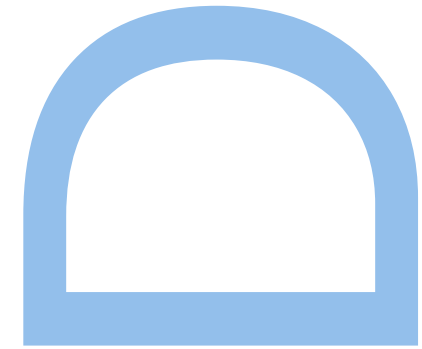
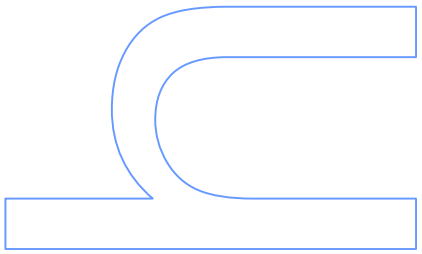
Programming with Sequence and Context Variables: Foundations and Applications



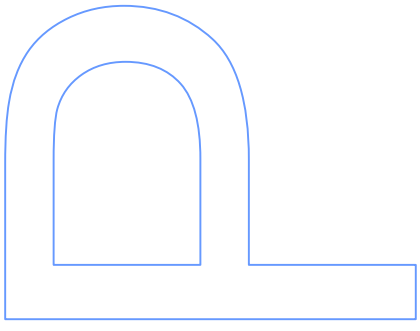
Besik Dundua



Programa Doutoral em Ciência de Computadores
Departamento de Ciência de Computadores
2014

Orientador
António Mário da Silva Marcos Florido, Professor Associado, Faculdade de Ciências da Universidade do Porto

Coorientador
Temur Kutsia, Professor, Johannes Kepler University Linz



Resumo

Esta tese aborda a integração de variáveis de sequência e de contexto em linguagens declarativas. As variáveis de sequência podem ser instanciadas por qualquer sequência finita de termos. As variáveis de contexto são variáveis de segunda ordem para contextos: termos com uma única ocorrência de uma variável ligada. Na primeira parte da tese, apresentamos uma instância do esquema de programação em lógica por restrições (CLP) com variáveis de sequência e de contexto bem como restrições expressas por expressões regulares. Em seguida apresentamos a linguagem baseada em reescrita PρLog, que extende o Prolog com regras de reescrita condicional sobre termos com variáveis de sequência e de contexto. Além disso PρLog tem uma linguagem de restrições regulares para restringir os valores de variáveis de sequência e de contexto usando expressões regulares e árvores regulares, respectivamente. Ilustramos o poder expressivo da linguagem com exemplos de aplicação ao processamento de dados estruturados (XML), raciocínio sobre a Web e especificação de estratégias de reescrita. Finalmente, integramos variáveis de sequência num calculo que modela programas funcionais: o "pattern calculus", definindo um novo cálculo e demonstrando propriedades importantes como a confluência.

Abstract

In this thesis we study integration of sequence and context variables into declarative programming. These variables are useful in various areas of computer science. Sequence variables can be instantiated by finite sequences of terms. Context variables are second order variables that stand for contexts: terms with a single occurrence of a single bound variable (called the hole). In the first part of the thesis, we develop a constraint solver for equational and membership constraints over sequences and contexts, incorporate it in the constraint logic programming schema, and explore the semantics of the obtained language. Next, we propose a rule-based system PρLog, which extends Prolog with strategic conditional rewriting rules and supports programming with sequence and context variables. In addition, PρLog permits regular constraints to restrict possible values of sequence and context variables by regular sequence expressions and regular tree (context) expressions, respectively. We illustrate expressive power of the system using examples and show its application in XML transformation, Web reasoning and for specifying rewriting strategies. Finally, we integrate sequence variables in pattern calculus (a formalism for functional programming) and propose a generic confluence proof, where the way pattern abstractions are applied in a non-deterministic calculus is axiomatized.

Acknowledgements

First and foremost, particular gratitude goes to my supervisors Mário Florido and Temur Kutsia. I would like to thank them for their valuable guidance and encouragement during my PhD studies. I can not speak about my supervisors separately, because both of them were very supportive and helpful, and without them this doctoral thesis would not exist.

Many thanks to my coauthors Sandra Alves, Jorge Coelho, Mircea Marin and Levan Uridia for their collaborations and interesting discussions. My office partners Marco Almeida, Cláudio Amaral, Margarida Carvalho, Bruno Oliveira, David Pereira and Andre Souto were always helpful and friendly to me and I wish to thank them.

I gratefully acknowledge António Porto, president of the department of computer science at university of Porto for providing necessary infrastructure and resources to accomplish my research work. I am very thankful to Sabine Broda for her encouragement and personal attention. My special thanks go to secretary Alexandra Ferreira for helping me to solve all kinds of bureaucratic problems in Portugal.

I also would like to thank my parents, wife, children for their patience and understanding.

Financial support is acknowledged from the LIACC through Programa de Financiamento Plurianual of the Fundação para a Ciência e Tecnologia (FCT) and from the FCT fellowship (ref. SFRH/BD/62058/2009)

Contents

1. Introduction

In this thesis, we address the problems of incorporating sequence and context variables into declarative programming. Sequence variables stand for finite sequences of terms. Context variables denote contexts that can be seen as unary functions with a single occurrence of the bound variable. They enhance expressive capabilities of a language, help to write short, neat, understandable code, hide away many tedious data processing details from the programmer.

Such expressive extensions naturally bring challenges: What are the calculi behind? How to define declarative and operational semantics? How to solve constraints, unification and matching problems that arise during computation? Does the evaluation order matter? Under which conditions do we have confluence and strong normalization? These and other, related problems are addressed in this thesis for calculi for constraint logic programming, rule-based programming, and functional programming. We first develop equation and constraint solving methods (for terms with sequence and context variables), since these techniques are in the heart of computational mechanisms of various styles of declarative programming. Then, we address an extension of constraint logic programming with sequence and context variables, its semantics and special cases. Next, we investigate a similar extension of rule-based programing. Finally, we study pattern calculi with sequence variables, which can be seen as a foundation of an extension of functional programming with sequence variables. Note that in this part we go beyond context variables and consider full higher-order variables. (Context variables are a special kind of second-order variables.)

Our interest in studying theories that involve sequence and context variables is stipulated by their interesting applications. In recent years, usefulness of sequence variables has been illustrated in practical applications related to XML [CF04, KM05a, CF06, CF07b, CFK07, CFK09], schema transformation operations [RF97, CD97], knowledge representation [Gen98, HM05, HM01, Men11], automated reasoning [Pau90, Gin91, Kut03, HV06, BCJ$^+$06], rewriting [Ham97, WB01, JR08], functional, functional logic, and rule-based programming

[MT03, Bol99, MK03, MK06], Common Logic [Com07], just to name a few. There are systems for programming with sequence variables. Probably the most prominent one is Mathematica [Wol03], with a powerful rule-based programming language that uses (essentially first order, equational) matching with sequence variables [Buc96]. Variadic symbols and sequence variables bring a great deal of expressiveness in this language, permitting writing a short, concise, readable code. For instance, one can implement a sorting algorithm (increasing order) in Mathematica just in two lines:

```
sort({X___,x_,Y___,y_,Y___}):= sort({X,y,Y,x,Y})/;x > y
sort({X___}):={X}
```

Identifiers with triple underscores such as `X___` are Mathematica notation for sequence variables that can be replaced also with the empty sequence, if necessary. Those with the single underscore `_` are individual variables. (In Mathematica, double underscores are reserved for variables that can be instantiated with nonempty sequences.) The first rule says that to sort a list that contains in some places two elements `x` and `y` that to not obey the given order, one should swap them and sort the obtained list. The notation `/; x>y` stands for the condition on applicability of the rule. It is applied if `x>y` holds. If the first rule is not applicable (that means, the list is sorted), then the second rule returns the list.

Context variables are placeholders for contexts, which are functional expressions whose applicative behavior is to replace a special constant (called the hole) with the expression given as argument. Context variables in programming give flexibility of data traversal in arbitrary depth. They have applications in compositional semantics of natural language [NPR97, Kol98, NV02, LNV05, NV05], program analysis [GT07], etc. There is a extension of Haskell programming language that supports programming with context variables [Moh96].

Several formalisms combining sequences and contexts have been proposed recently, motivated by various applications. Forest algebras [BW08] have been developed as an algebraic framework for classifying regular languages of finite labeled trees. Tree algebras [SW10] have been used for the characterization of XML database transformations. ρLog [MK06] was introduced as a rule-based transformation calculus with sequence and context variables.

In what follows, we give a brief summary of our contributions in integrating sequence and context variables into constraint logic programming, rule-based programming and functional programming.

Constraint Logic Programming

Constraint logic programming, CLP, [JM94, JMMS98] is one of the most successful areas of logic programming, combining logical deduction with constraint solving. In [JL87], Jaffar and Lassez introduced a CLP schema, denoted CLP(X), parametrized by the constraint domain X. Since then, various instances of this schema have been introduced and studied, introducing a new constraint domain, designing an efficient satisfiability and solving procedure for it, and putting it in the general framework. As examples, we could mention CLP($\mathbb{R}$) [JMSY92], CLP(FD) [CD96], and RISC-CLP(Real) [Hon91].

The domain we study in this thesis is combination of sequences and contexts. To the best of our knowledge, there is practically no work on constraint solving in this combined domain, although sequence unification and disunification [CD97, Kut02, Kut07, CFK07], context unification and constraint solving [Lev96, SS02, SSS02, Vil04, LSV06, Com98], and matching in a combined theory [KM05b] have been studied. The problem is difficult: Both sequence and context unification generalize word unification [Mak77] and have the infinitary unification type, i.e., some unification equations have infinite minimal complete set of solutions.

We extend the constraint logic programming schema to work over sequences and contexts. Constraints are existential formulas, constructed over equations and regular (sequence or context) language membership atoms. The obtained language is called CLP($\mathcal{SC}$). It generalizes CLP(Flex) [CF04], where constraints are conjunctions of equations between unranked terms (they are called flexible arity terms in [CF04]) with sequence variables. Moreover, as both sequences and contexts generalize words, this extension can be seen also as a generalization of CLP($\mathcal{S}$) [Raj94] and of string processing features of Prolog III [Col90]. We describe the semantics of CLP($\mathcal{SC}$) and investigate restrictions on programs leading to constraints in a special form for which the constraint solving algorithm is complete.

Contributions. The main contributions in constraint solving and constraint logic programming can be summarized as follows:

- Developing a constraint solving algorithm.
- Proving that the algorithm is sound and terminating, and brings constraints to a partially solved form.

- Identifying fragments of constraints that can be completely solved by the algorithm.
- Studying semantics of CLP($\mathcal{SC}$) programs.
- Investigating restrictions on programs leading to constraints in a special form for which the constraint solving algorithm is complete.

Some of these results have been reported in [DFKM14].

Rule-Based Programming

Rule-based programming is a paradigm that advocates usage of rules to program. It is experiencing a period of growing interest: Sophisticated calculi are being introduced, new systems are being implemented, interesting applications are being developed. In such programs, rules are applied, usually exhaustively, to transform an object replacing a sub-object in it with another one. The description of this sub-object is separated from the calculation of its replacement. Applicability of the rules can be restricted by conditions. Strategies provide additional control on transformations.

Some of the formalisms important for rule-based programming are term rewriting [BN98], rewriting logic [MOM02], rewriting calculus [CK01]. Specific systems that support rule-based style of programming include symbolic computation system Mathematica [Wol03], languages like CHR [Frü98], ELAN [BKK$^+$98], Maude [CDE$^+$02], ASF+SDF [vdBvDH$^+$01], Stratego [BKVV08]. We listed here some of the probably most mature ones: It is hard to give an exhaustive overview of all formalisms and systems in a brief introduction.

The ρLog calculus [MK06] has been influenced by the rewriting calculus, but there are some significant differences: ρLog adopts logic programming semantics (clauses are first class concepts, rules/strategies are expressed as clauses), uses top-position matching, and employs four different kinds of variables. PρLog (pronounced Pē-rō-log) [DK] is a system based on ρLog calculus and extends logic programming with strategic conditional transformation rules. PρLog is written in SWI-Prolog [WSTL10] and is available for downloading from `http://www.risc.jku.at/people/tkutsia/software.html`.

Contributions. The contributions in rule-based programming with sequence and context variables can be briefly summarized as follows:

- Description of PρLog: a system that extends Prolog with strategic conditional transformation rules, involving sequence and context variables and membership literals.
- Presenting case studies of implementing PρLog programs for XML processing, Web reasoning and rewriting strategies.

Some of these results have been published in [DKM09, CDFK10, Dun10].

Functional Programming

Lambda calculus, introduced by Alonzo Church in [Chu32], is the basis of functional programming languages. Functional languages heavily rely on pattern matching. To reflect it in lambda calculus, Peyton Jones in [PW87] proposed to generalize variable abstraction by pattern abstraction, permitting expressions of the form $\lambda P.T$, where the pattern P is an arbitrary term, not necessarily a variable. With this extension, the expression such as $\lambda(xy).(xy)$ is a well-formed term. The ordinary beta reduction is replaced by the rule $(\lambda P.M)N \rightarrow_\beta M\sigma$, where σ is the substitution that matches P with N.

Following this proposal, van Oostrom [vO90] introduced lambda calculus with patterns (later revisited in [KvOdV08]) and proved that the confluence property holds, provided that the patterns meet certain requirements. (Confluence means that terms can be reduced in more than one way, but the reduction eventually yields the same result.) After that, various calculi that incorporate pattern matching in lambda calculus have been proposed. Some representative ones are lambda-calculus with constructors [AMR06], rewriting calculus [CK01], pure pattern calculus [JK06, JK09], basic pattern matching calculus [Kah03], just to name a few. It turns out that the introduction of patterns, in general, leads to loss of confluence when no restrictions are imposed. These restrictions can be put on the syntactic form of patterns or on the reduction relation (with the help of a strategy).

Cirstea and Faure in [CF07a] studied confluence of pattern calculus parametrized by unitary matching function. Extending confluence results from unitary to non-unitary matching is very useful in practice but, as it has been underlined in [CF07a], is syntactically and semantically non-trivial and opens new challenges. This is the problem we approach in Chapter 6 of this thesis.

Contributions. Our contributions in pattern calculi can be briefly summarized as follows:

- Presenting a pattern calculus, where matching function can be finitary. The calculus permits sequence variables: sequence matching is finitary.
- Establishing sufficient properties the matching function should satisfy in order the pattern calculus to be confluent.
- Providing a generic confluence proof, from which one can obtain confluence proofs for concrete instantiations of underline matching.
- Describing concrete instances of the matching function that satisfy the sufficient conditions for confluence.

Some of the results of this chapter have been presented in [ADFK13, ADFK14].

Thesis Outline

The structure of the thesis follows the presentation of the results described above. After introducing the language in Chapter 2, we present the constraint solving algorithm in Chapter 3, and discuss constraint logic programming over sequences and contexts in Chapter 4. Chapter 5 is dedicated to rule-based programming over sequences and contexts and the description of the PρLog system. Chapter 6 is about the pattern calculus. The final chapter is conclusion.

2. Term Language

In this chapter we present the main syntactic categories of the term language. All the basic notions used throughout the thesis are defined here: first order terms, contexts, sequences, lambda terms, substitutions and related notions and operations.

2.1 Terms

We consider the alphabet $\mathcal{A}$ consisting of the following pairwise disjoint sets of symbols:

- $\mathcal{V}_{\mathsf{T}}$: term variables, denoted by $x, y, z, \ldots$,
- $\mathcal{V}_{\mathsf{S}}$: sequence variables, denoted by $\overline{x}, \overline{y}, \overline{z}, \ldots$,
- $\mathcal{F}$: function symbols, denoted by $f, g, a, b, \ldots$,
- Binary function symbol, denoted by $\wr$,
- Auxiliary symbols: parentheses and the comma.

We assume that $\mathcal{F}$ is finite. The letter $\mathcal{V}$ is used to denote the set of variables $\mathcal{V} := \mathcal{V}_{\mathsf{T}} \cup \mathcal{V}_{\mathsf{S}}$. *Terms* over the alphabet $\mathcal{A}$ are defined inductively as follows:

$$M, N ::= x \mid \ f \mid (M\, N) \mid (M\, \overline{x}) \mid (\lambda_{\chi} M.\, N) \mid (M \wr N)$$

where $(M\, N)$ stands for *term to term application* and $(M\, \overline{x})$ for *term to a sequence variable application*. In the *abstraction* $(\lambda_{\chi} M.N)$ we call the term M a *pattern*. The finite set χ of variables is supposed to specify which variables are bound by the abstraction. They are called *matchable* variables (because, as we will see later, they are available for matching in reductions). The set of terms over $\mathcal{F} \cup \{\wr\}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F} \cup \{\wr\}, \mathcal{V})$.

For example, terms are $((g\,a)\,b)$, $(((f\,x)\,a)\,\overline{x})$, $((\lambda_{\{y,z,\overline{y}\}}((f\,\overline{x})\,y).\,(x\,\overline{y}))\,y)$, $(\lambda_{\{x\}}\,x.\,(((g\,y)\,x)\,\overline{x}))$, $(\lambda_{\{x\}}x.\,(f\,(x\,\overline{x})))$ and $(\lambda_{\{x,\overline{x}\}}(((f\,x)\wr(g\,x))\,\overline{x}).\,((f\,x)\,((g\,y)\wr(g\,\overline{x})\wr(g\,y))))$, while $(\lambda_{\{\overline{x}\}}\,\overline{x}.\,(g\,\overline{x}))$ and $(\lambda_{\{x\}}\,x.\,(\overline{x}\,x))$ are not.

The letters M, N, Q, P, W are used to denote terms. The letters $\overline{M}$, $\overline{N}$, $\overline{Q}$, $\overline{P}$, $\overline{W}$ denote sequence variables or terms. Application associates to the left, therefore we can write $(M\,\overline{Q}_1\cdots\overline{Q}_n)$ for $(\cdots(M\,\overline{Q_1})\cdots\,\overline{Q_n})$. Note that we do not associate a fixed arity to symbols (we can say that our alphabet is *unranked*), hence, in this notation, the same symbol can be followed with different number of terms in different places.

Application also binds stronger than $\wr$. When there is no ambiguity, the outermost parentheses are omitted. For example, the terms above are written as follows:

- $((g\,a)\,b)$ as $g\,a\,b$,
- $(((f\,x)\,a)\,\overline{x})$ as $f\,x\,a\,\overline{x}$,
- $((\lambda_{\{y,z,\overline{y}\}}((f\,\overline{x})\,y).\,(x\,\overline{y}))\,y)$ as $(\lambda_{\{y,z,\overline{y}\}}f\,\overline{x}\,y.\,(x\,\overline{y}))\,y$,
- $(\lambda_{\{x\}}\,x.\,(((g\,y)\,x)\,\overline{x}))$ as $\lambda_{\{x\}}\,x.\,(g\,y\,x\,\overline{x})$,
- $(\lambda_{\{x,\overline{x}\}}(((f\,x)\wr(g\,x))\,\overline{x}).\,((f\,x)\,((g\,y)\wr(g\,\overline{x})\wr(g\,y))))$ as $\lambda_{\{x,\overline{x}\}}(f\,x\wr g\,x)\,\overline{x}.\,(f\,x\,(g\,y\wr g\,\overline{x}\wr g\,y))$.

The *sets of free and bound variables* of a term M, denoted $\mathtt{fv}(M)$ and $\mathtt{bv}(M)$ respectively, are defined inductively as follows:

$\mathtt{fv}(x) = \{x\}$	$\mathtt{bv}(x) = \emptyset$
$\mathtt{fv}(f) = \emptyset$	$\mathtt{bv}(f) = \emptyset$
$\mathtt{fv}(N\,Q) = \mathtt{fv}(N) \cup \mathtt{fv}(Q)$	$\mathtt{bv}(N\,Q) = \mathtt{bv}(N) \cup \mathtt{bv}(Q)$
$\mathtt{fv}(N\,\overline{x}) = \mathtt{fv}(N) \cup \{\overline{x}\}$	$\mathtt{bv}(N\,\overline{x}) = \mathtt{bv}(N)$
$\mathtt{fv}(\lambda_\chi P.N) = (\mathtt{fv}(P) \cup \mathtt{fv}(N)) \setminus \chi$	$\mathtt{bv}(\lambda_\chi P.N) = \mathtt{bv}(P) \cup \mathtt{bv}(N) \cup \chi$
$\mathtt{fv}(N \wr Q) = \mathtt{fv}(N) \cup \mathtt{fv}(Q)$	$\mathtt{bv}(N \wr Q) = \mathtt{bv}(N) \cup \mathtt{bv}(Q)$

For example, the term $g\,a\,b$ contains neither free nor bound variables, and the term $f\,x\,a\,\overline{x}$ contains x and $\overline{x}$ as free variables. In the term $(\lambda_{\{y,z,\overline{y}\}}f\,\overline{x}\,y.\,(x\,\overline{y}))\,y$, the variables $x, y, \overline{x}$ occur as free variables and $y, z, \overline{y}$ occur as bound variables. Hence, y has both free and bound occurrences. The term $\lambda_{\{x\}}\,x.\,(g\,y\,x\,\overline{x})$ contains x as a bound variable and y and $\overline{x}$ as free variables. In the term $\lambda_{\{x\}}x.\,(f\,(x\,\overline{x}))$, the variable $\overline{x}$ occurs freely, while x is a bound.

Finally, in the term $\lambda_{\{x,\overline{x}\}}(f\,x \wr g\,x)\,\overline{x}.\,(f\,x\,(g\,y \wr g\,\overline{x} \wr g\,y))$ the variable y occurs freely, while x and $\overline{x}$ are bound. A term M is called *linear* with respect to the set of variables V if each variable from V occurs in M freely at most once. We say a term is *closed* if there are no free variables. For example, $\lambda_{\{y,\overline{x},\overline{y}\}} f\,\overline{x}\,y.\,(y\,\overline{y})$ and $\lambda_{\{x\}} x.\,(f\,x)$ are closed terms, while $(\lambda_{\{y,z,\overline{y}\}} f\,\overline{x}\,y.\,(x\,\overline{y}))\,y$ and $\lambda_{\{x\}} x.\,(f\,(x\,\overline{x}))$ are not.

Unlike the λ-calculus, we abstract not only on variables but on terms. The abstraction $\lambda_\chi P.N$ binds those variables that are explicitly mentioned in χ. Note that in [CF07a], χ is supposed to be a subset of $\mathtt{fv}(P)$. We do not require $\chi \subseteq \mathtt{fv}(P)$. Later, when we define the notion of reduction, we will see that some reductions under the restriction $\chi \subseteq \mathtt{fv}(P)$ would transform terms to non-terms.

We adopt Barendregt's variable name convention [Bar84], i.e., free and bound variables have different names. This can be fulfilled by renaming bound variables. For example, a renamed copy of the term $(\lambda_{\{y,z,\overline{y}\}} f\,\overline{x}\,y.\,(x\,\overline{y})\,y)$ will be written as $(\lambda_{\{y',z,\overline{y}\}} f\,\overline{x}\,y'.\,(x\,\overline{y}))\,y$. We identify terms modulo α-equivalence, therefore expressions that differ only in the names of bound variables are identified. For example, a term $\lambda_{\{z\}}\,z.\,(g\,y\,z\,\overline{x})$ is α-equivalent to the term $\lambda_{\{x\}}\,x.\,(g\,y\,x\,\overline{x})$.

For a given set S, $\lceil s_1, \ldots, s_n \rceil$ denotes a finite *sequence* of elements $s_1, \ldots, s_n$ of S. In particular, the empty sequence is written as $\lceil\,\rceil$. We do not distinguish between a singleton sequence and its element and write s instead of $\lceil s \rceil$. Concatenation of sequences $\lceil s_1, \ldots, s_n \rceil$ and $\lceil s'_1, \ldots, s'_m \rceil$, written $\lceil s_1, \ldots, s_n \rceil \bowtie \lceil s'_1, \ldots, s'_m \rceil$, is the sequence $\lceil s_1, \ldots, s_n, s'_1, \ldots, s'_m \rceil$. Obviously, $\bowtie$ is associative and the empty sequence plays the role of its unit element.

We use $\tilde{S}$ to denote finite (possible empty) sequences of terms and sequence variables. For example, $\lceil g\,a\,b,\ f\,x\,a\,\overline{x},\ \overline{y},\ \lambda_{\{x,\overline{x}\}}(f\,x \wr g\,x)\,\overline{x}.\,(f\,x\,(g\,y \wr g\,\overline{x} \wr g\,y)) \rceil$ is a sequence. The set of sequences over $\mathcal{F} \cup \{\wr\}$ and $\mathcal{V}$ is denoted by $\mathcal{H}(\mathcal{F} \cup \{\wr\}, \mathcal{V})$.

The notions of free and bound variables are extended to sequences as follows: $\mathtt{fv}(\overline{x}) = \{\overline{x}\}$, $\mathtt{fv}(\lceil \overline{M}_1, \ldots, \overline{M}_n \rceil) = \cup_{i=1}^n \mathtt{fv}(\overline{M}_i)$, $\mathtt{bv}(\overline{x}) = \emptyset$, and $\mathtt{bv}(\lceil \overline{M}_1, \ldots, \overline{M}_n \rceil) = \cup_{i=1}^n \mathtt{bv}(\overline{M}_i)$. We generalize Barendregt's variable name convention for sequences, i.e., free and bound variables have different names in a sequence.

2.2 Substitutions

A *substitution* is a mapping from term variables to terms, and from sequence variables to sequences, such that all but finitely many variables are mapped to themselves. We use σ,θ and ρ to denote substitutions.

A substitution σ is represented as a finite set of pairs $\{v_1 \mapsto \sigma(v_1), \ldots, v_n \mapsto \sigma(v_n)\}$ where the v's are all those (term or sequence) variables which are not mapped to themselves by σ. The sets $Dom(\sigma) = \{v_1, \ldots, v_n\}$ and $Ran(\sigma) = \{\sigma(v_1), \ldots, \sigma(v_n)\}$ are called the *domain* and the *range* of σ, respectively. Obviously, $\sigma(v) = v$ iff $v \notin Dom(\sigma)$.

The set of variables of σ, denoted $var(\sigma)$, is defined as $var(\sigma) := Dom(\sigma) \cup \mathtt{fv}(Ran(\sigma))$.

The *application* of a substitution σ to a term M, written as $M\sigma$, replaces each free occurrence of a variable v in M with $\sigma(v)$. It is defined inductively:

$$
\begin{array}{ll}
x\sigma = \sigma(x), \text{ if } x \in Dom(\sigma). & (M\,\overline{x})\sigma = M\sigma\,\overline{N}_1 \ldots \overline{N}_n, \\
x\sigma = x, \text{ if } x \notin Dom(\sigma). & \qquad \text{if } \sigma(\overline{x}) = \lceil \overline{N}_1, \ldots, \overline{N}_n \rceil,\ n > 0. \\
f\sigma = f. & (M\,\overline{x})\sigma = M\sigma \text{ if } \sigma(\overline{x}) = \lceil\,\rceil. \\
(M\,N)\sigma = M\sigma\,N\sigma. & (M\,\overline{y})\sigma = M\sigma\,\overline{y} \text{ if } \overline{y} \notin Dom(\sigma). \\
(M \wr N)\sigma = M\sigma \wr N\sigma. & (\lambda_\chi P.N)\sigma = \lambda_\chi P\sigma.N\sigma.
\end{array}
$$

In the abstraction, it is assumed that $var(\sigma) \cap \mathtt{bv}(\lambda_\chi P.N) = \emptyset$. This can be achieved by properly renaming the bound variables. Hence, the equality here is α-equivalence. Note that the rule for substituting a sequence variable $\overline{x}$ by a sequence $\lceil \overline{N}_1, \ldots, \overline{N}_n \rceil$ in a term $M\overline{x}$ results into $(\cdots(M\sigma\,\overline{N}_1)\cdots\overline{N}_n)$, written as $M\sigma\,\overline{N}_1 \ldots \overline{N}_n$. When $n = 0$, then $\sigma(\overline{x}) = \lceil\,\rceil$ and $(M\,\overline{x})\sigma = M\sigma$.

The result $M\sigma$ of applying a substitution σ to a term M is called an *instance* of M. Application of a substitution σ to a sequence is defined as $\lceil\,\rceil\sigma = \lceil\,\rceil$ and $\lceil \overline{M}_1, \ldots, \overline{M}_n \rceil = \overline{M}_1\sigma \bowtie \cdots \bowtie \overline{M}_n\sigma$ for $n > 0$.

Example 2.1. *Let* $M = (\lambda_{\{y',z,\overline{y}\}} f\,\overline{x}\,y'.(x\,\overline{y}))\,y$ *and* $\sigma = \{x \mapsto \lambda_{\{x\}}\,x.(g\,y\,x\,\overline{x}),\ \overline{x} \mapsto \lceil g\,a\,b, \overline{z} \rceil,\ y \mapsto \lambda_{\{x\}}x.(f\,(x\,\overline{x})),\ \overline{y} \mapsto \lceil \lambda_{\{x\}}\,x.(g\,y\,x\,\overline{x}),\ f\,x\,a\,\overline{x} \rceil\}$. *Then* $M\sigma$ *is*

$$M\sigma = \Big(\lambda_{\{y',z,\overline{y}'\}} f\,(g\,a\,b)\,\overline{z}\,y'.(\lambda_{\{x\}}\,x.(g\,y\,x\,\overline{x}))\,\overline{y}'\Big)\,\lambda_{\{x\}}x.(f\,(x\,\overline{x}))$$

The *composition* of substitutions is defined in the standard way, as the composition of two mappings. We use juxtaposition to denote it, writing $\sigma\vartheta$ for the composition of σ and ϑ. For all M, we have $M\sigma\vartheta = (M\sigma)\vartheta$.

We say that a substitution σ is *more general* then θ, denoted $\sigma \leq \theta$, if there exist a substitution ρ such that $\sigma\rho = \theta$. The relation $\leq$ is also called the *subsumption ordering*. The *restriction* of a substitution σ to a set of variables $V \subset \mathcal{V}$, denoted $\sigma|_V$, is defined as a substitution with $\sigma|_V(v) = \sigma(v)$ if $v \in V$, and $\sigma|_V(v) = v$ otherwise.

2.3 Simple Terms, Contexts, and Simple Substitutions

In this section we introduce restrictions on terms from $\mathcal{T}(\mathcal{F} \cup \{\imath\}, \mathcal{V})$ to obtain simple terms and contexts.

We distinguish pairwise disjoint subsets of term variables in the syntactic level:

- $\mathcal{V}_{\mathsf{I}}$: individual variables, denoted by $X, Y, Z, \ldots$,
- $\mathcal{V}_{\mathsf{F}}$: function variables, denoted by $X_{\mathsf{f}}, Y_{\mathsf{f}}, Z_{\mathsf{f}}, \ldots$,
- $\mathcal{V}_{\mathsf{C}}$: context variables, denoted by $X_{\mathsf{c}}, Y_{\mathsf{c}}, Z_{\mathsf{c}}, \ldots$.

The reason why we make a distinction between these variables is the form of terms they can be replaced by. We will elaborate on details a bit later, when we will be discussing simple substitutions. Note that $\mathcal{V}_{\mathsf{I}} \cup \mathcal{V}_{\mathsf{F}} \cup \mathcal{V}_{\mathsf{C}} \subset \mathcal{V}_{\mathsf{T}}$.

Simple terms and contexts over the set of variables $\mathcal{V}_{\mathsf{ISFC}} = \mathcal{V}_{\mathsf{I}} \cup \mathcal{V}_{\mathsf{S}} \cup \mathcal{V}_{\mathsf{F}} \cup \mathcal{V}_{\mathsf{C}} \subset \mathcal{V}$ and set of function symbols $\mathcal{F}$ is defined as follows:

Simple term:	$t ::= X \mid X_{\mathsf{c}}\, t \mid T,$ where $T ::= f \mid X_{\mathsf{f}} \mid T\,\overline{t}.$
Simple term or sequence variable:	$\overline{t} ::= t \mid \overline{x}.$
Context:	$C ::= \lambda_{\{X\}} X.t,$ where X occurs only once in t.

The letters t, r are used to denote simple terms, $\overline{t}$ and $\overline{r}$ denote sequence variables or simple terms, and C and D stand for contexts. The set of simple terms is denoted by $\mathcal{T}^s(\mathcal{F}, \mathcal{V}_{\mathsf{ISFC}})$ and the set of contexts is denoted by $\mathcal{C}(\mathcal{F}, \mathcal{V}_{\mathsf{ISFC}})$.

Note that from those terms we have seen earlier, e.g., $g\,a\,b$, $f\,x\,a\,\overline{x}$, $(\lambda_{\{y',z,\overline{y}\}} f\,\overline{x}\,y'.\,(x\,\overline{y}))\,y$, $\lambda_{\{x\}}\,x.\,(g\,y\,x\,\overline{x})$, $\lambda_{\{x\}}x.\,(f\,(x\,\overline{x}))$ and $\lambda_{\{x,\overline{x}\}}(f\,x\wr g\,x)\,\overline{x}.\,(f\,x\,(g\,y\wr g\,\overline{x}\wr g\,y))$, only $g\,a\,b$ and $f\,x\,a\,\overline{x}$ are simple terms, provided that $x \in \mathcal{V}_{\mathsf{I}}$. The term$\lambda_{\{x\}}\,x.\,(g\,y\,x\,\overline{x})$ is a context, provided that $x, y \in \mathcal{V}_{\mathsf{I}}$.

The term $\lambda_{\{x\}}x.\,(f\,(x\,\overline{x}))$ is not a context because of the following: Either $x \notin \mathcal{V}_{\mathsf{F}}$ and $(x\,\overline{x})$ is not a simple term, or $x \in \mathcal{V}_{\mathsf{F}}$ and $\lambda_{\{x\}}x.\,(f\,(x\,\overline{x}))$ does not fall in the definition of contexts.

For notational convenience, contexts are written in a lambda free form, replacing a bound variable with the meta symbol $\circ$, called the *hole*. For instance, the context $\lambda_{\{X\}}\,X.\,g\,Y\,X\,\overline{x}$ will be written as $g\,Y\,\circ\,\overline{x}$. We write simple terms and contexts in a *decurried* form. For instance, $g\,a\,b$ and $f\,X\,a\,\overline{x}$ will be written as $g(a, b)$ and $f(X, a, \overline{x})$, respectively. The context $g\,Y\,\circ\,\overline{x}$ will be written as $g(Y, \circ, \overline{x})$. A context C may be applied to a simple term r (resp., context C'), written $C[r]$ (resp., $C[C']$), and the result is the term (resp., context) obtained from C by replacing the $\circ$ with r (resp., with C'). For example, an application of the context $g(Y, \circ, \overline{x})$ to the term $f(X, a, \overline{x})$, written as $g(Y, \circ, \overline{x})[f(X, a, \overline{x})]$, is a term $g(Y, f(X, a, \overline{x}), \overline{x})$.

Simple sequences are sequences consisting of simple terms and sequence variables. We use $\tilde{s}$ to denote simple sequences. A *simple substitution* is a substitution that maps individual variables to simple terms, sequence variables to simple sequences, function variables to function symbols and function variables, and context variables to contexts, such that all but finitely many individual, sequence, and function variables are mapped to themselves, and all but finitely many context variables are mapped to themselves applied to the hole.

Example 2.2. *Let* $t = X_{\mathsf{f}}(\overline{x}, Y, X_{\mathsf{c}}(X))$, $C = f(X_{\mathsf{c}}(Y), X_{\mathsf{f}}(\overline{y}, \circ), X)$ *and* $\sigma = \{X \mapsto f(\overline{x}, a),\ \overline{x} \mapsto \lceil a, f(b, X), \overline{y}\rceil,\ \overline{y} \mapsto \lceil\rceil,\ X_{\mathsf{f}} \mapsto f,\ X_{\mathsf{c}} \mapsto g(\overline{y}, \circ, b)\}$. *Then:*

$$t\sigma = f(a, f(b, X), \overline{y}, Y, g(\overline{y}, f(\overline{x}, a), b)).$$
$$C\sigma = f(g(\overline{y}, Y, b), f(\circ), f(\overline{x}, a)).$$

2.4 Equation Solving

Solving equations between terms is a key technique used in many areas of theorem proving, declarative programming, computational linguistics, etc. Unification, matching, and in general, constraint solving require solving term equations. Below we give a brief overview of some unification problems that are related to our constraint solving problem in Chapter 3.

For surveys about unification, we refer [BS01, Dow01].

The most basic unification problem is first-order unification, which, in our terminology, can be seen as the problem of solving equations between simple terms constructed over function symbols and term variables. If the same function symbol has different number of arguments in different places, these occurrences are treated as different symbols. Robinson [Rob65] developed an algorithm for solving first-order unification problems, which is still in use in various systems despite its exponential complexity. Its improvements have been proposed in, e.g., [Hue76, PW78, MM82, CB83, EIG88, RP89]. If a first-order unification problem is solvable, then it has a unique solution modulo subsumption ordering, called a most general unifier. Such problems are called problems of unitary unification type. First-order unification is an example of unitary unification.

If we slightly extend the syntax, permitting in the unification problems simple terms built over function symbols and term and sequence variables, we obtain sequence unification problems. This "small change", however, has a dramatic influence on solutions. Solvable sequence unification problems may have infinitely many solutions that are incomparable with respect to subsumption ordering. Such problems are called infinitary. Kutsia [Kut04, Kut07] proved decidability of sequence unification and developed a minimal and complete solving procedure for it.

If we permit context variables instead of sequence variables, we obtain context unification problems. Similarly to first-order unification, function symbols are identified not only by the name but also by the arity. Like sequence unification, context unification is infinitary. Its decidability turned out to be a very difficult problem, which remained open for more than 20 years, until Jeż [Jeż14] proved that it is in PSPACE. A complete solving procedure is due to Levy [Lev96]. Kutsia, Levy, and Villaret [KLV07, KLV10] studied the relationship between sequence and context unification and showed that sequence unification is equivalent to a variant of the left-hole fragment of context unification.

Permitting function variables in unification problems affects neither decidability nor the unification type. However, if we consider terms with arbitrary higher-order variables, then unification quickly becomes undecidable [Gol81].

In this thesis, we consider constraint solving problems over simple terms, as well as some special matching problems over arbitrary terms.

3. Constraint Solving

3.1 Introduction

Various forms of constraint solving are in the center of declarative programming paradigms. Unification is the main computational mechanism for logic programming. Matching plays the same role in rule-based and functional programming. Constraints over special domains are in the heart of constraint logic programming languages.

In this chapter we propose a solving algorithm for equational and membership constraints over simple terms, contexts, and sequences, as defined in the previous chapter. It will be used in Chapter 4, where we will be discussing constraint logic programming over sequences and contexts. Special fragments of this constraint solving problem have applications in rule-based and pattern-based programming, discussed respectively in Chapter 5 and Chapter 6.

We start with formulating the constraint language and then define its semantics. The primitive constraints are equations and membership constraints for regular sequence and context languages. We may have function symbols whose argument order does not matter (unordered symbols): A kind of generalization of the commutativity property to unranked terms. The algorithm works on the input in disjunctive normal form and transforms it to the partially solved form. It is sound and terminating. The latter property naturally implies that the solver is incomplete for arbitrary constraints, because the problem it solves is infinitary: There might be infinitely many incomparable solutions to constraints that involve sequence and context variables, see, e.g., [Kut02, Lev96, Vil04]. However, there are fragments of constraints for which the solver is complete, i.e., it computes all the solutions. One of such fragments is so called the well-moded fragment [DFKM14, KM06], 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 sequence matching and context matching (which are known to be NP-complete [SSS04, KM12]), plus some early failure detection rules. Another fragment for which the solver is complete is named after the Knowledge Interchange Format, KIF [Gen98], which is a computer-oriented language for the interchange of knowledge among disparate computer programs. In KIF, function symbols do not have a fixed arity and sequence variables are permitted only in the last argument positions. The KIF fragment we consider in this thesis can be characterized exactly in this way: There we have essentially a first-order language over unranked function symbols, and sequence variables appear only in the last argument positions.

3.2 Syntax

The constraint alphabet $\mathcal{CA}$ extends the alphabet $\mathcal{A}$ by the following symbols:

- The propositional constants true and false, the equality predicate $\doteq$, and the membership predicate in.
- The set $\mathcal{P}$ of predicate symbols, denoted by $p, q, \ldots$.
- Regular operators: eps (empty sequence), . (sequence concatenation), | (sequence choice), $*$ (sequence repetition), $\bullet$ (empty context), $\cdot$ (context concatenation), $+$ (context choice), $\star$ (context repetition).
- Logical connectives and quantifiers: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence), $\exists$ (existential quantifier), $\forall$ (universal quantifier).

Regular sequence expressions RS and *regular context expressions* RC are defined inductively:

$$\mathsf{RS} ::= \mathsf{eps} \mid \mathsf{RS}.\mathsf{RS} \mid \mathsf{RS}|\mathsf{RS} \mid \mathsf{RS}^* \mid f(\mathsf{RS})$$
$$\mathsf{RC} ::= \bullet \mid \mathsf{RC} \cdot \mathsf{RC} \mid \mathsf{RC} + \mathsf{RC} \mid \mathsf{RC}^\star \mid f(\mathsf{RS}, \mathsf{RC}, \mathsf{RS})$$

where eps is the regular expression for the empty sequence and is omitted whenever it appears as an argument of a function symbol. For example, an expression $f((a(\mathsf{eps})|b(\mathsf{eps}))^*).c(\mathsf{eps})^*$ will be written as $f((a|b)^*).c^*$. The meta symbol R will be used to denote both regular sequence expressions RS and regular context expressions RC.

We assume that there is a possibly empty subset of $\mathcal{F}$, whose elements have so called unordered property: $f(\overline{x}, X, \overline{y}, Y, \overline{z}) \doteq f(\overline{x}, Y, \overline{y}, X, \overline{z})$ for all $\overline{x}, Y, \overline{y}, X, \overline{z}$. This property generalizes commutativity to function symbols that do not have a fixed arity. We call such symbols unordered function symbols and denote their set by $\mathcal{F}_{\mathsf{u}}$. The meta symbol f_{u} varies over unordered function symbols. To distinguish, the set $\mathcal{F}_{\mathsf{o}} := \mathcal{F} \setminus \mathcal{F}_{\mathsf{u}}$ will be called the set of ordered function symbols. The meta symbol f_{o} will stand for the elements of $\mathcal{F}_{\mathsf{o}}$. We still use f when the distinction between ordered and unordered symbols does not matter.

Definition 3.1. A *formula* over the alphabet $\mathcal{CA}$ is defined inductively as follows:

a) true and false are formulas.

b) If t and r are simple terms, then $t \doteq r$ is a formula.

c) If C and D are contexts, then $C \doteq D$ is a formula.

d) If $\tilde{s}$ is a simple sequence and RS is a sequence regular expression, then $\tilde{s}$ in RS is a formula.

e) If C is a context and RC is a context regular expression, then C in RC is a formula.

f) If p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are simple terms, then $p(t_1, \ldots, t_n)$ is a formula. It is called an atomic formula or simply an atom.

g) If $\mathbf{F}_1$ and $\mathbf{F}_2$ are formulas, then so are $(\neg\mathbf{F}_1)$, $(\mathbf{F}_1 \vee \mathbf{F}_2)$, $(\mathbf{F}_1 \wedge \mathbf{F}_2)$, $(\mathbf{F}_1 \Rightarrow \mathbf{F}_2)$, and $(\mathbf{F}_1 \Leftrightarrow \mathbf{F}_2)$.

h) If $\mathbf{F}$ is a formula and $v \in \mathcal{V}_{\mathsf{ISFC}}$, then $\exists v.\mathbf{F}$ and $\forall v.\mathbf{F}$ are formulas.

The formulas defined by the items b)–e) are called *primitive constraints.* A *literal* $\mathbf{L}$ is an atom or a primitive constraint. A *constraint* is an arbitrary formula built over true, false and primitive constraints. Application of simple substitutions is extended to literals and conjunctions of literals in a natural way.

For example, $X_{\mathsf{f}}(\overline{x}, X_{\mathsf{c}}(X), b) \doteq f(\overline{y}, X) \wedge \mathsf{false} \wedge (\mathsf{true} \vee f(X_{\mathsf{c}}(g(\circ, X)), \overline{x})\ \mathsf{in}\ f((a|b)^*, g(a, \bullet)))$ is a constraint, where $X_{\mathsf{f}}(\overline{x}, X_{\mathsf{c}}(X), b) \doteq f(\overline{y}, X)$ and $f(X_{\mathsf{c}}(g(\circ, X)), \overline{x})\ \mathsf{in}\ f((a|b)^*, g(a, \bullet))$ are primitive constraints.

The language generated from $\mathcal{CA}$ is denoted by $\mathcal{L}(\mathcal{CA})$.

The *sets of free and bound variables* of a formula $\mathbf{F}$, denoted $\mathtt{fvar}(\mathbf{F})$ and $\mathtt{bvar}(\mathbf{F})$ respectively, are defined inductively as follows:

$\mathtt{fvar}(\mathsf{true}) = \emptyset$.

$\mathtt{fvar}(\mathsf{false}) = \emptyset$.

$\mathtt{fvar}(t \doteq s) = \mathtt{fv}(t) \cup \mathtt{fv}(s)$.

$\mathtt{fvar}(C \doteq D) = \mathtt{fv}(C) \cup \mathtt{fv}(D)$.

$\mathtt{fvar}(\tilde{s} \text{ in } \mathsf{RS}) = \mathtt{fv}(\tilde{s})$.

$\mathtt{fvar}(C \text{ in } \mathsf{RC}) = \mathtt{fv}(C)$.

$\mathtt{fvar}(p(t_1, \ldots, t_n)) = \Sigma_{i=1}^{n} \mathtt{fv}(t_i)$.

$\mathtt{fvar}(\neg \mathbf{F}) = \mathtt{fvar}(\mathbf{F})$.

$\mathtt{fvar}(\mathbf{F}_1 \diamond \mathbf{F}_2) = \mathtt{fvar}(\mathbf{F}_1) \cup \mathtt{fvar}(\mathbf{F}_2)$, where $\diamond \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$.

$\mathtt{fvar}(\mathsf{Q}v.\mathbf{F}) = \mathtt{fvar}(\mathbf{F}) \setminus \{v\}$, where $\mathsf{Q} \in \{\exists, \forall\}$.

$\mathtt{bvar}(\mathbf{F}) = \emptyset$, if $\mathbf{F}$ is true, false, or a literal.

$\mathtt{bvar}(\neg \mathbf{F}) = \mathtt{bvar}(\mathbf{F})$.

$\mathtt{bvar}(\mathbf{F}_1 \diamond \mathbf{F}_2) = \mathtt{bvar}(\mathbf{F}_1) \cup \mathtt{bvar}(\mathbf{F}_2)$, where $\diamond \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$.

$\mathtt{bvar}(\mathsf{Q}v.\mathbf{F}) = \mathtt{bvar}(\mathbf{F}) \cup \{v\}$, where $\mathsf{Q} \in \{\exists, \forall\}$.

3.3 Semantics

For a given set S, we denote by S^* the set of finite, possibly empty, sequences of elements of S, and by S^n the set of sequences of length n of elements of S. Given a sequence $s = \lceil s_1, s_2, \ldots, s_n \rceil \in S^n$, we denote by $perm(s)$ the set of sequences $\{\lceil s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(n)} \rceil \mid \pi$ is a permutation of $\{1, 2, \ldots, n\}\}$. The set of functions from a set S_1 to a set S_2 is denoted by $S_1 \longrightarrow S_2$. The notion $f : S_1 \longrightarrow S_2$ means that f belongs to $S_1 \longrightarrow S_2$.

A *structure* $\mathfrak{S}$ for a language $\mathcal{L}(\mathcal{CA})$ is a tuple $\langle \mathcal{D}, \mathcal{I} \rangle$ made of a non-empty *carrier set* of individuals $\mathcal{D}$ and an interpretation function $\mathcal{I}$ that maps each function symbol $f \in \mathcal{F}$ to a function $\mathcal{I}(f) : \mathcal{D}^* \longrightarrow \mathcal{D}$, and each n-ary predicate symbol $p \in \mathcal{P}$ to an n-ary relation $\mathcal{I}(p) \subseteq \mathcal{D}^n$. Moreover, if $f \in \mathcal{F}_{\mathsf{u}}$ then $\mathcal{I}(f)(s) = \mathcal{I}(f)(s')$ for all $s \in D^*$ and $s' \in perm(s)$. Given such a structure, we also define the operation $\mathcal{I}_{\mathsf{C}} : (\mathcal{D}^* \longrightarrow \mathcal{D}) \longrightarrow \mathcal{D}^* \longrightarrow \mathcal{D}^* \longrightarrow (\mathcal{D} \longrightarrow \mathcal{D}) \longrightarrow (\mathcal{D} \longrightarrow \mathcal{D})$ by $\mathcal{I}_{\mathsf{C}}(\psi)(\tilde{s}_1)(\tilde{s}_2)(\phi)(d) := \psi(\tilde{s}_1, \phi(d), \tilde{s}_2)$ for all $\psi : \mathcal{D}^* \longrightarrow \mathcal{D}$,

$\tilde{s}_1, \tilde{s}_2 \in \mathcal{D}^*$, $d \in \mathcal{D}$, and $\phi : \mathcal{D} \longrightarrow \mathcal{D}$.

A *variable assignment* for such a structure is a function with the domain $\mathcal{V}_{\mathsf{ISFC}}$ that maps individual variables to elements of $\mathcal{D}$; sequence variable to elements of $\mathcal{D}^*$; function variables to functions in $\mathcal{D}^* \longrightarrow \mathcal{D}$; and context variables to functions in $\mathcal{D} \longrightarrow \mathcal{D}$.

The interpretations of our syntactic categories with respect to a structure $\mathfrak{S} = \langle \mathcal{D}, \mathcal{I} \rangle$ and variable assignment ρ is shown below. The *interpretation* of simple sequences $[\![\tilde{s}]\!]_{\mathfrak{S},\rho}$ and of contexts $[\![C]\!]_{\mathfrak{S},\rho}$ are defined as follows:

$$
\begin{aligned}
[\![X]\!]_{\mathfrak{S},\rho} &:= \rho(X).\\
[\![f(\tilde{s})]\!]_{\mathfrak{S},\rho} &:= \mathcal{I}(f)([\![\tilde{s}]\!]_{\mathfrak{S},\rho}).\\
[\![X_{\mathsf{f}}(\tilde{s})]\!]_{\mathfrak{S},\rho} &:= \rho(X_{\mathsf{f}})([\![\tilde{s}]\!]_{\mathfrak{S},\rho}).\\
[\![X_{\mathsf{c}}(t)]\!]_{\mathfrak{S},\rho} &:= \rho(X_{\mathsf{c}})([\![t]\!]_{\mathfrak{S},\rho}).\\
[\![\overline{x}]\!]_{\mathfrak{S},\rho} &:= \rho(\overline{x}).\\
[\![\lceil \overline{t}_1, \ldots, \overline{t}_n \rceil]\!]_{\mathfrak{S},\rho} &:= [\![\overline{t}_1]\!]_{\mathfrak{S},\rho} \bowtie \cdots \bowtie [\![\overline{t}_n]\!]_{\mathfrak{S},\rho}.\\
[\![\circ]\!]_{\mathfrak{S},\rho} &:= Id_{\mathcal{D}}.\\
[\![f(\tilde{s}_1, C, \tilde{s}_2)]\!]_{\mathfrak{S},\rho} &:= \mathcal{I}_{\mathsf{C}}(\mathcal{I}(f))([\![\tilde{s}_1]\!]_{\mathfrak{S},\rho})([\![\tilde{s}_2]\!]_{\mathfrak{S},\rho})([\![C]\!]_{\mathfrak{S},\rho}).\\
[\![X_{\mathsf{f}}(\tilde{s}_1, C, \tilde{s}_2)]\!]_{\mathfrak{S},\rho} &:= \mathcal{I}_{\mathsf{C}}(\rho(X_{\mathsf{f}}))([\![\tilde{s}_1]\!]_{\mathfrak{S},\rho})([\![\tilde{s}_2]\!]_{\mathfrak{S},\rho})([\![C]\!]_{\mathfrak{S},\rho}).\\
[\![X_{\mathsf{c}}(C)]\!]_{\mathfrak{S},\rho} &:= \rho(X_{\mathsf{c}}) \circledcirc [\![C]\!]_{\mathfrak{S},\rho}, \text{ where } \circledcirc \text{ stands for function composition.}
\end{aligned}
$$

Note that terms are interpreted as elements of $\mathcal{D}$, sequences as elements of $\mathcal{D}^*$, and contexts as elements of $\mathcal{D} \longrightarrow \mathcal{D}$. We may omit ρ and write simply $[\![E]\!]_{\mathfrak{S}}$ for the interpretation of a variable-free (i.e., *ground*) expression E.

Overloading the notation, we use $[\![.]\!]$ for the interpretation of regular sequence and context expressions. First, we introduce the following notation:

$$
\begin{aligned}
&[\![\mathsf{RC}]\!]^0_{\mathfrak{S}} := \{Id_{\mathcal{D}}\}, \text{ where } Id_{\mathcal{D}} : \mathcal{D} \longrightarrow \mathcal{D} \text{ is the identity function on } \mathcal{D}.\\
&[\![\mathsf{RC}]\!]^{n+1}_{\mathfrak{S}} := \{C_1 \circledcirc C_2 \mid C_1 \in [\![\mathsf{RC}]\!]_{\mathfrak{S}}, C_2 \in [\![\mathsf{RC}]\!]^n_{\mathfrak{S}}\} \text{ for } n \geq 0.\\
&[\![\mathsf{RC}]\!]^{\star}_{\mathfrak{S}} := \bigcup_{n \geq 0} [\![\mathsf{RC}]\!]^n_{\mathfrak{S}}.
\end{aligned}
$$

Now, the *interpretation of regular expressions* is defined by structural induction as follows:

$$[\![\mathsf{eps}]\!]_{\mathfrak{S}} := \{\lceil\ \rceil\}.$$

$$
\begin{aligned}
[\![\mathsf{RS}_1.\mathsf{RS}_2]\!]_{\mathfrak{S}} &:= \{\tilde{s}_1 \bowtie \tilde{s}_2 \mid \tilde{s}_1 \in [\![\mathsf{RS}_1]\!]_{\mathfrak{S}}, \tilde{s}_2 \in [\![\mathsf{RS}_2]\!]_{\mathfrak{S}}\}.\\
[\![\mathsf{RS}_1|\mathsf{RS}_2]\!]_{\mathfrak{S}} &:= [\![\mathsf{RS}_1]\!]_{\mathfrak{S}} \cup [\![\mathsf{RS}_2]\!]_{\mathfrak{S}}.\\
[\![\mathsf{RS}^*]\!]_{\mathfrak{S}} &:= [\![\mathsf{RS}]\!]^*_{\mathfrak{S}}.\\
[\![f(\mathsf{RS})]\!]_{\mathfrak{S}} &:= \{\mathcal{I}(f)(\tilde{s}) \mid \tilde{s} \in [\![\mathsf{RS}]\!]_{\mathfrak{S}}\}.\\
[\![\bullet]\!]_{\mathfrak{S}} &:= \{Id_{\mathcal{D}}\}.\\
[\![\mathsf{RC}_1 \cdot \mathsf{RC}_2]\!]_{\mathfrak{S}} &:= \{C_1 \circledcirc C_2 \mid C_1 \in [\![\mathsf{RC}_1]\!]_{\mathfrak{S}}, C_2 \in [\![\mathsf{RC}_2]\!]_{\mathfrak{S}}\}.\\
[\![\mathsf{RC}_1 + \mathsf{RC}_2]\!]_{\mathfrak{S}} &:= [\![\mathsf{RC}_1]\!]_{\mathfrak{S}} \cup [\![\mathsf{RC}_2]\!]_{\mathfrak{S}}.\\
[\![\mathsf{RC}^\star]\!]_{\mathfrak{S}} &:= [\![\mathsf{RC}]\!]^\star_{\mathfrak{S}}.\\
[\![f(\mathsf{RS}_1, \mathsf{RC}, \mathsf{RS}_2)]\!]_{\mathfrak{S}} &:= \{\mathcal{I}_{\mathsf{C}}(\mathcal{I}(f))(\tilde{s}_1)(\tilde{s}_2)(C) \mid C \in [\![\mathsf{RC}]\!]_{\mathfrak{S}}, \tilde{s}_1 \in [\![\mathsf{RS}_1]\!]_{\mathfrak{S}}, \tilde{s}_2 \in [\![\mathsf{RS}_2]\!]_{\mathfrak{S}}\}.
\end{aligned}
$$

The formulae true, false, and literals are *interpreted* with respect to a structure $\mathfrak{S}$ and a variable assignment ρ as follows:

$\mathfrak{S} \models_\rho \mathsf{true}$.

Not $\mathfrak{S} \models_\rho \mathsf{false}$.

$\mathfrak{S} \models_\rho t_1 \doteq t_2$ iff $[\![t_1]\!]_{\mathfrak{S},\rho} = [\![t_2]\!]_{\mathfrak{S},\rho}$.

$\mathfrak{S} \models_\rho C_1 \doteq C_2$ iff $[\![C_1]\!]_{\mathfrak{S},\rho} = [\![C_2]\!]_{\mathfrak{S},\rho}$.

$\mathfrak{S} \models_\rho \tilde{s}\ \mathsf{in\ RS}$ iff $[\![\tilde{s}]\!]_{\mathfrak{S},\rho} \in [\![\mathsf{RS}]\!]_{\mathfrak{S}}$.

$\mathfrak{S} \models_\rho C\ \mathsf{in\ RC}$ iff $[\![C]\!]_{\mathfrak{S},\rho} \in [\![\mathsf{RC}]\!]_{\mathfrak{S}}$.

$\mathfrak{S} \models_\rho p(t_1, \ldots, t_n)$ iff $\mathcal{I}(p)([\![t_1]\!]_{\mathfrak{S},\rho}, \ldots, [\![t_n]\!]_{\mathfrak{S},\rho})$ holds.

Interpretation of an arbitrary formula with respect to a structure and a variable assignment is defined in the standard way. Also, the notions $\mathfrak{S} \models \mathbf{F}$ for validity of an arbitrary formula $\mathbf{F}$ in $\mathfrak{S}$, and $\models \mathbf{F}$ for validity of $\mathbf{F}$ in any structure are defined as usual.

An *intended structure* is a structure $\mathfrak{I}$ with a carrier set $\mathcal{T}^s(\mathcal{F})$ (the set of ground simple terms) and interpretation $\mathcal{I}$ defined for every $f \in \mathcal{F}$ by $\mathcal{I}(f)(\tilde{s}) := f(\tilde{s})$. It follows that $\mathcal{I}_{\mathsf{C}}(\mathcal{I}(f))(\tilde{s}_1)(\tilde{s}_2)(C) := f(\tilde{s}_1, C, \tilde{s}_2)$. Thus, intended structures identify terms, sequences and contexts with themselves. Also, $[\![\mathtt{R}]\!]_{\mathcal{I}}$ is the same in all intended structures, and will be denoted by $[\![\mathtt{R}]\!]$. Other remarkable properties of intended structures $\mathfrak{I}$ are: $\mathfrak{I} \models_\rho t_1 \doteq t_2$ iff $t_1\rho = t_2\rho$, $\mathfrak{I} \models_\rho C_1 \doteq C_2$ iff $C_1\rho = C_2\rho$, $\mathfrak{I} \models_\rho \tilde{s}\,\mathsf{in\,RS}$ iff $\tilde{s}\rho \in [\![\mathsf{RS}]\!]$, and $\mathfrak{I} \models_\rho C\,\mathsf{in\,RC}$ iff $C\rho \in [\![\mathsf{RC}]\!]$.

A ground substitution ρ is a *solution* of a constraint $\mathcal{C}$ if $\mathfrak{I} \models \mathcal{C}\rho$ for all intended structures $\mathfrak{I}$.

3.4 Solver

In this section we present a constraint solver. It is based on rules, transforming a constraint in *disjunctive normal form* (DNF) into a constraint in DNF. We say a constraint is in DNF, if it has a form $\mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$, where $\mathcal{K}$'s are conjunctions of true, false, and primitive constraints. The number of solver rules is not small (as it is usual for such kind of solvers, cf., e.g., [DPPR00, Com98]). To make their comprehension easier, we group them so that similar ones are collected together in subsections. Within each subsection, for better readability, the rule groups are put between horizontal lines.

Before going into the details, we introduce a more conventional way of writing expressions, some kind of syntactic sugar, that should make reading easier. In the rest of this chapter, the letter F is a meta symbol for function symbols and function variables. The symmetric closure of $\doteq$ is denoted by $\simeq$. We write $F_1 \doteq F_2$ instead of $F_1() \doteq F_2()$, $X_{\mathsf{c}} \doteq C$ instead of $X_{\mathsf{c}}(\circ) \doteq C$, and X_{c} in RC instead of $X_{\mathsf{c}}(\circ)$ in RC.

3.4.1 Logical Rules

The logical rules perform logical transformations of the constraints and have to be applied in constraints, at any depth modulo associativity and commutativity of disjunction and conjunction. $\mathbf{F}$ stands for any formula. We denote the whole set of rules by Log.

$$\mathbf{F} \wedge \mathbf{F} \rightsquigarrow \mathbf{F} \qquad \mathsf{true} \wedge \mathbf{F} \rightsquigarrow \mathbf{F} \qquad \mathsf{false} \wedge \mathbf{F} \rightsquigarrow \mathsf{false} \qquad \tilde{s} \simeq \tilde{s} \rightsquigarrow \mathsf{true}$$

$$\mathbf{F} \vee \mathbf{F} \rightsquigarrow \mathbf{F} \qquad \mathsf{true} \vee \mathbf{F} \rightsquigarrow \mathsf{true} \qquad \mathsf{false} \vee \mathbf{F} \rightsquigarrow \mathbf{F} \qquad C \simeq C \rightsquigarrow \mathsf{true}$$

$$\lceil\ \rceil\ \mathsf{in}\ \mathsf{RS} \rightsquigarrow \mathsf{true},\ \text{if } \lceil\ \rceil \in [\![\mathsf{RS}]\!] \qquad \circ\ \mathsf{in}\ \mathsf{RC} \rightsquigarrow \mathsf{true},\ \text{if } \circ \in [\![\mathsf{RC}]\!]$$

3.4.2 Failure Rules

In the second group there are rules for failure detection. The first two rules detect function symbol clash:

(F1) $f_1(\tilde{s}_1) \simeq f_2(\tilde{s}_2) \rightsquigarrow \mathsf{false}, \quad \text{if } f_1 \neq f_2.$

(F2) $f_1(\tilde{s}_1, C_1, \tilde{s}_2) \simeq f_2(\tilde{s}_3, C_2, \tilde{s}_4) \rightsquigarrow \mathsf{false}, \quad \text{if } f_1 \neq f_2.$

The next three rules perform occurrence check. Peculiarity of this operation for our language is that the variable occurrence into a term/context does not always leads to failure. For instance, the equation $X \doteq X_{\mathsf{c}}(X)$, where the variable X occurs in $X_{\mathsf{c}}(X)$, still has a solution $\{X_{\mathsf{c}} \mapsto \circ\}$. Therefore, the occurrence check should fail on equations of the form $var \doteq nonvar$ only if no instance of the non-variable expression $nonvar$ can become the variable var. To achieve this, the rules below require the non-variable terms to contain F (the first two rules) and t (the third rule), which can not be erased by a substitution application:

(F3) $X \simeq C[F(\tilde{s})] \rightsquigarrow \mathsf{false}, \qquad \text{if } X \in var(C[F(\tilde{s})]).$

(F4) $X_{\mathsf{c}} \simeq C_1[F(C_2)] \rightsquigarrow \mathsf{false}, \qquad \text{if } X_{\mathsf{c}} \in var(C_1[F(C_2)]).$

(F5) $\overline{x} \simeq \lceil \tilde{s}_1, t, \tilde{s}_2 \rceil \rightsquigarrow \mathsf{false}, \qquad \text{if } \overline{x} \in var(\lceil \tilde{s}_1, t, \tilde{s}_2 \rceil).$

Further, we have two more rules which lead to failure in the case when the hole is unified with a context whose all possible instances are nontrivial contexts (guaranteed by the presence of F), and when the empty sequence is attempted to match to an inherently nonempty sequence (guaranteed by t):

(F6) $\circ \simeq C_1[F(C_2)] \rightsquigarrow \mathsf{false}.$ (F7) $\lceil\, \rceil \simeq \lceil \tilde{s}_1, t, \tilde{s}_2 \rceil \rightsquigarrow \mathsf{false}.$

The other rules deals with the membership atoms. The first two of them lead to failure because the language generated by the regular expression does not contain the empty sequence (in the first rule) and the hole (in the second rule):

(F8) $\lceil\, \rceil \text{ in } f(\mathsf{RS}) \rightsquigarrow \mathsf{false}.$ (F9) $\circ \text{ in } f(\mathsf{RS}_1, \mathsf{RC}, \mathsf{RS}_2) \rightsquigarrow \mathsf{false}.$

The following two rules give failure because of the top function symbol clash between a term/context and the corresponding regular expression:

(F10) $f_1(\tilde{s})$ in $f_2(\mathsf{RS}) \rightsquigarrow$ false, if $f_1 \neq f_2$.

(F11) $f_1(\tilde{s}_1, C, \tilde{s}_2)$ in $f_2(\mathsf{RS}_1, \mathsf{RC}, \mathsf{RS}_2) \rightsquigarrow$ false, if $f_1 \neq f_2$.

The next two rules are justified by the fact that a nonempty sequence can not belong to the language consisting of the empty sequence only, and a nontrivial context can not be a member of the language containing only the hole:

(F12) $\lceil \tilde{s}_1, t, \tilde{s}_2 \rceil$ in eps $\rightsquigarrow$ false.

(F13) $C_1[F(\tilde{s}_1, C_2, \tilde{s}_2)]$ in $\bullet \rightsquigarrow$ false.

We denote this set of rules (F1) – (F13) by Fail.

3.4.3 Deletion Rules

There are five rules which delete identical terms, sequence variables or context variables from both sides of an equation. They are more or less self-explanatory. Just note that under unordered head, we delete an arbitrary occurrence of a term (that is not a sequence variable).

(Del1) $X_{\mathsf{c}}(t_1) \simeq X_{\mathsf{c}}(t_2) \wedge \rightsquigarrow t_1 \doteq t_2$.

(Del2) $X_{\mathsf{c}}(C_1) \simeq X_{\mathsf{c}}(C_2) \rightsquigarrow C_1 \doteq C_2$.

(Del3) $f_{\mathsf{u}}(\tilde{s}_1, \overline{t}, \tilde{s}_2) \simeq f_{\mathsf{u}}(\tilde{s}_3, \overline{t}, \tilde{s}_4) \rightsquigarrow f_{\mathsf{u}}(\tilde{s}_1, \tilde{s}_2) \doteq f_{\mathsf{u}}(\tilde{s}_3, \tilde{s}_4)$.

(Del4) $\lceil \overline{x}, \tilde{s}_1 \rceil \simeq \lceil \overline{x}, \tilde{s}_2 \rceil \rightsquigarrow \tilde{s}_1 \doteq \tilde{s}_2$.

(Del5) $\overline{x} \simeq \lceil \tilde{s}_1, \overline{x}, \tilde{s}_2 \rceil \rightsquigarrow \tilde{s}_1 \doteq \lceil\, \rceil \wedge \tilde{s}_2 \doteq \lceil\, \rceil$,

In the last rule $\tilde{s}_1$ is not the empty sequence.

We denote the set of rules (Del1) - (Del5) by Del.

3.4.4 Membership Rules

We start with membership rules for sequences. When the sequence $\tilde{s}$ in the membership atom $\tilde{s}$ in RS is ground, then these rules simply provide the membership check. Non-ground sequences require more special treatment and before giving rules for membership atom transformation, we elaborate on the form of regular sequence expressions.

To solve membership constraints for sequences of the form $\lceil t, \tilde{s} \rceil$ with t a term, we rely on the possibility of computing the linear form of a regular sequence expression, that is, to express it as a choice of concatenations of regular sequence expressions that identifies all plausible membership constraints for t and $\tilde{s}$. Formally, the *linear form* of a regular expression RS, denoted $lf_s(\mathsf{RS})$, is a finite set of pairs $(f(\mathsf{RS}_1), \mathsf{RS}_2)$ called *monomials*, which is defined recursively as follows:

$lf_s(\mathsf{eps}) = \emptyset$.

$lf_s(f(\mathsf{RS})) = \{(f(\mathsf{RS}), \mathsf{eps})\}$.

$lf_s(\mathsf{RS}_1|\mathsf{RS}_2) = lf_s(\mathsf{RS}_1) \cup lf_s(\mathsf{RS}_2)$.

$lf_s(\mathsf{RS}_1.\mathsf{RS}_2) = lf_s(\mathsf{RS}_1) \odot \mathsf{RS}_2$, if $\lceil\,\rceil \notin [\![\mathsf{RS}_1]\!]$.

$lf_s(\mathsf{RS}_1.\mathsf{RS}_2) = lf_s(\mathsf{RS}_1) \odot \mathsf{RS}_2 \cup lf_s(\mathsf{RS}_2)$, if $\lceil\,\rceil \in [\![\mathsf{RS}_1]\!]$.

$lf_s(\mathsf{RS}^*) = lf_s(\mathsf{RS}) \odot \mathsf{RS}^*$.

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 $l \odot \mathsf{eps} = l$, and $l \odot \mathsf{RS} = \{(f(\mathsf{RS}_1), \mathsf{RS}_2.\mathsf{RS}) \mid (f(\mathsf{RS}_1), \mathsf{RS}_2) \in l, \mathsf{RS}_2 \neq \mathsf{eps}\} \cup \{(f(\mathsf{RS}_1), \mathsf{RS}) \mid (f(\mathsf{RS}_1), \mathsf{eps}) \in l\}$, if $\mathsf{RS} \neq \mathsf{eps}$. Instead of explicitly listing all syntactic conditions to the other rules (e.g., MS1, MS4, MS5, MS6, MS7 and MS12) that would prevent (MS8) to be their alternative, we decided to put a syntactically rather implicit condition to (MS8) to guarantee determinism. Note also the intersection in the rule (MS8): RS is closed under this operation, see, e.g., [CDG+07].

(MS1) $\lceil \overline{x}_1, \ldots, \overline{x}_n \rceil$ in $\mathsf{eps} \wedge \mathcal{K} \rightsquigarrow \wedge_{i=1}^{n}\, \overline{x}_i \doteq \lceil\,\rceil \wedge \mathcal{K}\theta$,

where $n > 0$ and $\theta = \{\overline{x}_1 \mapsto \lceil\,\rceil, \ldots, \overline{x}_n \mapsto \lceil\,\rceil\}$

(MS2) $\lceil t, \tilde{s} \rceil$ in $\mathsf{RS} \wedge \mathcal{K} \rightsquigarrow \bigvee_{(f(\mathsf{RS}_1), \mathsf{RS}_2) \in lf_s(\mathsf{RS})} \Big(t \;\mathsf{in}\; f(\mathsf{RS}_1) \wedge \tilde{s} \;\mathsf{in}\; \mathsf{RS}_2 \wedge \mathcal{K} \Big)$,

where $\tilde{s} \neq \lceil\,\rceil$ and $\mathsf{RS} \neq \mathsf{eps}$.

(MS3) $\lceil \overline{x}, \tilde{s} \rceil$ in $f(\mathsf{RS}) \wedge \mathcal{K} \rightsquigarrow$

$$\Big(\overline{x} \text{ in } f(\mathsf{RS}) \wedge \tilde{s} \doteq \lceil\,\rceil \wedge \mathcal{K}\Big) \vee \Big(\overline{x} \doteq \lceil\,\rceil \wedge \tilde{s} \text{ in } f(\mathsf{RS}) \wedge \mathcal{K}\Big),$$

where $\tilde{s} \neq \lceil\,\rceil$.

(MS4) t in $\mathsf{RS}^* \rightsquigarrow t$ in RS.

(MS5) t in $\mathsf{RS}_1.\mathsf{RS}_2 \wedge \mathcal{K} \rightsquigarrow$

$$\Big(t \text{ in } \mathsf{RS}_1 \wedge \lceil\,\rceil \text{ in } \mathsf{RS}_2 \wedge \mathcal{K}\Big) \vee \Big(\lceil\,\rceil \text{ in } \mathsf{RS}_1 \wedge t \text{ in } \mathsf{RS}_2 \wedge \mathcal{K}\Big).$$

(MS6) t in $\mathsf{RS}_1|\mathsf{RS}_2 \wedge \mathcal{K} \rightsquigarrow \Big(t \text{ in } \mathsf{RS}_1 \wedge \mathcal{K}\Big) \vee \Big(t \text{ in } \mathsf{RS}_2 \wedge \mathcal{K}\Big)$.

(MS7) $\lceil \overline{x}, \tilde{s} \rceil$ in $\mathsf{RS}_1|\mathsf{RS}_2 \wedge \mathcal{K} \rightsquigarrow \Big(\lceil \overline{x}, \tilde{s} \rceil \text{ in } \mathsf{RS}_1 \wedge \mathcal{K}\Big) \vee \Big(\lceil \overline{x}, \tilde{s} \rceil \text{ in } \mathsf{RS}_2 \wedge \mathcal{K}\Big)$.

(MS8) v in $\mathsf{RS}_1 \wedge v$ in $\mathsf{RS}_2 \rightsquigarrow v$ in RS,

where $v \in \mathcal{V}_\mathsf{I} \cup \mathcal{V}_\mathsf{S}, [\![\mathsf{RS}]\!] = [\![\mathsf{RS}_1]\!] \cap [\![\mathsf{RS}_2]\!]$, and neither v in RS_1 nor v in RS_2 can be transformed by other rules.

Next, we have rules that constrain singleton sequences to be in a term language. They proceed by the straightforward matching or decomposition of the structure. Note that in (MS11), we require the arguments of the unordered function symbol to be terms. The rule (MS9) does not distinguish whether f is ordered or unordered. We also have (MS12), where a sequence variable is constrained by a regular sequence expression in the form $f(\mathsf{RS})$. It forces the sequence variable to be instantiated by a single term.

(MS9) $X_\mathsf{f}(\tilde{s})$ in $f(\mathsf{RS}) \wedge \mathcal{K} \rightsquigarrow$

$$X_\mathsf{f} \doteq f \wedge f(\tilde{s})\{X_\mathsf{f} \mapsto f\} \text{ in } f(\mathsf{RS}) \wedge \mathcal{K}\{X_\mathsf{f} \mapsto f\}.$$

(MS10) $f_\mathsf{o}(\tilde{s})$ in $f_\mathsf{o}(\mathsf{RS}) \rightsquigarrow \tilde{s}$ in RS.

(MS11) $f_\mathsf{u}(\tilde{s})$ in $f_\mathsf{u}(\mathsf{RS}) \wedge \mathcal{K} \rightsquigarrow \bigvee_{\tilde{s}' \in perm(\tilde{s})} \Big(\tilde{s}' \text{ in } \mathsf{RS} \wedge \mathcal{K}\Big)$,

where $\tilde{s}$ is a sequence of terms.

(MS12) $\overline{x}$ in $f(\mathsf{RS}) \wedge \mathcal{K} \rightsquigarrow \overline{x} \doteq X \wedge X$ in $f(\mathsf{RS}) \wedge \mathcal{K}\{\overline{x} \mapsto X\}$, where X is fresh.

To solve membership constraints for contexts of the form $F(\tilde{s}_1, C, \tilde{s}_2)$, we use a similar

approach like we did to the one for solving membership constraints for sequences. We rely on the possibility to compute the linear form of a regular context expression, that is, to express it as a choice of concatenations of regular context expressions that identify all plausible membership constraints for $F(\tilde{s}_1, \circ, \tilde{s}_2)$ and C. Formally, the *linear form* of a regular context expression RC, denoted $lf_c(\mathsf{RC})$, is a finite set of pairs $(f(\mathsf{RS}_1, \bullet, \mathsf{RS}_2), \mathsf{RC})$ called *monomials*, which is defined recursively as follows:

$$lf_c(\bullet) = \emptyset.$$
$$lf_c(\mathsf{RC}^\star) = lf_c(\mathsf{RC}) \boxdot \mathsf{RC}^\star.$$
$$lf_c(f(\mathsf{RS}_1, \mathsf{RC}, \mathsf{RS}_2)) = \{(f(\mathsf{RS}_1, \bullet, \mathsf{RS}_2), \mathsf{RC})\}.$$
$$lf_c(\mathsf{RC}_1 + \mathsf{RC}_2) = lf_c(\mathsf{RC}_1) \cup lf_c(\mathsf{RC}_2).$$
$$lf_c(\mathsf{RC}_1 \cdot \mathsf{RC}_2) = lf_c(\mathsf{RC}_1) \boxdot \mathsf{RC}_2, \text{ if } \bullet \notin [\![\mathsf{RC}_1]\!].$$
$$lf_c(\mathsf{RC}_1 \cdot \mathsf{RC}_2) = lf_c(\mathsf{RC}_1) \boxdot \mathsf{RC}_2 \cup lf_c(\mathsf{RC}_2), \text{ if } \bullet \in [\![\mathsf{RC}_1]\!].$$

These equations involve an extension of concatenation $\boxdot$ that acts on a linear form and a regular expression and returns a linear form. It is defined as $l \boxdot \bullet = l$, and $l \boxdot \mathsf{RC} = \{(f(\mathsf{RS}_1, \bullet, \mathsf{RS}_2), \mathsf{RC}_1 \cdot \mathsf{RC}) \mid (f(\mathsf{RS}_1, \bullet, \mathsf{RS}_2), \mathsf{RC}_1) \in l, \mathsf{RC}_1 \neq \bullet\} \cup \{(f(\mathsf{RS}_1, \bullet, \mathsf{RS}_2), \mathsf{RC}) \mid (f(\mathsf{RS}_1, \bullet, \mathsf{RS}_2), \bullet) \in l\}$, if $\mathsf{RC} \neq \bullet$.

The rules for contexts are given below. Like for their counterparts for sequences, these rules provide membership test when the contexts in membership atoms are ground. $F(\tilde{s}_1, \circ, \tilde{s}_2)$ is a singleton context and it can be splitted in only one possible way: into $F(\tilde{s}_1, \circ, \tilde{s}_2)$ and $\circ$. The rules (MC1)–(MC8) are counterparts of the rules (MS1)–(MS8) respectively. Bellow we do not list syntactic conditions to the rules that would prevent (MC8) to be their alternative. Instead, we put a syntactically implicit condition to (MC8) to guarantee determinism, like we have done for the rule (MS8). For the intersection in the rule (MC8) we note that RC is also closed under this operation. This can be shown by translating the RC expressions into regular sequence automata.

(MC1) $X_{\mathsf{c}1}(\cdots X_{\mathsf{c}n}(\circ)\cdots)$ in $\bullet \wedge \mathcal{K} \rightsquigarrow \wedge_{i=1}^{n} X_{\mathsf{c}i} \doteq \circ \wedge \mathcal{K}\theta$
where $n > 0$ and $\theta = \{X_{\mathsf{c}1} \mapsto \circ, \ldots, X_{\mathsf{c}n} \mapsto \circ\}$.

(MC2) $F(\tilde{s}_1, C, \tilde{s}_2)$ in $\mathsf{RC} \wedge \mathcal{K} \rightsquigarrow$

$$\bigvee_{(f(\mathsf{RS}_1,\bullet,\mathsf{RS}_2),\mathsf{RC}_1)\in lf_c(\mathsf{RC})} \Big(F(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } f(\mathsf{RS}_1,\bullet,\mathsf{RS}_2)\wedge C \text{ in } \mathsf{RC}_1\wedge\mathcal{K}\Big),$$

where $C \neq \circ$ and $\mathsf{RC} \neq \bullet$.

(MC3) $X_{\mathsf{c}}(C) \text{ in } f(\mathsf{RS}_1,\bullet,\mathsf{RS}_2)\wedge\mathcal{K} \rightsquigarrow$

$$\Big(X_{\mathsf{c}} \text{ in } f(\mathsf{RS}_1,\bullet,\mathsf{RS}_2)\wedge C\simeq\circ\wedge\mathcal{K}\Big)\vee\Big(X_{\mathsf{c}}\simeq\circ\wedge C \text{ in } f(\mathsf{RS}_1,\bullet,\mathsf{RS}_2)\wedge\mathcal{K}\Big),$$

where $C \neq \circ$.

(MC4) $F(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } \mathsf{RC}^\star \rightsquigarrow F(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } \mathsf{RC}$.

(MC5) $F(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } \mathsf{RC}_1\cdot\mathsf{RC}_2\wedge\mathcal{K} \rightsquigarrow$

$$\Big(F(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } \mathsf{RC}_1\wedge\circ \text{ in } \mathsf{RC}_2\wedge\mathcal{K}\Big)\vee\Big(\circ \text{ in } \mathsf{RC}_1\wedge F(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } \mathsf{RC}_2\wedge\mathcal{K}\Big).$$

(MC6) $F(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } \mathsf{RC}_1+\mathsf{RC}_2\wedge\mathcal{K} \rightsquigarrow$

$$\Big(F(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } \mathsf{RC}_1\wedge\mathcal{K}\Big)\vee\Big(F(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } \mathsf{RC}_2\wedge\mathcal{K}\Big).$$

(MC7) $X_{\mathsf{c}}(C) \text{ in } \mathsf{RC}_1+\mathsf{RC}_2\wedge\mathcal{K} \rightsquigarrow \Big(X_{\mathsf{c}}(C) \text{ in } \mathsf{RC}_1\wedge\mathcal{K}\Big)\vee\Big(X_{\mathsf{c}}(C) \text{ in } \mathsf{RC}_2\wedge\mathcal{K}\Big)$.

(MC8) $X_{\mathsf{c}} \text{ in } \mathsf{RC}_1\wedge X_{\mathsf{c}} \text{ in } \mathsf{RC}_2 \rightsquigarrow X_{\mathsf{c}} \text{ in } \mathsf{RC}$,

where $[\![\mathsf{RC}]\!] = [\![\mathsf{RC}_1]\!]\cap[\![\mathsf{RC}_2]\!]$, and neither $X_{\mathsf{c}} \text{ in } \mathsf{RC}_1$ nor $X_{\mathsf{c}} \text{ in } \mathsf{RC}_2$ can be transformed by other rules.

The next two rules decompose both the structure of the context and the regular expression, where the regular expression has the form $f(\mathsf{RS}_1,\mathsf{RC},\mathsf{RS}_2)$. Note that we do not distinguish whether f is ordered or unordered. This set of rules corresponds the (MS9)–(MS11) rules for sequences.

(MC9) $f(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } f(\mathsf{RS}_1,\mathsf{RC},\mathsf{RS}_2)\wedge\mathcal{K} \rightsquigarrow f(\tilde{s}_1,\tilde{s}_2) \text{ in } f(\mathsf{RS}_1,\mathsf{RS}_2)\wedge\circ \text{ in } \mathsf{RC}\wedge\mathcal{K}$.

(MC10) $X_{\mathsf{f}}(\tilde{s}_1,\circ,\tilde{s}_2) \text{ in } f(\mathsf{RS}_1,\mathsf{RC},\mathsf{RS}_2)\wedge\mathcal{K} \rightsquigarrow$

$$X_{\mathsf{f}}\simeq f\wedge f(\tilde{s}_1\theta,\circ,\tilde{s}_2\theta) \text{ in } f(\mathsf{RS}_1,\mathsf{RC},\mathsf{RS}_2)\wedge\mathcal{K}\theta,$$

where $\theta = \{X_{\mathsf{f}}\mapsto f\}$.

We denote the set of rules (MS1)–(MS12) and (MC1)–(MC10) by Memb.

3.4.5 Decomposition Rules

Like the membership rules, each of the decomposition rules operates on a conjunction of constraint literals and gives back either a conjunction again, or a disjunction of conjunctions. These rules should be applied to disjuncts of constraints in DNF, to preserve the DNF structure.

(D1) $f_{\mathsf{o}}(\tilde{s}_1) \simeq f_{\mathsf{o}}(\tilde{s}_2) \rightsquigarrow \tilde{s}_1 \doteq \tilde{s}_2$.

(D2) $f_{\mathsf{u}}(\tilde{s}_1) \simeq f_{\mathsf{u}}(\tilde{s}_2) \wedge \mathcal{K} \rightsquigarrow \bigvee_{\tilde{s}' \in perm(\tilde{s}_2)} \big(\tilde{s}_1 \doteq \tilde{s}' \wedge \mathcal{K}\big)$,

where $\tilde{s}_2$ is a sequence of terms, $\tilde{s}_1$ and $\tilde{s}_2$ are disjoint.

(D3) $\lceil t_1, \tilde{s}_1 \rceil \simeq \lceil t_2, \tilde{s}_2 \rceil \rightsquigarrow t_1 \doteq t_2 \wedge \tilde{s}_1 \doteq \tilde{s}_2$,

where $\tilde{s}_1 \neq \lceil\,\rceil$ or $\tilde{s}_2 \neq \lceil\,\rceil$.

(D4) $f(\tilde{s}_1, C_1, \tilde{s}_2) \simeq f(\tilde{s}_3, C_2, \tilde{s}_4) \rightsquigarrow f(\tilde{s}_1, \tilde{s}_2) \doteq f(\tilde{s}_3, \tilde{s}_4) \wedge C_1 \doteq C_2$.

We denote the set of rules (D1)–(D4) by Dec.

3.4.6 Variable Elimination Rules

This set of rules eliminate variables from the given constraint, keeping only a single equation for them. The first four rules replace a variable with the corresponding expression, provided that the variable does not occur in the expression:

(E1) $X \simeq t \wedge \mathcal{K} \rightsquigarrow X \doteq t \wedge \mathcal{K}\theta$,

where $X \notin var(t)$, $X \in var(\mathcal{K})$ and $\theta = \{X \mapsto t\}$. If t is a variable then in addition it is required that $t \in var(\mathcal{K})$.

(E2) $\overline{x} \simeq \tilde{s} \wedge \mathcal{K} \rightsquigarrow \overline{x} \doteq \tilde{s} \wedge \mathcal{K}\theta$,

where $\overline{x} \notin var(\tilde{s})$, $\overline{x} \in var(\mathcal{K})$, and $\theta = \{\overline{x} \mapsto \tilde{s}\}$. If $\tilde{s} = \overline{y}$ for some $\overline{y}$, then in addition it is required that $\overline{y} \in var(\mathcal{K})$.

(E3) $\quad X_{\mathsf{c}} \simeq C \wedge \mathcal{K} \rightsquigarrow X_{\mathsf{c}} \doteq C \wedge \mathcal{K}\theta,$

where $X_{\mathsf{c}} \notin var(C)$, $X_{\mathsf{c}} \in var(\mathcal{K})$, and $\theta = \{X_{\mathsf{c}} \mapsto C\}$. If C has the form $Y_{\mathsf{c}}(\circ)$, then in addition it is required that $Y_{\mathsf{c}} \in var(\mathcal{K})$.

(E4) $\quad X_{\mathsf{f}} \simeq F \wedge \mathcal{K} \rightsquigarrow X_{\mathsf{f}} \doteq F \wedge \mathcal{K}\theta,$

where $X_{\mathsf{f}} \neq F$, $X_{\mathsf{f}} \in var(\mathcal{K})$, and $\theta = \{X_{\mathsf{f}} \mapsto F\}$. If F is a function variable, then in addition it is required that $F \in var(\mathcal{K})$.

The rules (E5) and (E6) for sequence variable elimination assign to a variable an initial part of the sequence in the other side of the selected equation. The sequence has to be a sequence of terms in (E5). In (E6), only a split of the prefix of the sequence is relevant. The rest is blocked by the term t due to occurrence check: No instantiation of $\overline{x}$ can contain it.

(E5) $\quad \lceil \overline{x}, \tilde{s}_1 \rceil \simeq \tilde{s}_2 \wedge \mathcal{K} \rightsquigarrow \bigvee_{\tilde{s}_2 = \lceil \tilde{s}', \tilde{s}'' \rceil} \Big(\overline{x} \doteq \tilde{s}' \wedge \tilde{s}_1\theta \doteq \tilde{s}'' \wedge \mathcal{K}\theta \Big)$

where $\tilde{s}_2$ is a sequence of terms, $\overline{x} \notin var(\tilde{s}_2)$, $\theta = \{\overline{x} \mapsto \tilde{s}'\}$, and $\tilde{s}_1 \neq \lceil\,\rceil$.

(E6) $\quad \lceil \overline{x}, \tilde{s}_1 \rceil \simeq \lceil \tilde{s}, t, \tilde{s}_2 \rceil \wedge \mathcal{K} \rightsquigarrow$
$$\bigvee_{\tilde{s} = \lceil \tilde{s}', \tilde{s}'' \rceil} \Big(\overline{x} \doteq \tilde{s}' \wedge \tilde{s}_1\theta \doteq (\tilde{s}'', t, \tilde{s}_2)\theta \wedge \mathcal{K}\theta \Big)$$

where $\tilde{s}$ is a sequence of terms, $\overline{x} \notin var(\tilde{s})$, $\overline{x} \in var(t)$, $\theta = \{\overline{x} \mapsto \tilde{s}'\}$, and $\tilde{s}_1 \neq \lceil\,\rceil$.

The rules (E7) and (E8) below can be seen as counterparts of (E5). In the rule (E8) we need conservative decomposition of contexts. Before giving those rules, we define the notion of conservativity.

We will speak about the *main path* of a context as the sequence of symbols (path) in its

tree representation from the root to the hole. For instance, the main path in the context $f(X_{\mathsf{c}1}(a), X_{\mathsf{f}}(X_{\mathsf{c}2}(b), g(\circ)), \overline{x})$ is $fX_{\mathsf{f}}g$, and in $f(X_{\mathsf{c}1}(a), X_{\mathsf{f}}(X_{\mathsf{c}2}(b), X_{\mathsf{c}3}(\circ)), \overline{x})$ – $fX_{\mathsf{f}}X_{\mathsf{c}3}$. A context is called *strict* if its main path does not contain context variables. For instance, the context $f(X_{\mathsf{c}1}(a), X_{\mathsf{f}}(X_{\mathsf{c}2}(b), g(\circ)), \overline{x})$ is strict, while $f(X_{\mathsf{c}1}(a), X_{\mathsf{f}}(X_{\mathsf{c}2}(b), X_{\mathsf{c}3}(\circ)), \overline{x})$ is not, because $X_{\mathsf{c}3}$ is in its main path $fX_{\mathsf{f}}X_{\mathsf{c}3}$. We say that a context C is *decomposed* in two contexts C_1 and C_2 if $C = C_1[C_2]$.

We say that a context C is *conservative*, if for any instance $C\rho$ of C and for any decomposition $D_1[D_2]$ of $C\rho$ there exists a decomposition $C_1[C_2]$ of C such that $D_1 = C_1\rho$ and $D_2 = C_2\rho$. Strict contexts satisfy this property. Non-strict contexts violate it, as the following example shows: The context $C = X_{\mathsf{c}}(\circ)$ has two decompositions into $C_1[C_2]$ with $C_1 = \circ$, $C_2 = X_{\mathsf{c}}(\circ)$ and $C_1 = X_{\mathsf{c}}(\circ)$, $C_2 = \circ$. Let $\rho = \{X_{\mathsf{c}} \mapsto f(g(\circ))\}$. Then $C\rho = f(g(\circ))$. One of its decomposition with $D_1 = f(\circ)$, $D_2 = g(\circ)$ is not an instance of any of the decompositions of C.

The rules (E7) and (E8) are formulated now as follows:

(E7) $\quad X_{\mathsf{c}}(t_1) \simeq t_2 \wedge \mathcal{K} \rightsquigarrow \vee_{t_2 = C[t]} \Big(X_{\mathsf{c}} \doteq C \wedge t_1\theta \doteq t \wedge \mathcal{K}\theta \Big),$

where t_2 does not contain individual, sequence, and context variables, $t_1 \neq \circ$, and $\theta = \{X_{\mathsf{c}} \mapsto C\}$.

(E8) $\quad X_{\mathsf{c}}(C_1) \simeq C_2 \wedge \mathcal{K} \rightsquigarrow \vee_{C_2 = C[C']} \Big(X_{\mathsf{c}} \doteq C \wedge C_1\theta \doteq C'\theta \wedge \mathcal{K}\theta \Big),$

where C_2 is strict, $X_{\mathsf{c}} \notin var(C)$, $C_1 \neq \circ$, and $\theta = \{X_{\mathsf{c}} \mapsto C\}$.

Finally, there are two rules for function variable elimination. Their behavior is standard:

(E9) $\quad X_{\mathsf{f}}(\tilde{s}_1) \simeq F(\tilde{s}_2) \wedge \mathcal{K} \rightsquigarrow X_{\mathsf{f}} \doteq F \wedge F(\tilde{s}_1)\theta \doteq F(\tilde{s}_2)\theta \wedge \mathcal{K}\theta.$

where $X_{\mathsf{f}} \neq F$, $\theta = \{X_{\mathsf{f}} \mapsto F\}$, and $\tilde{s}_1 \neq \lceil\,\rceil$ or $\tilde{s}_2 \neq \lceil\,\rceil$.

(E10) $\quad X_{\mathsf{f}}(\tilde{s}_1) \simeq X_{\mathsf{f}}(\tilde{s}_2) \wedge \mathcal{K} \rightsquigarrow \bigvee_{f \in \mathcal{F}} \Big(X_{\mathsf{f}} \doteq f \wedge f(\tilde{s}_1)\theta \doteq f(\tilde{s}_2)\theta \wedge \mathcal{K}\theta \Big),$

where $\theta = \{X_{\mathsf{f}} \mapsto f\}$, and $\tilde{s_1} \neq \tilde{s_2}$.

We denote the set of rules (E1)–(E10) by Elim.

3.5 Solved and Partially Solved Constraints

We say a variable is *solved* in a conjunction of primitive constraints $\mathcal{K} = \mathbf{c}_1 \wedge \cdots \wedge \mathbf{c}_n$, if there is a $\mathbf{c}_i$, $1 \leq i \leq n$, such that

- the variable is X, $\mathbf{c}_i$ is $X \doteq t$, and X occurs neither in t nor elsewhere in $\mathcal{K}$, or
- the variable is $\overline{x}$, $\mathbf{c}_i$ is $\overline{x} \doteq \tilde{s}$, and $\overline{x}$ occurs neither in $\tilde{s}$ nor elsewhere in $\mathcal{K}$, or
- the variable is X_{f}, $\mathbf{c}_i$ is $X_{\mathsf{f}} \doteq F$ and X_{f} occurs neither in F nor elsewhere in $\mathcal{K}$, or
- the variable is X_{c}, $\mathbf{c}_i$ is $X_{\mathsf{c}} \doteq C$, and X_{c} occurs neither in C nor elsewhere in $\mathcal{K}$, or
- the variable is X, $\mathbf{c}_i$ is X in $f(\mathsf{RS})$ and X does not occur in membership constraints elsewhere in $\mathcal{K}$, or
- the variable is $\overline{x}$, $\mathbf{c}_i$ is $\overline{x}$ in RS, $\overline{x}$ does not occur in membership constraints elsewhere in $\mathcal{K}$ and RS has the form $\mathsf{RS}_1.\mathsf{RS}_2$ or RS_1^*, or
- the variable is X_{c}, $\mathbf{c}_i$ is X_{c} in RC, X_{c} does not occur in membership constraints elsewhere in $\mathcal{K}$ and RC has the form $\mathsf{RC}_1 \cdot \mathsf{RC}_2$, RC_1^*, or $f(\mathsf{RS}_1, C, \mathsf{RS}_2)$.

In this case we also say that $\mathbf{c}_i$ is *solved in* $\mathcal{K}$. Moreover, $\mathcal{K}$ is called *solved* if for any $1 \leq i \leq n$, $\mathbf{c}_i$ is solved in it. $\mathcal{K}$ is *partially solved*, if for any $1 \leq i \leq n$, $\mathbf{c}_i$ is solved in $\mathcal{K}$, or has one of the following forms:

- Membership atom:
  - $f_{\mathsf{u}}(\tilde{s_1}, \overline{x}, \tilde{s_2})$ in $f_{\mathsf{u}}(\mathsf{RS})$.
  - $X_{\mathsf{c}}(t)$ in $f(\mathsf{RS})$.
  - $\lceil \overline{x}, \tilde{s} \rceil$ in RS where RS has a form $\mathsf{RS}_1.\mathsf{RS}_2$ or RS'^*.
  - $X_{\mathsf{c}}(C)$ in RC where RC has a form either $\mathsf{RC}_1 \cdot \mathsf{RC}_2$, $\mathsf{RC}_1^\star$, or $f(\mathsf{RS}_1, \mathsf{RC}', \mathsf{RS}_2)$, where $\mathsf{RC}' \neq \bullet$.

- Equation:

  - $\lceil \overline{x}, \tilde{s}_1 \rceil \doteq \lceil \overline{y}, \tilde{s}_2 \rceil$ where $\overline{x} \neq \overline{y}$, $\tilde{s}_1 \neq \lceil\,\rceil$ and $\tilde{s}_2 \neq \lceil\,\rceil$.
  - $\lceil \overline{x}, \tilde{s}_1 \rceil \doteq \lceil \tilde{s}, \overline{y}, \tilde{s}_2 \rceil$, where $\tilde{s}$ is a sequence of terms, $\overline{x} \notin var(\tilde{s})$, $\tilde{s}_1 \neq \lceil\,\rceil$, and $\tilde{s} \neq \lceil\,\rceil$. The variables $\overline{x}$ and $\overline{y}$ are not necessarily distinct.
  - $f_{\mathsf{u}}(\tilde{s}_1, \overline{x}, \tilde{s}_2) \doteq f_{\mathsf{u}}(\tilde{s}_3, \overline{y}, \tilde{s}_4)$ where $\lceil \tilde{s}_1, \overline{x}, \tilde{s}_2 \rceil$ and $\lceil \tilde{s}_3, \overline{y}, \tilde{s}_4 \rceil$ are disjoint.
  - $X_{\mathsf{c}}(t) \doteq r$ where $r \neq X_{\mathsf{c}}(t')$ contains individual, context or sequence variables,
  - $X_{\mathsf{c}}(C_1) \doteq C_2$ where $C_2 \neq X_{\mathsf{c}}(C_3)$ and C_2 is not strict.

A constraint is *solved*, if it is either true or a non-empty quantifier-free disjunction of solved conjunctions. A constraint is *partially solved*, if it is either true or a non-empty quantifier-free disjunction of partially solved conjunctions.

3.6 The Algorithm

In this section we present an algorithm that converts a constraint with respect to rules specified in the Section3.4 into a partially solved one. First, we define the rewrite step

step := first(Log, Fail, Del, Dec, Elim, Memb).

When applied to a constraint, step transforms it by the *first* applicable rule of the solver, looking successively into the sets Log, Fail, Del, Dec, Elim, and Memb.

The constraint solving algorithm implements the strategy solve which is defined as a repeatedly application of the step:

solve := NF(step).

That means, step is applied to a constraint repeatedly as long as possible. It remains to show that this definition yields an algorithm, which amounts to proving that a normal form is reached by NF(step) for any constraint $\mathcal{C}$.

3.7 Properties of the Constraint Solver

Termination. We prove that the solver halts for any input constraint.

Theorem 3.2. solve *terminates on any input constraint.*

Proof. We define a *complexity measure* $cm(\mathcal{C})$ for quantifier-free constraints in DNF, and show that $cm(\mathcal{C}') < cm(\mathcal{C})$ holds whenever $\mathcal{C}' = \mathsf{step}(\mathcal{C})$.

For a sequence $\tilde{s}$ and for a context C (resp. regular sequence expression RS and regular context expression RC), we denote by $size(\tilde{s})$ and by $size(C)$ (resp. by $size(\mathsf{RS})$ and by $size(\mathsf{RC})$) its denotational length, e.g., $size(\mathsf{eps}) = 1$, $size(\circ) = 1$, $size(X_{\mathsf{c}}(f(a)), X_{\mathsf{f}}(\overline{x})) = 5$, $size(X_{\mathsf{f}}(f_{\mathsf{o}}(a), f(\circ), b, \overline{x})) = 7$, $size(f(f(a.b^*))) = 6$, and $size(f(a, f(\bullet, b)^\star, a|b)) = 9$. The complexity measure $cm(\mathcal{K})$ of a conjunction of primitive constraints $\mathcal{K}$ is the tuple $\langle N_1, M_1, N_2, M_2, M_3\rangle$ defined as follows ($\{\!|\}$ stands for a multiset):

- N_1 is the number of unsolved variables in $\mathcal{K}$.
- $M_1 := \{\!|size(\tilde{s}) \mid \tilde{s} \text{ in } \mathsf{RS} \in \mathcal{K}, \tilde{s} \neq \lceil\,\rceil|\!\} \uplus \{\!|size(C) \mid C \text{ in } \mathsf{RC} \in \mathcal{K}, C \neq \circ|\!\}$.
- N_2 is the number of primitive constraints in the form $\overline{x}$ in RS in $\mathcal{K}$.
- $M_2 := \{\!|size(\mathsf{RS})) \mid \tilde{s} \text{ in } \mathsf{RS} \in \mathcal{K}|\!\} \uplus \{\!|size(\mathsf{RC}) \mid C \text{ in } \mathsf{RC} \in \mathcal{K}|\!\}$.
- $M_3 := \{\!|size(t_1) + size(t_2) \mid t_1 \doteq t_2 \in \mathcal{K}|\!\} \uplus \{\!|size(C_1) + size(C_2) \mid C_1 \doteq C_2 \in \mathcal{K}|\!\}$.

Where $\uplus$ stands for multiset union.

The complexity measure $cm(\mathcal{C})$ of a constraint $\mathcal{C} = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$ is defined as a multiset $\{\!|cm(\mathcal{K}_1), \ldots, cm(\mathcal{K}_n)|\!\}$. Measures are compared by multiset extension of the lexicographic ordering on tuples. The Log rules strictly reduce the measure. For the other rules, the table below shows which rule reduces which component of the measure, which implies termination of the algorithm solve.

Rule	N_1	M_1	N_2	M_2	M_3
(E1)-(E10),(MS1),(MS9),(MC1),(MC10)	$>$				
(F10)-(F13),(MS2),(MS3),(MS8),(MS10),(MS11),(MC2),(MC3),(MC8),(MC9)	$\geq$	$>$			
(MS12)	$\geq$	$\geq$	$>$		
(F8),(F9),(MS4)-(MS7),(MC4)-(MC7)	$\geq$	$\geq$	$\geq$	$>$	
(F1)-(F7),Dec,Del	$\geq$	$\geq$	$\geq$	$\geq$	$>$

□

Soundness and Partial Completeness. Here we show that the solver reduces a constraint to its equivalent constraint.

Lemma 3.3. *If* $\mathsf{step}(\mathcal{C}) = \mathcal{D}$*, then* $\mathfrak{I} \models \forall\Big(\mathcal{C} \Leftrightarrow \overline{\exists}_{var(\mathcal{C})}\mathcal{D}\Big)$ *for all intended structures* $\mathfrak{I}$*.*

Proof. By case distinction on the inference rules of the solver, selected by the strategy first in the application of step. We illustrate here two cases, when the selected rules are (E5) and (MS2). For other rules the lemma can be shown in a similar manner. In (E5), $\mathcal{C}$ has a disjunct $\mathcal{K} = (\overline{x}, \tilde{s}_1) \doteq \tilde{s}_2 \wedge \mathcal{K}'$ where $\tilde{s}_2$ is a sequence of terms, $\overline{x} \notin var(\tilde{s}_2)$, and $\mathcal{D}$ is the result of replacing $\mathcal{K}$ in $\mathcal{C}$ with the disjunction $\mathcal{C}' = \bigvee_{\tilde{s}_2=(\tilde{s}',\tilde{s}'')}(\overline{x} \doteq \tilde{s}' \wedge \tilde{s}_1\theta \doteq \tilde{s}'' \wedge \mathcal{K}'\theta)$ where $\theta = \{\overline{x} \mapsto \tilde{s}'\}$. Therefore, it is sufficient to show that $\mathfrak{I} \models \forall(\mathcal{K} \Leftrightarrow \overline{\exists}_{var(\mathcal{C})}\mathcal{C}')$. Since $var(\mathcal{C}') = var(\mathcal{K})$, this amounts to show that for all ground substitutions ρ of $var(\mathcal{K})$ we have $\mathfrak{I} \models (\overline{x}\rho, \tilde{s}_1\rho) \doteq \tilde{s}_2\rho \wedge \mathcal{K}'\rho$ iff $\mathfrak{I} \models (\bigvee_{\tilde{s}_2=(\tilde{s}',\tilde{s}'')}(\overline{x} \doteq \tilde{s}' \wedge \tilde{s}_1\theta \doteq \tilde{s}'' \wedge \mathcal{K}'\theta))\rho$.

- Assume $\mathfrak{I} \models (\overline{x}\rho, \tilde{s}_1\rho) \doteq \tilde{s}_2\rho \wedge \mathcal{K}'\rho$. We can split $\tilde{s}_2\rho$ into $\tilde{s}'\rho$ and $\tilde{s}''\rho$ such that $\overline{x}\rho = \tilde{s}'\rho$ and $\tilde{s}_1\rho = \tilde{s}''\rho$. Now, we show $v\theta\rho = v\rho$ for all $v \in var(\overline{x}, \tilde{s}_1, \tilde{s}_2)$. Indeed, if $v \neq \overline{x}$, the equality trivially holds. If $v = \overline{x}$, we have $\overline{x}\theta\rho = \tilde{s}'\rho = \overline{x}\rho$. Hence, $\mathfrak{I} \models (\bigvee_{\tilde{s}_2=(\tilde{s}',\tilde{s}'')}(\overline{x} \doteq \tilde{s}' \wedge \tilde{s}_1\theta \doteq \tilde{s}'' \wedge \mathcal{K}'\theta))\rho$.
- Assume $\mathfrak{I} \models (\bigvee_{\tilde{s}_2=(\tilde{s}',\tilde{s}'')}(\overline{x} \doteq \tilde{s}' \wedge \tilde{s}_1\theta \doteq \tilde{s}'' \wedge \mathcal{K}'\theta))\rho$. Then there exists the split $\tilde{s}_2 = (\tilde{s}', \tilde{s}'')$ such that $\mathfrak{I} \models (\overline{x}\rho \doteq \tilde{s}'\rho \wedge \tilde{s}_1\theta\rho \doteq \tilde{s}''\rho \wedge \mathcal{K}'\theta\rho)$. Again, we can show $v\theta\rho = v\rho$ for all $v \in var(\overline{x}, \tilde{s}_1, \tilde{s}_2)$. Hence, $\mathfrak{I} \models (\overline{x}\rho, \tilde{s}_1\rho) = \tilde{s}_2\rho \wedge \mathcal{K}'\rho$.

Now, let the selected rule be (MS2). $\mathcal{C}$ has a disjunct $\mathcal{K} = (t, \tilde{s})$ in $\mathsf{RS} \wedge \mathcal{K}'$ with $\tilde{s} \neq \lceil\rceil$ and $\mathsf{RS} \neq \mathsf{eps}$. Then $\mathcal{D}$ is the result of replacing $\mathcal{K}$ in $\mathcal{C}$ with $\mathcal{C}' = \bigvee_{(f(\mathsf{RS}_1),\mathsf{RS}_2)\in lf_s(\mathsf{RS})}(t$ in $f(\mathsf{RS}_1) \wedge \tilde{s}$ in $\mathsf{RS}_2 \wedge \mathcal{K}')$. Therefore, to show $\mathfrak{I} \models \forall(\mathcal{C} \Leftrightarrow \overline{\exists}_{var(\mathcal{C})}\mathcal{D})$, it is enough to show that $\mathfrak{I} \models \forall(\mathcal{K} \Leftrightarrow \overline{\exists}_{var(\mathcal{C})}\mathcal{C}')$. Since $var(\mathcal{C}') = var(\mathcal{K})$, this amounts to showing that for all ground substitutions ρ of $var(\mathcal{K})$ we have $\mathfrak{I} \models (t\rho, \tilde{s}\rho)$ in $\mathsf{RS} \wedge \mathcal{K}'\rho$ iff $\mathfrak{I} \models (\bigvee_{(f(\mathsf{RS}_1),\mathsf{RS}_2)\in lf_s(\mathsf{RS})}(t$ in $f(\mathsf{RS}_1) \wedge \tilde{s}$ in $\mathsf{RS}_2 \wedge \mathcal{K}'))\rho$.

- Assume $\mathfrak{I} \models (t\rho, \tilde{s}\rho)$ in $\mathsf{RS} \wedge \mathcal{K}'\rho$. Proposition 5 of [Ant96] can be easily extended for regular sequence expressions, obtaining the following statement: For all RS, $[\![\mathsf{RS}]\!] = o(\mathsf{RS}) \cup [\![lf_s(\mathsf{RS})]\!]$, where $[\![lf_s(\mathsf{RS})]\!] = \bigcup_{\langle f(\mathsf{RS}_1),\mathsf{RS}_2\rangle\in lf_s(\mathsf{RS})}[\![f(\mathsf{RS}_1).\mathsf{RS}_2]\!]$, and $o(\mathsf{RS})$ is defined as follows: If $\lceil\rceil \in [\![\mathsf{RS}]\!]$, then $o(\mathsf{RS}) = \{\lceil\rceil\}$, otherwise $o(\mathsf{RS}) = \emptyset$. Then $\mathfrak{I} \models (t\rho, \tilde{s}\rho)$ in $\mathsf{RS} \wedge \mathcal{K}'\rho$ implies $\mathfrak{I} \models (t\rho, \tilde{s}\rho)$ in $lf_s(\mathsf{RS}) \wedge \mathcal{K}'\rho$ by the definitions of intended structures and entailment. Hence, we can conclude $\mathfrak{I} \models (\bigvee_{(f(\mathsf{RS}_1),\mathsf{RS}_2)\in lf_s(\mathsf{RS})}(t\rho$ in $f(\mathsf{RS}_1) \wedge \tilde{s}\rho$ in $\mathsf{RS}_2 \wedge \mathcal{K}'\rho))$.

- Assume $\mathfrak{I} \models (\bigvee_{(f(\mathsf{RS}_1),\mathsf{RS}_2)\in lf_s(\mathsf{RS})}(t\rho \text{ in } f(\mathsf{RS}_1) \wedge \tilde{s}\rho \text{ in } \mathsf{RS}_2 \wedge \mathcal{K}'\rho))$. Then we have $\mathfrak{I} \models (t\rho, \tilde{s}\rho) \text{ in } lf_s(\mathsf{RS}) \wedge \mathcal{K}'\rho$. By the extended version of Proposition 5 of [Ant96], stated in the previous item, we can conclude $\mathfrak{I} \models (t\rho, \tilde{s}\rho) \text{ in } \mathsf{RS} \wedge \mathcal{K}'\rho$.

□

Theorem 3.4. *If* $\mathsf{solve}(\mathcal{C}) = \mathcal{D}$, *then* $\mathfrak{I} \models \forall\Big(\mathcal{C} \Leftrightarrow \overline{\exists}_{var(\mathcal{C})}\mathcal{D}\Big)$ *for all intended structures* $\mathfrak{I}$, *and* $\mathcal{D}$ *is either partially solved or the* false *constraint.*

Proof. $\mathfrak{I} \models \forall\big(\mathcal{C} \Leftrightarrow \overline{\exists}_{var(\mathcal{C})}\mathcal{D}\big)$ follows from Lemma 3.3 and the following property: If $\mathfrak{I} \models \forall\big(\mathcal{C}_1 \Leftrightarrow \overline{\exists}_{var(\mathcal{C}_1)}\mathcal{C}_2\big)$ and $\mathfrak{I} \models \forall\big(\mathcal{C}_2 \Leftrightarrow \overline{\exists}_{var(\mathcal{C}_2)}\mathcal{C}_3\big)$, then $\mathfrak{I} \models \forall\big(\mathcal{C}_1 \Leftrightarrow \overline{\exists}_{var(\mathcal{C}_1)}\mathcal{C}_3\big)$. The property itself relies on the fact that $\mathfrak{I} \models \forall\big(\overline{\exists}_{var(\mathcal{C}_1)}\overline{\exists}_{var(\mathcal{C}_2)}\mathcal{C}_3 \Leftrightarrow \overline{\exists}_{var(\mathcal{C}_1)}\mathcal{C}_3\big)$, which holds because all variables introduced by the rules of the solver in $\mathcal{C}_3$ are fresh not only for $\mathcal{C}_2$, but also for $\mathcal{C}_1$.

As for the partially solved constraint, by the definition of solve and Theorem 3.2, $\mathcal{D}$ is in a normal form. Assume by contradiction that it is not partially solved. By inspection of the solver rules, based on the definition of partially solved constraints, we can see that there is a rule that applies to $\mathcal{D}$. But this contradicts the fact that $\mathcal{D}$ is in a normal form. Hence, $\mathcal{D}$ is partially solved. □

Theorem 3.5. *If the constraint* $\mathcal{D}$ *is solved, then* $\mathfrak{I} \models \exists\mathcal{D}$ *for all intended structures* $\mathfrak{I}$.

Proof. Since $\mathcal{D}$ is solved, each disjunct $\mathcal{K}$ in it has a form $v_1 \doteq e_1 \wedge \cdots \wedge v_n \doteq e_n \wedge v'_1 \text{ in } \mathsf{R}_1 \wedge \cdots \wedge v'_m \text{ in } \mathsf{R}_m$ where $m, n \geq 0$, $v_i, v'_j \in \mathcal{V}_{\mathsf{ISFC}}$ and e_i is an expression corresponding to v_i. Moreover, $v_1, \ldots, v_n, v'_1, \ldots, v'_m$ are distinct and $[\![\mathsf{R}_j]\!] \neq \emptyset$ for all $1 \leq j \leq m$. Assume ρ'_i is a grounding substitution for e_i for all $1 \leq i \leq n$, and let e'_j be an element of $[\![\mathsf{R}_j]\!]$ for all $1 \leq j \leq m$. Then $\rho = \{v_1 \mapsto e_1\rho'_1, \ldots, v_n \mapsto e_n\rho'_n, v'_1 \mapsto e'_1, \ldots, v'_m \mapsto e'_m\}$ solves $\mathcal{K}$. Therefore, $\mathfrak{I} \models \exists\mathcal{D}$ holds. □

3.8 Solving Constraints in Special Forms

As we have seen in the previous section, not all the constraints can be reduced to a solved form or to false. Therefore, it is interesting to find certain special cases, for which our solver is complete. This is the problem we address in this section, considering two such fragments: well-moded and KIF constraints.

3.8.1 Well-Moded Constraints

First, we define the notion of *well-moded*[DFKM14, KM06] sequence of primitive constraints. Later this notion will be generalized to sequences of arbitrary literals and will be used in the context of constraint logic programming.

A sequence of primitive constraints $\mathbf{c}_1, \ldots, \mathbf{c}_n$ is *well-moded* if the following conditions are satisfied:

a) If for some $1 \leq i \leq n$, $\mathbf{c}_i$ is $t_1 \doteq t_2$, then $var(t_1) \subseteq \bigcup_{j=1}^{i-1} var(\mathbf{c}_j)$ or $var(t_2) \subseteq \bigcup_{j=1}^{i-1} var(\mathbf{c}_j)$.

b) If for some $1 \leq i \leq n$, $\mathbf{c}_i$ is $C_1 \doteq C_2$, then $var(C_1) \subseteq \bigcup_{j=1}^{i-1} var(\mathbf{c}_j)$ or $var(C_2) \subseteq \bigcup_{j=1}^{i-1} var(\mathbf{c}_j)$.

c) If for some $1 \leq i \leq n$, $\mathbf{c}_i$ is a membership primitive constraint, then the inclusion $var(\mathbf{c}_i) \subseteq \bigcup_{j=1}^{i-1} var(\mathbf{c}_j)$ holds.

A conjunction of primitive constraints $\mathcal{K}$ is well-moded if there exists a sequence of primitive constraints $\mathbf{c}_1, \ldots, \mathbf{c}_n$ which is well-moded and $\mathcal{K} = \bigwedge_{i=1}^{n} \mathbf{c}_i$ modulo associativity and commutativity of $\wedge$. A constraint $\mathcal{C} = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$ is well-moded if each $\mathcal{K}_i$, $1 \leq i \leq n$, is well-moded.

A remarkable property of well-moded constraints is that the solver can bring them to a solved form or to false. To show it, we first need to prove the following lemma:

Lemma 3.6. *Let $\mathcal{C}$ be a well-moded constraint and* $\mathsf{step}(\mathcal{C}) = \mathcal{C}'$*, then $\mathcal{C}'$ is either well-moded,* true *or* false.

Proof. Let $\mathcal{C} = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$ be a well-moded constraint. By Theorem 3.4, $\mathcal{C}'$ is either false or a partially solved constraint (which can be also true, in particular). Assume $\mathcal{C}'$ is neither true nor false. By the definition of well-modedness, each $\mathcal{K}_i$, $1 \leq i \leq n$, is well-moded. We have to show that well-modedness is preserved by each rule of the solver used to perform the step. In fact, the only nontrivial cases are transformations by the variable elimination rules.

For illustration, assume $\mathcal{K}_i = \lceil \overline{x}, \tilde{s}_1 \rceil \doteq \tilde{s}_2 \wedge \mathcal{K}_i'$, where $\tilde{s}_2$ is a nonempty sequence of terms, $\overline{x} \notin var(\tilde{s}_2)$, and we want to show that $\mathsf{step}(\mathcal{K}_i)$ is well-moded. Then the step is performed

by the (E5) rule: $\mathsf{step}(\lceil \overline{x}, \tilde{s}_1 \rceil \doteq \tilde{s}_2 \wedge \mathcal{K}'_i) = \bigvee_{\tilde{s}_2 = \lceil \tilde{s}', \tilde{s}'' \rceil} (\overline{x} \doteq \tilde{s}' \wedge \tilde{s}_1\theta \doteq \tilde{s}'' \wedge \mathcal{K}'_i\theta)$, where $\theta = \{\overline{x} \mapsto \tilde{s}'\}$.

By the definition of well-modedness, we can assume $\mathbf{c}_1, \ldots, \mathbf{c}_{j-1}, \lceil \overline{x}, \tilde{s}_1 \rceil \doteq \tilde{s}_2, \mathbf{c}_{j+1}, \ldots, \mathbf{c}_n$ is a well-moded sequence of primitive constraints, obtained by permutation of primitive constraints taken from $\lceil \overline{x}, \tilde{s}_1 \rceil \doteq \tilde{s}_2 \wedge \mathcal{K}'_i$. For brevity, let $\tilde{\mathbf{c}}_1$ stand for the sequence $\mathbf{c}_1, \ldots, \mathbf{c}_{j-1}$ and $\tilde{\mathbf{c}}_2$ for $\mathbf{c}_{j+1}, \ldots, \mathbf{c}_n$. Then well-modedness of $\tilde{\mathbf{c}}_1, \lceil \overline{x}, \tilde{s}_1 \rceil \doteq \tilde{s}_2, \tilde{\mathbf{c}}_2$ implies that we have $var(\lceil \overline{x}, \tilde{s}_1 \rceil) \subseteq var(\tilde{\mathbf{c}}_1)$ or $var(\tilde{s}_2) \subseteq var(\tilde{\mathbf{c}}_1)$. We will show that the sequence $\tilde{\mathbf{c}}_1\theta, \overline{x} \doteq \tilde{s}', \tilde{s}_1\theta \doteq \tilde{s}'', \tilde{\mathbf{c}}_2\theta$ is well-moded.

First, assume $var(\lceil \overline{x}, \tilde{s}_1 \rceil) \subseteq var(\tilde{\mathbf{c}}_1)$. It implies that $\tilde{\mathbf{c}}_1$ is not empty. Since $\overline{x} \notin var(\tilde{s}_2)$, $\tilde{s}_2 = \lceil \tilde{s}', \tilde{s}'' \rceil$, and $\theta = \{\overline{x} \mapsto \tilde{s}'\}$, we have $var(\tilde{s}') \subseteq var(\tilde{\mathbf{c}}_1\theta)$. Also, $var(\tilde{s}_1\theta) = var(\tilde{s}_1) \cup var(\tilde{s}') \setminus \{\overline{x}\} \subseteq var(\tilde{\mathbf{c}}_1\theta)$ and $var(\tilde{\mathbf{c}}_2\theta) = var(\tilde{\mathbf{c}}_2) \cup var(\tilde{s}') \setminus \{\overline{x}\}$. Therefore, we have that $\tilde{\mathbf{c}}_1\theta, \overline{x} \doteq \tilde{s}', \tilde{s}_1\theta \doteq \tilde{s}'', \tilde{\mathbf{c}}_2\theta$ is well-moded in this case.

Now assume $var(\tilde{s}_2) \subseteq var(\tilde{\mathbf{c}}_1)$. Then we have $var(\tilde{s}_2) \subseteq var(\tilde{\mathbf{c}}_1\theta)$, because $var(\tilde{\mathbf{c}}_1\theta) = var(\tilde{\mathbf{c}}_1) \setminus \{x\}$ and $x \notin var(\tilde{s}_2)$. Then we get $var(\tilde{s}') \subseteq var(\tilde{\mathbf{c}}_1\theta)$ and $var(\tilde{s}'') \subseteq var(\tilde{\mathbf{c}}_1\theta)$, which implies well-modedness of $\tilde{\mathbf{c}}_1\theta, \overline{x} \doteq \tilde{s}', \tilde{s}_1\theta \doteq \tilde{s}'', \tilde{\mathbf{c}}_2\theta$ also in this case.

Note that the sequence $\tilde{\mathbf{c}}_1\theta, \overline{x} \doteq \tilde{s}', \tilde{s}_1\theta \doteq \tilde{s}'', \tilde{\mathbf{c}}_2\theta$ is a permutation of a disjunct of constraints taken from $\mathsf{step}(\mathcal{K}_i)$, as we obtained it by the rule (E5). Since we considered an arbitrary split of $\tilde{s}_2$, we actually proved that $\mathsf{step}(\mathcal{K}_i)$ is well-moded.

The reasoning is similar, when the selected primitive constraint has the form on which some other rule of the solver works. We do not consider each of those cases here.

Hence, from the well-modedness of $\mathcal{K}_i$ we proved the well-modedness of $\mathsf{step}(\mathcal{K}_i)$ (when it is neither true nor false). From this, we conclude that the well-modedness of $\mathcal{C}$ implies the well-modedness of $\mathsf{step}(\mathcal{C})$, when the latter is neither true nor false. □

Theorem 3.7. *Let $\mathcal{C}$ be a well-moded constraint and $\mathsf{solve}(\mathcal{C}) = \mathcal{C}'$, where $\mathcal{C}' \neq \mathsf{false}$. Then $\mathcal{C}'$ is solved.*

Proof. From Lemma 3.6, by simple induction we get that if $\mathcal{C}' \neq \mathsf{false}$, then it is either true or well-moded. true is already solved. Consider the case when $\mathcal{C}'$ is well-moded. Let $\mathcal{C}' = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_m$. Since $\mathcal{C}' \neq \mathsf{false}$, by Theorem 3.4 $\mathcal{C}'$ is partially solved. So, each $\mathcal{K}_j$, $1 \leq j \leq m$, is partially solved and well-moded. By definition, $\mathcal{K}_j$ is well-moded if there exists a permutation of its literals $\mathbf{c}_1, \ldots, \mathbf{c}_i, \ldots, \mathbf{c}_n$ which satisfies the well-modedness property.

Assume $\mathbf{c}_1, \ldots, \mathbf{c}_{i-1}$ are solved. By this assumption and the definition of well-modedness, each of $\mathbf{c}_1, \ldots, \mathbf{c}_{i-1}$ is an equation whose one side is a variable that occurs neither in its other side nor in any other primitive constraint. Then well-modedness of $\mathcal{K}_j$ guarantees that the other side of these equations are ground terms. Assume by contradiction that $\mathbf{c}_i$ is partially solved, but not solved. If $\mathbf{c}_i$ is a membership constraint, well-modedness of $\mathcal{K}_j$ implies that $\mathbf{c}_i$ does not contain variables and, therefore, can not be partially solved. Now let $\mathbf{c}_i$ be an equation. Since all variables in $\mathbf{c}_1, \ldots, \mathbf{c}_{i-1}$ are solved, they can not appear in $\mathbf{c}_i$. From this fact and well-modedness of $\mathcal{K}_j$, $\mathbf{c}_i$ should have at least one ground side. But then it can not be partially solved. The obtained contradiction shows that $\mathcal{C}'$ is solved. □

Example 3.8. *Let* $\mathcal{C} = X_{\mathsf{c}}(g_{\mathsf{u}}(\overline{x}, \overline{y})) \doteq f_{\mathsf{o}}(b, g_{\mathsf{u}}(a, g_{\mathsf{u}})) \wedge X_{\mathsf{c}}$ in $f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x}$ in a^*. *Then* solve *performs the following derivation (the subscript* $\{\mathsf{Rule}_1, \ldots, \mathsf{Rule}_n\}$ *of* $\rightsquigarrow$ *specifies which set of rules are applied in the transformation):*

$$
\begin{aligned}
\mathcal{C} \rightsquigarrow_{\{\mathsf{Elim}\}} & X_{\mathsf{c}} \doteq \circ \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = f_{\mathsf{o}}(b, g_{\mathsf{u}}(a, g_{\mathsf{u}})) \wedge X_{\mathsf{c}} \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \vee \\
& X_{\mathsf{c}} \doteq f_{\mathsf{o}}(\circ, g_{\mathsf{u}}(a, g_{\mathsf{u}})) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = b \wedge X_{\mathsf{c}} \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \vee \\
& X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = g_{\mathsf{u}}(a, g_{\mathsf{u}}) \wedge X_{\mathsf{c}} \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \vee \\
& X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, g_{\mathsf{u}}(\circ, g_{\mathsf{u}})) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = a \wedge X_{\mathsf{c}} \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \vee \\
& X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, g_{\mathsf{u}}(a, \circ)) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = g_{\mathsf{u}} \wedge X_{\mathsf{c}} \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \\
\rightsquigarrow_{\{\mathsf{Fail}\}} & X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = g_{\mathsf{u}}(a, g_{\mathsf{u}}) \wedge X_{\mathsf{c}} \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \vee \\
& X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, g_{\mathsf{u}}(a, \circ)) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = g_{\mathsf{u}} \wedge X_{\mathsf{c}} \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \\
\rightsquigarrow_{\{\mathsf{Elim}\}} & X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = g_{\mathsf{u}}(a, g_{\mathsf{u}}) \wedge f_{\mathsf{o}}(b, \circ) \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \vee \\
& X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, g_{\mathsf{u}}(a, \circ)) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = g_{\mathsf{u}} \wedge \\
& \qquad f_{\mathsf{o}}(b, g_{\mathsf{u}}(a, \circ)) \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \\
\rightsquigarrow_{\{\mathsf{Memb,Log}\}} & X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = g_{\mathsf{u}}(a, g_{\mathsf{u}}) \wedge \overline{x} \text{ in } a^* \vee \\
& X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, g_{\mathsf{u}}(a, \circ)) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = g_{\mathsf{u}} \wedge \\
& \qquad f_{\mathsf{o}}(b, g_{\mathsf{u}}(a, \circ)) \text{ in } f_{\mathsf{o}}(b^*, \bullet) \wedge \overline{x} \text{ in } a^* \\
\rightsquigarrow_{\{\mathsf{Memb,Fail}\}} & X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge g_{\mathsf{u}}(\overline{x}, \overline{y}) = g_{\mathsf{u}}(a, g_{\mathsf{u}}) \wedge \overline{x} \text{ in } a^* \\
\rightsquigarrow_{\{\mathsf{Dec}\}} & X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \lceil \overline{x}, \overline{y} \rceil = \lceil a, g_{\mathsf{u}} \rceil \wedge \overline{x} \text{ in } a^* \vee \\
& X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \lceil \overline{x}, \overline{y} \rceil = \lceil g_{\mathsf{u}}, a \rceil \wedge \overline{x} \text{ in } a^* \\
\rightsquigarrow_{\{\mathsf{Elim}\}} & X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \overline{x} = \lceil \, \rceil \wedge \overline{y} = \lceil a, g_{\mathsf{u}} \rceil \wedge \overline{x} \text{ in } a^* \vee \\
& X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \overline{x} = a \wedge \overline{y} = g_{\mathsf{u}} \wedge \overline{x} \text{ in } a^* \vee
\end{aligned}
$$

$$X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \overline{x} = \lceil a, g_{\mathsf{u}} \rceil \wedge \overline{y} = \lceil\,\rceil \wedge \overline{x} \text{ in } a^*\vee$$
$$X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \overline{x} = \lceil\,\rceil \wedge \overline{y} = \lceil g_{\mathsf{u}}, a \rceil \wedge \overline{x} \text{ in } a^*\vee$$
$$X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \overline{x} = g_{\mathsf{u}} \wedge \overline{y} = a \wedge \overline{x} \text{ in } a^*\vee$$
$$X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \overline{x} = \lceil g_{\mathsf{u}}, a \rceil \wedge \overline{y} = \lceil\,\rceil \wedge \overline{x} \text{ in } a^*$$
$$\rightsquigarrow_{\{\mathsf{Elim,Memb,Fail,Log}\}} X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \overline{x} = \lceil\,\rceil \wedge \overline{y} = \lceil a, g_{\mathsf{u}} \rceil\vee$$
$$X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \overline{x} = a \wedge \overline{y} = g_{\mathsf{u}}\vee$$
$$X_{\mathsf{c}} \doteq f_{\mathsf{o}}(b, \circ) \wedge \overline{x} = \lceil\,\rceil \wedge \overline{y} = \lceil g_{\mathsf{u}}, a \rceil$$

The obtained constraint is solved.

3.8.2 Constraints in the form of Knowledge Interchange Format (KIF)

A simple term is in the *KIF form* (*KIF-term*) if it does not contain context variables, sequence variables occur only below ordered function symbols,* and they occupy only the last argument position in each subterm where they appear. For example, the term $f_{\mathsf{o}}(X, f_{\mathsf{o}}(a, \overline{x}), f_{\mathsf{u}}(X, b), \overline{x})$ is in the KIF form, while $f_{\mathsf{o}}(\overline{x}, a, \overline{x})$, $f_{\mathsf{u}}(X, f_{\mathsf{o}}(a, \overline{x}), f_{\mathsf{u}}(X, b), \overline{x})$, and $f_{\mathsf{o}}(X_{\mathsf{c}}(X), \overline{x})$ are not. This fragment originates the language of Knowledge Interchange Format [Gen98], where sequence variables are subject of this restriction.

A sequence $(\tilde{s}, \overline{t})$ is in the KIF form, if $\tilde{s}$ is a sequence of KIF-terms and $\overline{t}$ is either a KIF-term or a sequence variable. A primitive constraint $t_1 \doteq t_2$ is in the KIF-form if t_1 and t_2 are in the KIF-form, and a membership atom $\tilde{s}$ in RS is in the KIF-form, if $\tilde{s}$ is a KIF-sequence. The notion of KIF form extends naturally to constraints and states, requiring their literals should be in the KIF form.

Theorem 3.9. *Let $\mathcal{C}$ be a KIF-constraint and* solve$(\mathcal{C}) = \mathcal{C}'$*, where $\mathcal{C}' \neq$* false*. Then $\mathcal{C}'$ is solved.*

Proof. Note that for KIF-constraints, solve does not use rules that transform context equations and context membership constraints. solve also never uses the rules (E5), (E6), and (MS3). As for the other ones, it is easy to see that one of them necessarily applies to a constraint which is not solved. Hence, at the end we either get false or a solved constraint. □

*If the language does not contain unordered function symbols, then sequence variables are permitted under function variables as well.

We illustrate how to solve a simple KIF constraint:

Example 3.10. *Let* $\mathcal{C} = f_{\mathsf{o}}(X, \overline{x}) \doteq f_{\mathsf{o}}(g_{\mathsf{o}}(\overline{y}), a, \overline{y}) \wedge \overline{x}$ in $a^* \wedge \overline{y}$ in $a.a(b^*)^*$. *Then* solve *performs the following derivation:*

$$\begin{aligned}
\mathcal{C} &\rightsquigarrow_{\{\mathsf{Dec},\mathsf{Elim}\}} X \doteq g_{\mathsf{o}}(\overline{y}) \wedge \overline{x} \doteq \lceil a, \overline{y} \rceil \wedge \lceil a, \overline{y} \rceil \text{ in } a^* \wedge \overline{y} \text{ in } a.a(b^*)^* \\
&\rightsquigarrow_{\{\mathsf{Memb},\mathsf{Log}\}} X \doteq g_{\mathsf{o}}(\overline{y}) \wedge \overline{x} \doteq \lceil a, \overline{y} \rceil \wedge \overline{y} \text{ in } a^* \wedge \overline{y} \text{ in } a.a(b^*)^* \\
&\rightsquigarrow_{\{\mathsf{Memb}\}} X \doteq g_{\mathsf{o}}(\overline{y}) \wedge \overline{x} \doteq \lceil a, \overline{y} \rceil \wedge \overline{y} \text{ in } a.a^*
\end{aligned}$$

The obtained constraint is solved.

4. Constraint Logic Programming for Sequences and Contexts

4.1 Introduction

Constraint logic programming (CLP) [JM94, JMMS98] extends logic programming by generalizing the term equations of logic programming to constraints from a given domain. In this chapter we present constraint logic programming over the domain of sequences and contexts, CLP($\mathcal{SC}$). That means, we integrate the constraint solving algorithm studied in the previous chapter into the constraint logic programming schema to obtain CLP($\mathcal{SC}$). We describe the declarative and operational semantics of CLP($\mathcal{SC}$) and identify two fragments of programs which give rise to well-moded and KIF constraints.

4.2 CLP($\mathcal{SC}$) Programs

A CLP($\mathcal{SC}$) program is a finite set of *clauses* of the form $\forall(\mathbf{L}_1 \wedge \cdots \wedge \mathbf{L}_n \rightarrow \mathbf{A})$, usually written as $\mathbf{A} \leftarrow \mathbf{L}_1, \ldots, \mathbf{L}_n$, where $\mathbf{A}$ is an atom and $\mathbf{L}_1, \ldots, \mathbf{L}_n$ are literals ($n \geq 0$). A *query* is a formula of the form $\exists(\mathbf{L}_1 \wedge \cdots \wedge \mathbf{L}_n)$, $n \geq 0$, usually written as $\mathbf{L}_1, \ldots, \mathbf{L}_n$.

Given a program P, its Herbrand base $\mathcal{B}_\mathsf{P}$ is, naturally, the set of all atoms $p(t_1, \ldots, t_n)$, where p is an n-ary user-defined predicate in $\mathcal{P}$ and $\lceil t_1, \ldots, t_n \rceil \in \mathcal{T}^s(\mathcal{F})^n$. Then an intended interpretation of P corresponds uniquely to a subset of $\mathcal{B}_\mathsf{P}$. An *intended model* of P is an intended interpretation of P that is its model.

As usual, we will write $\mathsf{P} \models \mathsf{G}$ if G is a query which holds in every model of P. Since our programs consist of positive clauses, the following facts hold:

a) Every program P has a least $\mathcal{HC}$-model, which we denote by $lm(\mathsf{P}, \mathcal{HC})$.

b) If G is a query then $\mathsf{P} \models \mathsf{G}$ iff $lm(\mathsf{P}, \mathcal{HC})$ is a model of G.

Bellow, for illustration we give two CLP($\mathcal{SC}$) program examples: implementation of the general rewriting and recursive path ordering with status.

Example 4.1. *The general rewriting mechanism can be implemented with one* CLP($\mathcal{SC}$) *clauses:* $rewrite(X_{\mathsf{c}}(X), X_{\mathsf{c}}(Y)) \leftarrow rule(X, Y)$. *It is assumed that there are clauses which define the* rule *predicate. The rewrite clause says a term* $X_{\mathsf{c}}(X)$ *can be rewritten to* $X_{\mathsf{c}}(Y)$ *if there is a rule such that* $rule(X, Y)$ *succeeds.*

An example of the definition of the rule *predicate is*

$$rule(X_{\mathsf{f}}(\overline{x}_1, \overline{x}_2), X_{\mathsf{f}}(\overline{y})) \leftarrow \ \overline{x}_1 \ \mathsf{in} \ f_{\mathsf{o}}(a^*).b^*, \ \overline{x}_1 \doteq \lceil X, \overline{z} \rceil, \ \overline{y} \doteq \lceil X, f_{\mathsf{o}}(\overline{z}) \rceil,$$

where the constraint $\overline{x}_1 \ \mathsf{in} \ f_{\mathsf{o}}(a^*).b^*$ *requires* $\overline{x}_1$ *to be instantiated by sequences from the language generated by the regular sequence expression* $f_{\mathsf{o}}(a^*).b^*$ *(that is, from the language* $\{f_{\mathsf{o}}, f_{\mathsf{o}}(a), f_{\mathsf{o}}(a, a), \ldots, (f_{\mathsf{o}}, b), (f_{\mathsf{o}}(a), b), \ldots, (f_{\mathsf{o}}(a, \ldots, a), b, \ldots, b), \ldots\}$*).*

With this program, the query $\leftarrow rewrite(f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), b)), X)$ *has two answers:* $\{X \mapsto f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), f_{\mathsf{o}}))\}$ *and* $\{X \mapsto f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), f_{\mathsf{o}}(b)))\}$.

Example 4.2. *The recursive path ordering (rpo)* $>_{\mathrm{rpo}}$ *is a well-known term ordering [Der82] used to prove termination of rewriting systems. Its definition is based on a precedence order* $\succ$ *on function symbols, and on extensions of* $>_{\mathrm{rpo}}$ *from terms to tuples of terms. There are two kinds of extensions: lexicographic* $>_{\mathrm{rpo}}^{lex}$*, when terms in tuples are compared from left to right, and multiset* $>_{\mathrm{rpo}}^{mul}$*, when terms in tuples are compared disregarding the order. The status function* τ *assigns to each function symbol either* lex *or* mul *status. Then for all (ranked) terms* r, t*, we define* $r >_{\mathrm{rpo}} t$*, if* $r = f(r_1, \ldots, r_m)$ *and*

a) either $r_i = t$ *or* $r_i >_{\mathrm{rpo}} t$ *for some* r_i*,* $1 \leq i \leq m$*, or*

b) $t = g(t_1, \ldots, t_n)$*,* $r >_{\mathrm{rpo}} t_i$ *for all* $i, 1 \leq i \leq n$*, and either*

i) $f \succ g$*, or (b)* $f = g$ *and* $(r_1, \ldots, r_n) >_{\mathrm{rpo}}^{\tau(f)} (t_1, \ldots, t_n)$.

To implement this definition in CLP($\mathcal{SC}$)*, we use the predicate* rpo *for* $>_{\mathrm{rpo}}$ *between two terms, and four helper predicates: rpo_all to implement the comparison* $r >_{\mathrm{rpo}} t_i$ *for all*

i; prec to implement the comparison depending on the precedence; ext to implement the comparison with respect to an extension of $>_{\text{rpo}}$*; and* status *to give the status of a function symbol. The predicate* lex *implements* $>_{\text{rpo}}^{lex}$ *and* mul *implements* $>_{\text{rpo}}^{mul}$*. The symbol* f_{o} *is an unranked ordered function symbol taken from* $\mathcal{F}_{\mathsf{o}}$*, and* f_{u} *is an unordered unranked function symbol taken from* $\mathcal{F}_{\mathsf{u}}$*. We write both symbols in infix notation. As one can see, the implementation is rather straightforward and closely follows the definition.* $>_{\text{rpo}}$ *requires four clauses, since there are four alternatives in the definition:*

$$1.\quad rpo(X_{\mathsf{f}}(\overline{x}, X, \overline{y}), X). \qquad rpo(X_{\mathsf{f}}(\overline{x}, X, \overline{y}), Y) \leftarrow rpo(X, Y).$$

$$2a.\quad rpo(X_{\mathsf{f}}(\overline{x}), Y_{\mathsf{f}}(\overline{y})) \leftarrow rpo\_all(X_{\mathsf{f}}(\overline{x}), f_{\mathsf{o}}(\overline{y})), prec(X_{\mathsf{f}}, Y_{\mathsf{f}}).$$

$$2b.\quad rpo(X_{\mathsf{f}}(\overline{x}), X_{\mathsf{f}}(\overline{y})) \leftarrow rpo\_all(X_{\mathsf{f}}(\overline{x}), f_{\mathsf{o}}(\overline{y})), ext(X_{\mathsf{f}}(\overline{x}), X_{\mathsf{f}}(\overline{y})).$$

rpo_all is implemented with recursion:

$$rpo\_all(X, f_{\mathsf{o}}). \qquad rpo\_all(X, f_{\mathsf{o}}(Y, \overline{y})) \leftarrow rpo(X, Y), rpo\_all(X, f_{\mathsf{o}}(\overline{y})).$$

The definition of prec *as an ordering on finitely many function symbols is straightforward. More interesting is the definition of* ext*:*

$$ext(X_{\mathsf{f}}(\overline{x}), X_{\mathsf{f}}(\overline{y})) \leftarrow status(X_{\mathsf{f}}, lex), lex(f_{\mathsf{o}}(\overline{x}), f_{\mathsf{o}}(\overline{y})).$$
$$ext(X_{\mathsf{f}}(\overline{x}), X_{\mathsf{f}}(\overline{y})) \leftarrow status(X_{\mathsf{f}}, mul), mul(f_{\mathsf{u}}(\overline{x}), f_{\mathsf{u}}(\overline{y})).$$

The predicate status *can be given as a set of facts,* lex *needs one clause, and* mul *requires three:*

$$lex(f_{\mathsf{o}}(\overline{x}, X, \overline{y}), f_{\mathsf{o}}(\overline{x}, Y, \overline{z})) \leftarrow rpo(X, Y).$$
$$mul(f_{\mathsf{u}}(X, \overline{x}), f_{\mathsf{u}}). \qquad mul(f_{\mathsf{u}}(X, \overline{x}), f_{\mathsf{u}}(X, \overline{y})) \leftarrow mul(f_{\mathsf{u}}(\overline{x}), f_{\mathsf{u}}(\overline{y})).$$
$$mul(f_{\mathsf{u}}(X, \overline{x}), f_{\mathsf{u}}(Y, \overline{y})) \leftarrow rpo(X, Y),\ mul(f_{\mathsf{u}}(X, \overline{x}), f_{\mathsf{u}}(\overline{y})).$$

That's all. This examples illustrates the benefits of all four kinds of variables we have and unordered function symbols.

4.3 Operational Semantics

In this section we describe the operational semantics of CLP($\mathcal{SC}$), following the approach for the CLP schema given in [JMMS98]. A *state* is a pair $\langle \mathsf{G} \parallel \mathcal{C} \rangle$, where G is the sequence of literals and $\mathcal{C} = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$, where each of the $\mathcal{K}$'s are conjunctions of true, false, and primitive constraints. The *definition of an atom* $p(t_1, \ldots, t_m)$ *in program* P, $defn_{\mathsf{P}}(p(t_1, \ldots, t_m))$, is the set of rules in P such that the head of each rule has a form $p(r_1, \ldots, r_m)$. We assume that $defn_{\mathsf{P}}$ each time returns fresh variants. We also assume that $\Box$ denotes *the empty sequence of literals* (understood as the empty conjunction).

A state $\langle \mathbf{L}_1, \ldots, \mathbf{L}_n \parallel \mathcal{C} \rangle$ can be *reduced with respect to* P as follows: Select a literal $\mathbf{L}_i$. Then:

- If $\mathbf{L}_i$ is true or a primitive constraint literal and $\mathsf{solve}(\mathcal{C} \wedge \mathbf{L}_i) \neq \mathsf{false}$, then it is reduced to $\langle \mathbf{L}_1, \ldots, \mathbf{L}_{i-1}, \mathbf{L}_{i+1}, \ldots, \mathbf{L}_n \parallel \mathsf{solve}(\mathcal{C} \wedge \mathbf{L}_i) \rangle$.
- If $\mathbf{L}_i$ is true or false or a primitive constraint literal and $\mathsf{solve}(\mathcal{C} \wedge \mathbf{L}_i) = \mathsf{false}$, then it is reduced to $\langle \Box \parallel \mathsf{false} \rangle$.
- If $\mathbf{L}_i$ is an atom $p(t_1, \ldots, t_m)$, then it is reduced to

  $$\langle \mathbf{L}_1, \ldots, \mathbf{L}_{i-1}, t_1 \doteq r_1, \ldots, t_m \doteq r_m, \mathsf{B}, \mathbf{L}_{i+1}, \ldots, \mathbf{L}_n \parallel \mathcal{C} \rangle$$

  for some $(p(r_1, \ldots, r_m) \leftarrow \mathsf{B}) \in defn_{\mathsf{P}}(\mathbf{L}_i)$.
- If $\mathbf{L}_i$ is a atom and $defn_{\mathsf{P}}(\mathbf{L}_i) = \emptyset$, then it is reduced to $\langle \Box \parallel \mathsf{false} \rangle$.

A *derivation from a state* $\mathbb{S}$ in a program P is a finite or infinite sequence of states $\mathbb{S}_0 \rightarrowtail \mathbb{S}_1 \rightarrowtail \cdots \rightarrowtail \mathbb{S}_n \rightarrowtail \cdots$ where $\mathbb{S}_0$ is $\mathbb{S}$ and there is a reduction from each $\mathbb{S}_{i-1}$ to $\mathbb{S}_i$, using rules in P. A *derivation from a query* G in a program P is a derivation from $\langle \mathsf{G} \parallel \mathsf{true} \rangle$. The *length* of a (finite) derivation of the form $\mathbb{S}_0 \rightarrowtail \mathbb{S}_1 \rightarrowtail \cdots \rightarrowtail \mathbb{S}_n$ is n. A derivation is *finished* if the last query cannot be reduced, that is, if its last state is of the form $\langle \Box \parallel \mathcal{C} \rangle$ where $\mathcal{C}$ is partially solved or false. If $\mathcal{C}$ is false, the derivation is said to be *failed*.

Example 4.3. *Consider again the program given in Example 4.1. One of the finished derivations from the query* $\leftarrow rewrite(f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), b)), X)$ *is the following:*

$$\langle rewrite(f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), b)), X) \parallel \mathsf{true} \rangle \rightarrowtail$$
$$\langle X_{\mathsf{c0}}(X_0) \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), b)), X_{\mathsf{c0}}(Y_0) \doteq X, rule(X_0, Y_0) \parallel \mathsf{true} \rangle \rightarrowtail$$

$$\langle X_{\mathsf{c}0}(Y_0) \doteq X, \mathit{rule}(X_0, Y_0) \parallel$$
$$X_{\mathsf{c}0} \doteq \circ \wedge X_0 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),b))\vee$$
$$X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(\circ) \wedge X_0 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),b)\vee$$
$$X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(\circ,b)) \wedge X_0 \doteq f_{\mathsf{o}}(a,a)\vee$$
$$X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),\circ)) \wedge X_0 \doteq b\vee$$
$$X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(\circ,a),b)) \wedge X_0 \doteq a\vee$$
$$X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,\circ),b)) \wedge X_0 \doteq a\rangle \rightarrowtail$$
$$\langle \mathit{rule}(X_0, Y_0) \parallel$$
$$Y_0 \doteq X \wedge X_{\mathsf{c}0} \doteq \circ \wedge X_0 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),b))\vee$$
$$f_{\mathsf{o}}(Y_0) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(\circ) \wedge X_0 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),b)\vee$$
$$f_{\mathsf{o}}(f_{\mathsf{o}}(Y_0,b)) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(\circ,b)) \wedge X_0 \doteq f_{\mathsf{o}}(a,a)\vee$$
$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),Y_0)) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),\circ)) \wedge X_0 \doteq b\vee$$
$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(Y_0,a),b)) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(\circ,a),b)) \wedge X_0 \doteq a\vee$$
$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,Y_0),b)) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,\circ),b)) \wedge X_0 \doteq a\rangle \rightarrowtail$$
$$\langle X_0 \doteq X_{\mathsf{f}}(\overline{x}_1, \overline{x}_2), Y_0 \doteq X_{\mathsf{f}}(\overline{y}), \overline{x}_1 \text{ in } f_{\mathsf{o}}(a^*).b^*,\ \overline{x}_1 \doteq (X', \overline{z}),\ \overline{y} \doteq (X', f_{\mathsf{o}}(\overline{z})) \parallel$$
$$Y_0 \doteq X \wedge X_{\mathsf{c}0} \doteq \circ \wedge X_0 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),b))\vee$$
$$f_{\mathsf{o}}(Y_0) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(\circ) \wedge X_0 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),b)\vee$$
$$f_{\mathsf{o}}(f_{\mathsf{o}}(Y_0,b)) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(\circ,b)) \wedge X_0 \doteq f_{\mathsf{o}}(a,a)\vee$$
$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),Y_0)) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),\circ)) \wedge X_0 \doteq b\vee$$
$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(Y_0,a),b)) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(\circ,a),b)) \wedge X_0 \doteq a\vee$$
$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,Y_0),b)) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,\circ),b)) \wedge X_0 \doteq a\rangle \rightarrowtail$$
$$\langle Y_0 \doteq X_{\mathsf{f}}(\overline{y}), \overline{x}_1 \text{ in } f_{\mathsf{o}}(a^*).b^*,\ \overline{x}_1 \doteq (X', \overline{z}),\ \overline{y} \doteq (X', f_{\mathsf{o}}(\overline{z})) \parallel$$
$$Y_0 \doteq X \wedge X_{\mathsf{c}0} \doteq \circ \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),b) \wedge \overline{x}_2 \doteq \lceil\,\rceil\vee$$
$$Y_0 \doteq X \wedge X_{\mathsf{c}0} \doteq \circ \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil\,\rceil \wedge \overline{x}_2 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),b)\vee$$
$$f_{\mathsf{o}}(Y_0) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil\,\rceil \wedge \overline{x}_2 \doteq \lceil f_{\mathsf{o}}(a,a), b\rceil\vee$$
$$f_{\mathsf{o}}(Y_0) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq f_{\mathsf{o}}(a,a) \wedge \overline{x}_2 \doteq b\vee$$
$$f_{\mathsf{o}}(Y_0) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil f_{\mathsf{o}}(a,a), b\rceil \wedge \overline{x}_2 \doteq \lceil\,\rceil\vee$$
$$f_{\mathsf{o}}(f_{\mathsf{o}}(Y_0,b)) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(\circ,b)) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil\,\rceil \wedge \overline{x}_2 \doteq \lceil a, a\rceil\vee$$
$$f_{\mathsf{o}}(Y_0) \doteq X \wedge X_{\mathsf{c}0} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq a \wedge \overline{x}_2 \doteq a\vee$$

$$f_{\mathsf{o}}(Y_0) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil a, a \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), Y_0)) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), \circ)) \wedge X_{\mathsf{f}} \doteq b \wedge \overline{x}_1 \doteq \lceil \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(Y_0, a), b)) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(\circ, a), b)) \wedge X_{\mathsf{f}} \doteq a \wedge \overline{x}_1 \doteq \lceil \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, Y_0), b)) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, \circ), b)) \wedge X_{\mathsf{f}} \doteq a \wedge \overline{x}_1 \doteq \lceil \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \rangle \rightarrowtail$$

$$\langle \overline{x}_1 \ \mathsf{in}\ f_{\mathsf{o}}(a^*).b^*,\ \overline{x}_1 \doteq (X', \overline{z}),\ \overline{y} \doteq (X', f_{\mathsf{o}}(\overline{z})) \parallel$$

$$f_{\mathsf{o}}(\overline{y}) \doteq X \wedge X_{\mathsf{c0}} \doteq \circ \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), b) \wedge \overline{x}_2 \doteq \lceil \rceil \vee$$

$$f_{\mathsf{o}}(\overline{y}) \doteq X \wedge X_{\mathsf{c0}} \doteq \circ \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil \rceil \wedge \overline{x}_2 \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), b) \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil \rceil \wedge \overline{x}_2 \doteq \lceil f_{\mathsf{o}}(a, a), b \rceil \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq f_{\mathsf{o}}(a, a) \wedge \overline{x}_2 \doteq b \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil f_{\mathsf{o}}(a, a), b \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y}), b)) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(\circ, b)) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil \rceil \wedge \overline{x}_2 \doteq \lceil a, a \rceil \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq a \wedge \overline{x}_2 \doteq a \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil a, a \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), f_{\mathsf{o}}(\overline{y}))) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), \circ)) \wedge X_{\mathsf{f}} \doteq b \wedge \overline{x}_1 \doteq \lceil \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y}), a), b)) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(\circ, a), b)) \wedge X_{\mathsf{f}} \doteq a \wedge \overline{x}_1 \doteq \lceil \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, f_{\mathsf{o}}(\overline{y})), b)) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, \circ), b)) \wedge X_{\mathsf{f}} \doteq a \wedge \overline{x}_1 \doteq \lceil \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \rangle \rightarrowtail$$

$$\langle \overline{x}_1 \doteq (X', \overline{z}),\ \overline{y} \doteq (X', f_{\mathsf{o}}(\overline{z})) \parallel$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq f_{\mathsf{o}}(a, a) \wedge \overline{x}_2 \doteq b \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge \overline{x}_1 \doteq \lceil f_{\mathsf{o}}(a, a), b \rceil \wedge \overline{x}_2 \doteq \lceil \rceil \vee$$

$$\langle \overline{y} \doteq (X', f_{\mathsf{o}}(\overline{z})) \parallel$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge X' \doteq f_{\mathsf{o}}(a, a) \wedge \overline{z} \doteq \lceil \rceil \wedge \overline{x}_2 \doteq b \vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(\overline{y})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge X' \doteq f_{\mathsf{o}}(a, a) \wedge \overline{z} \doteq b \wedge \overline{x}_2 \doteq \lceil \rceil \rangle \rightarrowtail$$

$$\langle \square \parallel$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), f_{\mathsf{o}})) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge$$

$$X' \doteq f_{\mathsf{o}}(a, a) \wedge \overline{z} \doteq \lceil \rceil \wedge \overline{x}_2 \doteq b$$

$$\vee$$

$$f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a, a), f_{\mathsf{o}}(b))) \doteq X \wedge X_{\mathsf{c0}} \doteq f_{\mathsf{o}}(\circ) \wedge X_{\mathsf{f}} \doteq f_{\mathsf{o}} \wedge$$

$$X' \doteq f_{\mathsf{o}}(a, a) \wedge \overline{z} \doteq b \wedge \overline{x}_2 \doteq \lceil \rceil \rangle.$$

From the obtained solved constraint, we can extract two possible instantiations for the variable

X: $\{X \mapsto f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),f_{\mathsf{o}}))\}$ *and* $\{X \mapsto f_{\mathsf{o}}(f_{\mathsf{o}}(f_{\mathsf{o}}(a,a),f_{\mathsf{o}}(b)))\}$.

4.4 Well-Moded and KIF Programs

In this section we consider syntactic restrictions on programs that lead to well-moded and KIF[Gen98] constraints during derivations. As we have seen in Chapter 3, such constraints are interesting, because they can be completely solved by solve.

4.4.1 Well-Moded Programs

A mode for an n-ary predicate symbol p is a function $m_p : \{1,\ldots,n\} \longrightarrow \{\mathsf{i},\mathsf{o}\}$. If $m_p(i) = \mathsf{i}$ (resp. $m_p(i) = \mathsf{o}$) then the position i is called an *input* (resp. *output*) *position* of p. The predicates in and $\doteq$ have only output positions. For a literal $\mathbf{L} = p(t_1,\ldots,t_n)$ (where p can be also in or $\doteq$), we denote by $invar(\mathbf{L})$ and $outvar(\mathbf{L})$ the sets of variables occurring in terms in the input and output positions of p.

A sequence of literals $\mathbf{L}_1,\ldots,\mathbf{L}_n$ is *well-moded* if the following hold:

a) For all $1 \leq i \leq n$, $invar(\mathbf{L}_i) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j)$.

b) If for some $1 \leq i \leq n$, $\mathbf{L}_i$ is $t_1 \doteq t_2$, then $var(t_1) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j)$ or $var(t_2) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j)$.

c) If for some $1 \leq i \leq n$, $\mathbf{L}_i$ is $C_1 \doteq C_2$, then $var(C_1) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j)$ or $var(C_2) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j)$.

d) If for some $1 \leq i \leq n$, $\mathbf{L}_i$ is a membership atom, then the inclusion $var(\mathbf{L}_i) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j)$ holds.

A conjunction of literals G is *well-moded* if there exists a well-moded sequence of literals $\mathbf{L}_1,\ldots,\mathbf{L}_n$ such that $\mathsf{G} = \bigwedge_{i=1}^{n} \mathbf{L}_i$ modulo associativity and commutativity. A *formula in DNF is well-moded* if each of its disjuncts is. A *state* $\langle \mathbf{L}_1,\ldots,\mathbf{L}_n \parallel \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_m \rangle$ *is well-moded*, where $\mathcal{K}$'s are conjunctions of true, false, and primitive constraints, if the formula $(\mathbf{L}_1 \wedge \cdots \wedge \mathbf{L}_n \wedge \mathcal{K}_1) \vee \cdots \vee (\mathbf{L}_1 \wedge \cdots \wedge \mathbf{L}_n \wedge \mathcal{K}_m)$ is well-moded. A *clause* $\mathbf{A} \leftarrow \mathbf{L}_1,\ldots,\mathbf{L}_n$ *is well-moded* if the following hold:

a) For all $1 \leq i \leq n$, $invar(\mathbf{L}_i) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j) \cup invar(\mathbf{A})$.

b) $outvar(\mathbf{A}) \subseteq \bigcup_{j=1}^{n} outvar(\mathbf{L}_j) \cup invar(\mathbf{A})$.

c) If for some $1 \leq i \leq n$, $\mathbf{L}_i$ is $t_1 \doteq t_2$, then $var(t_1) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j) \cup invar(\mathbf{A})$ or $var(t_2) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j) \cup invar(\mathbf{A})$.

d) If for some $1 \leq i \leq n$, $\mathbf{L}_i$ is $C_1 \doteq C_2$, then $var(C_1) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j) \cup invar(\mathbf{A})$ or $var(C_2) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j) \cup invar(\mathbf{A})$.

e) If for some $1 \leq i \leq n$, $\mathbf{L}_i$ is a membership atom, then $outvar(\mathbf{L}_i) \subseteq \bigcup_{j=1}^{i-1} outvar(\mathbf{L}_j) \cup invar(\mathbf{A})$.

A CLP($\mathcal{SC}$) program is *well-moded* if all its clauses and query are well-moded.

Example 4.4. *In Example 4.1, if the first argument is the input position and the second argument is the output position in the user-defined predicates, it is easy to see that the program is well-moded. In Example 4.2, for well-modedness we need to define both positions in the user-defined predicates to be the input ones.*

We now show that well-modedness is preserved by program derivation steps. The first lemma is the following one:

Lemma 4.5. *Let $\mathcal{K}_1 \wedge \mathcal{K}_2 \wedge v \doteq e$ be a well-moded conjunction, where $\mathcal{K}_1$ and $\mathcal{K}_2$ are (maybe empty) conjunctions of literals, v is a variable and e is the corresponding expression such that v does not occur in e and $\theta = \{v \mapsto e\}$ is a simple substitution. Then $\mathcal{K}_1 \wedge \mathcal{K}_2\theta \wedge v \doteq e$ is well-moded.*

Proof. We consider two cases. First, when $v \doteq e$ is the leftmost literal containing v in a well-moded sequence corresponding to $\mathcal{K}_1 \wedge \mathcal{K}_2 \wedge v \doteq e$, and second, when this is not the case.

Case 1. Let $\tilde{\mathbf{L}}_1, v \doteq e, \tilde{\mathbf{L}}_2$ be a well-moded sequence corresponding to $\mathcal{K}_1 \wedge \mathcal{K}_2 \wedge v \doteq e$ such that $\tilde{\mathbf{L}}_1$ does not contain v. Then well-modedness requires the variables e to appear in $\tilde{\mathbf{L}}_1$. Consider the sequence $\tilde{\mathbf{L}}_1, v \doteq e, \tilde{\mathbf{L}}_2[\theta]$, where the notation $\tilde{\mathbf{L}}[\theta]$ stands for such an instance of $\tilde{\mathbf{L}}$, in which θ affects only literals from $\mathcal{K}_2$. Then $\tilde{\mathbf{L}}_1, v \doteq e$ is well-moded and it can be safely extended by $\tilde{\mathbf{L}}_2[\theta]$ without violating well-modedness, because the variables in $v \doteq e$ still precede literals from $\tilde{\mathbf{L}}_2[\theta]$, and the relative order of the other variables does not change. Hence, $\tilde{\mathbf{L}}_1, v \doteq e, \tilde{\mathbf{L}}_2[\theta]$ is a well-moded sequence that corresponds to $\mathcal{K}_1 \wedge \mathcal{K}_2\theta \wedge v \doteq e$.

Case 2. Let $\tilde{\mathbf{L}}_1, \mathbf{L}, \tilde{\mathbf{L}}_2, v \doteq e, \tilde{\mathbf{L}}_3$ be a well-moded sequence corresponding to $\mathcal{K}_1 \wedge \mathcal{K}_2 \wedge v \doteq e$, where $\mathbf{L}$ is the leftmost literal that contains v in an output position. Then $\tilde{\mathbf{L}}_1, \mathbf{L}, v \doteq e, \tilde{\mathbf{L}}_2, \tilde{\mathbf{L}}_3$ is also a well-moded sequence (corresponding to $\mathcal{K}_1 \wedge \mathcal{K}_2 \wedge v \doteq e$), because v still appears in an output position in $\mathbf{L}$ left to $v \doteq e$, the variables in e still precede literals from $\tilde{\mathbf{L}}_3$, and the relative order of the other variables does not change. For literals in $\tilde{\mathbf{L}}_2$ that contain variables from e such a reordering does not matter.

Note that v does not appear in $\tilde{\mathbf{L}}_1$: If it was there in some literal in an output position, then $\mathbf{L}$ would not be the leftmost such literal. If it was there in some literal $\mathbf{L}'$ in an input position, then well-modedness of the sequence would require v to appear in an output position in another literal $\mathbf{L}''$ that is even before $\mathbf{L}'$, i.e., to the left of $\mathbf{L}$ and it would again contradict the assumption that $\mathbf{L}$ is the leftmost literal containing v in an output position.

Let $\tilde{\mathbf{L}}_1, \mathbf{L}[\theta], v \doteq e, \tilde{\mathbf{L}}_2[\theta], \tilde{\mathbf{L}}_3[\theta]$ be a sequence of all literals taken from $\mathcal{K}_1 \wedge \mathcal{K}_2 \wedge v \doteq e$. we distinguish two cases, depending whether θ affects $\mathbf{L}$ or not.

θ affects $\mathbf{L}$. Then it replaces v in $\mathbf{L}$ with e, i.e., $\mathbf{L}[\theta] = \mathbf{L}\theta$. Then the variables of e appear in output positions in $\mathbf{L}\theta$ and, hence, placing $v \doteq e$ after $\mathbf{L}\theta$ in the sequence would not destroy well-modedness. As for the $\mathbf{L}\theta$ itself, we have two alternatives:

a) $\mathbf{L}\theta$ is an equation, say $s \doteq t\theta$, obtained from $\mathbf{L} = (s \doteq t)$ by replacing occurrences of v in t by e. In this case, by well-modedness of $\tilde{\mathbf{L}}_1, \mathbf{L}, v \doteq e, \tilde{\mathbf{L}}_2, \tilde{\mathbf{L}}_3$, variables of s appear in $\tilde{\mathbf{L}}_1$ and s does not contain v. Then the same property is maintained in $\tilde{\mathbf{L}}_1, \mathbf{L}\theta, v \doteq e, \tilde{\mathbf{L}}_2[\theta], \tilde{\mathbf{L}}_3[\theta]$, since s remains in $\mathbf{L}\theta$ and $\tilde{\mathbf{L}}_1$ does not change.

b) $\mathbf{L}\theta$ is an atom. Then replacing v by e in an output position of $\mathbf{L}$, which gives $\mathbf{L}\theta$, does not affect well-modedness.

Hence, we got that $\tilde{\mathbf{L}}_1, \mathbf{L}, v \doteq e$ is well-moded. Now we can safely extend this sequence with $\tilde{\mathbf{L}}_2[\theta], \tilde{\mathbf{L}}_3[\theta]$, because variables in new occurrences of e in $\tilde{\mathbf{L}}_2[\theta], \tilde{\mathbf{L}}_3[\theta]$ are preceded by $v \doteq e$, and the relative order of the other variables does not change. Hence, the sequence $\tilde{\mathbf{L}}_1, \mathbf{L}\theta, v \doteq e, \tilde{\mathbf{L}}_2[\theta], \tilde{\mathbf{L}}_3[\theta]$ is well-moded.

θ does not affects $\mathbf{L}$. Then $\mathbf{L}[\theta] = \mathbf{L}$, the sequence $\tilde{\mathbf{L}}_1, \mathbf{L}, v \doteq e$ is well-moded and it can be safely extended with $\tilde{\mathbf{L}}_2[\theta], \tilde{\mathbf{L}}_3[\theta]$, obtaining the well-moded sequence $\tilde{\mathbf{L}}_1, \mathbf{L}, v \doteq e$, $\tilde{\mathbf{L}}_2[\theta], \tilde{\mathbf{L}}_3[\theta]$.

Hence, we showed also in *Case 2* that there exists a well-moded sequence of literals, namely,

$\tilde{\mathbf{L}}_1, \mathbf{L}[\theta], v \doteq e, \tilde{\mathbf{L}}_2[\theta], \tilde{\mathbf{L}}_3[\theta]$, that corresponds to $\mathcal{K}_1 \wedge \mathcal{K}_2\theta \wedge v \doteq e$. Hence, $\mathcal{K}_1 \wedge \mathcal{K}_2\theta \wedge v \doteq e$ is well-moded. □

Lemma 4.6. *Let* P *be a well-moded* CLP($\mathcal{SC}$) *program and* $\langle \mathsf{G} \parallel \mathcal{C} \rangle$ *be a well-moded state. If* $\langle \mathsf{G} \parallel \mathcal{C} \rangle \rightarrowtail \langle \mathsf{G}' \parallel \mathcal{C}' \rangle$ *is a reduction using clauses in* P*, then* $\langle \mathsf{G}' \parallel \mathcal{C}' \rangle$ *either is also a well-moded state, or* $\mathsf{G}' = \square$ *and* $\mathcal{C}' = \mathsf{false}$*, or* $\mathsf{G}' = \square$ *and* $\mathcal{C}' = \mathsf{true}$*.*

Proof. Let $\mathsf{G} = \mathbf{L}_1, \ldots, \mathbf{L}_i, \ldots, \mathbf{L}_n$, $\mathcal{C} = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_m$, and $\langle \mathsf{G} \parallel \mathcal{C} \rangle$ be a well-moded state. Assume that $\mathbf{L}_i$ is the selected literal in the reduction that gives $\langle \mathsf{G}' \parallel \mathcal{C}' \rangle$ from $\langle \mathsf{G} \parallel \mathcal{C} \rangle$. We consider four possible cases, according to the definition of the operational semantics:

Case 1. Let $\mathbf{L}_i$ be a primitive constraint and $\mathcal{C}' \neq \mathsf{false}$. Let $\mathcal{D}$ denote the DNF of $\mathcal{C} \wedge \mathbf{L}_i$. Then $\mathcal{C}' = \mathsf{solve}(\mathcal{D})$.

In order to prove that $\langle \mathsf{G}' \parallel \mathcal{C}' \rangle = \langle \mathsf{G}' \parallel \mathsf{solve}(\mathcal{D}) \rangle$ is well-moded, by the definition of solve, it is sufficient to prove that $\langle \mathsf{G}' \parallel \mathsf{step}(\mathcal{D}) \rangle$ is well-moded. Since, obviously, $\langle \mathsf{G}' \parallel \mathcal{D} \rangle$ is a well-moded state, we have to show that state well-modedness is preserved by each rule of the solver.

Since $\mathcal{C}' \neq \mathsf{false}$, the step is not performed by any of the failure rules of the solver. For the rules MS1–MS8, MS10–MS11, MC1–MC9, D1–D4, it is pretty easy to verify that $\langle \mathsf{G}' \parallel \mathsf{step}(\mathcal{D}) \rangle$ is well-moded. Therefore, we consider the other rules in more detail. We denote the disjunct of $\mathcal{D}$ on which the rule is applied by $\mathcal{K}_\mathcal{D}$. The cases below are distinguished by the rules:

Del. Here the same variable is removed from both sides of the selected equation. Assume $\tilde{\mathbf{L}}_1, s \doteq t, \tilde{\mathbf{L}}_2$ is a well-moded sequence corresponding to $\mathsf{G}' \wedge \mathcal{K}_\mathcal{D}$, and $s \doteq t$ is the selected equation affected by one of the deletion rules. Well-modedness of $\tilde{\mathbf{L}}_1, s \doteq t, \tilde{\mathbf{L}}_2$ requires that the variable deleted at this step from $s \doteq t$ should occur in an output position in some other literal in $\tilde{\mathbf{L}}_1$. Let $s' \doteq t'$ be the equation obtained by the deletion step from $s \doteq t$. Then $\tilde{\mathbf{L}}_1, s' \doteq t', \tilde{\mathbf{L}}_2$ is again well-moded, which implies that $\mathsf{G}' \wedge \mathsf{step}(\mathcal{K}_\mathcal{D})$ is well-moded and, therefore, that $\langle \mathsf{G}' \parallel \mathsf{step}(\mathcal{D}) \rangle$ is well-moded.

MS9. Let $\mathsf{G}' \wedge \mathcal{K}_\mathcal{D}$ be represented as $\mathsf{G}' \wedge X_\mathsf{f}(\tilde{s})$ in $f(\mathsf{RS}) \wedge \mathcal{K}'$, where $X_\mathsf{f}(\tilde{s})$ in $f(\mathsf{RS})$ is the equation affected by the rule. Note that then $\mathsf{G}' \wedge X_\mathsf{f}(\tilde{s})$ in $f(\mathsf{RS}) \wedge X_\mathsf{f} \doteq f \wedge \mathcal{K}'$ is also well-moded. Applying Lemma 4.5, we get that $\mathsf{G}' \wedge X_\mathsf{f}(\tilde{s})\theta$ in $f(\mathsf{RS}) \wedge X_\mathsf{f} \doteq f \wedge \mathcal{K}'\theta$

is well-moded, where $\theta = \{X_{\mathsf{f}} \mapsto f\}$. But it means that $\mathsf{G}' \wedge \mathsf{step}(\mathcal{K}_{\mathcal{D}})$ is well-moded, which implies that $\langle \mathsf{G}' \parallel \mathsf{step}(\mathcal{D}) \rangle$ is well-moded.

MS12. Let $\mathsf{G}' \wedge \mathcal{K}_{\mathcal{D}}$ be represented as $\mathsf{G}' \wedge \overline{x}$ in $f(\mathsf{RS}) \wedge \mathcal{K}'$, where $\overline{x}$ in $f(\mathsf{RS})$ is the equation affected by the rule. Note that then $\mathsf{G}' \wedge \overline{x} \doteq X \wedge X$ in $f(\mathsf{RS}) \wedge \mathcal{K}'$ is also well-moded. Applying Lemma 4.5, we get that $\mathsf{G}' \wedge \overline{x} \doteq X \wedge X$ in $f(\mathsf{RS}) \wedge \mathcal{K}'\theta$ is well-moded, where $\theta = \{\overline{x} \mapsto X\}$. Then we get well-modedness of $\mathsf{G}' \wedge \mathsf{step}(\mathcal{K}_{\mathcal{D}})$, which implies well-modedness of $\langle \mathsf{G}' \parallel \mathsf{step}(\mathcal{D}) \rangle$.

MC10. Analogous to the MS12 rule above.

E1–E4. Well-modedness of $\mathsf{G}' \wedge \mathsf{step}(\mathcal{K}_{\mathcal{D}})$ is a direct consequence of Lemma 4.5.

E5. Let $\mathsf{G}' \wedge \mathcal{K}_{\mathcal{D}}$ be represented as $\mathsf{G}' \wedge \lceil \overline{x}, \tilde{s}_1 \rceil \simeq \tilde{s}_2 \wedge \mathcal{K}'$, where $\lceil \overline{x}, \tilde{s}_1 \rceil \simeq \tilde{s}_2$ is the equation affected by the rule and $\overline{x} \notin var(\tilde{s}_2)$. Note that then $\mathsf{G}' \wedge \overline{x} \doteq \tilde{s}' \wedge \tilde{s}_1 \doteq \tilde{s}'' \wedge \mathcal{K}'$ is also well-moded for some $\tilde{s}'$ and $\tilde{s}''$ with $\lceil \tilde{s}', \tilde{s}'' \rceil = \tilde{s}_2$. Applying Lemma 4.5, we get that $\mathsf{G}' \wedge \overline{x} \doteq \tilde{s}' \wedge \tilde{s}_1\theta \doteq \tilde{s}'' \wedge \mathcal{K}'\theta$ is well-moded, where $\theta = \{\overline{x} \mapsto \tilde{s}'\}$. Since $\tilde{s}'$ and $\tilde{s}''$ were arbitrary, it implies that $\mathsf{G}' \wedge \mathsf{step}(\mathcal{K}_{\mathcal{D}})$ and, therefore, $\langle \mathsf{G}' \parallel \mathsf{step}(\mathcal{D}) \rangle$ is well-moded.

E6–E8. Similar to the case of the rule E5.

Case 2. Let $\mathbf{L}_i$ be a primitive constraint and $\mathcal{C}' = \mathsf{false}$, where $\mathcal{C}' = \mathsf{solve}(\mathcal{C} \wedge \mathbf{L}_i)$. Then by the operational semantics we have $\mathsf{G}' = \square$ and the theorem trivially holds.

Case 3. Let $\mathbf{L}_i$ be an atom $p(t_1, \ldots, t_k, \ldots, t_l)$. Assume that P contains a clause of the form $p(r_1, \ldots, r_k, \ldots, r_l) \leftarrow \mathsf{B}$. Assume also $\{1, \ldots, k\}$ are the input positions of p and $\{k+1, \ldots, l\}$ are the output ones. Then we have

$$\begin{aligned}
&\mathsf{G} = \mathbf{L}_1, \ldots, \mathbf{L}_{i-1}, p(t_1, \ldots, t_k, \ldots, t_l), \mathbf{L}_{i+1}, \ldots, \mathbf{L}_n,\\
&\mathsf{G}' = \mathbf{L}_1, \ldots, \mathbf{L}_{i-1}, t_1 \doteq r_1, \ldots, t_k \doteq r_k, \ldots, t_l \doteq r_l, \mathsf{B}, \mathbf{L}_{i+1}, \ldots, \mathbf{L}_n,\\
&\mathcal{C}' = \mathcal{C} = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_m.
\end{aligned}$$

From well-modedness of $\langle \mathsf{G} \parallel \mathcal{C} \rangle$, we know that for all $1 \leq j \leq m$ there exists a sequence $\tilde{\mathbf{L}}_j$ of literals taken from $\mathbf{L}_1, \ldots, \mathbf{L}_{i-1}, \mathbf{L}_{i+1}, \ldots, \mathbf{L}_n$ and $\mathcal{K}_j$ such that $var(\lceil t_1, \ldots, t_k \rceil) \subseteq outvar(\tilde{\mathbf{L}}_j)$. From well-modedness of $p(r_1, \ldots, r_k, \ldots, r_l) \leftarrow \mathsf{B}$, we know $var(\lceil r_{k+1}, \ldots, r_l \rceil) \subseteq outvar(\mathsf{B}) \cup var(\lceil r_1, \ldots, r_k \rceil)$. Input variables of literals from B remain within output variables of literals from B and in terms $r_1, \ldots, r_k$. Therefore, we can construct a well-

moded sequence from all literals in $\mathsf{G}' \wedge \mathcal{K}_j$ for all $1 \leq j \leq m$. It means that $\langle \mathsf{G}' \parallel \mathcal{K}_j \rangle$ is well-moded for all $1 \leq j \leq m$, which implies that $\langle \mathsf{G}' \parallel \mathcal{C}' \rangle$ is well-moded.

Case 4. If $defn_P(\mathbf{L}_i) = \emptyset$, then $\mathsf{G}' = \square$, $\mathcal{C}' = \mathsf{false}$, and the theorem trivially holds. □

The theorem below is the main theorem for well-moded CLP($\mathcal{SC}$) programs. It states that any finished derivation from a well-moded query leads to a solved constraint or to a failure:

Theorem 4.7. *Let* $\langle \mathsf{G} \parallel \mathsf{true} \rangle \rightarrowtail \cdots \rightarrowtail \langle \square \parallel \mathcal{C}' \rangle$ *be a finished derivation with respect to a well-moded* CLP($\mathcal{SC}$) *program, starting from a well-moded query* G*. If* $\mathcal{C}' \neq \mathsf{false}$*, then* $\mathcal{C}'$ *is solved.*

Proof. We prove a slightly more general statement: If $\langle \mathsf{G} \parallel \mathsf{true} \rangle \rightarrowtail \cdots \rightarrowtail \langle \mathsf{G}' \parallel \mathcal{C}' \rangle$ is a derivation with respect to a well-moded program, starting from a well-moded query G and ending with G' that is either $\square$ or consists only of atomic formulas without arguments (propositional constants). If $\mathcal{C}' \neq \mathsf{false}$, then $\mathcal{C}'$ is solved.

To prove this statement, we use induction on the length n of the derivation. When $n = 0$, then $\mathcal{C}' = \mathsf{true}$ and it is solved. Assume the statement holds when the derivation length is n, and prove it for the derivation with the length $n + 1$. Let such a derivation be $\langle \mathsf{G} \parallel \mathsf{true} \rangle \rightarrowtail \cdots \rightarrowtail \langle \mathsf{G}_n \parallel \mathcal{C}_n \rangle \rightarrowtail \langle \mathsf{G}_{n+1} \parallel \mathcal{C}_{n+1} \rangle$. There are two possibilities to make the last step:

a) G_n has a form (modulo permutation) $\mathbf{L}, p_1, \ldots, p_n$, where $\mathbf{L}$ is primitive constraint, the p's are propositional constants, $\mathsf{G}_{n+1} = p_1, \ldots, p_n$, and $\mathcal{C}_{n+1} = \mathsf{solve}(\mathcal{C}_n \wedge \mathbf{L})$.

b) G_n has a form (modulo permutation) $q, p_1, \ldots, p_n$, where q and p's are propositional constants, $\mathsf{G}_{n+1} = p_1, \ldots, p_n$, and $\mathcal{C}_{n+1} = \mathcal{C}_n$.

In the first case, note that by Lemma 4.6, $\langle \mathsf{G}_n \parallel \mathcal{C}_n \rangle$ is well-moded. Since the p's have no influence on well-modedness (they are just propositional constants), $\mathcal{C}_n \wedge \mathbf{L}$ is well-moded and hence it is solvable. By Lemma 3.7 we get that if $\mathcal{C}_{n+1} = \mathsf{solve}(\mathcal{C}_n \wedge \mathbf{L}) \neq \mathsf{false}$ then $\mathcal{C}_{n+1}$ is solved.

In the second case, since G_n consists of propositional constants only, by the induction hypothesis we have that if $\mathcal{C}_n$ is not false, then it is solved. But $\mathcal{C}_n = \mathcal{C}_{n+1}$. It finishes the proof. □

4.4.2 Programs in the KIF Form

An atom $p(t_1, \ldots, t_n)$ is in the KIF form, if each t_i, $1 \leq i \leq n$, is a KIF-term. A CLP($\mathcal{SC}$) program is in the KIF form, if it is constructed from literals in the KIF form. Note that the programs in examples 4.1 and 4.2 are not KIF programs. One could rewrite them in this form, but the code size would become a bit larger. KIF programs, like the well-moded ones discussed above, also show a good behavior: Reductions preserve the KIF form, and finished non-failed derivations lead to solved constraints. These results are formally stated below.

Lemma 4.8. *Let* P *be a* CLP($\mathcal{SC}$) *program in the KIF form and* $\langle \mathsf{G} \parallel \mathcal{C} \rangle$ *be a KIF state. If* $\langle \mathsf{G} \parallel \mathcal{C} \rangle \rightarrowtail \langle \mathsf{G}' \parallel \mathcal{C}' \rangle$ *is a reduction using clauses in* P*, then* $\langle \mathsf{G}' \parallel \mathcal{C}' \rangle$ *is also a KIF state or* $\mathsf{G}' = \square$ *and* $\mathcal{C}' = \mathsf{false}$ *or* $\mathsf{G}' = \square$ *and* $\mathcal{C}' = \mathsf{true}$.

Proof. Inspecting the rules of the solver, one can easily see that KIF constraints are transformed into KIF constraints, which implies the lemma. □

Theorem 4.9. *Let* $\langle \mathsf{G} \parallel \mathsf{true} \rangle \rightarrowtail \cdots \rightarrowtail \langle \square \parallel \mathcal{C}' \rangle$ *be a finished derivation with respect to a* CLP($\mathcal{SC}$) *program in the KIF form, starting from a KIF query* G*. If* $\mathcal{C}' \neq \mathsf{false}$*, then* $\mathcal{C}'$ *is solved.*

Proof. Similar to the proof of Theorem 4.7. □

5. Rule-Based Programming

5.1 Introduction

This chapter is about the rule-based programming tool PρLog, based on the ρLog calculus [MK06] for conditional transformations of unranked sequences.

Programming with rules has been experiencing a period of growing interest since the nineties when rewriting logic [MOM02] and rewriting calculus [CK01] have been developed and several systems and languages (ASF-SDF [vdBvDH$^+$01], CHR [Frü98], Claire [CJL04], ELAN [BKK$^+$98], Maude [CDE$^+$02], Stratego [BKVV08], just to name a few) emerged. The ρLog calculus has been influenced by the rewriting calculus, but there are some significant differences: ρLog adopts logic programming semantics (clauses are first class concepts, rules/strategies are expressed as clauses), uses top-position matching, and employs four different kinds of variables that we have already seen in the previous chapter.

PρLog (pronounced Pē-rō-log) [DK] is an experimental tool that extends logic programming with strategic conditional transformation rules, combining Prolog with ρLog calculus. PρLog programs consist of clauses. Like CLP($\mathcal{SC}$), it supports programming with individual, sequence, function, and context variables. The clauses either define user-constructed strategies by (conditional) transformation rules or are ordinary Prolog clauses. Prolog code can be used freely within PρLog programs. One can include its predicates in PρLog rules, which is especially convenient when arithmetic calculations or input-output features are needed.

PρLog inference mechanism is essentially the same as SLDNF-resolution, multiple results are generated via backtracking, its semantics is compatible with semantics of normal logic programs [Llo87] and, hence, Prolog was a natural choice to base PρLog on: The inference mechanism comes for free, as well as the built-in arithmetic and many other useful features of the Prolog language. Following Prolog, PρLog is also untyped, but values of sequence

and context variables can be constrained by regular sequence or context languages (as in CLP($\mathcal{SC}$)). Essentially, the matching mechanism used in PρLog can be seen as an implementation of the well-moded constraint solving from Chapter 3.

5.2 An Overview of PρLog

As it has already been mentioned, PρLog is an implementation of the ρLog calculus in Prolog, extending the host language with strategy-based conditional sequence transformation rules. ρLog deals with simple sequences, transforming them by conditional rules. Transformations are non-deterministic and may yield several results. Logic programming seems to be a suitable framework for such non-deterministic computations. Strategies provide a control on rule applications in a declarative way. Strategy combinators help the user to construct more complex strategies from simpler ones. Rules apply matching to the whole input sequence (or, if it is a single term, apply at the top position).

In this chapter we follow the PρLog language notation, writing in the `typewriter` font the syntactic categories given in Section 2.3. PρLog uses the following conventions for the variables names: Individual variables start with `i_` (like, e.g., `i_Var` for a named variable or `i_` for the anonymous variable), sequence variables start with `s_`, function variables start with `f_`, and context variables start with `c_`. PρLog also uses prolog variables (named prolog variables beginning with the capital letters and anonymous prolog variables are printed as the underscore) written in the `typewriter` font. The symbol $\mathcal{V}_{\mathtt{an}}$ stands for the set of anonymous variables. The function symbols and the hole symbol is written as `f,g,h` and `hole`. Regular operators are written as: `eps` (empty sequence), `sconc` (sequence concatenation), `sor` (sequence choice), `sstar` (sequence repetition), `hole` (empty context), `cconc` (context concatenation), `cor` (context choice), and `cstar` (context repetition). Note, that we use the same symbol `hole` for a context constructor and for a regular operator. Note also, PρLog syntax requires simple sequences to be put within round brackets instead of ceiling corner brackets.

In this section we give a brief overview of basic features of PρLog, explaining them mostly on examples instead of giving formal definitions.

5.2.1 Programs and Queries

A ρLog *atom* (ρ-atom) is a quadruple consisting of a simple term *st* (a *strategy*), two simple sequences $\tilde{s}_1$ and $\tilde{s}_2$, and a set of regular constraints R where each variable is constrained only once, written as *st* `::` $\tilde{s}_1$ `==>` $\tilde{s}_2$ `where` R. Intuitively, it means that the strategy *st* transforms $\tilde{s}_1$ to $\tilde{s}_2$ when the variables satisfy the constraint R. We call $\tilde{s}_1$ the left hand side and $\tilde{s}_2$ the right hand side of this atom. When R is empty, we omit it and write *st* `::` $\tilde{s}_1$ `==>` $\tilde{s}_2$. The negated atom is written as *st* `::` $\tilde{s}_1$ `=\=>` $\tilde{s}_2$ `where` R. A ρLog *literal* (ρ-literal) is a ρ-atom or its negation. Literals, denoted by **L** is a common name for ρ- or Prolog literals. A PρLog *clause* is either a Prolog clause, or a clause of the form *st* `::` $\tilde{s}_1$ `==>` $\tilde{s}_2$ `where` R `:-`B (in the sequel called a ρ-clause) where B is a (possibly empty) conjunction of literals.

A PρLog *program* is a sequence of PρLog clauses and a *query* is a conjunction of ρ- and Prolog literals. There is a restriction on variable occurrence imposed on clauses: ρ-clauses and queries can contain only individual, sequence, function and context variables, while Prolog clauses and queries can contain only Prolog variables. If a Prolog literal occurs in a ρ-clause or query, it may contain only ρLog individual variables that internally get translated into Prolog variables.

Example 5.1. *The following code illustrates how factorial can be written in PρLog*

```
factorial :: 0 ==> 1.
factorial :: i_N ==> i_Fact :-
     i_N > 0,
     i_N1 is i_N - 1,
     factorial :: i_N1 ==> i_Fact1,
     i_Fact is i_Fact1 * i_N.
```

A sample PρLog query can have a form:

```
?((i_N is 2*3, factorial ::  i_N ==> i_X), Subst).
```

It combines Prolog and PρLog subqueries: The first one computes $2*3$*, the second one gives the factorial of 6. The solving substitution is returned in* `Subst`*.*

5.2.2 Operational Semantics

PρLog inference mechanism is based essentially on SLDNF-resolution adapted to ρ-clauses. Sequence solving algorithm integrated in the PρLog system is obtained by reformulation of the constraint solver studied in Chapter 3. The algorithm takes as an input a constraint and returns as an output a substitution which solves the constraint. In these rules below, P stands for a program, Q denotes a query and solve is the sequence solving algorithm. The strategy `id` is the built-in and stands for identity. The rules have the form $\mathsf{G}_1 \rightsquigarrow \mathsf{G}_2$, transforming the query G_1 into a new query G_2.

R: **Resolvent**

$$st \texttt{ :: } \tilde{s}_1 \texttt{ ==> } \tilde{s}_2 \texttt{ where } \mathsf{R} \wedge \mathsf{G} \rightsquigarrow$$
$$\sigma(\mathsf{B} \wedge (\texttt{id :: } \tilde{s}_2' \texttt{ ==> } \tilde{s}_2 \texttt{ where } \mathsf{R}) \wedge \mathsf{G})$$

where st is not `id`, there exists a clause $st' \texttt{ :: } \tilde{s}_1' \texttt{ ==> } \tilde{s}_2' \texttt{ where } \mathsf{R}' \texttt{ :- } \mathsf{B}$ in P such that $\sigma \in \mathsf{solve}(st' \doteq st \wedge \tilde{s}_1' \doteq \tilde{s}_1 \wedge \mathsf{R}')$.

Id: **Identity**

$$\texttt{id :: } \tilde{s}_1 \texttt{ ==> } \tilde{s}_2 \texttt{ where } \mathsf{R} \wedge \mathsf{G} \rightsquigarrow \sigma(\mathsf{G}) \qquad \text{if } \sigma \in \mathsf{solve}(\tilde{s}_2 \doteq \tilde{s}_1 \wedge \mathsf{R})$$

NF: **Negation as Failure**

$$(st \texttt{ :: } \tilde{s}_1 \texttt{ =\=> } \tilde{s}_2 \texttt{ where } \mathsf{R}) \wedge \mathsf{G} \rightsquigarrow \mathsf{G}$$

if there exists a finitely failed SLDNF-derivation tree for $st \texttt{ :: } \tilde{s}_1 \texttt{ ==> } \tilde{s}_2 \texttt{ where } \mathsf{R}$ with respect to P.

We do not define here the standard notions like derivations, finitely failed SLDNF-derivation tree, etc. They can be found in the literature elsewhere, see, e.g, [AB94] for a survey. For Prolog clauses the usual SLDNF-resolution rules apply.

These rules can be applied in different (finitely many) ways to the same selected query and the same program clause, because there can be more than one solution σ. We need to impose *well-modedness* restrictions on ρ-clauses and queries to guarantee that in the derivations generated constraints has finite solutions.

More specifically, well-modedness for PρLog programs extends the same notion for constraint logic programs, given in Chapter 4, to negative literals and literals with anonymous variables. The input and the output position of the relation $\cdot \texttt{ :: } \cdot \texttt{ ==> } \cdot$ is $\mathsf{i}(\cdot \texttt{ :: } \cdot \texttt{ ==> } \cdot) = \{1, 2\}$ and $\mathsf{o}(\cdot \texttt{ :: } \cdot \texttt{ ==> } \cdot) = \{3\}$ respectively. A mode is defined (uniquely) for a Prolog predicate as well. A clause is *moded* if all its predicate symbols are moded. We assume that all ρ-clauses

are moded. As for the Prolog clauses, we require modedness only for those ones that define a predicate that occurs in the body of some ρ-clause. If a Prolog literal occurs in a query in conjunction with a ρ-clause, then its relation and the clauses that define this relation are also assumed to be moded.

Definition 5.2. A query $\mathbf{L}_1, \ldots, \mathbf{L}_n$ is well-moded iff it satisfies the following conditions for each $1 \leq i \leq n$:

- $invar(\mathbf{L}_i) \subseteq \cup_{j=1}^{i-1} outvar(\mathbf{L}_j) \setminus \mathcal{V}_{\mathsf{an}}$.
- If $\mathbf{L}_i$ is a negative literal, then $outvar(\mathbf{L}_i) \subseteq \cup_{j=1}^{i-1} outvar(\mathbf{L}_j) \cup \mathcal{V}_{\mathsf{an}}$.
- If $\mathbf{L}_i$ is a ρLog literal, then its strategy term is ground.

A clause $\mathbf{L}_0$`:-`$\mathbf{L}_1, \ldots, \mathbf{L}_n$ is well-moded, iff the following conditions are satisfied for each $1 \leq i \leq n$:

- $invar(\mathbf{L}_i) \cup outvar(\mathbf{L}_0) \subseteq \cup_{j=0}^{i-1} outvar(\mathbf{L}_j) \setminus \mathcal{V}_{\mathsf{an}}$.
- If $\mathbf{L}_i$ is a negative literal, then $outvar(\mathbf{L}_i) \subseteq \cup_{j=1}^{i-1} outvar(\mathbf{L}_j) \cup \mathcal{V}_{\mathsf{an}} \cup invar(\mathbf{L}_0)$.
- If $\mathbf{L}_0$ and $\mathbf{L}_i$ are ρLog literals with the strategy terms st_0 and st_i, respectively, then $var(st_i) \subseteq var(st_0)$.

A PρLog program is *well-moded* if all its clauses and query are well-moded.

PρLog allows only well-moded program clauses and queries. There is no restriction on the Prolog clauses if the predicate they define is not used in a ρ-clause.

Example 5.3. *The query* `str1 :: a ==> i_X, str2 :: i_Y ==> i_Z` *is not well-moded, because the variable* `i_Y` *in the input position of the second subgoal does not occur in the output position of the first subgoal. On the other hand,* `str1 :: a ==> i_X, str2 :: i_X ==> i_Z` *is well-moded.*

If we change the last goal by `str1 :: a ==> i_X, str2 :: i_X =\=> i_Z`*, well-modedness will get violated again, because the variable* `i_Z`*, occurring in the negative literal, does not appear in the output position of the previous subgoal. Examples of well-moded queries involving negative literals are, e.g.,* `str1 :: a ==> (i_X, i_Z), str2 :: i_X =\=> i_Z` *and* `str1 :: a ==> i_X, str2 :: i_X =\=> i_.`

5.2.3 Predefined Strategies and Strategy Combinators

Strategies can be combined to express in a compact way many tedious small step transformations. These combinations give more control on transformations. PρLog provides a library of several predefined strategy combinators. Most of them are standard. The user can write her own strategies in PρLog or extend the Prolog code of the library. Some of the predefined strategies and their intuitive meanings are the following:

- `id ::` $\tilde{s}_1$ `==>` $\tilde{s}_2$ succeeds if the sequence $\tilde{s}_1$ and $\tilde{s}_2$ are identical (or can be made identical by $\tilde{s}_2$ matching $\tilde{s}_1$) and fails otherwise.

- `compose`$(\mathtt{st}_1, \mathtt{st}_2, \ldots, \mathtt{st}_n)$, $n \geq 2$, first transforms the input sequence by $\mathtt{st}_1$ and then transforms the result by `compose`$(\mathtt{st}_2, \ldots, \mathtt{st}_n)$ (or by $\mathtt{st}_2$, if $n = 2$). Via backtracking, all possible results can be obtained. The strategy fails if either $\mathtt{st}_1$ or `compose`$(\mathtt{st}_2, \ldots, \mathtt{st}_n)$ fails.

- `choice`$(\mathtt{st}_1, \ldots, \mathtt{st}_n)$, $n \geq 1$, returns a result of a successful application of some strategy $\mathtt{st}_i$ to the input sequence. It fails if all $\mathtt{st}_i$'s fail. By backtracking it can return all outputs of the applications of each of the strategies $\mathtt{st}_1, \ldots, \mathtt{st}_n$.

- `first_one`$(\mathtt{st}_1, \ldots, \mathtt{st}_n)$, $n \geq 1$, selects the first $\mathtt{st}_i$ that does not fail on the input sequence and returns only one result of its application. `first_one` fails if all $\mathtt{st}_i$'s fail. Its variation, `first_all`, returns via backtracking all the results of the application to the input sequence of the first strategy $\mathtt{st}_i$ that does not fail.

- `nf`(`st`), when terminates, computes a normal form of the input sequence with respect to `st`. It never fails because if an application of `st` to a sequence fails, then `nf(st)` returns that sequence itself. Backtracking returns all normal forms.

- `iterate`$(\mathtt{st}, N)$ starts transforming the input sequence with `st` and returns a result (via backtracking all the results) obtained after N iterations for a given natural number N.

- `map1`(`st`) maps the strategy `st` to each term in the input sequence and returns the result sequence. Backtracking generates all possible output sequences. `st` should operate on a single term and not on an arbitrary sequence. `map1`(`st`) fails if `st` fails on at least one term from the input sequence. `map` is a variation of `map1` where the single-term

restriction is removed. It should be used with care because of high non-determinism. Both `map1` and `map`, when applied to the empty sequence, return the empty sequence.

- `interactive` takes a strategy from the user, transforms the input sequence by it and waits for further user instruction (either to apply another strategy to the result sequence or to finish).

- `rewrite(st)` applies to a single term (not to an arbitrary sequence) and rewrites it by `st` (which also applies to a single term). Via backtracking, it is possible to obtain all the rewrites. The input term is traversed in the leftmost-outermost manner. Note that `rewrite(st)` can be easily implemented inside PρLog:

```
rewrite(i_Str) :: c_Context(i_Redex) ==> c_Context(i_Contractum) :-
     i_Str :: i_Redex ==> i_Contractum.
```

5.2.4 System Components

The system consists of the following parts:

a) the parser, in file `parse.pl`,

b) the library `library.pl` that implements a collection of common strategy operators, such as, e.g., `compose`, `choice`, `nf`, `first_one`, `rewrite`, `map1`, etc.

c) the compiler `compile.pl` which translates PρLog clauses and queries into Prolog clauses and goals, and

d) the solver `solve.pl` that implements the matching algorithm.

A typical PρLog session consists of the following steps: (1) Write a PρLog program in a file, say, *`filename`*`.rho`; (2) Start Prolog and load the main PρLog file `prholog.pl`; (3) Compile the PρLog program, by calling `consult(`*`filename`*`.rho')`; and (4) Query PρLog to answer a goal `G` by calling

```
?(G,R).
```

where `R` is a fresh logic variable. This call translates `G` into a semantically equivalent Prolog goal, solves it by SLDNF resolution, and translates the Prolog result into a list of PρLog

bindings that are assigned to `R`. In the next subsections we will give more insights of each system component.

5.2.5 Examples Implemented in PρLog

We give below few examples that demonstrate the use some of the PρLog features, including the strategies we just mentioned. The users can define own strategies in a program either by writing clauses for them or using abbreviations of the form `str_1 := str_2`. Such an abbreviation stands for the clause `str_1 :: s_X ==> s_Y :- str_2 :: s_X ==> s_Y`.

Example 5.4. *Let* `str1` *and* `str2` *be two strategies defined as follows:*

```
str1 :: (s_1, a, s_2) ==> (s_1, f(a), s_2).
str2 :: (s_1, i_x, s_2, i_x, s_3) ==> (s_1, i_x, s_2, s_3).
```

Manipulating strategies in the goal we get different answers:

- *The goal* `str1 :: (a, b, a, f(a)) ==> s_X` *returns two answers (instantiations of the sequence variable* `s_X`*):* `(f(a), b, a, f(a))` *and* `(a, b, f(a), f(a))`*. Multiple answers are computed by backtracking. They are two because* `(s_1, a, s_2)` *matches* `(a, b, a, f(a))` *in two ways, with the matchers* {`s_1`↦`eps`, `s_2`↦`(b, a, f(a))`} *and* {`s_1`↦`(a, b)`, `s_2`↦`f(a)`}*, respectively.*
- *If we change the previous goal by* `str1 :: (a, b, a, f(a)) ==> (s_X, f(a), s_Y)`*, then PρLog will return four answers that correspond to the following instantiations of* `s_X` *and* `s_Y`*:*
  - *a)* `s_X`↦`eps`, `s_Y`↦`(b, a, f(a))`.
  - *b)* `s_X`↦`(f(a), b, a)`, `s_Y`↦`eps`.
  - *c)* `s_X`↦`(a, b)`, `s_Y`↦`f(a)`.
  - *d)* `s_X`↦`(a, b, f(a))`, `s_Y`↦`eps`.
- *The goal* `str1 :: (a, b, a, f(a)) =\=> s_` *fails, because its positive counterpart succeeds. On the other hand,* `str1 :: (a, b, a, f(a)) =\=> (b, s_)` *succeeds.*
- *The composition* `compose(str1, str2) :: (a, b, a, f(a)) ==> s_X` *gives two answers:* `(f(a), b, a)` *and* `(a, b, f(a))`*,*

- *On the goal* `choice(str1, str2) :: (a, b, a, f(a)) ==> s_X` *we get three answers:* `(f(a), b, a, f(a))`, `(a, b, f(a), f(a))`, *and* `(a, b, f(a))`.
- `nf(compose(str1, str2)) :: (a, b, a, f(a)) ==> s_X`, *which computes a normal form of the composition, returns* `(f(a), b)` *twice, computing it in two different ways.*
- *The goal* `first_one(str1, str2) :: (a, b, a, f(a)) ==> s_X` *returns only one answer* `(f(a), b, a, f(a))`. *This is the first output computed by the first applicable strategy,* `str1`.
- *Finally,* `first_all(str1, str2) :: (a, b, a, f(a)) ==> s_X` *computes two instantiations:* `(f(a), b, a, f(a))` *and* `(a, b, f(a), f(a))`. *These are all the answers returned by the first applicable strategy,* `str1`.

Example 5.5. *The two PρLog clauses below flatten nested occurrences of the head function symbol of a term. The code is written using function and sequence variables, which makes it reusable, since it can be used to flatten terms with different heads and different numbers of arguments:*

```
flatten_one :: f_Head(s_1, f_Head(s_2), s_3) ==> f_Head(s_1, s_2, s_3).
flatten := nf(flatten_one).
```

The first clause flattens one occurrence of the nested head. The second one (written in the abbreviated form) defines the `flatten` *strategy as the normal form of* `flatten_one`. *Here are some examples of queries involving these strategies:*

- `flatten_one :: f(a, f(b, f(c)), f(d)) ==> i_X` *gives* `f(a, b, f(c), f(d))`.
- `flatten :: f(a, f(b, f(c)), f(d)) ==> i_X` *returns* `f(a, b, c, d)`.
- *We can map the strategy* `flatten` *to a sequence, which results in flattening each element of the sequence. For instance, the goal* `map1(flatten) :: (a,f(f(a)),g(a, g(b))) ==> s_X` *returns the sequence* `(a, f(a), g(a, b))`.

Example 5.6. *The* `replace` *strategy takes a term and a sequence of replacement rules in the form* `(lhs_1 -> rhs_1 ... lhs_n -> rhs_n)`, *chooses a subterm in the given term that can be replaced by a rule* `lhs_i -> rhs_i`, *and returns the result of the replacement.* `replace_all` *computes a normal form with respect to the given replacement rules.*

```
replace :: (c_Context(i_X), s_1, i_X -> i_Y, s_2) ==>
           (c_Context(i_Y), s_1, i_X -> i_Y, s_2).

replace_all :: (i_Term, s_Rules) ==> i_Instance :-
                nf(replace) :: (i_Term, s_Rules) ==> (i_Instance, s_).
```

With `replace_all`*, one can, for example, compute an instance of a term under an idempotent substitution:* `replace_all :: (f(x, g(x, y)), x -> z, y -> a) ==> i_X` *gives* `f(z, g(z, a))`*. (We can take the conjunction of this goal with the cut predicate to avoid recomputing the same instance several times.) The same code can be used to compute a normal form of a term under a ground rewrite system, the sort of a term if the rules are sorting rules, etc.*

Example 5.7. *This is a bit longer example that shows how one can specify a simple propositional proving procedure in PρLog. We assume that the propositional formulas are built over negation (denoted by '*`-`*') and disjunction (denoted by '*`v`*'). The corresponding PρLog program starts with the Prolog operator declaration that declares disjunction an infix operator:*

```
:- op(200, xfy, v).
```

Next, we describe inference rules of a Gentzen-like sequent calculus for propositional logic. The rules operate on sequents and are represented as `sequent(ant(`sequence of formulas`), cons(`sequence of formulas`))`*.* `ant` *and* `cons` *are tags for the antecedent and consequent, respectively. There are five inference rules in the calculus: The axiom rule, negation left, negation right, disjunction left, and disjunction right.*

```
axiom :: sequent(ant(s_, i_Formula, s_), cons(s_, i_Formula, s_)) ==>  eps.

neg_left :: sequent(ant(s_F1, -(i_Formula), s_F2), cons(s_F3)) ==>
     sequent(ant(s_F1, s_F2), cons(i_Formula, s_F3)).

 neg_right :: sequent(ant(s_F1), cons(s_F2, -(i_Formula), s_F3)) ==>
      sequent(ant(s_F1, i_Formula), cons(s_F2, s_F3)).

 disj_left :: sequent(ant(s_F1, i_Formula1 v i_Formula2, s_F2), i_Cons) ==>
```

```
      (sequent(ant(s_F1, i_Formula1, s_F2), i_Cons),
       sequent(ant(s_F1, i_Formula2, s_F2), i_Cons)).

disj_right :: sequent(i_ant, cons(s_F1, i_Formula1 v i_Formula2, s_F2)) ==>
     sequent(i_ant, cons(s_F1, i_Formula1, i_Formula2, s_F2)).
```

Next, we need to impose control on the applications of the inference rules and define success and failure of the procedure. The control is pretty straightforward: To perform an inference step on a given sequence of sequents, we select the first sequent and apply to it the first applicable inference rule, in the order specified in the arguments of the strategy `first_one` *below. When there are no sequents left, the procedure ends with success. Otherwise, if no inference step can be made, we have failure.*

```
success :: eps ==> true.

inference_step :: (sequent(i_Ant, i_Cons), s_Sequents) ==>
                  (s_New_sequents, s_Sequents) :-
      first_one(axiom, neg_left, neg_right, disj_left, disj_right) ::
                  sequent(i_Ant, i_Cons) ==> s_New_sequents.

failure :: (sequent(i_Ant, i_Cons), s_Sequents) ==> false.
```

Finally, we specify the proof procedure as repeatedly applying the first possible strategy between `success`, `inference_step`, *and* `failure` *(in this order) until none of them is applicable:*

```
prove := nf(first_one(success, inference_step, failure)).
```

Note that it does matter in which order we put the clauses for the inference rules or the control in the program. What matters, is the order they are combined (e.g. as it is done in the strategy `first_one`*).*

What we described here is just one way of implementing the given propositional proof procedure in PρLog. One could do it differently as well, for instance, by writing recursive clauses like it has been shown in [MK06]. However, we believe that the version above is more declarative and naturally corresponds to the way the procedure is described in textbooks.

Note that there can be several clauses for the same strategy in a PρLog program. In this case they behave as usual alternatives of each other (when a query with this strategy is being evaluated) and are tried in the order of their appearance in the program, i.e. top-down.

Example 5.8. *The following program illustrates how bubble sort can be implemented in PρLog.*

```
swap(f_Ordering) :: (s_X, i_I, s_Y, i_J, s_Z) ==>
                          (s_X, i_J, s_Y, i_I, s_Z)  :-
                               not( f_Ordering(i_I,i_J)).

bubble_sort(f_Ordering) := first_one(nf(swap(f_Ordering))).
```

This algorithm takes two elements from a given sequence and compares them with respect to `f_Ordering`*. If the elements are not determined to be ordered by* `f_Ordering`*, then they are swapped.* `nf` *applies* `swap` *repeatedly until impossible, which leads to a sorted sequence.*

Note that, `bubble_sort(f_Ordering) := first_one(nf(swap(f_Ordering)))` *is an abbreviation of the clause*

```
bubble_sort(f_Ordering) :: s_X ==> s_Y :-
               first_one(nf(swap(f_Ordering))) :: s_X ==> s_Y.
```

The query `bubble_sort(=<)::(1,3,4,3,2) ==> s_X` *returns* `s_X = (1, 2, 3, 3, 4)`*.*

5.3 Case Study 1: XML Processing and Web Reasoning

In this section, we illustrate how PρLog can be used in XML querying, validation, and reasoning, pretty naturally and concisely expressing problems coming from these areas. For these applications, PρLog uses the unranked tree model, represented as a Prolog term. Below we assume that the XML input is provided in the translated form.

5.3.1 Querying

Maier in [Mai98] gives a list of query operations that are desirable for an XML query language: selection, extraction, reduction, restructuring, and combination. They all should

be expressible in a single query language. A comparison of five query languages on the basis of these queries is given in [BC00]. Here we demonstrate, on the *car dealer office* example, how these queries can be expressed in PρLog.

Example 5.9. *A car dealer office contains documents from different car dealers and brokers. There are two kinds of documents. The* `manufacturer` *documents list the manufacturer's name, year, and models with their names, front rating, side rating, and rank. The* `vehicle` *documents list the vendor, make, model, year, color and price. They are presented by XML data of the following form:*

```
<manufacturer>
   <mn-name>Mercury</mn-name>
   <year>1998</year>
   <model>
     <mo-name>Sable LT</mo-name>
     <front-rating>
        3.84
     </front-rating>
     <side-rating>
        2.14
     </side-rating>
     <rank>9</rank>
   </model> ...
</manufacturer>
```

```
<vehicle>
   <vendor>
      Scott Thomason
   </vendor>
   <make>Mercury</make>
   <model>Sable LT</model>
   <year>1999</year>
   <color>
      metallic blue
   </color>
   <price>26800</price>
</vehicle>
```

We assume that sequences of these elements are wrapped respectively by `<list-manuf>` *and* `<list-vehicle>` *tags. To save space, in the queries below we use metavariable M to refer to the document consisting of the list of manufacturers, i.e., the document with the root tag* `<list-manuf>`*. Similarly, the metavariable V denotes the document with the root tag* `<list-vehicle>`*.*

Selection and Extraction: We want to select and extract `<manufacturer>` elements where some `<model>` has `<rank>` less or equal to 10.

```
select_and_extract :: list_manuf(s_,c_Manuf(rank(i_Rank)),s_) ==>
```

```
                        c_Manuf(rank(i_Rank)) :-
    i_Rank =< 10.
```

Given the goal `select_and_extract :: M ==> i_M`, this code generates all solutions, one after the other, via backtracking. The alternatives are generated according to the ways the term `list_manuf(s_,c_Manuf(rank(i_Rank)), s_)` matches M. If a `<manufacturer>` element contains two or more models with the rank ≤ 10, it will be returned several times. However, with a little modification of the code we can make sure that no such duplicated answers are computed:

```
select_and_extract :: list_manuf(s_,manufacturer(s_X),s_) ==>
                      manufacturer(s_X) :-
    select :: manufacturer(s_X) ==> manufacturer(s_X).

select :: c_Manuf(rank(i_Rank)) ==> c_Manuf(rank(i_Rank)) :-
    i_Rank =< 10,
    !.
```

Reduction: From the `<manufacturer>` elements, we want to drop the `<model>` subelements whose `<rank>` is greater than 10. Besides that, we also want to elide the `<front_rating>` and `<side_rating>` elements from the remaining models. It can be done in various ways in PρLog. One of such implementations is given below. `reduction` is defined as the normal form of transforming each `manufacturer` element inside `list_manuf`. A single `manufacturer` element is transformed by `reduction_step` depending whether it contains a model with the rank ≤ 10:

```
reduction :: list_manuf(s_1) ==> list_manuf(s_2) :-
    map1(nf(reduction_step)) :: s_1 ==> s_2,
    !.

reduction_step :: manufacturer(s_1,model(s_,rank(i_R)),s_2) ==>
                  manufacturer(s_1,s_2) :-
    i_R > 10.
reduction_step :: manufacturer(s_1,model(i_Name,i_,s_,rank(i_R)),s_2) ==>
```

```
                manufacturer(s_1,model(i_Name,rank(i_R)),s_2) :-
    i_R =< 10.
```

Then the query `reduction :: M ==> i_List` produces the list of reduced `manufacturer` elements.

Join: We want our query to generate pairs of `<manufacturer>` and `<vehicle>` elements where `<mn-name>` = `<make>`, `<mo-name>` = `<model>` and `<year>` = `<year>`. The implementation is straightforward:

```
join :: (list_manuf(s_,
            manufacturer(mn_name(i_Manuf_Name),year(i_Year),
               s_,model(mo_name(i_Model_Name),s_X),s_),s_),
         list_vehicle(s_,
            vehicle(i_Vendor,make(i_Manuf_Name),
               model(i_Model_Name),year(i_Year),i_Price),s_)
        ) ==>
        (manufacturer(mn_name(i_Manuf_Name),year(i_Year),
            model(mo_name(i_Model_Name),s_X)),
         vehicle(i_Vendor,make(i_Manuf_Name),
            model(i_Model_Name),year(i_Year),i_Price)
        ).
```

The query `join :: (M,V) ==> (i_Manuf,i_Vehicle)` returns a desired pair. One can compute all such pairs via backtracking.

Restructuring: We want our query to collect `<car>` elements listing their `make`, `model`, `vendor`, `rank`, and `price` subelements, in this order.

```
restructuring :: (list_manuf(s_,
                      manufacturer(mn_name(i_Manuf_Name),s_,
                          model(mo_name(i_Model_Name),s_X,i_Rank),
                          s_),
                      s_),
```

```
 list_vehicle(s_,
     vehicle(i_Vendor,make(i_Manuf_Name),
         model(i_Model_Name),year(i_Year),
         i_Color,i_Price),
     s_)
 )  ==>
car(make(i_Manuf_Name),model(i_Model_Name),
     i_Vendor,i_Rank,i_Price).
```

The query `restructuring :: (`M, V`) ==> i_Car` returns a car element. Backtracking gives all the answers.

5.3.2 Incomplete Queries

Often, the structure of a Web document to be queried is unknown to a query author, even if the schema to which the document conforms is familiar to her. The reason is that the schemas allow much flexibility for documents, expressed in terms of arbitrary repetition of substructures, or optional or alternative structures. Even more often, the query author is interested not in the entire structure of the document but only in its relevant parts. Therefore, a pattern-based Web querying language should be able to express such incomplete queries. Schaffert in [Sch04] classifies incomplete queries (four kinds of incompleteness: in breadth, in depth, with respect to order and with respect to optional elements) and explains how they are dealt with in the Xcerpt Language. Here we show how they can be expressed in PρLog. As we will see, it can be done pretty naturally, without introducing additional constructs for them.

Incompleteness in Breadth. In languages that have wildcards only for single terms, expressing incompleteness in breadth requires a special construct that allows to omit those wildcards for neighboring nodes in the data tree. In PρLog, we do not need any extra construct because of sequence variables. Anonymous sequence variables can be used as wildcards for arbitrary sequence of nodes. Furthermore, if needed, we can use named sequence variables to extract arbitrary sequence of nodes without knowing the exact structure. This is a very convenient feature. The second `select_and_extract` clause from the previous section is a good example. There the anonymous sequence variable `s_` helps to omit the irrelevant part

of the document. The named sequence variable `s_X` helps to extract a sequence of nodes, without knowing its length and structure.

Incompleteness in Depth. This kind of incompleteness allows to select data items that are located at arbitrary, unknown depth and skip all structure in between. The context variables in PρLog make this operation straightforward: Just place the query subterm you are interested in under an anonymous context variable. For instance, to extract only the rank values from the manufacturer elements in Example 5.9, we can write a simple clause:

```
select_rank :: c_(rank(i_Rank)) ==> i_Rank.
```

Here the anonymous context variable `c_` helps to descend to arbitrary depth, ignoring all the structure in between. We can do even more: If needed, we can extract the entire context above the query subterm without knowing the depth and the structure of the context. For this, it is enough just to put there a named context variable. This has been done in the first `select_and_extract` clause in the previous section with the `c_Manuf` variable:

```
select_and_extract :: list_manuf(s_,c_Manuf(rank(i_Rank)),s_) ==>
                      c_Manuf(rank(i_Rank)) :-
    i_Rank =< 10.
```

In fact, this clause also demonstrates how incompleteness in breadth and depth can be combined in a single rule in PρLog.

Incompleteness with respect to Order. It allows to specify neighboring nodes in a different order than the one in that they occur in the data tree. Since PρLog syntax does not allow unordered function symbols, we need a bit of more coding to express incomplete queries with respect to order. For instance, assume that we do not know in which order the `front_rating` and `side_rating` elements occur in the model in Example 5.9 and write the clause that extract them:

```
extract_ratings :: c_(model(s_X)) ==> (i_Front,i_Side) :-
    id :: model(s_X) ==> model(s_1,front_rating(i_Front),s_2),
    id :: model(s_1,s_2) ==> model(s_,side_rating(i_Side),s_).
```

In the first subgoal of the body of this rule, the `id` strategy forces the term `model(s_1, front_rating(i_Front),s_2)` to match `model(s_X)`, extracting the value for front rating `i_Front`. To find the side rating, we force matching `model(s_,side_rating(i_Side),s_)` to `model(s_1,s_2)` that is obtained from `model(s_X)` by deleting `front_rating(i_Front)`. This deletion comes for free from the previous match and we can take an advantage of it, since there is no need to keep `front_rating` in the structure where `side_rating` is looked for. One can notice that in this example we also used incomplete queries in depth and breadth.

Incompleteness with respect to Optional Elements. It allows to query for certain substructures if they exist, but still let the query succeed if they do not exist. Since sequence variables can be instantiated with the empty sequence as well, such incomplete queries can be trivially expressed in PρLog.

5.3.3 Validation

PρLog permits regular constraints in its clauses for context and sequence variables. They, in particular, can be used to check whether an XML document conforms to certain DTD that can be expressed by means of regular sequence expressions. We demonstrate this in the following example:

Example 5.10. *Let the DTD below define the structure of XML documents, which contain manufacturer elements.*

```
<!ELEMENT list-manuf (manufacturer*)>
<!ELEMENT manufacturer (mn-name, year, model*)>
<!ELEMENT model (mo-name, front-rating, side-rating, rank)>
<!ELEMENT mn-name (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT front-rating (#PCDATA)>
<!ELEMENT side-rating (#PCDATA)>
<!ELEMENT rank (#PCDATA)>
```

Then the validation task becomes a PρLog clause where the DTD is encoded in a regular constraint:

```
validate :: s_X ==> true
   where [s_X in list_manuf(manufacturer(mn_name(i_),year(i_),
                                model(mo_name(i_),front_rating(i_),
                                  side_rating(i_),rank(i_))*)*)]
```

(With `i_` *in the constraint we abbreviate the set of all ground terms with respect to the given finite alphabet.) To check whether a certain document conforms this DTD, we take a PρLog term T that represents that document and write the query* `validate :: T ==> true`*. The matching algorithm will try to match* `s_X` *to T and check whether the constraints are satisfied. If the document conforms the DTD, the query will succeed, otherwise it will fail.*

Validation test can be tailored in XML transformations, to make sure that the result of the transformation conforms a given schema. Similar to the `validate` *clause above, it is straightforward to express such tasks in PρLog. We do not elaborate on the details here.*

5.3.4 Basic Web Reasoning

Reasoning plays a crucial role in making data processing on the Web more "intelligent". Semantic Web adds metadata to Web resources, which can be used to make retrieval "semantic". To query both data and metadata, languages need to have certain reasoning capabilities. In this section we demonstrate basic reasoning capabilities of PρLog using the *Clique of Friends* example from [Sch04].

Example 5.11 (Clique of Friends)**.** *This example illustrates some basic reasoning for the Semantic Web. It does not use any particular Semantic Web language itself.*

Consider a collection of address books where each address book has an owner and a set of entries, some of which are marked as `friend` *to indicate that the person associated with this entry is considered a friend by the owner of the address book. In XML, this collection of address books can be represented in a straightforward manner as the following example illustrates:*

```
<address-books>
   <address-book>
      <owner>Donald Duck</owner>
      <entry>
```

```
            <name>Daisy Duck</name>
            <friend/>
         </entry>
         <entry>
            <name>Scrooge McDuck</name>
         </entry>
      </address-book>
      <address-book>
         <owner>Daisy Duck</owner>
         <entry>
            <name>Gladstone Duck</name>
            <friend/>
         </entry>
         <entry>
            <name>Ratchet Gearloose</name>
            <friend/>
         </entry>
      </address-book>
</address-books>
```

The collection contains two address books, the first owned by `Donald Duck` *and the second by* `Daisy Duck`*. Donalds address book has two entries, one for* `Scrooge`*, the other for* `Daisy`*, and only* `Daisy` *is marked as* `friend`*. Daisys address book again has two entries, both marked as* `friend`*.*

The clique-of-friends *of Donald is the set of all persons that are either direct friends of Donald (i.e. in the example above only* `Daisy`*) or friends of a friend (i.e.* `Gladstone` *and* `Ratchet`*), or friends of friends of friends (none in the example above), and so on. To retrieve these friends, we have to define the relation "being a friend of" and its transitive closure.*

Transitive closure of a relation can be easily defined in PρLog. It can be even written in a generic way, parametrized by the strategy that defines the relation:

```
transitive_closure(i_Strategy) :: s_X ==> s_Y :-
    i_Strategy :: s_X ==> s_Y.
transitive_closure(i_Strategy) :: s_X ==> s_Z :-
```

```
i_Strategy :: s_X ==> s_Y,
transitive_closure(i_Strategy) :: s_Y ==> s_Z.
```

The relation of "being a friend of" with respect to the address books document is defined as follows:

```
friend_of(address_books(s_,
            address_book(owner(i_X),s_,entry(name(i_Y),friend),s_),
            s_)) :: i_X ==> i_Y.
```

The query `transitive_closure(friend_of(`T`)) :: Donald_Duck ==> i_Y`*, where* T *is the PρLog term corresponding to the address book XML document above, will return one after the other the friend and the friends of the friend of* `Donald_Duck: Daisy_Duck`*,* `Gladstone_Duck`*, and* `Ratchet_Gearloose`*.*

5.4 Case Study 2: Implementation of Rewriting Strategies

In this section we illustrate how rewriting strategies can be implemented in PρLog. It can be done in a pretty succinct and declarative way. The code for leftmost-outermost and outermost rewriting is shorter than the one for leftmost-innermost and innermost rewriting, because it takes an advantage of PρLog's built-in term traversal strategy.

5.4.1 Leftmost-Outermost and Outermost Rewriting

As mentioned above, the `rewrite` strategy traverses a term in leftmost-outermost order to rewrite subterms. For instance, if the strategy `strat` is defined by two rules

```
strat :: f(i_X) ==> g(i_X).
strat :: f(f(i_X)) ==> i_X.
```

then for the goal `rewrite(strat) :: h(f(f(a)), f(a)) ==> i_X` we get, via backtracking, four instantiations for `i_X`, in the following order: `h(g(f(a)), f(a))`, `h(a, f(a))`, `h(f(g(a)), f(a))`, and `h(f(f(a)), g(a))`.

If we want to obtain *only one result*, then it is enough to add the cut predicate at the end of the goal. Indeed, `rewrite(strat) :: h(f(f(a)), f(a)) ==> i_X, !` returns only `h(g(f(a)), f(a))`.

To get *all the results of leftmost-outermost rewriting,* we have to find the first redex and rewrite it in all possible ways (via backtracking), ignoring all the other redexes. This can be done by using an anonymous variable for checking reducibility, and then putting the cut:

```
rewrite_left_out(i_Str) :: c_Context(i_Redex) ==> c_Context(i_Contractum) :-
        i_Str :: i_Redex ==> i_,
        !,
        i_Str :: i_Redex ==> i_Contractum.
```

The goal `rewrite_left_out(strat) :: h(f(f(a)), f(a)) ==> i_X` gives two instantiations for `i_X`: `h(g(f(a)), f(a))` and `h(a, f(a))`.

To return *all the results of outermost rewriting* we find an outermost redex and rewrite it. Backtracking returns all the results for all outermost redexes.

```
   rewrite_out(i_Str) :: i_X ==> i_Y :-
        i_Str :: i_X ==> i_,
        !,
        i_Str :: i_X ==> i_Y.

   rewrite_out(i_Str) :: f_F(s_1, i_X, s_2) ==> f_F(s_1, i_Y, s_2) :-
        rewrite_out(i_Str) :: i_X ==> i_Y.
```

The goal `rewrite_out(strat) :: h(f(f(a)), f(a)) ==> i_X` gives three answers, in this order: `h(g(f(a)), f(a))`, `h(a, f(a))`, and `h(f(f(a)), g(a))`.

5.4.2 Leftmost-Innermost and Innermost Rewriting

Implementation of innermost strategy in PρLog is slightly more involved than the implementation of outermost rewriting. It is not surprising since the outermost strategy takes an advantage of the PρLog built-in term traversal strategy. For innermost rewriting, we could have modified the PρLog source by simply changing the order of two rules in the matching

algorithm to give preference to the rule that descends deep in the term structure. It would change the term traversal strategy from leftmost-outermost to leftmost-innermost. Another way would be to build term traversal strategies into PρLog (like it is done in ELAN and Stratego, for instance) that would give the user more control on traversal strategies, giving her a possibility to specify the needed traversal inside a PρLog program.

However, here our aim is different: We would like to demonstrate that rewriting strategies can be implemented quite easily inside PρLog. For the outermost strategy it has already been shown. As for the innermost rewriting, if we want to obtain *only one result by leftmost-innermost strategy*, we first check whether any argument of the selected subterm rewrites. If not, we try to rewrite the subterm and if we succeed, we cut the alternatives. The way how matching is done guarantees that the leftmost possible redex is taken:

```
rewrite_left_in_one(i_Str) :: c_Ctx(f_F(s_Args)) ==> c_Ctx(i_Contractum) :-
        rewrites_at_least_one(i_Str) :: s_Args =\=> i_,
        i_Str :: f_F(s_Args) ==> i_Contractum,
        !.

   rewrites_at_least_one(i_Str) :: (s_, i_X, s_) ==> true :-
        rewrite(i_Str) :: i_X ==> i_,
        !.
```

To get *all results of leftmost-innermost rewriting*, we first check whether the selected subterm is an innermost redex. If yes, the other redexes are cut off and the selected one is rewritten in all possible ways:

```
   rewrite_left_in(i_Str) :: c_Context(f_F(s_Args)) ==>
                             c_Context(i_Contractum) :-
        rewrites_at_least_one(i_Str) :: s_Args =\=> i_,
        i_Str :: f_F(s_Args) ==> i_,
        !,
        i_Str :: f_F(s_Args) ==> i_Contractum.
```

If `strat` is the strategy defined in the previous section, then we have only one answer for the goal `rewrite_left_in(strat) :: h(f(f(a)),f(a)) ==> i_X` and it is the term `h(f(g(a)), f(a))`. The same term is returned by `rewrite_left_in_one`.

Finally, `rewrite_in` computes *all results of innermost rewriting* via backtracking:

```
rewrite_in(i_Str) :: f_F(s_Args) ==> i_Y :-
     rewrites_at_least_one(i_Str) :: s_Args =\=> i_,
     i_Str :: f_F(s_Args) ==> i_Y.

rewrite_in(i_Str) :: f_F(s_1, i_X, s_2) ==> f_F(s_1, i_Y, s_2) :-
     rewrite_in(i_Str) :: i_X ==> i_Y.
```

The goal `rewrite_in(strat) :: h(f(f(a)), f(a)) ==> i_X` returns two instantiations of `i_X`: `h(f(g(a)), f(a))` and `h(f(f(a)), g(a))`.

6. Pattern Calculi

6.1 Introduction

Pattern calculi extend the λ-calculus with pattern matching capabilities. Instead of abstracting from a variable, they permit abstractions from a pattern: a λ-term that specifies the form of the argument. The more flexible the patterns are, the more powerful the calculus is. Patterns are the most expressive ones: They can be instantiated, generated, and reduced. λ-calculus with patterns [vO90, KvOdV08], pure pattern calculus [JK06, JK09], ρ-calculus [CK01], λ-calculus with first-order constructor patterns [PW87] are some examples of the integration of pattern matching and λ-calculus.

Pattern calculi are expressive, but there is also a price to pay for that: Confluence is lost and various restrictions have to be imposed to recover it. Some restrictions concern the form of patterns (e.g., rigid pattern condition of [vO90]). Some others permit arbitrary patterns, but affect matching. [JK09] gives an abstract confluence proof, where patterns can be any term, but matching should satisfy a so called rigid matching condition. This approach is general and captures different pattern calculi, such as the pure pattern calculus, λ-calculus with patterns, and ρ-calculus. Yet another abstract proof of confluence is presented in [CF07a]. This approach can also be applied to various calculi, including λ-calculus with patterns, ρ-calculus, and a simplified version of the pure pattern calculus. All the mentioned papers consider unitary matching.

In this chapter we study confluence of pattern calculus parametrized with finitary matching. Computing finitely many matchers makes the resulting calculus non-deterministic, but rather than having a purely non-deterministic evaluation, we define a reduction which introduces a formal sum of terms, representing all possible results of a non-deterministic computation.

Our patterns, like those in [JK09], can be any term. They may contain both free and bound

variables and redexes and, hence, can be instantiated and reduced. To prove confluence, we impose restrictions on the matching function. They guarantee that no new free variable appears during reduction, and that reductions are stable both by substitution and by reduction. These properties, denoted by $\mathbf{H}_0$, $\mathbf{H}_1$, and $\mathbf{H}_2$, in their turn, make sure that the calculus is confluent. The matching function restrictions generalize the analogous ones, introduced for the unitary case in [CF07a]. The proof uses the method due to Tait and Martin-Löf [Bar84].

6.2 Core Pattern Calculus with Finitary Matching

In this section we use syntactic categories and notions introduced in Section 2.1 and in Section 2.2. We consider the binary functional symbol $\wr$ to be associative, commutative, and idempotent. Moreover, term application distributes over $\wr$ both from the left and from the right and we write ACID for this property. First-order ACID unification and matching are decidable and the rewrite system corresponding to the ACID equations is convergent [ANR04]. Normal forms of terms, obtained from this system, are called ACID *normal forms*.We consider terms in the ACID normal form with respect to $\wr$ and application. For example, the ACID normal form of the term $\lambda_{\{x,\overline{x}\}}(f\,x\wr g\,x)\,\overline{x}.\,f\,x\,(g\,y\wr g\,\overline{x}\wr g\,y)$ is a term $\lambda_{\{x,\overline{x}\}}(f\,x\,\overline{x}\wr g\,x\,\overline{x}).\,(f\,x\,g\,y\wr f\,x\,g\,\overline{x})$.

Evaluation in the core pattern calculus is given by three binary relations β_{p}, D_{l}, and D_{r} on terms, written in the form of reduction rules below. The relation β_{p} defines the way how pattern-abstractions are applied. It is parametrized by a *pattern matching function* solve, which is defined on $\wr$ free terms P, Q and returns a finite set of substitutions. We denote it by $\mathsf{solve}(P \ll_\chi Q)$. The relations D_{l} and D_{r} define how abstraction distributes over $\wr$:

$$\begin{aligned}
\beta_{\mathsf{p}}:&\quad (\lambda_\chi P.N)\,Q \to N\sigma_1 \wr \cdots \wr N\sigma_n,\ \text{where } \mathsf{solve}(P \ll_\chi Q) = \{\sigma_1,\ldots,\sigma_n\},\ n \geq 1.\\
\mathsf{D}_{\mathsf{l}}:&\quad \lambda_\chi(P_1 \wr P_2).N \to \lambda_\chi P_1.N \wr \lambda_\chi P_2.N.\\
\mathsf{D}_{\mathsf{r}}:&\quad \lambda_\chi P.(N_1 \wr N_2) \to \lambda_\chi P.N_1 \wr \lambda_\chi P.N_2.
\end{aligned}$$

A reducible expression, or *redex,* is any expression to which these rules can be applied.

A binary relation $\to_R$ on sequences is compatible if it is defined by the inference rules below.

Because of commutativity of $\wr$, one rule for the sum is sufficient.

$$\frac{M \to_R M'}{MN \to_R M'N} \qquad \frac{M \to_R M'}{NM \to_R NM'} \qquad \frac{M \to_R M'}{M\overline{x} \to_R M'\overline{x}}$$

$$\frac{M \to_R M'}{M \wr N \to_R M' \wr N} \qquad \frac{P \to_R P'}{\lambda_\chi P.N \to_R \lambda_\chi P'.N} \qquad \frac{N \to_R N'}{\lambda_\chi P.N \to_R \lambda_\chi P.N'}$$

$$\frac{M_1 \to_R M_1', \ldots, M_n \to_R M_n' \qquad n > 0}{\lceil \tilde{S}_1, M_1, \tilde{S}_2, M_2, \ldots, \tilde{S}_n, M_n, \tilde{S}_{n+1} \rceil \to_R \lceil \tilde{S}_1, M_1', \tilde{S}_2, M_2', \ldots, \tilde{S}_n, M_n', \tilde{S}_{n+1} \rceil}$$

In what follows, $\to_C$ denotes the compatible closure of the union of the relations $\beta_{\mathsf{p}}, \mathsf{D}_{\mathsf{l}}$ and D_{r}. The relation $\twoheadrightarrow_C$ denotes the reflexive and transitive closure of $\to_C$.

We say that $\to_C$ is *confluent*, if for all terms M, N, and Q, $M \twoheadrightarrow_C N$ and $M \twoheadrightarrow_C Q$ implies that there exists a term W such that $N \twoheadrightarrow_C W$ and $Q \twoheadrightarrow_C W$. Confluence of a calculus means confluence of the relation $\to_C$ for that calculus.

We are interested in particular instantiations of the solve function for studying confluence. The first such example is sequence matching:

Example 6.1 (Matching with Sequence Variables)**.** *As a concrete example, consider matching between terms P and Q together with the set of variables χ. A matching equation is written as $P \ll^?_\chi Q$. Its solution (a matcher) is a substitution σ such that $P\sigma = Q$. A matching problem Π is a finite set of matching equations. A state is a pair $\langle \Pi, \theta \rangle$ of a matching problem and a substitution. The matching procedure [Kut07, KM12] with sequence variables is reformulated here as a set of state transformations of the form $\langle \{P \ll^?_\chi Q\} \cup \Pi, \theta \rangle \rightsquigarrow \langle (\Pi \cup \Pi')\sigma, \theta\sigma \rangle$. Each of the transformations is characterized by a rule of the form $M \ll^?_\chi N \rightsquigarrow_\sigma \Pi'$. There are four rules, for deletion, decomposition, term and sequence variable elimination: (ε stands for the identity substitution. Rules are applied modulo α-equivalence.)*

$$\begin{aligned}
\mathsf{Del}: &\quad M \ll^?_\chi M \rightsquigarrow_\varepsilon \emptyset, \text{ where } \mathtt{fv}(M) \cap \chi = \emptyset.\\
\mathsf{Dec}: &\quad M_1\, M_2 \ll^?_\chi N_1\, N_2 \rightsquigarrow_\varepsilon \{M_1 \ll^?_\chi N_1, M_2 \ll^?_\chi N_2\},\\
\mathsf{TVE}: &\quad x \ll^?_\chi N \rightsquigarrow_{\{x \mapsto N\}} \emptyset, \text{ where } x \in \chi.\\
\mathsf{SVE}: &\quad M\,\overline{x} \ll^?_\chi N\,\overline{N}_1 \cdots \overline{N}_n \rightsquigarrow_{\{\overline{x} \mapsto \lceil \overline{N}_1, \ldots, \overline{N}_n \rceil\}} \{M \ll^? N\},\\
&\quad \text{where } \overline{x} \in \chi \text{ and } n \geq 0.
\end{aligned}$$

We can define $\mathsf{solve}(P \ll_\chi Q)$ *as a partial function from the triple* P, Q, χ *to a set of substitutions with the following conditions:*

C1. *If* $P = x$ *for some variable* x*,* Q *is* $\wr$*-free and does not contain bound sequence variables, then we have one of the following cases:*

- *If* $\chi = \emptyset$ *and* $Q = x$*, then* $\mathsf{solve}(P \ll_\chi Q) = \{\varepsilon\}$.
- *If* $\chi = \{x\}$*, then* $\mathsf{solve}(P \ll_\chi Q) = \{\{x \mapsto Q\}\}$.
- *If the conditions in the previous two items do not hold, then* $\mathsf{solve}(P \ll_\chi Q)$ *is undefined.*

C2. *If* $P \neq x$ *for any* x*, assume the following conditions are satisfied:*

C2.1: $\chi \subseteq \mathtt{fv}(P)$.

C2.2: P *and* Q *are redex-free and* $\wr$*-free.*

Then we transform $P \ll^?_\chi Q$ *by the rules above in all possible ways as long as possible and substitutions* σ *from the success states* $\langle \emptyset, \sigma \rangle$ *are collected in the set* $\mathcal{M}$ *of computed matchers (which is complete and finite [KM12]).*

C3. *Otherwise,* $\mathsf{solve}(P \ll_\chi Q)$ *is undefined.*

Now we give an example that explains why we do not require $\chi \subseteq \mathtt{fv}(P)$ in the term $\lambda_\chi P.N$.

Example 6.2. *As we noted in Chapter 2, in the definition of terms we do not require* $\chi \subseteq \mathtt{fv}(P)$ *in* $\lambda_\chi P.N$*. We also remarked that otherwise the set of terms would not be closed under reduction. Now, when reduction is defined, we can illustrate this point. Consider a term* $\lambda_{\{x\}}((\lambda_{\{y\}}\, y.\, z)\, x).M$*. If we reduce the redex by the* β_p *rule, we get* $\lambda_{\{x\}} z.M$*. However,* $\{x\} \subseteq \mathtt{fv}(z)$ *does not hold. Hence,* $\lambda_{\{x\}} z.M$ *would not be a term, had we defined terms* $\lambda_\chi P.N$ *under the requirement* $\chi \subseteq \mathtt{fv}(P)$.

Our goal is to study confluence of the core pattern calculus. It is tempting to leave out χ from $\lambda_\chi P.N$ and assume that the abstraction binds all variables in the pattern. However, it causes a serious problem: Bound variables can be freed and confluence does not hold even for the following simple case:

$$\lambda(\lambda x.c)y.y \to_{\beta_\mathsf{p}} \lambda c.y$$

$$\lambda(\lambda x.c)y.y =_\alpha \lambda(\lambda x.c)z.z \to_{\beta_\mathsf{p}} \lambda c.z$$

In comparison, having χ explicit, $\lambda_{\{y\}}(\lambda_{\{x\}}x.c)y.y$ reduces to two α-equivalent terms:

$$\lambda_{\{y\}}(\lambda_{\{x\}}x.c)y.y \to_{\beta_\mathsf{p}} \lambda_{\{y\}}c.y$$
$$\lambda_{\{y\}}(\lambda_{\{x\}}x.c)y.y =_\alpha \lambda_{\{z\}}(\lambda_{\{x\}}x.c)z.z \to_{\beta_\mathsf{p}} \lambda_{\{z\}}c.z$$

Example 6.3. *We show a maximal sequence of $\to_C$ reductions which involves* solve *defined in the Example 6.1 and the* ACID *property of $\wr$.*

$$\begin{array}{ll}
\lambda_{\{x\}}\, x.\, (x((\lambda_{\{\overline{x},\overline{y}\}} f\, \overline{x}\, \overline{y}.\, (g\, \overline{x}))\, (f\, a))) \to_C & \textit{(by } \beta_\mathsf{p}\textit{)} \\
\lambda_{\{x\}}\, x.\, (x\, (g \wr g\, a)) = & \textit{(by the ACID property of } \wr\textit{)} \\
\lambda_{\{x\}}\, x.\, (x\, g \wr x\, g\, a) \to_C & \textit{(by } \mathsf{D_r}\textit{)} \\
(\lambda_{\{x\}}\, x.\, (x\, g)) \wr (\lambda_{\{x\}}\, x.\, (x\, g\, a)). & \textit{(normal form)}
\end{array}$$

Note that $\wr$ is introduced at the β_p step by solve*: We have* solve$(f\, \overline{x}\, \overline{y} \ll_{\{\overline{x},\overline{y}\}} f\, a) = \{\{\overline{x} \mapsto \lceil\rceil, \overline{y} \mapsto a\}, \{\overline{x} \mapsto a, \overline{y} \mapsto \lceil\rceil\}\}$*. The* SVE *rule is responsible for that.*

6.3 Confluence of the Core Pattern Calculus

The function solve defined in Example 6.1 above leads to diverging reductions:

Example 6.4. *Let* solve *be the function based on the matching with sequence variables as defined in Example 6.1. Then $M = (\lambda_{\{\overline{z}\}} f\, \overline{z}.\, (\lambda_{\{\overline{x},\overline{y}\}} f\, \overline{x}\, \overline{y}.\, f\, \overline{x})(f\, \overline{z}))\, (f\, a\, b)$ reduces to two different normal forms:*

a) $M \to_C (\lambda_{\{\overline{z}\}} f\, \overline{z}.\, (f \wr f\, \overline{z}))\, (f\, a\, b) \to_C f \wr f\, a\, b.$

b) $M \to_C (\lambda_{\{\overline{x},\overline{y}\}} f\, \overline{x}\, \overline{y}.\, f\, \overline{x})\, (f\, a\, b) \to_C f \wr f a \wr f\, a\, b.$

Example 6.5. *This example shows that if in $(\lambda_\chi P.N)Q$ a free variable appears in Q, reduction may be non-confluent. Let* solve *be the function as defined in Example 6.1. Then the term $M = (\lambda_{\{x\}}\, x.\, (\lambda_{\{\overline{x},\overline{y}\}} f\, \overline{x}\, x\, \overline{y}.\, f\, \overline{x})(f\, a\, x\, b))\, a$ reduces to two different normal forms:*

a) $M \to_C (\lambda_{\{x\}}\, x.\, (f\, a))\, a \to_C f\, a.$

b) $M \to_C (\lambda_{\{\overline{x},\overline{y}\}} f\, \overline{x}\, a\, \overline{y}.\, (f\, \overline{x}))(f\, a\, a\, b) \to_C f \wr f a.$

Note that the condition "Q does not contain bound sequence variables" required in C1 is crucial, since without it the confluence is lost:

Example 6.6. $M = (\lambda_{\{x\}}\, x.(x\,x))((\lambda_{\{\overline{x},\overline{y}\}} f\,\overline{x}\,\overline{y}.\,(f\,\overline{x}))(f\,a))$ *reduces to two different normal forms:*

a) $M \to_C (\lambda_{\{x\}}\, x.(x\,x))\, f \wr (\lambda_{\{x\}}\, x.(x\,x))(f\,a) \to_C f\, f \wr f\, a\, (f\,a)$.

b) $M \to_C ((\lambda_{\{\overline{x},\overline{y}\}} f\,\overline{x}\,\overline{y}.\,(f\,\overline{x}))(f\,a))((\lambda_{\{\overline{x},\overline{y}\}} f\,\overline{x}\,\overline{y}.\,(f\,\overline{x}))(f\,a)) \to_C f\, f \wr f\,(f\,a) \wr f\,a\, f \wr f\,a\,(f\,a)$

Our goal is to impose restrictions on solve so that confluence is guaranteed. They will be sufficient conditions: For each solve, which satisfies them, the calculus will be confluent. For this, some more notions have to be defined.

The confluence proof will be based on the method due to Tait and Martin-Löf [Bar84] . It requires the notion of *parallel reduction* which is defined as follows:

$$\frac{}{\overline{\overline{M}} \rightrightarrows_C \overline{\overline{M}}} \qquad \frac{\overline{M}_1 \rightrightarrows_C \overline{M}'_1 \quad \dots \quad \overline{M}_n \rightrightarrows_C \overline{M}'_n}{\lceil \overline{M}_1, \dots, \overline{M}_n \rceil \rightrightarrows_C \lceil \overline{M}'_1, \dots, \overline{M}'_n \rceil} \qquad \frac{M \rightrightarrows_C M' \quad \overline{M} \rightrightarrows_C \overline{M}'}{M\,\overline{M} \rightrightarrows_C M'\,\overline{M}'}$$

$$\frac{P \rightrightarrows_C P' \quad N \rightrightarrows_C N'}{\lambda_\chi P.N \rightrightarrows_C \lambda_\chi P'.N'} \qquad \frac{P_1 \rightrightarrows_C P'_1 \quad P_2 \rightrightarrows_C P'_2 \quad N \rightrightarrows_C N'}{\lambda_\chi (P_1 \wr P_2).N \rightrightarrows_C \lambda_\chi P'_1.N' \wr \lambda_\chi P'_2.N'}$$

$$\frac{M \rightrightarrows_C M' \quad N \rightrightarrows_C N'}{M \wr N \rightrightarrows_C M' \wr N'} \qquad \frac{P \rightrightarrows_C P' \quad N_1 \rightrightarrows_C N'_1 \quad N_2 \rightrightarrows_C N'_2}{\lambda_\chi P.(N_1 \wr N_2) \rightrightarrows_C \lambda_\chi P'.N'_1 \wr \lambda_\chi P'.N'_2}$$

$$\frac{P \rightrightarrows_C P' \quad N \rightrightarrows_C N' \quad Q \rightrightarrows_C Q'}{(\lambda_\chi P.N)Q \rightrightarrows_C N'\sigma_1 \wr \cdots \wr N'\sigma_n}, \text{ where } \mathsf{solve}(P' \ll_\chi Q') = \{\sigma_1, \dots, \sigma_n\},\ n \geq 1.$$

The definition is extended to substitutions having the same domain by setting $\sigma \rightrightarrows_C \sigma'$ if for all $v \in \mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$, we have $v\sigma \rightrightarrows_C v\sigma'$.

Now we define the conditions to be imposed on solve. As already mentioned, the terms are in the ACID normal form.

$\mathbf{H}_0$: If $\sigma \in \mathsf{solve}(P \ll_\chi Q)$, then $\mathrm{dom}(\sigma) = \chi$ and $\mathtt{fv}(Ran(\sigma)) \subseteq \mathtt{fv}(Q) \cup (\mathtt{fv}(P) \setminus \chi)$.

$\mathbf{H}_1$: If $\mathsf{solve}(P \ll_\chi Q) = \{\sigma_1, \dots, \sigma_n\}$, $n \geq 1$, then for all θ with $var(\theta) \cap \chi = \emptyset$, we have $\mathsf{solve}(P\theta \ll_\chi Q\theta) = \{(\sigma_1\theta)|_\chi, \dots, (\sigma_n\theta)|_\chi\}$.

$\mathbf{H}_2$: If $\mathsf{solve}(P \ll_\chi Q) = \{\sigma_1, \dots, \sigma_n\}$, $n \geq 1$, $P \rightrightarrows_C P'$, and $Q \rightrightarrows_C Q'$, then $\mathsf{solve}(P' \ll_\chi Q') = \{\sigma'_1, \dots, \sigma'_m\}$, $m \geq 1$, and

(a) for all $1 \leq i \leq n$ there exists $1 \leq j \leq m$ such that $\sigma_i \rightrightarrows_C \sigma'_j$ and

(b) for all $1 \leq j \leq m$ there exists $1 \leq i \leq n$ such that $\sigma_i \rightrightarrows_C \sigma'_j$.

These conditions extend those for the core dynamic pattern calculus from [CF07a]. They correspond to the preservation of free variables, stability by substitution, and stability by reduction in the finitary matching case. We briefly explain the intuition behind them:

- *Preservation of free variables:* $\mathbf{H}_0$ guarantees that no new free variables appear during reduction, which is a natural requirement when one defines a higher-order calculus. When the term $(\lambda_\chi P.N)Q$ is reduced with the help of $\mathsf{solve}(P \ll_\chi Q) = \{\sigma_1, \ldots, \sigma_n\}$, we want to have $\cup_{i=1}^{n} \mathtt{fv}(N\sigma_i) \subseteq \mathtt{fv}((\lambda_\chi P.N)Q)$. It is easy to check that that the algorithms solving higher-order matching problems or matching problems in non-regular theories do not verify $\mathbf{H}_0$.

- *Stability by substitution:* When we have an application $(\lambda_\chi P.N)Q$, it may happen that Q contains free variables. Either we wait until those free variables get instantiated and then perform reduction, or we reduce $(\lambda_\chi P.N)Q$ and then instantiate the variables which come from the set $\mathtt{fv}(Q) \cup (\mathtt{fv}(P) \setminus \chi)$. The results in both cases should be the same. This is what $\mathbf{H}_1$ requires.

- *Stability by reduction:* When the application term $(\lambda_\chi P.N)Q$ is reduced, it is not necessary P and Q to be in a normal form. One can either reduce the application immediately, or first transform P and Q into P' and Q' and only afterwards try to reduce the application. This subsequent reduction of application should not fail. Even more, each substitution in $\mathsf{solve}(P' \ll_\chi Q')$ should be derivable from a substitution in $\mathsf{solve}(P \ll_\chi Q)$, and each substitution in $\mathsf{solve}(P \ll_\chi Q)$ should be reducible to a substitution in $\mathsf{solve}(P' \ll_\chi Q')$. It is not necessary these two sets to contain the same number of elements. Stability of reduction is required by $\mathbf{H}_2$.

When these conditions do not hold, confluence, in general, is not guaranteed. For instance, looking back to Example 6.4, we see that the solve there, which was based on sequence matching, violates $\mathbf{H}_1$. Just take $P = f(\overline{x}, \overline{y})$, $Q = f(\overline{z})$, and $\theta = \{\overline{z} \mapsto \lceil a, b \rceil\}$. Then we get $\mathsf{solve}(P \ll_{\{\overline{x},\overline{y}\}} Q) = \{\{\overline{x} \mapsto \epsilon, \overline{y} \mapsto \overline{z}\}, \{\overline{x} \mapsto \overline{z}, \overline{y} \mapsto \epsilon\}\}$ and $\mathsf{solve}(P \ll_{\{\overline{x},\overline{y}\}} Q\theta) = \{\{\overline{x} \mapsto \epsilon, \overline{y} \mapsto \lceil a, b \rceil\}, \{\overline{x} \mapsto a, \overline{y} \mapsto b\}, \{\overline{x} \mapsto \lceil a, b \rceil, \overline{y} \mapsto \epsilon\}\}$.

Example 6.7. *Sometimes, non-confluence is caused by so called "redex breaking". For instance,* $(\lambda_{\{x,y\}}(x\,y).(y\,x))(\lambda_{\{z\}}z.z)(f\,a)$ *is transformed either to* $(f\,a)\lambda_{\{z\}}z.z$ *or to* $(a\,f)$ *and the redex* $(\lambda_{\{z\}}z.z)(f\,a)$ *breaks. Note that in this case* $\mathbf{H}_2$ *is violated: Take* $P = (x\,y)$, $Q = (\lambda_{\{z\}}z.z)(f\,a)$. *Then* $P \rightrightarrows_C P' = P$, $Q \rightrightarrows_C Q' = (f\,a)$, $\mathsf{solve}(P \ll_{\{x,y\}} Q) = \{\{x \mapsto \lambda_{\{z\}}z.z, y \mapsto (f\,a)\}\}$, $\mathsf{solve}(P' \ll_{\{x,y\}} Q') = \{\{x \mapsto f, y \mapsto a\}\}$ *and the violation of* $\mathbf{H}_2$ *is obvious.*

Now we will show that $\mathbf{H}_0, \mathbf{H}_1$, and $\mathbf{H}_2$ are sufficient for proving confluence of our calculus. We assume that the relations $\rightarrow_C$ and $\rightrightarrows_C$ in the lemmas and in the theorem below use a solve which satisfies $\mathbf{H}_0$, $\mathbf{H}_1$, and $\mathbf{H}_2$. The notation $\wr_{i=1}^{n} M_i$ abbreviates $M_1 \wr \cdots \wr M_n$.

Lemma 6.8. *The following inclusion hold:* $\rightarrow_C \subseteq \rightrightarrows_C \subseteq \twoheadrightarrow_C$.

Proof. First we prove $\rightarrow_C \subseteq \rightrightarrows_C$.

When the reduction occurs at the head position, the inclusion follows from the definition of $\rightrightarrows_C$. Indeed, assume $M = (\lambda_\chi M_1.M_2)M_3$ reduces by β_p to $N = \wr_{i=1}^{n} M_2\sigma_i$, where $Sol(M_1 \ll_\chi M_3) = \{\sigma_1, \ldots, \sigma_n\}$. Then $M \rightrightarrows_C N$ holds, because $\rightrightarrows_C$ is reflexive. If $M = \lambda_\chi(M_1 \wr M_2).M_3$ reduces to $N = \lambda_\chi M_1.M_3 \wr \lambda M_2.M_3$ by $\mathsf{D_l}$ or $M = \lambda_\chi M_1.(M_2 \wr M_3)$ reduces to $N = \lambda_\chi M_1.M_2 \wr \lambda_\chi M_1.M_3$ by $\mathsf{D_r}$ then again by reflexivity of $\rightrightarrows_C$we have $M \rightrightarrows_C N$. If the reduction does not occur at the head position, the inclusion follows from the compatibility of $\rightrightarrows_C$.

Now we prove $\rightrightarrows_C \subseteq \twoheadrightarrow_C$, that is we show $h_1 \rightrightarrows_C h_2$ implies $h_1 \twoheadrightarrow_C h_2$ by induction on the derivation length of $h_1 \rightrightarrows_C h_2$.

- Let $\tilde{S}_1 = M = \tilde{S}_2 = N$. Then the result follows from reflexivity of $\twoheadrightarrow_C$.
- Let $\tilde{S}_1 = \lceil \overline{M}_1, \ldots, \overline{M}_n \rceil \rightrightarrows_C \tilde{S}_2 = \lceil \overline{M}'_1, \ldots, \overline{M}'_n \rceil$ with $\overline{M}_1 \rightrightarrows_C \overline{M}'_1, \ldots, \overline{M}_n \rightrightarrows_C \overline{M}'_n$. Then by the induction hypothesis (IH) we have $\overline{M}_1 \twoheadrightarrow_C \overline{M}'_1, \ldots, \overline{M}_n \twoheadrightarrow_C \overline{M}'_n$. By compatibility of $\twoheadrightarrow_C$ we can conclude $\lceil \overline{M}_1, \ldots, \overline{M}_n \rceil \twoheadrightarrow_C \lceil \overline{M}'_1, \ldots, \overline{M}'_n \rceil$.
- Let $\tilde{S}_1 = M_1 X \rightrightarrows_C \tilde{S}_2 = N_1 X$ with $M_1 \rightrightarrows_C N_1$. Then by IH we have $M_1 \twoheadrightarrow_C N_1$. By compatibility of $\twoheadrightarrow_C$ we also have $M_1 X \twoheadrightarrow_C N_1 X$.
- Let $\tilde{S}_1 = M_1 M_2 \rightrightarrows_C \tilde{S}_2 = N_1 N_2$ with $M_1 \rightrightarrows_C N_1$ and $M_2 \rightrightarrows_C N_2$. Then by IH we have $M_1 \twoheadrightarrow_C N_1$ and $M_2 \twoheadrightarrow_C N_2$. By compatibility of $\twoheadrightarrow_C$ we also have $M_1 M_2 \twoheadrightarrow_C N_1 M_2$ and $N_1 M_2 \twoheadrightarrow_C N_1 N_2$. By transitivity of $\twoheadrightarrow_C$ we finally get $M_1 M_2 \twoheadrightarrow_C N_1 N_2$.

- If $\tilde{S}_1 = \lambda_\chi M_1.M_2 \rightrightarrows_C \tilde{S}_2 = \lambda_\chi N_1.N_2$ with $M_1 \rightrightarrows_C N_1$ and $M_2 \rightrightarrows_C N_2$. Similar to the previous case.

- Let $\tilde{S}_1 = \lambda_\chi (M_1 \wr M_2).M_3 \rightrightarrows_C \tilde{S}_2 = \lambda_\chi N_1.N_3 \wr \lambda_\chi N_2.N_3$ with $M_1 \rightrightarrows_C N_1$, $M_2 \rightrightarrows_C N_2$ and $M_3 \rightrightarrows_C N_3$. Then by IH we have $M_1 \twoheadrightarrow_C N_1$, $M_2 \twoheadrightarrow_C N_2$ and $M_3 \twoheadrightarrow_C N_3$. By compatibility of $\twoheadrightarrow_C$ we have $\lambda_\chi (M_1 \wr M_2).M_3 \twoheadrightarrow_C \lambda_\chi (N_1 \wr N_2).N_3$. By $\mathsf{D_I}$ we have $\lambda_\chi (N_1 \wr N_2).N_3$ reduces to $\lambda_\chi N_1.N_3 \wr \lambda_\chi N_2.N_3$ and hence we conclude $\lambda_\chi (M_1 \wr M_2).M_3 \twoheadrightarrow_C \lambda_\chi N_1.N_3 \wr \lambda_\chi N_2.N_3$.

- If $\tilde{S}_1 = \lambda_\chi M_1.(M_2 \wr M_3) \rightrightarrows_C \tilde{S}_2 = \lambda_\chi N_1.N_2 \wr \lambda_\chi N_1.N_3$ with $M_1 \rightrightarrows_C N_1$, $M_2 \rightrightarrows_C N_2$ and $M_3 \rightrightarrows_C N_3$. Similar to the previous case.

- Let $\tilde{S}_1 = M_1 \wr M_2 \rightrightarrows_C \tilde{S}_2 = N_1 \wr N_2$ with $M_1 \rightrightarrows_C N_1$ and $M_2 \rightrightarrows_C N_2$. Then by IH we have $M_1 \twoheadrightarrow_C N_1$ and $M_2 \twoheadrightarrow_C N_2$. By compatibility of $\twoheadrightarrow_C$, $M_1 \wr M_2 \twoheadrightarrow_C N_1 \wr M_2$ and $M_2 \wr N_1 \twoheadrightarrow_C N_2 \wr N_1$. Since $\wr$ is ACID, we can write $M_2 \wr N_1 \twoheadrightarrow_C N_2 \wr N_1$ as $N_1 \wr M_2 \twoheadrightarrow_C N_1 \wr N_2$. By the transitivity of $\twoheadrightarrow_C$ we get $M_1 \wr M_2 \twoheadrightarrow_C N_1 \wr N_2$.

- Let $\tilde{S}_1 = (\lambda_\chi M_1.M_2)M_3 \rightrightarrows_C \tilde{S}_2 = \wr_{i=1}^{n} N_2\sigma_i$ with $M_1 \rightrightarrows_C N_1$, $M_2 \rightrightarrows_C N_2$, $M_3 \rightrightarrows_C N_3$ and $\mathsf{solve}(N_1 \ll_\chi N_3) = \{\sigma_1, \ldots, \sigma_n\}$. By the IH, $M_1 \twoheadrightarrow_C N_1$, $M_2 \twoheadrightarrow_C N_2$, $M_3 \twoheadrightarrow_C N_3$. By compatibility and transitivity of $\twoheadrightarrow_C$ we have $\lambda_\chi M_1.M_2 \twoheadrightarrow_C \lambda_\chi N_1.N_2$, hence, $(\lambda_\chi M_1.M_2)M_3 \twoheadrightarrow_C (\lambda_\chi N_1.N_2)N_3$. By β_p we have $(\lambda_\chi N_1.N_2)N_3 \to_C \wr_{i=1}^{n} N_2\sigma_i$. Hence, $(\lambda_\chi M_1.M_2)M_3 \twoheadrightarrow_C \wr_{i=1}^{n} N_2\sigma_i$.

□

Lemma 6.9 (Fundamental Lemma)**.** *For all terms M, M' and substitutions θ, θ' with* $\mathrm{dom}(\theta) = \mathrm{dom}(\theta')$*, if $M \rightrightarrows_C M'$ and $\theta \rightrightarrows_C \theta'$, then $M\theta \rightrightarrows_C M'\theta'$.*

Proof. By induction on the length of derivation of $M \rightrightarrows_C M'$.

- If $M = M'$, we can proceed by structural induction on M, using the definitions of substitution application and parallel reduction.

  - Let $M = x$ and $x \in \mathrm{dom}(\theta)$. By the definition of parallel reduction for substitutions we have $x\theta \rightrightarrows_C x\theta'$.
  - Let $M = y$ and $y \notin \mathrm{dom}(\theta)$. By the assumptions $\theta \rightrightarrows_C \theta'$ and $y \notin \mathrm{dom}(\theta)$, and by the definition of parallel reduction for substitutions we have $y \notin \mathrm{dom}(\theta')$. By

the definition of substitution application we also have $y\theta = y\theta' = y$ and hence by reflexivity of $\rightrightarrows_C$ we conclude $y\theta \rightrightarrows_C y\theta'$.

- Let $M = f$. By the definition of substitution application we have $f\theta = f$ and $f\theta' = f$. By reflexivity of $\rightrightarrows_C$ we can conclude $f\theta \rightrightarrows_C f\theta'$.
- Let $M = M_1\overline{x}$ with $M_1 \rightrightarrows_C M_1$. By the IH we have $M_1\theta \rightrightarrows_C M_1\theta'$. We consider two cases:
  - $\overline{x} \notin \mathrm{dom}(\theta)$. By the definition of parallel reduction for substitutions we have $\overline{x} \notin \mathrm{dom}(\theta')$. Therefore, by the definitions of substitution application we also have $(M_1\overline{x})\theta = (M_1\theta)\overline{x}$ and $(M_1\overline{x})\theta' = (M_1\theta')\overline{x}$. Since $M_1\theta \rightrightarrows_C M_1\theta'$, by the definition of parallel reduction we conclude $(M_1\overline{x})\theta \rightrightarrows_C (M_1\overline{x})\theta'$.
  - $\overline{x} \in \mathrm{dom}(\theta)$. By the definition of parallel reduction for substitutions and by the assumption $\theta \rightrightarrows_C \theta'$, we have $\overline{x}\theta \rightrightarrows_C \overline{x}\theta'$. We have two sub-cases: (1) $\overline{x}\theta = \epsilon$ and $\overline{x}\theta' = \epsilon$. Since $M_1\theta \rightrightarrows_C M_1\theta'$, by the definition of substitution application we have $(M_1\overline{x})\theta = M_1\theta \rightrightarrows_C M_1\theta' = (M_1\overline{x})\theta'$. (2) $\overline{x}\theta = (\overline{N}_1, \ldots, \overline{N}_n)$ and $\overline{x}\theta' = (\overline{N}'_1, \ldots, \overline{N}'_n)$ with $\overline{N}_1 \rightrightarrows_C \overline{N}'_1, \ldots, \overline{N}_n \rightrightarrows_C \overline{N}'_n$. Since $M_1\theta \rightrightarrows_C M_1\theta'$, by the definition of substitution application and by n time application of the definition of parallel reduction we have $(M_1\overline{x})\theta = (\cdots((M_1\theta)\overline{N}_1)\cdots\overline{N}_n) \rightrightarrows_C (\cdots((M_1\theta')\overline{N}'_1)\cdots\overline{N}'_n) = (M_1\overline{x})\theta'$.
- Let $M = M_1M_2$ with $M_1 \rightrightarrows_C M_1$ and $M_2 \rightrightarrows_C M_2$. By the IH we have $M_1\theta \rightrightarrows_C M_1\theta'$ and $M_2\theta \rightrightarrows_C M_2\theta'$. By the definitions of parallel reduction and substitution application, we conclude $(M_1M_2)\theta \rightrightarrows_C (M_1M_2)\theta'$.
- Let $M = \lambda_\chi M_1.M_2$ with $M_1 \rightrightarrows_C M_1$ and $M_2 \rightrightarrows_C M_2$. We proceed similarly to the previous case.
- Let $M_1 \wr M_2 \rightrightarrows_C M_1 \wr M_2$ with $M_1 \rightrightarrows_C M_1$ and $M_2 \rightrightarrows_C M_2$. We proceed similarly to the previous cases.
- Let $(\lambda_\chi M_1.M_2)M_3 \rightrightarrows_C (\lambda_\chi M_1.M_2)M_3$ with $M_1 \rightrightarrows_C M_1, M_2 \rightrightarrows_C M_2$ and $M_3 \rightrightarrows_C M_3$. By the IH we have $M_1\theta \rightrightarrows_C M_1\theta'$, $M_2\theta \rightrightarrows_C M_2\theta'$ and $M_3\theta \rightrightarrows_C M_3\theta'$. By the definitions of parallel reduction and substitution application, we conclude $((\lambda_\chi M_1.M_2)M_3)\theta \rightrightarrows_C ((\lambda_\chi M_1.M_2)M_3)\theta'$.
- Let $\lambda_\chi(M_1 \wr M_2).M_3 \rightrightarrows_C \lambda_\chi(M_1 \wr M_2).M_3$ with $M_1 \rightrightarrows_C M_1, M_2 \rightrightarrows_C M_2$ and $M_3 \rightrightarrows_C M_3$. By the IH we have $M_1\theta \rightrightarrows_C M_1\theta'$,$M_2\theta \rightrightarrows_C M_2\theta'$ and $M_3\theta \rightrightarrows_C M_3\theta' M_3\theta \rightrightarrows_C M_3\theta'$. By the definitions of parallel reduction and substitution application, we conclude $(\lambda M_1 \wr M_2.M_3)\theta \rightrightarrows_C (\lambda M_1 \wr M_2.M_3)\theta'$.

- If $\lambda M_1.(M_2 \wr M_3) \rightrightarrows_C \lambda M_1.(M_2 \wr M_3)$ with $M_1 \rightrightarrows_C M_1, M_2 \rightrightarrows_C M_2$ and $M_3 \rightrightarrows_C M_3$. We proceed similar to the previous cases.

- If $M \neq M'$, we need to consider cases depending on the rules of parallel reduction.
  - Let $M = M_1\overline{x} \rightrightarrows_C M' = M'_1\overline{x}$ with $M_1 \rightrightarrows_C M'_1$. By the IH we have $M_1\theta \rightrightarrows_C M'_1\theta'$. We consider two cases:
    - $\overline{x} \notin \mathrm{dom}(\theta)$. By the definition of parallel reduction for substitutions we have $\overline{x} \notin \mathrm{dom}(\theta')$. Therefore, by the definitions of substitution application we also have $(M_1\overline{x})\theta = (M_1\theta)\overline{x}$ and $(M'_1\overline{x})\theta' = (M'_1\theta')\overline{x}$. Since $M_1\theta \rightrightarrows_C M'_1\theta'$, by the definition of parallel reduction we conclude $(M_1\overline{x})\theta \rightrightarrows_C (M'_1\overline{x})\theta'$.
    - $\overline{x} \in \mathrm{dom}(\theta)$. By the definition of parallel reduction for substitutions and by the assumption $\theta \rightrightarrows_C \theta'$, we have $\overline{x}\theta \rightrightarrows_C \overline{x}\theta'$. We have two sub-cases: (1) $\overline{x}\theta = \epsilon$ and $\overline{x}\theta' = \epsilon$. Since $M_1\theta \rightrightarrows_C M'_1\theta'$, by the definition of substitution application we have $(M_1\overline{x})\theta = M_1\theta \rightrightarrows_C M'_1\theta' = (M'_1\overline{x})\theta'$. (2) $\overline{x}\theta = (\overline{N}_1, \ldots, \overline{N}_n)$ and $\overline{x}\theta' = (\overline{N'}_1, \ldots, \overline{N'}_n)$ with $\overline{N}_1 \rightrightarrows_C \overline{N'}_1, \ldots, \overline{N}_n \rightrightarrows_C \overline{N'}_n$. Since $M_1\theta \rightrightarrows_C M'_1\theta'$, by the definition of substitution application and by n time application of the definition of parallel reduction we have $(M_1\overline{x})\theta = (\cdots((M_1\theta)\overline{N}_1)\cdots\overline{N}_n) \rightrightarrows_C (\cdots((M'_1\theta')\overline{N'}_1)\cdots\overline{N'}_n) = (M_1\overline{x})\theta'$.
  - Let $M = M_1M_2 \rightrightarrows_C M' = M'_1M'_2$ with $M_1 \rightrightarrows_C M'_1$ and $M_2 \rightrightarrows_C M'_2$. By the IH we have $M_1\theta \rightrightarrows_C M'_1\theta'$ and $M_2\theta \rightrightarrows_C M'_2\theta'$. By the definitions of parallel reduction and substitution application, we conclude $(M_1M_2)\theta \rightrightarrows_C (M'_1M'_2)\theta'$.
  - Let $M = \lambda_\chi M_1.M_2 \rightrightarrows_C M' = \lambda_\chi M'_1.M'_2$ with $M_1 \rightrightarrows_C M'_1$ and $M_2 \rightrightarrows_C M'_2$. We proceed similarly to the previous case.
  - Let $M = M_1 \wr M_2 \rightrightarrows_C M' = M'_1 \wr M'_2$ with $M_1 \rightrightarrows_C M'_1$ and $M_2 \rightrightarrows_C M'_2$. We proceed similarly to the previous cases.
  - Let $M = \lambda_\chi(M_1 \wr M_2).M_3 \rightrightarrows_C M' = \lambda_\chi M'_1.M'_3 \wr \lambda_\chi M'_2.M'_3$ with $M_1 \rightrightarrows_C M'_1$, $M_2 \rightrightarrows_C M'_2$ and $M_3 \rightrightarrows_C M'_3$. By the IH we have $M_1\theta \rightrightarrows_C M'_1\theta'$,$M_2\theta \rightrightarrows_C M'_2\theta'$ and $M_3\theta \rightrightarrows_C M'_3\theta'$. By the definitions of parallel reduction and substitution application, we conclude $(\lambda_\chi(M_1 \wr M_2).M_3)\theta \rightrightarrows_C (\lambda_\chi M'_1.M'_3 \wr \lambda_\chi M'_2.M'_3)\theta'$.
  - Let $M = \lambda_\chi M_1.(M_2 \wr M_3) \rightrightarrows_C M' = \lambda_\chi M'_1.M'_2 \wr \lambda_\chi M'_1.M'_3$ with $M_1 \rightrightarrows_C M'_1, M_2 \rightrightarrows_C M'_2$ and $M_3 \rightrightarrows_C M'_3$. We proceed similarly to the previous case.
  - Let $M = (\lambda_\chi M_1.M_2)M_3 \rightrightarrows_C M' = \wr_{i=1}^{n} M'_2\sigma_i$ with $M_1 \rightrightarrows_C M'_1$, $M_2 \rightrightarrows_C M'_2$, $M_3 \rightrightarrows_C M'_3$ and $\mathsf{solve}(M'_1 \ll_\chi M'_3) = \{\sigma_1, \ldots, \sigma_n\}$ where M'_1 and M'_3 do not

contain $\wr$. By the IH, we have $M_1\theta \rightrightarrows_C M_1'\theta'$, $M_2\theta \rightrightarrows_C M_2'\theta'$ and $M_3\theta \rightrightarrows_C M_3'\theta'$. By the definition of parallel reduction we have $(\lambda M_1\theta.M_2\theta)M_3\theta \rightrightarrows_C \wr_{i=1}^{k}(M_2'\theta')\eta_i$, where $\mathsf{solve}(M_1'\theta' \ll_\chi M_3'\theta') = \{\eta_1, \ldots, \eta_k\}$. We want to show $(\wr_{i=1}^{n} M_2'\sigma_i)\theta' = \wr_{i=1}^{k}(M_2'\theta')\eta_i$. Since $\mathsf{solve}(M_1' \ll_\chi M_3') = \{\sigma_1, \ldots, \sigma_n\}$, by $\mathbf{H}_1$ we have $\mathsf{solve}(M_1'\theta' \ll_\chi M_3'\theta') = \{(\sigma_1\theta')_{|\chi}, \ldots, (\sigma_n\theta')_{|\chi}\}$ (one can always guarantee $var(\theta') \cap \chi = \emptyset$, renaming bound variables). Since $\mathsf{solve}(M_1'\theta' \ll_\chi M_3'\theta') = \{\eta_1, \ldots, \eta_k\}$, we have $\{(\sigma_1\theta')_{|\chi}, \ldots, (\sigma_n\theta')_{|\chi}\} = \{\eta_1, \ldots, \eta_k\}$ and, hence, $n \geq k$. So, we have to show that $\sigma_i\theta' = \theta'(\sigma_i\theta')_{|\chi}$ holds for all $1 \leq i \leq n$, i.e., for all v, we have to show $v\sigma_i\theta' = v\theta'(\sigma_i\theta')_{|\chi}$. (Then the claim will follow by the ACID property of $\wr$.) There are two cases:

* $v \in \chi$. Then $v \in \mathrm{dom}(\sigma_i)$ by $\mathbf{H}_0$. We show $v\theta'(\sigma_i\theta')_{|\chi} = v\sigma_i\theta'$. Since we can assume $\mathrm{dom}(\theta') \cap \chi = \emptyset$, by the substitution application we have $v\theta'(\sigma_i\theta')_{|\chi} = v(\sigma_i\theta')_{|\chi}$. Since $\mathrm{dom}(\theta') \cap \mathrm{dom}(\sigma_i) = \emptyset$ (it can always be guaranteed through a renaming of bound variables), and since $v \in \mathrm{dom}(\sigma_i)$, by the substitution composition we get $v(\sigma_i\theta')_{|\chi} = v\sigma_i\theta'$.
* $v \notin \chi$. Then $v \notin \mathrm{dom}(\sigma_i)$ by $\mathbf{H}_0$. We show $v\theta'(\sigma_i\theta')_{|\chi} = v\sigma_i\theta'$. By the substitution application, $v\sigma_i\theta' = v\theta'$. Since $\mathrm{dom}(\sigma_i\theta')_{|\chi} = \chi$, we can rename bound variables so that $\mathrm{dom}(\sigma_i\theta')_{|\chi} \cap \mathtt{fv}(Ran(\theta')) = \emptyset$. By substitution application and composition, we get $v\theta'(\sigma_i\theta')_{|\chi} = v\theta'$.

□

Lemma 6.10 (Diamond Property)**.** *For all M, N, and Q, if $M \rightrightarrows_C N$ and $M \rightrightarrows_C Q$, then there exists W such that $N \rightrightarrows_C W$ and $Q \rightrightarrows_C W$.*

Proof. The proof is by induction on the structure of M.

- Let $M = x$ or $M = f$. Since we necessarily have $M = N = Q$, the result holds trivially.
- Let M is an abstraction. We consider the following cases:
  - If $M = \lambda_\chi M_1.M_2$ then we have $N = \lambda_\chi N_1.N_2$ with $M_1 \rightrightarrows_C N_1, M_2 \rightrightarrows_C N_2$ and $Q = \lambda_\chi Q_1.Q_2$ with $M_1 \rightrightarrows_C Q_1, M_2 \rightrightarrows_C Q_2$. Applying the IH to M_1 and to M_2 we get that there exist two terms W_1 and W_2 such that $N_1 \rightrightarrows_C W_1, Q_1 \rightrightarrows_C W_1$ and $N_2 \rightrightarrows_C W_2, Q_2 \rightrightarrows_C W_2$. So, by the definition of parallel reduction, we can conclude $N = \lambda_\chi N_1.N_2 \rightrightarrows_C \lambda_\chi W_1.W_2 = W$ and $Q = \lambda_\chi Q_1.Q_2 \rightrightarrows_C \lambda_\chi W_1.W_2 = W$.

– If $M = \lambda_\chi(M_1 \wr M_2).M_3$ then we have the following possible cases:

  a) $N = \lambda_\chi N_1.N_3 \wr \lambda_\chi N_2.N_3$ with $M_1 \rightrightarrows_C N_1, M_2 \rightrightarrows_C N_2, M_3 \rightrightarrows_C N_3$ and $Q = \lambda_\chi Q_1.Q_3 \wr \lambda_\chi Q_2.Q_3$ with $M_1 \rightrightarrows_C Q_1, M_2 \rightrightarrows_C Q_2, M_3 \rightrightarrows_C Q_3$. Applying the IH to M_1, M_2 and M_3 we get that there exist terms W_1, W_2 and W_3 such that $N_1 \rightrightarrows_C W_1, Q_1 \rightrightarrows_C W_1, N_2 \rightrightarrows_C W_2, Q_2 \rightrightarrows_C W_2$ and $N_3 \rightrightarrows_C W_3, Q_3 \rightrightarrows_C W_3$. So, by the definition of parallel reduction, we can conclude $N = \lambda_\chi N_1.N_3 \wr \lambda_\chi N_2.N_3 \rightrightarrows_C \lambda_\chi W_1.W_3 \wr \lambda_\chi W_2.W_3 = W$ and $Q = \lambda_\chi Q_1.Q_3 \wr \lambda_\chi Q_2.Q_3 \rightrightarrows_C \lambda_\chi W_1.W_3 \wr \lambda_\chi W_2.W_3 = W$.

  b) $N = \lambda_\chi(N_1 \wr N_2).N_3$ with $M_1 \rightrightarrows_C N_1, M_2 \rightrightarrows_C N_2, M_3 \rightrightarrows_C N_3$ and $Q = \lambda_\chi Q_1.Q_3 \wr \lambda_\chi Q_2.Q_3$ with $M_1 \rightrightarrows_C Q_1, M_2 \rightrightarrows_C Q_2, M_3 \rightrightarrows_C Q_3$. Applying the IH to M_1, M_2 and M_3 we get that there exist terms W_1, W_2 and W_3 such that $N_1 \rightrightarrows_C W_1, Q_1 \rightrightarrows_C W_1, N_2 \rightrightarrows_C W_2, Q_2 \rightrightarrows_C W_2$ and $N_3 \rightrightarrows_C W_3, Q_3 \rightrightarrows_C W_3$. So, by the definition of parallel reduction we can conclude $N = \lambda_\chi(N_1 \wr N_2).N_3 \rightrightarrows_C \lambda_\chi W_1.W_3 \wr \lambda_\chi W_2.W_3 = W$ and $Q = \lambda_\chi Q_1.Q_3 \wr \lambda_\chi Q_2.Q_3 \rightrightarrows_C \lambda_\chi W_1.W_3 \wr \lambda_\chi W_2.W_3 = W$. The proof proceeds similarly, when $N = \lambda_\chi N_1.N_3 \wr \lambda_\chi N_2.N_3$ and $Q = \lambda_\chi Q_1 \wr Q_2.Q_3$.

  c) The case for $N = \lambda_\chi(N_1 \wr N_2).N_3$ and $Q = \lambda_\chi(Q_1 \wr Q_2).Q_3$ is covered by the case for $M = \lambda_\chi M_1.M_2$.

– If $M = \lambda_\chi M_1.(M_2 \wr M_3)$ then we have the following cases:

  a) $N = \lambda_\chi N_1.N_2 \wr \lambda_\chi N_1.N_3$ with $M_1 \rightrightarrows_C N_1, M_2 \rightrightarrows_C N_2, M_3 \rightrightarrows_C N_3$ and $Q = \lambda_\chi Q_1.Q_2 \wr \lambda_\chi Q_1.Q_3$ with $M_1 \rightrightarrows_C Q_1, M_2 \rightrightarrows_C Q_2, M_3 \rightrightarrows_C Q_3$. Applying the IH to M_1, M_2 and M_3 we get that there exist terms W_1, W_2 and W_3 such that $N_1 \rightrightarrows_C W_1, Q_1 \rightrightarrows_C W_1, N_2 \rightrightarrows_C W_2, Q_2 \rightrightarrows_C W_2$ and $N_3 \rightrightarrows_C W_3, Q_3 \rightrightarrows_C W_3$. So, by the definition of parallel reduction we can conclude $N = \lambda_\chi N_1.N_2 \wr \lambda_\chi N_1.N_3 \rightrightarrows_C \lambda_\chi W_1.W_2 \wr \lambda_\chi W_1.W_3 = W$ and $Q = \lambda_\chi Q_1.Q_2 \wr \lambda_\chi Q_1.Q_3 \rightrightarrows_C \lambda_\chi W_1.W_2 \wr \lambda_\chi W_1.W_3 = W$.

  b) $N = \lambda_\chi N_1.(N_2 \wr N_3)$ with $M_1 \rightrightarrows_C N_1, M_2 \rightrightarrows_C N_2, M_3 \rightrightarrows_C N_3$ and $Q = \lambda_\chi Q_1.Q_2 \wr \lambda_\chi Q_1.Q_3$ with $M_1 \rightrightarrows_C Q_1, M_2 \rightrightarrows_C Q_2, M_3 \rightrightarrows_C Q_3$. Applying the IH to M_1, M_2 and M_3 we get that there exist terms W_1, W_2 and W_3 such that $N_1 \rightrightarrows_C W_1, Q_1 \rightrightarrows_C W_1, N_2 \rightrightarrows_C W_2, Q_2 \rightrightarrows_C W_2$ and $N_3 \rightrightarrows_C W_3, Q_3 \rightrightarrows_C W_3$. So, by the definition of parallel reduction, we can conclude $N = \lambda_\chi N_1.(N_2 \wr N_3) \rightrightarrows_C \lambda_\chi W_1.W_2 \wr \lambda_\chi W_1.W_3 = W$ and $Q = \lambda_\chi Q_1.Q_2 \wr \lambda_\chi Q_1.Q_3 \rightrightarrows_C \lambda_\chi W_1.W_2 \wr \lambda_\chi W_1.W_3 = W$. The proof proceeds similarly, when $N = \lambda_\chi N_1.N_3 \wr \lambda_\chi N_2.N_3$ and $Q = \lambda_\chi Q_1.(Q_2 \wr Q_3)$.

c) The case for $N = \lambda_\chi N_1.(N_2 \wr N_3)$ and $Q = \lambda_\chi Q_1.(Q_2 \wr Q_3)$ is covered by the case for $M = \lambda_\chi M_1.M_2$.

- Let M is a term application, then we have following cases:

  – If $M = M_1\overline{x}$ and $N = N_1\overline{x}$ and $Q = Q_1\overline{x}$ with $M_1 \rightrightarrows_C N_1$ and $M_1 \rightrightarrows_C Q_1$ then by IH to M_1 there exist W_1 such that $N_1 \rightrightarrows_C W_1, Q_1 \rightrightarrows_C W_1$. By the definition of parallel reduction, we can conclude $N_1\overline{x} \rightrightarrows_C W_1\overline{x}$ and $Q_1\overline{x} \rightrightarrows_C W_1\overline{x}$.

  – If $M = M_1M_2$ and $N = N_1N_2$ and $Q = Q_1Q_2$ with $M_1 \rightrightarrows_C N_1$, $M_2 \rightrightarrows_C N_2$ and $M_1 \rightrightarrows_C Q_1$, $M_2 \rightrightarrows_C Q_2$, then by IH to M_1 and M_2 we get that there exist two terms W_1 and W_2 such that $N_1 \rightrightarrows_C W_1, Q_1 \rightrightarrows_C W_1$ and $N_2 \rightrightarrows_C W_2, Q_2 \rightrightarrows_C W_2$. We conclude that $N = N_1N_2 \rightrightarrows_C W_1W_2$ and $Q = Q_1Q_2 \rightrightarrows_C W_1W_2$.

  – If $M = (\lambda_\chi M_1.M_2)M_3$ we consider the following cases:

    a) $N = \wr_{i=1}^{n} N_2\theta_i$ and $Q = \wr_{i=1}^{k} Q_2\theta'_i$ with

    * $M_1 \rightrightarrows_C N_1$ and $M_1 \rightrightarrows_C Q_1$
    * $M_2 \rightrightarrows_C N_2$ and $M_2 \rightrightarrows_C Q_2$
    * $M_3 \rightrightarrows_C N_3$ and $M_3 \rightrightarrows_C Q_3$
    * $\mathsf{solve}(N_1 \ll_\chi N_3) = \{\theta_1, \ldots, \theta_n\}$
    * $\mathsf{solve}(Q_1 \ll_\chi Q_3) = \{\theta'_1, \ldots, \theta'_k\}$

    then we apply the IH on M_1, M_2 and M_3 and get

    * $N_1 \rightrightarrows_C W_1$ and $Q_1 \rightrightarrows_C W_1$
    * $N_2 \rightrightarrows_C W_2$ and $Q_2 \rightrightarrows_C W_2$
    * $N_3 \rightrightarrows_C W_3$ and $Q_3 \rightrightarrows_C W_3$

    Applying $\mathbf{H}_2$ to N_1 and N_3 we get $\mathsf{solve}(W_1 \ll_\chi W_3) = \{\theta''_1, \ldots, \theta''_m\}$ where each θ_i reduces with the parallel reduction to at least one θ''_j and every θ''_j is the parallel reduction of some θ_i. Then by applying Lemma 6.9 we conclude that $N_2\theta_i \rightrightarrows_C W_2\theta''_j$. By the definition of parallel reduction and by the ACID property of $\wr$, we have $\wr_{i=1}^{n} N_2\theta_i \rightrightarrows_C \wr_{i=1}^{n} W_2\theta''_i$. Again by applying $\mathbf{H}_2$ to Q_1 and Q_3 we have $\mathsf{solve}(W_1 \ll_\chi W_3) = \{\theta''_1, \ldots, \theta''_m\}$ with each θ_i reduces with parallel reduction to at least one θ''_j and every θ''_j is the parallel reduction of some θ_i. By Lemma 6.9, we conclude that (i) for each $1 \leq i \leq n$ there exists $1 \leq j \leq m$ such that $Q_2\theta'_i \rightrightarrows_C W_2\theta''_j$, and (ii) for each $1 \leq j \leq m$ there exists $1 \leq i \leq n$ such that $Q_2\theta'_i \rightrightarrows_C W_2\theta''_j$. Therefore, by the definition of parallel reduction and by the ACID property of $\wr$, we have $\wr_{i=1}^{k} Q_2\theta'_i \rightrightarrows_C \wr_{i=1}^{n} W_2\theta''_i$.

b) $N = \wr_{i=1}^{n} N_2\theta_i$ and $Q = (\lambda_\chi Q_1.Q_2)Q_3$ where Q_3 is $\wr$-free with

* $M_1 \rightrightarrows_C N_1$ and $M_1 \rightrightarrows_C Q_1$,
* $M_2 \rightrightarrows_C N_2$ and $M_2 \rightrightarrows_C Q_2$,
* $M_3 \rightrightarrows_C N_3$ and $M_3 \rightrightarrows_C Q_3$,
* $\mathsf{solve}(N_1 \ll_\chi N_3) = \{\theta_1, \ldots, \theta_n\}$.

Then we apply the IH to M_1, M_2 and M_3 and get

* $N_1 \rightrightarrows_C W_1$ and $Q_1 \rightrightarrows_C W_1$,
* $N_2 \rightrightarrows_C W_2$ and $Q_2 \rightrightarrows_C W_2$,
* $N_3 \rightrightarrows_C W_3$ and $Q_3 \rightrightarrows_C W_3$.

Applying $\mathbf{H}_2$ to N_1 and N_3 we get $\mathsf{solve}(W_1 \ll_\chi W_3) = \{\theta''_1, \ldots, \theta''_m\}$, where each θ_i reduces by the parallel reduction to at least one θ''_j, and each θ''_j is the parallel reduction of some θ_i. By Lemma 6.9, we get (i) for each $1 \leq i \leq n$ there exists $1 \leq j \leq m$ such that $N_2\theta_i \rightrightarrows_C W_2\theta''_j$, and (ii) for each $1 \leq j \leq m$ there exists $1 \leq i \leq n$ such that $N_2\theta_i \rightrightarrows_C W_2\theta''_j$. By the definition of parallel reduction and by the ACID property of $\wr$, we have $\wr_{i=1}^{n} N_2\theta_i \rightrightarrows_C \wr_{i=1}^{m} W_2\theta''_i$. On the other hand, we have $(\lambda_\chi Q_1.Q_2)Q_3 \rightrightarrows_C \wr_{i=1}^{m} W_2\theta''_i$. The proof proceeds similarly, when $N = (\lambda_\chi N_1.N_2)N_3$ and $Q = \wr_{i=1}^{n} Q_2\theta_i$, where N_3 is $\wr$-free.

c) $N = \wr_{i=1}^{n} N_2\theta_i$ and $Q = (\lambda_\chi Q_1.Q_2)Q'_3 \wr (\lambda_\chi Q_1.Q_2)Q''_3$ where $Q'_3 \wr Q''_3 = Q_3$ with

* $M_1 \rightrightarrows_C N_1$ and $M_1 \rightrightarrows_C Q_1$,
* $M_2 \rightrightarrows_C N_2$ and $M_2 \rightrightarrows_C Q_2$,
* $M_3 \rightrightarrows_C N_3$ and $M_3 \rightrightarrows_C Q_3$,
* $\mathsf{solve}(N_1 \ll_\chi N_3) = \{\theta_1, \ldots, \theta_n\}$.

Then we apply the IH to M_1, M_2 and M_3 and get

* $N_1 \rightrightarrows_C W_1$ and $Q_1 \rightrightarrows_C W_1$,
* $N_2 \rightrightarrows_C W_2$ and $Q_2 \rightrightarrows_C W_2$,
* $N_3 \rightrightarrows_C W_3$ and $Q_3 \rightrightarrows_C W_3$.

Applying $\mathbf{H}_2$ to N_1 and N_3 we get $\mathsf{solve}(W_1 \ll_\chi W_3) = \{\theta''_1, \ldots, \theta''_m\}$, where each θ_i reduces by the parallel reduction to at least one θ''_j and every θ''_j is the parallel reduction of some θ_i. By Lemma 6.9, we get (i) for each $1 \leq i \leq n$ there exists $1 \leq j \leq m$ such that $N_2\theta_i \rightrightarrows_C W_2\theta''_j$, and (ii) for each $1 \leq j \leq m$ there exists $1 \leq i \leq n$ such that $N_2\theta_i \rightrightarrows_C W_2\theta''_j$. By the definition of parallel reduction and by the ACID property of $\wr$, we have

$\wr_{i=1}^{n} N_2\theta_i \rightrightarrows_C \wr_{i=1}^{m} W_2\theta''_i$. On the other hand, we know $Q'_3 \wr Q''_3 = Q_3 \rightrightarrows_C W_3$, which means there exist W'_3 and W''_3 such that $Q'_3 \rightrightarrows_C W'_3$, $Q''_3 \rightrightarrows_C W''_3$, and $W'_3 \wr W''_3 = W_3$. But since W_3 is $\wr$-free, we have $W_3 = W'_3 = W''_3$. Hence, we can conclude that $(\lambda_\chi Q_1.Q_2)Q'_3 \wr (\lambda_\chi Q_1.Q_2)Q''_3 \rightrightarrows_C (\lambda_\chi W_1.W_2)W'_3 \wr (\lambda_\chi W_1.W_2)W''_3 = (\lambda_\chi W_1.W_2)W_3 \rightrightarrows_C \wr_{i=1}^{m} W_2\theta''_i$.

d) The case when $N = (\lambda_\chi N_1.N_2)N_3$ and $Q = (\lambda_\chi Q_1.Q_2)Q_3$ is covered by the case $M = M_1M_2$.

- Let $M = M_1 \wr M_2$ and $N = N_1 \wr N_2$ and $Q = Q_1 \wr Q_2$ with $M_1 \rightrightarrows_C N_1, M_2 \rightrightarrows_C N_2$ and $M_1 \rightrightarrows_C Q_1, M_2 \rightrightarrows_C Q_2$, then applying IH to M_1 and M_2 we get that there exist two terms W_1 and W_2 such that $N_1 \rightrightarrows_C W_1$, $Q_1 \rightrightarrows_C W_1$ and $N_2 \rightrightarrows_C W_2$, $Q_2 \rightrightarrows_C W_2$. We conclude that $N = N_1 \wr N_2 \rightrightarrows_C W_1 \wr W_2$ and $Q = Q_1 \wr Q_2 \rightrightarrows_C W_1 \wr W_2$.

□

Theorem 6.11. *The core pattern calculus with finitary matching is confluent if* solve *satisfies* $\mathbf{H}_0$, $\mathbf{H}_1$, *and* $\mathbf{H}_2$.

Proof. We need to show that the relation $\twoheadrightarrow_C$ is confluent. From Lemma 6.8 we get that the reflexive-transitive closure of the relations $\rightarrow_C$ and $\rightrightarrows_C$ are the same. By Lemma 6.10, the relation $\rightrightarrows_C$ has the diamond property and, therefore, its reflexive-transitive closure is confluent. Hence, $\twoheadrightarrow_C$ is confluent. □

Remark 6.12. *In $\lambda_\chi P.M$, we say that the pattern P is* linear *if it is a linear term with respect to χ. In pattern calculi, non-linear patterns usually lead to non-confluent reductions, which is a variation of Klop's counterexample [Klo80]. Since we do not impose any syntactic restrictions on patterns in general, Klop's counterexample can be encoded in our calculus as it is done for the ρ-calculus [Wac03]. However, the conditions $\mathbf{H}_0, \mathbf{H}_1$ and $\mathbf{H}_2$ are strong enough to restrict non-joinable reductions of the given term. For example, consider* $\mathsf{solve}(f\,x\,x \ll_{\{x\}} f\,M\,M)$ *where M is a redex. If we reduce M to $M' \neq M$ by parallel reduction, the condition $\mathbf{H}_2$ is violated: We have $f\,M\,M \rightrightarrows_C f\,M'\,M$ and $f\,x\,x \rightrightarrows_C f\,x\,x$.*

6.4 Instantiations of solve

Sequence Matching

We now define solve slightly differently from that in Example 6.1. In particular, we add one more item in the condition C2:

C2.3. Q does not contain free variables.

Now we show that this solve satisfies $\mathbf{H}_0$, $\mathbf{H}_1$, and $\mathbf{H}_2$ (and, thus, makes the calculus confluent). They trivially hold when $\mathsf{solve}(P \ll_\chi Q)$ is undefined or is empty. Otherwise, $\mathbf{H}_0$ follows from the C1, C2.1 and the conditions that require that TVE and SVE rules are applied if x and $\overline{x}$ are from χ. If $\mathsf{solve}(P \ll_\chi Q)$ is defined, then both P and Q are redex- and $\wr$-free, or $P = x$ and Q is $\wr$-free. If P, Q are both redex- and $\wr$-free then $\mathbf{H}_2$ trivially holds. If $P = x$ then by condition C1 we know Q does not contain bound sequence variables and is $\wr$-free, which means that Q necessarily reduces by parallel reduction to a $\wr$-free term Q'. Then we have $\mathsf{solve}(x \ll_{\{x\}} Q) = \{\{x \mapsto Q\}\}$ and $\mathsf{solve}(x \ll_{\{x\}} Q') = \{\{x \mapsto Q'\}\}$. By the definition of parallel reduction for substitutions, we have $\{x \mapsto Q\} \rightrightarrows_C \{x \mapsto Q'\}$ and, hence, $\mathbf{H}_2$ holds. The condition $\mathbf{H}_1$ holds trivially.

Example 6.13. *Let* solve *be the function based on the matching with sequence variables as defined in this section, then* solve *makes the first reduction in Example 6.4 impossible, i.e.* $M = (\lambda_{\{\overline{z}\}} f\,\overline{z}.\,(\lambda_{\{\overline{x},\overline{y}\}} f\,\overline{x}\,\overline{y}.\,(f\,\overline{x}))(f\,\overline{z}))\,(f\,a\,b)$ *reduces to the normal form only via this reduction:* $M \to_C (\lambda_{\{\overline{x},\overline{y}\}} f\,\overline{x}\,\overline{y}.\,(f\,\overline{x}))\,(f\,a\,b) \to_C f \wr f\,a \wr f\,a\,b$.

Remark 6.14. *Sequence matching, as we defined it, permits an encoding of untyped* λ*-calculus in our core pattern calculus:* $\mathsf{solve}(P \ll_\chi Q)$ *always succeeds when* $P = x$ *with* $\chi = \{x\}$ *and* Q *is a* λ*-term with the computed matcher* $\{x \mapsto Q\}$*. Hence, for each* β*-reduction of a* λ*-term there exists a* $\to_C$ *reduction of the corresponding pattern-term.*

A term is a *rigid value* if it is a closed term that contains no redexes. For instance, the term Q that satisfies the conditions C2.2 and C2.3 of the sequence matching function defined above is a rigid value. Given a reduction $(\lambda_\chi P.N)Q \to_{\beta_\mathsf{p}} N\sigma_1 \wr \cdots \wr N\sigma_n$, we say that it is performed by the *call-by-rigid-value* strategy, if the term Q is a rigid value.

Remark 6.15. *The sequence matching function* solve *above is undefined when* P *is not a variable and* Q *is not a rigid value. Reductions performed based on that* solve *are call-by-*

rigid-value reductions for non-variable patterns. Note that those patterns do not contain redexes, but can be non-linear (in the sense of pattern linearity as defined in Remark 6.12) above. Nevertheless, Klop's counterexample to confluence can not be encoded due to the call-by-rigid-value strategy imposed by the conditions C2.2 *and* C2.3.

Unordered Sequence Matching

As we have seen in Chapter 3 some elements of $\mathcal{F}$ can have unordered property and the meta symbol f_{u} is used to vary over those elements. Matching between terms where some symbols may have unordered property generalizes the commutative matching and is called unordered sequence matching. To deal with unordered function symbols in a matching equation, the sequence matching algorithm given in Example 6.1 can be reformulated in following way:

$$
\begin{aligned}
\mathsf{Del}: &\ M \ll^?_\chi M \rightsquigarrow_\varepsilon \emptyset, \text{ where } \mathtt{fv}(M) \cap \chi = \emptyset.\\
\mathsf{Dec}: &\ M_1\, M_2 \ll^?_\chi N_1\, N_2 \rightsquigarrow_\varepsilon \{M_1 \ll^?_\chi N_1, M_2 \ll^?_\chi N_2\},\\
&\ \text{where } N_1 \text{ does not have a form } f_{\mathsf{u}}\, \overline{N}_1 \cdots \overline{N}_n.\\
\mathsf{FSU}: &\ f_{\mathsf{u}}\overline{M}_1 \dots \overline{M}_m \ll^?_\chi f_{\mathsf{u}}\, \overline{N}_1 \dots \overline{N}_n \rightsquigarrow_\varepsilon \{f_{\mathsf{o}}\overline{M}_1 \dots \overline{M}_m \ll^?_\chi f_{\mathsf{o}}\, \overline{N}_{\pi(1)} \dots \overline{N}_{\pi(n)}\},\\
&\ \text{where } m \text{ or } n \text{ is not } 0,\ \pi \text{ is a permutation of } (1,\dots,n), \text{ and}\\
&\ f_{\mathsf{o}} \text{ is some ordered function symbol.}\\
\mathsf{TVU}: &\ x\overline{M}_1 \dots \overline{M}_m \ll^?_\chi f_{\mathsf{u}}\, \overline{N}_1 \dots \overline{N}_n \rightsquigarrow_{\{x \mapsto f_{\mathsf{u}}\}} \{f_{\mathsf{o}}\overline{M}_1 \dots \overline{M}_m \ll^?_\chi f_{\mathsf{o}}\, \overline{N}_{\pi(1)} \dots \overline{N}_{\pi(n)}\},\\
&\ \text{where } m > 0,\ \pi \text{ is a permutation of } (1,\dots,n) \text{ and } f_{\mathsf{o}} \text{ is some ordered}\\
&\ \text{function symbol.}\\
\mathsf{TVE}: &\ x \ll^?_\chi N \rightsquigarrow_{\{x \mapsto N\}} \emptyset, \text{ where } x \in \chi.\\
\mathsf{SVE}: &\ M\, \overline{x} \ll^?_\chi N\, \overline{N}_1 \cdots \overline{N}_n \rightsquigarrow_{\{\overline{x} \mapsto \lceil \overline{N}_1, \dots, \overline{N}_n \rceil\}} \{M \ll^? N\}, \text{ where } \overline{x} \in \chi \text{ and } n \geq 0.
\end{aligned}
$$

solve here is defined slightly differently than in Example 6.1. The conditions C1–C3 are inherited from the sequence matching function in that example and, in addition, one more restriction is added to C1: Q should not contain unordered symbols. With the arguments similar to that for the sequence matching above we can show that the solve defined for the unordered sequence matching verifies $\mathbf{H}_0, \mathbf{H}_1$ and $\mathbf{H}_2$.

Remark 6.16. *Unordered sequence matching like free sequence matching verifies* $\mathbf{H}_0, \mathbf{H}_1$, *and* $\mathbf{H}_2$, *when the reduction is applied with the call-by-rigid-value strategy. Hence, the core pattern calculus with the unordered sequence matching is confluent and generalizes the*

confluence result for the (left-)distributive ρ-calculus, formulated for finitary matching with algebraic patterns and the call-by-rigid-value strategy in, e.g., [Fau07].

Example 6.17. *Let* $M = (\lambda_{\{\overline{z}\}} f_{\mathsf{u}}\, \overline{z}.\, (\lambda_{\{\overline{x},\overline{y}\}} f_{\mathsf{u}}\, \overline{x}\, \overline{y}.\, (g\, \overline{x}))(f_{\mathsf{u}}\, \overline{z}))\, (f_{\mathsf{u}}\, a\, b)$ *and* solve *be the function based on the unordered sequence matching as defined in this section. Then* M *reduces in only one way:* $M \to_C (\lambda_{\{\overline{x},\overline{y}\}} f_{\mathsf{u}}\, \overline{x}\, \overline{y}.\, (g\, \overline{x}))\, (f_{\mathsf{u}}\, a\, b) \to_C g \wr g\, a \wr g\, a\, b \wr g\, b \wr g\, b\, a$.

Sequence Matching with Linear Algebraic Patterns.

The free and unordered sequence matching functions above require Q to be a rigid value in $\mathsf{solve}(P \ll_\chi Q)$ (unless P is a variable). We now try to relax this restriction on Q. Instead, the form of patterns will be constrained.

Algebraic terms are defined by the grammar $A := x \mid f q_1 \cdots q_n$, $n \geq 0$, where q is defined as $q := \overline{x} \mid A$.

In a term $\lambda_\chi P.M$, the pattern P is an *algebraic pattern* if it is an algebraic term. It is a *linear algebraic pattern* if it is algebraic and linear with respect to χ.

For the definition of $\mathsf{solve}(P \ll^?_\chi Q)$, we take the solve for sequence matching, but make three modifications: Q may contain redexes, Q may not contain bound sequence variable occurrences, and P should be a linear algebraic term with respect to χ. Such solving problems arise when one tries to reduce $(\lambda_\chi P.M)Q$, where P is a linear algebraic pattern in $\lambda_\chi P.M$, and Q contains no bound sequence variables.

To make the conditions for solve more precise, now they look as follows:

C0. Q is $\wr$-free and does not contain bound sequence variables.

C1. If $P = x$ for some variable x, then we have one of the following cases:

- If $\chi = \emptyset$ and $Q = x$, then $\mathsf{solve}(P \ll_\chi Q) = \{\varepsilon\}$.
- If $\chi = \{x\}$, then $\mathsf{solve}(P \ll_\chi Q) = \{\{x \mapsto Q\}\}$.
- If the conditions in the previous two items do not hold, then $\mathsf{solve}(P \ll_\chi Q)$ is undefined.

C2. If $P \neq x$ for any x, assume the following conditions are satisfied:

C2.1: $\chi \subseteq \mathtt{fv}(P)$.

C2.2: P is a linear algebraic term with respect to χ.

C2.3. Q does not contain free variables.

Then we transform $P \ll^?_\chi Q$ by the rules above in all possible ways as long as possible and substitutions σ from the success states $\langle \emptyset, \sigma \rangle$ are collected in the set $\mathcal{M}$ of computed matchers (which is complete and finite [KM12]).

C3. Otherwise, $\mathsf{solve}(P \ll_\chi Q)$ is undefined.

Then the conditions $\mathbf{H}_0$ and $\mathbf{H}_1$ remain valid. For $\mathbf{H}_2$, note that even if Q may contain redexes, they are never propagated to the left-hand side of matching problems (due to linearity of patterns), and the redexes are not broken between pattern variables (due to the algebraic property). Linearity of the pattern P together with the restriction of bound sequence variable occurrences in Q guarantee that from solvable problems one can not obtain unsolvable ones by parallel reduction: Q never reduces to a term that contains the $\wr$ operator and we can not have cases similar to that in Remark 6.12 above. Hence, in the matching process, the left-hand sides of matching equations remain algebraic patterns (and, therefore, redex-free and $\wr$-free), the redexes from Q are preserved in the ranges of the matchers (and do not reduce to terms with $\wr$), and the instantiations of free variables in the pattern are not considered for matching again (since the variables occur only once).

These conditions imply $\mathbf{H}_2$. Sequence matching with algebraic patterns is an example of the matching function that does not require the call-by-value strategy.

6.5 Pattern Calculus with Finitary Matching

Pattern calculus with finitary matching extends the core pattern calculus by a set ξ of rewrite rules. This set, like solve, is a parameter of the calculus and serves the purpose of expressing some pattern-based calculi as instances of the pattern calculus with finitary matching. The core calculus might not be expressive enough for them.

The proof of confluence of the pattern calculus with finitary matching is analogous to the proof of the same result for the pattern calculus with unitary matching from [Fau07]. It uses the Yokouchi-Hikita's lemma [YH90, CHL96], which states that if R_1 and R_2 are two relations defined on the same set $\mathcal{T}(\mathcal{F} \cup \{\wr\}, \mathcal{V})$ of terms such that (a) R_1 is confluent and strongly normalizing, (b) R_2 verifies the diamond property, and (c) R_1 and R_2 form the

Yokouchi-Hikita's diagram, then the relation $R_1^* R_2 R_1^*$ is confluent. What it means R_1 and R_2 to form the Yokouchi-Hikita's diagram, is that for all $M, N, Q \in \mathcal{T}(\mathcal{F} \cup \{\wr\}, \mathcal{V})$ such that $M \to_{R_1} N$ and $M \to_{R_2} Q$ there should exist $W \in \mathcal{T}(\mathcal{F} \cup \{\wr\}, \mathcal{V})$ such that $N \to_{R_1^* R_2 R_1^*} W$ and $Q \to_{R_1^*} W$.

The confluence theorem is now formulated as follows:

Theorem 6.18. *The pattern calculus with finitary matching is confluent if*

- solve *satisfies* $\mathbf{H}_0, \mathbf{H}_1, \mathbf{H}_2$,
- *the set* ξ *of reduction rules is strongly normalizing and confluent, and*
- *the relations* $\rightrightarrows_C$ *and* ξ *satisfy Yokouchi-Hikita's diagram.*

Proof. Using the same argument as in the proof of Theorem 6.11 in [Fau07]: Apply Yokouchi-Hikita's lemma with R_1 being the compatible relation induced by ξ and with R_2 being $\rightrightarrows_C$. The diamond property of $\rightrightarrows_C$ has been proved in Lemma 6.10. To conclude the proof, as a consequence of Lemma 6.8, we can remark that the reflexive-transitive closure of $\to_{C \cup \xi}$ and $\twoheadrightarrow_\xi \rightrightarrows_C \twoheadrightarrow_\xi$ are equal. □

7. Conclusion

In this thesis we showed that incorporating sequence and context variables in declarative programing yields flexible, expressive framework with interesting and useful applications. Below we make the contributions more detailed and discuss ideas for future work.

7.1 Constraint Solving

We contributed with a new algorithm for solving equational and membership constraints over simple sequences and contexts. The problem generalizes both sequence and context unification, hence any complete solving procedure is non-terminating, because those problems are infinitary. Our algorithm is sound, terminating, and incomplete, computing partially solved constraints. Those partially solved constraints still may have infinitely many solutions or, in some cases when the solution set is finite, may explicitly provide a complete solution. We identified two such finitary fragments: well-moded and KIF, and showed how the algorithm computes a completely solved form for them. Both fragments are practically interesting and important. The well-moded fragment may arise in conditional rewriting over sequences and context, and the KIF fragment provides an expressive language for ontologies.

7.2 Constraint Logic Programming

We integrated the constraint solving algorithm into the constraint logic programming schema, obtaining a constraint logic programming language CLP($\mathcal{SC}$) over the domain of sequences and contexts. We studied the semantics of this language and identified two special cases of CLP($\mathcal{SC}$) programs, well-moded and KIF programs, that lead to constraints with the corresponding names, for which the solver computes a complete set of solutions. Although

the constraints we consider in this thesis are positive, in well-moded and KIF fragments we can easily enrich them with negation. Well-modedness then guarantees that the eventual test for disequality and non-membership in constraints will be performed on ground sequences and contexts, which can be effectively decided. In the KIF case, disequality and non-membership concerns a single sequence or context variable.

With the help of free and unordered unranked function symbols and sequence variables, we can easily express such well-known data structures as lists and multisets. This gives us the possibility to encode problems over those domains in CLP($\mathcal{SC}$). Context and function variables give additional expressive power to for making the encoding more compact and flexible. This expressive power, combined with the powerful constraint solving mechanism, makes CLP($\mathcal{SC}$) an useful tool.

7.3 Rule-based Programming

We described PρLog, a system for conditional rule-based transformation of sequences with strategies. It is built on top of Prolog and supports programming with individual, sequence, function, and context variables. The inference mechanism is based on Prolog's. The programs may contain Prolog clauses and predicates alongside the clauses specific to PρLog. We described the operational semantics and the strategy language of this system, and illustrated its applications on two use cases: XML processing and Web reasoning, and implementation of rewriting strategies. The PρLog code is usually compact and declaratively clear. The users familiar with logic programming and Prolog can very quickly start using it since its syntax is similar to that of Prolog and the semantics is based on logic programming.

7.4 Pattern Calculus

Pattern calculus is a formalism for functional programming. Confluence is one of the most important desirable properties for such formalisms. Being established earlier for pattern calculi with unitary matching, it was a challenging task to find conditions that guarantee confluence when matching is finitary. We studied such an extension, considering a pattern calculus with sequence variables, where even syntactic matching is finitary. Even if one collects all alternative reductions into one term, confluence does not hold in general and

special conditions are need to guarantee it. The conditions that we established are sufficient not only for calculi with sequence matching, but for arbitrary calculi with finitary matching. We gave three concrete instances of the matching function that satisfy these conditions: free and unordered sequence matchings, and the sequence matching with linear algebraic patterns.

7.5 Further Work

There are a couple of directions of research on the thesis topics that can be further explored. For instance, it would be interesting to find other fragments of constraints over sequences and contexts that can be solved efficiently. Applying CLP($\mathcal{SC}$) to problems in bioinformatics (e.g., modeling membrane computing) is another interesting idea. As for PρLog, one can think about implementing the matching module in some lower-level language to make it more efficient. Currently matching, like the whole PρLog system, is implemented in Prolog. Adding types to the pattern calculus with sequence variables and investigating conditions for strong normalization is yet another challenging task.

Bibliography

[AB94] Krzysztof R. Apt and Roland Bol. Logic programming and negation: A survey. *J. Logic Programming*, 19:9–71, 1994. 5.2.2

[ADFK13] Sandra Alves, Besik Dundua, Mário Florido, and Temur Kutsia. A confluent pattern calculus with hedge variables. In N. Hirokawa and V. van Oostrom, editors, *2nd International Workshop on Confluence, IWC 2013*, pages 41–45, 2013. 1

[ADFK14] Sandra Alves, Besik Dundua, Mário Florido, and Temur Kutsia. Confluence of pattern-based calculi with finitary matching. In Tudor Jebelean, Wei Li, and Dongming Wang, editors, *Third International Seminar on Program Verification, Automated Debugging and Symbolic Computation, PAS 2014*, 2014. 1

[AMR06] Ariel Arbiser, Alexandre Miquel, and Alejandro Ríos. A lambda-calculus with constructors. In Frank Pfenning, editor, *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2006. 1

[ANR04] Siva Anantharaman, Paliath Narendran, and Michaël Rusinowitch. Unification modulo *acui* plus distributivity axioms. *J. Autom. Reasoning*, 33(1):1–28, 2004. 6.2

[Ant96] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, March 1996. 3.7

[Baa07] Franz Baader, editor. *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*. Springer, 2007. 7.5

[Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984. Revised edition. 2.1, 6.1, 6.3

[BC00] Angela Bonifati and Stefano Ceri. Comparative analysis of five xml query languages. *SIGMOD Rec.*, 29(1):68–79, March 2000. 5.3.1

[BCJ+06] Bruno Buchberger, Adrian Craciun, Tudor Jebelean, Laura Kovács, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus Rosenkranz, and Wolfgang Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *J. Applied Logic*, 4(4):470–504, 2006. 1

[BKK+98] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *Electr. Notes Theor. Comput. Sci.*, 15:55–70, 1998. 1, 5.1

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008. 1, 5.1

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998. 1

[Bol99] Harold Boley. *A Tight, Practical Integration of Relations and Functions*, volume 1712 of *Lecture Notes in Computer Science*. Springer, 1999. 1

[BS01] Franz Baader and Wayne Snyder. Unification theory. In Robinson and Voronkov [RV01], pages 445–532. 2.4

[Buc96] Bruno Buchberger. Mathematica as a rewrite language. In Tetsuo Ida, Atsushi Ohori, and Masato Takeich, editors, *Functional and Logic Programming - 2nd Fuji International Workshop, FLOPS 1996, Shonan Village, Japan, June 4-6, 1996. Proceedings*, Lecture Notes in Computer Science, pages 1–13. Springer, 1996. 1

[BW08] Mikolaj Bojanczyk and Igor Walukiewicz. Forest algebras. In Jörg Flum, Erich Grädel, and Thomas Wilke, editors, *Logic and Automata*, volume 2

of *Texts in Logic and Games*, pages 107–132. Amsterdam University Press, 2008. 1

[CB83] Jacques Corbin and Michel Bidoit. A rehabilitation of robinson's unification algorithm. In Richard E. Mason, editor, *Information Processing 83*, pages 909–914. Elsevier Science Publishers Ltd., 1983. 2.4

[CD96] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(FD). *J. Log. Program.*, 27(3):185–226, 1996. 1

[CD97] Eric Chasseur and Yves Deville. Logic program schemas, constraints, and semi-unification. In Fuchs [Fuc98], pages 69–89. 1

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002. 1, 5.1

[CDFK10] Jorge Coelho, Besik Dundua, Mário Florido, and Temur Kutsia. A rule-based approach to XML processing and web reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *Web Reasoning and Rule Systems - Fourth International Conference, RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings*, volume 6333 of *Lecture Notes in Computer Science*, pages 164–172. Springer, 2010. 1

[CDG+07] Hubert Comon, Max Dauchet, Remi Gilleron, Cristof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available from: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007. 3.4.4

[CF04] Jorge Coelho and Mário Florido. CLP(Flex): Constraint Logic Programming applied to XML processing. In Robert Meersman and Zahir Tari, editors, *CoopIS/DOA/ODBASE (2)*, volume 3291 of *LNCS*, pages 1098–1112. Springer, 2004. 1

[CF06] Jorge Coelho and Mário Florido. Veriflog: A constraint logic programming approach to verification of website content. In Heng Tao Shen, Jinbao Li, Minglu Li, Jun Ni, and Wei Wang, editors, *Advanced Web and Network Technologies, and Applications, APWeb 2006 International Workshops: XRA,*

IWSN, MEGA, and ICSE, Harbin, China, January 16-18, 2006, Proceedings, volume 3842 of *Lecture Notes in Computer Science*, pages 148–156. Springer, 2006. 1

[CF07a] Horatiu Cirstea and Germain Faure. Confluence of pattern-based calculi. In Baader [Baa07], pages 78–92. 1, 2.1, 6.1, 6.3

[CF07b] Jorge Coelho and Mário Florido. XCentric: logic programming for XML processing. In Irini Fundulaki and Neoklis Polyzotis, editors, *9th ACM International Workshop on Web Information and Data Management (WIDM 2007), Lisbon, Portugal, November 9, 2007*, pages 1–8. ACM, 2007. 1

[CFK07] Jorge Coelho, Mário Florido, and Temur Kutsia. Sequence disunification and its application in collaborative schema construction. In Mathias Weske, Mohand-Said Hacid, and Claude Godart, editors, *WISE Workshops*, volume 4832 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2007. 1

[CFK09] Jorge Coelho, Mário Florido, and Temur Kutsia. Collaborative schema construction using regular sequence types. In *Proceedings of the IEEE International Conference on Information Reuse and Integration, IRI 2009, 10-12 August 2009, Las Vegas, Nevada, USA*, pages 290–295. IEEE Systems, Man, and Cybernetics Society, 2009. 1

[CHL96] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2):362–397, 1996. 6.5

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. Part I. *Annals of Mathematics*, 33(2):346–366, 1932. 1

[CJL04] Yves Caseau, François-Xavier Josset, and François Laburthe. Claire: Combining sets, search and rules to better express algorithms. *CoRR*, cs.PL/0405091, 2004. 5.1

[CK01] Horatiu Cirstea and Claude Kirchner. The rewriting calculus - parts I and II. *Logic Journal of the IGPL*, 9(3), 2001. 1, 1, 5.1, 6.1

[Col90] Alain Colmerauer. An introduction to Prolog III. *Commun. ACM*, 33(7):69–90, 1990. 1

[Com98] Hubert Comon. Completion of rewrite systems with membership constraints. Part II: constraint solving. *J. Symb. Comput.*, 25(4):421–453, 1998. 1, 3.4

[Com07] Common Logic Working Group. Common Logic Working Group Documents: Common Logic Standard. http://common-logic.org/, 2007. 1

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982. 4.2

[DFKM14] Besik Dundua, Mário Florido, Temur Kutsia, and Mircea Marin. Constraint logic programming for hedges: A semantic reconstruction. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 285–301. Springer, 2014. 1, 3.1, 3.8.1

[DK] Besik Dundua and Temur Kutsia. PρLog. Version 0.9. Available from: http://www.risc.uni-linz.ac.at/people/tkutsia/software.html. 1, 5.1

[DKM09] Besik Dundua, Temur Kutsia, and Mircea Marin. Strategies in PρLog. In Maribel Fernández, editor, *International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009*, volume 15 of *EPTCS*, pages 32–43, 2009. 1

[Dow01] Gilles Dowek. Higher-order unification and matching. In Robinson and Voronkov [RV01], pages 1009–1062. 2.4

[DPPR00] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, 2000. 3.4

[Dun10] Besik Dundua. Inference mechanism of PρLog. In *Reports of Enlarged Session of the Seminar of I. Vekua Institute of Applied Mathematics*, 2010. 1

[EIG88] Gonzalo Escalada-Imaz and Malik Ghallab. A practically efficient and almost linear unification algorithm. *Artif. Intell.*, 36(2):249–263, 1988. 2.4

[Fau07] Germain Faure. *Structures et modèles de calculs de réécriture.* PhD thesis, Université Henri Poincaré, Nancy, France, 2007. 6.16, 6.5, 6.5

[Frü98] Thom W. Frühwirth. Theory and practice of constraint handling rules. *J. Log. Program.*, 37(1-3):95–138, 1998. 1, 5.1

[Fuc98] Norbert E. Fuchs, editor. *Logic Programming Synthesis and Transformation, 7th International Workshop, LOPSTR'97, Leuven, Belgium, July 10-12, 1997, Proceedings*, volume 1463 of *Lecture Notes in Computer Science*. Springer, 1998. 7.5

[Gen98] Michael R. Genesereth. Knowledge Interchange Format, draft proposed American National Standard (dpANS). Technical Report NCITS.T2/98-004, 1998. Available from http://logic.stanford.edu/kif/dpans.html. 1, 3.1, 3.8.2, 4.4

[Gin91] Matthew L. Ginsberg. The MVL theorem proving system. *SIGART Bull.*, 2(3):57–60, 1991. 1

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981. 2.4

[GT07] Sumit Gulwani and Ashish Tiwari. Computing procedure summaries for interprocedural analysis. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *LNCS*, pages 253–267. Springer, 2007. 1

[Ham97] Makoto Hamana. Term rewriting with sequences. In *Proceedings of the First International Theorema Workshop*, number 97-20 in RISC Technical Report series, Hagenberg, Austria, 1997. 1

[HM01] Pat J. Hayes and Christopher Menzel. Semantics of Knowledge Interchange Format. http://reliant.teknowledge.com/IJCAI01/HayesMenzel-SKIF-IJCAI2001.pdf, 2001. 1

[HM05] Pat J. Hayes and Christopher Menzel. Simple Common Logic. In *W3C Workshop on Rule Languages for Interoperability*. W3C, 2005. 1

[Hon91] Hoon Hong. RISC-CLP(Real): logic programming with non-linear constraints over the reals. In Frédéric Benhamou and Alain Colmerauer, editors, *WCLP*, pages 133–159, 1991. 1

[Hue76] Gerard Huet. Résolution d'Équations dans des langages d'ordre $1, 2, \ldots, \omega$. These d'État, Université de Paris VII, 1976. 2.4

[HV06] Ian Horrocks and Andrei Voronkov. Reasoning support for expressive ontology languages using a theorem prover. In Jürgen Dix and Stephen J. Hegner, editors, *Foundations of Information and Knowledge Systems, 4th International Symposium, FoIKS 2006, Budapest, Hungary, February 14-17, 2006, Proceedings*, volume 3861 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2006. 1

[Jeż14] Artur Jeż. Context unification is in PSPACE. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 244–255. Springer, 2014. 2.4

[JK06] C. Barry Jay and Delia Kesner. Pure pattern calculus. In Peter Sestoft, editor, *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 100–114. Springer, 2006. 1, 6.1

[JK09] C. Barry Jay and Delia Kesner. First-class patterns. *J. Funct. Program.*, 19(2):191–225, 2009. 1, 6.1

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 111–119. ACM Press, 1987. 1

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994. 1, 4.1

[JMMS98] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. The semantics of constraint logic programs. *J. Log. Program.*, 37(1-3):1–46, 1998. 1, 4.1, 4.3

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, 1992. 1

[JR08] Florent Jacquemard and Michaël Rusinowitch. Closure of hedge-automata languages by hedge rewriting. In Voronkov [Vor08], pages 157–171. 1

[Kah03] Wolfram Kahl. Basic pattern matching calculi: Syntax, reduction, confluence, and normalisation. Technical Report 16, Software Quality Research Laboratory, McMaster Univ., 2003. 1

[Klo80] Jan Willem Klop. *Combinatory reduction systems.* PhD thesis, Mathematisch Centrum, Amsterdam, 1980. 6.12

[KLV07] Temur Kutsia, Jordi Levy, and Mateu Villaret. Sequence unification through currying. In Baader [Baa07], pages 288–302. 2.4

[KLV10] Temur Kutsia, Jordi Levy, and Mateu Villaret. On the relation between context and sequence unification. *J. Symb. Comput.*, 45(1):74–95, 2010. 2.4

[KM05a] Temur Kutsia and Mircea Marin. Can context sequence matching be used for querying XML? In Laurent Vigneron, editor, *Proceedings of the 19th International Workshop on Unification UNIF'05*, pages 77–92, Nara, Japan, 22 April 2005. 1

[KM05b] Temur Kutsia and Mircea Marin. Matching with regular constraints. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2005. 1

[KM06] Temur Kutsia and Mircea Marin. Solving regular constraints for hedges and contexts. In Jordi Levy, editor, *Proceedings of the 20th International Workshop on Unification UNIF'06*, pages 89–107, Seattle, USA, 11 August 2006. 3.1, 3.8.1

[KM12] Temur Kutsia and Mircea Marin. Solving, reasoning, and programming in common logic. In *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, Timisoara, Romania,*

September 26-29, 2012, pages 119–126. IEEE Computer Society, 2012. 3.1, 6.1, 6.4

[Kol98] Alexander Koller. Evaluating context unification for semantic underspecification. In *Proceedings of the Third ESSLLI Student Session*, pages 188–199, 1998. 1

[Kut02] Temur Kutsia. Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols. RISC Report Series 02-09, Research Institute for Symbolic Computation (RISC), University of Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, May 2002. PhD Thesis. 1, 3.1

[Kut03] Temur Kutsia. Equational prover of THEOREMA. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2003. 1

[Kut04] Temur Kutsia. Solving equations involving sequence variables and sequence functions. In Bruno Buchberger and John A. Campbell, editors, *Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 22-24, 2004, Proceedings*, volume 3249 of *Lecture Notes in Computer Science*, pages 157–170. Springer, 2004. 2.4

[Kut07] Temur Kutsia. Solving equations with sequence variables and sequence functions. *J. Symb. Comput.*, 42(3):352–388, 2007. 1, 2.4, 6.1

[KvOdV08] Jan Willem Klop, Vincent van Oostrom, and Roel C. de Vrijer. Lambda calculus with patterns. *Theor. Comput. Sci.*, 398(1-3):16–31, 2008. 1, 6.1

[Lev96] Jordi Levy. Linear second-order unification. In Harald Ganzinger, editor, *Rewriting Techniques and Applications, 7th International Conference, RTA-96, New Brunswick, NJ, USA, July 27-30, 1996, Proceedings*, volume 1103 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 1996. 1, 2.4, 3.1

[Llo87] John Wylie Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987. 5.1

[LNV05] Jordi Levy, Joachim Niehren, and Mateu Villaret. Well-nested context unification. In Robert Nieuwenhuis, editor, *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, volume 3632 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2005. 1

[LSV06] Jordi Levy, Manfred Schmidt-Schauß, and Mateu Villaret. Stratified context unification is np-complete. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2006. 1

[Mai98] David Maier. Database desiderata for and XML query language. Available from: http://www.w3.org/TandS/QL/QL98/pp/maier.html, 1998. 5.3.1

[Mak77] Gennady S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, 32(2):129–198, 1977. In AMS, (1979). 1

[Men11] Christopher Menzel. Knowledge representation, the World Wide Web, and the evolution of logic. *Synthese*, 182(2):269–295, 2011. 1

[MK03] Mircea Marin and Temur Kutsia. On the implementation of a rule-based programming system and some of its applications. In Boris Konev and Renate Schmidt, editors, *Proceedings of the 4th International Workshop on the Implementation of Logics*, pages 55–69, Almaty, Kazakhstan, 2003. 1

[MK06] Mircea Marin and Temur Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006. 1, 1, 5.1, 5.7

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982. 2.4

[Moh96] Markus Mohnen. Context patterns in haskell. In Werner E. Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, IFL'96, Bad Godesberg, Germany, September 16-18, 1996, Selected Papers*, volume 1268 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 1996. 1

[MOM02] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theor. Comput. Sci.*, 285(2):121–154, 2002. 1, 5.1

[MT03] Mircea Marin and Dorin Tepeneu. Programming with sequence variables: The Sequentica package. In Peter Mitic, Philip Ramsden, and Janet Carne, editors, *Challenging the Boundaries of Symbolic Computation. Proceedings of 5th International Mathematica Symposium*, pages 17–24, London, 2003. Imperial College Press. 1

[NPR97] Joachim Niehren, Manfred Pinkal, and Peter Ruhrberg. A uniform approach to underspecification and parallelism. In Philip R. Cohen and Wolfgang Wahlster, editors, *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, 7-12 July 1997, Universidad Nacional de Educación a Distancia (UNED), Madrid, Spain.*, pages 410–417. Morgan Kaufmann Publishers / ACL, 1997. 1

[NV02] Joachim Niehren and Mateu Villaret. Parallelism and tree regular constraints. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002, Tbilisi, Georgia, October 14-18, 2002, Proceedings*, volume 2514 of *Lecture Notes in Computer Science*, pages 311–326. Springer, 2002. 1

[NV05] Joachim Niehren and Mateu Villaret. Describing lambda terms in context unification. In Philippe Blache, Edward P. Stabler, Joan Busquets, and Richard Moot, editors, *Logical Aspects of Computational Linguistics, 5th International Conference, LACL 2005, Bordeaux, France, April 28-30, 2005, Proceedings*, volume 3492 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2005. 1

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In *Logic and Computer Science*, pages 361–386. Academic Press, 1990. 1

[PW78] Mike Paterson and Mark N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978. 2.4

[PW87] Simon L. Peyton Jones and Philip Wadler. *The Implementation of Functional Programming Languages*, chapter 4: Structured Types and the Semantics of Pattern Matching. Prentice Hall, 1987. 1, 6.1

[Raj94] Arcot Rajasekar. Constraint logic programming on strings: Theory and applications. In Maurice Bruynooghe, editor, *Logic Programming, Proceedings*

of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994, page 681. MIT Press, 1994. 1

[RF97] Julian Richardson and Norbert E. Fuchs. Development of correct transformation schemata for Prolog programs. In Fuchs [Fuc98], pages 263–281. 1

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. 2.4

[RP89] Peter Ruzicka and Igor Prívara. An almost linear Robinson unification algorithm. *Acta Inf.*, 27(1):61–71, 1989. 2.4

[RV01] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. 7.5

[Sch04] Sebastian Schaffert. *Xcerpt: a rule-based query and transformation language for the web*. PhD thesis, University of Munich, 2004. 5.3.2, 5.3.4

[SS02] Manfred Schmidt-Schauß. A decision algorithm for stratified context unification. *J. Log. Comput.*, 12(6):929–953, 2002. 1

[SSS02] Manfred Schmidt-Schauß and Klaus U. Schulz. Solvability of context equations with two context variables is decidable. *J. Symb. Comput.*, 33(1):77–122, 2002. 1

[SSS04] Manfred Schmidt-Schauß and Jürgen Stuber. The complexity of linear and stratified context matching problems. *Theory Comput. Syst.*, 37(6):717–740, 2004. 3.1

[SW10] Klaus-Dieter Schewe and Qing Wang. XML database transformations. *J. UCS*, 16(20):3043–3072, 2010. 1

[vdBvDH+01] Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The asf+sdf meta-environment: a component-based language development environment. *Electr. Notes Theor. Comput. Sci.*, 44(2):3–8, 2001. 1, 5.1

[Vil04] Mateu Villaret. *On Some Variants of Second-Order Unification*. PhD thesis, Technical University of Catalonia, Barcelona, 2004. 1, 3.1

[vO90] Vincent van Oostrom. Lambda calculus with patterns. Technical Report IR-228, Vrije Universiteit, Amsterdam, 1990. 1, 6.1

[Vor08] Andrei Voronkov, editor. *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*. Springer, 2008. 7.5

[Wac03] Benjamin Wack. Klop counter example in the ρ-Calculus. Draft notes, LORIA, Nancy, 2003. 6.12

[WB01] Manfred Widera and Christoph Beierle. A term rewriting scheme for function symbols with variable arity. Technical Report 280, Praktische Informatik VIII, FernUniversität Hagen, Germany, 2001. 1

[Wol03] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, 5th edition, 2003. 1, 1

[WSTL10] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *CoRR*, abs/1011.5332, 2010. 1

[YH90] Hirofumi Yokouchi and Teruo Hikita. A rewriting system for categorical combinators with multiple arguments. *SIAM J. Comput.*, 19(1):78–97, 1990. 6.5

Index