UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

Relational and Allegorical Semantics for Constraint Logic Programming

PhD Thesis

Emilio Jesús Gallego Arias
Ingeniero en Informática

July 2012

Departamento de Lenguajes y
Sistemas Informáticos e Ingeniería de Software
Facultad de Informática
Universidad Politécnica de Madrid

Relational and Allegorical Semantics for Constraint Logic Programming

presented in partial fulfillment of the degree of
Doctor in Software and Sistemas

Author: **Emilio Jesús Gallego Arias**
Ingeniero en Informática
European Master in Computational Logic
Universidad Politécnica de Madrid

Advisor: **Julio Mariño y Carballo**
Profesor Titular de Universidad (Interino)
Universidad Politécnica de Madrid

Advisor: **James B. Lipton**
Associated Professor of Computer Science
Wesleyan University

Madrid, July 2012

Thesis Committee:

- **Prof. Murdoch J. Gabbay**
  School of Mathematical & Computer Sciences
  Heriot Watt University

- **Prof. Manuel Hermenegildo**
  IMDEA Software Institute **and**
  Facultad de Informática
  Universidad Politécnica de Madrid

- **Prof. Francisco Javier López Fraguas**
  Facultad de Informática
  Universidad Complutense de Madrid

- **Prof. Roger D. Maddux**
  Department of Mathematics
  Iowa State University

- **Prof. Narciso Martí Oliet**
  Facultad de Informática
  Universidad Complutense de Madrid

- **Prof. Fernando Orejas**
  Departament de Llenguatges i Sistemes Informàtics
  Universitat Politècnica de Catalunya

Contents

List of Figures v

Resumen vii

Summary ix

Preface xi
- Declarative Programming xii
- Structure of the Work xiv
- Acknowledgments xiv

Chapter 1. Introduction 1
- 1.1. Relational Methods and Logic Programming 1
- 1.2. Objectives and Motivation 3
- 1.3. The Thesis 4
- 1.4. Contributions 6
- 1.5. Previous and Related Work 6
- 1.6. Notation 8

Chapter 2. Programming Language Theory 11
- 2.1. Theories and Formalisms 12
- 2.2. Operational Semantics 15
- 2.3. Denotational Semantics 15
- 2.4. Algebraic Semantics 17

Chapter 3. Constraint Logic Programming 19
- 3.1. Basic Syntax 19
- 3.2. Constraints 20
- 3.3. Examples 22
- 3.4. Denotational semantics 24
- 3.5. Operational Semantics 25
- 3.6. Impure features 26
- 3.7. Related Work 27

Chapter 4. The Calculus of Binary Relations 29
- 4.1. Relation Syntax 29
- 4.2. The Equational Theory **DRA** 29
- 4.3. Semantics 30
- 4.4. Properties of Binary Relations 30
- 4.5. Related Work 31

Chapter 5. Logic and Relations 33

5.1. The Relational Language R_Σ 33
5.2. The Equational Theory $\mathbf{QRA}_\Sigma$ 34
5.3. Semantics for R_Σ 35
5.4. Logic and Relation Algebra 36

Chapter 6. Relational Semantics for Constraint Logic Programming 39
6.1. Introduction 39
6.2. An Introductory Example 40
6.3. Translation of CLP Programs 43
6.4. Computing with Relations: Operational Semantics 52
6.5. Operational Equivalence 56
6.6. Related Work 69
6.7. Conclusions and Future Work 71

Chapter 7. First-Order Unification Using Variable-Free Relation Algebra 73
7.1. The Unification Problem 73
7.2. Solved forms for **U**-terms 75
7.3. The algorithm 76
7.4. An Example 82
7.5. Related work 84
7.6. Conclusions and Future Work 85

Chapter 8. Category Theory 87
8.1. Basic concepts 87
8.2. Functors 88
8.3. Diagrams 89
8.4. Limits 90
8.5. Lawvere Categories 91
8.6. Regular Categories and Relations 92
8.7. Allegories and Tabular Allegories 94
8.8. Related Work 95

Chapter 9. Allegorical Semantics for Logic Programming 97
9.1. Introduction 97
9.2. Regular Lawvere Categories 99
9.3. Σ-Allegories 101
9.4. Pullbacks in Regular Lawvere Categories 102
9.5. Computing with Diagrams 106
9.6. The Pullback Algorithm 108
9.7. Translation of the Program 113
9.8. The Categorical Machine 117
9.9. Memory Diagrams and Implementation Discussion 123
9.10. Related Work 124
9.11. Conclusions and Future Work 125

Chapter 10. Extensions to Logic Programming in Tabular Allegories 127
10.1. Constraints 127
10.2. Algebraic Data Types 129
10.3. Functions and Functional-Logic Programming 130
10.4. Monadic Constructions 133
10.5. Related Work 134

10.6. Conclusions and Future Work 135

Chapter 11. Evaluation 137
11.1. Scientific Results 139
11.2. The Prototype Implementation 140
11.3. Future Work 140

Bibliography 143

List of Figures

3.1 Code for the Queens Problem using Prolog(CLPFD) 23

4.1 The Equational Theory **DRA**. 30

5.1 The equational theory $\mathbf{QRA}_\Sigma$. 35
5.2 Standard interpretation of binary relations 36

6.1 Constraint meta-reductions 54
6.2 Preliminary rewriting rules for SLD_1 simulation. 54
6.3 Rewriting system for SLD. 55
6.4 Resolution Transition System 63
6.5 Relational Transition Rules 65

7.1 Definition of $dep : \mathbf{U} \to \mathcal{P}(\mathbb{N})$ 81

8.1 Basic Commutative Diagrams 88

9.1 Helper relations for the translation 113
9.2 Composition of Relations as a Computation Rule 118

10.1 Jugdments for Type-Checking and Arrow Adjustment 130
10.2 Approaches to Function Reduction in the Presence of Free Variables 132
10.3 Extended Arrow Normalization for Strict Evaluation 132

Resumen

El cálculo de relaciones binarias fue creado por De Morgan en 1860 para ser posteriormente desarrollado en gran medida por Peirce y Schröder. Tarski, Givant, Freyd y Scedrov demostraron que las álgebras relacionales son capaces de formalizar la lógica de primer orden, la lógica de orden superior así como la teoría de conjuntos.

A partir de los resultados matemáticos de Tarski y Freyd, esta tesis desarrolla semánticas denotacionales y operacionales para la programación lógica con restricciones usando el álgebra relacional como base.

La idea principal es la utilización del concepto de *semántica ejecutable*, semánticas cuya característica principal es el que la ejecución es posible utilizando el razonamiento estándar del universo semántico, este caso, razonamiento ecuacional. En el caso de este trabajo, se muestra que las *álgebras relacionales distributivas* con un operador de punto fijo capturan toda la teoría y metateoría estándar de la programación lógica con restricciones incluyendo los árboles utilizados en la búsqueda de demostraciones.

La mayor parte de técnicas de optimización de programas, evaluación parcial e interpretación abstracta pueden ser llevadas a cabo utilizando las semánticas aquí presentadas. La demostración de la corrección de la implementación resulta extremadamente sencilla.

En la primera parte de la tesis, un programa lógico con restricciones es traducido a un conjunto de términos relacionales. La interpretación estándar en la teoría de conjuntos de dichas relaciones coincide con la semántica estándar para CLP. Las consultas contra el programa traducido son llevadas a cabo mediante la reescritura de relaciones. Para concluir la primera parte, se demuestra la corrección y equivalencia operacional de esta nueva semántica, así como se define un algoritmo de unificación mediante la reescritura de relaciones.

La segunda parte de la tesis desarrolla una semántica para la programación lógica con restricciones usando la teoría de alegorías — versión categórica del álgebra de relaciones — de Freyd. Para ello, se definen dos nuevos conceptos de *Categoría Regular de Lawvere* y Σ-Alegoría, en las cuales es posible interpretar un programa lógico.

La ventaja fundamental que el enfoque categórico aporta es la definición de una máquina categórica que mejora e sistema de reescritura presentado en la primera parte. Gracias al uso de *relaciones tabulares*, la máquina modela la ejecución eficiente sin salir de un marco estrictamente formal.

Utilizando la reescritura de diagramas, se define un algoritmo para el cálculo de *pullbacks* en Categorías Regulares de Lawvere. Los dominios de las tabulaciones aportan información sobre la utilización de memoria y variable libres, mientras que el estado compartido queda capturado por los diagramas. La especificación de la máquina induce la derivación formal de un juego de instrucciones eficiente.

El marco categórico aporta otras importantes ventajas, como la posibilidad de incorporar tipos de datos algebraicos, funciones y otras extensiones a Prolog, a la vez que se conserva el carácter 100% declarativo de nuestra semántica.

Summary

The calculus of binary relations was introduced by De Morgan in 1860, to be greatly developed by Peirce and Schröder, as well as many others in the twentieth century. Using different formulations of relational structures, Tarski, Givant, Freyd, and Scedrov have shown how relation algebras can provide a variable-free way of formalizing first order logic, higher order logic and set theory, among other formal systems.

Building on those mathematical results, we develop denotational and operational semantics for Constraint Logic Programming using relation algebra.

The idea of *executable semantics* plays a fundamental role in this work, both as a philosophical and technical foundation. We call a semantics *executable* when program execution can be carried out using the regular theory and tools that define the semantic universe. Throughout this work, the use of pure algebraic reasoning is the basis of denotational and operational results, eliminating all the classical non-equational meta-theory associated to traditional semantics for Logic Programming. All algebraic reasoning, including execution, is performed in an algebraic way, to the point we could state that the denotational semantics of a CLP program is directly executable.

Techniques like optimization, partial evaluation and abstract interpretation find a natural place in our algebraic models. Other properties, like correctness of the implementation or program transformation are easy to check, as they are carried out using instances of the general equational theory.

In the first part of the work, we translate Constraint Logic Programs to binary relations in a modified version of the distributive relation algebras used by Tarski. Execution is carried out by a rewriting system. We prove adequacy and operational equivalence of the semantics.

In the second part of the work, the relation algebraic approach is improved by using allegory theory, a categorical version of the algebra of relations developed by Freyd and Scedrov.

The use of allegories lifts the semantics to *typed* relations, which capture the number of *logical variables* used by a predicate or program state in a declarative way. A logic program is interpreted in a Σ-allegory, which is in turn generated from a new notion of Regular Lawvere Category.

As in the untyped case, program translation coincides with program interpretation. Thus, we develop a categorical machine directly from the semantics. The machine is based on relation composition, with a pullback calculation algorithm at its core. The algorithm is defined with the help of a notion of diagram rewriting. In this operational interpretation, types represent information about memory allocation and the execution mechanism is more efficient, thanks to the faithful representation of shared state by categorical projections.

We finish the work by illustrating how the categorical semantics allows the incorporation into Prolog of constructs typical of Functional Programming, like abstract data types, and strict and lazy functions.

Preface

The use of software is increasing from day to day while more of our crucial needs become dependent on computers. Unfortunately, software malfunction is also pervasive and causes all types of problems, ranging from the frustration of the occasional gamer suffering a crash in her video game, to failures in critical systems that jeopardize human life.

Software malfunction may be caused by a wide variety of factors, from human error to hardware failure. Preventing and handling all the possible causes of failure is impossible. However, the great majority of errors can be traced back to human mistakes.

Correctness of program construction is dependent on the correctness of three layers. First, the programmer is supplied with *requirements* or *specifications* that she must convert into code. Such specifications are commonly written by humans in English. Obviously, specifications themselves are subject to mistakes and what is worse, a natural language specification is extremely hard to check for errors.

The second layer is the code itself. Programmers will write code in order to implement the specification. With imperative programming languages, the gap between the specification and implementation is commonly so big that the programmer will have a hard time to produce a correct program. It is difficult for a human to foresee all the possible execution states of a complex system. However, contrary to a natural language specification, a computer program has a precise mathematical meaning, so we can use tools or strongly-typed languages to help detect many common mistakes.

The third layer is the system layer. No matter how perfect our code or specification is, bugs in the toolchain, runtime environment or operating system may cause our program to crash or return incorrect results.

It has been argued [Brooks, 1995] that correctness of computer software is highly dependent on the skills of a particular programmer. Brooks also argues that team work only makes matters worse in most cases.

While we consider Brook's view reasonable at the time, recent advances in programming language theory make it somewhat obsolete. The use of special purpose programs allows us to check the correctness of software. Theorem provers, type systems and verifiers reduce the human impact in software correctness and in some cases the full correctness of a computer system can be asserted.

On the other hand, despite the large amount of progress in the design and implementation of programming languages in the last 50 years, we should consider it still a young discipline with significant remaining challenges. At the user level, bugs remain commonplace, and the programmer should be able to obtain much more help from her tools in order to avoid mistakes. At the language level, formally proving correctness properties of compilers and interpreters remains very difficult and costly.

The vast majority of programs in widespread use have been written using imperative languages. Providing direct access to hardware features was an important design requirement, and came at the cost of safety, expressiveness and re-usability. Thus, software today tends to be efficient, but safety and correctness problems are often severe.

In the hope of improving this situation, declarative programming languages place correctness before faithfulness to the underlying hardware model. Originally considered impractical, theoretical and

practical advances have allowed declarative languages to compete with the imperative world on all fronts. As previously mentioned, in order to check the correctness of a system we must use a mathematical framework or *theory*. Using it, we assign a precise *meaning* to programs, and formally define our notion of *correctness* and *mistake*. Given that a formal system is defined by a set of rules, building an automated tool to help us with proofs and reasoning is now feasible. Unfortunately well known results like the Gödel incompleteness theorem, the undecidability of the halting problem or more generally Rice's theorem [1] mean that we cannot build a general purpose *error finder* program for most classes of common mistakes.

A quote from Martin Davis may be appropriate to illustrate the situation we face when trying to prove the correctness of a system:

> As Goedel himself pointed out already in his 1933 lecture, instead of a single system we are confronted by a hierarchy of systems, and as we rise in the hierarchy, more and more previously unproved propositions become provable. [...] In this world what is important is systems that can be used to calibrate the strength of propositions.

No matter what amount of formalism we incorporate into programming languages, some upper layer or specification won't be a candidate for mechanization. Fortunately, this doesn't amount to a problem in practice, new generation theorem provers and programming paradigms are close to proving correctness of computer software at a level that can be considered safe.

Declarative Programming

Functional programming is based on the concept of mathematical function. Its theoretical foundations were pioneered by Curry and Schönfinkel, leading to the invention of λ-calculus [Barendregt, 1984, Cardone and Hindley, 2006] by Church, where each function carries a *type*. The notion of type plays today an important role in helping the programmer prevent mistakes. Types contain in a logical way important information about a function's parameters and result. Armed with this information, a compiler may reject ill-typed function composition and application, which undoubtedly signals an error by the programmer. Different type systems [Pierce, 2002] are possible depending on the logic chosen. Advanced logics will allow for more safety checks to be encoded in the type system. Other features such as combinators and higher-order functions are great tools for writing high performance programs and encouraging code reuse [Hudak and Jones, 1994].

Functional programming languages like Erlang [Armstrong, 2007], Haskell [Jones and Hughes, 1999] and ML [Milner, 1973, Leroy et al., 2011] enjoy wide use and acceptance by the industry. Complex systems have been implemented, while maintaining strong correctness guarantees, achieving excellent performance, and using less man-hours than imperative languages. For more details, the interested reader should consult the *Commercial Users of Functional Programming* website `http://www.cufp.org`.

The *Logic Programming* paradigm (LP) is based on interpreting logical theories as programs and theorem proving as computation. In particular, Prolog is based on a subset of first order logic [van Dalen, 1983] called Horn Clauses and on the resolution proof method [Robinson, 1965] as the theorem proving strategy. Running a program amounts to querying the LP system for a proof of an existentially quantified conjunctive formula. If there exists a proof for it, the answer will often include information about the instantiations of the existentially quantified variables that were performed during the proof. For instance, consider a predicate `related` formalizing the existence of a familiar link between two persons. Then, the query `related(john, X)` is interpreted as an existentially quantified formula and the system will successively return the instances of `X`, that is to say, persons, that the system can prove related to John. In

[1] Recall that Gödel's second incompleteness theorem states that any consistent formal system with certain complexity cannot prove its own consistency. The undecidability of the halting problem means that there is no Turing machine that can determine whether any arbitrary Turing machine will halt for all its inputs. Rice's theorem states that for any non-trivial property of a partial function, there is no general and effective method to decide whether an algorithm computes a partial function with that property.

standard semantics, predicates are interpreted as relations on the universe generated by the program's signature.

LP is a mature and important paradigm. Prolog [Sterling and Shapiro, 1986, O'Keefe, 1990] is the *standard* language, but multiple extensions and variations exist, such as Constraint Logic Programming [Jaffar and Maher, 1994, Marriott and Stuckey, 1998], Functional Logic Programming [Hanus et al., 2003, , ed., López-Fraguas and Sánchez-Hernández, 1999], Answer Set Programming [Niemelä, 1999], Higher-Order Logic Programming [Miller et al., 1991, Nadathur and Miller], and Concurrent Constraint Logic Programming [Shapiro, 1989, Tick, 1995] to name a few.

Prolog excels in specialized problems but for medium and large scale programming it doesn't work well. The lack of types means that the compiler has a hard time catching bugs. The non-deterministic nature of proof search hinders modularity. Programmer productivity suffers. Backtracking often causes bugs to propagate among modules, and makes compartmentalized debugging very hard unless meta-logical primitives are used. In order to overcome this problem, very interesting proposals have been developed. [Bueno et al., 1999] uses program analysis to support types and prove properties of Prolog programs, but the pervasive use of non-logical features like cuts and meta-predicates takes programs too far away from their original logical semantics.

Hybrid approaches like Mercury [Somogyi et al., 1996] or Curry provide types. However, they both lack standard denotational semantics or advanced type systems, making reasoning over them harder than over pure functional languages. Although some proposals have been made [Mariño, 2002], the formalization of logical variables remains an important problem in the context of these languages. λ-Prolog is an interesting alternative, providing adequate support for logic programming with types and semantics based on uniform proof systems. But the Prolog programmer suddenly loses source code compatibility with her previously existing programs. While in research environments, the adoption of new languages and technologies is quick and even encouraged, expecting industry programmers to ditch all their existing codebase is too stringent a request and will effectively stop adoption of new technology.

While both functional and logic programming enjoy solid logical foundations, the programming experience varies enormously. In FP, formulas (types) play the role of specifications, while the programmer is mainly expected to write a proof. Computation is performed by proof normalization, usually in the form of reduction. In LP, implicative formulas carry all the computational content, and automated proof search takes care of building a proof and providing the needed instantiations for it. This differentiation is rapidly changing with systems like COQ or Agda, which offer the programmer a mixed view. The boundary between proofs and types is diluted, as types may contain programs. On the other hand, the programmer is free to write its own proofs or use sophisticated proof search methods, leaving the programmer in-between both paradigms.

All these systems are based on advanced denotational semantics, where types have a prominent role from the programmer's point of view. What is more, FP systems have successfully adapted most of the advancements of LP systems, like free variables and proof search, without altering compatibility with previously written programs.

On the other hand, Prolog systems have trouble incorporating in declarative manner some of the features of FP languages, and many popular implementations are still based on designs derived from the Warren Abstract Machine.

The work carried out in this thesis constitutes a step towards the enhancement of both the implementation and semantics of constraint logic programming systems, and to the modernization of logic programming languages by allowing them to incorporate new features which are convenient for the programmer without giving up the declarative semantics.

Structure of the Work

Most of the contributions of this thesis were developed as papers, with chapter 6 corresponding to [Gallego Arias et al., 2012b], chapter 7 to [Gallego Arias et al., 2011], chapter 9 to [Gallego Arias and Lipton, 2012] and chapter 10 to [Gallego Arias et al., 2012a].

For the sake of readability and extended content we present the thesis as a merged version of the above papers. The thesis is divided into 10 chapters, with two main parts.

First we introduce the work and survey some needed material. In Chapter 1, we provide an introduction to the thesis itself. Chapters 2, 3, 4, and 5 briefly introduce programming language theory, logic programming, the calculus of binary relations and Tarski's results on the formalization of logic using relation algebra.

The first part presents the results of the translation of CLP programs to binary relations. Chapter 6 develops relational semantics for constraint logic programming, while Chapter 7 develops a unification algorithm fully-within the pure relational theory.

The second part comprises the categorical version of the results of the first one. Chapter 8 presents the necessary concepts about category theory used in Chapters 9 and 10, where the categorical semantics are developed.

The dissertation is closed in Chapter 11, where we evaluate the work and outline general future work.

Related work, state of the art and specific future work is separately discussed in each chapter.

Acknowledgments

Tras estos años de trabajo, la verdad tengo que agradecer mucho más de lo que puedo poner en esta pequeña sección. Muchas personas me han ayudado y apoyado, en la mayor parte de manera inmerecida. Acordarme de todos es imposible, gracias.

El primer agradecimiento particular va para a mis directores, Julio y Jaime, que han compartido generosamente sus conocimientos e ideas conmigo, me han ayudado mucho y han tenido mucha paciencia y aguante.

El segundo agradecimiento es para la Comunidad de Madrid, que generosamente ha tenido a bien contratarme durante 4 años para la realización de este trabajo. También quiero agradecer a la UPM, Wesleyan y los proyectos DESAFIOS y PROMESAS sus aportaciones económicas para viajes y estancias.

Quiero dar las gracias también a todos mis compañeros del grupo Babel, especialmente a Ángel y Pablo que leyeron una versión preliminar de esta tesis por sus comentarios y a Juanjo por su ayuda. Álvaro(s), Ana, Guillem, Victor, Lars, Clara y Susana que siempre me apoyaron y dieron excelentes consejos. Thanks to Jamie, our chats about your work were enlightening for a programmer like me.

Muchas gracias a Ángeles, eres una excelente profesional.

En el plano personal, es imposible cuantificar la ayuda y ánimos que mis padres, hermanos, tías, tíos, primos y amigos me han proporcionado. Cómo han sido capaces de aguantar mi larga lista de quejas y mi pertinaz pesimismo, es todo un misterio. A ellos, les dedico esta tesis.

Muchas gracias a Carol y a su familia por todos estos años ayudándome, animándome y teniendo mucha paciencia conmigo, siento mucho no haber acabado la tesis antes.

CHAPTER 1

Introduction

In this thesis we develop two theories or *semantics* for Constraint Logic Programming using relational methods. In particular, the first semantics is built upon a custom distributive relational algebra, directly based on Tarski and Givant's work [Tarski and Givant, 1987]. The second semantics uses Freyd and Scedrov's allegory theory to improve certain aspects of the purely relational first one.

In general terms, each predicate of a logic program is translated into a coreflexive relation, that is to say, a relation contained in the identity relation. Coreflexive relations are closely related to the notion of subset and the set-theoretical semantics of the coreflexive relation resulting from the translation of a predicate yields a semantics equivalent to the classical set-theoretical semantics for CLP.

The main features and benefits of our relational approach are:

- The necessary meta-theory for CLP is fully captured by the relational theory. Handling of logic variables and proof search are carried out using a purely relational approach. The compilation process and tools are able to reason about programs without using any extra-relational theories.
- In addition to an adequate denotational model, the semantics provides a good framework for program *execution*. The formalization of the full CLP theory and meta-theory by relations means that execution is just another instance of equational reasoning.
- The use of category theory opens the door to the development of extensions for Prolog and the reuse of existing semantic developments, such as types and functions, in a declarative and backwards-compatible way.

Existing categorical and algebraic semantics for Constraint Logic Programming fail to address the three criteria outlined above. While the existing semantics constitute an excellent and inspiring body of theoretical work, in our opinion they are far from becoming the basis of an actual implementation. Important aspects for programmers, such as efficiency or declarative extensions are often not even considered, nor is it clear how they would fit in the framework.

The new relational semantics are validated by a proof of *adequacy*, where it is shown that for any program, its classical semantics and the set-theoretical interpretation of its relational translation coincide. The equivalence of the relational machine with classical operational semantics for SLD proof is proven by an *operational equivalence* theorem, stating that both systems will return the same answer for any query and program.

1.1. Relational Methods and Logic Programming

Logic programming languages, at least in principle, are based on the notion of computing directly with a logical description of a problem, that is to say, with specifications. Computation is reduced to proof-search in such a way as to ideally separate the programmer from some of the procedural aspects of computing, to eliminate concerns about *how* results are obtained and replace them with definitions that specify *what* is to be computed. One quickly finds, however, that many procedural concerns must be taken into account, not only in the choice of definitions, but in the metalogical management of proof search, including, in particular, treatment of logical variables, avoiding name clashes, etc. This meta-logical layer lives outside the specification semantics.

Most semantics identify a program-level logical variable with an existential variable in the logic. This choice may seem harmless to the uninitiated reader, however, it has a big impact, particularly for building an efficient implementation which can be related to the semantics.

The behavior of logical variables is usually unspecified, with the implementors forced to use a low-level design in the name of efficiency, a choice which implies the creation of a considerable gap between the specification of the system and its implementation.

Relation algebras were shown by Tarski and Givant [1987] to capture several logical theories, including set theory. The Tarski-Givant theorems on formalizing mathematics without variables suggest a way to translate logic programs to variable-free terms in the relation calculus, essentially supplying a built-in abstract syntax.

Particular relation algebras capture several meta-theoretic issues like proof search, minimal semantics, and existentially quantified logical variables. What is more important, all these previously meta-logical aspects are now captured equationally. Implementations may be based on rewriting, and the gap between a particular implementation and its relational specification is smaller than in the non-relational case.

How to transfer Tarski's work to real-world logic programming? What would be the advantages of such an approach? Previous work by Lipton and Broome on variable-free logic programming in relation calculi [Broome and Lipton, 1994, Lipton and Chapman, 1998] led us to formulate our initial hypothesis: **Relation algebra and allegory theory constitute good frameworks for the development of denotational and operational semantics for constraint logic programming.**

Undoubtedly, there is a price to pay for the incorporation of meta-logical concepts into the theory itself, as the new semantics will be more complex, and the gained benefits may be not worth it. We believe that we have proven that this is not the case, and indeed, as the second part of this thesis shows, the formulation of the categorical machine is quite succinct and elegant, captures all the necessary concepts for an efficient implementation, and displays excellent properties of modularity, adaptability and extensibility.

Why are the relational semantics well-suited for execution and formalization of logical variables? We will discuss the issue in depth in the relevant chapters, but the main fact is that relation composition (or pullbacks in the categorical case) is a good model of existential quantification, and relational operators capture all the proof search meta-theory; all of this while maintaining a clear and concise set-theoretical denotational interpretation.

For instance, the fact that a query R has exactly two answers may be expressed as $R \to^! S_1 \cup S_2$, assuming $\to^!$ to be the normalizing rewriting relation in charge of proof search. (Note that the equation $R \cup R = R$ shouldn't be present in our theory to precisely reason about the number of solutions).

The combinatorial — or variable-free — nature of relation algebra stems from the interpretation of relation composition. Differently from the functional case, the composition of two relations captures existential quantification, as the relation $R; S$, "R composed with S", relates a with b, $a(R; S)b$ if exists a z such that $aRz \wedge zSb$.

This carefully chosen example (from [Lipton and Chapman, 1998]) illustrates in a simple way the use relation algebra to *interpret* a logic program and *execute* a query against it. Consider the Prolog program for the reflexive transitive closure of a simple graph, and two queries:

```
edge(a,b).
edge(b,c).
edge(a,l).
edge(l,c).

conn(X,X).
conn(X,Y) :- edge(X,Z),conn(Z,Y).

% Queries:
```

```
?- conn(a,c).
?- conn(X,c).
```

We introduce the binary relation symbols $conn$ and $edge$ (and let the font difference suffice to distinguish between program predicates and the new symbols), and translate the program into a pair of relation equations. The $conn$ identifier stands for a relation defined via an equation and $edge$ stands for an expression which explicitly describes a finite binary relation on the Herbrand universe $\mathcal{H}$ of the program. For readability, we denote composition of relations by semicolon.

$$\begin{aligned} edge &= \{(a,b),(b,c),(a,l),(l,c)\} \\ conn &= id \cup edge; conn \end{aligned}$$

where id denotes the identity relation.

The first query above is represented by $\{(a,c)\} \cap conn$ and the second by $(\mathbf{1};(c,c)) \cap conn$ where $\mathbf{1}$ is the universal relation. Hence $(\mathbf{1};(c,c))$ represents the set of all pairs whose second component is c. A solution to the first query would have be to be of the form $\{(a,c)\}$, confirming that $(a,c) \in conn$,, or otherwise $\mathbf{0}$, the empty relation. A solution to the second query should be a more explicit description, such as $\{(a,c),(b,c),(c,c),(l,c)\}$.

The computation of the second query can be carried out using simple rewriting rules that capture basic relation identities. A natural first strategy is to unfold the recursive definition of $conn$:

$$\mathbf{1};(c,c) \cap conn = (\mathbf{1};(c,c) \cap id) \cup (\mathbf{1};(c,c) \cap edge; conn) = (c,c) \cup (\mathbf{1};(c,c) \cap edge; conn)$$

obtaining the first answer, (c,c), given that $\mathbf{1};(c,c) \cap id$ is equal to (c,c). We proceed to unfold the right component of the union in the search for more answers:

$$\begin{aligned} \mathbf{1};(c,c) \cap edge; conn &= \mathbf{1};(c,c) \cap edge; (id \cup edge; conn) \\ &= \mathbf{1};(c,c) \cap (edge; id \cup edge; edge; conn) \\ &= \mathbf{1};(c,c) \cap (edge \cup edge; edge; conn) \\ &= (\mathbf{1};(c,c) \cap edge) \cup (\mathbf{1};(c,c) \cap edge; edge; conn) \\ &= (b,c) \cup (l,c) \cup (\mathbf{1};(c,c) \cap edge; edge; conn) \end{aligned}$$

Continuing this way, we will eventually obtain the term $\mathbf{1};(c,c) \cap (edge; edge)$ which is equal to $\{(a,c)\}$. The next unfolding gives us $\mathbf{1};(c,c) \cap (edge)^{(3)}; conn$. Notice $edge$ is finite, for any $n > 2$, $(edge)^{(n)} = \mathbf{0}$, thus we are done. The query evaluates to $(c,c) \cup \{(b,c),(l,c)\} \cup \{(a,c)\}$ which is simply $\{(c,c),(b,c),(l,c),(a,c)\}$.

1.2. Objectives and Motivation

In the light of the previous examples, the main objective of this work was to provide a declarative framework for Constraint Logic Programming covering all aspects that are involved in the definition of a real world declarative language, from the high level denotational semantics down to an efficient implementation using a virtual machine.

The typical approach to declarative programming language design is to build operational and denotational semantics, and then prove their equivalence. Usually, the resulting semantics are too far apart, requiring difficult proofs of operational correspondence and adequacy of the translation. Even with recent advances, the gap is there.

In our approach, relational terms are used for both the denotational and the operational representation of the program, thus we call our semantics "executable", given that computation is an instance of regular reasoning.

So the initial hypothesis which this work was started under was that using binary relations for interpreting logic programs would close the specification-implementation gap significantly and allow an efficient execution engine to be defined by relational rewriting.

The hypothesis, if correct, would bring many benefits. Analyzers, verifiers, and partial evaluators could be implemented using the same semantic framework, and would share most of the implementation of the execution engine. Denotational and operational semantics share the same theory, so obtaining the denotation of a particular execution state would be an immediate and well defined operation. Proof search strategies and meta-logical operations such as cut could be specified in declarative way. The problematic notion of *logical variable* would be removed from the metalevel and fully specified.

The second set of incentives was the exciting possibilities that the use of relations opens in order to extend logic programming with algebraic data types and functions in the style of [Bird and de Moor, 1996a]. The framework could be adapted in order to obtain a semantics for functional logic programs.

In more technical terms, the precise initial hypothesis could be stated as: the theory of distributive relation algebras with a fixpoint operator fully captures the theory of logic programming and that a rewriting system over relational terms would provide efficient execution of SLD queries.

The theoretical part of the hypothesis is proven in the first part of this thesis. However, the achievement of an efficient execution model was not possible for technical limitations associated with so-called relational quasiprojections, defined later in this thesis.

So in order to achieve the objective of an efficient execution model, the second part of the thesis was born. The new hypothesis was that the slightly weaker category theory version of relation algebra — allegory theory — could be used in order to build an efficient and still declarative execution model. The use of category theory should allow much simpler proofs, simplifying key structures and allowing them to be correct by construction.

Briefly, the notion of tabular allegory revealed itself as an ideal foundation for declarative extensions to the logic programming paradigm. Existing work in types, monads and other constructs typical of functional programming seemed to fit very well within our developments. Given the exciting possibilities that the use of categorical tools has had for the functional programmer, we felt obligated to study how the categorical semantics could accommodate them.

Apart from the primary motivations and objectives, we had a less defined secondary one: the desire to explore the suitability of computing in general with relations. Studying computation in the full relational calculus was out of our scope, but we sought some new insights and understanding of computation within the theory of binary relations. This work was a good opportunity to explore the appropriate and adequate fragments or relation algebra for use in programming languages and systems implementation.

1.3. The Thesis

The contents of the thesis are organized in two main sections. The first part deals with the formalization of Constraint Logic Programming in a custom distributive relation algebra. The second part formalizes Logic Programming using category and allegory theory.

1.3.1. Relational Semantics. In the first part, we interpret Constraint Logic Programs in distributive relation algebras enriched with quasiprojections, constants for the signature of the program, primitive constraint predicates, and a fixpoint operator. The fixpoint operator is not fundamental and may be replaced by adjoining predicate definition equations to the theory.

We study how the theory of distributive relation algebras captures the fragment of first order logic consisting of $=$, $\wedge$, $\vee$ and $\exists$, notions corresponding to unification, conjunction, disjunctive clauses and local variable creation. The modular law plays an important role in capturing the notion of *predicate call*.

Quasiprojections formalize a quasi-product type, with the difference that a quasiprojeciton hd relates *all* sequences of finite length to its first element, whereas a clasical projection has a domain of fixed-length. Quasiprojecitons were introduced in Tarski-Givant to overcome certain logical limitations. In our semantics they are used for formalizing most of parameter passing and renaming apart. The union of all term sequences of finite-length is the carrier of our relational algebra.

In order to capture the definition of recursive predicates, we may use a fixpoint operator. However, the first order nature of predicate definition means that the fixpoint operator can be accurately replaced by a set of relational constants and its corresponding equations.

We prove a semantic adequacy result and define a rewriting system for relational terms to simulate SLD resolution. The rewriting system is proven equivalent to a traditional operational semantics for CLP using SLD proof search. We define a new algebraic semantics for SLD resolution based on first-order logic and transition systems in order to carry out the proof.

In our approach, constraint solvers are handled as *black boxes*. The satisfiability of a constraint is delegated to a specialized engine.

After the formalization of CLP using relation algebra, we study the implementation of constraint solvers using the calculus of relations. The problem of first-order unification is given a relational solution, encoding terms as variable-free relations and developing an algorithm correct and complete for the original problem.

The newly developed relational theory meets all our requirements but one: we can execute the semantics but we cannot derive an efficient execution strategy from it. The use of generic projection relations — also known as quasi-projections — means that the theory doesn't capture the number of free variables in use in a given state of the execution of query. This information is a requirement in order to derive an efficient machine for constraint logic programming. In each predicate call, we may need to generate new logical variables. Adding a meta-level counter to keep track of the number of used variables would solve that problem, but this approach would defeat the spirit and purpose of the work. We develop a fix for this problem using category theory.

1.3.2. Allegorical Semantics. The second part starts by introducing the new categorical constructions able to interpret logic programming. Freyd's allegories and tabular allegories capture relational algebra in a categorical way, but some extra structure is needed in order to support the signature of a CLP program.

We develop the notion of Regular Lawvere Categories, prove them regular and study its models. As for any regular category, Regular Lawvere Categories generate a corresponding tabular allegory, which is then completed to define a Σ-allegory, an allegory where arrows have a tabulation iff they possess a union-free representation.

The internal logic of Σ-allegories captures all the theory of CLP save for recursive predicate definition, which is incorporated externally. Predicates are translated into arrows in a Σ-allegory. The denotational semantics of programs in this setting is studied, with the adequacy theorem being very similar to the non-categorical case.

The big change from the first part happens when studying the operational semantics of programs in the categorical setting. A notion of diagram computation is used in order to define an algorithm for pullback calculation in Regular Lawvere Categories. In a tabular allegory, relation composition is defined in terms of pullbacks of their tabulations, so we are able to compute composition of union-free arrows in a Σ-allegory.

The categorical machine is then specified, based on diagram rewriting and the primitive of relation composition, plus some algebraic rules for predicate call and backtracking. We define a compact representation of its rules and prove it operationally equivalent to classical SLD semantics.

We comment on some implementation details, like the design of a instruction set for the machine and the notion of memory diagram, which captures memory requirements for term storage.

To end the thesis, we discuss how to enrich our *tabular allegories* with constraints, data types and functions. The validity of this approach is shown as these new features are integrated seamlessly into the existing framework. The newly added structure *propagates* in two directions: models of the categories are naturally extended, and the relational composition algorithm is modified to handle the new constructions in a modular way.

1.4. Contributions

We distinguish two kind of contributions. First, we advocate some particular approaches and guidelines for programming language design and implementation. Then, a set of technical results and their development support the philosophical choices previously advocated.

The guidelines defended in this thesis are:

- Prolog should be extended with convenient features for the programmer, without losing compatibility with existing programs.
- Relation algebra is an adequate framework for reasoning about proof search.
- Algebraic semantics and denotational semantics should capture the full theory of a programming language, not just a denotation. Execution should belong to regular object-level reasoning.
- Implementations should be straightforwardly derived from the algebraic semantics.
- Algebraic semantics that don't support efficient execution should be avoided.
- Category theory is a very good framework for programming language theory and meta-theory.
- A particular categorical semantics shouldn't exclude the reuse of previous results.
- Enrichment of categories should be done in an "incremental" fashion. That is to say, for example, instead of requiring all coproducts to exist in a category, we will strictly require the minimal amount needed.

And the list of technical contributions is:

- New relation theory which captures most of the usual meta-theory of Constraint Logic Programming, including first-order variables and terms, Clark's equality theory and "black-box" constraints. For convenience, we rely on a meta-theoretical first order name definition to implement recursion.
- New translation procedure for Constraint Logic Programs to relational terms. Theorem: adequacy of the translation.
- New algebraic operational semantics for SLD proof search.
- New rewriting system for relation terms simulating SLD proof search.
- Theorem: correctness of the system, proof of operational equivalence.
- New algorithm for solving the first order unification by rewriting of ground relational terms. Proof of correctness and completeness.
- New notion of Regular Lawvere Categories. Definition, models and proof of regularity.
- New notion of Σ-allegories. Definition and models.
- New translation procedure from CLP programs to Sigma allegories. Adequacy of the translation.
- New notion of computing modulo the theory of Sigma Allegories using categorical diagrams.
- Algorithm for the calculation of pullbacks in Regular Lawvere Categories. Proof of correctness and completeness.
- Design of a categorical machine for CLP. Theorem: correctness and proof of operational equivalence.
- New notion of "memory" diagram, allowing the categorical machine to fully capture the memory usage of the program.
- Extension to the categorical model and machine to include algebraic data types. New type checking and intermediate code inference scheme.
- Extension to the categorical model and machine to include strict and lazy functions.
- Exploration of other extensions such as monads and residuating operational models.

1.5. Previous and Related Work

Detailed comment on related work is performed individually in each chapter. Here, we review prior results with similar goals to ours, and justify the differences that led us to take a different path.

1.5.1. Previous Work. We survey previous work that constitutes a direct predecessor of this thesis.

The calculus of binary relations was introduced by De Morgan in 1860, to be greatly developed by Peirce and Schröder [Pratt, 1992]. Tarski and Givant [Tarski and Givant, 1987], and Freyd and Scedrov [Freyd and Scedrov, 1991] showed relation algebras were able to capture first order logic, higher order logic and set theory.

In an attempt to transfer the cited mathematical work to computer science, Broome and Lipton [1994] applied some of the equivalence results to Logic Programming. In that seminal work, a relational version of the Mal'cev [Mal'tsev, 1961] quantifier elimination algorithm is presented, together with some preliminary results on the translation of logic programs. Lipton and Chapman [1998] focused in Prolog compilation and execution. Such work was used by the author of this thesis to build a preliminary implementation [Gallego Arias, 2004]. The implementation effort revealed the need for several improvements which constituted the starting point for this thesis.

Work by the author on operational semantics and constraint systems [Gallego Arias and Mariño, 2005, Gallego Arias et al., 2007, Gallego Arias et al., 2007] is a direct predecessor of the approach used for constraint integration into the relational paradigm.

The category theory part of this thesis has several quasi-predecessors. While the work with the pure relational calculus part is an improvement of previous existing work, the categorical part is a completely new formulation. Inspiration for its development came from three areas:

- Category theory: The use of categorical logic [Lambek and Scott, 1986] to capture different logical formalisms including λ-calculus was considered by the author to be a superior alternative to other approaches. Freyd and Scedrov's tabular allegories are the most attractive framework for a categorical version of relation algebra and provided technical constructions very useful for the development of this thesis.
- Categorical semantics for Logic programming: Representative works are [Asperti and Martini, 1989, Corradini and Asperti, 1992, Kinoshita and Power, 1996, Finkelstein et al., 2003, Amato et al., 2009, Amato and Lipton, 2001, Lipton and McGrail, 1998]. The fundamental idea in categorical semantics for logic programming is the representation of a logical variable as a projection. Lawvere Categories [Lawvere, 1968, 1969a, Hyland and Power, 2007] and indexed monoidal categories [Mac Lane, 1998] are two common categorical tools used for the definition of the semantics.

  Specifically, works in [Finkelstein et al., 1994, Amato and Lipton, 2001, Finkelstein et al., 2003, Amato et al., 2009] are very close in spirit to the work developed in this thesis. However, the approach used here is fundamentally different from these works and it becomes difficult to establish a direct ascendancy.

  We couldn't find any work on categorical semantics which handled the topic of implementation.
- Efficient abstract machines and categorical abstract machines: Most Prolog implementations are based on derivatives of the WAM [Warren, 1983, Ait-Kaci, 1991]. A very fast machine for the execution of Prolog, its specification is tied to a particular operational strategy and far apart from the actual logical semantics. Efforts in other areas like functional programming have proven that is its possible to build efficient abstract machines whose specification is very close to the actual mathematical models, like the ones in [Cousineau et al., 1987, Jones, 1985].

1.5.2. Related Work. Although originating from a very different context than Tarski's equipollence result, the work by the Algebra of Programming group [Bird and de Moor, 1996a] should be seen as the closest related work to this thesis. Their use of tabular allegories to specify and transform *relational programs* is very similar to the approach used here, with the important difference of our use of Lawvere Categories.

We started with the relational interpretation of first order logic developed by Tarski. In the search of enhancements we settled for the use of tabular allegories. They started with categorical semantics for functional programming and moved to tabular allegories to get some of the advantages of reasoning in the relation calculus. Their efforts were continued by McPhee [1995], but unfortunately, McPhee's work did not develop an executable model.

Later, Seres et al. [1999] provided an embedding of logic programs into the lazy functional language Haskell. Then, valid algebraic reasoning in Haskell can be used to optimize logic programs. There are many differences with our approach. The embedding relies in the full theory and metatheory of Haskell in order to reason about logic programs. Implementation details are left to the Haskell compiler. We consider the embedding too shallow for our purposes, predicates cannot profit from Haskell type system, and even if this is proposed as future work we cannot see how they could achieve this objective with their current embedding.

Several algebraic or compositional semantics exist for logic programming. The works of de Boer et al. [1997], van Emden [2006] develop an algebraic semantics based on Tarski's cylindrical algebras, but no comment on implementation or extensions is given. The same applies for the compositional semantics of Comini and Meo [1999], Comini et al. [2001].

Interesting related work are the linear logic semantics for allowed logic programs of Cerrito [1990] and Jeavons [1997]. Indeed, in this last semantics, $\otimes$ distributes over $\oplus$, mimicking the relational algebra distributive law for $\cap$ over $\cup$.

A notable and seminal work is the work of de Bakker and de Roever [1972], de Roever [1974], where a (typed) relational calculus is developed for program schemes. Later, the non-recursive subset was proven complete in [Frias and Maddux, 1998]. Without any doubt, the work on relational program schemes served as an inspiration for the treatment of logic programming started by Lipton and Broome.

1.6. Notation

This thesis studies relationships among several branches of mathematics, logic, programming and category theory. Thus, notation can be sometimes complex or difficult to follow. Before becoming thoroughly familiar with the calculi involved the main challenge for the reader may be to determine to what domain a written expression belongs. We try to shed some light in our choices of notation, with the hope of simplifying the reading of the thesis.

We have slightly abused and overloaded vector notation throughout the thesis, with the hope to make it more readable. The use of vectors ranges from simple sequences $\vec{x} = x_1, \dots, x_n$ to complex expressions containing lots of subvectors $\vec{p} = p_1(\vec{u}_1), \dots, p_n(\vec{u}_n)$, where $\vec{u}_k = u_{k_1}(\vec{t}_{k_1}), \dots, u_{k_m}(\vec{t}_{k_m})$, and the same for $\vec{t}$, etc.

The basic domains involved are:

- **First Order Logic** First order logic formulas and terms are written using the standard logical operators, lowercase for variables and terms, and uppercase for metavariables denoting predicates: $\forall x_1.\ P(t(x_1)) \rightarrow \exists x.\ P(t(x_1))$. Constraints are assumed to belong to the set of first order formulas.
- **Relation Algebra** For relation algebra, we can distinguish relational terms, with relational variables written in uppercase $A \cap (B \cup C)$ and equations $A \cap B = B \cap A$. Elements of a carrier $A^\dagger$ of the algebra, are usually super scripted with the carrier name, such as t^A.
- **Category Theory** For category theory, arrows are written in lowercase letters f, g, categories in calligraphic font $\mathcal{C}$, and objects in uppercase, $f : A \rightarrow B$. Functors are written with uppercase letters, and natural transformations in greek lowercase: η.

Some expressions — that we call compound — mix notation from some of these basic domains. They are:

- **Constraint Logic Programming** CLP programs are written using typewriter font:

```
nat(o).
nat(s(X)) <- nat(X).
```

Their associated logical meaning (e.g. $\forall x.nat(x) \rightarrow nat(s(x))$) is written in first order logic notation. Operational semantics, queries and transitions between them are written in the usual style, and we'll usually prefer using the logical meaning of the program to the actual code. In some examples, we may capitalize variables trying to underscore the connection of the logical variable with the one in the program.

- **Rewriting System** We will use a rewriting system for relational terms. For the sake of clarity, we use the same notation for terms of the rewriting system and relational terms. So, $A \cap B$ actually is both a relational term and a rewriting term, distinguished by the context. More details on the encoding can be found at the beginning of Section 6.4.
- **Logical and Relational States** We will define several mixed structures, such as logical and relational states. The definition of these intermediate structures uses notation for several domains, as they may include a mix of say, first order formulas and relations. For instance, a very common pattern is the representation of relational terms that from the application of a translation function $K : FirstOrderLogic \rightarrow Relation$. Given a formula φ, we write $K(\varphi)$ for the relational term it denotes.

CHAPTER 2

Programming Language Theory

The objective of this section is to briefly present some notions of theoretical concepts about programming languages. We use the term programming language theory or *semantics* to refer to any formalism able to reason about programs. In this section, we will focus on a very narrow segment of formalisms with a mathematical base. The reader familiar with general notions of mathematical semantics of programming languages may safely skip this chapter completely.

A very important use of PL theory or semantics is definition of properties of programs. Some examples are:

- Termination.
- Run-time errors.
- Space and time consumption.
- Order of execution.
- Data integrity.
- Security properties.
- Maintainability.

The number of existing frameworks for the study of those properties is immense, some of them are based on mathematical principles some not. If our program is given a meaning using a formal or mathematical system, we may speak about *formalization* of its properties.

Indeed, a very desirable property of a semantics is that it is akin to *mechanization*, that is to say, that the system used is suitable for automated reasoning. This way, we can write programs that can help us when *checking other programs*, easing the manual effort and freeing the programmer from often tedious manual proofs.

Given some semantics and a formalization of some property, there are several approaches to check whether a program meets it. We may roughly split them into the following non-exclusive approaches:

Program Analysis and Verification: We write programs in a PL that allows them to run regardless of their compliance with the desired property. An external tool then examines the program or its trace and determines whether the property holds.

Correctness by Construction: Programs are built from pieces that when properly combined, produce programs that *never go wrong* with respect to some criterion. Usually, programs are written in a *safe* language. The compiler won't allow the program to run unless it can check certain correctness properties.

Testing: An automated system generates a set of inputs for a program. Then the program is run and the system checks the output. Tests usually have a notion of *coverage*, allowing the assertion of some property with a varying degree of confidence.

A particular case of the correctness by construction approach is the family of declarative languages, family to which both functional programming and logic programming belong. A program is declarative if it specifies "what the desired result is" instead of "how to get the answer". Usually, declarative languages include in their definition rigorous mathematical semantics. Ideally, a declarative programming language should allow the programmer to forget control flow, easing her work towards a correct program.

2.1. Theories and Formalisms

What does "formal" mean? When we speak of *correctness* and *proof of correctness*, we need to define the notions themselves, including that of proof. Humans can easily adapt to varying degrees of formalism, but as of today, computers don't enjoy of the same ability. Even the formal language used in mathematics falls often short when working in an automated system. A "formal" proof may be accepted by most mathematicians, but it may be impossible or very difficult to check using a theorem prover. [1]

Problems often overlooked are the definition and encoding of syntax and structures. A definition of some structure may be simple and usable for programming, however, it may not meet some desired properties. What is more problematic, in order to define syntax we need some new syntax for its definition! This egg-and-chicken problem is what we call the theory/meta-theory barrier. In order to define some theory or semantics we need a theory of the defining language itself!

Let's use a typical example to illustrate some of the aspects of formalization, theory and metatheory. Assume a system based on logic and provability. The system is equipped with a set of statements or *formulas* $\mathcal{T}$, with elements $\varphi_1, \varphi_2, \ldots$, and a provability relation $\vdash$, relating a finite number of elements of $\mathcal{T}$ with a single one. For the moment, we don't worry about the structure of $\mathcal{T}$ itself. We write $\varphi_1, \varphi_2 \vdash \varphi_{14}$ for φ_{14} is derivable from φ_1 and φ_2. If a multiset of formulas Γ allows the derivation of a statement φ, we write $\Gamma \vdash \varphi$. The semantics for this system is determined by the provability relation, which we could interpret as a subset $\vdash \subseteq (\mathcal{T}^* \times \mathcal{T})$ of the cartesian product of finite sequences of formulas and a formula.

The reader can see that we have used several informal statements in English, a *Kleene* start and some set theory in order to define just the basic aspects of our system. We didn't even specify what are the elements of $\mathcal{T}$ or $\vdash$.

Indeed, the $\vdash$ relation completely determines the behavior of our logical system, so we may refer to it as the theory of the system. When speaking about the specification of $\vdash$ itself, we will speak of *metatheory*.

The bare minimum that the metatheory of the system (or the theory of $\vdash$) must contain is the syntax of the formulas and a finite description of the elements of $\vdash$. If $\vdash$ has an infinite number of elements we need some additional machinery — usually an induction scheme — in order to finitely define it. Theory and meta-theory are usually very different frameworks. Indeed, given a description of meta-theory for $\vdash$, it is a common case that determining its associated theory is impossible.

Beyond the minimum requirements for a meta-theory, we may enrich it in order to reason about our theory, which is in this case the $\vdash$ relation. Let's assume a syntax for formulas such that if $\varphi \in \mathcal{T}$, then $\neg\varphi \in \mathcal{T}$, where we intend $\neg\varphi$ to mean "φ is not true". For our definition to make sense, we need to check a fundamental property of $\vdash$, *consistency*. Undoubtedly, the lack of consistency will be a bug in $\vdash$. If we allow the derivation of φ and its negation $\neg\varphi$ from any set of premises Γ, it will usually imply that the system can prove any formula — *"ex falso quodlibet"* — making it useless for reasoning, and contradicting our intuitions about the meaning of $\neg$.

How to formally specify the consistency property in the meta-theory? Here is an attempt:

$$\forall \Gamma \in T^*. \neg \exists \varphi \in T. (\Gamma \vdash \varphi \wedge \Gamma \vdash \neg\varphi)$$

In the above definition, we use conjunction and quantification over formulas, so if we want to define consistency, the metatheory won't be exactly trivial. We'll need new rules to handle universal quantification over formulas.

We may extend the metatheory for the logical system in several ways. We may negate the above formula to get a definition of inconsistency:

$$\exists \Gamma \in T^*. \exists \varphi \in T. (\Gamma \vdash \varphi \wedge \Gamma \vdash \neg\varphi)$$

[1] Note that we refer to *proof checking*, as proof search for an actual complex mathematical proof is beyond current capabilities.

or use a different formulation for the definition of consistency:

$$\forall\Gamma \in T^*.\forall\varphi \in T.\neg(\Gamma \vdash \varphi \wedge \Gamma \vdash \neg\varphi)$$

Other interesting properties, like $\neg(\Gamma \vdash \varphi) \Leftrightarrow \Gamma \vdash \neg\varphi$ are also possible to define.

An obvious fact is that the proof of consistency for $\vdash$ will depend on the correctness of the metatheory. Trying to *mechanize* or *automate* a theory lifts correctness problems to the metatheory and implementation of the tool used for the automation. But the situation is not so dim as it may look. Today, there exist systems whose results may be taken as 100% correct for all practical purposes. Theorem provers like Coq or Isabelle are very reliable and can produce a *proof certificate*, easily double-checked by a different tool. Fortunately, proof checking is a much easier problem than proof search.[2]

A common aim is for systems where the metatheory is simpler than the defined theory. For instance, if we use sequents to formalize provability in linear logic we write rules of the form:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}$$

Linear logic [Girard, 1987] has a very rich theory and a complex proof search. The meta-theory of sequents and formula syntax may be considered "simpler" than the theory of linear logic itself which is the set $\vdash$ of premises and conclusions. Checking the correctness of a system that manipulates sequents and implements LL notion of syntactical derivability is not difficult. In general, there is a link between the logical complexity of the correctness statements that we intend to check with the complexity of the needed meta-theory.

How does the semantics of a programming language look? Indeed, the above logical definitions are a good base upon which to build a programming language. in the rest of this thesis we will explore in depth the theoretical aspects of programming languages based on logic and proof search. For the rest of this section, we'll use a simple programming language: The λ-calculus.

We start with the needed meta-theory: Assume a countable set of variables $x, y, ... \in Var$. The set of terms Λ of the λ-calculus is defined by the following grammar:

$$\begin{array}{lcll} \Lambda & ::= & x & \text{if} \quad x \in Var \\ & | & (M\ N) & \text{if} \quad M, N \in \Lambda \\ & | & \lambda x.M & \text{if} \quad x \in Var, M \in \Lambda \end{array}$$

Variables M, N, ranging over terms and induction over terms live at the syntax meta-level. The λ-calculus is commonly seen as a functional notation. Intuitively, a term of the for $(M\ N)$ denotes the application of a function M to an argument N and a term of the form $\lambda x.M$ represents a function which takes a parameter x and returns M. Terms in Λ have a notion of variable x. We call x an object level variable or simply a variable and we call M, N meta-variables.

The syntax of Λ is not by itself very useful. We want to capture the intuitions about functions just stated, which implies that we need a rule for function application. We will define such theory using equality. Through history [Tanner, 1962], equational reasoning has appeared in numerous forms.

Equality between relation expressions is a central concept in this thesis, and computation is formulated via reduction rules, an oriented version of equality. In fact the central translation of logic to relation algebra defined in subsequent chapters transforms logical deduction into equational reasoning in relation algebra.

Rules for equality are simple: it is the most general relation that is symmetric, reflexive and transitive. That is to say, for arbitrary elements a, b and c, equality obeys:

$$\begin{array}{lcl} a = a & & \\ a = b & \Rightarrow & b = a \\ a = b \text{ and } b = c & \Rightarrow & a = c \end{array}$$

[2]to the point that it is one of the prime examples of a P vs NP.

where a rule $R \Rightarrow S$ means that if R is true, so is S.

A good example of the power of equality is combinatory logic [Curry and Feys, 1958]. Combinatory logic is the theory of equality plus two constants $\mathbf{K}$ and $\mathbf{S}$, a binary operator (application), and the equations:

$$\begin{array}{rcl} \mathbf{K}xy & = & x \\ \mathbf{S}xyz & = & ((xz)yz) \end{array}$$

Every computable function can be encoded as a term in combinatory logic. Note that the metatheory is really simple, we don't even need the notion of object level variable as in the case of the λ-calculus. This implies that equality and some basic syntax is in itself enough for computation. Why then use a more complex formulation like the λ-calculus? Indeed, as we stated before, meta-theory plays a crucial role in proving properties about a system. For instance, proving confluence of combinatory logic without using the confluence proof of the lambda calculus was quite hard. [3]

Back to the λ-calculus, the idea is that terms that represent the same function should be equal. The names of *captured* or local variables shouldn't affect a function's result. This property is called α-conversion. The relation between application and abstraction is called β-conversion. The standard ($\alpha\beta$) theory of the λ-calculus is defined by the rules:

$$\begin{array}{rcll} \lambda x.M & =_\alpha & \lambda y.N[x/y] & \quad (y \text{ free in } N) \\ (\lambda x.M)N & =_\beta & M[x/N] & \end{array}$$

The transitive nature of equality allows us to repeatedly apply the above equations in order to prove equality among arbitrary terms.

The metatheory of the λ-calculus doesn't enjoy the simplicity of combinatory logic. The presence of variables in Λ's syntax and the nature of function application forces the definition of $=_\alpha$ and $=_\beta$ to depend on the use a meta-level capture-avoiding substitution $[x/M]$, which in turn depends on the notion of *being free in a term*.

The intuition behind capture avoiding substitution is simple, but its formalization is far from it — the interested reader may see an example in [Gabbay and Mathijssen, 2008]. Indeed, the presence of the meta-level substitution in the rules defining the theory complicates induction and formal reasoning over Λ in multiple ways. What do we mean by "far from simple"? Let's define capture-avoiding substitution by induction on the syntax of Λ:

$$\begin{array}{lcll} z[x/M] & = & M & \text{if} \quad x = z \\ z[x/M] & = & z & \text{if} \quad x \neq z \\ (M\ N)[x/E] & = & (M[x/E]\ N[x/E]) & \text{if} \quad \text{true} \\ (\lambda z.N)[x/E] & = & (\lambda z.N) & \text{if} \quad z = x \\ (\lambda z.N)[x/E] & = & (\lambda z.N[x/E]) & \text{if} \quad z \neq x, z \text{ fresh for } E \end{array}$$

In the above definition, we see some side conditions based on equality, and an English sentence "z fresh for E", whose meaning is that z does not occur free in E. The existence of this sentence means that our definition is a *non formal* definition, which translates to *not directly usable* in automated reasoning. Trying to formalize "fresh" is harder than it looks, given that it is always possible to α-convert the term E in order to make z fresh for E. Much work on those topics has been done by Gabbay and others, see [Gabbay and Pitts, 1999, Dowek et al., 2010].

We end this brief discussion about formalization and meta-theory observing again, the existence of a tradeoff when choosing how to split theory and meta-theory, mainly depending on what properties we want to define, capture and prove.

Logic programming is a good example. In classical Herbrand semantics, proof search is placed at a meta-level. Thus, set-theoretic semantics doesn't capture notions as the number of failures or other properties related to the resolution proof method. In exchange, the semantics are based on standard sets.

[3] see problem 1 at `http://tlca.di.unito.it/opltlca/`

2.2. Operational Semantics

Operational semantics is the study and specification of the dynamic behavior of a program. The exact definition widely varies depending on the context and the culture of the programming language. Some authors may consider an interpreter of a language its operational semantics[4], while others may use highly abstract notions.

The study of operational semantics was initiated in [Landin, 1965, Plotkin, 1975], to be fully refined in [Plotkin, 1981]. Much work has been done after that initial work. Natural Semantics [Launchbury, 1993], Modular Structural Operational Semantics [Mosses, 2004] and categorical operational semantics [Staton, 2008] are just some pointers to more current work.

Formal operational semantics are usually presented in two forms. *Small step* semantics take the form of a set of transitions between states of a program, where each transition captures a single computation step. Rewriting and transition systems are a common formal tool to write semantics in small step style. For instance, a small step operational semantics for combinatory logic is given by the rewriting system:

$$\begin{array}{lcl} \mathbf{K}xy & \to & x \\ \mathbf{S}xyz & \to & (xz(yz)) \end{array}$$

together with the condition that the rewriting is outermost. Better examples of small-step operational semantics arise in imperative languages, where the state represents the program's memory and each statement modifies it.

Small-step semantics are often convenient when we can prove properties of the rewriting system like confluence and orthogonality [Baader and Nipkow, 1998]. Rewriting systems also enjoy an advantage for modeling the operational behavior of concurrent systems, where big step semantics have a harder time capturing interaction.

Big step style operational semantics define evaluation of an expression by mapping it to its final form or output. A good example is the definition of an evaluator for the call-by-name strategy of the λ-calculus:

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.B \Downarrow \lambda x.B}$$

$$\frac{M \Downarrow M' \quad N[x/B] \Downarrow B'}{M\,N \Downarrow B'}\ (M' \equiv \lambda x.B) \qquad \frac{M \Downarrow M'}{M\,N \Downarrow M'\,N}\ (M' \not\equiv \lambda x.B)$$

In these rules, an expression of the form $t \Downarrow t'$ means that t' is the fully evaluated form of t.

Formal operational semantics opens the door for rigorously defining and checking interesting properties of a program's behavior.

For example, in typed systems, we may prove that evaluation is type preserving. Other examples of important properties that have been formalized for different ptogramming languages follow. An operational semantics is compositional if the evaluation of a compound term only depends on the evaluation of the subterms and a notion of composition. A programming language is strongly normalizing if every well formed program always terminates. Characterization of observational equivalence is key to create equivalence classes of programs, etc…

2.3. Denotational Semantics

Denotational semantics interpret programs as objects in some mathematical universe. Work in denotational semantics for programming languages was started by Scott and Strachey [Stoy, 1981, Winskel, 1993]. In logic, we may identify denotational semantics with work in model theory, ranging back to the early twentieth century.

[4]Note however the dependency introduced on the operational semantics of the interpreter itself, c.f.Reynolds [1998]

Mathematical universes or *domains* are usually defined by recursive equations. For instance, the domain for the untyped λ-calculus is the solution to the equation $D \approx D \to D$, where $D \to D$ is a *function space*.

As a small example, we will present a denotational semantics for the simply typed λ-calculus, $\lambda^{\to}$. Take syntax for the simply typed lambda calculus to be the previously defined set Λ. Define by induction a set of types τ, with base case ι, and inductive case $\tau_1 \to \tau_2$. A context Γ is a set of expressions of the form $x : \sigma$, with x a variable and $sigma$ a type. The *typing judgments* for $\lambda^{\to}$ are:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \quad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau}$$

where $\Gamma \vdash t : \sigma$ stands for "t has type σ with context Γ". A term t is well typed iff there exists a type σ and context Γ such that $\Gamma \vdash t : \sigma$.

We use the semantics from Plotkin [Plotkin, 1980]. Consider a fixed non-empty set $\mathcal{D}$. We define the *full type hierarchy* as follows. $\mathcal{D}_\iota = \mathcal{D}$, and $\mathcal{D}_{\sigma \to \tau} = (\mathcal{D}_\sigma \to \mathcal{D}_\tau)$ the set of all functions from $\mathcal{D}_\sigma$ to $\mathcal{D}_\tau$. An environment ρ is a σ-indexed function from variables $x : \sigma$ to elements of $\mathcal{D}_\sigma$. We write $\rho[x \mapsto d]$, for $x : \sigma$ and $d \in \mathcal{D}_\sigma$ for the new environment that maps x to d. We write $\{\}$ for the empty environment.

Then, for a closed well-typed term $t : \sigma$ of $\lambda^{\to}$, we define its interpretation $[\![t]\!]_{\{\}} \in \mathcal{D}_\sigma$:

$$\begin{array}{lcl} [\![x]\!]_\rho & = & \rho x \\ [\![(M\ N)]\!]_\rho & = & [\![M]\!]_\rho([\![N]\!]_\rho) \\ [\![\lambda x.M]\!]_\rho(d) & = & [\![M]\!]_{\rho[x \mapsto d]} \end{array}$$

the function $[\![]\!]$ assigns to each well-typed closed lambda term $t : \sigma$ an element on the domain $\mathcal{D}_\sigma$.

As it is the case with operational semantics, the term denotational semantics is used to encompass a wide variety of formalisms. As we will see, the denotational semantics of predicate in a logic program is the set of all the ground instances that are provable by resolution. This semantics, while called denotational, doesn't have very good properties, for instance is not compositional, take the program:

```
nat(o).
nat(s(X)) :- nat(X).
```

The union of the individual semantics of the clauses is not equivalent to their semantics taken together. In the first case, $[\![nat(o)]\!] = \{o\}$ and the semantics of the second clause is $\emptyset$, as this clause by itself cannot prove any formula using resolution. However, the semantics of both clauses together is the set $\{o, s(o), s(s(o)), \ldots\}$.

We may formalize some properties of interest about denotational semantics. Assume a set of programs $\mathcal{P}$, a domain $\mathcal{D}$, a relation of observational equivalence $\approx$: $(\mathcal{P} \times \mathcal{P})$, and an interpretation $[\![]\!] : \mathcal{P} \to \mathcal{D}$, then we have:

- Soundness, which checks that two terms which are observationally equivalent have the same denotation:

$$\forall p_1, p_2 \in \mathcal{P}.(p_1 \approx p_2 \Rightarrow [\![p_1]\!] = [\![p_2]\!])$$

- Full abstraction, when two programs have the same denotation they are observationally equivalent:

$$\forall p_1, p_2 \in \mathcal{P}.([\![p_1]\!] = [\![p_2]\!] \Rightarrow p_1 \approx p_2)$$

- Full completeness, when each element in the semantic domain has a program:

$$\forall d \in \mathcal{D}.\exists p \in \mathcal{P}.[\![p]\!] = d$$

Usually, full completeness results state that there exist a unique program for a given element.

2.4. Algebraic Semantics

Algebraic semantics are a special case of denotational semantics, where we interpret programs using *algebra*. That is to say, a set of terms, variables and operators with equivalences between them. Algebraic theories are usually presented as a set of equations.

Pure algebraic semantics for the λ-calculus are out of scope of this short introduction, we refer the interested reader to some related work by Salibra [Salibra, 2000] and Gabbay and Mathijssen [2010].

Fully algebraic semantics must capture all the meta-theory involved in a language. When that happens, the semantics can be seen as a middle ground between operational and denotational semantics. Orienting an equation readily gives a notion of computation, and mechanizing theorem proving with equational specifications is often straightforward, given that the transitivity and symmetry of equality are the only rules involved.

As an example of an algebraic specification we will use an example with a long tradition in computer science, algebraic data types. We may specify a list data type in algebraic style. Define operators:

$$\begin{array}{ll} empty & \rightarrow \mathsf{List} \\ cons(\mathsf{List}, \mathsf{Elem}) & \rightarrow \mathsf{List} \\ tail(\mathsf{List}) & \rightarrow \mathsf{List} \\ head(\mathsf{List}) & \rightarrow \mathsf{Elem} \end{array}$$

the following equations define the behavior of the operators:

$$\begin{array}{lcl} head(empty) & = & undefined \\ head(cons(l, e)) & = & e \\ tail(empty)) & = & undefined \\ tail(cons(l, e)) & = & l \end{array}$$

Note that our algebraic specification allows the occurrence of ill-constructed lists that are undefined. This particular example may be fixed by using sorted algebra. Define two sorts $\mathsf{NEList} < \mathsf{List}$, meaning that every List is also a NEList. Then the operators and rules are:

$$\begin{array}{ll} empty & \rightarrow \mathsf{List} \\ cons(\mathsf{List}, \mathsf{Elem}) & \rightarrow \mathsf{NEList} \\ tail(\mathsf{NEList}) & \rightarrow \mathsf{List} \\ head(\mathsf{NEList}) & \rightarrow \mathsf{Elem} \end{array}$$

$$\begin{array}{lcl} head(cons(l, e)) & = & e \\ tail(cons(l, e)) & = & l \end{array}$$

Using *sorts*, we have improved the semantics of our algebraically defined lists. A generalized notion of sorted algebra is provided by category theory. A category is defined by a collection of typed (or sorted) *arrows* between *objects*. Arrows can be composed if their sorts match, composition is associative and there is an identity arrow for every object.

The use of category theory for defining semantics of programming languages is a prominent case of algebraic semantics. Categories are a generalization of algebraic concepts, often highlighting the connections among algebra, logic and topology.

For instance, we may interpret simply typed λ-calculus in the so-called cartesian closed categories. Assuming the types are correct, the theory of a cartesian closed category may be equationally presented

as follows:

$$
\begin{array}{rcl}
(f \circ g) \circ h & = & f \circ (g \circ h) \\
Id \circ f & = & f \\
f \circ Id & = & f \\
Term \circ f & = & Term \\
Fst \circ \langle f, g \rangle & = & f \\
Snd \circ \langle f, g \rangle & = & g \\
\langle Fst \circ h, Snd \circ h \rangle & = & h \\
f \times g & = & \langle f \circ Fst, g \circ Snd \rangle \\
App \circ (\Lambda(f) \times f) & = & f \\
\Lambda(App \circ (f \times Id) & = & f
\end{array}
$$

The reader can observe how the CCC theory includes models such as the category Set of sets and functions. At the same time, this equational theory has served as the base for the implementation of λ-calculus interpreters and functional programming languages [Cousineau et al., 1987].

CHAPTER 3

Constraint Logic Programming

Constraint Logic Programming is programming with predicates in a certain fragment of logic, with the help of external domain-specific logic engines called constraint solvers. More broadly speaking, it is programming with executable specifications: code that has independent mathematical meaning consistent with its input-output behavior. Specifications are often formalized as relations.

Logic Programming is made possible thanks to *efficient* theorem proving methods. A program is viewed as a logical theory, then the system is fed with a formula or query, trying to prove it is logical consequence of the program. If there exists a proof, a summary of the proof itself is considered the answer.

There are almost as many different LP variants as there are logical formal systems. In the rest of this work, we focus in Logic Programming with first order Horn clauses and constraints. In this setting, a program is a set of Horn clauses with exactly one consequence and a query is an existentially quantified conjunction. The information returned by an answer is the witnesses for the existentially quantified variables.

3.1. Basic Syntax

Assume a permutative convention on symbols, i.e., unless otherwise stated explicitly, distinct names f, g stand for different entities (e.g. function symbols) and the same with distinct names i, j, for indices. A first-order language consists of a signature $\Sigma = \langle \mathcal{C}_\Sigma, \mathcal{F}_\Sigma \rangle$, given by $\mathcal{C}_\Sigma$, the set of constant symbols, and $\mathcal{F}_\Sigma$, the set of term formers or function symbols. We usually write $f, g, \ldots$ for elements of $\mathcal{F}_\Sigma$ and a, b for elements of $\mathcal{C}_\Sigma$. $\mathcal{P}$ denotes a set of predicate symbols. We usually write $p_i, p_j, q_i, q_j, \ldots$ for elements of $\mathcal{P}$. Function $\alpha : \mathcal{P} \cup \mathcal{F}_\Sigma \to \mathbb{N}$ returns the arity of its predicate argument. We assume a set $\mathcal{X}$ of so-called *logic variables* whose members are denoted $x_i, y_i, z_i, \ldots$.

We write $\mathcal{T}_\Sigma$ for the set of closed terms over Σ. We write $\mathcal{T}_\Sigma(\mathcal{X})$ for the set of open terms (in the variables in $\mathcal{X}$) over Σ. We drop Σ when understood from context. We write t, u, v for terms of $\mathcal{T}_\Sigma$. We write sequences of variables, terms and predicates using vector notation: $\vec{x} = x_1, \ldots, x_m$, $\vec{t} = t_1, \ldots, t_n$, $\vec{p} = p_1, \ldots, p_k$. The length of a sequence is written $|\cdot|$, thus $|\vec{x}| = m$, $|\vec{t}| = n$ and $|\vec{p}| = k$. $t[\vec{x}]$ denotes a term t from $\mathcal{T}_\Sigma(\mathcal{X})$ using variables in $\vec{x}$, and $\vec{t}[\vec{x}]$ denotes a sequence of terms using variables in $\vec{x}$.

We proceed to define standard concepts of atoms, predicates, programs and clauses. The reader may check [Lloyd, 1984] for more information.

Definition 3.1 (Atom)**.** *Given a predicate symbol $p \in \mathcal{P}$, with arity $\alpha(p) = n$ and terms $t_1, \ldots, t_n \in \mathcal{T}_\Sigma(\mathcal{X})$, an atom is an expression of the form $p(t_1, \ldots, t_n)$. We say an atom is* pure *if it only contains variables as arguments. This notion is naturally extended to compound formulas.*

Thus, $p(\vec{t}[\vec{x}]))$ denotes an atom composed of a predicate p and arguments $t_1[\vec{x}], \ldots, t_n[\vec{x}]$. $\vec{p}(\vec{t}[\vec{x}])$ denotes a sequence of atoms $p_1(\vec{t_1}[\vec{x}]), \ldots, p_n(\vec{t_n}[\vec{x}])$. We may drop $[\vec{x}]$ or even $\vec{t}[\vec{x}]$ when the context allows and just write $\vec{p}$ for $\vec{p}(\vec{t}[\vec{x}])$. Given a sequence $\vec{p}$ of n terms, variables or atoms, we write $\vec{p}_{|2}$ for the sequence of $n-1$ elements starting with the second one, that is to say, $\vec{p}_{|2} = p_2, \ldots, p_n$.

Definition 3.2 (Horn Clause)**.** *Given atoms $p(\vec{t}), q_1(\vec{u}_1), \ldots, q_n(\vec{u}_n)$, with $n \geq 0$, we define a Horn Clause as a named expression of the following form:*

$$cl : p(\vec{t}) \leftarrow q_1(\vec{u}_1), \ldots, q_n(\vec{u}_n)$$

The atom $p(\vec{t})$ before the $\leftarrow$ symbol is called the head *of the clause and the sequence of atoms to its right, the* tail*. Horn Clauses with an empty tail are called* facts*.*

A Horn Clause is logically interpreted as an implication from right to left. Operationally, given an atom $p(\vec{t})$ in a query, we will try to match it to the head of some clause. If we find a match, then we proceed to prove all the atoms in its tail.

Definition 3.3 (Logic Program)**.** *A Logic Program PL is a finite set of Horn clauses.*

For convenience, we formally define renaming apart:

Definition 3.4 (Renaming Apart)**.** *Given two sequences of variables $\vec{x}$, $\vec{y}$, we say they are renamed apart if there is no i, j such that $x_i \in \vec{x}$, $y_j \in \vec{y}$ and $x_i \equiv y_j$. That is to say, the names are all different.*

Let σ be a renaming over a set of variables, $\sigma\mathcal{X} \to \mathcal{X}$. Given $\vec{x}$ and $\vec{y}$, we say that σ is a renaming apart for $\vec{x}$ from $\vec{y}$ if $\sigma(\vec{x})$ and $\vec{y}$ are renamed apart.

3.2. Constraints

We can enrich Def. 3.3 if we allow a broader class of formulas to occur in Horn Clauses instead of just atoms. Constraint Logic Programming is the enrichment that occurs when using *constraint formulas*, a class of conjunction and existential quantification close formulas to be checked for satisfiability with an external tool. For all purposes, the external tool or *constraint solver* is an opaque entity, which we will interact with using a *tell* operation, where we post a new formula and the solver will return true it is satisfiable or fail otherwise. We also assume the ability to manipulate and construct constraints. We base our presentation on standard material from [Jaffar and Maher, 1994, Saraswat, 1992].

We first define $\Sigma_{\mathcal{D}}$-structures, that is to say, the connection of syntax of the constraint formulas to the constraint domain itself. The use of an external constraint solver means that we lack any inference rules for constraint formulas, so $\Sigma_{\mathcal{D}}$-structures constitute full descriptions for constraint domains.

Definition 3.5 ($\Sigma_{\mathcal{D}}$-structure)**.** *Let $\Sigma_{\mathcal{D}} \subseteq \Sigma$ be a subset of the signature of the program, and $\mathcal{CP}_{\mathcal{D}} \subseteq \mathcal{P}$ a subset of the predicate symbols of the program. A $\Sigma_{\mathcal{D}}$-structure $\mathcal{D}$ consists of a set D and an assignment for elements of $\Sigma_{\mathcal{D}}$, and predicate symbols of $\mathcal{CP}_{\mathcal{D}}$ in D in the usual way: constant symbols are mapped to individuals in D, function symbols of arity n to n-ary functions on D and predicate symbols of arity n to n-ary relations on D. We write $f^{\mathcal{D}}$, $p^{\mathcal{D}}$ for the D-interpretation of function and predicate symbols in D. The equality predicate $=$ — whose standard meaning is domain equality — is required in $\mathcal{CP}_{\mathcal{D}}$.*

Example 3.6. *A standard example is the $\Sigma_{\mathbb{N}}$-structure given by the set $\mathbb{N}$, constant o interpreted as 0, unary function symbol s interpreted as $x \mapsto x + 1$, binary function symbol $+$ interpreted as addition and predicate symbols $<$, $>$, interpreted as the less than and greater than relations.*

We may have a family of structures. Let $\mathfrak{D}$ be an indexed set of $\Sigma_{\mathcal{D}}$-structures. We write $\mathcal{D}_i$ for an element of $\mathfrak{D}$, and we require the sets $\mathcal{CP}_{\mathcal{D}_i} \subseteq \mathcal{P}$ to be disjoint in order to not to require an overloading mechanism.

Definition 3.7 (Constraint Formulas)**.** *The set $\mathcal{L}_{\mathcal{D}}$ of constraint formulas built from $(\Sigma_{\mathcal{D}}, \mathcal{CP}_{\mathcal{D}})$ is inductively defined as:*

- *For every $p \in \mathcal{CP}_{\mathcal{D}}$ of arity n and $t_1, \ldots, t_n$ in $\mathcal{T}_{\Sigma_{\mathcal{D}}}(\mathcal{X})$, $p(t_1, \ldots, t_n) \in \mathcal{L}_{\mathcal{D}}$. It is called an* atomic *constraint formula.*
- *If $\varphi, \psi \in \mathcal{L}_{\mathcal{D}}$, then $\varphi \wedge \psi \in \mathcal{L}_{\mathcal{D}}$.*
- *If $\varphi \in \mathcal{L}_{\mathcal{D}}$, then $\exists x.\varphi \in \mathcal{L}_{\mathcal{D}}$.*

We call nonatomic constraint formulas compound formulas.

$\mathcal{L}_\mathcal{D}$ is closed under conjunction and existential quantification and it is easily seen closed under substitution. We commonly call an element of the set $\mathcal{L}_\mathcal{D}$ a *constraint*. We write φ, ψ, ϕ for constraints.

Remark 3.8. *We force $\mathcal{L}_\mathcal{D}$ to be built using terms from $\mathcal{T}_\Sigma(\mathcal{X})$ in order to follow standard practice in constraint logic programming, where the arguments of any constraint formula can be interpreted and manipulated as a normal term. However, the standard equality predicate may not correspond to equality in the constraint domain. A typical example is the use of the `.=.` predicate to denote equality in finite domain solvers. To cite another example in the domain of the natural numbers `3 + 4 .=. 7` is true but the term equation `3 + 4 = 7` may fail since $=$ represents term equality and `3+4` is a compound term formed with $+ \in \mathcal{F}$, and $3, 4 \in \mathcal{C}$ and hence, not identical to 7.*

A common case is when $\mathcal{D}$ is $\mathbb{R}$ or $\mathbb{N}$ with the usual representation of numbers.

Definition 3.9 (Constraint Domain)**.** *A constraint domain is given by a pair $(\mathcal{D}, \mathcal{L}_\mathcal{D})$ consisting of a $\Sigma_\mathcal{D}$-structure and the set of constraint formulas.*

Example 3.10 (Herbrand Constraint Domain)**.** *Suppose $\mathcal{H}$ is the Herbrand universe for a signature Σ, that is to say, the free $\Sigma_\mathcal{H}$-structure, and let $\mathcal{L}_\mathcal{H}$ stand for all open formulas over the signature of $\mathcal{H}$ built up from atoms using conjunction, existential quantification and equality of terms using variables from $\mathcal{X}$, the set of so-called logic variables. Then the Herbrand Constraint Domain is the pair $(\mathcal{H}, \mathcal{L}_\mathcal{H})$.*

A $\Sigma_\mathcal{D}$ structure induces a mapping or interpretation from elements of $\mathcal{L}_\mathcal{D}$ to the lattice of truth values $\bot, \top$.

Definition 3.11 (Constraint Interpretation)**.** *The interpretation function for closed formulas $[\![\cdot]\!]^\mathcal{D} : \mathcal{L}_\mathcal{D} \to \{\bot, \top\}$ is defined by induction on the structure of the formulas:*

$$\begin{array}{lcl} [\![p(t_1, \ldots, t_n)]\!]^\mathcal{D} & = & \begin{cases} \bot & \textit{if } (t_1^\mathcal{D}, \ldots, t_n^\mathcal{D}) \notin p^\mathcal{D} \\ \top & \textit{if } (t_1^\mathcal{D}, \ldots, t_n^\mathcal{D}) \in p^\mathcal{D} \end{cases} \\ [\![\varphi \wedge \psi]\!]^\mathcal{D} & = & min([\![\varphi]\!]^\mathcal{D}, [\![\psi]\!]^\mathcal{D}) \\ [\![\exists x.\varphi]\!]^\mathcal{D} & = & max([\![\varphi[a/x]]\!]^\mathcal{D} \mid a \in D) \end{array}$$

Definition 3.12 (Constraint Satisfaction)**.** *A closed constraint formula φ is satisfied in a domain $\mathcal{D}$ or $\mathcal{D} \models \varphi$ if $[\![\varphi]\!]^\mathcal{D} = \top$.*

So far, we have defined interpretation for closed constraint formulas, now we extend the notion to constraints with free variables.

Definition 3.13 (Consistency of a Constraint)**.** *Let $\mathcal{D}$ be a constraint domain. We say an open constraint c with variables $\vec{x}$ is consistent or satisfiable if $\mathcal{D} \models \exists\, \vec{x}.c$. e.g. $x^2 = 2$ is satisfiable in $\mathbb{R}$ but $x^2 = -1$ is not.*

The reader should note that it is common practice in CLP to identify the *satisfiability* of a constraint c, usually written $\mathcal{D} \models c$ with *consistency* $\mathcal{D} \models \exists\, \vec{x}.c$. We avoid this abuse of formal notation but will use the words consistency and satisfiability of constraints interchangeably. The problem of determining when a constraint is satisfiable is solved by a query to the constraint solver.

A constraint system is equipped with a formal entailment relation which must meet several requirements.

Definition 3.14 (Constraint System, Saraswat [1992])**.** *A Constraint System is a pair $(\mathcal{L}_\mathcal{D}, \vdash)$:*

(1) *$\mathcal{L}_\mathcal{D}$ is a set of constraint formulas.*

(2) $\vdash \subseteq Fin(\mathcal{L}_\mathcal{D}) \times \mathcal{L}_\mathcal{D}$ *satisfies the following inference rules:*

$$\frac{}{\Gamma, d \vdash d} \qquad \frac{\Gamma_1 \vdash d \quad \Gamma_2, d \vdash e}{\Gamma_1, \Gamma_2 \vdash e}$$

$$\frac{\Gamma, d, e \vdash f}{\Gamma, d \wedge e \vdash f} \qquad \frac{\Gamma_1 \vdash d \quad \Gamma_2, \vdash e}{\Gamma_1, \Gamma_2 \vdash d \wedge e}$$

$$\frac{\Gamma, d \vdash e}{\Gamma, \exists X.d \vdash e} \qquad \frac{\Gamma \vdash d[t/X]}{\Gamma \vdash \exists X.d}$$

Constraint solvers must be sound, that is to say, every deduction using $\vdash$ must be correct respect to $\models$. The opposite notion, *completeness* is convenient but will not always hold. An incomplete solver may allow some derivations to happen where a constraint cannot be satisfied. Such an state is logically inconsistent. Analyzing the compromise of using such constraint solvers is beyond the scope of this work, but we will prove that when using a incomplete constraint solver our system gives equivalent answers to those that would be given by a classical CLP system.

Definition 3.15 (Constraint Store)**.** *The notion of* constraint store *refers to the internal state of a constraint solver. For our purposes, a constraint store is a constraint formula.*

We now extended the notion of Logic Program to the notion of Constraint Logic Program.

Definition 3.16 (Horn Clause with Constraints)**.** *Given atoms $p(\vec{t})$, and a sequence $A_1, \ldots, A_n$, with $n \geq 0$, where A_i is an atom ($A_i \equiv q_i(\vec{u}_i)$) or a constraint ($A_i \equiv \varphi_i$), we define a Horn Clause with Constraints as a named expression of the following form:*

$$cl : p(\vec{t}) \leftarrow A_1, \ldots, A_n$$

Abusing notation, we use $\vec{p}$ (and $\vec{q}$, etc...) for denoting sequences $A_1, \ldots, A_n$ where A_i may be a constraint or an atom $p_i(\vec{t}[\vec{x}])$.

Definition 3.17 (Constraint Logic Program)**.** *A Constraint Logic Program PL is a finite set of Horn Clauses with Constraints.*

3.3. Examples

A basic example is the definition of peano addition:

```
add(o, X, X).
add(s(X), Y, s(Z) :- add(X, Y, Z).
```

The first clause states that $o + X = X$ for all instances of X. The second clause states that if $Z = X + Y$, then $s(Z) = s(X) + Y$. We may query the add predicate with different instantiations of their arguments. For instance, if we query it will all its variables free, we obtain an infinite set of answers, procedurally generated:

```
?- add(X,Y,Z).

X = o,
Z = Y ? ;

X = s(o),
Z = s(Y) ? ;

X = s(s(o)),
```

```
queens(N, L, Lab) :-
        length(L, N),     % L has length N.
        domain(L, 1, N),
        safe(L),
        labeling(Lab, L).

safe([]).
safe([X|L]) :-
        noattack(L, X, 1),
        safe(L).

noattack([], _, _).
noattack([Y|L], X, I) :-
        diff(X, Y, I),
        I1 is I + 1,
        noattack(L, X, I1).

diff(X,Y,I):-
        X   #\= Y,
        X   #\= Y+I,
        X+I #\= Y.
```

FIGURE 3.1. Code for the Queens Problem using Prolog(CLPFD)

```
Z = s(s(Y)) ?
```

We must note that while add will return an answer in most cases, it is not complete, for instance, the following query will hang the Prolog interpreter.

```
add(s(X),s(Y),s(Y)).
```

Thus, while a highly declarative paradigm, standard logic programming doesn't fully abstract the programmer from operational issues.

We have stated that using an external constraint solver provides significant advantages, mainly in efficiency and modularity of code. A good example is the program of Fig. 3.1, that uses a constraint solver over the natural numbers to succinctly solve the N-Queens problem.

In this case, the predicates primitive to the constraint solver are

- `domain(L, M, N)`: It states that all the elements l_i of the list `L` of numbers (or free variables) are between `M` and `N`, that is to say, $m \leq l_i \leq n$.
- `labeling(S, L)`: This is a *control* predicate, and is true if the list `L` ground, that is to say, composed of numbers.
- `#=`: This is the $\neq$ relation for natural numbers.

The list of variables `L` of length `N`, is used to represent the positions of the queens. For instance, a list $[1, 3, 2]$ means that in column 1, the queen is in row 1, in column 2 the queen is in row 3, etc…

Then, we impose the following conditions on the numbers. All the elements of the list must be different, as having two queens in the same row means that they are not safe. We also impose the constraint that no queens are in diagonal, that for any positions X_i, X_j, with $i < j$ this is achieved by imposing $X_i \neq X_j + j - i$ and $X_i \neq X_j - j - i$. This particular example runs very fast and it is very difficult to beat with a hand-programmed solution. Note that, as is standard in logic programming, the

predicate `queens` may be used to find a solution or it may be called to *check* if a given configuration is a solution of the N-Queens problem.

3.4. Denotational semantics

The classical denotational semantics for a logic program is its set of ground theorems. To give a constructive description of this set in term of a sequence of approximations, the standard Apt-van Emden-Kowalski interpretation operator may be used, see [Lloyd, 1984]. In order to understand how this operator works, we define the logical interpretation of a Horn Clause:

Definition 3.18 (Logical Interpretation of Horn Clauses)**.** *The logical interpretation of a Horn Clause:*

$$p(\vec{t}) \leftarrow q_1(\vec{u}_1), \ldots, q_n(\vec{u}_n)$$

is the universal closure of the following formula

$$\exists \vec{x}.(q_1(\vec{u}_1) \wedge \ldots \wedge q_n(\vec{u}_n)) \rightarrow p(\vec{t})$$

where $\vec{x}$ are the variables free in the tail not occurring in the head.

As such, the logical interpretation of our program is a theory. The denotational semantics is the set of all the ground atoms that can be proved true by resolution. The *Herbrand Base* of a logic program is the set of all atoms that can be built using Σ and $\mathcal{P}$. Then, an interpretation I is a subset of the Herbrand base and it its a *Hebrand Model* of a logic program P iff for every ground instance of a clause $p \leftarrow \vec{q}$, when $\vec{q} \in I$ $p \in I$.

A model of a logic program P can be constructed using an operator T_P on interpretations, which is informally defined as:

$$\begin{array}{lcl} T_P^0 & = & \text{all ground instances of all the facts in } P \\ T_P^{n+1} & = & T_P^n \cup \{\text{all the ground instances of heads whose tails are in } T_P^n\} \end{array}$$

T_P is monotone and it has a *least fixed point* which corresponds with the minimal model of P. The use of constraints complicates this structure. Our interpretations are no longer sets of ground atoms, but they must have an attached constraint carrying logical information.

Definition 3.19. *The constrained base S_P is the set of pairs $\langle p(\vec{x}), \varphi[\vec{x}]\rangle$, where $p(\vec{x})$ ranges over all the pure atoms that can be formed from predicate symbols in the original set of clauses and $\varphi[\vec{x}]$ over every possible constraint formula in the program with free variables $\vec{x}$.*

Assume all clauses in P to be in *general form* (see Def. 6.10), that is to say, every atom is pure and all the constraints are grouped in the front, so clauses are of the form $cl : p(\vec{x}) \leftarrow \varphi, \vec{q}(\vec{y})$, then:

Definition 3.20. *Define $T_P :: Pow(S_P) \rightarrow Pow(S_P)$ as follows.*

$$T_P(S) = S \cup S'$$

where S' is generated as follows. For all clauses $cl : p(\vec{x}) \leftarrow \varphi[\vec{x}, \vec{y}], \vec{q}(\vec{y})$ in P, if $\langle \vec{q}_i(\vec{z}_i), \varphi_i[\vec{z}_i]\rangle \in S$ for all $i \in \{1, n\}$, with $\vec{z}$ renamed apart from $\vec{x}, \vec{y}$, define $\psi = \exists \vec{y}.(\varphi[\vec{x}\vec{y}] \wedge (\bigwedge_{i=1}^{n}(q_i(\vec{z}_i) = q_i(\vec{y}_i) \wedge \varphi_i[\vec{z}_i])))$. Then, $\langle p(\vec{x}), \psi\rangle$ is added to S' iff $\mathcal{D} \models \psi$.

Each iteration uses all the available clauses in order to get enlarge the set of provable facts. For this to happen, each element of the tail of a Horn Clause with constraints must be already in the interpretation.

Lemma 3.21. *T_P is monotone.*

PROOF. Immediate. For all B, $A \subseteq A \cup B$. □

Define ${T_P}^1$ to be T_P, ${T_P}^{n+1}$ to be $T_P \circ {T_P}^n$ and ${T_P}^\omega$ to be the pointwise union of all the ${T_P}^n$. Then we have the following theorem.

Theorem 3.22. *T_P has a least fix-point $T_P{}^{\omega}(\emptyset)$, which is the least Herbrand model of the program P.*

Proof. By the Tarski-Knaster theorem. □

For more information, the reader may consult [Fitting, 2002, Jaffar and Maher, 1994].
The operational semantics presented in Chapter 6 uses relation rewriting to implement T_P and $T_P{}^{\omega}$.

3.5. Operational Semantics

We follow Jaffar and Maher [1994]: a *resolution computation with constraints* is a set of derivations induced by a transition system over program states. Program states are sequences consisting of atoms or constraints and an accumulated constraint formula which serves as *constraint store*. That is to say, the constraint store will hold all the knowledge "acquired" during the execution of the program.

Definition 3.23 (Query)**.** *A query Q is a sequence $A_1, \ldots, A_n$, with $n \geq 1$, where A_i is an atom ($A_i \equiv q_i(\vec{u}_i)$) or a constraint ($A_i \equiv \varphi_i$), logically interpreted as $\exists \vec{x}.(A_1 \wedge \ldots \wedge A_n)$, where $\vec{x}$ is the set of all free variables in Q.*

We say that a query Q with free variables $\vec{x}$ succeeds against a program P, when $P \vdash \exists \vec{x}\, Q$ for the provability relation $\vdash$ induced by resolution, that is to say, when a resolution proof exists. As we will see below, such a proof not only contains a witness for $\vec{x}$, but a constraint over them allowing richer answers than plain substitution.

Definition 3.24 (Program State)**.** *A program state is a sequence $\langle A_1, \ldots, A_n \mid \varphi \rangle$, where the A_i are atoms or constraints and φ is a constraint formula. We call $A_1, \ldots, A_n$ the* resolvent *and φ the* constraint store*. We write $\Box$ for an empty resolvent. For any φ, $\langle \Box \mid \varphi \rangle$ is an* empty state.

We focus on "ideal" constraint logic programming systems. In such systems, whenever a new constraint is found, it is immediately "posted", which amounts to adding the constraint to the store and checking the satisfiability of the resulting formula, as shown in Def. 3.25. This implies that the constraint store of every program state found in the execution of a query will be tacitly assumed satisfiable. Note that some solvers are "not complete" or semi sound, that is to say, for some formulas they will allow inconsistent constraint stores until an special predicate that checks for satisfiability is called.

Proof search in an ideal CLP system is performed by two transitions, depending on whether the next element of the resolvent is a constraint or a defined predicate:

Definition 3.25. *A constraint step for a state $\langle \varphi, \ldots, p_n(\vec{u}_n) \mid \psi \rangle$, where φ is a constraint is:*

$$\langle \varphi, \vec{p} \mid \psi \rangle \quad \rightarrow_c \quad \langle \vec{p} \mid \psi \wedge \varphi \rangle$$

iff $\psi \wedge \varphi$ is satisfiable, that is to say $\mathcal{D} \models \exists \vec{x}.\ \psi \wedge \varphi$ for all variables $\vec{x}$ free in $\psi \wedge \varphi$.

In the above step the appropriate constraint solvers are called in order to determine the satisfiability of the constraint store $\psi \wedge \varphi$. Standard practice in CLP is to select the constraint solver by requiring constraint predicates to be disjoint among solvers. Detailed operational semantics which are generic for a family of constraint solvers can be found in [Gallego Arias et al., 2007]; a similar solution would work here but we think making explicit the fact that we may have several constraint solvers is not worth the increased complexity of notation.

Definition 3.26. *A resolution step for a state $\langle p(\vec{t}[\vec{x}]), \vec{p} \mid \varphi \rangle$ using clause $cl : p(\vec{u}[\vec{y}]) \leftarrow \vec{q}(\vec{v}[\vec{z}])$ is:*

$$\langle p(\vec{t}[\vec{x}]), \vec{p} \mid \varphi \rangle \quad \xrightarrow{cl}_r \quad \langle \vec{q}(\vec{v}[\sigma(\vec{z})]), \vec{p} \mid \varphi \wedge (\vec{u}[\sigma(\vec{y})] = \vec{t}[\vec{x}]) \rangle$$

iff $\varphi \wedge (\vec{u}[\sigma(\vec{y})] = \vec{t}[\vec{x}])$ is satisfiable and σ is a renaming apart of $\vec{y}$ and $\vec{z}$ from $\vec{x}$.

Note that the presence of a Herbrand constraint solver is necessary in order to implement the resolution step.

Definition 3.27. *A transition for a program state $Q \to Q'$ is either a constraint step $Q{\to}_c Q'$ or a resolution step $Q \xrightarrow{cl}_r Q'$. We say that a state Q has a derivation to Q' iff there's a sequence of transitions $Q \to Q_1 \to \cdots \to Q'$.*

Definition 3.28. *A state $\langle \vec{p} \,|\, \varphi \rangle$ is said to* succeed *iff it has a derivation that leads to an empty ($\langle \square \,|\, \psi' \rangle$) state.*

Definition 3.29. *A state $\langle \vec{p} \,|\, \varphi \rangle$ is final iff no derivation exists starting from this state. We represent this state of affairs by*

$$\langle \vec{p} \,|\, \varphi \rangle \not\to$$

if $\vec{p}$ is not the empty resolvent, we say that the state fails.

3.6. Impure features

The non-complete nature of the SLD resolution proof search strategy and some practical issues mean that the logic programmer is forced to use what we call *impure* features in some programs. While highly convenient, the use of impure predicates alter the semantics in a considerable way and greatly difficults program analysis.

We detail some of the most common impure features and discuss their impact on the declarative semantics.

Var predicate: : The `var` predicate is true if its argument is a free variable. While this predicate is very useful (mostly in conjunction with cut) to avoid unwanted instantiations, it breaks the logical semantics. Consider the program:

```
a(X) :- var(X).
a(3).
```

The query `a(X)` will complete with success and without instantiating X. This implies that $\forall X.a(X)$ is a consequence of the theory. However, the query `a(4)`, fails, so we cannot rely anymore on logical results involving free variables.

Cut predicate: : The ! (cut) predicate is used to control proof search. When found, it has the effect of discarding the current choice point. Let's study this program:

```
a :- b, !, c.
a :- d.

b :- true.

c :- fail.
d :- true.
```

From a logical point of view, a can be proved as $d \to a$ in the program. However, the cut in the first clause of a will impede backtracking and make the query `a` fail.

Assert predicate: : The `assert` predicate is used to add clauses to the program dynamically. Assert can be seen as a form of implication, given that it will augment the program in a similar way than the rule for logical implication does:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B.}$$

However, semantics for CLP doesn't contemplate the use of implication and the dynamic nature of assert leads again to weird logical behaviors. Even a bigger problem is formed by the predicate `retract`, which removes a clause from the program. There are interesting proposals to use Linear Logic to capture some of the features of retract.

Term predicate: : The `=..` predicate is used to create new terms in an ad-hoc way. Basically, what the predicate does is to decompose or create a term into its function part and argument part:

```
?- t(f) =.. M.
M = [t,f] ?

?- M =.. [t,f,a,b].
M = t(f,a,b) ?
```

While useful for meta-programming, the use of this predicate makes impossible to statically determine the universe of terms for the program, so program models may be incomplete.

3.7. Related Work

The body of work in Logic Programming is huge and there exists no complete account of all of it. Some introductory books for Prolog are [Sterling and Shapiro, 1986, O'Keefe, 1990]. For constraint and concurrent constraint programming, the reader may consult [Jaffar and Maher, 1994, Jaffar and Lassez, 1987, Marriott and Stuckey, 1998, Shapiro, 1989].

Most of the concepts presented here are included in standard books like [Lloyd, 1984]. Popular Prolog systems are SWI-Prolog, SICSTUS-Prolog and Ciao-Prolog.

Remarkable extensions to the Logic Programming paradigm are Higher-Order Logic Programming [Miller et al., 1991], Concurrent Constraint Logic Programming[Shapiro, 1989], Linear Logic programming [Hodas and Miller, 1994] and Answer Set programming [Niemelä, 1999].

CHAPTER 4

The Calculus of Binary Relations

The calculus of binary relations – also known as relation algebra – has diverse and important applications in computer science and mathematics. The composition, union and intersection operations provide a rich algebra in which many computing problems can be easily expressed [Bird and de Moor, 1996b].

Relation algebras may be seen as an algebraic formalization of set-theoretic binary relations. Sometimes, we describe a relation in terms of its intended set-theoretic semantics, that is to say, a set of ordered pairs. It may be convenient for the reader to think of some relations in terms of their semantics, however, all computation and formal reasoning with relational expressions is done directly in the relation algebra by inference from the equational theories to be presented below. The relation algebra formalism therefore should not be identified with its set-theoretic semantics.

4.1. Relation Syntax

The relational language R is defined by the following grammar:

$$\begin{array}{rcl} \mathsf{R}_{atom} & ::= & id \mid di \mid \mathbf{1} \mid \mathbf{0} \\ \mathsf{R} & ::= & \mathsf{R}_{atom} \mid \mathsf{R}^{\circ} \mid \mathsf{R} \cup \mathsf{R} \mid \mathsf{R} \cap \mathsf{R} \mid \mathsf{RR} \mid \mathsf{R} \setminus \mathsf{R} \end{array}$$

The constants $\mathbf{1}, \mathbf{0}, id, di$ respectively denote the universal relation (whose standard semantics is the set of all ordered pairs on a certain set), the empty relation, the identity (diagonal) relation, and identity's complement. Operators $\cup$ and $\cap$ represent union and intersection whereas juxtaposition RS represents relation composition and "$\setminus$" denotes difference or relative complement. For better readability we write sometimes $R; S$ for RS. We write xRy if x is related to y by R. A binary relation may be naturally understood as the set of pairs (x, y) such that xRy is true. Then, R° is the formal converse of R, i.e., $xRy \iff yR^{\circ}x$. We speak of the *domain* of a relation for the set $\{x|\exists y.xRy\}$ and its *codomain* is the domain of R°. Domain and codomain are can be expressed as relations (if we view a set S as the binary relation $\{(x, x) : x \in S\}$) as follows: $R; R^{\circ} \cap id$ and $R^{\circ}; R \cap id$. Note that xRy is not a term of our relational language, but it plays the role of an expression used with the intent of helping the reader to grasp the set-theoretic semantics of relations. As we will see, its formal interpretation is $xRy \iff (x, y) \in [\![R]\!]$.

4.2. The Equational Theory DRA

We specify relation algebras as an equational theory over terms of the language R. We write $R \subseteq S$ for $R \cap S = R$. The theory **DRA** of Distributive Relation Algebras is displayed in Fig. 4.1. The

$$
\begin{array}{ccc}
R \cap R = R & R \cap S = S \cap R & R \cap (S \cap T) = (R \cap S) \cap T \\
R \cup R = R & R \cup S = S \cup R & R \cup (S \cup T) = (R \cup S) \cup T
\end{array}
$$

$$R\,id = R \qquad R\mathbf{0} = \mathbf{0} \qquad \mathbf{0} \subseteq R \subseteq \mathbf{1}$$

$$R \cup (S \cap R) = R = (R \cup S) \cap R$$

$$R(S \cup T) = RS \cup RT \qquad (S \cup T)R = SR \cup TR$$

$$R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$$

$$(R \cup S)^\circ = R^\circ \cup S^\circ \qquad (R \cap S)^\circ = S^\circ \cap R^\circ$$

$$R^{\circ\circ} = R \qquad (RS)^\circ = S^\circ R^\circ$$

$$R(S \cap T) \subseteq RS \cap RT \qquad RS \cap T \subseteq (R \cap TS^\circ)S$$

$$id \cup di = \mathbf{1} \qquad id \cap di = \mathbf{0}$$

FIGURE 4.1. The Equational Theory **DRA**.

modular law $RS \cap T \subseteq (R \cap TS^\circ)S$ has provably equivalent left and right equational formulations: $T \cap RS = R(R^\circ T \cap S) \cap T$ and $T \cap RS = (TS^\circ \cap R)S \cap T$. Both are derivable from **DRA**.

4.3. Semantics

Fix a set $\mathcal{B}$, then we interpret a term of R as a set of pairs of elements of $\mathcal{B}$, that is to say, as a subset of $\mathcal{B} \times \mathcal{B}$.

Definition 4.1. *Given a set $\mathcal{B}$, a $\mathcal{B}$-**interpretation** is a mapping $[\![_]\!]_{\mathcal{B}} : \mathsf{R} \to Pow(\mathcal{B} \times \mathcal{B})$:*

$$
\begin{array}{lcl}
[\![id]\!]_{\mathcal{B}} & = & \{(u,u) \mid u \in \mathcal{B}\} \\
[\![di]\!]_{\mathcal{B}} & = & \{(u,v) \mid u \neq v, u, v \in \mathcal{B}\} \\
[\![R^\circ]\!]_{\mathcal{B}} & = & \{(u,v) \mid (v,u) \in [\![R]\!]_{\mathcal{B}}\} \\
[\![R \cup S]\!]_{\mathcal{B}} & = & [\![R]\!]_{\mathcal{B}} \cup [\![S]\!]_{\mathcal{B}} \\
[\![R \cap S]\!]_{\mathcal{B}} & = & [\![R]\!]_{\mathcal{B}} \cap [\![S]\!]_{\mathcal{B}} \\
[\![R \setminus S]\!]_{\mathcal{B}} & = & [\![R]\!]_{\mathcal{B}} \setminus [\![S]\!]_{\mathcal{B}} \\
[\![RS]\!]_{\mathcal{B}} & = & \{(x,y) \mid \exists v((x,v) \in [\![R]\!]_{\mathcal{B}} \wedge (v,y) \in [\![S]\!]_{\mathcal{B}})\} \\
[\![\mathbf{1}]\!]_{\mathcal{B}} & = & \mathcal{B} \times \mathcal{B} \\
[\![\mathbf{0}]\!]_{\mathcal{B}} & = & \emptyset
\end{array}
$$

The proof that equational reasoning in DRA is sound with respect to $\mathcal{B}$-interpretations is straightforward.

4.4. Properties of Binary Relations

We close the section by recalling a few elementary definitions from the theory of relations.

Definition 4.2. *A binary relation R is* **reflexive** *if $id \subseteq R$,* **coreflexive** *if $R \subseteq id$,* **functional** *if $R^\circ R$ is coreflexive, and* **injective** *if RR° is coreflexive.*

When stating that a certain relation expression is, e.g. injective or coreflexive, we mean that it is so when interpreted in the standard semantics. When we mean that it can be proved so in one of the equational theories defined in this paper, we will so indicate.

Coreflexive relations will play a critical role in this thesis, since they can be seen as a natural encoding of sets as binary relations.

Definition 4.3. *The formal diagonal, domain and range for a relation R are:*

$$
\begin{array}{rcl}
\Delta(R) & = & R \cap id \\
Dom(R) & = & R\mathbf{1} \cap id = RR^\circ \cap id \\
Ran(R) & = & R^\circ \mathbf{1} \cap id = R^\circ R \cap id
\end{array}
$$

4.5. Related Work

The calculus of binary relations was introduced by De Morgan in 1860, to be greatly developed by Peirce and Schröder. The reader is advised to read [Pratt, 1992] for more information on the historical development.

In [Dougherty and Gutiérrez, 2006], a notion of normal form for terms of the equational theory E_R of representable binary relations, expressing composition, converse, and lattice operations is defined. This extends the work of [Andréka and Bredikhin, 1995] with respect to the decidability of equational relational theories.

On the computer science side, several relation-based programming languages exist [Dwyer, 1994, Cattrall, 1992, Cattrall and Runciman, 1992, 1993, Hall, 1984]. In [Maddux, 1996] some introductory content is provided along with a full relational approach to program interpretation.

Of particular interest is the specification language Alloy [Jackson, 2002b,a]. At its core, the Alloy language is a simple but expressive logic based on the notion of relations, and was inspired by the Z specification language and Tarski's relational calculus.

There are many results about relational query languages and complexity. For instance, an interesting result is in [Vardi, 1982, Immerman, 1982] where it is proven that the language of Relational Calculus over a totally ordered finite domain plus least fixed points of monotone operators precisely expresses all queries computable in polynomial time — in the size of the domain — on a Turing machine.

CHAPTER 5

Logic and Relations

The foundation of the translation from logic to relations lies in the work of Tarski and Givant [1987], who observed that *variable-free* binary relations equipped with the so-called quasi-projection operators faithfully capture all of classical first-order logic. In their categorical formulation of the relation calculus Freyd and Scedrov [1991] show that so-called tabular allegories satisfying certain conditions faithfully capture all of higher-order intuitionistic logic. Both of these formalisms effectively eliminate logical variables in different ways.

The elimination of first-order variables in logic via a mathematical translation is of considerable interest, and a significant number of mathematical formalisms have been developed to achieve this aim over the past century. In computer science, and especially in the case of declarative programming languages, it provides a completely fresh approach to compilation, not unlike the translation of lambda calculus into combinators [Turner, 1979] or the program transformation performed by Warren's Abstract Machine [Ait-Kaci, 1991].

One of the main results due to Tarski, Maddux and Givant is the equipollence theorem, that states that every first-order sentence φ over a theory with a pairing operator has a semantically equivalent equational counterpart $X_\varphi = 1$ in the theory **QRA** of relation algebras with quasi-projections. Tarski and Givant also prove a stronger proof-theoretic version of this result, and exhibit a bijective recursive transformation of sentences φ to their associated relation expressions X_φ and of first-order proofs of the former to equational derivations of the latter.

We will use here a slightly different setting based on the work of Broome and Lipton [1994], which we call the theory $\mathbf{QRA}_\Sigma$ of distributive relation algebra with quasiprojections for a given signature Σ, usually of a logic program or a theory. $\mathbf{QRA}_\Sigma$ captures algebraically Clark's Equality Theory [Lloyd, 1984]. It should be noted that the absence of negation in $\mathbf{QRA}_\Sigma$ is no handicap, vis-a-vis first-order formulas with negation over the Herbrand Universe, because of the well-known results of Mal'cev [Mal'tsev, 1961, Maher, 1988] that any such formula is equivalent to a two-quantifier formula in which negation occurs only immediately preceding equations between terms. This can be modelled in $\mathbf{QRA}_\Sigma$ using di for disequality.

5.1. The Relational Language R_Σ

We extend the language R to support the interpretation of first-order sentences on a language $\mathcal{L}_\Sigma$, with signature Σ, equality, and arbitrary primitive predicate symbols $r_1, r_2, \ldots$ of different arities. Additionally, we incorporate a notion of relation variable and a capture-avoiding substitution which are used to define a fixpoint operator.

Fix a first-order signature $\Sigma = (\mathcal{C}_\Sigma, \mathcal{F}_\Sigma)$. The relation language R_Σ is built from a countable set of relation variables $\overline{x}, \overline{y}, \overline{z}$, etc, all in R_{var} (not to be confused with first-order logic-variables), a set of relation constants for terms $\mathsf{R}_\mathcal{T}$ built from Σ, a set of relation constants for primitive predicates $\mathsf{R}_{\mathcal{CP}}$, and a fixed set of relation constants and operators detailed below. Let us begin with $\mathsf{R}_\mathcal{T}$. Each constant $a \in \mathcal{C}_\Sigma$ defines a constant $(a, a) \in \mathsf{R}_\mathcal{T}$, each function symbol $f \in \mathcal{F}_\Sigma$ defines a constant R_f in $\mathsf{R}_\mathcal{T}$. Each predicate $r \in \mathcal{CP}$ defines a constant r in $\mathsf{R}_{\mathcal{CP}}$. Formally:

$$\mathsf{R}_{\mathcal{CP}} \;=\; \{\mathsf{r} \mid r \in \mathcal{CP}\} \quad \mathsf{R}_{\mathcal{T}} \;=\; \{\mathsf{R}_f \mid f \in \mathcal{F}_\Sigma, \} \cup \{(a,a) \mid a \in \mathcal{C}_\Sigma\}$$

The full relation language R_Σ is given by the following grammar:

$$\begin{array}{rcl} \mathsf{R}_{atom} & ::= & \mathsf{R}_{var} \mid \mathsf{R}_{\mathcal{T}} \mid \mathsf{R}_{\mathcal{CP}} \mid id \mid di \mid \mathbf{1} \mid \mathbf{0} \mid hd \mid tl \\ \mathsf{R}_\Sigma & ::= & \mathsf{R}_{atom} \mid \mathsf{R}_\Sigma{}^\circ \mid \mathsf{R}_\Sigma \cup \mathsf{R}_\Sigma \mid \mathsf{R}_\Sigma \cap \mathsf{R}_\Sigma \mid \mathsf{R}_\Sigma \mathsf{R}_\Sigma \mid \mathsf{R}_\Sigma \setminus \mathsf{R}_\Sigma \mid \mathbf{fp}\, \mathsf{R}_{var} \,.\, \mathsf{R}_\Sigma \end{array}$$

The constants $\mathbf{1}, \mathbf{0}, id, di$ respectively denote the universal relation (whose standard semantics is the set of all ordered pairs on a certain set), the empty relation, the identity (diagonal) relation, and identity's complement. Operators $\cup$ and $\cap$ represent union and intersection whereas juxtaposition RR represents relation composition and "$\setminus$" denotes difference or relative complement. For better readability we write sometimes $R; S$ for RS. R° is the formal converse of R, i.e., the relation it denotes is the one obtained by reversing the ordered pairs in the relation denoted by R. In the set-theoretic semantics, the domain of the set denoted by R is $\{x | (x, _) \in R\}$ and its codomain is $\{x | (_, x) \in R\}$. **fp** represents a standard fixpoint operator.

5.2. The Equational Theory $\mathbf{QRA}_\Sigma$

We start from **DRA**, the theory of distribute relation algebras, and add quasiprojections to obtain **QRA**, which is a subtheory of the theory used by Tarksi and Givant. The complementation operator will not be used. We augment **QRA** to $\mathbf{QRA}_\Sigma$ to include a fixpoint operator and some rules for the elements of $\mathsf{R}_{\mathcal{T}}$. Fig. 5.1 contains the full $\mathbf{QRA}_\Sigma$ theory.

Such a relational theory is close in spirit to the positive fragment of the untyped μ-relation calculus $\mathcal{M}U$ introduced by deRoever (see e.g. [de Roever, 1974, de Bakker and de Roever, 1972, Frias and Maddux, 1998]).

The role of $\mathbf{QRA}_\Sigma$ will be clearer in subsequent chapters. We will derive a proof search strategy and unification algorithm just by orienting some of the equations into an orthogonal rewrite system. This allows us to claim correctness almost for free.

5.2.1. Projections. Tarski showed how to axiomatize equationally the notion of products and projections of the first and second coordinate in a relational variable-free manner. hd and tl represent the head and tail relations over arbitrary sequences and are equationally axiomatized as follows:

$$hd(hd)^\circ \cap tl(tl)^\circ \subseteq id \qquad (hd)^\circ hd = (tl)^\circ tl = id \qquad (hd)^\circ tl = \mathbf{1}$$

The first condition formalizes the fact that pairs are uniquely determined by their first and second coordinates, the second, that hd and tl are functional relations (equivalently, their converses are injective), and the third, that any two elements can occur as the head or the tail of a pair. We define helper relations P_i for the i-th projection:

Definition 5.1. *Define the countable sequence $\{P_i \mid 1 \leq i\}$ of projection relations as follows:*

$$P_1 \;=\; hd \qquad P_2 \;=\; tl; hd \qquad \cdots \qquad P_n \;=\; tl^{n-1}; hd \qquad \ldots$$

In the standard semantics to be discussed below, P_i is a relation between a formal vector with at least $i+1$ components and its i-th component. That is to say, P_i's set-theoretic interpretation consists of pairs (u, u_i), where u is a sequence $u_1, \ldots, u_n$ and $i \leq n$. Note that P_i is a projection which domain is any sequence of arbitrary length.

It is convenient to introduce, for every function symbol f of arity n in the signature Σ the derived *labelled projections* f_i^n defined by $\mathsf{R}_f; P_i$.

$$R \cap R = R \qquad R \cap S = S \cap R \qquad R \cap (S \cap T) = (R \cap S) \cap T$$
$$R \cup R = R \qquad R \cup S = S \cup R \qquad R \cup (S \cup T) = (R \cup S) \cup T$$
$$R\,id = R \qquad R\mathbf{0} = \mathbf{0} \qquad \mathbf{0} \subseteq R \subseteq \mathbf{1}$$
$$R \cup (S \cap R) = R = (R \cup S) \cap R$$
$$R(S \cup T) = RS \cup RT \qquad (S \cup T)R = SR \cup TR$$
$$R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$$
$$(R \cup S)^\circ = R^\circ \cup S^\circ \qquad (R \cap S)^\circ = S^\circ \cap R^\circ$$
$$R^{\circ\circ} = R \qquad (RS)^\circ = S^\circ R^\circ$$
$$R(S \cap T) \subseteq RS \cap RT \qquad RS \cap T \subseteq (R \cap TS^\circ)S$$
$$id \cup di = \mathbf{1} \qquad id \cap di = \mathbf{0} \qquad \mathbf{fp}\, \overline{x}.\mathcal{E}(\overline{x}) = \mathcal{E}(\mathbf{fp}\, \overline{x}.\mathcal{E}(\overline{x}))$$

$$hd(hd)^\circ \cap tl(tl)^\circ \subseteq id \qquad (hd)^\circ hd = (tl)^\circ tl = id \qquad (hd)^\circ tl = \mathbf{1}$$
$$\mathbf{1}(c,c)\mathbf{1} = \mathbf{1} \qquad (c,c)R(c,c) = (c,c) \cap R \qquad (c,c) \subseteq id \qquad (c,c) \cap (d,d) = \mathbf{0}$$
$$\mathsf{R}_f ; {\mathsf{R}_f}^\circ \subseteq id \qquad {\mathsf{R}_f}^\circ ; \mathsf{R}_f \subseteq id \qquad \mathsf{R}_f \cap \mathsf{R}_g = \mathbf{0} \qquad f_i^n ; \ldots ; g_j^m \cap id = 0$$

FIGURE 5.1. The equational theory $\mathbf{QRA}_\Sigma$.

5.2.2. Clark's Equality Theory. The intended meaning of the relational constants is captured by a relational version of Clark's Equality Theory:

(1)	$c \neq c'$	c, c' any pair of distinct constants
(2)	$f(\vec{t}) \neq c$	c a constant, f a functor
(3)	$f(\vec{t}) \neq g(\vec{u})$	f, g any pair of distinct functors
(4)	$f(\vec{t}) = f(\vec{u}) \to \vec{t} = \vec{u}$	f any functors
(5)	$\tau(x) \neq x$	$\tau(x)$ any term structure in which x is free

In order to capture the above equations relationally, we add several axioms to our theory. CET axiom 1 is captured by $(c,c) \cap (d,d) = \mathbf{0}$, whereas axiom 2 follows from $(c,c) \subseteq id$ and $f_i^n \cap id = \mathbf{0}$. Axiom 3 follows from $\mathsf{R}_f \cap \mathsf{R}_g = \mathbf{0}$.

Equation (4) follows from injectivity of R_f — that is to say, $\mathsf{R}_f ; {\mathsf{R}_f}^\circ \subseteq id$. Occurs check is captured by the relational axiom scheme $f_i^n ; \ldots ; g_j^m \cap id = \mathbf{0}$.

For more details on the axiomatization of CET, the reader may consult Chapter 7, where a unification algorithm based on this axiomatization is developed, and [Broome and Lipton, 1994], where the current axiomatization was introduced.

5.3. Semantics for R_Σ

Let Σ be a signature, let A be a Σ-algebra, let the interpretations of the terms $t \in \mathcal{T}_\Sigma$ be denoted by t^A and A^* be the set of all *finite* sequences in A. $A^\dagger = A \cup A^* \cup A^{**} \cup \cdots \cup A^{*^{(n)}} \cup \cdots$ all hereditarily finite sequences over A.

Let R_A be the set of pairs of members of $A^\dagger$. We then make the power set of R_A into a model of the relation calculus by interpreting atomic relation terms in a certain canonical way, and the operators in their standard set-theoretic interpretation. Notice that we interpret hd and tl in this model as operations on sequences similar to the head and tail operations on lists. The intention is to identify expressions such as $\langle a, \langle b, c\rangle\rangle$ and $\langle\langle a, b\rangle, c\rangle$ with $\langle a, b, c\rangle$, that is, our tuples are *strict*. We use the notational convention of writing vector variables $\vec{x}$ to denote sequences $\langle x_1, \ldots, x_m\rangle$ for any $m > 0$.

Definition 5.2. *Given a first-order model $A^\dagger$ and a R_{var} closing-environment $\eta : \mathsf{R}_{var} \to A^\dagger$, a relational $A^\dagger$-**interpretation** is a mapping $[\![_]\!]_\eta^{A^\dagger}$ of relational terms into R_A satisfying Fig. 5.2.*

$$
\begin{array}{lcl}
[\![(c,c)]\!]_\eta^{A^\dagger} & = & \{(c^A, c^A)\} \\
[\![id]\!]_\eta^{A^\dagger} & = & \{(u,u) \mid u \in A^\dagger\} \\
[\![di]\!]_\eta^{A^\dagger} & = & \{(u,v) \mid u \neq v\} \\
[\![hd]\!]_\eta^{A^\dagger} & = & \{(\langle t_1, t_2, \ldots, t_m\rangle, t_1) \mid t_i \in A^\dagger, m \geq 1\} \\
[\![tl]\!]_\eta^{A^\dagger} & = & \{(\langle t_1, t_2, \ldots, t_m\rangle, \langle t_2, \ldots, t_m\rangle) \mid t_i \in A^\dagger, m \geq 1\} \\
[\![R \cup S]\!]_\eta^{A^\dagger} & = & [\![R]\!]_\eta^{A^\dagger} \cup [\![S]\!]_\eta^{A^\dagger} \\
[\![R \cap S]\!]_\eta^{A^\dagger} & = & [\![R]\!]_\eta^{A^\dagger} \cap [\![S]\!]_\eta^{A^\dagger} \\
[\![R \setminus S]\!]_\eta^{A^\dagger} & = & [\![R]\!]_\eta^{A^\dagger} \setminus [\![S]\!]_\eta^{A^\dagger} \\
[\![RS]\!]_\eta^{A^\dagger} & = & \{(x,y) \mid \exists v((x,v) \in [\![R]\!]_\eta^{A^\dagger} \wedge (v,y) \in [\![S]\!]_\eta^{A^\dagger})\} \\
[\![\mathbf{1}]\!]_\eta^{A^\dagger} & = & \mathsf{R}_A \\
[\![\mathbf{0}]\!]_\eta^{A^\dagger} & = & \emptyset \\
[\![\mathsf{R}_f]\!]_\eta^{A^\dagger} & = & \{(x,\vec{y}) \mid x = f^A(t_1, \ldots, t_n) \wedge \vec{y} = \langle t_1, \ldots, t_n\rangle\} \\
[\![\mathsf{r}]\!]_\eta^{A^\dagger} & = & \{(\vec{x}\vec{u}, \vec{x}\vec{u}) \mid \vec{x} = \langle t_1, \ldots, t_n\rangle \wedge r^A(t_1, \ldots, t_n), \vec{u} \in A^\dagger\} \\
[\![\overline{x}]\!]_\eta^{A^\dagger} & = & \eta(\overline{x}) \\
[\![\mathbf{fp}\ \overline{x}.E]\!]_\eta^{A^\dagger} & = & [\![E]\!]_{\eta[\overline{x} \mapsto [\![\mathbf{fp}\ \overline{x}.E]\!]_\eta^{A^\dagger}]}^{A^\dagger}
\end{array}
$$

FIGURE 5.2. Standard interpretation of binary relations

It is easy to check that equational reasoning in $\mathbf{QRA}_\Sigma$ is sound for any such interpretation.

5.4. Logic and Relation Algebra

The $\mathbf{QRA}_\Sigma$ theory is indeed quite powerful as it may be used to interpret and execute constraint logic programs. That is the main purpose of the next chapter.

In this section, we present a translation from formulas φ of a consisting of the signature Σ, equality, and predicate symbols $r_1, r_2, \ldots$ of different arities to relation expressions. The main difference with the translation of logic programs is the absence of defined predicates and recursion, and the need to quantify the number of free variables in φ beforehand. Then, we present an already known adequacy result in a form more closely suited to our needs and that we feel closer for the computer scientist.

Definition 5.3 (Strict Projections)**.** *For a given n, define:*

$$S_1 = P_1 \qquad S_2 = P_2 \qquad \ldots \qquad S_{n-1} = P_{n-1} \qquad S_n = tl^n.$$

Observe that, in the standard interpretation for $1 \leq i \leq n$, we have $(\vec{u}, v) \in [\![S_k]\!]$ iff v is the kth component of $\vec{u}$. The difference with P_i is that S_n rules out sequences of length greater than n.

Definition 5.4 (Partial Identities)**.**

$$Q_i^n = \bigcap_{j \neq i \leq n} S_j(S_j)^\circ \qquad id_n = \bigcap_{j \leq n} S_j(S_j)^\circ$$

Observe that $(\vec{u}, \vec{v}) \in [\![id_n]\!]$ means $\vec{u}$ and $\vec{v}$ are vectors of length at least n and have the same components u_j for all $j \in \{1 \ldots n\}$. $\vec{u}[\![Q_i]\!]\vec{v}$ means all but the i-th component of $\vec{u}$ and $\vec{v}$ agree.

We define a translation $(_)^r$ for formulas. Let n be a natural number greater than the largest number of variables occurring in any sentence to be considered below. Let $x_1, \ldots, x_s$ be all the variables, free or bound, that may occur in φ. Recall all equational atomic formulas φ may be taken *elementary*: either

$x_i = a, x_i = x_j$ or $x_{i_0} = f(x_{i_1}, \ldots, x_{i_n})$. First we translate the atomic formulas as follows:

$$\begin{array}{rcl} (x_i = a)^r &=& S_i(a,a)S_i{}^\circ \cap id_n \\ (x_i = x_j)^r &=& S_iS_j{}^\circ \cap id_n \\ (x_{i_0} = f(x_{i_1}, \ldots, x_{i_n}))^r &=& \bigcap_j S_{i_j}S_j{}^\circ; \mathsf{R}_f; S_jS_{i_j}{}^\circ \cap id_n \\ (r(x_{i_1}, \ldots, x_{i_n}))^r &=& \bigcap_j S_{i_j}S_j{}^\circ; \mathsf{r}; S_jS_{i_j}{}^\circ \cap id_n \end{array}$$

The nonatomic formulas are translated as follows:

$$\begin{array}{rclcrcl} (\varphi \wedge \psi)^r &=& (\varphi)^r \cap (\psi)^r & \quad & (\varphi \vee \psi)^r &=& (\varphi)^r \cup (\psi)^r \\ (\exists x_i\varphi)^r &=& Q_i(\varphi)^r Q_i \cap id_n & & (\neg\varphi)^r &=& id_n \setminus (\varphi)^r \end{array}$$

Finally, in the case where the function symbol f will only be interpreted as an injective function, either because we are restricting attention to a class of models in which this is the case, as in the term model for a signature, or because it is implied by the axioms of the theory being used, we can simplify the translation of one of the atomic clauses above:

$$(x_{i_0} = f(x_{i_1}, \ldots, x_{i_n}))^r = \bigcap_{j=1}^n S_{i_o}; f_j^n; S_{i_j}^o$$

The result below is one of many established in Tarski and Givant [1987] It was independently established by Freyd in a categorical setting [Freyd and Scedrov, 1991]. The statement of the theorem and its proof are new, but draw on similar ideas independently proved by Freyd, Maddux and Tarski.

Theorem 5.5 (Freyd, Maddux, Tarski)**.** *Let $\mathcal{L}$ be a first-order language, $\mathfrak{A}$ a model over $\mathcal{L}$. Let ψ be a first-order formula over $\mathcal{L}$, and let $x_1, \ldots, x_n$ contain all the variables, free or bound, that may occur in ψ. Then, if ψ is a closed formula $\mathfrak{A} \models \psi \iff [\![(\psi)^r]\!] = [\![id_n]\!]$. If ψ is open, with free variables $x_{i_1}, \ldots, x_{i_m}$ among $x_1, \ldots, x_n$*

$$[\![(\psi)^r]\!] = \{(\langle a_1, \ldots, a_n\rangle, \langle a_1, \ldots, a_n\rangle) \mid \mathfrak{A} \models \psi[a_1/x_1, \ldots, a_m/x_m]\}$$

Note that $\mathfrak{A} \models \psi[a_1/x_1, \ldots, a_m/x_m]$ is equivalent to $\mathfrak{A} \models \psi[a_{i_1}/x_{i_1}, \ldots, a_{i_m}/x_{i_m}]$ because only the x_{i_j} occur freely in ψ, In particular, the first conclusion of the theorem is just a special case of the second, where none of the x_{i_j} are free in ψ.

Proof. The proof is a straightforward induction on the structure of formulas:

- If $\psi \equiv (x_i = a)$, then let $\vec{u}$ be an n-tuple of terms in $\mathfrak{A}$, and suppose $(\vec{u}, \vec{u}) \in [\![(x_i = a)^r]\!] = [\![S_i(a,a)S_i^\circ \cap id_n]\!]$. This implies that $u_i = a$, hence $\mathfrak{A} \models (x_i = a)[u_1/x_1, \ldots, u_n/x_n]$. Conversely $u_i = a$ forces $(\vec{u}, \vec{u}) \in [\![(x_i = a)^r]\!]$.
- If $\psi \equiv (x_i = x_j)$, then suppose $(\vec{u}, \vec{u}) \in [\![(x_i = x_j)^r]\!] = [\![S_iS_j^\circ \cap id_n]\!]$. Then we have $u_i = u_j$ and $\mathfrak{A} \models (x_i = x_j)[\vec{u}/\vec{x}]$. The converse is also immediate.
- If $\psi \equiv x_{i_0} = f(x_{i_1}, \ldots, x_{i_n})$, then suppose $(\vec{u}, \vec{u}) \in [\![(x_{i_0} = f(x_{i_1}, \ldots, x_{i_n}))^r]\!]$, that is to say, in

  $$\bigcap_j S_{i_j}S_j{}^\circ; \mathsf{R}_f; S_jS_{i_j}{}^\circ \cap id_n.$$

  This is clearly equivalent to $u_{i_1} = f^A(u_{i_2}, \ldots, u_{i_m})$ and hence the conclusion of the theorem for this case.
- $\psi \equiv r(x_{i_1}, \ldots, x_{i_n})$, then suppose $(\vec{u}, \vec{u}) \in [\![(r(x_{i_1}, \ldots, x_{i_n}))^r]\!]$ i.e. $(\vec{u}, \vec{u})$ lies in

  $$\bigcap_j [\![S_{i_j}S_j{}^\circ; \mathsf{r}; S_jS_{i_j}{}^\circ]\!] \cap [\![id_n]\!].$$

  Now suppose $\vec{v}$ is a sequence such that $\vec{u}[\![S_{i_j}S_j{}^\circ]\!]\vec{v}$. Then for each j between 1 and m $v_j = u_{i_j}$, so $\langle(u_{i_1}, \ldots, u_{i_m}, u_{i_1}, \ldots, u_{i_m})\rangle \in [\![\mathsf{r}]\!]$, which implies that $\mathfrak{A} \models r(u_{i_1}, \ldots, u_{i_m})$. Conversely $\mathfrak{A} \models r(a_1, \ldots, a_m)$ implies that for any $\vec{u}$ with all $u_{i_j} = a_j$, $\vec{u}, \vec{u}) \in [\![(r(x_{i_1}, \ldots, x_{i_n}))^r]\!]$.
- If $\psi \equiv \varphi_1 \wedge \varphi_2$, then suppose $(\vec{u}, \vec{u}) \in [\![(\varphi_1 \wedge \varphi_2)^r]\!]$. Then $(\vec{u}, \vec{u}) \in [\![(\varphi_1)^r]\!]$ and $(\vec{u}, \vec{u}) \in [\![(\varphi_2)^r]\!]$. By the induction hypothesis, $\mathfrak{A} \models \varphi_1[\vec{u}/\vec{x}]$ and $\mathfrak{A} \models \varphi_2[\vec{u}/\vec{x}]$ and hence $\mathfrak{A} \models (\varphi_1 \wedge \varphi_2)[\vec{u}/\vec{x}]$

- If $\psi \equiv \neg\varphi$, suppose $(\vec{u}, \vec{u}) \in [\![(\neg\varphi)^r]\!]$. Then $(\vec{u}, \vec{u}) \in id_n \setminus [\![(\varphi)^r]\!]$ which is equivalent to saying that $(\vec{u}, \vec{u}) \in [\![(\varphi)^r]\!]$ is not the case. Using the induction hypothesis, this is equivalent to $\mathfrak{A} \not\models \varphi[\vec{u}/\vec{x}]$, and hence to $\mathfrak{A} \models \neg\varphi[\vec{u}/\vec{x}]$.
- If $\psi \equiv \exists x_i \varphi$, then if $(\vec{u}, \vec{u}) \in [\![(\exists x_i \varphi)^r]\!]$ we must have $(\vec{u}, \vec{u}) \in [\![Q_i(\varphi)^r Q_i]\!]$. This means there is an n-tuple of terms $\vec{v}$ with $v_j = u_j$ for every j between 1 and n except i, and $(\vec{v}, \vec{v}) \in [\![(\varphi)^r]\!]$. By the induction hypothesis, this means $\mathfrak{A} \models \varphi[\vec{v}/\vec{x}]$. But this is equivalent to $\mathfrak{A} \models \exists x_i \varphi[\vec{u}/\vec{x}]$. The converse is left to the reader.

□

The proof can be specialized to the case of a language consisting only of a signature Σ, with equality as the sole predicate symbol, with relation structure $\mathsf{R}_{\mathcal{H}_\Sigma}$ induced by the term model $\mathcal{H}_\Sigma$. The only case that changes is the one for the atomic formula $x_{i_0} = f(x_{i_1}, \ldots, x_{i_m})$, since we use the modified translation

$$(x_{i_0} = f(x_{i_1}, \ldots, x_{i_n}))^r = \bigcap_{j=1}^{n} S_{i_o}; f_j^n; S_{i_j}^\circ$$

But if $(\vec{u}, \vec{u})$ is a member of

$$\bigcap_j [\![S_{i_0}; f_j^n; S_{i_j}^\circ]\!] \cap [\![id_n]\!],$$

then $u_{i_1} = f(u_{i_2}, \ldots, u_{i_{n+1}})$ and $\mathcal{H}_\Sigma \models \varphi[\vec{u}/\vec{x}]$. The converse is immediate.

CHAPTER 6

Relational Semantics for Constraint Logic Programming

In this chapter we explore how constraint logic programs (CLP) may be profitably understood, extended and compiled in terms of an underlying equational relational calculus. First, we define a translation of CLP to terms of the relational language R_Σ defined in Sec. 5.1. We relate relational semantics with standard set-theoretic semantics for logic programs using the standard interpretation $[\![\cdot]\!]_\eta^{A^\dagger}$ of binary relations. Thus, the set-theoretic semantics of the relational translation of a program corresponds to the traditional denotational semantics presented in Sec. 3.4. Second, we define a rewriting system for query execution against translated constraint logic programs. We show that answers obtained by resolution correspond to normal forms in our rewriting system.

6.1. Introduction

As seen in Chapter 3, the syntax and operational semantics of logic programs can be presented in a concise and elegant way. However, while the operational semantics is based on elegant logical foundations — namely the resolution proof method — reasoning over the induced provability relation is not trivial and requires unrelated meta-logical machinery. Program states have associated proof-search trees. Logical variables act as pointers, thus an instantiation of a variable must be propagated across the corresponding tree, and each resolution step performs a renaming apart in order to avoid name clashes. In implementations, the use of a custom abstract machine is the norm, an approach that bears little resemblance with the original proof-theoretical definition.

A compositional operational semantics for logic programming usually defines resolution as a transition over program states. In that approach, the handling of a disjunctive proof step generates a set of derivations, requiring a custom notion of composition and set comprehension. While this approach is sound, the use of ad-hoc definitions poses a barrier to the understanding of the semantics and reasoning over its theory.

We use relation algebra to avoid that class of problems. The rich nature of the relation calculus excludes the need for *ad-hoc* operators and definitions. For instance, in our relational semantics, disjunction corresponds to the $\cup$ operator, a compositional notion may be understood as $R;(S \cup T) = R;S \cup R;T$, etc... We consider the $\mathbf{QRA}_\Sigma$ theory to be a suitable and elegant framework for denotational and operational reasoning over constraint logic programs.

The equational nature of relation algebra fosters the comprehensive use of equational reasoning, an approach that has several benefits in our setting. For instance, the use of algebraic rules for the definition of the operational semantics allows to reason easily about execution strategies, making it possible to quickly adapt our framework to other strategies like breadth-first search or Andorra [Warren, 1987]. The equation

$$R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$$

captures the handling of disjunctive clauses. Choosing whether to reduce the operands of $\cup$ in parallel or sequentially is a matter of slight changes in the rewriting rules or strategy. Impure features like cut are captured by the rule $S \subseteq R \cup S$. The inclusion of execution in regular relational reasoning makes the new semantics an ideal vehicle for abstract interpretation, optimization, and program analysis.

The basis of the translation from constraint logic programs to relational terms is the semantic version of the equipollence theorem presented in Sec. 5.4, which states that for any first order formula with $\vec{x}$ free variables $\varphi[\vec{x}]$, the set-theoretical interpretation $[\![(\varphi[\vec{x}])^r]\!]$ of its relational translation $(\varphi[\vec{x}])^r$ is the set of ground pairs $(\vec{a}, \vec{a})$, such that $\varphi[\vec{a}]$ is true.

Previous work by Broome and Lipton [1992, 1994] and Lipton and Chapman [1998] showed that pure logic programs can be interpreted by relational terms. A query is reduced by a rewriting system derived from the axioms of relation algebra, until a normal form is found. Then, the Mal'tsev [1961] and Maher [1988] quantifier elimination algorithm is used to provide a first-order logical description of the set of answers.

In this chapter, we extend the existing scheme by replacing equations over first order terms with arbitrary constraints. The relational formalism manipulates constraint formulas, but the satisfiability of a particular constraint is answered by an external solver. From our point of view, we assume nothing about the implementation of a particular constraint solvers, handling them as opaque "black boxes".

The inclusion of a fixed point **fp** operator in $\mathbf{QRA}_\Sigma$ is a divergence from the equational flavor of our theory. However, we profit from the fact that predicate names are unique within a program and replace the fixpoint operator by a notion of instantiation of unique names. This implements the first order recursion found in Prolog in a simple way and brings us a great benefit: relational variables are no longer present in the terms resulting from the translation. This way, every predicate corresponds to a variable-free first order ground term of R_Σ.

The use of ground terms permits the use of rewriting, overcoming the practical and theoretical difficulties that the existence of logical variables causes in equational reasoning. We define SLD resolution using a rewriting system over terms of R_Σ, extending the denotational semantics with an operational interpretation. We may speak of *executable* semantics, given that the denotational interpretation function and the compilation function are the same. Computation is built into the relational formalism, making the relational calculus into a new vehicle for proof search and automated deduction. The rewriting system takes care of parameter passing and search strategy, while the constraint solver takes care of unification and domain-specific constraint checking. Checking the correctness of the implementation is straightforward.

However, the proof of operational equivalence between the rewriting system and the traditional operational semantics is not straightforward. A new algebraic notion of state and new logic-style *folding* operational semantics are introduced. The required meta-theory of SLD proof search is formalized, and operational equivalence of the three systems follows.

Summarizing, we will prove that the Constraint Logic Programming Domain is equivalent to the logical-style transition system, which will also be shown equivalent to the relational-style transition system. The rewriting system will then be proven to implement the relational transition system, concluding the proof.

Structure of the Chapter. We start the chapter with an informal example of how the translation works in Sec. 6.2. In Sec. 6.3, we define the translation of constraints and CLP programs. We prove the translation correct and adequate. In Sec. 6.4, we define a rewriting system over relational terms that implements SLD resolution. We discuss the rewriting-theoretical properties of the system and prove it is confluent. In Sec. 6.5, we prove that the rewriting system previously defined is equivalent to the classical operational semantics for CLP defined in Sec. 3.5.. To end the chapter, we discuss related and future work in Sections 6.6 and 6.7.

6.2. An Introductory Example

We develop an example to give an early glimpse into how the translation works, with the aim of guiding the reader through the most important concepts and ideas present in the rest of the chapter.

Until the full details of the relational formalism are introduced in later sections, we will temporarily use familiar expressions denoting sets of pairs to represent binary relations as well as an informal language to discuss them. Thus, this example should be regarded as an informal one.

The key idea of the semantics is the translation of a predicate p to a relational term $\bar{p}$. The source of the translation function is then the definition of p, whereas the result is a ground relation term which conforms the declarative semantics of the predicate and its *executable code*. The execution of a query m against the predicate p is relationally represented as $\bar{m} \cap \bar{p}$. The query is executed by rewriting modulo the relational theory. By carefully unfolding the definition of $\bar{p}$ we obtain — in the case of terminating computations — a union of relational terms in normal form, corresponding to the answers of a traditional CLP system. Distinct clauses are translated using the union operator. Parameter passing is handled by projection operators.

We start by considering the classical Prolog program for implementing addition:

```
add(o,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

In order to better understand the logical meaning of the add predicate, we first transform it to a form with pure heads:

```
add(X,Y,Z) :- X = o, Y = Z.
add(X,Y,Z) :- X = s(X1), Z = s(Z1), add(X1,Y,Z1).
```

The Clark completed form [Lloyd, 1984] presents the logical interpretation of the original program when its minimal semantics are considered:

$$\begin{aligned} add(X,Y,Z) \leftrightarrow \quad & (X = o \wedge Y = Z) \vee \\ & \exists X', Z'.\ (X = s(X') \wedge Z = s(Z') \wedge add(X',Y,Z')) \end{aligned}$$

The signature of the program is $\{o/0, s/1\}$, thus the associated *Herbrand Universe* $\mathcal{H}$ is the infinite set $\{o, s(o), s(s(o)), \ldots\}$ in bijection with the standard model for the natural numbers.

We now illustrate the relational translation of the preceding code in an incremental way. In a nutshell, the objective is to translate a predicate symbol, such as add, to a term which represents a *coreflexive binary* relation. Recall that a relation R is coreflexive if $R \subseteq id$. Coreflexive relations are a natural way to encode sets in the relation calculus. For example, the natural numbers can be represented as the coreflexive relation $\{(0,0),(1,1),(2,2),....\}$.

Using quasi-projections, we can represent arbitrarily long tuples of terms (from the Herbrand Universe) as expressions built up from binary relations. Let's use brackets $[,,]$ as a notational convention to construct sequences or *vectors of ground terms*. We use parentheses to represent pairs of such sequences, as in $([o, s(o), s(o)], [s(o), o])$.

A notational convenience for coreflexive relations is to write them using angle brackets. Thus $\langle t \rangle$ is shorthand for the relation consisting of the pair (t,t), and for example, $\langle a,b,c \rangle = P_1; (a,a); P_1^\circ \cap P_2; (b,b); P_2^\circ \cap P_3; (c,c); P_3^\circ$ with interpretation $[\![\langle a,b,c \rangle]\!] = \{([a,b,c],[a,b,c])\}$.

The semantics of the relation resulting from the translation of `add` is the set of pairs $([X,Y,Z],[X,Y,Z])$, such that $add(X,Y,Z)$ is provable by resolution and X, Y, Z are ground terms, that is to say, the infinite set

$$[\![\langle o,o,o \rangle \cup \langle o, s(o), s(o) \rangle \cup \langle o, s(s(o)), s(s(o)) \rangle \cup \ldots]\!]$$

This set is in bijection with the *Herbrand interpretation* of add induced by SLD resolution.

Using the $\langle\rangle$ relation, we may assign a relational meaning to ground facts:

```
parent(john, jim).
parent(jim, dana).
```

For the above example, the relational semantics are $\overline{parent} = \langle john, jim \rangle \cup \langle jim, dana \rangle$. However, the first clause of add, contains a free variable. This is not a problem for the semantics: now we'll have a

relation whose semantics are an infinite number of pairs, but we must extend the $\langle\rangle$ relation to translate variables to *ground relational terms*.

We use a capital letter X in a position of the pairing relation to signal it should contain all the terms of the signature. Thus, $[\![\langle o, X, X\rangle]\!] = \{([o, a, a], [o, a, a]) \mid a \in \mathcal{H}\}$. A very important fact is that any capital letter X, Y, etc. occurring inside $\langle\rangle$ **is local to it**. So $\langle o, s(o)\rangle \subseteq \langle X, Y\rangle$, but $\langle o, s(o)\rangle \not\subseteq \langle X, X\rangle$.

Then, the relational counterpart of `add(o,X,X)` is $\langle o, X, X\rangle$. The exact details of the translation are in Sec. 6.3. For reference, the exact relational term for $\langle o, X, X\rangle$ is $R; R^\circ$, where $R = P_1; (o, o) \cap P_2; P_1^\circ \cap P_3; P_1^\circ$[1]. Thus $\langle o, X, X\rangle$ still corresponds to a ground relational finite term of the relational language R_Σ.

Arbitrary facts can be now translated using this scheme. However, recursive definitions require extra machinery in order to capture parameter passing and renaming apart. For the second clause of add, we'd like to express relationally the informal equation:

$$\overline{add} = \langle o, X, X\rangle \cup \langle s(X'), Y, s(Z') \mid \langle X', Y, Z'\rangle \in \overline{add}\rangle$$

where we have extended the $\langle\rangle$ operator to support a notion of recursive comprehension. Unfolding the definition we'd get:

$$\overline{add} = \langle o, X, X\rangle \cup \langle s(o), X, s(X)\rangle \cup \langle s(X'), Y, s(Z') \mid \langle X', Y, Z'\rangle \in \overline{add}\rangle$$

as desired. Note the similarity to the classical Kowalski-Van Emden interpretation. Indeed the relational approach is a rigorous implementation of it, where all the meta-theory is fully incorporated into the object level calculus.

Allowing this informal notion of parameter passing, we may compute answers to queries using some rewriting rules. Consider the query `?- add(o,s(o),X).`:

$$\langle o, s(o), X\rangle \cap \underbrace{\overline{add}}_{\text{[unfold]}} \rightarrow$$

$$\underbrace{\langle o, s(o), X\rangle \cap (\langle o, X, X\rangle \cup \langle s(X'), y, s(Z') \mid \langle X', Y, Z'\rangle \in \overline{add}\rangle)}_{[A \cap (B \cup C) = (A \cap B) \cup (A \cap C)]} \rightarrow$$

$$\underbrace{(\langle o, s(o), X\rangle \cap \langle o, X, X\rangle)}_{\text{[primitive intersection]}} \cup (\langle o, s(o), X\rangle \cap \langle s(X'), Y, s(Z') \mid \langle X', Y, Z'\rangle \in \overline{add}\rangle) \rightarrow$$

$$\langle o, s(o), s(o)\rangle \cup \underbrace{(\langle o, s(o), X\rangle \cap \langle s(X'), Y, s(Z') \mid \langle X', Y, Z'\rangle \in \overline{add}\rangle)}_{[\textit{magic}]} \rightarrow$$

$$\underbrace{\langle o, s(o), s(o)\rangle \cup \emptyset}_{[A \cup 0 = A]} \rightarrow$$

$$\langle o, s(o), s(o)\rangle$$

Note the *magic* step. We introduced the comprehension notion in an informal way, and while the meaning is clear, we don't have a set of rewriting rules for it. We now formalize the comprehension notion and illustrate the full relational flavor of the formalism to be developed in this chapter. Recall that the logical interpretation of add used a total of five logic variables. We modify the translation for the second clause of add in to add two new slots in the semantics, corresponding to X' and Z' in the comprehension. We may understand them as placeholders used to store the elements of the recursive call.

However, the addition of two new slots creates a problem. Now, the relation representing the first clause of add has a semantics of sequences of length 3, while the second clause is interpreted as a relation between sequences of length 5. We overcome this problem introducing helper projection relations. Define the helper binary relations I (a truncated identity relation) and W (a truncated permutation relation),

[1] Note that this relation is coreflexive only up to the first element of the sequence.

such that $\forall x, y, z, y', z' \in \mathcal{H}$:

$$\begin{array}{lll} ([x,y,z], & [x,y,z,x',z']) & \in [\![I]\!] \\ ([x,y,z,x',z'], & [x',y,z']) & \in [\![W]\!] \end{array}$$

I° and W° stand for the converses (or inverses) of the respective relations, that is, semantically, $R^\circ = \{(y,x) \mid (x,y) \in R\}$. For instance,

$$([o, s(o), s(s(s(o)))], [o, s(o), s(s(o)), o, s(s(s(o)))]) \in [\![W^\circ]\!]$$

Note that I and W are particular to the add predicate. Using these new relations, we can implement the parameter passing in a fully relational way.

The complete relational translation of $\overline{add}$ is:

$$\overline{add} = \langle o, X, X\rangle \cup I(\langle s(X'), Y, s(Z'), X', Y'\rangle \cap W; \overline{add}; W^\circ)I^\circ$$

The I relation hides the local slots which are temporal in nature, whereas the W relation permutes the underlying vector in order to place the parameters in their correct place. Together they form a complete parameter passing mechanism for CLP, including renaming apart. We proceed with a slightly more complex query, using true relational rewriting:

$$\begin{array}{ll}
\langle s(o), s(o), X\rangle \cap \overline{add} & \mapsto \\
\ldots \quad \text{[similar to the previous example]} & \\
\langle s(o), s(o), X\rangle \cap I(\langle s(X'), Y, s(Z'), X', Z'\rangle \cap W; \overline{add}; W^\circ)I^\circ & \mapsto \ (*) \\
I(\langle s(o), s(o), s(Z'), o, Z'\rangle \cap W\langle o, X, X\rangle W^\circ)I^\circ & \mapsto \\
I(\langle s(o), s(o), s(Z'), o, Z'\rangle \cap W(\langle o, s(o), Z\rangle \cap \langle o, X, X\rangle)W^\circ)I^\circ & \mapsto \\
I(\langle s(o), s(o), s(Z'), o, Z'\rangle \cap W\langle o, s(o), s(o)\rangle W^\circ)I^\circ & \mapsto \\
I(\langle s(o), s(o), s(Z'), o, Z'\rangle \cap \langle s(o), s(o), s(s(o)), _, _\rangle I^\circ & \mapsto \\
I(\langle s(o), s(o), s(s(o)), o, s(o)\rangle I^\circ & \mapsto \\
\langle s(o), s(o), s(s(o))\rangle &
\end{array}$$

Note that we have omitted the second clause of add when instantiating $\overline{add}$ for the sake of brevity. The most interesting step is $(*)$, where the leftmost relation expression moves inside the instantiated part of $\overline{add}$ (using a variant of the so-called modular law) in order to resolve the query.

6.3. Translation of CLP Programs

We define the translation of Constraints and CLP programs into relations.

6.3.1. Translation of Constraints. The main objective of this section is to define a translation $\dot{K} : \mathcal{L}_\mathcal{D} \to \mathsf{R}_\Sigma$, such that a constraint $\varphi[\vec{x}] \in \mathcal{L}_\mathcal{D}$ is translated to a relational term whose semantics is precisely the set of elements which make $\varphi[\vec{x}]$ true:

$$(\vec{a}\vec{u}, \vec{a}\vec{u}') \in [\![\dot{K}(\varphi[\vec{x}])]\!] \iff \mathcal{D} \models \varphi[\vec{a}]$$

$\vec{a} \in \mathcal{T}_\Sigma$ is a sequence of ground terms, $|\vec{a}| = |\vec{x}| = n$, and $\vec{u}, \vec{u}'$ any arbitrary sequences.

Remark 6.1. *Note that the result of the translation is not a coreflexive but for the first n elements of the sequence. In previous versions, true coreflexive relations were used by intersecting the final term with id. Compare the above equation with:*

$$(\vec{a}\vec{u}, \vec{a}\vec{u}) \in [\![\dot{K}(\varphi[\vec{x}])]\!] \iff \mathcal{D} \models \varphi[\vec{a}]$$

A relation R such that $R \subseteq I_n$ is called n-coreflexive. The choice of using n-coreflexive relations allows us to drop the $\cap\, id$ suffix.

This difference is not significant form the point of view of the program semantics as a n-coreflexive relation will become a $n+k$-coreflexive relation when intersected with another $n+k$-coreflexive.

Indeed, we may restate this remark as "if our program has n *variables, we don't care about elements on the sequence beyond* n*".*

We assume variables occurring in a constraint to be previously normalized — maybe by a program preprocessor — such that they are a subset of $\{x_1, \ldots, x_n\}$. This means each variable within a constraint is uniquely identified by a natural number. The relational terms themselves resulting from the translation are *ground*, not making use of any relational variable. This as one of the main advantages and allows the use of rewriting and other forms of reasoning dependent on ground representations.

We first examine how parameter passing to constraint predicates work. Let's use the constraint $s(x_2) < s(s(x_3))$ as a running example. In the relational signature, we have relations R_s and $<$ corresponding to the term former s and constraint predicate $<$ respectively. The variables x_2, x_3 will be eliminated. Recall the set-theoretic semantics of both relations:

$$\begin{array}{lcl} [\![s]\!]^{A^\dagger}_\eta & = & \{(x,t) \mid \vec{t} \in \mathcal{T}_\Sigma \wedge x = s^A(t)\} \\ [\![<]\!]^{A^\dagger}_\eta & = & \{([x,y]\vec{u}, [x,y]\vec{u}' \mid x,y \in \mathcal{T}_\Sigma, x < y, \vec{u}, \vec{u}' \in A^\dagger\} \end{array}$$

then, we build the relation $R \equiv P_1; s; P_2^\circ \cap P_2; s; s; P_3^\circ$ with semantics:

$$\begin{array}{lcll} [\![P_1; s; P_2^\circ \cap P_2; s; s; P_3^\circ]\!]^{A^\dagger}_\eta & = & \{([p_1,p_2]\vec{u}, [x_1,x_2,x_3]\vec{v}) \mid & x_1,x_2,x_3 \in \mathcal{T}_\Sigma, \vec{u}, \vec{v} \in A^\dagger, \\ & & & p_1 = s^A(x_2) \wedge \\ & & & p_2 = s^A(s^A(x_3))\} \end{array}$$

that is to say, it is the relation between a vector with the two first elements $s(x_2)$ and $s(s(x_3))$ and a vector where the three first elements are all the terms of $\mathcal{T}_\Sigma$, that is to say, they behave as variables. The semantics of the relation $\overline{<}; R$ is:

$$\begin{array}{lcll} [\![\overline{<}; R]\!]^{A^\dagger}_\eta & = & \{ & ([p_1,p_2]\vec{u}, [x_1,x_2,x_3]\vec{v}) \mid x_1,x_2,x_3 \in \mathcal{T}_\Sigma, \vec{u}, \vec{v} \in A^\dagger, \\ & & & p_1 = s^A(x_2) \wedge p_2 = s^A(s^A(x_3)) \wedge p_1 < p_2\} \end{array}$$

Note how relation composition behaves semantically as an existential quantifier. Let $S \equiv \overline{<}; (P_1; s; P_2^\circ \cap P_2; s; s; P_3^\circ)$, then $S^\circ; S$ is the constraint we are looking for, with semantics:

$$\begin{array}{lcll} [\![S^\circ; S]\!]^{A^\dagger}_\eta & = & \{([x_1,x_2,x_3]\vec{u}, [x_1,x_2,x_3]\vec{u}') \mid & x_1,x_2,x_3 \in \mathcal{T}_\Sigma, \vec{u}, \vec{u}' \in A^\dagger, \\ & & & s(x_2) < s(s(x_3))\} \end{array}$$

Conjunction of constraints is directly translated to intersection. Existential quantification is modeled by a *hiding relation* Q_i, such that a constraint $\varphi[\vec{x}]$ with n free variables has a semantics of a vector with n significant terms, the formula $\exists x_i.\ \varphi[\vec{x}]$ will have a semantics with $n-1$ significant terms, where the i position becomes free. In the case of unsatisfiability, its semantics will be the empty set. Recall from the equipollence theorem that the translation of a closed formula is true iff it is provably equal to $\mathbf{1}$. Similarly, a the semantics of a constraint with no free variables will be equal to $[\![\mathbf{0}]\!]$ or $[\![\mathbf{1}]\!]$.

Definition 6.2 (Hiding Relation)**.** *The hiding relation* Q_i *for position* i *is defined as:*

$$Q_i = \bigcap_{j \neq i} P_j; P_j^\circ$$

Observe that $u[\![Q_i]\!]v$ *means that all but the* i*-th component of* u *and* v *agree.*

Note that Q_i is defined as an infinite intersection of relational projections. We think the use of Q_i is the best in terms of readability. However, it is not the only choice, as a partial identity relation I_n — which is defined by a fixpoint-free finite relational term — could be used for the same purpose. The following remarks outline some of the cons and pros of our choice:

- In Chapter 5, the hiding relation Q_i^n for vectors of size n is defined. If we used Q_i^n here, the translation would become dependent on the number of free variables of a constraint. Using Q_i doesn't impose a limit on the number of variables.

- Q_i can be finitely defined within $\mathbf{QRA}_\Sigma$ by using the fixpoint operator, either using the presented form or using its union form $Q_i = \bigcup_{n \geq i} Q_i^n$.
- If we normalize constraints such that the existentially quantified variables have a greater index than all the free ones at the quantifier occurrence, Q_i can be replaced by the partial identity relation I_n, which "hides" all the variables greater than n.
- CLP languages like Prolog lack the $\wedge$ and $\exists$ operators for constraints. The standard mechanisms for existential quantification (using a new variable in the tail) and conjunction (the `,` predicate) are used in order to build constraint formulas.
- In the next section, we present a translation procedure for CLP programs, which uses I_n for existentially quantified variables. In the light of these remarks, the translation for programs can be adapted to arbitrary constraint formulas either using a constraint normalization procedure or by generating new predicates.

We define two translation functions, one for terms and one for constraint formulas:

Definition 6.3 (Term Translation)**.** *The $K : \mathcal{T}_\Sigma(\mathcal{X}) \to \mathsf{R}_\Sigma$ translation function for terms is:*

$$\begin{array}{lcl} K(a) & = & (a,a)\mathbf{1} \\ K(x_i) & = & P_i^\circ \\ K(f(t_1,\ldots,t_n)) & = & \bigcap_{i\leq n} f_i^r K(t_i) \end{array}$$

Definition 6.4 (Constraint Translation)**.** *The $\dot{K} : \mathcal{L}_\mathcal{D} \to \mathsf{R}_\Sigma$ translation function for constraint formulas with n free variables is:*

$$\begin{array}{lcl} \dot{K}(p(t_1,\ldots,t_n)) & = & \mathsf{p}; (\bigcap_{i\leq n} P_i; K(t_i)); (\bigcap_{i\leq n} K(t_i)^\circ; P_i^\circ); \mathsf{p} \\ \dot{K}(\varphi \wedge \theta) & = & \dot{K}(\varphi) \cap \dot{K}(\theta) \\ \dot{K}(\exists x_i.\, \varphi) & = & Q_i; \dot{K}(\varphi); Q_i \end{array}$$

Lemma 6.5. *Given a vector of variables $\vec{x}$ and a term $t[\vec{x}]$ of length n:*

$$(t^A[\vec{a}], \vec{a}\vec{u}) \in [\![K(t[\vec{x}])]\!]_\eta^{A^\dagger}$$

where $\vec{a} \in \mathcal{T}_\Sigma^n$ and $\vec{u}$ an arbitrary term sequence.

Proof. By induction over the structure of terms. □

Lemma 6.6. *For any normal form constraint formula $\varphi[\vec{x}]$ with n free variables*

$$(\vec{a}\vec{u}, \vec{a}\vec{u}') \in [\![\dot{K}(\varphi[\vec{x}])]\!] \iff \mathcal{D} \models \varphi[\vec{a}]$$

This basically amounts to a soundness and completeness result.

Proof. By induction over the structure of the formulas:

- For the case $p(\vec{t}[\vec{x}])$, we want

$$(\vec{a}\vec{u}, \vec{a}\vec{u}') \in [\![\dot{K}(p(\vec{t}[\vec{x}]))]\!] \iff \mathcal{D} \models p(\vec{t}[\vec{a}])$$

Assume a unary constraint predicate (the argument extends easily to more). Unfolding the translation:

$$(\vec{a}\vec{u}, \vec{a}\vec{u}') \in [\![\mathsf{p}; P_1; K(t[\vec{x}]); K(t[\vec{x}])^\circ; P_1^\circ; \mathsf{p}]\!] \iff \mathcal{D} \models p(\vec{a})$$

We focus on the semantics of $\mathsf{p}; P_1; K(t[\vec{x}])$. By Lem. 6.5 and the semantics of projections is straightforward to prove that $[\![P_1; K(t[\vec{x}])]\!] = \{([t^A[\vec{a}]]\vec{u}, \vec{a}\vec{v}) \mid \vec{a} \in \mathcal{T}_\Sigma^n, \vec{u}, \vec{v} \in A^\dagger\}$. Recall that the semantics of relation composition behaves as an existential quantifier, then:

$$[\![\mathsf{p}; P_1; K(t[\vec{x}])]\!] = \{(x,y) \mid (\exists v.(x,v) \in [\![p]\!] \wedge (v,y) \in \{([t^A[\vec{a}]]\vec{u}, \vec{a}\vec{v}) \mid \vec{a} \in \mathcal{T}_\Sigma^n\})\}$$

given that $\mathsf{p} \subseteq id$, we can rewrite the equations:

$$[\![\mathsf{p}; P_1; K(t[\vec{x}])]\!] = \{([t^A(\vec{a})]\vec{u}, \vec{a}\vec{v}) \mid \vec{a} \in \mathcal{T}_\Sigma^n \wedge p(t^A[\vec{a}])\}$$

which establishes the lemma.

- For the case $\varphi \wedge \theta$,

$$(\vec{a}\vec{u}, \vec{a}\vec{u}') \in [\![\dot{K}(\varphi) \cap \dot{K}(\theta)]\!] \iff \mathcal{D} \models \varphi[\vec{a}] \wedge \theta[\vec{a}]$$

By induction hypothesis, the lemma holds for $\varphi[\vec{a}]$ and $\theta[\vec{a}]$. By the semantics of the intersection operator this case is straightforward.

- For the case $\exists x_i.\varphi$, we have that $\varphi[\vec{x}]$ has n open variables, so $(\exists x_i.\varphi)[\vec{x}']$ has $n-1$ open variables. By induction hypothesis we know that

$$(\vec{a}\vec{u}, \vec{a}\vec{u}') \in [\![K(\varphi)]\!] \iff \mathcal{D} \models \varphi[\vec{a}]$$

holds, thus it is immediate that:

$$(\vec{a}_{i-1}[t]\vec{a}_{i+1}\vec{u}, \vec{a}_{i-1}[t]\vec{a}_{i+1}\vec{u}') \in [\![Q_i; \dot{K}(\varphi); Q_i]\!] \iff \mathcal{D} \models (\exists x_i.\varphi)[\vec{a}_{i-1}\vec{a}_{i+1}]$$

for arbitrary $t \in \mathcal{T}_\Sigma$ and $\vec{u}$ holds.

□

6.3.2. Adequacy of the translation. We prove that $\vdash$ corresponds to $\subseteq$ for the relational translation of constraints.

Theorem 6.7 (Adequacy of Constraint Translation)**.** *Given constraint formulas* $\Gamma = \{\gamma_1, \ldots, \gamma_n\}$, φ *of a constraint system* $(\mathcal{L}_\mathcal{D}, \vdash)$ *then* $\Gamma \vdash \varphi$ *iff* $[\![\dot{K}(\Gamma)]\!] \subseteq [\![\dot{K}(\varphi)]\!]$.

Proof. We only show a few cases, as the rest are similar: For

$$\frac{}{\Gamma, d \vdash d}$$

the result is immediate. For

$$\frac{\Gamma_1 \vdash d \quad \Gamma_2, \vdash e}{\Gamma_1, \Gamma_2 \vdash d \wedge e}$$

We need to check that $[\![\dot{K}(\Gamma_1)]\!] \cap [\![\dot{K}(\Gamma_2)]\!] \subseteq [\![\dot{K}(d)]\!] \cap [\![\dot{K}(e)]\!]$ is implied by $[\![\dot{K}(\Gamma_1)]\!] \cap [\![\dot{K}(\Gamma_2)]\!] \subseteq [\![\dot{K}(d)]\!]$ and $[\![\dot{K}(\Gamma_1)]\!] \cap [\![\dot{K}(\Gamma_2)]\!] \subseteq [\![\dot{K}(e)]\!]$. This is a consequence of $A \subseteq B \equiv A \cap B = A$, then, $A \subseteq B$ and $A \subseteq C$ implies $A \subseteq B \cap C$.

For

$$\frac{\Gamma \vdash d[t/x_i]}{\Gamma \vdash \exists x_i.d}$$

We to check that $[\![\dot{K}(\Gamma)]\!] \subseteq [\![\dot{K}(d[t/X])]\!] \subseteq [\![Q_i\dot{K}(d)Q_i]\!]$, which is true as $A \subseteq Q_i; A; Q_i$. □

Remark 6.8. *In this setting,* $[\![\top]\!] = \mathcal{D}$, $[\![\bot]\!] = \emptyset$.

6.3.3. Examples of Constraint Domains.

6.3.3.1. *CET.* The simplest constraint solver example is obtained by taking the domain to be $\mathcal{H}$ with the necessary equality axioms to support unification formalized by Clark. Among other things, this constraint domain is capable of specifying unification problems and renaming apart.

Let t_1, t_2 be $\mathcal{T}_\Sigma(\mathcal{X})$ terms with n free variables, then their unification problem is the constraint formula $t_1 = t_2$. In this case Herbrand, equality corresponds to equality in our term model, thus we may be interpret equality by $\cup$. The problem of whether t_1, t_2 are unifiable is reduced to the satisfiability of $K(t_1) \cap K(t_2)$.

We can model a term t containing a *renamed apart* or *fresh* variable x_i with the formula $\exists x_i.x_n = t$, where x_n allows to use the term outside the quantifier. For instance, unification modulo renaming apart as done in Prolog is expressed as:

$$\exists \vec{x}.\ (x_{n+1} = t[\vec{x}]) \wedge x_n = u$$

where $t[\vec{x}]$ is a term in the head of some clause in the program, $\vec{x}$ are its n free variables, u is the term in the original query. This construction is a cornerstone for the modeling of parameter passing in the relational operational semantics.

6.3.3.2. $\mathbb{R}$. Assume $\mathcal{D}$ to be $\mathbb{R}$, the binary term former $+$ to be in Σ and the constraint predicates $\{<, =_{\mathbb{R}}\}$, interpreted as:

$$([a_1, a_2]\vec{u}, [a_1, a_2]\vec{u}') \in [\![\dot{K}(=_{\mathbb{R}})]\!]) \iff \mathbb{R} \models [\![a_1]\!]_{\mathbb{R}} = [\![a_2]\!]_{\mathbb{R}}$$
$$([a_1, a_2]\vec{u}, [a_1, a_2]\vec{u}') \in [\![\dot{K}(<)]\!] \iff \mathbb{R} \models [\![a_1]\!]_{\mathbb{R}} < [\![a_2]\!]_{\mathbb{R}}$$

These two predicates also interpret the symbol $+$ as addition in $\mathbb{R}$, so $[2, 1 + 1]\vec{u} \in [\![\dot{K}(=_{\mathbb{R}})]\!]$.

Remark 6.9. *Delegating term interpretation to the constraint solver is standard in constraint logic programming, where the constraint solver interprets some term in $\mathcal{T}_{\Sigma}(\mathcal{X})$ into an element of the corresponding $\Sigma_{\mathcal{D}}$ structure.*

The use of relations makes it easy to interpret $+$ by the real addition operation over the natural numbers. Let $[\![+]\!] = \{([a_1, a_2], [a_3]) : a_1 + a_2 = a_3\}$, then $[\![\dot{K}(<)]\!]$ is the regular less than relation, whereas if we interpret $[\![+]\!] = \{([a_1, a_2], [a_3]) : a_3 = +^A(a_1, a_2)\}$ then $[\![\dot{K}(<)]\!]$ is forced to contain tuples like $(+^A(1, 3), +^A(2, 3))$, etc…

6.3.4. Translation of Constraint Logic Programs to Relations. The translation procedure for a Constraint Logic Program is a modified version of the translation for constraints. This is necessary to handle the two main differences between CLP and pure constraint reasoning: the existence of recursively defined predicates, and the existence of disjunctive clauses. We use a sightly different scheme for existential quantification occurring in the tail of the clauses, replacing the quasi-identity relation Q_i by a partial identity relation I_n.

We process the program to obtain its completed database form as defined in [Clark, 1977]. This reflects the fact that we want to work with respect to the minimal semantics of a program. The main syntactic difference of the completed database is that we require all atoms referring to defined predicates to be pure, that is to say, no term may occur as argument. Then, we define a translation from completed programs to relational terms.

6.3.4.1. *Program Processing.* We process our program P to obtain a completed form P' suitable for translation to relational terms. Assume that variables occurring in all clauses are distinct to newly introduced variables $\vec{x} = x_1, \ldots, x_n$.

Conjunction of literals and constraints $p_1(\vec{u}_1[\vec{x}_1]), \ldots, p_n(\vec{u}_n[\vec{x}_n])$ are represented as $\vec{p}(\vec{u}[\vec{x}])$, note that p_i may refer to a defined predicate or to a constraint. $t[\vec{x}]$ exhibits the fact that a term t has all its variables in $\vec{x}$, and the same holds for vectors of terms $\vec{t}[\vec{x}]$. Given $m < n$, we write $\exists^n_m.\varphi$ for $\exists x_{m+1}, \ldots, x_n$. We write $\exists^{n\uparrow}$ for $(\exists x_{n+1}, \ldots).\varphi$, that is to say, all variables with index greater than n are existentially quantified.

The operational semantics that will be presented in Sec. 6.5 could easily allow arbitrary choices of variable names. But the variable-free nature of the relational calculus imposes a "canonical" naming to logic variables: the correspondence to the position they point to in the sequences occurring in the semantics. So a transformed clause will have all its variables ranging from 1 to n.

The operational semantics for constraint logic programming given by Def. 3.27 doesn't have a concept of substitution. The constraint store is opaque, so we must assume that any variable posted to it is still used. Under this assumption, the number of names used for variables increases monotonically with each derivation. This canonical naming scheme gives rise to a canonical renaming apart scheme: if a program state is using up to n variables, the canonical renaming apart for a variable x_i is x_{n+1}.

On the other hand, our relational system doesn't directly represent predicate calls such as $p(f(a, b))$, but the parameter information is always kept in the constraint store: $x_1 = f(a, b) \wedge \overline{p}(x_1)$. We could extend our relational language or use syntactic sugar for this purpose, but we think it is easier to use states

and clauses that are in "General Purified Form", — a name inspired by Clark's General Form concept — meaning that every state and clause is rewritten in a logically correct way so that every atom has only variables as arguments; if that is the case, we say an atom is *pure*. The first element of the tail of a clause in GPF form is a constraint $\vec{x} = \vec{t}$ for all the variables $\vec{x}$ appearing as arguments. Requiring constraints to be pure is not necessary, as purity's main reason of existence is to help in parameter passing.

Definition 6.10 (General Purified Form for Clauses)**.** *For a clause*

$$p(\vec{t}[\vec{y}]) \leftarrow \vec{q}(\vec{v}[\vec{y}])$$

let $h = \alpha(p)$*,* $y = \alpha(\vec{y})$*,* $v = \alpha(\vec{v})$*, and* $m = h + y + v$*. Define the following vectors of variables:*

$$\begin{array}{lcccccl} \vec{x} & = & \vec{x}_h\vec{x}_t & = & \vec{x}_h\vec{x}_y\vec{x}_v & = & x_1,\ldots,x_h,x_{h+1},\ldots,x_{h+y},x_{h+y+1},\ldots,x_m \\ \vec{x}_h & & & & & = & x_1,\ldots,x_h \\ \vec{x}_t & & & = & \vec{x}_y\vec{x}_v & = & x_{h+1},\ldots,x_{h+y},x_{h+y+1},\ldots,x_m \\ \vec{x}_y & & & & & = & x_{h+1},\ldots,x_{h+y} \\ \vec{x}_v & & & & & = & x_{h+y+1},\ldots,x_m \end{array}$$

then the clause's GPF form is:

$$p(\vec{x}_h) \leftarrow \exists^{h\uparrow}.((\vec{x}_h = \vec{t}[\vec{x}_y] \wedge \vec{x}_v = \vec{v}[\vec{x}_y]), \vec{q}(\vec{x}_v))$$

This definition is easily extended to programs.

As an example, take the second clause of the predicate add:

$$add(s(x), y, s(z)) \leftarrow add(x, y, z)$$

its GPF form is:

$$add(x_1, x_2, x_3) \leftarrow \exists^{3\uparrow}.(\langle x_1, x_3, x_6, x_7, x_8\rangle = \langle s(x_4), s(x_5), x_4, x_2, x_5\rangle, add(x_6, x_7, x_8))$$

We may replace redundant equations of the form $x_i = x_j$, obtaining:

$$add(x_1, x_2, x_3) \leftarrow \exists^{3\uparrow}.(x_1 = s(x_4), x_3 = s(x_5), add(x_4, x_2, x_5))$$

The GPF form is the first step in order to handle disjunctive clauses. The second is to perform Clark's completion. A predicate:

$$p \leftrightarrow cl_1 \vee \cdots \vee cl_n$$

will be translated to the relational term:

$$\overline{p} = C_1 \cup \cdots \cup C_n$$

Recall that for a (unique) clause $p \leftarrow q$, we cannot logically derive $p \rightarrow q$, thus obtaining the double implication. But as Clark noted, the correspondence holds in the minimal semantics Our relational framework uses an explicit fixpoint operator, so the validity of the completion holds.

Definition 6.11 (Completion of a Predicate)**.** *For a predicate* p *with clauses in GPF form:*

$$\begin{array}{lcl} p(\vec{x}_h) & \leftarrow & tl_1 \\ \vdots & & \\ p(\vec{x}_h) & \leftarrow & tl_k \end{array}$$

we define its completed form as

$$p(\vec{x}_h) \leftrightarrow tl_1 \vee \cdots \vee tl_k$$

Again, this definition easily extends to programs.

For the predicate add, with clauses in GPF form:

$$\begin{array}{lcl} add(x_1, x_2, x_3) & \leftarrow & (x_1 = o, x_2 = x_3) \\ add(x_1, x_2, x_3) & \leftarrow & \exists^{3\uparrow}.(x_1 = s(x_4), x_3 = s(x_5), add(x_4, x_2, x_5)) \end{array}$$

its completed form:

$$\begin{array}{rcl} add(x_1, x_2, x_3) & \leftrightarrow & (x_1 = o, x_2 = x_3) \\ & \vee & \exists^{3\uparrow}.(x_1 = s(x_4), x_3 = s(x_5), add(x_4, x_2, x_5)) \end{array}$$

The relational translation for a completed predicate is then almost straightforward. Constraints are translated using $\dot{K}$, conjunction is mapped to $\cap$ and $\vee$ to $\cup$. Thus, the translation only needs to take care of the existential quantifier occurring in some of the tails and recursive definitions and parameter passing.

6.3.4.2. *Existential Quantification or Local Variables.* Variables *local* to a clause are existentially quantified. We could mimic the method used in constraint translation, using a Q_i relation. However, although the definition of Q_i doesn't pose a problem for constraints given that the *black-box* constraint logic model frees us from having to deal with the operational issues of the constraint solver, for defined predicates this is not the case. A *partial identity* relation, which act as an existential quantifier for all variables of index greater than a given number, will prove very useful for our purposes.

Definition 6.12. *The relation of partial identity up to the first n components is defined as:*

$$I_n := \bigcap_{1 \le i \le n} P_i (P_i)^\circ$$

with semantics

$$(\vec{a}\vec{u}, \vec{a}\vec{u}') \in [\![I_n]\!] \iff |\vec{a}| = n \quad where \quad \vec{a}, \vec{u}, \vec{u}' \in A^\dagger$$

The logical effect this new relation has is to "hide" or existentially quantify all the variables higher than n.

Remark 6.13. *We use I_n for the translation thanks to the careful definition of the GPF: for a predicate of arity n, all local variables are clustered together to have an index greater than n. Indeed, it is easy to check that $[\![I_n]\!]$ coincides with $\bigcup_{i>n} [\![Q_i]\!]$.*

6.3.4.3. *Recursive Definitions.* Recursive definitions such as the one of add would be usually translated using the fixed point operator **fp**. That is to say, for a relational term E representing a recursive predicate, we would use a relational variable as a placeholder and construct a fixed point equation. Imagine $E(R)$ is the translation of add:

$$\overline{add} \equiv \mathbf{fp}\ R.\ E(R)$$

When rewriting, we are forced to use capture-avoiding substitution in order to unfold the fixed point operator. However, given that predicate names are unique, we may remove the need for capture-avoiding substitution as such as no clash will occur if we name the variables appropriately. The existence of substituion is still a problem when rewriting.

The solution chosen is to use a *definitional approach*. For every predicate p, calling $p_1, \ldots, p_n$ we add a ground relational term $\overline{p}$ and an equation to $\mathbf{QRA}_\Sigma$:

$$\overline{p} = E(\overline{p_1}, \ldots, \overline{p_n})$$

So we incorporate the recursive nature of predicate definitions into $\mathbf{QRA}_\Sigma$ and we avoid the use of a fixpoint operator in our translation. This approach is clearer for the reader, sound for our purposes and compatible with rewriting.

6.3.4.4. *Parameter Passing.* Calls to a defined predicate $p_i(\vec{x}_i)$ occurring in a tail are translated to the corresponding predicate name $\overline{p_i}$ plus a permutation. The permutation is needed as in general variables $\vec{x}_i$ won't occur in the right order, as the add example shows, with a call to $add(x_4, x_2, x_5)$. If we define a permutation $w = (41)(52)$, the vector $\vec{x} = \langle x_1, x_2, x_3, x_4, x_5 \rangle$ is permuted to $w(\vec{x})) = \langle x_4, x_2, x_5, x_1, x_3 \rangle$. Indeed, we define a variable permutation for constraint formulas as follows:

Definition 6.14 (Variable Permutation)**.** *Given a permutation $\pi : \{1..n\} \to \{1..n\}$, the w_π function over formulas and terms is defined from π as:*

$$\begin{array}{ll} w_\pi(x_i) & = x_{\pi(i)} \\ w_\pi(t(t_1, \ldots, t_n) & = t(w_\pi(t_1), \ldots, w_\pi(t_n)) \\ w_\pi(P(t_1, \ldots, t_n)) & = P(w_\pi(t_1), \ldots, w_\pi(t_n)) \\ w_\pi(\varphi \wedge \psi) & = w_\pi(\varphi) \wedge w_\pi(\psi) \\ w_\pi(\exists x_i.\, \varphi) & = \exists x_{\pi(i)}.\, w_\pi(\varphi) \end{array}$$

The corresponding relation is given in the following definition:

Definition 6.15 (Switching Relation)**.** *The* switching relation W_π*, associated to a permutation π is:*

$$W_\pi = \bigcap_{j=1}^{n} P_{\pi(j)}(P_j)^\circ.$$

Lemma 6.16. *Fix a permutation π and its associated logic and relational version w and W. Then:*

$$[\![\dot{K}(w(P(x_1, \ldots, x_n)))]\!] = [\![W\dot{K}(P)W^\circ]\!]$$

Proof. Straightforward, This is just a restatement of the claim:

$$(\vec{a}\vec{u}, \vec{a}\vec{u}) \in R \iff (\vec{a}'\vec{u}, \vec{a}'\vec{u}) \in [\![WRW^\circ]\!]$$

where $\vec{a} = a_1, \ldots, a_n$ and $\vec{a}' = a_{\pi(1)}, \ldots, a_{\pi(n)}$. □

Thus, the call to $add(x_4, x_2, x_5)$ is relationally translated to $W_\pi; \overline{add}; W_\pi^\circ$, for $\pi = (41)(53)$.

6.3.4.5. *The Translation Function.* We are ready to define the translation for predicates:

Definition 6.17 (Relational Translation of Predicates)**.** *The translation function Tr from completed predicates to relational equations is:*

$$Tr(p(\vec{x}_h) \leftrightarrow cl_1 \vee \cdots \vee cl_k) = (\overline{p} = Tr_{cl}(cl_1) \cup \cdots \cup Tr_{cl}(cl_k))$$

where

$$Tr_{cl}(\exists^{h\uparrow}.\vec{p}) = I_h; (Tr_l(p_1) \cap \cdots \cap Tr_l(p_n)); I_h$$

where

$$Tr_l(\varphi) = \dot{K}(\varphi) \qquad Tr_l(p_i(\vec{x}_i)) = W_\pi; \overline{p_i}; W_\pi^\circ \quad \textit{such that } \pi(\vec{x}_i) = x_1, \ldots, x_{\alpha(p_i)}$$

Given the symmetrical occurrence of the W, P and I terms and their reciprocals, we may abuse notation and write them as unary operators on the left:

$$I_n(R) \equiv I_n; R; I_n \qquad W_\pi(R) \equiv W_\pi; R; W_\pi^\circ$$

Add Example: Recall that the completed form for add is:

$$\begin{array}{lll} add(x_1, x_2, x_3) & \leftrightarrow & (x_1 = o, x_2 = x_3) \\ & \vee & \exists^{3\uparrow}.(x_1 = s(x_4), x_3 = s(x_5), add(x_4, x_2, x_5)) \end{array}$$

then, the recursively defined relational term resulting of its translation is:

$$\overline{add} = \dot{K}(x_1 = o \wedge x_2 = x_3) \cup I_3((\dot{K}(x_1 = s(x_4) \wedge x_3 = s(x_5)) \cap W(\overline{add})))$$

6.3.5. Adequacy of Program Translation. We show that the translation preserves the intended meaning of the program. To this end, we related the semantics defined in Sec. 3.4 with the set-theoretical interpretation of a translated CLP program.

Recall that a translated program is a set of recursive definitions for each predicate:

$$\begin{array}{ll}\overline{p}_1 = & F_1(\overline{p}_{1_1}, \dots, \overline{p}_{k_1}) \\ \dots & \\ \overline{p}_n = & F_n(\overline{p}_{1_n}, \dots, \overline{p}_{k_n})\end{array}$$

where F is a term over the relational signature, example $F(\overline{p}_1, \overline{p}_2) = \overline{p}_1 \cap I_n \overline{p}_2 I_n$.

We iterate this set of equations with an interpretation operator in order to construct the semantics. We will always interpret predicate symbols as the empty set, $[\![\overline{p}]\!] = \emptyset$. The iteration operator substitutes all the $\overline{p}$ symbols by their definition. The program is unfolded using the set of defining equation and we obtain a correspondence with the classical fix-point semantics for CLP.

Definition 6.18. *The relational substitution operator $R_P(E)$ takes a set of relational definitions and returns an augmented set of definitions in following way: for every equation $\overline{p} = F(\overline{p}_1, \dots, \overline{p}_n) \in E$, add a new equation to the output:*

$$\overline{p} = F(F_1(\overline{p}_{1_1}, \dots, \overline{p}_{k_1}), \dots, F_n(\overline{p}_{1_n}, \dots, \overline{p}_{k_n}))$$

where $\overline{p}_i = F_i(\overline{p}_{1_i}, \dots, \overline{p}_{k_i}) \in E$.

Definition 6.19. *Given the set of equations E:*

$$\begin{array}{ll}\overline{p}_1 = & F_1(\overline{p}_{1_1}, \dots, \overline{p}_{k_1}) \\ \dots & \\ \overline{p}_n = & F_n(\overline{p}_{1_n}, \dots, \overline{p}_{k_n})\end{array}$$

the semantics for a relational program $\{\![E]\!\}$ is the extension of $[\![]\!]$ to E:

$$\begin{array}{ll}\overline{p}_1 = & [\![F_1(\overline{p}_{1_1}, \dots, \overline{p}_{k_1})]\!] \\ \dots & \\ \overline{p}_n = & [\![F_n(\overline{p}_{1_n}, \dots, \overline{p}_{k_n})]\!]\end{array}$$

We study how the semantics of the original program and its translation relate. For simplicity we assume a program with one recursive predicate, but the argument is easily extend to multiple predicates.

For a predicate p with n clauses, its completed form is $p \leftrightarrow cl_1 \vee \dots \vee cl_n$, with the relational translation $\overline{p} = F(\overline{p})$, where $F(x) = \Theta_1 \cup \ldots \cup \Theta_n$. Depending if the i-th clause was a fact when non-completed, each Θ_i will have the following form:

- If it was a fact, its is of the form $p(\vec{x}) \leftarrow \varphi$. Its relational counterpart is $\dot{K}(\varphi)$.
- If it was a regular clause, it is of the form $p(\vec{x}) \leftarrow \varphi_i, p(\vec{y})$. Its relational counterpart is $I_n(\dot{K}(\varphi_i) \cap W(p))I_n$.

Lemma 6.20. $[\![I_n(\dot{K}(\varphi_i) \cap W(\overline{p}))I_n]\!] = \emptyset$

Proof. $[\![\overline{p}]\!] = \emptyset$, the lemma follows by properties of relational composition and intersection. $\square$

In order to understand how the scheme works, assume the definition of $\overline{p}$ to be the following equation:

$$\overline{p} = \dot{K}(\varphi_1) \cup I_n(\dot{K}(\varphi_2) \cap W(\overline{p}))$$

Applying the substitution operator we obtain:

$$\overline{p} = \dot{K}(\varphi_1) \cup I_n(\dot{K}(\varphi_2) \cap (W(\dot{K}(\varphi_1) \cup I_n(\dot{K}(\varphi_2) \cap W(\overline{p})))))$$

which is equivalent to:

$$\dot{K}(\varphi_1) \cup I_n(\dot{K}(\varphi_2) \cap W(\dot{K}(\varphi_1))) \cup I_n(\dot{K}(\varphi_2) \cap I_n(\dot{K}(\varphi_2) \cap W(\overline{p})))$$

with the part relevant for the meaning being:

$$\dot{K}(\varphi_1) \cup I_n(\dot{K}(\varphi_2) \cap W(\dot{K}(\varphi_1)))$$

as the last part contains the symbol $\overline{p}$ which is interpreted as $\emptyset$. This way, the relational semantics are built by successively accumulating constraints.

Theorem 6.21 (Adequacy of the relational translation)**.** *Given* $\langle p(\vec{x}), \varphi_i \rangle \in T_P^n(\emptyset)$*,* $1 <= i <= k$*, and* $\overline{p} = F(\overline{p}) \in R_E^n(PL)$*, where* PL *is the original translation of the program. Let* $|\vec{a}| = \alpha(p)$*. Then,* $(\vec{a}\vec{u}, \vec{a}\vec{u}) \in [\![F(\overline{p})]\!] \iff \varphi_1(\vec{a}) \vee \ldots \vee \varphi_k(\vec{a})$*.*

Proof. By induction over n. Base case is immediate, The T_P operator will add all the tailless clauses, which correspond to the semantics of the initial relational equations.

For the inductive case, we have that the theorem holds for n. The T_P operator for the clause $p(\vec{x}) \leftarrow \varphi, p(\vec{y})$ and each $\langle p(\vec{x}), \varphi_i \rangle$ will construct new pairs $\langle p(\vec{x}), c \rangle$ such that

$$c = \exists.\vec{x}\ (\varphi \wedge (p(\vec{z}) = p(\vec{y})) \wedge \varphi_i)$$

whereas the relational instantiation operator will take the existing equation

$$\overline{p} = \dot{K}(\varphi_1) \cup \ldots \cup \dot{K}(\varphi_k) \cup I_n(\dot{K}(\varphi) \cap W(\overline{p}))$$

to

$$\overline{p} = \dot{K}(\varphi_1) \cup \ldots \cup \dot{K}(\varphi_k) \cup I_n(\dot{K}(\varphi) \cap W(\dot{K}(\varphi_1) \cup \ldots \cup \dot{K}(\varphi_k) \cup I_n(\dot{K}(\varphi) \cap W(\overline{p}))))$$

which is semantically equivalent to:

$$\dot{K}(\varphi_1) \cup \ldots \cup \dot{K}(\varphi_k) \cup \dot{K}(\exists^{n\uparrow}.\ \varphi \wedge w(\varphi_1)) \cup \cdots \cup \dot{K}(\exists^{n\uparrow}.\ \varphi \wedge w(\varphi_k))$$

Each $\dot{K}(\exists^{n\uparrow}.\ \varphi \wedge w(\varphi_1))$ corresponds to the newly added $c = \exists.\vec{x}\ (\varphi \wedge (p(\vec{z}) = p(\vec{y})) \wedge \varphi_i)$. This completes the proof.

In Sec. 6.5 this last correspondence is explored in more detail. □

6.4. Computing with Relations: Operational Semantics

We have established the declarative semantics for our translated programs, but we lack an effective method to compute an answer to a query. In this section, we will develop a rewriting system based on the equational theory $\mathbf{QRA}_\Sigma$, which will be proven equivalent to the traditional operational semantics for CLP.

6.4.1. Rewriting signature. The representation of relational terms in our rewriting systems is given by the following term-forming operations over the sort $\mathcal{T}_R$ of relational terms for rewriting: $\mathsf{I} : (\mathbb{N} \times \mathcal{T}_R) \to \mathcal{T}_R$, $\mathsf{W} : (\mathsf{Perm} \times \mathcal{T}_R) \to \mathcal{T}_R$, $\mathsf{K} : \mathcal{L}_\mathcal{D} \to \mathcal{T}_R$, $\odot : (\mathcal{T}_R \times \mathcal{T}_R) \to \mathcal{T}_R$, $\cup : (\mathcal{T}_R \times \mathcal{T}_R) \to \mathcal{T}_R$. and $\cap : (\mathcal{T}_R \times \mathcal{T}_R) \to \mathcal{T}_R$.

In addition to the above ground representation, we define patterns over $\mathcal{T}_R$-terms in a standard way using a set of variables: let i, j, k, n, m etc, range over $\mathbb{N}$; let w_i, w_j etc, range over Perm; let R, S, T etc, range over $\mathcal{T}_R$-terms. We write $I_n(R)$ for $\mathsf{I}(n, A)$, write $W_i(R)$ for $\mathsf{W}(w_i, A)$, write $\dot{K}(\vec{t})$ for $\mathsf{K}(\vec{t})$, write RS for $\odot(R, S)$, write $R \cup S$ for $\cup(R, S)$, and write $R \cap S$ for $\cap(R, S)$.

Remark 6.22. *We give term former operators for* $\mathcal{T}_R$ *in order to comply with standard rewriting practice. As previously stated, we use the same notation for terms of the rewriting system and terms in the mathematical world of relation algebra. This can been seen as an extension to "allow" rewriting over the mathematical theory. For convenience we write* $I_n(R)$ *for* $I_n R I_n$*,* $W_i(R)$ *for* $W_i R W_i^\circ$ *and* $W_i^\circ(R)$ *for* $W_i^\circ R W_i$*.*

Rewrite rules are of the form $\rho : l \to r$ where ρ is the rule's name (optional), l and r patterns, and l not a variable.

A rewrite rule $\rho : l \to r$ matches a term t iff $l\sigma = t$. We allow subterm matching: if there exists a position p such that $l\sigma = t_{|p}$, then t reduces to $t[r\sigma]_{|p}$. When matching succeeds, t rewrites to $r\sigma$.

When a the term being reduced, I may happen that more than one reducible position in the term or *redex* exists. We call a rewriting strategy non-deterministic when the redex can be freely chosen. We say a strategy is parallel-outermost when the redex chosen is not a subterm of any other redex and if various of those redexes exists, all of them are reduced.

We say a strategy is left-outermost if it tries to reduce the outermost term, and if this is not possible then selects the leftmost redex of the tree of reduction candidates for the sub-terms.

6.4.2. Rewriting and the Modular Law. A critical equation for the simulation of resolution is the *modular law*. This example illustrates how the use of the modular law can improve termination properties when we consider relation terms as programs that can be evaluated.

Let us consider the binary relations over the set of closed terms induced by the signature $a, b, s^1, [\cdot]^2$, that is to say, two constants, a unary successor function and a pairing function. Define $A \equiv \dot{K}([s(s(a)), a)])$ and $R \equiv \dot{K}([s(X), X])$, that is to say, the successor relation. The transitive closure R^* is also defined. We assume that membership in R is computed in one step by a black box, in the following way:

$$\dot{K}([s(a), a]) \cap R \mapsto \dot{K}([s(a), a]) \qquad \dot{K}([a, b]) \cap R \mapsto \mathbf{0}$$

Thus we want to compute the value of $A \cap (R \cup R^2 \cup \cdots \cup R^n \cup \cdots)$, to obtain either the result $\mathbf{0}$ or $A \equiv \dot{K}([s(s(a)), a])$. If we proceed in the obvious way, we obtain.

$$A \cap R \ \cup \ A \cap R^2 \ldots$$

Proceeding from left to right, the computation will terminate if $A \cap R^k = A$ for some k. In fact, this will occur when $k = 2$. But what if we pick a term, such as $B \equiv \dot{K}([a, b])$ which does not lie in R^*? If we proceed in the same way we require a non-terminating computation to yield the output $\mathbf{0}$. For example, after n steps we have the term:

$$\mathbf{0} \cup \cdots \cup \mathbf{0} \cup B \cap R^n \ldots.$$

To solve this problem we introduce a new rewrite rule corresponding to the *equational left-modular law*:

$$T \cap RS \mapsto R(R^o T \cap S) \cap T.$$

We will also add the rule $R^* \mapsto R \cup RR^*$, in order to work with a definition of transitive closure, and avoid writing infinite expressions. We then unfold the computation

$$\begin{array}{rcl}
\dot{K}([a, b]) \cap R^* & \mapsto & \dot{K}([a, b]) \cap (R \cup RR^*) \\
& \mapsto & \dot{K}([a, b]) \cap R \ \cup \ \dot{K}([a, b]) \cap (RR^*) \\
& \mapsto & \mathbf{0} \cup \ R(R^\circ \dot{K}([a, b]) \cap R^*) \cap \dot{K}([a, b]) \\
& \mapsto & \mathbf{0} \cup \ R(\mathbf{0} \cap R^*) \cap \dot{K}([a, b]) \\
& \mapsto & \mathbf{0} \cup \ \mathbf{0} \\
& \mapsto & \mathbf{0}
\end{array}$$

If R, S are coreflexive relations, $S; R = S \cap R$, so $R^\circ; \dot{K}([a, b]) = R^\circ \cap \dot{K}([a, b]) = \mathbf{0}$.

6.4.3. Meta-reductions. Our choice to abstract away the notion of constraint solver means that our rewriting engine doesn't know how to perform satisfiability checking.

We formalize the interface as meta-reductions. Each time such a reduction is required, the rewriting engine will call the constraint solver, which will return the appropriate reply. From the rewriting system point to view, the constraint solver remains a black-box.

$$
\begin{array}{llll}
m_1 : I_m(\dot{K}(\psi)) & \overset{P}{\longmapsto} & \dot{K}(\exists^{m\uparrow}.\,\psi) & x_i \in \vec{x} \iff i > m \\
m_2 : W_i(\dot{K}(\psi)) & \overset{P}{\longmapsto} & \dot{K}(w_i(\psi)) & \\
m_3 : \dot{K}(\psi_1) \cap \dot{K}(\psi_2) & \overset{P}{\longmapsto} & \dot{K}(\psi_1 \wedge \psi_2) & \mathcal{D} \models \psi_1 \wedge \psi_2 \\
m_3 : \dot{K}(\psi_1) \cap \dot{K}(\psi_2) & \overset{P}{\longmapsto} & \mathbf{0} & \mathcal{D} \not\models \psi_1 \wedge \psi_2
\end{array}
$$

FIGURE 6.1. Constraint meta-reductions

$$
\begin{array}{lll}
R \cap (S \cap T) & \overset{P}{\longmapsto} & (R \cap S) \cap T \\
R \cap (S \cup T) & \overset{P}{\longmapsto} & (R \cap S) \cup (R \cap T) \\
(R \cup S) \cap T & \overset{P}{\longmapsto} & (R \cap T) \cup (S \cap T) \\
R \cap W_i(S) & \overset{P}{\longmapsto} & W_i(W_i^{\circ}(R) \cap S) \\
R \cap I_n(S) & \overset{P}{\longmapsto} & I_n(I_n(R) \cap S) \cap R \\
\overline{q} & \overset{P}{\longmapsto} & \Theta_1 \cup \cdots \cup \Theta_n \\
\mathbf{fp}x.\mathcal{E}(x) & \overset{P}{\longmapsto} & \mathcal{E}(\mathbf{fp}x.\mathcal{E}(x))
\end{array}
$$

FIGURE 6.2. Preliminary rewriting rules for SLD_1 simulation.

The main meta-reductions are presented in Fig. 6.1. A proposed interface (using Haskell notation) may look like this:

```
restrict :: Int         -> CStore -> CStore
rename   :: Permutation -> CStore -> CStore
check    :: CStore      -> CStore -> Maybe CStore
prim     :: CPred       -> CStore
```

Lemma 6.23. *All meta-reductions are sound.*

PROOF. By soundness of constraint translation. □

6.4.4. Rewriting system for SLD. Orienting an specialized version of relational theory equations, we get a rewriting system with a notion of *normal form*, such that the translation for a query reaches normal form iff a finite branch or *answer* in the SLD tree exists. Additionally the notion of irreducible term coincides with the notion of a finite (or closed) SLD tree.

The approach of rewriting in relation algebra was proposed in [Broome and Lipton, 1994], and the first concrete rewriting system proposal for SLD programs appears in [Lipton and Chapman, 1998], and while completeness is conjectured some important details were missing.

A SLD theory. The spirit of SLD resolution is captured with the set of rules shown in Fig. 6.2. Control strategy is reflected as the distributivity of $\cap$ over $\cup$ — the first three rules. Parameter passing and eager failure are carried out by the fourth and fifth reductions, and the existence of recursive computations is handled by the two last rules, which are equivalent.

This "simple" rewriting system captures SLD resolution and is easy to read, but its rewriting theoretical properties are not good enough for actual use. The system is highly non-confluent and too dependent on careful application of the rules using a particular order.

In order to avoid the theoretical problems, we define a specialized rewriting system in Fig. 6.3. Achieving confluence is not possible, given that proof search may not terminate, but we prove local confluence and will show in Sec. 6.5 that the system has a normal form if the associated SLD-tree of the program has a finite branch. Local confluence plus termination implies confluence.

$$
\begin{array}{lll}
m_1 & : I_m(\dot{K}(\psi)) & \overset{P}{\longmapsto} \dot{K}(\exists^{m\uparrow}.\psi) \\
m_1* & : I_m(\mathbf{0}) & \overset{P}{\longmapsto} \mathbf{0} \\
m_2 & : W_\pi(\dot{K}(\psi)) & \overset{P}{\longmapsto} \dot{K}(w_\pi(\psi)) \\
m_2* & : W_\pi(\mathbf{0}) & \overset{P}{\longmapsto} \mathbf{0} \\
m_3 & : \dot{K}(\psi_1) \cap \dot{K}(\psi_2) & \overset{P}{\longmapsto} \dot{K}(\psi_1 \wedge \psi_2) \\
m_3* & : \dot{K}(\psi_1) \cap \dot{K}(\psi_2) & \overset{P}{\longmapsto} \mathbf{0} \\
m_4 & : \dot{K}(\psi) \cap \overline{q} & \overset{P}{\longmapsto} \dot{K}(\psi) \cap (\Theta) \\
p_1 & : \mathbf{0} \cup R & \overset{P}{\longmapsto} R \\
p_2 & : \mathbf{0} \cap R & \overset{P}{\longmapsto} \mathbf{0} \\
p_3 & : W_\pi(R \cup S) & \overset{P}{\longmapsto} W_\pi(R) \cup W_\pi(S) \\
p_4 & : I_n(R \cup S) & \overset{P}{\longmapsto} I_n(R) \cup I_n(S) \\
p_5 & : (R \cup S) \cap T & \overset{P}{\longmapsto} (R \cap T) \cup (S \cap T) \\
p_6 & : \dot{K}(\psi) \cap (R \cup S) & \overset{P}{\longmapsto} (\dot{K}(\psi) \cap R) \cup (\dot{K}(\psi) \cap S) \\
p_7 & : \dot{K}(\psi) \cap (R \cap W_\pi(\overline{q_i})) & \overset{P}{\longmapsto} (\dot{K}(\psi) \cap R) \cap W_\pi(\overline{q_i}) \\
p_8 & : \dot{K}(\psi) \cap W_\pi(\overline{q}) & \overset{P}{\longmapsto} W_\pi^\circ(W_\pi(\dot{K}(\psi)) \cap \overline{q}) \\
p_9 & : \dot{K}(\psi) \cap I_m(R) & \overset{P}{\longmapsto} I_m(I_m(\dot{K}(\psi)) \cap R) \cap \dot{K}(\psi)
\end{array}
$$

FIGURE 6.3. Rewriting system for SLD.

Lemma 6.24. *In the equational theory QRA, from $SS^\circ \subset id$ we can infer $A \cap SR = S(S^\circ A \cap R)$. From $S^\circ S \subset id$ we can infer $A \cap RS = (AS^\circ \cap R)S$.*

PROOF. By the modular law we have, in the first case, $A \cap SR = S(S^\circ A \cap R) \cap A$. But $S(S^\circ A \cap R) \subseteq SS^\circ A \cap SR \subseteq idA \cap SR = A \cap SR$. Thus $S(S^\circ A \cap R) \cap A$ reduces to $S(S^\circ A \cap R)$. The argument for the second claim is symmetric. □

Lemma 6.25. *The rewriting system is sound.*

PROOF. All of the rules are easy consequences of relation algebra, except for p_9. For p_9, we apply the equational version of the modular law to obtain the derivation:

$$
\begin{array}{ll}
\dot{K} \cap IRI & =_{[I\dot{K}I \supseteq \dot{K}]} \\
\dot{K} \cap I\dot{K}I \cap IRI & \subseteq_{[RS \cap T \subseteq (R \cap TS^\circ)S]} \\
\dot{K} \cap (IR \cap I\dot{K}II^\circ)I & \subseteq_{[RS \cap T \subseteq R(R^\circ T \cap S)]} \\
\dot{K} \cap I(R \cap I^\circ I\dot{K}II^\circ)I & =_{[I^\circ I = I]} \\
\dot{K} \cap I(R \cap I\dot{K}I)I &
\end{array}
$$

The opposite direction $\dot{K} \cap IRI \supseteq \dot{K} \cap I(I\dot{K}I \cap R)I$ is immediate. □

Lemma 6.26. *The rewriting system in Fig. 6.3 is locally confluent iff we give higher priority to p_7 over p_8.*

PROOF. We study critical pairs and prove that all the existing ones join. Our systems has three critical pairs:

- m_1 overlaps with p_8, so using p_8: $\dot{K}(\psi_1) \cap I_m(\dot{K}(\psi_2)) \overset{P}{\longmapsto} I_m(I_m(\dot{K}(\psi_1)) \cap \dot{K}(\psi_2)) \cap \dot{K}(\psi_1) \overset{P}{\longmapsto} I_m(\dot{K}(\exists^{m\uparrow}.\psi_1) \cap \dot{K}(\psi_2)) \cap \dot{K}(\psi_1) \overset{P}{\longmapsto} I_m(\dot{K}(\exists^{m\uparrow}.\psi_1 \wedge \psi_2)) \cap \dot{K}(\psi_1) \overset{P}{\longmapsto} \dot{K}(\exists^{m\uparrow}.(\exists^{m\uparrow}.\psi_1 \wedge \psi_2)) \cap \dot{K}(\psi_1) \overset{P}{\longmapsto} \dot{K}(\exists^{m\uparrow}.(\exists^{m\uparrow}.\psi_1 \wedge \psi_2) \wedge \psi_1)$ which is logically equivalent to $\dot{K}(\psi_1 \wedge \exists^{m\uparrow}.\psi_2)$, that we obtain reducing with m_1.
- p_1 overlaps with p_5, so using p_5: $\dot{K}(\psi) \cap (\mathbf{0} \cup R) \overset{P}{\longmapsto} (\dot{K}(\psi) \cap \mathbf{0}) \cup (\dot{K}(\psi) \cap R) \overset{P}{\longmapsto} \mathbf{0} \cup (\dot{K}(\psi) \cap R) \overset{P}{\longmapsto} \dot{K}(\psi) \cap R$, which is what we get using p_1 directly.

- p_7 overlaps with p_8, and indeed this overlapping is not solvable without assigning a priority to some of the rules. The overlapping term is of the form $\dot{K}(\psi_1) \cap (\dot{K}(\psi_2) \cap W(\overline{q}))$, and p_7 has higher priority than p_8 this gets rewritten to $(\dot{K}(\psi_1) \cap \dot{K}(\psi_2)) \cap W(\overline{q})$ which leads to a non-problematic term $\dot{K}(\psi_1 \wedge \psi_2) \cap W(\overline{q})$.

□

Corollary 6.27. *The two first cases of overlapping can be eliminated by a transformation of the system.*

Remark 6.28. *A left-outermost rewriting strategy always gives priority to p_7 over p_8.*

So in order to simulate SLD, the rewriting system must be used under the *left outermost strategy* defined earlier.

Lemma 6.29. *No orthogonal rewriting exists for resolution using the signature presented.*

Proof. For we cannot have an orthogonal rewriting system that can carry out the $\dot{K} \cap W(q) \to I((I(\dot{K}') \cap \dot{K}) \cap W(p))$ reduction to if we want to keep the $\dot{K} \cap \dot{K} \to \dot{K}$ rewriting rule. This is mainly due to the fact that our signature doesn't distinguish if a term $\dot{K}$ comes from a head of a clause or from a query. □

6.5. Operational Equivalence

We prove that rewriting relational terms is equivalent to "traditional" SLD operational semantics such as the ones in [Jaffar and Maher, 1994, Comini et al., 2001].

First, we modify the classical operational semantics in order to obtain a transition system without the meta-logical operation of renaming apart. For this purpose, we require all clause definitions and states to be in *purified* form, where every predicate occurs of the form $p(\vec{x})$ with $\vec{x}$ variables. This concept is in close correspondence to Clark's general form. A new transition system which doesn't use a renaming apart operation is defined and proven equivalent to the standard one.

Then, we define a folding of all derivations from a state s to states $\{s_1, \dots, s_n\}$ into a new *resolution state* $s \to \cdots \to (s_1 \,\|\, \dots \,\|\, s_n)$. A resolution state captures all the proof search information needed for resolution. When a state has no further derivation, it is deleted and the next one is selected. We define a new transition system for this notion of state and prove its operational equivalence with the traditional one. Some important optimizations like linear-selection must be reflected.

The resulting transition system for a given *resolution state* is deterministic. Every successful query has a unique derivation which reflects all the proof search needed for reaching success. This is in contrast to the classical approach, where a query produces a set of non-successful derivations plus a successful one.

The rewriting system presented in Sec. 6.4 induces a transition system for relational expressions. We establish an isomorphism between resolution states and relational expressions and prove that there is a simulation between both transition systems.

6.5.1. Operational Semantics in Logic Style for SLD-resolution. Traditional operational semantics for constraint logic programming is defined as a non-deterministic transition system between states representing queries. Operational issues like clause selection order, returning from a predicate call, and backtracking are not made explicit.

6.5.1.1. *General Purified From for Program States.* We use the convention of writing a constraint store without its top level implicit existential quantifier. Recall that a constraint $\varphi[\vec{x}]$ is said to be satisfiable iff $\mathcal{D} \models \exists\vec{x}.\varphi[\vec{x}]$. In our case the top level quantifier is always redundant so we will omit it unless otherwise noted.

We write GPF for general purified form. For a state Q, we write Q' for its GPF form, and for a program P, we write P' for its GPF form as defined in Sec. 6.3. We now define the purified forms for states:

Definition 6.30 (General Purified Form for Program States)**.** *For a program state*

$$\langle \vec{p}(\vec{u}[\vec{x}]) \mid \varphi[\vec{x}] \rangle$$

with $\vec{x} = x_1, \ldots, x_m$*, let* $k = \alpha(\vec{u})$ *and let* $\vec{x}' = x_{m+1}, \ldots, x_{m+k}$*. Then its GPF is:*

$$\langle \vec{p}(\vec{x}') \mid \varphi[\vec{x}] \wedge \vec{x}' = \vec{u}[\vec{x}] \rangle$$

Lemma 6.31. *Let* φ *be the constraint store of a state* Q*, and* φ' *the constraint store of* Q'*. Then,* $\mathcal{D} \models \varphi$ *iff* $\mathcal{D} \models \varphi'$*.*

Proof. A consequence of soundness. Take a formula $\exists \vec{x}.\varphi$, then, for $\vec{x}'$ fresh, and any sequence of terms $\vec{t}$ from $\mathcal{T}_\Sigma(\mathcal{X})$, $\exists \vec{x}.\varphi \iff \exists \vec{x}.\varphi \wedge \vec{t} = \vec{t} \iff (\exists \vec{x}.\varphi \wedge \vec{x}' = \vec{t})[\vec{x}'/\vec{t}] \iff \exists \vec{x}\vec{x}'.\varphi \wedge \vec{x}' = \vec{t}$. □

The GPF transformation creates states that just differ from the original ones in the number of variables in use, but otherwise they are equivalent from the operational point of view. In order to capture this equivalence we need to define a notion of state equivalence.

Definition 6.32 (State Equivalence)**.** *Let*

$$Q_1 = \langle \vec{p}(\vec{t}[\vec{x}_1]) \mid \psi_1[\vec{x}_1] \rangle$$
$$Q_2 = \langle \vec{p}(\vec{t}[\vec{x}_2]) \mid \psi_2[\vec{x}_2] \rangle$$

be two states with the same resolvent but possibly different set of variables and constraint stores. We say they are equivalent and write $Q_1 \approx_{\mathcal{D}} Q_2$ *when* $\mathcal{D} \models \exists \vec{x}_1\ \psi_1[\vec{x}_1]$ *iff* $\mathcal{D} \models \exists \vec{x}_2\ \psi_1[\vec{x}_2]$*.*

Lemma 6.33. *Let* $Q_1 \approx_{\mathcal{D}} R_1$*.* $Q_1 \to Q_2$ *iff for some state* R_2*,* $R_1 \to R_2$ *and* $Q_2 \approx_{\mathcal{D}} R_2$*.*

Proof. Immediate consequence of the soundness of the constraint solver. The same resolvent guarantees that the choice of every step is identical. Then, for every step, either a resolution or a constraint one, we have $\psi_1 \iff \psi_2$, thus for a newly added constraint φ arising from either a resolution or a constraint step, it is the case that $\psi_1 \wedge \varphi \iff \psi_2 \wedge \varphi$. □

The above lemma allows us to take a quotient of the transition relation over states that are equivalent, given that $\approx_{\mathcal{D}}$ is (trivially) an equivalence relation. This operational equivalence is fundamental in relating a state C_n belonging to a derivation starting from a state Q'_1 (in GPF form) with a state Q'_n with is the GPF form of Q_n coming from non-GPF Q_1. They key of the quotient is that it allows us to consider equal states that just differ in the number of "temporary" variables arising from its GPF conversion.

Lemma 6.34 (GPF Equivalence)**.** *For a state* Q_1 *and its GPF form* Q'_1*:*

- *A derivation* $Q_1 \to_c Q_2$ *exists iff* $Q'_1 \to_c C_2$ *does and* $C_2 \approx_{\mathcal{D}} Q'_2$*.*
- *A derivation* $Q_1 \to_r Q_2$ *exists iff* $Q'_1 \to_r C_1 \to_c C_2$ *does and* $C_2 \approx_{\mathcal{D}} Q'_2$*.*

Proof. We annotate the number of variables in use in each constraint store in order to help the reader to follow the proof. Let $|\vec{x}| = m$, $|\vec{u}| = |\vec{x}'| = k$. Recall that $\vec{x}' = x_{m+1}, \ldots, x_{m+k}$ then

$$\begin{array}{lcll} Q_1 & = & \langle \vec{p}(\vec{u}[\vec{x}]) \mid \varphi[\vec{x}] \rangle & \text{[m]} \\ Q'_1 & = & \langle \vec{p}(\vec{x}') \mid \varphi[\vec{x}] \wedge \vec{x}' = \vec{u}[\vec{x}] \rangle & \text{[m+k]} \end{array}$$

We know that $Q_1 \approx_{\mathcal{D}} Q'_1$, so a derivation will always exist for Q_1 iff it exists for Q'_1. Now we check that $Q'_2 \approx C_2$.

- If $p_1 \equiv \psi$, then we have $Q_1 \to_c Q_2$ and $Q'_1 \to_c C_2$. The new states are:

$$\begin{array}{lcll} Q_2 & = & \langle \vec{p}_{|2}(\vec{u}_{|2}[\vec{x}]) \mid \varphi[\vec{x}] \wedge \psi \rangle & \text{[m]} \\ Q'_2 & = & \langle \vec{p}_{|2}(\vec{x}') \mid \varphi[\vec{x}] \wedge \psi \wedge \vec{x}' = \vec{u}_{|2}[\vec{x}] \rangle & \text{[m+k]} \\ C_2 & = & \langle \vec{p}_{|2}(\vec{x}') \mid \varphi[\vec{x}] \wedge \psi \wedge \vec{x}' = \vec{u}_{|2}[\vec{x}] \rangle & \text{[m+k]} \end{array}$$

They are the same identical state given that we don't purify constraints.

- If p_1 is a defined predicate with clause:

$$\begin{array}{lll} cl: & p_1(\vec{t}[\vec{y}]) & \leftarrow \vec{q}(\vec{v}[\vec{y}]) \\ cl': & p_1(\vec{x}_h) & \leftarrow \exists_{h+1}^{m'}.((\vec{x}_h = \vec{t}[\vec{x}_y] \wedge \vec{x}_v = \vec{v}[\vec{x}_y]), \vec{q}(\vec{x}_v)) \end{array}$$

Let $j = |\vec{y}|, \vec{x}_\sigma = x_{m+1}, \ldots, x_{m+j}, j' = j'_1 + j'_2, j'_1 = |\vec{v}|\ j'_2 = |\vec{u}_{|2}|, \vec{x}'_q = x_{m+j+1}, \ldots, x_{m+j+j'_1}$, $\vec{x}'_p = x_{m+j+j'_1+1}, \ldots, x_{m+j+j'}$. The states Q_2 and Q'_2 arising from the derivation rules are:

$$\begin{array}{lll} Q_2 & = & \langle \vec{q}(\vec{v}[\vec{x}_\sigma]), \vec{p}_{|2}(\vec{u}_{|2}[\vec{x}]) \,|\, \varphi[\vec{x}] \wedge \vec{u}_1[\vec{x}] = \vec{t}[\vec{x}_\sigma] \rangle \qquad \text{[m+j]} \\ Q'_2 & = & \langle \vec{q}(\vec{x}'_q)), \vec{p}_{|2}(\vec{x}'_p) \,|\, \varphi[\vec{x}] \wedge \vec{u}_1[\vec{x}] = \vec{t}[\vec{x}_\sigma] \wedge \vec{x}'_q = \vec{v}[\vec{x}_\sigma] \wedge \vec{x}'_p = \vec{u}_{|2}[\vec{x}] \rangle \qquad \text{[m+j+j']} \end{array}$$

Let $\vec{x}_{h\sigma}$, etc..., the $m + k$ shifted vectors of variables arising from renaming them apart from variables in Q'_1. The states C_1, C_2 are:

$$\begin{array}{lll} C_1 & = & \langle (\vec{x}_{h\sigma} = \vec{t}[\vec{x}_{y\sigma}] \wedge \vec{x}_{v\sigma} = \vec{v}[\vec{x}_{y\sigma}]), \vec{q}(\vec{x}_{v\sigma}), \vec{p}_{|2}(\vec{x}'_{|2}) \\ & & \,|\, \varphi[\vec{x}] \wedge \vec{x}' = \vec{u}[\vec{x}] \wedge \vec{x}_{h\sigma} = \vec{x}'_1 \rangle \qquad \text{[m+k+m']} \\ C_2 & = & \langle \vec{q}(\vec{x}_{v\sigma}), \vec{p}_{|2}(\vec{x}'_{|2}) \\ & & \,|\, \varphi[\vec{x}] \wedge \vec{x}' = \vec{u}[\vec{x}] \wedge \vec{x}_{h\sigma} = \vec{x}'_1 \wedge \vec{x}_{h\sigma} = \vec{t}[\vec{x}_{y\sigma}] \wedge \vec{x}_{v\sigma} = \vec{v}[\vec{x}_{y\sigma}] \rangle \qquad \text{[m+k+m']} \end{array}$$

we will apply vector splitting and variable renaming to go from the constraint store of C_2 to the one belonging to Q'_2. We omit the number of used variables but the reader can easily check that the elimination equates them.

$$\begin{array}{lr} \varphi[\vec{x}] \wedge \vec{x}' = \vec{u}[\vec{x}] \wedge \vec{x}_{h\sigma} = \vec{x}'_1 \wedge \vec{x}_{h\sigma} = \vec{t}[\vec{x}_{y\sigma}] \wedge \vec{x}_{v\sigma} = \vec{v}[\vec{x}_{y\sigma}] & \Leftrightarrow \\ \{\vec{x}' = \vec{x}'_1\vec{x}'_{|2}, \vec{u}[\vec{x}] = \vec{u}_1[\vec{x}]\vec{u}_{|2}[\vec{x}]\} & \\ \varphi[\vec{x}] \wedge \vec{x}'_1 = \vec{u}_1[\vec{x}] \wedge \vec{x}'_{|2} = \vec{u}_{|2}[\vec{x}] \wedge \vec{x}_{h\sigma} = \vec{x}'_1 \wedge \vec{x}_{h\sigma} = \vec{t}[\vec{x}_{y\sigma}] \wedge \vec{x}_{v\sigma} = \vec{v}[\vec{x}_{y\sigma}] & \Leftrightarrow \\ \{\vec{x}'_1, \vec{x}_{h\sigma} \text{ elimination}\} & \\ \varphi[\vec{x}] \wedge \vec{u}_1[\vec{x}] = \vec{t}[\vec{x}_{y\sigma}] \wedge \vec{x}_{v\sigma} = \vec{v}[\vec{x}_{y\sigma}] \wedge \vec{x}'_{|2} = \vec{u}_{|2}[\vec{x}] & \Leftrightarrow \\ \{\text{renaming}\} & \\ \varphi[\vec{x}] \wedge \vec{u}_1[\vec{x}] = \vec{t}[\vec{x}_\sigma] \wedge \vec{x}'_q = \vec{v}[\vec{x}_\sigma] \wedge \vec{x}'_p = \vec{u}_{|2}[\vec{x}] & \end{array}$$

by soundness, $C_2 \approx_\mathcal{D} Q'_2$.

□

The derivation set of an state in GPF form is in close correspondence with its original one, with the caveat that for every resolution step its GPF form will perform two transitions. But that behavior is innocuous from an operational point of view, indeed, the derivation set will be bigger but given that the new induced derivation is deterministic the final derivations of answers for both Q and Q' will coincide.

Corollary 6.35. *A derivation $Q \rightarrow \cdots \rightarrow \langle \Box \,|\, \varphi \rangle$ exists iff $Q' \rightarrow \cdots \rightarrow \langle \Box \,|\, \varphi \rangle$ exists.*

Remark 6.36. *There is no technical difficulty in defining a GPF form that doesn't use a canonical $1 \ldots n$ scheme for variables. However it complicates the notation and the canonical scheme used here is closer to the one* hidden *behind the relational approach.*

6.5.1.2. *Call-Return Transition System.* The use of a GPF is a necessary step in our road to eliminate the renaming apart operation, which in this section will be fully removed using the following property of the existential quantifier with regards to constraint satisfaction:

$$\varphi[\vec{x}] \wedge \exists \vec{y}.\, \psi[\vec{y}] \iff \varphi[\vec{x}] \wedge \psi[\vec{y}_\sigma] \qquad \vec{y}_\sigma = \sigma_{\vec{x}}(\vec{y})$$

where $\sigma_{\vec{x}}(\vec{y})$ is a renaming apart of $\vec{y}$ for $\vec{x}$. The reader may see that formula on the right hand side corresponds to the scheme generated by a resolution step. We replace it by the left one, obtaining a new resolution scheme free of renaming apart.

However, parameter passing still remains a problem, as, e.g. we must equate a predicate call $p(x_8, x_9)$ with a clause head $p(x_1, x_2)$ whose tail may contain x_8 or x_9. Several possibilities exist to remedy this

problem, but most of them amount to having a notion of *scope* creation when performing a resolution step.

Thanks to Cor. 6.35 we restrict ourselves to states and programs in GPF form. We augment our notion of flat state to an hereditary notion of state closed under "sub-states", with n variables that can be inspected, that is to say, not hidden in the constraint store. We correspondingly define a new system free of renaming apart with "call" and "return" derivations. This system effectively represents in a logical way the notion of call-frame present in most Prolog implementations.

Definition 6.37 (Call-Return State)**.** *The set $\mathcal{CS}$ of call-return states is defined inductively as:*

- $\langle \vec{p} \,|\, \varphi[\vec{x}] \rangle_n$*, where $p_i \equiv P_i(\vec{x}_i)$ is an atom representing a call to a defined predicate P_i or $p_i \equiv \psi$ a constraint, $\vec{x}_i$ is a vector of variables, n a natural number, and $\varphi[\vec{x}]$ is a constraint store.*
- $\left\langle^{\pi} CS, \vec{p} \,|\, \varphi[\vec{x}] \right\rangle_n$*, where CS is a call-return state, π is a permutation, $\vec{p}$ a vector of atoms similar to the previous case, n a natural number and $\varphi[\vec{x}]$ a constraint store.*

The second clause of the definition captures the notion of nested states and includes the permutations used for parameter passing. The natural number n records the number of *out* variables, needed for return when the inner state reaches the empty resolvent. Indeed, we simulate parameter passing in a resolution step by profiting from the fact that the head or every clause is of the form $p(x_1, \ldots, x_n)$. Then, for call we permute variables in the current constraint store so the parameters for p have the name $x_1, \ldots, x_n$. Then, we build the new constraint for the inner predicate by using the existential quantifier to *hide* existing variables bigger than n.

For instance, given a state $\langle p_1(x_2) \,|\, \varphi[x_1, \ldots, x_3] \rangle$ and clause

$$p_1(x_1) \leftarrow \exists_1^3.(\varphi'[x_1, \ldots, x_3], q(x_1, x_2))$$

we build a new inner state $\langle q(x_1, x_2) \,|\, \exists_1^3.\varphi[\pi(\vec{x})] \rangle_1$, where $\pi(x_2) = x_1, \pi(x_1) = x_2$, or in *swap* or transposition form, $\pi = (x_1\ x_2)$, so we have an inner constraint $\exists_1^3.\varphi[x_2, x_1, x_3]$. The return operation just hides all the variables bigger than n and inverts the permutation. For a state $\left\langle^{\pi} \langle \square \,|\, \psi \rangle_n, \vec{p} \,|\, \varphi \right\rangle$, we will return the constraint $\pi^{-1}(\exists^{n\uparrow}.\psi)$, obtaining the state $\langle \vec{p} \,|\, \varphi \wedge \pi^{-1}(\exists^{n\uparrow}.\psi) \rangle$.

Recall how a permutation interacts with an existential formula:

$$w_\pi(\exists x_i.\, \varphi) = \exists x_{\pi(i)}.\, w_\pi(\varphi)$$

We define constraint store operators Δ and ∇ for the call and return manipulations. $\Delta_n^\pi(\varphi)$ may be read as call with φ into a context with n open variables and permutation π. $\nabla_n^\pi(\varphi, \psi)$ may be read as return φ from context (π, n) into ψ:

$$\begin{aligned} \Delta_n^\pi(\varphi) &= \exists^{n\uparrow}.\pi(\varphi) \\ \nabla_n^\pi(\varphi, \psi) &= \psi \wedge \pi^{-1}(\exists^{n\uparrow}.\, \varphi) \end{aligned}$$

Call-return include regular program states. Thus, the standard renaming apart transitions of Def. 3.27 may apply to base call-return states, although we will never generate a *sub-state*. But in order to use their full power, we use the new call-return system defined below:

Definition 6.38 (Call-Return Transition System)**.** *For the sake of space we write $\vec{p}$ for $\vec{p}(\vec{x})$, etc...*

$$\langle \psi, \vec{p} \,|\, \varphi \rangle_n \xrightarrow{constraint}_{cr} \langle \vec{p} \,|\, \varphi \wedge \psi \rangle_n \quad \boxed{\textit{if } p_1 \equiv \psi \textit{ and } \varphi \wedge \psi \textit{ satisfiable}}$$

$$\langle p(\vec{x}_1), \vec{p} \,|\, \varphi \rangle_n \xrightarrow{call/cl_i}_{cr} \left\langle^{\pi} \langle \vec{q} \,|\, \Delta^{\pi}_{h}(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \quad \boxed{\textit{if } cl_i : p(\vec{x}_h) \leftarrow \exists^m_h.\vec{q} \in P' \textit{ and } \pi(\vec{x}_1) = \vec{x}_h}$$

$$\left\langle^{\pi} \langle \Box \,|\, \psi \rangle_m, \vec{p} \,|\, \varphi \right\rangle_n \xrightarrow{return}_{cr} \langle \vec{p} \,|\, \nabla^{\pi}_m(\psi, \varphi) \rangle_n$$

$$\left\langle^{\pi} PS, \vec{p} \,|\, \varphi \right\rangle_n \xrightarrow{sub}_{cr} \left\langle^{\pi} PS', \vec{p} \,|\, \varphi \right\rangle_n \quad \boxed{\textit{if } PS \neq \langle \Box \,|\, \varphi' \rangle_h, \textit{ and } PS \rightarrow_p PS'}$$

We prove the call-return transition system equivalent to the standard one.

Lemma 6.39. *Given a state $Q_1 = \langle p_1(\vec{x}_1), \vec{p}_{|2}(\vec{x}) \,|\, \varphi \rangle_n$ and a program P, both in GPF:*

$$\begin{array}{ll} Q_1 \rightarrow \ldots \rightarrow \langle \vec{p}_{|2}(\vec{x}) \,|\, \varphi' \rangle_n & \\ & \textit{iff} \\ Q_1 \rightarrow_{cr} \ldots \rightarrow_{cr} \langle \vec{p}_{|2}(\vec{x}) \,|\, \varphi'' \rangle_n & \end{array}$$

and it is the case that $\langle \vec{p}_{|2}(\vec{x}) \,|\, \varphi' \rangle_n \approx_{\mathcal{D}} \langle \vec{p}_{|2}(\vec{x}) \,|\, \varphi'' \rangle_n$.

Proof. By induction over the length of the first derivation. They key point of the proof is to research the behavior the new sub-state notion induces.
Base Case: The base case is a derivation of length 1, corresponding either to a constraint step or a *empty* clause $p_1(\vec{x}_h) \leftarrow$.

- If p is a constraint, the proof is immediate as the constraint transition is the same in both systems.
- If p is a defined predicate with empty clause, then the proof is also immediate as the transition for the first system is:

$$\begin{array}{ll} \langle p(\vec{x}_p), \vec{p}(\vec{x}) \,|\, \varphi \rangle_n & \rightarrow_r \\ \langle \vec{p}(\vec{x}) \,|\, \varphi \wedge \vec{x}_{h\sigma} = \vec{x}_p \rangle_n & \end{array}$$

  and for the call-return one is:

$$\begin{array}{ll} \langle p(\vec{x}_p), \vec{p}(\vec{x}) \,|\, \varphi \rangle_n & \xrightarrow{call}_{cr} \\ \left\langle^{\pi} \langle \Box \,|\, \exists^{h\uparrow}.\pi(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n & \xrightarrow{return}_{cr} \\ \langle \vec{p} \,|\, \varphi \wedge \pi^{-1}(\exists^{h\uparrow}.\exists^{h\uparrow}.\pi(\varphi)) \rangle_n & \end{array}$$

  $(\varphi \wedge \pi^{-1}(\exists^{h\uparrow}.\exists^{h\uparrow}.\pi(\varphi))) \Leftrightarrow \varphi$ and $(\varphi \wedge \vec{x}_{h\sigma} = \vec{x}_p) \Leftrightarrow \varphi$, completing the proof.

Inductive Case: The inductive case is when p_1 is a defined predicate with a non-empty clause:

$$p_1(\vec{x}_h) \leftarrow \exists^n_m.\vec{q}(\vec{x}').$$

Note that the $\vec{x}$ occurring in the states and in the clause are different, we will use $\vec{x}'$ for the one coming from the clause, but is equally a sequence tux. $\vec{x}$ and $\vec{x}'$ only differ in length. We have a derivation of length $i+1$. The derivations for both transition systems are:

$$\langle \vec{p}(\vec{x}) \,|\, \varphi[\vec{x}] \rangle \rightarrow_r \langle \vec{q}(\vec{x}_\sigma), \vec{p}_{|2}(\vec{x}) \,|\, \varphi[\vec{x}] \wedge \vec{x}_1 = \vec{x}_{h\sigma} \rangle \overbrace{\rightarrow \cdots \rightarrow}^{i} \langle \vec{p}_{|2}(\vec{x}) \,|\, \varphi[\vec{x}] \wedge \vec{x}_1 = \vec{x}_{h\sigma} \wedge \varphi'[\vec{x}_\sigma] \rangle$$

$$\begin{array}{l} \langle \vec{p}(\vec{x}) \,|\, \varphi[\vec{x}] \rangle_m \xrightarrow{call}_{cr} \left\langle^{\pi} \langle \vec{q}(\vec{x}') \,|\, \exists^{h\uparrow}.\pi(\varphi[\vec{x}]) \rangle_h, \vec{p}_{|2} \,|\, \varphi[\vec{x}] \right\rangle_m \overbrace{\rightarrow \cdots \rightarrow}^{i'} \\ \left\langle^{\pi} \langle \Box \,|\, \exists^{h\uparrow}.\pi(\varphi[\vec{x}]) \wedge \varphi'[\vec{x}'] \rangle_h, \vec{p}_{|2} \,|\, \varphi[\vec{x}] \right\rangle_m \xrightarrow{return}_{cr} \\ \langle \vec{p}_{|2}(\vec{x}) \,|\, \varphi[\vec{x}] \wedge \pi^{-1}(\exists^{h\uparrow}.(\exists^{h\uparrow}.\pi(\varphi[\vec{x}]) \wedge \varphi'(\vec{x}'))) \rangle_m \end{array}$$

with $\pi(\vec{x}_1)$. We must be able to apply the induction hypothesis for the derivations of length i and i', which amounts to checking equivalence of the substate with a restricted notion of the second one. Then, we must check logical equivalence of the resulting constraint store after return.

We use the fact that derivations for the first atom or constraint of a resolvent doesn't depend on the rest of it:

$$\begin{array}{ll} \langle \vec{p}(\vec{x}) \,|\, \varphi[\vec{x}] \rangle & \rightarrow \cdots \rightarrow \langle \square, \vec{p}_{|2}(\vec{x}) \,|\, \varphi[\vec{x}] \wedge \varphi'[\vec{x}_1] \rangle \quad \text{iff} \\ \langle p_1(\vec{x}_1) \,|\, \varphi[\vec{x}] \rangle & \rightarrow \cdots \rightarrow \langle \square \,|\, \varphi[\vec{x}] \wedge \varphi'[\vec{x}_1] \rangle \end{array}$$

Then, we check the equivalence of the two states:

$$\langle \vec{q}(\vec{x}') \,|\, \exists^{h\uparrow}.\pi(\varphi[\vec{x}]) \rangle \approx_{\mathcal{D}} \langle \vec{q}(\vec{x}_\sigma) \,|\, \varphi[\vec{x}] \wedge \vec{x}_1 = \vec{x}_{h\sigma} \rangle$$

Thus, the precise statement needed to prove state equivalence is:

$$\exists \vec{x}'.\exists^{h\uparrow}.\pi(\varphi[\vec{x}]) \iff \exists \vec{x}\vec{x}_\sigma.(\varphi[\vec{x}] \wedge \vec{x}_1 = \vec{x}_{h\sigma})$$

Let $m = |\vec{x}|$ and $k = |\vec{x}'|$. Thus $\vec{x} = x_1, \ldots, x_m$, $\vec{x}' = x_1, \ldots, x_k$ and $\vec{x}_\sigma = x_{m+1}, \ldots, x_{m+k}$. The captured variables inside the $\exists^{h\uparrow}$ quantifier are $x_{h+1}, \ldots, x_m$. Let $\vec{x}_r = \vec{x}/\vec{x}_1$. Then, $\pi(\vec{x}) = x_1, \ldots, x_h, \pi(\vec{x}_r)$. Renaming apart $\pi(\vec{x}_r)$ to $\vec{x}_{r'} = x_{k+1}, \ldots, x_{k+m}$ we can eliminate the inner quantifier:

$$\exists \vec{x}'\vec{x}_{r'}.\varphi[\vec{x}_h\vec{x}_{r'}] \iff \exists \vec{x}\vec{x}_\sigma.(\varphi[\vec{x}] \wedge \vec{x}_1 = \vec{x}_{h\sigma})$$

The may match $\vec{x}'$ to $\vec{x}_\sigma$, but $\vec{x}_{r'}$ is missing h variables. If we add new h variables $\vec{x}_{h'}$ and add the equation $\vec{x}_{h'} = \vec{x}_h$ we get the desired equivalence:

$$\exists \vec{x}'\vec{x}_{r'}\vec{x}_{h'}.(\varphi[\vec{x}_h\vec{x}_{r'}] \wedge \vec{x}_{h'} = \vec{x}_h) \iff \exists \vec{x}\vec{x}_\sigma.(\varphi[\vec{x}] \wedge \vec{x}_1 = \vec{x}_{h\sigma})$$

We apply the induction hypothesis. Actually, we are applying induction as many times as elements or constraints $\vec{q}$ has. We could recast this lemma to make this fact more explicit:

$$\langle \vec{p}(\vec{x}) \,|\, \varphi \rangle \overbrace{\rightarrow \cdots \rightarrow}^{i} \langle \square, \vec{p}_{|2}(\vec{x}) \,|\, \varphi' \rangle \overbrace{\rightarrow \cdots \rightarrow}^{j} \langle \square \,|\, \varphi'' \rangle$$

but we think the current presentation is clearer.

After applying the induction hypothesis, the following equivalence remains to be proven:

$$\exists \vec{x}\vec{x}_\sigma.(\varphi[\vec{x}] \wedge \vec{x}_1 = \vec{x}_{h\sigma} \wedge \varphi'[\vec{x}_\sigma]) \iff \exists \vec{x}.(\varphi[\vec{x}] \wedge \pi^{-1}(\exists^{h\uparrow}.(\exists^{h\uparrow}.\pi(\varphi[\vec{x}]) \wedge \varphi'(\vec{x}'))))$$

We focus on the formula on the right. Similarly to the previous case, we apply the permutation using the knowledge of the variables involved:

$$\exists \vec{x}.(\varphi[\vec{x}] \wedge \exists \vec{x}'/\vec{x}_1.(\varphi'(\pi^{-1}(\vec{x}')) \wedge \exists \vec{x}'/\vec{x}_1(\varphi[\vec{x}]))))$$

Renaming apart $\vec{x}'$ and adding the new variables needed with their corresponding equations, we get:

$$\exists \vec{x}\vec{x}_\sigma.(\varphi[\vec{x}] \wedge \vec{x}_1 = \vec{x}_{h\sigma} \wedge \varphi'[\vec{x}_\sigma] \wedge \exists \vec{x}'/\vec{x}_1(\varphi[\vec{x}]))$$

which is clearly equivalent to:

$$\exists \vec{x}\vec{x}_\sigma.(\varphi[\vec{x}] \wedge \vec{x}_1 = \vec{x}_{h\sigma} \wedge \varphi'[\vec{x}_\sigma])$$

Concluding the proof. ☐

Corollary 6.40. *Given a state $Q_1 = \langle \vec{p} \,|\, \varphi \rangle_n$ and a program P, both in GPF:*

$$\begin{array}{lllll} Q_1 & \rightarrow & \ldots & \rightarrow & \langle \square \,|\, \varphi' \rangle_n \\ & & & & \qquad \textit{iff} \\ Q_1 & \rightarrow_{cr} & \ldots & \rightarrow_{cr} & \langle \square \,|\, \varphi'' \rangle_n \end{array}$$

and $\langle \square \,|\, \varphi' \rangle_n \approx_{\mathcal{D}} \langle \square \,|\, \varphi'' \rangle_n$.

Proof. By induction over the length of the resolvent and Lem. 6.39. ☐

6.5.1.3. *Folding of SLD derivations.* We incorporate proof search information into our call-return states. The set of derivations arising from all the possible transitions from a call-return state is folded into a single derivation of *resolution states*. Algebraically, proof search is a tree thus we will extend our states with a parallel constructor $((PS_1 \,\|\, PS_2))$. The leaf of the proof tree may be a successful state or a failure. We need to make failure explicit so we introduce a new $\langle fail \rangle$ state constructor. Thus, a *resolution state* captures all the meta theory of constraint logic programming but recursion. Recursion is captured by instantiation of a predicate symbol, emulating a general fixpoint operator. The *call* transition is a meta transition equivalent to the meta-rewriting step we perform in Sec. 6.4.

Definition 6.41 (Resolution States)**.** *The set $\mathcal{PS}$ of resolution states is inductively defined as:*

- $\langle fail \rangle$.
- $\langle \vec{p} \,|\, \varphi \rangle_n$*, where* $p_i \equiv P_i(\vec{x}_i)$ *is an atom representing a call to a defined predicate* P_i*, or a constraint* $p_i \equiv \psi$*,* $\vec{x}_i$ *is a vector of variables,* φ *is a constraint store and* n *a natural number.*
- $\left\langle^{\pi} PS, \vec{p} \,|\, \varphi \right\rangle_n$*, where* PS *is a resolution state, and* π *is a permutation.*
- $\left\langle^{\pi}\blacktriangleright PS, \vec{p} \,|\, \varphi \right\rangle_n$*, the "select state". It represents the state just before selecting a clause to proceed with proof search.*
- $(PS_1 \,\|\, PS_2)$*, representing a choice in the proof search.*

The need for the new parallel state is obvious. The need for the select state comes from the fact that a resolution step algebraically is split into two task: clause selection and parameter passing. While a state like $\langle (A \,\|\, B), \vec{p} \,|\, \varphi \rangle$ looks right, we cannot define a correct algebraic interpretation for it in Sec. 6.5.3, as $(A \,\|\, B)$ is living in a different "variable naming space" than $\vec{p}$.

Thus, the new "select" state represents the moment where we have shifted the name space in order to call a defined predicate but we have not yet done the actual parameter passing by creating the new constraint store. Combining the two steps in one transition would mean that we are performing more work than what SLD resolution does, thus losing the operational equivalence. If we were to prove equivalence with regards to a non-selecting strategy like breadth first the select state would not be needed.

Definition 6.42 (Resolution Transition System)**.** *The new transition system* $\rightarrow_P \subseteq (\mathcal{PS} \times \mathcal{PS})$ *for the operational semantics of SLD search for constraint logic programming is shown in Fig. 6.4*

The equivalence of this semantics with respect to sets of call-return derivations is almost immediate. We establish that backtracking and selection works as expected.

Lemma 6.43 (Clause Selection)**.** *Given clauses:*

$$\begin{array}{c} cl_1 : p(\vec{x}_h) \leftarrow \exists^{h\uparrow}.\vec{q} \\ cl_2 : p(\vec{x}_h) \leftarrow \exists^{h\uparrow}.\vec{r} \end{array}$$

and a state $\langle p(\vec{x}), \vec{p} \,|\, \varphi \rangle$*. It will have the following derivation set using the call-return system:*

$$\left\{ \begin{array}{l} \langle p(\vec{x}), \vec{p} \,|\, \varphi \rangle_n \xrightarrow{call/cl_1}_{cr} \left\langle^{\pi} \langle \vec{q} \,|\, \Delta_h^{\pi}(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \not\rightarrow_{cr} \\ \langle p(\vec{x}), \vec{p} \,|\, \varphi \rangle_n \xrightarrow{call/cl_2}_{cr} \left\langle^{\pi} \langle \vec{r} \,|\, \Delta_h^{\pi}(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \xrightarrow{constraint}_{cr} \left\langle^{\pi} \langle \vec{r}_{|2} \,|\, r_1 \wedge \Delta_h^{\pi}(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \end{array} \right\}$$

iff it has this derivation in the resolution system:

$$\langle p(\vec{x}), \vec{p} \,|\, \varphi \rangle_n \rightarrow_p \ldots \rightarrow_p \left\langle^{\pi} \langle \vec{r}_{|2} \,|\, r_1 \wedge \Delta_h^{\pi}(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n$$

$$
\begin{array}{rcl}
\langle \psi, \vec{p} \,|\, \varphi \rangle_n & \xrightarrow{constraint}_p & \langle \vec{p} \,|\, \varphi \wedge \psi \rangle_n \\
\langle \psi, \vec{p} \,|\, \varphi \rangle_n & \xrightarrow{fail}_p & \langle fail \rangle \\
 & & \boxed{\text{if } \varphi \wedge \psi \text{ is not satisfiable}} \\
\langle p(\vec{x}), \vec{p} \,|\, \varphi \rangle_n & \xrightarrow{call}_p & \left\langle^{\pi} \blacktriangleright (\langle \vec{q}_1 \,|\, \top \rangle_h \,\|\, \ldots \,\|\, \langle \vec{q}_k \,|\, \top \rangle_h), \vec{p} \,|\, \varphi \right\rangle_n \\
 & & \boxed{\text{if } p(\vec{x}_h) \leftarrow \exists^{h\uparrow}.(\vec{q}_1 \vee \ldots \vee \vec{q}_k) \in P',\ \pi(\vec{x}) = \vec{x}_h} \\
\left\langle^{\pi} \blacktriangleright (\langle \vec{q} \,|\, \psi \rangle_h \,\|\, PS), \vec{p} \,|\, \varphi \right\rangle_n & \xrightarrow{select}_p & \left(\left\langle^{\pi} \langle \vec{q} \,|\, \psi \wedge \Delta^{\pi}_h(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \,\|\, \left\langle^{\pi} \blacktriangleright PS, \vec{p} \,|\, \varphi \right\rangle_n \right) \\
\left\langle^{\pi} \langle \Box \,|\, \psi \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n & \xrightarrow{return}_p & \langle \vec{p} \,|\, \nabla^{\pi}_h(\psi, \varphi) \rangle_n \\
\left\langle^{\pi} \langle fail \rangle, \vec{p} \,|\, \varphi \right\rangle_n & \xrightarrow{return}_p & \langle fail \rangle \\
\left\langle^{\pi} PS, \vec{p} \,|\, \varphi \right\rangle_n & \xrightarrow{sub}_p & \left\langle^{\pi} PS', \vec{p} \,|\, \varphi \right\rangle_n \\
 & & \boxed{\text{if } PS \neq \langle \Box \,|\, \psi \rangle,\ PS \neq \langle fail \rangle, \text{ and } PS \rightarrow_p PS'} \\
(\langle fail \rangle \,\|\, PS) & \xrightarrow{backtrack}_p & PS \\
(PS_1 \,\|\, PS_2) & \xrightarrow{seq}_p & (PS'_1 \,\|\, PS_2) \\
 & & \boxed{\text{if } PS \neq \langle fail \rangle, \text{ and } PS_1 \rightarrow_p PS'_1}
\end{array}
$$

(We omit the case in *select* where the left side has no *PS* component which happens when the number of clauses for a given predicate is one ($k = 1$))

FIGURE 6.4. Resolution Transition System

PROOF. The derivation set is only possible if $q_1 \wedge \Delta^{\pi}_h(\varphi)$ is not satisfiable. We check the transitions using $\rightarrow_p$ (we label *sub* and *seq* transitions with the actual atomic ones):

$$
\begin{array}{l}
\langle p(\vec{x}) \,|\, \varphi \rangle_n \xrightarrow{call}_p \left\langle^{\pi} \blacktriangleright (\langle \vec{q} \,|\, \top \rangle_h \,\|\, \langle \vec{r} \,|\, \top \rangle_h), \vec{p} \,|\, \varphi \right\rangle_n \xrightarrow{select}_p \\
\left(\left\langle^{\pi} \langle \vec{q} \,|\, \Delta^{\pi}_h(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \,\|\, \left\langle^{\pi} \blacktriangleright \langle \vec{r} \,|\, \top \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \right) \xrightarrow{fail}_p \\
\left(\left\langle^{\pi} \langle fail \rangle, \vec{p} \,|\, \varphi \right\rangle \,\|\, \left\langle^{\pi} \blacktriangleright \langle \vec{r} \,|\, \top \rangle, \vec{p} \,|\, \varphi \right\rangle \right) \xrightarrow{return}_p \left(\langle fail \rangle \,\|\, \left\langle^{\pi} \blacktriangleright \langle \vec{r} \,|\, \top \rangle, \vec{p} \,|\, \varphi \right\rangle \right) \xrightarrow{backtrack}_p \\
\left\langle^{\pi} \blacktriangleright \langle \vec{r} \,|\, \top \rangle, \vec{p} \,|\, \varphi \right\rangle \xrightarrow{select}_p \left\langle^{\pi} \langle \vec{r} \,|\, \Delta^{\pi}_h(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \xrightarrow{constraint}_{cr} \left\langle^{\pi} \langle \vec{r}_{|2} \,|\, r_1 \wedge \Delta^{\pi}_h(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n
\end{array}
$$

Clearly, if $\rightarrow_p$ could carry out any other transition the derivation set would be different. $\Box$

Lemma 6.44 (Backtracking)**.** *Given clauses:*

$$
\begin{array}{l}
cl_1 : p(\vec{x}_h) \leftarrow \exists^{h\uparrow}.q(\vec{x}_1), \vec{q} \\
cl_2 : p(\vec{x}_h) \leftarrow \exists^{h\uparrow}.r(\vec{x}_2), \vec{r} \\
cl_3 : q(\vec{x}_i) \leftarrow \exists^{i\uparrow}.\vec{s} \\
cl_4 : r(\vec{x}_j) \leftarrow \exists^{j\uparrow}.\vec{t}
\end{array}
$$

and a state $\langle p(\vec{x}), \vec{p} \,|\, \varphi \rangle$. It will have the following derivation set using the call-return system:

$$
\left\{
\begin{array}{ll}
\langle p(\vec{x}), \vec{p} \,|\, \varphi \rangle_n \xrightarrow{call/cl_1}_{cr} & \left\langle^{\pi} \langle q(\vec{x}_1), \vec{q} \,|\, \Delta^{\pi}_h(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \xrightarrow{call/cl_3}_{cr} \\
 & \left\langle^{\pi} \left\langle^{\pi_1} \langle \vec{s} \,|\, \Delta^{\pi_1}_i(\Delta^{\pi}_h(\varphi)) \rangle_h, \vec{q} \,|\, \Delta^{\pi}_h(\varphi) \right\rangle, \vec{p} \,|\, \varphi \right\rangle_n \not\rightarrow_{cr} \\
\langle p(\vec{x}), \vec{p} \,|\, \varphi \rangle_n \xrightarrow{call/cl_2}_{cr} & \left\langle^{\pi} \langle r(\vec{x}_2), \vec{r} \,|\, \Delta^{\pi}_h(\varphi) \rangle_h, \vec{p} \,|\, \varphi \right\rangle_n \xrightarrow{call/cl_4}_{cr} \\
 & \left\langle^{\pi} \left\langle^{\pi_2} \langle \vec{t} \,|\, \Delta^{\pi_2}_j(\Delta^{\pi}_h(\varphi)) \rangle_h, \vec{r} \,|\, \Delta^{\pi}_h(\varphi) \right\rangle, \vec{p} \,|\, \varphi \right\rangle_n \xrightarrow{constraint}_{cr} \\
 & \left\langle^{\pi} \left\langle^{\pi_2} \langle \vec{t}_{|2} \,|\, t_1 \wedge \Delta^{\pi_2}_j(\Delta^{\pi}_h(\varphi)) \rangle_h, \vec{r} \,|\, \Delta^{\pi}_h(\varphi) \right\rangle, \vec{p} \,|\, \varphi \right\rangle_n
\end{array}
\right\}
$$

iff it has this derivation using the resolution system:

$$
\langle p(\vec{x}), \vec{p} \,|\, \varphi \rangle \rightarrow_p \ldots \rightarrow_p \left\langle^{\pi} \left\langle^{\pi_2} \langle \vec{t}_{|2} \,|\, t_1 \wedge \Delta^{\pi_2}_j(\Delta^{\pi}_h(\varphi)) \rangle_h, \vec{r} \,|\, \Delta^{\pi}_h(\varphi) \right\rangle, \vec{p} \,|\, \varphi \right\rangle_n
$$

Proof. We check the transitions as in the previous lemma.

$$\langle p(\vec{x}), \vec{p} \,|\, \varphi\rangle_n \xrightarrow{call}_p \big\langle^{\pi} \blacktriangleright (\langle q(\vec{x}_1), \vec{q} \,|\, \top\rangle_h \,\rule{2pt}{8pt}\, \langle r(\vec{x}_2), \vec{r} \,|\, \top\rangle_h), \vec{p} \,|\, \varphi\big\rangle_n \xrightarrow{select}_p$$
$$\big(\big\langle^{\pi} \langle q(\vec{x}_1), \vec{q} \,|\, \Delta^{\pi}_h(\varphi)\rangle_h, \vec{p} \,|\, \varphi\big\rangle_n \,\rule{2pt}{8pt}\, \big\langle^{\pi} \blacktriangleright \langle r(\vec{x}_2), \vec{r} \,|\, \top\rangle_h, \vec{p} \,|\, \varphi\big\rangle_n\big) \xrightarrow{call}_p$$
$$\big(\big\langle^{\pi} \big\langle^{\pi_1} \langle \vec{s} \,|\, \Delta^{\pi_1}_i(\Delta^{\pi}_h(\varphi))\rangle_h, \vec{q} \,|\, \Delta^{\pi}_h(\varphi)\big\rangle, \vec{p} \,|\, \varphi\big\rangle_n \,\rule{2pt}{8pt}\, \big\langle^{\pi} \blacktriangleright \langle r(\vec{x}_2), \vec{r} \,|\, \top\rangle_h, \vec{p} \,|\, \varphi\big\rangle_n\big) \xrightarrow{fail}_p$$
$$\big(\big\langle^{\pi} \big\langle^{\pi_1} \langle \mathit{fail}\rangle, \vec{q} \,|\, \Delta^{\pi}_h(\varphi)\big\rangle, \vec{p} \,|\, \varphi\big\rangle_n \,\rule{2pt}{8pt}\, \big\langle^{\pi} \blacktriangleright \langle r(\vec{x}_2), \vec{r} \,|\, \top\rangle_h, \vec{p} \,|\, \varphi\big\rangle_n\big) \xrightarrow{return}_p$$
$$\big(\big\langle^{\pi} \langle \mathit{fail}\rangle, \vec{p} \,|\, \varphi\big\rangle_n \,\rule{2pt}{8pt}\, \big\langle^{\pi} \blacktriangleright \langle r(\vec{x}_2), \vec{r} \,|\, \top\rangle_h, \vec{p} \,|\, \varphi\big\rangle_n\big) \xrightarrow{return}_p \big(\langle \mathit{fail}\rangle \,\rule{2pt}{8pt}\, \big\langle^{\pi} \blacktriangleright \langle r(\vec{x}_2), \vec{r} \,|\, \top\rangle_h, \vec{p} \,|\, \varphi\big\rangle_n\big)$$
$$\xrightarrow{backtrack}_p \big\langle^{\pi} \blacktriangleright \langle r(\vec{x}_2), \vec{r} \,|\, \top\rangle_h, \vec{p} \,|\, \varphi\big\rangle_n \xrightarrow{select}_p \big\langle^{\pi} \langle r(\vec{x}_2), \vec{r} \,|\, \Delta^{\pi}_h(\varphi)\rangle_h, \vec{p} \,|\, \varphi\big\rangle_n \xrightarrow{call}_p$$
$$\big\langle^{\pi} \big\langle^{\pi_2} \langle \vec{t} \,|\, \Delta^{\pi_2}_j(\Delta^{\pi}_h(\varphi))\rangle_h, \vec{r} \,|\, \Delta^{\pi}_h(\varphi)\big\rangle, \vec{p} \,|\, \varphi\big\rangle_n \xrightarrow{constraint}_p$$
$$\big\langle^{\pi} \big\langle^{\pi_2} \langle \vec{t}_{|2} \,|\, t_1 \wedge \Delta^{\pi_2}_j(\Delta^{\pi}_h(\varphi))\rangle_h, \vec{r} \,|\, \Delta^{\pi}_h(\varphi)\big\rangle, \vec{p} \,|\, \varphi\big\rangle_n$$

□

Corollary 6.45. *For a query $\langle \vec{p} \,|\, \varphi\rangle$, the first successful SLD derivation in its set of derivations using the call-return transition system is*

$$\langle \vec{p} \,|\, \varphi\rangle \rightarrow_{cr} \ldots \rightarrow_{cr} \langle \Box \,|\, \varphi'\rangle$$

iff the derivation

$$\langle \vec{p} \,|\, \varphi\rangle \rightarrow_{p} \ldots \rightarrow_{p} (\langle \Box \,|\, \varphi'\rangle \,\rule{2pt}{8pt}\, PS)$$

exists.

Proof. By induction over the length of the successful derivation, repeatedly applying Lem. 6.43 and Lem. 6.44. □

Theorem 6.46. *The transition systems of Def. 3.27 and Fig. 6.4 are computationally equivalent, that is to say, for a given query they will return the same answer constraint.*

Proof. The standard transition system is equivalent to the call-return system by Lem. 6.34 and Lem. 6.39. The call-return system is equivalent to the resolution transition system by Cor. 6.45. □

6.5.2. Operational Semantics in Relational Style for SLD-resolution. The logical based operational semantics presented above have been carefully designed in order to have a close correspondence with the relational terms that occur in the relational execution of a constraint logic program. Indeed, each state will be interpreted algebraically in the next section.

The rewriting system of Sec. 6.4 is too fine-grained or small-step to relate it directly to the resolution operational semantics. Thus, we define a set $\mathcal{RS}$ of relational states and a transition system over them such that every state and transition possess a correspondence with a resolution state.

We use the helper notation $\overrightarrow{W(\overline{p})_{\cap}} \equiv R_1 \cap \ldots \cap R_n$, where $R_i \equiv \dot{K}(\varphi_i)$ or $R_i \equiv W_{\pi_i}(\overline{p}_i)$.

Definition 6.47 (Relational States)**.** *The set $\mathcal{RS}$ of relational states is inductively defined as:*

- $\mathbf{0}$, *failure.*
- $I_n(\dot{K}(\varphi) \cap \overrightarrow{W(\overline{p})_{\cap}})$, *base query.*
- $I_n(W^{\circ}_{\pi}(\dot{K}(\varphi) \cap RS) \cap \overrightarrow{W(\overline{p})_{\cap}})$, *selection.*
- $I_n(W^{\circ}_{\pi}(RS \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\overline{p})_{\cap}})$, *subquery.*
- $(RS_1 \cup RS_2)$, *parallel.*

Recall that a predicate p is translated to a relational term such that we have the equation $\overline{p} = \Theta_1 \cup \cdots \cup \Theta_k$, and $\Theta_i = I_{\alpha(p)}(\dot{K}(\varphi_i) \cap \overrightarrow{W(\overline{q})_{\cap}})$.

Definition 6.48 (Relational Transition System)**.** *The transition system for relational states is defined by the rules of Fig. 6.5.*

$$
\begin{array}{lcl}
I_n((\dot{K}(\varphi) \cap \dot{K}(\psi)) \cap \overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{constraint}_r & I_n(\dot{K}(\varphi \wedge \psi) \cap \overrightarrow{W(\overline{p})_\cap}) \\
 & & \boxed{\text{if } \varphi \wedge \psi \text{ satisfiable}} \\
I_n((\dot{K}(\varphi) \cap \dot{K}(\psi)) \cap \overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{fail}_r & \mathbf{0} \\
 & & \boxed{\text{if } \varphi \wedge \psi \text{ not satisfiable}} \\
I_n((\dot{K}(\varphi) \cap W_\pi(\overline{p})) \cap \overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{call}_r & I_n(W^\circ_\pi(\dot{K}(\pi(\varphi)) \cap \Theta) \cap \overrightarrow{W(\overline{p})_\cap}) \\
 & & \boxed{\text{with } \overline{p} = \Theta} \\
I_n(W^\circ_\pi(\dot{K}(\varphi) \cap (\Theta_1 \cup \Theta)) \cap \overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{select}_r & I_n(W^\circ_\pi(\Theta'_1 \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\overline{p})_\cap}) \cup \\
 & & I_n(W^\circ_\pi(\dot{K}(\varphi) \cap \Theta) \cap \overrightarrow{W(\overline{p})_\cap}) \\
 & & \boxed{\Theta_1 \equiv I_m(\overrightarrow{W(\overline{q})_\cap})} \\
 & & \boxed{\Theta'_1 \equiv I_m(\dot{K}(\exists^{m\uparrow}.\varphi) \cap \overrightarrow{W(\overline{q})_\cap})} \\
I_n(W^\circ_\pi(I_m(\dot{K}(\varphi_1)) \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{return}_r & I_n(\dot{K}(\pi^{-1}(\varphi \wedge (\exists^{m\uparrow}.\ \varphi_1))) \cap \overrightarrow{W(\overline{p})_\cap}) \\
I_n(W^\circ_\pi(\mathbf{0} \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{return}_r & \mathbf{0} \\
I_n(W^\circ_\pi(RS \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{sub}_r & I_n(W^\circ_\pi(RS' \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\overline{p})_\cap}) \\
 & & \boxed{\text{if } RS \rightarrow_r RS'} \\
(\mathbf{0} \cup RS) & \xrightarrow{backtrack}_r & RS \\
(RS_1 \cup RS_2) & \xrightarrow{seq}_r & (RS'_1 \cup RS_2) \\
 & & \boxed{\text{if } RS_1 \rightarrow_r RS'_1}
\end{array}
$$

FIGURE 6.5. Relational Transition Rules

6.5.3. The Equivalence. We define an isomorphism between the states in logical and relational style. We check that both transition systems are equivalent, that is to say, the isomorphism induces a simulation between them. After that we check that the rewriting system of Sec. 6.4 implements the relational transition system.

Recall that a query is just a state with $\top$ as constraint store. The transition procedure coincides with the base case of the iso.

Definition 6.49. *We define functions* $\mathsf{R} : \mathcal{PS} \to \mathcal{RS}$ *and* $\mathsf{R}^{-1} : \mathcal{RS} \to \mathcal{PS}$ *by induction over the structure of the states:*

$$
\begin{array}{ll}
\mathsf{R}(\langle fail \rangle) & = \mathbf{0} \\
\mathsf{R}(\langle \vec{p} \,|\, \varphi \rangle_n) & = I_n(\dot{K}(\varphi) \cap \overrightarrow{W(\overline{p})_\cap}) \\
\mathsf{R}(\langle^\pi \blacktriangleright PS, \vec{p} \,|\, \varphi \rangle_n) & = I_n(W^\circ_\pi(\dot{K}(\pi(\varphi)) \cap \mathsf{R}(PS)) \cap \overrightarrow{W(\overline{p})_\cap}) \\
\mathsf{R}(\langle^\pi PS, \vec{p} \,|\, \varphi \rangle_n) & = I_n(W^\circ_\pi(\mathsf{R}(PS) \cap \dot{K}(\pi(\varphi))) \cap \overrightarrow{W(\overline{p})_\cap}) \\
\mathsf{R}((PS_1 \,\rule{0.2em}{0.8em}\, PS_2)) & = (\mathsf{R}(PS_1) \cup \mathsf{R}(PS_2))
\end{array}
$$

In the second case, note that $\vec{p}$ is assumed to be in purified form for defined predicate calls. Thus, for each element p_i of $\vec{p}$ we get a relational term $\dot{K}(\varphi_i)$ if it is a constraint or $W_{\pi_i}(\overline{P_i})$ if it is a call to a defined predicate, where the permutation π is the one which selects the correct variables for its arguments. A

similar remark is valid for the definition of R^{-1}*:*

$$\begin{array}{ll}
\mathsf{R}^{-1}(\mathbf{0}) & = \langle fail \rangle \\
\mathsf{R}^{-1}(I_n(\dot{K}(\varphi) \cap \overrightarrow{W(\bar{p})_\cap})) & = \langle \vec{p} \,|\, \varphi \rangle_n \\
\mathsf{R}^{-1}(I_n(W^\circ_\pi(\dot{K}(\varphi) \cap RS) \cap \overrightarrow{W(\bar{p})_\cap})) & = \left\langle^\pi \blacktriangleright \mathsf{R}^{-1}(RS), \vec{p} \,|\, \pi^{-1}(\varphi) \right\rangle_n \\
\mathsf{R}^{-1}(I_n(W^\circ_\pi(RS \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\bar{p})_\cap})) & = \left\langle^\pi \mathsf{R}^{-1}(RS), \vec{p} \,|\, \pi^{-1}(\varphi) \right\rangle_n \\
\mathsf{R}^{-1}((RS_1 \cup RS_2)) & = \left(\mathsf{R}^{-1}(RS_1) \,\|\, \mathsf{R}^{-1}(RS_2)\right)
\end{array}$$

Note that R is an extension of the translation function of Sec. 6.3.4.

Lemma 6.50. R *is an isomorphism.*

Proof. By induction over the structure of the states. For the failure, base and parallel states the proof is immediate. We check the subquery case:

$$\mathsf{R}^{-1}(\mathsf{R}(\left\langle^\pi PS, \vec{p} \,|\, \varphi \right\rangle_n)) = \mathsf{R}^{-1}(I_n(W^\circ_\pi(\mathsf{R}(PS) \cap \dot{K}(\pi(\varphi))) \cap \overrightarrow{W(\bar{p})_\cap})) =$$
$$\left\langle^\pi \mathsf{R}^{-1}(\mathsf{R}(PS)), \vec{p} \,|\, \pi^{-1}(\pi(\varphi)) \right\rangle_n =_{\{\mathrm{IH}\}} \left\langle^\pi PS, \vec{p} \,|\, \varphi \right\rangle_n$$

and the select case:

$$\mathsf{R}^{-1}(\mathsf{R}(\left\langle^\pi \blacktriangleright PS, \vec{p} \,|\, \varphi \right\rangle_n)) = \mathsf{R}^{-1}(I_n(W^\circ_\pi(\dot{K}(\pi(\varphi)) \cap \mathsf{R}(PS)) \cap \overrightarrow{W(\bar{p})_\cap})) =$$
$$\left\langle^\pi \blacktriangleright \mathsf{R}^{-1}(\mathsf{R}(PS)), \vec{p} \,|\, \pi^{-1}(\pi(\varphi)) \right\rangle_n =_{\{\mathrm{IH}\}} \left\langle^\pi \blacktriangleright PS, \vec{p} \,|\, \varphi \right\rangle_n$$

□

Lemma 6.51. *The resolution transition system (Fig. 6.4) and the relational transition system (Def. 6.48) are equivalent.*

Proof. We check that the relation $R \subseteq (\mathcal{PS} \times \mathcal{RS})$ induced by the isomorphism R is a simulation relation, that is to say:

$$\forall RS, PS.\ (PS, RS) \in R \Rightarrow ((PS \rightarrow_p PS' \iff RS \rightarrow_r RS') \wedge (PS', RS') \in R)$$

Given the one to one nature of our relation and the fact that the transition systems are deterministic, the truth of the above statement can be reduced to checking that any of the following properties:

$$\begin{array}{lll}
PS \rightarrow_p PS' & \Rightarrow & \mathsf{R}(PS) \rightarrow_r \mathsf{R}(PS') \\
RS \rightarrow_r RS' & \Rightarrow & \mathsf{R}^{-1}(PS) \rightarrow_p \mathsf{R}^{-1}(PS')
\end{array}$$

holds for every transition of the system. Note that one case implies the other. We check the non-obvious transitions constraint, fail, call, select, and return. In order to help the reader, we show both transitions, then perform the check outlined above.

• $constraint$:

$$\begin{array}{lll}
\langle \psi, \vec{p} \,|\, \varphi \rangle_n & \xrightarrow{constraint}_p & \langle \vec{p} \,|\, \varphi \wedge \psi \rangle_n \\
I_n((\dot{K}(\varphi) \cap \dot{K}(\psi)) \cap \overrightarrow{W(\bar{p})_\cap}) & \xrightarrow{constraint}_r & I_n(\dot{K}(\varphi \wedge \psi) \cap \overrightarrow{W(\bar{p})_\cap})
\end{array}$$

and the corresponding check:

$$\mathsf{R}(\langle \psi, \vec{p} \,|\, \varphi \rangle_n) = I_n((\dot{K}(\varphi) \cap \dot{K}(\psi)) \cap \overrightarrow{W(\bar{p})_\cap}) \xrightarrow{constraint}_r$$
$$I_n(\dot{K}(\varphi \wedge \psi) \cap \overrightarrow{W(\bar{p})_\cap}) = \mathsf{R}(\langle \vec{p} \,|\, \varphi \wedge \psi \rangle_n)$$

• $fail$:

$$\begin{array}{lll}
\langle \psi, \vec{p} \,|\, \varphi \rangle_n & \xrightarrow{fail}_p & \langle fail \rangle \\
 & & \boxed{\text{if } \varphi \wedge \psi \text{ is not satisfiable}} \\
I_n((\dot{K}(\varphi) \cap \dot{K}(\psi)) \cap \overrightarrow{W(\bar{p})_\cap}) & \xrightarrow{fail}_r & \mathbf{0}
\end{array}$$

the simulation check is:

$$\begin{array}{ll} \mathsf{R}(\langle\psi,\vec{p}\,|\,\varphi\rangle_n) = I_n((\dot{K}(\varphi)\cap\dot{K}(\psi))\cap\overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{fail}_r \\ \mathbf{0} = \mathsf{R}(\langle \mathit{fail}\rangle) & \end{array}$$

• $call$:

$$\begin{array}{lll} \langle p(\vec{x}),\vec{p}\,|\,\varphi\rangle_n & \xrightarrow{call}_p & \left\langle^{\pi}\blacktriangleright(\langle\vec{q}_1\,|\,\top\rangle_h \,\Vert\, \ldots \,\Vert\, \langle\vec{q}_k\,|\,\top\rangle_h),\vec{p}\,|\,\varphi\right\rangle_n \\ & & \boxed{\text{if } p(\vec{x}_h)\leftarrow\exists^{h\uparrow}.(\vec{q}_1\vee\ldots\vee\vec{q}_k)\in P',\ \pi(\vec{x})=\vec{x}_h} \\ I_n((\dot{K}(\varphi)\cap W_\pi(\overline{p}))\cap\overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{call}_r & I_n(W_\pi^\circ(\dot{K}(\pi(\varphi))\cap\Theta)\cap\overrightarrow{W(\overline{p})_\cap}) \\ & & \boxed{\text{with } \overline{p}=\Theta} \end{array}$$

$\mathsf{R}(\langle\vec{q_i}\,|\,\top\rangle_h) = I_h(\overrightarrow{W(\overline{q_i})_\cap}) \equiv \Theta_i$, thus $\mathsf{R}((\langle\vec{q}_1\,|\,\top\rangle_h \,\Vert\, \ldots \,\Vert\, \langle\vec{q}_k\,|\,\top\rangle_h)) = \Theta_1\cup\cdots\cup\Theta_k \equiv \Theta$. The check is:

$$\begin{array}{ll} \mathsf{R}(\langle\vec{p}\,|\,\varphi\rangle_n) = I_n(\dot{K}(\varphi)\cap\overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{call}_r \\ I_n(W_\pi^\circ(\dot{K}(\pi(\varphi))\cap\Theta)\cap\overrightarrow{W(\overline{p})_\cap}) = \mathsf{R}(\left\langle^{\pi}\blacktriangleright(\langle\vec{q}_1\,|\,\top\rangle_h \,\Vert\, \ldots \,\Vert\, \langle\vec{q}_k\,|\,\top\rangle_h),\vec{p}\,|\,\varphi\right\rangle_n) & \end{array}$$

• $select$:

$$\begin{array}{lll} \left\langle^{\pi}\blacktriangleright(\langle\vec{q}\,|\,\top\rangle_h \,\Vert\, PS),\vec{p}\,|\,\varphi\right\rangle_n & \xrightarrow{select}_p & \left(\left\langle^{\pi}\langle\vec{q}\,|\,\Delta_h^\pi(\varphi)\rangle_h,\vec{p}\,|\,\varphi\right\rangle_n \,\Vert\, \left\langle^{\pi}\blacktriangleright PS,\vec{p}\,|\,\varphi\right\rangle_n\right) \\ I_n(W_\pi^\circ(\dot{K}(\varphi)\cap(\Theta_1\cup\Theta))\cap\overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{select}_r & I_n(W_\pi^\circ(\Theta_1'\cap\dot{K}(\varphi))\cap\overrightarrow{W(\overline{p})_\cap})\ \cup \\ & & I_n(W_\pi^\circ(\dot{K}(\varphi)\cap\Theta)\cap\overrightarrow{W(\overline{p})_\cap}) \\ & & \boxed{\Theta_1\equiv I_m(\overrightarrow{W(\overline{q})_\cap})} \\ & & \boxed{\Theta_1'\equiv I_m(\dot{K}(\exists^{m\uparrow}.\varphi)\cap\overrightarrow{W(\overline{q})_\cap})} \end{array}$$

The check is:

$$\begin{array}{ll} \mathsf{R}(\left\langle^{\pi}\blacktriangleright(\langle\vec{q}\,|\,\top\rangle_h \,\Vert\, PS),\vec{p}\,|\,\varphi\right\rangle_n) = I_n(W_\pi^\circ(\dot{K}(\varphi)\cap(I_h(\overrightarrow{W(\overline{q})_\cap})\cup\mathsf{R}(PS)))\cap\overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{select}_p \\ I_n(W_\pi^\circ(I_h(\dot{K}(\exists^{m\uparrow}.\varphi)\cap\overrightarrow{W(\overline{q})_\cap})\cap\dot{K}(\pi(\varphi)))\cap\overrightarrow{W(\overline{p})_\cap})\ \cup & \\ I_n(W_\pi^\circ(\dot{K}(\pi(\varphi))\cap\mathsf{R}(PS))\cap\overrightarrow{W(\overline{p})_\cap}) = \mathsf{R}((\left\langle^{\pi}\langle\vec{q}\,|\,\Delta_h^\pi(\varphi)\rangle_h,\vec{p}\,|\,\varphi\right\rangle_n \,\Vert\, \left\langle^{\pi}\blacktriangleright PS,\vec{p}\,|\,\varphi\right\rangle_n)) & \end{array}$$

• $return$:

$$\begin{array}{lll} \left\langle^{\pi}\langle\Box\,|\,\psi\rangle_h,\vec{p}\,|\,\varphi\right\rangle_n & \xrightarrow{return}_p & \langle\vec{p}\,|\,\nabla_h^\pi(\psi,\varphi)\rangle_n \\ I_n(W_\pi^\circ(I_m(\dot{K}(\psi))\cap\dot{K}(\varphi))\cap\overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{return}_r & I_n(\dot{K}(\pi^{-1}(\varphi\wedge(\exists^{m\uparrow}.\,\psi)))\cap\overrightarrow{W(\overline{p})_\cap}) \end{array}$$

the check:

$$\begin{array}{ll} \mathsf{R}(\left\langle^{\pi}\langle\Box\,|\,\psi\rangle_h,\vec{p}\,|\,\varphi\right\rangle_n) = I_n(W_\pi^\circ(I_m(\dot{K}(\psi))\cap\dot{K}(\pi(\varphi)))\cap\overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{return}_p \\ I_n(\dot{K}(\pi^{-1}(\pi(\varphi)\wedge(\exists^{m\uparrow}.\,\psi)))\cap\overrightarrow{W(\overline{p})_\cap}) = I_n(\dot{K}(\varphi\wedge\pi^{-1}(\exists^{m\uparrow}.\,\psi))\cap\overrightarrow{W(\overline{p})_\cap}) = & \\ \mathsf{R}(\langle\vec{p}\,|\,\nabla_n^\pi(\varphi,\psi)\rangle) & \end{array}$$

• $return$ second case:

$$\begin{array}{lll} \left\langle^{\pi}\langle \mathit{fail}\rangle,\vec{p}\,|\,\varphi\right\rangle_n & \xrightarrow{return}_p & \langle \mathit{fail}\rangle \\ I_n(W_\pi^\circ(\mathbf{0}\cap\dot{K}(\varphi))\cap\overrightarrow{W(\overline{p})_\cap}) & \xrightarrow{return}_r & \mathbf{0} \end{array}$$

is immediate. □

Lemma 6.52. *The relational transition system of Def. 6.48 is implemented by the rewriting system in Fig. 6.3.*

PROOF. Given that our rewriting system is locally confluent, we can easily check that the transition system is just a collapsing of a particular rewriting chain, omitting uninteresting states. We check the most relevant transitions constraint, fail, call, select and return.

• $constraint$:

$$I_n((\dot{K}(\varphi) \cap \dot{K}(\psi)) \cap \overrightarrow{W(\overline{p})_\cap}) \xrightarrow{constraint}_r I_n(\dot{K}(\varphi \wedge \psi) \cap \overrightarrow{W(\overline{p})_\cap})$$

This transition is implemented by the rewriting rule m_3.

• $fail$:

$$I_n((\dot{K}(\varphi) \cap \dot{K}(\psi)) \cap \overrightarrow{W(\overline{p})_\cap}) \xrightarrow{fail}_r \mathbf{0}$$

This transition is implemented by the rewriting chain:

$$\begin{array}{ll} I_n((\dot{K}(\varphi) \cap \dot{K}(\psi)) \cap \overrightarrow{W(\overline{p})_\cap}) & \xmapsto{P} (m_3*) \\ I_n(\mathbf{0} \cap \overrightarrow{W(\overline{p})_\cap}) & \xmapsto{P} (p_2) \\ I_n(\mathbf{0}) & \xmapsto{P} (m_1*) \\ \mathbf{0} & \end{array}$$

• $call$:

$$I_n((\dot{K}(\varphi) \cap W_\pi(\overline{p})) \cap \overrightarrow{W(\overline{p})_\cap}) \xrightarrow{call}_r I_n(W_\pi^\circ(\dot{K}(\pi(\varphi)) \cap \Theta) \cap \overrightarrow{W(\overline{p})_\cap}) \quad \boxed{\text{with } \overline{p} = \Theta}$$

the transition is implemented by the rewriting chain:

$$\begin{array}{ll} I_n((\dot{K}(\varphi) \cap W_\pi(\overline{p})) \cap \overrightarrow{W(\overline{p})_\cap}) & \xmapsto{P} (p_8) \\ I_n(W_\pi(W_\pi^\circ(\dot{K}(\varphi)) \cap \overline{p}) \cap \overrightarrow{W(\overline{p})_\cap}) & \xmapsto{P} (m_2) \\ I_n(W_\pi(\dot{K}(\pi(\varphi)) \cap \overline{p}) \cap \overrightarrow{W(\overline{p})_\cap}) & \xmapsto{P} (m_4) \\ I_n(W_\pi^\circ(\dot{K}(\pi(\varphi)) \cap \Theta) \cap \overrightarrow{W(\overline{p})_\cap}) & \end{array}$$

• $select$:

$$\begin{array}{ll} I_n(W_\pi^\circ(\dot{K}(\varphi) \cap (\Theta_1 \cup \Theta)) \cap \overrightarrow{W(\overline{p})_\cap}) \xrightarrow{select}_r & I_n(W_\pi^\circ(\Theta_1' \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\overline{p})_\cap}) \cup \\ & I_n(W_\pi^\circ(\dot{K}(\varphi) \cap \Theta) \cap \overrightarrow{W(\overline{p})_\cap}) \\ & \boxed{\Theta_1 \equiv I_m(\overrightarrow{W(\overline{q})_\cap})} \\ & \boxed{\Theta_1' \equiv I_m(\dot{K}(\exists^{m\uparrow}.\varphi) \cap \overrightarrow{W(\overline{q})_\cap})} \end{array}$$

This transition is implemented by the rewriting chain:

$$\begin{array}{ll} I_n(W_\pi^\circ(\dot{K}(\varphi) \cap (\Theta_1 \cup \Theta)) \cap \overrightarrow{W(p)_\cap}) & \xmapsto{P} (p_6) \\ I_n(W_\pi^\circ((\dot{K}(\varphi) \cap \Theta_1) \cup (\dot{K}(\varphi) \cap \Theta)) \cap \overrightarrow{W(p)_\cap}) & \xmapsto{P} (p_3) \\ I_n((W_\pi^\circ(\dot{K}(\varphi) \cap \Theta_1) \cup W_\pi^\circ(\dot{K}(\varphi) \cap \Theta)) \cap \overrightarrow{W(p)_\cap}) & \xmapsto{P} (p_5) \\ I_n((W_\pi^\circ(\dot{K}(\varphi) \cap \Theta_1) \cap \overrightarrow{W(p)_\cap}) \cup (W_\pi^\circ(\dot{K}(\varphi) \cap \Theta) \cap \overrightarrow{W(p)_\cap})) & \xmapsto{P} (p_4) \\ I_n(W_\pi^\circ(\dot{K}(\varphi) \cap \Theta_1) \cap \overrightarrow{W(p)_\cap}) \cup I_n(W_\pi^\circ(\dot{K}(\varphi) \cap \Theta) \cap \overrightarrow{W(p)_\cap}) & \xmapsto{P} (p_9) \\ I_n(W_\pi^\circ(I_m(I_m(\dot{K}(\varphi)) \cap \overrightarrow{W(q)_\cap}) \cap \dot{K}(\varphi)) \cap \overrightarrow{W(p)_\cap}) \cup I_n(W_\pi^\circ(\dot{K}(\varphi) \cap \Theta) \cap \overrightarrow{W(p)_\cap}) & \xmapsto{P} (m_1) \\ I_n(W_\pi^\circ(I_m(\dot{K}(\exists^{m\uparrow}.\, \varphi) \cap \overrightarrow{W(q)_\cap}) \cap \dot{K}(\varphi)) \cap \overrightarrow{W(p)_\cap}) \cup I_n(W_\pi^\circ(\dot{K}(\varphi) \cap \Theta) \cap \overrightarrow{W(p)_\cap}) & \end{array}$$

• $return$:

$$I_n(W_\pi^\circ(I_m(\dot{K}(\psi)) \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\overline{p})_\cap}) \xrightarrow{return}_r I_n(\dot{K}(\pi^{-1}(\varphi \wedge (\exists^{m\uparrow}.\, \psi))) \cap \overrightarrow{W(\overline{p})_\cap})$$

the transition is implemented by the rewriting chain:

$$\begin{array}{ll} I_n(W^\circ_\pi(I_m(\dot{K}(\psi)) \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\bar{p})_\cap}) & \overset{P}{\longmapsto} (m_1) \\ I_n(W^\circ_\pi(\dot{K}(\exists^m\uparrow.\ \psi) \cap \dot{K}(\varphi)) \cap \overrightarrow{W(p)_\cap} & \overset{P}{\longmapsto} (m_3) \\ I_n(W^\circ_\pi(\dot{K}(\exists^m\uparrow.\ \psi \wedge \varphi)) \cap \overrightarrow{W(p)_\cap} & \overset{P}{\longmapsto} (m_2) \\ I_n(\dot{K}(\pi^{-1}(\exists^m\uparrow.\ \psi \wedge \varphi)) \cap \overrightarrow{W(p)_\cap} & \end{array}$$

• $return$ second case:

$$I_n(W^\circ_\pi(\mathbf{0} \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\bar{p})_\cap}) \quad \xrightarrow{return}_r \quad \mathbf{0}$$

the transition is implemented by the rewriting chain:

$$\begin{array}{ll} I_n(W^\circ_\pi(\mathbf{0} \cap \dot{K}(\varphi)) \cap \overrightarrow{W(\bar{p})_\cap}) & \overset{P}{\longmapsto} (p_2) \\ I_n(W^\circ_\pi(\mathbf{0}) \cap \overrightarrow{W(p)_\cap}) & \overset{P}{\longmapsto} (m_2*) \\ I_n(\mathbf{0} \cap \overrightarrow{W(p)_\cap}) & \overset{P}{\longmapsto} (p_2) \\ I_n(\mathbf{0}) & \overset{P}{\longmapsto} (m_1*) \\ \mathbf{0} & \end{array}$$

The sub and seq rules are a consequence of the rewriting strategy used. □

Theorem 6.53. *The rewriting system accurately simulates SLD-resolution.*

Proof. Lem. 6.52 and Theorem 6.46. Indeed, when SLD-resolution diverges, the relational rewriting system does it in the same way. □

Remark 6.54. *This proof technique can be applied to non-deterministic and parallel CLP systems with only slight modifications.*

6.6. Related Work

The origins of this work should be traced to the work of Tarski and Givant. In [Tarski and Givant, 1987], it is proven that the calculus of binary relations with quasi-projections captures set theory in a variable-free manner. In addition, they show that a first-order theorem φ over a theory with a pairing operator has a semantically equivalent equational counterpart $X_\varphi = 1$ in the theory **QRA** of relation algebras with quasi-projections. They also exhibit a bijective recursive transformation of sentences φ to their associated relation expressions X_φ and of first-order proofs of the former to equational derivations of the latter.

Using a variation of this technique in order to compile and execute logic programs was first proposed in in [Broome and Lipton, 1994], and the first concrete rewriting system proposal for SLD programs appears in [Lipton and Chapman, 1998], and while completeness is conjectured some important details are missing. Some other successful applications of rewriting with relations can be seen in [Dougherty and Gutiérrez, 2006], where the theory of representable relation algebras is proved decidable using normal forms of relational terms.

The use of rewriting for solving equational problems is a topic which has been studied in-depth. Some surveys are [Baader and Nipkow, 1998, Dershowitz and Plaisted, 2001]. The work by Common [Comon, 1991] is certainly interesting to our approach and provides insight on how to extend our work beyond Clark's equality theory.

Other interesting approaches that use rewriting and algebraic methods in order to model logic programming can be found in [O'Donnell, 1998, Bonacina and Hsiang, 1992, Kirchner, 1994, 1993].

In addition to the semantics for relations just given, natural categorical models exist: tabular distributive allegories [Freyd and Scedrov, 1991] provide a semantics for a considerably more general notion of logic program, over a finite product category [Finkelstein et al., 1994, 2003, Kinoshita and Power, 1996, Corradini and Montanari, 1992].

Categorical treatments of logic programming provide alternative ways of algebraicizing the subject, which are, in a sense, variable free (see e.g. [Asperti and Martini, 1989, Diaconescu, 1995, Kinoshita and Power, 1996, Finkelstein et al., 1994, 2003, Pym, 1997, 2001a,b]). Corradini and Montanari [Corradini and Montanari, 1992] have given a categorical analysis of logic program execution in terms of transition systems. None of these approaches have as yet been applied to compilation or the definition of an abstract machine for logic programs, although this might be an interesting alternative to the work in this paper.

Combinatory approaches to logic programming have appeared elsewhere in the literature: Bellia and Occhiuto develop an algebra of programs that captures unification, rewriting and narrowing in [Bellia and Occhiuto, 1993]. We feel that our approach is not really comparable, given its applicative nature. We define a more expressive algebra of logic programs, which admits first-order queries, extensions to equational logic, negation and higher order logic, as well as to show that such an algebra can be found within the relation calculus, a well-understood mathematical formalism which easily incorporates other programming paradigms.

Although not directly comparable to our approach, in the reader can see in [Richard, 1995][Kulaš and Beierle, 2001] two different approaches to Logic Programming that use rewriting and rewriting logic as a core mechanism.

Other algebraic proposal such as the one in [Bruni et al., 2001] can be understood as close in spirit to this work, although they keep using meta-logical operations such as renaming apart.

Some semantics based on observables are [Falaschi et al., 1989, Comini et al., 2001]. For instance, the first work defines a non-ground Herbrand universe of tuples (A, σ), with A a non-ground atom and σ the accumulated substitution. Apart from the cardinality considerations that this model may bring in the presence of an infinite number of variables, the semantics presented here is canonical, in the sense that a relation is a canonical representative of the non-ground atom A.

Some semantics based on cylindrical algebras are [van Emden, 2006, de Boer et al., 1997].

In [Elbl, 1999], a new semantics for depth-first logic programming is presented. Biquantale semantics, as the author names it, is declarative, since it is based on a notion of validity in a certain class of models, and its calculus is related to the notion of logic state presented here.

It is worth noting that horn clauses can be interpreted without a database completion, as done in [Gaifman and Shapiro, 1989], where $\leftarrow$ is interpreted as the intuitionistic logical implication.

The work presented here owes a substantial debt to the logic program transformation ideas of Clark [Lloyd, 1984] and Warren [Ait-Kaci, 1991].

Our use of relation algebra formalism as an executable algebra of logic programs, can be understood to belong to the Backus tradition [Backus, 1978]. Seres and Spivey define in [Seres et al., 1999] an algebra for logic programs, using a functional-programming style.

A great deal of work has been done in specification refinement using relational specifications based on Hoare's work [Bird and de Moor, 1996b, Naumann, 1994, Backhouse and Hoogendijk, 1993], as well as on relational approaches to hardware design [Brown and Hutton, 1994, Sheeran and Jones, 1990]. In light of this work, the relational translation described here may provide a new formal link between logic programming, hardware specification, and program synthesis.

In [Berghammer and Hoffmann, 2000] a relational specification is given for depth-first search in a directed graph. A correctness proof is sketched. The authors comment on the advantages of the relational formalism.

Diverse approaches to the formalization of data types in the relation calculus has been studied extensively by the Eindhoven group [Backhouse and Hoogendijk, 1993] and by Bird and de Moor [Bird and de Moor, 1996b].

Work by the RUBY group at Oxford on hardware [Brown and Hutton, 1994, Sheeran and Jones, 1990], and on program synthesis via relations by Bird and de Moor[Bird and de Moor, 1996b], Maddux[Maddux, 1991, 1996], Naumann [Naumann, 1994] and Backhouse [Backhouse and Hoogendijk, 1993], to name a few of the many researchers in this field, suggests that the relational paradigm can

provide significant computational insights at almost every level of the field. Exploration –via relations– of possible connections between program synthesis and a more general notion logic of programming seems particularly tempting.

6.7. Conclusions and Future Work

We have developed a full relational framework for Constraint Logic Programming and proved rigorously both denotational and operational equivalence to the classical approach. We think the approach used has two significant advantages:

- The CLP paradigm is fully captured using relation algebra. This means that we don't need any meta-logical notion. Thus, programs may be fully analyzed, transformed and optimized withing the framework.
- The relational interpretation is used as intermediate code for execution. The only barrier between both worlds is the encoding used in rewriting. We feel it is not possible for specification and implementation to become closer.

The sum of these two facts brings interesting possibilities. For instance, the execution engine can be used in order to perform abstract interpretation, or a debugger may profit from the comprehensiveness of the semantics to get a full trace of the execution tree.

The semantic approach we have used in order to integrate external entities such a the constraint solvers has been useful from a technical point of view. The "black box" approach used here, allows a better handling of the technical aspects when compared to native formulas over the classical Clark's Equality Theory. Also, it provides a good basis for building heterogeneous systems where some parts of the computation is handled by a relational engine and others are not.

Obviously, the correctness of an implementation will also depend on the correctness of the rewriting engine and the correct transcription of the rewriting rules. The nature of our rewriting systems means that modification of the search strategy can be done easily and in a modular way, seeming particularly well suited for parallel execution. The operational equivalence proof can be easily adapted to several other proof search strategies. We would also like to exploit the rich semantics of relational formalisms to obtain new notions of observables, and abstract interpretation, as well as to extend the relational compilation to higher-order logic programming.

To some extent this work is a foray into the terrain of relation-based computing as a separate discipline, with logic programming as an extended case-study . Some instances of "pure" relational programming languages, of limited expressive power, were studied in [Broome and Lipton, 1994]. A useful rewriting system for the full relation calculus didn't seem very well motivated in the absence of some computational paradigm and we thought it would help, at the start, to anchor such a system in logic programming-inspired reductions. Together with Chapter 7, we have obtained good experience of the difficulties involved when computing with relations, and we have advanced our understanding on what is the appropriate framework.

The main lines for future work are two:

- The system presented here is usable, but performance is not comparable to real-world implementations. A fix using category theory has been developed in Chapter 9. However, the main culprit happens in call transitions, where constraint stores are duplicated. A different fix using graph-rewriting [Plump, 1999] and a richer set of relational constraint store operators may be possible. others.
- The second main line of work is to incorporate to our semantics impure features and extensions, such as `var`, `cut`, negation and implication, which are needed in order for the framework to be usable.

It should be noted that the absence of negation in $\mathbf{QRA}_\Sigma$ is no handicap, vis-a-vis first-order formulas with negation over the Herbrand Universe, because of the well-known results of Mal'cev [Mal'tsev, 1961,

Maher, 1988] that any such formula is equivalent to a two-quantifier formula in which negation occurs only immediately preceding equations between terms, which can be modelled in $\mathbf{QRA}_{\Sigma}$ using di for disequality.

Apart from the main line of future work, some parallel work is being performed in defining how some extension fit in our framework. Abstract interpretation [Cousot and Cousot, 1992, Hermenegildo et al., 2005, Puebla et al., 1999], partial evaluation[Consel and Danvy, 1993, Jones et al., 1993, Puebla and Ochoa, 2006] and coinductive approaches such as the ones in [Komendantskaya and Power, 2011b,a, Komendantskaya et al., 2010] are good candidates.

CHAPTER 7

First-Order Unification Using Variable-Free Relation Algebra

The relational representation provides an abstract syntax (i.e., an axiomatic treatment of free and bound variables, quantifiers, abstraction, etc) for logic programs and a formal treatment of logic variables and unification, where meta-logical procedures (name clashes, substitution, etc) are now captured within an object-level algebraic theory.

In the previous chapter, unification was incorporated meta-logically using a constraint solver to solve intersection of relational terms, with the execution details left unspecified. In some regards, this black-box treatment is a natural choice, since this is precisely where constraint systems are added to logic programming engines in conventional constraint logic programming languages, such as CLP(X) [Jaffar and Maher, 1994]. However, the benefits of variable-elimination may be lost here if unification is carried out using typical algorithms that operate directly on the variable names in terms. Then variables must be restored just for the sake of this step.

In this chapter we adapt the framework and develop an algorithm for deciding equality constraint problems in the Herbrand domain, this being equivalent to the classical first-order unification problem. The algorithm proceeds by rewriting relational-term representations.

Although first-order unification is a well-studied and solved problem, an algebraic approach has several advantages. First, cumbersome and underspecified meta-logical procedures (name clashes, substitution, etc.) and their properties (invariance under substitution of ground terms, equality's congruence with respect to term forming, etc.) are captured algebraically within the framework. Second, other constraint problems can be accommodated, for example, existential quantification in the logic can be interpreted as a new operation formalizing renaming apart.

Section 7.1 defines the unification problem. Section 7.2 defines solved forms for our relational representation. Section 7.3 presents the algorithm itself. A detailed example can be found in Section 7.4. In the final part of the chapter we examine related and future work, and wind up with our conclusions.

7.1. The Unification Problem

A first-order unification problem can be represented as a first order formula with equality φ. Then asking whether two terms can be unified is equivalent to the decision problem $\dot{K}(\varphi) = \mathbf{0}$, with $\mathbf{0}$ being the empty relation, as is implied by the equipollence theorem.

Concretely, assume terms t_1, t_2, with free variables $\vec{y}$, and fresh $x_1, z \notin \vec{y}$. We may encode the unification problem $t_1 \approx t_2$ using atomic formulas of the form $x_i = t$:

$$t_1 \approx^? t_2 \iff \exists x_1.\exists \vec{y}.(x_1 = t_1 \wedge x_1 = t_2)$$

Notice the generality and expressiveness of the encoding, thanks to the presence of the existential quantifier. For example, Prolog-like unification with renaming apart (assume $\nu x.t$ means rename apart x in t) can be encoded as a constraint:

$$\begin{aligned} \nu x.t_1 \approx^? t_2 &\iff \exists x_1.\exists \vec{y}.((\exists x.x_1 = t_1) \wedge x_1 = t_2) \\ &\iff \exists x_1.\exists \vec{y}.\exists z.(x_1 = t_1[z/x] \wedge x_1 = t_2) \end{aligned}$$

The encoding can be made more powerful. Adding negation would allow us to handle disunification problems, as well as universal quantification. Summarizing:

$$\begin{aligned} t_1 \approx^? t_2 &\iff \exists x_1.\exists \vec{y}.(x_1 = t_1 \wedge x_1 = t_2) \\ &\iff \dot{K}(\exists x_1.\exists \vec{y}.(x_1 = t_1 \wedge x_1 = t_2)) \neq \mathbf{0} \end{aligned}$$

As the resulting relational expression is ground, and the relational theory is equationally defined, rewriting seems appropriate to handle relational terms. However, two requisites are proven difficult to implement. First, the occur-check, which forces us to make our rewriting system conditional and to compute a side-condition. Second, identifying functions. The rewriting system must match multiple finite sequences of relation compositions as one.

In a relational setting we have no variables and no syntactic notion of substitution. Therefore, the natural output of a unification procedure is some sort of normalized constraint with the right semantics. The next paragraphs will develop this idea.

We focus on the translation K instead of the translation $\dot{K}$ for in $\dot{K}(\exists \vec{y}.(x_1 = t_1 \wedge x_1 = t_2)$, the names of the variables $\vec{y}$ are effectively erased, and we lose the notion of transitivity of the unification relation $\approx$. Indeed,

$$\exists \vec{y}.(x_1 = t_1 \wedge x_1 = t_2) \iff t_1 \approx t_2$$

so we can reduce the unification problem to $K(t_1) \cap K(t_2)$.

The translation K from terms $t \in \mathcal{T}_\Sigma(\mathcal{X})$ to terms in R_Σ is defined in a way that every ground instance of t is in $[\![K(t)]\!]$. We will also define the set of relational terms in K's image inductively and call them $\mathbf{U}$-terms. The result of applying the most general unifier of two terms $t_1, t_2 \in \mathcal{T}_\Sigma(\mathcal{X})$ to any of them is then represented by $K(t_1) \cap K(t_2)$. It is therefore of interest to define and compute a normal form for such an expression which conveys all the desired information.

Definition 7.1. *$K : \mathcal{T}_\Sigma(\mathcal{X}) \to \mathsf{R}$ is defined by induction on $\mathcal{T}_\Sigma(\mathcal{X})$ terms:*

$$\begin{aligned} K(a) &= (a,a)\mathbf{1} \\ K(x_i) &= (P_i)^\circ \\ K(f(t_1,\ldots,t_n)) &= \bigcap_{i \le n} f_i^n K(t_i) \end{aligned}$$

For example, $K(f(x_1, g(a, x_2)))$ yields $f_1^2 P_1^\circ \cap f_2^2(g_1^2(a,a)\mathbf{1} \cap g_2^2 P_2^\circ)$.

In general, the relations semantically denoted by the terms $K(t)$ are non-coreflexive, that is, domain and co-domain are not equal. Indeed, consider the base case of the definition, $K(a) = (a,a)\mathbf{1}$. The semantics of the resulting relation term is the set $\{(a^A, \vec{u}) | \vec{u} \text{ any sequence of terms}\}$. Indeed, we could informally state that the domain any relation generated by $K(t)$ is the set of t's instantiations whereas the co-domain is the set of instantiations of the variables of t. So, for a ground term t, it follows that $[\![K(t)]\!] = \{(t^A, \vec{u}) | \vec{u} \text{ any sequence of terms}\}$. The fact that in the ground case the co-domain is the set of all sequences of terms reflects the invariability of t under substitution.

We may easily deduce the following equation from Lemma 6.5:

$$[\![K(t_1) \cap K(t_2)]\!] = \begin{cases} [\![K(t_1\sigma)]\!] & \text{where } \sigma = \mathbf{mgu}(t_1, t_2) \text{ if exists} \\ \emptyset & \text{otherwise} \end{cases} \tag{6}$$

Definition 7.2 ($\mathbf{U}$-terms)**.** *The set of $\mathbf{U}$-terms (relational terms in K's image) is defined inductively:*

- $\mathbf{0} \in \mathbf{U}$, $(a,a)\mathbf{1} \in \mathbf{U}$ *for every* $(a,a) \in \mathsf{R}_\mathcal{T}$, *and* $P_i^\circ \in \mathbf{U}$ *for every* $i \in \mathbb{N}$.
- *If* $R \in \mathbf{U}$ *then* $f_i^n R \in \mathbf{U}$ *for every* $f_i^n \in \mathsf{R}_\mathcal{T}$.
- *If* $R_1, \ldots, R_n \in \mathbf{U}$ *then* $R_1 \cap \cdots \cap R_n \in \mathbf{U}$.

Definition 7.3 (Unification problem)**.** *The unification problem in this relational setting is to reduce, given terms $t_1, t_2 \in \mathcal{T}_\Sigma(\mathcal{X})$, $K(t_1) \cap K(t_2)$ into an equivalent $\mathbf{U}$-term which can be proved equal or different to $\mathbf{0}$.*

Indeed, the reduction is compositional in the sense that the resulting term can be intersected with a new one to perform unification again.

7.2. Solved forms for U-terms

It is standard practice in decision problems to use a *solved form* which has a trivial decision procedure. The decision problem is then reduced to developing a method for producing solved forms [Comon, 1991]. At the core of our decision procedure is the theory $\mathbf{QRA}_\Sigma$, for it provides a strong enough axiomatization of the original underlying algebra of finite trees.

Given the term $T = R \cap S$, we informally say that R is constrained by S in T and vice versa. We also say that R is obtained from $R \cap S$ by dropping the constraint S.

Definition 7.4 (P-constraint completeness)**.** *A term R is* P-constraint complete *or* $\Xi(R)$ *iff for all subterms t_1 and t_2 of R of the form:*

$$t_1 = P_i^\circ \cap R_1 \cap \cdots \cap R_m$$
$$t_2 = P_j^\circ \cap S_1 \cap \cdots \cap S_n$$

if $i = j$ then the equality $t_1 = t_2$ holds modulo $\cap$-commutativity.

This formally captures the notion that every P_i° appearing in a term must have the same set of constraints. Some examples of P-constraint-complete terms are: $P_1^\circ \cap P_2^\circ$, $f_1^2(P_1^\circ \cap R) \cap g_1^2(R \cap P_1^\circ)$ and $f_1^2(P_1^\circ \cap R) \cap g_1^2(S \cap P_2^\circ)$. Some non-P-constraint-complete terms are: $P_1^\circ \cap f_1^1 P_1^\circ$ and $f_1^2(P_1^\circ \cap R) \cap g_1^2(S \cap P_1^\circ)$.

Definition 7.5 (Indexed P-constraint completeness)**.** *A term R is* P-constraint complete on i *or* $\Xi_i(R)$ *iff for all subterms t_1, t_2 of R of the form:*

$$t_1 = P_j^\circ \cap R_1 \cap \cdots \cap R_m$$
$$t_2 = P_k^\circ \cap S_1 \cap \cdots \cap S_n$$

if $i = j = k$ then the equality $t_1 = t_2$ holds modulo $\cap$-commutativity.

Definition 7.6 (Solved form)**.** $\mathbf{U}$*-terms in solved form are inductively defined for all a, i, j, f, g as:*

R			*for* $R \in \{(a,a)\mathbf{1}, P_i^\circ, \mathbf{0}\}$
R	$\cap$	S	*for* $R, S \in \{(a,a)\mathbf{1}, P_i^\circ\}$
$(a,a)\mathbf{1}$	$\cap$	$f_i^j R$	
$f_i^n R$			*if R in solved form.*
P_i°	$\cap$	$f_j^n R$	*if R in solved form.*
$f_i^m R$	$\cap$	$g_j^n S$	*if $f \neq g$.*
$f_i^n R$	$\cap$	$f_j^n S$	*if R, S in solved form and $\Xi(R \cap S)$.*
R_1	$\cap \cdots \cap$	R_n	*if every pair $R_i \cap R_j, i \neq j$ in solved form.*

If a term t is in solved form we say $solved(t)$. Let $\mathbf{U_S} = \{t \in \mathbf{U} \mid solved(t)\}$ and $\mathbf{U_N} = \{t \in \mathbf{U} \mid \neg solved(t)\}$. By definition, $\mathbf{U_S}$ and $\mathbf{U_N}$ partition $\mathbf{U}$. The inductive definition of unsolved forms is thus obtained from the logical negation of the solved form definition:

Definition 7.7 (Unsolved form)**.** $\mathbf{U}$*-terms in unsolved form are those terms not in solved form. Inductively, for all i, j, f:*

$f_i^n R$			*if R not in solved form.*
P_i°	$\cap$	$f_j^n R$	*if R not in solved form.*
$f_i^n R$	$\cap$	$f_i^n S$	
$f_i^n R$	$\cap$	$f_j^n S$	*if $\neg\Xi(R \cap S)$ or any of R, S not in solved form.*
R_1	$\cap \cdots \cap$	R_n	*if $R_i \cap R_j$ not in solved form for some $i, j, i \neq j$.*

The decision procedure for solved forms is an exhaustive check for incompatible intersections.

Lemma 7.8 (Validity of solved forms)**.** *For any term $t \in \mathbf{U_S}$, we can always decide whether $t = \mathbf{0}$ or, what is the same, whether $[\![t]\!] = \emptyset$.*

PROOF. By cases on t:

- R and $R \cap S$ for $R, S \in \{(a,a)\mathbf{1}, P_i^\circ\}$: Follows directly by term semantics.
- $(a,a)\mathbf{1} \cap f_i^j R$: Always $\mathbf{0}$, because the relation domains are disjoint.
- $f_i^n R$: We check R for validity.
- $P_i^\circ \cap f_j^n R$: We check R for validity.
- $f_i^m R \cap g_j^n S$: Always $\mathbf{0}$, because relation domains are disjoint ($f \neq g$).
- $f_i^n R \cap f_j^n S$: We check R and S for validity. This check is enough because no term $f_i^n R \cap f_i^n S$ is in solved form. Consequently, the domain of the resulting relations is known. Given that $\Xi(R \cap S)$ holds, the codomain is also known, as every projection P_i° into the codomain shares the same set of constraints, so validity of the codomain is effectively reduced to validity of constraints.
- $R_1 \cap \cdots \cap R_n$: Every pair $R_i \cap R_j$ with $i \neq j$ is in solved form, thus the term's validity depends on the pairs' validity.

□

7.3. The algorithm

This section defines an algorithm for computing normal forms of $\mathbf{U}$-terms. The core of the algorithm consists of two term rewriting systems whose effect can be explained by analogy with the classic non-deterministic algorithm (ND) [Martelli and Montanari, 1982]. The first rewriting system (system $\to_{\mathcal{L}}$ defined in Sec. 7.3.2) performs what we call left-factoring, analogous to generation of new equations from a common root term in ND, which is now algebraically understood as the distribution of composition over intersection. The following diagram illustrates ($\mathcal{E}$ stands for a multiset of equations):

$$\{f(t_1,\ldots,t_n) = f(u_1,\ldots,u_n), \mathcal{E}\} \quad \Rightarrow \quad \{t_1 = u_1, \ldots, t_n = u_n, \mathcal{E}\}$$
$$f_i^n R \cap f_i^n S \quad \to_{\mathcal{L}} \quad f_i^n(R \cap S)$$

The other step in ND, equation elimination, is now algebraically understood as constraint propagation (system $\to_{\mathcal{R}}$ defined Sec. 7.3.4):

$$\{x = t, \mathcal{E}\} \quad \Rightarrow \quad \{\mathcal{E}[t/x]\} \qquad \text{when } x \notin t$$
$$F(P_i \cap R) \cap G(P_i \cap S) \quad \to_{\mathcal{R}} \quad F(P_i \cap R \cap S) \cap G(P_i \cap S \cap R)$$

System $\to_{\mathcal{R}}$ is conditional on what we call *functorial compatibility*, a novel way of performing the occur-check which was motivated by $\mathbf{QRA}_\Sigma$'s axiom $f_i^n \cap id = \mathbf{0}$. The two rewriting systems are carefully composed with the help of a constraint-propagation one (system $\to_{\mathcal{S}}$ defined in Section 7.3.3) to guarantee termination.

7.3.1. Rewriting preliminaries. The representation of $\mathbf{U}$-terms in our rewriting systems is given by the following term-forming operations: $\mathsf{c} : \mathcal{C}_\Sigma \to \mathbf{U}$, $\mathsf{t} : (\mathcal{F}_\Sigma \times \mathbb{N} \times \mathbb{N}) \to \mathbf{U}$, $\mathsf{P} : \mathbb{N} \to \mathbf{U}$, $\odot : (\mathbf{U}_1 \times \cdots \times \mathbf{U}_n) \to \mathbf{U}$, and $\cap : (\mathbf{U}_1 \times \cdots \times \mathbf{U}_n) \to \mathbf{U}$, with $n \geq 2$ and n-ary $\odot$ and $\cap$. In addition to the above ground representation, we define patterns of $\mathbf{U}$-terms in a standard way using a set of variables. Let i, j, k etc, range over $\mathbb{N}$. Let a, b, c etc, range over $\mathcal{C}_\Sigma$. Let f, g, h etc, range over $\mathcal{F}_\Sigma$. Let R, S, T etc, range over $\mathbf{U}$-terms. We write $(a,a)\mathbf{1}$ for $\mathsf{c}(a)$, write f_i^j for $\mathsf{t}(f,i,j)$, write P_i° for $\mathsf{P}(i)$, write $R_1 \ldots R_n$ for $\odot(R_1,\ldots,R_n)$, and write $R_1 \cap \cdots \cap R_n$ for $\cap(R_1,\ldots,R_n)$.

Rewrite rules are of the form $\rho : l \to r$ with ρ the rule's name, l and r patterns, and l not a variable. Conditional rewrite rules are of the form $\rho : l \to r \Leftarrow C$ belonging to type III CTRS [Klop, 1992] (see Def. 7.14). We write $\to^!$ for the normalization relation derived from a terminating and confluent

(convergent) relation $\to$. We write $\circ$ for composition of rewriting relations. We use $\equiv$ for syntactic identity (modulo AC).

We use A and AC rewriting for $\odot$ and $\cap$. More precisely, we define the equational theories for relational terms $A_\odot = \{R(ST) = (RS)T\}$ and $AC_\cap = \{R \cap (S \cap T) = (T \cap S) \cap T,\ R \cap S = S \cap R\}$. The AC rewriting used is described in [Dershowitz and Plaisted, 2001, p577–581]: associative term formers are flattened and rewrite rules are extended with dummy variables to take into account the arity of term-forming operations. Matching efficiency can be improved by using ordered rewriting.

A rewrite rule $\rho : l \to r$ matches a term t iff $l\sigma = t$ modulo $A_\odot$ and $AC_\cap$. A sequence of integers p is called a position. $f(t_1, \ldots, t_n)_{|\langle i,l \rangle} \equiv t_{i|l}$, given $1 \le i \le n$ and $t_{|\langle\rangle} = t$. We allow subterm matching: if there exists a position p such that $l\sigma = t_{|p}$ (modulo previous AC), then t reduces to $t\{t_{|p} \mapsto r\sigma\}$. Importantly, we also allow matching over *functorial variables* which represent the largest composition of f_i^n terms.[1] We use F, G, H etc, for functorial variables. The expression $length(F)$ delivers the length of functorial variable F. For example, the term $r_1^2 s_2^2((a,a)\mathbf{1} \cap (b,b)\mathbf{1} \cap t_1^1 P_1^\circ)$ matches $F(R \cap GS)$ with $\sigma = \{F \mapsto r_1^2 s_2^2,\ R \mapsto (a,a)\mathbf{1} \cap (b,b)\mathbf{1},\ G \mapsto t_1^1,\ S \mapsto P_1^\circ\}$. Matching over functorial variables is a sort of specialized list-matching. Note that such a variable always matches the largest sequence possible.

7.3.2. Left-factoring rewriting system.

Definition 7.9. *The rewriting system $\mathcal{L}$ consists of the set of rewrite rules (where i ranges over all indices and f over all function symbols):*

$$f_i^n R \cap f_i^n S \quad \to_{\mathcal{L}} \quad f_i^n(R \cap S)$$

Lemma 7.10. *$\mathcal{L}$ is sound.*

Proof. Soundness is a consequence of the following equation:

$$RS \cap RT = R(S \cap T) \tag{7}$$

which holds in **DRA** when R is functional (which is the case for every f_i^n):

$$R(S \cap T) \supseteq_{\textbf{[by } id \,\supseteq\, R^\circ R\textbf{]}} R(S \cap R^\circ RT) \supseteq_{\textbf{[by modular law]}} RS \cap RT$$

Conversely, $RS \cap RT \supseteq R(S \cap T)$ by **DRA**. □

Lemma 7.11. *$\mathcal{L}$ is terminating and confluent.*

Proof. To prove termination it suffices to give a lexicographic path ordering on terms [Dershowitz, 1982]. The ordering is $\cap \succ \odot$. The system has no critical pairs, so it is locally confluent, local confluence plus termination implies confluence [Knuth and Bendix, 1983]. □

7.3.3. Split-rewriting system.

Definition 7.12. *The rewriting system $\mathcal{S}$ consists of the rewrite rules:*

$$F(R \cap G(P_i^\circ \cap S)) \to_{\mathcal{S}} FR \cap FG(P_i^\circ \cap S)$$

Lemma 7.13. *$\mathcal{S}$ is sound, terminating and confluent.*

Proof. Soundness and closure properties are immediate from (7). Termination is proven giving the lexicographic path ordering $\odot \succ \cap$ and confluence follows from the fact that $\to_{\mathcal{S}_i}$ is terminating and locally confluent. □

We will also use a parametrized version of $\mathcal{S}$, written $\mathcal{S}_i$ where i is fixed.

[1] We use “functorial” in connection with function symbols (term formers), not with functors in category theory.

7.3.4. Constraint-propagation rewriting system. The purpose of this system is to propagate constraints over P_i° terms. Constraint propagation has two main technical difficulties. First, addressing occur-check to avoid infinite rewriting. Second, propagating constraints before checking for term clashes. Both difficulties can be addressed by introducing a decidable notion of functorial compatibility:

Definition 7.14. *The convergent rewriting relation $\rightarrow_\Delta$ is defined as:*

$$(f_i^n)^\circ f_i^n \rightarrow_\Delta id \qquad (f_i^n)^\circ g_j^m \rightarrow_\Delta \mathbf{0} \qquad (f_i^n)^\circ f_j^n \rightarrow_\Delta \mathbf{1} \qquad id f_i^n \rightarrow_\Delta f_i^n$$
$$(f_i^n)^\circ \mathbf{1} \rightarrow_\Delta \mathbf{1} \qquad \mathbf{1} f_i^n \rightarrow_\Delta \mathbf{1} \qquad (f_i^n)^\circ \mathbf{0} \rightarrow_\Delta \mathbf{0} \qquad \mathbf{0} f_i^n \rightarrow_\Delta \mathbf{0}$$

The above convergent rewriting relation gives rise to their associated function:

Definition 7.15 (Functorial delta)**.** *Given functorials F and G, we define $\Delta(F,G)$ as follows:*

$$\begin{array}{lll} \Delta(F,G) = S, & F^\circ G \rightarrow_\Delta^! S & \textit{if } length(G) \geq length(F) \\ \Delta(F,G) = S, & G^\circ F \rightarrow_\Delta^! S & \textit{if } length(G) < length(F) \end{array}$$

Lemma 7.16.

$$\Delta(F,G) = \begin{cases} \mathbf{0} & \textit{if } [\![F \cap G]\!] = \emptyset \\ id & \textit{if } F \equiv G. \\ S & \textit{if } G \equiv FS \\ S & \textit{if } F \equiv GS \\ \mathbf{1} & \textit{otherwise.} \end{cases}$$

Proof. By induction on functorial terms. □

Remark 7.17. *We may understand functorials as pointers to a position in a term. From this standpoint, Δ handles "pointer interference". That is, whether the functorials "point" to the same, to different but compatible, or to incompatible positions in a term. The delicate case is when one functorial points to a sub-position of another.*

Definition 7.18 (Syntactic difference)**.** *The syntactic difference $\Theta(R_1 \cap \cdots \cap R_m, S_1 \cap \cdots \cap S_n)$ between two arbitrary-length intersection of terms is defined as the term $\bigcap_{i \in D} S_i$ such that $(\forall j \in \{1..m\}.R_j \not\equiv S_i) \iff i \in D$. Abusing notation, we use $\{\}$ for the empty intersection.*

Lemma 7.19. $R \cap \Theta(R,S) \equiv S \cap \Theta(S,R)$

Proof. By induction on the length of R. □

Recall that $\equiv$ is syntactic identity modulo AC (Section 7.3.1). Should we use the idempotency axiom $A \cap A \equiv A$ to check equality modulo ACI, we would get an extended version of Lemma 7.19:

$$R \cap S \equiv R \cap \Theta(R,S) \equiv S \cap \Theta(S,R) \equiv S \cap R$$

Corollary 7.20. *For any term R, $\Theta(R,R) = \{\}$.*

Definition 7.21. *The rewriting system $\mathcal{R}$ consists of the rewrite rules:*

$$\begin{array}{ll} R0: & P_i^\circ \cap F(P_i^\circ \cap S) \rightarrow_\mathcal{R} \mathbf{0} \\ R1: & F(P_i^\circ \cap R) \cap G(P_i^\circ \cap S) \rightarrow_\mathcal{R} \mathbf{0} \quad \Leftarrow \Delta(F,G) = \mathbf{0} \vee \Delta(F,G) = f_i^n \ldots g_j^m \\ R2: & F(P_i^\circ \cap R) \cap G(P_i^\circ \cap S) \rightarrow_\mathcal{R} F(P_i^\circ \cap R \cap \Theta(R,S)) \cap G(P_i^\circ \cap S \cap \Theta(S,R)) \\ & \qquad \Leftarrow (\Theta(R,S) \neq \{\} \vee \Theta(S,R) \neq \{\}) \wedge (\Delta(F,G) = \mathbf{1} \vee \Delta(F,G) = id) \end{array}$$

Notice Θ helps us deal conveniently with the equivalence of terms $R \cap S$ and $S \cap R$. We will also use a parametrized version of $\mathcal{R}$, written $\mathcal{R}_i$, where i is fixed.

Lemma 7.22. *$\mathcal{R}$ is sound.*

PROOF. $R0$ is sound beacuse every term matching the left hand side can be factored into a term of the form $id \cap f_i^n \ldots g_j^m$ using $RP_i^\circ \cap SP_i^\circ = (R \cap S)P_i^\circ$ which is a version of Eq. (8) below. $R1$ is sound for two reasons:

(1) If $\Delta(F, G) = \mathbf{0}$ then F and G are incompatible and the left hand side rewrites to $\mathbf{0}$ by left-factoring.
(2) If $\Delta(F, G) = f_i^n \ldots g_j^m$ then there is a common prefix F' of F and G such that $F \cap G = F'(id \cap f_i^n \ldots g_j^m)$ and the right-hand-side rewrites to $\mathbf{0}$ by $\mathbf{QRA}_\Sigma$'s occur-check axiom.

That $R2$ is sound follows from the equation:

$$F(P_i^\circ \cap R) \cap GP_i^\circ = F(P_i^\circ \cap R) \cap G(P_i^\circ \cap R) \tag{8}$$

Recall that a relation is injective when $RR^\circ \subseteq id$. Recall Lemma 7.10 which states that $RT \cap ST = (R \cap S)T$ for T injective. Using these facts we prove the $\subseteq$ direction:

$$F(P_i^\circ \cap R) \cap GP_i^\circ =_{\textbf{[left+right factoring]}} (F \cap G)P_i^\circ \cap FR \subseteq_{\textbf{[modular law]}}$$
$$(F \cap G)((F^\circ \cap G^\circ)FR \cap P_i^\circ) \subseteq_{\textbf{[by } (F^\circ \cap G^\circ)F \subseteq F^\circ F = id\textbf{]}} (F \cap G)(R \cap P_i^\circ)$$
$$=_{\textbf{[injectivity of } (R \cap P_i^\circ)\textbf{]}} F(R \cap P_i^\circ) \cap G(R \cap P_i^\circ)$$

The $\supseteq$ direction is easier:

$$F(P_i^\circ \cap R) \cap GP_i^\circ \supseteq_{\textbf{[monotonicity of } \cap\textbf{]}} F(P_i^\circ \cap R) \cap G(P_i^\circ \cap R)$$

□

The following example illustrates why the compatibility check is needed to ensure termination. Take the term $P_1^\circ \cap f_1^1(P_1^\circ \cap R)$. If we propagate restrictions by orienting (8) we get an infinite rewrite: $P_1^\circ \cap f_1^1(P_1^\circ \cap R) \to P_1^\circ \cap f_1^1(R \cap P_1^\circ \cap f_1^1(P_1^\circ \cap R)) \to P_1^\circ \cap f_1^1(R \cap P_1^\circ \cap f_1^1(P_1^\circ \cap R \cap f_1^1(P_1^\circ \cap R))) \ldots$

Definition 7.23. *Let $\succ_C$ be the order relation:*

$$F(P_i^\circ \cap R) \cap G(P_i^\circ \cap S) \succ_C F(P_i^\circ \cap R \cap \Theta(R, S)) \cap G(P_i^\circ \cap S \cap \Theta(S, R))$$

where $\Theta(R, S) \neq \{\}$ and $\Theta(S, R) \neq \{\}$.

Informally, think of $\succ_C$ being defined over the measure "number of different elements of R and S".

We prove $\succ_C$ is well-founded using Lemma 7.19.

Lemma 7.24. *$\succ_C$ is well-founded.*

PROOF. $\succ_C$ has bounded depth, with no chain of length longer than two. Suppose $\succ_C$ had a chain of length three:

$$\begin{array}{ll} F(P_i^\circ \cap R) \cap G(P_i^\circ \cap S) & \succ_C \\ F(P_i^\circ \cap R \cap \Theta(R, S)) \cap G(P_i^\circ \cap S \cap \Theta(S, R)) & \succ_C \\ F(P_i^\circ \cap R \cap \Theta(R, S) \cap \Theta(R \cap \Theta(R, S), S \cap \Theta(S, R))) \cap & \\ G(P_i^\circ \cap S \cap \Theta(S, R) \cap \Theta(S \cap \Theta(S, R), R \cap \Theta(R, S))) & \end{array}$$

and note that in order for the chain to exist $\Theta(R \cap \Theta(R, S), S \cap \Theta(S, R)) \neq \{\}$. But using Lemma 7.19, $R \cap \Theta(R, S) \equiv S \cap \Theta(S, R)$, so by Corollary 7.20, $\Theta(R \cap \Theta(R, S), S \cap \Theta(S, R)) = \{\}$, contradicting the existence of the chain. □

Lemma 7.25. *$\mathcal{R}$ terminates.*

PROOF. Rules $R0$ and $R1$ rewrite to $\mathbf{0}$. For rule $R2$ we define a well-founded order that contains the relation induced by $R2$. Notice that the finiteness of the terms involved in the rewriting ensures that the transformation for handling the n-ary term former $\cap$ doesn't affect termination. □

Lemma 7.26. *$\mathcal{R}$ is confluent (modulo commutativity).*

PROOF. We give the same lexicographic path ordering as Lemma 7.11 and $\mathcal{R}$ has no overlapping rules. □

7.3.5. The algorithm. Unfortunately, a naïve application of the previous rewriting rules does not necessarily reach a solved form. Take for example the term $r_1^2(P_1^\circ \cap s_1^1(P_2^\circ \cap (a,a)\mathbf{1})) \cap r_2^2(P_2^\circ \cap s_1^1(P_1^\circ \cap (b,b)\mathbf{1}))$. If we apply the rewriting strategy $\rightarrow^!_{\mathcal{S}} \circ \rightarrow^!_{\mathcal{R}} \circ \rightarrow^!_{\mathcal{L}}$ we do not reach a solved form because $\rightarrow^!_{\mathcal{S}}$ destroys constraints over P_1°, impeding the constraint propagation step to work properly. One solution is to complete the constraints for each P_i° one at a time.

Lemma 7.27. *Given a rewriting $t \rightarrow^!_{\mathcal{S}_i} t'$, every P_i° in t' occurs at* top level relative to functorials*, i.e., t' has the form:*

$$P_i^\circ \cap F(P_i^\circ \cap R) \cap \dots \cap G(P_i^\circ \cap S) \cap \dots$$

Proof. If the P_i° term occurs deeper in the term, t' has the form:

$$F(R \cap G(S \cap \cdots H(P_i^\circ \cap T))) \cap \dots$$

which is not a normal form for $\rightarrow^!_{\mathcal{S}_i}$. □

Definition 7.28 (Individual constraint propagation)**.** *The rewrite relation $\rightarrow_{\mathcal{U}_i}$ parametrized on i is defined as:*

$$\rightarrow^!_{\mathcal{L}} \circ \rightarrow^!_{\mathcal{S}_i} \circ \rightarrow^!_{\mathcal{R}_i} \circ \rightarrow^!_{\mathcal{L}}$$

Lemma 7.29. *$\Xi_i(t')$ holds for every term t and reduction $t \rightarrow_{\mathcal{U}_i} t'$.*

Proof. Assume a term t exists such that $t \rightarrow_{\mathcal{U}_i} t'$ and $\neg\Xi_i(t')$. There are subterms of t' of the form $P_i^\circ \cap R$ and $P_i^\circ \cap S$, with $\Theta(R,S) \neq \{\}$ or $\Theta(S,R) \neq \{\}$. Let $t \rightarrow^!_{\mathcal{L}} u$, then no subterm of the form $FR \cap FS$ exists in u. Let $u \rightarrow^!_{\mathcal{S}_i} v$, then every P_i° in v is of the form described in Lemma 7.27. Let $v \rightarrow^!_{\mathcal{R}_i} w$, then $\Xi_i(w)$ because w is a normal form of $\rightarrow_{\mathcal{R}_i}$ and $P_i^\circ \cap R$ and $P_i^\circ \cap S$ with $\Theta(R,S) \neq \{\}$ and $\Theta(S,R) \neq \{\}$ cannot occur at the top-level, and every P_i° is at the top level. Let finally $w \rightarrow^!_{\mathcal{L}} t'$. For $\rightarrow^!_{\mathcal{L}}$ to break P-constraint completeness, w must have a term of the form $FP_i^\circ \cap FR$ such that after $\rightarrow^!_{\mathcal{L}}$, R becomes a new constraint on P_i°. However this cannot happen, for u had no such terms and v had only terms of the form $F(P_i^\circ \cap R) \cap FG(P_i^\circ \cap R)$, which are rewritten to **0** by $R1$. □

Lemma 7.30. *Given t such that $\Xi_i(t)$ holds then $t \rightarrow^!_{\mathcal{S}_i} t'$ and $\Xi_i(t')$.*

Proof. Follows from $\rightarrow_{\mathcal{S}_i}$ rules because if $\Xi(t)$ then no term of the form $F(P_i^\circ \cap GR)$ with P_i° in R can occur in t. Such occurrence is the necessary condition for $\rightarrow^!_{\mathcal{S}_i}$ to destroy P-constraint completeness. □

Lemma 7.31. *Given t such that $\Xi_i(t)$ holds then $t \rightarrow^!_{\mathcal{R}_i} t$.*

Proof. Follows from the definition of $\rightarrow_{\mathcal{R}_i}$. Note that if $R0$ or $R1$ are applicable for a term t, this easily implies $\neg\Xi(t)$. □

Lemma 7.32. *$\rightarrow_{\mathcal{U}_i}$ reaches a fixed point, in other words, given a term t then $t \rightarrow_{\mathcal{U}_i} t' \rightarrow_{\mathcal{U}_i} t''$ and $t' = t''$.*

Proof. $\Xi_i(t')$ holds by Lemma 7.29. In the second $\rightarrow_{\mathcal{U}_i}$ step, $t' \rightarrow^!_{\mathcal{L}} t'$ holds. By Lemma 7.30, $t' \rightarrow^!_{\mathcal{S}_i} u$ and $\Xi_i(u)$. By Lemma 7.31, we have $u \rightarrow^!_{\mathcal{R}_i} u$. Finally, we need to prove $t' \rightarrow^!_{\mathcal{S}_i} u \rightarrow^!_{\mathcal{L}} t'$. This is proven by the fact that $\rightarrow^!_{\mathcal{L}}$ *undoes* everything $\rightarrow^!_{\mathcal{S}_i}$ does on already factorized terms, which is the case of t'. Given rewriting rules $F(R \cap G(P_i^\circ \cap S)) \rightarrow_{\mathcal{S}_i} FR \cap FG(P_i^\circ \cap S)$ and $FT \cap FU \rightarrow_{\mathcal{L}} F(T \cap U)$, their composition happens with substitution $\{T \mapsto R,\ U \mapsto G(P_i^\circ \cap S)\}$, resulting in the rewriting rule $F(R \cap G(P_i^\circ \cap S)) \rightarrow_{\mathcal{L} \circ \mathcal{S}_i} F(R \cap G(P_i^\circ \cap S))$ which is the identity. □

Definition 7.33 (P-dependency)**.** *Given a term $P_i^\circ \cap R$, we say i P-depends on j if $j \in dep(R)$ where $dep : \mathbf{U} \rightarrow \mathcal{P}(\mathbb{N})$ is defined in Fig. 7.1. We can build the P-dependency for a term t taking all its subterms in the form $P_i^\circ \cap R$.*

$$
\begin{array}{lcl|lcl}
dep(P^\circ_j) & = & \emptyset & dep'(P^\circ_j) & = & \{j\} \\
dep(R \cap S) & = & dep(R) \cup dep(S) & dep'(R \cap S) & = & dep'(R) \cup dep'(S) \\
dep(f^n_i R) & = & dep'(R) & dep'(f^n_i R) & = & dep'(R)
\end{array}
$$

FIGURE 7.1. Definition of $dep : \mathbf{U} \to \mathcal{P}(\mathbb{N})$

Lemma 7.34 (Constraint destruction)**.** *Given $\Xi_j(t)$, $t \to_{\mathcal{U}_i} t'$ can make $\neg\Xi_j(t')$ iff j P-depends on i and $i \neq j$.*

PROOF. The only way $\to_{\mathcal{U}_i}$ can add a new constraint to a P°_j by constraint propagation is if the term is in the form $P^\circ_j \cap FR$ and P°_i occurs in R which is precisely the definition of P-dependency. □

Lemma 7.35 (Occurs check)**.** *If i P-depends on i in a term t then $t \to_{\mathcal{U}_i} \mathbf{0}$.*

PROOF. Such P-dependency means $P^\circ_i \cap FR$ and P°_i occurs in R which gets rewritten to $\mathbf{0}$ by $\to^!_{\mathcal{R}_i}$. □

Definition 7.36 (Solved form algorithm)**.** *Given a term $t \in \mathbf{U}$, containing P°_i terms with $i \in \{1 \dots n\}$ and $n \geq 1$, we define the rewriting relation $\to_{\mathcal{U}_{\downarrow n}}$ as $\to_{\mathcal{U}_1} \circ \cdots \circ \to_{\mathcal{U}_n}$, that is:*

$$\to^!_{\mathcal{L}} \circ \to^!_{\mathcal{S}_1} \circ \to^!_{\mathcal{R}_1} \circ \to^!_{\mathcal{L}} \circ \to^!_{\mathcal{S}_2} \circ \to^!_{\mathcal{R}_2} \circ \to^!_{\mathcal{L}} \circ \cdots \circ \to^!_{\mathcal{S}_n} \circ \to^!_{\mathcal{R}_n} \circ \to^!_{\mathcal{L}}$$

For $n = 0$, there are no P°_i in t and we define $\to_{\mathcal{U}_{\downarrow 0}}$ as $\to^!_{\mathcal{L}}$.

Lemma 7.37. *There is a finite k such that $t \to^k_{\mathcal{U}_{\downarrow n}} t'$ and $\Xi(t')$.*

PROOF. The case $n = 0$ follows from Lemma 7.11 with $k = 1$. The case $n = 1$ follows from Lemma 7.32 with $k = 1$. In the case $n > 1$, for a step $u \to_{\mathcal{U}_i} u'$ then $\Xi_i(u')$ by Lemma 7.29. However, by Lemma 7.34 such a step can make $\neg\Xi_j(u')$ hold iff j P-depends on i. Suppose the P-dependency graph of t is acyclic. Then there is a set of terminal edges E such that forall $l \in E$, Ξ_l holds after $\to_{\mathcal{U}_l}$ and no other step $\to_{\mathcal{U}_i}$ can make Ξ_l false, for they depend on no j. Once the term is Ξ_l for every $l \in E$, E can be removed from the graph. The process can be repeated with the new set E' of terminal edges a finite number of times, for the graph is acyclic and the number of edges is finite. Finally if the P-dependency graph of the original term is cyclic, then by Lemma 7.35 the term would get rewritten to $\mathbf{0}$ in the $\to_{\mathcal{U}_i}$ iteration corresponding to any i on the cycle. Thus when the process ends, $\Xi_l(t')$ for all $l \in \{1 \dots n\}$ which is equivalent to $\Xi(t')$. □

Lemma 7.38. *$\to_{\mathcal{U}_{\downarrow n}}$ reaches a fixed point.*

PROOF. The proof is similar to the one in Lemma 7.32, but using Lemma 7.37, for once $\Xi(t)$ holds, $\to_{\mathcal{U}_{\downarrow n}}$ application does not modify t. □

Lemma 7.39. *The fixed point for $\to_{\mathcal{U}_{\downarrow n}}$ is in solved form.*

PROOF. We check by induction that no unsolved term can be a fixed point:

- $f^n_i R$ is unsolved when R is unsolved, this means one of the cases apply for R.
- $P^\circ_i \cap f^n_j R$, if $P^\circ_i \notin R$ and R not in solved form: If R is in unsolved form then one of the cases below applies. Otherwise, if P°_i is a subterm of R when $\to^!_{\mathcal{R}_i}$ fires, it rewrites to $\mathbf{0}$.
- $f^n_i R \cap f^n_i S$: Terms of this form match the left side of $\to_{\mathcal{L}}$.
- $f^n_i R \cap f^n_j S$, we have two cases:
  - $\Xi(R \cap S)$ doesn't hold: Lemma 7.38 implies that $\to_{\mathcal{U}_n}$ reaches a fixed point for t precisely when $\Xi(t)$ holds, so $\neg\Xi(R \cap S)$ is a contradiction.
  - Any of R, S are in unsolved form: for the term in unsolved form one of the cases of the definition of unsolved applies.

- $R_1 \cap \cdots \cap R_n$, if some pair $R_i \cap R_j, i \neq j$ is in unsolved form: If the pair R_i, R_j is in unsolved form one of the cases above apply.

□

Definition 7.40 (Relational unification)**.** *We define the relation between* $\mathbf{U}$*-terms with maximum index* n*, denoted* $t \rightarrow_{\{UNIF,n\}} t'$*, as follows: If* $t \rightarrow^{!}_{\mathcal{U}_{\downarrow n}} s$ *and* s *is not valid then* $t' = \mathbf{0}$*; otherwise,* $t' = s$*.*

Theorem 7.41. *Terms* t_1, t_2 *with* n *different variables are not unifiable iff* $K(t_1) \cap K(t_2) \rightarrow_{\{UNIF,n\}} \mathbf{0}$*.*

Proof. By Lemma 7.39 we know that $\rightarrow^{!}_{\mathcal{U}_{\downarrow n}}$ brings any $\mathbf{U}$-term to solved form. As we have a complete decision procedure for solved terms by Lemma 7.8, we can decide if $K(t_1) \cap K(t_2)$ is $\mathbf{0}$, which means by Eq. 6 that t_1, t_2 are not unifiable. □

Definition 7.42 (Relational unifiers)**.** *We say a term* t *in solved form has a unifier* R *for* i *if* $P_i^\circ \cap R$ *is a subterm of* t*.*

For example, suppose we have the term $P_1^\circ \cap (a,a)\mathbf{1} \cap P_2^\circ$. This means in $\mathcal{T}_\Sigma(\mathcal{X})$ that $x_1 = x_2 = a$.

7.4. An Example

We will use the example given in [Martelli and Montanari, 1982]:

$$\begin{aligned} t_1 &= f(g(h(a,x_5),x_2),x_1,h(a,x_4),x_4) \\ t_2 &= f(x_1,g(x_2,x_3),x_2,b) \end{aligned}$$

First, we use K-translation to translate the term into a suitable relational form:

$$\begin{aligned} K(t_1) &= f_1^4(g_1^2(h_1^2(a,a) \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ) \cap f_2^4 P_1^\circ \cap f_3^4(h_1^2(a,a) \cap h_2^2 P_4^\circ) \cap f_4^4 P_4^\circ \\ K(t_2) &= f_1^4 P_1^\circ \cap f_2^4(g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) \cap f_3^4 P_2^\circ \cap f_4^4(b,b)\mathbf{1} \end{aligned}$$

So $K(t_1) \cap K(t_2)$ is:

$$\begin{array}{lll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ) & \cap f_1^4 P_1^\circ & \cap \\ f_2^4 P_1^\circ & \cap f_2^4(g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_4^\circ) & \cap f_3^4 P_2^\circ & \cap \\ f_4^4 P_4^\circ & \cap f_4^4(b,b)\mathbf{1} & \end{array}$$

Now we apply our rewriting strategy. For the sake of brevity, we will rewrite variables in the optimal order, which is easily calculable after factoring. We first factor the term:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ \cap P_1^\circ) & \cap \\ f_2^4(P_1^\circ \cap g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_4^\circ \cap P_2^\circ) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

One of the orders induced by Γ is $1, 3, 2, 4, 5$. We will start with variable P_5°. The first step is to split on 5. We apply $\rightarrow_{\mathcal{S}_5}$:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1}) \cap g_2^2 P_2^\circ \cap P_1^\circ) \cap f_1^4 g_1^2 h_2^2 P_5^\circ & \cap \\ f_2^4(P_1^\circ \cap g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_4^\circ \cap P_2^\circ) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

No constraint can be propagated, so we come back and split on 4:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ \cap P_1^\circ) & \cap \\ f_2^4(P_1^\circ \cap g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap P_2^\circ) \cap f_3^4 h_2^2 P_4^\circ & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

At this point some propagation takes place:

$$\begin{array}{ll} f_3^4 h_2^2 P_4^\circ \cap f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \to_{\mathcal{R}} \\ f_3^4 h_2^2(P_4^\circ \cap (b,b)\mathbf{1}) \cap f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

The full term is now (after factoring again):

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ \cap P_1^\circ) & \cap \\ f_2^4(P_1^\circ \cap g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap P_2^\circ \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

We factor for 2:

$$\begin{array}{ll} f_1^4 g_2^2 P_2^\circ & \cap \\ f_2^4 g_1^2 P_2^\circ & \cap \\ f_3^4(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cdots \end{array}$$

and we get propagated the constraint $h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})$ to the other terms yielding a result:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) \cap P_1^\circ) & \cap \\ f_2^4(P_1^\circ \cap g_1^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap P_2^\circ \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

For P_3° there is only one occurrence, so we go to propagate on P_1°.

$$\begin{array}{ll} f_1^4(P_1^\circ \cap g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1}))) & \cap \\ f_2^4(P_1^\circ \cap g_1^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) \cap g_2^2 P_3^\circ) & \cap \end{array}$$

which results in

$$\begin{array}{lll} f_1^4(P_1^\circ \cap & g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_1^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ & g_2^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1}) \cap g_2^2 P_3^\circ) & \end{array}$$

after factoring becomes:

$$\begin{array}{lll} f_1^4(P_1^\circ \cap & g_1^2(h_1^2(a,a)\mathbf{1} \cap P_2^\circ h_2^2(P_4^\circ \cap P_5^\circ \cap (b,b)\mathbf{1})) & \cap \\ & g_2^2(P_2^\circ \cap P_3^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \end{array}$$

The final term:

$$\begin{array}{ll} f_1^4(P_1^\circ \cap g_1^2(h_1^2(a,a)\mathbf{1} \cap P_2^\circ h_2^2(P_4^\circ \cap P_5^\circ \cap (b,b)\mathbf{1})) & \cap \\ \qquad g_2^2(P_2^\circ \cap P_3^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_2^4(P_1^\circ \cap g_1^2(h_1^2(a,a)\mathbf{1} \cap P_2^\circ h_2^2(P_4^\circ \cap P_5^\circ \cap (b,b)\mathbf{1})) & \cap \\ \qquad g_2^2(P_2^\circ \cap P_3^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap P_2^\circ \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

We checked the term is valid: we couldn't find an invalidating constraint. Now, we erase the P_i° terms to help see better what actually happened:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1}) \cap g_2^2(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1})) & \cap \\ f_2^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1}) \cap g_2^2(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1})) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1}) & \cap \\ f_4^4(b,b)\mathbf{1} & \end{array}$$

Enhanced visualization:

$$\begin{array}{ll} f_1(g_1(h_1\mathbf{a} \cap h_2\mathbf{b}) \cap g_2(h_1\mathbf{a} \cap h_2\mathbf{b})) & \cap \\ f_2(g_1(h_1\mathbf{a} \cap h_2\mathbf{b}) \cap g_2(h_1\mathbf{a} \cap h_2\mathbf{b})) & \cap \\ f_3(h_1\mathbf{a} \cap h_2\mathbf{b}) & \cap \\ f_4\mathbf{b} & \end{array}$$

If we worry about space we can borrow categorical notation to stretch the text:

$$f\langle g\langle h\langle \mathbf{a}, \mathbf{b}\rangle, h\langle \mathbf{a}, \mathbf{b}\rangle\rangle, g\langle h\langle \mathbf{a}, \mathbf{b}\rangle, h\langle \mathbf{a}, \mathbf{b}\rangle\rangle, h\langle \mathbf{a}, \mathbf{b}\rangle, \mathbf{b}\rangle$$

or indeed with $\delta : 1 \to 2, \delta(f) = \langle f, f\rangle$:

$$f\langle gh\langle \mathbf{a}, \mathbf{b}\rangle\delta^2, h\langle \mathbf{a}, \mathbf{b}\rangle, \mathbf{b}\rangle$$

which can be translated back to

$$f(g(h(a, b), h(a, b)), g(h(a, b), h(a, b)), h(a, b), b)$$

7.5. Related work

Unification. Unification was first proposed in [Robinson, 1965] and axiomatized in [Maher, 1988, Mal'tsev, 1961]. Classical algorithms for first-order unification are surveyed in [Baader and Snyder, 2001], such as [Martelli and Montanari, 1982, Paterson and Wegman, 1978] which define fast and well-understood algorithms targeted to implementors using imperative languages. Other interesting approaches to unification include categorical views [Goguen, 1989, Rydeheard and Burstall, 1986], unification for lambda terms [Huet, 1975, Dowek, 2001] and a remarkable one [Dougherty, 1993], which studies unification in the typed combinatorial paradigm [Curry and Feys, 1958]. Nominal unification [Urban et al., 2003] is an alternative approach to meta-theory formalization that uses nominal logic. Indeed, nominal techniques [Gabbay, 2011] have been established as a solid approach to meta-theory, binding, and syntax formalization giving rise to nominal rewriting [Fernández and Gabbay, 2007] and nominal algebra [Gabbay and Mathijssen, 2009].

C-expressions [Bellia and Occhiuto, 1999, 1993] also provide a combinatorial unification algorithm, based on applicative terms instead of relations. Unification using explicit substitutions is handled in the higher-order case in [Dowek et al., 2000].

In practical terms, our work is not different from these approaches because the result of the algorithm is a unifier.

However, our work is very different in its theoretical foundations and implications. In a sense, all the combinatorial approaches are based on applicative structures, where a term $f(x)$ is represented by the function f such that $f(x) = \mathsf{f}(\mathsf{x})$, so variables are implicit. The relational approach allows us to go a step further by *explicity* handling variables at the object level by means of projections. In fact, the relational term P_i can be effectively identified with the ith variable. Because of this, some primitives that are difficult to define in other frameworks, such as the term-destructor function $f^i : \mathcal{T}_\Sigma \to (\mathcal{T}_{\Sigma 1} \times \cdots \times \mathcal{T}_{\Sigma i})$, are easy to define in ours.

Comparison with cylindric algebras. Tarski and Givant [Tarski and Givant, 1987] discuss in depth the benefits of variable elimination via binary relation algebras, including their relation to cylindric algebras. (It is interesting to note that Nemeti deems both approaches complementary in his review of Tarski and Givant's book [Németi, 1990]). In our context, binary relation algebras provide a mathematically rigorous notion of compilation in which target code retains the declarative content of the source code and is extensible to constraints and many other logic programming formalisms. The use of binary relations, as compared to other relational formalisms in general and cylindric algebras in particular, greatly simplifies the rewrite rules. First, there is no need for indexed joins, i.e., relations obtained by "composing" two n-ary relations with respect to a particular component. Second, functions can be interpreted "as themselves", i.e., as single-valued relations.

Variable elimination and abstract syntax. Other efforts to eliminate variables in logic programming include the n-ary relation based combinatory work of [Hamfelt and Nilsson, 1998] and [Bellia and Occhiuto, 1993] where a new polynomial algebra is developed for the purpose of eliminating variables. These results can be viewed as a special case of the research in abstract syntax aimed at axiomatizing the treatment of renaming, freshness, instantiation, and quantification as a mathematical theory in its own right. We refer the reader to [Gabbay et al., 2007, Gabbay and Pitts, 1999, Cheney and Urban,

2004] and to [Cheney, 2004], the latter an excellent and detailed description of the field, its aims and its history. Finally, category theory itself can be viewed as a device to eliminate variables from mathematics. Categorical tools have been increasingly employed to formalize abstract syntax (see [Fiore et al., 1999, Gabbay and Pitts, 1999, Freyd and Scedrov, 1991, Finkelstein et al., 2003, Amato et al., 2009]).
Tabular allegories. Freyd shows in [Freyd and Scedrov, 1991] that there is a natural canonical extension of a category to a relations-enriched category (an allegory) in which the original arrows are recoverable as functional relations. Our work shares a special relationship with Freyd's tabular allegories. An allegory is a category whose $HomSets$ are enriched with semi-lattice structure[Freyd and Scedrov, 1991]. An allegory is tabular iff every relation $R : A \to B$ has a tabulation $C, f : C \to A, g : C \to B$ such that $R = f^\circ g$ and $(x; f = y; f) \wedge (x; g = y; g) \Rightarrow x = y$ (; denotes diagrammatic composition).

This informally means that we have an object C such that it uniquely determines all the pairs of elements belonging to the relation. As an example, fix the set $A = \{1, 2, 3\}$ and $B = \{t, f\}$, then the relation $R : B \to B = \{(t, u), (t, v), (v, t)\}$ is tabulated by $f(1) = t, f(2) = t, f(3) = v, g(1) = u, g(2) = v, g(3) = t$.

In our encoding, the left tabulation corresponds to the term structure and the right tabulation captures *equations among variables*. Instantiation of a variable is a change of the domain of the tabulations. As an example imagine the relational term $K(m(x_1, x_2, x_3)) \cap K(m(x_1, x_1, a))$. Then the domain C of the tabulations is isomorphic to the Herbrand domain and the tabulations f, g would be defined (assume C is $\mathcal{H}_\Sigma$):

$$\begin{aligned} f(x) &= m(x, x, a) \\ g(x) &= \langle x, x, a \rangle \end{aligned}$$

In this setting, we can interpret our algorithm as a check for disjoint images of the tabulations.

7.6. Conclusions and Future Work

We have presented an algorithm for first-order unification using rewriting in a variable-free relation algebra. The simple systems $\to_{\mathcal{L}}$ and $\to_{\mathcal{R}}$ suffice to decide unification for occur-check-free pairs of terms. Function Δ and the split-rewriting system are introduced to deal with occur-check at the expense of losing simplicity. We can regain simplicity by using a more powerful notion of rewriting and matching, obtaining as a result an efficient algorithm largely in the spirit of [Martelli and Montanari, 1982]. Furthermore, a dual right-factoring version of the algorithm exists where constraints are accumulated over f_i^n terms and factorization happens for P_i° terms, using $FP_i^\circ \cap GP_i^\circ = (F \cap G)P_i^\circ$. We plan to relate this kind of duality with other duality/symmetry notions such as deep inference [Guglielmi, 2007].
Substitution. In the relational world we have no variables and no substitution notion. This raises an interesting paradox: the heart of unification is the computation of a substitution acting on variables. How can we unify in such a setting? The answer is that substitution gets replaced by constraint propagation and occur-check translates to functorial compatibility in such a way that *pure* (conditional) rewriting is enough!
Abstract syntax and algebraic approach. Another advantage of performing unification in the relational setting is that variable elimination is one way of formalizing abstract syntax, effectively giving syntax, syntactic operations (renaming, substitution, etc.), and syntactic machinery (contexts, multi-equations, etc.) a rigorous axiomatization by coding them out of existence into a particular formalism and making them as declarative as the object language. Even properties are captured algebraically, as mentioned in the introduction. For example, invariance under substitution of ground terms is reflected in our framework by $K(t)^\circ K(t) = \mathbf{1}$ (the equation is true iff t is ground) and equality's congruence with respect to term forming ($f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)$ iff $t_1 = u_1 \wedge \cdots \wedge t_n = u_n$) is captured in **DRA** as $FS \cap FR = F(S \cap R)$, provided F is a mapping.

Extending the framework. Our framework can be seamlessly extended in order to formalize and decide other notions of unification. For instance, a common unification pattern found in logic programming is unification among *renamed apart* terms. Consider a restriction operator ν such that $\nu x_1.t = x_1$ is equivalent to $t\sigma = x_1$, where σ is a renaming apart of x_1, and $\nu x_1.t = x_1$ is equivalent to $t = x_1$ iff x_1 does not occur in t.

This variable-restriction concept can be faithfully represented in our framework using the partial projection relation Q_i, which relates an open sequence with the ones where the ith element may be any term. (In [Lipton and Chapman, 1998] Q_i is understood as an existential quantifier.) Our framework is modified as follows. First, extend K with case $K(\nu x_i.t) = K(t)Q_i$. In words, the ith position of $K(\nu x_i.t)$'s codomain is free, whereas for $K(t)$ it contains the set of possible groundings for x_i. The new definition of K extends $\mathbf{U}$, so a new decision procedure is needed. It is defined in modular fashion by adding a new rewriting subsystem for Q_i elimination. Some rules of this system are $P_jQ_i \rightarrow P_j$ and $(f_i^n P_j \cap R)Q_i \rightarrow f_i^n P_j Q_i \cap RQ_i$.

Performing unification modulo additional theories is possible in our framework and we think it should not be difficult to derive the corresponding algorithm. As supporting evidence, see [Lipton and Chapman, 1998] on how to represent disunification problems with relations.

Categorical interpretation. Defining the full set of rules for the system in the preceding paragraph will result in a too complex rewriting system. We believe such a complexity is not inherent to our approach, but to the formalism used, in this case rewriting.

We could modify our rewriting procedure — adding more *structure* awareness to it — in order to avoid most of the cumbersome rewritings, like the splitting/factoring, where terms are split to be joined again. However we believe the algorithm's true nature will be revealed in a categorical interpretation.

A deeper algebraic study of the algorithm reveals some interesting properties. For instance, a "functorial" variable directly maps to an arrow in a category. In fact, normal forms are just a representation of the tabulation for the intersection of two relations.

Another example of a categorical construction is the "functorial compatibility" test, $F^\circ G$, which is just a tabulation for the compatibility relation.

Establishing a categorical semantics where the current algorithm can be compared with existing pullback construction algorithms is performed in Chapter 9.

CHAPTER 8

Category Theory

Category Theory is an area of mathematics that examines in an abstract whay the properties of particular mathematical concepts, formalizing them as collections of *arrows* or *morphisms* and *objects*, with these collections satisfying some basic conditions.

Many areas of mathematics can be formalized by category theory, but in our humble opinion, Category Theory is a natural framework for Constructive Mathematics and Computer Science. Category Theory was developed in the 40s by Saunders Mac Lane and Samuel Eilenberg, and thus should be considered a young area of mathematics. In Computer Science, category theory has brought significant advances and improvement in both theoretical and practical fields.

A remarkable feature of Category Theory is its ability to represent different areas of mathematics using a unified framework, and revealing previously unknown connections. In Andrej Bauer words:

> Developments in categorical logic reveal important and deep interaction between logic and category theory (or algebra, if you will), especially sheaf theory and topos theory. For example, they elucidate the relationship between Cohen forcing and Grothendieck's constructions which originated from algebraic geometry, if I am not mistaken, certainly an amazing connection.
>
> Recent developments in type theory reveal important and deep intereaction between type theory and homotopy theory. It is perhaps too early to pass a definitive judgment, but the connections certainly look promising.

This chapter doesn't intend to be a comprehensive introduction to Category Theory, for that the reader may consult standard textbooks, some of them are [Mac Lane, 1998, Barr and Wells, 1999, Jacobs, 1999, Pitts, 2000, Borceux, 1994, Freyd and Scedrov, 1991, Lambek and Scott, 1986, Makkai and Reyes, 1977]. The purpose of the chapter is to present the definitions and notions used in Chapters 9 and 10 for defining the categorical machine for logic programs.

8.1. Basic concepts

A category $\mathcal{C} = \langle \mathcal{O}, \mathcal{A} \rangle$ consist on a collection of objects $\mathcal{O}$ and arrows $\mathcal{A}$. For every object $A \in \mathcal{O}$, an identity arrow exists, $id_A \in \mathcal{A}$. Given arrows $f : A \to B$ and $g : B \to C$, its composition $f;g : A \to C$ is defined. Diagrammatically:

$$A \xrightarrow{f} B \xrightarrow{g} C$$

$$A \xrightarrow{f;g} C$$

We use the so-called *diagramatic* order of composition, where an arrow the composition of an arrow $f : A \to B$ with $g : B \to C$ is written as $f;g$. Note that most category theory texts employ what we call the functional order of composition, in which this same composition would be written $g \circ f$. As with relation, we omit the ; symbol when there is no confusion and write fg for $f;g$.

The composition operation obeys the laws of associativity and identity, that is, for $f : A \to B$, $g : B \to C$ and $h : C \to D$, we have $(f;g);h = f;(g;h)$ and $id_A;f = f = f;id_B$.

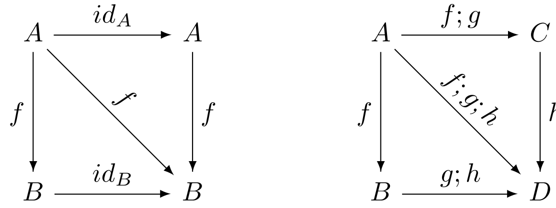


FIGURE 8.1. Basic Commutative Diagrams

Often arrow equality is expressed by means of *commutative diagrams*. A diagram is commutative iff for any path in the diagram with the same origin and target, the composition of the arrows of both paths is equal. The basic laws of a category are shown in diagrammatic style in Fig. 8.1. As the reader we may have noticed, the basic laws of a category are an extension of the associativity and identity laws of a monoid. Indeed, a monoid is category with a unique object, and arrows as elements.

Another example of a category is a partially ordered set. Objects are elements of the set, and an arrow from A to B exists iff $A \leq B$.

Often, categories allow different representations for the same mathematical concept. For instance, the natural numbers may be seen as a category with one object and arrows the numbers, then composition is may be the addition operation. Or we may model them as a partially ordered set.

An important category is Set, the category of sets and functions between them. In this category, objects are set and arrows are set-theoretic functions, that is to say, a function f from A to B is a subset of the cartesian product $A \times B$, such that $\forall x \in A.\exists! y.(x, y) \in f$.

Definition 8.1 (Monic Arrow)**.** *An arrow f is monic if for all morphisms g, h, $g; f = h; f$ implies that $g = h$.*

In some categories, this corresponds to the notion of "injectivity". We denote monics with an special purpose arrow symbol:

$$A \overset{f}{\rightarrowtail} B$$

Correspondingly, we say an arrow is epic if for all morphisms g, h, $f; g = f; h$ implies that $g = h$.

Indeed, the monic/epic concept is a good example of what we call "duality". From a category $\mathcal{C}$, we may form its dual category $\mathcal{C}^\circ$ by reversing all the arrows and keeping the same objects. Thus, monics in $\mathcal{C}$ are epics in $\mathcal{C}^\circ$.

Definition 8.2 (Isomorphism)**.** *An arrow $f : A \to B$ is an isomorphism iff exists an arrow $f^{-1} : B \to A$ such that $f; f^{-1} = id_A$.*

Definition 8.3 (Subobject)**.** *For an object A, a subobject is an isomorphism-closed class of monic arrows.*

In some categories, the subobject definition corresponds to the notion of subset of A. It should be noted that in category theory, equality is usually defined up to isomorphism. However, in computer science, that approach is not so convenient, and we require "canonical" representatives of most constructions, including subobjects.

8.2. Functors

A functor is an homomorphism of categories, that is to say, a mapping that respects the categorical structure.

Definition 8.4. *Given categories $\mathcal{C} = \{\mathcal{O}, \mathcal{A}\}$, $\mathcal{D} = \{\mathcal{O}', \mathcal{A}'\}$, a functor $F : \mathcal{C} \to \mathcal{D}$ is composed by an object map $F_{\mathcal{O}} : \mathcal{O} \to \mathcal{O}'$ and an arrow map $F_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}'$ such that $F_{\mathcal{A}}(id_A) = id_{F_{\mathcal{O}}(A)}$ and $F_{\mathcal{A}}(f; g) = F(f); F(g)$.*

It is easy to check that there is an identity functor and that functor composition is associative. In Lawvere's thesis [Lawvere, 1968], the category Cat where objects are categories and arrows are functors is defined. To avoid size problems, we assume that categories are *small*, that is to say, their collection of arrows and objects are actually sets.

Can we take this generalization further and build a category where objects are functors and arrows are transformations between them? What is a morphism of functors?

Definition 8.5 (Natural Transformation)**.** *Given functors $F, D : \mathcal{C} \to \mathcal{D}$, a natural transformation η_X an object-indexed family of arrows $\eta_A : F(A) \to G(A)$ such that the following diagram commutes for all arrows $f : A \to B$:*

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \downarrow{\scriptstyle \eta_A} & & \downarrow{\scriptstyle \eta_B} \\ G(A) & \xrightarrow{G(f)} & G(B) \end{array}$$

Thus, the functor category $Fun(\mathcal{C}, \mathcal{D})$ has as objects functors from $\mathcal{C}$ to $\mathcal{D}$ and as arrows the natural transformations.

May commonly used categories are functor categories, of particular importance are the hom-functors. Assuming $\mathcal{C}$ a small category, there exists a hom-set $Hom(A, B)$ which is the set of morphisms from A to B. Then, the hom-functor $Hom(A, -) : \mathcal{C} \to Set$ maps each object X in $\mathcal{C}$ to the set $Hom(A, X)$ and each arrow $f : X \to Y$ to the function $Hom(A, f) : Hom(A, X) \to Hom(A, Y)$ given by mapping each arrow $g : A \to X$ in $Hom(A, X)$ to $g; f : A \to Y$. By a symmetric argument, a functor $Hom(-, A)$ exists.

The Yoneda Lemma states that any small category $\mathcal{C}$ can be embedded in the category of contravariant functors $Fun(\mathcal{C}^{\circ}, Set)$ from $\mathcal{C}$ to Set via a functor $h : \mathcal{C} \to Fun(\mathcal{C}^{\circ}, Set)$. h is called the *Yoneda Embedding*.

8.3. Diagrams

In this section, we follow [Barr and Wells, 1999] conventions and definitions.

Definition 8.6 (Graph)**.** *A graph $\mathcal{G} = \langle \mathcal{G}_V, \mathcal{G}_E \rangle$ is composed of a set $\mathcal{G}_V$ of nodes and a set $\mathcal{G}_E = (\mathcal{G}_V \times \mathcal{G}_V)$ of edges.*

Definition 8.7 (Graph Homomorphism)**.** *A graph homomorphism $F : \mathcal{G}_V \to \mathcal{J}_V$ is a maps such that if $(u, v) \in \mathcal{G}_E$, then $(F(u), F(v)) \in \mathcal{J}_E$.*

Definition 8.8 (Diagram)**.** *Let $\mathcal{G}$ and $\mathcal{J}$ be graphs. A diagram in $\mathcal{G}$ of shape $\mathcal{J}$ is a homomorphism $D : \mathcal{J} \to \mathcal{G}$ of maps. We call $graj$ the shape graph of diagram D.*

Remark 8.9. *We mostly follow [4.1.14 Barr and Wells, 1999] in our conventions and definitions for diagrams.*

There exist several notions of model for a diagram. We use the notion of the model of a graph in a category.

Definition 8.10 (Free Category from a Graph)**.** *Let $\mathcal{G}$ be a graph. Then, its free generated category $F(\mathcal{G}) = \{\mathcal{O}, \mathcal{A}\}$ is built as follows: Let $\mathcal{O} = \mathcal{G}_V$, then, name each edge of the graph. The set $\mathcal{A}_E^0$ is*

generated as follows: add an arrow id_A for each object in $\mathcal{O}$, add every vertex $f : u \to v$. Then, $\mathcal{A}_E^{n+1}$ is generated from all arrows $f : u \to v$ and $g : v \to w$ by adding $f; g$. $\mathcal{A}_E$ is the limit of $\mathcal{A}_E^n$, and $\mathcal{A}$ is the quotient of $\mathcal{A}_E$ by the following equations: $id_A; f = f$, $f; id_A = f$, $f; (g; h) = (f; g); h$.

Definition 8.11 (Model of a Graph)**.** *Given a graph $\mathcal{G}$, a model of $\mathcal{G}$ in a category $\mathcal{C}$ is a functor from $F(A)$ to $\mathcal{C}$.*

When a model is an embedding, the may see a diagram as a subcategory of the original one.

8.4. Limits

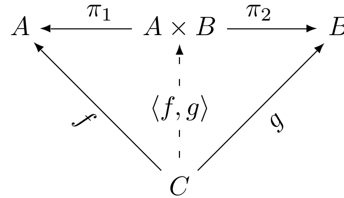
Limits are usually defined in terms of diagrams and cones, but we won't describe them in those terms, presenting the concrete limits used in this work.

Definition 8.12 (Initial Object)**.** *We say an object A is initial, if for any other object B, there is a unique arrow $? : A \to B$.*

Definition 8.13 (Terminal Object)**.** *We say an object A is terminal, if for any other object B, there is a unique arrow $! : B \to A$.*

The above definitions imply that if an initial or terminal object exists, it is unique up to isomorphism. Note also how they form an example of *dual* definitions.

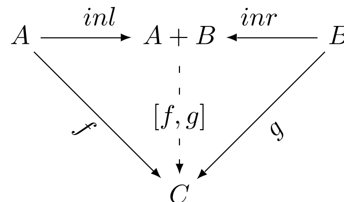
Definition 8.14 (Binary Product)**.** *A binary product of objects A, B is an object $A \times B$, together with arrows $\pi_1^{A\times B} : A \times B \to A$, $\pi_2^{A\times B} : A \times B \to B$ called projections such that for any arrows $f : C \to A$, $g : C \to B$ there exists a unique arrow $\langle f, g\rangle : C \to A \times B$ such that $\langle f, g\rangle; \pi_1 = f$ and $\langle f, g\rangle; \pi_2 = g$.*



Note that we omit the superscript from the projections as it is usually clear from the context. We may generalize the definition of product from binary to n-ary products easily, by involving more objects and requiring the existence of the corresponding projections.

The dual notion of the product is the coproduct:

Definition 8.15 (Coproduct)**.** *A coproduct of objects A and B, is an object $A + B$ with arrows inl, inr and a unique $[,]$ such that the following diagram commutes:*



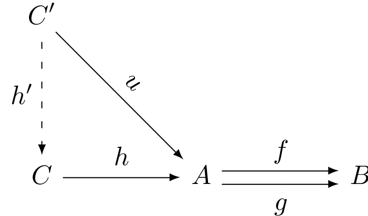
The product's definition implies that a product $A \times (B \times C)$ is isomorphic to $(A \times B) \times C$, however, in some case having a stronger equivalence is convenient:

Definition 8.16 (Strictly Associative Product)**.** *A product $(A \times B) \times C$ is strictly associative if it is* equal, *not isomorphic, to $A \times (B \times C)$, that is to say, it is the same object.*

Thanks to this definition, we may write $A \times B \times C$ for strictly associative products.

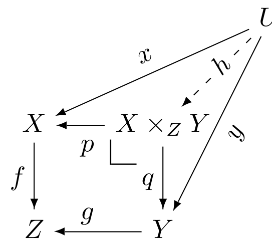
Now we introduce limits that are meant to represent in an universal way equalities among arrows. The first one is the equalizer:

Definition 8.17 (Equalizer)**.** *Let $f, g : A \to B$ be arrows in a category $\mathcal{C}$. An arrow $h : C \to A$ such that $h; f = h; g$ is said to be the* equalizer *of f,g if for all other arrows $u : C' \to A$ exists a unique $h' : C' \to C$ such that $u = h'; h$. In diagrammatic form:*



The generalization of this notion to arrows that don't share the same domain leads to the definition of pullback:

Definition 8.18 (Pullback)**.** *Given arrows f, g with a common co-domain, we say that $(p, q, X \times_Z Y)$ is a pullback if $p; f = q; g$ and for any arrows x, y such that $x; f = y; g$, exists and unique h such that $x = h; p$ and $y = h; q$. In diagrammatic form:*



For any of the notions defined above, we say that a category has it (products, coproducts, etc. . .) if for every object (or arrow) of the category, the limit exists. For instance, if we say that a category has products, then a product diagram exists for any two arbitrary objects A, B. Or if the category has equalizers, then for any arrows $f, g : A \to B$, there exist and arrow $h : C \to A$ such that it is an equalizer of f, g.

Definition 8.19 (Complete Category)**.** *Let $\mathcal{C}$ be a category. We say it is complete (or cartesian) category if it has all finite limits.*

The above definition is equivalent to stating that $\mathcal{C}$ has equalizers and all finite products, or that $\mathcal{C}$ has pullbacks and a terminal object or that $\mathcal{C}$ has equalizers, binary products, and a terminal object, as every all finite limits can be constructed from these basis limits.

8.5. Lawvere Categories

A Lawvere category is a category $\mathcal{C}$ with strictly associative products and a denumerable set $\{T^0, T^1, \ldots, T^n, \ldots\}$ of distinct objects, where each object T^n is the n-th power of the object T^1, thus the product $T^m \times T^n$ is T^{m+n}. We write i for T^i. Note that addition is associative, so this notation fits well our notion of category as $((1\times1)\times1) = (1\times2) = 3$. This means $(id_2 \times id) : 2\times1 \to 2\times1 = id_3 : 3 \to 3$, or for $f : 2 \to 2$, $(f \times id_2) = \langle f; \pi_1, f; \pi_2, id_1, id_1\rangle$, etc. . .

We write 0 for the terminal object. We write $!_A : A \to 0$ for the terminal arrow. Lawvere Categories are a natural framework for categorically representing algebraic theories. Examples of such categories $\mathcal{C}$ may be seen in [Lawvere, 1968], although we recommend the reader the explanation of [Hyland and Power, 2007]. The presentation in [Borceux, 1994] is also recommended.

Arrows from $N \to 1$ are called *operators* and arrows from $0 \to 1$ are called constants. For instance, in the Lawvere category with operators $+ : 2 \to 1$, $s : 1 \to 1$ and constant o, arrows such as $\{!_2; o, (s \times s; s); +\}; + : 2 \to 1$ exists, which is a representation of the term $o + (s(x_1) + s(s(x_2)))$.

A logic program signature Σ has a corresponding Lawvere Category $\mathcal{C}_\Sigma$.

Definition 8.20 (Lawvere Category of a Logic Program)**.** *For a given Σ, we start with the free Lawvere category, that is to say, the Lawvere category without operations and constants. Then we add arrows as follows:*

- *For every constant $a \in \mathcal{T}_\Sigma$, we freely adjoin an arrow $a : 0 \to 1$.*
- *For every function symbol $f \in \mathcal{T}_\Sigma$ with arity $\alpha(f) = N$, we freely adjoin an arrow $f : N \to 1$.*

Note that the newly adjoined arrows are monic, given that we didn't impose any equational related to them theory to the category.

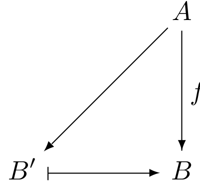
Definition 8.21 (Models for Lawvere Categories)**.** *A model of a Lawvere Category $\mathcal{C}$ is a functor $F : \mathcal{C} \to Set$ which preserves finite products and pullbacks. A homomorphism of $\mathcal{C}$-models is a natural transformation.*

The category of models $Mod(\mathcal{C}, Set)$ for $\mathcal{C}$ is the usual functor category.

8.6. Regular Categories and Relations

Several definitions exist for Regular Categories [Butz, 1998, Borceux, 1994, Johnstone, 2003, Freyd and Scedrov, 1991]; we use the latter presentation. A regular category is basically a complete category where arrows satisfy certain *exactness* conditions.

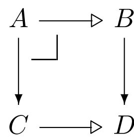
Definition 8.22 (Image)**.** *If B' is an object, we say B' allows $f : A \to B$ if f* factors through B', *i.e. if there are arrows $s : A \to B'$ and $t : B' \to B$ such that $s; t = f$. The image of $f : A \to B$, if it exists, is the smallest subobject that allows f.*



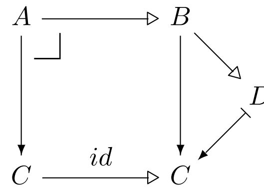
Definition 8.23 (Cover)**.** *An arrow $f : A \to B$ is a cover if its image is entire (i.e. an isomorphism) We denote covers by $A \twoheadrightarrow B$. Every cover is an epimorphism but the converse is not true. We denote covers by*

$$A \twoheadrightarrow B$$

Definition 8.24 (Regular Category)**.** *We say $\mathcal{C}$ is a Regular Category if it has products, equalizers, images and pullback transfer covers. Diagrammatically:*



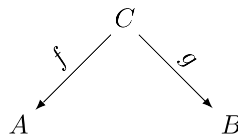
Note that the properties of covers together with the existence of images allow us to construct a cover-monic factorization of any arrow in a regular category:



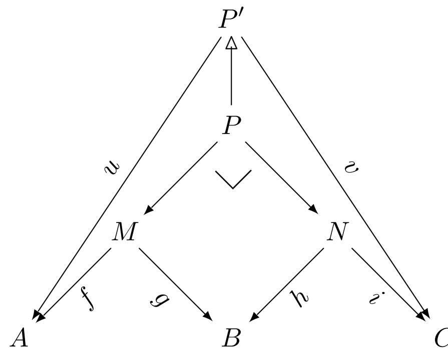
Regular Categories give rise to categories of relations.

Definition 8.25 (Monic Pair)**.** *$f : C \to A$ and $g : C \to B$ is a monic pair iff $\langle f, g\rangle : C \mapsto A \times B$ is monic.*

A monic pair (f, g) is a subobject of $A \times B$, thus, we may see it as relation from A to B:

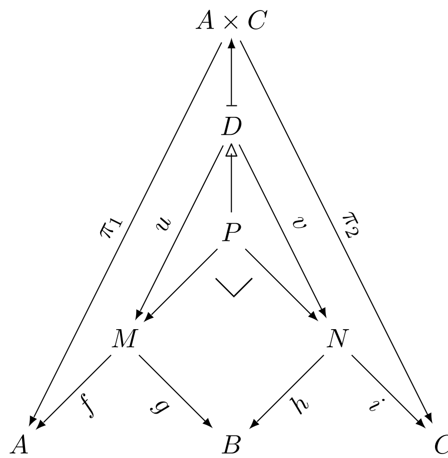


Definition 8.26 (Composition of Relations)**.** *The composition (u, v) of a relation (f, g) with (h, i) is defined by the following diagram:*

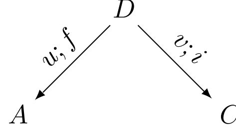


The purpose of the cover is to ensure that (u, v) remain a monic pair.

Relation composition may be defined in an alternative way, which is indeed the form we will use for our machine. In this diagram:



the composition of (f, g) with (h, i) is $(u; f, v; i)$:



Relations are not exclusive of regular categories, but their composition is well defined only if their category is regular.

Lemma 8.27. *Composition of relations is associative iff the category is regular.*

Proof. See Johnstone [Johnstone, 2003] or Freyd [Freyd and Scedrov, 1991]. Basically, pullbacks don't transfer covers, associativity fails. If associativity fails, there exists a pullback with doesn't transfer a cover. □

Definition 8.28 (Categories of Relations)**.** *For a regular category $\mathcal{C}$, the category $Rel(\mathcal{C})$ of relations has the same objects as $\mathcal{C}$, arrows $A \to B$ are monic pairs $(f : C \to A, g : C \to B)$ and composition is defined as above. $\mathcal{C}$ is a sub-category of $Rel(\mathcal{C})$. The inclusion functor sends an arrow $f : A \to B$ to the pair (id, f). If a morphism of $Rel(\mathcal{C})$ is in $\mathcal{C}$, we call it a* map.

This is not the only categorical definition for categories of relations, for instance, in [Mulry, 2002], it is proven that a category of relations is equivalent to a category of functions with a powerset monad.

Given a relation (f, g), we say (g, f) is its reciprocal. The natural order-isomorphism $Sub(A \times B) \approx Rel(A, B)$ yields a semi-lattice structure on $Rel(A, B)$. If a morphism of $Rel(\mathcal{C})$ is in $\mathcal{C}$, we call it a *map*.

8.7. Allegories and Tabular Allegories

An allegory is a enriched notion of category, such that in addition to composition, the intersection and reciprocation is defined for arrows. They provide a natural categorical framework for working with relation algebras.

Definition 8.29 (Allegory)**.** *An allegory $\mathcal{R} = \{\mathcal{O}, \mathcal{A}\}$ is an enriched category, with objects $\mathcal{O}$ and* relations *$\mathcal{A}$. We write $R; S : A \to C$ for composition of relations $R : A \to B$ and $S : B \to C$. When there is no confusion possible we may also write RS for $R; S$. We add two new operations:*

- *For every relation $R : A \to B$ and $S : A \to B$, $(R \cap S) : A \to B$ is a relation.*
- *For every relation $R : A \to B$, $R^\circ : B \to A$ is a relation.*

We write $R \subseteq S$ for $R \cap S = R$. The new operations obey the following laws:

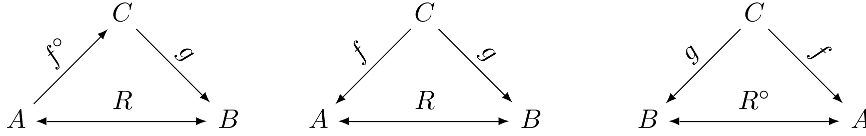
$$\begin{array}{rclrcl} R \cap R & = & R & R \cap S & = & S \cap R \\ R \cap (S \cap T) & = & (R \cap S) \cap T & R^{\circ\circ} & = & R \\ (RS)^\circ & = & S^\circ; R^\circ & (R \cap S)^\circ & = & (R^\circ \cap S^\circ) \\ R; (S \cap T) & \subseteq & (R; S \cap R; T) & (R; S \cap T) & \subseteq & (R \cap T; S^\circ); S \end{array}$$

A *map* is a relation such that $R^\circ; R \subseteq id$ and $id \subseteq R; R^\circ$. We use capital letters for relations and small letters for maps. A relation R is coreflexive iff $R \subseteq id$. For an allegory $\mathcal{R}$, we shall denote its subcategory of maps by $Map(\mathcal{R})$.

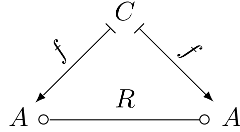
Definition 8.30 (Tabulation)**.** *A pair of maps f, g tabulates a relation R iff $f^\circ; g = R$ and $f; f^\circ \cap g; g^\circ =$ 1.*

The latter condition is equivalent to stating that f, g form a monic pair. It is easy to prove that a tabulation is unique up to isomorphism. A coreflexive relation $R \subseteq id$ is tabulated by a pair of the form

(f, f). If $R = f^\circ; g$, then $R^\circ = g^\circ; f$. Some diagrams showing tabulations for a relation R:



A coreflexive relation $R \subseteq id$ is tabulated by (f°, f):



Definition 8.31 (Tabular Allegory)**.** *An allegory $\mathcal{R}$ is a tabular allegory iff every relation has a tabulation.*

Let $\mathcal{R}$ be a tabular allegory, then $Map(\mathcal{R})$ is a regular category. The following lemma tells us that a tabular allegory is the relational extension generated by its maps and that the concepts of regular category and tabular allegory coincide:

Lemma 8.32. *If $\mathcal{R}$ is a tabular allegory then $\mathcal{R} \approx Rel(Map(\mathcal{R}))$. If $\mathcal{C}$ is a regular category then $\mathcal{C} \approx Map(Rel(\mathcal{C}))$. If $\mathcal{R} \approx Rel(\mathcal{C})$ then $Map(\mathcal{R}) \approx \mathcal{C}$.*

Proof. See [Freyd and Scedrov, 1991] 2.147 and 2.148, 2.154. □

Composition of relations in a tabular allegory is thus defined in the same way than for categories of relations arising from a Regular Category. See Def. 8.26.

The enrichment of an allegory with a $\cup$ operation, yields the concept of distributive allegory.

Definition 8.33 (Distributive Allegory)**.** *A distributive allegory is an allegory with a new relation denoted $\mathbf{0}_{AB}$ for every object A, B, and given relations R, S with the same type, $R \cup S$ is an arrow. They obey the following laws:*

$$
\begin{array}{rclrcl}
R \cup R & = & R & R \cup S & = & S \cup R \\
R \cup (S \cup T) & = & (R \cup S) \cup T & \mathbf{0} \cup S & = & S \\
R \cup (R \cap S) & = & R & R; \mathbf{0} & = & \mathbf{0} \\
R(S \cup T) & = & RS \cup RT & R \cap (S \cup T) & = & (R \cap S) \cup (R \cup T)
\end{array}
$$

Distributive tabular allegories exists, with Pre-Logoi as the category of maps. However, we won't make use of them in this work.

8.8. Related Work

Beyond standard textbooks, categorical logic [Lambek and Scott, 1986] is particularly interesting for the computer scientist. Certain categories are suitable as frameworks for constructive logic. In particular, cartesian closed categories [Lawvere, 1969b, 1968] are able to adequately interpret the simply typed lambda calculus, and C-monoids provide a model for the untyped one.

This fact has been thoroughly used to give semantics to programming languages, and indeed, very interesting transfer of categorical concepts to programming language practice has happened, like monads [Wadler, 1995, Moggi, 1991].

The main purpose of Lawvere Categories is to provide a categorical version of algebraic theories. In computer science, they have been used as an alternative to monads [Hyland and Power, 2007, Hyland et al., 2006] and to build categorical semantics based on indexed categories over them [Amato et al., 2009, Amato and Lipton, 2001, Kinoshita and Power, 1996].

CHAPTER 9

Allegorical Semantics for Logic Programming

In this chapter we explore the correspondence of constraint logic programs with Σ-allegories. In a Σ-allegory, predicates are arrows, and the equational theory of the allegory supports a notion of categorical machine. In order to carry out resolution using the machine, we develop an algorithm for arrow (relation) composition.

The resulting system bears a strong resemblance with traditional abstract machines for logic programming such as the WAM. Indeed, in the categorical machine, types can be understood as memory cells and projections as pointers. We formalize the machine and briefly discuss some issues that an implementer may face.

9.1. Introduction

In Chapter 6, we developed operational and denotational semantics for constraint logic programming using the theory **QRA** of distributive relation algebra with a quasi-projection operator. The presented semantics have very pleasant properties, but individual relations are assumed to range over the same domain, forcing our models to be pairs of elements of the union of the set of all finite sequences of terms generated by the signature of the program.

While this semantic choice allows for a simpler presentation, the useful part of the interpretation for a particular query or predicate is a fixed-length sequence $\mathcal{T}_{\Sigma}^{n} \subseteq A^{\dagger}$ where n is their arity. For instance, $[\![\overline{add}]\!]$ "contains" an infinite number of sequences of length greater or equal than 3, but in all cases, the meaningful information about the predicate is stored in the three first elements.

Developing an efficient rewriting system for the execution of logic programs becomes difficult with this approach. The representation doesn't directly capture the bounds[1] on the useful part of a sequence so, when a predicate call happens, we must split the constraint store in two parts — the one belonging to the caller environment and the one needed by the called predicate — and merge back the results at return time. Propagating constraint operations that happened inside a procedure call to the outer context is delayed.

We remedy this shortcoming by using *typed* relations. The theory of allegories [Freyd and Scedrov, 1991], provides a categorical setting for distributive relation algebras. In this setting, relations are typed and types can be interpreted as a fixed-length sequence of terms. hd and tl relations are replaced by the categorical product, capturing in an adequate way the shared context required to have an efficient execution model.

The use of *tabular relations* is key on our work. We say a relation $R : A \leftrightarrow B$ is tabulated by a monic arrow $f : C \to A \times B$ if every pair of the relation is in its image. We may split f into its components $f;\pi_1 : C \to A$ and $f;\pi_2 : C \to B$, and state that the pair $(f;\pi_1, f;\pi_2)$ tabulates R. Such a concept is fundamental for two reasons: composition — and intersection — of tabular relations is defined in terms of their tabulations and the types of the tabulations have an important meaning in the proposed implementation.

[1] A given relational term can be analyzed in order to obtain such bounds, but note that this is not practical in execution, as the analysis would have to be performed in every predicate call.

Together with allegories, we make extensive use of the so-called Lawvere Categories. aWork in [Amato and Lipton, 2001, Amato et al., 2009, Finkelstein et al., 2003, Corradini and Asperti, 1992, Asperti and Martini, 1989, Kinoshita and Power, 1996] shows that Lawvere Categories, when used as a base for an indexed category form an adequate model for Logic Programming plus common extensions, such as constraints. However, the above semantics didn't lead to an implementation effort, as the level of abstraction imposed by the categorical theory is quite high.

A very important concept in Lawvere Categories is the notion of *strictly associative product*. Given types A, B (or *objects* in categorical language), we write $A \times B$ for their cartesian product. As usual $A \times (B \times C)$ is isomorphic to $(A \times B) \times C$. We say our products are *strictly associative* if the isomorphism is an equality. That is, $(A \times B) \times C = A \times (B \times C)$. We are thus allowed to write $A \times B \times C$. This reveals to be a crucial fact for relating the categorical semantics with the an actual implementation, since if we interpret a chosen type H as a memory cell, then a memory region of size n may be interpreted as H^n iff our products are strictly associative.

The idea is to use a Lawvere Category $\mathcal{C}$ as the category of maps for a tabular allegory. Thus, types in the allegory will correspond to types in $\mathcal{C}$ and arrows in the allegory are all the relations tabulated by the arrows of $\mathcal{C}$. That way, we can choose to define an Allegory $\mathcal{R}$ or a Regular Lawvere Category $\mathcal{C}$. Once one is defined, we may obtain an equivalent structure as follows: $Rel(\mathcal{C}) \approx \mathcal{R}$ and $Map(\mathcal{R}) \approx \mathcal{C}$. The resulting Allegory or Regular Lawvere Category are a satisfactory model for the conjunctive fragment of Logic Programming.

This construction is not straightforward. The first problem we face is that Lawvere Categories are not — in general — complete. They lack equalizers, and indeed this is considered a *feature*. We fix this problem adjoining an initial object to our Lawvere Categories. This simple addition makes the category regular.

Tabular allegories directly generated from a Regular Lawvere Category $\mathcal{C}$ cannot support logic programming in full, as they are not distributive, that is to say, the lack union arrows $R \cup S$.

A distributive allegory is tabulated by a Pre-Logos, however, extending a Lawvere Category to become a Pre-Logos diminishes some of the advantages we enjoy, in particular, arrows cannot be interpreted as functions between terms anymore.

In order to remedy this we define the target of our translation to be special distributive allegories called Σ-allegories. A Σ-allegory is constructed as the distributive completion of a tabular allegory $\mathcal{R}$, where $Map(\mathcal{C})$ is a Regular Lawvere Category. Note that in a Σ-allegory, arrows that don't enjoy a union-free representation may lack a tabulation. This is not a problem, as we will handle disjunction within the algebraic theory of allegory itself, without any mention of tabulations. Thus, Σ-allegories replace most of the ad hoc theory used in logic programming, with a well known theory, category theory.

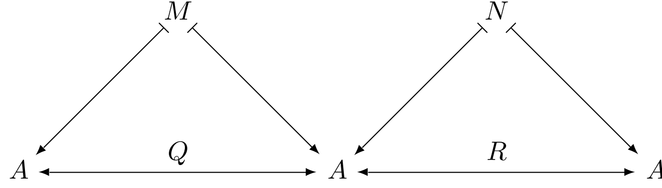
A key difference with respect to Chapter 6, is that a clause is translated to an arrow which is defined in terms of composition of tabular relations. Intersection is not used.

The composition of tabular relations is fully characterized by the pullback of its tabulations. In our translation, relation composition models unification, parameter passing, renaming apart, allocation of new temporary variables and garbage collection.

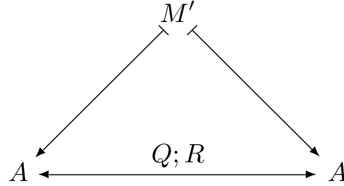
The fact that pullbacks model all that primitives present in logic programming is not new, but the recasting of the pullback concept as composition in a tabular allegory and the use for logic programming semantics is, as far as we can tell, new. Indeed, it opens up the door to new semantic constructions previously unknown. The fact that our models are composition based allows the use of typical semantic constructions from functional and functional logic programming.

Queries to the program are also constructed using relation composition. For execution, we want to follow our previous approach of "executable semantics". In the category theory setting, diagram transformation is an adequate device to define a computation notion. A query Q against a clause R is

represented categorically as the composition of two tabular relations Q and R:



then, our machine must find a normal form of the diagram such that:



Given that composition of tabular relations is defined in terms of pullbacks, the core of our logic program machine is a categorical-inspired algorithm for the construction of a pullback diagram.

An implementation of such algorithm should yield an efficient interpreter for Prolog. By efficient, we mean that the implementation can achieve the same $O(-)$ complexity as machines like the WAM [Warren, 1983] without sacrificing its declarative nature.

Diagrams accurately capture sharing, but we may go further and interpret categorical projections as pointers. Strictly associative products, their projections and their associated equational theory form a good model of the notion of memory cells, pointers and variable substitution and de-referencing.

If we interpret the categorical diagrams arising at the execution of a given Prolog query we find surprising discoveries. The domain of a tabulation represents the number of distinct “logic variables” used by the relation or program fragment, whereas the co-domain represents the “local” storage or number or terms in use for the machine in that particular moment. Going even further, the co-domain almost exactly matches the notion of “register” used in a traditional Warren Abstract Machine! ([Warren, 1983, Ait-Kaci, 1991].

Substitution and de-referencing is formalized by the equations $\langle f, g\rangle; \langle h, i\rangle = \langle f; h, g; i\rangle$ and $\langle f_1, f_2\rangle; \pi_i = f_i$. This corresponds to the traditional pointer-based representation for terms. A projection π_i is a pointer to a cell in the heap, arrow normalization using the product’s equations is de-referencing and composition is the replacement of the pointed cell contents.

We hope that all of the above notions will allow us to develop a "cell-based" instruction set for the implementation of the pullback algorithm, and thus, an instruction set for Prolog.

Structure Of The Chapter. In Sec. 9.2 we define Regular Lawvere Categories. Their associated distributive allegories, Σ-allegories are presented in Sec. 9.3. Pullbacks are a fundamental tool for our purpose, we discuss their existence and semantics in Sec. 9.4.

In Sec. 9.5 we introduce a notion of categorical computation. The algorithm for pullback calculation, which will form the core of the machine is presented separately in Sec. 9.6.

Sec. 9.7 presents the compilation procedure of a logic program, which amounts to translating predicates to arrows in a Σ-allegory. Sec. 9.8 presents the algebraic specification of the categorical machine. Some discussion about implementation is carried out in Sec. 9.9.

Finally, we discuss related work in Sec. 9.10, and detail future work and conclusions in Sec. 9.11.

9.2. Regular Lawvere Categories

We define a completion procedure for a Lawvere Category to make it regular. As seen in Sec. 8.7, regular categories are in correspondence to tabular allegories.

Definition 9.1 (Regular Lawvere Category (RLC))**.** *Given a Lawvere Category $\mathcal{C}$, we build its regular completion $\hat{\mathcal{C}}$ by adjoining an initial object $\bot$, the corresponding initial arrows $?_A : \bot \to A$ for every object A and applying the quotient $?_A; f = ?_B$ for any arrow $f : A \to B$.*

It is obvious that this procedure equates arrows that were not equal in $\mathcal{C}$.

Remark 9.2. *Adjoining objects and arrows to an existing category is logical procedure involving the graph of a category. The reader may consult[I.4 Lambek and Scott, 1986] for a rigorous definition and commentary.*

In $\hat{\mathcal{C}}$, every pair of arrows has an equalizer, so asking for the *existence* of an equalizer makes no sense anymore, as arrows not having an equalizer in $\mathcal{C}$ are equalized by $\bot$ in $\hat{\mathcal{C}}$. Indeed, the new question is, "what is the domain of the equalizer?" The domain of the arrows will play a fundamental role in proof search.

Lemma 9.3. *In a Regular Lawvere Category, every arrow has an image.*

Proof. By induction on arrows:

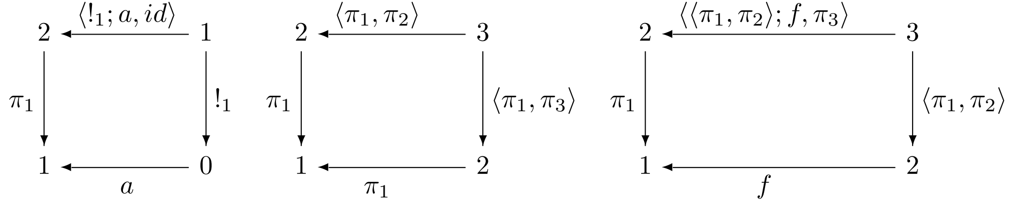
- For a initial $?_A : \bot \to A$, its image is itself, as there's no arrow with codomain $\bot$, thus $?_A$ is a monic.
- For a terminal arrow $!_A : A \to 0$, its image is id.
- Constants and operators are monics, thus they are their own image.
- The image of projections π_i is id.
- For an arrow $\langle f, g\rangle$, if it is a monic, then its image is itself, otherwise the image it is built from the images of f and g.

□

Lemma 9.4. *In a Regular Lawvere Category, pullbacks transfer covers.*

Proof. The only covers in a RLC are the projections, its associated arrows such as $\langle \pi_3, \pi_2\rangle : 3 \to 2$ and the terminal arrows.

A single projection may be pulled back against three kind of arrows:



In all cases, the pullbacks transfer covers. For arrows of domain $M > 1$, they are covers iff they are of the form $\langle \pi_{i_1}, \dots, \pi_{i_M}\rangle$, and it is easy to check by induction that they transfer covers. □

Lemma 9.5. *The regular completion of a Lawvere Category is indeed a Regular Category.*

Proof. Every pair of arrows has an equalizer, thanks to the existence of the initial object. Every arrow has an image as can be checked inductively over the set of arrows. Pullbacks transfer covers. □

Models for Regular Lawvere Categories are models of Lawvere Categories in which the initial arrows are mapped to initial arrows in Set. The most interesting model is the initial one, which we use for our logic programming semantics:

Definition 9.6 (Initial Model)**.** *Given a choice $\langle , \rangle$ of product in Set, and a choice of symbols for the signature Σ generating the Regular Lawvere Category $\mathcal{C}$ and set $\mathcal{T}_\Sigma$, the initial model of a Regular*

Lawvere Category $\mathcal{C}$ — that is to say, the initial object in $Mod(\mathcal{C}, Set)$ — is the functor $[\![]\!]$, with object and arrow components $([\![]\!]_O, [\![]\!]_A)$:

$$[\![\bot]\!]_O = \emptyset \qquad [\![0]\!]_O = \{\bullet\} \qquad [\![N]\!]_O = \mathcal{T}_\Sigma^N \quad N > 0$$

$$\begin{array}{lcl}
[\![?_N]\!]_A & = & \emptyset \xrightarrow{\emptyset} [\![N]\!]_O \\
[\![!_N]\!]_A & = & \lambda x.\, \bullet \\
[\![c : 0 \to 1]\!]_A & = & \lambda \bullet .\, c \\
[\![f : N \to 1]\!]_A & = & \lambda(n_1, \ldots, n_N).\, f(n_1, \ldots, n_N) \\
[\![\pi_i : N \to 1]\!]_A & = & \lambda(n_1, \ldots, n_N).\, n_i \\
[\![\langle t_1, \ldots, t_N\rangle : M \to N]\!]_A & = & \lambda n.\langle [\![n]\!]_A ; [\![t_1]\!]_A, \ldots, [\![n]\!]_A ; [\![t_N]\!]_A\rangle
\end{array}$$

The proof that $[\![]\!]_A$ is initial follows from the fact that $\mathcal{T}_\Sigma$ is the initial term model of the signature.

Given a Regular Lawvere Category $\mathcal{C}$, we call $Rel(\mathcal{C})$ a pre-Σ-allegory. Such allegories are the base upon which proper Σ-allegories will be built in Sec. 9.3.

9.3. Σ-Allegories

Regular Lawvere Categories are not enough to model disjunctive clauses in logic programs, as they cannot tabulate (and thus generate) arrows arising from disjunctive clauses. Chapter 6 provides strong support for us to think that distributive allegories should be the adequate framework for logic programming. However, in order to achieve an efficient implementation we would be looking for a *tabular distributive allegory*. A distributive allegory is tabulated by a Pre-Logos [Freyd and Scedrov, 1991], which is a regular category whose subobjects form a complete lattice, not just a semi-lattice. We may extend a Regular Lawvere Category into a syntactical Pre-Logos, but the added complexity is unnecessary, as tabulations are really needed to implement unification, that is to say, the conjunctive part.

In this section, we define a middle ground between a tabular allegory and a distributive tabular allegory which we call Σ-allegory.

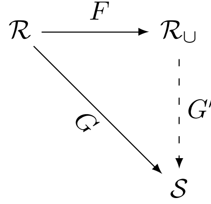
In contrast with Σ-allegories, where the union of arrows is externally characterized, the traditional categorical framework for our logical system would indeed be a *geometric category* or a Pre-Logos in Freyd's terminology.

Recall that theories involving $\wedge$, $\top$, and $\exists$ are regular theories, which can be interpreted in a regular category. If we add $\vee$ and $\bot$ to the logic, we obtain *coherent categories*. Geometric categories extend our reasoning to infinitary $\vee$, requiring the existence of all small colimits.

Definition 9.7 (Σ-Allegory)**.** *Given a Regular Lawvere Category $\mathcal{C}$, we define a Σ-allegory $\mathcal{R}_\cup$ as the distributive allegory generated from the allegory $\mathcal{R} \approx Rel(\mathcal{C})$ by freely adding all (infinitary) union arrows and taking the quotient by the distributive laws. This means that an inclusion functor $F : \mathcal{R} \to \mathcal{R}_\cup$ exists. Recall that all the arrows in $\mathcal{R}$ are tabular, thus, it is easy to see that all the arrows in $\mathcal{R}_\cup$ that possess a union-free representation are tabular.*

Note that using this definition, arrows that are the an infinite union of arrows of $\mathcal{R}$ exist. The above procedure may be logically seen as freely creating all the finite words $f_1 \cup f_2 \cup \cdots \cup f_n$ and infinite words $f_1 \cup f_2 \cup \cdots \cup f_\omega$, where f_i are arrows in $\mathcal{R}$, then taking their quotient by the distributive axioms. Indeed, the procedure yields an inclusion functor F from lr to $lr_\cup$, such that given a functor G from allegories that preserves the allegory structure, there exists a unique functor G' that preserves the allegory

structure such that:



The compromise of using Σ-allegories means that the conjunctive part of resolution is handled by pullbacks of tabulations whereas backtracking is controlled by pure allegorical laws.

The standard interpretation for a sigma allegory is a functor to Rel, the allegory of sets and relations.

Definition 9.8 (Interpretation of Σ-allegories)**.** *Let $\mathcal{R}$ be a Σ-allegory. The functor $[\![]\!]_S : \mathcal{R} \to Rel$ is defined as the identity on objects. For primitive arrows belonging to the associated RLC, we use the $[\![]\!]_A$,interpretation, then:*

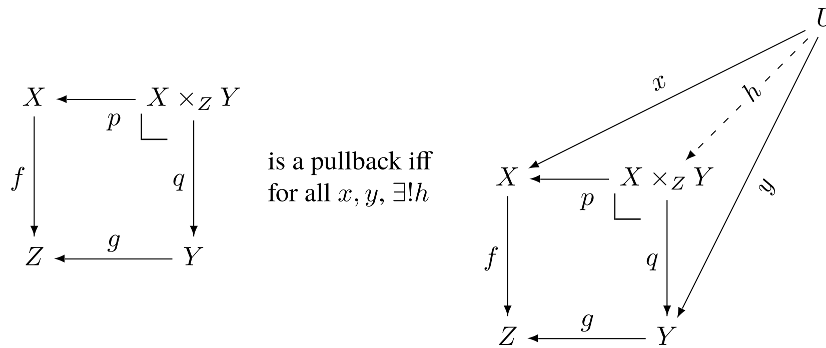
$$\begin{array}{lcl}
[\![f]\!]_S & = & [\![f]\!]_A \\
[\![R^\circ]\!]_S & = & [\![R]\!]_S^\circ \\
[\![R;S]\!]_S & = & [\![R]\!]_S;[\![S]\!]_S \\
[\![R \cap S]\!]_S & = & [\![R]\!]_S \cap [\![S]\!]_S \\
[\![R \cup S]\!]_S & = & [\![R]\!]_S \cup [\![S]\!]_S
\end{array}$$

9.4. Pullbacks in Regular Lawvere Categories

Pullbacks are the fundamental concept we will base the resolution machine on. In the next sections, we will develop a pullback calculation algorithm. Before that, we think it convenient to have a closer look at pullbacks in Regular Lawvere Categories. Understanding the inner working of pullbacks is a fundamental pre-requisite in order to understand how the algorithm and the translation works.

The objects of a RLC are the natural numbers plus a distinguished object $\bot$, with it the initial object and 0 the terminal one. The usual projections and product formers are present. Arrow are constants $c : 0 \to 1$, and operators $f, g : N \to 1$, where $N \geq 1$. Note that $\bot \times N = \bot$, and for M, N distinct from $\bot$, $M \times N = M + N$.

9.4.1. A Pullback Tutorial. We recall the pullback definition from Sec. 8.4: Given arrows f, g with a common co-domain, we say that $(p, q, X \times_Z Y)$ is a pullback if $p; f = q; g$ and for any arrows x, y such that $x; f = y; g$, exists and unique h such that $x = h; p$ and $y = h; q$. Pullbacks are defined diagrammatically as:

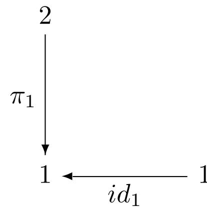


That is to say, for arrows f and g we must find a pair of arrows p, q such that the diagram commutes and are *universal* for any other commuting arrows.

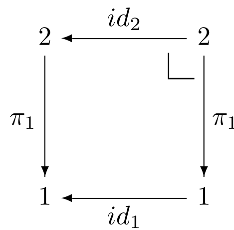
We may also interpret pullbacks in RLC as their pullbacks in Set. A pullback among two arrows in Set $f : N \to M$ and $g : N' \to M$, is set-theoretically understood as $\{f(x) = g(y) \mid x \in N \land y \in N'\}$.

Note the very important fact that x and y are disjoint variables. The pullback operation is unification between *fully renamed apart* terms! Pullbacks may be easily used to generate *fresh* variables. In an allegory setting, this corresponds to composition with the reciprocal of a projection, namely π_i°.

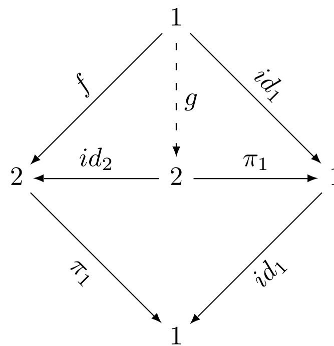
Let's present an example. What is the pullback of the following diagram?



an obvious choice for making the diagram commute would be:

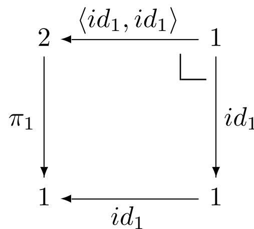


This corresponds to the set theoretic interpretation, $\{((x, y), z) \in 2 \times 1 \mid \pi_1(x, y) = z\}$, which indeed makes z redundant, as the set defined in the comprehension is isomorphic to 2. We check the universal property for the object 1:

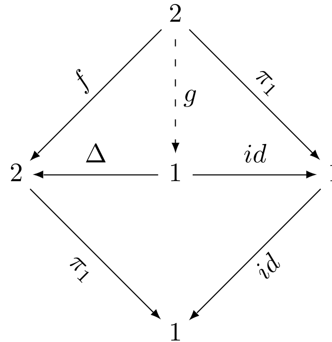


For the outer diagram to commute, it is required that $f = \langle id, h\rangle$, where $h : 1 \to 1$ is an arbitrary arrow. The inner diagram must commute, so $g = f$ is forced. The choice of id_2 guarantees that the universal property holds.

What happens if we pick a different choice for the pullback?



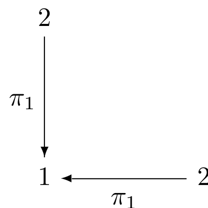
Is the above diagram a pullback? Let $\Delta = \langle id_1, id_1\rangle$ and let's check the universal property:



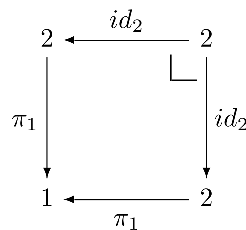
The diagram forces $g = \pi_1$, and f is the solution to $f = \pi_1; \langle id, id\rangle$, thus $f = \langle \pi_1, \pi_1\rangle$. But for the outer diagram to commute, f could be any arbitrary arrow $\langle \pi_1, _\rangle$, like $\langle \pi_1, \pi_2\rangle$, and in this case there's no solution for g. The universal property fails. Had we chosen 3 as the pullback object, is easy to see that the uniqueness property would fail.

Given that the objects in our category are the natural numbers, the following informal guidelines apply: If a candidate pullback object is too "small", the universal property will fail for the arrow h cannot be constructed. If the candidate object is too "big", the uniqueness property will fail.

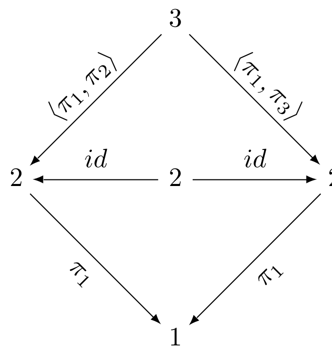
How do pullbacks model renaming apart and variable destruction? Consider the following diagram:



we may take again an obvious choice for making the diagram commute:

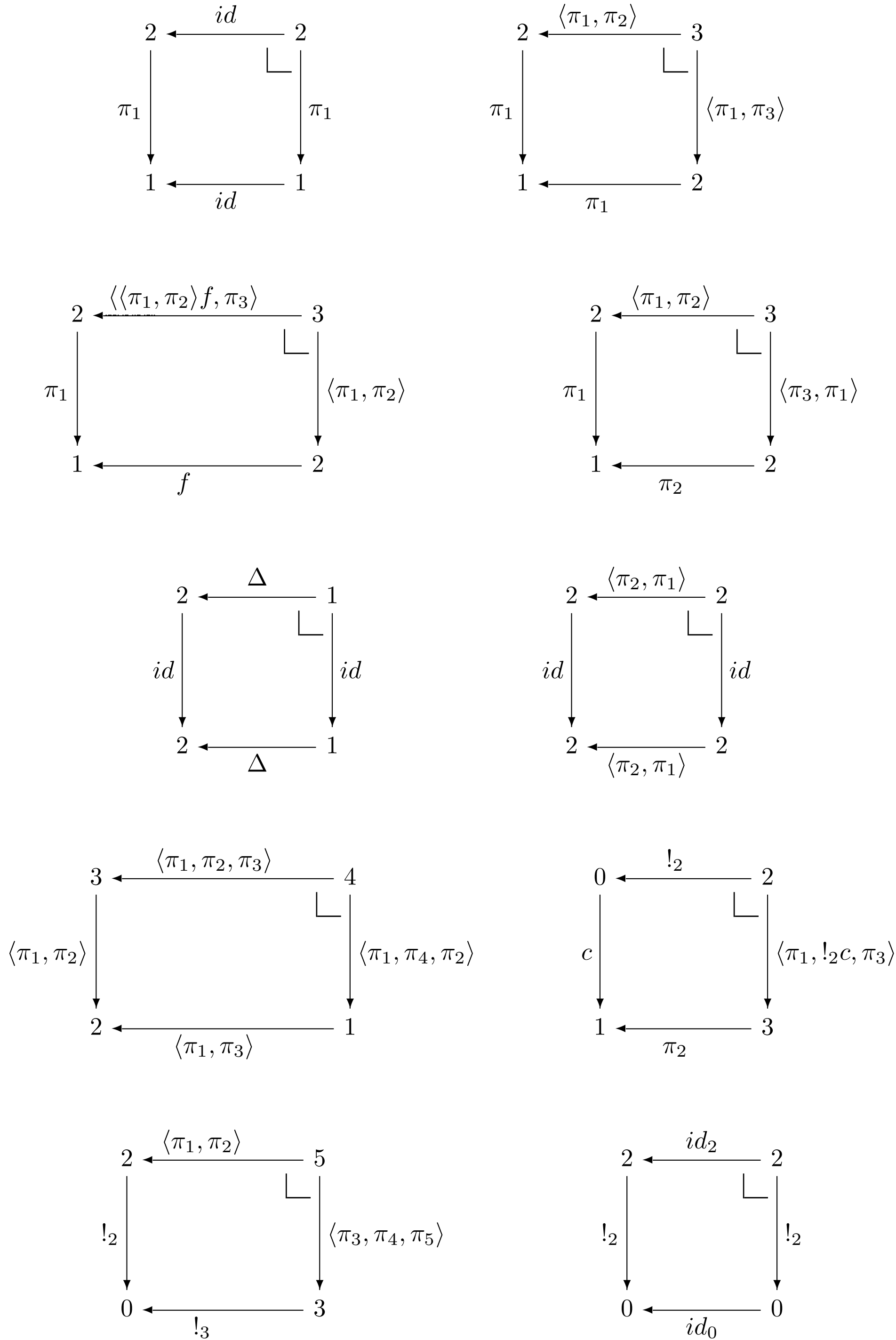


This diagram looks better than our previous failed candidate. But it is not a pullback. It fails the universal property precisely at this commuting square:



There's no unique arrow that makes the two upper triangles commute. Indeed, the correct pullback for the original diagram is the out square we have just drawn as a counterexample.

9.4.2. A Catalog of Pullbacks. For reference, we draw common pullbacks in a Regular Lawvere Category:



9.5. Computing with Diagrams

The theory of a category identifies objects that are the same, that is to say, $f; id = f$ are the same conceptual arrow. However, the representation $f; id$ and f are two different syntactic entities.

Indeed, this representation problem is present in diagrams, note that

$$A \xrightarrow[f]{} A$$

$$A \xrightarrow[f]{} A \xrightarrow[id]{} A$$

$$A \xrightarrow[id]{} A \xrightarrow[f]{} A$$

denote the same arrow $f : A \to A$ but are all three different diagrams. We want to use categories in two different ways:

- As a semantic and foundational device. Each logic program will have a an associated Σ-allegory.
- As a computational paradigm. Our computational notion is a form of diagram transformation.

Our objective is to use diagrams as a computation device in such a way that we may directly relate a computational state to its categorical counterpart, making soundness trivial to prove, using the diagram inclusion functor into a category.

Diagrams are defined in Sec. 8.3, in the rest of this section, we'll focus the computation of pullbacks using diagrams. Assume a Regular Lawvere Category $\mathcal{C}$, with operators $f : 3 \to 1$ and $g : 2 \to 1$. The diagram:

$$\begin{array}{ccc} 3 & & \\ {\scriptstyle f}\downarrow & & \\ 1 & \xleftarrow{g} & 2 \end{array}$$

has a pullback by definition, given that $\mathcal{C}$ is complete. Mathematically, it exists and we have the diagram:

$$\begin{array}{ccc} 3 & \xleftarrow{f'} & 3 \times_1 2 \\ {\scriptstyle f}\downarrow & \lrcorner & \downarrow{\scriptstyle g'} \\ 1 & \xleftarrow{g} & 2 \end{array}$$

However, we have no idea of what is the canonical representation for f', g', we have to *compute* it.

In light of Sec. 9.4, we may develop an *ad-hoc* or semantic-based algorithm. However that approach would make us to lose track of the categorical roots of the problem. The objective is to use the *categorical theory* to perform computation.

What is a *canonical representation* for arrows. Fortunately, as RLC categories are freely generated from logic programs, every arrow has a normal form induced by the categorical theory.

But we need to use that notion with care. Arrows like:

$$3 \xrightarrow{\langle g, \langle \pi_1, \pi_2 \rangle; f \rangle} 2 \xrightarrow{\langle \pi_2, \pi_1 \rangle} 2$$

have a normal form induced by the equality theory of the product:

$$3 \xrightarrow{\langle \langle \pi_1, \pi_2 \rangle; f, g \rangle} 2$$

but arrows involving the initial object like:

$$\bot \xrightarrow{\langle ?_1, ?_1 \rangle} 2 \xrightarrow{\langle \pi_1, \pi_1, \pi_2 \rangle} 3$$

the normal form is:

$$\bot \xrightarrow{\langle ?_1, ?_1, ?_1 \rangle = ?_3} 3$$

Now, imagine we want to rewrite the arrow $\langle ?_1, ?_1 \rangle$ to $\langle \pi_1, \pi_2; ?_1 \rangle : 1 \times \bot \to 2$. As $1 \times \bot = \bot$, if we fully use the categorical theory for arrow normalization, we may lose the rewriting step.

The possibility of using a countable number of meta-arrows and meta-objects was taken into consideration. But it turns out that for our purposes, using arrows is enough. The fact that we don't need to use meta-arrows is closely related to the use of ground terms in Sections 6 and 7.

Thus, the two basic steps of our computational paradigm are: arrow rewriting, and arrow normalization based on a version of underlying equational theory of the category.

A very important choice for designing a categorically-inspired algorithm using this approach is whether to use commutative or non-commutative diagrams. It is obvious that the result of a pullback calculation algorithm will be a commutative diagram, but should we start with a commutative diagram and modify it until it is a pullback, or should we start with a non-commutative diagram and rewrite arrows until the diagram is commutative, and thus a pullback?

We developed both approaches and found that using non-commutative diagrams reflects much better what is happening in the computational world. Imagine a pullback which is intended to unify two terms. In representation of terms in memory, they are not equal, but we keep instantiating variables until they become equal. This should be interpreted categorically as a non-commutative diagram and it is the approach we use for the actual pullback algorithm given presented in Sec. 9.6.

In order to illustrate the commutative approach, we briefly present the example of a pullback calculation. Start with the diagram:

$$\begin{array}{ccc} 2 & & \\ \downarrow{\scriptstyle \langle \pi_1, \pi_1 \rangle} & & \\ 2 & \xleftarrow{\langle \pi_2, \pi_1 \rangle} & 2 \end{array}$$

We reduce the computation of a pullback to the computation of an equalizer. The transformed diagram is:

$$\bot \xrightarrow{?_4} 4 \underset{\langle \pi_3, \pi_4 \rangle}{\overset{\langle \pi_1, \pi_2 \rangle}{\rightrightarrows}} 2 \underset{\langle \pi_2, \pi_1 \rangle}{\overset{\langle \pi_1, \pi_1 \rangle}{\rightrightarrows}} 2$$

Remember that $2 \times 2 = 4$, as products are strictly associative. We normalize the arrows and obtain the following diagram:

$$\bot \xrightarrow{?_4} 4 \underset{\langle \pi_4, \pi_3 \rangle}{\overset{\langle \pi_1, \pi_1 \rangle}{\rightrightarrows}} 2$$

As previously hinted, we could (and will) use the non-commutative approach, so replacing id for $?_4$ we would get a non-commutative diagram:

$$4 \xrightarrow{id_4} 4 \underset{\langle \pi_4, \pi_3 \rangle}{\overset{\langle \pi_1, \pi_1 \rangle}{\rightrightarrows}} 2$$

Using the non-commutative approach will be specified in the next section, so the previous diagram:

$$\bot \xrightarrow{?_4} 4 \underset{\langle \pi_4, \pi_3 \rangle}{\overset{\langle \pi_1, \pi_1 \rangle}{\rightrightarrows}} 2$$

is the starting diagram for our computation.. Note that $?_4 = \langle ?_1, ?_1, ?_1, ?_1 \rangle$, so a more convenient diagram for our purposes is:

$$\bot \xrightarrow{\langle ?_1, ?_1, ?_1, ?_1 \rangle} 4 \underset{\langle \pi_4, \pi_3 \rangle}{\overset{\langle \pi_1, \pi_1 \rangle}{\rightrightarrows}} 2$$

The problem is clear, our algorithm must rewrite the $?_1$ arrows in order to construct the equalizer. The first step is induced by comparison of the π_1 and π_4:

$$1 \times \bot \xrightarrow{\langle \pi_1, \pi_2; ?_1, \pi_2; ?_1, \pi_1 \rangle} 4 \underset{\langle \pi_4, \pi_3 \rangle}{\overset{\langle \pi_1, \pi_1 \rangle}{\rightrightarrows}} 2$$

the next step is induce by π_1 and π_3:

$$1 \times \bot \xrightarrow{\langle \pi_1, \pi_2; ?_1, \pi_1, \pi_1 \rangle} 4 \underset{\langle \pi_4, \pi_3 \rangle}{\overset{\langle \pi_1, \pi_1 \rangle}{\rightrightarrows}} 2$$

Now, we update the "free" positions in order to obtain the equalizer, finishing the computation:

$$2 \xrightarrow{\langle \pi_1, \pi_2, \pi_1, \pi_1 \rangle} 4 \underset{\langle \pi_4, \pi_3 \rangle}{\overset{\langle \pi_1, \pi_1 \rangle}{\rightrightarrows}} 2$$

Going back to the original diagram, we get:

$$2 \xrightarrow{\langle \pi_1, \pi_2, \pi_1, \pi_1 \rangle} 4 \underset{\langle \pi_3, \pi_4 \rangle}{\overset{\langle \pi_1, \pi_2 \rangle}{\rightrightarrows}} 2 \underset{\langle \pi_2, \pi_1 \rangle}{\overset{\langle \pi_1, \pi_1 \rangle}{\rightrightarrows}} 2$$

thus the pullback is:

$$\begin{array}{ccc} 2 & \xleftarrow{\langle \pi_1, \pi_2 \rangle} & 2 \\ {\scriptstyle \langle \pi_1, \pi_1 \rangle} \downarrow & & \downarrow {\scriptstyle \langle \pi_1, \pi_1 \rangle} \\ 2 & \xleftarrow{\langle \pi_2, \pi_1 \rangle} & 2 \end{array}$$

The reader will immediately see why the non-commutative approach is easier to understand. Here, the arrows are always equal, and we must rewrite trying to reach a maximal *ordering* without breaking equality, whereas in the non-commutative approach we just rewrite to make components of product arrows equal.

9.6. The Pullback Algorithm

We develop a pullback calculation algorithm for an Regular Lawvere Category $\mathcal{C}$ generated from an arbitrary signature Σ. We follow the approach introduced in Sec. 9.5 to develop the algorithm.

In order to improve the presentation, we reduce the pullback problem to its equivalent equalizer formulation. We start with a non-commutative diagram and rewrite it until we reach a commutative one, which will be an equalizer.

The diagram rewriting used in the algorithm doesn't change the shape of the diagram, allowing for a non-diagrammatic representation. Arrow composition captures substitution, and is implemented by normalization modulo the equational theory of the categorical product.

Definition 9.9 (Pullback Problem)**.** *A pullback problem is given by two arrows $f : N \to M$ and $g : N' \to M$. In diagrammatic form:*

$$\begin{array}{ccc} N & & \\ \downarrow f & & \\ M & \xleftarrow{g} & N' \end{array}$$

We call $N + N'$ the type *of the pullback problem.*

Definition 9.10 (Arrow Normalization)**.** *The arrow normalization relation $\to_R$ is given by the rewriting system:*

$$\begin{array}{lll} h;\langle f,g\rangle & \to_R & \langle h;f,h;g\rangle \\ \langle f,g\rangle;\pi_1 & \to_R & f \\ \langle f,g\rangle;\pi_2 & \to_R & g \\ f;!_N & \to_R & !_M \qquad f : M \to N \end{array}$$

It is easily seen that $\to_R$ is confluent and terminating. We write $\to^!_R$ for the associated normalizing relation based on $\to_R$.

We don't need rewriting cases for the initial arrow. If we need to equalize two arrows with the initial one, the pullback is determined to be the initial object and the computation stops.

Definition 9.11 (Pre-Starting Diagram)**.** *For a pullback problem, build the pre-starting diagram $\mathcal{P}$:*

$$N \times N' \underset{\pi_2;g}{\overset{\pi_1;f}{\rightrightarrows}} M$$

Given that products are strictly associative π_2 behaves as renaming operation $\pi_i \to \pi_{i+N}$ for all projections occurring in g.

Take the example of $f : 1 \to 1 = \langle \pi_1 \rangle$, $g : 2 \to 1 = \langle \langle \pi_1, \pi_2 \rangle; f \rangle$. Then $1 \times 2 = 3$, so $\pi_2 : 3 \to 2$ is equal to $\langle \pi_2, \pi_3 \rangle$, and $\pi_2; g = \langle \langle \pi_2, \pi_3 \rangle; f \rangle$.

The construction of the starting diagram involves then formally replacing the representation of $N \times N'$ by $N + N'$.

Definition 9.12 (Starting Diagram)**.** *Given $N \times N' = N + N'$, we make a choice of representation $p_1 : N + N' \to N = \langle \pi_1, \dots, \pi_N \rangle$ and $p_2 : N + N' \to N' = \langle \pi_{N+1}, \dots, \pi_{N+N'} \rangle$. Then, let $p_1; f \to^!_R f'$ and $p_2; g \to^!_R g'$, the starting diagram $\mathcal{P}$ is:*

$$N + N' \xrightarrow{id_{N+N'} = \langle \pi_1, \dots, \pi_{N+N'} \rangle} N + N' \underset{g'}{\overset{f'}{\rightrightarrows}} M$$

The elements of the diagram f', g', $N + N'$ and M, will remain static throughout all the algorithm, so our algorithm state will only needs to include leftmost arrow in the starting diagram.

Definition 9.13 (Algorithm State)**.** *For a pullback problem of type N, the algorithm state is $(S \mid h)$, $h : N \to N$ an arrow and S an ordered multiset of equations $f \approx g$ between arrows $f, g : N \to 1$.*

Definition 9.14 (Helper Substitution)**.** *The helper substitution function $\vec{\sigma}$ is defined as $\vec{\sigma}\,(i, f : N \to 1, h : N \to N) = h'$, where $\langle \pi_1, \dots, \pi_{i-1}, f, \pi_{i+1}, \dots, \pi_N \rangle; h \to^!_R h'$. Basically, this function replaces any π_i in h by f.*

If we visualize the arrow $h : N \to N$ as a function mapping N memory "cells" to N "registers", $\vec{\sigma}$ is the action of replacing the i-th memory cell by f.

Definition 9.15 (Pullback Calculation Algorithm)**.** *The input of the algorithm is two arrows $f_0 : N \to M$ and $g_0 : N' \to M$. First, build the starting diagram $\mathcal{P}$, which produces arrows f'_0 and g'_o, and a type of the problem $N + N' = N_T$. Diagrammatically:*

$$N_T \xrightarrow{id_{N_T} = \langle \pi_1, \ldots, \pi_{N_T} \rangle} N_T \underset{g'_0}{\overset{f'_0}{\rightrightarrows}} M$$

f'_0 and g'_0 are of the form $\langle f_1, \ldots, f_M \rangle$, $\langle g_1, \ldots, g_M \rangle$. Then build the initial set $S = \{f_1 \approx g_1, \ldots, f_M \approx g_M\}$. The initial state is $(S \mid \langle \pi_1, \ldots, \pi_{N_T} \rangle)$.

The algorithm proceeds to transform the state $(S \mid h)$ iteratively until $S = \emptyset$ using the following rules:

- *Pick an equation from S such that $S = \{f \approx g\} \cup S'$. Compute $h; f \to^!_R f'$ and $h; g \to^!_R g'$. Then, obtain the new state by case analysis on $f' \approx g'$:*

$$\begin{array}{llcll}
!_M; a & \approx & !_M; b & \Rightarrow & Fail \\
!_M; a & \approx & h; f & \Rightarrow & Fail \\
g; f & \approx & g'; f' & \Rightarrow & Fail \\
!_M; a & \approx & !_M; a & \Rightarrow & (S' \mid h) \\
!_M; a & \approx & \pi_i & \Rightarrow & (S' \mid \vec{\sigma}\,(i, !_M; a, h)) \\
\pi_i & \approx & \pi_j & \Rightarrow & (S' \mid \vec{\sigma}\,(j, \pi_i, h)) \\
\pi_i & \approx & g; f & \Rightarrow & (S' \mid \vec{\sigma}\,(i, g; f, h)) \\
g; f & \approx & g'; f & \Rightarrow & (\{g_1 \approx g'_1\} \cup \cdots \cup \{g_n \approx g'_n\} \cup S' \mid h)
\end{array}$$

Note that the type of h is not changed thorough the algorithm, so when $S = \emptyset$, our diagram will be commutative but may not be an equalizer for having an incorrect domain.

Adjusting h to be a monic arrow — that is to say, we inspect h and determine the K elements of the product N_T that are not referenced in h — is enough. Compose $h : N_T \to N_T$ with a monic extension of $id_{N_T - K}$ to N_T such that the occurrences of projections in h are assigned different indexes to obtain $h' : (N_T - K) \to N_T$. Then the equalizer is h'. If the algorithm fails, the equalizer is the initial arrow:

$$\bot \xrightarrow{?_{N_T}} N_T \underset{g'}{\overset{f'}{\rightrightarrows}} M$$

The process of converting h to a monic is similar to *garbage collection* and memory de-fragmentation. Imagine that $h : 4 \to 4$ is $h = \langle \pi_3, \pi_3, \pi_2; f, \pi_3; f \rangle$. In this case, π_1 and π_4 don't occur in h, so we may safely adjust the type of h to $h : 2 \to 4$. This is done categorically using the monic extension $e : 2 \to 4$, let's use for instance $e = \langle \pi_1, \pi_2, \pi_1, \pi_1 \rangle$, so $h' = e; h$ is $h' = \langle \pi_1, \pi_1, \pi_2; f, \pi_1; f \rangle$. Viewed computationally, the term sequence for h' is using just two variables versus 4 for h.

As an example, we calculate the pullback of arrows $f : 3 \to 2 = \langle \langle \pi_1, \pi_2 g, \pi_2, \pi_1 h \rangle f, \pi_3 \rangle$ and $g : 3 \to 2 = \langle \langle !_3 a, \pi_1, \pi_2 g, \pi_3 \rangle f, \pi_3 h \rangle$. The type of the problem is 6. The starting diagram is:

$$6 \xrightarrow{\langle \pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6 \rangle} 6 \underset{\langle \langle !_6 a, \pi_4, \pi_5 g, \pi_5 \rangle f, \pi_6 h \rangle}{\overset{\langle \langle \pi_1, \pi_2 g, \pi_2, \pi_1 h \rangle f, \pi_3 \rangle}{\rightrightarrows}} 2$$

The algorithm proceeds:

$$\begin{array}{lll}
(\{\langle \pi_1, \pi_2 g, \pi_2, \pi_1 h\rangle f \approx \langle !_6 a, \pi_4, \pi_5 g, \pi_5\rangle f, \pi_3 \approx \pi_6 h\} & \mid \langle \pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6\rangle) & \to \\
(\{\pi_1 \approx !_6 a, \pi_2 g \approx \pi_4, \pi_2 \approx \pi_5 g, \pi_1 h \approx \pi_5, \pi_3 \approx \pi_6 h\} & \mid \langle \pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6\rangle) & \to \\
(\{\pi_2 g \approx \pi_4, \pi_2 \approx \pi_5 g, \pi_1 h \approx \pi_5, \pi_3 \approx \pi_6 h\} & \mid \langle !_6 a, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6\rangle) & \to \\
(\{\pi_2 \approx \pi_5 g, \pi_1 h \approx \pi_5, \pi_3 \approx \pi_6 h\} & \mid \langle !_6 a, \pi_2, \pi_3, \pi_2 g, \pi_5, \pi_6\rangle) & \to \\
(\{\pi_1 h \approx \pi_5, \pi_3 \approx \pi_6 h\} & \mid \langle !_6 a, \pi_5 g, \pi_3, \pi_5 g g, \pi_5, \pi_6\rangle) & \to \\
(\{\pi_3 \approx \pi_6 h\} & \mid \langle !_6 a, !_6 a h g, \pi_3, !_6 a h g g, !_6 a h, \pi_6\rangle) & \to \\
(\emptyset & \mid \langle !_6 a, !_6 a h g, \pi_6 h, !_6 a h g g, !_6 a h, \pi_6\rangle) &
\end{array}$$

We perform the renaming step. There's only one projection used, π_6, so we compose h with $\langle \pi_1, \pi_1, \pi_1, \pi_1, \pi_1, \pi_1\rangle$ to obtain the final equalizer:

$$1 \xrightarrow{\langle !_1 a, !_1 a h g, \pi_1 h, !_1 a h g g, !_1 a h, \pi_6\rangle} 6 \underset{\langle\langle !_6 a, \pi_4, \pi_5 g, \pi_5\rangle f, \pi_6 h\rangle}{\overset{\langle\langle \pi_1, \pi_2 g, \pi_2, \pi_1 h\rangle f, \pi_3\rangle}{\rightrightarrows}} 2$$

Note that in the interest of mimicking the actual behavior of many Prolog implementations, we didn't implement the occur-check in the rules. Without the occur-check, it is impossible to develop termination or soundness results. Let $\Delta = \langle \pi_1, \pi_1\rangle$ in the following example:

$$\begin{array}{ccc}
 & & 1 \\
 & & \downarrow \langle \pi_1, \Delta; f\rangle \\
 & & 2 \xleftarrow{\langle \pi_1, \langle \Delta; f, \Delta; f\rangle; f\rangle} 1
\end{array}$$

The pullback of the above diagram is the initial object $\perp$, as there is no arrow g such that $g; \langle \pi_1, \pi_1\rangle = g; \pi_1$ is true in the original Lawvere category. However, the pullback algorithm won't terminate and proceed to construct the infinite arrow:

$$\ldots \quad 1 \xrightarrow{\Delta; f} 1 \xrightarrow{\Delta; f} 1 \xrightarrow{\Delta; f} 1$$

In that case, the resulting diagram won't be a pullback, and the algorithm may even fail to terminate.

In order to prove some properties of the algorithm, we consider a slightly different version from the one presented here. We noted that the domain of the substitution arrow h is not modified. This choice was done to remain in consonance with what would happen in an implementation, but now we'd like to track which variables are in use.

We modify the substitution function $\vec{\sigma}$ as follows:

Definition 9.16 (Reducing Helper Substitution)**.** *Let i be a natural number, $f : (N-1) \to 1$ and $h : N \to N$ arrows. Then, the helper substitution function is $\vec{\sigma}(i, f, h) = h' : (N-1) \to N$, where*

$$\langle \pi_1, \ldots, \pi_{i-1}, f, \pi_i, \ldots, \pi_{N-1}\rangle; h \to_R^! h'$$

Now, the categorical framework tracks which variables occur in a term, in a similar fashion to structural approaches like the one in [McBride, 2003]. So for a term in N free vars, the substitution operation returns a term with $N-1$ free variables. This scheme is, by itself, enough to guarantee termination.

The rules of the algorithm are now modified as follows:

$$\begin{array}{lllll}
!_M; a & \approx & \pi_i & \Rightarrow & (S' \mid \vec{\sigma}(i, !_{M-1}; a, h)) \\
\pi_i^M & \approx & \pi_j & \Rightarrow & (S' \mid \vec{\sigma}(j, \pi_i^{M-1}, h)) \\
\pi_i & \approx & g; f & \Rightarrow & (S' \mid \vec{\sigma}(i, R; g; f, h)) \quad \text{if } g; f \text{ not using } \pi_i
\end{array}$$

Note the new side condition of $g; f : M \to 1$ *not using* π_i. What this means is that the projection π_i doesn't occur in $g; f$, that is to say, the arrow is invariant under any embedding $R : (M - 1) \to M$. Then, $\vec{\sigma}$ uses such and embedding to create a new diagram with a domain of $M - 1$.

Lemma 9.17. *The modified algorithm terminates.*

PROOF. For the fail rules, the proof is immediate. Let m, n be natural numbers and $(m, n) > (m', n')$ if $m > m'$ or $m = m'$ and $n > n'$. Assume that the last rule (equation introduction) is performed automatically for every equation after a substitution. Then, assigning m to the type of the diagram, and n the number of equations, we see that when a substitution happens, it may augment the number of equations, but it will decrease the type of the diagram. □

Lemma 9.18. *For a final state ($\emptyset \mid h : E \to M$), the starting diagram commutes, that is to say $h; f'_0 = h; g'_0$*

PROOF. The equational theory of a regular Lawvere category is straightforward, and mainly imposes that $\langle f, g\rangle = \langle f', g'\rangle$ iff $f = f'$ and $g = g'$. The normalized disagreement set of a diagram is built by performing the transformation $\langle f, g\rangle = \langle f', g'\rangle \Rightarrow \{f = f' \cup g = g'\}$

It is easy to check that for a given diagram and state of the with equations S, the normalized disagreement set coincides. Then, if $S = \emptyset$, it means that the diagram is commutative. □

We should remark that $\vec{\sigma}$ is always sound thanks to its implementation as arrow composition, given that $f = g \Rightarrow h; f = h; g$. In other words, no substitution can make a commutative diagram into a non-commutative one.

Lemma 9.19. *Given a family of arrows h_N in a regular Lawvere category, such that for every h_i, $h_i; f = h_i; g$, the equalizer is arrow h_j such that it is a monic and has greater domain.*

PROOF. Suppose that the arrow h_j is not a monic. Then, the uniqueness property of the equalizer trivially fails.

Suppose that the arrow h_j is not the one with the greatest domain that is a monic. Then, the monic arrow with the greatest index cannot be factored through h_j, as we cannot factor a monic $M + 1 \to N$ through a monic $M \to N$. The existence property fails. □

Lemma 9.20. *The algorithm returns an arrow h that is monic and has maximal domain.*

PROOF. An arrow $h : M \to N$ is monic in a RLC iff all π_i from 1 to M occur in the normalized representation of h. Thus, h must be the monic arrow with the maximum number of distinct π_i that makes the diagram commute. The end of the algorithm guarantees that the arrow is monic. The disagreement set shows that the algorithm eliminates the minimum number of π_i terms in order to make the diagram commute, so h is the required arrow. □

Lemma 9.21. *Given a starting diagram:*

$$N + N' \xrightarrow{id_{N+N'} = \langle \pi_1, \ldots, \pi_{N+N'}\rangle} N + N' \underset{g'}{\overset{f'}{\rightrightarrows}} M$$

If the algorithm ends with state ($\emptyset \mid h$), $h' : E \to M$ is the equalizer of $f'g'$.

PROOF. A consequence of the previous lemmas. □

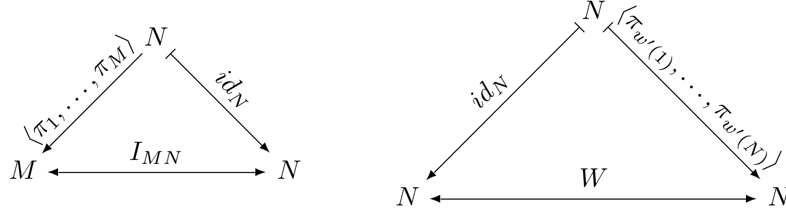


FIGURE 9.1. Helper relations for the translation

9.7. Translation of the Program

The compilation procedure for a logic program is its interpretation in a Σ-allegory. The translation is almost identical to the one used in Chapter 6. In that version, the lack of typing information and tabulations forced the duplication of the constraint store $\dot{K}(\varphi)$ on predicate call:

$$\underbrace{\dot{K}(\varphi)} \cap I_m; R; I_m^\circ \rightarrow I_m; (I_m; \underbrace{\dot{K}(\varphi)}; I_m^\circ \cap R); I_m^\circ \cap \underbrace{\dot{K}(\varphi)}$$

Within the $\mathbf{QRA}_\Sigma$ theory, the only available rule for algebraic reasoning in a predicate call is the modular law, which forces two occurrences of the relation $\dot{K}(\varphi)$ in the rightmost term. This not only amounts to an unnecessary duplication, but it delays the propagation of substitutions happening in the inside context, and provides an execution model quite different from the one used in practice.

How can we avoid this? Using $\mathbf{QRA}_\Sigma$, we lack any information about the elements of the vector affected by $\dot{K}(\varphi)$. Performing some non-algebraic analysis to determine which relational projections are used may be an approach, but we don't consider this approach to fit well withing our objectives of a declarative framework. However, types and category theory do keep track of this information.

Before defining the categorical translation, we define categorical versions of the non-typed relations W and I. The difference between typed and untyped W is minimal, behaving as a permutation in the same manner than its non-categorical counterpart.

Definition 9.22 (W Relation)**.** *Given a projection $w : N \rightarrow M$, with $N \geq M$ and $K = N - M$, we denote by $w' : N \rightarrow N$ any of its extensions to a permutation such that the following equations are satisfied: $\{w'(K) = w^{-1}(1), \ldots, w'(K + M) = w^{-1}(M)\}$. Fixed a w', W is tabulated by $(N, \langle \pi_{w'(1)}, \ldots, \pi_{w'(N)} \rangle)$. See Fig. 9.1.*

Basically the W relation permutes a vector, putting the last interesting elements in the end of the vector. We don't care about the order of the other elements as they won't play a role. For $w : N \rightarrow N$, W behaves as a regular permutation.

However, the typed I_{MN} relation is very different from the untyped I_M. I_M is a partial identity between vectors of *unbounded size*, whereas I_{MN} is the partial identity relation between vectors of size M and N:

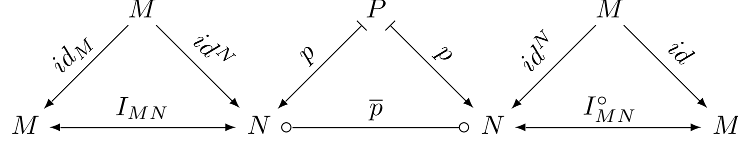
Definition 9.23 (I Relation)**.** *The relation or partial identity I_{MN}, with $M < N$ is defined by the tabulations $(\langle \pi_1, \ldots, \pi_M \rangle, id_N)$. See Fig. 9.1.*

We visualized I_M as a "cleaner" relation for the elements $M + 1$ in the sequence, such that $(\{\ldots, a_{m+i}, \ldots\}, \{\ldots, a_{m+i}, \ldots\}) \in I_m; R; I_m$ for all $R \subseteq id$ and $a \in \mathcal{T}_\Sigma$. Now that the vector size is fixed, I_{MN} plays the role of a creator — and its converse a *destroyer* — of local variables. The intuition that the reciprocal of a projection creates a new variable is formalized, indeed, $I_{12} = \pi_1^\circ$, the converse of the categorical projection, creates a new local variable.

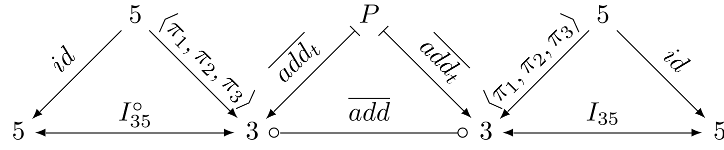
Using a direct lifting of the approach used in Chapter 6, we define the translation of a call to a predicate p of arity N from context of arity M as:

$$\dot{K} \cap (I_{MN}; \overline{p}; I_{MN}^{\circ})$$

Diagrammatically:



We may use a more concrete example using the standard add predicate implementing Peano addition. As its translation $\overline{add}$ is coreflexive, it is tabulated by a monic $\overline{add_t}$:



This approach is semantically correct, but poses a problem: neither the relation $I_{MN}; \overline{p}; I_{MN}^{\circ}$ or $I_{35}; \overline{add}; I_{35}^{\circ}$ are coreflexive. Thus, for a query $K \cap I_{MN}; \overline{p}; I_{MN}^{\circ}$ we would need to compute not only the composition, but also a intersection.

Furthermore, we want the machine and semantics to be fully-based on relation composition, allowing us to base the machine on relation composition and to incorporate interesting categorical semantic developments from functional programming. We profit from the key fact that for a coreflexive relation $R \subseteq id$ then $R; S = R \cap S$. That is to say, if we transform $I_{MN}; \overline{p}; I_{MN}^{\circ}$ to a coreflexive relation we may eliminate all intersections and base our machine on pullbacks.

We will abuse notation and profit from the fact that a coreflexive relation is uniquely tabulated by a monic $f^{\circ}; f$ to write f for $f^{\circ}; f$ when it can be deduced from the context.

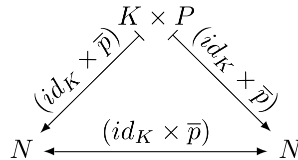
How can we convert $I_{MN}^{\circ}; \overline{p}; I_{MN}$ to a coreflexive relation? Clearly, given that $I_{MN}^{\circ}; \overline{p}; I_{MN} \not\subseteq id$, then any coreflexive substitute may alter soundness. However, this is not a problem as $I_{MN}^{\circ}; \overline{p}; I_{MN}$ will always occur in a context where it is intersected with a coreflexive relation. Thus we have the following derivation:

$$\dot{K} \cap I_{MN}; \overline{p}; I_{MN}^{\circ} = (\dot{K} \cap id) \cap I_{MN}; \overline{p}; I_{MN}^{\circ} = \dot{K} \cap (id \cap I_{MN}; \overline{p}; I_{MN}^{\circ})$$

and:

$$id \cap I_{MN}; \overline{p}; I_{MN}^{\circ} = (\overline{p} \times id_{N-M})^{\circ}; (\overline{p} \times id_{N-M})$$

We reverse the product so the values of $\overline{p}$ are placed in the end of the sequence. So the wrapping in an environment of type N for a predicate $\overline{p} : P \to M$, with $K = N - M$ is:



As we can see in the diagram, the K free variables that form the wrapping are tabulated from a common object K. This makes the relation coreflexive iff $\overline{p}$ is and will enable the propagation of substitutions happening when computing $\overline{p}$. For instance, imagine the following wrapper for a predicate tabulated by

$\overline{p_1} : 1 \to 1$ in a context 3:

$$\begin{array}{ccc} & 2 \times 1 & \\ {}^{(id_2+\overline{p_1})}\swarrow & & \searrow^{(id_2+\overline{p_1})} \\ 3 & \xleftrightarrow{(id_2 \times \overline{p_1})} & 3 \end{array}$$

Then, for whatever reason the first register and the third one got unified, which is done by composing $u = \langle \pi_1, \pi_2, \pi_1 \rangle$ with the tabulations:

$$\begin{array}{ccc} & 2 & \\ {}^{u;(id_2+\overline{p_1})}\swarrow & & \searrow^{u;(id_2+\overline{p_1})} \\ 3 & \xleftrightarrow{u;(id_2 \times \overline{p_1})} & 3 \end{array}$$

Then, any instantiation of $\overline{p_1}$ will also affect the first position of the vector.

We now proceed to define the full translation procedure: First, each clause of the program is rewrote to general form. The set of n variables occurring in the clause is renamed to y_1 to y_n. Every term occurring as an argument in the head and tail is assigned to a new variable x_i. After that process, clauses are of the form:

$$cl_i : p(\vec{x}') \leftarrow \vec{x} = \vec{t}[\vec{y}], p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n).$$

$\vec{x}'$ a prefix of $\vec{x}$, $\vec{x}_i$ a selection of variables in $\vec{x}$ and $\vec{t}$ a sequence of terms using variables in $\vec{y}$. Informally, we may see $\vec{y}$ as a set of *global variables*, whereas the $\vec{x}$ plays the role of term-containing registers.

Replace $\vec{x}_i$ for projections $w_i(\vec{x})$ such that $w_i(\vec{x}) = \vec{x}_i$. Clauses are now of the form:

$$p(\vec{x}') \leftarrow \vec{x} = \vec{t}[\vec{y}], p_1(w_1(\vec{x})), \ldots, p_n(w_n(\vec{x})).$$

Define $M = |\vec{x}'|$, $N = |\vec{x}|$. We first build a coreflexive relation between sequences of terms $K(\vec{t})$, of type $|\vec{t}| \to |\vec{t}|$. Such relation, is tabulated by an arrow $|\vec{y}| \to |\vec{t}|$, that is to say, it constructs the terms from a supply of fresh variables corresponding to $\vec{y}$.

Definition 9.24 (Term Translation)**.** *The translation function K takes a sequence of terms $\vec{t}$, using $\vec{y} \equiv [y_1, \ldots, y_{|\vec{y}|}]$ variables and returns a coreflexive tabular relation $K(\vec{t}) : |\vec{t}| \to |\vec{t}|$ with tabulation $f : |\vec{y}| \to |\vec{t}|$.*

$$\begin{array}{lcl} K(\vec{t}[\vec{y}]) & = & \langle K_{\vec{y}}(t_1), \ldots, K_{\vec{y}}(t_n) \rangle^\circ ; \langle K_{\vec{y}}(t_1), \ldots, K_{\vec{y}}(t_n) \rangle \\ \textit{where} & & \\ K_{\vec{y}}(a) & = & !_{|\vec{y}|} ; a : |\vec{y}| \to 1 \\ K_{\vec{y}}(y_i) & = & \pi_i : |\vec{y}| \to 1 \\ K_{\vec{y}}(f(t_1, \ldots, t_n)) & = & \langle K_{\vec{y}}(t_1), \ldots, K_{\vec{y}}(t_n) \rangle ; f : \alpha(\vec{y}) \to 1 \end{array}$$

As before, $\wedge$ corresponds to intersection of relations. Then we wrap the predicates into a relational projection W_i generated from w_i. Say $K_i = N - \alpha(p_i)$. Using the defined wrappers we build the relational term for the tail:

$$K(\vec{t}); W_1; (id_{K_1} \times \overline{p_1}); W_1^\circ; \ldots; W_n; (id_{K_n} \times \overline{p_n}); W_n^\circ$$

Note that we use composition in place of intersection as both $K(\vec{t})$ and each term $(id_{K_i} \times \overline{p_i})$ are coreflexive. We are close to the final form. As $|\vec{x}'| \leq |\vec{x}|$, the type of the above term is not the correct one for $\overline{p}$. Then, we wrap the translation of the tail with the partial identity relation $I_{MN} : M \to N$. The final translation for the clause is the arrow:

$$\overline{cl_i} = I_{MN}; K(\vec{t}); W_1; (id_{K_1} \times \overline{p_1}); W_1^\circ; \ldots; W_n; (id_{K_n} \times \overline{p_n}); W_n^\circ; I_{MN}^\circ$$

Now we clearly see why I_{MN} plays the role of *environment* creation and destruction. A predicate p consisting of several clauses is translated using $\cup$:

$$\begin{array}{l} p(\vec{x}) \leftarrow cl_1 \vee \cdots \vee cl_m \quad \Rightarrow \\ \overline{p} = \overline{cl_1} \cup \cdots \cup \overline{cl_m} \end{array}$$

where $\overline{cl_i}$ is the arrow corresponding to the translation of the clause cl_i.

Our approach to the handling of recursive predicates is similar to the one used in the non-categorical case. We add a set of primitive symbols for predicates and extend the equational theory of the allegory. For instance, the predicate nat :

$$\overline{nat} = K(X_1 = o) \cup I_{12}; K(X_1 = s(y_1), X_2 = y_1); (id_1 \times \overline{nat}); I_{21}^{\circ}$$

or in full categorical form:

$$\overline{nat} = !_1; o \cup I_{12}; \langle \pi_1; s, \pi_1 \rangle; (id_1 \times \overline{nat}); I_{21}^{\circ}$$

the equation is added to the category so $\overline{nat}$ may be viewed as the limit of the sequence:

$$\begin{array}{ll} !_1; o & \cup \\ I_{12}; \langle \pi_1; s, \pi_1 \rangle; (id_1 \times !_1; o); I_{21}^{\circ} & \cup \\ I_{12}; \langle \pi_1; s, \pi_1 \rangle; (id_1 \times I_{12}; \langle \pi_1; s, \pi_1 \rangle; (id_1 \times !_1; o)); I_{21}^{\circ} & \cup \\ \ldots & \end{array}$$

which could actually be represented as:

$$!_1; o \cup !_1; o; s \cup !_1; o; s; s \cup !_1; o; s; s; s \cup \ldots$$

Theorem 9.25 (Adequacy of the Translation)**.** *Given a predicate p of arity N translated to the arrow $\overline{p} : N \to N$, the initial model maps $\overline{p}$ to the subobject $\xrightarrow{[\![\overline{p}]\!]_S} \mathcal{T}_{\Sigma}^{N}$ such that its image is precisely the set of ground terms making p true.*

Proof. The proof proceeds in a similar manner than the one of Theorem 6.21, given that the translation is close to identical. For each non-recursive clause, is easy to check that the semantics of its translation are the relations such that $p(\vec{t})$ iff $(\vec{t}, \vec{t}) \in [\![\overline{p}]\!]_S$. Indeed, a non-recursive clause corresponds uniquely to an arrow of the underlying regular Lawvere category.

For the recursive case, we assign $[\![\overline{p}]\!]_A^0 = \emptyset$ and use the theory of the allegory to unfold the definition imitating a fixpoint. This way we obtain an arrow $A_1 \cup A_2 \cup \ldots$, which is in coincidence whose interpretation is the coreflexive relation that represents the subset resulting from the van Emden-Kowalski fixpoint operator. $\square$

9.7.1. An Example of the Translation. We use as example of the translation the prolog predicate `add` implementing Peano addition:

```
add(o,X,X).
add(s(X), Y, s(Z)) :- add(X, Y, Z).
```

We perform the renaming procedure:

```
add(X1,X2,X3) :- X1 = o, X2 = Y1, X3 = Y1.
add(X1,X2,X3) :- X1 = s(Y1), X2=Y2, X3 = s(Y3), X4=Y1, X5=Y3,
                 add(X4, X2, X5).
```

Note that we have two *kinds* of variables, the ones starting by X which may only appear as arguments to predicates and the Y variables, which represent the "real" variables used inside the predicate. Externally, `add` only uses three X variables, but internally it needs two more. In our relational translation, we will capture this fact by using a relation $I_{35} : 3 \to 5$ that takes care of creating $X4$ and $X5$. Recall that $\langle f, g \rangle$

is the categorical product constructor. Then, storing all our X variables in such a product, we may try to express add in a relational pseudo-notation:

$$\begin{array}{rcl}\overline{add} & = & \langle o, Y1, Y1\rangle \\ & \cup & I_{35}; (\langle s(Y1), Y2, s(Y3), Y1, Y3\rangle \cap (id_2 \times \overline{add})); I^{\circ}_{35}\end{array}$$

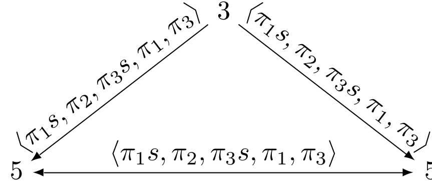
the recursive call to $\overline{add}$ is wrapped into a vector of size 5, but we are calling it with the wrong parameters! The above expression is equivalent to $add(X3, X4, X5)$. We need to call it with the right parameters, so we compose the call with a permutation of the vector. We replace Y variables by categorical projections and the actual translation is:

$$\begin{array}{rcl}\overline{add} & = & \langle o, \pi_1, \pi_1\rangle^{\circ}; \langle o, \pi_1, \pi_1\rangle \\ & \cup & I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle^{\circ}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle; W; (id_2 \times \overline{add}); W^{\circ}; I^{\circ}_{35}\end{array}$$

where $I_{35} : 3 \to 5 = \langle \pi_1, \pi_2, \pi_3\rangle^{\circ}$ and $W : 5 \to 5 = \langle \pi_1, \pi_3, \pi_4, \pi_2, \pi_5\rangle$. We abuse notation and write f for a coreflexive relation $f^{\circ}; f$. With this abuse in mind, the translation is:

$$\begin{array}{rcl}\overline{add} & = & \langle o, \pi_1, \pi_1\rangle \\ & \cup & I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle; W; (id_2 \times \overline{add}); W^{\circ}; I^{\circ}_{35}\end{array}$$

The tabulation of $\langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3\rangle$ is:



The reader can see how domain of the tabulations reflects the number of free variables in use by the machine, information which is usually associated to global storage. The codomain of the tabulations — the actual domain of the relations — should be interpreted as the number or working "temporal registers" that are used for parameter passing and unification.

9.8. The Categorical Machine

Given that we know how to compute relation composition for union-free arrows in Σ-allegories, the actual categorical machine has to take care of unfolding predicate definitions and distributing disjunctive arrows. Thus, the machine itself is quite simple as all the heavy work has been lifted to the pullback algorithm.

We present the machine as an algebraic specification. Each state corresponds to a diagram, and each step corresponds to diagram rewriting. The main rule is the rewriting induced by relation composition as can be seen in Fig. 9.2.

We may summarize the basic principles of the allegorical computational paradigm here:

- Based on relation composition.
- Predicates are coreflexive relations, thus tabulated by monics.
- Domain of the tabulations represent the number of logical variables in use.
- Co-domain of a tabulation, or what it is the same, the type of the relations represents the number of "temporal" registers in use.

Every predicate of our logic program is translated to an arrow in a Σ-allegory. Then a query will be translated as an empty predicate and normalized by the execution engine. The engine is based on rewriting a diagram representation of the actual categorical constructs. The basic rule, which relates relation composition with the calculation of a pullback can be seen in Fig. 9.2, together with the diagram rewriting used.

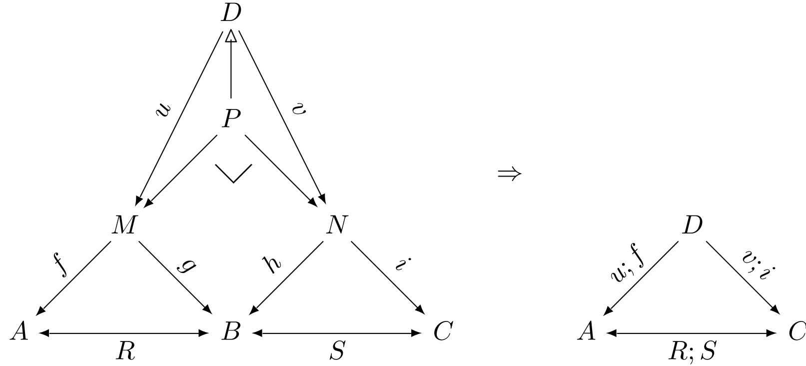


FIGURE 9.2. Composition of Relations as a Computation Rule

9.8.1. Transition Rules. We define the categorical machine as a set of transition rules over relations. We write $(f \mid g)$ for tabular relations. Then, $(f \mid g); (f' \mid g')$ is rewrote to $(h; f \mid h'; g')$ using the pullback (h, h') of g, f'. This corresponds to a substitution, where the arrow $h : M \to N$ takes a current state of the machine using N variables to a state using M variables, and $h' : M' \to N'$ does the same, usually instantiating the translations of a clause to the right variables. This mechanism is also used for variable creation/destruction. The pair of arrows (h, h') above the transition arrow denotes the result of the pullback.

A union $R_1 \cup \cdots \cup R_n$ is used to represent disjunctive search, while predicate calls are represented as $(f \mid \langle g, [R] \rangle)$, where R is the relation pertaining to the call in-progress. Note that g and the left tabulation of R share the same domain, allowing the propagation of substitutions resulting from reducing R to the outer context.

$$\begin{array}{llll}
(f \mid g); (f' \mid g') & \xrightarrow{(h,h')} & (h; f \mid h'; g') & \\
(f \mid \langle g_K, g_N \rangle); (id_K \times \overline{p_N}) & \Rightarrow & (f \mid \langle g_K, [(g_N \mid g_N); p_1] \rangle) \cup & \\
 & & \vdots \qquad\qquad\qquad\qquad \cup & \\
 & & (f \mid \langle g_K, [(g_N \mid g_N); p_n \rangle]) & \\
(f \mid \langle g, [(g' \mid g')] \rangle) & \Rightarrow & (f \mid \langle g, g \rangle) & \\
(f \mid \langle g, [E] \rangle) & \Rightarrow & (h; f \mid \langle h; g, [E'] \rangle) & \text{iff } E \xrightarrow{(h,h')} E' \\
R \cup S & \Rightarrow & R' \cup S & \text{iff } R \Rightarrow R' \\
\mathbf{0} \cup S & \Rightarrow & S &
\end{array}$$

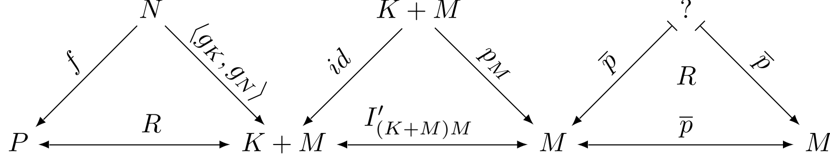
The first rule represents composition of tabular relations. The second one represents predicate call. First, disjunctive predicates are unfolded using the rule $f; (R \cup S) = f; R \cup f; S$. Computing the predicate call is performed by the relation $(g_N \mid g_N); p_1$. The third rule deals with return. The three last rules encode the search strategy of the machine.

Lemma 9.26 (Soundness)**.** *The machine is sound, that is to say, for a rule $R \Rightarrow S$, then $R = S$ in the theory.*

PROOF. The base case is straightforward, as the diagram of Fig. 9.2 shows. The cases involving $\cup$ are a direct consequence of the axioms of distributive allegories.

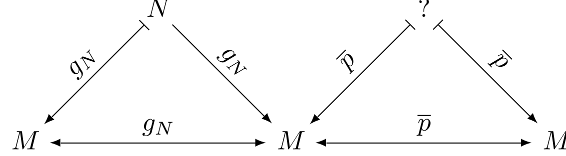
The remaining cases are the ones handling predicate call. There are several ways to show the correctness, most of them based on unfolding the pullback the occurs in a predicate call and proving some properties of it. Instead, in this proof we use the equivalence $(id_K \times \overline{p}) = I'_{NM}; \overline{p}; I'_{MN} \cap id$. So the problem of reducing $R; (I_{NM}; \overline{p}; I_{MN} \cap id)$ can be tackled in two steps: $(R; I_{NM}; \overline{p}; I_{MN}) \cap R$.

Let $(f \mid \langle g, [R]\rangle)$ be a machine state, we let it be represented by the diagram:

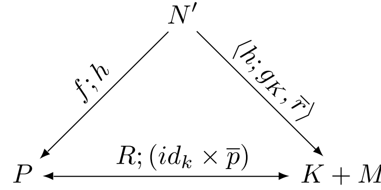


where $p_M = \langle \pi_{K+1}, \ldots, \pi_{K+M}\rangle$.

By routine calculation, this amounts to reducing:



which will return the result of the pullback (h, h') along with a coreflexive arrow $\overline{r}$. Then, we construct the following diagram after return:



□

Theorem 9.27 (Operational equivalence)**.** $\langle p_1(\vec{t_1}), \ldots, p_n(\vec{t_n})\rangle \to \cdots \to \square$ *is the SLD derivation with substitution* σ *iff*

$$K(\vec{t}); W_1; (id_{K_1} \times \overline{p_1}); W_1^\circ; \ldots; W_n; (id_{K_n} \times \overline{p_n}); W_n^\circ \Rightarrow K(\sigma(\vec{u})) \cup R$$

Proof. In order to prove the equivalence, we must fix an increasing renaming scheme for the non-constrained transition of Def. 3.26. Then, if a state is using variables from x_1 to x_n, we rename apart a clause $cl : p(\vec{u}[\vec{y}]) \leftarrow \vec{q}(\vec{v}[\vec{z}])$ (with $\vec{y} \subseteq \vec{z}$ to $cl' : p(\vec{u}[\vec{y}']) \leftarrow \vec{q}(\vec{v}[\vec{z}'])$ in such a way that $\vec{z}'$ is $x_n, \ldots, x_{n+|\vec{z}|})$. Then, let δ be the m.g.u of $\vec{u}[\vec{y}']$ and $\vec{t}[\vec{x}]$ and the resolution transition is:

$$\langle p(\vec{t}[\vec{x}]), \vec{p}\rangle \quad \xrightarrow{cl}_r \quad \langle \delta(\vec{q}(\vec{v}[\vec{z}'])), \vec{p})\rangle$$

We check a single step, the rest follow by induction over the length of the derivation, the length of the query and Lemmas 6.43 and 6.44. The SLD strategy corresponds to our choice of always executing first the left part of a union, where incomplete derivations correspond to **0**.

Let's abuse notation and write here $\vec{u}$ for $K(\vec{u})$, etc... Then, for a single step, the relational term:

$$(\vec{t} \mid \vec{t}); W_1; (id_{K_1} \times \overline{p_1}); W_1^\circ; \ldots; W_n; (id_{K_n} \times \overline{p_n}); W_n^\circ$$

rewrites to

$$(\vec{t} \mid \langle \vec{t_r}, \vec{t_1}\rangle); (id_{K_1} \times \overline{p_1}); \ldots; W_n; (id_{K_n} \times \overline{p_n}); W_n^\circ$$

and to

$$(\vec{t} \mid \langle \vec{t_r}, [\vec{t_1}; \overline{p_1}]\rangle); \ldots; W_n; (id_{K_n} \times \overline{p_n}); W_n^\circ$$

where unfolding the definition of p_1 we get:

$$(\vec{t} \mid \langle \vec{t_r}, [\vec{t_1}; I_{MN}; \vec{u}; W; (id_{K_1'} \times \overline{q_1}); W^\circ; \ldots; W; (id_{K_m'} \times \overline{q_m}); W^\circ; I_{NM}]\rangle); \\ \ldots; W_n; (id_{K_n} \times \overline{p_n}); W_n^\circ$$

where the subquery is clear. The use of pullbacks guarantees that the renaming apart condition holds. □

9.8.2. Examples: The Machine in Action. For the first example we'll use the translation of the predicate add, whose translation is:

$$\begin{array}{rcl}\overline{add} & = & \langle o, \pi_1, \pi_1 \rangle \\ & \cup & I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle; W; (id_2 \times \overline{add}); W^\circ; I^\circ_{35}\end{array}$$

A query $add(s(X), Y, Z)$ is translated to $\langle \pi_1 s, \pi_2, \pi_3 \rangle; \overline{add}$. The execution trace is:

$$\begin{array}{lr}
\langle \pi_1 s, \pi_2, \pi_3 \rangle; \overline{add} & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3 \rangle; \langle o, \pi_1, \pi_1 \rangle) \cup \ldots & \Rightarrow \\
\mathbf{0} \cup \langle \pi_1 s, \pi_2, \pi_3 \rangle; I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle; W; (id_2 \times \overline{add}); W^\circ; I^\circ_{35} & \Rightarrow \\
\langle \pi_1 s, \pi_2, \pi_3 \rangle; I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle; W; (id_2 \times \overline{add}); W^\circ; I^\circ_{35} & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3 \rangle \mid \langle \pi_1 s, \pi_2, \pi_3, \pi_4, \pi_5 \rangle); \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle; W; (id_2 \times \overline{add}); W^\circ; I^\circ_{35} & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3 s \rangle \mid \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle); W; (id_2 \times \overline{add}); W^\circ; I^\circ_{35} & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3 s \rangle \mid \langle \pi_1 s, \pi_3 s, \pi_1, \pi_2, \pi_3 \rangle); (id_2 \times \overline{add}); W^\circ; I^\circ_{35} & \Rightarrow \\
(\langle \pi_1 s, \pi_2, \pi_3 s \rangle \mid \langle \pi_1 s, \pi_3 s, [\langle \pi_1, \pi_2, \pi_3 \rangle; \langle o, \pi_1, \pi_1 \rangle] \rangle; W^\circ; I^\circ_{35} \cup \ldots & \Rightarrow \\
(\langle os, \pi_1, \pi_1 s \rangle \mid \langle os, \pi_1 s, [\langle o, \pi_1, \pi_1 \rangle] \rangle; W^\circ; I^\circ_{35} \cup \ldots & \Rightarrow \\
(\langle os, \pi_1, \pi_1 s \rangle \mid \langle os, \pi_1 s, o, \pi_1, \pi_1 \rangle); W^\circ; I^\circ_{35} \cup \ldots & \Rightarrow \\
(\langle os, \pi_1, \pi_1 s \rangle \mid \langle os, \pi_1, \pi_1 s, o, \pi_1 \rangle); I^\circ_{35} \cup \ldots & \Rightarrow \\
\langle os, \pi_1, \pi_1 s \rangle \cup \ldots &
\end{array}$$

The component $\langle os, \pi_1, \pi_1 s \rangle$ is translated back to the answer $X = o, Z = s(Y)$.

We present the above example in diagrammatic form so the reader may understand better how the machine works. The example query `add(s(X), Y, Z)` is translated to the relation:

$$\langle \pi_1 s, \pi_2, \pi_3 \rangle \cap \overline{add}$$

Expanding the definition of $\overline{add}$ we obtain:

$$\langle \pi_1 s, \pi_2, \pi_3 \rangle \cap \langle o, \pi_1, \pi_1 \rangle \cup I_{35}; \langle \pi_2 s, \pi_1, \pi_3 s, \pi_2, \pi_3 \rangle; W; (id_2 \times \overline{add}); W^\circ; I^\circ_{35}$$

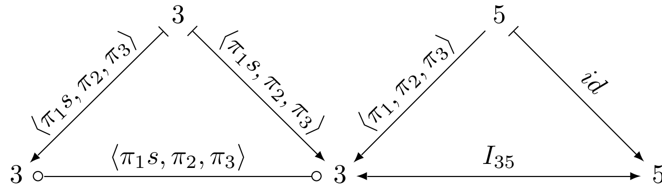
using the distributivity rule:

$$\begin{array}{lr}\langle \pi_1 s, \pi_2, \pi_3 \rangle; \langle o, \pi_1, \pi_1 \rangle & \cup \\ \langle \pi_1 s, \pi_2, \pi_3 \rangle; I_{35}; \langle \pi_2 s, \pi_1, \pi_3 s, \pi_2, \pi_3 \rangle; W; (id_2 \times \overline{add}); W^\circ; I^\circ_{35} & \end{array}$$

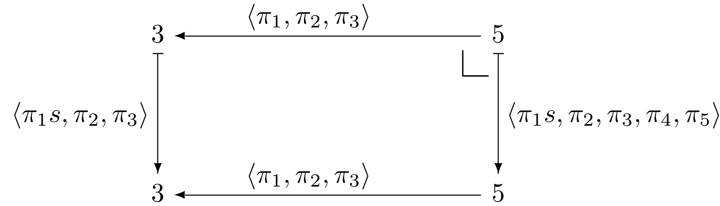
That first component is rewritten to 0, we focus on:

$$\langle \pi_1 s, \pi_2, \pi_3 \rangle; I_{35}; \langle \pi_2 s, \pi_1, \pi_3 s, \pi_2, \pi_3 \rangle; W; (id_2 \times \overline{add}); W^\circ; I^\circ_{35}$$

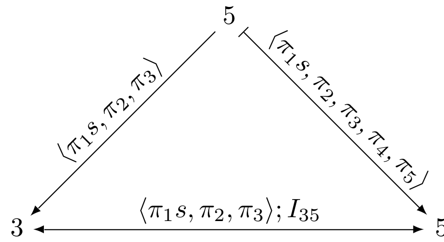
We calculate the first composition:



the corresponding pullback is:



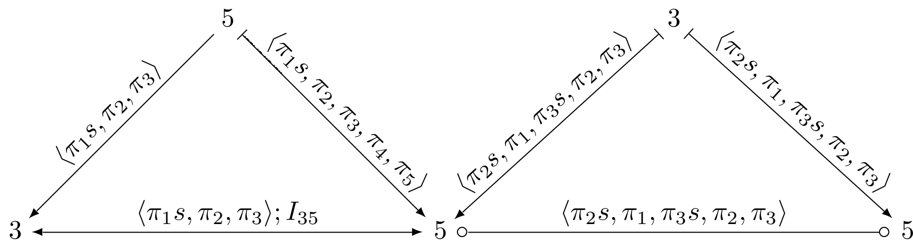
thus the composed relation is:



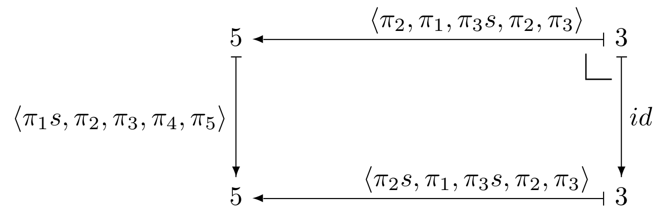
now we proceed to reduce:

$$(\langle \pi_1 s, \pi_2, \pi_3 \rangle ; I_{35}); \langle \pi_2 s, \pi_1, \pi_3 s, \pi_2, \pi_3 \rangle$$

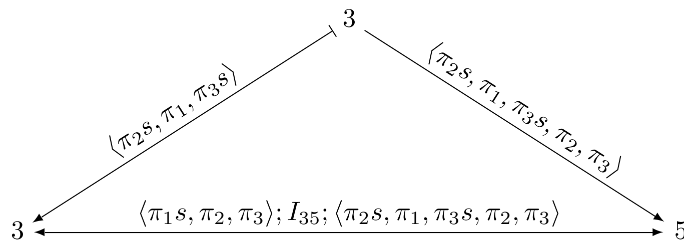
which is expressed diagrammatically as:



with pullback:



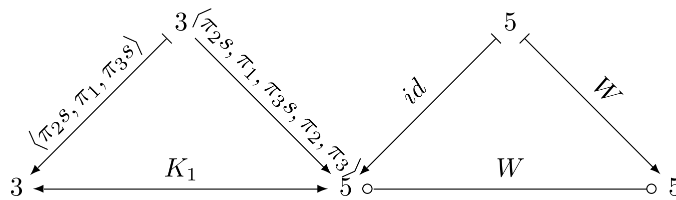
and composed relation:



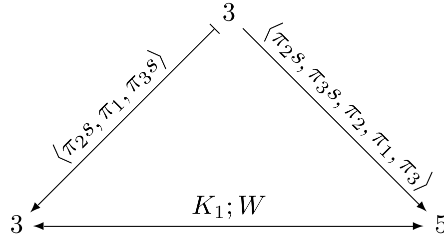
Let's write K_1 for $\langle \pi_1 s, \pi_2, \pi_3 \rangle ; I_{35} ; \langle \pi_2 s, \pi_1, \pi_3 s, \pi_2, \pi_3 \rangle$. The next step is:

$$K_1 ; W$$

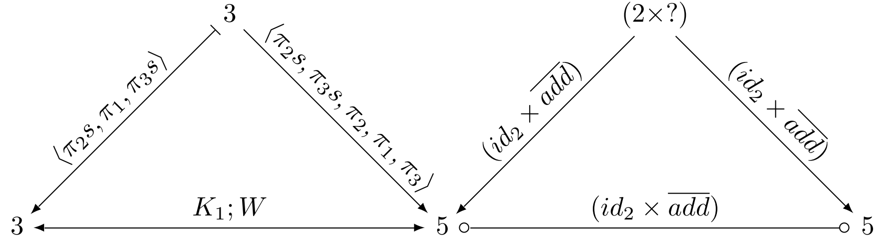
which given $W = \langle \pi_1, \pi_3, \pi_4, \pi_2, \pi_5 \rangle$, its diagrammatic form is:



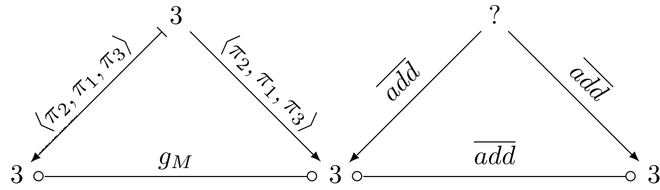
with trivial pullback and result:



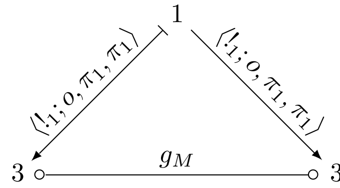
Notice how the parameters are in the right order for the "call" to add. We now reduce the call:



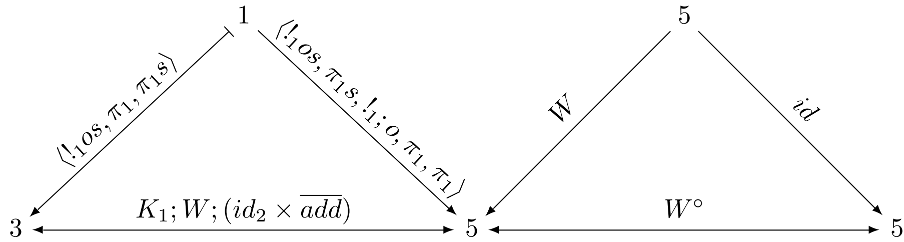
Now we apply the call rule so we have to reduce the diagram:



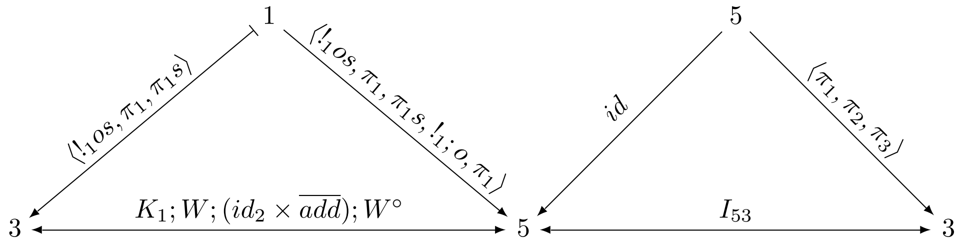
which will reduce to:



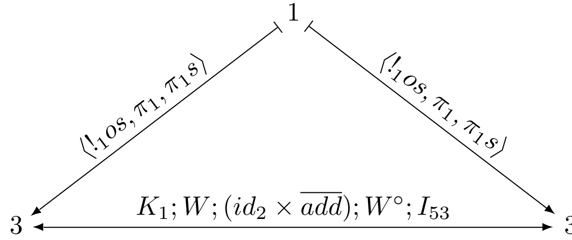
The return rule propagates the result and we proceed with the next step:



then to:



then to the first answer, $\langle !_1 os, \pi_1, \pi_1\rangle$:



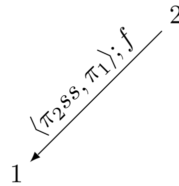
9.9. Memory Diagrams and Implementation Discussion

Before discussing implementation strategies, we need to have a closer look to the representation of our diagrams. If we think about the domain of a tabulation as a memory cell and its codomain as a register, a tabular relation from from $M \to N$ uses $M + N$ registers.

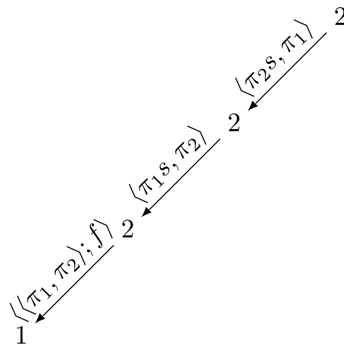
However, this notion of register is not fully adequate in the sense that a we may build a relation $R : 1 \to 1$ with left tabulation $s; s; s; s; s; s; s; s : 1 \to 1$. So far, the notion of diagram presented here tracks the memory usage of *shared variables* but we need to inspect the arrows in order to determine the amount of memory needed for a register.

We can solve this problem by introducing a notion of *memory diagram*, a diagram whose arrows are all "atomically representable" arrows, which indeed will exactly track the memory needs of the machine at any given state.

For instance, take the diagram:



Then, we may say that the register X_1 of the machine should be $X_1 = \langle \pi_2 ss, \pi_1\rangle; f$. We know we need to memory cells for the variables, but the amount of memory needed for storing the arrow is not known. We may use a different diagram for the same arrow:



Indeed, this new diagram tells us that we need 3 memory cells for the arrow $\langle\langle \pi_1, \pi_2\rangle; f\rangle$, one for f itself and two pointers, we need another 3 memory cells for $\langle \pi_1 s, \pi_2\rangle$, two pointers and the symbol s another 3 for $\langle \pi_2 s, \pi_1\rangle$ and lastly, 2 cells for the variables, totaling 11 memory cells.

Note that this notion of diagram is not minimal with respect to memory use for an already known term, we could represent the term with just 9 memory cells. However, it captures faithfully the memory behavior that happens in variable substitution.

Definition 9.28 (Memory arrow)**.** *An arrow is in* memory form *iff it is of the form* $\langle a_1, \dots, a_n \rangle$*, and every* a_i *is of the form* $\langle \pi_{i_1}, \dots, \pi_{i_m} \rangle; f$ *for* $f : m \to 1$ *an arrow generated by the signature of the program.*

A memory diagram is a diagram where every arrow is in *memory form*.

How does this new representation affect the pullback algorithm? We eliminate normalization for projections, at the price of adding a *deref* operation before comparing terms. Substitution is just composition or diagrams pasting. The *deref* operation is just the normalization for the projections.

A further enhancement to the concept of memory diagram is given by creating $\mathbb{N}$ copies of the object T — which stands for 1 in our diagrams — with their corresponding products. Then, the T_i object stands for the memory cell i, and the denotational model captures the instantiation of a variable as the variation of the tabulation domain from $(T_1 \times T_2 \times T_3)$ to $(T_1 \times T_3)$. This may yield a memory behavior closer to a standard WAM without garbage collection.

An actual implementation should be based in those main points:

- The interpretation of projections as pointers. Any π_i appearing inside an arrow is a pointer to a cell i.
- The interpretation of the relations domains as registers. The set of equations S used in the pullback algorithm should be understood as equations between registers $X_i \approx Y_j$.

We think the categorical model still leaves freedom for different design choices.

Inspired by Aït-Kaci's [Ait-Kaci, 1991] presentation of the WAM, the instruction set for an actual implementation could be designed following the structure of the machine:

(1) Instructions for comparison and substitution of terms. Register management. Used for pullback calculation.
(2) Instructions for register handling in predicate calls.
(3) Instructions for backtracking.

Thus, for a pullback of $\langle !_1; f, \pi_1 \rangle$ and $\langle a, b \rangle$, we will first emit instructions for such that $X_1 = !_1; f$, $X_2 = \pi_1$, $Y_1 = a$ and $Y_2 = b$. Then, we would emit `test` instructions, and depending on the result `put` or `fail`. We may adopt the WAM approach and omit the Y registers and use test instructions with and argument: `testc a, X1` and `testc b, X2`.

The most difficult part will be getting the optimizations right. A real compiler will feature two optimizations engines, the first one at the relational level able of equational reasoning and pullback computation, the second one at the instructions level mainly performing peephole optimization.

9.10. Related Work

Several categorical semantics for logic programming exist [Kinoshita and Power, 1996, Finkelstein et al., 2003, Amato and Lipton, 2001, Corradini and Asperti, 1992, Asperti and Martini, 1989, Amato et al., 2009]. The most important difference with our work is that all of them are based on a notion of *indexed category* and don't make a proposal for a concrete implementation. They use pullbacks to model renaming apart and unification, however, the use of allegory theory here helps us to internalize more of the logic and we get rid of functors, achieving a simpler model.

A different line of work is the interpretation of logic programming as functional programs. The most representative works are [Seres et al., 1999, Todoran and Papaspyrou, 2000, Brisset and Ridoux, 1993, Pirog and Gibbons, 2011].

In [van Emden, 2006], a similar effort to our semantics is developed, but the framework chosen is Tarski's cylindrical algebras instead Freyd's allegories. The author doesn't consider the implementation and efficiency of his approach.

In [Braßel and Christiansen, 2008], the authors study relational semantics for lazy functional logic programming language, modeling adequately the interactions between function call and non-determinism.

In [Bruni et al., 2001] the authors propose a diagram-based semantics for Logic Programming. An very interesting related work is [McPhee, 1995]. This is the only proposal that we know of for the use of tabular allegories in programming. Unfortunately, McPhee's work does not develop an executable model.

The use of category theory as a foundational tool for a machine is not new, functional programming languages have long used categorical machines [Guy Cousineau and Robinet, 1985, Cousineau et al., 1987].

Sketches and their associated theories [Barr and Wells, 1999] are closely related to our use of diagrams.

Wolfang Gehrke's thesis [Gehrke, 1995] also provides an interesting approach to computation in categories. It defines a rewriting system in order to prove the decidability of the free theory of a monad over a category.

Several approaches to virtual machine generation [Morales et al., 2005, Diehl et al., 2000] and compiler verification [Russinoff, 1992] for Prolog exist, but in our opinion they are not directly comparable to the work presented here.

This work was done taking no hint from existing abstract machines like the WAM. However, at a late stage of development we realized the categorical model shared some interesting similarities to the WAM, to the point we are seeing this work as being close to a formalization/derivation of a machine similar to the WAM's spirit.

Temporal registers of the WAM are in close correspondence with the notion of register presented here, and a particular implementation of the pullback algorithm may mimic closely WAM behavior. We understand those similarities as an interesting point regarding optimal designs for the efficient implementation of logic programming.

9.11. Conclusions and Future Work

Σ-allegories constitute a much richer semantic framework than the one strictly needed for interpreting and executing pure Prolog. In Chapter 10 we will explore how the categorical framework may be used to extend the classical logic programming model.

The first important benefit of our use of categorical concepts is the small gap from the categorical specification to the actual machine and proposed implementation. The algebraic connection between the different layers of the machine is preserved, reasoning in a layer is immediately reflected by the others. This allows us to reason using the a very convenient algebraic style, immediately witnessing the impact of any high-level change on the machine itself. In the other direction, a good example is the effect that memory layout have on incorporating T_i objects representing memory cells has on the base Lawvere Category itself. Our philosophy is that in a fully algebraic framework, efficient execution should belong to regular reasoning. Real world implementations usually depart from this view in the name of efficiency, one key objective of this work is to achieve efficiency without abandoning the algebraic approach. It is also worth noting that in our framework, we replace all the custom theory and meta-theory used in logic programming with category theory. The precise statement is that a Σ-allegory captures all the needed theory and meta-theory for a Logic Program with signature Σ, from set-theoretical semantics down to efficient execution.

The correctness of the machine is easy to check. Composition of relations together with the equation $R; (S \cup T) = R; S \cup R; T)$ capture in a simple way the operational semantics and memory layout of Prolog. Our framework is well suited to prove semantic properties, given that our semantics are compositional and use the well established frameworks of category theory and relation algebra. Third, the use of such frameworks favors the reuse of existing technologies in other areas of programming.

The second — and in our opinion, most innovative benefit — is the possibility of seamlessly extending Prolog using constructions typical of functional programming in a fully *declarative way*. In Chapter 10, we enrich the categories used here, adding algebraic data types, constraints, functions and monads to Prolog, all of it without losing source code compatibility with existing programs.

We are pursuing 4 big areas of future work. First, we'd like to explore some alternatives to our semantics foundations. The use of distributive tabular allegories instead of Σ-allegories is not out of the question, however the extension of the notion of RLC to PreLogoi comes at an important cost and added inner complexity of the associated categorical models.

For handling recursion, we'd like to depart from the external approach used in this chapter and provide a true internal fixpoint arrow in our semantics. We are studying some approaches like the one used in [Crole and Pitts, 1990] or [Eppendahl, 2003a].

Exploring the structure of constructions like $Map(Split(Cor(A)))$ is also in our agenda.

Second, we'd like to explore and generalize our notion of computation with diagrams. The choices made in this work have provided good results yielding a short and concise declarative specification.

However, alternative approaches lik Burroni's graphical algebras [Burroni, 1981], an in particular his equational characterization of equalizers [Lambek and Scott, 1986] may provide an interesting framework for our computational needs. The mail from Pierre Argeron[2] has more references about graphical algebras and related concepts.

After investigating more the semantics foundations, we plan to mechanize the machine and the compiler. In [Aameri and Winter, 2011], the authors propose a first-order encoding for allegories. This is related our previous relation rewriting approach and indeed we consider their work very useful for mechanizing our theory. An encoding of allegories in a dependently-typed programming language is presented in [Kahl, 2011]. We think Kahl's approach may help us to formalize and certify the compilation scheme presented here.

In this regard, we are thinking if approaches like[Morrisett et al., 1998] could serve us as inspiration for our work.

Finally, we are actively working on an instruction set. We don't want to tie it to an specific theorem proving strategy like SLD. Building a real-world efficient implementation of the machine presented in this chapter poses in itself a thesis, mainly for the effort involved in writing the optimizers, but we will likely develop a concrete instruction set proposal that can be used as a basis for the full development.

[2] http://www.mta.ca/~cat-dist/catlist/1999/graphical-algebras

CHAPTER 10

Extensions to Logic Programming in Tabular Allegories

In this chapter, we sketch some extensions for the framework defined in Chapter 9. The natural step is to profit from the richness of the categorical framework and enrich a regular Lawvere category with constraints, functions and types.

Although the extensions developed should be viewed as preliminary proposals to be further developed, this chapter serves two very important purposes for our work: it shows the *feasibility* of extending our framework, and what we think it is a crucial fact, it supports the *adequacy* of our framework for them. As the reader will see, most extensions are accomplished in a modular fashion and with minimal modifications to the existing framework.

When we took the decision to evolve the non-categorical framework to a category theory based one, one of our main objectives was to borrow semantic constructions from other paradigms such as functional programming.

On the other hand, categories are in general well suited for enrichment, and this is the way we incorporate constraints, data types and functions to the semantics. Given that the algebraic laws governing the execution of the machine are directly based on the laws holding in the semantics, the new computation rules follow in a straightforward way from the semantics enrichment.

We discuss each extension in its own section. First, we describe the intent of the extension, second we detail the semantic changes needed and analyze their impact on the execution engine.

In Sec. 10.1 we present constraints, in Sec. 10.2 we discuss algebraic data types, in Sec. 10.3 the topic of adding functions is handled, to close with a brief discussion of monadic constructs in Sec. 10.4. We discuss related work in Sec. 10.5 and close the chapter with conclusions and future work in Sec. 10.6.

10.1. Constraints

Constraint Logic Programming [Jaffar and Maher, 1994, Marriott and Stuckey, 1998] support is pervasive among Prolog systems. In Chapter 6 we presented a relational semantics for Constraint Logic Programming where constraint solvers were handled as complete black-boxes. However, when developing the categorical machine of Chapter 9, we did not follow the black-box approach and chose to investigate logic programming without constraints. The main motivation was to study the whole stack of the categorical machine, including how low-level details of a particular implementation should be handled. We also wanted to follow standard practice of abstract machines for Prolog, that usually handle terms and variables in a primitive way.

It turns out that enriching the categorical semantics and machine with constraint is straightforward. Categorically, a constraint is just a monic. Thus, we enrich the RLC notion with "opaque" objects representing the constraint store and monic arrows corresponding to constraints. A monic arrow is in close correspondence to subsets in the set-theoretic models of the RLC category.

We'll handle the operations of existential quantification and conjunction using facilities provides by standard Logic Programming, as it is commonly done in Prolog.

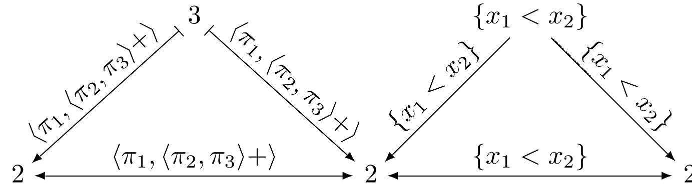
Definition 10.1 (Constraint Enriched Regular Lawvere Category)**.** *Given a constraint domain $\mathcal{D}$ and a set of primitive constraint predicates $\mathcal{CP}$, we enrich a Regular Lawvere Category $\mathcal{C}$ adding a set of arrows and objects representing all the constraint formulas that can be formed from primitive predicates*

$\mathcal{D}$. *For example, we write* $\{x_1 + x_2 = x_3\} : \{x_1 + x_2 = x_3\} \to 3$. *Note that we share the name of the arrow and the domain, as no confusion is possible. Note that the* x_i *notation refers to the* i*-th position of the co-domain type, as opposed to projections* π_i *which name elements of the domain.*

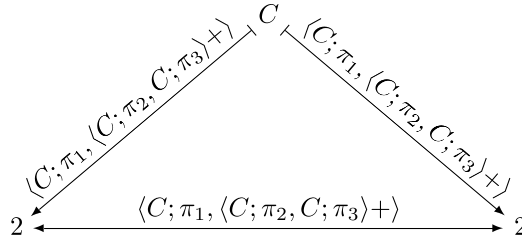
The initial model for a Constraint Enriched Regular Lawvere Category is a functor mapping the a constraint store to its corresponding subset, and the corresponding constraint formula to the corresponding inclusion monic.

Pullbacks between constraints correspond to the $\wedge$ operation, for instance the pullback of $\{x_1 + x_2. = .x_3\}$ and $(\{x_1 < x2\} \times id)$ is the constraint $\{x_1 + x_2 = x_3 \wedge x_1 < x2\}$.

Let's use the $\mathbb{N}$ domain as example, which gives rise to the Constraint Logic Programming over Finite Domains (CLPFD) paradigm. The constraint predicates `.=.` and `.<.` are common occurrence in such domains. In our category, we will write $\{x_1 =_{\mathbb{N}} x_2\}$ and $\{x_1 <_{\mathbb{N}} x_2\}$ for them, dropping the $\mathbb{N}$ subscript when it can be understood from the context. The constraint formula `X1 .<. X2 + X3` is compiled to:



The compilation scheme assumes that the constraint solver understands the term former $+$ from $\mathcal{T}_\Sigma$ as the addition in the domain, as it is standard practice in CLP. We extend the pullback algorithm of Sec. 9.6 to call the constraint solver when a constraint arrow is detected. In this example, the pullback algorithm will return a pullback with left component $C : C \mapsto 3 = \{x_1 < x_2 + x_3\}$ and with right component proprietary to the solver. After normalization, we obtain the diagram:



All the three variables are "generated" from C by the projections. Thus the constraint store C plays a role of a generator for the three variables involved. It is not possible to know their real value without resorting to C.

The categorical construction guarantees the sharing of the constraint store among all registers. Subsequent pullbacks will modify $C : C \to 3$ by composing it with another arrow $C' \xrightarrow{C'} C$.

The pullback algorithm is modified by adding the following rules:

$$\begin{array}{lcllcl}
C; \pi_i & \approx & \pi_j & \Rightarrow & (S' \mid S(j, C; \pi_i, h)) \\
C; \pi_i & \approx & !_N; a & \Rightarrow & (S' \mid S(C, tell(C; \pi, !_N; a), h)) \\
C; \pi_i & \approx & g; f & \Rightarrow & (S' \mid S(C, tell(C; \pi_i, g; f), h)) \\
C; \pi_i & \approx & C'; \pi_j & \Rightarrow & (S' \mid S(\{C, C'\}, tell(C; \pi_i, C'; \pi_j), h))
\end{array}$$

where $tell(C, C') = \{C \wedge C'\}$ or the initial arrow if the conjunction is not satisfiable. The auxiliary substitution function S is extended to handle references to constraint stores in h.

The categorical approach opens up very interesting possibilities with regard to native or half-native constraint solvers. Traditional problems of constraint solvers specification has the handling of variables are solved by the use of categories, thus we believe that defining a symbolic representation for C may

allow developing an algebraic theory for a some constraint solvers, in the same manner that we developed the theory for resolution.

For instance, we think that pure Prolog solvers like the one in [Gallego Arias, 2006] may be fully formalized and efficiently executed using this approach.

Indeed, we could see the pullback algorithm is very close to be a constraint solver for the Clark's equality theory.

10.2. Algebraic Data Types

Up to this point, Regular Lawvere Categories are built from the program's signature Σ, giving rise to the base type 1 — T from now on — which is interpreted by the initial model as $\mathcal{T}_\Sigma$, the set of all ground terms generated by Σ.

However, there's no obstacle in extending the base types from T to an arbitrary collections, as long as we keep the strictly-associative product requirement for them. We will follow an approach similar to the one used in [Bird and de Moor, 1996a], allowing the definition of recursive polynomial data types. For each new type base type A, we add a new object A and its corresponding *strictly associative* products to a Regular Lawvere Category. The type definition is seen as an extension to the signature and new arrows are adjoined, the existence of an initial algebra follows. Note that polynomial types exclude the definition of *higher order types* $T \to T$.

Two important points must be taken into consideration: as in AOP, types are always defined over the category of maps of a given tabular allegory. Recall that for an allegory $\mathcal{R}$ and a category $Map(\mathcal{R})$ the notion of product and coproduct don't coincide. We don't make claims such as $Map(\mathcal{R})$ has coproducts. Indeed, we will only claim that coproducts exists for selected objects based on the type definitions provided for the user.

Definition 10.2 (Polynomial Functor)**.** *A functor F built from constants, products and coproducts is said to be* polynomial*.*

Definition 10.3 (Type Declaration)**.** *A type declaration is a construction of the form:*

:- **data** $Type$ = C_1 | ... | C_n.

where each C_i is of the form $f(Type_1, \dots, Types_m)$.

Some examples are:

```
:- data Nat   :- o | s(Nat).
:- data TermN :- l(T, Nat).
:- data List  :- nil | cons(T, List).
```

Now we define the effect of type declarations in the semantics.

Definition 10.4 (Type Completion)**.** *For a type declaration $Type = C_1 \mid \cdots \mid C_n$ we enrich the generating Lawvere Category of the Program as follows:*

- *We add a new object $Type$.*
- *For each $C_i = f(Type_1, \dots, Type_n)$, we add an arrow $f : N \to Type$.*
- *We add an arrow $cast_{Type} : Type \to T$.*
- *We define a polynomial (bi,tri) functor $F_{Type} : \mathcal{C} \to \mathcal{C}$ representing the structure of the type.*

We assume the data constructors occurring in type definitions to be disjoint. Otherwise the compiler should emit a name-clash error.

For instance, the Nat example will add the object Nat and the arrows $o : 0 \to Nat$ and $s : Nat \to Nat$. The functor F_{Nat} is defined on types as $F_{Nat}A = 0 + A$ and on arrows as $F_{Nat}f = [id_0, f]$. The initial algebra for Nat is $\alpha_{Nat} : F_{Nat} \to Nat = [o, s]$.

$$\frac{\Gamma \vdash R \Rightarrow S : T}{(\overline{p} : T) \in \Gamma \vdash \overline{p} = R \Rightarrow \overline{p} = S : T} \qquad \frac{\Gamma \vdash R \Rightarrow R : T \qquad \Gamma \vdash R \Rightarrow R : T}{\Gamma \vdash R \cup S \Rightarrow R' \cup S' : T}$$

$$\frac{\Gamma \vdash R \Rightarrow R' : T_1 \to T \qquad \Gamma \vdash S \Rightarrow S' : T \to T_2}{\Gamma \vdash R; S \Rightarrow R'; S' : T_1 \to T_2} \qquad \frac{\Gamma \vdash R \Rightarrow R' : T_2 \to T_1}{\Gamma \vdash R^\circ \Rightarrow R'^\circ : T_1 \to T_2}$$

$$\frac{\Gamma \vdash f_1 \Rightarrow f'_1 : T_0 \to T_1 \qquad \dots \qquad \Gamma \vdash f_n \Rightarrow f'_n : T_0 \to T_n}{\Gamma \vdash \langle f_1, \dots, f_n \rangle \Rightarrow \langle f'_1, \dots, f'_n \rangle : T_0 \to (T_1 \times \dots \times T_n)} \qquad \frac{}{(\overline{p} : T) \in \Gamma \vdash \overline{p} \Rightarrow \overline{p} : T}$$

$$\frac{}{(f : T \to T_f) \in \Gamma \vdash f \Rightarrow f : T \to T_f} \qquad \frac{}{(c : 0 \to T) \in \Gamma \vdash !_M; c \Rightarrow !_{T_o}; c : T_o \to T}$$

$$\frac{N = (T_1 \times \dots \mathbf{T}_i \dots \times T_n) \qquad N' = (T_1 \times \dots T_i \dots \times T_n)}{\Gamma \vdash \pi_i^N \Rightarrow \pi_i^{N'} : N' \to T_i}$$

FIGURE 10.1. Jugdments for Type-Checking and Arrow Adjustment

The initial model of the Regular Lawvere Category is extended to interpret data types to the corresponding subset of $\mathcal{T}_\Sigma$. The arrow $cast_{Type}$ is then the inclusion of the signature of the type inside the general signature of the program. We may annotate predicates with types:

Definition 10.5 (Type Annotation)**.** *A type annotation is a construction of the form:*

`:- pred M :` $(Type_1, \dots, Type_n)$`.`

If a predicate p *is not annotated it is assumed to be of type* T^N*, where* N *is the arity of* p*.*

Type Checking: We type-checking could be defined over the Prolog source code, as in [Hill and Lloyd, 1994, Mycroft and O'Keefe, 1984].

However, we want to profit from the categorical framework and extend our approach to type-inference, so the procedure we present uses the unmodified translation procedure of the previous chapter to produce ill-typed arrows as the result of the translation.

Then, the judgment system in Fig. 10.1 checks and rewrites ill-typed arrows until we get a well-formed arrow or a type-checking failure.

A judgment $\Gamma \vdash R \Rightarrow S : T$ means that S has type T and that S was replaced for R in the original arrow. In the rules, we write $\mathbf{T}$ for the type base type interpreted as $\mathcal{T}_\Sigma$. The main action for fixing arrows is adjusting the types of the projections to match their intended type. The environment Γ is built from type declarations and annotations, and checked for conflicting data constructors.

Type Casting: For every type A, a *casting* predicate $cast_A : (A \times T)$ exists, whose only purpose is to allow the programmer to call existing predicates with the representation of a term. Operationally, the cast predicate is equivalent to = and its tabulation is based on the $cast_A$ arrow.

Operational Behavior: Operational semantics and pullback algorithms don't need any modification, as type-checking is performed at compile time.

10.3. Functions and Functional-Logic Programming

The basic primitive in categorical semantics is the arrow, which is seen as a function in many categories, where arrow composition corresponds to function composition. Indeed, a models for a RLC

is a functor to Set, where arrows stand for functions, so it is a natural step to enrich our semantics with functions.

In order to illustrate some of the advantages that category theory brings, we briefly compare our proposal with other attempts to incorporate functions to Prolog whose semantics don't fully capture the intended addition.

A good example may be the addition of functional notation to Prolog implemented by source code transformation, as defined in [Casas et al., 2006]. In that approach, a function

```
f(a) := b
```

is translated to the predicate

```
f(a,b).
```

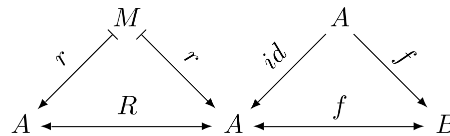
`f` has a formal semantics as a predicate, but that semantics doesn't formally capture the fact that the it is a function, that is to say, $\forall X.\exists! Y.f(X, Y)$. On the other hand, analyzers and compilers are missing the information that `f` is a function, and cannot optimize accordingly. Some paradigms like Functional Logic Programming address some of the semantics problems, but they usually come at the cost of losing compatibility with existing Prolog programs.

In the semantics level, the first question to explore is what exact notion of function do we want? If we do not want to extend our notion of model, a function in our program will correspond to an arrow in Set, which represents a total function.

We will maintain the computation theory of the function separate from the categorical theory in order to simplify the presentation, but as long as the reduction theory is algebraic, extending a Regular Lawvere Category to capture it would pose no problems to our semantics.

In this proposal we omit exponential objects, limiting functions to range over the polynomial types defined in Sec. 10.2. Making our category cartesian closed is something we cannot do straightforwardly. First, the undecidability of higher-order unification [Goldfarb, 1981] means that our pullback procedure would become very complex, and of course, undecidable. However, in our opinion a restricted notion of exponential object is desirable and we have plans to incorporate it. We discuss our ideas in Sec. 10.6.

Note that the semantics accommodates composing functions directly with predicates:



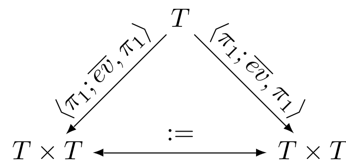
but we don't have plans to define a syntax for that. However, using our semantic framework for Functional Logic Languages like Curry [Braßel and Christiansen, 2008, Hanus et al., 2003] and Toy [López-Fraguas and Sánchez-Hernández, 1999] seems appropriate, although we need to define our restricted notion of exponential object prior to obtaining an allegorical semantics for them.

We first discuss the addition of total computable functions, to proceed to discuss the use of partial computable ones.

Syntax: Functions are incorporated to programs using a new `:=` predicate. For instance, given the (primitive) function $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ we may write:

```
p(X) :- Y := +(3,3), q(X,Y).
```

The tabulation of the `X := f` predicate is



	Yes	**No**
Instantiating	Narrowing	Residuation
Call by Value	Strict	Lazy

FIGURE 10.2. Approaches to Function Reduction in the Presence of Free Variables

$$\begin{array}{llll} h; \langle f, g\rangle & \to_R & \langle h; f, h; g\rangle & \\ \langle f, g\rangle; \pi_1 & \to_R & f & \\ \langle f, g\rangle; \pi_2 & \to_R & g & \\ f; !_N & \to_R & !_M & f : M \to N \\ f; \overline{ev} & \to_R & e & f \to_! e \end{array}$$

FIGURE 10.3. Extended Arrow Normalization for Strict Evaluation

where $\overline{ev}$ is a new special operation $\overline{ev} : T \to T$ marking the term as a function. Models identify this arrow with the semantics of the identity arrow.

The translation of p is then:

$$I_{13}; \langle \pi_1, \pi_2, \langle !_2; 3, !_2; 3\rangle; +\rangle; (id_1 \times :=); W; \overline{q}; W^\circ; I_{13}^\circ$$

What role does $\overline{ev}$ play in the pullback algorithm? Indeed, depending where we plug in its handling rules we will obtain a lazy or strict evaluation strategy. For the rest of the section we assume that our functions are partial computable ones. In this case, the evaluation strategy [Plotkin, 1975] and the behavior of the functions in the presence of free variables [Antoy et al., 1994, Bonnier and Maluszynski, 1988]. Fig 10.2 become very important choices, as the same program will yield different results under different strategies. Fig. 10.2 illustrates the different choices available. A residuating strategy involves significant challenges and we won't consider it here.

How does a model look like when partial functions are evaluated under a strict strategy? The functors now won't target set, but the category of pointed sets, where each object has a distinguished element $\perp$ and composition of arrows is strict. Predicates remain modeled as subobjects of this pointed category. If we use a lazy strategy, the target category for models would be a category with objects CPO and arrows continuous functions.

Operational Considerations: In order to support lazy and strict instantiating strategies, we need to modify the pullback algorithm to specially handle the function marker $\overline{ev}$.

- **Strict**: Assume a strict evaluation relation for our functions $\to_!$. Then, we extend the rewriting system used for arrow normalization such that when a function is found, it will evaluate it. The use of := triggers arrow normalization, obtaining an strict operational behavior. The modified theory can be seen in Fig. 10.3.

  For the example given previously, the state of the machine before the call to q would be:

  $$(\langle \pi_1\rangle | \langle (3+3), [\langle \pi_1, 6\rangle; \overline{q}]\rangle; W^\circ; I_{13}^\circ$$

  In the implementation, the operator := will be compiled to a special instruction. It is not needed to construct a term, calling the functional evaluator is enough.
- **Lazy**: We assume a small-step relation $\to_L$ such that for a function definition like $ones = f(1, ones)$ then $ones \to_L f(1, ones; \overline{ev})$. In this case, the normalization theory is not modified and := wraps a function f to a term $f; \overline{ev}$.

We modify the pullback algorithm to call $\rightarrow_L$ whenever it needs to check equality for a term $f;\overline{ev}$. The following rules are added:

$$\begin{array}{lcllcl} f;\overline{ev} & \approx & \pi_i & \Rightarrow & (S' \mid S(i, f;\overline{ev}, h)) & \\ f;\overline{ev} & \approx & f' & \Rightarrow & (\{g \approx f'\} \cup S \mid h) & \text{if} \quad f \rightarrow_L g \wedge f' \neq \pi_j \end{array}$$

In this case, the state of the machine prior to the call to $\overline{q}$ would be:

$$(\langle \pi_1 \rangle | \langle (3+3), [\langle \pi_1, \overline{ev}(3+3) \rangle; \overline{q}] \rangle; W^\circ; I_{13}^\circ$$

10.4. Monadic Constructions

Our categorical model allows us to experiment by adding any particular structure to our semantics. A very interesting possibility is the notion of monad [Eckmann, 1969], whose application to computer sience [Moggi, 1991], in particular to functional programming [Wadler, 1995] in order to differentiate computations from functions has been very successful.

Our intention when using monads in logic programming is to allow the differentiation of a logical predicate from a *computational* one, that is to say, a predicate whose proof may possibly involve some side effects. Monads provide a representation for side effects within the logical model itself, in a structured manner.

Without the use of exponential types in our categorical framework, very interesting monads are left out. However some monads whose functor component is polynomial are still useful, and we think that the discussion below will serve as a very useful reference for future work.

A monad over a category $\mathcal{C}$ is a triple (T, η, μ), where $T : \mathcal{C} \rightarrow \mathcal{C}$ is a functor, $\eta : \mathsf{Id}_{\mathcal{C}} \rightarrow M$ (also named return) and $\mu : T^2 \rightarrow T$ (also named join) are natural transformations such that the following diagrams commute:

$$\begin{array}{ccc} T^3A & \xrightarrow{\mu_{TA}} & T^2A \\ {\scriptstyle T\mu_A}\downarrow & & \downarrow{\scriptstyle \mu_A} \\ T^2A & \xrightarrow[\mu_A]{} & TA \end{array} \qquad \begin{array}{ccccc} TA & \xrightarrow{\eta_{TA}} & T^2A & \xleftarrow{T\eta_A} & TA \\ & {\scriptstyle id_{TA}}\searrow & \downarrow{\scriptstyle \mu_A} & \swarrow{\scriptstyle id_{TA}} & \\ & & TA & & \end{array}$$

Composition of monadic arrows $f : A \rightarrow TB$ and $g : B \rightarrow TB$ is defined as $f;^* g : A \rightarrow TC = f; Tg; \mu$. This composition operator $;^*$ is called Kleisli composition. Informally, we may say that a monad is a *decoration* of the codomain of an arrow. For any $f : A \rightarrow B$, there exists a decorated version $f; \eta_B : A \rightarrow TB$. The operation $;^*$ is then interpreted as the threading of a result through a computation.

What is the relationship to our framework? In our setting, predicates are translated to arrows in an allegory, also named relations. So far, we have interpreted predicates as coreflexive relations, so for a predicate p of arity N, the corresponding relation $\overline{p}$ has type $N \rightarrow N$. Enriching our base Lawvere Category with a monad T, we may construct predicates with type $N \rightarrow TN$ and replace conjunction — that we interpret as composition — by the Kleisli composition.

Let's see an actual example of a monad in our framework. The first component of a monad is a functor T. In Sec. 10.2 we defined polynomial functors. Can η and μ be defined for some of them? The answer is yes. For instance, the functor $Maybe\ A = 0 + A$ — recalls that 0 is the terminal object — can be readily made into a monad with natural transformations $\eta_A(f) = f; inr_A$ and $\mu_{MaybeA}(f) = [id_0, f]$. Another interesting monad is the Writer monad. Assume a list data type $[T]$, then $Writer\ A = A \times [T]$. That is to say, we lift any type A to the product of A with a list of terms. We don't support user-defined monads, for we lack the necessary type system. The compiler has to provide them in a primitive way, but we show a sketch of what a user defined version of the Writer would look like:

```
:- data Writer a = w (a × [Term]).
:- fun return_w : a -> Writer a
:- fun bind_w : Writer (Writer a) -> Writer a
```

```
return_w(a)             ::= w(a, []).
join_w(w(w(a, L1), L2) ::= w(a, L1 ++ L2).
:- fun log_w : Term -> Writer <>
log_w(X) ::= w(<>, [X]).
```

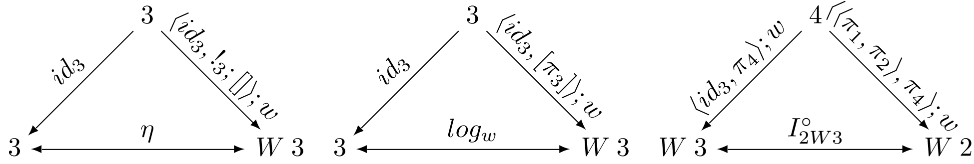
Then, we may define a predicate using the monadic facilities:

```
:- pred do : Term -> Term.
do(a). do(b).
:- pred p : (Term × Term) -> Writer (Term × Term).
p(X,Y) :- do(X), do(Y), log(m(X,Y)).
```

The categorical translation is:

$$\overline{p} = I_{23}; \langle \pi_1, \pi_2, \langle \pi_1, \pi_2 \rangle; m \rangle; W_1; \overline{do}; W_1^\circ; W_w; \overline{do}; W_2^\circ; \eta;^* ; log_w; I_{2W3}^\circ$$

where $W\,3 = (3 \times [Term])$ and the non-trivial arrows are tabulated by:



Using monads and types, we may structure Prolog programs allowing significant more code reuse and abstracting from the underlying implementation of data types. The maybe monad may be very useful for predicates that want to fail in a controlled fashion, whereas the reader and writer patterns turn global read-only state and logging into modular and declarative features of Prolog.

Unfortunately, the state monad [1] $StateA = (S, A)^S$ is not a polynomial functor. If we were to extend the semantic framework, our pullback algorithm would face the problem of equalizing two arrows $f : C \to (S, A)^S$, $g : C' \to (S, A)^S$. We know of strategies for a first order encoding of $(S, A)^S$, which would allow us to implement the state monad, but we feel that resorting to it would depart from our fully-declarative spirit.

Despite the lack of an implementation strategy for monads whose functor lifts to an exponential type, we can use them at the semantic level to reason about logic programs involving computations and side effects within our allegories.

10.5. Related Work

There is a huge body of related work on most of the topics discussed in this chapter. The main difference that the approach presented here has with many other approaches trying to integrate all those features in Logic Programming is the use of a new semantic framework (Σ-allegories) which *natively* supports them, and in addition, provides a clear guidance on their implementation.

Languages such as Functional Logic ones [Hanus et al., 2003, López-Fraguas and Sánchez-Hernández, 1999, Somogyi et al., 1996], Higher-Order ones [Miller et al., 1991, Nadathur and Miller] and purely logic ones [Hill and Lloyd, 1994] break source compatibility with Prolog. In particular, λ-Prolog has a polymorphic type system [Kwon et al., 1994] and higher-order support, allowing the implementation of some monads [Bekkers and Tarau, 1995].

Semantics for monadic predicates and data types in algebraic models for logic programming that are based on indexed categories are discussed in [Lipton and McGrail, 1998, Amato et al., 2009].

Ciao Prolog [Hermenegildo et al., 2011, Bueno et al., 1999] shares our philosophy of preserving compatibility with existing Prolog programs. Their use of program analysis and transformation is very

[1] the IO monad is isomorphic to a state monad whose state is the whole world.

interesting and it supports functions [Casas et al., 2006], a syntactic notion of higher-order programming [Cabeza et al., 2004] and regular types (similar to our polynomial functors). All those features are implemented using desugaring to Prolog, so the underlying execution engine remains the WAM. A new compiler for Ciao-Prolog that uses the analysis information generated by the Ciao's preprocessor to optimize the compilation process is being developed [Morales et al., 2006].

The approach used in [Bird and de Moor, 1996a] seems like a good path for extending our data-type support.

Exploiting the relation between monads, continuations and Lawvere Theories [Hyland and Power, 2007, Hyland et al., 2006], the effects-handler style or programming proposed by Eff [Bauer and Pretnar, 2012, Plotkin and Pretnar, 2009, 2008] seems to be better suited to our semantic framework and Logic Programming than Monads, which require exponential types and a notion of lazy evaluation.

Our framework also supports being extended with Deduction modulo [Dowek et al., 1998], an approach where deduction (and in this case resolution) is performed with respect to an equational theory.

10.6. Conclusions and Future Work

.

We have showcased some examples on how the categorical machine and algebraic framework of Section 9 can be extended with declarative features. Constraint support, polynomial data types and functions are presented in what we think is a mostly definitive form. Some design choices were made based on our own experience with Prolog, but we are open to modifying them once we have a real system and feedback from programmers arrives.

Regarding the constraints extension, the straightforward support for objects representing constraint domains like $\mathbb{N}$ or $\mathbb{R}$, and the way our semantics captures metatheoretical issues like variables and substitution favors the development of algebraic specifications of constraint solvers. The modeling of the full operational and denotational behavior of a constraint solver within our categorical semantics seems feasible.

The incorporation of algebraic data types was done with two clear guidelines. First, we wanted to remain fully compatible with standard Prolog programs, feature which needs the ability of mixing typed predicates with untyped ones. We must support primitive types too. Second, we wanted the types be the closest possible to the categorical formalism for data types pioneered in functional programming.

Given the versatility of the base data type T, we may consider that our proposal includes a restricted form of non-typed polymorphism; but we sought for proper polymorphism based on the Girard-Reynolds isomorphism [Wadler, 2007, Girard et al., 1989].

Integrating higher-order types (i.e: exponential objects) is the most important priority for future work. Due to semantic considerations, we must be careful in order to avoid problems with inconsistency in the allegories used here. Division or power allegories are a candidate to be explored. Another interesting possibility in the data type field is the automatic generation of type checking/instantiating predicates as Ciao-Prolog does. In Ciao, a type predicate serves two purposes: type-checking and element-generation, depending on a call with ground or free arguments. We think that initial algebras will allow us to internally model this behavior. Supporting coalgebraic constructions [Komendantskaya and Power, 2011b,a, Jacobs and Rutten, 1997] and types is also a possibility for future work.

For function support, our main line of work is the precise definition of the models for both strict and lazy functions. Next, we plan to add an internal notion of function reduction. This enrichment will go together with higher-order types, and surely implies that the machine will gain a new primitive beyond relation composition: application. We are exploring what would be the best syntax to incorporate functions to Prolog without altering the “flavour” of the programs so much that we have created a new language.

An important challenge is the integration of residuating strategies. In residuation, we function reduction is delayed until all their arguments are ground. While the categorical machine itself doesn't

have a problem to support this execution model, the denotational models for residuation are unknown. We think that discussing how support them on our machine gives an interesting example of how we can model less "pure" features. We may use a new definition for `:=` such that for `X := f(Y)`, we assign $\overline{r}(X, f(\overline{v}(Y, f(X))))$ to X and substitute $\overline{v}(Y, f(X), X)$ for Y. Note that this substitution is performed globally. So now, our variables may be of the form π_i or $\overline{v}(Y, f(X), Y)$. We need to may modify both our pullback algorithm and arrow normalization to account for the new representation of variables.

In the implementation, we may accordingly replace memory cells representing variables with a compound structure such that when a variable is instantiated it will check if the variables affecting the function are already instantiated.

Monads whose functor component is a polynomial one are fully defined in this paper. However, as we have seen in Sec. 10.4, the monads whose functor performs a lifting to an exponential object — like IO and State — crucially depend on the incorporation of a true applicative notion to our model. In the spirit of retaining a "logical" style of programs, we are wondering what syntactic sugar would be a good match to the **do** notation used in Haskell.

Meta predicates like `var` and cut (!) may be added to our operational model very easily. In the first case, `var` would consist checking if an arrow is a projection, where cut would discard elements of a union. However, the standard model for our enriched Lawvere Category doesn't support these notions. We are studying what would be an appropriate notion of model in order to support these features in a declarative way.

CHAPTER 11

Evaluation

The work presented in this thesis is part of a wider initiative started in the early nineties to develop relational semantics for logic and functional programming languages. Several key features distinguishes our work:

- **The focus on real world applicability, efficiency and compiler construction:** Even when working in deep theoretical issues, we never stopped wondering about the impact of any change on the actual implementation.
- **The logical interpretation of the relations:** We never abandoned the philosophy advocated by Tarski and Lipton in the equipollence theorem: the one to one correspondence of a coreflexive relation and a first order formula.
- **The use of constraints as core piece:** The use of constraints simplified the presentation of our theory, allowing us to compartmentalize quantifier elimination and proof search in the semantics. This significantly eased the proofs and allowed us to present a much clearer picture of the general scheme.
- **The use of executable semantics:** We strictly adhered to the principle that execution should belong to regular reasoning. At some moments, we were tempted to abandon this approach, specially when technical problems with the quasiprojections arose, but in the end our choice to stick to this design choice payed great benefits.

The contributed relational semantics, including the operational part, turn out to be what we hoped for, but efficient execution is not possible without resorting to — for us — unacceptable use of additional meta-theory. Formalization of variables by quasi-projections implies that the number of variables in use cannot be captured by the theory, condition *sine qua non* to achieve a *declarative* and simultaneously *efficient* executable semantics. Thus, the original objective of using the theory of distributive relational algebras in order to construct an efficient machine for Constraint Logic Programming couldn't be achieved.

However, that shortcoming had a huge positive impact: it forced us to replace quasi-projections by projections, a simple change with deep implications. Ever relations became typed, and we ended with an equational system almost equivalent to allegory theory. The work in categorical semantics was very productive, and we obtained better results than we originally hoped for when switching to category theory. Given that our expectations were surpassed we believe this work lays a solid foundation upon more work can and should be performed.

Finding the right categorical framework for Constraint Logic Programming that would meet our requirements was a hard task. But once achieved, we were surprised by its elegance and good properties. Algebraic Data Types and strict and lazy functions were seamlessly added, with little or nil impact on the original algorithm and categories. We just needed to add the needed structure and rules in a modular fashion and everything fitted together perfectly.

The preliminary instruction set we derived was surprisingly close to the WAM! We consider that as strong evidence that our design choices are the right ones with regards to efficiency. Some time ago, we hoped that the efficient execution model derived from the executable semantics would provide a radically different machine than the WAM. The opposite happened.

This work ends at the gates of defining a compiler which should be fully competitive with more mature ones. The big missing part is the optimization stage. We realized that the final design of a particular instruction set inspired by this semantics couldn't be done without taking into account the optimization stages. In light of that, we choose to stop the work of this book here. However, we have built several prototypes of a compiler and machine and have gained significant experience towards future work.

How the relational calculus can help in program transformation and optimization was comprehensively studied in [Bird and de Moor, 1996a], but in our opinion, building a real program optimizer using the calculus of relations won't be easy. Our work constitutes a step in that direction, but the full work is enough material for another dissertation.

We comment on the technical contributions:

- **Relational theory for the formalization of Constraint Logic Programming:** The relational formalization of Constraint Logic Programming is elegant and has attractive properties. However, computation over it is more complex than wanted and we prefer the categorical approach used in the second part. On the other hand, the work carried out here brought some ideas on how to develop a modified notion of rewriting that could fix the main problem.
- **Translation procedure for Constraint Logic Programs:** The translation procedure is straightforward and is the right base to build a verified compiler.
- **Algebraic operational semantics for SLD:** We should explore how to extended the presented semantics with algebraic constraint store operators.
- **Rewriting system for SLD proof search:** The definition of the rewriting system includes a few ideas on how to handle lack of confluency when rewriting relations, one of the main problems when trying to compute in a relational theory.
- **Proof of operational equivalence:** The proof is very detailed and a candidate for mechanization.
- **Algorithm for first order unification:** The work here is very interesting and very good example of the problems faced when trying to compute in the pure relational calculus. The most revealing discovery of the work is the use of two notions of normal form for terms, one for outlining properties of the domain and the other for the "inspection" of the codomain. This is in contrast with standard calculi that enjoy a unique notion of normal form.
- **Regular Lawvere Categories:** The concept of Regular Lawvere Categories is an idea of Prof. Lipton, further developed by me. They are a simple, but extremely useful notion given the regular nature of the category of maps of a tabular allegory.
- **Σ-Allegories:** Instead of requiring tabular distributive allegories for the semantics of CLP programs, we built the weakest categorical structure needed for an adequacy theorem. The partial-tabular nature of Σ-allegories constitutes an interesting notion from the logical point of view and is an example of "just the required structure" approach.
- **Translation procedure to Σ-allegories:** The translation of a CLP predicate to an arrow is an interesting example of how to profit from the denotational semantics for program optimization and simplification. Using the relational laws, we get rid of any intersection in a translated predicate, allowing our allegorical machine to be based just on relation composition and greatly simplifying it.
- **Notion of computing using categorical diagrams:** The jump from the mathematical world of category theory to the world of computation involves significant challenges. Fully-equational representation of categories is not always possible, or involves complex tricks. Simple categorical identities like $id; f = f$ may have extreme computational importance, and the diagrams for $id; f$ and f are completely distinct from a computational point of view.
- **Algorithm for pullbacks computation in Regular Lawvere Categories:** The algorithm is an elegant and efficient implementation of unification with renaming apart. In the case of

Regular Lawvere Categories, the proof of correctness is easy, but for more complex structures, universal properties of diagrams will play a fundamental role. A very interesting point is how the specification of the algorithm plus a specific interpretation of arrows and objects practically leads to a canonical instruction set.

- **Categorical Machine for CLP:** The most interesting bit of the categorical machine is the idea that a state corresponds to a diagram. Then, thanks to reasoning with diagrams we are able to use a convenient representation, closely matching the standard operational semantics for CLP. The proof of equivalence and the specification of the machine itself becomes much simpler.
- **Notion of "memory" diagram:** As originally defined, the categorical machine fully captures the number of free variables in a particular state of program execution, but memory requirements for the terms itself are not. By imposing a normal form condition over the diagrams, we obtain states that fully capture the memory layout of the real machine. The downside here is that we must add a *deref* operation to the pullback algorithm.
- **Algebraic Data Types:** The addition of algebraic data types was straightforward and didn't pose a particular challenge. However, in order to remain compatible with Prolog we had to develop some type inference. Instead of working with a representation of the original CLP program, we developed our type inference specification using a notion of pre-arrow, where composition may not be well defined. Then, the type inference engine tries to rewrite a pre-arrow in order to reach a real arrow. If it is not possible to form and arrow, the algorithm fails and the program doesn't type check.
- **Strict and lazy functions:** It was very surprising how few changes we had to make in order to accommodate functions into our framework. We think this contribution is in need of more development, we accordingly comment on the future work section.

What would I have done differently? I would use the categorical approach from the start, but the failure of the untyped relation algebra to provide an efficient execution model was almost impossible to predict. If I were to start today a project similar to this I would plan and design it so that all the proofs and definitions would be carried out in a theorem prover like Coq, Isabelle or Agda.

11.1. Scientific Results

The work of this thesis has produced the following papers. I am the principal contributor in all of them.

- *Short Technical Report*:
  Emilio Jesús Gallego Arias, James Lipton, and Julio Mariño. Extensions to Logic Programming in Tabular Allegories. *Technical Report, May 2012*
- *Journal paper*:
  Emilio Jesús Gallego Arias, James Lipton, and Julio Mariño. Logic Programming with a Relational Machine. *Submitted.*
- *Conference paper*:
  Emilio Jesús Gallego Arias, and James Lipton. Logic Programming in Tabular Allegories. *Accepted. Technical Communications of the ICLP 2012, LIPICS series.*
- *Journal paper*:
  Emilio Jesús Gallego Arias, James Lipton, Julio Mariño, and Pablo Nogueira. First-order unification using variable-free relational algebra. *Logic Journal of IGPL*, 19(6):790-820, 2011.
- *Conference paper*:
  Emilio Jesús Gallego Arias, Julio Mariño, and José; María Rey Poza. A generic semantics for constraint functional logic programming. *In Proc. of the 16th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2007)*, 2007.

- *Conference paper*:
  Emilio Jesús Gallego Arias and Julio Mariño and José María Rey Poza A proposal for disequality contraints in Curry *Electr. Notes Theor. Comput. Sci., 177:269-285*, 2007.
- *Conference paper*:
  Emilio Jesús Gallego-Arias and Julio Mariño. An overview of the Sloth2005 Curry system. *In Michael Hanus, editor, Workshop on Curry and Functional Logic Programming*, ACM press, 2005.

In addition to publication, the work carried out in this thesis has been part of the following research projects:

- **DESAFIOS10** (Desarrollo de software de alta calidad, fiable, distribuido y seguro) Reference: TIN2009-14599-C03-00.
- **PROMETIDOS** (PROgrama de METodos rIgurosos de DesarollO de Sofwtare de la Comunidad de Madrid) Reference: P2009/TIC-1465.
- **DESAFIOS** (Desarrollo de Software de Alta Calidad, Fiable y Seguro) Reference: TIN2006-15660-C02-02.
- **PROMESAS** (Programa en métodos para el desarrollo de software fiable, de alta calidad y seguro) S-0505/TIC/0407.
- **SOFFIE** (Software Fiable y Extensible de Alta Calidad) Reference: TIC2003-01036

11.2. The Prototype Implementation

The key goal of this work was to build a real-world compiler. Indeed, two prototypes were built during the development of the theoretical work. Both compilers are written using Haskell and are directly implement the translation and operational semantics defined in this book. They are available online: `http://babel.ls.fi.upm.es/projects/ram`.

The first prototype was based on the pure relational calculus version and helped to laid out the final treatment of constraints. However, soon it became clear that some typing information was needed in order to build an efficient implementation.

The second prototype was a modification of the first one to use categorical arrows as intermediate code. However, soon after work started that prototype was deemed obsolete by the implementers. As good as Haskell is as a practical declarative programming language, we have no hope of building a fully certified Prolog compiler using it.

With this new requisite in mind, we have started to implement a new compiler from scratch in the dependently-typed programming language Agda, that not only directly supports all the definitions and proofs used in this work, but indeed is able to certify its translation procedure by including a version of the equipollence theorem. Indeed, when a logic program is compiled by this compiler, both a relational term and a proof of logical equivalence is produced.

While the two previous prototypes may bear some historical interest and may be useful for students or Haskell programmers, we do not find them worthy of research interested more than mentioning their existence and the crucial role the feedback played in the development of this work.

11.3. Future Work

We may distinguish four lines of further work. First, we think there are open questions about computing modulo the theory of distributive relation algebras. The work here proves that rewriting, or even other forms like [Dougherty and Gutiérrez, 2006] are difficult to control in the light of such a rich theory. However, the use of deductive, constraint or uniform systems may allow a much better control over relational computation. In all cases, efficient computing of constraint solving algorithms will require some form of additional meta-theory beyond equality, but it may be well the case that this enrichment is worthwhile and can be considered declarative.

As a second line of future work, the categorical semantics may be improved in several ways. Regarding the structure of the Σ-allegories, internalization of the fixpoint operator is a priority. Extension of our Regular Lawvere categories to Pre Logos is open work, and will make Σ-allegories tabular.

Full support for Monads and Higher-Order functions will require a notion of exponential object. This will allow us to fully interpret Functional Logic Programming in our framework. However, such extensions must be done with care, as naively adding exponential types may create consistency problems. A notion of exponential object will undoubtedly extend the categorical machine beyond relation composition. A notion of application will be needed.

The use of impure features in LP is widespread. The theory of allegories has no problem supporting them, at the expense of losing the standard set-theoretical models. Metapredicates like `var`, if directly accommodated will break the associativity of composition. The categorical approach of enriching our category with a non-associative notion of monoid may be the solution to model these kinds of primitives. Then, the image of a functor acting as a model of our $\Sigma - \otimes$-categories would a non-associative algebra or tree. A similar approach may be used for denotationally modeling difficult features like residuation.

This work serves as use example of the internal logic of categories for programming language metatheory formalization. Building a logical framework based on such internal logic may be a worthwhile goal.

Extending the paradigm of Constraint Logic Programming constitutes the third line of future work. By allowing the programmer some access to modify the underlying theory, custom definition of proof search strategies by algebraic laws may be incorporated into Prolog. A study of what particular extensions would suppose a greater benefit for the Prolog programmer is not well known. Syntax for the extensions needs more work.

The last line of work is fully centered on the categorical machine and compiler. While the execution model presented here is quite efficient, more work is needed to be really competitive with real implementations. The derivation of the instruction set from the categorical algorithm must be formalized and proven correct. Two optimizers must be designed and implemented: a relational optimizer and a peephole optimizer for instructions, which will generate abstract machine code competitive with WAM-based implementations.

Bibliography

Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990, 1990. IEEE Computer Society. ISBN 0-8186-2073-0.

Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3183-0.

Bahar Aameri and Michael Winter. A first-order calculus for allegories. In Harrie de Swart, editor, *Relational and Algebraic Methods in Computer Science*, volume 6663 of *Lecture Notes in Computer Science*, pages 74–91. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21069-3. URL `http://dx.doi.org/10.1007/978-3-642-21070-9_8`. 10.1007/978-3-642-21070-9_8.

Hassan Ait-Kaci. *Warren's Abstract Machine. A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts. London, England, 1991.

Gianluca Amato and James Lipton. Indexed categories and bottom-up semantics of logic programs. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 438–454. Springer, 2001. ISBN 3-540-42957-3.

Gianluca Amato, James Lipton, and Robert McGrail. On the algebraic structure of declarative programming languages. *Theoretical Computer Science*, 410(46):4626 – 4671, 2009. ISSN 0304-3975. doi: DOI:10.1016/j.tcs.2009.07.038. URL `http://www.sciencedirect.com/science/article/B6V1G-4WV15VS-7/2/5475111b9a9642244a208e9bd1fcd46a`. Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi.

Hahnal Andréka and Dimitry A. Bredikhin. The equational theory of union-free algebras of relations. *Algebra Universalis*, 33:516–532, 1995. ISSN 0002-5240. URL `http://dx.doi.org/10.1007/BF01225472`. 10.1007/BF01225472.

Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *Proc. 21st. ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, Oregon, 1994. ACM Press.

Joe Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: 10.1145/1238844.1238850. URL `http://doi.acm.org/10.1145/1238844.1238850`.

Andrea Asperti and Simone Martini. Projections instead of variables: A category theoretic interpretation of logic programs. In *ICLP*, pages 337–352, 1989.

Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-45520-0.

Franz Baader and Wayne Snyder. Unification theory. In Robinson and Voronkov [2001], pages 445–532. ISBN 0-444-50813-9, 0-262-18223-8.

Roland Carl Backhouse and Paul F. Hoogendijk. Elements of a relational theory of datatypes. In Bernhard Möller, Helmuth Partsch, and Stephen Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 7–42. Springer, 1993. ISBN 3-540-57499-9.

John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. 21(8):613–641, August 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/

359576.359579.

Hendrik Pieter Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 103 of *Studies in Logic and Mathematics*. North Holland, 1984.

Michael Barr and Charles Wells. *Category theory for computing science (3. ed.)*. Sentre de REcherches Mathématiques, 1999. ISBN 2-921120-31-3.

Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *ArXiv e-prints*, March 2012.

Yves Bekkers and Paul Tarau. Monadic constructs for logic programming. In *ILPS*, pages 51–65, 1995.

Marco Bellia and M. Eugenia Occhiuto. C-expressions: A variable-free calculus for equational logic programming. *Theor. Comput. Sci.*, 107(2):209–252, 1993.

Marco Bellia and M. Eugenia Occhiuto. Lazy linear combinatorial unification. *Journal of Symbolic Computation*, 27(2):185–206, 1999. URL `citeseer.ist.psu.edu/bellia98lazy.html`.

Rudolf Berghammer and Thorsten Hoffmann. Relational depth-first-search with applications. In Jules Desharnais, editor, *RelMiCS*, pages 11–20, 2000.

Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice-Hall, 1996a. URL `http://www.cs.ox.ac.uk/publications/books/algebra/`.

Richard S. Bird and Oege de Moor. The algebra of programming. In Manfred Broy, editor, *NATO ASI DPD*, pages 167–203, 1996b. ISBN 3-540-60947-4.

Maria Paola Bonacina and Jieh Hsiang. On rewrite programs: Semantics and relationship with prolog. *J. Log. Program.*, 14(1&2):155–180, 1992.

Staffan Bonnier and Jan Maluszynski. Towards a clean amalgamation of logic programs with external procedures. In *ICLP/SLP*, pages 311–326, 1988.

Francis Borceux. *Handbook of Categorical Algebra 2. Categories and Structures*, volume 51 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994. ISBN 052144179X.

Bernd Braßel and Jan Christiansen. A relation algebraic semantics for a lazy functional logic language. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *RelMiCS*, volume 4988 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2008. ISBN 978-3-540-78912-3.

Pascal Brisset and Olivier Ridoux. Continuations in lambda-prolog. In *ICLP*, pages 27–43, 1993.

Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-83595-9.

Paul Broome and James Lipton. Combinatory logic programming: computing in relation calculi. In *ILPS '94: Proceedings of the 1994 International Symposium on Logic programming*, pages 269–285, Cambridge, MA, USA, 1994. MIT Press. ISBN 0-262-52191-1.

Paul Broome and Jim Lipton. Constructive relational programming. In ARO-Report, editor, *9th Army Conference on Applied Mathematics and Computing*, volume 92, 1992.

Carolyn Brown and Graham Hutton. Categories, allegories and circuit design. In *LICS*, pages 372–381. IEEE Computer Society, 1994.

Roberto Bruni, Ugo Montanari, and Francesca Rossi. An interactive semantics of logic programming. *Theory and Practice of Logic Programming*, 1(06):647–690, 2001. doi: 10.1017/S1471068401000035.

F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP8/95.0.7.20, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, November 1999.

Albert Burroni. Algèbres graphiques (sur un concept de dimension dans les langages formels. *Cahiers de topologie et géométrie différentielle*, XXII(3), 1981.

Carsten Butz. Regular categories and regular logic. Technical Report LS-98-2, BRICS, October 1998.

Daniel Cabeza, Manuel V. Hermenegildo, and James Lipton. Hiord: A type-free higher-order logic programming language with predicate abstraction. In Michael J. Maher, editor, *ASIAN*, volume 3321 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2004. ISBN 3-540-24087-X.

Felice Cardone and J. Roger Hindley. History of Lambda-calculus and Combinatory Logic. Technical Report MRRS-05-06, Swansea University Mathematics Department Research Report, 2006.

Amadeo Casas, Daniel Cabeza, and Manuel V. Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in lp systems. In Masami Hagiya and Philip Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2006. ISBN 3-540-33438-6.

Dave Cattrall and Colin Runciman. A relational programming system with inferred representations. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 475–476. Springer Berlin / Heidelberg, 1992. ISBN 978-3-540-55844-6. URL `http://dx.doi.org/10.1007/3-540-55844-6_156`. 10.1007/3-540-55844-6_156.

Dave Cattrall and Colin Runciman. Widening the representation bottleneck: A functional implementation of relational programming. In *FPCA*, pages 191–200, 1993.

D.M. Cattrall. *The Design and Implementation of a Relational Programming Systema*. PhD thesis, University of York, 1992.

Serenella Cerrito. A linear semantics for allowed logic programs. In *LICS* DBL [1990], pages 219–227. ISBN 0-8186-2073-0.

James Cheney and Christian Urban. alpha-prolog: A logic programming language with names, binding and a-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004. ISBN 3-540-22671-0.

James Robert Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004. Advisor – Morrisett, Greg.

Keith L. Clark. Negation as failure. In Gallaire and Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1977.

Marco Comini and Maria Chiara Meo. Compositionality properties of *ld*-derivations. *Theor. Comput. Sci.*, 211(1-2):275–309, 1999.

Marco Comini, Giorgio Levi, and Maria Chiara Meo. A theory of observables for logic programs. *Inf. Comput.*, 169(1):23–80, 2001.

Hubert Comon. Disunification: A survey. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 322–359, 1991. URL `citeseer.ist.psu.edu/comon91disunification.html`.

Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *POPL*, pages 493–501, 1993.

Andrea Corradini and Andrea Asperti. A categorial model for logic programs: Indexed monoidal categories. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 666 of *Lecture Notes in Computer Science*, pages 110–137. Springer, 1992. ISBN 3-540-56596-5.

Andrea Corradini and Ugo Montanari. An algebraic semantics for structured transition systems and its applications to logic programs. *Theor. Comput. Sci.*, 103(1):51–106, 1992.

Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Sci. Comput. Program.*, 8(2):173–202, 1987.

Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).

Roy L. Crole and Andrew M. Pitts. New foundations for fixpoint computations. In *LICS* DBL [1990], pages 489–497. ISBN 0-8186-2073-0.

Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North Holland, 1958.

J. W. de Bakker and Willem P. de Roever. A calculus for recursive program schemes. In *ICALP*, pages 167–196, 1972.

Frank S. de Boer, Alessandra Di Pierro, and Catuscia Palamidessi. An algebraic perspective of constraint logic programming. *J. Log. Comput.*, 7(1):1–38, 1997.

Willem P. de Roever. Recursion and parameter mechanisms: An axiomatic approach. In Jacques Loeckx, editor, *ICALP*, volume 14 of *Lecture Notes in Computer Science*, pages 34–65. Springer, 1974. ISBN 3-540-06841-4.

Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

Nachum Dershowitz and David A. Plaisted. Rewriting. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 535–610. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

Razvan Diaconescu. A category-based equational logic semantics to constraint programming. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *COMPASS/ADT*, volume 1130 of *Lecture Notes in Computer Science*, pages 200–221. Springer, 1995. ISBN 3-540-61629-2.

Stephan Diehl, Pieter H. Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Comp. Syst.*, 16(7):739–751, 2000.

Daniel J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114(2): 273–298, 1993. URL `citeseer.ist.psu.edu/255654.html`.

Daniel J. Dougherty and Claudio Gutiérrez. Normal forms for binary relations. *Theor. Comput. Sci.*, 360(1):228–246, 2006. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2006.03.023.

Gilles Dowek. Higher-order unification and matching. In Robinson and Voronkov [2001], pages 1009–1062. ISBN 0-444-50813-9, 0-262-18223-8.

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique, April 1998.

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher order unification via explicit substitutions. *Inf. Comput.*, 157(1-2):183–235, 2000.

Gilles Dowek, Murdoch J. Gabbay, and Dominic P. Mulligan. Permissive Nominal Terms and their Unification: an infinite, co-infinite approach to nominal techniques (journal version). *Logic Journal of the IGPL*, 18(6):769–822, 2010. doi: 10.1093/jigpal/jzq006.

B. Dwyer. Programming using binary relations: a proposed programming languague. Technical report, University of Adelaide, 1994. URL `http://cs.adelaide.edu.au/~dwyer/TR95-10.html`.

Beno Eckmann, editor. *Seminar on Triples and Categorical Homology Theory*, number 80 in Lecture Notes in Mathematics, 1969. Springer-Verlag.

M. Hanus (ed.), H. Kuchen, and J.J. Moreno-Navarro et al. Curry: An integrated functional logic language. Technical report, RWTH Aachen, 2000. URL `http://www.informatik.uni-kiel.de/~mh/curry/report.html`.

Birgit Elbl. A declarative semantics for depth-first logic programs. *J. Log. Program.*, 41(1):27–66, 1999.

Adam Eppendahl. *Categories and Types for Axiomatic Domain Theory*. PhD thesis, University of London, 2003a.

Moreno Falaschi, Giorgio Levi, Catuscia Palamidessi, and Maurizio Martelli. Declarative modeling of the operational behavior of logic languages. *Theor. Comput. Sci.*, 69(3):289–318, 1989.

Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting (journal version). *Information and Computation*, 205(6):917–965, June 2007. doi: 10.1016/j.ic.2006.12.002.

Stacy E. Finkelstein, Peter J. Freyd, and James Lipton. Logic programming in tau categories. In Leszek Pacholski and Jerzy Tiuryn, editors, *CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 1994. ISBN 3-540-60017-5.

Stacy E. Finkelstein, Peter J. Freyd, and James Lipton. A new framework for declarative programming. *Theor. Comput. Sci.*, 300(1-3):91–160, 2003.

Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. *Symposium on Logic in Computer Science*, 0:193, 1999. ISSN 1043-6871. doi: http://doi.ieeecomputersociety.org/10.1109/LICS.1999.782615.

Melvin Fitting. Fixpoint semantics for logic programming a survey. *Theor. Comput. Sci.*, 278(1-2): 25–51, 2002.

P.J. Freyd and A. Scedrov. *Categories, Allegories*. North Holland Publishing Company, 1991.

Marcelo F. Frias and Roger D. Maddux. Completeness of a relational calculus for program schemes. In *LICS*, pages 127–134, 1998.

Murdoch Gabbay and Aad Mathijssen. Capture-avoiding substitution as a nominal algebra (journal version). *Formal Aspects of Computing*, 20(4-5):451–479, January 2008.

Murdoch J. Gabbay. Foundations of nominal techniques: logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic*, 17(2):161–229, 2011. doi: 10.2178/bsl/1305810911.

Murdoch J. Gabbay and Aad Mathijssen. Nominal universal algebra: equational logic with names and binding. *Journal of Logic and Computation*, 19(6):1455–1508, December 2009. doi: 10.1093/logcom/exp033.

Murdoch J. Gabbay and Aad Mathijssen. A nominal axiomatisation of the lambda-calculus. *Journal of Logic and Computation*, 20(2):501–531, April 2010. doi: 10.1093/logcom/exp049.

Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224, Washington, DC, USA, 1999. IEEE Computer Society Press. ISBN 0-7695-0158-3.

Murdoch J. Gabbay, Samuel Rota Buló, and Andrea Marin. Denotations for functions in which variables are first-class denotational citizens — or, variables are data. Unpublised manuscript, 2007.

Haim Gaifman and Ehud Y. Shapiro. Fully abstract compositional semantics for logic programs. In *POPL*, pages 134–142. ACM Press, 1989. ISBN 0-89791-294-2.

Emilio Jesús Gallego Arias, Julio Mariño Carballo, and José María Rey Poza. A proposal for disequality constraints in curry. *Electron. Notes Theor. Comput. Sci.*, 177:269–285, June 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2007.01.014. URL `http://dx.doi.org/10.1016/j.entcs.2007.01.014`.

Emilio Jesús Gallego Arias, Julio Mariño, and José María Rey Poza. A generic semantics for constraint functional logic programming. In *Proc. of the 16th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2007)*, 2007.

Emilio Jesús Gallego Arias. Desarrollo de un compilador de prolog a código relacional. Asignatura Sistemas Informáticos, FI-UPM, September 2004.

Emilio Jesús Gallego Arias. An efficient clpfd implementation in pure prolog, 2006. Included in Ciao-Prolog.

Emilio Jesús Gallego Arias and James Lipton. Logic programming in tabular allegories. In *Technical Communications of the 27 International Conference on Logic Programming*, LIPICS, 2012.

Emilio Jesús Gallego Arias and Julio Mariño. An overview of the Sloth2005 Curry system: system description. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, pages 66–69, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-069-8. doi: http://doi.acm.org/10.1145/1085099.1085113.

Emilio Jesús Gallego Arias, James Lipton, Julio Mariño, and Pablo Nogueira. First-order unification using variable-free relational algebra. *Logic Journal of IGPL*, 19(6):790–820, 2011. doi: 10.1093/jigpal/jzq011. URL `http://jigpal.oxfordjournals.org/content/19/6/790.abstract`.

Emilio Jesús Gallego Arias, James Lipton, and Julio Mariño. Extensions to logic programming in tabular allegories: Algebraic data types, functions, constraints and monads. Technical report, Universidad Politécnica de Madrid, 2012a.

Emilio Jesús Gallego Arias, James Lipton, and Julio Mariño. Constraint logic programming with a relational machine. Technical report, Universidad Politécnica de Madrid, 2012b. `http://babel.ls.fi.upm.es/~egallego/iclp/clprm.pdf`.

Wolfgang Gehrke. Problems in rewriting applied to categorical concepts by the example of a computational comonad. In Jieh Hsiang, editor, *RTA*, volume 914 of *Lecture Notes in Computer Science*, pages 210–224. Springer, 1995. ISBN 3-540-59200-8.

J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

Joseph Goguen. What is unification? A categorical view of substitution, equation and solution. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*, pages 217–261. Academic, 1989. URL `citeseer.ist.psu.edu/166243.html`.

Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981.

Alessio Guglielmi. A system of interaction and structure. *ACM Transactions on Computational Logic*, 8(1):1–64, 2007. URL `http://cs.bath.ac.uk/ag/p/SystIntStr.pdf`.

Pierre-Louis Curien Guy Cousineau and Bernard Robinet, editors. *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*. Springer, 1985.

Patrick A. V. Hall. Relational algebras, logic, and functional programming. In Beatrice Yormark, editor, *SIGMOD Conference*, pages 326–333. ACM Press, 1984.

Andreas Hamfelt and Jørgen Fischer Nilsson. Inductive synthesis of logic programs by composition of combinatory program schemes. In P. Flener, editor, *LOPSTR'98, 8th. International Workshop on Logic-Based Program Synthesis and Transformation*, volume 1559 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 1998.

Michael Hanus, Sergio Antoy, Herbert Kuchen, Francisco J. López-Fraguas, Wolfgang Lux, Juan José Moreno-Navarro, and Frank Steiner. *Curry: An Integrated Functional Logic Language*, 0.8 edition, April 2003. URL `http://www.informatik.uni-kiel.de/~mh/curry/report.html`. Editor: Michael Hanus.

Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Sci. Comput. Program.*, 58(1-2):115–140, 2005.

Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and German Puebla. An overview of ciao and its design philosophy. *CoRR*, abs/1102.5497, 2011.

Patricia M. Hill and John W. Lloyd. *The Gödel programming language*. MIT Press, 1994. ISBN 978-0-262-08229-7.

Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110:32–42, 1994.

Paul Hudak and Mark P. Jones. Haskell vs. ada vs. C++ vs awk vs ... an experiment in software prototyping productivity. Technical report, 1994. URL `citeseer.ist.psu.edu/hudak94haskell.html`.

Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electr. Notes Theor. Comput. Sci.*, 172:437–458, 2007.

Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.

Neil Immerman. Relational queries computable in polynomial time (extended abstract). In Lewis et al. [1982], pages 147–152. ISBN 0-89791-070-2.

Daniel Jackson. Alloy: A new technology for software modelling. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, page 20. Springer, 2002a. ISBN 3-540-43419-4.

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11 (2):256–290, 2002b.

Bart Jacobs. *Categorical Logic and Type Theory*. Elsevier Science, 1999.

Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL*, pages 111–119. ACM, 1987.

Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994. URL `http://citeseer.ist.psu.edu/jaffar94constraint.html`.

John S. Jeavons. An alternative linear semantics for allowed logic programs. *Ann. Pure Appl. Logic*, 84 (1):3–16, 1997.

Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium*. Oxford University Press, 2003.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993. ISBN 978-0-13-020249-9.

S. Peyton Jones and J. Hughes. *Report on the Programming Language Haskell 98. A Non-strict Purely Functional Language*, February 1999.

Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1985.

Wolfram Kahl. Dependently-typed formalisation of relation-algebraic abstractions. In Harrie de Swart, editor, *Relational and Algebraic Methods in Computer Science*, volume 6663 of *Lecture Notes in Computer Science*, pages 230–247. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21069-3. URL `http://dx.doi.org/10.1007/978-3-642-21070-9_18`. 10.1007/978-3-642-21070-9_18.

Yoshiki Kinoshita and A. John Power. A fibrational semantics for logic programs. In Roy Dyckhoff, Heinrich Herre, and Peter Schroeder-Heister, editors, *ELP*, volume 1050 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 1996. ISBN 3-540-60983-0.

Hélène Kirchner. Some extensions of rewriting. In Hubert Comon and Jean-Pierre Jouannaud, editors, *Term Rewriting*, volume 909 of *Lecture Notes in Computer Science*, pages 54–73. Springer, 1993. ISBN 3-540-59340-3.

Hélène Kirchner. On the use of constraints in automated deduction. In Andreas Podelski, editor, *Constraint Programming*, volume 910 of *Lecture Notes in Computer Science*, pages 128–146. Springer, 1994. ISBN 3-540-59155-9.

Jan Willem Klop. Term rewriting systems. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, 1992. URL `citeseer.ist.psu.edu/klop92term.html`.

Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 342–376. Springer, Berlin, Heidelberg, 1983.

Ekaterina Komendantskaya and John Power. Coalgebraic semantics for derivations in logic programming. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 268–282. Springer, 2011a. ISBN 978-3-642-22943-5.

Ekaterina Komendantskaya and John Power. Coalgebraic derivations in logic programming. In Marc Bezem, editor, *CSL*, volume 12 of *LIPIcs*, pages 352–366. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011b. ISBN 978-3-939897-32-3.

Ekaterina Komendantskaya, Guy McCusker, and John Power. Coalgebraic semantics for parallel derivation strategies in logic programming. In Michael Johnson and Dusko Pavlovic, editors, *AMAST*, volume 6486 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2010. ISBN 978-3-642-17795-8.

M. Kulaš and C. Beierle. Defining Standard Prolog in rewriting logic. In K. Futatsugi, editor, *Proc. of the 3rd Int. Workshop on Rewriting Logic and its Applications (WRLA'2000), Kanazawa*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing polymorphic typing in a logic programming language. *Comput. Lang.*, 20(1):25–42, 1994.

Joachim Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, Cambridge, 1986.

Peter J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, 1965.

John Launchbury. A natural semantics for lazy evaluation. In *POPL*, pages 144–154, 1993.

F. William Lawvere. *Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the context of Functorial Semantics of Algebraic Theories*. PhD thesis, Columbia University, 1968.

F. William Lawvere. Adjointness in foundations. *Dialectica*, 23:281–296, 1969a.

F. William Lawvere. Diagonal arguments and cartesian closed categories. In *Category Theory, Homology Theory and their Applications, II (Battelle Institute Conference, Seattle, Wash., 1968, Vol. Two)*, pages 134–145. Springer-Verlag, Berlin, 1969b.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system (release 3.12): Documentation and user's manual, jul 2011. URL `http://caml.inria.fr/distrib/ocaml-3.12/ocaml-3.12-refman.pdf`.

Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors. *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, 1982. ACM. ISBN 0-89791-070-2.

James Lipton and Robert McGrail. Encapsulating data in logic programming via categorial constraints. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *PLILP/ALP*, volume 1490 of *Lecture Notes in Computer Science*, pages 391–410. Springer, 1998. ISBN 3-540-65012-1.

Jim Lipton and Emily Chapman. Some notes on logic programming with a relational machine. In Ali Jaoua, Peter Kempf, and Gunther Schmidt, editors, *Using Relational Methods in Computer Science*, Technical Report Nr. 1998-03, pages 1–34. Fakultät für Informatik, Universität der Bundeswehr München, July 1998.

John W. Lloyd. *Foundations of logic programming*. Springer-Verlag New York, Inc., New York, NY, USA, 1984. ISBN 0-387-13299-6.

Francisco Javier López-Fraguas and Jaime Sánchez-Hernández. *TOY*: A multiparadigm declarative system. In Paliath Narendran and Michaël Rusinowitch, editors, *RTA*, volume 1631 of *Lecture Notes in Computer Science*, pages 244–247. Springer, 1999. ISBN 3-540-66201-4.

Saunders Mac Lane. *Categories for the Working Mathematician (Graduate Texts in Mathematics)*. Springer, 2nd edition, September 1998. ISBN 0387984038. URL `http://www.worldcat.org/isbn/0387984038`.

Roger D. Maddux. Introductory course on relation algebras, finite-dimensional cylindric algebras, and their interconnections. In H. Andreka, J. D. Monk, and I. Nemeti, editors, *Algebraic Logic (Proc. Conf. Budapest 1988)*, volume 54 of *Colloq. Math. Soc. J. Bolyai*, pages 361–392. North Holland, Amsterdam, 1991.

Roger D. Maddux. Relation-algebraic semantics. *Theor. Comput. Sci.*, 160(1&2):1–85, 1996.

Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings, Third Annual Symposium on Logic in Computer Science, 5-8 July 1988, Edinburgh, Scotland, UK*, pages 348–357. IEEE Computer Society, 1988.

Micheal Makkai and Gonzalo E. Reyes. *First Order Categorical Logic*, volume 611 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 1977.

Anatolij Ivanovich Mal'tsev. On the elementary theories of locally free universal algebras. *Soviet Math*, pages 768–771, 1961.

Julio Mariño. *Semantics and Analysis of Functional Logic Programs*. PhD thesis, Universidad Politécnica de Madrid, Facultad de Informática, May 2002. Supervisor: Juan José Moreno Navarro.

Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, mar 1998. ISBN 0262133415. URL `http://www.worldcat.org/isbn/0262133415`.

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

Conor McBride. First-order unification by structural recursion. *J. Funct. Program.*, 13(6):1061–1075, 2003.

Richard McPhee. Towards a relational programming language, 1995.

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1-2):125–157, 1991.

Robin Milner. Models of LCF. Technical report, Stanford, CA, USA, 1973.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. ISSN 0890-5401. doi: DOI:10.1016/0890-5401(91)90052-4. URL `http://www.sciencedirect.com/science/article/B6WGK-4DX4K5F-P/2/a5ad300540b29452e7e41eed22de9e89`. Selections from 1989 IEEE Symposium on Logic in Computer Science.

José F. Morales, Manuel Carro, Germán Puebla, and Manuel V. Hermenegildo. A generator of efficient abstract machine implementations and its application to emulator minimization. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2005. ISBN 3-540-29208-X.

José F. Morales, Manuel Carro, and Manuel V. Hermenegildo. Towards description and optimization of abstract machines in an extension of prolog. In Germán Puebla, editor, *LOPSTR*, volume 4407 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2006. ISBN 978-3-540-71409-5.

J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In David B. MacQueen and Luca Cardelli, editors, *POPL*, pages 85–97. ACM, 1998. ISBN 0-89791-979-3.

Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

P. S. Mulry. Lifting results for categories of algebras. *Theor. Comput. Sci.*, 278(1-2):257–269, 2002. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(00)00338-8.

Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23(3): 295–307, 1984.

Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in AI and Logic Programming, Volume 5: Logic Programming*. Oxford. URL `http://citeseer.ist.psu.edu/article/nadathur86higherorder.html`.

David A. Naumann. A recursion theorem for predicate transformers on inductive data types. *Inf. Process. Lett.*, 50(6):329–336, 1994.

István Németi. Review of [Tarski and Givant, 1987] (untitled). The Journal of Symbolic Logic, Vol 55. No. 1, pp 350–352, March 1990.

Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.

Michael J. O'Donnell. Equational logic programming. In Dov M. Gabbay, C.J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5: Logic Programming. Oxford University Press, 1998.

Richard A. O'Keefe. *The Craft of Prolog*. MIT, 1990.

Mike Paterson and Mark N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Maciej Pirog and Jeremy Gibbons. A functional derivation of the Warren Abstract Machine. 2011. URL `http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/wam.pdf`. Submitted for publication.

Andrew M. Pitts. Categorical logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 5. Algebraic and Logical Structures*, pages 39–128. Oxford University Press, 2000. ISBN 0-19-853781-6. URL `http://www.oup.co.uk/isbn/0-19-853781-6`.

Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1: 125–159, 1975.

Gordon D. Plotkin. Lambda-Definability in the Full Type Hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H.B.˜Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, London, 1980.

Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.

Gordon D. Plotkin and Matija Pretnar. A logic for algebraic effects. In *LICS* DBL [2008], pages 118–129. ISBN 978-0-7695-3183-0.

Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. ISBN 978-3-642-00589-3.

Detlef Plump. Term graph rewriting. In Hartmut Ehrig, G. Engels, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

Vaughan R. Pratt. Origins of the calculus of binary relations. In *Logic in Computer Science*, pages 248–254, 1992. URL `citeseer.ist.psu.edu/2576.html`.

Germán Puebla and Claudio Ochoa. Poly-controlled partial evaluation. In Annalisa Bossi and Michael J. Maher, editors, *PPDP*, pages 261–271. ACM, 2006. ISBN 1-59593-388-3.

Germán Puebla, Manuel V. Hermenegildo, and John P. Gallagher. An integration of partial evaluation in a generic abstract interpretation framework. In *PEPM*, pages 75–84, 1999.

David J. Pym. Functorial kripke models of the π-calculus. In *Mathematical Sciences, Semantics Programme, Workshop on Categories and Logic Programming*, 1997.

David J. Pym. Functorial kripke-beth-joyal models of the lambda pi -calculus i: type theory and internal logic, 2001a.

David J. Pym. Functorial kripke-beth-joyal models of the lambda pi-calculus ii: the lf logical framework, 2001b.

John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

Gilles Richard. A rewrite approach for constraint logic programming. In *In Proceedings of International Symposium LATIN'95, number 911 in LNCS*, pages 469–482. Springer-Verlag, 1995.

John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321250.321253.

John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

David M. Russinoff. A verified prolog compiler for the Warren Abstract Machine. *Journal of Logic Programming*, 13:367–412, 1992.

David E. Rydeheard and Rod M. Burstall. A categorical unification algorithm. In *Proceedings of a tutorial and workshop on Category theory and computer programming*, pages 493–505, New York, NY, USA, 1986. Springer-Verlag New York, Inc. ISBN 0-387-17162-2.

Antonino Salibra. On the algebraic models of lambda calculus. *Theor. Comput. Sci.*, 249(1):197–240, 2000.

Vijay A. Saraswat. The category of constraint systems is cartesian-closed. In Andre Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symp. on Logic in Computer Science, LICS 1992*, pages 341–345. IEEE Computer Society Press, June 1992.

Silvija Seres, J. Michael Spivey, and C. A. R. Hoare. Algebra of logic programming. In *ICLP*, pages 184–199, 1999.

Ehud Y. Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3): 413–510, 1989.

Mary Sheeran and Geraint Jones. Circuit design in ruby. In J. Staunstrup, editor, *Formal Methods in VLSI Design*. North-Holland, Sep 1990. Lecture notes on Ruby from a summer school in Lyngby, Denmark,.

Zoltan Somogyi, Fergus James Henderson, and Thomas Charles Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29, 1996.

Sam Staton. General structural operational semantics through categorical logic. In *LICS* DBL [2008], pages 166–177. ISBN 978-0-7695-3183-0.

Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press, 1986.

Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1981. ISBN 0262690764.

R. C. H. Tanner. On the role of equality and inequality in the history of mathematics. *The British Journal for the History of Science*, 1(2):159–169, December 1962. ISSN 0007-0874 (print), 1474-001X (electronic). doi: http://dx.doi.org/10.1017/S0007087400001333. URL `http://www.jstor.org/stable/4025130`.

Alfred Tarski and Steven Givant. *A Formalization of Set Theory Without Variables*, volume 41 of *Colloquium Publications*. American Mathematical Society, Providence, Rhode Island, 1987.

Evan Tick. The deevolution of concurrent logic programming languages. *J. Log. Program.*, 23(2): 89–123, 1995.

Eneia Todoran and Nikolaos S. Papaspyrou. Continuations for parallel logic programming. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '00, pages 257–267, New York, NY, USA, 2000. ACM. ISBN 1-58113-265-4. doi: http://doi.acm.org/10.1145/351268.351297. URL `http://doi.acm.org/10.1145/351268.351297`.

D. A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9:31 – 49, 1979.

Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. In Matthias Baaz and Johann A. Makowsky, editors, *CSL*, volume 2803 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2003. ISBN 3-540-40801-0.

Dirk van Dalen. *Logic and Structure*. Springer Verlag, Berlin, 1983.

Maarten H. van Emden. Compositional semantics for the procedural interpretation of logic. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2006. ISBN 3-540-36635-0.

Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In Lewis et al. [1982], pages 137–146. ISBN 0-89791-070-2.

Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5.

Philip Wadler. The girard-reynolds isomorphism (second edition). *Theor. Comput. Sci.*, 375(1-3): 201–226, 2007.

David H. D. Warren. An abstract Prolog instruction set. Technical note 309, SRI International, Menlo Park, CA, October 1983.

David H. D. Warren. Or-parallel execution models of prolog. In Hartmut Ehrig, Robert A. Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT, Vol.2*, volume 250 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 1987. ISBN 3-540-17611-X.

Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993. ISBN 978-0-262-23169-5.