $R(\mathcal{X}_j)$, by

$$\mathsf{CTLB}(\mathcal{X}_j) = \bigcup\{\mathsf{CTLB}(C \overset{l}{\implies} E \cap M) \mid (E, M, C) \in R(\mathcal{X}_j) \ \wedge\ E \text{ a source}\}$$

The extra elements of $R(\mathcal{X}_j)$ that are ignored for $\mathsf{CTLB}(\mathcal{X}_j)$ are used to prune transitive edges in G', and thus to avoid merging redundant sets into $R(\mathcal{X}_j)$.

Algorithm 6.7 Suppose $R(\mathcal{X}_k)$ has been computed for all $\mathcal{X}_k$ with index less than $o(\mathcal{X}_j)$. Compute $R(\mathcal{X}_j)$ as follows.

Initially, let $R(\mathcal{X}_j) = \{\}$. Consider each edge $E \xrightarrow{M,C} \mathcal{X}_j$ in G' in turn, ordered by decreasing top-level variable index of E (expressions without top-level variables come last in no particular order). If $(E, M, C) \in R(\mathcal{X}_j)$, ignore this edge. Otherwise add (E, M, C) to $R(\mathcal{X}_j)$ and if $E = \mathcal{X}_k$ let $I = \{(E', M' \cap M, C' \wedge C) \mid (E', M', C') \in R(\mathcal{X}_k)\}$ and add set I to $R(\mathcal{X}_j)$ as well. Note that if $M = 1$ and $C = \mathsf{true}$, then $I = R(\mathcal{X}_k)$.

We now show how the algorithm avoids transitive edges $\mathcal{X}_i \xrightarrow{M,C} \mathcal{X}_j$ when there are edges $\mathcal{X}_i \xrightarrow{M,C} \mathcal{X}_k$, and $\mathcal{X}_k \xrightarrow{1,\mathsf{true}} \mathcal{X}_j$ in G'. Suppose we are computing $R(\mathcal{X}_j)$. Then the above edges imply that any triple (E, M', C') added to $R(\mathcal{X}_j)$ through $\mathcal{X}_i \xrightarrow{M,C} \mathcal{X}_j$ is also an element of $R(\mathcal{X}_k)$. The algorithm avoids merging these elements twice into $R(\mathcal{X}_j)$ by considering edge $\mathcal{X}_k \xrightarrow{1,\mathsf{true}} \mathcal{X}_j$ before $\mathcal{X}_i \xrightarrow{M,C} \mathcal{X}_j$ since $o(\mathcal{X}_i) < o(\mathcal{X}_k)$. After merging $R(\mathcal{X}_k)$ into $R(\mathcal{X}_j)$, $R(\mathcal{X}_j)$ contains the triple $(\mathcal{X}_i, M, C)$. Thus when next considering the edge $\mathcal{X}_i \xrightarrow{M,C} \mathcal{X}_j$, the algorithm skips it correctly, since all elements $\{(E, C \wedge C', M \cap M') \mid (E, C', M') \in R(\mathcal{X}_i)\}$ are already present in $R(\mathcal{X}_j)$.

6.2.2 Condition resolution

Given a system of constraints S, certain conditions C will be true in all solutions of S. Such information may be equally interesting to a program analysis as the explicit bounds on set variables themselves. This section describes how to resolve the status of all conditions in a closed constraint graph G. It is possible to classify conditions into two categories: 1) either a condition is true in all solutions, or 2) it is false in some solution. It is interesting to note that for the conditions of the 2nd category, there exist solutions of G for which these conditions are all simultaneously false.

For notational convenience, we define $\mathsf{CTLB}(\mathcal{X}) \cap M = \bigcup\{\mathsf{CTLB}(C \overset{l}{\Longrightarrow} E \cap M) \mid C \overset{l}{\Longrightarrow} E \in \mathsf{CTLB}(\mathcal{X})\}$. Assume that we have computed $\mathsf{CTLB}(\mathcal{X}_j)$ for all variables $\mathcal{X}_j$. Recall that a condition $\bigwedge_i \mathcal{X}_i \cap M_i$ is true in all solutions if each atomic condition $\mathcal{X}_j \cap M_i$ is true in all solutions. An atomic condition $\mathcal{X} \cap M$ in turn is true, if the set $\mathsf{CTLB}(\mathcal{X}) \cap M$ contains an element $\mathsf{true} \overset{l}{\Longrightarrow} E$, or some element $C \overset{l}{\Longrightarrow} E'$ where C is true. We use this observation to build a condition dependency graph G_C as follows. Let the nodes of G_C be the set of all conditions (conjuncts and atomic) appearing in $\mathsf{CTLB}(\mathcal{X}_j)$ for some $\mathcal{X}_j$. There is an *implication* edge from condition C to atomic condition $\mathcal{X} \cap M$, iff $C \overset{l}{\Longrightarrow} c(\dots)$ appears in $\mathsf{CTLB}(\mathcal{X}) \cap M$. Furthermore, there is a *containment* edge from atomic condition $\mathcal{X} \cap M$ to every condition C' containing $\mathcal{X} \cap M$ as a conjunct.

Algorithm 6.8 The following naive algorithm marks all conditions that are true in every solution:

```
Repeat until no more conditions are marked
```

```
  For each condition $C$ in $G_C$
    if $C$ is not marked then
      if $C$ is atomic of the form $\mathcal{X} \cap M$ then
         if $\mathsf{CTLB}(\mathcal{X}) \cap M$ contains true $\overset{l}{\Longrightarrow} E$, then
           mark $C$
         fi
      else ($C$ is not atomic)
         inspect all $C'$ s.t. $C' \rightarrow C$ is a containment edge.
         If all $C'$ are marked, then mark $C$ fi
      fi
    else ($C$ is marked)
      for all implication edges $C \rightarrow C'$, mark $C'$
    fi
  endfor
endrepeat
```

Clearly, if the algorithm marks a condition C, then C is true in all solutions. On the other hand, suppose there is an atomic condition $\mathcal{X} \cap M$ not marked by the algorithm. Then we can add the edge $\mathcal{X} \xrightarrow{M,\mathsf{true}} 0$ to G without making the graph inconsistent.

There is clearly a better algorithm which visits each edge in G_C at most once.

Algorithm 6.9 Associate a counter $cnt(C)$ with each condition C. For atomic conditions, set the counter to 1. For non-atomic conditions, set the counter to the number of its atomic conjuncts.

```
dec-and-propagate($C$) =
  if $cnt(C)$ > 0 then
    $cnt(C)$ := $cnt(C)$ - 1
    if $cnt(C)$ == 0 then
      mark $C$
      for each outgoing edge $C \rightarrow C'$ in $G_C$
        dec-and-propagate($C'$)
      endfor
    fi
  fi

For each atomic condition $C$ of the from $\mathcal{X} \cap M$
  if $\mathsf{CTLB}(\mathcal{X}) \cap M$ contains an element true $\overset{l}{\Longrightarrow} E$, then
    dec-and-propagate($C$)
  fi
```

The identification of conditions that are true in all solutions can be performed incrementally and at the same time that the constraint graph closure is computed. It is then possible to take advantage of the information about conditions to add only edges $E \xrightarrow{M,\mathsf{true}} E'$ to the graph where the condition is true. Heintze describes such an algorithm for computing the results of Set-Based Analysis in his dissertation [38]. This algorithm is based on a variation of the standard transitive closure rule that adds only edges whose condition is true. The algorithm requires substantial book-keeping of suspended edges, *i.e.*, edges $E \xrightarrow{M,C} E'$ that would normally be added through the transitive or structural closure, but where C is not known to be true. Furthermore, whenever a new edge is added to the graph, the condition of some suspended constraint may become true, and thus suspended edges need to be examined.

The implementation of BANE described in the second part of this dissertation does not suspend conditional constraints. It is based on the inductive transitive closure rule (ITCR) and solves conditional constraints as described in Section 6.1. Algorithm 6.9 is used to explicitly solve conditions when computing the transitive lower bounds TLB from the conditional transitive lower bounds CTLB. As we show in Section 9.4, this approach enables computing the TLB on-demand, resulting in substantially smaller constraint graphs than obtained using the standard algorithm for Set-Based Analysis.

6.3 Complexity

We briefly discuss the worst-case time complexity for deciding the consistency of set of mixed constraints. The general problem of deciding the consistency of the closure of a constraint graph is at least NEXPTIME. This lower bound follows from a result by McAllester and Heintze [59], who show complexity bounds for various refinements of Set-Based Analysis (SBA). The hardest form of SBA they study is NEXPTIME-complete. Since all their SBA variations can be formulated using the Set-constraint part of mixed constraints, resolution of mixed constraints is at least NEXPTIME.

6.4 Discussion and Related Work

The use of inclusion constraints in program analysis goes back to an early paper by Reynolds. Reynolds [74] describes an analysis for inferring the shape of data-structures in Lisp programs. It provides the basic idea of later analyses based on inclusion constraints [48, 2, 41], although the same ideas were independently rediscovered in some cases.

Jones and Muchnick [48] describe an abstract interpretation approach to inferring a description of list data-structures at each program point of imperative flow-chart programs. Their system is very similar to the one of Reynolds, but is not expressed using constraints.

Decidability and complexity results for a number of variations of set constraints have been studied by numerous researchers [40, 33, 34, 10, 5, 13, 14, 6]. Heintze popularized set-constraints for program analysis through his work on set-based analysis (SBA) of logic and functional programs [38]. He expresses set-based analysis using constraints of the form $\mathsf{op}(\mathcal{X}_1..\mathcal{X}_n) \subseteq \mathcal{X}$, where the right-hand side of all constraints is a set-variable, and the left-hand side is an arbitrary monotone set-operator op. His formulation of solutions to the constraints is somewhat non-standard, since the interpretation of certain set-operators constrains the solution. As a result, his formulation only applies to the least solution of the constraints. Least solutions always exist since all set-operators are monotone. The constraint resolution proposed by Heintze differs from our approach in that it uses the standard transitive closure rule we described earlier. We will contrast our implementation of Set-constraints with the standard set-constraint resolution described in Heintze's dissertation further in Chapter 9, where we show that constraint resolution based on standard transitive closure is severely limited in its ability to scale to large constraint problems due to the fact that it explicitly computes the transitive lower bound of all variables.

Independently of Heintze, Aiken and Wimmers developed set-constraint decision procedures for type inference of FL [2, 3]. The Set-constraint resolution of mixed constraints is based on their work, which we already described in Chapter 2.

FlowTerm-constraints can be viewed as constraints arising in the type inference system PTB_μ+int studied by Palsberg and O'Keefe [67] and Palsberg, Wand, and O'Keefe [69]. The system PTB_μ+int stands for partial types with top and bottom and integer types. It is an extension of Thatte's system of partial types [81] with bottom. Constraints arising in this system are non-structural subtyping relations, meaning that two types can be related even if they don't have the same structure. The non-structural relation matches our inclusion relation on FlowTerm-ideals. Palsberg et al. [67] show that their type system is equivalent to a flow analysis and give a cubic time algorithm to infer the flow relations. The algorithm is essentially the full constraint graph closure we described in this chapter.

Term-constraints are most similar to equality constraints between first-order terms, but augmented with top and bottom. Without top, Term-constraints correspond to the *conditional unification* constraints proposed by Steensgaard [79]. A constraint $E_1 \leq E_2$ in his system is satisfied, if either E_1 is bottom, or $E_1 = E_2$, but there is no top in his language. Steensgaard describes a nearly linear time algorithm for solving such constraints. The constraints proposed by Henglein for inference of global tagging optimization of Lisp programs [46] and for binding-time analysis in partial evaluation [45] are similar to Term-constraints with top, but no bottom. A constraint $E_1 \leq E_2$ in these systems is satisfied if either E_2 is top, or $E_1 = E_2$. However, in certain variations of the tagging and binding-time analyses, further constraints are placed on E_1 if E_2 is top, namely that if E_1 has the form $c(E'_1, .., E'_n)$, then $E'_i = \top$ for $i = 1..n$. Henglein shows that such constraints can be solved in nearly linear time. These extra constraints on the sub-expressions of E_1 cannot be expressed in our current formulation of mixed constraints.

The notion of *row* types was introduced by Wand [86]. He later pointed out that his algorithm for solving equality constraints between type and row expressions may not terminate due to the introduction of fresh row variables [87]. The fix to Wand's non-termination problem is similar to the termination argument we used for splitting closed rows. Both ideas are based on the formalization of record types by Rémy [73]. Rémy proposed an extension of the ML type system with extensible record types. Rémy recognized that Row-variables need an annotation indicating the set of labels that may not be present. He writes this annotation as a superscript χ^L, whereas we introduced domain constraints. Type inference for Rémy's record extensions are based on equality constraints between closed rows. Record subtyping is obtained in his system solely through parametric polymorphism. Jategaonkar and Mitchell [47] consider a type system and type inference for an extension of ML with records and atomic subtyping. They provide an inference algorithm based on unification [75] and atomic subtype resolution [63]. Ohori [65] presents a type system similar to the one of Rémy for adding extensible records. Type inference is again based on equality constraints and implemented through an extension of unification. Record subtyping is obtained through bounded parametric polymorphism. So far, all record type inference systems described are based on equality constraints that are solved through an extension to the unification algorithm. The only type inference system we are aware of that uses inclusion constraints between records is by Stansifer [78]. He studied type inference in the presence of record and variant types. The constraints arising in his type derivations between variant rows correspond to constraints between minimal Row-expressions, and the constraints on record types correspond to constraints on maximal Row-expressions. Stansifer did not give any constraint resolution rules or consistency criteria for the generated constraints. The resolution of minimal and maximal Row-constraints presented here can be seen as filling in that gap.

Particular instances of mixed constraint systems have been described in the literature in the past. Type systems with record types have already been described above. They fall into the category of mixed Term and Term-Row expressions and constraints. Another category of mixed expressions and constraints arise in *effect systems* [54, 55, 49, 80, 82]. Effect systems generalize type systems in that they also infer a description of some behavioral aspects of evaluation, usually side effects. Examples of side effects are reads and writes to the store, sending or receiving messages, and non-local control transfers (for example exceptions). Effects are naturally described as sets. Classic effect systems are instances of mixed Term and Set expressions and constraints, with the additional property that there are no mixed Set-constructors, *i.e.*, Term expressions don't appear in effects. Lucassen and Gifford [55] describe type and effect inference rules using a subset relation on types induced by the subset relation of effect sets contained in the types. Such constraints correspond to mixed FlowTerm-Set constraints. However, they do not show how to solve such constraints and, in fact, in a later paper drop the subset constraints for equality constraints, obtaining a Term-Set instance of mixed constraints, which they solve with generalized unification [49]. Similarly, Tofte and Talpin [82] use a mixture of types and sets in an effect system to infer allocation and deallocation points of memory regions at compile-time. Their inference rules are also based on equality constraints which they solve using a generalized unification procedure. In Section 8.2 we describe an effect system for computing uncaught exceptions in Standard ML programs based on three different mixed constraint systems, one of which we have published earlier [25]. The exception inference system generalizes previous effect systems in that effect sets also contain mixed constructors.

Mossin [64] independently studied a form of mixed constraints in the context of flow analysis for higher order functional programs. In his formulation, flow analysis computes information about how values flow through a program. Every expression constructing a value is tagged with a label. The result of a flow analysis is then a mapping of program points to sets of labels. He formulates flow analysis as a type system, where standard types for functional programs are augmented with label annotations. Constraints between annotated types are similar to mixed FlowTerm-Set constraints. However, Mossin's formulation separates the type inference from the flow analysis. During flow analysis, the type structure of the program is already known, and only constraints between the label annotations are generated and solved. Furthermore, his resolution algorithm is based on the fact that the label sets only contain constants and variables and that no non-trivial upper bounds on label sets are added as constraints. As a result, the label set constraint graph is trivially consistent and needs not be closed under transitive or structural rules. This property results in an pseudo-linear time algorithm for computing the flow graph. Pseudo-linear refers to the fact that the algorithm is linear in the size of the type structure of a program, but the type structure itself can be exponential in the program size. However, many researchers have argued that in practice programmers don't write programs with huge types, for such programs would be hard to understand. Thus in practice, the size of the type structure is often of the same order as the program size. Individual queries about labels sets can be answered in linear time by a depth-first search in the flow graph. Computing all queries results in a quadratic algorithm.

Independently, Heintze and McAllester [42] developed essentially the same idea for performing closure analysis of ML programs. Each lambda expression is labelled and types are augmented with label annotations. They compute a flow graph similar to Mossin in pseudo-linear time and are able to answer individual queries in linear time.

There is a large body of work on atomic subtyping constraints starting with Mitchell [63] and later Fuh and Mishra [32], which we do not review here since the current formulation of mixed constraints does not have a notion of atomic subtyping. It would however be entirely plausible to add another sort for expressions with an atomic subtype relation.

Part II

BANE

Chapter 7

BANE: An Implementation of Mixed Constraints

This chapter describes the implementation of the BANE library, discussing important engineering decisions and algorithmic ideas supporting the resolution of large constraint problems. BANE (the Berkeley ANalysis Engine) implements a resolution engine for mixed constraints based on the algorithm for solving conditional constraints presented in Chapter 6.

BANE is implemented as a library in Standard ML of New Jersey [9] and encompasses about 20,000 lines of code.

Section 7.1 describes implementation aspects common to all sorts, in particular the representation used for constraint graphs. The subsequent sections describe aspects specific to the implementation of each sort. Finally, Section 7.6 describes BANE's support for polymorphic constraint-based analysis.

7.1 Constraint Graph Representation

This section describes BANE's representation of constraint graphs. Recall from Chapter 5 and Chapter 6 that constraints can be simplified to inductive constraints on variables. In terms of graph edges, each inductive constraint represents an edge between two mixed expressions, where at least one endpoint is a variable. This fact enables a graph representation where edges are stored as adjacency lists on variable nodes. Note that graph representations based on adjacency matrices are not practical since they require space quadratic in the number of nodes even if the graph is sparse. BANE's basic assumption is that the final (closed) constraint graph is sparse, *i.e.*, the number of edges will be roughly linear in the number of nodes. Constraint problems with inductively closed graphs containing a quadratic number of edges are not expected to scale well. As we show in later sections, BANE uses techniques to restrain the blowup of edges and maintains sparse graphs for many constraint problems that would otherwise exhibit a quadratic number of edges.

We use ML datatype declarations to specify the shape of the data structures described in this chapter. An ML datatype declaration has the form datatype dty = C_1 [of ty_1] ... C_n [of ty_n]

where dty is the name of the declared datatype, and the C_i are value constructors. Value constructors are either constants, or they carry an argument of type ty_i. A value of a datatype dty is either one of the constants C_i, or a constructor C_i applied to a value of type ty_i.

The next subsection describes the representation of nodes in our constraint graphs, followed by a subsection on the representation of edges. Section 7.1.3 describes *alias edges*, a special kind of edge to represent equality constraints. Section 7.1.4 describes how sets of mixed expressions are represented efficiently. Section 7.1.5 describes a technique for performing online cycle elimination in the constraint graph which forms one of the core techniques of BANE for scaling to large constraint problems.

7.1.1 Nodes

Nodes are essentially mixed expressions. The datatype for mixed expressions used by BANE has the following form

```
datatype me = Var of varinfo
            | Cons of {con:constructor, args: me list}
            | Row of {fields: {l:label, e:me} list, rest:me}
            | EmptyRow
            | Zero
            | One
            | LUnion of me list
            | RInter of me list
            | LInter of {v : me, m : me}
            | RPat of {e : me, m : me}
            | Cond {co:condition, e : me}
            | Neg of constructor list
```

The value constructors for mixed expressions consist of a variable constructor `Var` with associated information of type `varinfo` which we describe in the next section, a value constructor for constructor expressions `Cons` with an argument record containing a value for the constructor `con` and the list of argument mixed expressions (`args`). A record in ML {l$_1$: ty_1, ..., l$_n$: ty_n} is similar to a struct in C with fields $l_1..l_n$ containing values of types $ty_1..ty_n$. An ML type of the form ty `list` is a datatype for lists of elements of type ty. The value constructor `Row` is used to represent Row-composition $\langle l : E_l\rangle_A \circ E$, where `fields` is the list of label-expression pairs $(l : E_l)$ and `rest` holds the composed expression E. The constants `EmptyRow`, `Zero`, and `One` represent the empty Row $\langle\rangle$, the minimal expression 0, and the universal expression 1. Value constructor `LUnion` forms L-unions for Set-expressions, *i.e.*, unions appearing in L-contexts. Similarly, `RInter` forms intersections appearing in R-contexts. Value constructor `LInter` represents L-intersections. This constructor thus makes the invariant explicit that L-intersections must be of the form $\mathcal{X} \cap M$, *i.e.*, a variable intersected with an M-expression. Similarly, constructor `RPat` forms R-patterns Pat$[E, M]$ which abbreviate the R-union $E \cap M \cup \neg M$. Value constructor `Cond` is used to represent conditional expressions $C \overset{l}{\Longrightarrow} E$ and $C \overset{r}{\Longrightarrow} E$. The particular kind is always apparent from the context. Finally, constructor `Neg` represents cofinite negations of Set-constructors $\neg\{c_1, \ldots, c_n\}$. We don't give the explicit form of constructor values, conditions, and labels. Constructor values simply contain the name and the signature of a constructor, and conditions are sets of atomic conditions of the form $\mathcal{X} \cap M$.

A couple of remarks about the expression representation are in order. The actual implementation uses a less verbose representation, where the constants `EmptyRow`, `Zero`, and `One` are represented as special cases of the other constructors. L-unions and R-intersections formed by `LUnion` and `RInter` are never empty, *i.e.*, 0 and 1 are uniquely represented by `Zero` and `One`. M-expressions are formed using the constructors `Zero`, `One`, `Cons`, `Neg`, and `LUnion`. BANE uses a common datatype to represent expressions for all sorts. Expressions for a particular sort only use a subset of all the value constructors given above. The use of the common datatype is motivated by the need to avoid code blowup. There is a good amount of code that can be shared among the sorts $\mathbf{s}, \mathbf{ft}, \mathbf{t}$, and among the Row-sorts $\mathbf{r}(\mathbf{s}), \mathbf{r}(\mathbf{ft}), \mathbf{r}(\mathbf{t})$. If each sort is given a separate datatype, we end up with six distinct mutually recursive datatypes and code for one sort cannot be applied to expressions of other sorts. Since constructor arguments can refer to any other sort, an extra common datatype that represents expressions of any sort is needed nevertheless. Although a clean separation of sorts at the ML-typelevel would be desirable, ML's lack of mutually recursive modules makes such an approach impractical.

For similar reasons, we do not use separate datatypes to represent L-compatible and R-compatible expressions. It is important to note that the common representation of mixed expressions is only visible internally in BANE. At the library interface, expressions of different sorts are given distinct ML types. Thus an analysis written using BANE as a library enjoys the benefits of ML-typechecking for avoiding programming errors related to the confusion of distinct sorts.

The nodes of constraint graphs are now represented by mixed expression values of type `me`, with the restriction that Set-expressions do not contain L-unions and R-intersections at top-level. Constraints involving L-unions and R-intersections are readily broken up into constraints without L-unions and R-intersections using the resolution rules given earlier

$$\begin{aligned} E_1 \cup E_2 \subseteq_s E_3 &\iff E_1 \subseteq_s E_3 \ \wedge \ E_2 \subseteq_s E_3 \\ E_1 \subseteq_s E_2 \cap E_3 &\iff E_1 \subseteq_s E_2 \ \wedge \ E_1 \subseteq_s E_3 \end{aligned}$$

7.1.2 Edges

As outlined before, edges are represented as adjacency lists on variable nodes. The adjacency lists are stored in the variable information `varinfo` associated with each variable node.

```
datatype varinfo = LU of {lb : meset, ub : meset}
```

The field `lb` contains a set of mixed expressions (type `meset`) representing lower bounds on the variable, *i.e.*, if $E \in$ `lb` of a variable $\mathcal{X}$, then the edge $E \to \mathcal{X}$ is present. Similarly, `ub` contains a set of mixed expressions representing upper bounds on the variable.

Each graph edge is represented uniquely as either a predecessor edge, or a successor edge. Recall from the previous chapter that edges are either of the form $E \xrightarrow{M,C} \mathcal{X}$ where E is a source, $\mathcal{X} \xrightarrow{M,C} E$ where E is a sink, or the constraint contains top-level variables on both sides. (Recall that sources and sinks E are mixed expression of the form 0, 1, $c(\dots)$, $\langle l : E_l \rangle_A \circ K$, or $\neg\{c_1, \dots, c_n\}$, where K is 0, 1, or $\langle\rangle$.)

The first form is always represented by placing the expression $C \overset{l}{\Longrightarrow} E \cap M$ in the set of lower bounds of $\mathcal{X}$. Similarly, the second form is represented by placing $C \overset{r}{\Longrightarrow} \mathsf{Pat}[E, M]$ in the set of upper bounds of $\mathcal{X}$.

Recall that we associate an index (a unique integer) $o(\mathcal{X})$ with each variable $\mathcal{X}$ thus creating a total order on the variables. The representation of edges where both sides contain top-level variables depends on the respective order of the top-level variables. These remaining edges are of the form $\langle l : E_l \rangle_A \circ [\mathcal{Y}] \xrightarrow{1,C} \mathcal{X}$, $\mathcal{X} \xrightarrow{1,C} \langle l : E_l \rangle_A \circ [\mathcal{Y}]$, and $\mathcal{X} \xrightarrow{M,C} \mathcal{Y}$, where $[\mathcal{Y}]$ stands for Row-mask expressions involving variable $\mathcal{Y}$ (Section 5.4.4). The resolution of Row-constraints in Section 5.4 guarantees that we can transform constraints into inductive constraints, *i.e.*, $o(\mathcal{Y}) < o(\mathcal{X})$ in the edges $\langle l : E_l \rangle_A \circ [\mathcal{Y}] \xrightarrow{1,C} \mathcal{X}$, and $\mathcal{X} \xrightarrow{1,C} \langle l : E_l \rangle_A \circ [\mathcal{Y}]$. Thus, the first form is represented by adding $C \overset{l}{\Longrightarrow} \langle l : E_l \rangle_A \circ [\mathcal{Y}]$ to the lower bounds of $\mathcal{X}$, and the second form is represented by adding $C \overset{r}{\Longrightarrow} \langle l : E_l \rangle_A \circ [\mathcal{Y}]$ to the upper bounds of $\mathcal{X}$.

Finally, an edge $\mathcal{X} \xrightarrow{M,C} \mathcal{Y}$ is represented by adding $C \overset{l}{\Longrightarrow} \mathcal{X} \cap M$ to the lower bounds of $\mathcal{Y}$, if $o(\mathcal{X}) < o(\mathcal{Y})$, and by adding $C \overset{r}{\Longrightarrow} \mathsf{Pat}[\mathcal{Y}, M]$ to the upper bounds of $\mathcal{X}$, if $o(\mathcal{Y}) < o(\mathcal{X})$. We call this representation of the edges the *inductive form* (IF) of the graph. Inductive form graphs have the property that the top-level variables $\mathcal{Y}$ in any lower or upper bound expression of variable $\mathcal{X}$ have indices strictly less than $\mathcal{X}$. We thus call all edges in an inductive graph *inductive edges* since they correspond to inductive constraints.

The order $o(\cdot)$ on variables used by BANE is simply the order in which the variables are generated, and we call this order the gen order. The particular choice of $o(\cdot)$ influences the number of edges in the closed graph. In practice, we have compared the gen order to randomly generated orders and found that there was no significant difference. The gen order however has some pleasing properties that we exploit to prove termination when introducing fresh variables.

7.1.3 Alias Edges

In the subsequent sections it is convenient to use an additional kind of graph edge called an *alias edge*. An alias edge $\mathcal{X} \overset{=}{\rightarrow} E$ is a directed edge from a variable $\mathcal{X}$ to an expression E. An alias edge $\mathcal{X} \overset{=}{\rightarrow} E$ is inductive, if each top-level variable $\mathcal{Y} \in \mathsf{TLV}(E)$ has index lower than $o(\mathcal{X})$. We only consider constraint graphs with inductive alias edges. An alias edge abbreviates an equality between $\mathcal{X}$ and E that could be represented with two edges $E \rightarrow \mathcal{X}$ and $\mathcal{X} \rightarrow E$. Note that if the alias edge is inductive, so are the two constraints it abbreviates. The advantage of alias edges however does not come from saving a single edge in the graph. Instead, we maintain an invariant on X, that if $\mathcal{X} \overset{=}{\rightarrow} E$ is an inductive alias edge in a constraint graph G, then there are no other alias edges $\mathcal{X} \overset{=}{\rightarrow} E'$, no L-inductive edges $E' \rightarrow \mathcal{X}$ and no R-inductive edges $\mathcal{X} \rightarrow E'$ in G. If there exists an alias edge $\mathcal{X} \overset{=}{\rightarrow} E$

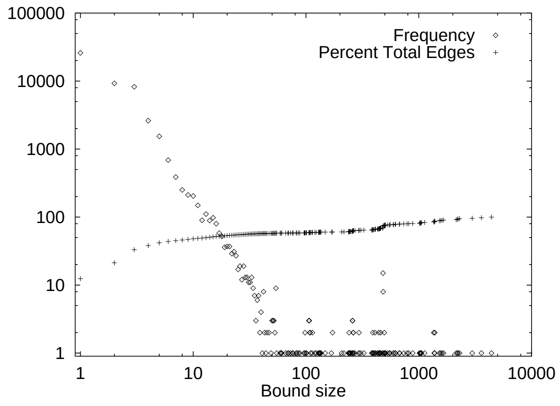

Figure 7.1: Example frequency of bound sizes

in G, then we say that $\mathcal{X}$ is aliased in G.

This invariant is enforced by the `varinfo` datastructure on variables by adding a variant `Alias` containing the aliased expression E.

```
datatype varinfo = LU of {lb : meset, ub : meset} | Alias of me
```

The advantage of alias edges comes from the possibility to compress alias paths. An *alias path* $\mathcal{X} \overset{=}{\longrightarrow}^* E$ is a sequence of alias edges and variables $\mathcal{X}_1 \ldots \mathcal{X}_n$, such that $\mathcal{X} = \mathcal{X}_1$, and $\mathcal{X}_i \overset{=}{\longrightarrow} \mathcal{X}_{i+1}$, and $\mathcal{X}_n \overset{=}{\longrightarrow} E$. If G contains an alias path $\mathcal{X} \overset{=}{\longrightarrow}^* E$, then the unique alias edge $\mathcal{X} \overset{=}{\longrightarrow} \mathcal{X}_2$ may be replaced by $\mathcal{X} \overset{=}{\longrightarrow} E$ without changing the solutions of the constraints represented by G. This path compression is equivalent to Tarjan's path compression in the union-find algorithm.

For the moment we only consider non-recursive aliases that are L- and R-compatible.

Definition 7.1 *An alias edge $\mathcal{X} \overset{=}{\longrightarrow} E$ is non-recursive, if E is non-recursive in $\mathcal{X}$. We say that E is non-recursive in $\mathcal{X}$ if $\mathcal{X}$ does not appear in E, and for all aliased variables $\mathcal{Y}$ in E with alias paths $\mathcal{Y} \overset{=}{\longrightarrow}^* E'$, E' is non-recursive in $\mathcal{X}$.*

Assuming that $\mathcal{X}$ is not aliased in the current graph, and its lower bounds are `lb`$(\mathcal{X})$ and its upper bounds are `ub`$(\mathcal{X})$, an alias edge $\mathcal{X} \overset{=}{\longrightarrow} E$ can be introduced into a graph by replacing the variable info associated with $\mathcal{X}$ by `Alias` E, and adding the constraint $L \subseteq_s E$ for each $L \in$ `lb`$(\mathcal{X})$ and the constraint $E \subseteq_s U$ for each $U \in$ `lb`$(\mathcal{X})$.

7.1.4 Expression Hashing and Bound Representation

Edges in the constraint graph are represented as sets of mixed expressions in the upper and lower bounds of variables. During constraint resolution, some of these sets grow to substantial size. Even if we assume that the final graph is sparse, there can still be many nodes whose lower or upper bounds have size proportional to the number of nodes in the graph. Such cases do happen in practice, and it is thus crucial that operations for membership test and element addition on these sets be constant amortized time. Figure 7.1

shows an example edge distribution for a constraint graph obtained with the points-to analysis described in Section 8.1. The graph contains 59600 variables nodes and 209139 edges. The x-axis represents the size of a variable bound (number of expressions in lower and upper-bounds). The frequency plot represents the number of variables in the graph with a particular bound size. There are numerous variables with bound sizes larger than 100 and even larger than 1000 expressions. Note that there is one variable with a bound size of 4500. The second plot (Percent Total Edges) shows that most edges appear on variables with small bounds. A point (x, y) of the second plot states that y percent of the total number of edges appear in bounds of variables with bound size less or equal to x. For example, 12% of edges appear on variables with only one expression in their upper and lower-bounds.

Since expressions themselves can be large, we do not want to structurally compare them. Instead, we associate a unique integer $\mathsf{id}(E)$ with each mixed expression E built in BANE. The association of expressions and their id's is stored as an extra field on the expression itself. In order to generate the same id if the same expression is built twice, BANE uses hash-consing, *i.e.*, each syntactically distinct expression is represented at most once in memory. Hash consing guarantees the unique mapping between expressions and their id's, and furthermore supports maximal sharing of expressions, thereby reducing memory requirements. The expression hash table is implemented using *weak pointers*, (pointers that do not keep an object alive when otherwise unreachable) in order to reclaim storage of expressions no longer in use. The hash table grows dynamically when full by doubling the table size and rehashing.

The unique id associated with each expression E enables the use of hash-tables to represent sets of expressions. A set is represented as a pair of a list and a small hash-table. The list contains the elements in the set (no duplicates), and the hash-table contains the unique ids of all expressions in the set. To test membership of E in the set, it suffices to test whether $\mathsf{id}(E)$ is in the hash-table. Adding an expression E to a set reduces to testing membership of E, and if not present, consing E onto the head of the list and adding $\mathsf{id}(E)$ to the hash-table. The hash tables used for sets are initially very small (2 elements) and grow dynamically by doubling the table size when full and rehashing.

The use of individual hash tables per bound instead of using a global hash table representing the presence of edges in the constraint graph is motivated by the desire to prune unreachable variables from constraint graphs efficiently (Section 7.6). Using local hash tables, unreachable variables and their edges can be left to the garbage collector. In the presence of a global hash table, edges incident on unreachable variables would have to be removed individually.

7.1.5 Online Cycle Detection and Elimination

The performance of constraint resolution can be improved by simplifying the constraint graph at various times during the resolution. Prior work on constraint graph simplification has shown that periodic simplification performed during resolution helps to scale to larger analysis problems [24, 30, 58], but absolute performance is still unsatisfactory. One problem is deciding the frequency at which to perform simplifications to keep a well-balanced cost-benefit tradeoff. Simplification frequencies in past approaches range from once for an entire module to once for every program expression.

BANE contains a novel technique for a particular constraint graph simplification, namely detecting and eliminating cycles. As we will show in the chapter on experiments, cycles in the constraint graph are a key inhibitor for scaling to large Set-constraint problems. This section describes the algorithm used in BANE for detecting and eliminating cycles in constraints. The algorithm is applicable to all sorts.

Cycles in the constraint graph are sequences of edges $\mathcal{X}_1 \xrightarrow{1,\mathsf{true}} \mathcal{X}_2 \xrightarrow{1,\mathsf{true}} \cdots \xrightarrow{1,\mathsf{true}} \mathcal{X}_n$ where $\mathcal{X}_1 = \mathcal{X}_n$. Such cycles imply that $\mathcal{X}_1 = \mathcal{X}_2 = \cdots = \mathcal{X}_{n-1}$ in all solutions of the constraints. In general, let V be a strongly connected component in the graph, *i.e.*, V is a set of variables such that for all pairs $(\mathcal{X}, \mathcal{Y})$ of variables in V, there is a path $\mathcal{X} \xrightarrow{1,\mathsf{true}}{}^{*} \mathcal{Y}$ and a path $\mathcal{Y} \xrightarrow{1,\mathsf{true}}{}^{*} \mathcal{X}$ in G. Then the variables of V are equal in all solutions. The idea is to collapse the strongly connected component V to a single representative variable of V.

BANE takes the extreme approach to simplification frequency by performing cycle detection and elimination *online*, *i.e.*, at every update of the constraint graph. At first glance, this approach seems overly expensive, since the best known algorithm for online cycle detection performs a full depth-first search for half of all edge additions [77]. The detection algorithm used by BANE is actually only partial and does not detect all cycles. It is the partial aspect of it that makes it practical. Section 9.2 will show that our partial detection costs only a small constant overhead per edge addition by only traversing a few edges during the search. Nevertheless, many cycles are detected, boosting performance by more than an order of magnitude for large constraint problems.

Note that it is not sufficient to use the well-known linear time algorithm to eliminate cycles in the initial unclosed constraint graph. The structural rules used during constraint graph closure add new edges to the graph which may form cycles that are not apparent in the initial constraint graph. Thus cycle detection needs to be performed during the resolution.

We first show how to perform the detection of cycles, and then how cycles are collapsed.

Partial Detection Algorithm

We would like to know whether adding an edge $\mathcal{X} \xrightarrow{1,\mathsf{true}} \mathcal{Y}$ to a constraint graph G closes a cycle. Thus we need to know whether G contains a path $\mathcal{Y} \xrightarrow{1,\mathsf{true}}{}^{*} \mathcal{X}$. This question may be answered in several ways. If we maintained transitive edges between all variables, then we could simply query the presence of edge $\mathcal{Y} \xrightarrow{1,\mathsf{true}} \mathcal{X}$. However, maintaining full transitive closure is expensive to compute and requires quadratic space in practice. Another way to answer the query is to perform a depth-first search on the graph, starting at $\mathcal{Y}$, searching for $\mathcal{X}$, or doing a depth-first-search on the reverse graph starting at $\mathcal{X}$ and searching for $\mathcal{Y}$. Doing a full depth-first search however is still expensive, since on every edge addition a very large part of the graph may be explored. What we would like is a way to restrict our search by traversing only certain kinds of edges.

Fortunately, the inductive graph representation provides a very natural restriction. Recall that each edge $\mathcal{X} \xrightarrow{1,\mathsf{true}} \mathcal{Y}$ is stored as a lower bound on $\mathcal{Y}$ if $o(\mathcal{X}) < o(\mathcal{Y})$, or as an upper bound on $\mathcal{X}$ if $o(\mathcal{Y}) < o(\mathcal{X})$. Thus, inductive form doesn't even allow a

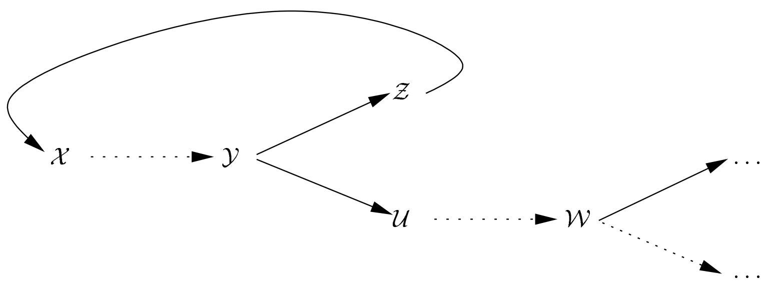

Figure 7.2: Cycle detection in an example graph

standard depth-first search, since not all successors of a variable $\mathcal{X}$ are apparent in the upper bounds of $\mathcal{X}$. Only successors with lower index than $\mathcal{X}$ are present. Similarly, only predecessors with lower index than $\mathcal{X}$ are present in its lower bounds. Performing a forward depth-first search in the inductive graph using only the inductive edges corresponds to a restricted forward depth-first search in the graph, where an edge $\mathcal{X} \rightarrow \mathcal{Y}$ is only traversed if $o(\mathcal{Y}) < o(\mathcal{X})$. Similarly, performing a backward depth-first search in the inductive graph using only inductive edges corresponds to a depth-first search in the reversed graph, where again an edge $\mathcal{X} \rightarrow \mathcal{Y}$ is only traversed if $o(\mathcal{Y}) < o(\mathcal{X})$.

Let an *inductive path* from $\mathcal{Y}$ to $\mathcal{X}$ be a sequence of edges $\mathcal{Z}_1 \rightarrow \mathcal{Z}_2 \rightarrow \cdots \mathcal{Z}_n$ such that $\mathcal{Y} = \mathcal{Z}_1$, $\mathcal{Z}_n = \mathcal{X}$, and the sequence of indices $o(\mathcal{Z}_1), o(\mathcal{Z}_2), \ldots, o(\mathcal{Z}_n)$ is strictly decreasing. Similarly, let a *reverse inductive path* from $\mathcal{Y}$ to $\mathcal{X}$ be a sequence of edges $\mathcal{Z}_1 \rightarrow \mathcal{Z}_2 \rightarrow \cdots \mathcal{Z}_n$ such that $\mathcal{Y} = \mathcal{Z}_1$, $\mathcal{Z}_n = \mathcal{X}$, and the sequence of indices $o(\mathcal{Z}_1), o(\mathcal{Z}_2), \ldots, o(\mathcal{Z}_n)$ is strictly increasing.

Algorithm 7.2 The observation on depth-first search in inductive graphs translates into the following strategy for detecting cycles when adding an edge $\mathcal{X} \rightarrow \mathcal{Y}$.

- If $o(\mathcal{X}) < o(\mathcal{Y})$ search for an inductive path $\mathcal{Y} \rightarrow^* \mathcal{X}$ along successor or alias edges, starting from the upper bounds of $\mathcal{Y}$. Prune the search whenever a variable $\mathcal{Z}$ is reached with index lower than $o(\mathcal{X})$, since in that case there are no inductive paths from $\mathcal{Z}$ to $\mathcal{X}$.

- If $o(\mathcal{Y}) < o(\mathcal{X})$ search for a reverse inductive path $\mathcal{Y} \rightarrow^* \mathcal{X}$, but search for the path in reverse, *i.e.*, starting from $\mathcal{X}$, following predecessor or alias edges to lower indexed variables. Prune the search whenever a variable $\mathcal{Z}$ is reached with index lower than $o(\mathcal{Y})$, since in that case there are no reverse inductive paths from $\mathcal{Y}$ to $\mathcal{Z}$.

We illustrate the search algorithm using the example graph of Figure 7.2. The edge $\mathcal{X} \cdots\!\!\blacktriangleright \mathcal{Y}$ is the edge we are about to add and which triggers the cycle detection. Plain edges in the graph represent successor edges, *i.e.*, edges $\mathcal{Y} \longrightarrow \mathcal{Z}$ where the index $o(\mathcal{Y})$ is larger than the index $o(\mathcal{Z})$. Dotted edges represent predecessor edges, *i.e.*, edges $\mathcal{U} \cdots\!\!\blacktriangleright \mathcal{W}$ where the index $o(\mathcal{U})$ is smaller than $o(\mathcal{W})$. The example assumes that the index $o(\mathcal{X})$ is smaller than $o(\mathcal{Y})$. We are thus in the first case of Algorithm 7.2, searching for an inductive path $\mathcal{Y} \rightarrow^* \mathcal{X}$ along successor edges. In our notation, this path corresponds to a path using

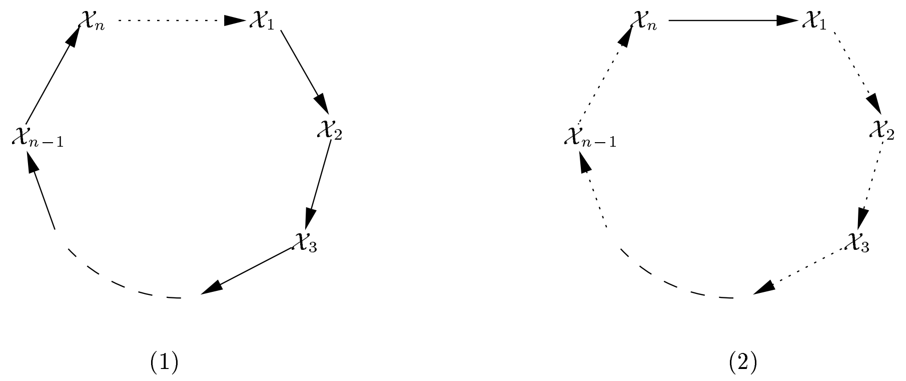

Figure 7.3: The two kinds of cycles detected

only plain arrows from $\mathcal{Y}$ to $\mathcal{X}$. The search finds such a path through $\mathcal{Z}$. Note that the search also explores node $\mathcal{U}$, since it is reachable along a plain edge from $\mathcal{Y}$. However, node $\mathcal{W}$ and it successors are not explored, since $o(\mathcal{W}) > o(\mathcal{U})$.

The above strategy only detects cycles of the two following kinds, illustrated in Figure 7.3.

1. G contains an inductive path $\mathcal{X}_1 \rightarrow^* \mathcal{X}_n$ and we add edge $\mathcal{X}_n \rightarrow \mathcal{X}_1$. In this case $o(\mathcal{X}_n) < o(\mathcal{X}_1)$ and we find the inductive path by searching along successor edges starting from $\mathcal{X}_1$.

2. G contains a reverse inductive path $\mathcal{X}_1 \rightarrow^* \mathcal{X}_n$ and we add edge $\mathcal{X}_n \rightarrow \mathcal{X}_1$. In this case $o(\mathcal{X}_1) < o(\mathcal{X}_n)$ and we find the reverse inductive path by searching along predecessor edges starting from $\mathcal{X}_n$.

Clearly cycles of length 2 are always detected. For larger cycles, this approach seems overly restrictive, since the probability given a random index assignment of closing one of the above cycles decreases exponentially in the size of the cycle. However, the inductive transitive closure rule (ITCR) used to close the constraint graph adds some transitive edges between variables. In fact, if $\mathcal{X}_1..\mathcal{X}_n$ form a cycle that has not been detected with the above strategy, then ITCR adds the transitive edge $\mathcal{X}_{j-1} \rightarrow \mathcal{X}_{j+1}$, through $\mathcal{X}_j$, where $\mathcal{X}_j$ is the variable with maximum index on the cycle. As a result, the cycle length is shortened by one edge, giving cycle detection another chance. In the worst case, a cycle of length 2 will be found between the two variables of minimum index on the cycle. Thus, inductive transitive closure and the above cycle detection mechanism always finds a sub-cycle of every strongly connected component in the graph.

We illustrate this mechanism for cycles of length 3. Every 3-cycle has the form of one of the cycles in Figure 7.4. In the figure, the variable order assumed is given by the variable indices. Variable $\mathcal{X}_3$ is thus the variable with maximum index on these cycles. Consider the 3-cycle on the left of Figure 7.4. Since $\mathcal{X}_2 \subseteq_s \mathcal{X}_3$ is an L-inductive constraint and $\mathcal{X}_3 \subseteq_s \mathcal{X}_1$ is an R-inductive constraint, inductive transitive closure adds the transitive edge $\mathcal{X}_2 \longrightarrow \mathcal{X}_1$ to the graph. This edge addition generates a 2-cycle $\mathcal{X}_1, \mathcal{X}_2$ which is de-

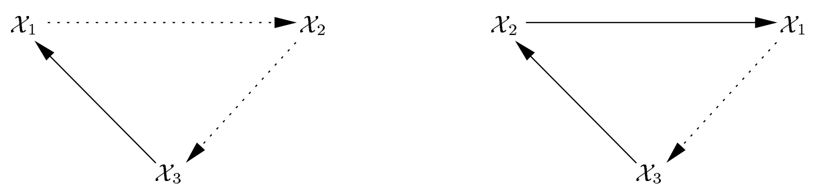

Figure 7.4: Possible 3-cycles

tected. Similarly, inductive transitive closure adds the edge $\mathcal{X}_1 \dashrightarrow \mathcal{X}_2$ to the right graph of Figure 7.4, forming a 2-cycle that is detected.

Collapsing Cycles

Once a cycle $\mathcal{X}_1..\mathcal{X}_n$ is found, it can be collapsed to avoid performing redundant work on the cycle's edges in the future. Collapsing the cycle requires choosing a representative $\mathcal{X}_r$ among $\mathcal{X}_1..\mathcal{X}_n$ and introducing alias edges from the remaining variables on the cycle to the representative. Furthermore, the lower bounds of each variable on the cycle must be added as a lower bound constraint on $\mathcal{X}_r$ and similarly for the upper bounds.

The choice of the representative $\mathcal{X}_r$ for inductive graphs is given by the invariant that alias edges must be inductive, *i.e.*, $\mathcal{X} \overset{=}{\longrightarrow} \mathcal{X}_r$ implies that the index of $\mathcal{X}$ is larger than $o(\mathcal{X}_r)$. Thus the representative must be the variable with minimum index on the detected cycle.

For the cycles we detect with the above strategy, the representative is the lower indexed variable of the edge $\mathcal{X}_n \longrightarrow \mathcal{X}_1$ that triggered the detection. Furthermore, there is an important optimization that we can do when adding the bounds of the variables on the cycle to the representative. If $\mathcal{X}_1..\mathcal{X}_n$ is the cycle we detected while adding edge $\mathcal{X}_n \longrightarrow \mathcal{X}_1$, then either $o(\mathcal{X}_n) < o(\mathcal{X}_1)$, in which case $\mathcal{X}_n$ is the representative and the lower bounds of the variables on the cycle are already present on $\mathcal{X}_n$ or $o(\mathcal{X}_1) < o(\mathcal{X}_n)$, in which case $\mathcal{X}_1$ is the representative and the upper bounds of the variables on the cycle are already present on $\mathcal{X}_1$. The reasoning is as follows:

- If $o(\mathcal{X}_n) < o(\mathcal{X}_1)$ then there is an inductive path $\mathcal{X}_1 \longrightarrow^* \mathcal{X}_n$ and $\mathcal{X}_n$ is the variable with minimum index on the cycle. The inductive path implies that for any edge $E \xrightarrow{M,C} \mathcal{X}_j$ of a source E to a variable $\mathcal{X}_j$ on the cycle, there is already an edge $E \xrightarrow{M,C} \mathcal{X}_n$ in the graph (Section 6.2).

  Similarly, for any edge $\mathcal{Z} \xrightarrow{M,C} \mathcal{X}_j$ where $o(\mathcal{Z}) < o(\mathcal{X}_n)$ the transitive closure has already added the edge $\mathcal{Z} \xrightarrow{M,C} \mathcal{X}_n$. For any edge $\mathcal{Z} \xrightarrow{M,C} \mathcal{X}_j$ where $o(\mathcal{Z}) < o(\mathcal{X}_j)$, but $o(\mathcal{Z}) > o(\mathcal{X}_n)$, there exists a variable $\mathcal{X}_i$ on the cycle with index larger or equal to $o(\mathcal{X}_n)$ and index less than $o(\mathcal{X}_j)$, such that the edge $\mathcal{Z} \xrightarrow{M,C} \mathcal{X}_i$ is R-inductive, *i.e.*, represented as an upper bound on $\mathcal{Z}$. Since $\mathcal{X}_i$ is aliased to $\mathcal{X}_n$ after the cycle is collapsed, it is not necessary to add the R-inductive edge $\mathcal{Z} \xrightarrow{M,C} \mathcal{X}_n$ involving the representative explicitly.

The above discussion establishes that when collapsing a cycle involving an inductive path to the minimum indexed variable, only the upper bounds of the variables on the cycle must be added to the representative. The lower bounds are already present.

- otherwise, $o(\mathcal{X}_1) < o(\mathcal{X}_n)$ in which case there is a reverse inductive path $\mathcal{X}_1 \rightarrow^* \mathcal{X}_n$ and $\mathcal{X}_1$ is the variable with minimum index on the cycle. Analogously to the above case one can establish that all upper bounds of variables on the cycle are already added as upper bound constraints on $\mathcal{X}_1$ and only lower bounds must be added to $\mathcal{X}_1$.

On Searching for Cycles

The choice of the minimum indexed variable as the representative for the cycle was given by our invariant on inductive alias edges. Here we argue that this choice is actually crucial to keep the cycle detection cost low. Consider for each variable $\mathcal{X}$ in a constraint graph G the set of *reachable successors* and the set of *reachable predecessors* defined as follows. A variable $\mathcal{Y}$ is a reachable successor of $\mathcal{X}$ if $o(\mathcal{Y}) < o(\mathcal{X})$ and there is a successor edge $\mathcal{X} \rightarrow \mathcal{Y}$ or alias edge $\mathcal{X} \overset{=}{\rightarrow} \mathcal{Y}$ in G, or there exists a variable $\mathcal{Z}$ with index $o(\mathcal{Y}) < o(\mathcal{Z}) < o(\mathcal{X})$ and an edge $\mathcal{X} \rightarrow \mathcal{Z}$ or alias edge $\mathcal{X} \overset{=}{\rightarrow} \mathcal{Z}$, and $\mathcal{Y}$ is a reachable successor of $\mathcal{Z}$. Reachable predecessors are defined analogously, *i.e.*, a variable $\mathcal{Y}$ is a reachable predecessor of $\mathcal{X}$ if $o(\mathcal{Y}) < o(\mathcal{X})$ and there is a predecessor edge $\mathcal{Y} \rightarrow \mathcal{X}$ or alias edge $\mathcal{X} \overset{=}{\rightarrow} \mathcal{Y}$ in G, or there exists a variable $\mathcal{Z}$ with index $o(\mathcal{Y}) < o(\mathcal{Z}) < o(\mathcal{X})$ and an edge $\mathcal{Z} \rightarrow \mathcal{X}$ or alias edge $\mathcal{X} \overset{=}{\rightarrow} \mathcal{Z}$, and $\mathcal{Y}$ is a reachable predecessor of $\mathcal{Z}$. The cycle detection cost is essentially defined by the number of reachable successors and predecessors of each variable in the graph. When adding an edge $\mathcal{Y} \rightarrow \mathcal{X}$ and $\mathcal{X}$ is the higher indexed variable, cycle detection visits each variable $\mathcal{Z}$ of index equal to or higher than $o(\mathcal{Y})$ in the reachable successors of $\mathcal{X}$. Similarly, when adding an edge $\mathcal{X} \rightarrow \mathcal{Y}$ and $\mathcal{X}$ is the higher indexed variable, cycle detection visits each variable $\mathcal{Z}$ of index equal to or higher than $o(\mathcal{Y})$ in the reachable predecessors of a variable $\mathcal{X}$.

Collapsing cycles to the minimum indexed variable on the cycle decreases the size of the reachable predecessor or successor sets of the variables on the cycle. Suppose we detect a cycle $\mathcal{X}_1..\mathcal{X}_n$ when adding edge $\mathcal{X}_n \rightarrow \mathcal{X}_1$ and the index of $\mathcal{X}_1$ is less than the index of $\mathcal{X}_n$. Then variables $\mathcal{X}_1..\mathcal{X}_n$ form an anti-inductive path. We add alias edges from each $\mathcal{X}_i$ to $\mathcal{X}_1$. Since $\mathcal{X}_1$ is in the predecessor set of every $\mathcal{X}_i$ on the cycle, the set of predecessors of $\mathcal{X}_i$ is reduced to the set of predecessors of $\mathcal{X}_1$. Consider the predecessor edges that are added to $\mathcal{X}_1$ as a result of collapsing the cycle. These edges are the predecessor edges $\mathcal{Z} \rightarrow \mathcal{X}_i$ of variables $\mathcal{X}_i$ on the cycle where $o(\mathcal{Z}) < o(\mathcal{X}_1)$. These edges are also added as predecessor edges on $\mathcal{X}_1$ if we add the edge $\mathcal{X}_n \rightarrow \mathcal{X}_1$ instead of collapsing the cycle, since the inductive transitive closure will add edges $\mathcal{X}_i \rightarrow \mathcal{X}_1$ for each $\mathcal{X}_i$ on the cycle.

Now consider the reachable successors of each $\mathcal{X}_i$. After collapsing the cycle, the reachable successors of each $\mathcal{X}_i$ are equal to the reachable successors of $\mathcal{X}_1$. We've already seen that closing the graph under the edge $\mathcal{X}_n \rightarrow \mathcal{X}_1$ instead of collapsing the cycle adds edges $\mathcal{X}_i \rightarrow \mathcal{X}_1$ for each $\mathcal{X}_i$. Thus the set of reachable successors of each $\mathcal{X}_i$ includes $\mathcal{X}_1$ in that case. Collapsing the cycle can only make the set smaller.

The argument for the case where we collapse a cycle $\mathcal{X}_1..\mathcal{X}_n$ and $o(\mathcal{X}_1) > o(\mathcal{X}_n)$ is analogous. Now consider what happens, if we collapse the cycle to the highest indexed variable $\mathcal{X}_n$ instead of $\mathcal{X}_1$. Collapsing the cycle involves adding all successor edges of each $\mathcal{X}_i$ to $\mathcal{X}_n$. Since $o(\mathcal{X}_n) > o(\mathcal{X}_i)$, the reachable successors of $\mathcal{X}_n$ after collapsing the cycle is the union of reachable successors of any $\mathcal{X}_i$. This set is potentially very large. A similar argument applies to the reachable predecessors when collapsing the dual kind of cycles to the maximal index variable.

7.1.6 General Constraint Resolution Algorithm

This section briefly describes how constraints are transformed into inductive constraint graphs. The algorithm given incorporates cycle detection and alias edges, topics not discussed in Chapters 5 or 6.

Algorithm 7.3 Given a constraint system S, let the constraint graph G be initially empty. Apply the function `resolve` below to each constraint $C \implies E_1 \subseteq_s E_2$ in S.

```
resolve($C$,$E_1$,$E_2$)

  if $E_1 \subseteq_s E_2$ is L-inductive ($E_2 = \mathcal{X}_2$) then
    if $\mathcal{X}_2$ is aliased to $E_2'$, then
      resolve($C$,$E_1$,$E_2'$)
    else if $E_1$ is a variable $\mathcal{X}_1$ and $C$ is true then
      run cycle detection on $\mathcal{X}_1 \rightarrow \mathcal{X}_2$
      if there is a cycle $\mathcal{X}_{i_1}..\mathcal{X}_{i_n}$ then
        for each $\mathcal{X}_j \neq \mathcal{X}_1$ in $\mathcal{X}_{i_1}..\mathcal{X}_{i_n}$ do
          add an alias edge $\mathcal{X}_j \overset{=}{\rightarrow} \mathcal{X}_1$;
          add upper-bounds of $\mathcal{X}_j$ to $\mathcal{X}_1$
        endfor
      else
        update_lower_bound($C$,$\mathcal{X}_1$,$\mathcal{X}_2$)
      endif
    else ($E_1$ is a source)
      update_lower_bound($C$,$E_1$,$\mathcal{X}_2$)
    endif

  else if $E_1 \subseteq_s E_2$ is R-inductive ($E_1 = \mathcal{X}_1$) then
    if $\mathcal{X}_1$ is aliased to $E_1'$, then
      resolve($C$,$E_1'$,$E_2$)
    else if $E_2$ is a variable $\mathcal{X}_2$ and $C$ is true then
      run cycle detection on $\mathcal{X}_1 \rightarrow \mathcal{X}_2$
      if there is a cycle $\mathcal{X}_{i_1}..\mathcal{X}_{i_n}$ then
        for each $\mathcal{X}_j \neq \mathcal{X}_2$ in $\mathcal{X}_{i_1}..\mathcal{X}_{i_n}$ do
          add an alias edge $\mathcal{X}_j \overset{=}{\rightarrow} \mathcal{X}_2$;
          add lower-bounds of $\mathcal{X}_j$ to $\mathcal{X}_2$
        endfor
      else
        update_upper_bound($C$,$\mathcal{X}_1$,$\mathcal{X}_2$)
      endif
    else ($E_2$ is a sink)
      update_upper_bound($C$,$\mathcal{X}_1$,$E_2$)
      endif

    else ($E_1 \subseteq_s E_2$ is not inductive)
      apply a resolution rule of Chapter~5
      if the constraint is inconsistent, stop.
      else
         $E_1 \subseteq_s E_2$ is equivalent to a set of simpler
         constraints $C_1 \implies (E_{11} \subseteq_{s_1} E_{21}), \ldots, C_n \implies (E_{1n} \subseteq_{s_n} E_{2n})$
         for each $i = 1..n$ do
           resolve($C \wedge C_i$,$E_{i1}$,$E_{i2}$)
         endfor


  update_lower_bound($C$,$E$,$\mathcal{X}$)

    if $C \overset{l}{\implies} E$ not in lb($\mathcal{X}$) then
      add $C \overset{l}{\implies} E$ to lb($\mathcal{X}$)
      for each $U$ in ub($\mathcal{X}$) do
         resolve($C$,$E$,$U$)
      endfor
    endif


  update_upper_bound($C$,$\mathcal{X}$,$E$)

    if $C \overset{r}{\implies} E$ not in ub($\mathcal{X}$) then
      add $C \overset{r}{\implies} E$ to ub($\mathcal{X}$)
      for each $L$ in lb($\mathcal{X}$) do
         resolve($C$,$L$,$E$)
      endfor
    endif
```

Our restriction to inductive alias edges guarantees that the constraint $E_1 \subseteq_s E_2$ that results from expanding an aliased variable $\mathcal{X}$ has top-level variables with indices strictly lower than $o(\mathcal{X})$. Thus, if the constraint has any top-level variables left, further rewrite steps must eventually transform all constraints involving top-level variables into inductive constraints. For constraints without top-level variables either the constraint is trivially satisfied, the constraint has no solution, or one of the structural resolution rules 5.7, 5.59, 5.21, or 5.27 is applied, which generate constraints on sub-expressions of E_1 and E_2. The restriction to non-recursive aliases guarantees that any variables in the sub-expressions of E_1 and E_2 can be expanded as well without generating an infinite expansion.

In practice, BANE implements a specialized `resolve` function for each sort. The specific details of these functions are discussed in the following sections. The `resolve` functions are mutually recursive due to the need to invoke resolution of constraints of other sorts at sort interfaces. To avoid having to write the resolve functions as a set of mutually recursive functions in BANE (which would hinder extension and modularity of the code), constructor signatures contain a set of function pointers, akin to a dispatch table in object-oriented language implementations.

7.2 Set Sort

This section describes implementation features of BANE specific to the resolution of Set-constraints. The implementation of Set-constraint resolution corresponds closely to Algorithm 7.3 given above. Besides online cycle detection that is applicable to all sorts, BANE contains another novel technique called *projection merging* that improves the scaling behavior of Set-constraints even further.

We first introduce projections and then describe the projection merging algorithm.

7.2.1 Projections

Projections come up frequently in program analysis problems where they correspond to selecting a particular component of a data-structure.

A projection $\mathsf{c}^{-i}(E)$ denotes the set $\bigcup\{\mu[\![A_i]\!]\sigma \mid c(A_1,\ldots,A_{a(c)}) \subseteq E\}$. Projections are not primitive expressions in BANE, but can be expressed indirectly by a translation of a projection $\mathsf{c}^{-i}(E)$ into a fresh variable $\mathcal{X}$, along with the constraint $E \subseteq_{\mathsf{s}} \mathsf{Pat}[c(1^{\iota_1},\ldots,\mathcal{X},\ldots,1^{\iota_n}), c(1^{\iota_1},\ldots,1^{\iota_n})]$, where $c : \iota_1\cdots\iota_n \to \mathsf{s}$ and $\mathcal{X}$ appears in the ith position. The notation 1^{ι_j} stands for 1 if $\iota_j = t$, *i.e.*, c is covariant in the jth argument, and 0 if $\iota_j = \bar{t}$, *i.e.*, c is contravariant in the jth argument. If c is covariant in the projected position (i), the constraint on the fresh variable $\mathcal{X}$ only requires $\mathcal{X}$ to be a superset of the desired projection. In practice, this is not a limitation as long as $\mathcal{X}$ is only used in L-contexts. To see this, note that if $\mathcal{X}$ appears only in L-contexts, then every constraint on $\mathcal{X}$ is of the form $\mathcal{X} \subseteq_s E$, constraining $\mathcal{X}$ by an upper-bound. Thus, the lower-bounds on $\mathcal{X}$ arise solely from the projection. Similarly, if c is contravariant in the projected position (i), then the constraint on $\mathcal{X}$ only requires it to be a subset of the projection. Again, this is not a limitation as long as $\mathcal{X}$ is used only in R-contexts.

The resolution of the Pat constraint proceeds by filtering out all sets of the form $c(E_1,\ldots,E_n)$ from E and making them a subset of $c(1^{\iota_1},\ldots,\mathcal{X},\ldots,1^{\iota_n})$.

$$\begin{array}{rl} & E \subseteq_{\mathsf{s}} \mathsf{Pat}[c(1^{\iota_1},\ldots,\mathcal{X},\ldots,1^{\iota_n}), c(1^{\iota_1},\ldots,1^{\iota_n})] \\ \iff & E \cap c(1^{\iota_1},\ldots,1^{\iota_n}) \subseteq_{\mathsf{s}} c(1^{\iota_1},\ldots,\mathcal{X},\ldots,1^{\iota_n}) \end{array}$$

Given a constraint $c(E_1,\ldots,E_n) \subseteq E$, the following sequence of constraints is generated:

$$\begin{array}{rl} & c(E_1,\ldots,E_n) \cap c(1,\ldots,1) \subseteq_{\mathsf{s}} c(1^{\iota_1},\ldots,\mathcal{X},\ldots,1^{\iota_n}) \\ \iff & c(E_1,\ldots,E_n) \subseteq_{\mathsf{s}} c(1^{\iota_1},\ldots,\mathcal{X},\ldots,1^{\iota_n}) \\ \iff & E_i \subseteq_{\iota_i} \mathcal{X} \end{array}$$

The Pat formulation of projections imposes a fair amount of overhead. BANE thus introduces an abbreviation for projection patterns of the form $\mathsf{Pat}[c(1,\ldots,E,\ldots,1), c(1,\ldots,1)]$ with the syntax $\mathsf{ProjPat}(\mathsf{c},i,E)$, along with the resolution rule:

$$d(E_1,\ldots,E_{a(d)}) \subseteq \mathsf{ProjPat}(\mathsf{c},i,E) \iff \begin{cases} E_i \subseteq_{\iota_j} E & \text{if } c = d \\ 0 \subseteq_s 1 & \text{otherwise} \end{cases}$$

The resolution of the abbreviation avoids an intersection and only accesses the position that is being projected.

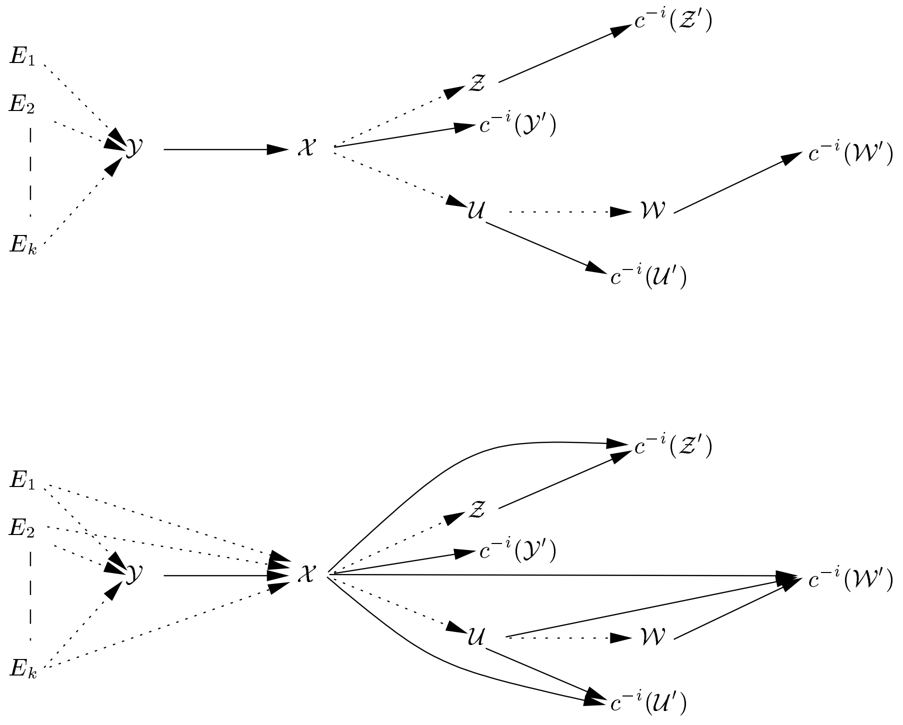

Figure 7.5: Transitive edges to projection patterns

7.2.2 Projection Merging

As shown in the previous section, a projection $\mathsf{c}^{-i}(E)$ is translated by BANE into a fresh variable $\mathcal{X}$, and the constraint $E \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{X})$. Since $\mathsf{ProjPat}(\mathsf{c}, i, \mathcal{X})$ is a sink, it may propagate backwards in the constraint graph along reverse inductive paths through the inductive transitive closure rule (ITCR). As a result, large numbers of these projection patterns may accumulate on a single variable. Together with large numbers of forward propagated lower bounds, this accumulation can cause a quadratic work and space blowup.

Figure 7.5 depicts the backward propagation of projection patterns. The graphs contain a number of sources $E_1..E_k$, variables, and projection patterns. Projection patterns $\mathsf{ProjPat}(\mathsf{c}, i, E)$ are abbreviated in the graphs by $c^{-i}(E)$. The top graph depicts the situation before transitive closure is applied. The relative order of the variables is again shown by using dotted or plain edges. The bottom graph shows some of the edges added by the inductive transitive closure rule. Note how the closure adds edges from variable $\mathcal{X}$ to all projection patterns, since there are reverse inductive paths from $\mathcal{X}$ to all variables with projection patterns in their upper bounds. On the other hand, the sources $E_1..E_k$ are propagated forward along edge $\mathcal{Y} \longrightarrow \mathcal{X}$. The bottom graph does not yet show the transitive constraints through variable $\mathcal{X}$ that are now added between all sources $E_1..E_k$ and all projection patterns. The apparent redundancy is that the projection of $\mathcal{X}$ w.r.t. c and argument i is computed multiple times, once for each projection pattern.

The basic idea behind projection merging is that a set of projection constraints

$$\{\mathcal{X} \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{c}, i, E_1), \mathcal{X} \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{c}, i, E_2), \ldots, \mathcal{X} \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{c}, i, E_n)\}$$

is satisfied if all constraints $c^{-1}(\mathcal{X}) \subseteq_{\iota_i} E_j$ for $j = 1..n$ are satisfied. Observe that for any solution σ, the projection of a fixed variable w.r.t. a fixed constructor c and index i is unique, and that we can replace the above system of constraints with the following system:

$$\{\mathcal{X} \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{X}^1), \mathcal{X}^1 \subseteq_{\iota_i} E_1, \mathcal{X}^1 \subseteq_{\iota_i} E_2, \ldots, \mathcal{X}^1 \subseteq_{\iota_n} E_n\}$$

where $\mathcal{X}^1$ is a fresh variable generated as a function of $\mathcal{X}$. The number of constraints is the same, but consider the transitive constraints that ensue. Given lower-bounds $\mathcal{Y}_1 \subseteq \mathcal{X}, \ldots, \mathcal{Y}_k \subseteq \mathcal{X}$, the original system propagates the n projection patterns to all $\mathcal{Y}_i$ where $o(\mathcal{Y}_i) < o(\mathcal{X})$. In the new constraint system, the only projection pattern propagated to each $\mathcal{Y}_i$ is $\mathsf{ProjPat}(\mathsf{c}, i, \mathcal{X}^1)$.

This transformation introduces fresh variables. Consequently, the termination of the resolution algorithm must be proven with the addition of this new rule. We first need to be more precise in the formulation of the new resolution rule. Assume that each variable $\mathcal{X}$ is part of a family $\mathcal{X}^s$, where s is a sequence of constructor-index pairs.

$$S \cup \{\mathcal{X}^s \subseteq \mathsf{ProjPat}(\mathsf{c}, i, E)\} \iff S \cup \{\mathcal{X}^s \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{X}^{s(ci)}), \mathcal{X}^{s(ci)} \subseteq E\} \qquad (7.1)$$

This rule applies *before* any transitive rules so that $\mathsf{ProjPat}(\mathsf{c}, i, E)$ is not propagated. Note that since the rule generates the variable $\mathcal{X}^{s(ci)}$ as a function of $\mathcal{X}$, constructor c and index i, there is only a single projection pattern per variable, constructor, index triple.

For the proof of termination, we restrict ourselves to the case where projection expressions only contain variables $\mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y})$. The following invariant relates the indices of variables appearing in projection constraints:

Lemma 7.4 *Let S_0 be a system of initial constraints. All variables in S_0 are assumed to have indices ≤ 0. Further assume that variables generated by Rule 7.1 are assigned consecutive indices starting from 1. Let S be any system of constraints obtained from S_0 by applying some of the resolution rules. The relation $(o(\mathcal{X}) > 0 \vee o(\mathcal{Y}) > 0) \implies o(\mathcal{X}) < o(\mathcal{Y})$ holds for any constraint $\mathcal{X} \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y})$ in S.*

Proof: Note that the lemma holds for S_0, since $o(\mathcal{X}) \leq 0$ for all $\mathcal{X}$. We show that any new constraints involving a projection maintain the invariant.

Rule 7.1 We have $o(\mathcal{X}^s) < o(\mathcal{X}^{s(ci)})$ for the constraint $\mathcal{X}^s \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{X}^{s(ci)})$ generated from $\mathcal{X}^s \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y})$, since $\mathcal{X}^{s(ci)}$ is generated after $\mathcal{X}$.

Transitivity: From $\mathcal{X} \subseteq \mathcal{Z} \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y})$ transitivity generates $\mathcal{X} \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y})$ whenever $o(\mathcal{X}) < o(\mathcal{Z})$. By assumption, $(o(\mathcal{Z}) > 0 \vee o(\mathcal{Y}) > 0) \implies o(\mathcal{Z}) < o(\mathcal{Y})$, thus $(o(\mathcal{X}) > 0 \vee o(\mathcal{Y}) > 0) \implies o(\mathcal{X}) < o(\mathcal{Y})$.□

□ Say that $\mathcal{Y}$ is a parent of $\mathcal{X}^{s(ci)}$ whenever Rule 7.1 is applied with $E = \mathcal{Y}$. Let the ancestor relation $\mathsf{anc}(\mathcal{X}, \mathcal{Y}^{s(ci)})$ be the transitive closure of the parent relation.

Lemma 7.5 *Given a constraint* $\mathcal{X} \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y}^{s(ci)})$ *it holds that* $o(\mathcal{X}) < o(\mathcal{Z})$ *for any ancestor* $\mathcal{Z}$ *of* $\mathcal{Y}^{s(ci)}$ *with* $o(\mathcal{Z}) > 0$.

Proof: By induction on the length of the ancestor relation anc^n.

Base: $\mathcal{Z}$ is a parent of $\mathcal{Y}^{s(ci)}$: Then $\mathcal{Y}^s \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Z})$, and $o(\mathcal{Y}^s) < o(\mathcal{Z})$ by Lemma 7.4. If $\mathcal{X} \neq \mathcal{Y}^s$, then we must have $\mathcal{X} \subseteq \mathcal{Y}^s$ and also $o(\mathcal{X}) < o(\mathcal{Y}^s)$ for transitivity to apply. In any case $o(\mathcal{X}) < o(\mathcal{Z})$.

Induction: Assume there exists an ancestor $\mathcal{Z}$, such that $\mathsf{anc}^{n+1}(\mathcal{Z}, \mathcal{X})$, and not $\mathsf{anc}^n(\mathcal{Z}, \mathcal{X})$. Then there exists a parent $\mathcal{W}^{t(dj)}$ of $\mathcal{Y}^{s(ci)}$ such that $\mathsf{anc}^n(\mathcal{W}^{t(dj)}, \mathcal{Z})$. The constraint $\mathcal{Y}^s \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{W}^{t(dj)})$ must result from transitivity of $\mathcal{Y}^s \subseteq \mathcal{W}^t$ and the constraint $\mathcal{W}^t \subseteq \mathsf{ProjPat}(\mathsf{d}, j, \mathcal{W}^{t(dj)})$. This implies $o(\mathcal{Y}^s) < o(\mathcal{W}^t)$ and by induction we have $o(\mathcal{W}^t) < o(\mathcal{Z})$ and thus $o(\mathcal{Y}^s) < o(\mathcal{Z})$. If $\mathcal{X} \neq \mathcal{Y}^s$ then $\mathcal{X} \subseteq \mathcal{Y}^s$ and $o(\mathcal{X}) < o(\mathcal{Y}^s)$. In either case, $o(\mathcal{X}) < o(\mathcal{Z})$.□

□ Now let V_0 be the set of variables appearing in the original

finite constraint system S_0, and let I be the set of *initial* generated variables $\mathcal{X}$, *i.e.*, the set of variables $\{\mathcal{X} \mid \exists \mathcal{Y} \in V_0, \text{s.t. } \mathcal{Y} \text{ parent of } \mathcal{X}\}$. Since for each occurrence of an expression $\mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y})$ in the original constraints, there is exactly one variable $\mathcal{X}$, such that $\mathcal{Y}$ is a parent of $\mathcal{X}$, the set I is finite. Also note that every generated variable has an ancestor in I. Now consider any constraint $\mathcal{X} \subseteq \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Z})$ obtained during resolution. From Lemma 7.5, we know that the index of $\mathcal{X}$ is less than the index of $\mathcal{Z}$ and all ancestors of $\mathcal{Z}$, and thus there exists $\mathcal{W} \in I$, such that $o(\mathcal{X}) < o(\mathcal{W})$. But this implies that $\mathcal{X} \in I \cup V_0$. Termination follows directly.

Theorem 7.6 (Termination of Projection Merging) *The number of variables generated through projection merging during resolution is bounded by* $2A|S_0| \cdot |C|$, *where* C *is the set of constructors appearing in* S_0 *and* A *is the maximum arity of any constructor in* C.

Note that termination is also guaranteed in the case where the resolution of other sorts introduces fresh Set-variables (for example Rule 5.13), since any fresh variable $\mathcal{X}$ will have index $o(\mathcal{X}) > 0$ and the only constraints the resolution of other sorts can add on $\mathcal{X}$ is a variable constraint $\mathcal{X} \subseteq_{\mathsf{s}} \mathcal{Y}$ or $\mathcal{Y} \subseteq_{\mathsf{s}} \mathcal{X}$. Then Lemma 7.4 still holds, since a constraint $\mathcal{X} \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Z})$ on the fresh variable $\mathcal{X}$ can only be generated by applying the inductive transitive closure to $\mathcal{X} \subseteq_{\mathsf{s}} \mathcal{Y}$ and $\mathcal{Y} \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Z})$, in which case $\mathcal{Y}$ has higher index than $o(\mathcal{X})$.

Section 9.3 shows the impact of projection merging on constraint resolution times. For very large constraint problems, projection merging can improve resolution times by an order of magnitude.

7.3 Term Sort

BANE's implementation of Term-constraints makes two simplifying assumptions.

- Solutions where a Term-variable must be $\top_{\mathsf{t}}$ are not of interest and can be discarded.

- Conditions on conditional constraints are assumed to be eventually true in all solutions and are discarded.

These two assumptions enable a faster constraint resolution algorithm which is essentially the algorithm proposed by Steensgaard for *conditional unification* [79].

Note that the above assumptions retain soundness of the resolution, but discard completeness in favor of efficiency. These assumptions represent only the current implementation of BANE and are not fundamental limitations of the mixed constraint approach. To avoid unexpected effects due to the assumption on conditions, BANE resolves the condition status of any conditional Term-constraint using the algorithm given in Section 6.2.2 and issues a warning if the condition cannot be proven true in all solutions.

To take advantage of Steensgaard's algorithm, Term-constraints are not represented in inductive form. Instead, constraints of the form $\mathcal{X} \subseteq_{\mathsf{t}} E$ are always represented as a successor edge in the graph, by adding E to the upper bound set of $\mathcal{X}$ ($\mathtt{ub}(\mathcal{X})$). The algorithm essentially assumes that each variable $\mathcal{X}$ is $\perp$ until proven otherwise by a lower bound of the form $c(\dots)$. A constraint of the form $c(E_1,\dots,E_n) \subseteq_{\mathsf{t}} \mathcal{X}$ causes $\mathcal{X}$ to be equal to $c(E_1,\dots,E_n)$ by the second assumption above. The constraint resolution thus replaces the `varinfo` of $\mathcal{X}$ with an alias edge to $c(E_1,\dots,E_n)$ and adds the constraints $c(E_1,\dots,E_n) \subseteq_{\mathsf{t}} U$ for all $U \in \mathtt{ub}(\mathcal{X})$. An occurs check guarantees that the alias is non-recursive.[1]

Figure 7.6 gives the pseudo-code for Term-constraint resolution. Note that the only transitive constraints are generated when an alias edge is introduced. No lower bounds are ever added to any variable. For a variable $\mathcal{X}$, the algorithm traverses each successor edge of $\mathcal{X}$ at most once, *i.e.*, exactly when an alias edge for $\mathcal{X}$ is introduced.

The non-inductive representation defies the cycle detection strategy outlined earlier. If desired, inductive edges can be added to lower bounds solely for cycle detection. So far, we have not encountered the need to add cycle elimination to Term-constraints.

Constraints of the form $c(E_1,\dots,E_n) \subseteq_{\mathsf{t}} c(E'_1,\dots,E'_n)$ are simplified by generating equality constraints $E_i =_{\iota_i} E'_i$ for $i = 1..n$. The equality constraints between E_i and E'_i are implemented using Robinson's unification algorithm whenever possible [75]. Since Term and FlowTerm expressions can be identified structurally, unification applies to them in all cases. For Set-expressions, equality can be implemented using unification as long as both sides are structurally equal up to variables. When the structure doesn't match, symmetric constraints $E_i \subseteq_s E'_i$ and $E'_i \subseteq_s E_i$ are asserted.

7.4 FlowTerm Sort

BANE's implementation of FlowTerm-constraints makes similar simplifying assumptions as for Term constraints.

- Solutions where a FlowTerm-variable must be $\perp_{\mathsf{ft}}$ or $\top_{\mathsf{ft}}$ are deemed uninteresting and are discarded.

[1] If recursive unification is used for the structural constraints and no mixed constructors of sort Term are used, then recursive aliases can be permitted.

```
term-resolve($C$,$E_1$,$E_2$)

  if $C$ not true in current constraints, issue warning endif
  if $E_1 = \mathcal{X}_1$ then
    if $\mathcal{X}_1$ is aliased to $E_1'$ then
      term-resolve(true,$E_1'$,$E_2$)
    else
      add $E_2$ to ub($\mathcal{X}$)
    endif
  else ($E_1 = c(E_{11}, \ldots, E_{1n})$)
    if $E_2 = \mathcal{X}_2$ then
      if $\mathcal{X}_2$ is aliased to $E_2'$ then
        term-resolve(true,$E_1$,$E_2'$)
      else
        add alias edge $\mathcal{X}_2 \overset{=}{\longrightarrow} E_1$
        for each $U \in$ ub($\mathcal{X}_2$) do
          term-resolve(true,$E_1$,$U$)
        endif
      endif
    else  ($E_2 = d(E_{21}, \ldots, E_{2m})$)
      if $c \neq d$ stop, no solution
      else
        for each $i = 1..n$
          $s$ is sort of $i$th argument of $c$
          dispatch-unify($E_{1i}$,$E_{2i}$) to sort $s$
        endfor
      endif
    endif
  endif
```

Figure 7.6: Specialized Term resolution

.

- Conditions on conditional constraints are assumed to be true in all solutions and are discarded.

Again, these assumptions present implementation choices and are not fundamental. As for Term-constraints, BANE issues a warning for conditional FlowTerm-constraints whose condition cannot be proven true.

The first assumption above (discarding $\bot_{\mathsf{ft}}$ and $\top_{\mathsf{ft}}$ solutions) enables the use of the following resolution rules:

$$\begin{aligned} c(E_1,\ldots,E_n) \subseteq_{\mathsf{ft}} \mathcal{X} &\iff \mathcal{X} = c(\mathcal{X}^{c1},\ldots,\mathcal{X}^{cn}) \ \wedge\ E_i \subseteq_{\iota_i} \mathcal{X}^{ci} \\ \mathcal{X} \subseteq_{\mathsf{ft}} c(E_1,\ldots,E_n) &\iff \mathcal{X} = c(\mathcal{X}^{c1},\ldots,\mathcal{X}^{cn}) \ \wedge\ \mathcal{X}^{ci} \subseteq_{\iota_i} E_i \end{aligned}$$

where the variables $\mathcal{X}^{ci}$ are fresh variables similar to the fresh variables introduced during projection merging. These variables are uniquely determined by variable $\mathcal{X}$, constructor c, and index i. The first resolution rule above is justified by the fact that in all solutions where $\mathcal{X} \neq \top_{\mathsf{ft}}$, $\mathcal{X}$ must be of the form $c(\mathcal{X}^{c1},\ldots,\mathcal{X}^{cn})$. The second rule is justified similarly.

The equality constraint on $\mathcal{X}$ can be represented using an alias edge as outlined in Figure 7.7. Termination of the resolution using the specialized rules can be proven similarly to the case of projection merging, by showing that only a finite number of fresh variables are generated. Termination hinges crucially on the fact that we expand the aliasing of a variable $\mathcal{X}$ only when an inductive constraint on $\mathcal{X}$ is found and not when $\mathcal{X}$ is merely an upper or lower bound on some other higher indexed variable $\mathcal{Y}$. Consider the example constraint $c(\mathcal{X}) \subseteq_{\mathsf{ft}} \mathcal{X}$ which forces a naive algorithm into an infinite loop. The resolution of this constraint proceeds as follows: (assuming no other initial constraints)

- The constraint is L-inductive, so the algorithm in Figure 7.7 generates the fresh variable $\mathcal{X}^{c1}$ and adds the alias edge $\mathcal{X} \xrightarrow{=} c(\mathcal{X}^{c1})$.
- Then the constraint $\mathcal{X} \subseteq_{\mathsf{ft}} \mathcal{X}^{c1}$ is generated. Since $\mathcal{X}^{c1}$ was generated after $\mathcal{X}$, we have $o(\mathcal{X}^{c1}) > o(\mathcal{X})$ and the constraint is L-inductive. Thus the algorithm checks the aliasing of $\mathcal{X}^{c1}$, which is not aliased and we add $\mathcal{X}$ to the lower bounds $\mathtt{lb}(\mathcal{X}^{c1})$ of $\mathcal{X}^{c1}$.

If the algorithm expanded the aliasing of the lower indexed variable $\mathcal{X}$ instead, the algorithm would enter an infinite loop.

The intuitive termination argument is that whenever a variable $\mathcal{X}$ is found to be of the form $c(\mathcal{X}^{c1},\ldots,\mathcal{X}^{cn})$, the addition of the alias edge $\mathcal{X} \xrightarrow{=} c(\mathcal{X}^{c1},\ldots,\mathcal{X}^{cn})$ only causes variables $\mathcal{Y}$ with lower index than $\mathcal{X}$ to be expanded as well, since all upper and lower bounds of $\mathcal{X}$ in inductive form have lower indices.

The above algorithm is novel to the best of our knowledge. Similar previously published constraint resolution algorithms based on the expansion of a variable $\mathcal{X}$ to $c(\mathcal{X}^{c1},\ldots,\mathcal{X}^{cn})$ are restricted to non-recursive constraints [42, 64].

7.5 Row Sorts

BANE implements only one kind of Row-variable for each of the three Row-sorts $\mathbf{r}(\mathbf{s})$,$\mathbf{r}(\mathbf{t})$, and $\mathbf{r}(\mathbf{ft})$, namely closed Row-variables. Minimal and maximal Row-expressions can still be

```
flowterm-resolve($C$,$E_1$,$E_2$)

  if $C$ not true in current constraints, issue warning endif
  if $E_1 \subseteq_{\mathsf{ft}} E_2$ is L-inductive then
    $E_2$ is a variable $\mathcal{X}_2$
    if $\mathcal{X}_2$ is aliased to $E_2'$ then
      term-resolve(true,$E_1$,$E_2'$)
    else
      if $E_1 = c(E_{11}, \ldots, E_{1n})$ then
        add alias edge $\mathcal{X}_2 \overset{=}{\longrightarrow} c(\mathcal{X}^{c1}, \ldots, \mathcal{X}^{cn})$ and transitive constraints
        for each $i = 1..n$
          $s$ is the sort of the $i$th argument of $c$
          dispatch-resolve(true,$E_{1i}$,$\mathcal{X}^{ci}$)
        endfor
      else
        update_lb(true,$E_1$,$\mathcal{X}$)
      endif
    endif

  else if $E_1 \subseteq_{\mathsf{ft}} E_2$ is R-inductive then
    $E_1$ is a variable $\mathcal{X}_1$
    if $\mathcal{X}_1$ is aliased to $E_1'$ then
      term-resolve(true,$E_1'$,$E_2$)
    else
      if $E_2 = c(E_{21}, \ldots, E_{2n})$ then
        add alias edge $\mathcal{X}_1 \overset{=}{\longrightarrow} c(\mathcal{X}^{c1}, \ldots, \mathcal{X}^{cn})$ and transitive constraints
        for each $i = 1..n$
          $s$ is the sort of the $i$th argument of $c$
          dispatch-resolve(true,$\mathcal{X}^{ci}$,$E_{2i}$)
        endfor
      else
        update_ub(true,$\mathcal{X}$,$E_2$)
      endif
    endif

  else ($E_1 = c(E_{11}, \ldots, E_{1n})$ and $E_2 = d(E_{21}, \ldots, E_{2m})$)
    if $c \neq d$ stop, no solution
    else
      for each $i = 1..n$
        $s$ is sort of $i$th argument of $c$
        dispatch-resolve(true,$E_{1i}$,$E_{2i}$) to sort $s$
      endfor
    endif
  endif
```

Figure 7.7: Specialized FlowTerm-resolution

used, but are always of the forms $\langle l : E_l \rangle_A \circ 0$ and $\langle l : E_l \rangle_A \circ 1$. Conditional constraints are handled as for Term and FlowTerm-constraints.

Besides these assumptions, the resolution of Row-constraints is implemented using the general resolution algorithm given above and the resolution rules described in Section 5.4. The only implementation optimization is the use of an alias edge when a variable $\mathcal{X}$ is split into $\langle l : \mathcal{X}_l \rangle_A \mathcal{X}'$. Note that the absence of minimal and maximal Row-variables obviates the need to represent Row-masks.

Domain constraints are not currently implemented in BANE. With the absence of minimal and maximal Row-variables, termination can be guaranteed if the original constraints observe the following invariant: If a Row-variable $\mathcal{X}$ appears in two composition expressions $\langle l : E_l \rangle_A \circ \mathcal{X}$ and $\langle l : E_l' \rangle_B \circ \mathcal{X}$ in the original constraints, then $A = B$. This is an idea of Rémy (for example [70]).

7.6 Polymorphic Analysis

BANE provides support for polymorphic constraint-based program analysis through polymorphic constrained mixed expressions. Polymorphic constrained expressions have the form

$$\forall \mathcal{X}_1..\mathcal{X}_n.E \backslash S$$

where $\mathcal{X}_1..\mathcal{X}_n$ are the *quantified variables*, E is the underlying mixed expression, and S is the system of constraints of the polymorphic expression.

Polymorphic constrained expressions are the natural generalization of polymorphic types. Polymorphic types arise in languages supporting parametric polymorphism. For example, in ML one can write a function `length` that returns the length of lists containing any type of elements. The type of such a `length` function is $\forall \alpha.\alpha\ \mathsf{list} \to \mathsf{int}$, where α is a type variable standing for the type of elements of the list. The type of the `length` function is polymorphic in the element type α. One way to think about the type of the length function is that it has all types $\tau\ \mathsf{list} \to \mathsf{int}$ for any type τ. The justification for the polymorphic type of `length` is that the length function does not assume anything about the element types. In other words, the type of the elements is not constrained. One thus refers to the quantified variables of such polymorphic type as *universally* quantified variables.

In languages with parametric polymorphism and type inference, polymorphic types are usually inferred for *let-bound* variables, *i.e.*, variables `x` bound in expressions of the form

```
let x = e
in
   e'
end
```

Here, the `x` is the let-bound program variable, e is the let-bound expression, and e' the let-body. The standard semantics of such `let`-expressions is that `x` is bound to the value v which is the result of evaluating e. Occurrences of `x` within the let-body e' then refer to the value v. The *let-expansion* of the body e' is the expression $e'[e/\mathtt{x}]$, *i.e.*, e' where all occurrences of `x` are replaced with e (while avoiding name capture). In conventional programming language semantics, the let-expansion is equivalent to the `let`-expression provided that e has no side-effects. Type systems for such languages generally state that a `let`-expression is well-typed if and only if the let-expansion is well-typed (assuming e has no side effects).

The basic idea of polymorphic type inference is to infer that the let-expansion of a `let`-expression is well-typed without repeatedly performing the type inference for the let-bound expression e at all occurrences of `x` in e'. This is achieved by inferring the most general polymorphic type $\forall.\vec{\alpha}.\tau$ for e and instantiating this type to possibly different types τ_i at each occurrence of `x` within e'. Instantiating a polymorphic type means replacing the quantified type variables $\vec{\alpha}$ with actual types in the underlying type τ. Coming back to our example of the length function, instantiating the polymorphic type $\forall\alpha.\alpha\ \mathsf{list} \rightarrow \mathsf{int}$ corresponds to selecting the appropriate type $\tau\ \mathsf{list} \rightarrow \mathsf{int}$ among all types of `length` such that τ is the type of the elements to which `length` is actually applied in the particular occurrence.

For improved accuracy, program analyses are sometimes performed on the let-expansion of a program. The let-expansion of a program may in principle be exponential in size w.r.t. to the original program size, although in practice such blowups seem rare (at least for standard type systems). The idea of polymorphic type systems can be carried over to constraint-based program analysis. In this scenario, the analysis of the let-bound expression e may yield a mixed expression E along with a constraint system S. Let V be the set of variables in S that were generated as part of the analysis of e and are not shared with the analysis of any other part of the program. Call V the *local variables* of S. If we performed the same analysis on the let-expansion of the `let`-expression, we would infer a mixed expression E_i and constraint system S_i for every occurrence i of `x` in the body e'. Furthermore, E_i, S_i and E_j, S_j will differ in exactly the local variables only. Thus, we can achieve the same effect if we generate the quantified expression $\forall V.E\backslash S$ for the analysis of the let-bound expression e and instantiate the quantified expression at each occurrence of `x` in the body. In this case, instantiation refers to replacing each quantified variable $v \in V$ with a fresh variable in E and S. Note that this step is only valid if the program semantics are such that the let-expansion is actually equivalent to the original `let`-expression. Furthermore, the constraint system S inferred for the let-bound expression e must usually be consistent.

To support this style of polymorphic analysis, BANE provides the following features:

- Multiple constraint graphs and automatic tracking of local variables.
- Quantification and instantiation
- Constraint simplification

The next subsections deal with each of these aspects in turn.

7.6.1 Multiple Constraint Graphs

If a mixed expression E and associated constraints S are to be generalized it is convenient to keep the constraints S separate from other constraints S' that may arise in other parts of an analysis. To guarantee that constraints S have a solution, they need to be solved before quantification. Solving the constraints means building a closed inductive constraint graph for S. In order to keep several systems S and S' in solved form, BANE provides the ability to maintain several distinct constraint graphs.

BANE's approach to multiple constraint graphs is imperative. BANE has the notion of a *current constraint graph* (CCG). Constraints are always added to the current constraint graph. There are operations to create fresh empty constraint graphs and to make a constraint graph *current*. The information associated with a constraint graph is a set of variable-`varinfo` pairs, where the variable information associated with each variable $\mathcal{X}$ contains the predecessor and successor edges of $\mathcal{X}$ or an alias edge.

Each variable node has a special field called the *varinfo-cache*. A variable $\mathcal{X}$ is deemed *current*, if its varinfo-cache points to the varinfo data associated with $\mathcal{X}$ in the current constraint graph. The varinfo-cache enables direct access to the edges of the CCG through variable nodes without the need for a lookup operation. A constraint graph G is made current by making each variable of G current, *i.e.*, for each variable-varinfo pair $(\mathcal{X}, vi)$ in G the varinfo-cache of $\mathcal{X}$ is set to vi.

Each variable in a constraint graph G is either *local* or *free* w.r.t. to G. When a fresh variable $\mathcal{X}$ is created, it is added to the current constraint graph and is then local to the CCG. When a constraint $E \subseteq_s E'$ is to be added to the current constraint graph, each variable occurring within E or E' that is not current must be added to the CCG as a free variable with an initially empty varinfo structure. Thus the local variables of G are the variables created while G is the current constraint graph.

7.6.2 Quantification and Instantiation

BANE provides a function to form a polymorphic constrained expression $\forall V.E\backslash S$ from a given mixed expression E and the current constraint graph. The quantified variables V are the local variables of the CCG, and the constraints S are simply the constraints represented by the edges of the CCG.

It is not always desirable to quantify all local variables, for example variables that are later inspected to extract results from the analysis should not be quantified. Variables to be excluded from quantification can be explicitly mentioned during the quantification step.

Instantiation of a polymorphic constrained expression $\forall V.E\backslash S$ proceeds by creating a substitution σ from V to a set of fresh variables V', adding the constraint $\sigma(E) \subseteq_s \sigma(E')$ to the current constraint for each constraint $E \subseteq_s E'$ in S, and returning the instantiation $\sigma(E)$. The set of fresh variables V' are local to the current constraint graph.

7.6.3 Simplification

Naively quantifying mixed expressions and their constraint graphs obtained from the analysis of let-bound expressions and instantiating the resulting polymorphic expressions at each occurrence of the let-bound variable in the let-body as outlined at the beginning of this section is often not practical. The constraint graphs associated with polymorphic expressions may be large, containing many quantified variables. Since each instantiation generates fresh variables, it is fairly simple to obtain an exponential blowup of variables and graph edges.

This blowup is reduced by *simplifying* the set of constraints S extracted from the current constraint graph when quantifying over the local variables. Simplification of a polymorphic constraint system $\forall V.E \backslash S$ w.r.t. the set of variables V involves computing a hopefully smaller constraint system S' that is equivalent to S w.r.t. the free variables of $\forall V.E \backslash S$, *i.e.*, variables occurring in S or E but not in V. We say that S and S' are equivalent w.r.t. to a set of free variables F, if for any solution σ of S, there exists a solution σ' for S' that agrees with σ on F, and for any solution σ' of S' there exists a solution σ of S such that σ and σ' agree on F.

We make use of an alternative formulation of equivalent constraint systems. We characterize a constraint system S not by its solutions, but by the local (or quantified) variables and the possible consistent extensions of S. We say that S and S' are equivalent w.r.t. a set of *local* variables V, if for any set of constraints S'' such that no variable of V occurs in S'', the constraints $S \cup S''$ are consistent if and only if the constraints $S' \cup S''$ are consistent.

Simplification of constraint systems has been studied in the past by several researchers, including the author [24, 29, 72, 7, 58]. BANE takes a less sophisticated approach to constraint system simplification than for example Flanagan [29] or Pottier [72] in that no entailment relation of constraint systems is used. BANE only prunes unreachable variables and constraints, and performs *minimization-maximization.*

Reachability classifies each variable in the current constraint graph as *unreachable*, *L-reachable*, *R-reachable*, or both *L- and R-reachable.* We represent these four choices by two bits associated with each variable, an L-bit and an R-bit. The reachability of a variable $\mathcal{X}$ in a polymorphic constrained expression $\forall V.E \backslash S$ states in what contexts variable $\mathcal{X}$ may appear in any extension of the constraints S with S'' not involving any variables of V.

If a variable $\mathcal{X}$ is unreachable in a consistent system S, then no future constraints can further bound $\mathcal{X}$. Thus, $\mathcal{X}$ has no influence on whether an extension $S \cup S''$ is consistent or not. Thus, S is equivalent to S' where $\mathcal{X}$ and all edges involving $\mathcal{X}$ have been deleted.

If variable $\mathcal{X}$ is L-reachable, but not R-reachable, then future constraints can only add upper bounds on $\mathcal{X}$. In this case $\mathcal{X}$ may be minimized without affecting the consistency of any extensions $S \cup S''$. Similarly, if $\mathcal{X}$ is R-reachable, but not L-reachable, future constraints can only lower bound $\mathcal{X}$. In this case $\mathcal{X}$ may be maximized without affecting the consistency of any extensions $S \cup S''$.

The following algorithm computes the reachability of variables for a polymorphic constrained expression $\forall V.E \backslash S$ obtained from a constraint graph G with local variables V and expression E.

Algorithm 7.7 Let `L-mark` and `R-mark` be the two procedures defined below. Initially, mark the L- and R-bit of each free variable and call `L-mark` and `R-mark` on E. Then apply the following steps until no more variables can be marked.

- If $\mathcal{X}$ is L-reachable, then call `L-mark` on each lower bound L in `lb`$(\mathcal{X})$ or on E if $\mathcal{X}$ is aliased to E.

- If $\mathcal{X}$ is R-reachable, then call `R-mark` on each upper bound R in $\texttt{ub}(\mathcal{X})$ or on E if $\mathcal{X}$ is aliased to E.

where

```
L-mark($E$)
  case $E$ of
    $\mathcal{X}$ => mark L-bit of $\mathcal{X}$
  | $c(E_1, \ldots, E_n)$ =>
       for each $i = 1..n$ do
         if $c$ is a Term-constructor then
           call L-mark($E_i$) and R-mark($E_i$)
         else
           if $c$ is covariant in $i$ then
             call L-mark($E_i$)
           else ($c$ contravariant in $i$)
             call R-mark($E_i$)
           endif
         endif
       endfor
  | $\langle l : E_l \rangle_A \circ E$ =>
       call L-mark($E$);
       for each $l \in A$ do
         call L-mark($E_l$)
       endfor
  | $\mathcal{X} \cap M$ => call L-mark($\mathcal{X}$)
  | Pat$[E, M]$ => call L-mark($E$)
  | _ => ;
  endcase
```

and `R-mark` is the dual of `L-mark`.

If we know that each instantiation E' of the polymorphic constrained expression $\forall V.E \backslash S$ generated from G will only appear on the left of a constraint, then we don't need to call `R-mark` on E, and analogously if each instantiation only appears on the right.

The marking correctly captures the reachability of variables in the constraint graph since `L-mark`(E) correctly classifies the reachability of variables during the resolution of a constraint $E \subseteq_s E'$ using the structural resolution rules, and similarly for `R-mark`. Furthermore, the repeat steps correspond to applying the inductive transitive closure rule.

To minimize a variable $\mathcal{X}$, add an alias edge $\mathcal{X} \overset{=}{\longrightarrow} \bigsqcup \texttt{lb}(\mathcal{X})$ from $\mathcal{X}$ to the union of its lower bounds, provided this union is expressible in the sort of $\mathcal{X}$ and $\bigsqcup \texttt{lb}(\mathcal{X})$ is not recursive in $\mathcal{X}$. Minimization is always possible for Set-variables. Term variables never have lower bounds, so they can be minimized to 0. FlowTerm and Row-variables can be minimized only if their lower bound is empty or contains a single expression E. In the first case the variable is aliased to 0, in the second case to E.

Similarly, to maximize a variable $\mathcal{X}$, add an alias edge $\mathcal{X} \overset{=}{\longrightarrow} \bigsqcap \texttt{ub}(\mathcal{X})$ from $\mathcal{X}$ to the intersection of its upper bounds, provided this intersection is expressible in the sort of $\mathcal{X}$ and $\bigsqcap \texttt{ub}(\mathcal{X})$ is not recursive in $\mathcal{X}$. Maximization is always possible for Set-variables. Term, FlowTerm and Row-variables can be maximized only if their upper bound is empty or contains a single expression E. In the first case the variable is aliased to 1, in the second case to E.

Chapter 8

Example Analyses

This chapter describes two example program analyses expressed using mixed constraints and implemented using the BANE library. The first analysis is a points-to analysis for C based on Andersen's algorithm [8]. It uses inclusion constraints between Set-expressions and Row-expressions. The second analysis is an exception inference for ML developed by the author [25, 26]. We study three distinct versions of this analysis. The versions differ in their use of particular sorts and constraints. Experiments involving these analyses are used in the next chapter to evaluate the scaling ability of BANE and the impact of particular implementation techniques.

8.1 Points-to Analysis for C

Points-to analysis for C computes for each expression in a program a set of abstract memory locations (variables and heap) that the expression may evaluate to. From this information, a *points-to* graph can be derived. Graph nodes represent abstract memory locations and edges represent points-to relations. More precisely, an edge from a location l_1 to a location l_2 in the points-to graph states that the abstract location l_1 may at some point during evaluation of the program contain a pointer to abstract location l_2. Abstract locations refer to program variables or syntactic occurrences of applications of memory allocation functions (for example `malloc`). Figure 8.1 shows the points-to graph computed by Andersen's analysis for a simple C program.

Points-to analysis has been well studied (for example [88, 15, 53, 44, 23, 79]). Here we examine the particular points-to analysis proposed by Andersen [8], which is based on Set-constraints. Our interest in this study is not primarily the precision of the points-to information computed, but the execution time required to compute the information. Past implementations of points-to analyses based on Set-constraints suggest that the approach does not scale to large programs. As we will show through extensive measurements in Chapter 9, points-to analysis derived by solving a system of set constraints is practical even for very large programs.

Andersen's points-to analysis is formulated as a non-standard type inference system, based on a collection of constants representing abstract locations $\{l_1, .., l_n\}$ and a collection of Set-variables, one per location $\{\mathcal{X}_1, .., \mathcal{X}_n\}$. Set-constraints are used to model

```
a = &b;
a = &c;
*a = &d;
```

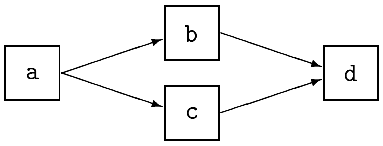

Figure 8.1: Example points-to graph

the flow of abstract locations through the program. The analysis infers a non-standard type from the following language for each expression e in the program.

$$E ::= l \mid \mathcal{X}_l \mid *E$$

A type is either a location l, a type variable $\mathcal{X}_l$ associated with a location l, or a *dereference* type $*E$. Type variables $\mathcal{X}_l$ denote the set of locations pointed-to by location l. A dereference type `*E` refers to the set of locations $\bigcup\{\mathcal{X}_l \mid l \in E\}$, *i.e.*, to locations l', such that there exists l in the set of locations of E, and l points to l' (or $l' \in \mathcal{X}_l$). The type of an expression denotes the set of locations that the expression may evaluate to. The minimal solution of the example in Figure 8.1 using Andersen's formulation is

$$\begin{aligned} \mathcal{X}_{l_{\mathtt{a}}} &= \{l_{\mathtt{b}}, l_{\mathtt{c}}\} \\ \mathcal{X}_{l_{\mathtt{b}}} &= \{l_{\mathtt{d}}\} \\ \mathcal{X}_{l_{\mathtt{c}}} &= \{l_{\mathtt{d}}\} \end{aligned}$$

Andersen's analysis is a *global context* and *flow*-insensitive analysis. The analysis is global in that it requires the entire program to derive the pointer relationships. Context insensitive means that the effect of a function call on the points-to relation does not depend on the particular call-site. In other words, the effect of all calls to a particular function are merged. Context insensitivity is generally faster than context-sensitive approaches but may result in less precise information, since information from one call site flows back to all other call-sites of the same function. Flow-insensitive means that the analysis treats the statements of a C program as an unordered collection, evaluated in no particular order.

The crux of any points-to formulation using Set-constraints is in the handling of indirect assignments of the form

$$*e_1 = e_2;$$

Andersen's formulation uses a non-standard Set-expression $*E$ to refer to the locations pointed to by E. The above statement is typed in Andersen's formulation using the following type rule:

$$\frac{\begin{array}{c} e_1 : E_1 \\ e_2 : E_2 \\ E_2 \subseteq *E_1 \end{array}}{*e_1 = e_2 : E_2}$$

where the constraint $E_2 \subseteq *E_1$ expresses the intuitive meaning of assignment, that any location e_2 should be in the points-to set of each location pointed to by e_1. The presence of the non-standard Set-expression $*E$ forces Andersen's algorithm for solving the constraints to be specialized to this application. Andersen uses thus a non-standard resolution rule associated with $*E$ for closing constraint systems:

$$E_1 \subseteq *E_2 \ \wedge \ l \subseteq E_2 \implies E_1 \subseteq \mathcal{X}_l$$

The rule should be read as follows. If $E_1 \subseteq *E_2$ and $l \subseteq E_2$ are constraints in S, then add the constraint $E_1 \subseteq \mathcal{X}_l$ to S. As is apparent in this rule, Andersen's formulation contains an implicit association between each location l and the Set-variable $\mathcal{X}_l$ that represents the points-to set of l. It is this implicit association that results in a specialized resolution algorithm for Andersen's points-to analysis. To express Andersen's analysis in BANE, we first show how to reformulate the types and constraints such that BANE's generic Set-constraint resolution can be applied. Section 8.1.2 then describes the constraint generation. Finally, Section 8.1.3 illustrates the points-to analysis with a complete example.

8.1.1 Re-Formulation using Standard Set Constraints

The association between a location l_i and its points-to set $\mathcal{X}_{l_i}$ is implicit in Andersen's formulation and results in an ad-hoc resolution algorithm. We use a different formulation that makes this association explicit and enables the use of BANE's generic Set-constraint solver. We model locations by pairing location names and points-to set variables with a constructor $\mathsf{ref}(l_i, \mathcal{X}_{l_i})$ akin to reference types in languages like ML [61].

Unlike the type system of ML, which is equality-based, Andersen's points-to analysis uses inclusion constraints. It is well known that subtyping of references is unsound in the presence of update operations (e.g., Java arrays [35]). A sound approach is to turn inclusions between references into equality for their contents: $\mathsf{ref}(L_1, \mathcal{X}) \subseteq \mathsf{ref}(L_2, \mathcal{Y}) \Leftrightarrow L_1 \subseteq L_2 \ \wedge \ \mathcal{X} = \mathcal{Y}$.

We adapt this technique to a purely inclusion-based system using a novel approach.[1] We intuitively treat a reference $l_{\mathtt{x}}$ as an object with a location name and two methods $\mathsf{get} : \mathsf{void} \to \mathcal{X}_{l_{\mathtt{x}}}$ and $\mathsf{set} : \mathcal{X}_{l_{\mathtt{x}}} \to \mathsf{void}$, where the points-to set of the location acts both as the range of the get function and the domain of the set function. Updating a location corresponds to applying the set function to the new value. Dereferencing a location corresponds to applying the get function.

Translating this intuition, we make ref a ternary constructor. The first argument captures a set of location names, the second argument corresponds to the get function (represented by the range), and the third argument corresponds to the set function, represented by its domain. Since functions are contravariant in the domain, ref is contravariant in this third argument. Our location constructor is thus a pure Set-constructor with three arguments having the signature

$$\mathsf{ref} : \mathbf{s}\ \mathbf{s}\ \overline{\mathbf{s}} \to \mathbf{s}$$

A location for a variable x is then represented by an expression $\mathsf{ref}(l_{\mathtt{x}}, \mathcal{X}_{l_{\mathtt{x}}}, \overline{\mathcal{X}_{l_{\mathtt{x}}}})$ (to improve readability we overline contravariant arguments). We now show how to dereference and update these locations using constraints. Dereferencing an expression E representing a set

[1]It has recently come to our attention that this idea has also been suggested to Pottier by Cardelli and independently by Trifonov and Smith as described in Pottier's dissertation [72], page 154.

of locations is equivalent to computing the projection of the second argument to the ref constructor. We have already seen how to model projection in BANE (Section 7.2.1) by generating a fresh variable $\mathcal{T}$ (in this section we use $\mathcal{T}$ for temporary variables that are not associated directly with abstract locations) and adding the constraint

$$E \subseteq_{\mathsf{s}} \mathsf{Pat}[\mathsf{ref}(1, \mathcal{T}, 0), \mathsf{ref}(1, 1, 0)]$$

The constraint makes $\mathcal{T}$ an upper bound on all locations pointed to by E. As explained in Section 7.2.1, BANE provides a specialized abbreviation for this common pattern, called a projection pattern. We can thus use the equivalent constraint

$$E \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T})$$

Updating the points-to set of each location represented by an expression E with a set of locations represented by E' is similar to a dereference, but involves the contravariant argument of the ref constructor. It suffices to assert the constraint

$$E \subseteq_{\mathsf{s}} \mathsf{Pat}[\mathsf{ref}(1, 1, \overline{E'}), \mathsf{ref}(1, 1, 0)]$$

To illustrate how the above constraint adds E' to the points-to set of any location in E, consider the following example. Suppose $E = \mathcal{T}$ and $\mathsf{ref}(l_{\mathtt{x}}, \mathcal{X}_{l_{\mathtt{x}}}, \overline{\mathcal{X}_{l_{\mathtt{x}}}}) \subseteq_{\mathsf{s}} E$. Then the transitive constraint $\mathsf{ref}(l_{\mathtt{x}}, \mathcal{X}_{l_{\mathtt{x}}}, \overline{\mathcal{X}_{l_{\mathtt{x}}}}) \subseteq_{\mathsf{s}} \mathsf{ref}(1, 1, \overline{E'})$ generated through $\mathcal{T}$ and the above constraints is equivalent to $E' \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathtt{x}}}$ (due to contravariance), which is the desired effect. As above for dereference, we can express an equivalent constraint with a projection pattern

$$E \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 3, E')$$

To extract the points-to set as a set of location names of a location $\mathsf{ref}(l_{\mathtt{x}}, \mathcal{X}_{l_{\mathtt{x}}}, \mathcal{X}_{l_{\mathtt{x}}})$ it suffices to project the first field of all ref constructors of $\mathcal{X}_{l_{\mathtt{x}}}$. Let $\mathcal{Y}_{l_{\mathtt{x}}}$ stand for the location names in the points-to set of location $l_{\mathtt{x}}$. Add the constraint

$$\mathcal{X}_{l_{\mathtt{x}}} \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 1, \mathcal{Y}_{l_{\mathtt{x}}})$$

The location names in the points-to set of `x` is the minimal solution of variable $\mathcal{Y}_{l_{\mathtt{x}}}$ which coincides with the transitive lower bound $\mathsf{TLB}(\mathcal{Y}_{l_{\mathtt{x}}})$ in this case, since the solution only contains location constants.

8.1.2 Constraint Generation

Types and constraints are generated by applying the type rules in Figure 8.2 to the abstract syntax tree of a C program. The rules assign a type (Set-expression) to each program expression and generate a system of Set-constraints as side conditions. C has the notion of l- and r-values, where an l-value is a location that can be updated, whereas r-values are values that cannot be updated. Quantities with l-values (such as variables) are automatically converted to r-values if the context requires it. For example the assignment `y = x;` of a variable `x` to a variable `y` implicitly converts the location of `x` (an l-value) to the value stored in location `x` (an r-value) whereas `y` is not converted, since it is the location `y` that must

$$\frac{}{\mathtt{x} : \mathsf{ref}(l_{\mathtt{x}}, \mathcal{X}_{l_{\mathtt{x}}}, \overline{\mathcal{X}_{l_{\mathtt{x}}}})} \qquad \text{(Var)}$$

$$\frac{e : E}{\&e : \mathsf{ref}(0, E, \overline{1})} \qquad \text{(Addr)}$$

$$\frac{e : E \quad E \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}) \quad \mathcal{T} \text{ fresh}}{*e : \mathcal{T}} \qquad \text{(Deref)}$$

$$\frac{\begin{array}{c} e_1 : E_1 \qquad\qquad e_2 : E_2 \\ E_2 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}) \quad E_1 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 3, \mathcal{T}) \\ \mathcal{T} \text{ fresh} \end{array}}{e_1 \mathtt{=} e_2 : E_2} \qquad \text{(Asst)}$$

$$\frac{\begin{array}{c} \mathtt{f} : \mathsf{ref}(l_{\mathtt{f}}, \mathcal{X}_{l_{\mathtt{f}}}, \overline{0}) \quad \mathtt{x}_i : \mathsf{ref}(l_{\mathtt{x}_i}, \mathcal{X}_{l_{\mathtt{x}_i}}, \overline{\mathcal{X}_{l_{\mathtt{x}_i}}}) \\ \mathsf{lam}(\mathcal{A}_{\mathtt{f}}, \mathcal{R}_{\mathtt{f}}) \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathtt{f}}} \\ \mathcal{A}_{\mathtt{f}} \subseteq_{\mathsf{r(s)}} \langle a_i : \mathcal{X}_{l_{\mathtt{x}_i}} \rangle_{i=1..n} \circ 1 \\ \mathcal{A}_{\mathtt{f}}, \mathcal{R}_{\mathtt{f}} \text{ fresh} \end{array}}{\mathtt{f(x}_1\mathtt{,...,x}_n\mathtt{)}\ \{\ldots\} : 0} \qquad \text{(Fun)}$$

$$\frac{\begin{array}{c} e : E \quad E \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}) \\ \mathcal{T} \subseteq_{\mathsf{s}} \mathcal{R}_{\mathtt{f}} \\ \mathcal{T} \text{ fresh} \end{array}}{\mathtt{return(}e\mathtt{)} : 0} \qquad \text{(Ret)}$$

$$\frac{\begin{array}{c} e_0 : E_0 \qquad\qquad e_i : E_i \\ E_i \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_i) \qquad i = 0..n \\ \mathcal{T}_0 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 2, \mathcal{T}') \quad \mathcal{T}_0 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 1, \mathcal{A}) \\ \langle a_i : \mathcal{T}_i \rangle_{i=1..n} \circ 0 \subseteq_{\mathsf{r(s)}} \mathcal{A} \\ \mathcal{T}_i, \mathcal{T}', \mathcal{A} \text{ fresh} \end{array}}{e_0\mathtt{(}e_1\mathtt{,...,}e_n\mathtt{)} : \mathsf{ref}(0, \mathcal{T}', \overline{1})} \qquad \text{(App)}$$

Figure 8.2: Constraint generation for Andersen's analysis

be updated. To avoid separate rules for l- and r-values the type rules in Figure 8.2 lift all values to l-values and infer types for l-values only. Where necessary, the types of l-values are converted explicitly to the types of the corresponding r-values by adding dereference constraints. The (Var) rule for example gives `x` the type $\mathsf{ref}(l_{\mathtt{x}}, \mathcal{X}_{l_{\mathtt{x}}}, \overline{\mathcal{X}_{l_{\mathtt{x}}}})$ which is the type of the location of `x` (and thus the type of the l-value of `x`).

We now describe the remaining rules in Figure 8.2. Rule (Addr) for typing the address-of operator (Addr) creates a reference to the type of its operand E by nesting E inside a ref constructor. Note that an expression $\&e$ can never be used as an l-value in C, but we nevertheless lift its type. Since the resulting value can only appear in a context requiring an r-value, the type $\mathsf{ref}(0, E, \overline{1})$ will be explicitly converted back to the type of the corresponding r-value, namely E. Thus the ref type assigned in (Addr) has no location name and the contravariant argument is 1, since these fields will be ignored by the l-to-r conversion step. The type rule (Deref) for the dereferencing operator does the opposite, removing a ref constructor by projecting the covariant points-to set argument of E through a projection constraint as discussed above. The first constraint in the assignment rule (Asst) transforms the type E_2 of the right-hand l-value e to the type $\mathcal{T}$ of its corresponding r-value as in (Deref). The second constraint $E_1 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 3, \mathcal{T})$ is the assignment constraint described above, which makes $\mathcal{T}$ a subset of the points-to set of E_1 and expresses exactly the intuitive meaning of assignment: the points-to set E_1 of the left-hand side contains at least the points-to set E_2 of the right-hand side.

C functions are given types formed by the binary mixed constructor lam (for lambda) with signature

$$\mathsf{lam} : \overline{\mathsf{r(s)}}\ \mathsf{s} \to \mathsf{s}$$

which is analogous to the standard function type constructor $\cdot \to \cdot$. The first argument of lam is contravariant of sort $\mathsf{Set\text{-}Row}$ and stands for the domain of a function, in this case a record of the formal parameters, where the ith argument is labeled by a_i. The second argument of lam is covariant and represents the range or return value of a function. Since C allows function pointers, we also lift all functions to l-values. Rule (Fun) describes how a function declaration is typed. The type for the l-value of the function `f` is a location $\mathsf{ref}(l_{\mathtt{f}}, \mathcal{X}_{l_{\mathtt{f}}}, \overline{0})$ labeled by $l_{\mathtt{f}}$ with points-to set $\mathcal{X}_{l_{\mathtt{f}}}$. The location of `f` cannot be updated, so the third argument is 0. The location of `f` only points to the function `f` and thus the points-to set $\mathcal{X}_{l_{\mathtt{f}}}$ only contains the set $\mathsf{lam}(\mathcal{A}_{\mathtt{f}}, \mathcal{R}_{\mathtt{f}})$ which is expressed with the constraint $\mathsf{lam}(\mathcal{A}_{\mathtt{f}}, \mathcal{R}_{\mathtt{f}}) \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathtt{f}}}$. $\mathsf{lam}(\mathcal{A}_{\mathtt{f}}, \mathcal{R}_{\mathtt{f}})$ is the function type of `f`, $\mathcal{A}_{\mathtt{f}}$ is a Row-variable standing for the sequence of formal parameters of `f` and $\mathcal{R}_{\mathtt{f}}$ is the type of the r-value returned by `f`. The Row-constraint $\mathcal{A}_{\mathtt{f}} \subseteq_{\mathsf{r(s)}} \langle a_i : \mathcal{X}_{l_{\mathtt{x}_i}} \rangle_{i=1..n} \circ 1$ models the flow of actual argument values to the formal parameters. The l-types of the formal parameters $\mathtt{x}_i$ are $\mathsf{ref}(l_{\mathtt{x}_i}, \mathcal{X}_{l_{\mathtt{x}_i}}, \overline{\mathcal{X}_{l_{\mathtt{x}_i}}})$. Thus any lower bound on the domain $\mathcal{A}_{\mathtt{f}}$ of `f`'s function type will end up as a lower bound on $\langle a_i : \mathcal{X}_{l_{\mathtt{x}_i}} \rangle_{i=1..n} \circ 1$ and consequently, each individual argument flows into the points-to set $\mathcal{X}_{l_{\mathtt{x}_i}}$ of the corresponding formal parameter. The use of a maximal $\mathsf{Row}(\circ 1)$ serves the purpose of ignoring any extraneous arguments that might be passed to `f`.[2] Note that the types of the formal arguments are updatable locations, since in the body of function `f`, the formal parameters are assignable like any other variable. The correspondence between the function return type $\mathcal{R}_{\mathtt{f}}$ and the body of `f` is dealt with in the (Ret) rule for typing `return` statements. The rule assumes that the `return` statement appears in the body of function `f`. The type E of the l-value of e is converted to the type of the corresponding r-value $\mathcal{T}$ using the now familiar projection constraint. The constraint $\mathcal{T} \subseteq_{\mathsf{s}} \mathcal{R}_{\mathtt{f}}$ then constrains the return set $\mathcal{R}_{\mathtt{f}}$ to contain $\mathcal{T}$. Since the rule is applied to all `return` statements in the body of `f`, the return set $\mathcal{R}_{\mathtt{f}}$ contains the union of all possible return values.

Finally, consider the rule for function application. Here e_0 is the function expression, *i.e.*, an expression whose L-value is a location pointing to a function.[3] The l-values of

[2]We have found numerous C programs containing function applications that pass more arguments than the function expects.

[3]C actually contains implicit dereferencing for function pointers appearing in the function position of an application. Here we take the opposite approach, lifting constant functions to l-values and always performing the dereference at the application point.

```
(1)  f(r) { return(r); }

(2)  g(p,q,h) { *p = (*h)(q); }

(3)  a = &b;
(4)  a = &c;
(5)  g(a, &d, &f);
```

Figure 8.3: More complex C example

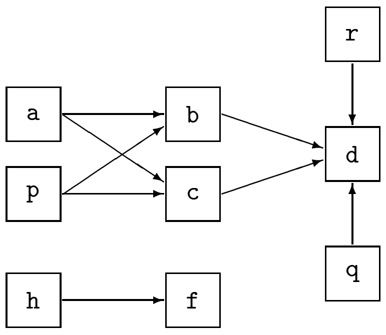

Figure 8.4: Points-to graph of program in Figure 8.3

the actual arguments e_i have types E_i. The first set of constraints in the rule converts the types of the l-values of $e_0..e_n$ to the types $\mathcal{T}_i$ of the corresponding r-values. The constraint $\mathcal{T}_0 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 2, \mathcal{T}')$ projects the return set of the function type $\mathcal{T}_0$ into $\mathcal{T}'$ and the constraint $\mathcal{T}_0 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 1, \mathcal{A})$ makes the fresh Row-variable $\mathcal{A}$ a lower bound on the domain of the function (recall that lam is contravariant in the first argument). The final constraint $\langle a_i : \mathcal{T}_i \rangle_{i=1..n} \circ 0 \subseteq_{\mathsf{r(s)}} \mathcal{A}$ models the flow of the actual arguments $\mathcal{T}_i$ to the domain of the function (through variable $\mathcal{A}$). We use a minimal Row-expression ($\circ 0$) to avoid inconsistent constraints when functions are applied to fewer arguments than expected. The final type of the function application is $\mathcal{T}'$ lifted to an l-value as in the rule (Addr).

8.1.3 A Complete Points-to Example

To illustrate the type and constraint generation rules, consider Figure 8.3 which contains an example C program. The function pointers and indirect assignments generate the points-to graph of Figure 8.4 which contains the points-to graph of the simple example of Figure 8.1 as a sub-graph. Line (1) declares an identity function `f`. Line (2) declares function `g` with three arguments `p`, `q`, and `h`. The body of `g` applies `h` (which is thus a function parameter) to `q` and assigns the result indirectly through `p`. Lines (3) and (4) assign the address of variables `b` and `c` to `a`. Line (5) calls `g`, passing `a` as `p`, the address of variable `d` as `q`, and the identity function `f` as `h`.

We now give a detailed account of the type judgments and constraints generated for this example. The type judgments and constraints generated for line (1) in Figure 8.3 are

$$\begin{array}{rll} \mathtt{f} : \mathsf{ref}(l_{\mathtt{f}}, \mathcal{X}_{l_{\mathtt{f}}}, \overline{0}) & \mathtt{r} : \mathsf{ref}(l_{\mathtt{r}}, \mathcal{X}_{l_{\mathtt{r}}}, \overline{\mathcal{X}_{l_{\mathtt{r}}}}) & \\ \mathsf{lam}(\mathcal{A}_{\mathtt{f}}, \mathcal{R}_{\mathtt{f}}) \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathtt{f}}} & \mathcal{A}_{\mathtt{f}} \subseteq_{\mathsf{r(s)}} \langle a_1 : \mathcal{X}_{l_{\mathtt{r}}} \rangle \circ 1 & (8.1) \\ \mathsf{ref}(l_{\mathtt{r}}, \mathcal{X}_{l_{\mathtt{r}}}, \overline{\mathcal{X}_{l_{\mathtt{r}}}}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_1) & \mathcal{T}_1 \subseteq_{\mathsf{s}} \mathcal{R}_{\mathtt{f}} & \end{array}$$

The constraints on the second line arise from applying Rule (Fun) and the constraints on the last line arise from applying Rule (Ret). Note that the second to last constraint implies $\mathcal{X}_{l_{\mathtt{r}}} \subseteq_{\mathsf{s}} \mathcal{T}_1$, which together with the last constraint implies

$$\mathcal{X}_{l_{\mathtt{r}}} \subseteq_{\mathsf{s}} \mathcal{R}_{\mathtt{f}} \tag{8.2}$$

stating that the return set of `f` contains whatever `r` points-to. The type judgments and constraints for line (2) are as follows. Applying Rule (Fun) to `g` generates

$$\begin{array}{rll} \mathsf{g} : \mathsf{ref}(l_{\mathsf{g}}, \mathcal{X}_{l_{\mathsf{g}}}, \overline{0}) & \mathtt{p} : \mathsf{ref}(l_{\mathtt{p}}, \mathcal{X}_{l_{\mathtt{p}}}, \overline{\mathcal{X}_{l_{\mathtt{p}}}}) & \\ \mathtt{q} : \mathsf{ref}(l_{\mathtt{q}}, \mathcal{X}_{l_{\mathtt{q}}}, \overline{\mathcal{X}_{l_{\mathtt{q}}}}) & \mathtt{h} : \mathsf{ref}(l_{\mathtt{h}}, \mathcal{X}_{l_{\mathtt{h}}}, \overline{\mathcal{X}_{l_{\mathtt{h}}}}) & \\ \mathsf{lam}(\mathcal{A}_{\mathsf{g}}, \mathcal{R}_{\mathsf{g}}) \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathsf{g}}} & \mathcal{A}_{\mathsf{g}} \subseteq_{\mathsf{r(s)}} \langle a_1 : \mathcal{X}_{l_{\mathtt{p}}}, a_2 : \mathcal{X}_{l_{\mathtt{q}}}, a_3 : \mathcal{X}_{l_{\mathtt{h}}} \rangle \circ 1 & (8.3) \end{array}$$

Applying (Deref) and (App) to `(*h)(q)` generates

$$\begin{array}{rll} \mathsf{ref}(l_{\mathtt{h}}, \mathcal{X}_{l_{\mathtt{h}}}, \overline{\mathcal{X}_{l_{\mathtt{h}}}}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_2) & \mathcal{T}_2 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_3) & (8.4) \\ \mathsf{ref}(l_{\mathtt{q}}, \mathcal{X}_{l_{\mathtt{q}}}, \overline{\mathcal{X}_{l_{\mathtt{q}}}}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_4) & & \\ \mathcal{T}_3 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 1, \mathcal{A}_1) & \langle a_1 : \mathcal{T}_4 \rangle \circ 0 \subseteq_{\mathsf{r(s)}} \mathcal{A}_1 & (8.5) \\ \mathcal{T}_3 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 2, \mathcal{T}_1') & \mathtt{(*h)(q)} : \mathsf{ref}(0, \mathcal{T}_1', \overline{1}) & (8.6) \end{array}$$

which imply among other constraints

$$\mathcal{X}_{l_{\mathtt{h}}} \subseteq_{\mathsf{s}} \mathcal{T}_2 \tag{8.7}$$

$$\mathcal{X}_{l_{\mathtt{q}}} \subseteq_{\mathsf{s}} \mathcal{T}_4 \tag{8.8}$$

Applying (Deref) to `*p` and applying (Asst) to the entire assignment we obtain

$$\begin{array}{rl} \mathsf{ref}(l_{\mathtt{p}}, \mathcal{X}_{l_{\mathtt{p}}}, \overline{\mathcal{X}_{l_{\mathtt{p}}}}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_5) & \mathtt{*p} : \mathcal{T}_5 \\ \mathsf{ref}(0, \mathcal{T}_1', \overline{1}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_6) & \mathcal{T}_5 \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 3, \mathcal{T}_6) \end{array}$$

which imply

$$\mathcal{X}_{l_{\mathtt{p}}} \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 3, \mathcal{T}_6) \tag{8.9}$$

$$\mathcal{T}_1' \subseteq_{\mathsf{s}} \mathcal{T}_6 \tag{8.10}$$

The type judgments and constraints for line (3) are

$$\begin{array}{rl} \mathsf{b} : \mathsf{ref}(l_{\mathsf{b}}, \mathcal{X}_{l_{\mathsf{b}}}, \overline{\mathcal{X}_{l_{\mathsf{b}}}}) & \mathsf{\&b} : \mathsf{ref}(0, \mathsf{ref}(l_{\mathsf{b}}, \mathcal{X}_{l_{\mathsf{b}}}, \overline{\mathcal{X}_{l_{\mathsf{b}}}}), \overline{1}) \\ \mathsf{a} : \mathsf{ref}(l_{\mathsf{a}}, \mathcal{X}_{l_{\mathsf{a}}}, \overline{\mathcal{X}_{l_{\mathsf{a}}}}) & \mathsf{ref}(0, \mathsf{ref}(l_{\mathsf{b}}, \mathcal{X}_{l_{\mathsf{b}}}, \overline{\mathcal{X}_{l_{\mathsf{b}}}}), \overline{1}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_7) \\ & \mathsf{ref}(l_{\mathsf{a}}, \mathcal{X}_{l_{\mathsf{a}}}, \overline{\mathcal{X}_{l_{\mathsf{a}}}}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 3, \mathcal{T}_7) \end{array}$$

which imply that $\mathsf{ref}(l_{\mathsf{b}}, \mathcal{X}_{l_{\mathsf{b}}}, \overline{\mathcal{X}_{l_{\mathsf{b}}}}) \subseteq_{\mathsf{s}} \mathcal{T}_7 \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathsf{a}}}$ by contravariance of the third argument of ref. The constraints generated for line (4) are similar and we thus have

$$\begin{aligned} \mathsf{ref}(l_{\mathsf{b}}, \mathcal{X}_{l_{\mathsf{b}}}, \overline{\mathcal{X}_{l_{\mathsf{b}}}}) &\subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathsf{a}}} \\ \mathsf{ref}(l_{\mathsf{c}}, \mathcal{X}_{l_{\mathsf{c}}}, \overline{\mathcal{X}_{l_{\mathsf{c}}}}) &\subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathsf{a}}} \end{aligned} \tag{8.11}$$

Finally, line (5) generates the following type judgments and constraints.

$$\begin{array}{rll} \mathtt{g} : \mathsf{ref}(l_{\mathtt{g}}, \mathcal{X}_{l_{\mathtt{g}}}, \overline{0}) & \mathtt{a} : \mathsf{ref}(l_{\mathsf{a}}, \mathcal{X}_{l_{\mathsf{a}}}, \overline{\mathcal{X}_{l_{\mathsf{a}}}}) & \\ \mathtt{d} : \mathsf{ref}(l_{\mathtt{d}}, \mathcal{X}_{l_{\mathtt{d}}}, \overline{\mathcal{X}_{l_{\mathtt{d}}}}) & \mathtt{\&d} : \mathsf{ref}(0, \mathsf{ref}(l_{\mathtt{d}}, \mathcal{X}_{l_{\mathtt{d}}}, \overline{\mathcal{X}_{l_{\mathtt{d}}}}), \overline{1}) & \\ \mathtt{f} : \mathsf{ref}(l_{\mathtt{f}}, \mathcal{X}_{l_{\mathtt{f}}}, \overline{0}) & \mathtt{\&f} : \mathsf{ref}(0, \mathsf{ref}(l_{\mathtt{f}}, \mathcal{X}_{l_{\mathtt{f}}}, \overline{1}), \overline{0}) & \\ \mathsf{ref}(l_{\mathsf{a}}, \mathcal{X}_{l_{\mathsf{a}}}, \overline{\mathcal{X}_{l_{\mathsf{a}}}}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_8) & & \\ \mathsf{ref}(0, \mathsf{ref}(l_{\mathtt{d}}, \mathcal{X}_{l_{\mathtt{d}}}, \overline{\mathcal{X}_{l_{\mathtt{d}}}}), \overline{1}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_9) & & \\ \mathsf{ref}(0, \mathsf{ref}(l_{\mathtt{f}}, \mathcal{X}_{l_{\mathtt{f}}}, \overline{1}), \overline{0}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_{10}) & & \\ \mathsf{ref}(l_{\mathtt{g}}, \mathcal{X}_{l_{\mathtt{g}}}, \overline{0}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_{11}) & \mathcal{T}_{11} \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 1, \mathcal{A}_2) & (8.12) \\ \langle a_1 : \mathcal{T}_8, a_2 : \mathcal{T}_9, a_3 : \mathcal{T}_{10} \rangle \circ 0 \subseteq_{\mathsf{r(s)}} \mathcal{A}_2 & & (8.13) \end{array}$$

These constraints imply

$$\mathcal{X}_{l_{\mathsf{a}}} \subseteq_{\mathsf{s}} \mathcal{T}_8 \tag{8.14}$$

$$\mathsf{ref}(l_{\mathtt{d}}, \mathcal{X}_{l_{\mathtt{d}}}, \overline{\mathcal{X}_{l_{\mathtt{d}}}}) \subseteq_{\mathsf{s}} \mathcal{T}_9 \tag{8.15}$$

$$\mathsf{ref}(l_{\mathtt{f}}, \mathcal{X}_{l_{\mathtt{f}}}, \overline{0}) \subseteq_{\mathsf{s}} \mathcal{T}_{10} \tag{8.16}$$

$$\mathcal{X}_{l_{\mathtt{g}}} \subseteq_{\mathsf{s}} \mathcal{T}_{11} \tag{8.17}$$

By (8.3), (8.12), and (8.17) we obtain $\mathsf{lam}(\mathcal{A}_{\mathtt{g}}, \mathcal{R}_{\mathtt{g}}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 1, \mathcal{A}_2)$ and thus $\mathcal{A}_2 \subseteq_{\mathsf{r(s)}} \mathcal{A}_{\mathtt{g}}$ by contravariance of the first argument of lam. Then by (8.13) and (8.3) we obtain

$$\mathcal{T}_8 \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathsf{p}}} \tag{8.18}$$

$$\mathcal{T}_9 \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathsf{q}}} \tag{8.19}$$

$$\mathcal{T}_{10} \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathsf{h}}} \tag{8.20}$$

connecting up the actual arguments with the formal parameters of $\mathtt{g}$. By (8.16), (8.20) we obtain

$$\mathsf{ref}(l_{\mathtt{f}}, \mathcal{X}_{l_{\mathtt{f}}}, \overline{0}) \subseteq_{\mathsf{s}} \mathcal{X}_{l_{\mathsf{h}}} \tag{8.21}$$

stating that $\mathtt{h}$ points to function $\mathtt{f}$. Furthermore, by (8.7), and (8.4), we have $\mathsf{ref}(l_{\mathtt{f}}, \mathcal{X}_{l_{\mathtt{f}}}, \overline{0}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{ref}, 2, \mathcal{T}_3)$ and thus $\mathcal{X}_{l_{\mathtt{f}}} \subseteq_{\mathsf{s}} \mathcal{T}_3$. By (8.1), (8.5), and (8.6) we obtain $\mathsf{lam}(\mathcal{A}_{\mathtt{f}}, \mathcal{R}_{\mathtt{f}}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 1, \mathcal{A}_1)$ connecting the argument Row, and $\mathsf{lam}(\mathcal{A}_{\mathtt{f}}, \mathcal{R}_{\mathtt{f}}) \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{lam}, 2, \mathcal{T}_1')$ connecting the result set, which are equivalent to

$$\mathcal{A}_1 \subseteq_{\mathsf{r(s)}} \mathcal{A}_{\mathtt{f}} \tag{8.22}$$

$$\mathcal{R}_{\mathtt{f}} \subseteq_{\mathsf{s}} \mathcal{T}_1' \tag{8.23}$$

By (8.22), (8.5), and (8.1) we obtain $\langle a_1 : \mathcal{T}_4\rangle \circ 0 \subseteq_{\mathsf{r(s)}} \langle a_1 : \mathcal{X}_{l_\mathtt{r}}\rangle \circ 1$ and thus $\mathcal{T}_4 \subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{r}}$, which together with (8.8) implies

$$\mathcal{X}_{l_\mathtt{q}} \subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{r}} \tag{8.24}$$

stating that `r` points-to whatever `q` points to. By (8.2), (8.23), and (8.10) we also have

$$\mathcal{X}_{l_\mathtt{q}} \subseteq_\mathsf{s} \mathcal{T}_6 \tag{8.25}$$

Variable $\mathcal{T}_6$ is used in the assignment of `*p = (*h)(q);` and we now follow argument `a` in the call to `g`. By (8.11), (8.14), and (8.18) we have

$$\begin{aligned} \mathsf{ref}(l_\mathtt{b}, \mathcal{X}_{l_\mathtt{b}}, \overline{\mathcal{X}_{l_\mathtt{b}}}) &\subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{p}} \\ \mathsf{ref}(l_\mathtt{c}, \mathcal{X}_{l_\mathtt{c}}, \overline{\mathcal{X}_{l_\mathtt{c}}}) &\subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{p}} \end{aligned} \tag{8.26}$$

Furthermore by (8.9), we obtain $\mathsf{ref}(l_\mathtt{b}, \mathcal{X}_{l_\mathtt{b}}, \overline{\mathcal{X}_{l_\mathtt{b}}}) \subseteq_\mathsf{s} \mathsf{ProjPat}(\mathsf{ref}, 3, \mathcal{T}_6)$ and $\mathsf{ref}(l_\mathtt{c}, \mathcal{X}_{l_\mathtt{c}}, \overline{\mathcal{X}_{l_\mathtt{c}}}) \subseteq_\mathsf{s} \mathsf{ProjPat}(\mathsf{ref}, 3, \mathcal{T}_6)$ which are equivalent to $\mathcal{T}_6 \subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{b}}$ and $\mathcal{T}_6 \subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{c}}$. But then by (8.25) we obtain

$$\begin{aligned} \mathcal{X}_{l_\mathtt{q}} &\subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{b}} \\ \mathcal{X}_{l_\mathtt{q}} &\subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{c}} \end{aligned} \tag{8.27}$$

which establishes that `b` and `c` point-to whatever `q` points to. It remains to track the actual argument `&d` to formal parameter `q`. By (8.15) and (8.19) we obtain

$$\mathsf{ref}(l_\mathtt{d}, \mathcal{X}_{l_\mathtt{d}}, \overline{\mathcal{X}_{l_\mathtt{d}}}) \subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{q}} \tag{8.28}$$

and thus

$$\begin{aligned} \mathsf{ref}(l_\mathtt{d}, \mathcal{X}_{l_\mathtt{d}}, \overline{\mathcal{X}_{l_\mathtt{d}}}) &\subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{r}} \\ \mathsf{ref}(l_\mathtt{d}, \mathcal{X}_{l_\mathtt{d}}, \overline{\mathcal{X}_{l_\mathtt{d}}}) &\subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{b}} \\ \mathsf{ref}(l_\mathtt{d}, \mathcal{X}_{l_\mathtt{d}}, \overline{\mathcal{X}_{l_\mathtt{d}}}) &\subseteq_\mathsf{s} \mathcal{X}_{l_\mathtt{c}} \end{aligned} \tag{8.29}$$

The final points-to graph is directly read off relations (8.11), (8.26), (8.28), (8.29), and (8.21). Note the edge `h` $\rightarrow$ `f` in the resulting points-to graph of Figure 8.4 showing that function parameter `h` may point to function `f`. This formulation of points-to analysis thus performs a *control-flow* analysis simultaneously with the pointer analysis.

8.2 ML Exception Inference

This section describes an exception inference for ML developed by the author. The standard ML type system gives no information about the set of exceptions that an expression may raise. Knowing only the types, a programmer must assume that each expression e has the worst possible effect: Every imaginable exception may be raised during evaluation of e. The exception inference described here gives the programmer more precise information about possible exceptions. We present our analysis for Mini-ML but discuss implementation issues for full SML.

The syntax of Mini-ML is a typed lambda calculus with exceptions, `raise` and `handle` expressions, pairs, and `case` expressions with pattern matching.

$$\begin{array}{rcl} e & ::= & x \mid c \mid d(e) \mid (e_1, e_2) \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e_1\, e_2 \mid \\ & & e_0\ \texttt{handle}\ x\ \texttt{=>}\ e_1 \mid \texttt{raise}\ e \mid \texttt{case}\ e_0\ \texttt{of}\ p\ \texttt{=>}\ e_1\ \Box\ x\ \texttt{=>}\ e_2 \mid \\ & & \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \\ p & ::= & x \mid c \mid d(p) \mid (p_1, p_2) \end{array}$$

The language has a standard call-by-value semantics which we outline informally. Exception values are built from exception constructors, where c stands for a constant exception and d for a unary exception constructor that can be applied to a value of fixed type (we use d without an argument to refer to either a constant or a unary exception constructor). Exception values built from c and d are first class values that can be passed around before being raised in a raise-expression or pattern matched in a case-expression. Handle expressions evaluate e_0 while catching any exception. If an exception is caught, it is bound to identifier x and the handler body e_1 is evaluated. Raise-expressions evaluate the argument e (which must be an exception value) and raise it.[4] Case expressions evaluate e_0 and match the result against pattern p. If the resulting value matches, expression e_1 is evaluated, otherwise, the value is bound to identifier x and the default branch e_2 is evaluated. Finally, a let-expression binds variable x to the value of e_1 and evaluates e_2. We use let-expressions to illustrate polymorphic analysis. Patterns p consist of variable patterns x (matching any value and binding it to the identifier x), constant patterns c, constructor patterns $d(p)$ with argument pattern p and pair patterns (p_1, p_2). We assume that patterns are linear, *i.e.*, every identifier occurs at most once in any given pattern.

Our implementation of the exception inference deals with the entire ML language and our examples in this section do use ML features not present in the above language. The exception inference is formulated as a non-standard type and effect system [54] which infers a type expression T and an effect expression E for each program expression e of the source program. The effect E of an expression e denotes the set of exceptions that may be raised during evaluation of e. The following grammar defines type and effect expressions.

$$\begin{array}{l} T ::= \mathsf{exn}(E) \mid \langle T_1, T_2 \rangle \mid T_1 \xrightarrow{E} T_2 \mid \mathcal{T} \\ E ::= \mathcal{E} \mid 0 \mid c \mid d(T) \mid E \cap E \mid E \cup E \mid \neg\{d\} \end{array}$$

where $\mathcal{T}$ ranges over type variables and $\mathcal{E}$ ranges over effect variables. The type $T_1 \xrightarrow{E} T_2$ is a function type with domain T_1, range T_2 and effect E, where E represents the effect that results from applying a function of this type. Effect expressions consist of constant exception constructors c, and unary exception constructors d, one for every exception constructor in the source program. Because exceptions can carry values, the grammar for type and effect expressions is mutually recursive, a feature not commonly seen in effect systems.

We map these type and effect expressions onto mixed expressions by choosing to represent types with FlowTerm-expressions and effects with Set-expressions using the

[4]For clarity, we made `raise` expressions explicit even though `raise` could be treated as a function with a type binding in the initial environment.

following constructors and signatures:

$$\begin{aligned} \mathsf{exn} &: \mathbf{s} \to \mathbf{ft} \\ \mathsf{pair} &: \mathbf{ft}\ \mathbf{ft} \to \mathbf{ft} \\ \cdot \xrightarrow{\cdot} \cdot &: \overline{\mathbf{ft}}\ \mathbf{s}\ \mathbf{ft} \to \mathbf{ft} \\ c &: \mathbf{s} \quad \text{for all constant exceptions } c \\ d &: \mathbf{ft} \to \mathbf{s} \quad \text{for all unary exceptions } d \end{aligned}$$

Pair types $\langle T_1, T_2 \rangle$ are represented using a binary constructor pair, but we will write the more convenient form $\langle T_1, T_2 \rangle$. The type expressions T correspond to the standard type expressions of ML (for our subset of the language) except for the addition of the exception annotations E in the types $\mathsf{exn}(E)$ and $T_1 \xrightarrow{E} T_2$. If we erase these exception annotations, we recover the standard ML types. We will also use polymorphic constrained expressions $Q = \forall V.T \backslash S$ to analyze let-bound expressions polymorphically. We use the convention that if V and S are empty, then $Q = T$.

In the standard ML type system, all exception values have type exn giving no indication of the constructors used to create the exception. Our exception annotation E on $\mathsf{exn}(E)$ provides information about the exception constructors of a particular exception value. For example the constant exception `Subscript` (raised when accessing an array outside its domain), can be given the type exn(Subscript), where Subscript is the constant Set-constructor corresponding to the `Subscript` exception. Similarly, the annotation on function types provides information about the exceptions that may be raised when applying a function. If `sub` is the function that returns an array element at a given index and `sub` may raise the `Subscript` exception, then we assign `sub` the type $\langle \mathcal{T}\ \mathsf{array}, \mathsf{int} \rangle \xrightarrow{\mathsf{Subscript}} \mathcal{T}$, where $\mathcal{T}$ array is the type of an array containing elements of type $\mathcal{T}$ and int is the type of integers. (We discuss later in this section how such types and ML datatypes are added to our inference system).

Since exception values and functions are first-class values in ML that can be passed as arguments and stored in data-structures, our type and effect language must be rich enough to express dependencies between exception values and effects. We illustrate these aspects with a few more examples. Consider the ML function `raise`, which is used to raise an exception. Its ML type is `raise` : exn $\to \mathcal{T}$. Using our refined type language, the type becomes `raise` : $\mathsf{exn}(\mathcal{E}) \xrightarrow{\mathcal{E}} \mathcal{T}$, capturing the fact that applying `raise` to an exception of type $\mathsf{exn}(\mathcal{E})$ causes the observable effect $\mathcal{E}$. The variables $\mathcal{T}$ and $\mathcal{E}$ are implicitly quantified here, but we omit the quantifier. In general, we can infer constrained polymorphic expressions as discussed in Section 7.6 for let-bound expressions that observe the so-called *value restriction* [90, 62]. We assume that all let-bound expressions in valid Mini-ML programs are values and that their types can therefore be generalized. Let-expressions that do not satisfy the value restriction can be eliminated by replacing them with (`fn` x => e_2) e_1.

Consider a function `catchSubscript` that calls a function argument, and if the `Subscript` exception is raised, returns the default value `d`.

```
exception Subscript
```

$$\frac{\begin{array}{l} A(x) = \forall V.T \backslash S \\ \text{fresh substitution } \sigma \text{ on } V \end{array}}{A \vdash x : \sigma(T) \mathbin{!} 0,\ \sigma(S)} \quad [\text{VAR}] \qquad\qquad A \vdash c : \mathsf{exn}(c) \mathbin{!} 0,\ \emptyset \quad [\text{CON0}]$$

$$\frac{A \vdash e : T \mathbin{!} E,\ S}{A \vdash d(e) : \mathsf{exn}(d(T)) \mathbin{!} E,\ S} \quad [\text{CON1}] \qquad \frac{\begin{array}{c} A \vdash e_1 : T_1 \mathbin{!} E_1,\ S_1 \\ A \vdash e_2 : T_2 \mathbin{!} E_2,\ S_2 \end{array}}{A \vdash (e_1,e_2) : \langle T_1,T_2\rangle \mathbin{!} E_1 \cup E_2,\ S_1 \cup S_2} \quad [\text{PAIR}]$$

$$\frac{A[x \mapsto \mathcal{T}] \vdash e : T \mathbin{!} E,\ S}{A \vdash \texttt{fn}\ x\ \texttt{=>}\ e : (\mathcal{T} \xrightarrow{E} T) \mathbin{!} 0,\ S} \quad [\text{ABS}] \qquad \frac{\begin{array}{l} A \vdash e_1 : T_1 \mathbin{!} E_1,\ S_1 \\ A \vdash e_2 : T_2 \mathbin{!} E_2,\ S_2 \\ S_3 = \{T_1 \subseteq_{\mathsf{ft}} T_2 \xrightarrow{\mathcal{E}} \mathcal{T}\} \end{array}}{A \vdash e_1\ e_2 : \mathcal{T} \mathbin{!} E_1 \cup E_2 \cup \mathcal{E},\ S_1 \cup S_2 \cup S_3} \quad [\text{APP}]$$

$$\frac{\begin{array}{l} A \vdash e : T \mathbin{!} E,\ S \\ S' = \{T \subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E})\} \end{array}}{A \vdash \texttt{raise}\ e : \mathcal{T} \mathbin{!} E \cup \mathcal{E},\ S \cup S'} \quad [\text{RAISE}]$$

$$\frac{\begin{array}{l} A \vdash e_0 : T_0 \mathbin{!} E_0,\ S_0 \\ A[x \mapsto \mathsf{exn}(E_0)] \vdash e_1 : T_1 \mathbin{!} E_1,\ S_1 \\ S_2 = \{T_0 \subseteq_{\mathsf{ft}} \mathcal{T}, T_1 \subseteq_{\mathsf{ft}} \mathcal{T}\} \end{array}}{A \vdash e_0\ \texttt{handle}\ x\ \texttt{=>}\ e_1 : \mathcal{T} \mathbin{!} E_1,\ S_0 \cup S_1 \cup S_2} \quad [\text{HANDLE}]$$

$$\frac{\begin{array}{l} A \vdash e_0 : T_0 \mathbin{!} E_0,\ S_0 \\ \vdash_{\mathrm{p}} p : (T_p, R_2..R_n, A_p, S_p) \\ S = \{T_0 \subseteq_{\mathsf{ft}} T_p\} \\ A, A_p \vdash e_1 : T_1 \mathbin{!} E_1,\ S_1 \\ A[x \mapsto R_i] \vdash e_2 : T_i \mathbin{!} E_i,\ S_i \quad i = 2..n \\ S' = \{T_i \subseteq_{\mathsf{ft}} \mathcal{T} \mid i = 1..n\} \end{array}}{A \vdash \texttt{case}\ e_0\ \texttt{of}\ p\ \texttt{=>}\ e_1\ \square\ x\ \texttt{=>}\ e_2 : \mathcal{T} \mathbin{!} \bigcup_{i=0..n} E_i,\ S \cup S' \cup \bigcup_{i=0..n} S_i} \quad [\text{CASE}]$$

$$\frac{\begin{array}{l} A \vdash e_1 : T_1 \mathbin{!} E_1,\ S_1 \\ Q = \forall V.T_1 \backslash S_1 \qquad V \text{ local to } S_1 \\ A[x \mapsto Q] \vdash e_2 : T_2 \mathbin{!} E_2,\ S_2 \end{array}}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \mathcal{T}_2 \mathbin{!} E_1 \cup E_2,\ S_1 \cup S_2} \quad [\text{LET}]$$

Figure 8.5: Type and exception inference rules for expressions

$$\vdash_{\mathrm{p}} x : (\mathcal{T}, \{\mathcal{T}'\}, [x \mapsto \mathcal{T}], \emptyset) \qquad \text{[PVAR]}$$

$$\vdash_{\mathrm{p}} c : (\mathsf{exn}(\mathcal{E}), \{\mathsf{exn}(\mathcal{E} \cap \neg\{c\})\}, [], \emptyset) \qquad \text{[PCON0]}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{p}} p : (T, \{R_1..R_n\}, A, S_p) \\ S = S_p \cup \{\mathcal{E} \subseteq_{\mathsf{s}} \mathsf{Pat}[d(T), d(1)]\}\end{array}}{\vdash_{\mathrm{p}} d(p) : (\mathsf{exn}(\mathcal{E}), \{\mathsf{exn}(\mathcal{E} \cap \neg\{d\} \cup d(R_1) \cup .. \cup d(R_n))\}, A, S)} \qquad \text{[PCON1]}$$

$$\frac{\begin{array}{c}\vdash_{\mathrm{p}} p_1 : (T_1, \{R_1..R_n\}, A_1, S_1) \\ \vdash_{\mathrm{p}} p_2 : (T_2, \{R'_1..R'_{n'}\}, A_2, S_2) \\ R = \{(T_1, R'_1), .., (T_1, R'_{n'}), (R_1, T_2), .., (R_n, T_2)\}\end{array}}{\vdash_{\mathrm{p}} (p_1, p_2) : (\langle T_1, T_2 \rangle, R, A_1 \oplus A_2, S_1 \cup S_2)} \qquad \text{[PPAIR]}$$

Figure 8.6: Type and exception inference rules for patterns

```
let catchSubscript = fn f => fn d =>
                  f () handle Subscript => d
in ... end
```

Note our use of the abbreviation e `handle` p `=>` e' for the expression e `handle x => case x of` p `=> e' [] y => raise y`. The value `()` is a dummy value of type `unit` and is used as an argument (result) of functions that have no non-trivial argument (result). We can assign the type

$$\texttt{catchSubscript} : (\mathsf{unit} \xrightarrow{\mathcal{E}} \mathcal{T}) \xrightarrow{0} \mathcal{T} \xrightarrow{\mathcal{E} \cap \neg\{\mathsf{Subscript}\}} \mathcal{T}$$

to `catchSubscript`. The type illustrates the dependencies between the exceptions carried by the function argument `f` and the exceptions of `catchSubscript`. Given a function `f` : $\mathsf{unit} \xrightarrow{\mathcal{E}} \mathcal{T}$ which may raise an exception from the set $\mathcal{E}$, we know that the expression `f()` has type $\mathcal{T}$ and effect $\mathcal{E}$. The `handle` expression prevents the `Subscript` exception from escaping the body of `catchSubscript`. As a result, we know that evaluating `catchSubscript` can result in any exceptions raised by the argument function, except `Subscript` which is expressed with the Set-expression $\mathcal{E} \cap \neg\{\mathsf{Subscript}\}$. Set expressions make it convenient to describe such types concisely.

8.2.1 Type and Constraint Generation

Figure 8.5 shows the type inference rules for expressions. The rules assign types, effects, and a system of constraints to each expression in the source program. Judgments have the form $A \vdash e : T \; ! \; E,\; S$, meaning that under the type assumptions A (mapping identifiers to type expressions), expression e has type T and may raise the exceptions denoted by E.

Both T and E are constrained by the constraint system S. All type and effect variables appearing in the rules are assumed to be fresh.

Figure 8.6 contains type rules for patterns. Judgments for patterns have the form $\vdash_{\mathrm{p}} p : (T, \{R_1..R_k\}, A, S)$ meaning that pattern p has type T, values not matched by the pattern have any one of the remainder types $R_1..R_k$, and any variable x occurring in the pattern has type $A(x)$ as given by the bindings A. Again, types T, $R_1..R_n$, and the range of A are constrained by the constraints in S.

We now discuss the type rules. Except for the `handle` expression, exceptions propagate from sub-expressions to enclosing expressions. The effect of an expression is thus the union of the effect of its sub-expressions. Identifiers are typed using Rule [VAR] by looking up the type assumption for x in the type environment A. The possibly polymorphic type $\forall V.T \backslash S$ is instantiated through a substitution σ mapping all variables V to fresh variables. The result type is $\sigma(T)$ with associated constraints $\sigma(S)$. No exceptions are raised by referring to an identifier, thus the effect is 0. If V and S are empty, the substitution is the identity and the result type is T. Rule [CON0] types constant exceptions c by assigning the type $\mathsf{exn}(c)$. There are no associated constraints and the effect is 0, since no exceptions are raised. Unary exception constructors $d(e)$ are typed using Rule [CON1]. If the inferred type and effect of the argument expression e is T and E constrained by S, then the type of the constructor application is $\mathsf{exn}(d(T))$ with effect E and constraints S. Type inference for a pair expression (e_1, e_2) proceeds by typing e_1 and e_2, yielding T_i, E_i, and S_i $(i = 1, 2)$. The result type is $\langle T_1, T_2 \rangle$ and the effect is the union $E_1 \cup E_2$, since the exceptions raised by the pair expression are either the exceptions raised by e_1 or by e_2. The type and effect are constrained by $S_1 \cup S_2$. Rule [ABS] infers the type for a lambda abstraction `fn` x `=>` e by typing the body e where the type assumptions A are extended with a binding for x to a fresh variable $\mathcal{T}$. If under this assumption the type and effect of the body is T and E, then the lambda abstraction has function type $\mathcal{T} \xrightarrow{E} T$. Evaluating a lambda expression does not raise any exceptions, since the effects of the body are delayed and are only observable when the function is applied. Function application $e_1\ e_2$ is typed by Rule [APP] by inferring types and effects for e_1 and e_2 and constraining the type T_1 of e_1 to be a sub-type of $\mathcal{T}_2 \xrightarrow{\mathcal{E}} \mathcal{T}$, where $\mathcal{E}$ and $\mathcal{T}$ are fresh variables. The constraint makes $\mathcal{E}$ an upper bound on the effects of function T_1 and $\mathcal{T}$ an upper bound on the result type of T_1. Since function types are contravariant in the domain, the constraint makes the argument type T_2 a lower bound on the domain of T_1. The effect of the application is the union of the effect E_1 of e_1, E_2 of e_2, and the effect $\mathcal{E}$ produced by applying the function (and thus evaluating its body). The above constraint is standard for type systems with sub-typing. The sub-typing present in our system is the subtyping of mixed $\mathsf{FlowTerm}$ and Set-constraints.

Rule [RAISE] infers type T and effect E for the argument e. The constraint $T \subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E})$ makes the effect variable $\mathcal{E}$ an upper bound on the set of exceptions carried by the argument e. The type of the `raise` expression is a fresh type variable $\mathcal{T}$ which is akin to saying that the `raise` expression has any type (we could have used 0 instead). The effect of the expression is the effect E of evaluating e unioned with the set of exceptions $\mathcal{E}$ extracted from the exception type T. Rule [HANDLE] for `handle` expressions works in the opposite way. If the type and effect of expression e_0 is T_0 and E_0, then the handler body is typed under assumptions A extended with a binding for x to $\mathsf{exn}(E_0)$. The set of exceptions

E_0 that may be raised by e_0 is effectively converted to the exception type $\mathsf{exn}(E_0)$. The effect of the `handle` expression is simply the effect E_1 of the handler-body, since the effects of e_0 are caught.

Before we describe the rule for `case` expressions, we discuss the pattern rules in Figure 8.6. Recall that pattern judgments have the form $\vdash_{\mathrm{p}} p : (T, \{R_1..R_n\}, A, S)$ assigning pattern p a type T, remainder types $R_1..R_n$, a variable environment A, and constraints S. The type T of the pattern is intended as an upper bound on the type of the value against which the pattern is matched. The remainder types $R_1..R_n$ are possible types for the values not matched by this pattern. We use a set instead of a single remainder type, since not all remainder types can be expressed using a single type expression. We will discuss this issue in the context of the pair pattern. Assumptions A bind pattern variables to types and constraints S constrain the pattern type T, $R_1..R_n$, and the types in A. Pattern variables are assigned a fresh type variable $\mathcal{T}$ by Rule [PVAR]. The remainder type, *i.e.*, the type of values not matched by this pattern is $\mathcal{T}'$—a fresh unconstrained variable—since a variable pattern matches everything (we could have used 0 instead). The assumptions returned by [PVAR] binds x to $\mathcal{T}$. Rule [PCON0] assigns type $\mathsf{exn}(\mathcal{E})$ to constant exception patterns c. The remainder type is $\mathsf{exn}(\mathcal{E} \cap \neg\{c\})$ expressing that if we match c against some value of type $T \subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E})$, then the type of the values not matched by c is $\mathsf{exn}(\mathcal{E} \cap \neg\{c\})$, *i.e.*, exceptions of T, except the c exception. Rule [PCON1] types patterns $d(p)$ by inferring a type T and remainder types $R_1..R_n$ for p. The type of $d(p)$ is then $\mathsf{exn}(\mathcal{E})$, where $\mathcal{E}$ is constrained by $\mathcal{E} \subseteq_{\mathsf{s}} \mathsf{Pat}[d(T), d(1)]$. The constraint makes $d(T)$ an upper bound on the exceptions $\mathcal{E}$ of the value to be matched. It also constrains the matched type against the pattern argument type T, thus indirectly constraining the remainder types $R_1..R_n$. The remainder type is $\mathsf{exn}(\mathcal{E} \cap \neg\{d\} \cup d(R_1) \cup .. \cup d(R_n))$, expressing that any exceptions $\mathcal{E} \cap \neg\{d\}$ are not matched by the pattern, as well as any exceptions $d(R_i)$, where the argument pattern p does not match. Pair patterns (p_1, p_2) are typed with the pair type $\langle T_1, T_2 \rangle$ where T_1 and T_2 are inferred for p_1 and p_2. Since our patterns are linear, the variable bindings A_1 and A_2 have disjoint domains and can be composed into $A_1 \oplus A_2$.

The remainder types of pair patterns are tricky. Suppose p_1 had a single remainder type R_1 and p_2 a single remainder type R_2. The remainder type of the pair pattern (p_1, p_2) is *not* $\langle R_1, R_2 \rangle$. The remainder $\langle R_1, R_2 \rangle$ expresses the types that match neither p_1 nor p_2. However, the pair pattern only matches if both p_1 and p_2 match. Thus the remainder types must be either $\langle R_1, T_2 \rangle$ (if p_1 doesn't match), or $\langle T_1, R_2 \rangle$ (if p_2 doesn't match). These two types cannot be expressed as a single type without losing precision. The only *single* safe remainder type is $\langle T_1, T_2 \rangle$ which is the same type as the type for the pair pattern. Using a single remainder type thus results in no filtering when pair patterns are used.

We now return to the type rule for `case` expressions. Rule [CASE] infers type T_0 and effect E_0 for expression e_0 which is to be matched against p. If the type for the pattern is T_p with remainder types $R_2..R_n$, and variable bindings A_p, then we constrain the type of the expression to be matched with the pattern type $T_0 \subseteq_{\mathsf{ft}} T_p$, and infer type T_1 and effect E_1 for the branch expression e_1 under the assumptions A extended with the bindings for the pattern variables A_p. The default branch is typed once per remainder type $R_2..R_n$, by binding x to R_i. The type of the entire `case` expression is the fresh type variable $\mathcal{T}$ which is an upper bound of $T_1..T_n$. Similarly, the effects of the `case` expression are the effects

```
exception Subscript
exception Fail of exn
let substFail = fn f => fn d =>
                 (f d) handle x =>
                        case x of
                          Fail(y) => raise y
                        | z => raise z
in  ..
end
```

Figure 8.7: Example Mini-ML program

of e_0, e_1, and e_2 (under all remainder types). Using this type rule becomes impractical if the number of remainder types is large. Furthermore, precision is only recovered by using multiple remainder types, if pair and exception constructors are treated as strict, a feature not currently supported by BANE for FlowTerm-constructors. Thus our implementation uses a single remainder type by approximating the remainder of pair patterns as described above.

The final Rule [LET] concerns `let` expressions. If we infer type T_1, effect E_1 under constraints S_1 for the let-bound expression e_1, then we can form the polymorphic constrained type $\forall V.T_1 \backslash S_1$ representing all possible typings of e_1. V is the set of local variables of S (in this case local variables are all variables generated during the type and constraint generation of e_1). The let-body e_2 is then typed using assumptions A extended with a binding for x to $\forall V.T_1 \backslash S_1$. The type of the `let` expression is the type of the let-body T_2 and the effect is the union of effects E_1 and E_2. If we compare this rule to standard inference rules of let-polymorphic type systems, we notice the absence of any reference to the assumptions A in choosing the type variables V to be quantified over. In the standard inference rules of Hindley-Milner type inference expressed using Robinson's unification algorithm to solve equality constraints and explicit substitutions, the type T_1 can only be generalized over type variables not appearing in the assumptions A. This restriction is necessary because the equality constraints arising in Hindley-Milner type inference are eliminated before the quantification. If we constrained a fresh variable $\mathcal{X}$ to be equal to a variable $\mathcal{Y}$ appearing in the assumptions, and $\mathcal{X}$ appears in type T_1, then the unification of $\mathcal{X}$ and $\mathcal{Y}$ either substitutes $\mathcal{X}$ into A, or $\mathcal{Y}$ into T_2. In either case, the variable appearing in T_2 is not quantified and with reason, since the environment can further constrain it. However, if we left the equality constraint $\mathcal{X} = \mathcal{Y}$ unsolved and formed the constrained quantified type $\forall \mathcal{X}.T_2 \backslash \{\mathcal{X} = \mathcal{Y}\}$ instead, the reference to the assumptions A is not necessary, since no variable in A can be local to S. Furthermore, the constraint $\mathcal{X} = \mathcal{Y}$ appearing in the polymorphic type guarantees that any further constraints on $\mathcal{Y}$ will be reflected on $\mathcal{X}$ in all instantiations of the polymorphic type. In our inference we use inclusion constraints instead of equality, but the argument is the same.

8.2.2 An Example Inference

This subsection uses the example program in Figure 8.7 to illustrate the type and constraint generation rules. The example program assumes that `Subscript` is a constant exception and `Fail` is an exception constructor with an exception argument. The function `substFail` expects a function argument `f` and an argument `d` and proceeds by applying `f` to `d`. Any exceptions raised by the application and matching the `Fail(y)` pattern cause the exception argument `y` to be raised. Other exceptions are re-raised. We derive the polymorphic constrained type

$$\begin{aligned}\texttt{substFail} :&\forall \mathcal{T}_1, \mathcal{T}_{\mathtt{d}}, \mathcal{E}_1, \mathcal{E}_5.(\mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_1} \mathcal{T}_1) \xrightarrow{0} \mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_6 \cup \mathcal{E}_1 \cap \neg\{\mathsf{Fail}\}} \mathcal{T}_1 \backslash \\ &\mathcal{E}_1 \subseteq_{\mathsf{s}} \mathsf{Pat}[\mathsf{Fail}(\mathsf{exn}(\mathcal{E}_5)), \mathsf{Fail}(1)]\end{aligned}$$

for this function. The type expresses that if the argument `f` has type $\mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_1} \mathcal{T}_1$, the argument `d` type $\mathcal{T}_{\mathtt{d}}$, then the result of `substFail` is $\mathcal{T}_1$. So far this corresponds to the type $(\alpha \to \beta) \to \alpha \to \beta$ that the standard ML type system would derive. The exception annotations however tell us that if applying `f` raises exceptions $\mathcal{E}_1$, then the exceptions raised by `substFail` are the exceptions $\mathcal{E}_1$, except for the `Fail` exceptions, plus any exceptions carried by a `Fail` exception of $\mathcal{E}_1$.

We break up the body of `substFail` into expressions

$$\begin{aligned} e_{\mathtt{substFail}} &= \texttt{fn f => fn d =>}\ e_{\mathtt{handle}} \\ e_{\mathtt{handle}} &= e_{\mathtt{app}}\ \texttt{handle x =>}\ e_{\mathtt{case}} \\ e_{\mathtt{app}} &= \texttt{f d} \\ e_{\mathtt{case}} &= \texttt{case x of Fail(y) => raise y | z => raise z} \end{aligned}$$

Let $A = [\mathtt{f} \mapsto \mathcal{T}_{\mathtt{f}}, \mathtt{d} \mapsto \mathcal{T}_{\mathtt{d}}]$ be the assumptions resulting from applying Rule [ABS] twice to $e_{\mathtt{substFail}}$. Rule [HANDLE] first derives the judgment for expression $e_{\mathtt{app}}$ under A

$$\frac{\begin{array}{c} A \vdash \mathtt{f} : \mathcal{T}_{\mathtt{f}}\ !\ 0,\ \emptyset \\ A \vdash \mathtt{d} : \mathcal{T}_{\mathtt{d}}\ !\ 0,\ \emptyset \end{array}}{A \vdash \mathtt{f\ d} : \mathcal{T}_1\ !\ \mathcal{E}_1,\ S_{\mathtt{app}}}$$

where $S_{\mathtt{app}} = \{\mathcal{T}_{\mathtt{f}} \subseteq_{\mathsf{ft}} \mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_1} \mathcal{T}_1\}$ and the premisses are obtained by applying Rule [VAR] to `f` and `d`. From this judgment, Rule [HANDLE] constructs assumptions $A' = A[\mathtt{x} \mapsto \mathsf{exn}(\mathcal{E}_1)]$ to be used for the inference of $e_{\mathtt{case}}$. Applying Rule [CASE] to $e_{\mathtt{case}}$ proceeds by deriving $A' \vdash \mathtt{x} : \mathsf{exn}(\mathcal{E}_1)\ !\ 0,\ \emptyset$ using Rule [VAR], and the pattern judgment

$$\vdash_{\mathrm{p}} \mathtt{Fail(y)} : (\mathsf{exn}(\mathcal{E}_2), \{\mathsf{exn}(\mathcal{E}_2 \cap \neg\{\mathsf{Fail}\} \cup \mathsf{Fail}(\mathcal{T}'_{\mathtt{y}}))\}, A_p, S_{\mathtt{pat}})$$

where $A_p = [\mathtt{y} \mapsto \mathcal{T}_{\mathtt{y}}]$ and $S_{\mathtt{pat}} = \{\mathcal{E}_2 \subseteq_{\mathsf{s}} \mathsf{Pat}[\mathsf{Fail}(\mathcal{T}_{\mathtt{y}}), \mathsf{Fail}(1)]\}$. This judgment in turn is derived from the pattern judgment $\vdash_{\mathrm{p}} \mathtt{y} : (\mathcal{T}_{\mathtt{y}}, \mathcal{T}'_{\mathtt{y}}, A_p, \emptyset)$ of `y`. Rule [CASE] proceeds by typing the branch expression `raise y` under assumption A', A_p using [RAISE] resulting in

$$A', A_p \vdash \mathtt{raise\ y} : \mathcal{T}_3\ !\ \mathcal{E}_3,\ S_{\mathtt{raise1}}$$

where $S_{\mathtt{raise1}} = \{\mathcal{T}_{\mathsf{y}} \subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E}_3)\}$. Then the default branch is typed under assumptions $A'' = A'[\mathtt{z} \mapsto \mathsf{exn}(\mathcal{E}_2 \cap \neg\{\mathsf{Fail}\} \cup \mathsf{Fail}(\mathcal{T}'_{\mathsf{y}}))]$ where `z` is bound to the remainder type obtained from the pattern judgment of `Fail(y)`. Applying Rule [Raise] again results in

$$A'' \vdash \mathtt{raise\ z} : \mathcal{T}_4 \,!\, \mathcal{E}_4,\ S_{\mathtt{raise2}}$$

where $S_{\mathtt{raise2}} = \{\mathsf{exn}(\mathcal{E}_2 \cap \neg\{\mathsf{Fail}\} \cup \mathsf{Fail}(\mathcal{T}'_{\mathsf{y}})) \subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E}_4)\}$ which is equivalent to $\{\mathcal{E}_2 \cap \neg\{\mathsf{Fail}\} \cup \mathsf{Fail}(\mathcal{T}'_{\mathsf{y}}) \subseteq_{\mathsf{s}} \mathcal{E}_4\}$. Rule [CASE] now concludes with the judgment

$$A' \vdash e_{\mathtt{case}} : \mathcal{T}_5 \,!\, \mathcal{E}_3 \cup \mathcal{E}_4,\ S_{\mathtt{case}}$$

where $S_{\mathtt{case}} = S_{\mathtt{pat}} \cup S_{\mathtt{raise1}} \cup S_{\mathtt{raise2}} \cup \{\mathsf{exn}(\mathcal{E}_1) \subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E}_2), \mathcal{T}_3 \subseteq_{\mathsf{ft}} \mathcal{T}_5, \mathcal{T}_4 \subseteq_{\mathsf{ft}} \mathcal{T}_5\}$. The last two constraints simply make the result type $\mathcal{T}_5$ of $e_{\mathtt{case}}$ an upper bound on the result types $\mathcal{T}_3$ and $\mathcal{T}_4$ of the branches. Constraint $\mathsf{exn}(\mathcal{E}_1) \subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E}_2)$ results from relating the type of `x` to be matched with the type of the pattern. This constraint is equivalent to $\mathcal{E}_1 \subseteq_{\mathsf{s}} \mathcal{E}_2$. The [HANDLE] Rule now concludes with

$$\frac{\begin{array}{c} A \vdash e_{\mathtt{app}} : \mathcal{T}_1 \,!\, \mathcal{E}_1,\ S_{\mathtt{app}} \\ A' \vdash e_{\mathtt{case}} : \mathcal{T}_5 \,!\, \mathcal{E}_3 \cup \mathcal{E}_4,\ S_{\mathtt{case}} \end{array}}{A \vdash e_{\mathtt{handle}} : \mathcal{T}_2 \,!\, \mathcal{E}_3 \cup \mathcal{E}_4,\ S_{\mathtt{handle}}}$$

where $S_{\mathtt{handle}} = S_{\mathtt{app}} \cup S_{\mathtt{case}} \cup \{\mathcal{T}_1 \subseteq_{\mathsf{ft}} \mathcal{T}_2, \mathcal{T}_5 \subseteq_{\mathsf{ft}} \mathcal{T}_2\}$ and the last two constraints make the result type $\mathcal{T}_2$ of $e_{\mathtt{handle}}$ an upper bound of the type $\mathcal{T}_1$ of $e_{\mathtt{app}}$ and $\mathcal{T}_5$ of $e_{\mathtt{case}}$. The final judgment for $e_{\mathtt{substFail}}$ is obtained by concluding the initial [ABS] Rules and results in

$$[] \vdash e_{\mathtt{substFail}} : \mathcal{T}_{\mathtt{f}} \xrightarrow{0} \mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_3 \cup \mathcal{E}_4} \mathcal{T}_1 \,!\, 0,\ S_{\mathtt{handle}}$$

The final inductive constraints $S_{\mathtt{handle}}$ are summarized by

$$\begin{array}{rll} \mathcal{T}_3 \subseteq_{\mathsf{ft}} \mathcal{T}_5 & \mathcal{T}_4 \subseteq_{\mathsf{ft}} \mathcal{T}_5 & \mathcal{T}_5 \subseteq_{\mathsf{ft}} \mathcal{T}_2 \\ \mathcal{T}_1 \subseteq_{\mathsf{ft}} \mathcal{T}_2 & \mathcal{E}_1 \subseteq_{\mathsf{s}} \mathcal{E}_2 & \mathcal{E}_2 \subseteq_{\mathsf{s}} \mathsf{Pat}[\mathsf{Fail}(\mathcal{T}_{\mathsf{y}}), \mathsf{Fail}(1)] \\ \mathcal{E}_2 \cap \neg\{\mathsf{Fail}\} \cup \mathsf{Fail}(\mathcal{T}'_{\mathsf{y}}) \subseteq_{\mathsf{s}} \mathcal{E}_4 & \mathcal{T}_{\mathsf{y}} \subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E}_3) & \mathcal{T}_{\mathtt{f}} \subseteq_{\mathsf{ft}} \mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_1} \mathcal{T}_1 \end{array}$$

Using the constraint on $\mathcal{T}_{\mathsf{y}}$ we can introduce the fresh variable $\mathcal{E}_5$ and set $\mathcal{T}_{\mathsf{y}} = \mathsf{exn}(\mathcal{E}_5)$ with the constraint $\mathcal{E}_5 \subseteq_{\mathsf{s}} \mathcal{E}_3$. Applying minimization and maximization simplification (Section 7.6.3) to these constraints we obtain

$$\begin{array}{lll} \mathcal{T}_3 \overset{\min}{\Rightarrow} 0 & \mathcal{T}_4 \overset{\min}{\Rightarrow} 0 & \mathcal{T}_5 \overset{\min}{\Rightarrow} 0 \\ \mathcal{T}_2 \overset{\min}{\Rightarrow} \mathcal{T}_1 & \mathcal{T}'_{\mathsf{y}} \overset{\min}{\Rightarrow} 0 & \mathcal{E}_2 \overset{\min}{\Rightarrow} \mathcal{E}_1 \\ \mathcal{E}_4 \overset{\min}{\Rightarrow} \mathcal{E}_1 \cap \neg\mathsf{Fail} & \mathcal{E}_3 \overset{\min}{\Rightarrow} \mathcal{E}_5 & \mathcal{T}_{\mathtt{f}} \overset{\max}{\Rightarrow} \mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_1} \mathcal{T}_1 \end{array}$$

resulting in the final polymorphic constrained type

$$\begin{array}{c} \mathtt{substFail} : \forall \mathcal{T}_1, \mathcal{T}_{\mathtt{d}}, \mathcal{E}_1, \mathcal{E}_5 . (\mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_1} \mathcal{T}_1) \xrightarrow{0} \mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_5 \cup \mathcal{E}_1 \cap \neg\{\mathsf{Fail}\}} \mathcal{T}_1 \backslash \\ \mathcal{E}_1 \subseteq_{\mathsf{s}} \mathsf{Pat}[\mathsf{Fail}(\mathsf{exn}(\mathcal{E}_5)), \mathsf{Fail}(1)] \end{array}$$

To conclude our example, assume we apply `substFail` to arguments

$$\begin{aligned} \mathtt{f} &: \mathsf{exn}(\mathcal{E}_6) \xrightarrow{\mathcal{E}_6} \mathsf{int} \\ \mathtt{d} &: \mathsf{exn}(\mathsf{Fail}(\mathsf{Subscript})) \end{aligned}$$

Applying Rule [APP] twice results in the constraints

$$\begin{aligned} \mathsf{exn}(\mathcal{E}_6) \xrightarrow{\mathcal{E}_6} \mathsf{int} &\subseteq_{\mathsf{ft}} \mathcal{T}_{\mathtt{d}} \xrightarrow{\mathcal{E}_1} \mathcal{T}_1 \\ \mathsf{exn}(\mathsf{Fail}(\mathsf{Subscript})) &\subseteq_{\mathsf{ft}} \mathcal{T}_{\mathtt{d}} \\ \mathcal{E}_1 &\subseteq_{\mathsf{s}} \mathsf{Pat}[\mathsf{Fail}(\mathsf{exn}(\mathcal{E}_5)), \mathsf{Fail}(1)] \end{aligned}$$

which in inductive form are

$$\begin{array}{rll} \mathcal{T}_{\mathtt{d}} \subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E}_6 & \mathsf{int} \subseteq_{\mathsf{ft}} \mathcal{T}_1 & \mathcal{E}_6 \subseteq_{\mathsf{s}} \mathcal{E}_1 \\ \mathsf{exn}(\mathsf{Fail}(\mathsf{Subscript})) \subseteq_{\mathsf{ft}} \mathcal{T}_{\mathtt{d}} & \mathcal{E}_1 \subseteq_{\mathsf{s}} \mathsf{Pat}[\mathsf{Fail}(\mathsf{exn}(\mathcal{E}_5)), \mathsf{Fail}(1)] & \end{array}$$

Applying transitivity results in the constraints

$$\begin{aligned} \mathsf{exn}(\mathsf{Fail}(\mathsf{Subscript})) &\subseteq_{\mathsf{ft}} \mathsf{exn}(\mathcal{E}_6) \\ \mathsf{Fail}(\mathsf{Subscript}) &\subseteq_{\mathsf{s}} \mathcal{E}_6 \\ \mathsf{Fail}(\mathsf{Subscript}) &\subseteq_{\mathsf{s}} \mathsf{Pat}[\mathsf{Fail}(\mathsf{exn}(\mathcal{E}_5)), \mathsf{Fail}(1)] \\ \mathsf{Subscript} &\subseteq_{\mathsf{s}} \mathcal{E}_5 \end{aligned}$$

The result type of the application is $\mathcal{T}_1$ which can be minimized to int. Similarly, the overall effect is $\mathcal{E}_5 \cup \mathcal{E}_1 \cap \neg\{\mathsf{Fail}\}$, which can be minimized to $\mathsf{Subscript}$. The example shows that the type of `substFail` contains enough information to relate the type of the argument `d` with the exceptions raised by `f` and resulting in the filtering of Fail and extraction of the argument exception $\mathsf{Subscript}$ carried by Fail.

8.2.3 Exception Inference for Full SML

Some remarks about extending the described exception inference to full SML are in order.

- Standard ML has parameterized modules called functors. Our exception inference cannot directly analyze functors due to unresolved exception aliasing. We use a tool to expand all functor applications prior to performing the analysis.

- Exception declarations in SML are *generative*, meaning that a declaration produces a fresh exception constructor (distinct from all other constructors) at every evaluation. Exception declarations within `let` expressions can therefore give rise to an unbounded number of distinct exceptions, all sharing the same name. Since our exception inference matches exceptions by constructor name, the filtering of exceptions is only sound if a particular exception name refers to a unique exception constructor. Consequently, only exceptions declared at top-level can safely be filtered *by name*, since such exception declarations give rise to exactly one exception constructor. Our analysis implementation classifies exceptions as either *top-level*, in which case the exception can be filtered, or as *generative*, in which case the exception is never filtered. A special case in the pattern rules generates remainder types for generative exception patterns that do not filter the exception. In practice, we find that practically all exception declarations appear at top-level or can safely be moved to top-level without changing the semantics of the program. It is worth noting that this problem does not arise in the CAML dialect of ML, since CAML disallows exception declarations within `let` expressions [89].

- ML record types are modeled using closed Row-expressions.

- In practice it is necessary to know *where* an exception is raised in a program. Effects are thus modeled as a set of pairs $E@p$, where E is a Set-expression describing a set of exceptions, and P is a Set-expression describing a set of program source positions. In terms of the implementation, this amounts to adding a binary Set-constructor $\texttt{@} : \texttt{s}\ \texttt{s} \rightarrow \texttt{s}$ and adjusting Rule [Raise] where source positions are introduced, and Rule [Handle], where source positions are removed.

- Datatypes hide the internal structure of values. We must ensure that exceptions do not "disappear" into datatypes. To this end, we extend datatypes containing exception values (directly or through functions) with a single extra type parameter to capture these exceptions. To illustrate this technique, consider the following excerpt from a hash-table implementation of the SML/NJ library.

```
datatype 'a hash_table =
      HT of {not_found : exn,
             table     : ...,
             n_items   : ...}

fun mkTable (sizeHint,notFound) =
      (HT {not_found = notFound,
           table = ...,
           n_items = ...})
```

The function `mkTable` is used to create an empty hash-table. It takes an exception argument `notFound`, which is stored as part of the hash-table data structure. This exception value is raised during `lookup` and `remove` operations on keys that are not part of the table. In order to correctly report the exception raised by `lookup` or `remove`, we need to attach the exception used when creating the hash-table to the type of the hash-table. In general, we augment types with exception information by parameterizing types with an extra exception argument. In the example, the `hash_table` type constructor takes a second argument denoting the set of exceptions potentially stored in the hash-table data-structure. Thus, the type of `mkTable` is

$$\langle \mathsf{int}, \mathsf{exn}(\mathcal{E})\rangle \xrightarrow{0} (\mathcal{T}, \mathcal{E})\ \texttt{hash\_table}$$

which states that `mkTable` can be applied to a pair consisting of an integer and an exception (whose exception names are bound to $\mathcal{E}$), and it returns a hash-table data-structure containing elements of type $\mathcal{T}$ and exceptions $\mathcal{E}$. The dependency between the `hash_table` type and the exceptions raised by the `lookup` function appears clearly in the type of `lookup`:

$$(\mathcal{T}, \mathcal{E})\ \texttt{hash\_table} \rightarrow \texttt{key} \xrightarrow{\mathcal{E}@P} \mathcal{T}$$

Any exceptions carried by the hash-table may be raised when calling `lookup` (where P is the position of the `raise` expression within `lookup`).

In general, we infer for each datatype whether or not it carries any exception names and effects, and whether these effects appear covariantly, contravariantly, or non-variantly in the datatype. Our inference conflates all covariant (contravariant) exceptions carried by a datatype.

- Mutable references in ML are treated using the same trick applied in the Points-to analysis of Section 8.1. A reference is treated as an object with a get-method of type unit $\rightarrow \mathcal{X}$ and a set-method of type $\mathcal{X} \rightarrow$ unit, where $\mathcal{X}$ represents the contents of the reference cell. Such a cell is represented using a constructor with signature

  $$\mathsf{ref} : \mathbf{ft}\ \overline{\mathbf{ft}} \rightarrow \mathbf{ft}$$

  where the first FlowTerm-argument refers to the range of the get-method, and the second FlowTerm-argument refers to the domain of the set-method (and is thus contravariant). The types assigned by our system to ML operations on references are

  $$\begin{aligned} \texttt{ref} &: \mathcal{T} \rightarrow \mathsf{ref}(\mathcal{T}, \mathcal{T}) \\ ! &: \mathsf{ref}(\mathcal{T}, 0) \rightarrow \mathcal{T} \\ := &: \langle \mathsf{ref}(1, \mathcal{T}), \mathcal{T} \rangle \rightarrow \mathsf{unit} \end{aligned}$$

  The first function `ref` creates a fresh reference with contents $\mathcal{T}$. The dereference operator ! expects a ref type, but extracts only the covariant argument, corresponding to a call to the get-method. Dually, the assignment function := expects a ref type and an argument of type $\mathcal{T}$ and updates the cell by accessing the contravariant argument of ref, corresponding to a call to the set-method.

We have successfully applied an implementation of our exception inference combined with a visualization tool to detect two previously unknown bugs caused by uncaught exceptions in the tools ML-Lex and ML-Burg distributed with the SML/NJ compiler [27].

8.2.4 Precision-Efficiency Variations

We briefly discuss two variations of our analysis, showing how the mixed constraint formalism of BANE helps evaluate precision-efficiency tradeoffs. We refer to the system we have presented so far as the FlowTerm-Set exception inference system, since it is based on FlowTerm and Set expressions.

A coarsening of the analysis can be obtained by replacing the use of the FlowTerm-sort for the type structure of our inference with the Term-sort, and strengthening the FlowTerm-inclusion constraints $\subseteq_{\mathsf{ft}}$ to equality constraints between Term-expressions $=_{\mathsf{t}}$. This coarsening results in less precise exception information due to the back-flow of exception information resulting from the use of equality constraints. We refer to this system as the Term-Set system.

Going in the direction of more expressive, but also more expensive sorts, we can replace the FlowTerm-sort with the Set-sort, thus using Set-expressions for both the type and the effect structure. We call this system Set-Set system. Note that there is no gain in precision by going to this variation unless we refine the types further to take advantage of the Set-structure. A possible refinement would be to model constructors of a datatype separately.

Chapter 9 characterizes the precision-efficiency tradeoffs of the three variations of exception inference by running experiments on all systems and characterizing the number of exceptions found and the overall performance of the analysis.

8.2.5 Related Exception Work

We are aware of two earlier approaches to uncaught exception detection for ML. Guzmán and Suárez [37] describe an extended type system for ML similar to, but less powerful than the one presented here. They do not treat exceptions as first class values, and they ignore value-carrying exceptions. Yi [91] describes a collecting interpretation for estimating uncaught exceptions in ML. His analysis is presented as an abstract interpretation [17] and is much finer grained than Guzmán and Suárez' approach [37] or the system described here, but is also slow in practice.

Independently, Yi [93] developed an approach to exception inference of ML based on a control-flow analysis, followed by an exception analysis based on set constraints. Since control-flow depends on exception information, this approach conservatively approximates exception information during the control-flow analysis. This step is justified since exceptions rarely carry function arguments. The resulting analysis is comparable in precision and performance to the one presented here.

More recently, Pessaux and Leroy [70] have proposed an exception inference for CAML based on types and Row-expressions. Their work can be viewed as yet another refinement expressible in BANE, where exception sets are represented by labels present in Row-expressions instead of Set-expressions.

Chapter 9

Experiments

In this chapter we evaluate the implementation strategies employed in BANE and the benefits provided by a reusable mixed constraint framework. We show the impact of BANE's implementation strategies by running a number of experiments exercising each implementation feature in isolation. The experiments show that BANE enables the resolution of constraint problems that are orders of magnitude larger than previously reported results, and that precision-efficiency variations can be studied through appropriate sort selections.

Furthermore we experimentally contrast BANE's inductive transitive closure (for computing the consistency of constraint graphs) with the standard transitive closure algorithm.

The chapter is organized as follows. Section 9.1 discusses general aspects concerning the measurement methodology. Section 9.2 shows that for our experiments, Set-constraint graphs contain large strongly connected components (SCC) and that these components are a major cost during resolution. The section then shows the effectiveness of BANE's novel partial online cycle elimination strategy. Section 9.3 evaluates the effect of projection merging on constraint resolution. We show that projection merging speeds up the analysis of the largest programs by an order of magnitude, but has little effect on small programs. Section 9.4 studies how the use of inductive form (IF) compares to the standard form (SF) used traditionally for solving inclusion constraints. We show that SF is superior to IF only in the presence of cycles. When cycles are eliminated using our online strategy, then IF performs much better than SF without cycle elimination. We also show that BANE's strategy for cycle detection can be applied to SF resulting in similar speedups as for IF. Section 9.4 concludes with a discussion of a clear advantage of IF over SF, namely the ability to compute the transitive lower-bound on demand.

9.1 Measurement Methodology

All experiments were performed using a single processor on a SPARC Enterprise-5000 with 2GB of memory. Reported execution times are best out of three runs on a lightly loaded machine.

The absolute execution times include the execution of a fair amount of instrumentation code present in BANE. Removal of that code may yield better absolute times.

						Strongly Connected Components (SCC)					
						Initial graph			Final graph		
Benchmark	AST Nodes	LOC	Total #Vars	Nodes	Initial Edges	#Vars	#SCC	#Max	#Vars	#SCC	#Max
allroots	700	426	126	74	110	5	2	3	20	5	11
diff.diffh	935	293	186	133	184	6	3	2	13	6	3
anagram	1078	344	209	127	191	10	5	2	24	9	5
genetic	1412	323	226	137	196	4	2	2	14	5	4
ks	2284	574	326	225	329	6	3	2	39	3	34
ul	2395	441	196	146	192	4	2	2	6	3	2
ft	3027	1180	393	278	392	0	0	—	68	5	55
compress	3333	651	251	189	293	17	7	4	26	9	6
ratfor	5269	1532	605	565	747	20	7	6	89	8	65
compiler	5326	1888	442	597	725	12	6	2	35	8	20
assembler	6516	2980	971	849	942	9	4	3	88	10	27
ML-typecheck	6752	2410	794	722	976	29	9	9	248	15	144
eqntott	8117	2266	978	699	1066	50	15	12	177	22	55
simulator	10946	4216	1433	1261	1350	6	3	2	237	5	213
less-177	15179	11988	1848	1751	2195	30	12	4	273	18	202
li	16828	5761	3223	2653	3384	24	10	4	1267	6	1255
flex-2.4.7	29960	9345	3749	3363	4377	57	19	11	325	19	248
pmake	31148	18138	3312	3189	3938	63	24	8	669	21	592
make-3.72.1	36892	15213	4737	3781	5150	156	63	7	869	64	665
inform-5.5	38874	12957	4402	4535	5338	34	15	6	505	22	407
tar-1.11.2	41035	18293	4158	2957	5073	347	89	29	848	82	527
sgmls-1.1	44533	30941	4253	3953	5268	91	37	9	910	43	767
screen-3.5.2	49292	23919	6516	4849	6050	108	46	6	914	39	824
cvs-1.3	51223	31130	6971	5358	6936	340	87	31	743	95	379
espresso	56938	21537	6327	4855	7839	333	149	10	1396	171	976
gawk-3.0.3	71140	28326	6237	4653	7134	400	58	86	1343	45	1087
povray-2.2	87391	59689	7712	5863	8391	198	87	9	1509	75	1245

Table 9.1: Benchmark data common to all experiments

However, all experiments were run with all instrumentation code in place.

For the Points-to experiments, library functions were assumed to have no effect on the pointer graph, a naive assumption. For the exception inference experiment, conservative signatures of all library functions were provided.

9.2 Online Cycle Elimination

In this section we show that *cycle elimination* in the constraint graph is a crucial step to making inclusion constraint analyses scale to large problems with good performance. Cyclic constraints have the form $\mathcal{X}_1 \subseteq \mathcal{X}_2 \subseteq \mathcal{X}_3 \subseteq \ldots \subseteq \mathcal{X}_n \subseteq \mathcal{X}_1$ where the $\mathcal{X}_i$ are variables. All variables on such a cycle are equal in all solutions of the constraints, and thus the cycle can be collapsed to a single variable. Cycles arise and can be eliminated in the constraints of all sorts in BANE. The experiments here focus on Set-constraints.

We show that the partial online cycle detection employed in BANE is both efficient in practice with a small constant time overhead on every edge addition, and still able to find and eliminates on average 90% of all variables involved in cycles. For our benchmarks, this approach radically improves the scaling behavior, making points-to analysis of large programs practical. For example, online cycle elimination provides speedups of points-to analysis of large programs (more than 10000 lines) of up to a factor of 50.

The reason why cycle elimination produces such large speedups is that cycles may induce a non-linear amount of work that can be avoided when the cycle is collapsed. To see this suppose that V is a strongly connected component of size n and there are edges from a set of sources $E_1..E_m$ to some variables in V. In the worst case, closing the constraint graph will add edges from each source E_i to each variable in V, *i.e.*, mn edges. Furthermore,

Experiment	Description
IF-Plain	Inductive form, no cycle elimination
IF-Oracle	Inductive form, with full (oracle) cycle elimination
IF-Online	Inductive form, with online cycle elimination

Table 9.2: Experiments

if there are redundant paths within V, then the closure may try to add the same edge redundantly up to n times. If on the other hand the component V is collapsed to a single variable before adding edges $E_1..E_m$ incident on V, then the work to add them is reduced to m steps.

9.2.1 Measurements

The measurements in this section use our Set-constraint formulation of Andersen's Points-to analysis, run on the C benchmark programs shown in Table 9.1. For each benchmark, the table lists the number of abstract syntax tree (AST) nodes, the number of lines in the preprocessed source, the number of set variables, the total number of distinct nodes in the graph (sources, variables, and sinks), and the number of edges in the initial constraints (before closing the graph). Furthermore, the table contains the combined size of all non-trivial strongly connected components (#Vars), the number of components (#SCC), and the size of the largest component (#Max), both for the initial graph (before closure) and for the final graph (in any experiment). The difference in the combined size of SCCs between the initial and the final graph shows the need for online cycle elimination. Cycles are formed dynamically through new edges added by the structural closure rules. If all cycles were present in the initial graph, online cycle elimination would be unnecessary. For example, for the benchmark `povray`, only 13% of all variables involved in cycles are apparent in the initial constraint graph.

We performed the three experiments shown in Table 9.2. The first experiment, IF-Plain, runs the points-to analysis without performing cycle elimination. Experiment IF-Oracle precomputes the strongly connected components of the final graph and uses that information as an oracle during the analysis. Whenever a fresh Set-variable is created, the oracle predicts to which strongly connected component the variable will eventually belong in the final graph. We designate a unique witness variable of that component and add an alias edge from the fresh variable to the witness. As a result, the oracle experiment uses only a single variable (witness) for each strongly connected component, and thus the constraint graphs are acyclic at all times. Since the oracle experiment avoids all unnecessary work related to cycles in the constraint graph (perfect cycle elimination), it provides a lower bound[1] for the last experiment—IF-Online—which uses BANE's online cycle detection and elimination algorithm. (The prefix IF- of these experiments refers to the use of the inductive closure for the constraint graph. In Section 9.4 will we examine similar experiments using the standard transitive closure SF.)

[1]In fact, the oracle as implemented has true future knowledge, since it combines variables even before the cycle the variables belong to is present in the graph.

	IF-Plain			IF-Oracle		
Benchmark	Edges	Work	Time(s)	Edges	Work	Time(s)
allroots	291	34	0.13	522	35	0.14
diff.diffh	404	46	0.18	723	44	0.20
anagram	444	45	0.19	702	11	0.21
genetic	438	25	0.21	895	23	0.22
ks	2649	6134	0.50	1519	305	0.44
ul	306	51	0.31	1220	51	0.36
ft	3480	10496	0.71	1798	117	0.54
compress	463	67	0.35	1658	57	0.44
ratfor	4986	10558	1.20	4257	265	1.34
compiler	1268	265	0.72	3323	82	1.07
assembler	3173	2514	1.30	5479	444	1.92
ML-typecheck	18284	219631	7.07	6528	931	1.84
eqntott	5614	6908	1.41	5858	369	1.59
simulator	15061	292900	7.14	7179	1128	2.25
less-177	32237	455486	12.97	11894	2456	3.74
li	1146729	156695062	3577.03	15883	46302	9.63
flex-2.4.7	76662	1625239	37.32	29205	1160	13.49
pmake	390037	30756702	649.46	20622	25735	7.44
make-3.72.1	800101	104545496	2370.49	26811	91328	12.71
inform-5.5	196431	13856788	269.09	26491	20474	10.68
tar-1.11.2	248733	16414402	316.73	26133	15207	9.20
sgmls-1.1	998305	178723881	3690.27	45302	92659	21.22
screen-3.5.2	894383	96327887	1995.84	33965	64764	14.79
cvs-1.3	171643	5258119	123.59	35821	14243	12.46
espresso	916001	81191353	1671.40	39131	80024	17.25
gawk-3.0.3	1041525	106585396	2314.35	43717	57869	18.65
povray-2.2	2171746	343496620	8058.49	66258	150098	29.55

Table 9.3: Benchmark data for IF-Plain and IF-Oracle

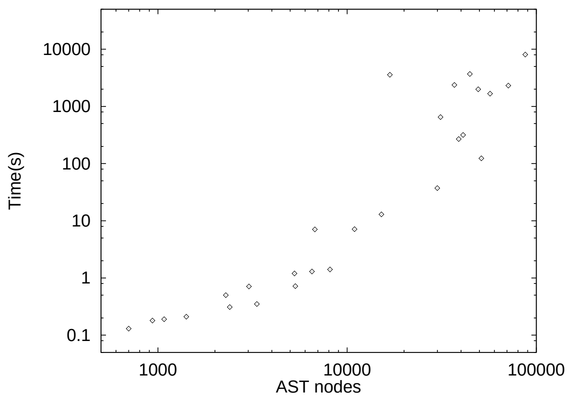

Figure 9.1: IF without cycle elimination

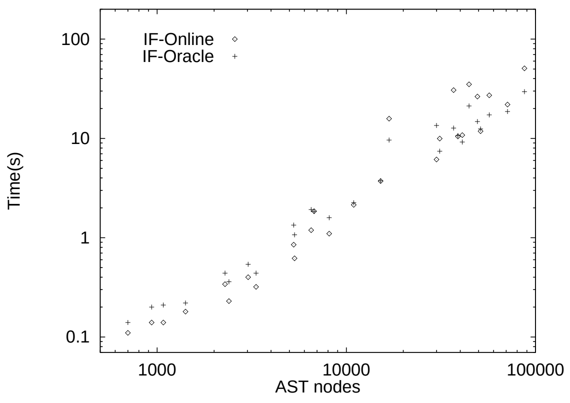

Figure 9.2: Analysis times with cycle detection and oracle

Table 9.3 shows the results for the first two experiments. For each benchmark and experiment, we report the number of edges in the final graph, the total number of edge additions (Work) including redundant ones, and the execution time in seconds. The reported CPU times are best out of three runs. The reported times include the time to compute the transitive lower bounds of all points-to variables to compute the points-to sets. Note the large number of redundant edge additions for IF-Plain. The low numbers for the oracle runs IF-Oracle in Table 9.3 show that the bulk of work and execution time is attributable to strongly connected components in the constraint graph. Without cycles, the points-to analysis scales very well. For example, the analysis time for `povray-2.2` with IF-Oracle is over 250 times faster than with IF-Plain.

		SCC	IF-Online			
Benchmark	AST	Elim.	Edges	Work	Time(s)	%CD
allroots	700	15	265	311	0.11	18.18%
diff.diffh	935	13	369	423	0.14	0.00%
anagram	1078	23	372	415	0.14	0.00%
genetic	1412	11	424	451	0.18	5.56%
ks	2284	33	1027	1909	0.34	11.76%
ul	2395	6	286	337	0.23	8.70%
ft	3027	54	897	1370	0.40	7.50%
compress	3333	24	397	487	0.32	0.00%
ratfor	5269	64	1758	2515	0.85	5.88%
compiler	5326	22	1047	1240	0.62	1.61%
assembler	6516	69	2174	2897	1.19	3.36%
ML-typecheck	6752	227	2858	7924	1.85	12.97%
eqntott	8117	150	2496	3805	1.10	7.27%
simulator	10946	177	3652	8243	2.14	14.49%
less-177	15179	235	7040	15377	3.72	19.62%
li	16828	1013	10100	100410	15.80	27.34%
flex-2.4.7	29960	281	7531	10052	6.14	2.93%
pmake	31148	582	10629	53327	9.97	20.26%
make-3.72.1	36892	798	18123	238148	30.66	36.24%
inform-5.5	38874	422	14139	51621	10.47	17.10%
tar-1.11.2	41035	745	15366	48610	10.79	19.46%
sgmls-1.1	44533	815	17358	208348	35.03	29.35%
screen-3.5.2	49292	823	19707	159340	26.42	25.47%
cvs-1.3	51223	652	20289	49522	11.82	14.47%
espresso	56938	1208	21671	141052	27.19	24.20%
gawk-3.0.3	71140	1122	18613	111359	21.93	20.79%
povray-2.2	87391	1355	51216	286044	50.76	28.09%

Table 9.4: Benchmark data for IF-Online

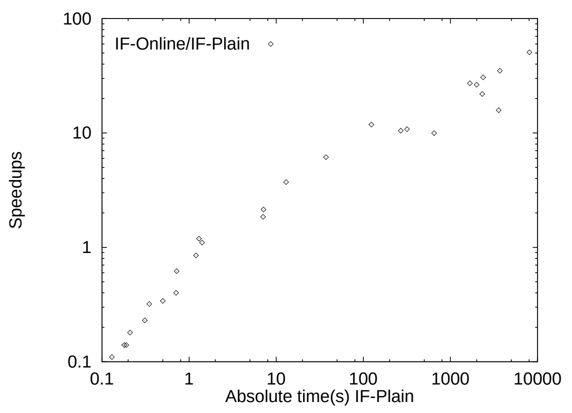

Figure 9.3: Speedups through online cycle detection

Figure 9.1 plots the analysis time for IF-Plain without cycle elimination against the number of AST nodes of the parsed program. As the size exceeds 15000 AST nodes there are many benchmarks where the analysis becomes impractical.

Table 9.4 reports the measurement results for the online cycle elimination experiment IF-Online. In addition to the information shown for the plain and oracle experiments, the table contains the number of variables that were eliminated through cycle detection and the fraction of analysis time spent in cycle detection (%CD). As the execution times and work counts show, online cycle elimination is very effective for medium and large programs. Figure 9.2 plots the analysis times for online cycle elimination and the oracle experiment

	Detection			Coverage		
Benchmark	Vst	Hit%	Len	Total%	Avg%	Max%
allroots	2.03	9.84%	2.00	75.00%	87.27%	36.36%
diff.diffh	1.53	7.30%	2.00	100.00%	100.00%	100.00%
anagram	1.66	26.16%	2.12	100.00%	100.00%	100.00%
genetic	1.59	5.37%	2.50	78.57%	85.00%	75.00%
ks	1.88	7.50%	2.53	86.84%	95.10%	85.29%
ul	1.24	16.66%	1.00	100.00%	100.00%	100.00%
ft	2.24	8.13%	2.60	79.41%	70.06%	83.64%
compress	1.68	17.95%	2.21	92.31%	96.30%	66.67%
ratfor	2.19	5.94%	2.82	77.11%	88.89%	69.49%
compiler	1.46	5.13%	2.36	62.86%	88.33%	40.00%
assembler	1.67	7.50%	2.35	78.41%	83.75%	70.37%
ML-typecheck	2.15	4.67%	3.23	93.42%	93.10%	95.80%
eqntott	2.32	10.52%	2.73	87.21%	91.09%	94.23%
simulator	2.29	3.71%	2.39	76.96%	94.27%	71.36%
less-177	2.38	2.28%	2.63	86.08%	95.11%	83.66%
li	1.91	2.04%	2.84	80.72%	96.73%	80.37%
flex-2.4.7	1.85	9.57%	2.74	88.92%	94.74%	88.48%
pmake	1.70	1.31%	2.65	88.18%	88.39%	88.68%
make-3.72.1	1.90	0.49%	3.13	93.22%	94.17%	93.61%
inform-5.5	1.60	1.51%	2.62	90.17%	94.34%	90.86%
tar-1.11.2	1.98	1.83%	3.31	90.63%	95.72%	89.55%
sgmls-1.1	1.65	0.47%	2.67	94.44%	94.40%	94.89%
screen-3.5.2	1.45	0.84%	2.86	95.37%	96.78%	94.44%
cvs-1.3	1.90	2.17%	2.52	90.06%	97.31%	85.83%
espresso	1.84	2.71%	2.68	87.92%	99.49%	83.28%
gawk-3.0.3	1.65	1.41%	2.73	87.04%	91.52%	86.45%
povray-2.2	2.25	0.60%	2.61	93.26%	95.89%	92.77%

Table 9.5: Cycle detection statistics

(note the scale change w.r.t. Figure 9.1). IF-Online stays relatively close to the oracle times indicating that while our cycle detection algorithm is not perfect, there is not much room for improvement.

Figure 9.3 shows the total speedup of IF-Online over IF-Plain. To show that our technique helps scaling, we plot the speedups vs. the absolute execution time of IF-Plain. As we go to larger problems that take longer to run without cycle elimination, the speedups also grow. Cycle elimination thus helps scaling and does not simply improve the execution time by a constant factor. For very small programs, the cost of cycle elimination outweighs the benefits, but the absolute extra cost is small ($<$ 1 second). For medium and large programs, online cycle elimination improves analysis times substantially, for large programs by more than an order of magnitude.

The absolute speedups of IF-Online over IF-Plain should be taken with a grain of salt. The inductive transitive closure actually performs very badly in the presence of cycles, due to the fact that it adds transitive edges between variables as well as transitive edges between sources and sinks. Constraint graph closure based on the standard transitive closure rule performs a little better in the presence of large strongly connected components than IF, but the scaling behavior is essentially the same. We will study the relationship between inductive form IF and standard form SF in more detail in Section 9.4.

Table 9.5 shows statistics on the cycle detection performed during the points-to analyses. For each benchmark, the table reports the average number of variables visited during a cycle detection (Vst), the hit rate of finding an actual cycle (Hit%), and the average cycle length (Len). Furthermore, we give three measures for the quality of the detection. The overall detection fraction (Total%) is the ratio between the number of variables in the final graph that we found to be on a cycle, over the total number of variables in non-trivial strongly connected components in the final graph. The second measure (Avg%) gives the

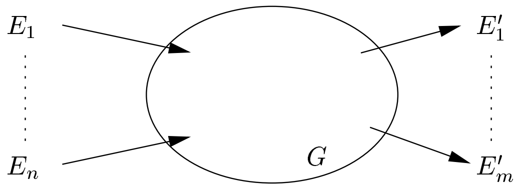

Figure 9.4: Graph Schema

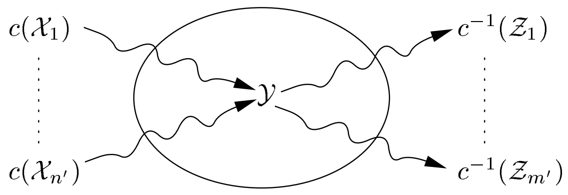

Figure 9.5: Paths from sources to $\mathcal{Y}$ to sinks

average detection fraction per strongly connected component, *i.e.*, for each SCC V in the final graph, the fraction $|V'|/|V|$, where V' is the largest single component we detected within V. Finally, the third measure (Max%) gives the detection fraction for the largest component in the final graph, *i.e.*, if V is the largest component, the fraction $|V'|/|V|$, where V' is the largest single component within V that we detected. The numbers clearly show that even though the average cycle length detected is small, detected cycles combine into larger detected components so that the overall coverage is high. Recall from Table 9.1 that the largest strongly connected component in the final graph is substantial for each benchmark. For example, the largest SCC in the final graph of `povray-2.2` contains 1299 variables. The 92.77% detection of the maximal component for `povray-2.2` in Table 9.5 states that we detected a single component consisting of 1205 variables within this largest component.

9.3 Projection Merging

This section takes a close look at the impact of the projection merging technique of BANE (Section 7.2.2) on constraint graph size and analysis execution times. We explain how projection merging can in the best case prevent the addition of a quadratic number of edges to a graph and show through measurements of the points-to analysis that for large constraint problems, projection merging results in speedups of another order of magnitude over cycle elimination alone.

To gain insight into how projection merging changes the structure of a constraint graph during closure, consider the schematic constraint graph in Figure 9.4. Expressions $E_1..E_n$ are sources and $E'_1..E'_m$ are sinks and we assume that the graph has size $O(n+m)$ and that there exist paths in the graph connecting each source E_i with each sink E'_j. Furthermore, assume that each source E_i has the form $c(\mathcal{X}_i)$ for some fixed unary constructor

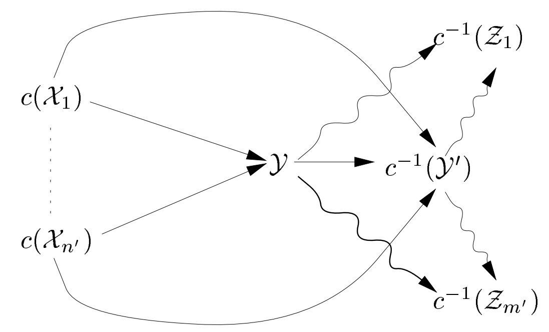

Figure 9.6: Paths added through projection merging

c, and that each sink E'_j has the form $\mathsf{ProjPat}(\mathsf{c}, 1, \mathcal{Z}_j)$. Closing this graph under any of the transitive closure rules and structural closure rules we have presented results in the addition of the nm edges $\mathcal{X}_i \longrightarrow \mathcal{Z}_j$, for $i = 1..n$, $j = 1..m$. The closure of a graph of size $O(n + m)$ may thus produces a graph of size $O(nm)$. Now observe that if the original graph has size $O(n + m)$, and there exists paths from each source E_i to each sink E_j, then these paths must share some variable nodes $\mathcal{Y}$. In fact there must exist variable nodes $\mathcal{Y}$, such that there are paths from a fraction $O(n)$ of the sources E_i to $\mathcal{Y}$ and paths from $\mathcal{Y}$ to a fraction $O(m)$ of the sinks E'_j. This situation is depicted in Figure 9.5, where we assume that n' of the sources E_i have paths to $\mathcal{Y}$ and there are paths from $\mathcal{Y}$ to m' of the sinks E'_j (we use straight edges to denote graph edges and wavy lines to denote paths. We also abbreviate $\mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Z}_i)$ in the graph by $c^{-1}(\mathcal{Z}_i)$). If our graph is being closed under the inductive transitive closure rule and only $O(n + m)$ transitive edges between variables are added[2], we can furthermore assume without loss of generality that variable $\mathcal{Y}$ is the variable with minimum index on all paths $E_i \longrightarrow^* E'_j$. Without projection merging, the inductive transitive closure rule eventually adds direct edges $E_i \longrightarrow \mathcal{Y}$ and $\mathcal{Y} \longrightarrow E'_j$ for all paths $E_i \longrightarrow^* \mathcal{Y}$ and $\mathcal{Y} \longrightarrow^* \mathcal{Z}_j$. The resulting transitive constraints $E_i \subseteq_{\mathsf{s}} E_j$ on $\mathcal{Y}$ then generate the $n'm'$ edges $\mathcal{X}_i \longrightarrow \mathcal{Z}_j$.

Now consider the impact of projection merging in the situation of Figure 9.5. Projection merging will create a new sink $\mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y}')$ with an edge $\mathcal{Y} \longrightarrow \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y}')$, and paths $\mathcal{Y}' \longrightarrow^* \mathcal{Z}_j$ for the m' sinks $\mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Z}_j)$ reachable from $\mathcal{Y}$. We still end up with edges $E_i \longrightarrow \mathcal{Y}$ added by the inductive transitive closure, but now we only add the n' transitive constraints $E_i \subseteq_{\mathsf{s}} \mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Y}')$, which results in the n' edges $\mathcal{X}_i \longrightarrow \mathcal{Y}'$. As a result, we have created paths $\mathcal{X}_i \longrightarrow^* \mathcal{Z}_j$ for the n' sources $c(\mathcal{X}_i)$ and the m' sinks $\mathsf{ProjPat}(\mathsf{c}, i, \mathcal{Z}_j)$ connected through $\mathcal{Y}$ (depicted in Figure 9.6). The sharing in these new paths mirrors the sharing present in the paths $E_i \longrightarrow^* E_j$, since all the new paths $\mathcal{X}_i \longrightarrow^* \mathcal{Z}_j$ pass through the fresh variable $\mathcal{Y}'$. As a result, we can expect the graph closed using projection merging to still have $O(n + m)$ edges, assuming again that the inductive transitive closure rule only adds a linear number of edges directly between variables.

[2]We describe a theoretical model for estimating the cost of closing constraint graphs under the inductive transitive closure rule in a separate publication [28]. For random graphs and any ordering o of the variables, the expected number of transitive edges added between variables is linear.

		IF-PM				Addtl	Reduction w.r.t. IF-Online		
Benchmark	AST	#Vars	Edges	Work	Time(s)	#Vars	Edges	Work	Time
allroots	700	212	365	59	0.16	86	0.73	0.58	0.69
diff.diffh	935	277	514	46	0.17	100	0.72	1.00	0.82
anagram	1078	316	502	18	0.20	108	0.74	1.39	0.70
genetic	1412	438	707	23	0.25	212	0.60	1.04	0.72
ks	2284	582	1288	321	0.52	266	0.80	2.13	0.65
ul	2395	258	326	83	0.28	67	0.88	0.61	0.82
ft	3027	598	1084	190	0.53	209	0.83	1.45	0.75
compress	3333	393	582	149	0.40	152	0.68	0.43	0.80
ratfor	5269	845	2158	356	1.01	292	0.81	1.51	0.84
compiler	5326	667	1276	199	0.91	258	0.82	0.83	0.68
assembler	6516	1494	3227	869	1.74	528	0.67	0.63	0.68
ML-typecheck	6752	1252	3184	1405	2.06	473	0.90	2.39	0.90
eqntott	8117	1608	3260	657	1.57	688	0.77	1.44	0.70
simulator	10946	2106	4452	666	2.08	703	0.82	6.15	1.03
less-177	15179	2441	6921	1812	3.22	607	1.02	4.11	1.16
li	16828	4061	11567	4475	6.54	872	0.87	19.21	2.42
flex-2.4.7	29960	5318	9811	1751	7.43	1664	0.77	1.01	0.83
pmake	31148	5304	12182	3997	6.86	2032	0.87	10.11	1.45
make-3.72.1	36892	7164	33820	150028	43.89	2558	0.54	1.38	0.70
inform-5.5	38874	8197	18920	3966	9.66	4161	0.75	8.85	1.08
tar-1.11.2	41035	5576	17282	3607	7.69	1562	0.89	8.57	1.40
sgmls-1.1	44533	6085	24181	13465	13.92	2245	0.72	13.69	2.52
screen-3.5.2	49292	8032	17309	5236	10.79	1760	1.14	25.39	2.45
cvs-1.3	51223	10618	23478	6223	12.20	3729	0.86	4.39	0.97
espresso	56938	10377	23186	6460	12.69	4247	0.93	17.65	2.14
gawk-3.0.3	71140	8563	20007	6296	14.54	2558	0.93	13.94	1.51
povray-2.2	87391	11812	38242	22654	23.91	4329	1.34	9.80	2.12
mume	312458	68233	218772	102215	170.24	16720	2.47	123.16	10.94
gs	504724	115976	245064	75254	272.73	19353	177.85	46.19	192.90
gcc	1168907	199424	707881	854424	808.27	63748	0.93	94.03	14.55

Table 9.6: Benchmark data for IF-PM

Table 9.6 contains the results of experiment IF-PM which consists of running the Points-to analysis of Section 8.1 on all the C benchmarks we used in the previous section and two additional large benchmarks, ghostscript (gs), and the GNU C-compiler (gcc). Each run used cycle elimination and projection merging. Projection merging without cycle elimination does not perform well at all, which is not surprising. As we have outlined above, projection merging creates paths between variables appearing in sources and variables appearing in sinks and these paths mirror the structure of the paths connecting the source and the sink. If the original paths contain cycles, projection merging results in similar additional cycles and we have already established in the previous section that cycles slow constraint resolution significantly.

Table 9.6 gives the total number of edges, total redundant work, and the total execution time in seconds for all benchmarks. Furthermore, we show reduction factors of these quantities with respect to the equivalent experiment without projection merging, and the number of additional variables generated by projection merging. The number of additional variables generated by projection merging varies, but is generally between 25-30%. This relatively large increase of variables explains that except for two benchmarks, projection merging results in a slight increase of the final graph size. The redundant work however is reduced between 1 and 100 fold for all but the small benchmarks which translates into substantial speedups in execution times. The speedups are thus primarily due to the reduced redundant work and not to reduction in the overall graph size. Redundant work results from redundant paths in the constraint graph. Benchmark gs seems to be a special case. Projection merging reduces the graph size of gs substantially more than for the other benchmarks. We currently have no explanation for this effect.

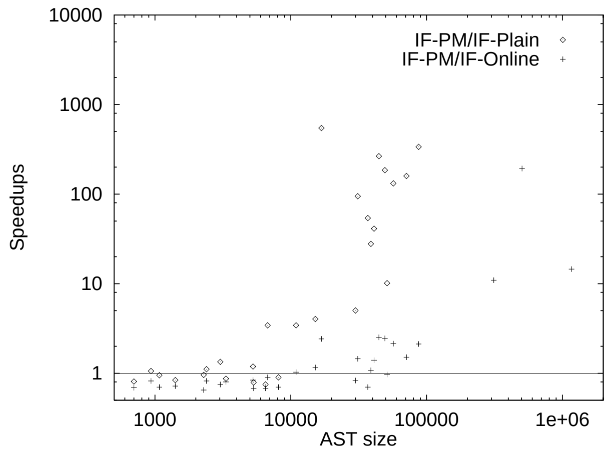

Figure 9.7: Speedups through projection merging

To see how projection merging helps reduce redundant work, consider again the schematic graph in Figure 9.4 and suppose that there are k variables $\mathcal{Y}_1..\mathcal{Y}_k$ for which the situation depicted in Figure 9.5 holds. In other words, there are k distinct paths between each source E_i and sink E'_j. Without projection merging, edge $\mathcal{X}_i \rightarrow \mathcal{Z}_j$ is added through each $\mathcal{Y}_k$, for $i = 1..n'$ and $j = 1..m'$, resulting in $(k-1)n'm'$ redundant work or $O(knm)$. Contrast this with the graph produced using projection merging. The structure of the paths created between $\mathcal{X}_i$ and $\mathcal{Z}_j$ mirror the paths between E_i and E'_j, and thus we create redundant paths through the fresh variables $\mathcal{Y}'_1..\mathcal{Y}'_k$. But the number of edges added is roughly $kn' + km'$ or $O(kn + km)$ without redundant work. The savings in redundant work can thus dwarf the savings in graph edges by a factor k. Our understanding of the exact effect of projection merging on the savings in redundant work is incomplete. We believe that there are positive interactions between projection merging and cycle elimination. Studying these effects however is tricky and time consuming, since the effects only show up on large constraint problems and cannot easily be reproduced in micro-benchmarks.

Figure 9.7 plots the speedup obtained through projection merging over cycle elimination alone, and the total speedup of projection merging and cycle elimination over IF-Plain. As is the case for cycle elimination, the speedups grow as we solve larger constraint problems, showing that projection merging substantially helps scaling. Since the largest three programs ran out of space for the IF-Plain experiment, projection merging and cycle elimination not only result in substantial speedups, but enable the analysis of large programs that cannot otherwise be analyzed.

9.4 Standard Form v.s. Inductive Form

In this section we compare the commonly used implementation strategy of set-based analysis [38], which represents constraint graphs in standard form (SF), with the inductive form (IF) used by BANE. Using our formulation of Andersen's points-to analysis as the example, we show the following four points.

1. Without cycle elimination, standard form performs better on the set of benchmarks than inductive form.

2. Inductive form with cycle elimination performs substantially better than standard form without cycle elimination.

3. Cycle detection and elimination for standard form is also effective. With cycle elimination, standard form performs somewhat worse than inductive form with cycle elimination.

4. Standard form computes the transitive lower bound (TLB) of all Set-variables explicitly. Often, the transitive lower bound of only a fraction of all variables needs to be inspected to extract useful information from an analysis. Inductive form enables the computation of TLB on demand which yields substantial overall speedup due to the fact that the total size of the TLB tends to grow quadratically with the size of the constraint problem.

The next subsections briefly recall from Section 6.2 the properties of the associated constraint graphs for the two representations under study SF and IF. Both forms use adjacency lists to represent edges. Every edge $(\mathcal{X}, \mathcal{Y})$ in a graph is represented exclusively either as a *predecessor edge* ($\mathcal{X} \in \mathit{pred}(\mathcal{Y})$) or as a *successor edge* ($\mathcal{Y} \in \mathit{succ}(\mathcal{X})$).

9.4.1 Standard Form

Standard form (SF) represents edges in constraint graphs as follows:

$\mathcal{X} \subseteq \mathcal{Y}$	$\mathcal{X} \longrightarrow \mathcal{Y}$	successor edge
$E \subseteq \mathcal{X}$	$E \dashrightarrow \mathcal{X}$	predecessor edge (E is a source)
$\mathcal{X} \subseteq E$	$\mathcal{X} \longrightarrow E$	successor edge (E is a sink)

We draw predecessor edges in graphs using dotted arrows and successor edges using plain arrows. New edges are added by the standard transitive closure rule (STCR). STCR can be expressed in terms of predecessor and successor edges:

$$L \dashrightarrow \mathcal{X} \longrightarrow R \quad \Leftrightarrow \quad L \subseteq_s R$$

Given a predecessor edge $L \dashrightarrow \mathcal{X}$ and a successor edge at $\mathcal{X} \longrightarrow R$, a new constraint $L \subseteq R$ is generated. We generate a constraint instead of an edge because only atomic constraints (constraints on variables) are represented in the graph. The constraint is thus first transformed into a set of atomic constraints using the structural resolution rules given in earlier chapters. Note that L is always a source in all transitive constraints $L \subseteq_{\mathsf{s}} R$ generated for SF. This follows from the fact that the only predecessor edges in the constraint graph are rooted at sources.

SF makes the transitive lower bound of all variables explicit by propagating sources forward to all reachable variables via the closure rule. The particular choice of successor

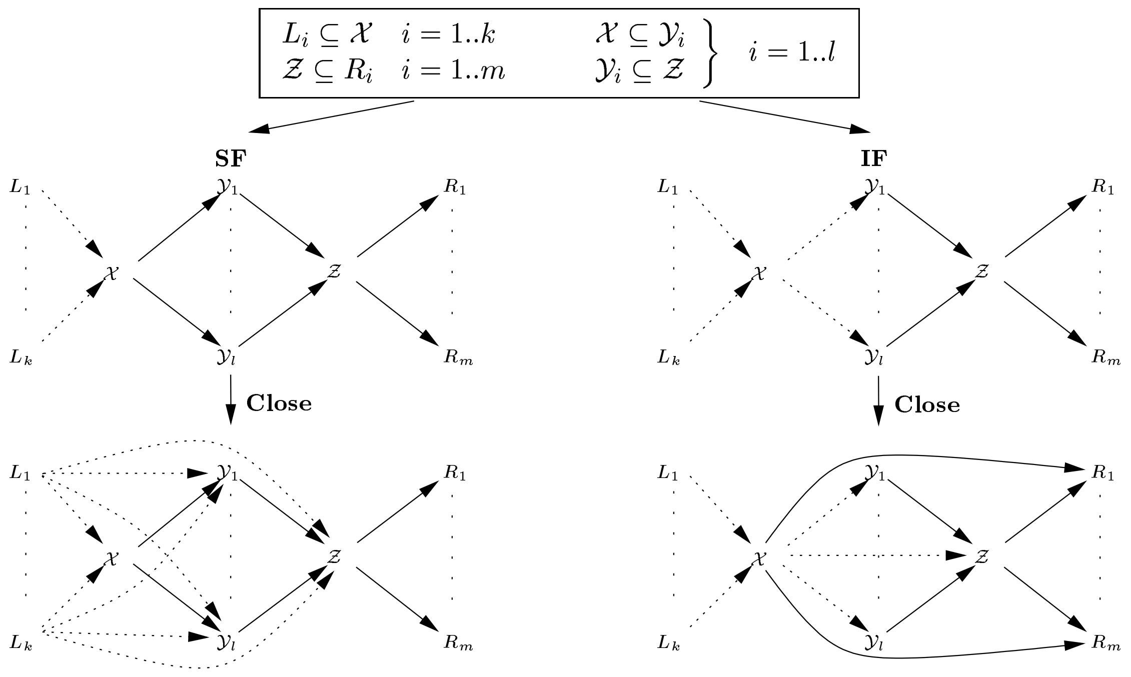
$L_i \subseteq \mathcal{X}$ $i = 1..k$
$\mathcal{Z} \subseteq R_i$ $i = 1..m$
$\mathcal{X} \subseteq \mathcal{Y}_i$
$\mathcal{Y}_i \subseteq \mathcal{Z}$
$i = 1..l$
SF
IF
Close
Close
L_1
L_k
$\mathcal{X}$
$\mathcal{Y}_1$
$\mathcal{Y}_l$
$\mathcal{Z}$
R_1
R_m

Figure 9.8: Example constraints in SF and IF

and predecessor representation is motivated by the need to implement the transitive closure rule locally. Given a variable $\mathcal{X}$, the closure rule must be applied exactly to all combinations of predecessor and the successor edges of $\mathcal{X}$.

Figure 9.8 shows an example system of constraints, the initial SF graph, and the resulting closed SF graph (left). The example assumes that set expressions $L_1 \ldots L_k$ are sources and $R_1 \ldots R_m$ are sinks. The closure of the standard form adds transitive edges from each source L_i to all variables reachable from $\mathcal{X}$ *i.e.*, $\mathcal{Y}_1 \ldots \mathcal{Y}_l, \mathcal{Z}$. Note that the edges from $L_1 \ldots L_k$ to $\mathcal{Z}$ are added l times each, namely along all l edges $\mathcal{Y}_i \longrightarrow \mathcal{Z}$. The total work of closing the graph is $2kl$ edge additions, of which $k(l-1)$ additions are redundant, plus the work resulting from the km constraints $L_i \subseteq R_j$ (not shown).

To see why cycle elimination can asymptotically reduce the amount of work to close a graph, suppose there is an extra edge $\mathcal{Z} \longrightarrow \mathcal{X}$ in Figure 9.8, forming a strongly connected component $\mathcal{X}, \mathcal{Y}_1, \ldots, \mathcal{Y}_l, \mathcal{Z}$. If we collapse this component before adding the transitive edges $L_i \dashrightarrow \mathcal{Y}_j$, none of the $2kl$ transitive edge additions $L_i \dashrightarrow \mathcal{Y}_j$ are performed (the km constraints $L_i \subseteq R_j$ are still produced of course).

9.4.2 Inductive Form

Inductive form (IF) exploits the fact that a variable-variable constraint $\mathcal{X} \subseteq \mathcal{Y}$ can be represented either as a successor edge ($\mathcal{Y} \in succ(\mathcal{X})$) or as a predecessor edge ($\mathcal{X} \in pred(\mathcal{Y})$). The representation for a particular edge is chosen as a function of a fixed total order

$o : V \to \mathbb{N}$ on the variables. Edges in the constraint graph are represented as follows:

$$\begin{array}{lll} \mathcal{X} \subseteq \mathcal{Y} & \mathcal{X} \longrightarrow \mathcal{Y} & \text{a successor edge if } o(\mathcal{X}) > o(\mathcal{Y}) \\ \mathcal{X} \subseteq \mathcal{Y} & \mathcal{X} \dashrightarrow \mathcal{Y} & \text{a predecessor edge if } o(\mathcal{X}) < o(\mathcal{Y}) \\ E \subseteq \mathcal{X} & E \dashrightarrow \mathcal{X} & \text{predecessor edge } (E \text{ is a source}) \\ \mathcal{X} \subseteq E & \mathcal{X} \longrightarrow E & \text{successor edge } (E \text{ is a sink}) \end{array}$$

The choice of the order $o(\cdot)$ can have substantial impact on the size of the closed constraint graph and the amount of work required for the closure. Choosing a good order is hard, and we have found that the gen order employed by BANE performs as well or better than other orders we have tried. For example, picking a random order performs about as well as the gen order, but the gen order is of course easier to generate.

The inductive transitive closure rule (ITCR), expressed in terms of predecessor and successor edges, is surprisingly the same as STCR. Transitive constraints are generated between L and R through $\mathcal{X}$, if there is a predecessor edge $L \dashrightarrow \mathcal{X}$ and a successor edge $\mathcal{X} \longrightarrow R$.

$$L \dashrightarrow \mathcal{X} \longrightarrow R \quad \Leftrightarrow \quad L \subseteq R$$

Notice that here L may be a source or a variable—unlike SF, where L is always a source. In IF the closure rule can therefore directly produce transitive edges between variables. (This is not to say that the closure of SF does not produce new edges between variables, but for SF such edges always involve the the application of a structural closure rule.)

Unlike SF, the transitive lower bounds of all variables are not explicit in IF. As was shown in Section 6.2.1, the transitive lower bounds can be computed efficiently for IF. Unless otherwise stated, all execution times reported in the rest of this section include the time to compute the transitive lower bounds of all variables.

The right side of Figure 9.8 shows the initial and final graph for the example constraints using IF. Note that some variable-variable edges in IF are predecessor edges (dotted), whereas all variable-variable edges in SF are successor edges (solid). The ordering on the variables assumed in the example is $o(\mathcal{X}) < o(\mathcal{Z}) < o(\mathcal{Y}_i)$. Note the extra variable-variable edge $\mathcal{X} \dashrightarrow \mathcal{Z}$ added by the closure rule for IF. As a result of this edge, the closure of IF adds edges from $\mathcal{X}$ to all R_i. Each of the variables $\mathcal{Y}_1, \ldots, \mathcal{Y}_l, \mathcal{Z}$ has a single predecessor edge to $\mathcal{X}$, and thus their transitive lower bound is equal to $\mathsf{TLB}(\mathcal{X}) = \{L_1, \ldots, L_k\}$. The total work of closing the graph is $l+m$ edge additions, of which $l-1$ additions are redundant, namely the addition of edge $\mathcal{X} \dashrightarrow \mathcal{Z}$ through all $\mathcal{Y}_i$, plus the work for the km transitive constraints $L_i \subseteq R_j$ (not shown). The work to compute the TLB is proportional to l.

9.4.3 Measurements

Our measurements use the same C benchmark programs shown in Table 9.1. We performed the three experiments shown in Table 9.7 which are analogous to the ones performed for inductive form in Section 9.2. SF-Plain corresponds to classic implementations of set-based analyses and does not perform any cycle elimination. The SF-Oracle experiment is similar to IF-Oracle in that it avoids all unnecessary work induced by cycles in the constraint

Experiment	Description
SF-Plain	Standard form, no cycle elimination
SF-Oracle	Standard form, with full (oracle) cycle elimination
SF-Online	Standard from, using IF online cycle elimination

Table 9.7: Points-to Experiments

	SF-Plain			SF-Oracle		
Benchmark	Edges	Work	Time(s)	Edges	Work	Time(s)
allroots	222	20	0.10	480	20	0.10
diff.diffh	335	41	0.13	690	45	0.17
anagram	365	24	0.15	672	11	0.18
genetic	366	29	0.15	864	26	0.20
ks	1271	1059	0.30	1698	506	0.31
ul	278	54	0.20	1209	53	0.30
ft	1051	496	0.34	1826	153	0.41
compress	387	47	0.28	1627	48	0.45
ratfor	2308	1112	0.69	4561	403	1.16
compiler	1849	293	0.62	3827	131	0.90
assembler	2465	569	1.08	5739	564	1.53
ML-typecheck	16803	129255	2.93	9234	2478	1.77
eqntott	2707	1052	0.96	5668	436	1.45
simulator	30011	266797	5.14	16610	4232	2.49
less-177	40610	279440	6.22	27004	9809	3.65
li	1356287	95149841	1199.05	389792	339648	21.06
flex-2.4.7	12659	5266	5.85	32711	1933	9.67
pmake	276802	9151132	117.40	122188	107489	11.96
make-3.72.1	697032	50125520	606.20	313423	353384	22.08
inform-5.5	260795	8831354	111.25	152261	81206	13.72
tar-1.11.2	270995	6215213	82.10	139649	65246	11.57
sgmls-1.1	1084322	113925061	1346.18	499575	601541	37.67
screen-3.5.2	664085	37987203	461.43	306694	257300	24.55
cvs-1.3	119323	925859	23.21	92380	57411	13.69
espresso	712612	25996610	360.47	387566	331837	29.07
gawk-3.0.3	754367	32595925	439.16	344836	215576	28.20
povray-2.2	2057472	162483247	2148.99	1108347	767024	69.55

Table 9.8: Benchmark data for SF-Plain and SF-Oracle

graph by eliminating cycles before they arise, using the precomputed strongly connected components of the final graph. Finally, SF-Online uses BANE's strategy for cycle detection and elimination applied to constraints in standard form.

Table 9.8 shows the results for the first two experiments. Figure 9.9 plots the analysis time for SF-Plain without cycle elimination against the number of AST nodes of the parsed program. For comparison, the graph also includes the numbers for IF-Plain, *i.e.*, the corresponding experiment for inductive form. Without cycle elimination, SF generally outperforms IF because cycles add many redundant variable-variable edges in IF that lead to redundant work. However, the scaling trend is roughly the same, showing that SF also does not scale well. As is the case for IF, strongly connected components in the constraint graph are a scaling inhibitor for SF as shown by the low numbers for SF-Oracle in Table 9.8.

To validate that our results are not a product of our particular implementation of SF, we compare our measurements against the times of an independent implementation of the same points-to analysis written in C by Shapiro and Horwitz [76]. Their implementation (SH) corresponds to SF without cycle elimination, and we empirically verify that our implementation of SF produces the same trend on our benchmark suite. The scatter plot in Figure 9.10 shows that our implementation of SF without cycle elimination is usually between 2 times faster and 2 times slower than SH (horizontal lines) on a subset of the

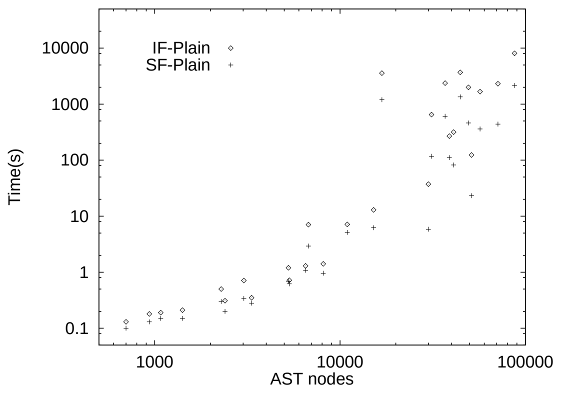

Figure 9.9: SF and IF without cycle elimination

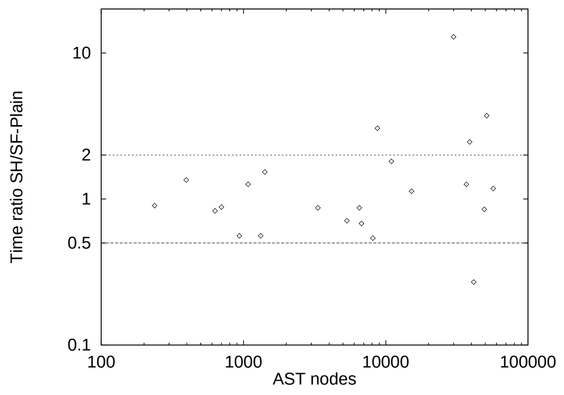

Figure 9.10: Relative execution times of Shapiro and Horwitz's SF implementation of C points-to analysis (SH) over SF-Plain

benchmarks[3] with a few exceptions where our implementation is significantly faster (`flex`, `li`, `cvs`, `inform`), and one program where our implementation is substantially slower (`tar`).

Table 9.9 reports the measurement results for SF-Online. In addition to the information shown for the plain and oracle experiments, the table contains the number of variables that were eliminated through cycle detection and fraction of analysis time spent in cycle detection (%CD).

Figure 9.11 plots the execution time of SF-Online against the number of AST nodes. For comparison, the graph also plots SF-Oracle, IF-Online, and IF-Oracle. The fastest analysis times are achieved by IF-Oracle, followed by SF-Oracle, IF-Online, and SF-Online. We can use the oracle experiments SF-Oracle and IF-Oracle to directly compare the graph representations of IF and SF, independently of cycle elimination. Figure 9.11 shows that IF-Oracle does better than SF-Oracle overall, and thus closing acyclic graphs using the inductive transitive closure rule is more efficient than using the standard transitive closure rule. This observation not only applies to the particular experiment described here. We showed elsewhere [28] that this result can be derived analytically by studying the average case behavior of STCR and ITCR on random acyclic graphs.

Figure 9.12 shows the speedup obtained through online cycle elimination applied to standard form (SF-Online over SF-Plain). For comparison, the graph also contains the speedup of IF-Online over SF-Plain, showing that with cycle elimination, inductive form does better, as predicted by comparing SF-Oracle and IF-Oracle.

The performance benefit of inductive over standard form is illustrated more clearly in Figure 9.13. In this plot, we can see that IF-Online is consistently faster than SF-Online for medium and large-sized programs (at least 10,000 AST nodes).[4] For large programs the difference is significant, with IF-Online outperforming SF-Online by over a factor of 2.5 for the largest program. For very small programs, IF is at most 50% slower than SF, which in

[3]Not all benchmarks ran through SH.

[4]The outlier is the program `flex`; although `flex` is a large program, it contains large initialized arrays. Thus as far as points-to analysis is concerned, it actually behaves like a *small* program.

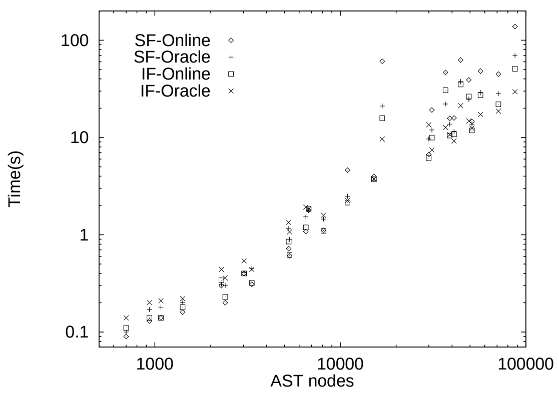

Figure 9.11: Analysis times with cycle detection and oracle

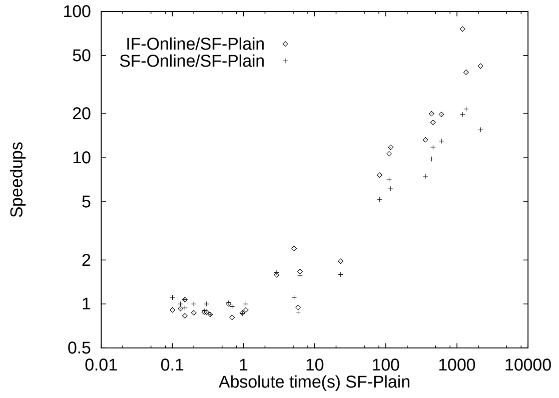

Figure 9.12: Speedups through online cycle detection

		SCC	SF-Online				
Benchmark	AST	#Vars	Elim.	Edges	Work	Time(s)	%CD
allroots	700	20	11	214	316	0.09	11.11%
diff.diffh	935	13	13	319	455	0.13	7.69%
anagram	1078	23	23	333	438	0.14	0.00%
genetic	1412	14	6	366	490	0.16	6.25%
ks	2284	38	24	957	2511	0.30	6.67%
ul	2395	6	6	268	360	0.20	0.00%
ft	3027	67	27	865	1420	0.40	12.50%
compress	3333	26	21	366	474	0.31	0.00%
ratfor	5269	83	35	1913	3521	0.72	5.56%
compiler	5326	35	12	1560	1983	0.61	0.00%
assembler	6516	88	46	2385	3380	1.08	2.78%
ML-typecheck	6752	243	162	8333	26557	1.79	12.29%
eqntott	8117	173	93	2175	4215	1.11	5.41%
simulator	10946	230	108	19142	112887	4.61	22.13%
less-177	15179	274	150	27998	69334	3.99	10.03%
li	16828	1256	872	635866	2348798	60.85	9.22%
flex-2.4.7	29960	315	177	10859	15621	6.68	7.19%
pmake	31148	661	446	144126	443113	19.16	15.08%
make-3.72.1	36892	858	642	329665	1357871	46.67	19.78%
inform-5.5	38874	468	343	99451	306814	15.74	19.70%
tar-1.11.2	41035	822	666	126487	348520	15.88	11.90%
sgmls-1.1	44533	863	744	368084	2198616	62.53	15.70%
screen-3.5.2	49292	864	737	282551	864976	39.07	17.12%
cvs-1.3	51223	724	554	78151	190880	14.59	11.45%
espresso	56938	1374	834	428677	1066940	48.22	20.01%
gawk-3.0.3	71140	1290	910	365473	1072313	44.81	15.09%
povray-2.2	87391	1456	1024	1182741	4557948	138.39	16.16%

Table 9.9: Benchmark data for SF-Online

absolute times means only fractions of seconds.

We can explain the performance difference of IF and SF by comparing the fraction of variables on cycles found by IF-Online and SF-Online (Figure 9.14). Throughout, SF finds only about half as many variables on cycles as IF, and the remaining cycles slow down SF. One reason for this difference is that for SF, the cycle detection only searches successor chains. The analog to predecessor chains in SF are increasing chains. Searching increasing chains in SF results in a higher detection rate (57%), but the much higher cost outweighs any benefits.

Projection merging hurts standard form, since every extra variable will induce extra source edges and extra redundant work. Furthermore, since sinks are not propagated

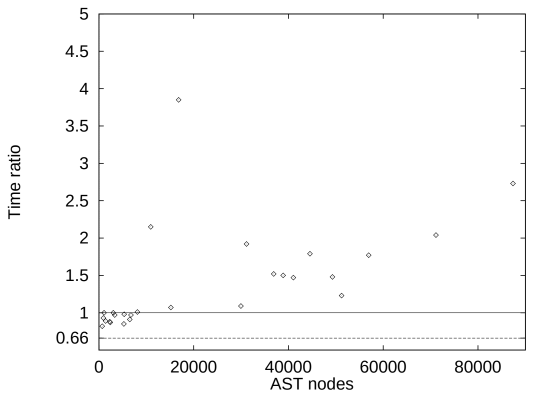

Figure 9.13: Speedups through inductive form

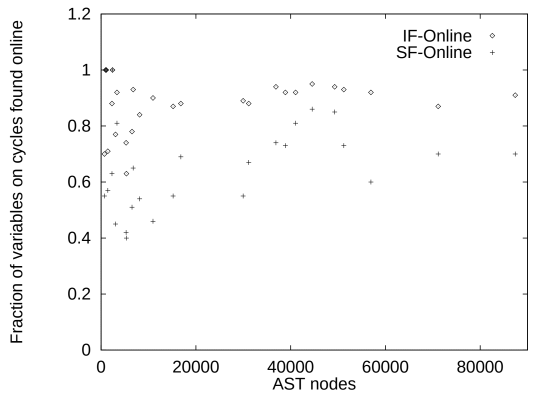

Figure 9.14: Fraction of variables on cycles found online

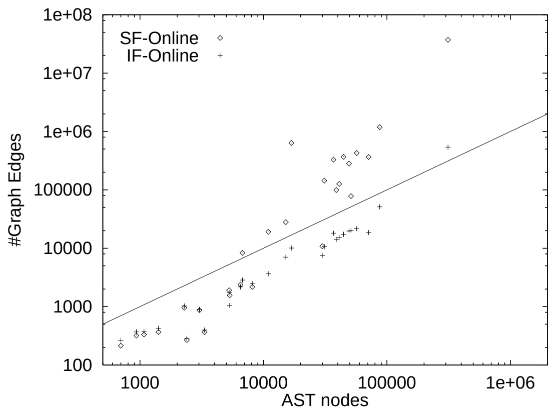

Figure 9.15: Final graph sizes of SF and IF

backwards, projection merging isn't needed.

9.4.4 TLB on demand

One clear advantage of inductive form over standard form is space usage during the resolution. At each step in the resolution process, the constraint graph for standard form contains the transitive lower bounds of each variable explicitly. Since the total size of the TLBs becomes fairly large for large constraint problems, the space usage during resolution under SF is a scaling inhibitor. Figure 9.15 plots the final graph size of IF and SF vs. the benchmark size. The diagonal line plots $y = x$, showing that for IF, the final graph size is smaller than the number of AST nodes, with the exception of `mume` where the graph size is about a third larger. The graph sizes under standard form however exceed the number of AST nodes by orders of magnitude for medium to large programs. In fact, the largest two benchmarks (gs and gcc) did not run to completion with 2GB of main memory under SF.

The transitive lower bound (TLB) under IF is computed using Algorithm 6.7 which

		IF-Online						
Benchmark	AST	Edges	QTLBsize	FTLBsize	BTime(s)	QTime(s)	FTime(s)	Speedup
allroots	700	265	16	34	0.10	0.01	0.00	0.91
diff.diffh	935	369	60	77	0.13	0.01	0.01	1.00
anagram	1078	372	48	89	0.14	0.00	0.01	1.07
genetic	1412	424	52	95	0.18	0.00	0.01	1.06
ks	2284	1027	171	448	0.33	0.01	0.02	1.03
ul	2395	286	24	28	0.23	0.00	0.00	1.00
ft	3027	897	158	215	0.40	0.00	0.01	1.02
compress	3333	397	33	44	0.31	0.01	0.01	1.00
ratfor	5269	1758	610	829	0.84	0.01	0.03	1.02
compiler	5326	1047	398	679	0.61	0.01	0.02	1.02
assembler	6516	2174	681	972	1.16	0.03	0.04	1.01
ML-typecheck	6752	2858	3092	3833	1.73	0.12	0.10	0.99
eqntott	8117	2496	316	605	1.08	0.02	0.05	1.03
simulator	10946	3652	8600	12448	2.03	0.11	0.20	1.04
less-177	15179	7040	16278	19933	3.62	0.10	0.30	1.05
li	16828	10100	453422	538661	15.18	0.62	5.01	1.28
flex-2.4.7	29960	7531	3858	6545	6.03	0.11	0.32	1.03
pmake	31148	10629	90576	118190	9.68	0.29	1.00	1.07
make-3.72.1	36892	18123	159226	273020	30.07	0.59	4.12	1.12
inform-5.5	38874	14139	31044	85711	10.08	0.39	0.78	1.04
tar-1.11.2	41035	15366	64543	108111	10.29	0.50	0.85	1.03
sgmls-1.1	44533	17358	219934	329640	34.28	0.75	4.69	1.11
screen-3.5.2	49292	19707	150763	248691	25.84	0.58	4.69	1.16
cvs-1.3	51223	20289	40274	62274	11.61	0.21	0.90	1.06
espresso	56938	21671	197535	351298	25.19	2.00	3.47	1.05
gawk-3.0.3	71140	18613	152493	306369	21.36	0.57	3.42	1.13
povray-2.2	87391	51216	624500	986988	48.14	2.62	13.07	1.21
mume	312458	539536	17897718	31474473	(113.16) 1800.66	(56.91) 61.89	938.07	(6.08) 1.47
gs	504724	43583903	52728554	–	51360.90	1014.02	∞	∞
gcc	1168907	659584	87997663	–	11485.26	272.06	∞	∞

Table 9.10: Benchmark data for on demand TLB

reduces to computing the transitive closure on an acyclic graph. Redundant work due to transitive edges can be avoided during this computation, whereas during closure under SF, the same redundant work cannot be avoided. Algorithm 6.7 can also be used for on-demand computation of the TLB by computing only the TLBs of variables reachable from the variables of interest. As a result, the full TLBs of the entire graph need never be explicitly represented in memory for IF and as little of the TLB can be computed as necessary. This last point is of interest in that for most analyses, the TLB is needed for only a fraction of all variables present in the constraints. In Points-to analysis for example, the points-to relation is mainly of interest at dereference expressions in the program. Our experiments actually compute the TLB for every program variable. To quantify the benefits of computing the TLB on demand under IF we also computed the full TLB explicitly in memory for each benchmark under the IF-Online experiment, *i.e.*, using cycle elimination, but not projection merging. Table 9.10 contains for each benchmark the size of the constraint graph under IF, the size of the TLB computed on demand (QTLBsize), the size of the full TLB (FTLBsize), the time to compute the closed graph before computing any TLBs (BTime), the time to compute the query-based TLB (QTime), and the time to compute the full TLB (FTime). The last column shows the speedup obtained by using the query-based TLB. The full TLB is roughly twice the size of query-based TLB for many benchmarks under this experiment. Computing the query-based TLB however is often substantially cheaper, indicating that the total size may not be a good indicator for performance. Computing the TLB involves merging sets and eliminating duplicates, which induces costs not apparent in the final TLB size. Even though computing the query-based TLB is roughly five times cheaper for most benchmarks, the overall speedups obtained through the query-based TLB for IF-Online are

Implementation	Description
Term-Set	Exception inference using Term-expressions for the type structure and Set-expressions for exception annotations. Based on equality constraints between Term-expressions.
FlowTerm-Set	Exception inference using FlowTerm-expressions for the type structure and Set-expressions for exception annotations. Based on inclusion constraints.
Set-Set	Exception inference using Set-expressions for both the type structure and the exception annotations. Based on inclusion constraints.

Table 9.11: Precision-efficiency variations for exception analysis

moderate, ranging from 1 to 1.5. The time to close the graph dominates the time to compute the TLBs.

The situation changes when we look at the execution times for closing the graph using projection merging. Projection merging reduces the time to compute the closed graph substantially, but has no direct influence on the time to compute the TLB. Thus, the speedup obtained with the query-based TLB computation is more important when using projection merging. The numbers in parentheses for the mume benchmark show the times and speedup obtained w.r.t. IF-PM. Note that with projection merging, the time to close the graph and compute the query-based TLB for mume is less than $\frac{1}{5}$ that of computing the full TLB alone on the final graph. Since standard form computes the full TLB in all cases, it can not compete with inductive form and query-based TLB on large constraint problems.

9.5 Precision-Efficiency Tradeoffs

This section evaluates BANE with respect to the precision-efficiency tradeoffs provided through mixed constraints. We implemented three versions of the exception inference system described in Section 8.2 using different sorts for the inference of the type structure. Table 9.11 describes the three variations. In theory, the precision increases from the Term-Set to the FlowTerm-Set systems. The precision difference between these systems stems from the use of equality constraints vs. the use of inclusion constraints between type expressions. Inclusion constraints model the direction of the flow of values through a program, whereas equalities model the flow of values undirectionally. As a result, the use of equality constraints may lead to *back-flow* of information. As an example of an expression where the two systems differ, consider the following program fragment:

```
let g = fn x => fn y => fn f => (case ... of .. => f x | .. => f y, x)
in
  ...
```

Here g is a function of three curried arguments x, y, and f. The result of applying g is a pair $\langle v_1, v_2 \rangle$, where v_1 is the result of applying f to either x or y, and v_2 is identical to argument x. An analysis based on equality constraints (Term-Set) equates the types of arguments `x` and `y` with the domain of function argument `f`. As a result, the type of the second component of the return value (v_2) is equal to the type of both `x` and `y`, and it appears as if the second argument `y` could be returned by `g`. Thus the type of `g` obtained by Term-Set is

$$\mathsf{g} : \mathcal{T}_1 \to \mathcal{T}_1 \to (\mathcal{T}_1 \xrightarrow{\mathcal{E}} \mathcal{T}_2) \xrightarrow{\mathcal{E}} \langle \mathcal{T}_2, \mathcal{T}_1 \rangle$$

On the other hand, consider the type of `g` inferred by the FlowTerm-Set system.

$$\mathsf{g} : \mathcal{T}_1 \to \mathcal{T}_2 \to (\mathcal{T}_3 \xrightarrow{\mathcal{E}} \mathcal{T}_4) \xrightarrow{\mathcal{E}} \langle \mathcal{T}_4, \mathcal{T}_1 \rangle \backslash \mathcal{T}_1 \subseteq_{\mathsf{ft}} \mathcal{T}_3 \ \wedge \ \mathcal{T}_2 \subseteq_{\mathsf{ft}} \mathcal{T}_3$$

Here the types of arguments `x` and `y` are not equated. The fact that `f` is potentially applied to `x` and `y` is reflected in the constraints $\mathcal{T}_1 \subseteq_{\mathsf{ft}} \mathcal{T}_3$ and $\mathcal{T}_2 \subseteq_{\mathsf{ft}} \mathcal{T}_3$ expressing that either argument flows into the domain of function `f`. The result type of `g` states precisely that the second component is of type $\mathcal{T}_1$, the type of `x`.

The effect of the different types for `g` is best observed by considering applying `g` to `u`, `v`, and `w` with the following types:

$$\begin{aligned} &\mathtt{u} : \mathsf{exn}(\mathcal{E}_1) \backslash \mathsf{Subscript} \subseteq_{\mathsf{s}} \mathcal{E}_1 \\ &\mathtt{v} : \mathsf{exn}(\mathcal{E}_2) \backslash \mathsf{Fail} \subseteq_{\mathsf{s}} \mathcal{E}_2 \\ &\mathtt{w} : \mathcal{T} \xrightarrow{0} \mathcal{T} \end{aligned}$$

In words, `u` is an exception value carrying at least exception Subscript, `v` is an exception value carrying at least exception Fail, and `w` is function, returning a value of the same type as its argument. Under Term-Set, the result type of the application is $\langle \mathcal{T}, \mathcal{T} \rangle$ with the constraints

$$\mathcal{T} = \mathsf{exn}(\mathcal{E}) \ \wedge \ \mathsf{Subscript} \subseteq_{\mathsf{s}} \mathcal{E} \ \wedge \ \mathsf{Fail} \subseteq_{\mathsf{s}} \mathcal{E}$$

If we extract the exception in the second position of the result and raise it we obtain the potential exception Fail, even though the application could never result in the Fail exception.

Under FlowTerm-Set, the result type of the above application still correctly models the possible value flows in the program. We obtain the pair type $\langle \mathcal{T}_4, \mathcal{T}_1 \rangle$ with the constraints

$$\begin{array}{lll} \mathcal{T}_4 = \mathsf{exn}(\mathcal{E}_3) & \mathsf{Subscript} \subseteq_{\mathsf{s}} \mathcal{E}_3 & \mathsf{Fail} \subseteq_{\mathsf{s}} \mathcal{E}_3 \\ \mathcal{T}_1 = \mathsf{exn}(\mathcal{E}_4) & \mathsf{Subscript} \subseteq_{\mathsf{s}} \mathcal{E}_4 & \end{array}$$

As the example suggests, these differences only show up in higher order programs.

Even though in theory there is a precision difference between the FlowTerm-Set and the Set-Set systems, our formulation of exception inference for ML does not expose it. The Set-Set system can in principle deal with types of the form $\mathsf{bool} \cup \mathsf{int}$, *i.e.*, a union of an integer and a boolean, whereas the FlowTerm-Set system would have to express this type as $\top$. No valid ML programs contain such types however, and thus the difference does not show up. Every ML type is formed by applying a single head constructor, and the FlowTerm sort can express such types accurately. It should be noted here that the use of a sort like FlowTerm or Term requires the analysis to be as polymorphic as the ML type system, for

Benchmark	Lines ML	Description
kb	630	An implementation of the Knuth-Bendix completion algorithm.
lex	1320	The Standard ML/NJ lexer generator.
yacc	2978	The Standard ML/NJ parser generator.
burg	8320	The Standard ML/NJ tree match generator.
pta	30226	The complete Points-to implementation including the BANE library.

Table 9.12: ML Benchmarks for exception analysis

Experiment	Description
Base	Exception inference using cycle elimination, projection merging, and min-max simplification of polymorphic constrained expressions.
NoCycle	Same as Base, but no cycle elimination or projection merging.
NoSimp	Same as Base, but no min-max simplification.

Table 9.13: Experiments for exception analysis

otherwise, the single head constructor assumption no longer holds. The Set-Set system on the other hand can analyze ML programs using less polymorphic types than are needed to infer the ML types. We have not explored this aspect further.

On the efficiency side, constraint resolution for the Term-Set system should be cheaper than for the FlowTerm-Set system through the use of equality constraints and the more compact representation that use enables. Similarly, resolution of constraints in the FlowTerm-Set system should be cheaper than for the Set-Set system, since relations between types containing no exception annotations can be represented by equalities in the FlowTerm system. For example, suppose we have the constraints

$$\mathcal{T}_1 \subseteq_{\mathsf{ft}} \mathcal{T}_2 \qquad\qquad \mathcal{T}_2 \subseteq_{\mathsf{ft}} \mathcal{T}_3 \qquad\qquad \mathcal{T}_2 \subseteq_{\mathsf{ft}} \mathcal{T}_4$$

and we add the constraint $\mathsf{bool} \subseteq_{\mathsf{ft}} \mathcal{T}_1$. This constraint forces $\mathcal{T}_1..\mathcal{T}_4$ to be equal to bool in our implementation, which can be compactly represented. On the other hand, the same constraints under the Set sort cannot be simplified.

We measured the actual precision and efficiency of the three exception systems on the five benchmarks shown in Table 9.12. The benchmarks span two orders of magnitude in terms of size, ranging from 600 lines to 30000 lines. The number of lines is the number of non-blank, non-comment lines after functor applications have been expanded. For each system and each benchmark, we performed the experiments shown in Table 9.13. The Base experiment uses all optimizations present in BANE. The NoCycle experiment evaluates the efficiency difference of the three systems when no cycle elimination is performed, and the last experiment—NoSimp—evaluates the efficiency difference of the three systems when polymorphic constrained expressions are not simplified using the min-max approach of Section 7.6.3.

	Exception Counts						Analysis Time (s)			Avg. # quantified vars		
Name	LTP	LTE	LTL	MFP	MFE	MFL	Base	NoCycle	NoSimp	Base	NoCycle	NoSimp
Term-Set												
kb	0	0	0	1	1	1	2.04	1.93	2.91	2.36	2.73	9.33
lex	0	0	0	16	7	10	9.12	9.03	18.02	21.05	21.49	52.71
burg	5	4	5	48	17	32	11.15	13.44	11.40	6.02	6.17	8.25
yacc	6	4	6	36	21	36	16.21	15.76	16.53	2.16	2.36	3.16
pta	22	11	22	261	41	180	7894.47	∞	∞	23.58	–	–
FlowTerm-Set												
kb	0	0	0	1	1	1	2.10	2.01	5.60	1.27	1.26	20.35
lex	0	0	0	16	7	10	3.05	2.91	10.43	0.35	0.35	28.88
burg	5	4	5	48	17	32	12.39	12.52	25.61	2.72	2.94	26.09
yacc	6	4	6	36	21	36	24.36	23.90	41.26	1.55	1.63	15.11
pta	22	11	22	261	41	180	198.30	∞	∞	2.10	–	–
Set-Set												
kb	0	0	0	1	1	1	2.15	2.42	5.14	1.29	1.37	14.96
lex	0	0	0	16	7	10	2.89	4.22	8.77	0.55	0.71	13.51
burg	5	4	5	48	17	32	9.44	11.64	20.36	1.83	2.21	12.05
yacc	6	4	6	36	21	36	27.12	39.56	49.00	1.56	2.04	11.59
pta	22	11	22	261	41	180	319.37	∞	∞	2.75	–	–

Table 9.14: Benchmark data of all implementations and experiments

Table 9.14 contains the results of our experiments. We measured the relative precision of the exception inference systems in terms of the number of reported uncaught exception-location pairs for the main function of each program (MFP). Recall from Section 8.2 that we infer sets of uncaught exception-location pairs $e@l$, where e is an exception, and l a program point. The pair expresses that the uncaught exception e may originate from location l. It is thus possible to infer two pairs $e@l_1$, and $e@l_2$ involving the same exception e, but two different locations l_1 and l_2. To clarify the differences in the precision of just the set of exceptions e and locations l, we also give the number of distinct exceptions (MFE) and locations (MFL) appearing in all exception-location pairs.

We also report the number of uncaught load-time exceptions, *i.e.*, exceptions that are potentially raised when loading and evaluating initialization code, but before calling the main function of the program (LTP, LTE, and LTL). The execution times reported are in seconds and are best out of three runs, using one processor of a lightly loaded 8 processor Enterprise-5000 machine with 2GB of main memory. Entries marked with ∞ ran out of space or did not run to completion within 24 hours.

The last three columns report the average number of quantified variables per polymorphic constrained expression inferred for each experiment. As we will see, the size of the quantified types has a strong influence on execution time.

As is apparent from the exception counts, the precision loss of the Term-Set implementation does not manifest itself. All implementations report exactly the same uncaught exceptions for the main function and the load-time exceptions. A likely explanation for this fact is that the absence of subtyping in Term-Set is counteracted by let-polymorphism. On the efficiency side there are even more surprises. The expected efficiency grade from Term-Set to FlowTerm-Set only shows up for the three benchmarks `kb`, `burg`, and `yacc`, although the running times for `kb` are essentially the same. The most surprising result is that for the largest benchmark, `pta`, the Term-Set system performs extremely poorly. We will study the reason for this anomaly below. The efficiency grade between FlowTerm-Set and Set-Set shows up only for the two largest benchmarks. For smaller benchmarks, the overhead introduced by fresh variables arising during the resolution of FlowTerm constraints may annul any advantages of the potential savings provided by the more compact representation of

FlowTerm constraints.

If we look at the average number of quantified variables for the Base experiments, we notice that the averages for the Term-Set system are up to ten times larger than for the other two systems, indicating that the efficiency advantage of the Term-Set system did not show up due to large polymorphic constrained expressions. This fact is supported by the NoSimp experiment where the expected efficiency grade is more apparent. The purpose of the min-max simplification is to reduce the size of polymorphic constrained expressions so as to avoid overhead when the expression is instantiated multiple times. Except for `lex` and `pta`, the execution times for the Term-Set system without simplification are barely different from the Base experiment, whereas for the other systems, execution times double without simplification. These facts suggest that min-max simplification in the Term-Set systems does not work well. Clearly, type expressions of sort Term cannot be minimized or maximized, since we assert equality constraints between type expressions in the Term-Set system. However, the bounds on Set-variables appearing in the constraints can be minimized or maximized, except for variables appearing inside Term-constructors (Section 7.6.3). Examining the types and constraints arising during the inference of `pta` with the Term-Set system, we notice that many constrained types contain a large number of constraints of the form:

$$\begin{aligned} \mathsf{IO}(\mathsf{exn}(\mathcal{E}_1))@l_1 &\subseteq_{\mathsf{s}} \mathcal{E} \\ \mathsf{IO}(\mathsf{exn}(\mathcal{E}_2))@l_2 &\subseteq_{\mathsf{s}} \mathcal{E} \\ \mathsf{IO}(\mathsf{exn}(\mathcal{E}_3))@l_3 &\subseteq_{\mathsf{s}} \mathcal{E} \\ &\vdots \end{aligned}$$

where the IO exception constructor is used by the input-output subsystem of SML and signals that an exception was caught during an input-output operation. The exception that was caught is provided to any handler as an argument carried by the IO-constructor. The distinct Set-variables $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots$ all have similar if not the same bounds, and the locations $l_1, l_2, \ldots$ are also mostly the same. These constraints on $\mathcal{E}$ cannot be minimized by the min-max simplification, since the variables $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots$ appear inside the Term-constructor exn and future constraints can result in equality constraints on these variables. Note that the FlowTerm-Set and Set-Set systems can in this same situation minimize the variables $\mathcal{E}_1, \mathcal{E}_2, \ldots$ which results in many equivalent lower-bounds on $\mathcal{E}$ (since the variable $\mathcal{E}_i$ have similar lower bounds). As a result, large numbers of such bounds do not appear in these systems.

The NoCycle experiments show that cycle elimination introduces some overhead for the smaller benchmarks, but is absolutely necessary for the largest benchmark `pta`. This largest benchmark exercises the higher order nature and the imperative features of ML more fully than the other benchmarks and presents a realistic real world analysis problem. The scaling behavior of the three systems should thus be judged mainly by this benchmark.

A first conclusion from the above experiments is that the expected precision-efficiency tradeoffs do not necessarily arise in practice. Unexpected interactions (in this case between Term constructors, and simplification) may annul the efficiency advantage of certain sorts. On the other hand, the precision of a cheaper system need not be worse in

	Exception Counts						Base Experiment	
Name	LTP	LTE	LTL	MFP	MFE	MFL	Time(s)	avg.QV
kb	0	0	0	1	1	1	1.41	2.36
lex	0	0	0	59	7	12	3.93	3.85
burg	5	4	6	415	17	34	7.94	2.89
yacc	6	4	6	493	21	36	15.65	1.91
pta	54	11	22	3541	41	198	127.09	2.50

Table 9.15: Benchmark data for Term-Set Base experiment with Cartesian-closed constructors.

practice. Being able to express these different systems in a common framework as provided by BANE allowed us to identify why one system performs better than another. As a result, we added another precision dial to BANE to deal with the performance problem of the Term-Set system described above. The precision dial allows Set-constructors to be marked as *Cartesian-closed.* This option tells BANE that if $c : s\ t \to \mathsf{s}$ is a Cartesian-closed constructor, then the union of two constructor expressions $c(E_1, E_2) \cup c(E_3, E_4)$ is to be treated as equivalent to $c(\mathcal{T}_1, \mathcal{T}_2)$ along with the constraints

$$E_1 \subseteq_s \mathcal{T}_1 \qquad E_2 \subseteq_s \mathcal{T}_1 \qquad E_3 \subseteq_t \mathcal{T}_2 \qquad E_2 \subseteq_t \mathcal{T}_2$$

where $\mathcal{T}_1$ is a fresh temporary variable of sort s, $\mathcal{T}_2$ is a fresh temporary variable of sort t. In other words, the union of two constructor expressions with the same cartesian-closed constructor is transformed into a single constructor expression where the arguments are unioned (the example assumes that both arguments to c are covariant). Applying this option to the constructor `@` combining exception names and locations allows us to simplify the large numbers of constraints arising in the Term-Set system. In the case of the IO constructor bounds shown above, the simplification results in equality constraints between $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots$ and we end up with a single lower bound $\mathsf{IO}(\mathsf{exn}(\mathcal{E}_1))@\mathcal{T}_2 \subseteq_{\mathsf{s}} \mathcal{E}$. The optimization however may result in a severe loss of precision since the association between an exception name and the location where it was raised is essentially lost.

Table 9.15 shows the results of the Base experiment for the Term-Set system when Cartesian-closed constructors are used. The last column shows that the number of quantified variables per polymorphic constrained expression inferred has been reduced ten-fold w.r.t. to the earlier Base experiment on Term-Set. The numbers are now of the same order as for the FlowTerm-Set and Set-Set systems. As a result, the execution times have been reduced, in particular for `pta`. The expected efficiency and scaling advantage of the Term-Set system finally shows. However, we have traded the efficiency gain for a loss in precision. The number of exception-location pairs inferred for `pta` jumped from 261 to 3541. Similar experiments for the FlowTerm-Set and Set-Set systems produced execution times that did not differ much from the times reported in Table 9.14 for the Base experiments. This fact is not surprising, since simplification for these systems already produced small polymorphic constrained expressions. Figure 9.16 summarizes the results. It shows the Base analysis times for all systems and benchmarks. The Term-Set-CC system refers to the Term-Set system with Cartesian-closed constructors. No system stands out with a clear performance advantage over the other. The maximum difference is a factor of 2 between Term-Set-CC and Set-Set on the `pta` benchmark. We expect that for even larger benchmarks, the efficiency relations seen on `pta` will prevail.

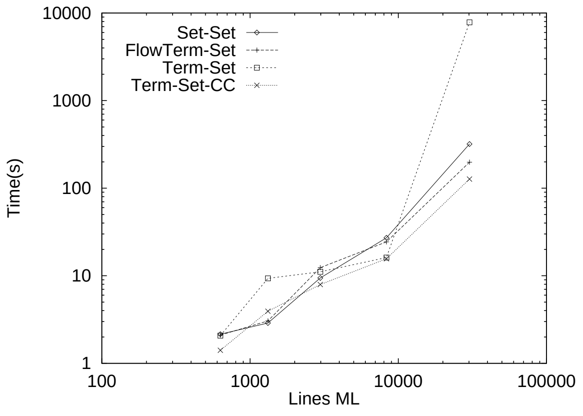

Figure 9.16: Comparison of the Base experiments

The sequence of experiments described in this section have shown that experimentation is crucial to figuring out practical precision-efficiency tradeoffs of program analyses. The expected tradeoffs have not shown-up initially but have uncovered a scaling inhibitor for the Term-Set system.

Chapter 10

Related Work

This chapter discusses related work not covered in Section 4.6 or Section 6.4. The related work is divided into three sections. The first section covers work on practical aspects of constraint resolution implementations of cubic or worse complexity. The second section covers sub-cubic time analyses and resolution techniques. Finally, the third section covers other approaches to program analysis tools and frameworks.

10.1 The Cubic-Time Bottleneck

Most work on program analysis focuses on the soundness and theoretical complexity of the algorithms. Algorithms based on inequalities (in particular inclusion constraints) generally require a form of dynamic transitive closure, causing such algorithms to have at least cubic worst-case time-complexity. Heintze and McAllester [43] show that the problem of determining membership for languages defined by 2-way nondeterministic pushdown automata (2NPDA) is linear time reducible to a standard flow analysis [67]. The best known algorithm for 2NPDA has cubic worst-case time complexity, showing that it is unlikely that sub-cubic time algorithms for standard flow analysis exist.

This so-called cubic-time bottleneck has led many researchers away from algorithms based on dynamic transitive closure and towards cheaper, but potentially less precise techniques. Relatively little work has focused on practical implementation aspects of cubic or worse constraint resolution algorithms, probably because straight-forward implementations do indeed attain the worst-case complexity even for relatively small programs.

The work described in this dissertation shows that through clever representation and constraint graph simplification, the cubic worst case complexity need not be attained in practice. Furthermore, the mixed constraint formalism provides an escape mechanism to use cheaper constraints, such as equality between Term expressions in cases where the practical complexity is indeed high.

Work on set-based analysis [38, 30, 29] and inclusion constraint-based type inference [71, 24, 83, 58, 72] has mostly focused on techniques to simplify constraints to achieve scalability. One exception is Heintze, who describes in his dissertation that hash-consing constructed expressions is important to efficiently test set-memberships, and that sets should be represented as hash tables. Flanagan and Felleisen prove that their particular form of set-constraints have minimal normal forms, but that computing the normal form is PSPACE-complete. They then develop a number of algorithms based on grammar simplifications that achieve good reductions in constraint size without computing the minimal form. They apply their techniques in MrSpidey [31], a static debugger for Scheme using set-based analysis. Their results focus mainly on the reduction of constraint system sizes, showing that the simplifications enable the analysis of medium sized programs (17000 lines of Scheme), whose analysis otherwise exhausts heap space. It would be interesting to combine inductive form and online cycle elimination with some of the more sophisticated algorithms they developed.

Pottier develops heuristics for simplifying and thus reducing the size of constraint systems. Pottier does not report measurements showing the benefit of his approach [72]. We examined similar techniques for simplifying constraint graphs at regular intervals [24]; we showed that the cost-benefit tradeoff of simplifications poses a problem in that frequent simplification is too expensive for some benchmarks, but necessary for others to achieve scalability.

Trifonov and Smith investigate the decidability of entailment for a class of set constraints slightly distinct from those of Flanagan and Felleisen. They only derive an approximation of entailment that could be used to simplify constraint systems, but do not investigate practical aspects. Marlow and Wadler developed a soft-typing system for Erlang [58] based on the set-expressions and set-constraints of Aiken and Wimmers [3]. They also report scaling problems and suggest several ways to simplify constraints that help scaling. The reported analysis times of their system are still rather slow.

Except for our own system, all of the above systems are based on a standard form representation of the constraints and standard transitive closure (so far as we know). As we have shown in Section 9.4, standard form has serious limitations for scaling to large problems, due to the explicit computation of the transitive lower bounds. Simplification techniques may help reduce the size of the transitive lower bound, but cannot in general compute a sparser closure than inductive form.

10.2 Sub-Cubic Time Formalisms

The lack of progress in achieving scalable implementations of algorithms based on dynamic transitive closure has encouraged interest in asymptotically faster algorithms that are either less precise or designed for special cases. An example of the former is Steensgaard's formulation of points-to analysis based on conditional unification [79]. Instead of a points-to set for each program point, his analysis infers points-to equivalence classes. If a pointer is found to point to two distinct classes, the classes are merged. This approach loses precision but results in a nearly linear time algorithm.

Shapiro and Horwitz [76] study points-to analysis w.r.t. the precision–efficiency tradeoff. They contrast an algorithm based on inclusion constraints [8] with the equality based algorithm of Steensgaard [79], and then describe a spectrum of algorithms with sub-cubic time complexity in between. They conclude that while Andersen's analysis is substantially more precise than Steensgaard's, its running time is impractical. However, our implementation of Andersen's points-to analysis is generally competitive with their implementation of Steensgaard's algorithm, suggesting that the precision-efficiency tradeoff needs to be reexamined.

In Section 6.4 we already discussed the the pseudo-linear time flow analysis of Mossin and the similar pseudo-linear time closure-analysis algorithm for functional programs of Heintze and McAllester [64, 42]. These analyses are examples of the second class of analyses, where particular properties of the program under analysis are exploited. In this case the exploited property is the bounded type size.

Defouw, Grove, and Chambers study precision-efficiency tradeoffs in receiver-class analysis of Cecil and Java programs [21]. They show that the receiver class analysis of Palsberg and Schwartzbach (a cubic time algorithm) does not scale well and propose less precise but sub-cubic time algorithms based on unification.

10.3 Program Analysis Frameworks

The desire to reuse substantial programming efforts in program analysis is not new. Frameworks for classic dataflow analysis (DFA) are numerous. These frameworks are specialized for producing dataflow analyses in the context of an optimizing compiler.

Kildall [50] characterizes a broad class of "global" (single procedure) dataflow analyses (DFA) as iterative fix-points in meet semi-lattices. Based on this formulation, he built a tool to automatically generate DFA-algorithms from specifications. According to Aho et al. [1] tools for automatically generated DFA-algorithms have not caught on because the amount of development time saved was not significant.

Venkatesh [85] and Yi et. al. [92] describe generic frameworks based on abstract interpretation [17]. Both frameworks allow semantic program analyses to be described in a high level language, which is then translated into an implementation language and linked with support libraries. Venkatesh gives no performance numbers, while Yi et. al. report numbers for inter-procedural alias analysis and constant propagation of 4–100 minutes for programs in the range of several thousand lines of C. An interesting aspect of the framework described by Yi et. al. is the idea of a projection, which can be used to coarsen abstract domains and thus trade precision for efficiency.

Dawson et al. [20] show the practicality of describing and implementing program analyses as logic programs. An analysis is defined by a translation from the source language to a Prolog program, where the evaluation of the Prolog program yields the analysis result. They illustrate their approach with a groundness analysis for logic programs and a strictness analysis for lazy functional programs. They claim their approach is also practical for classic DFA of imperative programs. Unfortunately, the order of horn clauses in the generated program has a direct influence on the efficiency of the analysis. They report reasonable strictness analysis times of 3 seconds for a 595 line functional program. However, they do not describe the scaling behavior. This approach corresponds essentially to solving equations between terms, and can be used e.g. for Hindley-Milner type inference. The authors refer to constraint logic programming (CLP) as an avenue to generalize the approach.

Attribute grammar frameworks provide a clean formalism to express semantic analyses of programs. A variety of approaches have been explored to overcome the circularity restriction (Paakki gives a comprehensive survey [66]). For example, logic attributes can be used to express Hindley-Milner type inference. Another approach is to compute iterative fix-points akin to abstract interpretation.

Chapter 11

Conclusions

Expressing a program analysis as a constraint problem provides a clean separation between the analysis specification (constraint generation) and its implementation (constraint resolution).

The first part of this dissertation developed the formalism of mixed constraints. Mixed constraints combine different kinds of constraint formalisms into a coherent new formalism. Mixed constraints give the program analysis designer more control over the precision-efficiency tradeoffs of an analysis. The second part of the dissertation presents and empirically evaluates an implementation of the mixed constraint formalism called BANE.

The clean separation of program analysis specification and implementation allows tuning the constraint representation and resolution and enables powerful optimizations that can be implemented completely independently from any particular program analyses. Furthermore, these optimizations and representation choices benefit any future analyses written using the same library.

We have demonstrated, for the first time, that program analyses based on set constraints are very practical, even for large programs. As our benchmark, we used a Points-to analysis for the C programming language, which computes an approximation of the memory graph and the global control-flow of a program. Keeping the constraint-graph size under control is crucial. The improved scaling results from a combination of three novel techniques:

- A non-standard constraint-graph representation based on inductive constraints,
- Online detection and elimination of cyclic constraints, and
- Merging of projection constraints.

All three techniques help reduce the number of edges, even at the cost of introducing new nodes (in the case of projection merging).

Some scaling obstacles appear only with very large constraint problems. Cyclic constraints are a problem already on medium sized graphs, whereas our inductive graph representation becomes important at much larger scales. Projection merging also is not essential up to very large constraint problems. Our measurements show that inferring the scaling behavior of an analysis implementation from a few small benchmarks is very misleading. Unfortunately, such inferences are still very common in the literature.

We compared our constraint graph representation to the standard graph representation commonly used in implementations of Set-Based Analysis (SBA). We showed that SBA implementations also benefit from cycle elimination, but that for very large constraint problems the standard graph representation requires orders of magnitude more space in practice than inductive constraint graphs for the same problem.

To evaluate the precision-efficiency choices provided by mixed constraints, we implemented and compared three versions of an exception inference system for Standard ML. We demonstrated the importance of empirical experimentation in determining the practicality of an analysis, by showing that expected efficiency advantages of less precise, but faster constraint formalisms may not manifest in a naive implementation.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1988.

[2] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science (LICS'92)*, pages 329–340. IEEE Computer Society Press, June 1992.

[3] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[4] A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Conference Record of the 21st Annual ACM SSymposium on Principles of Programming Languages*, pages 163–173, January 1994.

[5] Alexander Aiken, Dexter Kozen, Moshe Vardi, and Ed Wimmers. The complexity of set constraints. In *Computer Science Logic '93*, volume 832 of *Lecture Notes in Computer Science*, pages 1–17. Springer Verlag, September 1993.

[6] Alexander Aiken, Dexter Kozen, and Ed Wimmers. Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122(1):30–44, 1995.

[7] Alexander Aiken, Ed Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Science*, pages 47–76. Springer Verlag, September 1997.

[8] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[9] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[10] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Set constraints are the monadic class. In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (LICS'93)*, pages 75–83. IEEE Computer Society Press, June 1993.

[11] Kim B. Bruce and Giuseppe Longo. A modest model of records. In C. A. Gunter and J. C. Mitchell, editors, *Thoretical Aspects of Object-Oriented Programming*, Foundation of Computing, chapter 6. MIT Press, 2nd edition, 1994.

[12] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–174, 1988.

[13] Witold Charatonik and Leszek Pacholski. Negative set constraints with equality. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (LICS'94)*, pages 128–136. IEEE Computer Society Press, July 1994.

[14] Witold Charatonik and Leszek Pacholski. Set constraints with projections are in NEXPTIME. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 642–655, Los Alamitos, CA, USA, November 1994. IEEE Computer Society Press.

[15] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 25:6 in SIGPLAN notices, pages 296–310, June 1990.

[16] Patrick Cousot. Types as abstract interpretations. In *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, January 1997.

[17] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by contruction or approximation of fixed points. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[18] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the 1995 Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, June 1995.

[19] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Science*, pages 687–706. Springer Verlag, April 1994.

[20] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—A case study. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 31:5 in SIGPLAN notices, pages 117–126, May 1996.

[21] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Conference Record of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, January 1998.

[22] Alain Deutsch. On the complexity of escape analysis. In *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 358–371, January 1997.

[23] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 29:6 in SIGPLAN notices, pages 242–256, June 1994.

[24] Manuel Fähndrich and Alex Aiken. Making set-constraint based program analyses scale. In *First Workshop on Set Constraints at CP'96*, Cambridge, MA, August 1996. Available as Technical Report CSD-TR-96-917, University of California at Berkeley.

[25] Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 114–126. Springer Verlag, September 1997.

[26] Manuel Fähndrich and Alexander Aiken. Refined type inference for ML. In *Proceedings of the 1st Workshop on Types in Compilation*, 1997.

[27] Manuel Fähndrich, Jeffrey S. Foster, Alexander Aiken, and Jason Cu. Tracking down exceptions in Standard ML programs. Technical Report UCB//CSD-96-996, University of California, Berkeley, 1998.

[28] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 33:5 in SIGPLAN notices, pages 85–96, June 1998.

[29] Cormac Flanagan. *Componential Set-Based Analysis.* PhD thesis, Rice University, 1997.

[30] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 32:6 in SIGPLAN notices, pages 235–248, June 1997.

[31] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 31:5 in SIGPLAN notices, pages 23–32, May 1996.

[32] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *Proceedings of the 1988 European Symposium on Programming*, pages 94–114, 1988.

[33] Rémi Gilleron, Sophie Tison, and Marc Tommasi. Solving systems of set constraints using tree automata. In *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 505–514, 1992.

[34] Rémi Gilleron, Sophie Tison, and Marc Tommasi. Solving systems of set constraints with negated subset relationships. In *Foundations of Computer Science*, pages 372–380, November 1993.

[35] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, chapter 10, pages 199–200. Addison Wesley, 1996.

[36] Carl A. Gunter and Dana S. Scott. *Semantic Domains*, chapter 12, pages 633–674. Elsevier & MIT Press, 1992. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics.*

[37] Juan Carlos Guzmán and Ascánder Suárez. An extended type system for exceptions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 127–135, June 1994.

[38] Nevin Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.

[39] Nevin Heintze. Set based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 306–17, June 1994.

[40] Nevin Heintze and Joxan Jaffar. A decision procedure for a class of Herbrand set constraints. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 42–51. IEEE Computer Society Press, June 1990.

[41] Nevin Heintze and Joxan Jaffar. A decision procedure for a class of set constraints (extended abstract). In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 42–51. IEEE Computer Society Press, June 1990.

[42] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 32:6 in SIGPLAN notices, pages 261–272, June 1997.

[43] Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, pages 342–351. IEEE Computer Society Press, June 1997.

[44] Laurie J. Hendren, Joseph Hummel, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis of imperative programs. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 27:7 in SIGPLAN notices, pages 249–260, June 1992.

[45] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *5th ACM Conference Proceedings on Functional Programming Languages and Computer Architecture*, pages 448–72, 1991.

[46] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 205–215, June 1992.

[47] Lalita Jategaonkar and John Mitchell. ML with extended pattern matching and subtypes. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 198–211, July 1988.

[48] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[49] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

[50] Gary Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

[51] Dexter Kozen. Logical aspects of set constraints. In *CSL: 7th Workshop on Computer Science Logic*, pages 175–188. Springer Verlag, 1993.

[52] Dexter Kozen. Set constraints and logic programming. *Information and Computation*, 142(1), 1998.

[53] William Landi and Barbara G. Ryder. Safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 27:7 in SIGPLAN notices, pages 235–248, June 1992.

[54] John M. Lucassen. *Types and Effects—Towards the Integration of Functional and Imperative Programming.* Ph.D. thesis, MIT Laboratory for Computer Science, August 1987.

[55] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, January 1988.

[56] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymophic types. In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 165–174, January 1984.

[57] David B. MacQueen, Gordon D. Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1–2):95–130, October–November 1986.

[58] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proceedings of the International Conference on Functional Programming (ICFP '97)*, number 32:8 in SIGPLAN notices, pages 136–149, June 1997.

[59] David McAllester and Nevin Heintze. On the complexity of set-based analysis. In *Proceedings of the International Conference on Functional Programming (ICFP '97)*, number 32:8 in SIGPLAN notices, pages 150–63, June 1997.

[60] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[61] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[62] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[63] John Mitchell. Coercion and type inference (summary). In *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[64] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[65] Atsushi Ohori. A polymorphic record calculus and its compilation. *Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[66] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–256, June 1995.

[67] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995.

[68] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of the ACM Conference on Object-Oriented programming: Systems, Languages, and Applications*, pages 146–61, October 1991.

[69] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9(1):49–67, 1997.

[70] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 276–290, January 1999.

[71] François Pottier. Simplifying subtyping constraints. In *Proceedings of the SIGPLAN '96 International Conference on Functional Programming (ICFP '96)*, number 31:6 in SIGPLAN notices, pages 122–133, May 1996.

[72] François Pottier. *Type Inference in the Presence of Subtyping: From Theory to Practice.* PhD thesis, Université Paris VII, July 1998.

[73] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76, January 1989.

[74] John C. Reynolds. *Automatic Computation of Data Set Definitions*, pages 456–461. Information Processing 68. North-Holland, 1969.

[75] John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[76] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[77] Oded Shmueli. Dynamic cycle detection. *Information Processing Letters*, 17(4):185–188, 8 November 1983.

[78] Ryan Stansifer. Type inference with subtypes. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 88–97, January 1988.

[79] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[80] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[81] Satish Thatte. Type inference with partial types. In *Automata, Languages and Programming: 15th International Colloquium*, pages 615–629. Springer-Verlag Lecture Notes in Computer Science, vol. 317, July 1988.

[82] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the 21st Annual ACM SSymposium on Principles of Programming Languages*, pages 188–201, January 1994.

[83] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer Verlag, September 1996.

[84] Jan van Leeuwen. *Graph Algorithms*, chapter 10, page 542. Elsevier & MIT Press, 1992. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity.*

[85] G. A. Venkatesh. A framework for construction and evaluation of high-level specifications for program analysis techniques. In *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 24:7 in SIGPLAN notices, pages 1–12, 1989.

[86] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the 2nd Annual IEEE Symposium on Logic in Computer Science (LICS'87)*, pages 37–44. IEEE Computer Society Press, June 1987. Corrigendum, LICS'88, page 132.

[87] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the 3rd Annual IEEE Symposium on Logic in Computer Science (LICS'88)*, page 132. IEEE Computer Society Press, July 1988.

[88] William E. Weihl. Interprocedural data flow analysis in the presence of pointers. In *Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[89] Pierre Weis, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.

[90] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93-200, Rice University, February 1993.

[91] Kwangkeun Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Proceedings of the 1st International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 238–254. Springer Verlag, September 1994.

[92] Kwangkeun Yi and Williams Ludwell Harrison, III. Automatic generation and management of interprocedural program analyses. In *Conference Record of the 20th Annual ACM-SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 246–259, January 1993.

[93] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113. Springer Verlag, September 1997.

Table of Notations

Notation	Explanation
S	the set of sorts
s,t	a sort
c,d	a syntactic constructor
$\mathcal{X}$,$\mathcal{Y}$,$\mathcal{Z}$	a variable of any sort
E	a mixed expression or set-expression
$\mathbf{V}$	a semantic domain
v, w	an element of a semantic domain
f, g	a function value of a semantic domain
I, J, K, X, Y, Z	an ideal
$\mathcal{I}(D)$	the collection of ideals of D
$\mathbf{L}$	the universal set of labels
A, B, C	a finite set of labels
N	a finite or cofinite set of labels
S	a constraint set
Γ	a collection of constraint sets
δ	a domain-complement expression
$\alpha_{\mathcal{X}}$	a domain-complement variable
$\alpha(E)$	the domain-complement of Row-expression E