WEST UNIVERSITY OF TIMIŞOARA

DOMAIN: COMPUTER SCIENCE

HABILITATION THESIS

CANDIDATE:

MARIN MIRCEA, CONFERENŢIAR UNIV. DR.

WEST UNIVERSITY OF TIMIŞOARA

2017

WEST UNIVERSITY OF TIMIŞOARA

DOMAIN: COMPUTER SCIENCE

COMPUTATIONAL MODELS FOR DECLARATIVE PROGRAMMING

CANDIDATE:

MARIN MIRCEA, ASSOCIATE PROFESSOR, PhD

WEST UNIVERSITY OF TIMIŞOARA

2017

Abstract

Declarative programming is a well established programming paradigm with a large variety of styles which attempt to minimize the programming effort by describing *what* we know in terms of the problem domain instead of *how* to perform a sequence of actions which yield a desired outcome. Constraint solving, logic programming, functional programming, and rule-based programming are well known examples of such styles where programming amounts to expressing the logic of computation instead of describing the control flow.

Multiparadigm programming emerged from the desire to increase the expressive power and efficiency by combining some of the most important programming styles. The success story of the scheme CLP which allows to combine constraint programming with logic programming and implement it efficiently, ignited similar efforts to combine more declarative programming styles.

This thesis summarises my research activity after receiving a PhD degree from Johannes Kepler University in Linz. My achievements are concerned with the design and implementation of systems for multiparadigm declarative programming, and were developed along three major directions of research:

1. **Lazy narrowing calculi** and their use as computational models for functional logic programming.

2. **Constraint functional logic programming in open environments**. We designed and implemented a new system called Open CFLP which adapts the scheme CFLP($\mathcal{X}, \mathcal{S}, \mathcal{C}$) from my PhD thesis to a service model where domain-specific constraint solvers are distributed over the network and can be accessed remotely through standardised protocols. This model emerged from the following premises: (1) solvers evolve over time; they implement sophisticated algorithms developed by experts who wish to protect their intelligent and copy rights, (2) deploying them as services in an open environment

is beneficial for both sides: the developers who are willing to provide their solvers as services, and the clients wish to access these services remotely instead of downloading large and reasonably expensive code, and (3) solving services may be deployed dynamically, independent of the plan of the potential clients.

3. **Rule-based programming** from the point of view of combining FLP with (1) a declarative language to express strategies that constrain the order of rule application, and with (2) various constraint solving capabilities. My main contribution is ρLog, a system for rule-based programming with user-defined strategies, and its implementation as a Mathematica add-on package. The main novelty of ρLog are its matching constructs, like sequence variables, context variables, and membership constraints, which increase the expressive power of rule-based specifications beyond those of first order conditional rewrite systems, and make it suitable for new applications, such as XML document validation and transformation.

Other achievements, closely related to the research directions mentioned before, are:

- algorithms designed to solve matching and unification problems for the higher-order extensions envisioned by us: sequence variables, context variables, and membership constraints for their bindings.

- algorithms for type inference for systems with regular expression types, and for the implementation of operations on regular hedge languages.

The thesis is structured in five chapters. Chapter 1 gives an overview of my scientific, professional, and academic achievements, and indicates the main thematic directions of research which I pursued so far. Chapter 2 presents our achievements in the study of lazy narrowing as computational models for FLP. Chapter 3 describes the outcome of our effort to adapt constraint functional logic programming scheme proposed in my PhD thesis to the capabilities of an open environment, where constraint solvers can be deployed dynamically as services that can be accessed remotely. Chapter 4 gives an account to our results (theoretical properties and practical implementations) to define a calculus for rule-based programming with strategies, and algorithms for related subproblems. Finally, in Chapter 5, I present my career development plan by focusing on academic and scientific objectives.

Rezumat

Programarea declarativă este o paradigmă cu o varietate de stiluri menite să minimizeze efortul de programare punând accentul pe descrierea a *ceea ce ştim* referitor la problemă în loc să precizăm *cum* se efectuează o secvenţă de acţiuni care produc rezultatul dorit. Rezolvarea constrângerilor, programarea logică, programarea funcţională şi programarea bazată pe reguli sunt exemple binecunoscute de stiluri în care activitatea de programare constă în exprimarea logicii de calcul în locul descrierii unui flux de control.

Programarea multiparadigmă a provenit din dorinţa de a creşte puterea de expresie şi eficienţa combinând cele mai relevante stiluri de programare. Povestea de succes a schemei CLP care combină programarea cu constrângeri cu programarea logică, precum şi implementarea eficientă a acestei combinări, a declanşat eforturi similare de combinare a altor stiluri de programare declarativă.

Această teză conţine o descriere succintă a activităţii mele de cercetare după obţinerea doctoratului la universitatea Johannes Kepler University din Linz. Cercetarea mea s-a axat pe designul şi implementarea sistemelor de programare declarativă multiparadigmă şi a fost elaborată urmând trei direcţii tematice majore:

1. **Sisteme de calcul lazy narrowing** şi utilizarea lor ca modele de calcul pentru programarea logică şi funcţională (FLP).

2. **Programarea logică şi funcţională cu constrângeri în medii deschise**. Am contribuit la conceperea şi implementarea unui sistem nou numit Open CFLP care adaptează schema CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$ introdusă în teza mea de doctorat la un model orientat pe servicii în care rezolvitoarele de constrângeri pe domenii specifice sunt distribuite în reţea şi pot fi accesate la distanţă prin protocoale standard. Elaborarea acestui model a fost motivată de premisele următoare: (1) rezolvitoarele de constrângeri evoluează în timp; ele implementează algoritmi sofisticaţi dezvoltaţi de experţi dornici să-şi protejeze drepturile de autor, (2) oferirea lor ca servicii în medii deschise este benefică

pentru ambii participanţi: dezvoltatorii sunt dornici să le ofere ca servicii, iar clienţii sunt dornici să le acceseze de la distanţă în loc să descarce cod sursă relativ voluminos şi costisitor, şi (3) serviciile de rezolvare a constrângerilor pot fi oferite dinamic, independent de cerinţele potenţialilor clienţi.

3. **Programarea bazată pe reguli** din punctul de vedere al combinării FLP cu (1) un limbaj declarativ de exprimare a strategiilor care constrâng ordinea şi locul de aplicare a regulilor, şi cu (2) capabilităţi diverse de rezolvare a constrângerilor. Principala mea contribuţie a fost ρLog, un sistem de programare bazată pe reguli cu strategii definite de utilizator, şi implementarea lui ca Mathematica add-on package. O noutate sunt capabilităţile de matching cu variabile secvenţă, variabile context şi constrângeri de apartenenţă. Acestea măresc semnificativ puterea de expresie a specificaţiilor bazate pe reguli dincolo de puterea de expresie a sistemelor e rescriere conciţională de ordin I şi facilitează aplicaţii noi, precum validarea şi transformarea documentelor XML.

Alte rezultate, strâns legate de direcţiile de cercetare menţionate mai sus, sunt:

- algoritmi de rezolvare a problemelor de matching şi unificare pentru diversele extensii de ordin superior studiate de noi: variabile secvenţă, variabile context, şi constrângeri de apartenenţă.

- algoritmi de inferenţă de tip în sisteme cu tipuri exprimate cu expresii regulate, şi de implementare a unor operaţii asupra limbajelor de secvenţe regulate de arbori.

Teza cuprinde 5 capitole. Capitolul 1 este o prezentare succintă a rezultatelor mele ştiinţifice, profesionale şi academice, şi indică principalele direcţii tematice de cercetare pe care le-am urmat. În capitolul 2 prezint rezultatele obţinute în studiul sistemelor de lazy narrowing ca modele de calcul pentru FLP. Capitolul 3 descrie rezultatele eforturilor noastre de a adapta schema CFLP propusă în teza mea de doctorat la facilităţile unui mediu deschis în care rezolvitoarele de constrângeri pot fi oferite dinamic ca servicii accesibile de la distanţă. Capitolul 4 prezintă rezultatele obţinute de noi (proprietăţi teoretice şi implementări practice) legate de definirea unui calcul pentru programarea bazată pe reguli cu strategii, precum şi algoritmii propuşi de noi pentru subproblemele aferente. În final, în capitolul 5 prezint planul de dezvoltare a carierei punând accentul de obiectivele mele ştiinţifice şi academice.

Contents

Chapter 1

Scientific, Professional, and Academic Achievements

1.1 Overview

I received a PhD degree from Johannes Kepler University of Linz. My thesis "Functional Logic Programming with Distributed Constraint Solving" touched on various areas of research in Computer Science, closely related to declarative programming: computational models for functional logic programming (lazy narrowing calculi), constraint logic programming, and collaborative constraint solving. My PhD thesis was the outcome of my close cooperation with researchers from Research Institute for Symbolic Computation RISC-Linz, and Symbolic Computation Research Group (SCORE) from University of Tsukuba, Japan.

After defending my PhD thesis in 2000, I broadened my experience in constraint functional logic programming as a postdoc. I succeeded to obtain a 2-year postdoctoral fellowship (2000-2002) from the Japanese Society for the Promotion of Science (JSPS) at University of Tsukuba, Japan. The goal of my postdoctoral research was to adapt the scheme CFLP($\mathcal{X}, \mathcal{S}, \mathcal{C}$) proposed in my PhD thesis to open environments, which can be turned into hosts of a virtually unlimited number of solving services. The main achievement was Open CFLP, a system for collaborative constraint solving in open environments developed in close cooperation two researchers from the Symbolic Computation Research Group (SCORE) from University of Tsukuba: Professor Tetsuo Ida and his PhD student Norio Kobayashi. The system implemented an instance of the CFLP($\mathcal{X}, \mathcal{S}, \mathcal{C}$) scheme and was realised by integrating the symbolic computation capabilities of *Mathematica* with constraint solving

services deployed using CORBA middleware.

The architecture and capabilities of Open CFLP were reported in three journal articles [69, 91, 72].

A core component of Open CFLP are the solvers for the functional logic component of the system. To strengthen their capabilities, I worked on finding better higher-order lazy narrowing calculi for their execution model. My approach was to start from the first-order lazy narrowing calculi proposed for this purpose by the researchers of the SCORE group, and to collaborate with them to extend their calculi to the higher-order setting proposed at the end of the 1990s by Prehofer [124]. Our scientific results in this direction were published in two papers at conferences of type C [71, 70] and an accompanying technical report [105].

After finishing my postdoc studies, I continued research in symbolic computation (03/2003–09/2004) as researcher in the Symbolic Computation research group of the newly established Research Institute for Computational and Applied Mathematics (RICAM) of the Austrian Academy of Sciences. During this time, I established collaborations with researchers from the other four research groups of the RICAM institute, and strengthened my collaboration with the researchers involved in the development of the Theorema system at institute RISC-Linz. I continued the study of lazy narrowing as execution models for functional logic programming, and started research on a new direction: rule-based programming. My interest in rule-based programming started from the desire to provide programmatic support for strategies that control the collaboration between specialized theorem provers in Theorema to assist proof discovery. I soon realized that the system envisioned by me could have many more applications. The outcome of this research was the ρLog calculus and its implementation as a *Mathematica* adds-on package. My closest collaborators during my stay at RICAM were: Temur Kutsia from RISC-Linz institute, an expert in matching and unification procedures with sequence variables; Florina Piroi from RISC-Linz institute, who helped me extend ρLog with human-readable proof presentation capabilities by borrowing ideas and concepts from the design of the Theorema proof system; and Professor Aart Middeldorp from Innsbruck University, an expert in narrowing and lazy narrowing calculi. My research results with them were were published in 6 papers [85, 96, 103, 104, 96, 102]; the last one was presented at a B conference.

In October 2004 I became associate professor at the Department of Computer Science of Institute of Information Sciences and Electronics from University of Tsukuba, and held

that position until March 2011. My teaching duties included teaching the following courses: Mathematics for Computer Science, English in Technologies II, Information Processing, Advanced Topics in Term Rewriting, and Advanced Topics in Symbolic Computation. Besides teaching, I continued doing research on rule-based programming and closely related thematic directions, with funding from two JSPS grants:

1. April 2005 – March 2007: main investigator for

   JSPS Grant-in-Aid for Young Researchers (B)
   Project 17700025: *Rule-based Programming: Design and Applications*
   Budget (JPY): 1200 000 (2005), 1100 000 (2006)

   My scientific results during this period appeared in one journal article [97] and five other publications presented at: one A conference [78], two C conferences [89, 90], and two international workshops on unification [77, 80].

2. April 2008 – March 2011: main investigator for

   JSPS Grant-in-Aid for Scientific Research (C)
   Project 20500025: *Applications of rule-based programming to verification and transformation of XML*
   Budget (JPY): 1300 000 (2008), 1100 000 (2009), 1000 000 (2010)

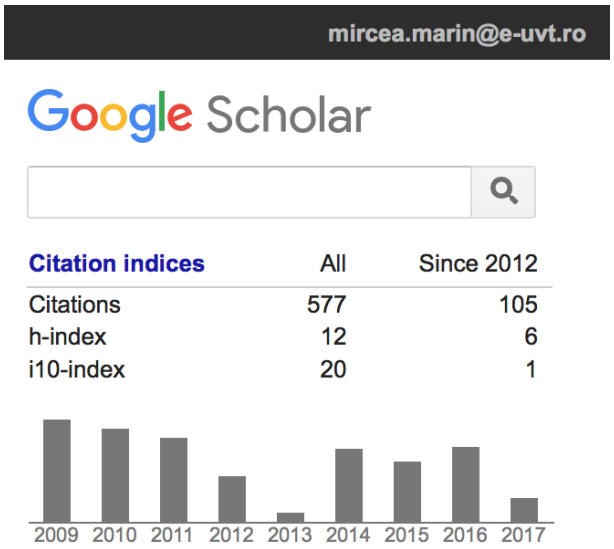
   My scientific achievements as associate professor affiliated with University of Tsukuba appeared in one journal article [100] and seven other publications presented at: one A conference [81], two B conferences [99, 101], two C conferences [86, 87], and two workshops [98, 39].

In 2011, I returned to Romania and started work as Scientific Researcher III at the Department of Computer Science from West University of Timisoara (April 2011 – September 2011). In October 2011, I became Assistant Professor at the Department of Computer Science of the Faculty of Mathematics and Informatics from West University of Timişoara, and since October 2015, I am Associate Professor at the same department. I teach topics such as Functional Programming, Advanced Functional and Logic Programming, Graph Theory and Combinatorics, and Advanced Data Structures.

At West University of Timişoara I became a member of the Theoretical Computer Science Group, whose research interests and scientific achievements can be checked at

`http://tcs.ieat.ro`. I found many common research interests with the members of this group, some of them new to me, including the study of exact and approximate learning algorithms from queries and counterexamples. I can mention here the discovery of a polynomial algorithm to learn cover context-free grammars from structural data. The algorithm was included in the proceedings of a type B conference [93], and also in an extended journal version [94]. I continue fruitful international collaborations with people interested in various aspects of declarative programming. Our recent achievements were published in the proceedings of an A conference [37], an A journal [38], and a B journal [82].

Google Scholar makes the following evaluation of my overall scientific production:



My other professional achievements include:

- Since 1999, I was every year a PC committee member of the international SYNASC symposia organised at West University of Timişoara. Also, since 2011, I am chair of the special track *Advances in the Theory of Computing* of SYNASC symposium.
- Local organizer of the Asian Symposium on Programming Languages and Systems (APLAS 2005) at University of Tsukuba, Japan.
- Steering committee member of the UNIF international meetings (2008-2013),
- Guest editor of a special issue on unification of the Logic Journal of IGPL.
- In 2014 and 2016: Visiting professor at Johannes Kepler University of Linz, in the frame of ERASMUS teaching mobilities. I gave lectures on advanced techniques of functional and logic programming, and rule-based programming.

- Editor of the proceedings of the 22nd International Workshop on Unification (UNIF 2008) organized in Hagenberg, Austria.

- Coauthor of the book "Principles of Functional and Logic Programming" published by Editura UVT in 2016.

1.2 Thematic directions of research

A constant topic of my research activities was the theoretical study of combinations of computational models for declarative programming, auxiliary algorithms (matching, unification, type reconstruction, etc.), and the implementation of systems based on my models.

I achieved significant results by following three thematic directions:

1. lazy narrowing as computational models for functional logic programming (FLP). I studied many classes of rewrite systems (higher order, conditional) for which we can define lazy narrowing calculi with essential properties (soundness, completeness, determinism) that make them practical for programming purposes.

2. a scheme for collaborative constraint functional logic programming (CFLP), and how to adapt it to an environment where constraint solvers are deployed dynamically as services which can be accessed remotely, and

3. rule-based programming from the perspective of integrating functional logic programming, advanced pattern matching features, constraint solving, and a declarative language to specify strategies that constrain the order of rule applications.

The first two directions are a continuation of the subject in my PhD thesis entitled "Functional Logic Programming with Distributed Constraint Solving" (2000). There, I proposed a scheme $\mathrm{CFLP}(\mathcal{X}, \mathcal{S}, \mathcal{C})$ for constraint functional logic programming, which describes the combination between: (1) a host language for FLP, whose behavior is driven by a lazy narrowing calculus $\mathcal{C}$, (2) a collection of solvers for subdomains of the constraint domain $\mathcal{X}$, and (3) a declarative language to specify a strategy $\mathcal{S}$ that defines a cooperation for the constraint solvers of the system. Schematically, I defined $\mathrm{CFLP}(\mathcal{X}, \mathcal{S}, \mathcal{C}) = \mathrm{FLP}(\mathcal{C}) + \mathrm{CP}(\mathcal{X}, \mathcal{S})$.

My first two thematic directions are strongly related with two important streams of development which flourished in the 1990 and are still very active:

- finding an adequate scheme for constraint functional logic programming [35, 41, 131, 130, 42, 108, 45, 40, 43, 60, 109, 110, 36].

- cooperative constraint solving, which aims to combine solvers acting on different admissible constraints in order to obtain a solver for systems of constraints that none of the individual solvers can handle alone [62, 129, 116, 117, 58, 59, 118, 44].

I became interested in rule-based programming in 2003, when I joined the Symbolic Computation research group of the RICAM institute of the Austrian Academy of Sciences. Starting from the observation that rule-based programming can be viewed as an extension of functional logic programming where rule application is constrained by strategies, I decided to elaborate this idea by drawing from my experience in functional logic programming, and by collaborating with my research colleagues from institutes RICAM and RISC-Linz in Austria. I received encouragements from the developers of the Theorema system at RISC-Linz institute, who were faced with a similar problem: providing good programmatic support for the strategies that govern the collaboration of specialized theorem provers in Theorema. This turned out to be a very rich line of research: first, I discovered the ρLog calculus and used it to implement a full-fledged system for rule-based programming on top of the basic rule-based programming and constraint solving capabilities of *Mathematica* [148], but soon afterwards we discovered several related subproblems, very interesting on their own right, that were waiting for algorithmic solutions. As a result, my third thematic direction spawned several related subdirections of research, like solving unification problems, matching problems, membership constraints, or type reconstruction problems.

The rest of this habilitation thesis is structured along the three thematic directions mentioned before. Chapter 2 presents our achievements in the study of lazy narrowing as computational models for FLP. Chapter 3 describes the outcome of our effort to adapt constraint functional logic programming scheme proposed in my PhD thesis to the capabilities of an open environment, where constraint solvers can be deployed dynamically as services that can be accessed remotely. Chapter 4 gives an account to our results (theoretical properties and practical implementations) to define a calculus for rule-based programming with strategies, and algorithms for related subproblems. In Chapter 5, I present my career development plan by focusing on academic and scientific objectives.

Chapter 2

Contributions to Functional Logic Programming

Functional logic programming (FLP) is a declarative programming style where the user specifies what he knows using a collection of rewrite rules (the program), and solves problems by posing asking questions expressed by systems of equations (the query) in the equational theory induced by his program. The underlying computational model is expected to compute a correct and, hopefully complete, set of answers to the queries posed by the user. This programming paradigm generalises both logic programming and functional programming (FP): Logic programs consist of Horn clauses which have a natural translation into conditional rewrite rules, whereas FP differs from FLP by solving a more restricted class of queries.

Functional logic programming owes its appeal to the wide range of applications that can be easily modeled in rewrite theories, and to the discovery of efficient execution principles that makes it relevant for practical implementations. (See [56] for a recent survey of these developments.) Users wish to have very general classes of rewrite systems (higher-order, conditional with various kinds of constraints, etc.), but these capabilities complicate significantly the design of a reliable and efficient model of computation.

A well-known method that can be turned in a computational model for FLP is narrowing. Discovered as a sound and complete method to solve unification problems in equational theories presented by confluent rewrite systems, it soon found applications in other other areas like functional logic programming [127, 7, 56], partial evaluation [2, 1], verification of cryptographic protocols [113], or program testing [21]. Formally, narrowing is an operation $t \leadsto^{\theta}_{\mathcal{R}} t'$ which specifies how we can reduce an object expressed by a term t with a rule cho-

sen from a collection of rules $\mathcal{R}$: we instantiate the variable parts of t with a substitution θ which creates a part (a.k.a. subterm) $t|_p$ of $t\theta$ that can be rewritten with a rule r from $\mathcal{R}$, and afterwards we replace $t|_p$ in $t\theta$ with the rewrite produced by rule r. Technically, attempts to narrow a term t with a rule from $\mathcal{R}$ are highly nondeterministic: we look at all non-variable parts of t that can be *unified* with the left-side l of a rule $(l \to r) \in \mathcal{R}$. When we find such a part $t|_p$ we compute the most general unifier θ of $t|_p$ with l, and perform the narrowing step $t \leadsto^{\theta}_{\mathcal{R}} t'$ where t' is the result of replacing the subterm $t\theta|_p = l\theta$ of $t\theta$ with $r\theta$.

Repeated applications of narrowing steps can be used to determine, for every input expressed by a term t, a set $\mathcal{A}$ of pairs $\langle \theta, v \rangle$ consisting of a substitution θ and a value v, such that the instantiation of t with θ, denoted by $t\theta$, has value v. This is related to what the computational model C of a functional logic programming language should do:

Given a *program* presented by a term rewrite system $\mathcal{R}$ for the functions used in the problem specification, and a *goal* described by a term t,

Find a set $Ans^C_{\mathcal{R}}(t)$ of pairs $\langle \theta, v \rangle$ where θ is a substitution for the variables in t and v is a value, such that the following conditions hold:

- Soundness: v is a value of $t\theta$ with respect to $\mathcal{R}$. In formal notation, $t\theta \to^*_{\mathcal{R}} v$.
- Completeness: for any substitution γ and value w such that $t\gamma \to^*_{\mathcal{R}} w$, there exist $\langle \theta, v \rangle \in Ans^C_{\mathcal{R}}(t)$ and a substitution σ such that $t\theta\sigma =_{\mathcal{R}} t\gamma$ and $v\sigma =_{\mathcal{R}} w$. Simply put, this means that $\langle \gamma, w \rangle$ is an $\mathcal{R}$-instantiation of a pair $\langle \theta, v \rangle \in Ans^C_{\mathcal{R}}(t)$.

The variables of t are called *logical variables*, and the elements of $Ans^C_{\mathcal{R}}(t)$ are the *answers* computed by C. If we define the set of *solutions* of t to be $Sol_{\mathcal{R}}(t) = \{\langle \theta, v \rangle \mid t\theta \to^*_{\mathcal{R}} v\}$, it follows that soundness holds when $Ans^C_{\mathcal{R}}(t) \subseteq Sol_{\mathcal{R}}(t)$, and completeness holds when every solution is an instantiation of a computed answer.

The connection between FLP and equational unification stems from the fact that functional logic languages restrict goals to systems of equations

$$G = s_1 \approx t_1 \;\&\&\; \ldots \;\&\&\; s_n \approx t_n \tag{2.1}$$

and solve them in the equational theory presented by $\mathcal{R}$ with the following approach:

- $\mathcal{R}$ is assumed to contain default rules to define the function symbols $\&\&$ and $\approx$ as boolean functions for logical conjunction and equality, such that $\{\theta \mid \langle \theta, v \rangle \in Sol_{\mathcal{R}}(E)\}$ is a complete set of $\mathcal{R}$-unifiers of G.

- they use a sound and complete calculus C to compute $Ans^C_{\mathcal{R}}(G)$.

A rule-based definition of `&&` as logical conjunction is straightforward, but there are many ways to define equality. The most common interpretations of $\approx$ are joinability ($s \approx t$ holds if $s \downarrow_{\mathcal{R}} t$), reducibility ($s \approx t$ holds if $s \rightarrow^*_{\mathcal{R}} t$), and strict equality ($s \approx t$ holds if there is a ground constructor term v such that $s \rightarrow^*_{\mathcal{R}} v$ and $t \rightarrow^*_{\mathcal{R}} v$).

Hullot [66] noticed that if $\mathcal{R}$ is a confluent and terminating rewrite system and $\approx$ is interpreted as joinability then the calculus which computes

$$\{\theta_1 \ldots \theta_n|_{vars(G)} \mid G \rightsquigarrow^{\theta_1}_{\mathcal{R}} \ldots \rightsquigarrow^{\theta_n}_{\mathcal{R}} G_n \quad \text{for some value } G_n\}$$

is a sound and complete method to solve equational goals of the form (2.1). This execution model is far from being adequate for functional logic languages, mainly because

1. It is highly inefficient due to the huge nondeterminism in selecting (1) the rewrite rule applicable in the next narrowing step, and (2) the part of the input that can be unified with the left side of the selected rule.

2. The class of rewrite systems considered by Hullot is to weak to model functional logic programs of practical interest.

These limitations have been largely eliminated by (1) extending the notion of narrowing to larger classes of rewrite systems, which are well suited for programming, and (2) defining viable strategies for these classes of rewrite systems, which reduce the aforementioned sources of high nondeterminism without losing completeness. Good surveys of the developments that laid the foundations of functional logic programming are [54, 56, 6].

Lazy narrowing calculi emerged as a viable alternative to narrowing strategies: They replace the rather complicated narrowing rule by a small set of more simple inference rules and reduce the high nondeterminism by imposing priorities among the inference rules applicable to a goal. In particular, lazy narrowing avoids the unification operation for θ in $t \rightsquigarrow^{\theta}_{\mathcal{R}} t'$ at the expense of simulating this step by applications of more primitive inference rules.

My contributions

I was interested to find suitable computational models for classes of rewrite systems with better support to represent problems in a neat and concise way. In functional logic programming, 'suitable' means sound, complete with respect to the set of solutions of interest, and reasonably efficient.

My objects of study were first-order conditional rewrite systems and various classes of higher-order rewrite systems in the simply-typed λ-calculus.

The starting point of my investigation was the first-order lazy narrowing calculus LNC and its various deterministic refinements [114], and my insight was that we can lift their nice properties (soundness, completeness w.r.t. various selection strategies) to higher-order counterparts. The higher-order lazy narrowing proposed by me were obtained in this way.

The presentation is organized in three subsections. The first one introduces our concepts notations, and the following two describe our calculi. These contributions have been published in [71, 70, 105, 102] and where achieved in close cooperation with members of the SCORE group from University of Tsukuba during my postdoc studies in Japan and shortly afterwards.

2.1 Preliminary notions

The inference rules of a lazy narrowing calculus C for a rewrite system $\mathcal{R}$ are expressions of the form $\dfrac{G_1, e, G_2}{G'\theta}$ where G_1, G_2, G' are sequences of equations, e is an equation, and θ is a substitution. The corresponding sequent style representation of this inference is

$$\frac{\mathcal{R} \vdash \exists \left(\bigwedge_{e' \in G_1} e' \wedge e \wedge \bigwedge_{e' \in G_2} e' \right)}{\mathcal{R} \vdash \exists \left([\theta] \wedge \bigwedge_{e' \in G'} e'\theta \right)}$$

where $[\theta]$ is the equational formula $\bigwedge_{x \in Dom(\theta)} (x \approx \theta(x))$ and $\exists F$ is the existential closure of a formula F. θ is the substitution *computed* by the inference rule.

Every application of an inference rule $\dfrac{G}{G'}$ with computed substitution θ corresponds to performing the C-step $G \overset{C}{\Rightarrow}_\theta G'$. We may write more subscripts, depending on the calculus C under consideration. A *C-derivation* is a sequence $G \overset{C}{\Rightarrow}_{\theta_1} G_1 \overset{C}{\Rightarrow}_{\theta_2} \ldots \overset{C}{\Rightarrow}_{\theta_n} G_n$ of C-steps, abbreviated $G \overset{C}{\Rightarrow}^*_\theta G_n$ where $\theta = \theta_1 \ldots \theta_n|_{vars(G)}$ is the composition of substitutions $\theta_1,\ldots,\theta_n$ restricted to the free variables of G. Note that we write $\theta_1 \ldots \theta_n$ for the functional composition $\theta_n \circ \ldots \circ \theta_1$, and $E\theta$ for the result of applying the homomorphic extension of θ to a syntactic construct (term, equation, goal) E.

To shorten the writing of lengthy sequences, I will abbreviate a sequence $s_1, \ldots, s_n$ by $\overline{s_n}$, and a sequence $s_m, s_{m+1}, \ldots, s_n$ by $\overline{s_{m,n}}$. E.g., $f(\overline{x_n})$ stands for $f(x_1, \ldots, x_n)$, and $\overline{s_n \approx t_n}$ stands for $s_1 \approx t_1, \ldots, s_n \approx t_n$. The empty sequence is denoted by $\Box$.

2.2 Lazy narrowing calculi for conditional rewrite systems

These calculi were introduced in the paper [102] presented by me at PPDP 2004. They are defined for first-order CTRSs and goals with the following syntax and semantics:

- Equations are of of three kinds: unoriented equations $s \approx t$ where $\approx$ is interpreted as joinability; oriented equations $s \triangleright t$ where $\triangleright$ is interpeted as reducibility; and `true`, which stands for an equation which trivially holds. We may write $s \triangleleft t$ instead of $t \triangleright s$, and $s \backsimeq t$ instead of $t \approx s$.

- A goal is a sequence of equations $e_1, \ldots, e_n$ which denotes their logical conjunction. The goal is *proper* if none of the component equations is `true`. We write $\Box$ for the empty goal, and $\top$ for any goal made of of `true` equations only.

- Rewrite rules are of the form $l \to r \Leftarrow c$ where s is a non-variable term, and the conditional part c is a proper goal. When $c = \Box$ we simply write $l \to r$.

A rewrite system $\mathcal{R}$ consisting of this kind of rules induces a conditional rewrite relation $\to_{\mathcal{R}}$ on goals, which can be defined as follows:

- $G_1, s \doteq t, G_2 \to_{\mathcal{R}} G_1, s[r\sigma]_p \doteq t, c\sigma, G_2$
  if $\doteq \in \{\approx, \triangleright, \backsimeq\}$, $(l \to r \Leftarrow c) \in \mathcal{R}$, and $s|_p = l\sigma$.

- $G_1, s \doteq s, G_2 \to_{\mathcal{R}} G_1, \mathtt{true}, G_2$ if $\doteq \in \{\approx, \triangleright\}$.

A solution of a goal G with respect to such a CTRS $\mathcal{R}$ is any substitution θ such that $G\theta \to_{\mathcal{R}}^* \top$. To solve proper goals, we started with an initial lazy narrowing calculus LCNC$_\ell$ and refined it to more deterministic versions. LCNC$_\ell$ consists of five inference rules:

[o] outermost narrowing: $\dfrac{f(\overline{s_n}) \doteq t, G}{\overline{s_n \triangleright l_n}, c, r \doteq t, G}$
if $\doteq \in \{\approx, \backsimeq, \triangleright\}$ and $f(\overline{l_n}) \to r \Leftarrow c$ is a fresh variant of a rule in $\mathcal{R}$.

[i] imitation: $\dfrac{f(\overline{s_n}) \doteq x, G}{(\overline{s_n \doteq x_n}, G)\theta}$
if $\doteq \in \{\approx, \backsimeq, \triangleright, \triangleleft\}$ and $\theta = \{x \mapsto f(\overline{x_n})\}$ with $x_1, \ldots, x_n$ fresh variables.

[d] decomposition: $\dfrac{f\overline{s_n}) \doteq f(\overline{t_n}), G}{\overline{s_n \doteq t_n}, G}$ if $\doteq \in \{\approx, \triangleright\}$.

[v] variable elimination: $\dfrac{s \doteq x, G}{G\theta} \quad \dfrac{x \doteq s, G}{G\theta}\, s \notin \mathcal{V}$
if $x \notin vars(s)$, $\doteq \in \{\approx, \triangleright\}$, and $\theta = \{x \mapsto s\}$.

[t] removal of trivial equations: $\dfrac{s \doteq s, G}{G}$ if $\doteq \in \{\approx, \triangleright\}$.

$\mathcal{V}$ stands here for the set of all variables. Every inference rule indicates that solving the goal above the line is reduced to solving the goal below, and may also mention a substitution θ computed by the reduction step. If θ is not specified, we assume implicitly that it is the empty substitution. If G is the goal above the line and G' the goal below the line, we may also depict the reduction step by $G \Rightarrow_\theta G'$. LCNC$_\ell$ solves a goal G by computing all derivations $G \Rightarrow_\theta^* \Box$, and reports the cosrresponding substitutions θ ad answers for G.

We have shown that LCNC$_\ell$ is sound [102, Theorem 1] but incomplete [102, Example 1]. To recover completeness, we imposed a natural extra condition called *determinism* on the syntactic structure of rewrite rules, and extended the notion of determinism to goals as well:

- If X is a set of variables and $G = e_1, \ldots, e_n$ is a proper goal, we say G is X-*deterministic* if the following condition holds for every equation e_i of G:

  if e_i is $s \approx t$ then $vars(e_i) \subseteq X \cup \bigcup_{j=1}^{i-1} vars(e_j)$, otherwise if e_i is $s_i \triangleright t_i$ then $vars(s_i) \subseteq X \cup \bigcup_{j=1}^{i-1} vars(e_j)$.

- Similarly, we say a CTRS $\mathcal{R}$ is *deterministic* if, for every rule $(r \to r \Leftarrow c) \in \mathcal{R}$, the conditional part c is $vars(l)$-deterministic.

Every proper goal G has a smallest set of variables $X \subseteq vars(G)$ for which it is X-deterministic. This set will be denoted by X_G in the following. We say that a solution θ of G is *normalized* if $x\theta$ is an $\mathcal{R}$-irreducible term for all $x \in X_G$.

Deterministic CTRSs are well suited for functional logic programming. E.g., the following deterministic CTRS provides an efficient way to compute Fibonacci numbers as integers represented in Peano arithmetic:

$$\begin{array}{ll} 0 + y \to y & \mathtt{s}(x) + y \to \mathtt{s}(x + y) \\ \mathtt{fst}(\langle x, y\rangle) \to x & \mathtt{snd}(\langle x, y\rangle) \to y \\ \mathtt{fib}(0) \to \langle 0, \mathtt{s}(0)\rangle & \mathtt{fib}(\mathtt{s}(x)) \to \langle y, y + z\rangle \Leftarrow \mathtt{fib}(x) \to \langle y, z\rangle \end{array}$$

and the one below is an easy-to-understand specification of quick-sort:

$$0 + y \to y \qquad \mathtt{s}(x) + y \to \mathtt{s}(x + y)$$

$$0 \leq x \to \mathtt{t} \qquad \mathtt{s}(x) \leq 0 \to \mathtt{f} \qquad \mathtt{s}(x) \leq s(y) \to x \leq y$$

$$\mathtt{split}(x, [y \mid z]) \to \langle u, [y \mid v]\rangle \Leftarrow x \leq y \triangleright \mathtt{t}, \mathtt{split}(x, z) \triangleright \langle u, v\rangle$$

$$\mathtt{split}(x, [y \mid z]) \to \langle u, [y \mid v]\rangle \Leftarrow x \leq y \triangleright \mathtt{f}, \mathtt{split}(x, z) \triangleright \langle u, v\rangle$$

$$\mathtt{append}([], y) \to y \quad \mathtt{append}([x|y], z) \to [x|\mathtt{append}(y, z)] \quad \mathtt{qsort}([]) \to []$$

$$\mathtt{qsort}([x|y]) \to \mathtt{append}(\mathtt{qsort}(u), [x \mid \mathtt{qsort}(v)]) \Leftarrow \mathtt{split}(x, y) \triangleright \langle u, v\rangle$$

It turns out that, if we assume $\mathcal{R}$ to be a deterministic CTRS and consider of interest only the normalized solutions of G, then LCNC$_\ell$ becomes sound and complete. Still, LCNC$_\ell$ is highly nondeterministic, and further refinements are needed in order to make it a viable model of computation for deterministic CTRSs.

First, we noticed that if we restrict LCNC$_\ell$ to apply inference rule [t] eagerly, and inference rule [v] eagerly to equations $x \triangleright t$ with $x \neq t$, we preserve completeness w.r.t. normalized solutions [102, Lemma 1]. Then, we defined a simple marking strategy to avoid computing (many) non-normalized solutions, and used it to define the calculus LCNC$^\dagger_\ell$.

The second refinement is based on a deep theoretical result: if $\mathcal{R}$ is a CTRS which satisfies the standardization theorem, then rule [v] can be applied eagerly to descendants of parameter-passing equations without loss of completeness. To enable this refinement, we extended LCNC$^\dagger_\ell$ to keep track of the descendants of parameter-passing equations, and added an extra restriction, called *freshness*, on the syntax of deterministic CTRS, that guarantees that standardization theorem holds. The resulted refinement is the calculus LCNC$^{\mathrm{eve}}_\ell$.

The most dramatic refinement is the calculus LCNC$^s_\ell$. It was obtained by adopting strictness to interpret equations. In the literature [8, 55] a substitution θ is called *strict solution of an unoriented equation* $s \approx t$ if $s\theta$ and $t\theta$ rewrite to the same ground term without defined symbols. We do not require groundness. We say that θ is a *strict solution of an oriented equation* $s \triangleright t$ if $s\theta \to^*_{\mathcal{R}} t\theta$ an $t\theta$ is a term without defined symbols. These notions are easily incorporated into the definition of conditional rewriting.

The inference rules of LCNC$^s_\ell$ are given below. Note that, in order to distinguish parameter-passing descendants from descendants of other oriented equations, we use the symbol $\blacktriangleright$ instead of $\triangleright$. We denote by $\mathcal{F}_D$ the set of defined function symbols, by $\mathcal{F}_C$ the set of symbols for data constructors, by $\mathcal{T}(\mathcal{F}_C, \mathcal{V})$ the set of terms made of data constructors and variables, and by $\mathrm{root}(t)$ the outermost symbol of a nonvariable term t.

[o] outermost narrowing: $\dfrac{f(\overline{s_n}) \doteq t, G}{\overline{s_n \triangleright l_n}, c, r \doteq t, G}$
if $\doteq \in \{\approx, \backsimeq, \triangleright, \blacktriangleright\}$ and $f(\overline{l_n}) \to r \Leftarrow c$ is a fresh variant of a rule in $\mathcal{R}$.

[i] imitation:

$$\frac{f(\overline{s_n}) \doteq x, G}{(\overline{s_n \doteq x_n}, G)\theta} \quad \text{if } \doteq \in \{\approx, \backsimeq, \triangleright\}, f \in \mathcal{F}_C, \text{ and } f(\overline{s_n}) \notin \mathcal{T}(\mathcal{F}_C, \mathcal{V})$$

$$\frac{f(s_1, \ldots, s_n) \blacktriangleright x, G}{(\overline{s_n \blacktriangleright x_n}, G)\theta}$$

where $\theta = \{x \mapsto f(\overline{x_n})\}$ with $x_1, \ldots, x_n$ fresh variables.

[d] decomposition:

$$\frac{g(\overline{s_n}) \doteq g(\overline{t_n}), G}{\overline{s_n \doteq t_n}, G} \qquad \text{if } \doteq \in \{\approx, \triangleright\} \text{ and } g \in \mathcal{F}_C$$

$$\frac{f(\overline{s_n}) \blacktriangleright f(\overline{t_n}), G}{\overline{s_n \blacktriangleright t_n}, G}$$

[v] variable elimination:

$$\frac{x \blacktriangleright s, G}{G\theta} s \notin \mathcal{V} \qquad \frac{s \doteq x, G}{G\theta} s \in \mathcal{T}(\mathcal{F}_C, \mathcal{V})$$

$$\frac{s \blacktriangleright x, G}{G\theta} \qquad \frac{x \doteq s, G}{G\theta}\, s \in \mathcal{T}(\mathcal{F}_C, \mathcal{V}) \setminus \mathcal{V}$$

if $x \notin vars(s)$, $\doteq \in \{\approx, \triangleright\}$, and $\theta = \{x \mapsto s\}$.

[t] removal of trivial equations:

$$\frac{s \doteq s, G}{G} \qquad \text{if } \doteq \in \{\approx, \triangleright\} \text{ and } s \in \mathcal{T}(\mathcal{F}_C, \mathcal{V})$$

$$\frac{s \blacktriangleright s, G}{G}$$

The strategy of priorities among the inference rules of LCNC^s_ℓ is illustrated in Table 2.1. Here, the symbol ";" separates (groups of) rules in decreasing order of priority. The subscript (1 or 2) in [o] indicates to which side (left or right) of the selected equation is applied the rule.

We have shown that, if we consider of interest only normalized strict solutions then the calculus LCNC^s_ℓ is complete for the class of deterministic CTRS [102, Theorem 5]. Also, we noticed that if we restrict ourselves to programs presented by left-linear fresh deterministic constructor CTRSs, then LCNC^s_ℓ becomes a fully deterministic calculus LCNC_d.

$s \approx t$

$\mathrm{root}(s)\backslash^{\mathrm{root}(t)}$	$\mathcal{F}_C$	$\mathcal{F}_D$	$\mathcal{V}$
$\mathcal{F}_C$	[t]; [d]	$[\mathrm{o}]_2$	[v]; [i]
$\mathcal{F}_D$	$[\mathrm{o}]_1$	$[\mathrm{o}]_1$	$[\mathrm{o}]_1$
$\mathcal{V}$	[v]; [i]	$[\mathrm{o}]_2$	[t]; [v]

$s \triangleright t$

$\mathrm{root}(s)\backslash^{\mathrm{root}(t)}$	$\mathcal{F}_C$	$\mathcal{F}_D$	$\mathcal{V}$
$\mathcal{F}_C$	[t]; [d]	×	[v]; [i]
$\mathcal{F}_D$	$[\mathrm{o}]_1$	[t]; $[\mathrm{o}]_1$	$[\mathrm{o}]_1$
$\mathcal{V}$	[v]	×	[t]; [v]

$s \blacktriangleright t$

$\mathrm{root}(s)\backslash^{\mathrm{root}(t)}$	$\mathcal{F}_C$	$\mathcal{F}_D$	$\mathcal{V}$
$\mathcal{F}_C$	[v]; [i]	$[\mathrm{o}]_2$	[t]; [v]
$\mathcal{F}_D$	$[\mathrm{o}]_1$	[t]; ($[\mathrm{o}]_1$, [d])	$[\mathrm{o}]_1$, [i], [v]
$\mathcal{V}$	[v]	[v]	[t]; [v]

Table 2.1: Selection strategy for the inference rules of LCNC_ℓ^s

	LCNC_ℓ			$\mathrm{LCNC}_\ell^{\mathrm{eve}}$			LCNC_d		
length	R	N	T	R	N	T	R	N	T
5	5	35	0.06	1	10	0.03	0	9	0.02
10	29	549	0.73	1	25	0.05	0	24	0.04
15	249	6837	12.63	1	39	0.07	0	38	0.06
20	281	51373	371.42	1	54	0.09	0	53	0.08
151	?	?	?	?	?	?	1	363	0.41

Table 2.2: Benchmarks for the goal $\mathtt{qsort}([\mathtt{s(0)}, \mathtt{0}, \mathtt{s(s(s(0)))}, \mathtt{s(s(0))}]) \approx x$

To assess the efficiency of our lazy narrowing calculi, we implemented them in ρLog, a system for rule-based programming implemented by us in *Mathematica* [97, 104], and ran benchmarks on a Windows PC with a 1.3GHz Centrino processor and 1 GB of RAM. Table 2.2 is an excerpt which illustrates how our lazy conditional narrowing calculi fare when computing strict normalized solutions for a goal with respect to the CTRS for the quicksort algorithm from page 19. Because the search space for solutions may be infinite, our implementation imposes a limit on the depth of the explored space (the first column of the table). The table indicates the number of nodes (N), time in seconds (T), and solutions found (R) for given depths. The "?" entries denote a timeout of 30 minutes. We observe a dramatic reduction of the search space for solutions as we move from LCNC_ℓ via $\mathrm{LCNC}_\ell^{\mathrm{eve}}$ to LCNC_d.

2.3 Higher-order lazy narrowing calculi

The results presented here are concerned with extensions of first-order lazy narrowing to the higher-order case, in a way that makes it useful for high-order functional logic programming. Our results were published in the paper [70] presented by us at the conference EUROCAST 2001 and in the final report of the Japanese Discovery Science project [71]. A more technical presentation of some of these results can be found in [105].

Pioneering achievements in this direction were reported in Prehofer's PhD thesis [124]. He showed how to adapt first-order lazy narrowing to the higher-order case as a framework for solving higher-order equations and found restricted classes of higher-order rewrite systems and equational goals that can form a basis for true high-order functional logic programming. As expected, the extension to the higher-order case introduces new sources of nondeterminism which complicate the identification of efficient strategies to find all solutions of interest for the new classes of programs and goals. The calculi proposed in [124] are LN and LNC. They are defined in the simply typed λ-calculus for goals made of oriented equations $s \to^? t$, and have good properties for functional logic programming (soundness, completeness, strategies which reduce significantly the huge space for solutions of interest) when programs are presented by certain classes of higher-order rewrite systems (HRS). Still, Prehofer's calculi are too nondeterministic to be of real practical use.

Our contributions in this direction can be summarised as follows:

1. We considered goals of the more general form $G\downharpoonright_W$ where W is a set of variables and G is a sequence of equations of two kinds:

   - oriented equations $(s \triangleright t)$, where $\triangleright$ is interpreted as reducibility, and
   - unoriented equations $s \approx t$, where $\approx$ is interpreted as joinability.

   Our logical interpretation of $G\downharpoonright_W$ with respect to a HRS $\mathcal{R}$ is

   $$\exists \left(\bigwedge_{(s \approx t) \in G} (s \downarrow_{\mathcal{R}} t) \wedge \bigwedge_{(s \triangleright t) \in G} (s \to^*_{\mathcal{R}} t) \right)$$

   whereas the operational interpretation is to compute a complete set of answer substitutions θ such that (1) $s\theta \downarrow_{\mathcal{R}} t\theta$ for all unoriented equations $s \approx t$ of G, (2) $s\theta \to^*_{\mathcal{R}} t\theta$ for all oriented equations $s \triangleright t$ of G, and (3) $X\theta$ is $\mathcal{R}$-irreducible whenever $X \in W$.

2. To solve goals of this kind, we developed a family of higher-order lazy narrowing calculi. They are refinements of Prehofer's calculus LN that perform better because we succeeded to lift deterministic refinements of the first-order lazy narrowing calculus LNC [115, 114] to the higher-order case.

The rest of this section is structured as follows. First, I introduce some preliminary notions. Next, I present the calculi proposed by us as better alternatives to Prehofer's calculi.

2.3.1 Preliminary notions

Our framework of study is the simply-typed λ-calculus, where terms are represented in their long $\beta\eta$-normal form and equality is assumed modulo α-convertibility. We denote by $\mathcal{F}$ the set of function symbols, by $\mathcal{V}$ the set of variables used in the construction of terms, and by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ the set of all simply-typed λ-terms. Also, we use the following naming conventions:

F, G, X, Y, Z, H, T, possibly subscripted, are free variables,
x, y, z, possibly subscripted, are bound variables,
f, g, h, c, possibly subscripted, are function symbols,
τ, τ', possibly subscripted, are simple types,
b is a base type,
s, t, u, v, l, r, possibly primed or subscripted, are terms.

The general form of a term in long $\beta\eta$-normal form is $\lambda\overline{x_n}.v(\overline{t_m})$ where $v(\overline{t_m})$ is a term of base type, and v is either a variable or a function symbol. Such a term is *flex* if v is a free variable, and *rigid* otherwise.

We write $BVars(t)$ for the bound variables in s, and $FV(t)$ for the free variables in s. A *relaxed higher-order pattern* is a term t with the property that all free variables in s have bound variables as arguments, i.e., if $X(\overline{s_n})$ is a subterm of t then $s_i{\downarrow_\eta} \in BV(t)$ for $1 \leq i \leq n$. A *(higher-order) pattern* is a relaxed pattern where the arguments of free variables are distinct bound variables. For instance, $\lambda x.F(x, \lambda z.y(z), x)$ is a relaxed pattern which is not a pattern, and $\lambda x, y.G(x, \lambda z.y(z))$ is a pattern. A pattern is *fully extended* if, for every subterm of the form $X(\overline{s_n})$, $\overline{s_n{\downarrow_\eta}}$ is the sequence of all bound variables in the scope of the subterm. A pattern is *linear* if no free variable occurs more than once in it.

Substitutions are mappings $\theta : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ that map variables to terms of the same type as them, and satisfy the condition that $Dom(\theta) := \{X \in \mathcal{V} \mid \theta(X){\downarrow_\eta} \neq X\}$ is a finite

set. Thus, every substitution θ can be represented completely as a set $\{X_1 \mapsto \theta(X_1), \ldots, X_n \mapsto \theta(X_n)\}$ where $\{X_1, \ldots, X_n\} = Dom(\theta)$. θ has a unique homomorphic extension θ^* on terms, and we write $t\theta$ instead of $\theta^*(t)$. Also, we prefer to write $\theta_1\theta_2$ for the functional composition $\theta_2 \circ \theta_1$ of substitutions θ_1, θ_2.

A *higher-order rewrite system (HRS in [124], PRS in [70, 71])* is a set of rewrite rules of the form $f(\overline{l_n}) \to r$ where $f(\overline{s_n})$ is a pattern, $f(\overline{l_n})$ and r have the same base type, and $FV(r) \subseteq FV(f(\overline{l_n}))$. The reasons for these syntactic restrictions stem from the facts that (1) patterns have a most general unifier (mgu) which can be computed in linear time [125], and (2) they simplify the definition of term rewriting [121] and narrowing to be similar to the first-order case. Still, a small technicality is needed: in order to rewrite or narrow a subterm $s|_p$ with a rule $l \to r$, we must $\overline{x_k}$-lift $l \to r$, where $\overline{x_k}$ are the bound variables in the scope of $s|_p$ (see details in [124]). Here, we only need to say what is an $\overline{x_k}$-lifter of a rewrite rule $l \to r$: It is the rewrite rule $l\rho \to r\rho$ where $\rho = \{\overline{X_n \to Y_n(\overline{x_k})\uparrow^\eta}\}$ where $Y_1, \ldots, Y_n$ are distinct fresh variables of appropriate types.

A PRS is *left-linear* (resp. *fully-extended*) if it consists of rewrite rules whose left side is a linear (resp. fully extended) pattern. We abbreviate with EPRS a fully-extended PRS, and with LEPRS a left-linear EPRS. Like in the first-order, case, we distinguish the set $\mathcal{F}_D$ of defined function symbols and the set $\mathcal{F}_C$ of data constructors. We assume $\mathcal{F} = \mathcal{F}_C \cup \mathcal{F}_D$, $\mathcal{F}_C \cap \mathcal{F}_D = \emptyset$, and allow to define by rewrite rules only symbols from $\mathcal{F}_D$.

Equations are pairs of terms s, t of the same type. They are either oriented $s \triangleright t$, or unoriented $s \approx t$. We may write $s \backsimeq t$ instead of $t \approx s$, and $s \triangleleft t$ instead of $t \triangleright s$. An equation is *flex*, or *immediately solvable*, if it is between flex terms. We consider goals of the form $G\downharpoonright_W$ where G is a sequence of such equations and W is a set of variables. A goal $G\downharpoonright_W$ is *immediately solvable* if al equations of G are immediately solvable.

For a given PRS $\mathcal{R}$, the set $Sol_\mathcal{R}(G\downharpoonright_W)$ of solutions of interest for $G\downharpoonright_W$ consists of the substitutions θ such that (1) $s\theta \downarrow_\mathcal{R} t\theta$ for all $(s \approx t) \in G$, (2) $s\theta \to^*_\mathcal{R} t\theta$ for all $(s \to t) \in G$, and (3) $\theta(X)$ is $\mathcal{R}$-normalized for all $X \in W$. When viewed as solutions of a goal whose logical variables are from a set W, it is useful to say which one is more general than another. For this purpose, we define the relation $\theta \leq \theta'\ [W]$ if there exists a substitution γ such that $\theta'(X) = \theta(X)\gamma$ for all $X \in W$. A *complete* set of solutions of $G\downharpoonright_W$ is a subset $M \subseteq Sol_\mathcal{R}(G\downharpoonright_W)$ such that, for every $\theta \in Sol_\mathcal{R}(G\downharpoonright_W)$ there exists $\sigma \in M$ with $\sigma \leq \theta\ [FV(G)]$. In general, it is unreasonable to look for a calculus C which computes a

complete set of solutions. It is widely accepted that:

n_1) Computing a complete set of unifiers for solvable goals is unreasonable, due to the high search space for them. However, immediately solvable goals always have solutions. and knowing this fact is often sufficient. (E.g., in the design of theorem provers which check the satisfiability of equational goals.)

n_2) It is more reasonable to compute a set PM of pairs $\langle \sigma, F\lfloor_{W'} \rangle$ where $F\lfloor_{W'}$ is immediately solvable, such that for every $\theta \in Sol_{\mathcal{R}}(G\lfloor_W)$ there exist $\langle \sigma, F\lfloor_{W'} \rangle \in PM$ and $\gamma \in Sol_{\mathcal{R}}(F\lfloor_{W'})$ with $\sigma\gamma \leq \theta\ [FV(G)]$. If we were able to solve all the immediately solvable components of these pairs, we would get a complete set of solutions of $G\lfloor_W$.

Starting from this state of facts, we designed higher-order lazy narrowing calculi which fulfil condition n_2). All calculi C consist of a small collection of inference rules which induce a reduction relation $G\lfloor_W \overset{C}{\Rightarrow}_{[\alpha],\theta} G'\lfloor_{W'}$, abbreviated $G\lfloor_W \Rightarrow_\theta G'\lfloor_{W'}$, where $[\alpha]$ is the name of the applied inference rule, and θ is the substitution inferred by the application. They produce a set PM of pairs $\langle \sigma, F\lfloor_{W'} \rangle$ corresponding to derivations $G\lfloor_W \overset{C}{\Rightarrow}^*_\theta F\lfloor_{W'}$. The search space for such derivations is huge, mainly because of the *don't know* nondeterminism in choosing which inference rule to apply next. This issue is addressed by attaching a strategy to every calculus C, which imposes priorities among the inference rules applicable to a goal and discards the need to apply the rules with low priority.

2.3.2 HOLN_0

This was the starting point of our investigation. HOLN_0 is designed to work for programs presented by confluent PRSs and consists of three groups of inference rules: pre-unification rules, lazy narrowing rules, and removal rules of flex equations.

Pre-unification rules

[i] Imitation
$(G_1, \lambda\overline{x_k}.X(\overline{s_m}) \approxeq \lambda\overline{x_k}.g(\overline{t_n}), G_2)\lfloor_W \Rightarrow_{[i],\theta} (G_1, \overline{\lambda\overline{x_k}.H_n(\overline{s_m}) \approxeq \lambda\overline{x_k}.t_n}, G_2)\theta\lfloor_{W\cup\{\overline{H_n}\}}$
where $\approxeq \in \{\approx, \not\approx, \rhd, \lhd\}$, $\theta = \{X \mapsto \lambda\overline{y_m}.g(\overline{H_n(\overline{y_m})})\}$

[p] Projection: If $\lambda\overline{x_k}.t$ is rigid, $m \geq 1$, and s_i and t have same return type, then
$(G_1, \lambda\overline{x_k}.X(\overline{s_m}) \approxeq \lambda\overline{x_k}.t, G_2)\lfloor_W \Rightarrow_{[p],\theta} (G_1, \overline{\lambda\overline{x_k}.X(\overline{s_m}) \approxeq \lambda\overline{x_k}.t}, G_2)\theta\lfloor_{W\cup\{\overline{H_n}\}}$
where $\approxeq \in \{\approx, \not\approx, \rhd, \lhd\}$, $\theta = \{X \mapsto \lambda\overline{y_m}.y_i(\overline{H_n(\overline{y_m})})\}$.

[d] Decomposition: If $v \in \mathcal{F} \cup \{\overline{x_k}\}$ and $\trianglerighteq \in \{\approx, \not\approx, \triangleright\}$ then
$(G_1, \lambda\overline{x_k}.v(\overline{s_n}) \trianglerighteq \lambda\overline{x_k}.v(\overline{t_n}), G_2)\downarrow_W \Rightarrow_{[d],\{\}} (G_1, \overline{\lambda\overline{x_k}.s_n \trianglerighteq \lambda\overline{x_k}.t_n}, G_2)\downarrow_W$

Lazy narrowing rules

[on] Outermost narrowing at non-variable position:
If $f(\overline{l_n}) \to r$ is an $\overline{x_k}$-lifted rule of $\mathcal{R}$ and $\trianglerighteq \in \{\approx, \not\approx, \triangleright\}$ then
$(G_1, \lambda\overline{x_k}.f(\overline{s_n}) \trianglerighteq \lambda\overline{x_k}.t, G_2)\downarrow_W \Rightarrow_{[on],\{\}} (G_1, \overline{\lambda\overline{x_k}.s_n \triangleright \lambda\overline{x_k}.l_n}, \lambda\overline{x_k}.r \trianglerighteq \lambda\overline{x_k}.t, G_2)\downarrow_W$

[ov] Outermost narrowing at variable position
If $f(\overline{l_n}) \to r$ is an $\overline{x_k}$-lifted rule of $\mathcal{R}$, $\trianglerighteq \in \{\approx, \not\approx, \triangleright\}$, and $\begin{cases} \lambda\overline{x_k}.X(\overline{s_m}) \text{ is not a pattern} \\ \text{or} \\ X \notin W \end{cases}$
then
$(G_1, \lambda\overline{x_k}.X(\overline{s_m}) \trianglerighteq \lambda\overline{x_k}.t, G_2)\downarrow_W$
$\Rightarrow_{[ov],\theta} (G_1\theta, \overline{\lambda\overline{x_k}.H_n(\overline{s_m\theta}) \triangleright \lambda\overline{x_k}.l_n}, \lambda\overline{x_k}.r \trianglerighteq \lambda\overline{x_k}.t\theta, G_2\theta)\downarrow_{W\cup\{\overline{H_n}\}}$
where $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$

Removal rules

[t] Trivial equation: If $\cong \in \{\approx, \triangleright\}$ then
$(G_1, \lambda\overline{x_k}.X(\overline{s_m}) \cong \lambda\overline{x_k}.X(\overline{s_m}), G_2)\downarrow_W \Rightarrow_{[t],\{\}} (G_1, G_2)\downarrow_W$

[fs] Flex same: If $\cong \in \{\approx, \triangleright\}$ and $X \in W$ then
$(G_1, \lambda\overline{x_k}.X(\overline{y_n}) \cong \lambda\overline{x_k}.X(\overline{y'_n}), G_2)\downarrow_W \Rightarrow_{[fs],\theta} (G_1, G_2)\theta\downarrow_{W\cup\{H\}}$
where $\theta = \lambda\overline{y_n}.H(\overline{z_p})$ with $\{\overline{z_p}\} = \{y_i \mid y_i = y'_i, 1 \le i \le n\}$.

[fd] Flex different: If $\begin{cases} X \in W \text{ and } T \in W & \text{when } \cong \text{ is } \approx \\ X \in W & \text{when } \cong \text{ is } \triangleright \end{cases}$ then
$(G_1, \lambda\overline{x_k}.X(\overline{y_m}) \cong \lambda\overline{x_k}.Y(\overline{y'_n}), G_2)\downarrow_W \Rightarrow_{[fd],\theta} (G_1, G_2)\downarrow_{W\cup\{H\}}$
where $\theta = \{X \to \lambda\overline{y_m}.H(\overline{z_p}), Y \to \lambda\overline{y'_n}.H(\overline{z_p})\}$ and $\{\overline{z_p}\} = \{\overline{y_m}\} \cap \{\overline{y'_n}\}$.

This calculus is highly nondeterministic and we did not impose any restriction on the selection of applicable rules. We showed that HOLN_0 is sound and complete. In fact, HOLN_o is strongly complete, where strong completeness means that the selection of equations in goals is irrelevant. Thus, an implementation is free to use any equation selection strategy without fear of losing completeness.

2.3.3 HOLN_1

This calculus is obtained from HOLN_0 by keeping track of the descendants of parameter-passing equations and imposing the strategy which does not apply rule [on] to descendants o parameter-passing equations $\lambda\overline{x_k}.f(\overline{s_n}) \triangleright \lambda\overline{x_k}.X(\overline{y_k})$. Descendants of parameter-passing equations are oriented equations introduced by applications of the rules [on] and [ov], and we keep track of them in exactly the same way as the first-order calculus LCNC, by using the symbol $\blacktriangleright$ instead of $\triangleright$. Thus, HOLN_1 is obtained from HOLN_0 as follows:

- with the exception of rule [on], all the other rules applicable to an equation $s \triangleright t$ are extended to apply to $s \blacktriangleright t$ in the same way.

We have shown that HOLN_1 is sound and strongly complete for confluent LEPRS [71, Theorem 3]. The proof of this result is based on the fact that the standardization theorem holds for confluent LEPRSs [142]. This result is the higher-order version of the eager-variable elimination strategy for the first-order lazy narrowing calculus LNC [115].

2.3.4 HOLN_2

This is a specialisation of HOLN_0 for confluent constructor LEPRS, i.e., confluent LEPRSs made of rules $f(\overline{l_n}) \to r$ with no defined symbols in $l_1, \ldots, l_n$. The calculus HOLN_2 has

1. a new inference rule [c] for parameter-passing descendants, defined as follows:

   [c] Constructor propagation:
   If $\trianglerighteq \in \{\approx, \backsimeq, \triangleright, \blacktriangleright\}$, $\exists s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k}) \in G_1$ and $s' = \lambda\overline{y_k}.s(\overline{x_k})$ then
   $(G_1, \lambda\overline{z_n}.X(\overline{t_k}) \trianglerighteq \lambda\overline{z_n}.u, G_2)\downarrow_W \Rightarrow_{[c],\{\}} (G_1, \lambda\overline{z_n}.s'(\overline{t_k}) \trianglerighteq \lambda\overline{z_n}.u, G_2)\downarrow_W$

2. a strategy which (1) always selects the leftmost equation on which a rule can act, and (2) gives the highest priority to rule [c].

This calculus is more deterministic than HOLN_0 because there are fewer rules applicable to descendants of parameter-passing equations: Their right sides will never contain defined symbols and therefore we will never have to choose between [on] and [d] when selecting them. We have shown that if $\mathcal{R}$ is a confluent LEPRS then HOLN_2 with this strategy is sound complete. Moreover, the calculus is also sound when restricted to goals without oriented equations [71, Theorem 4].

2.3.5 HOLN_3

This is a refinement of HOLN_2 which eliminates the effort of solving certain descendants of parameter-passing equations by simply removing them from the goal. Such equations are called *redundant*: $\lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k})$ is redundant in $(G_1, \lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k}), G_2)\lfloor_W$ if $\overline{y_k}$ is a permutation of $\overline{x_k}$ and $X \notin FV(G_1, \lambda\overline{x_k}.s, G_2)$. HOLN3 extends HOLN_2 with

[rm] Removal or redundant equations.

If $\overline{y_k}$ is a permutation of $\overline{x_k}$ and $X \notin FV(G_1, \lambda\overline{x_k}.s, G_2)$ then
$(G_1, \lambda\overline{x_k}.s \blacktriangleright \lambda\overline{x_k}.X(\overline{y_k}), G_2)\lfloor_W \Rightarrow_{[rm],\{\}} (G_1, G_2)\lfloor_W$

ad give to [c] priority over the other applicable rules. We have shown that HOLN_3 is sound and complete for confluent LEPRSs [71, Theorem 5].

2.4 Applications and further extensions

We studied two further extensions of HOLN and its refinements to make them useful for true high-order functional logic programming:

1. Programs presented by conditional rewrite systems (CTRS). This extension is motivated by the fact that the most natural way to define functions is by expressing the computation of the result in terms of a sequence of conditions that must be fulfilled to produce it, and CTRSs provide the means to do so. (A good example is the first order CTRS for quick-sort from page 19.)

2. Integration with solvers for constraints over various constraint domains. This extension has the same motivation as the design of schemes for constraint logic programming: to boost the performance of a declarative programming language with support from high-performance solver for domain-specific constraints.

Our achievements in this area have materialized in more powerful implementations of the system `CFLP` described in my PhD thesis. In [92, 70, 69, 72], we described the capabilities of refined implementations of `CFLP`. Their model of computation is an instance of my CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$ scheme which integrates

1. An interpreter for FLP which can be configured to work with one of the higher-order lazy narrowing calculi HOLN_i proposed by us. In addition, we implemented a higher-order extension of the first-order lazy narrowing calculus LCNC [52],

2. Specialized solvers for systems of linear equations, equations with invertible functions, systems of multivariate polynomial equations, and differential equations.

I defer to Chapter 3 the description of the capabilities brought by the integration of higher-order FLP with constraint solving and conclude this section with an interesting application which can be solved with an adjustment of HOLN: program transformation. This problem is concerned with the transformation of programs made of inefficient but easily understandable function definitions into programs that are more efficient.

The following example is taken from [71]: It shows how to use a clever adjustment of HOLN to implement a program transformation technique called *fusion*. Consider the problem of writing a program to check whether a list of numbers is steep, i.e, if each element is greater than or equal to the average of the elements that follow it. (By default, the empty list is steep.) We can do this with the program consisting of the rewrite rules

$$\begin{array}{ll} \texttt{steep}([\,]) \to \texttt{true} & \texttt{steep}(\texttt{cons}(a, X)) \to (a * \texttt{len}(X) \geq \texttt{sum}(X) \wedge \texttt{steep}(X) \\ \texttt{sum}([\,]) \to \texttt{0} & \texttt{sum}(\texttt{cons}(X, Y)) \to X + \texttt{sum}(Y) \\ \texttt{len}([\,]) \to \texttt{0} & \texttt{len}(\texttt{cons}(X, Y)) \to \texttt{1} + \texttt{len}(Y) \end{array}$$

where $[\,]$ is the empty list, $\texttt{cons}(s, t)$ is a list with head s and tail t, and $\texttt{c3} \in \mathcal{F}_C$ has function type $\texttt{bool} \to \texttt{float} \to \texttt{float} \to \texttt{tuple}$. To improve readability, we write terms built with the binary operators $+, -, \geq, \wedge$ in infix notation. These operators are properly interpreted by specialised constraint solvers, but the FLP component of `CFLP` treats them as data constructors. This definition of `steep` is straightforward but inefficient (quadratic complexity). It is possible—and desirable—to automatically compute an efficient (linear complexity) version `steepOptimal` of `steep`. To do so, we use the fusion calculation rule which stipulates that

if the functions $f : \texttt{list}(\alpha) \to \beta$, $g : \alpha \to \texttt{list}(\alpha) \to \texttt{list}(\alpha)$ and $h : \alpha \to \beta \to \beta$ satisfy the equations

$f(e) = e'$ for some values e and e', and $\forall H : \alpha, T : \texttt{list}(\alpha).f(g(H, T)) = h(H, f(T))$

then the following equality also holds: $\forall T : \texttt{list}(\alpha).f(\texttt{foldr}(g, e, T)) = \texttt{foldr}(h, e', T)$.

Here, $\texttt{foldr} : (\alpha \to \beta \to \beta) \to \beta \to \texttt{list}(\alpha) \to \beta$ is the folding function defined as usual:

$$\texttt{foldr}(\lambda x, y.F(x, y), X, [\,]) \to X$$

$$\texttt{foldr}(\lambda x, y.F(x,y), X, \texttt{cons}(H,T)) \rightarrow F(H, \texttt{foldr}(\lambda x, y.F(x,y), X, T))$$

Note that, if $g = \texttt{cons}$ and $e = [\,]$ then

$$\forall T : \texttt{list}(\alpha) : \texttt{foldr}(g, e, T) = \texttt{foldr}(\texttt{cons}, [\,], T) = T$$

and fusion provides a new way to define f, by $f = \lambda x.\texttt{foldr}(h, e', x)$.

We can make use of this rule by observing that:

1. the computation of $\texttt{steep}(\texttt{cons}(h,t))$ requires the computation of three additional quantities: $\texttt{steep}(t)$, $\texttt{sum}(t)$, and $\texttt{len}(t)$. Thus, the defining rules of `steep`, `sum` and `len` specify the computation of the function $f = \lambda x.\texttt{c3}(\texttt{steep}(x), \texttt{sum}(x), \texttt{len}(x))$ where $\texttt{c3} \in \mathcal{F}_C$ is of type $\texttt{bool} \rightarrow \texttt{float} \rightarrow \texttt{float} \rightarrow \texttt{tuple}$.

2. If we assume $e = [\,]$, $e' = \texttt{c3}(\texttt{true}, 0, 0)$, $g = \texttt{cons}$, and manage to find a function $h : \texttt{float} \rightarrow \texttt{tuple}$ such that $\lambda x, y.f(g(x,y)) = \lambda x, y.h(x, f(y))$ then, according to the fusion calculation rule, we can take the function $\lambda x.\texttt{foldr}(h, e', x)$ as an alternative definition of function f.

We can find h by solving the goal $\lambda x, y.f(g(x,y)) \approx \lambda x, y.H(x, f(y))$ with respect to the program for `steep`, `sum`, and `len`. HOLN can solve this goal if we adjust it as follows:

- For every $c \in \mathcal{F}_C$ of type $\overline{\tau_n} \rightarrow b$ we presuppose the existence of n defined functions $\texttt{sel} \cdot c \cdot i : b \rightarrow \tau_i$, called the *selectors* of c, for which the following equalities hold:

$$c(\overline{\texttt{sel} \cdot c \cdot n(t)}) = t \quad \text{and} \quad \texttt{sel} \cdot c \cdot i(c(\overline{t_n})) = t_i \;\text{ for all } 1 \leq i \leq n.$$

  We support these requirements by replacing implicitly all terms $c(\overline{\texttt{sel} \cdot c \cdot n(t)})$ with t, and by extending the programs under consideration with the rewrite rules

$$\texttt{sel} \cdot c \cdot i(c(\overline{X_i})) \rightarrow X_i \qquad \text{for all } 1 \leq i \leq n.$$

- We add a new inference rule:

  [fl] Flattening: If t is a rigid term and $\cong \in \{\approx, \backsimeq, \triangleright, \triangleleft\}$ then
  $(G_1, \lambda\overline{x_k}.H(\overline{s_m}, c(\overline{t_n}), \overline{u_p}) \cong \lambda\overline{x_k}.t, G_2)\lfloor_W \Rightarrow_{[fl],\theta}$
  $(G_1\theta, \lambda\overline{x_k}.H(\overline{s_m\theta}, H_n(\overline{x_k}), \overline{u_p\theta}) \cong \lambda\overline{x_k}.t\theta, G_2\theta)\lfloor_{W\cup\{H\}}$
  where $\theta = \{X \mapsto \lambda\overline{x_m}, y, \overline{z_p}.H(\overline{x_m}, \overline{\texttt{sel} \cdot c \cdot n(y)}, \overline{z_p})\}$

With these adjustments, our higher order lazy narrowing calculus finds the substitution

$$\begin{aligned}\theta = \{H \mapsto \lambda x_1, x_2.\mathtt{c3}(\ & x_1 * \mathtt{sel} \cdot \mathtt{c3} \cdot \mathtt{3}(x_2) \geq \mathtt{sel} \cdot \mathtt{c3} \cdot \mathtt{2}(x_2) \wedge \mathtt{sel} \cdot \mathtt{c3} \cdot \mathtt{1}(x_2), \\ & x_1 + \mathtt{sel} \cdot \mathtt{c3} \cdot \mathtt{2}(x_2), \\ & 1 + \mathtt{sel} \cdot \mathtt{c3} \cdot \mathtt{3}(x_2))\}\end{aligned}$$

as unique solution of the goal

$$\begin{aligned}&(\lambda x, y.\mathtt{c3}(\mathtt{steep}(\mathtt{cons}(x, y)), \mathtt{sum}(\mathtt{cons}(x, y)), \mathtt{len}(\mathtt{cons}(x, y))) \\ &\approx \lambda x, y.H(x, \mathtt{c3}(\mathtt{steep}(y), \mathtt{sum}(y), \mathtt{len}(y)))\downarrow_{\{H\}}\end{aligned}$$

Thus, $f = \lambda x.\mathtt{c3}(\mathtt{steep}(x), \mathtt{sum}(x), \mathtt{len}(x))$ and $\lambda x.\mathtt{foldr}(H\theta, e', x)$ are functions that compute the same values. It follows that `steep` computes the same values as the function `steepOptimal` which is $\lambda x.\mathtt{sel} \cdot \mathtt{c3} \cdot \mathtt{1}(\mathtt{foldr}(H\theta, e', x))$.

It is easy to see that the function `steepOptimal` has a definition with linear complexity, therefore it is preferable to use `steepOptimal` instead of `steep`.

Chapter 3

Contributions to Constraint Functional Logic Programming

Constraint functional logic programming (CFLP) boosts the expressive power and performance of functional logic programming with specifications of constraints of various kinds, and with the possibility to solve them efficiently with specialised constraint solvers. The first proposal to achieve CFLP = FLP+CP by the smoothless amalgamation of the principles of functional logic programming (FLP) with constraint programming (CP) emerged at the beginning of the 1990s [35] and was followed in the next two decades by a flurry of improved schemes and implementations [41, 131, 130, 84, 42, 108, 45, 40, 43, 60, 109, 110, 36].

Formally, constraint functional logic programs are like functional programs, but the conditional part of rewrite rules can contain constraints in addition to regular equations whose meaning can be inferred from the program only. Typical examples are membership constraints and arithmetic relations (equalities, inequalities) expressed in terms of various kinds of functions (polynomial, trigonometric, differential). The proper interpretation of constraints is handed over to specialized constraint solvers capable to solve them efficiently.

My contributions

During my postdoc studies, I extended the scheme CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$ described in my PhD thesis to an open model which allows to access and use a practically unlimited variety of constraint solvers which are deployed as solving services in an open environment (the Internet) through standardised protocols. For each problem, the user can (re)configure the constraint solving component of the system to allocate access and communicate only with certain solv-

ing services over the constraint domain $\mathcal{X}$, and can specify a strategy $\mathcal{S}$ that indicates how the domain-specific solvers collaborate to solve the given problem.

In [72] we mentioned the following reasons for trying to adapt the scheme CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$ to a service model for constraint solving capabilities:

1. solvers evolve over time; sophisticated algorithms may require long time efforts for their perfection,

2. solvers may be big and require specialised resources which are not downloadable,

3. solvers are willing to provide services, but not necessarily willing to allow the clients to copy the programs to protect the intelligent and copy rights,

4. solvers and the clients that use those services are independent and only communicate with through standardised protocols, and

5. services may be deployed dynamically, independent of the plan of the potential clients.

The new scheme is more general than other CFLP scheme proposals: most of them describe the integration of a (lazy) narrowing calculus with one or a small number of domain-specific constraint solvers, and the collaboration between the system components is, usually, built-in.

For this direction of research I received a 2-year postdoctoral fellowship (2000-2002) awarded by the Japanese Society for the Promotion of Science. I contributed to the realization of two implementations which were deployed as Add-On packages of the *Mathematica*:

1. Open CFLP, a system produced by a close cooperation with two researchers from the Symbolic Computation Research Group (SCORE) from University of Tsukuba: Professor Tetsuo Ida and his PhD student Norio Kobayashi. Open CFLP was implemented using the CORBA middleware technology.

2. A Jini service for collaborative constraint solving. This implementation was produced while I worked as researcher in the Symbolic Computation group of the RICAM institute affiliated with the Austrian Academy of Sciences (2003-2004).

The rest of this chapter is structured as follows. The next two sections describe the architectures and capabilities of our implementations: Open CFPL (Sect. 3.1) and the Jini service for collaborative constraint solving (Sect. Section 3.2). The chapter concludes with a description of an interesting application of Open CFLP: Origami paper folding (Sect. 3.3).

Descriptions of the Open CFLP system at various stages during its development were published in three journal articles [69, 91, 72]. The final implementation is described in [72].

A description of our Jini service for collaborative constraint solving was published in the proceedings of the ICCC 2004 conference [88].

The usage of Open CFLP was described in our paper presented at the fifth International Mathematica Symposium [68].

3.1 Open CFLP

We decided to implement Open CFLP as an Add-On package of *Mathematica*. We found *Mathematica* ideal for our purposes because:

- It has advanced support for matching and conditional rewriting in a high-order fragment of logic. These capabilities simplified significantly the implementation of our higher-order lazy narrowing calculi for the FLP component of the system. Moreover, *Mathematica* has built-in support to solve efficiently various kinds of equational constraints, including systems of linear, polynomial, and differential equations. Therefore, we could easily implement various domain-specific solvers and access them directly through an instance of the *Mathematica* kernel.

- It has an appealing syntax, similar to the mathematical notation of humans. It has, however, some peculiarities: it uses square brackets instead of parentheses to write terms, and function names which, usually, start with an uppercase letter. Thus, the *Mathematica* syntax for $\sin(x) + 3\ f(x, y^2)$ is `Sin`$[x] + 3\ f[x, y^2]$.

The following example illustrates how we can use Open CFLP to solve a concrete problem. The presentation may appear a bit contrived to make clear the essence of our collaborative CFLP style. Let us consider the problem of solving the system equations

$$y'(t) = k\ y(t), h(0) = 1, y(2) = 3, h(T) = 5$$

for variables $y, k,$ and T. To make the specification of this problem more succinct, we introduce the function `map`. For this purpose, Open CFLP provides the function `FLPProgram`, shown below, which takes as input a functional program consisting of rewrite rules, together with an argument that specifies the types of the function symbols used in the program:

```
R=FLPProgram[{map[λ[{t}],f[t],{}]→{},
              map[λ[{t}],f[t],[H|T]]→[f(H)|map[λ[{t}],f[t],T]]},
          Signature→{
             DefinedSymbols→map:(ℝ → ℝ)×TyList[ℝ] →TyList[ℝ]}}
        ]
```

The free variables are those underlined in the left sides of rules. (A specification of the language of Open CFLP can be found in [91]).

Next we specify the goal to be solved. Logically a goal is an existentially quantified conjunction of equations $\exists x_1, \ldots, x_m.e_1 \wedge \ldots \wedge e_n$, where $e_1, \ldots, e_n$ are equations. In the language of CFLP, we write it as `exists`$[\{x_1, \ldots, x_m\}, \{e_1, \ldots, e_n\}]$. There are three kinds of equations; oriented equation $s \triangleright t$, unoriented equation $s == t$, and strict unoriented equation $s === t$. The former two are relevant to the present discussion. If the equation $s \triangleright t$ or $s == t$ is in solved form, it is written as $s \mapsto t$. The variables occurring in the formula can be type-annotated. Thus, the goal to be solved is as follows:

```
G=exists[{y:ℝ → ℝ,k:ℝ,T:ℝ},{λ[{t},y'[t]==λ[{t},k y[t]],
                                map[λ[{t},y[t]]],{0,2,T})]=={1,3,5}}]
```

In Open CFLP we have access to an open environment where we can find solvers that provide services to solve our problem. Some of them may be on our computer, but here let us assume that the necessary solvers are on the Internet. We can not download the solvers, but are allowed only to use the service of the solvers. We only know the names of the services that are available somewhere on the network. In order to uniquely identify the solving service, we use URI to name the service. For example, the URI

```
http://www.score.is.tsukuba.ac.jp/OCFLP/HOLN
```

is the name of the service of solving equations over the domain of higher-order terms using calculus HOLN [71]. Access to the solver services is obtained by the broker component of Open CFLP, through calls of `FindSolvers`. For example

```
FindSolvers["http://www.score.is.tsukuba.ac.jp/OCFLP/HOLN"]
```

will find the solvers that offer the service of HOLN. Similarly, we can find the solvers for systems of differential equations and systems of linear equations:

```
aHOLN=FindSolvers["http://www.score.is.tsukuba.ac.jp/OCFLP/HOLN"]
aDeriv=FindSolvers["http://www.score.is.tsukuba.ac.jp/OCFLP/Deriv"]
aLin=FindSolvers["http://www.score.is.tsukuba.ac.jp/OCFLP/Linear"]
```

`FindSolvers` returns an *elementary collaborative*, which is a collection of basic solvers that perform the same solving service. Later on, I will explain the mechanism of finding such solvers. An elementary collaborative can be configured by `ConfigSolvers` to work on specific problems more effectively by supplying additional parameters. To some solvers, configuration is essential to equip them with necessary solving power. For example, in the case of HOLN solvers, we need to supply a FLP program `R` to solve equations with respect to `R`. Thus, we configure the HOLN solvers as follows:

```
aHOLN=ConfigSolvers[aHOLN,Prog→R]
```

The other two solvers need not be configured since we use their standard services.

The next step is the most important: We must specify a collaboration of solvers which is capable to solve our problem. For this example, an adequate collaboration strategy is to apply repeatedly the solvers `aHOLN`, `aDeriv` and `aLinear`, in this order, until we reach a fixed point. This collaboration is `repeat[seq[{aHOLN,aDeriv,aLin}]]`, and the system can be configured to use it with the command `NewCollaborative`:

```
aCollabo=NewCollaborative[repeat[seq[{}]]]
```

Afterwards, `aCollabo` refers to a *collaborative*. The final step is to apply it to the goal G:

```
ApplyCollaborative[aCollabo,{G}]
```

The system returns the solution

$\{\{$y↦ λ[{}.e^Log[3] t/2],k↦Log[3]/2,T↦2 Log[5]/Log[3]$\}\}$

A trace of the computation reveals that it unfolds as follows:

$\{G\}\Rightarrow_{\text{aHOLN}}\{G_1\}\Rightarrow_{\text{aDeriv}}\{G_2\}\Rightarrow_{\text{aLin}}\{G_3\}\Rightarrow_{\text{aHOLN}}\{G_3\}\Rightarrow_{\text{aDeriv}}\{G_3\}\Rightarrow_{\text{aLin}}\{G_3\}$

where

$\{G_1\}=\{$exists[{h,T1},{y↦ λ[{t},h[t]],T↦T_1,λ[{t},h'[t]]==λ[{t},k h[t]],
h[0]==1,h[2]==3,h[T_1==5,...]}]},

$\{G_2\}=\{$exists[{c,k,t},{h↦ λ[{t},c e^k t],T↦T_1,y↦ λ[{t},c e^k t],
c==1,c e^2 k==3,e^k T_1==5,...}]}

$\{G_3\}=\{$c↦1,T_1 ↦2 Log[5]/Log[3],k↦Log[3]/2,h↦ λ[{t},e^Log[3] t/2],
y↦ λ[{t},e^Log[3] t/2],T↦2 Log[5]/Log[3],...}

3.1.1 The languages of Open CFLP

The previous example illustrates the general steps we perform with Open CFLP to solve a problem: (1) define an FLP program `R`, (2) specify the problem `G` to be solved as a goal, (3) obtain sets $C_1, \ldots, C_n$ of elementary collaboratives; these are, in fact, the services provided by the elementary solvers in the open environment, (4) configure the collaborative for the FLP component to use program `R`, (5) specify our solving method C as a collaboration of elementary collaboratives $C_1, \ldots, C_n$, and (5) apply C to `G`.

The implementation of Open CFLP as an Add-On package extends the language of *Mathematica* with new functions that assist the user to perform the steps mentioned before: `FLPProgram`, `FindSolvers`, `ConfigureSolvers`, `NewCollaborative`, and `ApplyCollaborative`. They belong to the language $\mathcal{M}$ of Open CFLP, and are methods to access the capabilities of the software components of our system trough the user frontend of the system. This language is multi-tiered, with separate sublanguages for constraint solving, symbolic computation, and solver coordination which enable a succinct and modular programming style.

An important sublanguage of $\mathcal{M}$ is the one used to specify the collaboration between solvers as a collaborative. This language is denoted by $\mathcal{L}$, and is used to control the behavior of a software component, called *coordinator*, which coordinates the activities of the individual solvers. Whenever the user submits a request `ApplyCollaborative`[C, G], the coordinator starts working as told by the collaborative C. The sublanguage of collaboratives $C \in \mathcal{L}$ is defined by the grammar

$C ::=$	$\mathcal{B}$	elementary collaborative
	$\mid \mathcal{X}$	locally defined collaborative
	$\mid$ `seq`$[\{C_1, \ldots, C_n\}]$	sequential composition
	$\mid$ `if`(ϕ, C_1, C_2)	test
	$\mid$ `choice`$[\psi, \{C_1, \ldots, c_n\}]$	concurrency and choice
	$\mid$ `repeat`$[C]$	repetition

Languages like $\mathcal{L}$ have been studied as a core of coordination languages by several researchers (see [123] for a good survey). $\mathcal{L}$ is, in fact, a core of the coordination language BALI [118].

3.1.2 The system architecture

Open CFLP consists of four software components: user frontend, coordinator, broker, and solver. Figure 3.1 shows how these components are connected.

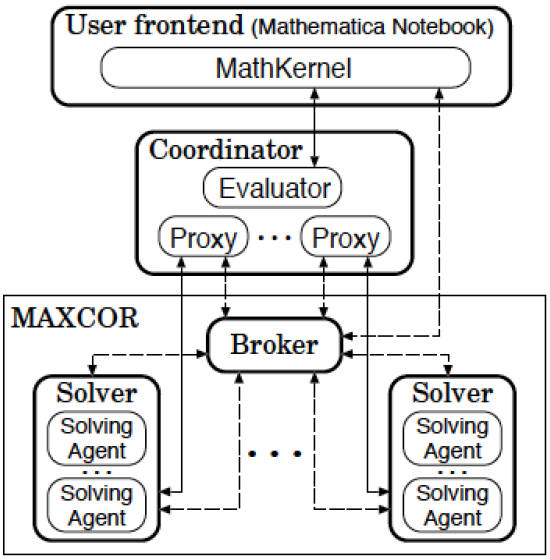


Figure 3.1: The architecture of Open CFLP

The way how the components interact is shown in the sequence diagram from Figure 3.2.

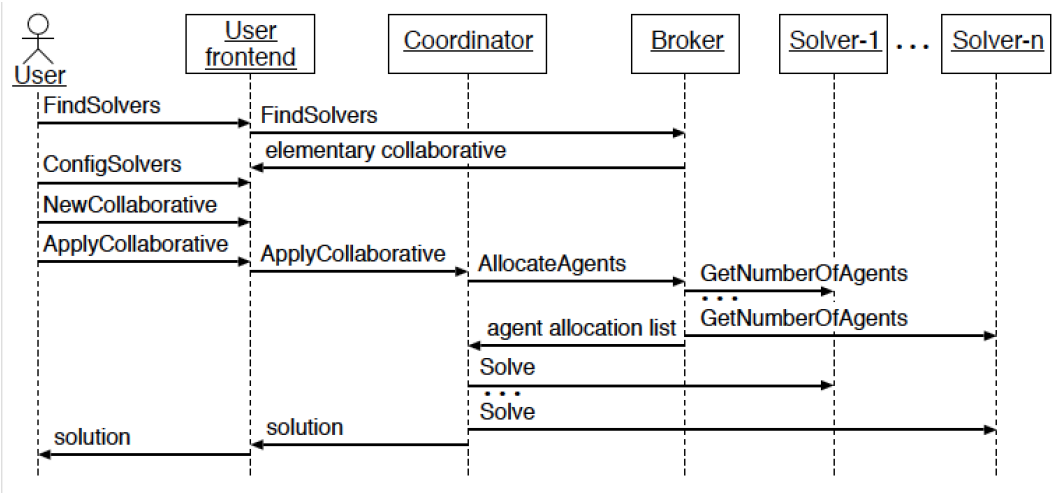


Figure 3.2: Interaction between components of Open CFLP

Functionally, the broker and the solvers are grouped together to form a framework called MAXCOR (MAth eXchange for CORBA). Since MAXCOR has many functionalities, we explain them in a separate subsection.

The **user frontend** (frontend for short) is a user interface to Open CFLP. It is a Mathe-

matica notebook equipped with a MathKernel and allows the user to define CFLP programs in $\mathcal{M}$, as well as to interact with the *Mathematica system*. The MathKernel executes locally defined collaboratives as well as Mathematica programs that run with Open CFLP.

Every **solver** corresponds to a solver provider. It implements a solving algorithm such as higher-order lazy narrowing, Gaussian elimination method, or Gröbner basis computation. When the operation `Solve` is invoked by the coordinator (cf. Fig. 3.2), the solver creates (possibly multiple) solving processes. We call the solving processes **solving agents**. The solving agents of a solver execute the same solving algorithm in parallel.

The **coordinator** interprets the collaboratives of $\mathcal{L}$. It consists of a main component called evaluator, which communicates with the frontend and the broker. When the user calls `ApplyCollaborative`$(C, \overline{G})$ with a collaborative C and list of goals $\overline{G}$, the evaluator receives C and $\overline{G}$ from the frontend, and allocates a buffer U where the solutions of $\overline{G}$ will be stored. Afterwards, it calls a procedure `Collabo`$[C, \overline{G}, U]$, which applies G to the goals in $\overline{G}$. (See implementation details in [72].) When the execution of `Collabo`$[C, \overline{G}, U]$, the evaluator fetches the solution stored on U and returns it to the frontend.

An important design issue is how to exploit parallelism when the system must solve a list of n goals $G_1, \ldots, G_n$ with a collaborative C. To simplify the discussion, suppose C is a basic collaborative, and our system found a collection of m solvers $s_1, \ldots, s_m$. The execution model of Open CFLP allows each solver s_i to spawn τ_i solving agents. Ideally, n goals are solved in parallel by n solving agents. The implementation of `Collabo` exploits the parallelism afforded by the open environment in the following way:

1. Contact the broker and obtain the number τ_i' of currently assignable solving agents for each solver s_i such that $\tau_i' \leq \tau_i$ and $\tau_1' + \ldots + \tau_m' = n' \leq n$.

2. Create a proxy of each solver s_i if $\tau_i' \neq 0$ for $i = 1, \ldots, m$.

3. Distribute n goals among n' solving agents via the proxies.

4. Configure the solving agents, if necessary, and then trigger the n' solving agents to solve the given goals.

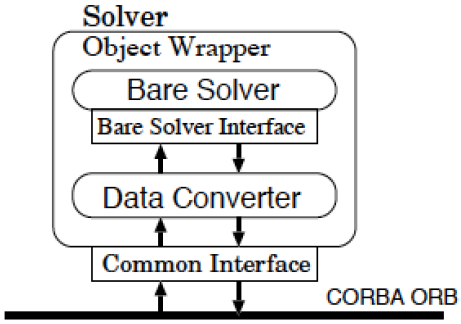
5. Probe the states of the computations via the proxies. Delete the proxies if all the computations are completed, and send the request to the broker to de-allocate all the n' involved solving agents.

6. Return the solution to the frontend.

3.1.3 MAXCOR

MAXCOR was designed as a framework which realizes the transparent communication between solvers. It consists of object wrappers for those solvers and the broker.

In general, we expect to produce collaborations among solvers which are heterogeneous, i.e., they are implemented in different programming languages, run on different platforms and have different data formats for describing constraints. Object wrappers are used to hide the heterogeneity of such solvers. This functionality is achieved by adopting a standard data format and a common protocol of communication. We chose MathML to represent data, and CORBA for the communication protocol. Object wrappers perform data conversion between MathML and the solvers' own data formats, and mediate the solvers' operations. Every object wrapper provides a CORBA-compliant common interface to communicate with a bare solver. The communication is via MathML data. Each wrapper has a data converter between MathML and the bare solvers' own data format. The general architecture of an object wrapper is shown below.



The broker

This software component mediates the connection between clients and providers. In our model, the providers are solvers who offer their capabilities as solving services of a particular kind described by an URI, and inform the broker about this. When the broker gets informed, it registers the solver into a service table which keeps track of all solvers associated so far with the same URI. The clients are proxies created by the coordinator, and they may ask the broker to find appropriate solving services for them.

The broker receives requests for a solving service from the frontend, via the command

`FindSolvers`$[uri]$. When this happens, the broker looks up the service table and returns to the frontend the list of solvers associated with uri.

The broker is also in charge to allocate solving agents for elementary collaboratives. Such requests are received from the coordinator via the command `AllocateAgents`$[C, n]$ where $C = \{s_1, \ldots, s_m\}$ is an elementary collaborative and $n > 0$. When this happens, the broker returns at most n solving agents as follows:

1. It asks each solver $s_i \in C$ how many solving agents it can provide, by invoking the operation `GetNumberOfAgents`. Assume the solver s_i returns the number τ_i.

2. Based on τ_i, the broker distributes n solving agents among m solvers according to its load balancing policy. Let $n' = \min(n, \tau_1 + \ldots + \tau_n)$. The broker decides τ'_i for each s_i such that $\tau'_i \leq \tau_i$ and $\tau_1 + \ldots + \tau'_m = n'$.

3. It informs each solver s_i that the broker decided to reserve τ'_i solving agents.

4. It returns the agent allocation list $\{\{s_1, \tau'_1\}, \ldots, \{s_m, \tau'_m\}\}$ to the coordinator.

Features of MAXCOR

MAXCOR was designed as an application of CORBA. The broker and the solvers are CORBA servers, and the frontend and the proxies in the coordinator are CORBA clients. Combined with the features of CORBA, we have achieved the following properties:

- portability: language and platform independence among solvers,
- data interoperability through MathML documents,
- interoperability among solvers,
- scalability and modularity, by having implemented solvers as CORBA servers, and
- location independence of solvers.

Because CORBA did not offer a broker for dynamic solver deployment, we implemented our own broker. Furthermore, the broker realizes an efficient use of distributed resources by implementing a load balancing policy for the solving agents.

3.1.4 Conclusion

A distinctive feature of Open CFLP is that it was implemented using the CORBA middleware technology and further has realized an broker-provider scheme for open collaborative constraint solving. By this approach, we achieved openness and extensibility of the system.

The coordinator is based on a control-driven coordination model similar to ConCoord [61] and ToolBus [12]. As far as we know, our collaborative constraint system fully implemented with open technologies, CORBA and MathML, with clear design goals of scientific problem solving was a new contribution in the field of collaborative constraint programming.

3.2 A Jini service for collaborative constraint solving

The development of a Jini service for collaborative constraint programming had similar motivations as the development of Open CFLP:

1. Constraint solving services can be easily published in a networked environment.
2. Potential clients can easily discover and access the constraint solving services published in the networked environment.
3. Robustness, easy administration.

There are several middleware technologies available nowadays which offer good solutions to achieve the demands mentioned before. I chose Jini [144], a middleware which relies on the Java capabilities of mobile code and device independence.

The rest of this section is structured as follows. First, I introduce the main concepts and our scheme for collaborative constraint solving. Next, I give a brief account to the Jini infrastructure and describe the implementation of the constraint solving system in Jini. Finally, I draw some conclusions.

3.2.1 The constraint solving scheme

For this system implementation, we chose a collaborative constraint solving scheme inspired by BALI [117], a domain independent environment equipped with a language and a suitable interpreter for solver collaborations. Our environment for constraint programming extends BALI in two important directions:

1. We adopt a higher order language for constraint specifications. Besides simple values, variables can denote unknown functions characterized by sets of higher-order constraints which can be solved by a collaboration of suitable constraint solvers.

2. We parameterize the scheme with a higher-order functional logic programming component (FLP). Several problems of practical interest have a more compact and natural specification in such a setting (Cf. [124]).

The central concepts of our framework are: constraint system, computational domain, component solver, and collaboration language. A constraint system defines the objects, operations and relations which can be used to write constraints. Formally, a constraint system is a triple $X = (\Sigma, \mathcal{V}, \mathcal{L})$ where

- Σ is a higher-order multi-sorted signature given by a set Σ_s of sorts, a set Σ_f of function symbols, and a set of predicate symbols Σ_p. Sorts are the building blocks of types, which are constructed by successive applications of the constructor $\rightarrow$ for function type s. For example, if $\mathbb{R}$ is the sort for real numbers, then $\mathbb{R} \rightarrow \mathbb{R}$ is the type of real functions. We denote by $\Sigma_s^{\rightarrow}$ the set of types over Σ_s.

- $\mathcal{V} = \{V_\tau\}_{\tau \in \Sigma_s^{\rightarrow}}$ where each V_τ is a countably infinite set of variables of type τ.

- $\mathcal{L}$ is the conjunctive closure of a set of atomic $(\Sigma, \mathcal{V})$-formulas. The elements of $\mathcal{L}$ are called constraints.

We can write terms like $\lambda[\{x{:}\mathbb{R}\}, x + 1]$ to denote the function which increments its input by one; or functional equalities like $\lambda[\{t{:}\mathbb{R}\}, v[t]] == \lambda[\{t{:}\mathbb{R}\}, R\, i[t]]$ which describes the variation in time of current i and voltage v in a resistor with resistance R.

A computation domain for a constraint system $\mathcal{X}$ is a structure $\mathcal{D}$ which assigns to every sort $\sigma \in \Sigma_s$ a non-empty set D_σ called the carrier set of σ; to every type $\tau = \tau_1 \rightarrow \tau_2$ the set D_τ of functions from D_{τ_1} to D_{τ_2}; and to every function $r \in \Sigma_f$ of type τ (resp. predicate $p \in \Sigma_p$) an interpretation $f^{\mathcal{D}} \in D_\tau$ (resp. relation $p^{\mathcal{D}} \subseteq D_\tau$). The notion of validity of a ground formula $F \in \mathcal{L}$ in a computation domain $\mathcal{D}$, denoted by $\mathcal{D} \models F$, is defined in the usual way. Free variables are interpreted as existentially quantified, and thus we write $\mathcal{D} \models F$ if there exists a ground substitution θ for the variables of F such that $\mathcal{D} \models F\theta$. Such a θ is called a *solution* of F.

The main concern of constraint solving is to decide the validity of formulas by identifying substitutions θ such that $\mathcal{D} \models F\theta$. This is the place where component solvers are employed. In our framework, a component solver for $\mathcal{X}$ is a computable function $S : \mathcal{L} \to \mathcal{L}^{\vee}$ such that such that, for all $C \in \mathcal{L}$, if $S(C) = C_1 \vee \ldots \vee C_n$ then $\mathcal{D} \models C$ iff $\exists i \in \{1, \ldots, n\}$ such that $D \models C_i$. The set $\mathcal{L}^{\vee}$ denotes the disjunctive closure of $\mathcal{L}$.

Constraint solvers compute finite representations which allow to detect easily whether a formula is valid and, hopefully, to identify it's set of solutions. In collaborative constraint solving, such representations are produced by successive applications of component solvers. The order in which the solvers are applied for reaching such a representation is driven by a strategy, which is an expression written in a suitable *collaboration language*. Our collaboration language is defined by a set $\mathcal{B}$ of basic strategies, and a number of operators which capture the most common ways of building strategies by composing solvers. The syntax for our strategies are defined by the grammar

$coll ::=$		*strategy:*
	b	basic strategy
$\mid$	$\texttt{seq}[coll_1, \ldots, coll_n]$	sequential composition
$\mid$	$\texttt{rep}[coll]$	repetition

The meaning of a strategy $coll$ is a function $\ulcorner coll \urcorner \in (\mathcal{L}^{\vee})^{\mathcal{L}}$ which is defined as follows:

- basic strategies refer to well-known component solvers for $\mathcal{X}$. The association of component solvers to basic strategies is programmatic, as we will explain later.
- $\ulcorner \texttt{seq}[coll] \urcorner$ and $\ulcorner \texttt{seq}[coll_1, \ldots, coll_n] \urcorner = \ulcorner [\urcorner coll_1] \circ \ulcorner \texttt{seq}[coll_1, \ldots, coll_n] \urcorner$, where the operation $\circ : (\mathcal{L}^{\vee})^{\mathcal{L}} \times (\mathcal{L}^{\vee})^{\mathcal{L}} \to (\mathcal{L}^{\vee})^{\mathcal{L}}$ is defined by $(f \circ g)(C) := \bigvee_{i=1}^{m} f(C_i)$ if $g(G) = C_1 \vee \ldots \vee C_m$.
- $\ulcorner \texttt{rep}[coll] \urcorner(C) := C$ if $\ulcorner coll \urcorner[C] = C$;
  otherwise $\ulcorner \texttt{rep}[coll] \urcorner(C) := \ulcorner \texttt{rep}[coll] \urcorner(\ulcorner coll \urcorner(C))$.

These concepts have been accommodated in a unified framework through an instance of the scheme $CFLP(\mathcal{X}, \mathcal{S}, \mathcal{C})$ where $\mathcal{X} = (\Sigma, \mathcal{V}, \mathcal{L})$ is a constraint system, $\mathcal{S}$ is the language for solver collaborations, and $\mathcal{C}$ is a lazy narrowing calculus for the functional logic (sub)language of $\mathcal{X}$. $\mathcal{X}$ is assumed to be open ended, i.e., is can be extended as needed in order to enable the specification of any problem of interest.

To solve a problem, the user specifies it as a constraint in $\mathcal{L}$. Eventually, he defines his own abstractions (functions, relations) to simplify the problem specification. Next, he looks for a number of constraint solving methods which are needed to solve the problem at hand. A suitable constraint solving strategy is then programmed by combining the constraint solving methods with the operators of $\mathcal{S}$.

3.2.2 The realization model

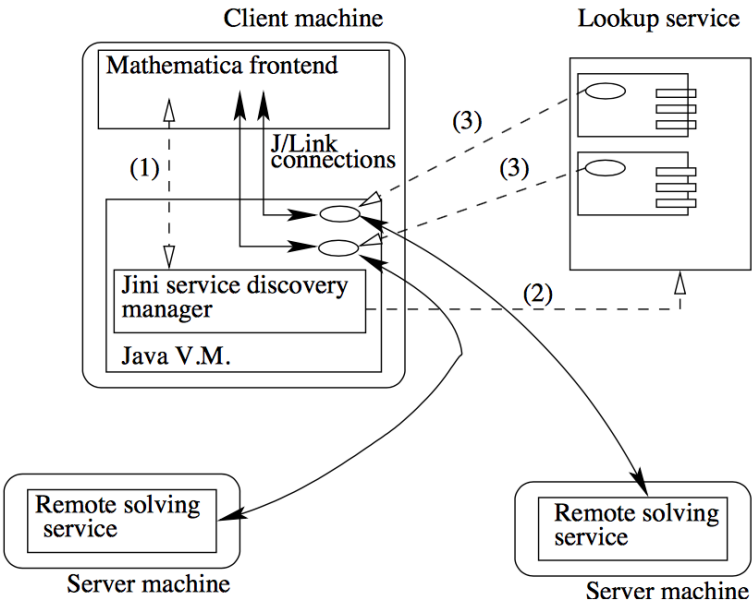
We used Jini to implement our instance of the scheme CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$. The central vision of Jini is the realization of a distributed computing environment that can support rapid configuration that will allow the configuration of devices and software being amended using a simple "plug and play" model. Like CORBA and Dcom, Jini provides an active and responsive distributed infrastructure. Moreover, since it is implemented on top of Java, Jini can exploit the benefits of code mobility and device independence.

The realization of a Jini system brings together three main concepts: (1) a **service** is a piece of independent functionality that is made available to other users and can be accessed remotely across the network; (2) a **client** is a hardware or software component which is interested to make use of a service; and (3) a **lookup service** helps clients to find and connect to services. This service acts as a broker between the needs of clients and the services available across the network.

Service providers make their services available to potential users by registering a service item with the lookup services found in its proximity. A service item contains: (1) a unique service identifier, (2) a proxy, which is a Java object intended to be downloaded by a client to mediate the commuication with the back-end implementation of the service. The purpose of the proxy is to shield the client from all the communication details with the server; and (3) a number of attributes which characterize the service.

A client is expected to contact discovery services located in his network proximity. He can use them to find the services he needs, by specifying the service interface he is interested in, and/or a set of service attributes. The lookup service finds the available service items which match the requirements of the client, and sends him a corresponding proxy. Afterwards, the client can access the service through the downloaded proxy.

The overall architecture of our system as a Jini service is shown below.



Potential clients interact with the distributed constraint solving system through a Mathematica frontend [148] connected to a Java virtual machine via J/Link connections. J/Link is a toolkit which zllows to create and access Java classes and objects from the true interpreted and interactive environment of a Mathematica session. Most importantly, the client can use the facilities of the Jini discovery manager to get access to remote constraint solving services. For instance, if the client wants a solver for derivative equations, he can call:

```
deriv = FindSolver[Domain→"ODE", ...];
```

The arguments of this call are Mathematica options which specify the attributes of the service we are interested in. In this case, we look for a solver for ordinary differential equations (ODE); alternatively, the client can use a Jini browser to view the services registered with the lookup services. This call triggers the following sequence of operations:

1. The service discovery manager (SDM) is asked to find a constraint solving service with the desired attributes.

2. SDM discovers the lookup services located in its network proximity, and asks them for service items which have the attributes requested by the client.

3. The lookup services which have such service items send to the clients the proxies to the corre- sponding remote services.

These three steps are labelled (1), (2), (3) in the figure above. From now on, the client can access the services via the proxy to which `deriv` refers. The access to remote solvers is

completely transparent to the user: all the remote communication details are encapsulated in the functionality of the proxy, and the user sees only the basic strategy `deriv` for a component solver for ordinary differential equations.

This realization model also takes care of resource allocation: constraint solving services are allocated to clients for exclusive use, and it is the responsibility of the client to deallocate the services at the end of his session. For example, in order to deallocate `deriv`, the user must call

```
Deallocate[deriv]
```

3.2.3 Conclusion

The usefulness of an open system for collaborative constraint solving depends on the availability of solving services from solver providers. To illustrate the usefulness of our system, we implemented a number of constraint solvers in Mathematica as providers for solving services, with the intent to deploy them in Jini communities. These solving services implement an interface specified by us. Thus, if a provider wants to make his constraint solving services accessible by our system, he must register a proxy which implements our interface.

The collaborative constraint solving system described here can solve systems of constraint symbolically, by computing exact symbolic solutions. There are several problems which can not be tackled in this way, but with efficient numeric methods. We keft the extension of our system with numeric solvers as a topic for future work.

3.3 Origami Programming

Origami is a Japanese art of paper folding which became a topic of active research due to its relation to art, geometry, theorem proving, and declarative programming. Our interest was to develop a tool which allows us to visualize Origami paper folds, and verify their geometric properties. We used Open CFLP and reported the results of our development in [68].

To make things formal, we proceeded as follows:

- We designed a typed algebra to describe the basic notions (point, segment, line, bisector, etc.) and properties of paper folds (parallelism, segment equality, etc.).

- We formalized the Origami operations by a set of six axioms due to Huzita [67] and modeled them by a set of rewrite rules.

- We explored the well-known geometry-algebra correspondence [34] to transform geometric specifications from our algebra into corresponding algebraic descriptions: systems of multivariate polynomial equations. The reduction is performed with a CFLP program which translates every geometric specification in the algebraic language of equational constraints. For example, we write `Pt[x,y]` for a point with coordinates `x`,`y`; `Line[a,b,c]` for the line of points `Pt[x,y]` for which $\mathtt{a}\,\mathtt{x} + \mathtt{b}\,\mathtt{y} + \mathtt{c} = 0$; and can define the relation $\mathtt{Pt}[\mathtt{x}, \mathtt{y}] \in \mathtt{Line}[\mathtt{a}, \mathtt{b}, \mathtt{c}]]$ by the rewrite rule

  ```
  Pt[x,y]∈Line[a,b,c]→a x+b y+c≈0.
  ```

  In this way, we reduced the decidability problem for geometric statements into an ideal membership problem, and relied on the availability of sophisticated constraint solving techniques, based on Gröbner basis computations [10, 34], to decide them.

- We used Open CFLP to work with our algebraic specifications and programs for the Origami operations, and a collaboration of three local solvers for our goals: (1) a solver for the FLP component, configured with the calculus LCNC_d, and (2) a solver for systems of multivariate polynomial equations; and (3) a solver based on variable elimination with back substitution.

We showed that this simple configuration of Open CFLP is powerful enough to illustrate how Origami can be used to solve some interesting algebraic problems, like finding the solutions of a quadratic equations or the cubic root of a rational number.

Chapter 4

Contributions to Rule-based Programming

Compared to the other declarative programming styles, rule-based programming adds a new declarative capability: to specify *rewrite strategies* which constrain which rule to apply when and where. Processes that can be modeled as the repeated application of a collection or rules under the control of a strategy occur everywhere. Typical examples are: (1) programming languages which use call-by-value or lazy strategies of evaluation; (2) theorem provers which use depth-first or breadth-first proof search strategies; and (3) symbolic computation engines which use sophisticated strategies to bring symbolic expressions to a canonical form. Whereas programming languages tend to rely on a built-in evaluation strategy, the other two examplified applications would really benefit from programmatic support for strategies. This is what proof assistants like LCF, Coq, and Isabelle provide under the name of tactics and tacticals. Other languages, such as Stratego [143], ELAN [15], Maude [24], provide sublanguages to define strategies which can be used to guide and control rule executions.

The increased interest in rule-based programming and the emergence of many new rule-based languages was influenced by two important theoretical results: the rewriting logic [111] and the ρ-calculus [22, 23].

Rewriting logic emerged as a computational logic based on the use of rewrite theories to represent with great generality (1) various models of computation (concurrency, programming languages, etc.), and (2) logical deduction. For computation, it represents states by equivalence classes in an equational theory, and local concurrent transitions by rewrite rules. For deductive purposes, it can represent formula-based data structures (e.g., sequents or sets

of formulas) by terms, and the inference rules of the logic by conditional rewrite rules. An interesting recent refinement of rewriting logic, inspired by the use of rewriting logic as a logical framework for deduction, was to make a clear *separation of concerns* between the specification of the inference system and the heuristics which guide the way in which rules are applied. This point of view introduced the usage of strategy languages to defined theory transformations parameterized by strategy modules [107].

The ρ-calculus makes all the ingredients of rewriting explicit. Terms, rules, rule application, and rule application strategies can all be explicitly represented in its syntax. Moreover, it has a small number of rules of evaluation, including the rule *Fire* which can be made parametric on the matching algorithm employed. As a result, the ρ-calculus can express rewriting modulo equational theories. This calculus was used to implement ELAN, and was later extended with strategy combinators in order to turn it into a powerful strategy language.

My contributions

I designed and studied the formal properties of a calculus for rule-based programming, which I called ρLog. Also, I implemented a rule-based programming system whose execution model is ρLog, and illustrated its usefulness for some interesting applications.

The original features of my calculus and its related implementation are:

1. Rewriting makes use of an advanced matching algorithm, which recognises the use of function variables, sequence variables, and context variables in the specification of patterns, and is capable to compute a complete set of matchers for them. Function variables are placeholders for function symbols, sequence variables are placeholders for sequences of terms, and context variables are placeholders for functions $\lambda x.t$ where t is a term which contains the free variable x only once. Patterns with variables of this kind enable concise specifications of many transformation rules of practical interest.

2. Whereas some systems (including *Mathematica*) can compute a complete set of matchers for patterns with function and sequence variables, the use of context variables was a novelty. I extended the matching algorithm to handle context variables too.

3. Programs are 3-CTRSs of a very general kind, where a rule's condition is a conjunction of atomic formulas of four kinds: labeled rewrites, equalities, membership constraints, and atomic formulas whose ground instances can be decided with *Mathematica*.

4. One can use regular expressions for hedges and contexts to specify memberships constraints for sequence variables and context variables. They are similar to the sort expressions and context expressions studied by H. Comon in [29, 30], and are useful for a range of new applications, such as XML validation and transformation.

5. The unrestricted presence of sequence variables and context variables renders an infinitary unification subproblems when deciding the validity of a conditional rewrite step. We avoided this problem by identifying *determinism* as a natural syntactic restriction on programs and goals. On the computational side, determinism allows to find a complete set of answers to existentially quantified queries by using only matching instead of unification.

6. ρLog makes a clear *separation of concerns* between the specification of the rewrite theory and the heuristics used to guide computation or deduction: The theory is represented by conditional rewrite rules labeled with unique identifiers attached to them (the basic strategies), and the heuristics are represented by strategies built with the combinators of the sublanguage for strategies.

7. For several deductive systems, we can model the inference rules by program clauses for basic strategies, proof heuristics by strategies expressible in ρLog, and proofs correspond to sequences of rewrite steps produced by unfolding a certain strategy. Our implementation can generate human-readable proofs from proof objects produced by tracing the derivations of ρLog. The implementation of this idea resulted from my close cooperation with developers of the THEOREMA software system [104, 103]. THEOREMA is a uniform framework for proving, solving, and simplifying formulae in all areas of mathematics, where proof objects play a central role.

In my opinion, ρLog implements a highly declarative programming style that is expressive enough to support concise implementations for: specifying and prototyping deductive systems, solvers for various equational theories, tools for querying and translating XML, evaluation strategies, etc. [96, 103, 104].

Closely related to the development of ρLog was my research on type systems, and matching and unification algorithms. For this reason, I decided present here my contributions in these subdirections of research too.

The rest of this chapter is structured as follows.

...

4.1 The ρLog system

The rule-based programming style of ρLog builds upon the idea of expressing computations as strategic compositions of basic computational steps, where each basic step can be decomposed into three elementary operations: (1) retrieve information from the input data, as candidates for computing a new result, (2) use some decision criterion to filter the irrelevant candidates, and (3) compute a result from a candidate that has passed the decision criterion. Such a basic computational step can be specified by a transformation rule

$$\texttt{apply}[str, l] \rightarrow r \Leftarrow cnd \quad \text{which we abbreviate by} \quad l \rightarrow_{str} r \Leftarrow cnd$$

where l is a pattern for the structure of the input data, str is an expression describing the strategic construct that acts on the input data, cnd is an algebraic specification of test to filter irrelevant data, and r is an expression that describes the computation of a result.

In our framework, rule-based programs are collections of transformation rules that provide a declarative semantics for answering queries. The expressive power of a rule-based programming language can be estimated by looking at: (1) the kind of queries that can be answered, (2) the strategic constructs recognised by the language, and (3) the programming constructs for specifying transformation rules.

4.1.1 The language

The main syntactic categories of our language are: terms, contexts, strategies, regular expressions, regular constraints, programs and queries.

The language of data

We use an algebra $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of terms to represent incompletely specified objects of interest. A peculiarity of our framework is the presence of four kind of variables: (1) ordinary individual variables $x, y, z \in \mathcal{V}_{\texttt{i}}$ for terms $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$; sequence variable $\overline{x}, \overline{y}, \overline{z} \in \mathcal{V}_{\texttt{s}}$ for finitary sequences of terms $ts \in \mathcal{TS}(\mathcal{F}, \mathcal{V})$; context variables $\overline{C} \in Ctx(\mathcal{F}, \mathcal{V})$ for contexts; and function variables $F, G \in \mathcal{V}_{\texttt{f}}$ for function symbols $f, g, h \in \mathcal{F} \setminus \{\diamond\}$. A context is just a term C with a single occurrence of the special function symbol $\diamond \in \mathcal{F}$, called *hole*, which we

interpret as the unary function $\lambda \diamond .C$. Like in logic programming, we also allow the usage of anonymous variables from the set $\mathcal{V}_{\mathtt{any}} := \{\__{\mathtt{i}}, \__{\mathtt{s}}, \__{\mathtt{s}}, \__{\mathtt{c}}\}$ to denote object parts which are of no interest to us: $\__{\mathtt{i}} \in \mathcal{V}_{\mathtt{i}}$ for an anonymous term; $\__{\mathtt{f}} \in \mathcal{V}_{\mathtt{f}}$ for an anonymous function symbol; $\__{\mathtt{s}} \in \mathcal{V}_{\mathtt{s}}$ for an anonymous term sequence; and $\__{\mathtt{c}} \in \mathcal{V}_{\mathtt{c}}$ for an anonymous context. To disambiguate notation, we write $\ulcorner\urcorner$ for the empty term sequence, and $\ulcorner ts_1, ts_2 \urcorner$ for the concatenation the term sequences ts_1, ts_2. The delimiters $\ulcorner$ and $\urcorner$ are mainly for readability purposes, and we do not display them inside terms. As usual, we abbreviate terms of the form $f[]$ with f.

Thus, $\mathcal{V} = \mathcal{V}_{\mathtt{i}} \uplus \mathcal{V}_{\mathtt{f}} \uplus \mathcal{V}_{\mathtt{s}} \uplus \mathcal{V}_{\mathtt{c}}$ and we revised the notion of substitution $\theta \in Subst(\mathcal{V})$ to bind variables to syntactic objects for their kind. The substitution instantiation $E\theta$ of a syntactic object (term, context, goal, etc.) was also revised accordingly (see [97] for details).

Strategies

In ρLog, strategies play a rôle similar to strategies in ELAN [16]: they specify restrictions of *practical interest* for the relation $\rightarrow^*_{\mathcal{R}}$ induced by a CTRS $\mathcal{R}$. More precisely, we enable the use of atomic formulas $s \rightarrow_{str} t$ in goals and the conditional parts of rules, where $\rightarrow_{str}$ is interpreted as the subrelation of $\rightarrow^*_{\mathcal{R}}$ which respects the constraints specified by strategy str. In this context, a strategy of practical interest can mean an evaluation strategy for programming purposes (lazy, eager, parallel outermost, etc.), a heuristic for theorem provinf when $\mathcal{R}$ models the inference rules of a system of deduction, etc.

We view this capability as a generalisation of the CFLP programming style: We replace the use of oriented equations $s \triangleright t$ (where $\triangleright$ is interpreted as $\rightarrow^*_{\mathcal{R}}$) with the use of reducibility atoms $s \rightarrow_{str} r$ (whose interpretation is usually more deterministic than $\rightarrow^*_{\mathcal{R}}$).

ρLog is a tiered system, where strategies are specified in a different term algebra $\mathcal{F}(\mathcal{F}_{\mathtt{st}}, \mathcal{V}_{\mathtt{st}})$, whose signature $\mathcal{F}_{\mathtt{st}}$ consists of:

1. Predefined strategy combinators `Id`, `Compose`, `Choice`, `Closure`, `NormalForm`, `OrElse` with built-in meanings:

   $s \rightarrow_{\mathtt{Id}} t$ if $s = t$

   $s \rightarrow_{\mathtt{Compose}[str_1,str_2]} t$ if $\exists u$ such that $s \rightarrow_{str_1} u$ and $u \rightarrow_{str_2} t$

   $s \rightarrow_{\mathtt{Choice}[str_1,str_2]} t$ if $s \rightarrow_{str_1} t$ or $s \rightarrow_{str_2} t$

   $s \rightarrow_{\mathtt{Closure}[str]} t$ if $\rightarrow^*_{str} t$ where $\rightarrow^*_{str}$ is the reflexive-transitive closure of $\rightarrow_{str}$

$s \to_{\texttt{NormalForm}[str]} t$ if $s \to^*_{str} t$ and there is no u such that $s \to_{str} u$

$s \to_{\texttt{OrElse}[str_1,str_2]} t$ if either (1) $s \to_{str_1} t$, or (2) $s \not\to_{str_1} t$ and $s \to_{str_2} t$.

The choice to predefine these combinators was driven by pragmatic reasons: They are useful for many applications.

2. User defined strategy combinators, also known as basic strategies. These are all the non-predefined function symbols ℓ from $\mathcal{F}_{\texttt{st}}$, and their must be defined by the programmer via one or more conditional rewrite rules of the form

$$l \to_{\ell[str_1,\ldots,str_n]} r \Leftarrow cnd$$

Membership constraints

Membersip constraints introduce another layer of declarative programming, for the admissible bindings of sequence variables and context variables. They are expressed by atomic constraints of two kinds: (1) $\overline{x} : \texttt{Rs}$ where $\overline{x} \in \mathcal{V}_{\texttt{s}}$ and $\texttt{Rh}$ is a regular constraint for term sequences, and (2) $\overline{C} : \texttt{Rc}$ where $\overline{C} \in \mathcal{V}_{\texttt{c}}$ and $\texttt{Rc}$ is a regular constraint for contexts.

The algebra of regular expressions and the interpreter of membership constraints form a separate component of our system. We considered many ways to specify regular constraints. Here, I indicate the version presented at LPAR 2005 [78].

In the literature, hedges and terms sequences are the same thing. Our regular hedge expressions $\texttt{Rh}, \texttt{Rh}_1, \texttt{Rh}_2$ and regular context expressions $\texttt{Rc}, \texttt{Rc}_1, \texttt{Rc}_2$ are defined as follows:

$$\texttt{Rh} ::= s \mid \overline{x} \mid \texttt{()} \mid \texttt{Rh}_1, \texttt{Rh}_2 \mid \texttt{Rh}_1 + \texttt{Rh}_2 \mid \texttt{Rh}_1^*$$

$$\texttt{Rc} ::= C \mid \texttt{Rc}_1.\texttt{Rc}_2 \mid \texttt{Rc}_1 + \texttt{Rc}_2 \mid \texttt{Rc}_1^\star$$

where $s \in \mathcal{TS}(\mathcal{F}, \mathcal{V} \setminus \mathcal{V}_{\texttt{any}}) \setminus \{\ulcorner \urcorner\}$ and $C \in Ctx(\mathcal{F}, \mathcal{V} \setminus \mathcal{V}_{\texttt{any}})$.

As expected, a regular hedge expression $\texttt{Rh}$ denotes a language $[\![\texttt{Rh}]\!]$ of hedges, and a regular context expression denotes a language $[\![\texttt{Rc}]\!]$ of contexts. These language are defined by induction on the syntactic structures of $\texttt{Rh}$ and $\texttt{Rc}$, where $\texttt{()}$ is interpreted as the empty sequence, ‘,’ as concatenation, ‘+’ as union, ‘*’ as finitary repeated concatenation of hedges, and ‘*’ as finitary repeated functional composition of contexts.

4.1.2 Programs and queries

The building blocks of programs and queries are expressions of the form

- $s \rightarrow_{str,\mathcal{C}} t$ where $\mathcal{C}$ is a set of membership constraints for the variables in s and t. This expression stands for the formula $s \rightarrow_{str} t \wedge \bigwedge_{mc \in \mathcal{C}} mc$.
- $s \not\rightarrow_{str,\mathcal{C}} t$ where $\mathcal{C}$ is a set of membership constraints for the variables in s and t. This expression is an abbreviation for the formula $\neg(s \rightarrow_{str} t \wedge \bigwedge_{mc \in \mathcal{C}} mc)$.

We call these kinds of expressions *reducibility literals*. For explanation purposes, I will sometimes write $s \dashrightarrow_{str,\mathcal{C}} t$ as an abbreviation for either $s \rightarrow_{str,\mathcal{C}} t$ or $s \not\rightarrow_{str,\mathcal{C}} t$. The purpose of the subscript $\mathcal{C}$ is to restrict the application of the strategy str on s to use matchers whose variable bindings satisfy the membership constraints in $\mathcal{C}$. To simplify matters, we simply write $s \rightarrow_{str} t$ instead of $s \rightarrow_{str,\emptyset} t$, and $s \not\rightarrow_{str} t$ instead of $s \not\rightarrow_{str,\emptyset} t$

In ρLog, programs are made of program clauses which look like conditional rewrite rules of the form

$$l \rightarrow_{str,\mathcal{C}} r \Leftarrow \bigwedge_{i=1}^{p} (t_i \dashrightarrow_{str_i,\mathcal{C}_i} t'_i).$$

When $p = 0$ we simply elide the conditional part and write $l \rightarrow_{str,\mathcal{C}} r$.

ρLog is designed to answer goals, which are queries of the form

$$\bigwedge_{i=1}^{n} (t_i \dashrightarrow_{str_i,\mathcal{C}_i} t'_i)$$

by using a form of SLDNF-resolution. Thus we consider rule-based programming as a special case of general logic programming, where the only defined symbol is the reducibility predicate $\rightarrow$. (Cf. [9] for a survey on general logic programming.)

The SLDNF-resolution principle works well for first-order term languages, but is not suitable for the language of ρLog, where sequence variables and context variables render an infinitary unification problem for terms. We have overcome this problem by identifying a class of queries and programs for which SLDNF-resolution can be performed by matching instead of unification. The advantage of matching versus unification is that matching is finitary and we succeeded to identify matching algorithms both for unconstrained matching and for matching with regular constraints [78]. However, in order to ensure the possibility to perform all resolution steps by matching instead of unification, we had to impose a restriction on the syntactic structure of goal and programs. We called this restriction *determinism*. To simplify explanations, I introduce the following notation:

- $vars(E)$ is the set of variables of a syntactic object E. E is *ground* if $vars(E) = \emptyset$.

- If $\mathcal{C}$ is a set of regular constraints then $cvars(\mathcal{C})$ is the set of all variables constrained by regular expressions in $\mathcal{C}$, and $evars(\mathcal{C})$ is the set of all non-anonymous variables that occur in regular expressions in $\mathcal{C}$ (the set of extra variables of $\mathcal{C}$). We also say that $\mathcal{C}$ constrains variables in $cvars(\mathcal{C})$.

Determinism

A program clause

$$t'_0 \rightarrow_{str,\mathcal{C}_0} t_{n+1} \Leftarrow \bigwedge_{i=1}^{n} (t_i \dashrightarrow_{str_i,\mathcal{C}_i} t'_i)$$

is *deterministic* if it satisfies the following four conditions:

1) $\bigcup_{i=1}^{n} vars(str_i) \subseteq vars(str)$,

2) For all $1 \leq i \leq n$, if the i-th literal of the conditional side of the program clause is negative then $vars(t'_i) \subseteq \mathcal{V}_{\texttt{any}} \cup \bigcup_{0 \leq j < i} vars(t'_j)$,

3) For all $1 \leq i \leq n+1$, $vars(t_i) \subseteq \bigcup_{0 \leq j < i} vars(t'_j) \setminus \mathcal{V}_{\texttt{any}}$,

4) For all $0 \leq i \leq n$:

   - $\mathcal{C}_i$ constrains any variable at most once, and
   - $cvars(\mathcal{C}_i) \subseteq vars(t'_i) \setminus \bigcup_{0 \leq j < i} vars(t'_j)$ and $evars(\mathcal{C}_i) \subseteq \bigcup_{0 \leq j < i} vars(t'_j)$.

A query $\bigwedge_{i=1}^{n} (t_i \dashrightarrow_{str_i,\mathcal{C}_i}) t'_i$
is *deterministic* if, for all $1 \leq n$, the following four conditions hold:

1) $vars(str_i) = \emptyset$ and $vars(t_i) \subseteq \bigcup_{1 \leq j < i} vars(t'_j) \setminus \mathcal{V}_{\texttt{any}}$,

2) If the i-th literal is negative them $vars(t'_i) \subseteq \mathcal{V}_{\texttt{any}} \cup \bigcup_{1 \leq j < i} vars(t'_j)$,

3) $\mathcal{C}_i$ constrains any variable at most once, and

4) $cvars(\mathcal{C}_i) \subseteq vars(t'_i) \setminus \bigcup_{1 \leq j < i} vars(t'_j)$ and $evars(\mathcal{C}_i) \subseteq \bigcup_{1 \leq j < i} vars(t'_j)$.

The implementation of built-in strategy combinators

The deterministic program $\mathcal{P}_{\texttt{str}}$ is powerful enough to define the intended meaning of the built-in strategy combinators of ρLog in a straightforward way:

$$\mathcal{P}_{\texttt{str}} := \{x \rightarrow_{\texttt{Id}} x.$$

$$x \rightarrow_{\texttt{Compose}[\sigma_1,\sigma_2]} z \Leftarrow (x \rightarrow_{\sigma_1} y) \wedge (y \rightarrow_{\sigma_2} z).$$

$$x \rightarrow_{\texttt{Choice}[\sigma_1,\sigma_2]} y \Leftarrow (x \rightarrow_{\sigma_1} y).$$

$$x \rightarrow_{\texttt{Choice}[\sigma_1,\sigma_2]} y \Leftarrow (x \rightarrow_{\sigma_2} y).$$

$$x \rightarrow_{\texttt{Closure}[\sigma]} y \Leftarrow (x \rightarrow_{\texttt{Id}} y).$$

$$x \rightarrow_{\texttt{Closure}[\sigma]} y \Leftarrow (x \rightarrow_{\sigma} z) \wedge z \rightarrow_{\texttt{Closure}[\sigma]} (y).$$

$$x \rightarrow_{\texttt{NormalForm}[\sigma]} y \Leftarrow (x \rightarrow_{\texttt{Closure}[\sigma]} y) \wedge (y \nrightarrow_{\sigma\ \texttt{-i}}).$$

$$x \rightarrow_{\texttt{OrElse}[\sigma_1,\sigma_2]} y \Leftarrow (x \rightarrow_{\sigma_1} y).$$

$$x \rightarrow_{\texttt{OrElse}]\sigma_1,\sigma_2]} y \Leftarrow (x \nrightarrow_{\sigma_1\ \texttt{-i}}) \wedge (x \rightarrow_{\sigma_2} y).$$

$\}$

where $x, y, z \in \mathcal{V}_{\texttt{i}}$ and $\sigma, \sigma_1, \sigma_2 \in \mathcal{V}_{\texttt{st}}$.

4.1.3 The calculus

ρLog is designed to answer deterministic queries in theories represented by programs made of deterministic clauses. In general, the structure of a ρLog program is $\mathcal{P} := \mathcal{P}_{\texttt{str}} \cup \mathcal{P}_{\texttt{u}}$ where

- $\mathcal{P}_{\texttt{str}}$ is the aforementioned deterministic program; It provides the built-in interpretation for the predefined strategy combinators.

- $\mathcal{P}_{\texttt{u}}$ is the part of the program written by the user. It provides definitions for his own strategy combinators $\ell \in \mathcal{F}_{\texttt{st}}$ through deterministic program clauses of the form

$$t \rightarrow_{\ell[str_1,\ldots,str_p],\mathcal{C}} t' \Leftarrow \bigwedge_{i=1}^{n} (t_i \dashrightarrow_{str_i,\mathcal{C}_i} t'_i)$$

The execution model of ρLog is, in essence, SLDNF-resolution with leftmost literal selection: every program clause $t \rightarrow_{str_0,\mathcal{C}} t_{n+1} \Leftarrow \bigwedge_{i=1}^{n}(t_i \dashrightarrow_{str_i,\mathcal{C}_i} t'_i)$ is logically equivalent to the clause $t \rightarrow_{str_0,\mathcal{C}} x \Leftarrow \bigwedge_{i=1}^{n}(t_i \dashrightarrow_{str_i,\mathcal{C}_i} t'_i) \wedge (t_{n+1} \rightarrow_{\texttt{Id}} x)$ where $x \in \mathcal{V}_{\texttt{i}}$ is a fresh individual variable, and therefore we can use resolution with respect to these equivalent clauses.

The resolution calculus of ρLog consists of three inference rules. They are shown in Fig. 4.1. There, $mcsm(E_1 \ll E_2, \mathcal{C})$ denotes the minimal complete set of matchers of E_1 and E_2 which satisfy the membership constraints from $\mathcal{C}$. In [78], we have shown that the set $mcsm(E_1 \ll E_2, \mathcal{C})$ is finite and gave an algorithm to compute it.

The main differences between SLDNF-resolution and the calculus of ρLog are:

$$\frac{(t \rightarrow_{str,\mathcal{C}} t' \wedge G)}{(\bigwedge_{i=1}^{n}(t_i \dashrightarrow_{str_i,\mathcal{C}_i} t'_i) \wedge (t_{n+1} \rightarrow_{\mathtt{Id},\mathcal{C}} t') \wedge G)\,\theta}$$

if $str \neq \mathtt{Id}$, $t_0 \rightarrow_{str_0,\mathcal{C}_0} t_{n+1} \Leftarrow \bigwedge_{i=1}^{n}(t_i \dashrightarrow_{str_i,\mathcal{C}_i} t'_i)$ is a fresh variant of a clause from $\mathcal{P}_{\mathtt{str}} \cup \mathcal{P}\mathtt{u}$, and $\theta \in mcsm(\mathtt{apply}(str_0, t_0) \ll \mathtt{apply}(str, t), \mathcal{C}_0)$.

$$\frac{(t \rightarrow_{\mathtt{Id},\mathcal{C}} t') \wedge G}{G\theta}$$

where $\theta \in mcsm(t' \ll t, \mathcal{C})$.

$$\frac{(t \nrightarrow_{str,\mathcal{C}} t') \wedge T}{G}$$

if there exists a finite failed SLDNF-derivation tree of $t \rightarrow_{str,\mathcal{C}} t'$ w.r.t. the program $\mathcal{P}_{\mathtt{str}} \cup \mathcal{P}_{\mathtt{u}}$.

Figure 4.1: The resolution calculus of ρLog

1. The use of $mcsm(\mathtt{apply}(str_0, t_0) \ll \mathtt{apply}(str, t), \mathcal{C})$ instead of

   $$mcsu((\mathtt{apply}(str_0, t_0) \rightarrow t') \equiv (\mathtt{apply}(str, t), \mathcal{C}) \rightarrow x)$$

   when $str \neq \mathtt{Id}$, and

2. The use of $mcsm(t \ll t', \mathcal{C})$ instead of

   $$mcsu((\mathtt{apply}(\mathtt{Id}, t) \rightarrow t') \equiv (\mathtt{apply}(\mathtt{Id}, x) \rightarrow x), \mathcal{C})$$

   when $str = \mathtt{id}$

where the notation $mcsu(E_1 \equiv E_2, \mathcal{C})$ is for a minimal complete set of unifiers between E and E' which satisfy the membership constraints from $\mathcal{C}$. We recall that the terms of our language may contain anonymous variables, and therefore the notions of unifier and matcher between terms must be defined with some care: First, every occurrence of an anonymous variable in the terms under consideration is replaced by a fresh new variable of the same sort; after computing the substitution (unifier or matcher), the variables that were newly introduced at the beginning are back-substituted into anonymous variables, and the bindings for anonymous variables are removed.

Unfortunately, $mcsu(E_1 \equiv E_2, \mathcal{C})$ is usually an infinite set, and *this* is the place where deterministic goals and queries overcome this problem: It can be shown that any ρLog-derivation with leftmost literal selection of a deterministic goal with respect to a deterministic program has the following properties:

(p1) every selected literal $t \dashrightarrow_{str,\mathcal{C}} t'$ has $vars(t) = vars(str) = \emptyset$,

(p2) every selected literal $t \dashrightarrow_{str,\mathcal{C}} t'$ has $vars(\mathcal{C}) \subseteq vars(t')$ and $evars(\mathcal{C}) = \emptyset$, and

(p3) every selected literal $t \nrightarrow_{str,\mathcal{C}} t'$ has $vars(t') \subseteq \mathcal{V}_{\mathtt{any}}$.

Therefore, whenever we select an atom $t \rightarrow_{str,\mathcal{C}} t_0$ for resolution, we have $vars(t) = vars(str) = \emptyset$, $cvars(C) \subseteq vars(t')$, $evars(C) = \emptyset$, and a simple proof by case distinction reveals that SLDNF-resolution with leftmost literal selection coincides with the corresponding inference rule of ρLog.

Hence, the resolution calculus of ρLog inherits all the properties of SLDNF-resolution with leftmost literal selection. In particular, our calculus is sound and incomplete because leftmost literal selection is an unfair selection strategy. Completeness can be recovered if we adopt additional restrictions to guarantee termination of $\mathcal{P}_{\mathtt{str}} \cup \mathcal{P}_{\mathtt{u}}$.

Our notion of deterministic clause is a generalisation of the notion of deterministic 3-CTRS [122] to the case when negative literals are allowed in the conditional part. Condition 2) of our definition of program clauses overcomes the well-known problematic interpretation of negative literals in general logic programming with SLDNF-resolution [9], by ensuring the fact that SLDNF-resolution with leftmost literal selection selects only negative literals with property (p3) mentioned above.

ρLog may also access powerful external libraries to carry out symbolic or numeric computations. The only requirement is the availability of an interface $eval()$ that enables to evaluate any ground term t to a boolean value $eval(t) \in \{0, 1\}$. This capability is embedded in the framework of ρLog via boolean atoms. A *boolean atom* is modeled as a reducibility atom of the form $t \rightarrow_{\mathtt{Id}} \mathtt{True}$ where $True \in \mathcal{F}$ is a special function symbol that indicates the evaluation of this literal should rely on the interface $eval()$ to some external libraries.

The inference rule for boolean atoms is

$$\frac{(t \rightarrow_{\mathtt{Id}} \mathtt{True}) \wedge G}{G} \qquad \text{if } eval(t) = 1.$$

Notes on the implementation

Our implementation of the calculus of ρLog takes advantage of the observation that the resolution steps can be regarded as rewrite steps without evaluating conditions with respect to the conditional term rewriting system $\mathcal{P}_{\mathtt{str}} \cup \mathcal{P}_{\mathtt{u}}$, in the sense defined by Bockmayr in [13]. This can be easily seen in Fig. 4.1 where, for the case $str \neq \mathtt{Id}$, we compute a matcher

between a subterm of the current goal and the left-hand side of the conditional rewrite rule. Since the *Mathematica* interpreter has a built-in mechanism for this kind of rewriting (with built-in backtracking), we found convenient to implement ρLog as a Mathematica package.

4.1.4 Applications

The most obvious application of ρLog is in automated reasoning, where programs model inference rules, and strategies may enforce an efficient exploration of the space for solutions. In [95] we have illustrated how to use this approach for fast prototyping of unification algorithms with sequence variables in free, flat, and restricted flat theories, and of lazy narrowing calculi for theories presented by both unconditional and conditional term rewrite systems. In [97], we have illustrated how to our system for automated deduction in propositional logic, by a straightforward translation of Gentzen's sequent calculus G' [47] into a ρLog program.

ρLog can also generate human-readable traces of its computation. These traces are proof certificates that the computed answers are indeed correct with respect to the program specification. The implementation of this capability was discussed in detail in [103].

Pattern matching with membership constraints makes ρLog useful for querying and manipulating data with regular (sub)structures, such as XML documents. We analysed the implementation of four XML query operations: selection and extraction, reduction, restructuring, and joins. These operations were identified in [83] as the main desiderata for an XML query language. We made a case study of a car dealer office with documents from different auto dealers and brokers. The encoding of these documents, and of the aforementioned operations in ρLog terms, programs, and queries was described in [77]. We draw the conclusion that matching with context variables and sequence variables can serve as a computational mechanism for a declarative rule-based XML query and transformation language. In our opinion, an advantage of such a language would be its flexibility and expressiveness: It would combine in itself the features of both path-based and pattern-based languages, and would easily support, for instance, a wide range of queries (selection and extraction, reduction, negation, restructuring, combination), parent-child and sibling relations and their closures, access by position, unordered matching, order-preserving result, partial and total queries, multiple results, and other properties. Moreover, rule-based paradigm would provide a clean declarative semantics.

We also noticed that ρLog is ideal for the implementation of various evaluation strate-

gies for programming purposes. To illustrate the idea, let's assume we want to evaluate expressions with respect to a confluent and terminating rewrite system

$$\mathcal{R} = \{l_1 \to r_1, \ldots, l_n \to r_n\}.$$

In ρLog, $\mathcal{R}$ can be encoded as a program with n labeled rules: $\{l_1 \to_{\ell_1} r_1, \ldots, l_n \to_{\ell_n} r_n\}$. We can compute the value of a ground expression t as the answer to the query

$$t \to_{\texttt{NormalForm}[str]} x$$

where str is a strategy which controls which redex(es) to rewrite with $\mathcal{R}$ in one reduction step. We indicate below the most common strategies used for this purpose. All of them are defined in terms of the auxiliary strategy $\texttt{R} = \texttt{Choice}[\ell_1, \ldots, \texttt{Choice}[\ell_{\texttt{n}-1}, \ell_{\texttt{n}}] \ldots]$

1. Unrestricted rewriting is allowed to pick up any redex. In this case, $str = \texttt{rw}[\texttt{R}]$ where the basic strategy `rw` is defined by the program clause

$$\overline{C}[x] \to_{\texttt{rw}[\sigma]} \overline{C}[y] \Leftarrow x \to_{\sigma} y.$$

2. Innermost rewriting picks up an innermost redex. We can define it as

$$str = \texttt{OrElse}[\texttt{inner}[\texttt{R}], \texttt{R}]$$

where the basic strategy `inner` is defined by the program clause

$$F[\overline{x}, x, \overline{y}] \to_{\texttt{inner}[\sigma]} F[\overline{x}, y, \overline{y}] \Leftarrow x \to_{\texttt{OrElse}[\texttt{inner}[\sigma],\sigma]} y.$$

3. Outermost rewriting picks up an outermost redex. We can define it as

$$str = \texttt{OrElse}[\texttt{R}, \texttt{outer}[\texttt{R}]]$$

where the basic strategy `outer` is defined by the program clause

$$F[\overline{x}, x, \overline{y}] \to_{\texttt{outer}[\sigma]} F[\overline{x}, y, \overline{y}] \Leftarrow x \to_{\texttt{OrElse}[\sigma,\texttt{outer}[\sigma]]} y.$$

4. Parallel innermost rewriting rewrites simultaneously all outer redexes. We can define it as $str = \texttt{p-i[R]}$ where `p-i` and its auxiliary strategies are defined by

$$\begin{aligned}
x \to_{\texttt{p-i}[\sigma]} y &\Leftarrow (x \to_{\texttt{OrElse}[\texttt{par-all}[\sigma],\sigma]} y) \\
F[x, \overline{x}] \to_{\texttt{par-all}[\sigma]} F[y, \overline{y}] &\Leftarrow (x \to_{\texttt{p-i}[\sigma]} y) \wedge (c(\overline{x}) \to_{\texttt{p-i-else-Id}[\sigma]} c[\overline{ys}]) \\
F[x, \overline{x}] \to_{\texttt{par-all}[\sigma]} F[y, \overline{y}] &\Leftarrow (x \not\to_{\texttt{p-i}[\sigma]\ \texttt{-i}}) \wedge (c(\overline{x}) \to_{\texttt{par-all}[\sigma]} c[\overline{ys}]) \\
F(x, \overline{x}) \to_{\texttt{p-i-else-Id}[\sigma]} F[y, \overline{y}] &\Leftarrow (x \to_{\texttt{OrElse}[\texttt{p-i}[\sigma],\texttt{Id}]} y) \wedge (c(\overline{x}) \to_{\texttt{p-i-else-Id}[\sigma]} c(\overline{y})) \\
F() \to_{\texttt{p-i-else-Id}} F()
\end{aligned}$$

Here, c is a data constructor which can take any number of arguments.

4.2 Related directions of research

4.2.1 Matching with membership constraints

The study of this problem was motivated by our desire to integrate (1) our membership constraints for sequence variables and context variables with rewriting with (2) rewriting with rules that use these kinds of variables in their patterns. When we started to study it, there were many disparate results for problems closely related to ours. Solving equations with context variables and membership constraints for them has already been investigated [29, 30]. It was known that context matching is NP-complete. Also, the problems of sequence matching and unification were already addressed, e.g., in [14, 50, 53, 73, 75, 74], and it was known that both are decidable. There was already a rich literature on matching with regular expressions, especially in the context of general-purpose programming languages and semistructured data querying. Regular expressions were being supported in Perl, Emacs-Lisp, XDuce [64], CDuce [11], Xtatic [48], and in XPath-based languages, just to name a few.

Our contribution

The novelty of our research was that we considered mixed problems where sequence variables, context variables, function variables, individual variables, and membership constraints can occur *simultaneously*. The presence of these kinds of variables allows matching to extract data of interest from term trees of arbitrary depth and breadth, respectively:

- context variables support vertical movement in terms at arbitrary depth, and function variables do the same in one depth level only.
- sequence and individual variables can be seen as the horizontal counterparts for context and sequence variables: Sequence variables match arbitrarily long sequences of terms, and individual variables match only a single term.

Another novelty was that we did consider non-ground regular expressions, and we did not impose the linearity restriction on the variables that occur in them. This gives a powerful data extraction mechanism.

Our main achievement was to find a sound, terminating, and complete matching algorithm for this problem. Also, we showed how to optimise the algorithm by early failure

detection and branching reduction heuristics, and discussed possible applications. This algorithm underlies the computation of the finite sets $mcsm(E_1, E_2, \mathcal{C})$ which are needed to implement the resolution calculus of ρLog.

These results were reported at the type A International Conference LPAR 2005 [78], and at the 22nd International Workshop on Unification [98]. A more technical description of these results was given in a technical report [79]. We give here a brief description of the results published by us in [78, 79], and of the extensions published in [98].

Matching without membership constraints

First, we considered the problem of solving CSM problems, which are systems of equations written as multisets of the form $\Gamma = \{s_1 \ll t_1, \ldots, s_n \ll t_n\}$, where all terms are from the term algebra $\mathcal{T}(\mathcal{F}, \mathcal{V} - \mathcal{V}_{\mathtt{any}})$ of ρLog, and $t_1, \ldots, t_n$ are ground terms. This means, $\bigcup_{i=1}^{n} vars(t_i) = \emptyset$. A *solution*, or *matcher*, of Γ is a substitution θ such that $s_i\theta = t_i$ for $1 \leq i \leq n$. A *complete set of matchers* of Γ is a set of substitutions S such that (i) each element of S is a matcher of Γ and (ii) for each matcher θ of Γ there exists a substitution $\sigma \in S$ such that $\sigma \leq \theta$. S is a *minimal complete set of matchers* (*mcsm*) if it is complete and two distinct elements of S are incomparable w.r.t. $\leq$.

To solve such systems of constraints, we specified a system $\mathfrak{I}$ of 11 transformation rules of the form $MS \Longrightarrow MS'$ where MS, MS' are *systems* of one of the following forms:

- a pair $\Gamma; \sigma$ where Γ is a CSM problem and σ is a substitution, or
- the symbol $\perp$ (failure) which denotes a system with no solutions.

The transformation rules of $\mathfrak{I}$ are shown below. We use the metavariable t (possibly subscripted) to range over terms, and s (possibly subscripted) to range over term sequences.

`T`: Trivial

$$\{t \ll t\} \cup \Gamma; \sigma \Longrightarrow \Gamma; \sigma$$

`IVE`: Individual Variable Elimination

$$\{x \ll t\} \cup \Gamma; \sigma \Longrightarrow \Gamma\theta; \sigma\theta \qquad \text{where } \theta = \{x \mapsto t\}$$

`FVE`: Function Variable Elimination

$$\{F(\overline{s_n}) \ll f(\overline{t_m})\} \cup \Gamma; \sigma \Longrightarrow \{f(\overline{s_m\theta}) \ll f(\overline{t_m})\} \cup \Gamma\theta; \sigma\theta \quad \text{where } \theta = \{F \mapsto f\}$$

PD: Partial Decomposition

$\{f(\overline{t_k}, \overline{x}, \overline{s_n}) \ll f(\overline{t'_m})\} \cup \Gamma; \sigma \Longrightarrow \{\overline{t_k \ll t'_k}, f(\overline{x}, \overline{s_n}) \ll f(\overline{t'_{k+1,m}})\} \cup \Gamma; \sigma$

if $0 < k \leq m$.

TD: Total Decomposition

$\{f(\overline{t_n}) \ll f(\overline{t'_n})\} \cup \Gamma; \sigma \Longrightarrow \{\overline{t_n \ll t'_n}\} \cup \Gamma; \sigma \quad$ if $f(\overline{t_n}) \neq f(\overline{t'_n})$.

SVD: Sequence Variable Deletion

$\{f(x, \overline{s_n}) \ll t\} \cup \Gamma; \sigma \Longrightarrow \{f(\overline{s_n\theta}) \ll t\} \cup \Gamma\theta; \sigma\theta \quad$ where $\theta = \{\overline{x} \mapsto \ulcorner\ \urcorner\}$

W: Widening

$\{f(x, \overline{s_n}) \ll f(t, \overline{t_n})\} \cup \Gamma; \sigma \Longrightarrow \{f(\overline{s_n\theta}) \ll f(\overline{t_n})\} \cup \Gamma\theta; \sigma\theta$

where $\theta = \{\overline{x} \mapsto \ulcorner t, \overline{x}\urcorner\}$

CVD: Context Variable Deletion

$\{\overline{C}(t) \ll t'\} \cup \Gamma; \sigma \Longrightarrow \{t\theta \ll t'\} \cup \Gamma\theta; \sigma\theta \quad$ where $\theta = \{\overline{C} \mapsto \diamond\}$

D: Deepening

$\{\overline{C}(t) \ll f(\overline{t_m})\} \cup \Gamma; \sigma \Longrightarrow \{\overline{C}(t\theta) \ll t_j\} \cup \Gamma\theta; \sigma\theta$

where $1 \leq j \leq m$ and $\theta = \{\overline{C} \mapsto f(\overline{t_{j-1}}, \overline{C}(\diamond), \overline{t_{j+1,m}})\}$.

SC: Symbol Clash

$\{f(\overline{s}_m) \ll g(\overline{t_n})\} \cup \Gamma \Longrightarrow \bot \qquad$ if $f \neq g$.

AD: Arity Disagreement

$\{f(\overline{t}_m,) \ll f(\overline{t'_n})\} \cup \Gamma \Longrightarrow \bot \qquad$ if $m \neq n$

$\{f(\overline{s}_m) \ll f()\} \cup \Gamma \Longrightarrow \bot \qquad$ if s_i is a term for some $1 \leq i \leq n$.

An $\mathfrak{I}$-derivation is a sequence of system transformations $\Gamma_1; \sigma_1 \Longrightarrow \Gamma_2; \sigma_2 \Longrightarrow \cdots$

To solve a CSM problem Γ, the matching procedure $\mathfrak{M}$ generates the a complete tree of derivations originating from the system $\Gamma; \varepsilon$, where ε is the empty (or identity) substitution. It's not hard to see that the leaves of this tree are either of the form $\emptyset; \sigma$, or $\bot$. Se denote by $Sol_{\mathfrak{M}}(\Gamma)$ the set of all substitutions σ produced by an $\mathfrak{I}$-derivation $\Gamma; \varepsilon \Longrightarrow^* \emptyset; \sigma$.

We have shown that $\mathfrak{M}$ terminates for any input problem Γ, thus $\mathfrak{M}$ is an algorithm. Moreover $\mathfrak{M}$ is sound and complete, and $Sol_{\mathfrak{M}}(\Gamma)$ is a mcsm of Γ [78, Theorem 1].

One way to optimize $\mathfrak{M}$ is by detecting failure early and avoiding branching whenever possible. We singled out four matching pretests on $t \ll t' \in \Gamma$ that detect early failure of Γ:

1. The number of symbol occurrences N different from context and sequence variables in t is greater than that in t'. For instance, if $t = f(\overline{C}(a), F(x), \overline{y})$ and $t' = f(a, a)$ then $N(t) = 4$, $N(t') = 3$ and, hence, $t \ll t'$ fails.

2. t contains a function symbol that does not occur in t' like. For instance if $f(\overline{x}, \overline{C}(a), b)$ and $t' = f(c, b)$ then $t \ll t'$ fails because a does not occur in t'.

3. $t = f(\overline{s_m})$, $t' = f(\overline{t'_n})$ and the sequence of non-variable heads of $\overline{s_m}$ os not a subsequence of the sequence of heads of $\overline{t'_n}$. For example, this matching pretest fails for $f(\overline{C}(a), g(x), \overline{x}, g(y)) \ll f(a, g(a), f(a))$ because the sequence g, g is not a subsequence of a, g, f.

4. The minimum depth of s is greater than the depth of t. The minimum depth of a term is computed as the depth without context variables. For instance, the minimum depth of $t = f(f(\overline{C}(F(\overline{x}, f(a)))), g(a, f(x)))$ is 4, and t does not match $t' = f(f(a, f(a)), g(a, f(b)))$ whose depth is 3.

Various such pretests are known in the term indexing literature; see, e.g. [136].

Another way to optimize $\mathfrak{M}$ is to avoid backtracking due to the possibility to apply more than one transformation rule to a system $e \cup \Gamma; \sigma$ with selected equation e. A good heuristics is to select as long as possible equations to which only one transition rule applies. Such examples are the transition rules `T`, `IVE`, `FVE`, `PDE`, `TD`, `SC`, and `AD`, which should be applied as long as possible. The efficiency of $\mathfrak{M}$ can be further improved by extending it with other transition rules of this kind, like the following two rules:

`[Sp]` Splitting

$$\{f(\overline{x}, \overline{s_m}, g(\overline{s''_r}), \overline{s'_p}) \ll f(\overline{t'_n})\} \cup \Gamma; \sigma$$
$$\Longrightarrow \{f(\overline{x}, \overline{s_m}) \ll f(\overline{t'_{j-1}}), g(\overline{s''_r}) \ll t'_j, f(\overline{s'_p}) \ll f(\overline{t_{j+1,n}})\} \cup \Gamma; \sigma$$

where $1 \leq j \leq n$ such that t_j has root symbol g.

`[TlD]` Tail Decomposition

$$\{f(\overline{x}, \overline{s_m}, \overline{y}, \overline{t_p}) \ll f(t'_n)\} \cup \Gamma; \sigma$$
$$\Longrightarrow \{f(\overline{x}, \overline{s_m}, \overline{y}) \ll f(\overline{t'_{n-p}}), t_1 \ll t'_{n-p+1}, \ldots, t_p \ll t'_n)\} \cup \Gamma; \sigma$$

Algorithm $\mathfrak{M}$ extended with membership constraints

Regular expressions provide a powerful mechanism for restricting data values. The classical approach to regular expression matching is based on automata. In this section we show that

regular the membership constraints considered by us for ρLog can be easily incorporated in the rule-based framework of CSM.

The regular expressions analysed by us are:

$\mathtt{Rh} \ ::= \ s \mid \overline{x} \mid \mathtt{()} \mid \mathtt{Rh}_1\mathtt{,}\mathtt{Rh}_2 \mid \mathtt{Rh}_1\mathtt{+}\mathtt{Rh}_2 \mid \mathtt{Rh}^*$

$\mathtt{Rc} \ ::= \ C \mid \mathtt{Rc}_1\mathtt{.}\mathtt{Rc}_2 \mid \mathtt{Rc}_1\mathtt{+}\mathtt{Rc}_2 \mid \mathtt{Rc}^\star$

The language $[\![\mathtt{Rh}]\!]$ denoted by a hedge expression $\mathtt{Rh}$ is defined recursively:

$[\![s]\!] := \{s\}$, $[\![\,\mathtt{()}\,]\!] := \{\ulcorner\,\urcorner\}$,
$[\![\mathtt{Rh}_1\mathtt{,}\mathtt{Rh}_2]\!] := \{\ulcorner s_1, s_2 \urcorner \mid s_1 \in [\![\mathtt{Rh}_1]\!], s_2 \in [\![\mathtt{Rh}_2]\!]\}$,
$[\![\mathtt{Rh}_1\mathtt{+}\mathtt{Rh}_2]\!] := [\![\mathtt{Rh}_1]\!] \cup [\![\mathtt{Rh}_2]\!]$,
$[\![\mathtt{Rh}^*]\!] := \{\ulcorner s_1, \ldots, s_p \urcorner \mid p \geq 0$ and $s_i \in [\![\mathtt{Rh}]\!]$ for all $1 \leq i \leq p\}$

Similarly, the language denoted by a context expression $\mathtt{Rc}$ is defined recursively by

$[\![C]\!] := \{C\}$, $[\![\mathtt{Rc}_1\,\mathtt{.}\,\mathtt{Rc}_2]\!] := \{C_1[C_2] \mid C_1 \in [\![\mathtt{Rc}_1]\!], C_2 \in [\![\mathtt{Rc}_2]\!]\}$,
$[\![\mathtt{Rc}_1\mathtt{+}\mathtt{Rc}_2]\!] := [\![\mathtt{Rc}_1]\!] \cup [\![\mathtt{Rc}_2]\!]$, and
$[\![\mathtt{Rc}^\star]\!] := \bigcup_{n\geq 0} [\![\mathtt{Rc}^n]\!]$ where $[\![\mathtt{Rc}^0]\!] := \{\diamond\}$, $[\![\mathtt{Rc}^{n+1}]\!] := \{C[C'] \mid C \in [\![\mathtt{Rc}]\!], C' \in [\![\mathtt{Rc}^n]\!]\}$.

Note the similarities and differences between our regular expressions and the regular (hedge) expressions and regular context expressions used in the literature (e.g., [65, 31]):

1. Our notions are slightly less general because, for computational reasons, we omitted some defining clauses, like $\mathtt{Rh} ::= \ldots \mid \mathtt{Rc.Rh}$

2. They are also more powerful, because we allow variables in their specification. To account for this fact, we extended the operation of substitution instantiation $E\theta$ to act on regular expressions too.

Regular CSM problems extend CSM problems with the presence of two kinds of regular constraints: $(s \in \mathtt{Rh}, f)$ where s is a hedge, and $(C \in \mathtt{Rc}, f)$ where C is a context variable or a context. In both cases, $f \in \{0, 1\}$ is a boolean flag. The intended reading of these regular constraints is: $(s \in \mathtt{Rh}, f)$ if $s \in [\![\mathtt{Rh}]\!] \setminus \{\ulcorner\,\urcorner \mid f = 1\}$, and $(C \in \mathtt{Rc}, f)$ if $C \in [\![\mathtt{Rc}]\!] \setminus \{\diamond \mid f = 1\}$. The use of a boolean flag f in the specification of regular constraints seems contrived, but was very useful to guarantee the termination of the matching algorithm revised to solve CSM problems with regular constraints. This technique of using flags in

constraints to guarantee termination is similar to that of Frisch and Cardelli [46] for dealing with ambiguity in matching sequences against regular expressions.

To solve CSM problems with regular constraints, we defined a revised version $\mathfrak{I}_R$ of the system of transformation rules $\mathfrak{I}$. Like $\mathfrak{I}$, $\mathfrak{I}_R$ operates on systems $\Gamma;\sigma$ where σ is a substitution, but Γ is now a CSM problem with regular constraints. Thes system $\mathfrak{I}_R$ includes all the rules from the system $\mathfrak{I}$, but SVD, W, CVD, and D need an extra condition on applicability: For the variables $\overline{x}$ and $\overline{C}$ in those transformation rules there should be no regular constraint $(\overline{x} \in \mathtt{Rh}; f)$ and $(\overline{C} \in \mathtt{Rc}; f)$ in the matching problem. There are additional rules in $\mathfrak{I}_R$ for the variables constrained by regular constraints. They are listed below. For the function symbols NonEmptySeq, NonEmptyCtx, and $\oplus$ used in these rules the following equalities hold: $\mathtt{NonEmptySeq}(\ulcorner\urcorner) = 0$ and $\mathtt{NonEmptySeq}(\overline{s_n}) = 1$ if $s_i \notin \mathcal{V}_\mathtt{s}$ for some $1 \leq i \leq n$; $\mathtt{NonEmptyCtx}(\diamond) = 0$ and $\mathtt{NonEmptyCtx}(C) = 1$ if C contains at least one symbol different from context variables and $\diamond$; $0 \oplus 0 = 1 \oplus 1 = 0$, and $1 \oplus 0 = 0 \oplus 1 = 1$.

ESRET: Empty Sequence in a Regular Expression for Hedges

$$\{f(\overline{x}, \overline{s_n}) \ll t, (\overline{x} \in \mathtt{()}, f)\} \cup \Gamma; \sigma \Longrightarrow \begin{cases} \{f(\overline{s_n\theta}) \ll t\} \cup \Gamma\theta; \sigma\theta & \text{if } f = 0 \\ \bot & \text{if } f = 1 \end{cases}$$

where $\theta = \{\overline{x} \mapsto \ulcorner\urcorner\}$.

TSRET: Term in a Regular Expression for Hedges

$\{f(\overline{x}, \overline{s_n}) \ll t, (\overline{x} \in s, f)\} \cup \Gamma; \sigma \Longrightarrow \{f(s, \overline{s_n\theta}) \ll t\} \cup \Gamma\theta; \sigma\theta$ where $\theta = \{\overline{x} \mapsto s\}$ if $s \notin \mathcal{V}_\mathtt{s}$.

SVRET: Sequence Variable in a Regular Expression for Hedges

$\{f(\overline{x}, \overline{s_n}) \ll t, (\overline{x} \in \overline{y}, f)\} \cup \Gamma; \sigma \Longrightarrow \{f(\overline{x}, \overline{s_n})\theta \ll t\} \cup \Gamma\theta; \sigma\theta$ where $\theta = \{\overline{x} \mapsto \overline{y}\}$ if $f = 0$, and $\theta = \{\overline{x} \mapsto \ulcorner y, \overline{y}\urcorner, \overline{y} \mapsto \ulcorner y, \overline{y}\urcorner\}$ with $y \in \mathcal{V}_\mathtt{i}$ a fresh variable.

ChRET: Choice in a Regular Expression for Hedges

$\{f(\overline{x}, \overline{s_n}) \ll t, (\overline{x} \in \mathtt{Rh}_1\mathtt{+}\mathtt{Rh}_2, f)\} \cup \Gamma; \sigma \Longrightarrow \{f(\overline{x}, \overline{s_n}) \ll t, (\overline{x} \in \mathtt{Rh}_i, g)\} \cup \Gamma; \sigma$ for $1 \leq i \leq 2$.

CRET: Concatenation in a Regular Expression for Hedges

$$\{f(\overline{x}, \overline{s_n}) \ll t, (\overline{x} \in \mathtt{Rh}_1\mathtt{,}\,\mathtt{Rh}_2, f)\} \cup \Gamma; \sigma \Longrightarrow \{f(\overline{x_1}, \overline{x_1}, \overline{s_n\theta}) \ll t, (\overline{x_1} \in \mathtt{Rh}_1, f_1), (\overline{x_2} \in \mathtt{Rh}_2, f_2)\} \cup \Gamma\theta; \sigma\theta$$

where $\overline{x_1}, \overline{x_2} \in \mathcal{V}_\mathtt{s}$ are fresh, $\theta = \{\overline{x} \mapsto \ulcorner\overline{x_1}, \overline{x_2}\urcorner\}$, $f_1 = 0$, and f_2 is computed as follows: If $f = 0$ then $f_2 = 0$ otherwise $f_2 = \mathtt{NonEmptySeq}(\overline{x_1}) \oplus 1$.

`RRET1`: Repetition in a Regular Expression for Hedges 1

$\{f(\overline{x}, \overline{s_n}) \ll t, (\overline{x} \in \mathtt{Rh}^*, 0)\} \cup \Gamma; \sigma \Longrightarrow \{f(\overline{s_n\theta}) \ll t\} \cup \Gamma\theta; \sigma\theta$ where $\theta = \{\overline{x} \mapsto \ulcorner\urcorner\}$.

`RRET2`: Repetition in a Regular Expression for Hedges 2

$\{f(\overline{x}, \overline{s_n}) \ll t, (\overline{x} \in \mathtt{Rh}^*, f)\} \cup \Gamma; \sigma$

$\Longrightarrow \{f(\overline{y}, \overline{x}, \overline{s_n\theta}) \ll t, (y \in \mathtt{Rh}, 1), (\overline{x} \in \mathtt{Rh}^*, 0)\} \cup \Gamma\theta; \sigma\theta$

where $\theta = \{\overline{x} \mapsto \ulcorner \overline{y}, \overline{x} \urcorner\}$ with $\overline{y}$ a fresh variable.

`HREC`: Hole in a Regular Expression for Contexts

$$\{\overline{C}(t) \ll t', (\overline{C} \in \diamond, f)\} \cup \Gamma; \sigma \Longrightarrow \begin{cases} \{t\theta \ll t'\} \cup \Gamma\theta; \sigma\theta & \text{if } f = 0, \\ \bot & \text{if } f = 1. \end{cases}$$

where $\theta = \{\overline{C} \mapsto \diamond\}$.

`CxREC`: Context in a Regular Expression for Contexts

$\{\overline{C}(t) \ll t', (\overline{C} \in C, f)\} \cup \Gamma; \sigma \Longrightarrow \{\overline{C}(t)\theta \ll t'\} \cup \Gamma\theta; \sigma\theta$

where $C \neq \diamond$, $\mathtt{root}(C) \notin \mathcal{V}_\mathsf{c}$, and $\theta = \{\overline{C} \mapsto C\}$.

`CVREC`: Context Variable in a Regular Expression for Contexts

$\{\overline{C}(t) \ll t', (\overline{C} \in \overline{D}(\diamond), f)\} \cup \Gamma; \sigma \Longrightarrow \{\overline{C}(t)\theta \ll t'\} \cup \Gamma\theta; \sigma\theta$

If $f = 1$ then $\theta = \{\overline{C} \mapsto F(\overline{x}, \overline{D}(\diamond), \overline{y}), \overline{D} \mapsto F(\overline{x}, \overline{D}(\diamond), \overline{y})\}$ where $F, \overline{x}, \overline{y}$ are fresh variables. If $f = 0$ then $\theta = \{\overline{C} \mapsto \overline{D}(\diamond)\}$.

`ChREC`: Choice in a Regular Expression for Contexts

$\{\overline{C}(t) \ll t', (\overline{C} \in \mathtt{Rc}_1\mathtt{+}\mathtt{Rc}_2, f)\} \cup \Gamma; \sigma \Longrightarrow \{\overline{C}(t) \ll t', (\overline{C} \in \mathtt{Rc}_i, f)\} \cup \Gamma; \sigma$

where $1 \leq i \leq 2$.

`CREC`: Concatenation in a Regular Expression for Contexts

$\{\overline{C}(t) \ll t', (\overline{C} \in \mathtt{Rc}_1\mathtt{.}\mathtt{Rc}_2, f\} \cup \Gamma; \sigma$

$\Longrightarrow \{\overline{C}(t)\theta \ll t', (\overline{C_1} \in \mathtt{Rc}_1, f_1), (\overline{C_2} \in \mathtt{Rc}_2, f_2)\} \cup \Gamma; \sigma$

where $\overline{C_2}, \overline{C_2} \in \mathcal{V}_\mathsf{c}$ are fresh, $\theta = \{\overline{C} \mapsto \overline{C_1}(\overline{C_2}(\diamond))\}$, $f_1 = 0$, and f_2 is computed as follows: $f_2 = 0$ if $f = 0$, otherwise $f_2 = \mathtt{NonEmptyCtx}(\overline{C_1}) \oplus 1$.

`RREC1`: Repetition in a Regular Expression for Contexts 1

$\{\overline{C}(t) \ll t', (\overline{C} \in \mathtt{Rc}^\star, 0)\} \cup \Gamma; \sigma \Longrightarrow \{t\theta \ll t'\} \cup \Gamma\theta; \sigma\theta$ where $\theta = \{\overline{C} \mapsto \diamond\}$

`RREC2`: Repetition in a Regular Expression for Contexts 2

$\{\overline{C}(t) \ll t', (\overline{C} \in \mathtt{Rc}^\star, f)\} \cup \Gamma; \sigma$

$\Longrightarrow \{\overline{C}(t)\theta \ll t', (\overline{C} \in \mathtt{Rc}^\star, 0), (\overline{D} \in \mathtt{Rc}, 1)\} \cup \Gamma\theta; \sigma\theta$

where $\overline{D} \in \mathcal{V}_\mathsf{c}$ is fresh and $\theta = \{\overline{C} \mapsto \overline{D}(\overline{C}(\diamond))\}$.

Note that the rules in $\mathfrak{I}_R$ work either on a selected matching equation, or on a selected pair of a matching equation and a regular constraint. No rule selects a regular constraint alone.

The method proposed my us to solve CSM problems with regular constraints is called $\mathfrak{M}_R$. It is defined in a similar way to the matching algorithm $\mathfrak{M}$, with the only difference that it uses the transition rules of $\mathfrak{I}_R$ instead of the rules of $\mathfrak{I}$.

Our main results about this method are: $\mathfrak{M}_R$ is terminating [78, Theorem 3], thus $\mathfrak{M}_R$ is an algorithm; it is sound [78, Theorem 2] and complete [78, Theorem 4].

Further improvements

In our paper presented at the 22nd International Symposium on Unification (UNIF 2008) [98], we generalised significantly the notions of context and membership constraint for sequence variable and context variable:

- Contexts are incomplete specifications of hedges, which are obtained by replacing one or more subelements of a hedge with the special symbol • (the hole). Every context with n occurrences of • is interpreted as the linear function $\lambda\overline{x_1}, \ldots, \overline{x_n}.C'$ where C' is obtained from C by replacing every i-th occurrence of C with a fresh sequence variable $\overline{x_i}$.

- the membership constraints for sequence variables are of the form $(\overline{x} \in \omega, \phi)$ with ω a *hedge pattern* and $\phi \in \{0, 1\}$, whereas the membership constraints for context variables are of the form $(\overline{C} \in \chi, \phi)$ with χ a *context pattern* and $\phi \in \{0, 1\}$.

- Hedge patterns may contain names of hedge patterns, and context patterns can contain names of context patterns. The meanings of names are specified by an environment $\mathcal{E}$ which indicates the patterns referred to by them. The use of names with associated environments in the specification of patterns was inspired by Hosoya's specification of regular expressions types [63].

- Like before, we allow multiple occurrences of variables in patterns.

- The matching problems consist of equations between hedges, and of membership constraints of the kinds mentioned abose.

We used the same transformational approach to define a matching method $\mathcal{M}$ in terms of a small set of transformation rules on systems for CSM problems. We showed that, under some reasobable syntactic restrictions, the method $\mathcal{M}$ terminates, is sound and complete.

4.2.2 Matching strategies with sequence variables

Sequence variables are an advanced feature of modern languages which allows to program in a declarative and easy to understand way. Functional programming with sequence variables relies on the implicit choice of a matcher which, in general, is not unique. For example, rewriting the list $\{1, 2\}$ with the rule $\{\overline{x}, \overline{y}\} \to \{\{\overline{x}\}, \{\overline{y}\}\}$ can yield any of the results $\{\{\}, \{1, 2\}\}$, $\{\{1\}, \{2\}\}$ or $\{\{1, 2\}, \{\}\}$, depending on the matcher we choose to use. There are many situations when the implicit choice of a language implementation is not what we want. It would be helpful to have a number of idioms for programming with sequence variables which enable the user to control the choice of the matcher.

My contribution

To this end, I have developed the package Sequentica, which extends the language of *Mathematica* with our programming idioms. Our extensions enable (1) to control the selection of matcher by annotating sequence variables with priorities and ranges for their lengths, and (2) to compute optimum values characterised by a score function which must be optimised.

We have published the formal description of our implementation with illustrative examples in two papers: one was presented at the 17th International Workshop on Unification [85], and the other was presented at the 5th International *Mathematica* Symposium [106].

The rest of this subsection draws from material published in [85] and assumes some familiarity of the reader with the functional programming capabilities of *Mathematica*. I precede the presentation of our contributions with a brief account of the programming capabilities of *Mathematica* with patterns, and refer the reader to [148, Sect. 2.3] for a complete description of all Mathematica patterns.

A pattern $patt$ may be named for later reference, e.g., we can write $x : patt$ for a pattern $patt$ named x. The separator : is usually omitted when $patt$ starts with an underline.

Pattern	Meaning
_	one term
__	sequence of 1 or more terms
___	sequence of 0 or more terms
__h	sequence of 1 or more terms, all of whose heads are h
___h	sequence of 0 or more terms, all of whose heads are h
__?*test*	sequence of 1 or more term which satisfy *test*
___?*test*	sequence of 0 or more term which satisfy *test*

Figure 4.2: Patterns in *Mathematica*

For example, we prefer to write x : _Real and y__?$test$ instead of x : _Real and y : __?$test$. Another useful specification is $patt/;cond$ which defines a pattern $patt$ whose meaning is enhanced with the requirement that $cond$ holds for the corresponding matcher. In this work, a sequence variable is considered to be the name of a pattern for a sequence of terms.

Extensions 1

This extension addresses the possibilities to (1) control the choice of a particular matcher instead of relying on some built-in pattern matching strategy; (2) confine the lengths of sequence variable bindings to certain intervals; and (3) impose equality constraints between the lengths of bindings of sequence variables. This can be achieved by annotating sequence variables with (1) binding priorities which specify the order in which the sequences are assigned bindings upon pattern matching, and (2) bounds for the lengths of their bindings. We proposed the extension with sequence annotations of the form

$$patt \leftarrow \{p, m, M\}$$

where $patt$ is a sequence pattern. This annotation assigns priority p to $patt$ and constrains the pattern to match a sequence of terms with length between m and M, inclusively. The sequence variables are assigned bindings in the increasing order of their priorities, by looking for the first matcher that is obtained by varying the sequence length from m to M.

The following examples illustrate the expressive power of this pattern annotation.

Example 1: The function defined by

$$\texttt{GetSubsequence}[x\_\_\_\leftarrow\{2,0,\infty\}, y\_\_\leftarrow\{1,\infty,1\}, y\_\_, z\_\_\_\leftarrow\{3,0,\infty\}] := \{y\}$$

yields the list of the longest nonempty subsequence in the input, which occurs repeatedly at least twice. The assigned priorities are: 1 for y, 2 for x and 3 for z, and thus we first try to bind the sequence variable y, next x, and finally z. The bindings for y are looked up starting from the largest possible length (denoted here by ∞) down to length 1.

Example 2: It is convenient to be able to constrain different sequences to have the same length. We achieve this by imposing that sequence variables annotated with same priority denote sequences of same length. The function

$$\texttt{GetPalindrome[}x\texttt{\_\_\_}\leftarrow\{2,0,\infty\}\texttt{,}\ y\texttt{\_\_}\leftarrow\{1,\infty,1\}, u\texttt{\_}, z\texttt{\_\_}\leftarrow\{1,\infty,1\}\texttt{,}\ t\texttt{\_\_\_}\leftarrow\{3,0,\infty\}\texttt{]/;}\{y\}\ \texttt{===}\ \texttt{Reverse[}\{z\}\texttt{]:=}\{y,u,z\}$$

yields the longest palindrome of odd length contained in the input sequence.

To avoid excessive annotations of sequence variables, we make the following assumptions: (1) it is sufficient to annotate one occurrence of a sequence variable; all other occurrences of that sequence variable are assumed to have the same annotation, (2) anonymous sequence variables have assigned distinct names, and (3) a sequence variable $y\texttt{\_\_}$ (resp. $y\texttt{\_\_\_}$) which is not explicitly annotated has an implicit annotation of the form $\leftarrow\{p,1,\infty\}$ (resp. $\leftarrow\{p,0,\infty\}$). The default priorities p are assigned in a leftmost-innermost manner, and are assumed to be larger than the priorities given explicitly. This means that the non-annotated sequence variables are the last ones which get assigned bindings.

Extension 2

Our second extension addresses the possibility to select matchers which render an expression optimal. To illustrate, let us consider the problem of defining a function `MaxVar[]` which yields the sublist of reals $l_i = \{v_1^i, \ldots, v_{n_i}^i\}$ of a list $l = \{l_1, \ldots, l_n\}$ for which the variation $\max(l_i) - \min(l_i)$ is maximum. We proposed the syntax call:

$$\texttt{MaxVar[}\{\texttt{\_\_\_}, \{x\texttt{\_\_Real}\}, \texttt{\_\_\_}\}\texttt{]:=BestFit[}\{x\}\texttt{,Max[}x\texttt{]-Min[}x\texttt{],Greater]}$$

to find the list $\{x\}$ computed by matching against $\{\texttt{\_\_\_}, \{x\texttt{\_\_Real}\}, \texttt{\_\_\_}\}$ which produces the greatest value of the expression `Max[`x`]-Min[`x`]` with respect to the comparison function `Greater`. More generally, we implemented support for two new definitional mechanisms:

$f[patt]$:=`BestFit[`$expr, optim, test$`]`

defines a partial function f which associates an input call $t = f[a_1, \ldots, a_n]$ with the instance $expr/.\theta$ of the matcher θ between t and $f[patt]$ which satisfies $test[optim/.\theta, optim/.\theta']$ for all other matchers θ' between t and $patt$. f is undefined for input calls which do not match $f[patt]$. Note that $expr/.\theta$ is the *Mathematica* operation to instantiate an expression $expr$ with a substitution θ.

$f[patt]$:=`BestFits[`$n, expr, optim, test$`]`

with $n > 0$ defines a function f which associates to an input call $t = f[a_1, \ldots, a_n]$ the list of instances $\{expr/.\theta_i \mid 1 \leq i \leq k\}$ produced by the longest list of matchers $\{\theta_1, \ldots, \theta_k\}$ between t and $f[patt]$, such that: (1) $k \leq n$, (2) $test[optim/.\theta_j, optim/.\theta_{j+1}]$ holds for all $j \in \{1, 2, \ldots, k-1\}$, and (3) $test[optim/.\theta_k, optim/.\theta]$ holds for all matchers $\theta \notin \{\theta_1, \ldots, \theta_k\}$.

In both cases, $test[]$ is assumed to define a total quasi-order on the set of values $optim/.\theta$ generated by the matchers θ of arbitrary expressions against $f[patt]$.

Algorithmic aspects and implementation issues

Before explaining the algorithmic challenges of implementing our extensions, we introduce a couple of useful notations and definitions.

We denote by $Matchers(expr, patt)$ the set of matchers between $expr$ and a pattern $patt$. We assume that `cut` is a distinguished variadic function symbol reserved for internal use. We use `cut` to mark positions of sequence variables in a pattern and to control lengths of sub-sequences by wrapping them in `cut[...]` subterms. Let $u(patt)$ be the result of removing all annotations from a sequence-annotated pattern $patt$, $Pos(patt)$ be the set of positions of a pattern $patt$, and $patt^+$ be the pattern obtained from $u(patt)$ by wrapping with cut all sequence pattern occurrences. A position $p \in Pos(patt)$ is a *sequence position* if there exists an occurrence of a (possibly annotated) sequence pattern immediately below p. The set of sequence positions of $patt$ is denoted by $Pos_{seq}(patt)$. The *stratification* of $Pos_{seq}(patt)$ is the partition $Pos^1_{seq}(patt) \cup \ldots \cup Pos^n_{seq}(patt)$ of the set $Pos_{seq}(patt)$ into n non-empty subsets, such that:

1. $\bigcup_{i=1}^{n} Pos^i_{seq}(patt) = Pos_{seq}(patt)$,

2. $\forall 1 \leq i \leq n, \forall p_1, p_2 \in Pos^i_{seq}(patt)$, if $p_1 \neq p_2$ then $p_1 \perp p_2$, and

3. $\forall 1 \leq i < n, \forall p_2 \in Pos^{i+1}_{seq}(patt), \exists p_1 \in Pos^i_{seq}(patt)$ such that $p_1 < p_2$.

The number n which shows up in the definition of stratification is called the *sequence depth* of $patt$ and is denoted by $\#(patt)$. When $n > \#(patt)$ we assume $Pos^n_{seq}(patt) = \emptyset$.

For example if

$$P = f[x_{--} \leftarrow \{1, \infty, 2\}, g[x_{--}, h[y_{--} \leftarrow \{2, \infty, 1\}, y_{--}, t_{--}]], h[z_{--}, t_{--} \leftarrow \{1, \infty, 2\}]]$$

then $u(P) = f[x_{--}, g[x_{--}, h[y_{--}, y_{--}, t_{--}]], h[z_{--}, t_{--}]]$,

$$P^+ = f[\texttt{cut}[x_{--}], g[\texttt{cut}[x_{--}], h[\texttt{cut}[y_{--}], \texttt{cut}[y_{--}], \texttt{cut}[t_{--}]]], h[\texttt{cut}[z_{--}], \texttt{cut}[t_{--}]]],$$

$Pos_{seq}(P) = \{\Lambda, 2, 3, 2.2\}$ where Λ stands for the root position of a tern, and $\#(P) = 3$. The stratification of $Pos_{seq}(P)$ consists of $Pos^1_{seq}(P) = \{\Lambda\}$, $Pos^2_{seq}(P) = \{2, 3\}$ and $Pos^3_{seq}(patt) = \{2.2\}$.

Suppose $i \leq \#(patt)+1$ and $Pos^i_{seq}(patt) = \{p_{i,1}, \ldots, p_{i,n_i}\}$. The i-th approximation of $patt$ is the pair $\langle S_i, patt_i \rangle$ where $S_i = \{s_{i,1}, \ldots, s_{i,n_i}\}$ is a set of fresh sequence variable identifiers, and $patt_i$ is obtained from $u(patt)$ by:

(1) dropping all conditions attached to (sub)patterns,

(2) replacing the patterns immediately below position $p_{i,j}$ with the sequence variable pattern $s_{i,j\,---}$, and

(3) wrapping with `cut` all sequence pattern occurrences different from $s_{i,j\,---}$.

For example, the approximations of the pattern P mentioned above are:
$\langle S_1, P_1 \rangle = \langle \{s_{1,1}\}, f[s_{1,1\,---}] \rangle$,
$\langle S_2, P_2 \rangle = \langle \{s_{2,1}, s_{2,2}\}, f[\texttt{cut}[x_{--}], g[s_{2,1\,---}], h[s_{2,2\,---}]] \rangle$,
$\langle S_3, P_3 \rangle = \langle \{s_{3,1}\}, f[\texttt{cut}[x_{--}], g[\texttt{cut}[x_{--}], h[s_{3,1\,---}]], h[\texttt{cut}[z_{--}], \texttt{cut}[t_{--}]]] \rangle$,
$\langle S_4, P_4 \rangle = \langle \emptyset, P^+ \rangle$.

Note that, if $\langle S_i, patt_i \rangle$ is an approximation of a pattern with sequence variables, then matching against $patt_i$ is unitary. This is so because $patt_i$ has no nodes with more than one sequence variable as immediate descendants.

The *i-th system of linear diophantine* constraints determined by *patt* is

$$Dioph(patt_i) : \begin{cases} linexpr_{i,1} = l_{s_i,1}, \\ \dots \\ linexpr_{i,n_i} = l_{s_i,n_i}, \\ length\_equalities, \\ l_{x_1} \in [m_1, M_1], \dots, l_{x_k} \in [m_k, M_k] \end{cases} \tag{4.1}$$

in variables $(l_{x_1}, \dots, l_{x_k}, l_{s_i,1}, \dots, l_{s_i,n_i})$, where

(a) $x_1, \dots, x_k$ is the sequence of all variable identifiers which appear in $patt$ immediately below the positions of $Pos^i_{seq}(patt)$. We assume that the enumeration $x_1, \dots, x_k$ is in the increasing order of sequence variable priorities,

(b) m_j, M_j are the bounds for the lengths l_{x_j} of the sequence variables x_j which are given in the sequence pattern annotations of $patt$, $(1 \leq j \leq k)$

(c) $length\_equalities$ are the equalities between the lengths of sequence variables $x_1, \dots, x_n$ which can be inferred from the sequence pattern annotations, and

(d) $linexpr_{i,j}$ is the linear expression in variables $l_{x_1}, \dots, l_{x_n}$ which defines the number of arguments of the sub-pattern of $patt$ at position $p_{i,j}$. $(1 \leq j \leq n_i)$

For example, the second system of linear diophantine equations of the previous pattern P is

$$Dioph(P_2) : \begin{cases} l_x + 1 = l_{s_{2,1}}, & (* \; linexpr_{2,1} = l_{s_{2,1}} \; *) \\ l_z + l_t = l_{s_{2,2}}, & (* \; linexpr_{2,2} = l_{s_{2,2}} \; *) \\ l_x = l_t, & (* \; length\_equalities \; *) \\ l_x, l_t \in [\infty..2], l_z \in [1..\infty]. \end{cases}$$

Given a substitution σ and a sequence variable x, we define $l_x/.\sigma := m + \sum_{k=1}^{n} p_i \cdot l_{x_i}$ where $x_1, \dots, x_n$ are all sequence variables which occur in the sequence $x/.\sigma$, p_i is the number of occurrences of x_i in the sequence $x/.\sigma$, and $m = \langle$length of sequence $x/.\sigma\rangle - \sum_{k=1}^{n} p_i$.

For example, for $P_2 = f[\texttt{cut}[x_{--}], g[s_{2,1}{---}], h[s_{2,2}{---}]]$ and $\sigma = \{x \mapsto \ulcorner 1, 2 \urcorner, s_{2,1} \mapsto \ulcorner 1, 2, h[y, y, t] \urcorner, s_{2,2} \mapsto \ulcorner z, t \urcorner\}$ where x, y, z, t are sequence variables, then $l_x/.\sigma = 2$, $l_{s_{2,1}}/.\sigma = 3$ and $l_{s_{2,2}}/.\sigma = l_z + l_t$. Note that $l_x/\sigma = l_x$ if $x \notin Dom(\sigma)$.

If $Dioph(patt_i)$ is a system of the form (4.1) then $Dioph(patt_i/.\sigma)$ is the system obtained by applying σ to all components l_{x_j}, $l_{s_{i,k}}$ of (4.1).

Let $expr$ be an input expression and $\langle S_i, patt_i\rangle$ be the i-th approximation of a sequence-annotated pattern $patt$. Suppose $expr$ matches $patt_i$ with the matcher σ, and $Sol = (\tilde{l}_{x_1}, \ldots, \tilde{l}_{x_k}, \tilde{l}_{s_1}, \ldots, \tilde{l}_{s_{n_i}})$ is a solution of $Dioph(patt_i)/.\sigma$. The *i-th cut* of $expr$ determined by $patt_i$ and Sol is the expression $cut_{patt_i,Sol}(expr)$ obtained from $expr$ by replacing every subterm $h[a_1, \ldots, a_p]$ of $expr$ at some position $p_{i,j} \in Pos^i_{seq}(patt)$ with the subterm $h[b_1, \ldots, b_q]$ computed as follows: if the sub-pattern of $patt$ at position $p_{i,j}$ is $h[v_1, \ldots, v_q]$ then

(a) $b_h = a_{u_h}$ if v_h is not a sequence variable pattern,

(b) $b_h = \texttt{cut}[a_{u_h}, \ldots, a_{u_h+\tilde{l}_{x_r}-1}]$ if $v_h = x_{r---}$

for all $h \in \{1, \ldots, q\}$, where $u_1 = 1$ and $u_h + \tilde{l}_{x_r} = u_{h+1}$ if $1 \leq h < q$.

For example, if P_i ($1 \leq i \leq 3$) are the approximations of the previous pattern P and $expr = f[\texttt{cut}[1, 2], g[1, 2, h[a, b, a, b, c]], h[d, c]]$, then

$$\sigma = \{x \mapsto \ulcorner 1, 2 \urcorner, s_{2,1} \mapsto \ulcorner 1, 2, h[a, b, a, b, c] \urcorner, s_{2,2} \mapsto \ulcorner d, c \urcorner\}$$

is a matcher of $expr$ against P_2. The diophantine system of constraints $Dioph(P_2)/.\sigma$ is

$$\begin{cases} 2 + 1 = 3, \\ l_z + l_t = 2, \\ 2 = l_t, \\ 2, l_t \in [\infty, 2], l_z \in [1, \infty] \end{cases}$$

in variables $(l_x, l_y, l_z, l_t, l_{s_{2,1}}, l_{s_{2,2}})$ has solution $(2, 2, 1, 1, 3, 2)$ and

$$cut_{P_2,Sol}(expr) = f[\texttt{cut}[1, 1], g[\texttt{cut}[1, 2], h[a, b, a, b, c]], h[\texttt{cut}[d], \texttt{cut}[c]]]$$

The values 2 for l_x, 3 for $l_{s_{2,1}}$ and 2 for $l_{s_{2,2}}$ are inferred from the bindings of σ, whereas the other values are inferred by solving the system $Dioph(P_2)/.\sigma$.

Implementation of extension 1

First, we must give a definition to matcher between inp_0 and a sequence-annotated pattern $patt$ which is sensitive to the intended meaning of our sequence annotations. Our definition takes into account the stratification of $patt$ and relies on the following observation: θ is a matcher between inp_0 and $u(patt)$ iff there exist $\sigma_1, Sol_1, \ldots, \sigma_n, Sol_n$ such that

(1) $n = \#(patt)$, $\sigma_i \in Matchers(inp_{i-1}, patt_i)$, Sol_i is solution of the system $Dioph(patt_i)/.\sigma_i$, and $inp_i = cut_{patt_i,Sol_i}(inp_{i-1})$ whenever $1 \leq i \leq n$, and

(2) $\theta \in Matchers(inp_n, patt^+)$.

According to this definition, all matchers can be enumerated by enumerating successive cuts inp_i of the input until we reach a cut $inp_{\#(patt)}$ that matches against $patt^+$. This enumeration is driven by the enumeration of the partial matchers σ_i, which is in the order imposed by the annotations of sequence variables. We define the matcher between inp_0 and the sequence-annotated pattern patt as the first matcher between inp_0 and $patt$ which is encountered by following this lookup strategy.

The computation of a matcher between inp and a sequence-annotated pattern $patt$ is triggered by the call $ExtPattMatch(inp, patt, 1)$. The auxiliary method $Matcher(inp, patt)$ returns a matcher between inp and $patt$ if there exists one, and fails if there is no matcher. Note that the calls of $Matcher()$from inside $ExtPattMatch()$ are deterministic. The method $uncut(inp)$ undoes all the cuts performed on an expression. For example

$$uncut(f[\texttt{cut}[1,2], g[\texttt{cut}[3,4], 5, \texttt{cut}[6,7]]])$$

yields $f[1, 2, g[3, 4, 5, 6, 7]]$.

Algorithm $ExtPattMatch$

Inputs: inp: input data; pat: sequence-annotated pattern; i: recursion index

Output: the matcher σ between $uncut(inp)$ and $patt$, if σ exists; `fail`, otherwise.

begin

if $(i = \#(patt) + 1)$

return $Matcher(inp, patt^+)$

else

(* Enumerate the sequence $\mathcal{S} \leftarrow \{Sol_1, Sol_2, \ldots, Sol_{p_i}\}$ of solutions of $Dioph(patt_i)./\sigma$ in the order of priority annotations given in $patt$ *)

next:

if $(j > p_i)$ **return** `fail`

else

$\sigma \leftarrow ExtPattMatch(cut_{patt_i, Sol_j}(inp), patt, i + 1)$

if $\sigma =$ `fail`

$j \leftarrow j + 1$

goto next

else

return σ

end

Sequentica provides an interpreter for sequence annotations. More precisely, in order to interpret function definitions with sequence annotations, we wrap them into a call

`Sequentica[`$defn_1, \ldots, defn_n$`]`

where each $defn_i$ is a sequence-annotated function definition of the form $lhs_i := rhs_i$. This call translates the definitions $defn_1, \ldots, defn_n$ into pure *Mathematica* code, and adds them to the global environment of the *Mathematica* session.

It is often desirable not to make the definitions global, but to handle them as a list of transformation rules. We can program such lists as follows:

R`=Sequentica[`lhs_1`:>`rhs_1`,...,`lhs_n`:>`rhs_n`]`

where $lhs_1, \ldots, lhs_n$ are patterns with annotated sequence variables. In this case, the identifier R is bound to the list $\{r1, \ldots, r_n\}$ of transformation rules produced by translating the sequence-annotated transformation rules into pure *Mathematica* code.

Seasoned *Mathematica* programmers can find the translation details of our sequence-annotated definitions (as functions or as rewrite rules) in [85].

It is important to point out that Sequentica relies heavily on an iterative solver for systems of linear diophantine constraints. This means that the solver does not compute all solutions at once, but enumerates them in the order imposed by the sequence variable annotations. Actually, it would be unreasonable to rely on a solver for linear diophantine constraints which computes all solutions at once because (1) the space of solutions can be very large, and (2) we are not interested in all solutions but only in the *first* one which realizes a matching (where *first* is defined w.r.t. the order which is inferred from the priorities attached to sequence variables). Our current implementation of the linear diophantine solver resembles the constraint solving algorithm described in [32, Sect. 3.3].

Implementation of extension 2

There is a notable algorithmic distinction between the first extension and the second one. The first extension requires to enumerate the solutions of systems of linear diophantine constraints in a certain order (which is controlled by the programmer) until we reach one which induces the desirable matcher, and then stop. The second extension also requires to enumerate the solutions of systems of linear diophantine equations, because these solutions induce possible candidates for pattern matching with sequence variables. However, in this case we can not stop when we reach a matcher because we have no knowledge that we have

reached the optimum we are looking for. Therefore, we must explore the *whole* space of solutions of our systems of linear diophantine constraints and maintain a buffer which stores the binding(s) found so far for which optimum is achieved.

The problem posed by this extension can be formalised as follows:

Given An input expression inp, a (possibly sequence-annotated) pattern $patt$, an integer $n \geq 1$, an expression $optim$ to be optimized, a predicate $test$ which is total on the set of instantiations $\{optim/.\theta \mid \theta \in Matcher(inp, u(patt))\}$, and a target expression $expr$

Find the list $L = \{expr/.\theta_1, \ldots, expr/.\theta_{\min(n,p)}\}$ where $\{\theta_1, \ldots, \theta_p\}$ is the list of matchers between inp and $patt$ ordered such that $test(expr/.\theta_i, expr/.\theta_{i+1})$ holds whenever $1 \leq i < p$.

The algorithm which solves this problem is straightforward to specify in terms of an auxiliary algorithm which enumerates all the matchers between $expr$ and $patt$. The algorithm $BestFits$ shown in Figure 4.3 yields

- `fail` if $p = 0$ (i.e., there are no matchers between inp and $patt$),
- the list $\{expr/.\theta_1, \ldots, expr/.\theta_{\min(n,p)}\}$ if $p > 0$

and makes use of three auxiliary methods: $getPos()$, $insert()$, and $replace()$. The call $getPos(\{v_1, \ldots, v_m\}, v, test)$ yields the largest index j for which $test(v_j, v)$ holds, and 0 if such a j does not exist. $insert(l, v, i)$ inserts v at position i in list l. If $l = \{\}$ and $i = 1$ then $insert(l, v, i)$ yields $\{v\}$. $replace(l, v, i)$ replaces the i-th element of list l with v.

The auxiliary data structure $optList$ is maintained and used to control the positions where to insert new values in L such that, if $\mathcal{E}_q = \{\theta'_1, \ldots, \theta'_q\}$ is the list of matchers enumerated so far, then $L = \{expr/.\theta_1, \ldots, expr/.\theta_{\min(n,q)}\}$ for a rearrangement $\{\theta_1, \ldots, \theta_q\}$ which satisfies the requirement that $test(optim/.\theta_i, optim/.\theta_{i+1})$ holds whenever $1 \leq i < q$.

An algorithm which enumerates the list $\mathcal{E}$ of all matchers between inp and $patt$ can be obtained by adjusting the algorithm $ExtPattMatch$ as follows: instead of resuming the algorithm with the first matcher σ found, we continue to look up for matchers until we reach the condition $j > p_i$.

The Sequentica package supports the second extension too: invocations of the method `Sequentica[]` recognize the definitions and the transformation rules which have definiens

of the form `BestFits`$[n, expr, optim, test]$ or `BestFit`$[expr, optim, test]$ and translates them accordingly. Seasoned *Mathematica* programmers can find in [85] how Sequentica performs these translations.

Algorithm *BestFits*
Description
Compute optimum values defined by a total relation *test*
on instances of expressions ranging over a given syntax domain
Input
inp: input expression
patt: sequence-annotated pattern
n: desired number of best results
expr: expression to be instantiated
optim: expression to be optimized
test: test predicate which characterizes the optimum
Output
`fail`, if $p = 0$
$\{expr/.\theta_1, \ldots, expr/.\theta_{\min(n,p)}\}$ otherwise
where $\{\theta_1, \ldots, \theta_p\}$ is an enumeration of matchers between *inp* and *patt*
such that $1 \leq i < \min(n, p)$ implies $test(optim/.\theta_i, optim/.\theta_{i+1})$
begin
(* Initialize the lists L and *optList* *)
$L \leftarrow \{\}$, $optList \leftarrow \{\}$
(* Generate an enum. $\mathcal{E} = \{\theta'_1, \ldots, \theta'_p\}$ of matchers between *patt* and *inp* *)
if $\mathcal{E} = \{\}$
return `fail`
else
for $i \leftarrow 1$ to p **do**
$idx \leftarrow getPos(optim/.\theta'_i, optList, test) + 1$
if $idx < n$
$L \leftarrow insert(L, expr/.\theta'_i, idx)$
$optList \leftarrow insert(optList, optim/.\theta'_i, idx)$
elseif $idx = n$
$L \leftarrow replace(L, expr/.\theta'_i, n)$
$optList \leftarrow replace(optList, optim/.\theta'_i, n)$
fi
od
return L
fi
end

Figure 4.3: Algoritm $BestFits$

4.2.3 Computational methods in algebras for regular hedge languages

Our first interest in regular hedge expressions was in connection with membership constraints that restrict the admissible bindings of sequence variables in matching problems. For such

uses, we allowed variables in the built-up of regular expressions, in order to increase the binding power of matchers to arbitrary width and depth in term-like structures.

Another important rôle of regular expressions in computed science is as type specifiers. For these purposes, we use variable-free (or ground) regular hedge expressions, which is a formalism equivalent to hedge automata [120] for the specification of regular hedge languages. A well-known use of ground regular hedge-expressions is in the specification of regular types of sequence variables used in typed languages for XML processing [63]. In this context, they are called regular expression types. For example, according to the RELAX NG standard for XML validation [141], an XML document is valid iff its content belongs to a regular hedge language.

The development of a rule-based transformation with regular expression types requires the development of computational methods for the various operations needed by a type system. In [63], Hosoya proposed various algorithms for the operations used in the implementation of language XDuce for XML processing.

Our approach and contributions

We studied the possibility to integrate ρLog with a type system with regular hedge expression types. To avoid excessive type annotations, we studied type inference systems for the variables in the patterns of rewrite rules that act on XML-validated input. In this framework, a validated input is a hedge from a known regular hedge language. Our transformation rules are more general than those used by Hosoya for XML processing because we allow nonlinear patterns. For XML transfomations driven by applications of rules of this kind, we were faced with the following type reconstruction problem:

Given inputs inp of a regular hedge type r, and a program rule $l \to r \Leftarrow cond$ where l is a nonlinear pattern for hedges which may contain sequence variables,

Compute the *maximal* tuple(s) of types $(r_1, \ldots, r_n)$ of the sequence variables $(\overline{x_1}, \ldots, \overline{x_n})$ in l, that guarantee that the type of l is a subtype of the type r of inp.

For this problem, maximality is defined with respect to the linear order $(r_1, \ldots, r_n) \leq (r'_1, \ldots, r'_n)$ iff $[\![r_i]\!] \subseteq [\![r'_i]\!]$ for $1 \leq i \leq n$, where $[\![r_i]\!]$ is the regular hedge language represented by r. A tuple of types $(r_1, \ldots, r_n)$ is *admissible* for $(\overline{x_1}, \ldots, \overline{x_n})$ if $\overline{x_n : r_n}$ imply

that the type of l is a subtype of r. $(r_1, \ldots, r_n)$ is *maximal* if it is admissible and there is no other admissible tuple $(r'_1, \ldots, r'_n)$ such that $(r_1, \ldots, r_n) < (r'_1, \ldots, r'_n)$.

In the design of XDuce, Hosoya addressed a similar problem: he considered transformation rules with linear patterns, and the type inference problem for each pattern variable $\overline{x_i}$ independently, without looking at the possible dependencies among their types. Our type inference problem is more general and, as it turned out, it is computationally harder. We found out that our type reconstruction problem is closely related to the factorization problem of regular hedge languages. A solid theory of factorizations of regular string languages was already established by Conway [33], but similar results for regular hedge languages were still missing. Therefore, we embarked on generalising Conway's factorization theory for regular hedge languages, and built up a rich algebra of computational methods which are useful to implement the type system envisioned by us for rule-based transformations.

The results of our research in this direction were presented at two type B conferences and two type C conferences: [99] presented at ADBIS 2009, [101] presented at DLT 2010, and [86, 87] presented at SYNASC conferences. They also formed the subject of a class C journal article [100]. To appreciate he relevance of our results, the remainder of this subsection gives a brief account of our achievements in this direction of research.

Preliminary notions

For any given set A, we write 2^A for the set of all subsets of a, A^* for the set of all finite strings of elements over A, and ϵ for the empty string. A string language over a finite set A is a subset of A^*. The most common operations on languages are set union, difference, intersection, complement, concatenation, asterate, left/right quotient. We recall that, if L, M are languages then (1) the *concatenation* $L.M := \{s_1 s_2 \mid s_1 \in L, s_2 \in M\}$; (2) the *asterate* of L is $L^* := \bigcup_{n>0} L^n$ where $L^0 := \{\epsilon\}$ and $L^n := L.L^{n-1}$ if $n > 0$; and (3) the *left quotient* (resp. *right quotient*) is $M^{-1}L := \{s \mid \exists s' \in M$ such that $s's \in L\}$ (resp. $LM^{-1} := \{s \mid \exists s' \in M$ such that $ss' \in L\}$).

The algebra $\mathrm{Reg}(A)$ of regular languages over A is of central importance in computer science, and there are many equivalent formalisms to represent them: finite automata, regular grammars, regular expressions, linear systems of (language) equations, etc. For example, a regular expression over alphabet A is defined by

$$r \ ::= \ 0 \mid 1 \mid a \mid r_1 + r_2 \mid r_1.r_2 \mid r_1^*$$

and their language interpretation is: $[\![0]\!] := \emptyset, [\![1]\!] := \{\epsilon\}, [\![a]\!] := \{a\}$, $[\![r_1 + r_2]\!] := [\![r_1]\!] \cup [\![r_2]\!], [\![r_1.r_2]\!] := [\![r_1]\!].[\![r_2]\!]$, and $[\![r_1^*]\!] := [\![r_1]\!]^*$. In this algebra, the computation of factors is an important operation for many applications. We recall that a *subfactorization* of $L \in \mathrm{Reg}(A)$ is a tuple $(F_1, \ldots, F_n)$ of regular languages such that $F_1.\cdots.F_n \subseteq L$. Subfactorizations are compared w.r.t. the ordering $\sqsubseteq$ defined by $(F_1, \ldots, F_n) \sqsubseteq (F'_1, \ldots, F'_n)$ iff $F_i \subseteq F'_i$ for $1 \leq i \leq n$. A *factorization* of $L \in \mathrm{Reg}(A)$ is a $\sqsubseteq$-maximal subfactorization of L, and a *factor* of L is a component F_i of a factorization $(F_1, \ldots, F_n)$ of L. The factors which occur first in a factorisation are called *left factors*. Conway [33] proved that regular languages have finitely many factors. Note that, if $\{S_1, \ldots, S_n\}$ is the set of left factors of $L \in \mathrm{Reg}(A)$ then $\{G_i \mid G_i = S_i^{-1}L, 1 \leq i \leq n\}$ is the set of right factors of L. Conway showed that we can compute an $n \times n$ matrix $FM(L)$ such that (1) each factor of L is one of $F_{i,j}$, (2) there exist unique indices $l, r \in \{1, \ldots, n\}$ such that $L = S_r = G_l = F_{l,r}$ and $S_i = F_{l,i}$, $G_i = F_{i,r}$ for each $i \in \{1, \ldots, n\}$. Moreover, we showed that

1. $\epsilon \in F_{i,i}$ for all $1 \leq i \leq n$.
2. $F_{i,j}.F_{j,k} \subseteq F_{i,k}$ whenever $i, j, k \in \{1, 2, \ldots, n\}$.
3. If L_1, L_2 are languages, then $L_1.L_2 \subseteq F_{i,j}$ iff there exists $1 \leq k \leq n$ such that $L_1 \subseteq F_{i,k}$ and $L_2 \subseteq F_{k,j}$.
4. If $L_1, \ldots, L_j$ are languages, then $L_1.\cdots.L_j \subseteq L$ iff there exist $k_1, \ldots, k_{j-1} \in \{1, \ldots, n\}$ such that $L_1 \subseteq F_{l,k_1}$, $L_j \subseteq F_{k_{j-1},r}$, and $L_i \subseteq F_{k_{i-1},k_i}$ for all $1 < i < j$.

Hedge languages are sets of hedges (or term sequences). Hedges $h \in \mathcal{H}(\Sigma, \mathcal{K})$ are defined over a finite alphabet Σ of variadic function symbols and a set $\mathcal{K}$ of constants, by

$h \ ::= \ \epsilon \mid k \mid a(h_1)h_2 \qquad$ where $a \in \Sigma$ and $k \in \mathcal{K}$.

When $\mathcal{K} = \emptyset$ we write $\mathcal{H}(\Sigma)$ instead of $\mathcal{H}(\Sigma, \emptyset)$.

A hedge language is a subset of $\mathcal{H}(\Sigma, \mathcal{K})$. For $a \in \Sigma$ and $L \subseteq \mathcal{H}(\Sigma, \mathcal{K})$ we define (1) the *projection* of a hedge language L with respect to $a \in \Sigma$ is the hedge language $\mathtt{Pj}_a(L) := \{h \mid a(h) \in L\}$, and (2) the *application* of a to L is the hedge language $\{a(h) \mid h \in L\}$.

There is a unique encoding of every hedge $h \in \mathcal{H}(\Sigma, \mathcal{K})$ into a binary tree $\mathrm{bin}(h) \in \mathcal{T}(\Sigma \cup \mathcal{K} \cup \{\#\})$. The binarization function bin is defined recursively: $\mathrm{bin}(\epsilon) := \#$, $\mathrm{bin}(k) := k(\#, \#)$, and $\mathrm{bin}(a(h_1)h_2) := a(\mathrm{bin}(h_1), \mathrm{bin}(h_2))$. Note that, in the construction of binary trees, the symbols of $\Sigma \cup \mathcal{K}$ have arity 2, and the new symbol $\#$ is the

only nullary function symbol. The *binarization* of a hedge language $L \subseteq \mathcal{H}(\Sigma, \mathcal{K})$ is $\mathrm{bin}(L) := \{\mathrm{bin}(h) \mid h \in L\}$.

The notion of regularity was extended from string languages to hedge languages by Thatcher [140, 139], who developed the basic theory of unranked tree automata and investigated their regular extensions. We denote by $\mathrm{HReg}(\Sigma, \mathcal{K})$ the class of regular hedge languages over Σ and $\mathcal{K}$, and write $\mathrm{HReg}(\Sigma)$ instead of $\mathrm{HReg}(\Sigma, \emptyset)$. There are many equivalent ways to define $L \in \mathrm{HReg}(\Sigma, \mathcal{K})$. We mention here three popular approaches:

1. $\mathrm{bin}(L)$ is accepted by a top-down tree automaton $(Q, I, \mathcal{F}, \Delta)$ (Cf. [63])

2. L consists of all hedges which fulfil a conformance relation $E \vdash h \in T$ where T is a regular expression type and E is an environment (i.e., collection of name definitions) for the names used in T (Cf. [63])

3. L is accepted by a hedge automaton (Q, Σ, F, Δ_M) (cf. [119])

Several theoretical results from regular string languages were generalized to regular hedge languages. Unfortunately, no attempts were made to define a factorization theory for regular hedge languages, or to check if the results established by Conway can be lifted to this more general setting. In the following we present our results in this direction.

A factorization theory for regular hedge languages

The notions of subfactorization, factorization, factors, and left factors carry over without changes from the algebra of regular languages to $\mathrm{HReg}(\Sigma)$. However, the computational complexity of these operations is higher than for regular languages, and is strongly affected by the language representations on which we operate.

In [99] we introduced *linear systems of hedge language equations* (LHS) as a formalism to represent regular hedge languages (RHLs for short). We have shown that this formalism is suitable for several algebraic computations, such as intersections, quotients, left and right factors of RHLs. We described algorithms for the translation between hedge automata and linear systems of hedge language equations, and for the computations mentioned before. A consequence of the existence of these algorithms is that RHLs have finitely many factors, and they can be placed in a factor matrix that has some nice properties as for regular string languages. The main problem noticed by us was the exponential complexity of the algorithms involved in the computation of the factor matrix.

In [101] we reconsidered the problem or regular RHL factorization and introduced a better representation of RHLs for this purpose: *reduced, complete, and deterministic linear hedge automata*. Our development of the revised theory for factorization is incremental:

1. First, we introduced *linear hedge automata* (LHA) $A = (Q, \Sigma, Q_f, \Delta)$ for RHLs, and defined the languages:

   - $L(A, q)$: the set of hedges accepted by A in a state $q \in Q$. and
   - $L(A, Q') := \bigcup_{q \in Q'} L(A, q')$.

2. Next, we described the correspondence between LHA and LHS for the same regular hedge language.

3. We presented an algorithm that computes for every LHA an equivalent LHA which is deterministic, reduced and complete.

4. We described an algorithm to compute the right factors of an RHL L. An auxiliary subcomputation of this algorithm is the set $\partial L := \{\{h\}^{-1}H \mid h \in \mathcal{H}(\Sigma)\}$ of left derivatives of H. To compute them, we defined a *differential calculus* for RHLs as a generalization of Antimirov's calculus with partial antiderivatives of regular expressions [4].

5. Next, we described how to compute LHAs for the right factors of L from the LHAs for the elements of ∂L. We have shown that, if $L = L(A, Q_f)$ for a deterministic, reduced and complete LHA $A = (Q, \Sigma, Q_f, \Delta)$ then the set of right factors of L is of the form $\{L(A, Q_i) \mid 1 \leq i \leq n\}$ where every Q_i is a subset of Q.

6. Finally, we have shown that the factor matrix of L is $(F_{i,j})$ of dimension $n \times n$, where every $F_{i,j}$ is the product antiderivative of $L(A, Q_i)$ with respect to $L(A, Q_j)$. An algorithm which computes an LHA for product antiderivatives of this kind is outlined in [101, Section 4.2].

In [101], our main insight to improve the efficiency for the computation of the factor matrix was that it is easier to compute the factors $F_{i,j}$ via computations of product antiderivatives between right factors than via product derivatives between left factors.

A type reconstruction algorithm

We noticed that factorizations of RHLs are useful in the implementation of type reconstruction algorithms for rule-based programming systems for XML processing. We first addressed this problem in [86], and described a more satisfactory solution in [87]. I recall here the problem and type reconstruction algorithm presented in [87].

Our type reconstruction problem was stated as follows:

Given (1) a rule pattern l which contains variables $\overline{x_1}, \ldots, \overline{x_n}$ from a countably infinite set $\mathcal{X}$ of sequence variables, and (2) an RHL L

Find all tuples of hedge languages $(L_1, \ldots, L_n)$ which are maximal w.r.t. the componentwise-inclusion ordering, such that the language denoted by the pattern instantiation

$$l\{\overline{x_1} \mapsto L_1, \ldots, \overline{x_n} \mapsto L_n\}$$

is a subset of L.

This problem occurs when we want to infer (or *type reconstruct*) the set T of maximal n-tuples of hedge languages for the variables $\overline{x_1}, \ldots, \overline{x_n}$ of the pattern of a transformation rule $l \to r \Leftarrow cond$, such that the following equality holds:

(GTR) $\{(\theta(\overline{x_1}), \ldots, \theta(\overline{x_n}) \mid \exists h \in L \text{ such that } \theta \text{ is a matcher of } l \text{ with } h\}$
$= \bigcup_{(L_1,\ldots,L_n)\in T} L_1 \times \ldots \times L_n.$

This is a type reconstruction problem because, in our framework, types represent regular hedge languages and every pair $(L_1, \ldots, L_n) \in T$ represents a valid assignment of types $\overline{x_1} : L_1, \ldots, \overline{x_n} : L_n$ to the pattern variables of the transformation rule.

The peculiarities of the type reconstruction problem posed by us are:

1. It is *globally precise*, in the sense that we look at tuples of types (which represent hedge languages) for which equality (GTR) holds. This is more demanding than the *locally precise* type reconstruction problem solved by the type inference system of XDuce [63], which computes, for every pattern variable $\overline{x_i}$ of l the largest hedge language L_i such that the following equality holds:

   (LTR) $\{\theta(\overline{x_i}) \mid \exists h \in L \text{ such that } \theta \text{ is a matcher of } l \text{ with } h\} = L_i.$

2. We consider a very general class of patterns, defined by the grammar:

$P ::= 1 \mid a(N) \mid P\,P \mid P{+}P \mid P\star \mid \overline{x}$

where $a \in \Sigma$, $\overline{x} \in \mathcal{X}$ is a sequence variable, N is a pattern name defined in an environment E consisting of definitions of the form $N = P$. We call these kinds of expressions *regular expression patterns*, or just *patterns*. They are similar to the patterns of XDuce, but more general because our patterns can be nonlinear.

Working with such patterns makes the definition of matchers sensitive to the presence of an environment E. Therefore, we defined the relation $h \in_E P \rightsquigarrow \sigma$ with the intended reading "σ is an extended matcher of hedge h by P in environment E" [87, Sect.II.(B)].

Our main insight was that the RHL factorization theory guarantees the following property for our type reconstruction problem:

- T is a finite set and all its elements $(L_1, \ldots, L_n)$ are tuples of RHLs.

For the efficient computation of the components of T we invented yet another representation or RHLs, by factor automata. A *factor automaton* is a triple $\langle Q', \Delta, Q''\rangle$ where Δ is the set of transition rules of a reduced, complete, and deterministic LHA $A = (Q, \Sigma, Q_f, \Delta)$, and Q', Q'' are subsets of Q. The language accepted by a factor automaton $FA = \langle Q', \Delta, Q''\rangle$ is $L(FA) := \{h \in \mathcal{H}(\Sigma) \mid \forall q'' \in Q''.\exists q' \in Q'.h\,q'' \to_\delta^* q'\}$.

When the LHA $A = (Q, \Sigma, Q_f, \Delta)$ is reduced, complete, and deterministic, there is a unique state $\mathtt{in}(\Delta) \in Q$ such that $\epsilon \to \mathtt{in}(\Delta)$. We write $\mathtt{st}(\Delta)$ for the set of states that occur in the rules of Δ, and abbreviate the factor automata $\langle Q', \Delta, \{\mathtt{in}(\Delta\})\rangle$ by $\langle Q', \Delta\rangle$. Note that $L(A) = L(\langle Q_f, \Delta\rangle)$. From now on, we assume implicitly that $A = (Q, \Sigma, Q_f, \Delta)$ is a reduced, complete, and deterministic LHA.

To facilitate the manipulation of RHLs represented by LHAs and/or factor automata, we introduced the following auxiliary notations and operations:

$$
\begin{array}{ll}
\overline{Q'} & := \{Q'' \mid L(\langle Q'', \Delta\rangle) \text{ is a right factor of } L(\langle Q', \Delta\rangle)\},\\
\mathtt{lf}(q) & :=\{(a(q'), q'') \mid (a(q')q'' \to q) \in \Delta\},\\
a(q') \setminus q & :=\{q'' \mid a(q'), q'' \in \mathtt{lf}(q)\},\\
a(q') \setminus Q'' & :=\bigcup_{q''\in Q''}(a(q') \setminus q''),\\
\mathtt{red}_{a(q)}(Q') & :=\{q'' \mid \exists q' \in Q' \text{ such that } (a(q)\,q' \to q'') \in \Delta\},
\end{array}
$$

where Q', Q'' are sets of tates, q, q', q'' are states, and $a \in \Sigma$.

We discovered some nice connections between factor automata and the factors $F_{i,j}$ of the RHL $L = L(A)$:

1. The set of right factors of L is of the form $\{L(\langle Q_i, \Delta\rangle) \mid 1 \leq i \leq n\}$.

2. $F_{i,j} = \mathcal{L}(\langle Q_i, \Delta, Q_j\rangle)$ for all $1 \leq i, j \leq n$.

Before defining our type reconstruction algorithm, we introduced an *internal* representation of types, by pairs of the form $\langle Q_1, \Delta, Q_2\rangle :: \mathcal{Q}$ where

- $\langle Q_1, \Delta, Q_2\rangle$ is a factor automaton, and
- $\mathcal{Q} = \overline{Q'}$ for some set of states Q' such that $Q_1 \in \mathcal{Q}$ and $Q_2 \in \mathcal{Q} \cup \{\{\mathtt{in}(\Delta)\}\}$.

The language of $A :: \mathcal{Q}$ is $L(A :: \mathcal{Q}) := L(A)$. This internal representation has the following useful properties:

1. $\mathtt{Pj}_a(L(\langle Q', \Delta, Q''\rangle :: \mathcal{Q})) = L(\langle Q_1, \Delta\rangle :: \overline{Q_1})$ for all $a \in \Sigma$. (The computations of Q_1 and $\overline{Q_1}$ were explained in [87, Sects.III-A and III-B].) Based on this property, we defined the operation $\mathtt{pj}_a(\langle Q', \Delta, Q''\rangle :: \mathcal{Q}) := \langle Q_1, \Delta\rangle :: \overline{Q_1}$.

2. $(L_1, \ldots, L_n)$ is a factorization of the RHL $L(\langle Q_1, \Delta, Q_{n+1}\rangle :: \mathcal{Q})$ iff there exist Q_2, $\ldots$, $Q_n \in \mathcal{Q}$ such that $L_i = L(\langle Q_i, \Delta, Q_{i+1}\rangle)$ for $1 \leq i \leq n$.

The time reconstruction algorithm proposed by us is designed to work with types τ with the internal representation mentioned before. We write $[\![\tau]\!]$ for the RHL represented by a type τ. Out type system also recognizes the following composite types:

- $\tau_1 \times \cdots \times \tau_n$, whose interpretation is $[\![\tau_1 \times \cdots \times \tau_n]\!] := [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$ for all $n \geq 1$.
- $\tau_1 + \ldots + \tau_n$, whose interpretation is $[\![\tau_1 + \ldots + \tau_n]\!] := \bigcup_{i=1}^{n} [\![\tau_i]\!]$ for all $n \geq 1$.

Thus, the type operators '$\times$' and '+' denote Cartesian product and union, respectively. With these preparations, our type reconstruction problem can be stated more formally as follows:

Given (1) a regular type τ_0 (the *input type*) and (2) a pattern P_0 with variables $\overline{x_1}, \ldots, \overline{x_n}$ and pattern names defined in an environment E

Compute a type

$$\tau_{1,1} \times \cdots \times \tau 1, n + \ldots + \tau_{m,1} \times \cdots \times \tau_{m,n} \tag{4.2}$$

such that there exists $h \in [\![\tau_0]\!]$ and $h \in P_0 \rightsquigarrow \sigma$ if and only if there exists $1 \leq k \leq m$ such that $\sigma(\overline{x_s}) \in [\![\tau_{k,s}]\!]$ for all $1 \leq s \leq n$.

The type reconstruction algorithm is presented in its full splendor in [87, Sect.IV]. The algorithm is called **TI** and is specified as a set of nine inference rules on judgments of the form $K \vdash (P, E) : A :: \mathcal{Q} \rightsquigarrow R$, where

- K is a set of so-called *assumptions*, which are expressions of the form $N : A'$ with N a pattern name and A' a factor automaton,
- P is a subpattern of P_0 in environment E,
- $A :: \mathcal{Q}$ is an internal representation of the input type, and
- R is an internal representation of the type computed for the tuple of variables in the pattern.

The intended reading of this judgement is: "assuming that all type inferences in K have been considered, the type inference for pattern P matched against inputs represented by A yields the internal representation R for the type of tuple of variables of P."

The paper concludes with a discussion on possible optimisations of the type reconstruction algorithm.

4.2.4 Order-sorted unification and matching with regular sorts

There are many other ways to constrain the bindings of sequence variables for programming purposes. Another powerful approach is by using an order-sorted type system which restricts the values of variables to be of regular expression sorts. More precisely, we took interest in the properties of Walther's order-sorted algebra [146] where sorts are strings s of sorts a finite poset $(\mathcal{B}, \preceq)$ of basic sorts, variables have sorts $s \in \mathcal{B}^*$ and functions symbols have sorts $s \to b$ where $s \in \mathcal{B}^*$ and $b \in \mathcal{B}$. We extend this framework by introducing regular expression sorts $\mathtt{R}$ over $\mathcal{B}$, allowing variables to be of sorts $\mathtt{R}$ and function symbols to have sorts $\mathtt{R} \to b$. The obtained signature corresponds to a finite bottom-up hedge automaton. Another extension is that overloading function symbols is allowed. Under some reasonable conditions imposed over the signature [51], terms have the least sort.

If we adopt a rule-based programming framework where the goals and conditional parts of rules can have equational constraints $\tilde{s} \doteq \tilde{t}$ between term sequences in this algebra, we are faced with an unification problem, which we call *order-sorted unification with regular expression sorts* (REOSU for short). When we impose the groundness restriction on $\tilde{t}$, we

are faced with *order-sorted matching problems with regular expression sorts* (REOSM for short): They are sets of matching equations $\tilde{s} \ll \tilde{t}$.

Our contribution

We extended first-order order-sorted unification by permitting regular expression sorts for variables and in the domains of function symbols. The obtained signature corresponds to a finite bottom-up hedge automaton. In this theory, we studied two problems: unification and matching.

We have shown that the unification problem is infinitary, and that the matching problem is finitary. We gave a complete unification procedure and proved decidability of unification. Also, for the matching problem we gave a complete and terminating matching algortihm.

Our results on unification in this theory were presented at the type A International Conference on Rewriting Techniques and Applications (RTA 2010) [81]. The journal version of this paper was published in a type B journal article [82], which also describes the matching algorithm. The remainder of this subsection gives a brief presentation of our results.

Preliminaries

We consider a finite poset $(\mathcal{B}, \preceq)$ of basic sorts, denoted by symbols s, r, possibly subscripted. $s \prec r$ means $s \preceq r$ and $s \neq r$. We write $\mathcal{R}$ for the set of regular expressions over $\mathcal{B}$, which are built in the standard way

$$\mathtt{R} ::= s \mid 1 \mid \mathtt{R}_1.\mathtt{R}_2 \mid \mathtt{R}_1 + \mathtt{R}_2 \mid \mathtt{R}_1^*$$

We write $[\![\mathtt{R}]\!]$ for the regular language represented by $\mathtt{R}$.

A *regular expression sort* is an element of $\mathcal{R}$, and a *functional expression sort* is an expression $\mathtt{R} \to s$ with $\mathtt{R} \in \mathcal{R}$ and $s \in \mathcal{B}$. The relation $\preceq$ on $\mathcal{B}$ is extended to words of basic sorts, sets of words, and regular expression sorts as follows: (1) if $w_1, w_2 \in \mathcal{B}^*$ then $w_1 \preceq w_2$ iff $w_1 = s_1 \dots s_b$, $w_2 = r_1 \dots r_n$ and $s_i \preceq r_i$ for all $1 \leq i \leq n$; if $W_1, W_2 \subseteq \mathcal{B}^*$ then $W_1 \preceq W_2$ iff for each $w_1 \in W_1$ there is $w_2 \in W_2$ such that $w_1 \preceq w_2$; and (3) if $\mathtt{R}_1, \mathtt{R}_2 \in \mathcal{R}$ then $\mathtt{R}_1 \preceq \mathtt{R}_2$ iff $[\![\mathtt{R}_1]\!] \preceq [\![\mathtt{R}_2]\!]$. Note that $\preceq$ is a quasi-order on the sets $\mathcal{B}^*$, $2^{\mathcal{B}^*}$, and $\mathcal{R}$. In particular, we can define the equivalence relation $\simeq$ on $\mathcal{R}$ by: $\mathtt{R}_1 \simeq \mathtt{R}_2$ iff $\mathtt{R}_1 \preceq \mathtt{R}_2$ and $\mathtt{R}_2 \preceq \mathtt{R}_1$. We extend this equivalence relation to functional sorts: $\mathtt{R}_1 \to s_1 \simeq \mathtt{R}_2 \to s_2$ iff $\mathtt{R}_1 \simeq \mathtt{R}_2$ and $s_1 = s_2$.

The *closure* $\overline{\mathtt{R}}$ of $\mathtt{R} \in \mathcal{R}$ is the regular expression defined as follows: $\overline{s} := \sum_{r \preceq s} r$, $\overline{1} := 1$, $\overline{\mathtt{R}_1.\mathtt{R}_2} := \overline{\mathtt{R}_1}.\overline{\mathtt{R}_2}$, $\overline{\mathtt{R}_1 + \mathtt{R}_2} := \overline{\mathtt{R}_1} + \overline{\mathtt{R}_2}$, and $\overline{\mathtt{R}^*} := \overline{\mathtt{R}}^*$. Closures of regular expressions enable the decidability of relations $\preceq$ and $\simeq$ on $\mathcal{R}$: (1) $\mathtt{R}_1 \preceq \mathtt{R}_2$ iff $[\![\overline{\mathtt{R}_1}]\!] \subseteq [\![\overline{\mathtt{R}_2}]\!]$, and (1) $\mathtt{R}_1 \simeq \mathtt{R}_2$ iff $[\![\overline{\mathtt{R}_1}]\!] = [\![\overline{\mathtt{R}_2}]\!]$.

The set of all $\preceq$-maximal elements of a set of sorts $S \subseteq \mathcal{R}$ is denoted $\max(S)$. $\mathtt{R}$ is a lower bound of S if $\mathtt{R} \preceq \mathtt{Q}$ for all $\mathtt{Q} \in S$. A lower bound $\mathtt{G}$ of S is a greatest lower bound, denoted $glb(S)$, if $\mathtt{R} \preceq \mathtt{G}$ for all lower bounds $\mathtt{R}$ of S. Note that if $glb(S)$ exists then it is unique modulo $\preceq$.

For each $\mathtt{R} \in \mathcal{R}$ we assume a countable set of variables $\mathcal{V}_\mathtt{R}$ such that $\mathcal{V}_{\mathtt{R}_1} = \mathcal{V}_{\mathtt{R}_2}$ iff $\mathtt{R}_1 \simeq \mathtt{R}_2$ and $\mathcal{V}_{\mathtt{R}_1} \cap \mathcal{V}_{\mathtt{R}_2} = \emptyset$ if $\mathtt{R}_1 \not\simeq \mathtt{R}_2$. Also, for each $\mathtt{R} \in \mathcal{R}$, $s \in \mathcal{B}$ we asume a set of function symbols $\mathcal{F}_{\mathtt{R} \to s}$ such that $\mathcal{F}_{\mathtt{R}_1 \to s_1} = \mathcal{F}_{\mathtt{R}_2 \to s_2}$ iff $\mathtt{R}_1 \to s_1 \simeq \mathtt{R}_2 \to s_2$. Moreover, the following conditions should be satisfied:

Preregularity: If $f \in \mathcal{F}_{\mathtt{R}_1 \to\ s_1}$ and $\mathtt{R}_2 \preceq \mathtt{R}_1$ then there is a $\preceq$-least element in the set $\{s \mid f \in \mathtt{R} \to s \text{ and } \mathtt{R}_2 \preceq \mathtt{R}\}$.

Finite overloading: For each f, the set $\{\mathcal{F}_{\mathtt{R} \to s} \mid \mathtt{R} \in \mathcal{R}, s \in \mathcal{B}, f \in \mathcal{F}_{\mathtt{R} \to s}\}$ is finite.

We say that $\mathtt{R}$ is a sort of x if $x \in \mathcal{V}_\mathtt{R}$. Similarly, $\mathtt{R} \to s$ is a sort of f if $f \in \mathcal{F}_{\mathtt{R} \to s}$. Function symbols from $\mathcal{F}_{1 \to s}$ are called *constants*. We use the letters $a, b, c$to denote them. We will write $f : \mathtt{R} \to s$ for $f \in \mathcal{F}_{\mathtt{R} \to s}$, $a : s$ for $a \in \mathcal{F}_{1 \to s}$, and $x : \mathtt{R}$ for $x \in \mathcal{V}_\mathtt{R}$. Setting $\mathcal{V} = \bigcup_{\mathtt{R} \in \mathcal{R}} \mathcal{V}_\mathtt{R}$ and $\mathcal{F} = \bigcup_{\mathtt{R} \in \mathcal{R}, s \in \mathcal{B}} \mathcal{F}_{\mathtt{R} \to s}$, we define the sets $\mathcal{T}_\mathtt{R}(\mathcal{F}, \mathcal{V})$ of terms of sort $\mathtt{R} \in \mathcal{R}$ over $\mathcal{V}$ and $\mathcal{F}$, and $\mathcal{TS}_\mathcal{R}(\mathcal{F}, \mathcal{V})$ of term sequences of sort $\mathtt{R} \in \mathcal{R}$ over $\mathcal{V}$ and $\mathcal{F}$, as the least sets satisfying the properties:

- $\mathcal{V}_\mathtt{R} \subseteq \mathcal{T}_\mathtt{R}(\mathcal{F}, \mathcal{V})$
- $\mathcal{T}_{\mathtt{R}'}(\mathcal{F}, \mathcal{V}) \subseteq \mathcal{T}_\mathtt{R}(\mathcal{F}, \mathcal{V})$ if $\mathtt{R}' \preceq \mathtt{R}$
- $\epsilon \in \mathcal{TS}_\mathtt{R}(\mathcal{F}, \mathcal{V})$ if $1 \preceq \mathtt{R}$
- $(t_1, \ldots, t_n) \in \mathcal{TS}_\mathtt{R}(\mathcal{F}, \mathcal{V})$ if there exist $\mathtt{R}_1, \ldots, \mathtt{R}_n \in \mathcal{R}$ such that $t_i \in \mathcal{T}_{\mathtt{R}_i}(\mathcal{F}, \mathcal{V})$ and $\mathtt{R}_1. \cdots .\mathtt{R}_n \preceq \mathtt{R}$.
- $f(\tilde{t}) \in \mathcal{T}_\mathtt{R}(\mathcal{F}, \mathcal{V})$ if $\mathtt{R} = s$, $f : \mathtt{R}' \to s$ and $\tilde{t} \in \mathcal{TS}_{\mathtt{R}'}(\mathcal{F}, \mathcal{V})$.

The set of terms over $\mathcal{V}$ and $\mathcal{F}$ is defined as $\mathcal{T}(\mathcal{F}, \mathcal{V}) = \bigcup_{\mathtt{R} \in \mathcal{R}} \mathcal{T}_\mathtt{R}(\mathcal{F}, \mathcal{V})$. We abbreviate terms $a(\varepsilon)$ with a. The *depth* of a term and a term sequence is defined in the standard way:

$depth(x) := 1$, $depth(f(\tilde{t})) := 1 + depth(\tilde{t})$, $depth(\epsilon) := 0$, and $depth(t_1, \ldots, t_n) := \max\{depth(t_i) \mid 1 \leq i \leq n\}$ if $n > 0$.

It can be shown that every term t has a least sort modulo $\simeq$, and we denote it by $lsort(t)$. In the same way, the $\preceq$-least sort of a term sequence $(t_1, \ldots, t_n)$, $n \geq 1$, is defined uniquely modulo $\simeq$ as $lsort(t_1).\cdots.lsort(t_n)$ and is denoted by $lsort(t_1, \ldots, t_n)$. When $n = 0$, i.e., for the empty sequence, $lsort(\epsilon) = 1$.

For a basic sort $s \in \mathcal{B}$, its semantics $sem(s)$ is the set $\mathcal{T}_s(\mathcal{F})$ of ground terms of sort s. The semantics of a regular sort is given by the set of ground term sequences of the corresponding sort: $sem(1) := \{\epsilon\}$, $sem(\mathtt{R}_1.\mathtt{R}_2) := \{(\tilde{s}_1, \tilde{s}_1) \mid \tilde{s}_1 \in sem(\mathtt{R}_1), \tilde{s}_2 \in sem(\mathtt{R}_2)\}$, $sem(\mathtt{R}_1 + \mathtt{R}_2) := sem(\mathtt{R}_1) \cup sem(\mathtt{R}_2)$, and $sem(\mathtt{R}^*) := sem(\mathtt{R})^*$. This definition, together with the definition of $\preceq$ and $\mathcal{T}_{\mathtt{R}}(\mathcal{F}, \mathcal{V})$, implies that if $\mathtt{R} \preceq \mathtt{Q}$, then $sem(\mathtt{R}) \subseteq sem(\mathtt{Q})$.

A substitution is a well-sorted mapping from variables to term sequences. Substitutions are denoted with lowercase Greek letters, where ε stands for the identity substitution. Well-sortedness of σ means that $lsort(\sigma(x)) \preceq lsort(x)$ for all $x \in \mathcal{V}$. The *depth* of a substitution is defined as $depth(\sigma) := \max\{depth(x\sigma) \mid x \in \mathcal{V}\}$.

We note that the relations $lsort(t\sigma) \preceq lsort(t)$ and $lsort(\tilde{t}\sigma) \preceq lsort(\tilde{t})$ hold for any term t, term sequence $\tilde{t}$ and substitution σ [81, Lemma 3].

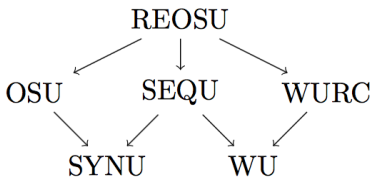
An equation is a pair of term sequences, written as $\tilde{s} \doteq \tilde{t}$. Its *depth* is the maximum between $depth(\tilde{s})$ and $depth(\tilde{t})$. A regular expression order sorted unification or, shortly, REOSU problem is a finite set of equations $\Gamma = \{\tilde{s}_1 \doteq \tilde{t}_1, \ldots, \tilde{s}_n \doteq \tilde{t}_n\}$. A substitution σ is a unifier of Γ if $\tilde{s}\sigma = \tilde{t}\sigma$ for all $1 \leq i \leq n$. A *minimal complete set of unifiers* of Γ is a set U of unifiers of Γ satisfying the following conditions:

Completeness: For any unifier θ of Γ there is $\sigma \in U$ such that $\sigma \leq \theta\ [var(\Gamma)]$.

Minimality: If $\sigma_1, \sigma_2 \in U$ such that $\sigma_1 \leq \sigma_2\ [var(\Gamma)]$ then $\sigma_1 = \sigma_2$.

The *depth* of a REOSU problem Γ is the maximum depth of the equations it contains.

REOSU extends some known problems as shown on the diagram below, illustrating its relations with syntactic unification (SYNU [128]), word unification (WU [135]), order-sorted unification (OSU [146]), sequence unification (SEQU [76]), and word unification with regular constraints (WURC [135]):



Following the arrows, the problems are related as follows:

- From OSU one can obtain SYNU by restricting the sort hierarchy to be empty.
- SEQU problems without sequence variables (i.e., with individual variables only) are SYNU problems.
- WU is a special case of SEQU with constants, sequence variables, and only one flexible arity function symbol for concatenation.
- WU is also a special case of WURC where none of the variables are constrained.
- From REOSU we can get OSU (but with finitely many basic sort symbols only, because this is what REOSU considers) if instead of arbitrary regular sorts in function domains we allow only words over basic sorts, restrict variables to be of only basic sorts, and forbid function symbol overloading.
- SEQU can be obtained if we restrict REOSU with only one basic sort, say s, the variables that correspond to sequence variables in SEQU have the sort s^*, individual variables are of the sort s, and function symbols have the sort $s^* \to s$.
- WURC can be obtained from REOSU by the same restriction that gives WU from SEQU and, in addition, identifying the constants there to the corresponding sorts.

Order-sorted unification described in [133, 147] extends OSU from [146] in a way that is not compatible with REOSU.

We recall the notion of linear form for regular expressions from [4] by adapting the notation to our setting and using the set of basic sorts $\mathcal{B}$ for alphabet. This notion, together with the split of a regular expression, will be needed later, in sort-related algorithms: When we decompose a hedge in the weakening process, we have to split the corresponding sort as well. Linear form helps to split a sort into a basic sort and another sort, while the split operation decomposes it into two (not necessarily basic) sorts.

A pair $(s, \mathtt{R})$ is called a *monomial*. A *linear form* of a regular expression $\mathtt{R}$, denoted by $lf(\mathtt{R})$, is a finite set of monomials defined recursively as follows:

$$\begin{array}{rclcrcl} lf(1) & = & \emptyset & & lf(\mathtt{R}^*) & = & lf(\mathtt{R}) \odot \mathtt{R}^* \\ lf(s) & = & \{(s,1)\} & & lf(\mathtt{R}.\mathtt{Q}) & = & lf(\mathtt{R}) \odot \mathtt{Q} \text{ if } \epsilon \notin [\![\mathtt{R}]\!] \\ lf(s+r) & = & lf(s) \cup lf(r) & & lf(\mathtt{R}.\mathtt{Q}) & = & lf(\mathtt{R}) \odot \mathtt{Q} \cup lf(\mathtt{Q}) \text{ if } \epsilon \in [\![\mathtt{R}]\!] \end{array}$$

These equations involve an extension of concatenation $\odot$ that acts on a linear form and a regular expression and returns a linear form. It is defined as $l \odot 1 := l$ and $l \odot \mathcal{Q} := \{(s, \mathtt{S}.\mathtt{Q}) \mid (s, \mathtt{S}) \in l, \mathtt{S} \neq 1\} \cup \{(s, \mathtt{Q}) \mid (s, 1) \in l\}$ if $\mathtt{Q} \neq 1$.

The set $\hat{lf}(\mathtt{R})$ is defined as $\{s.\mathtt{Q} \mid (s, \mathtt{Q}) \in lf(\mathtt{R})\}$.

A *split* of $\mathtt{S} \in \mathcal{R}$ is a pair $(\mathtt{Q}, \mathtt{R}) \in \mathcal{R}^2$ such that (1) $\mathtt{Q}.\mathtt{R} \preceq \mathtt{S}$ and (2) if $(\mathtt{Q}', \mathtt{R}') \in \mathcal{R}^2$ and $[\![\mathtt{Q}]\!] \subseteq [\![\mathtt{Q}']\!]$, $[\![\mathtt{R}]\!] \subseteq [\![\mathtt{R}']\!]$, $[\![\mathtt{Q}'.\mathtt{R}']\!] \subseteq [\![\mathtt{S}]\!]$, then $[\![\mathtt{Q}]\!] = [\![\mathtt{Q}']\!]$ and $[\![\mathtt{R}]\!] = [\![\mathtt{R}']\!]$.

Splits correspond to 2-factorizations in the following sense: $(\mathtt{Q}, \mathtt{R})$ is a split of $\mathtt{S}$ iff $(\overline{\mathtt{Q}}, \overline{\mathtt{R}})$ is a 2-factorization of $\overline{\mathtt{S}}$ [81, Lemma 4]. In [33] it has been shown that the 2-factorizations of a regular expression are finitely many modulo $\simeq$, and that they can be effectively computed. Thus, a regular expression has finitely many splits modulo $\simeq$ that can be effectively computed.

Relating REOS Signatures and Hedge Automata

Regular expression ordered sorts are related to regular hedge automata in the same way as ordered sorts are related to tree automata. Namely, a REOS signature is a finite bottom-up hedge automaton.

To illustrate this relation, we first recall the definition of nondeterministic finite hedge automaton (NFHA) from [31]: An NFHA over Σ is a tuple (Q, Σ, Q_f, Δ) where Q is a finite set of states, $Q_f \subseteq Q$ is a set of final states, and Δ is a finite set of transition rules of the following types: (1) $a(R) \to q$ where $a \in \Sigma$, $q \in Q$, and $R \subseteq Q^*$ is a regular language over Q, or (2) $q' \to q$ (called ϵ-transitions) where $q, q' \in Q$. Now, we can take our set of basic sorts $\mathcal{B}$ in the role of Q, the set $\mathcal{F}$ in the role of Σ, assume $Q_f = Q$, and define Δ as follows: For each $r \prec s$, the ϵ-transition rule $r \to s$ is in Δ. For each $f : \mathtt{R} \to s$, the rule $f(\mathtt{R}) \to s$ is also in Δ. It is easy to see that our ground terms are exactly the unranked trees recognised by this automaton. A ground term of sort s is an unranked tree recognised by the automaton at state s.

Sort-Related Algorithms

The sort-related algorithms of interest to us are: (1) to decide $\preceq$ on $\mathcal{R}$; (2) to compute $glb(\mathtt{S}, \mathtt{R})$ for $\mathtt{S}, \mathtt{R} \in \mathcal{R}$, and (3) to compute sort-weakening substitutions.

Deciding $\preceq$

We showed that $\mathtt{S} \preceq \mathtt{R}$ iff $[\![\overline{\mathtt{S}}]\!] \subseteq [\![\overline{\mathtt{R}}]\!]$. To decide $[\![\overline{\mathtt{S}}]\!] \subseteq [\![\overline{\mathtt{R}}]\!]$, we can use Antimirov's algorithm [3] that employs partial derivatives. The problem is PSPACE-complete, but Antimirov's rewriting approach has an advantage over the standard technique of translating regular expressions into automata: With it, in some cases solving derivations can have polynomial size, while any algorithm based on translation of regular expressions into DFAs causes an exponential blow-up.

Computing greatest lower bounds

A greatest lower bound of regular expressions would be their intersection, if we did not have an ordering on the basic sorts. It can be computed either by translating them into automata, or by Antimirov & Mosses's rewriting algorithm [5] for regular expressions extended with the intersection operator. The computation requires double exponential time.

Here we can employ the regular expression intersection algorithm [5] to compute a greatest lower bound, with one modification: To compute the intersection between two alphabet letters (i.e. between two basic sorts), instead of standard check whether they are the same, we compute the maximal elements in the set of their lower bounds. There can be several such maximal elements. This can be easily computed based on the ordering on basic sorts. Then we can take the sum of these elements and it will be a greatest lower bound. This construction allows to compute a greatest lower bound of two regular expressions, which is unique modulo $\simeq$.

An implementation of Antimirov-Mosses algorithm [138] requires only minor modifications to deal with the ordering on alphabet letters (basic sorts). Hence, for $\mathtt{S}$ and $\mathtt{R}$ we compute here $glb(\mathtt{S}, \mathtt{R})$ and we know that if $\mathtt{Q}$ is a regular expression with $[\![\mathtt{Q}]\!] = [\![\mathtt{S}]\!] \cap [\![\mathtt{R}]\!]$, then $glb(\mathtt{S}, \mathtt{R}) \simeq \mathtt{Q}$.

Computing Weakening Substitutions

Now we describe an algorithm that computes a substitution to weaken the sort of a term sequence towards a given sort. The following example illustrates a situation when such an algorithm is needed. Assume we want to unify the terms x and $f(y)$ where we know that $x : s$, $f : \mathtt{R}_1 \to s_1$, $f : \mathtt{R}_2 \to s_2$, $y : \mathtt{R}_2$, $s_1 \prec s \prec s_2$, and $\mathtt{R}_1 \prec \mathtt{R}_2$. We can not unify x with y directly because $lsort(f(y)) = s_2 \not\preceq s = lsort(x)$. But, if we weaken the least sort of $f(y)$ to s_1 then unification is possible. To weaken the least sort of $f(y)$, we take its instantiation under substitution $\{y \mapsto z\}$, where $z \in \mathcal{V}_{\mathtt{R}_1}$, which gives $lsort(f(z)) = s_1$. Thus, the substitution $\{y \mapsto z, x \mapsto f(z)\}$ is a unifier of x and $f(y)$, leading to the common instance $f(z)$.

A *weakening pair* is a pair $\tilde{t} \rightsquigarrow \mathtt{Q}$ made of a term sequence $\tilde{t}$ and a regular sort $\mathtt{Q}$. A substitution ω is called a *weakening substitution* of a set W of weakening pairs iff $lsort(\tilde{t}\omega) \preceq \mathtt{Q}$ for each $\tilde{t} \rightsquigarrow \mathtt{Q} \in W$.

Our weakening algorithm is called $\mathfrak{W}$, and works by applying exhaustively the following rules to pairs $W; \sigma$ where W is a set of weakening pairs and σ is a substitution:

R-w: Remove a Weakening Pair
$\{\tilde{t} \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow W; \sigma$ if $lsort(\tilde{t}) \preceq \mathsf{Q}$.

D1-w: Decomposition 1 in Weakening
$\{(f(\tilde{t}), \tilde{s}) \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow \{f(\tilde{t}) \rightsquigarrow \mathsf{s}, \tilde{s} \rightsquigarrow \mathsf{S}\} \cup W; \sigma$
if $lsort(f(\tilde{t}), \tilde{s}) \not\preceq \mathsf{Q}$, $var(f(\tilde{t}), \tilde{s}) \neq \emptyset$, $\tilde{s} \neq \epsilon$ and $\mathsf{s.S} \in \max(\hat{lf}(\mathsf{Q}))$.

D2-w: Decomposition 2 in Weakening
$\{(x, \tilde{s}) \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow \{x \rightsquigarrow \mathsf{Q}_1, \tilde{s} \rightsquigarrow \mathsf{Q}_2\} \cup W; \sigma$
if $lsort(x, \tilde{s}) \not\preceq \mathsf{Q}$, $\tilde{s} \neq \epsilon$ and $(\mathsf{Q}_1, \mathsf{Q}_2)$ is a split of Q.

AS-w: Argument Sequence Weakening
$\{f(\tilde{t}) \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow \{\tilde{t} \rightsquigarrow \mathsf{R}\} \cup W; \sigma$
where $lsort(f(\tilde{t})) \not\preceq \mathsf{Q}$, $var(f(\tilde{t})) \neq \emptyset$, $\mathsf{R.r}$ is a maximal sort such that $f \in \mathcal{F}_{\mathsf{R}\to\mathsf{r}}$ and $\mathsf{r} \preceq \mathsf{Q}$.

V-w: Variable Weakening
$\{x \rightsquigarrow \mathsf{Q}\} \cup W; \sigma \Longrightarrow W\sigma; \sigma\{x \mapsto w\}$
where $\mathrm{glb}(\{lsort(x), \mathsf{Q}\}) \neq \bot$ and w is a fresh variable from $\mathcal{V}_{\mathrm{glb}(\{lsort(x),\mathsf{Q}\})}$.

If none of the rules are applicable to $W; \sigma$ then it is transformed into $\bot$, indicating failure. By exhaustive search, transforming each $W; \sigma$ in all possible ways, we generate a complete search tree whose branches form *derivations*. The branches that end with $\bot$ are called failing branches. The branches that end with $\emptyset; \omega$ are called successful branches and ω is the substitution computed by $\mathfrak{W}$ along this branch. The set of all substitutions computed

by $\mathfrak{W}$ starting from $W;\epsilon$ is denoted by $weak(W)$. It is easy to see that the elements of $weak(W)$ are variable renaming substitutions.

It is essential that the signature has the finite overloading property, which guarantees that the rule AS-w does not introduce infinite branching. Since the linear form and split of a regular expression are both finite, the other rules do not cause infinite branching either.

We proved that $\mathfrak{W}$ is terminating (Thm. 1), sound (Thm. 2), and complete (Thm. 3).

The unification type

Let Γ_{re} be a REOSU problem and Γ_{seq} its version without sorts, i.e. a SEQU problem. Each unifier of Γ_{re} is either a unifier of Γ_{seq} or is obtained from a unifier of Γ_{seq} by composing it with a weakening substitution as follows: If $\sigma = \{x_1 \mapsto \tilde{t}_1, \ldots, x_n \mapsto \tilde{t}_n\}$ is a unifier of Γ_{seq}, then the set of weakening substitutions for σ is $\Omega(\sigma) = weak(\{\tilde{t}_1 \rightsquigarrow lsort(x_1), \ldots, \tilde{t}_n \rightsquigarrow lsort(x_n)\})$. For each $\omega_\sigma \in \Omega(\sigma)$, $\sigma\omega_\sigma$ is a unifier of Γ_{re}. Since SEQU is infinitary, the type of REOSU can be either infinitary or nullary. In [81] we have shown that REOSU is not nullary.

The unification procedure

One way to compute the unifiers of a REOSU problem is to compute with the SEQU procedure [76] the unifiers of the corresponding unsorted problem and then weaken every computed substitution to obtain their order-sorted instantiations. This approach is not uncommon in order-sorted unification: It is a modular method that reuses an existing solving procedure. (See, e.g., [133, 112, 57].) A drawback of this approach is that it is a generate-and-test method. It is not able to detect early enough derivations that fail because of sort incompatibility. Early failure detection requires weakening to be tailored in the unification rules.

To overcome these drawbacks, we have designed a set of ten transformation rules that act on pairs $\Gamma;\sigma$ made of a unification problem Γ and a substitution σ, and proved that they define a sound and complete rule-based procedure for REOSU problems [81, Theorems 4-5].

Decidability of REOSU

We proved decidability of REOSU through a translation into a word equation with regular constraints. Since the latter problem is known to be decidable, we concluded that REOSU is decidable too [81, Theorem 7].

The matching algorithm

A matching equation is a pair of term sequences $\tilde{s} \ll \tilde{t}$, where $\tilde{t}$ is ground. A *regular expression order sorted matching* problem or, shortly, a REOSM problem is a finite set of matching equations. A substitution φ is a *matcher* of a REOSM problem$\{\tilde{s}_1 \ll \tilde{t}_1, \ldots, \tilde{s}_n \ll \tilde{t}_n\}$ iff $\tilde{s}_i\varphi = \tilde{t}_i$ for all $\leq i \leq n$.

REOSM is a special case of REOSU. Unlike REOSU, in REOSM there is no need to compute weakening substitutions: Solving the regular language membership problem suffices. The rules of the REOSM procedure can be formulated as follows:

`T-M`: Trivial

$$\{\epsilon \ll \epsilon\} \uplus \Gamma; \varphi \Longrightarrow \Gamma; \varphi$$

`D-M`: Decomposition

$$\{(f(\tilde{t}), \tilde{t}') \ll (f(\tilde{s}), \tilde{s}')\} \uplus \Gamma; \varphi \Longrightarrow \{\tilde{t} \ll \tilde{s}, \tilde{t}' \ll \tilde{s}'\} \cup \Gamma; \varphi$$

`E-M`: Elimination

$$\{(x, \tilde{t}) \ll (\tilde{s}, \tilde{s}')\} \uplus \Gamma; \varphi \Longrightarrow \{\tilde{t}\vartheta \ll \tilde{s}'\} \cup \Gamma\vartheta; \varphi\vartheta$$

if $lsort(\tilde{s}) \preceq lsort(x)$ and $\vartheta = \{x \mapsto \tilde{s}\}$.

To match a term sequence $\tilde{s}$ to a ground term sequence $\tilde{t}$, we create the initial system $\{\tilde{s} \ll \tilde{t}\}; \epsilon$ and apply the rules exhaustively as long as it is possible. Problems to which no rule applies are transformed into $\bot$. The REOSM algorithm defined in this way is denoted by $\mathfrak{M}$.

The rule `E-M` is the only one which makes a choice: There can be various ways to split the term sequence in the right hand side of the selected equation into $\tilde{s}$ and $\tilde{s}'$ such that the rule condition is satisfied.

Derivations are sequences of rule applications. A derivation of the form $\Gamma; \epsilon \Longrightarrow^* \emptyset; \varphi$ is called a successful derivation and φ is called a computed substitution of Γ. We denote the set of substitutions computed by $\mathfrak{M}$ for Γ with $comp_{\mathfrak{M}}(\Gamma)$.

It is easy to check that the matching rules above are sound, i.e., every computed substitution of Γ is a matcher of Γ. In [82], we also showed that the matching procedure induced by the three rules of $\mathfrak{M}$ is terminating (Theorem 8.2) and that it computes a minimal complete set of matchers (Theorem 8.3). We also showed, that REOSM is NP-complete (Theorem 8.4) by defining a polynomial-time reduction to REOSU of 1-IN-3-SAT problems [132] which are known to be NP-complete.

4.2.5 Constraint logic programming for hedges

The manipulation of hedges has been intensively studied in recent years in the context of XML processing, rewriting, automated reasoning, knowledge representation, just to name a few. We have already noticed that, when working with unranked terms (i.e., terms built wit variadic function symbols), it becomes very convenient to use *hedge variables* (a.k.a. *sequence variables*), because they help to write neat and compact code. Recent years have seen the emergence of more and more languages that operate on unranked terms and hedges. An interesting newcomer was the CLP(Flex) schema [25], which extends the Constraint Logic Programming schema to work with hedges and is a basis for the XML processing language XCentric [27] and of a web site verification language VeriFLog [26].

To provide adequate support to constraint logic programming with hedges, we introduced a very general instantiation of the general constraint logic programming scheme for the domain H of hedges. We called this language CLP(H). In this scheme, hedges are finite sequences of unranked terms, built over variadic function symbols and three kinds of variables: for terms, for hedges, and for function symbols. Moreover, we may have function symbols whose argument order does not matter (unordered symbols): a kind of generalisation of the commutativity property to unranked terms.

As it turns out, such a language is very flexible and permits to write short, yet quite clear and intuitive code. Constraints involve equations between unranked terms and atoms for regular hedge language membership. We studied the algebraic semantics of CLP(H) programs and defined a sound, terminating and incomplete constraint solver. Also, we investigated two classes of constraints for which the solver defined by us returns a complete set of solutions, and described classes of programs that generate such constraints.

These results of ours were presented at the 12th International Symposium in Functional Logic Programming Flops 2014 [37], and published in a TPLP journal article [38].

The language CLP(H) generalizes CLP(Flex) with function variables, unordered functions, and membership constraints. Hence, as a special case, our paper describes the semantics of CLP(Flex). Moreover, as hedges generalize strings, CLP(H) can be seen also as a generalization of CLP over strings CLP(S) [126], string processing features of Prolog III [28], and CLP over regular sets of strings CLP(Σ^*) [145].

The flexibility and the expressive power of CLP(H) have their price: Equational constraints with hedge variables, in general, may have infinitely many solutions (Kutsia 2004;

2007). Therefore, any complete equational constraint solving procedure with hedge variables is nonterminating. The solver we describe in this paper is sound and terminating, hence incomplete for arbitrary constraints. However, there are fragments of constraints for which it is complete, i.e., computes all solutions. One such fragment is the so called *well-moded fragment*, where variables in one side of equations (or in the left hand side of the membership atom) are guaranteed to be instantiated with ground expressions at some point. This effectively reduces constraint solving to hedge matching [77, 78], plus some early failure detection rules. Another fragment for which the solver is complete is named after the Knowledge Interchange Format, KIF [49], where hedge variables are permitted only in the last argument positions. We identify forms of CLP(H) programs which give rise to well-moded or KIF constraints.

Chapter 5

Development plan

5.1 Background

I started my academic career in 1993, as Instructor (preparator) at the Department of Computer Science of University of Timişoara. My teaching duties in 1993–1995 included preparing seminars and labs for lectures on Functional Programming, Logic Programming, AI, and Data Structures.

In the next 9 years (1995-2004) I quit didactic activities and focused on doing research in the study of computational models for declarative programming: as PhD student at Research Institute for Symbolic Computation (1995-2004), as JSPS postdoc at Institute of Electronic Sciences and Engineering from University of Tsukuba (05/2000-04/2002), and as senior researcher at the Research Institute for Computational and Applied Mathematics (RICAM) of the Austrian Academy of Sciences (03/2003-09/2004). As a PhD student RISC-Linz, I worked on various research projects, including:

- *HPGP – High-Performance Generic Programming* (1996-1998), supported by an Austrian Science Foundation grant to develop an generic environment for high-performance mathematical libraries [134].

- *Constraint Solving for Functional Logic Programming* (07/1997-06/1999), supported by Research Institute for Advanced Information Technology (AITEC Japan) to develop a distributed software system made of a functional logic language interpreter and various constraint solving engines.

- *Theorema* project [20] initiated around 1995 by Bruno Buchberger for computer-

supported mathematical theorem proving and theory exploration. I was a voluntary member (1996-2000), and my contributions were acknowledged in [19, 18, 17].

During my PhD studies, I collaborated with researchers from RISC-Linz on topics related to symbolic computation and automated deduction (Bruno Buchberger and the developers of Theorema system), constraint logic programming and distributed computing (professors Hoon Hong and Wolfgang Schreiner), and with researchers from University of Tsukuba on topics related to functional logic programming and lazy narrowing (professor Tetsuo Ida).

After getting a PhD degree from Johannes Kepler University of Linz with the dissertation "Functional Logic Programming with Distributed Constraint Solving," I decided to enrich my experience in this field with a postdoc at University of Tsukuba, and refocused on the study of collaborative constraint solving systems in open environments where solvers can be deployed as discoverable services.

In March 2003, I returned to Austria as research scientist in the symbolic computation group of RICAM institute affiliated with Austrian Academy of Sciences. I refocused my research on the study of rule-based programming with programmatic support for strategies. I establishing new collaborations with researchers from RISC-Linz institute, and continued working with my former colleagues from University of Tsukuba.

In October 2004, I returned to teaching activities, as Assistant Professor at the Department of Computer Science from University of Tsukuba, Japan. Besides teaching duties, I continued to do research on rule-based programming and closely related topics. I focused on identifying more applications for this programming approach, and on extending my computational model with new capabilities (e.g., membership constraints, type inference systems) that increase its expressive power and range of applications. My work was funded by two grants having me as main investigator: a JSPS Grant-in-Aid for Young Researchers (04/2005–03/2007) for the project "Rule-based Programming: Design and Applications," and a JSPS Grant-in-Aid for Scientific Research (04/2008–03/2011) for the project "Applications of rule-based programming to verification and transformation of XML."

In October 2011, I changed my affiliation and became Assistant Professor at the Department of Computer Science from West University of Timişoara, and since 2015 I am Associate Professor. My teaching duties included lectures related to my research interests, like Logic Programming, Functional Programming, and Advanced Functional and Logic Programming. While continuing international collaborations with my old coworkers, I found

many common research interests with members from my department.

I coauthored the publication of one book, one journal article A, 2 journal articles B, one journal article C, five conferences in tier A, 8 conferences in tier B, and 12 conferences in tier C. According to the current evaluation criteria, my scientific production has the following citation rankings: 78 by A category, and 64.33 by B category.

5.2 Planned directions of research

I am very much interested in the study of computational models for multiparadigm declarative programming, and how to adapt them to make best use of the availability of distributed and parallel computing environments. The goal is to develop software tools that assist their users to solve problems by specifying their problem-specific knowledge and requirements in a formally precise way, and relying on the availability of a black-box execution model which is smart enough to compute efficiently all answers of interest.

Rewrite theories, which are at the core of both rule-based programming and constraint functional logic programming, provide an adequate representation for a wide variety of applications, including: automated deduction, software and hardware specification and verification, security, real-time and cyber-space systems, probabilistic systems, bioinformatics, and chemical systems. (See [111] for a convincing account.) This large spectrum of potential applications makes the study of rule-based programming paradigms and related tools a very appealing direction of research.

The study of rewrite theories and corresponding computational models is a well established and still very active area of research. We will continue to focus on the following directions of improvement:

D1. Rewrite theories for larger fragments of logic, where we can express more general notions. We saw that hedges, contexts, and regular membership constraints to write neat and compact specifications, but working with them poses significant challenges to language implementors. We will continue to analyse these formalisms, and to implement suitable computational procedures to work with them:

D1.1 unification and matching algorithms, and

D1.2 type inference systems

D2. Adapting the computational models to take advantage of the availability of distributed and parallel computing resources in their proximity. A well-known approach is to decompose the overall computational model into subprocesses that collaborate as prescribed by a strategy represented in a declarative language. We followed this approach in the implementation of Open CFLP, where the elementary collaboratives are interfaces to constraint solving services running remotely in a distributed environment.

    Nowadays, the availability of resources and middleware for cloud computing and grid computing simplifies a lot the design and implementation of the collaborative computational models envisioned by us, and many researchers from institute e-Austria Timişoara have core competencies in the areas of parallel computing (HPC&Cluster) and distributed computing (Cloud&Grid). Therefore, I intend to enlarge my network of cooperations with people from institute e-Austria, to develop the collaborative computational models envisioned by me, and to evaluate their performance.

D3. Efficient coordination models for distributed and/or parallel computing. This problem occurs in connection with the implementation of strategy combinators whose execution can be parallelised or pipelined. This is a highly technical issue that must be addressed: It does not affect the correctness of the computation model but can have a great impact on its performance. Therefore, users should have limited programmatic support to control these issues and leave the coordination details to the implementers of the computation model. A similar approach was adopted by the designers of Java 8 to support pipeline processing with streams: The user can choose to work with streams in `parallel` mode, and the way how this choice affects efficiency is left to the language implementation.

    We addressed some of these issues in the design of the broker of Open CFLP, by implementing a load balancing policy for its solving agents.

A recent research interest of mine is active learning. This is a family of machine learning methods which yield highly accurate characterisations of data sets by iterative selections of informative query instances answered by an oracle (e.g., a human). According to [137], this kind of work tries, essentially, to give a positive answer to the question: *Can machines learn with fewer labeled training instances if they are allowed to ask questions?*

This is a joint research theme with Gabriel Istrate from West University of Timişoara,

and we presented some of our results in two tier B publications [93, 94].

5.3 Scientific development plan

A constant rhythm of R&D is essential to the development of a high-profile scientist. This activity should address topics with a significant scientific and industrial value, and should be disseminated to attract both (1) funding agencies and partners who show interest in the use and further development of his achievements, and (2) researchers willing to collaborate and students willing to get expertise in that research direction and, hopefully, to follow an academic or closely related career afterwards.

To achieve these goals, I will focus on attaining the objectives mentioned below.

Attracting funds

My planned directions of research have applications for a wide variety of applications. Therefore, I feel motivated to build up a research team motivated to work in this field. My first target are students who can be involved for at least 1 or 2 years in projects developed within the university, and be payed from R&D grants and projects obtained through national or international competitions. There is no doubt that R&D grants and projects are essential to establish research teams, raise their professional level, and to sustain financially the academic activities in general. Students involved in such projects have high chances to become good researchers, and would become a valuable source to for people interested in an academic career. Since I returned to the Department of Computer Science of West University of Timişoara in 2011, I try constantly to attract funds to support both students and professors (including myself) in their research, by submitting proposals to national (UEFISCDI) and international (EU) competitions.

Establishing a strong interest group

I am fortunate to be member of a department with high potential energy to do valuable research, and with a couple of people with whom I share research interests. In our department we established the Theoretical Computer Science Group, an enthusiastic group of researchers within whom I found some new interdisciplinary directions of research. Its members published their results in journals like: Journal of Symbolic Computation, Theory and Prac-

tice of Logic Programming, Scientific Annals of Computer Science, Theoretical Computer Science, Information Processing Letters, and in conference proceedings like International Colloquium on Automata, Languages and Programming, and Symposium on Functional and Logic Programming (FLOPS). More information about the research interests and scientific achievements of this group can be found at the website `http://tcs.ieat.ro`

We wish to increase the scientific visibility and expertise of this group by

- Enlarging the group, by attracting students and fellow researchers eager to collaborate or to get acquainted with our research interests
- Identifying new research directions of common interest, as well as directions of interdisciplinary research between the members of the group
- Establishing reading groups, where to discuss new papers and and latest trends in ongoing research.
- Publishing high quality papers and presenting them at well-established international events.

Obviously, the success of this objective depends largely on the success of its members (including myself) to attract funds to support participation at major scientific events (conferences, summer schools, workshops), the establishment of an adequate research environment (documentation, equipment), and sustained collaboration with members of the international community.

Consolidating international collaborations

This is a necessary condition to obtain valuable scientific results. A recent outcome of my international collaborations was the publication of a scheme CLP(H) for constraint logic programming with hedges [38], elaborated together with researchers from Johannes Kepler University, Porto University, and Tbilisi State University. We continue working in this direction. In 2014 and 2016 I took advantage of ERASMUS teaching mobilities to visit fellow researchers form Johannes Kepler University and pursue joint research. I will continue to use all opportunities to strengthen my international network of collaborators.

Valuable scientific publications

Good papers published in journals or presented at high profile conferences increase the visibility and credibility of research. In the last years, I coauthored two type A papers [37, 38], and three type B papers [93, 94, 82]. These achievements give me confidence that our research results are well appreciated and will continue to get high visibility.

Strengthening the research position of UVT

Currently, our Department of Computer Science does active research in three research areas: distributed computing, and artificial intelligence, and theoretical computer science. Although we are only a handful of staff, there is potential to increase significantly the research track of our department. My goal is to add rule-based programming to this list, by contributing with my expertise accumulated abroad. To fulfil it, I plan to increase the visibility of my research and write more convincing project proposals to attract funding.

Another way to strengthen the research position of UVT is by organising and participating at scientific events. The Symbolic and Numeric Symposium on Scientific and Numeric Computing (SYNASC) is organised every year by our faculty at our university. Since its inception in 1999, I was every year a PC committee member of this international event. From 2011 I also started to act as PC chair of the special track "Advances in the Theory of Computing."

5.4 Academic development plan

Research activities should go hand in hand with didactic activities centered around the student. As professors, we should avoid teaching students just the basic IT technologies required by the current needs of industry. Instead, we should focus on familiarizing them with the principles of software engineering and programming paradigms that enable them to analyse rigorously a problem, find an algorithmic solution, and analyse the complexity of his/her solution. Students should be taught to appreciate the strengths and weaknesses of various programming styles, and how to use their core concepts to model the problems they must solve. Only afterwards should they decide what programming language(s) to choose to write down their concrete implementation.

My main academic objectives are the following.

Publication of materials to support my didactic activities

My intention is to publish materials about various declarative programming paradigms and substyles, as material to support my courses. The emphasis will be on the programming philosophy without delving into the intricacies of a particular programming language. So far, I published one book: "Principles and Practice of Functional Programming," which focuses on describing general principles of functional programming, and programming idioms specific to the functional way of thinking. I chose Scheme as a concrete functional language, but I expect students to have no difficulties to switch easily to another functional programming language, and to be able to appreciate the similarities and differences between them. I intend to prepare similar materials about the other declarative programming styles, as support for my lecture on Advanced Logic and Functional Programming.

Dissemination of knowledge related to my research concerns

I intend to prepare an advanced lecture on Advanced Functional and Logic Programming and Rule-based Programming, and include some parts of my research results in the lectures offered to our Master students. So far, I taught these lectures at RISC-Linz Institute and Johannes Kepler University of Linz, in the frame of two ERASMUS teaching mobilities, in 2014 and 2016, and they were well received.

Updated curriculum

Declarative programming concepts start to play a more and more prominent role in the design of modern programming languages, and programming concepts introduced long ago in functional programming languages (such as `lambdas`, closures, generic types, streams, `auto` variables) start to show up in the latest versions of popular programming languages (e.g., C++11, and Java 8).

These trends illustrate the advent of a new multiparadigm programming style, which aims to integrate object-oriented programming with advanced features of functional programming and logic programming. To fully appreciate the power of these language extensions, one should understand how they are integrated with the computational model of OOP.

I intend to update the curriculum of my lectures to give an account to the emerging declarative programming trends in modern programming languages, and offer a critical analysis of the way how they are simulated.

Chapter 6

Bibliography

[1] E. Albert. Partial evaluation of multi-paradigm declarative languages. *AI Commun.*, 14(4):235–237, 2001.

[2] M. Alpuente, M. Falaschi, and G. Vidal. Partial evaluation of functional logic programs. *ACM Trans. Program. Lang. Syst.*, 20(4):768–844, 1998.

[3] V. Antimirov. Rewriting regular inequalities (extended abstract). In H. Reichel, editor, *Fundamentals of Computation Theory, 10th International Symposium FCT'95*, volume 965 of *LNCS*, pages 116–125. Springer, 1995.

[4] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *TCS*, 155:291–319, 1996.

[5] V. M. Antimirov and P. Mosses. Rewriting extended regular expression. *TCS*, 143(1):51–72, 1995.

[6] S. Antoy. Evaluation strategies for functional logic programming. *JSC*, 40(1):875–903, 2005.

[7] S. Antoy. Programming with narrowing: A tutorial. *JSC*, 45:501–522, 2010.

[8] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.

[9] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19:9–71, 1994.

[10] T. Becker and V. Weispfenning. *Gröbner Bases: A Computational Approach to Commutative Algebra*. Graduate Texts in Mathematics. Springer, 1993.

[11] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric General-purpose Language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 51–63, New York, NY, USA, 2003. ACM.

[12] J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Procs. of COORDINATION '96 Cesena, Italy*, pages 75–88, Berlin, Heidelberg, 1996. Springer.

[13] A. Bockmayr. *Beiträge zur Theorie des logisch-funktionalen Programmierens*. PhD thesis, Universität Karlsruhe, Germany, 1990. In German.

[14] H. Boley. *A Tight, Practical Integration of Relations and Functions*, volume 1712 of *LNAI*. Springer-Verlag, 1999.

[15] P. Borovanski, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *TCS*, 285(2):155–185, 2002.

[16] P. Borovanski, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. *ENTCS*, 15:55 –70, 1998.

[17] B. Buchberger, K. Aigner, C. Dupre, T. Jebelean, F. Kriftner, M. Marin, K. Nakagawa, O. Podişor, E. Tomuţa, Y. Usenko, D. Văsaru, and W. Windsteiger. Theorema: An Integrated System for Computation and Deduction in Natural Style. RISC Report Series 98–25, RISC-Linz, Schloss Hagenberg, Austria, December 1998.

[18] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuţa, and D. Văsaru. A Survey of the Theorema project. In W. Kuechlin, editor, *Procs. of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation)*, pages 384–391, Maui, Hawaii, July 1997. ACM Press.

[19] B. Buchberger and M. Marin. Proving by Simplification. In B. Buchberger, T. Ida, and D. Vasaru, editors, *First International Theorema Workshop*, RISC, Hagenberg, Austria, June 9-10 1997. RISC-Linz Report Series No. 97-20.

[20] B. Buchberger and W. Windsteiger. The Theorema System. https://www.risc.jku.at/research/theorema/software/.

[21] J. Christiansen and S. Fischer. Easycheck – test data for free. In J. Garrigue and M. V. Hermenegildo, editors, *FLOPS 2008*, pages 322–336, Berlin, Heidelberg, 2008. Springer.

[22] H. Cirstea and C. Kirchner. The rewriting calculus - Part I. *Logic Journal of the IGPL*, 9(3):363–399, 2001.

[23] H. Cirstea and C. Kirchner. The rewriting calculus - Part II. *Logic Journal of the IGPL*, 9(3):401–434, 2001.

[24] M. Clavel and J. Meseguer. Reflection and Strategies in Rewriting Logic. *ENTCS*, 4:126 –148, 1996.

[25] J. Coelho and M. Florido. CLP(Flex): constraint logic programming applied to XML processing. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences. Proceedings, Part II*, volume 3291 of *LNCS*, pages 1098–1112, Agia Napa, Cyprus, October 2004. Springer.

[26] J. Coelho and M. Florido. VeriFLog: A constraint logic programming approach to verification of website content. In H. T. Shen, J. L. anf M. Li, J. Ni, and W. Wang, editors, *Procs. of APWeb 2006 International Workshops: XRA, IWSN, MEGA, and ICSE*, volume 3842 of *LNCS*, pages 148–156, Harbin, China, January 2006. ACM.

[27] J. Coelho and M. Florido. XCentric: logic programming for XML processing. In I. Fundulaki and N. Polyzotis, editors, *Procs. of WIDM 2007*, pages 1–8, Lisbon, Portugal, November 2007. ACM.

[28] A. Colmerauer. An introduction to Prolog III. *Commun. ACM*, 33(7):69–90, 1990.

[29] H. Comon. Completion of Rewrite Systems with Membership Constraints Part I: Deduction Rules. *JSC*, 25(4):397–419, 1998.

[30] H. Comon. Completion of Rewrite Systems with Membership Constraints Part II: Constraint Solving. *JSC*, 25(4):421–453, 1998.

[31] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. Release October 2007.

[32] E. Contejean. Solving Linear Diophantine Constraints Incrementally. In *Procs. of the Tenth Intl. Conference on Logic Programming*, pages 532–549, Budapest, Hungary, 1993. MIT Press.

[33] J. H. Conway. *Regular Algebra and Finite Machines*. Mathematics series. Chapman and Hall, 1971.

[34] D. A. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Undergraduate Texts in Mathematics. Springer-Verlag New York, Inc., NJ, USA, 2007.

[35] J. Darlington, Y. Guo, and H. Pull. Introducing Constraint Functional Logic Programming. Technical report, Imperial College, February 1991.

[36] R. del Vado Vírseda. Cooperation of algebraic constraint domains in higher-order functional and logic programming. In *AMAST 2010. Revised Selected Papers*, pages 180–200, Lac-Beauport, QC, Canada, 2010.

[37] B. Dundua, M. Florido, T. Kutsia, and M. Marin. Constraint Logic Programming for Hedges: A Semantic Reconstruction. In M. Codish and E. Sumii, editors, *12th International Symposium in Functional Logic Programming (FLOPS 2014). Proceedings*, volume 8475 of *LNCS*, pages 285–301, Kanazawa, Japan, June 2014. Springer.

[38] B. Dundua, M. Florido, T. Kutsia, and M. Marin. $CLP(\mathtt{H})$: Constraint logic programming for hedges. *TPLP*, 16(2):141–162, 2016.

[39] B. Dundua, T. Kutsia, and M. Marin. Strategies in PρLog. In M. Fernandez, editor, *Procs. of 9th Intl. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2009)*, volume 15 of *ENTCS*, pages 32–43, Brasilia, Brazil, 2009.

[40] A. J. Fernández, M. T. Hortalá-González, F. Sáenz-Pérez, and R. del Vado Vírseda. Constraint functional logic programming over finite domains. *TPLP*, 7(5):537–582, 2007.

[41] F. L. Fraguas. A General Scheme for Constraint Functional Logic Programming. In G. Levi and H. Kirchner, editors, *Algebraic and Logic Programming*, volume 632 of *LNCS*, pages 213–227, Berlin, 1992.

[42] F. L. Fraguas, M. R. Artalejo, and R. del Vado Virseda. A Lazy Narrowing Calculus for Declarative Constraint Programming. In *Procs. of PPDP'04*, pages 43–54, 2004.

[43] F. L. Fraguas, M. R. Artalejo, and R. del Vado Virseda. A New Generic Scheme for Functional Logic Programming with Constraints. *Higher Order and Symbolic Computation*, 20(1/2):73–122, 2007.

[44] S. Frank, P. Hofstedt, and P. R. Mai. A Flexible Meta-solver Framework for Constraint Solver Collaboration. In *Procs. of KI 2003*, pages 520–534, Hamburg, Germany, 2003.

[45] S. Frank, P. Hofstedt, and D. Reckmann. Meta-S - Combining Solver Cooperation and Programming Languages. In *19th Workshop on (Constraint) Logic Programming, Ulm, Germany, February 21-23, 2005*, pages 159–162, 2005.

[46] A. Frisch and L. Cardelli. Greedy regular expression matching. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004*, volume 3142 of *LNCS*, pages 618–629, Turku, Finland, 2004. Springer.

[47] J. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row Publishers, Inc., New York, NY, USA, 1985.

[48] V. Gapeyev and B. C. Pierce. Regular object types. In L. Cardelli, editor, *Procs. of ECOOP'03*, volume 2743 of *LNCS*, pages 151–175. Springer, 2003.

[49] M. Genesereth and R. E. Fikes. Knowledge interchange format version 3.0 reference manual. Technical Report Logic-92-1, Stanford University, Stanford, CA, USA, 1992.

[50] M. L. Ginsberg. The MVL Theorem Proving System. *SIGART Bull.*, 2(3):57–60, June 1991.

[51] J. A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *TCS*, 105(2):217–273, 1992.

[52] M. Hamada and A. Middeldorp. Strong completeness of a lazy conditional narrowing calculus. In *Procs. of 2nd Fuji Intl. Workshop on Functional and Logic Programming*, pages 14–32. World Scientific, 1997.

[53] M. Hamana. Term rewriting with sequences. In *Procs. of the First Int. Theorema Workshop*, 1997. Available as technical report 97-20, RISC, Johannes Kepler University.

[54] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19–20:583–628, 1994.

[55] M. Hanus. A unified computation model for functional and logic programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 80–93, New York, NY, USA, 1997. ACM.

[56] M. Hanus. *Functional Logic Programming: From Theory to Curry*, pages 123–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[57] J. Hendrix and J. Meseguer. Order-sorted unification revisited. In G. Kniesel and J. S. Pinto, editors, *Pre-proceedings of RULE'08*, pages 16–29, 2008.

[58] P. Hofstedt. Better communication for tighter cooperation. In *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, pages 342–358, 2000.

[59] P. Hofstedt. *Cooperation and coordination of constraint solvers*. PhD thesis, Dresden University of Technology, Germany, 2001.

[60] P. Hofstedt and P. Pepper. Integration of declarative and constraint programming. *TPLP*, 7(1-2):93–121, 2007.

[61] A. A. Holzbacher. A software environment for concurrent coordinated programming. In P. Ciancarini and C. Hankin, editors, *Procs. of COORDINATION '96 Cesena, Italy*, pages 249–266, Berlin, Heidelberg, 1996. Springer.

[62] H. Hong. Confluency of Cooperative Constraint Solvers. Technical Report 94-08, RISC-Linz, Castle of Hagenberg, Austria, 1994.

[63] H. Hosoya. *Foundations of XML Processing: The Tree-Automata Approach*. Cambridge University Press, 2010.

[64] H. Hosoya and B. C. Pierce. Regular Expression Pattern Matching for XML. *J. Functional Programming*, 13(6):961–1004, Nov. 2003.

[65] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, Jan. 2005.

[66] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *CADE 1980*, volume 87 of *LNCS*, pages 318–334. Springer, 1980.

[67] H. Huzita. Development of Origami Geometry. In *Procs. of First Intl. Meeting of Origami Science and Technology*, pages 143–158, 1989.

[68] T. Ida and M. Marin. Functional Logic Origami Programming with Open CFLP. In P. Mitic, P. Ramsden, and J. Carne, editors, *Challenging the Boundaries of Symbolic Computation: Procs. of the 5th International Mathematica Symposium (IMS 2003)*, pages 397–404. Imperial College Press, 2003.

[69] T. Ida, M. Marin, and N. Kobayashi. An Open Environment for Cooperative Equational Solving. *Wuhan University Journal of Natural Sciences*, 6(1):169–174, 2001.

[70] T. Ida, M. Marin, and T. Suzuki. Higher-order lazy narrowing calculus: A solver for higher-order equations. In R. Moreno-Diaz, B. Buchberger, and J. L. Freire, editors, *EUROCAST 2001*, volume 2178 of *LNCS*, pages 479–493, 2002.

[71] T. Ida, M. Marin, and T. Suzuki. Reducing Search Space in Solving Higher-Order Equations. In S. Arikawa and A. Shinohara, editors, *Progress in Discovery Science*, volume 2281 of *LNCS*, pages 19–30, 2002.

[72] N. Kobayashi, M. Marin, and T. Ida. Collaborative Constraint Functional Logic Programming System in an Open Environment. *IEICE Transactions on Information and Systems*, E86-D(1):63–70, 2003.

[73] T. Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols*. PhD thesis, Johannes Kepler University, Linz, Austria, 1992.

[74] T. Kutsia. Solving equations involving sequence variables and sequence functions. In B. Buchberger and J. A. Campbell, editors, *Procs. of Joint AISC'2004*, volume 3249 of *LNAI*, pages 157–170. Springer, 2002.

[75] T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Procs. of Joint AISC'2002–Calculemus'2002 Conference*, volume 2385 of *LNAI*, pages 290–304. Springer, 2002.

[76] T. Kutsia. Solving equations with sequence variables and sequence functions. *JSC*, 42(3):352–388, 2007.

[77] T. Kutsia and M. Marin. Can context sequence matching be used for querying XML? In L. Vigneron, editor, *Proceedings of the 19th International Workshop on Unification (UNIF'05)*, pages 77–92, Nara, Japan, 22 Apr. 2005.

[78] T. Kutsia and M. Marin. Matching with regular constraints. In G. Sutcliffe and A. Voronkov, editors, *Procs. of LPAR 2005*, volume 3835 of *LNAI*, pages 215–229, Berlin, Heidelberg, 2005. Springer-Verlag.

[79] T. Kutsia and M. Marin. Matching with Regular Constraints. Technical Report 05-05, RISC-Linz Institute, Austria, 2005.

[80] T. Kutsia and M. Marin. Solving regular constraints for hedges and contexts. In J. Levy, editor, *Procs. of the 20th Intl. Workshop on Unification (UNIF'06)*, pages 89–107, Seattle, USA, 11 Aug. 2006.

[81] T. Kutsia and M. Marin. Order-Sorted Unification with Regular Expression Sorts. In C. Lynch, editor, *Procs. of RTA 2010*, volume 6 of *LIPIcs-Leibniz International Proceedings in Informatics*, pages 193–208, Edinburgh, Scotland, U.K., 2010.

[82] T. Kutsia and M. Marin. Regular expression order-sorted unification and matching. *JSC*, 67:42–67, 2015.

[83] D. Maier. Database desiderata for an XML query language, 1998. Available from: `http://www.w3.org/TandS/QL/QL98/pp/maier.html`.

[84] M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD thesis, Johannes Kepler University, RISC-Linz Institute, Austria, 2000. Available as RISC Report Series 00-28.

[85] M. Marin. Functional Programming with Sequence Variables: The *Sequentica* Package. In J. Levy, M. Kohlhase, J. Niehren, and M. Villaret, editors, *Procs. of the 17th Intl. Workshop on Unification (UNIF'03)*, pages 65–78, Valencia, Spain, 8-9 June 2003. Available as Technical Report DSIC-II/12/03 of Universidad Politecnica de Valencia.

[86] M. Marin and A. Crăciun. Factorizations of regular hedge languages. In S. M. Watt, V. Negru, T. Ida, T. Jebelean, D. Petcu, and D. Zaharie, editors, *Procs. of SYNASC 2009*, pages 397–314, Timişoara, Romania, 2009. IEEE.

[87] M. Marin and A. Crăciun. Type Inference for Regular Expression Pattern Matching. In T. Ida, V. Negru, T. Jebelean, D. Petcu, S. Watt, and D. Zaharie, editors, *Procs. of SYNASC 2010*, pages 366–373, Timişoara, Romania, 2010. IEEE.

[88] M. Marin and M. Drăgan. A Jini service for collaborative constraint solving. In I. Dziţac, T. Maghiar, and C. Popescu, editors, *Procs. of International Conference on Computers and Communications (ICCC 2004)*, pages 235–240, Oradea, Romania, 2004.

[89] M. Marin and T. Ida. A Rule-Based Framework for Automated Reasoning. In S. il Pae and H. Park, editors, *Procs. of ASCM 2005*, pages 28–31, KIAS, Seoul, Korea, 2005.

[90] M. Marin and T. Ida. Rule-based Programming with ρLog. In D. Zaharie, D. Petcu, V. Negru, T. Jebelean, G. Ciobanu, A. Cicortaş, A. Abraham, and M. Paprzycki, editors, *Procs. of SYNASC 2005*, pages 31–38, Timişoara, Romania, 2005. IEEE Computer Society Press.

[91] M. Marin, T. Ida, and W. Schreiner. CFLP: a Mathematica Implementation of a Distributed Constraint Solving System. *Mathematica Journal*, 8(2):287–300, 2001.

[92] M. Marin, T. Ida, and T. Suzuki. Cooperative constraint functional logic programming. In *Procs. of International Symposium on Principles of Software Evolution (PSE 2000)*, pages 214–220. IEEE Computer Society, 2000.

[93] M. Marin and G. Istrate. *Learning Cover Context-Free Grammars from Structural Data*, volume 8687 of *LNCS*, pages 241–258. Springer, Bucharest, Romania, 2014.

[94] M. Marin and G. Istrate. Learning Cover Context-Free Grammars from Structural Data. *Sci. Ann. Comp. Sci.*, 24(2):253–286, 2014.

[95] M. Marin and T. Kutsia. On the implementation of a rule-based programming system and some of its applications. In B. Konev and R. Schmidt, editors, *Proceedings of the 4th International Workshop on the Implementation of Logics (WIL'03)*, pages 55–68, Almaty, Kazakhstan, 2003.

[96] M. Marin and T. Kutsia. A Rule-based Approach to the Implementation of Evaluation Strategies. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Procs. of SYNASC 2004*, pages 227–241, Timişoara, Romania, 2004. Mirton.

[97] M. Marin and T. Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.

[98] M. Marin and T. Kutsia. Matching with membership constraints for hedge and context variables. In M. Marin, editor, *Procs. of the 22th Intl. Workshop on Unification (UNIF'08)*, pages 55–68, Castle of Hagenberg, Austria, 18 July 2008.

[99] M. Marin and T. Kutsia. Linear Systems for Regular Hedge Languages. In J. Grundspenkis, M. Kirikova, Y. Manolopoulos, and L. Novickis, editors, *Procs, of Advances in Databases and Information Systems. Associated Workshops and Doctoral Consortium of the 13th East-European Conference, ADBIS 2009*, volume 5968 of *LNCS*, pages 104–112, Riga, Latvia, 2009. Springer.

[100] M. Marin and T. Kutsia. On the computation of quotients and factors of regular languages. *Frontiers of Computer Science in China*, 4(2):173–184, 2010.

[101] M. Marin and T. Kutsia. Regular hedge language factorization revisited. In S. Yu, editor, *Procs. of DLT 2010*, volume 6224 of *LNCS*, pages 328–339, London, ON, Canada, August 2010. Springer.

[102] M. Marin and A. Middeldorp. New completeness results for lazy conditional narrowing. In E. Moggi and D. S. Warren, editors, *PPDP 2004*, pages 120–131, New York, NY, USA, 2004. ACM.

[103] M. Marin and F. Piroi. Deduction and Presentation in ρLog. *ENTCS*, 93:161–182, 2004.

[104] M. Marin and F. Piroi. Rule-based programming with Mathematica. In *Procs. of Sixth International Mathematica Symposium (IMS 2004)*, pages 1–6, Banff, Alberta, Canada, 2004. Also available as RICAM-Report 2004-03.

[105] M. Marin, T. Suzuki, and T. Ida. Refinements of Lazy Narrowing for Left-Linear Fully-Extended Pattern Rewrite Systems. Technical Report ISE-TR-01-180, Institute of Information Sciences and Electronics, University of Tsukuba, 2001. 31 pages.

[106] M. Marin and D. Tepeneu. Programming with sequence variables: the Sequentica package. In P. Mitic, P. Ramsden, and J. Carne, editors, *Challenging the Boundaries of Symbolic Computation: Proceedings of the 5th International Mathematica Symposium*, pages 17–24. Imperial College Press, 2003.

[107] N. Marti-Oliet, J. Meseguer, and A. Verdejo. A Rewriting Semantics for Maude Strategies. *ENTCS*, 238(3):1–18, 2009.

[108] S. E. Martín and R. del Vado Vírseda. Designing an efficient computation strategy in *CFLP(FD)* using definitional trees. In *W(C)FLP'05*, pages 23–31, Tallinn, Estonia, 2005.

[109] S. E. Martín, A. J. Fernández, M. T. Hortalá-González, M. Rodríguez-Artalejo, F. Sáenz-Pérez, and R. del Vado Vírseda. Cooperation of constraint domains in the *TOY* system. In *PPDP'08*, pages 258–268, Valencia, Spain, 2008.

[110] S. E. Martín, M. T. Hortalá-González, M. Rodríguez-Artalejo, R. del Vado Vírseda, F. Sáenz-Pérez, and A. J. Fernández. On the cooperation of the constraint domain h, r, and f in cflp.

[111] J. Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721 – 781, 2012.

[112] J. Meseguer, J. A. Goguen, and G. Smolka. Order-sorted unification. *JSC*, 8(4):383–413, 1989.

[113] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *ENTCS*, 117:153 –182, 2005.

[114] A. Middeldorp and S. Okui. A deterministic lazy narrowing calculus. *JSC*, 25(6):733–757, 1998.

[115] A. Middeldorp, S. Okui, and T. Ida. Lazy narrowing: Strong completeness and eager variable elimination. *TCS*, 167(1):95–130, 1996.

[116] E. Monfroy. *Solver Collaboration for Constraint Logic Programming*. PhD thesis, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine, 1996.

[117] E. Monfroy. A solver collaboration in BALI. In *Procs. of the 1998 Joint Intl. Conference and Symposium on Logic Programming*, pages 349–350, Manchester, UK, 1998.

[118] E. Monfroy and F. Arbab. *Constraints Solving as the Coordination of Inference Engines*, pages 399–419. Springer, Berlin, Heidelberg, 2001.

[119] M. Murata. Extended path expressions for XML. In *Procs. of PODS 2001*, pages 126–137, Santa Barbara, California, USA, 2001.

[120] M. Murata. Hedge automata: a formal model for XML schemata, 2009. Available from `http://www.xml.gr.jp/relax/hedge_nice.html`.

[121] T. Nipkow. Orthogonal higher-order rewrite systems are confluent. In M. Bezem and J. F. Groote, editors, *Procs of TLCA'93*, volume 664 of *LNCS*, pages 306–317, Berlin, Heidelberg, 1993. Springer.

[122] E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *AAAECC*, 12(1):73–116, 2001.

[123] A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf. *Coordination of Internet Agents: Models, Technologies, and Applications*. Springer Science & Business Media, 2001.

[124] C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming*. Progress in Theoretical Computer Science. Birkhäuser, 1998.

[125] Z. Qian. Linear unification of higher-order patterns. In M. C. Gaudel and J. P. Jouannaud, editors, *Procs. of TAPSOFT'93. 4th International Joint Conference CAAP/FASE*, pages 391–405, Berlin, Heidelberg, 1993. Springer.

[126] A. Rajasekar. Constraint logic programming on strings: Theory and applications. In *Logic Programming, Procs. of ILPS'94*, page 681. MIT Press, 1994.

[127] U. Reddy. Narrowing as the operational semantics of functional languages. In *International Symposium on Logic Programming*, pages 138–151. IEEE Computer Soc. Press, 1985.

[128] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[129] M. Rueher. An architecture for cooperating constraint solvers on reals. In A. Podelski, editor, *Constraint Programming: Basics and Trends: 1994 Châtillon Spring School. Selected Papers*, pages 231–250, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[130] P. A. Sánchez, F. L. Fraguas, and M. R. Artalejo. Functional plus Logic Programming with Built-in and Symbolic Constraints. In *Procs. of PPDP'99*, volume 1702 of *LNCS*, pages 152–169, 1999.

[131] P. A. Sánchez, T. H. González, F. L. Fraguas, and E. U. Hernández. Functional Logic Programming with Real Numbers. In *Multi-Paradigm Logic Programming. Post-Conference Workshop of the JICSLP'96*, pages 47–58, Bonn, 1996.

[132] T. Schaefer. The complexity of satisfiability problems. In R. Lipton, W. Burkhard, W. Savitch, E. Friedman, and A. Aho, editors, *STOC*, pages 216–226. ACM, 1978.

[133] M. Schmidt-Schauss. *Computational Aspects of an Order-sorted Logic with Term Declarations*, volume 395 of *LNCS*. Springer, 1989.

[134] W. Schreiner, W. Danielczyk-Landerl, M. Marin, and W. Stöcher. A generic programming environment for high-performance mathematical libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 256–268, London, UK, UK, 2000. Springer-Verlag.

[135] K. U. Schulz. Makanin's Algorithm for Word Equations - Two Improvements and a Generalization. In *Procs. of the First Intl. Workshop on Word Equations and Related Topics*, IWWERT '90, pages 85–150, London, UK, 1992. Springer-Verlag.

[136] R. Sekar, I. V. Ramakrishnan, and A. Voronkov. Handbook of automated reasoning. chapter Term Indexing, pages 1853–1964. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 2001.

[137] B. Settles. From Theories to Queries: Active Learning in Practice. In *Procs. of Workshop on Active Learning and Experimental Design*, volume 16 of *JMLR*, pages 227–247, 2011.

[138] M. Sulzmann. regexpr-symbolic: Regular expressions via symbolic manipulation, 2009. `http://hackage.haskell.org/package/regexpr-symbolic`.

[139] J. Thatcher. There is a lot more to finite automata theory than you would have thought. Technical Report RC-2852 (#13407), IBM Thomas J. Watson Research Center, Yorktown, New York, 1970.

[140] J. Thatcher and J. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[141] E. van der Vlist. *RELAX NG*. O'Reilly, 2003.

[142] V. van Oostrom, August 2000. Personal communication.

[143] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. System Description of Stratego 0.5. In A. Middeldorp, editor, *Procs. of RTA 2001*, volume 2051 of *LNCS*, pages 357–361, Berlin, Heidelberg, May 2001. Springer.

[144] J. Waldo. *The Jini (TM) specifications*. Addison-Wesley, second edition, 2000.

[145] C. Walinsky. CLP(Σ^*): constraint logic programming with regular sets. In G. Levi and M. Martelli, editors, *Logic Programming, Proceedings of the Sixth International Conference*, pages 181–196, Lisbon, Portugal, June 1989. MIT Press.

[146] C. Walther. Many-sorted unification. *J. ACM*, 35(1):1–17, 1988.

[147] C. Weidenbach. Unification in sort theories and its applications. *Annals of Mathematics and Artificial Intelligence*, 18(2):261–293, 1996.

[148] S. Wolfram. *The Mathematica Book*. Wolfram Media, Inc., third edition, 2003.