Functional Logic Programming with Distributed Constraint Solving

Dissertation

zur Erlangung des akademischen Grades
"Doktor der technischen Wissenschaften"
in der Studienrichtung Informatik

verfaßt von
Mircea Marin
am
Institut für symbolisches Rechnen
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

April 2000

Erster Begutachter:	o. Univ.-Prof. Dr. Bruno Buchberger Technisch-Naturwissenschaftliche Fakultät Johannes Kepler Universität Linz
Zweiter Begutachter:	Univ. Prof. Dr. Tetsuo Ida Institute of Information Sciences and Electronics University of Tsukuba, Japan

Eidesstattliche Erklärung

Ich erkläre, daß ich die Dissertation selbständig verfaßt habe, andere als die angegebenen Quellen und Hilfsmittel nicht verwendet und mich auch sonst keiner unterlaubten bedient habe.

Mircea Marin
Linz, im April 2000

Zusammenfassung

Das Hauptziel dieser Arbeit ist der Entwurf effizienter Kalküle die als die operationale Semantik einer funktional-logischen Programmiersprache höherer Ordnung mit der Fähigkeit zum Lösen von Constraints dienen können, sowie deren verteilte Implementierung auf einem Computer-Netzwerk.

Die Hauptvorteile einer Logik höherer Stufe gegenüber einer Logik erster Stufe sind die Quantifizierung über Funktionen und Prädikate und ihre Abstraktionsmechanismen. Programmieren mit Funktionen höherer Stufe ist Standard in funktionalen Sprachen und kürzlich entwickelte Sprachen wie λ-Prolog illustrieren die praktische Nützlichkeit des logischen Programmierens höherer Stufe. Funktional-logische Programmiersprachen sind gegenwärtig hauptsächlich auf Logik erster Stufe eingeschränkt, obwohl die letzten Jahre Zeuge eines wachsenden Interesses waren, die operationalen Prinzipien von funktional-logischen Sprachen auf Logiken höherer Ordnung auszudehnen.

In dieser Arbeit präsentieren wir verschiedene Kalküle für ”lazy narrowing” für eine Logik höherer Stufe. Wir beweisen, dass unsere Kalküle wesentliche Eigenschaften wie Korrektheit und Vollständigkeit erfüllen, falls die funktional-logische Sprache gewisse Einschränkungen aufweist. Im allgemeinen sind die von uns untersuchten Einschränkungen in der funktional-logischen Forschungsgemeinde weit akzeptiert, bzw. sind sie Erweiterungen von Einschränkungen, die Standard in der funktional-logischen Programmierung erster Stufe sind.

Wir behaupten, dass die in dieser Arbeit vorgelegten Kalküle bessere Möglichkeiten für eine operationale Semantik für funktional-logisches Programmieren höherer Stufe sind, als wir bisher in der Literatur vorgefunden haben.

Um die Lösungskapazität einer funktional-logischen Sprache höherer Ordnung zu verbessern, zielen wir weiters auf die Integration der operationalen Prinzipien von ”lazy narrowing” höherer Stufe und vom gleichzeitigem Lösen von Constraints (”concurrent constraint solving”) ab. Concurrent

constraint solving ist bereits als ein gangbarer Ansatz bekannt, um die Lösungskapazitäten verschiedener Löser in einem System zu integrieren, das Probleme lösen kann, die kein einzelner Löser alleine zu behandeln imstande ist.

Wir defineren ein Schema CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$ für constraint logic programming, das die Integration eines lazy narrowing Kalküls $\mathcal{C}$ höherer Stufe (die funktional-logische Komponente) mit den operationalen Prinzipien einer Löser-Kooperation ber einem Constraint-Bereich $\mathcal{X}$ beschreibt. Die Löser-Kooperation ist durch eine Sammlung von Lösern $CS_1, \ldots, CS_n$ definiert, die zur Lößung eines gegebenen Problems mittels einer Strategie $\mathcal{S}$ zusammenarbeiten.

Weiters beschreiben wir ein experimentelles System, das wir in *Mathematica* geschrieben haben und das die Implementierung einer Instanz des von uns entwickelten Schemas darstellt. Das System heisst `CFLP` und besteht aus einem funktional-logischen Interpreter, der auf einer Maschine läuft, und einem verteilten Subsystems zur Constraint-Lösung. Das verteilte Subsystem besteht aus einer Anzahl von Lösern, die auf verschiedenen Maschinen laufen können, und einer speziellen Komponente, dem Constraint-Verwalter, der die Strategie $\mathcal{S}$ zur Koordination der Löser implementiert.

Zuletzt illustrieren wir Verhalten und Benutzung von `CFLP` mit verschiedenen Beispielprogrammen.

Abstract

The main goal of the thesis is the design of efficient calculi that can serve as operational semantics of a higher-order functional logic programming language with constraint solving capabilities, and their distributed implementation on a network of computers.

The main advantages of higher-order logic versus first-order logic are quantification over functions and predicates and its abstraction mechanism. Higher-order programming is standard in functional programming languages, and recent languages such as λ-Prolog illustrate the practical utility of higher-order logic programming. Currently, functional logic programming is mainly restricted to first-order logic, although recent years witnessed a growing interest to extend the operational principles of functional logic programming to higher-order logic.

In this thesis we present various lazy narrowing calculi for higher-order logic. We prove that our calculi satisfy essential properties, such as being sound and complete, if the functional logic programs satisfies certain restrictions. In general, the restrictions investigated by us are widely accepted by the functional logic community, or are higher-order generalizations of restrictions which are standard in first-order functional logic programming.

We claim that the calculi proposed in the thesis are better choices for an operational semantics of higher-order functional logic programming than what we have found in the literature.

Secondly, in order to improve the solving capability of a higher-order functional logic programming language, we aim at integrating the operational principles of higher-order lazy narrowing and of concurrent constraint solving. Concurrent constraint solving is already recognized as a viable approach to integrate the constraint solving capabilities of various constraint solvers in a system that can solve problems that none of the single solvers can handle alone.

We define a scheme CFLP($\mathcal{X}, \mathcal{S}, \mathcal{C}$) for constraint logic programming, which describes the integration of a higher-order lazy narrowing calculus $\mathcal{C}$ (the functional logic component) with the operational principle of a solver

cooperation over a constraint domain $\mathcal{X}$. The solver cooperation is defined by a collection of constraint solvers $CS_1, \ldots, CS_n$, which cooperate upon solving a given problem in accordance with a strategy $\mathcal{S}$.

Next, we describe an experimental system written in *Mathematica*, which is the implementation of an instance of the scheme described by us. The system is called `CFLP`, and it consists of a functional logic interpreter running on one machine, and a distributed constraint solving subsystem. The distributed constraint solving subsystem consists of a number of constraint solvers running on possibly different machines and a special component called constraint scheduler, which implements the strategy $\mathcal{S}$ to coordinate the solver cooperation.

Finally, we illustrate the behaviour and utility of `CFLP` with several example programs.

Acknowledgements

I am grateful to my supervisor, Prof. Dr. Bruno Buchberger for many reasons. Working with him in the frame of the *Theorema* project helped me to better understand the different aspects covered by the activity of a working mathematician: proving, solving and computing. He gave me his understanding, encouragement and support during my stay at RISC. His insight and experience have been a great influence to my entire work.

My sincere thanks go to my thesis supervisors, professors Tetsuo Ida and Wolfgang Schreiner, for their encouragement, guidance, source of inspiration, and support. I am indebted to Wolfgang Schreiner for the valuable discussions and comments on distributed computation.

My strong cooperation with Prof. Dr. Tetsuo Ida and the members of the SCORE group from University of Tsukuba, Japan, has been an incredible experience and an occasion to do valuable research in the field of constraint and functional logic programming.

I should also thank to Prof. Taro Suzuki for all the valuable discussions in the field of higher-order functional logic programming.

I would also like to thank to my colleagues and friends, without whom my stay at RISC would not have been so enjoyable.

This work has been supported by the project "Distributed Constraint Solving for Functional Logic Programming" in the AITEC Contract Research Programme, 1998-1999, and has been partially supported by Grant-in-Aid for Scientific Research on Priority Areas "Research on the Principles of Constructing Software with evolutionary Mechanisms", and Grant-in Aid for Scientific Research (B) 10480053.

Contents

Chapter 1

Introduction

Many interesting and complex problems from mathematics and sciences can be reduced to solving systems of equations over various constraint domains. The design and implementation of theoretical frameworks that support an easy formulation and efficient solving methods of such problems has received considerable interest during the last decade.

The most successful paradigms which support solving systems of equations over constraint domains are the outcome of integrating some form of declarative programming (e.g., logic programming, functional programming, or functional logic programming) and constraint solving. The declarative programming component provides a means to define one's own abstractions (user defined predicates and/or functions) over a constraint domain. We mention here the CLP($\mathcal{X}$) scheme [JL87] for constraint logic programming, and the proposals of a CFLP($\mathcal{X}$) scheme for constraint functional logic programming [DGP91b, DGP91a, LF92, LF94].

There have been, among others, two streams of development in the paradigm of constraint solving:

- cooperative constraint solving, and
- distributed constraint solving.

Cooperative constraint solving [Mon96, Hon92b, Hon94, Hon92a, Rue95] is concerned with the possibility of combining different constraint solvers which can solve different admissible constraints, in an attempt to obtain a more powerful solver that can solve systems of constraints that none of the individual solvers can handle alone. The central problem in cooperative constraint solving is the design of a suitable cooperation mechanism.

Distributed constraint solving [Leu93] refers to the following scenario. A distributed constraint system is composed of several machines called nodes.

Nodes communicate via message passing. During program execution, constraints are generated in the nodes incrementaly and asynchronously. The problem is to determine whether the constraints contained in the nodes are collectively satisfiable and determine the values of the variables satisfying these constraints whenever possible.

In this thesis we aim at the design and implementation of a scheme for declarative programming with constraints that integrates the advantages of functional programming, logic programming, cooperative constraint solving and distributed constraint solving. More precisely, our main goal is to design and implement a system that integrates:

1. functional logic programming,
2. higher-order equational logic,
3. cooperative constraint solving, and
4. distributed constraint solving.

The outcome is a scheme CFLP($\mathcal{X}$, $\mathcal{S}$, $\mathcal{C}$) for cooperative constraint functional logic programming. The scheme describes a system consisting of

1. a functional logic interpreter whose operational semantics is a lazy narrowing calculus $\mathcal{C}$,
2. a distributed constraint solving subsystem for solving constraints over a constraint domain $\mathcal{X}$. The distributed constraint solving system consists of
   (a) a number of constraint solvers which may run on different machines in a distributed environment,
   (b) a scheduler, which coordinates the constraint solving process carried out by the individual solvers in accordance with a cooperation strategy $\mathcal{S}$.

Motivation

There are various reasons why we have choosen functional logic programming as the starting point of our development. Firstly, functional logic programming is already the result of integrating two of the most successful declarative programing styles: logic programming and functional programming, in a way that captures the main advantages of both [Han97]. Sound and complete operational principles for functional logic programming have been identified and efficient implementations [AKP93, Smo95, HS95, Loo95, HAK$^+$00, SHC96, Nai91] witness its utility for practical applications. Secondly, our strong cooperation with the members of the SCORE group from University of Tsukuba, Japan gave us the opportunity to deeply understand the details of the design and implementation of an efficient functional logic programming calculus.

One of the most important subgoals in this thesis is to extend the functional logic programming style with higher-order constructs and to identify calculi for higher-order functional logic programming that can serve as operational semantics. Recent proposals to extend functional logic programming with support for higher-order constructs indicate the high potential of such a paradigm in modeling complex real-world problems [NI95, SNI97, Pre98, MMIY99]. Higher-order constructs such as function variables and λ-abstractions are widely used in functional programming, and higher-order logic programming languages, most notably λ-Prolog, have shown their practicality. The main challenge in adopting higher-order constructs in functional logic programming is the design of an efficient operational principle.

By integrating functional logic programming with constraint solving we aim at extending the functional logic scheme with the capacity of solving constraints over a given constraint domain. Among the formalisms for a constraint functional logic programming scheme CFLP($\mathcal{X}$) mentioned in the literature we recall the ones proposed by Darlington [DGP91b] and by López-Fraguas [LF92, LF94]. It turns out that defining a CFLP($\mathcal{X}$) scheme is more challenging then defining a CLP($\mathcal{X}$) scheme, mainly because of the complications of defining a clear semantics of the integrated model, and of the fact that constraint solving and the operational principle of functional logic programming are mutually dependent. Of particular interest is the scheme proposed by López-Fraguas, which can be formally described as:

$$\mathrm{CFLP}(\mathcal{X}, \mathcal{C}) = \mathrm{FLP}(\mathcal{C}) + \mathrm{CP}(\mathcal{X})$$

i.e., as the combination of a functional logic component whose operational semantics is given by a lazy narrowing calculus $\mathcal{C}$, and a constraint programming scheme CP($\mathcal{X}$). The CP($\mathcal{X}$) scheme is defined by the constraint domain $\mathcal{X}$ and its associated constraint solver.

This scheme can be improved if

1. we extend the scheme CP($\mathcal{X}$) by replacing the underlying constraint solver on $\mathcal{X}$ with a solver cooperation. A similar approach was proposed by Hong [Hon94, Hon92a] who studied this extension from the perspective of constraint logic programming

2. we define a distributed model for the CFLP scheme extended with a solver cooperation.

Our contribution

The main goal of the thesis is to define a suitable operational semantics for higher-order constraint functional logic programming. Our approach is to define various higher-order lazy narrowing calculi which are suitable for higher-order functional logic programming, and to extend their computing power with a cooperative constraint solving mechanism.

The starting point of our investigation is the lazy narrowing calculus LNC with leftmost equation selection strategy [MOI96], a sound and complete calculus for functional logic programming. It has been shown [MO98] that by imposing reasonable restrictions on the functional logic programs and on the equational goal, the nondeterminism between the inference rules of LNC can be completely eliminated without losing the important properties of soundness and completeness . This property makes LNC a good candidate for an operational semantics of functional logic programming.

Our first contribution relates to the usage of lazy narrowing with applicative term rewriting systems (ATRS for short). Applicative term algebras are more expressive than first-order term algebras because of the presence of higher-order variables. The generalization of the calculus LNC to applicative term algebras is straightforward and its soundness and completeness results are preserved. Unfortunately, LNC with ATRS is highly nondeterministic, mainly because of the many choices to perform outermost narrowing steps, and thus the search space for solutions is huge. In Chapter 4 we identify a refinement of LNC, called LNCA, which replaces the outermost narrowing rule of LNC with inference rules that are applied more deterministically, and prove that LNCA is sound and complete. We conjecture that LNCA can be refined towards more deterministic versions by following an approach similar to the deterministic refinement of LNC.

Our second contribution is in the field of algebras of simply-typed λ-terms and of functional logic programming with pattern rewrite systems (PRS for short). This theoretical framework is more expressive than the previous one because it supports λ-abstractions. Our development draws on two sources: the calculus LN with PRS [Pre98] and the deterministic refinements of the calculus LNC. LN can be viewed as a higher-order generalization of LNC, and LN satisfies soundness and completeness results which are similar to those of LNC. Therefore, we considered important to try to lift the deterministic refinements of LNC to a suitable extension of the calculus LN. The outcome is a collection of lazy narrowing calculi for simply-typed λ-algebras, which we prove to be sound and complete for certain classes of PRSs. Since the restrictions that define our classes of PRSs are higher-order versions of restrictions of TRSs which are standard in functional logic programming, we claim that they are reasonable for higher-order functional logic programming. We claim that the calculi proposed by us are better than the ones proposed so far in the literature.

Our third contribution relates to the possibility to combine the advantages of higher-order functional logic programming and cooperative constraint solving. We propose a scheme CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$ for cooperative constraint functional logic programming defined over algebras of simply-typed λ-terms. The scheme describes a system that integrates a functional logic programming system based on a lazy narrowing calculus $\mathcal{C}$ for PRSs with a cooperative constraint solving system. Formally

$$\mathrm{CFLP}(\mathcal{X}, \mathcal{S}, \mathcal{C}) = \mathrm{FLP}(\mathcal{X}, \mathcal{C}) + \mathrm{CP}(\mathcal{X}, \mathcal{S})$$

where $\mathcal{C}$ is a lazy narrowing calculus for PRS, $\mathcal{X}$ is the underlying constraint system, and $\mathcal{S}$ is a strategy that defines the way how the individual constraint solvers cooperate upon solving constraints over $\mathcal{X}$.

Fourth, we propose a distributed model of CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$. To illustrate the suitability of our distributed model, we describe the implementation of an instance of it called `CFLP`, and give some application examples for the resulted system.

Structure of the thesis

The structure of the thesis is as follows:

Chapter 2 introduces mathematical preliminaries that are used throughout the thesis. Basic concepts and properties related to universal algebra, general logic and equational logic are presented.

Chapter 3 gives a brief account to the functional programming framework.

Chapter 4 describes our first contribution to the field of higher-order functional logic programming: lazy narrowing with applicative term rewriting systems. We propose a new calculus called LNCA, which can be regarded as a deterministic refinement of the calculus LNC for ATRS, and give a detailed proof of its soundness and completeness.

Chapter 5 describes our second contribution to higher-order functional logic programming: lazy narrowing with pattern rewrite systems. We adopt the theoretical framework of higher-order equational reasoning proposed by Prehofer [Pre98] and define a lazy narrowing calculus for PRS, called $\mathrm{LN}_{\mathrm{ff}}$. The calculus $\mathrm{LN}_{\mathrm{ff}}$ can be regarded as an extension of the calculus LN proposed by Prehofer to solve systems of oriented equations. $\mathrm{LN}_{\mathrm{ff}}$ is designed to solve systems of both oriented and unoriented equations, and is extended with inference rules to perform full unification of higher-order patterns. In Sect. 5.3 we prove the soundness and completeness of $\mathrm{LN}_{\mathrm{ff}}$ with respect to certain equation selection strategies. Sections 5.4–5.9 describe refinements of $\mathrm{LN}_{\mathrm{ff}}$ towards more deterministic versions. Most of these refinements are inspired by similar refinements of the first order calculus LNC described in [MO98]. In Sect. 5.10 we define an extension of $\mathrm{LN}_{\mathrm{ff}}$ to conditional PRSs and discuss how some of the refinements of $\mathrm{LN}_{\mathrm{ff}}$ described in the previous sections can be lifted to the conditional case.

Chapter 6 gives an account to our third and fourth contributions. We define a cooperative constraint functional logic programming scheme CFLP($\mathcal{X}, \mathcal{S}, \mathcal{C}$) that combines the advantages of functional logic programming and cooperative constraint solving. A distributed model of the scheme is outlined in order to take advantage of the constraint solving resources available in a distributed environment.

Chapter 7 describes an instance of the distributed model of the scheme CFLP($\mathcal{X}, \mathcal{S}, \mathcal{C}$) defined in the previous chapter. We have implemented a system called `CFLP` consisting of a functional logic interpreter running on one machine and of various constraint solvers which can run on possibly different machines. The system is implemented completely in *Mathematica* and makes use of the *MathLink* communication protocol for interprocess communication via message passing.

Chapter 8 illustrates the practical utility of `CFLP` with examples.

Chapter 2

Mathematical Preliminaries

In this chapter we present some preliminary notions and results used in the thesis. Basic concepts and properties of universal algebra, general logic, and equational logic are presented. The presentation is carried out for the case of many-sorted signatures.

Notation

We first establish some notational conventions that will be used from now on. A (possibly empty) sequence $a_m, a_{m+1}, \ldots, a_n$ of syntactic objects is denoted by $\overline{a_{m,n}}$. The empty sequence is denoted by $\Box$. We write $\overline{a_n}$ instead of $\overline{a_{1,n}}$. The subscript n will be dropped when irrelevant, i.e., we will write $\overline{x}$ instead of $\overline{x_n}$ when n is irrelevant. This notation for sequences is extended to sequences of function applications and sequences of binary relations between expressions as follows:

- $f(\overline{a_{m,n}})$ denotes the expression $f(a_m, a_{m+1}, \ldots, a_n)$,
- $\overline{a_{m,n}(e)}$ denotes the sequence $a_m(e)$, $a_{m+1}(e), \ldots, a_n(e)$,
- If $\overline{a_{m,n}}, \overline{b_{m,n}}$ are sequences and $\cong$ an infix operator then $\overline{a_{m,n} \cong b_{m,n}}$ stands for $a_m \cong b_m, a_{m+1} \cong b_{m+1}, \ldots, a_n \cong b_n$.

Thus, we can write $\overline{a_n \cong b_n}$ instead of $a_1 \cong b_1, \ldots, a_n \cong b_n$.

Given a syntactic domain D, we denote by D^* the set of sequences of objects in D. If *rel* is a binary relation over D then *rel*$^+$ denotes the transitive closure of *rel*, and *rel** denotes the reflexive and transitive closure of *rel*.

2.1 Inductive Definitions

Most objects used in logic and computer science are defined inductively. By this we mean that we often define a set of objects as the smallest set of objects containing a given set X of objects, which is closed under a given set F of constructors. In this section we give a formal description of this notion.

Definition 1 (inductive closure) *Let A be a nonempty set, $X \subseteq A$ and F a set of functions $f : A^n \to A$.*

We say that a subset Y of A is inductive on X *iff*

- *$X \subseteq Y$, and*
- *for every $f : A^n \to A$, for every $\overline{y_n} \in Y$: $f(\overline{y_n}) \in Y$.*

The intersection of all inductive sets on X, denoted by X^+, is called the inductive closure of X under F.

Note that X^+ is an inductive set on X which is closed under F. Frequently, X^+ is called *the least set containing X and closed under F.*

Alternatively, we can define the sequence of sets $\{X_i\}_{i\geq 0}$ defined by:

$X_0 = X$ and

$X_{i+1} = X_i \cup \{f(\overline{x_n}) \mid (f : A^n \to A) \in F, \overline{x_n} \in X_i\}$,

and $X_+ := \bigcup_{i=0}^{\infty} X_i$. It can be shown that $X^+ = X_+$.

It is often the case that we define functions inductively over an inductive closure. The existence and uniqueness of such an inductive definition is guaranteed if the inductive closure has special properties, like being freely generated.

Definition 2 (freely generated set) *Let A be a nonempty set, F a set of functions on A and X_+ the inductive closure of X under F. We say that X_+ is* freely generated by X and F *if the following conditions hold:*

1. *the restriction of every function $f : A^n \to A$ in F to X_+^n is injective,*
2. *for every $f : A^m \to A$, $g : A^n \to A$ in F: $f(X_+^m) \cap g(X_+^n) = \emptyset$,*
3. *for every $f : A^n \to A$ in F and every $\overline{x_n} \in X_+$: $f(\overline{x_n}) \notin X$.*

In logic, terms, formulae and proofs are given by inductive definitions. Another important concept is that of a function defined recursively over an inductive set freely generated.

Let A be a nonempty set, X a subset of A, F a set of functions on A and X_+ the inductive closure of X under F. Let B be any nonempty set, and let G be the set of functions over the set B, such that there is a function $d : F \to G$ that associates with every function $f : A^n \to A$ in F a function $d(f) : B^n \to B$ in G.

Lemma 1 (unique homomorphic extension theorem) *If X_+ is freely generated by F and X then for every function $h : X \to B$ there is a unique function $h^* : X_+ \to B$ such that*

1. *for all $x \in X : h^*(x) = h(x)$, and*
2. *$h^*(f(x_1, \dots, x_n)) = d(f)(h^*(x_1), \dots, h^*(x_n))$.*

The properties 1. and 2. mean that h^* is a *homomorphism*, called the *unique homomorphic extension of h.*

2.2 Universal Algebra

In this section, the notion of universal algebra is briefly outlined. In order to support typed expressions we consider the formalism of many-sorted algebra.

Many Sorted Signature

For any set S, an *S-sorted* set is a family $\{A_s\}_{s \in S}$ of sets indexed by S. The operations and relations on sets are generalized to S-sorted sets in the componentwise way. For example, $\{A_s\}_{s \in S} \subseteq \{B_s\}_{s \in S}$ iff $A_s \subseteq B_s$ for all $s \in S$.

Definition 3 (signature) *A* many-sorted signature *(*signature *for short) is a pair $\Sigma := \langle S, \mathcal{F} \rangle$ such that*

- *S is a set of* sorts *(or* types*),*
- *$\mathcal{F}$ is a (possibly empty) set,*
- *Σ is equipped with a mapping*

$$\mathit{type} : \mathcal{F} \to S^* \times S$$

which assigns to any symbol $f \in \mathcal{F}$ an expression $\mathit{type}(f) \in S^* \times S$ called the type *of f.*

A symbol f of sort $\overline{\tau_n}, \tau$ is to be interpreted as an operation taking n arguments, the i-th argument being of type τ_i, and yielding a result of type τ. We will write $f : \overline{\tau_n} \to \tau$ whenever $f \in \mathcal{F}$ with $\mathit{type}(f) = \overline{\tau_n}, \tau$.

Many-sorted algebra

Suppose Σ is a many-sorted signature. We assume that ω ranges over S^*, f ranges over $\mathcal{F}$, and $\tau, \tau_1, \tau_2, \ldots$ range over S. The *arity* $\mathrm{ar}(f)$ of $f : \omega \to \tau$ is defined as the length $|\omega|$ of the sequence $\omega \in S^*$. Symbols of arity 0 are called *constants*.

Definition 4 (Σ-algebra) *Given a signature $\Sigma = \langle S, \mathcal{F} \rangle$, a* Σ-algebra *$\mathcal{A}$ is a pair $\langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$ where $\{A_\tau\}_{\tau \in S}$ is an S-sorted family of nonempty carrier sets and α is a map such that:*

- *$\alpha(f) \in A_\tau$ if $f : \tau$,*
- *$\alpha(f)$ is a function $\alpha(f) : A_{\tau_1} \times \ldots \times A_{\tau_n} \to A_\tau$ if $f : \overline{\tau_n} \to \tau$.*

$\{A_\tau\}_{\tau \in S}$ is called the *carrier* of $\mathcal{A}$ and is denoted by $|\mathcal{A}|$.

Just as functions and equivalence relations are defined for sets, we can extend these notions to their operation preserving counterparts for algebras and name them *homomorphism* and *congruence*.

Definition 5 (homomorphism) *A* homomorphism *$h : \mathcal{A} \to \mathcal{B}$ from a Σ-algebra $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$ to a Σ-algebra $\mathcal{B} = \langle \{B_\tau\}_{\tau \in S}, \beta \rangle$ is an S-indexed set of maps $h = \{h_\tau : A_\tau \to B_\tau\}_{\tau \in S}$ such that*

- *for every $f : \tau$, $h_\tau(\alpha(f)) = \beta(f)$,*
- *for every $f : \omega \to \tau$ such that $\omega = \overline{\tau_n}$ and $a_1 \in A_{\tau_1}, \ldots, a_n \in A_{\tau_n}$, $h_\tau(\alpha(f)(a_1, \ldots, a_n)) = \beta(f)(h_{\tau_1}(a_1), \ldots, h_{\tau_n}(a_n))$.*

A *monomorphism* is an injective homomorphism, an *epimorphism* is a surjective homomorphism, and an *isomorphism* is a bijective homomorphism.

The class of Σ-algebras is denoted by $Alg(\Sigma)$. Together with the Σ-homomorphisms, it forms a category denoted by $\underline{Alg}(\Sigma)$.

Definition 6 (initial Σ-algebra) *A Σ-algebra $\mathcal{A}$ is initial in a class C of Σ-algebras if $\mathcal{A} \in C$ and for any $\mathcal{B} \in C$ there exists a unique homomorphism from $\mathcal{A}$ to $\mathcal{B}$.*

Subalgebra

A Σ-algebra $\mathcal{B} = \langle \{B_\tau\}_{\tau \in S}, \beta \rangle$ is a *subalgebra* of a Σ-algebra $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$ if

- $\{B_\tau\}_{\tau \in S} \subseteq \{A_\tau\}_{\tau \in S}$, and
- for every $f : \tau$, $\beta(f) = \alpha(f)$, and

- for every $f : \overline{\tau_n} \to \tau : \beta(f) = \alpha(f)\lceil_{B_{\tau_1} \times \ldots \times B_{\tau_n}}$.

Given a Σ-algebra $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$, let $X = \{X_\tau\}_{\tau \in S} \subseteq \{A_\tau\}_{\tau \in S}$. The *least subalgebra* of $\mathcal{A}$ containing X is the subalgebra $[X]$ of $\mathcal{A}$ whose carrier is $\{[X_\tau]\}_{\tau \in S}$ where $[X_\tau] := \bigcup_{i=0}^{\infty} [X_\tau]_i$ and

$$[X_\tau]_0 = X_\tau \cup \{\alpha(f) \mid \mathit{type}(f) = \tau\},$$

$$[X_\tau]_{i+1} = [X_\tau]_i \cup \{\alpha(f)(\overline{x_n}) \mid f : \overline{\tau_n} \to \tau, x_1 \in [X_{\tau_1}]_i, \ldots, x_n \in [X_{\tau_n}]_i\}.$$

It is easy to see that the carrier of $[X]$ is the inductive closure of X under $F := \{\alpha(f) \mid f \in \mathcal{F}\}$.

We always assume that the carriers A_τ of a Σ-algebra $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$ are nonempty. To avoid having any carrier $[X_\tau]$ of $[X]$ empty, we assume that either there exists a constant $(f : \tau) \in \mathcal{F}$, or there is some $(f : \overline{\tau_n} \to \tau) \in \mathcal{F}$ such that $[X_{\tau_i}] \neq \emptyset$ if $i \in \{1, \ldots, n\}$.

A set of Σ-variables is an S-sorted set of symbols $\mathcal{V} := \{\mathcal{V}_\tau\}_{\tau \in S}$ such that $\mathcal{V} \cap \mathcal{F} = \emptyset$.

Term Algebra

Given a signature Σ and an S-sorted set $\mathcal{V} = \{\mathcal{V}_\tau\}_{\tau \in S}$ of variables , we define the following inductive closure of strings on $\mathcal{F}$ and $\mathcal{V}$ (involving symbols '(', ')', ','):

$$[T_\tau]_0 = \mathcal{V}_\tau \cup \{f \mid \mathit{type}(f) = \tau\},$$

$$[T_\tau]_{i+1} = [T_\tau]_i \cup \{f(\overline{t_n}) \mid f : \overline{\tau_n} \to \tau, t_1 \in [T_{\tau_1}]_i, \ldots, t_n \in [T_{\tau_n}]_i\}.$$

and define the set $\mathcal{T}(\mathcal{F}, \mathcal{V}) := \{\mathcal{T}(\mathcal{F}, \mathcal{V})\}_{\tau \in S}$ where $\mathcal{T}(\mathcal{F}, \mathcal{V})_\tau := \bigcup_{i=0}^{\infty} [T_\tau]_i$. We observe that if we interpret the variable symbols of $\mathcal{V}$ as mere constants then $\mathcal{T}(\mathcal{F}, \mathcal{V})$ has a structure of Σ-algebra which is freely generated by $\mathcal{F}$ and $\mathcal{V}$. This algebra is called the *term algebra* over Σ and $\mathcal{V}$. A Σ-*term* (or simply *term*) is an element of $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

The fact that $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is free on $\mathcal{V}$ implies that for every Σ-algebra $\mathcal{A}$ and every S-sorted function $v : \mathcal{V} \to |\mathcal{A}|$ there exists a unique homomorphic extension $v^* : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to |\mathcal{A}|$. An S-indexed function $v : \mathcal{V} \to |\mathcal{A}|$ is called $\mathcal{A}$-*valuation.*

An important theoretical result is that $\mathcal{T}(\mathcal{F}) := \mathcal{T}(\mathcal{F}, \{\emptyset\}_{\tau \in S})$ has a structure of initial Σ-algebra in $\mathit{Alg}(\Sigma)$. This Σ-algebra is called the *ground term algebra* on Σ or the *Herbrand universe.*

The main operations on Σ-terms are *replacement* and *substitution.*

The replacement operation can be easily described by using the notion of position.

Definition 7 (position) *The set $\mathcal{P}os(t)$ of* positions *in a Σ-term $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ is the set of sequences of natural numbers defined inductively as follows:*

$$\mathcal{P}os(t) := \begin{cases} \{\epsilon\} & \textit{if } t \in \mathcal{V}, \\ \{\epsilon\} \cup \{i \cdot p \mid 1 \leq i \leq n, p \in \mathcal{P}os(t_i)\} & \textit{if } t = f(t_1, \ldots, t_n). \end{cases}$$

Given a term t, we define the set of variable positions of t as $\mathcal{P}os_{\mathcal{V}}(t) := \{p \in \mathcal{P}os(t) \mid t|_p \in \mathcal{V}\}$, and the set of non-variable positions of t as $\mathcal{P}os_{\mathcal{F}}(t) := \{p \in \mathcal{P}os(t) \mid t|_p \notin \mathcal{V}\}$.

Definition 8 (subterm) *Let $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ and $p \in \mathcal{P}os(t)$. The* subterm of t at position p, *denoted by $t|_p$, is:*

$$t|_p := \begin{cases} t & \textit{if } p = \epsilon, \\ (t_i)|_q & \textit{if } t = f(t_1, \ldots, t_n) \textit{ and } p = i \cdot q. \end{cases}$$

By $\mathcal{V}(t)$ we denote the set of *variables occurring in* t, i.e. $\mathcal{V}(t) := \{t|_p \mid p \in \mathcal{P}os_{\mathcal{V}}(t)\}$. Positions are partially ordered by the prefix ordering $\leq$, i.e. $p \leq q$ if there exists an r such that $p \cdot r = q$. We write $p < q$ if $p \leq q$ and $p \neq q$. Positions p, q are *disjoint*, denoted $p \perp q$, if neither $p \leq q$ nor $q \leq p$.

Definition 9 (replacement) *If $p \in \mathcal{P}os(t)$ and s is a term such that $type(s) = type(t|_p)$ then $t[s]_p$ denotes the term obtained from t by replacing the subterm at position p by the term s.*

A *substitution* is a function $\theta : \mathcal{V} \to \mathcal{T}(\mathcal{F},\mathcal{V})$ such that $\mathcal{D}(\theta) := \{X \in \mathcal{V} \mid \theta(X) \neq X\}$ is finite. The set $\mathcal{D}(\theta)$ is called the *domain* of θ. If $\mathcal{D}(\theta) = \{\overline{X_n}\}$ then we may write θ as $\{X_1 \mapsto \theta(X_1), \ldots, X_n \mapsto \theta(X_n)\}$, abbreviated $\{\overline{X_n \mapsto \theta(X_n)}\}$. The *empty substitution*, denoted by ε, is the substitution with an empty domain. The *image* of θ is the set $\mathcal{I}(\theta) := \{\theta(X) \mid X \in \mathcal{D}(\theta)\}$, and the *range* of θ is $\mathcal{R}ng(\theta) := \mathcal{V}(\mathcal{I}(\theta))$. θ is called *ground* if $\mathcal{R}ng(\theta) = \emptyset$, and *idempotent* if $\mathcal{D}(\theta) \cap \mathcal{R}ng(\theta) = \emptyset$. We denote by $\mathcal{S}ubst(\mathcal{F},\mathcal{V})$ the set of substitutions. If t is a Σ-term then we write $t\theta$ instead of $\theta^*(t)$. The *composition* of two substitutions $\theta, \sigma \in \mathcal{S}ubst(\mathcal{F},\mathcal{V})$ is the substitution $\sigma\theta$ defined by $(\sigma\theta)(x) := (\sigma(x))\theta$. If $V \subseteq \mathcal{V}$ then the restriction $\sigma\!\upharpoonright_V \in \mathcal{S}ubst(\mathcal{F},\mathcal{V})$ is

$$\sigma\!\upharpoonright_V(X) := \begin{cases} \sigma(X) & \text{if } X \in \mathcal{D}(\sigma) \cap V, \\ X & \text{otherwise.} \end{cases}$$

We write $\sigma = \theta\ [V]$ iff $\sigma\!\upharpoonright_V = \theta\!\upharpoonright_V$.

A *renaming* is a substitution of the form $\theta := \{\overline{X_n \mapsto Y_n}\}$ with $\overline{Y_n}$ distinct variables.

We say that a syntactic object G' is a fresh variant of a syntactic object G if $G' = G\theta$ with θ a renaming such that $\mathcal{D}(\theta) = \mathcal{V}(G)$ and $\mathcal{R}ng(\theta)$ contains variables which did not occur so far.

Congruences, Quotient Algebras

Given a Σ-algebra $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$, a *congruence relation over* $\mathcal{A}$ is an S-sorted equivalence relation $\sim$ on $|\mathcal{A}|$ which is compatible with all function symbols, i.e. $\sim = \{\sim_\tau\}_{\tau \in S}$ and for all $a_1, b_1 \in A_{\tau_1}, \ldots, a_n, b_n \in A_{\tau_n}$, if $a_1 \sim_{\tau_1} b_1, \ldots, a_n \sim_{\tau_n} b_n$ and $f : \tau_1, \ldots, \tau_n \to \tau$ then $\alpha(f)(a_1, \ldots, a_n) \sim_\tau \alpha(f)(b_1, \ldots, b_n)$.

If $\sim$ is a congruence over $|\mathcal{A}|$ then $\mathcal{A}/_\sim := \langle \{A_\tau/_{\sim_\tau}\}_{\tau \in S}, \alpha/_\sim \rangle$ is a Σ-algebra, where

- $\alpha/_\sim(f) = [\alpha(f)]$ for every constant f,
- $\alpha/_\sim(f([a_1], \ldots, [a_n])) = [\alpha(f)(a_1, \ldots, a_n)]$ for every $f : \tau_1, \ldots, \tau_n \to \tau$ and $a_1 \in A_{\tau_1}, \ldots, a_n \in A_{\tau_n}$.

Here $[a]$ denotes the set $\{b \mid b \sim_\tau a\}$ if $a \in A_\tau$. The algebra $\mathcal{A}/_\sim$ is called the *quotient algebra of* $\mathcal{A}$ *associated with* $\sim$.

Conversely, every Σ-algebra $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$ induces a congruence relation $\sim^{\mathcal{A}}$ on $\mathcal{T}(\mathcal{F})$ defined by $t_1 \sim^{\mathcal{A}} t_2$ if $t_1^{\mathcal{A}} = t_2^{\mathcal{A}}$, where

$$t^{\mathcal{A}} := \begin{cases} \alpha(t) & \text{if } t \text{ is a constant in } \mathcal{F} \\ \alpha(f)(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}}) & \text{if } t = f(\overline{t_n}) \end{cases}$$

2.3 General Logic

General logic is a powerful formalism which is helpful in getting an intuitive understanding of the main logical notions related to the development of functional logic and concurrent constraint programming languages.

In this section we review the main concepts and properties of general logic which are used in this thesis. The main reference is [Mes89].

We will make use of a few basic categorical notions when introducing the main concepts of general logics, but familiarity with category theory is not necessary in order to obtain an intuitive understanding of the relevant logical notions. A reader unfamiliar with category theory may get an informal but easy to understand reading by translating some of the categorical concepts into their set-theoretic approximations, for example by translating "category" to "class" or "set", "functor" to "function", and so on. For a good introduction to the main categorical concepts we refer to [Pie91].

The key ingredients of a general axiomatic theory of logics are: a *syntax*, a notion of *entailment* of a sentence from a set of sentences, a notion of *model*, and a notion of *satisfaction* of a sentence by a model. It is desirable to have also a notion of *proof calculus*, which formalizes the proofs of entailments.

The syntax of a logic is typically given by a *signature* Σ together with a grammar which describes how to build *sentences.* For first-order logic, a typical signature consists of a set of function symbols and a set of predicates, each with a prescribed type, which are used to build up the usual sentences. To keep the formalism as general as possible, we only say that for each logic there is a category $\underline{Sign}$ of possible signatures for it, and a functor *sen* assigning to each signature Σ the set $sen(\Sigma)$ of all its Σ-sentences.

In general logic, the meaning of sentences can be established in two ways:

1. by the *satisfaction relation* between models and sentences, or
2. by the *entailment relation* between sentences.

Thus, in the framework of general logic, a logic is characterized by an *abstract deduction system* which defines the entailment relation of the logic, and a model class, called *institution* in [Mes89], which provides the interpretation of the sentences of the logic. General logic provides a precise axiomatization of logic by viewing logic as a harmonious relationship between its proof-theoretic structure and its model satisfaction structure.

2.3.1 Entailment Systems

The entailment system of a logic characterizes its proof -theoretic structure by asserting the *provability* of a sentence ϕ with respect to a set of sentences Γ. The sentences in Γ can be seen as *assumptions* and ϕ as a *conclusion.*

Formally, an entailment system can be defined as follows:

Definition 10 (entailment system) *An* entailment system *is a triple* $\mathcal{E} = \langle \underline{Sign}, sen, \vdash \rangle$ *with* $\underline{Sign}$ *a category of signatures,* $sen : \underline{Sign} \to \underline{Set}$ *a functor, and* $\vdash: \underline{Sign} \to \underline{Set}$ *a functor mapping each signature* Σ *to a relation* $\vdash_\Sigma \subseteq 2^{\mathrm{sen}(\Sigma)} \times sen(\Sigma)$ *such that the following properties are satisfied:*

reflexivity $\forall \phi \in sen(\Sigma),\ \{\phi\} \vdash_\Sigma \phi$

monotonicity *if* $\Gamma \vdash_\Sigma \phi$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Gamma' \vdash_\Sigma \phi$

transitivity *if* $\Gamma \vdash_\Sigma \phi_i$ *for* $i \in I$ *and* $\Gamma \cup \{\phi_i \mid i \in I\} \vdash_\Sigma \xi$ *then* $\Gamma \vdash_\Sigma \xi$.

Entailment systems define an abstract relation between sentences. By regarding the sentences as axioms, an entailment system deduces theorems with respect to the axioms.

Definition 11 (theory, theorem) *Let* $\langle \underline{Sign}, sen, \vdash \rangle$ *be an entailment system and* $\Gamma \subseteq sen(\Sigma)$. *We call* theory of Γ *the set* $\{\phi \mid \Gamma \vdash_\Sigma \phi\}$ *of deductive consequences of* Γ. *We denote by* Γ^* *the theory of* Γ. *A* theorem of Γ *is an element of* Γ^*.

Thus, for a given entailment system, a theory can be represented as a pair $T = \langle \Sigma, \Gamma^* \rangle$, where Γ are the axioms of T. Whenever a finite set of sentences Γ is given, Γ itself can be regarded as the *presentation* or the *axiomatization* of the theory. For simplicity, we may identify theories with their presentations, i.e. $T = \langle \Sigma, \Gamma \rangle$ instead of $\langle \Sigma, \Gamma^* \rangle$, since the presentations (programs) are of interest to us.

We associate to an entailment system $\langle \underline{Sign}, sen, \vdash \rangle$ the category $\underline{Th}$ of its theories, which has as objects pairs $T = \langle \Sigma, \Gamma \rangle$ with $\Gamma \subseteq sen(\Sigma)$. A morphism $H : \langle \Sigma, \Gamma \rangle \to \langle \Sigma', \Gamma' \rangle$ of $\underline{Th}$ is a signature morphism $H : \Sigma \to \Sigma'$ such that if $\phi \in \Gamma$ then $\Gamma' \vdash_{\Sigma'} \in H(\phi)$.

2.3.2 Models

The model theoretical semantics of a logic system is given by assigning a meaning to each syntactic entity of the logic. The axiomatization of the model theory is captured by the notion of *institution*.

Definition 12 (institution) *An* institution *is a tuple* $\langle \underline{Sign}, sen, Mod, \models \rangle$ *where*

- $\underline{Sign}$ *is the category of signatures,*
- $sen : \underline{Sign} \to \underline{Set}$ *is the functor which associates to each signature* Σ *the set* $sen(\Sigma)$ *of its sentences,*
- $Mod : \underline{Sign} \to \underline{Cat}$ *is a contravariant functor associating to each signature* Σ *the category* $\underline{Mod}(\Sigma)$ *of all its models. We denote by* $Mod(\Sigma)$ *the collection of objects of* $\underline{Mod}(\Sigma)$*,*
- $\models$ *is a function associating with each signature* Σ *a binary relation* $\models_\Sigma \subseteq Mod(\Sigma) \times sen(\Sigma)$ *called the* satisfaction relation *between models and* Σ*-sentences such that the following condition holds:*

$$\forall M' \in Mod(\Sigma'), \forall H : \Sigma \to \Sigma', \forall \phi \in sen(\Sigma) : \\ H^{op}(M') \models_\Sigma \phi \Leftrightarrow M' \models_{\Sigma'} H(\phi).$$

Given a set of sentences Γ, we denote by $\underline{Mod}(\Sigma, \Gamma)$ the subcategory of $\underline{Mod}(\Sigma)$ consisting of all models in $\underline{Mod}(\Sigma)$ satisfying all the sentences in Γ, i.e.:

$$Mod(\Sigma, \Gamma) := \{M \in Mod(\Sigma) \mid \forall \phi \in \Gamma . M \models_\Sigma \phi\}.$$

It can be shown that the relation $\models^*_\Sigma \subseteq 2^{\mathrm{sen}(\Sigma)} \times sen(\Sigma)$ defined as

$$\Gamma \models^*_\Sigma \phi \text{ iff } \forall M \in Mod(\Sigma, \Gamma) . M \models_\Sigma \phi$$

is an entailment relation. A sentence ϕ is a *logic consequence of* Γ iff $\Gamma \models^*_\Sigma \phi$.

2.3.3 Logic

A logic system is a sound combination of an entailment system with an institution. By soundness we mean that all deduced theorems of a presentation Γ must be its logic consequences. Formally:

Definition 13 (logic) *A logic is a tuple* $\mathcal{L} = \langle \underline{Sign}, sen, Mod, \vdash, \models \rangle$ *such that:*

1. $\mathcal{E} = \langle \underline{Sign}, sen, \vdash \rangle$ *is an entailment system,*
2. $\mathcal{I} = \langle \underline{Sign}, sen, Mod, \models \rangle$ *is an institution,*
3. *If* $\langle \underline{Sign}, sen, \models^* \rangle$ *is the generic entailment system associated with* $\mathcal{I}$ *then the following* soundness *condition holds: For any* $\Sigma \in Sign$, *and* $\phi \in sen(\Sigma)$
$$\forall \Gamma \subseteq_{\text{fin}} sen(\Sigma).\Gamma \vdash_\Sigma \phi \Rightarrow \Gamma \models^*_\Sigma \phi$$

$\mathcal{L}$ *is* complete *if*

$$\forall \Gamma \subseteq_{\text{fin}} sen(\Sigma).\Gamma \vdash_\Sigma \phi \Leftarrow \Gamma \models^*_\Sigma \phi$$

This definition emphasizes the proof-theoretic and model-theoretic sides of a logic. We have seen that any institution $\mathcal{I} = \langle \underline{Sign}, sen, \underline{Mod}, \models \rangle$ can be always associated with an entailment system $\mathcal{E} = \langle \underline{Sign}, sen, \vdash \rangle$ by taking $\vdash_\Sigma := \models^*_\Sigma$. On the other hand, it has been shown [Mes89] that any entailment system $\mathcal{E} = \langle \underline{Sign}, sen, \vdash \rangle$ can be associated with an institution $\mathcal{I} = \langle \underline{Sign}, sen, Mod, \models \rangle$. When the signature Σ is given, the logic may be given either as an entailment system $\langle sen(\Sigma), \vdash_\Sigma \rangle$ or as an institution $\langle sen(\Sigma), \underline{Mod}(\Sigma), \models_\Sigma \rangle$ where $sen(\Sigma)$ is called the *language* of the logic.

2.3.4 Proof Calculi

A proof calculus realizes the entailment relation of a logic. The entailment system of a logic can be realized by many different proof calculi. For example, in first order logic we have Hilbert style, natural style and sequent calculi among others, and the way in which proofs are represented and generated by rules of deduction is different for each of these calculi. Therefore, it is quite reasonable to axiomatize separately an abstract notion of deduction calculus, called *proof calculus*, for a given logic.

What a proof calculus does is to associate to each theory T a structure $P(T)$ of proofs that use axioms of T as hypotheses. $P(T)$ has an algebraic structure (e.g., a proof tree) that allows to obtain new proofs out of previously given proofs by operations that mirror the deduction rules of the calculus in question. Such a structure can be abstractly modeled by a particular category $\underline{Struc}$.

Definition 14 (proof calculus) *A* proof calculus *is a tuple*

$$\mathcal{C} = \langle \underline{Sign}, sen, \vdash, P, proof, \pi \rangle$$

where

1. $\langle \underline{Sign}, sen, \vdash \rangle$ *is an entailment system,*

2. $P : \underline{Th} \to \underline{Struc}$ *is a functor which associates a structure* $P(T)$ *to each theory* T*,*

3. $proof : \underline{Struc} \to \underline{Set}$ *is a functor which associates to each theory* T *its* proof set $proofs(T) := proof(P(T))$,

4. *a natural transformation* $\pi : proofs \to sen$ *such that for each theory* $T = \langle \Sigma, \Gamma \rangle$ *the function* $\pi_T : proofs(T) \to sen(\Sigma)$ *satisfies*

$$\phi \in \pi_T(proofs(T)) \Leftrightarrow \Gamma \vdash_\Sigma \phi$$

   i.e., $\pi_T(proofs(T)) = \Gamma^*$ *for any theory* $T = \langle \Sigma, \Gamma \rangle$. π_T *is called the* theorem projection function *of the theory* T.

Usually, proof calculi have a specialized nature, in the sense that only certain signatures are admissible as syntax (e.g., finite signatures), only certain axioms are allowed as axioms, and only certain sentences are allowed as conclusions. The obvious reason for imposing such restrictions is that proofs are more efficient under the given restrictions. For example, the restriction of axioms to Horn clauses in logic programming makes resolution much more efficient; the restriction of axioms to confluent rewrite systems in equational logic programming makes equational deduction enormously more efficient than unrestricted equational deduction. These considerations lead to the notion of *proof subcalculus*, which is just like a proof calculus, except that appropriate restrictions are imposed as follows:

- a subclass of *admissible signatures* is specified;

- for each admissible signature Σ, a family of sets $\Gamma \subseteq 2^{sen(\Sigma)}$, called sets of *admissible axioms* is also specified;

- for each admissible signature Σ, a subset $conc(\Sigma) \subseteq sen(\Sigma)$, called sets of *admissible conclusions* is also specified;

- the assignments $P(T)$, $proofs(T)$ and π_T are similar to those in a proof calculus, except that they are restricted to theories $T = \langle \Sigma, \Gamma \rangle$ having admissible signature and axioms, and π_T maps an admissible proof $p \in proofs(T)$ to an admissible conclusion $\pi_T(p) \in conc(\Sigma)$.

For effective computations, Meseguer proposes the notion of *effective proof subcalculus*, which is derived from the formalism of proof subcalculus by providing additional axioms that are useful in making the calculus mechanizable.

2.4 First-Order Logic with Equality

In this section we define the first-order logic with equality as an instance of general logic.

Syntax

First we define the language of first-order equational logic. We assume given a multi-sorted signature $\Sigma = \langle S, \mathcal{F} \cup \Pi \rangle$ with the following characteristics:

- S is a nonempty set of sorts which contains the special sort *bool*,
- $\mathcal{F}$ is a finite (possibly empty) set of function symbols,
- Π is a set of predicate symbols; it is assumed that Π satisfies the following conditions:
  - if $p \in \Pi$ then $p : \omega \to \texttt{bool}$
  - for every $\tau \in S$, the equality symbol $\approx_\tau : \tau, \tau \to bool$ is in Π
  - the constants $\texttt{true} : bool$ and $\texttt{false} : bool$ are in Π.
- $(\mathcal{F} \setminus \{\texttt{true}, \texttt{false}\}) \cap \Pi = \emptyset$

and an S-sorted set of variables $\mathcal{V} = \{\mathcal{V}_\tau\}_{\tau \in S \cup bool}$ such that $\mathcal{V}_\tau$ is an infinite set for every $\tau \in S$.

A Σ*-term* (*term* for short) is an element of $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We write $t : \tau$ whenever $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})_\tau$.

An *atomic* Σ*-formula* is an element of the set defined inductively as follows:

- if $p : bool \in \Pi$ then p is an atomic Σ-formula
- if $t_1 \in \mathcal{T}(\mathcal{F}, \mathcal{V})_{\tau_1}, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})_{\tau_n}$ and $p : \overline{\tau_n} \to bool$ then $p(\overline{t_n})$ is an atomic Σ-formula.

An atomic Σ-formula of the form $t_1 \approx_\tau t_2$, abbreviated $t_1 \approx t_2$, is called *equation*. We denote the set of equations by $\mathcal{E}q(\mathcal{F}, \mathcal{V})$. The set $sen(\Sigma)$ of Σ-sentences is the least set satisfying the following properties:

- every atomic Σ-formula is in $sen(\Sigma)$,

- if $G, H \in sen(\Sigma)$ then $(G \wedge H) \in sen(\Sigma)$ and $\neg F \in sen(\Sigma)$,
- if $X \in \mathcal{V}_\tau$ and $G \in sen(\Sigma)$ then $(\forall X : \tau.G) \in sen(\Sigma)$ and $(\exists X : \tau.G) \in sen(\Sigma)$.

Models, Satisfiability

The models of first-order equational logic are Σ-algebras. A Σ-algebra over a multi-sorted signature $\Sigma = \langle S, \mathcal{F} \cup \Pi \rangle$ of a first-order logic with equality is a Σ-algebra $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$ which satisfies the following conditions:

- $A_{bool} = \{true, false\}$ where $\{true, false\}$ is a boolean domain
- $\alpha(\texttt{true}) = true$, $\alpha(\texttt{false}) = false$ and $\alpha(\approx_\tau) = =_\tau$ where $=_\tau$ is the equality operator over A_τ.

The institution of a first-order equational logic is $\langle \Sigma, sen, Alg, \models \rangle$, where the satisfiability relation is defined as follows:

Definition 15 (satisfiability) *For any Σ-algebra $\mathcal{A}$ and $\mathcal{A}$-valuation $v : \mathcal{V} \to |\mathcal{A}|$ and $G \in sen(\Sigma)$ the relation $\mathcal{A}$* satisfies *G* w.r.t. *v, written as $\mathcal{A}, v \models G$, is inductively defined as follows:*

- *$\mathcal{A}, v \models_\Sigma p(\overline{t_n})$ iff $\alpha(p)(v^*(t_1), \ldots, v^*(t_n))$ holds,*
- *$\mathcal{A}, v \models_\Sigma \neg G$ iff $(\mathcal{A}, v \models_\Sigma G)$ does not hold,*
- *$\mathcal{A}, v \models_\Sigma (G \wedge H)$ iff $(\mathcal{A}, v \models_\Sigma G)$ and $(\mathcal{A}, v \models_\Sigma H)$,*
- *$\mathcal{A}, v \models_\Sigma \forall X : \tau.G$ iff $(A, v_X \models_\Sigma G)$ for all valuations $v_X : \mathcal{V} \to |\mathcal{A}|$ with $v_X(Y) = Y$ for all $Y \neq X$,*
- *$\mathcal{A}, v \models_\Sigma \exists X : \tau.G$ iff there exists $a \in A_\tau$ such that $A, v[X := a] \models_\Sigma (G)$ for some $a \in A_\tau$. Here $v[X := a]$ is the valuation defined by $v[X := a](Y) = v(Y)$ for all $Y \neq X$ and $v[X := a](X) := a$.*

The relation $A \models_\Sigma G$ holds iff for any valuation $v : \mathcal{V} \to |\mathcal{A}| : A, v \models_\Sigma G$ holds, where v ranges over the set of all $\mathcal{A}$-valuations.

Given a Σ-sentence $G \in sen(\Sigma)$ we denote by $\mathcal{V}(G)$ the set of free variables in G. If t is a Σ-term then $\mathcal{V}(t)$ denotes the set of variables in t.

Definition 16 (universal closure, existential closure) *The* universal closure *of a sentence G is the sentence $\forall \overline{X_n}.G$, abbreviated $\forall G$, where $\mathcal{V}(G) = \{X_1, \ldots, X_n\}$.*

The existential closure *of G is the sentence $\exists \overline{X_n}.G$, abbreviated $\exists G$, where $\mathcal{V}(G) = \{X_1, \ldots, X_n\}$.*

Entailment

Let E be a set of Σ-sentences. For testing whether a Σ-sentence ϕ is valid (i.e., if $E \models^*_\Sigma G$) there are two logical results of central importance:

- Gödel's Completeness theorem: there exists an entailment system $\mathcal{E} = \langle sen(\Sigma), \vdash_\Sigma \rangle$ such that $E \models^*_\Sigma G \Leftrightarrow E \vdash_\Sigma G$.

- Church's undecidability of validity: there is no decision procedure (i.e., a procedure which always terminates) for deciding whether a formula is valid. In another words, any procedure for testing the validity of a formula (or, according to Gödel's completeness theorem, the provability in a complete logic) must run forever when given certain non-true formulae as input.

Proof Calculus

We adopt here a sequent style representation to define an effective proof calculus. With this formalization, logical deduction becomes a search for certain sequent proofs. A sequent $E \vdash_\Sigma \phi$ denotes the state of a proof procedure which attempts to determine whether ϕ follows from E. A proof calculus which realizes the entailment relation of the logic is based on a set $\mathcal{C}_s$ of deduction rules (also called *inference rules*) of the form

$$\frac{s_1 \ \dots \ s_n}{s}$$

which assert the provability of the sequent s from the provabilities of the sequents $s_1, \dots, s_n$. A proof of the entailment of the sequent $s_0 = E \vdash_\Sigma \phi$ is a tree with root s_0 and empty sequents as leaves. The presence of a sequent s with sons $s_1, \dots, s_n$ in the proof tree is justified by the existence of an inference rule $\frac{s_1 \ \dots \ s_n}{s} \in \mathcal{C}_s$. The construction of a proof tree can be realized with the following search function on sets

$$search(S \cup \{s\}) = \left\{ search(S \cup \{s_1, \dots, s_n\}) \mid \frac{s_1 \ \dots \ s_n}{s} \in \mathcal{C}_s \right\}.$$

The entailment relation of an equational logic is realized by a proof calculus $\mathcal{C} = \langle \Sigma, sen, P, proof, \pi \rangle$ where

- $P : \underline{Th} \to \underline{Struc}$ is a functor mapping each equational theory $T = \langle \Sigma, E \rangle$ to the category $P(T)$ whose objects are sets of sequents $\{\overline{s_n}\}$ and whose morphisms are sequences of elementary steps of the form $S \cup \{s\} \stackrel{\mathcal{C}_s}{\Longrightarrow} S \cup \{\overline{s_n}\}$ where $\frac{s_1 \ \dots \ s_n}{s} \in \mathcal{C}_s$

- for each theory $T = \langle \Sigma, E \rangle$, the functor *proof* maps $P(T)$ to the set $\mathit{proof}(P(T)) = \{\{E \vdash_\Sigma \phi\} \overset{\mathcal{C}_s}{\Longrightarrow}{}^* \{\} \mid (\{E \vdash_\Sigma \phi\} \overset{\mathcal{C}_s}{\Longrightarrow}{}^* \{\}) \in P(T)\}$
- for each theory $T = \langle \Sigma, E \rangle$, we define

$$\pi_T(\mathit{proof}(P(T))) := \{\phi \mid (\{E \vdash_\Sigma \phi\} \overset{\mathcal{C}_s}{\Longrightarrow}{}^* \{\}) \in \mathit{proof}(P(T))\}.$$

The generation of a proof with a search function is highly nondeterministic, because of:

1. the choice of the node which is expanded next,
2. the choice of the inference rule to be applied to the selected node.

For programming purposes, effective proof subcalculi which refine the naive generation of proofs by *search* have been developed.

2.4.1 Effective Proof Subcalculi

We recall here the most popular proof subcalculi used in first-order equational deduction: demodulation, unification, paramodulation, term rewriting, and narrowing.

Demodulation

Demodulation is an effective proof subcalculus which realizes entailment relations of the form $E \vdash_\Sigma s \approx t$ where E is a set of equations and $s \approx t$ is an equation. From the soundness condition of equational logic, the following implication must hold $E \models^*_\Sigma s \approx t \Rightarrow E \vdash_\Sigma s \approx t$.

An important theoretical result is that $\mathit{Alg}(\Sigma)/_{\sim^E}$ is an initial algebra in $\mathit{Alg}(\Sigma, E)$, where $\sim^E$ is the congruence induced by E on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. (See [MG85] for a detailed proof.) Therefore, deciding whether $E \models^*_\Sigma s \approx t$ amounts to deciding whether $\mathit{Alg}(\Sigma)/_{\sim^E} \models_\Sigma s \approx t$.

G.Birkhoff [Bir35] gave the following system of inference rules for realizing the entailments of the form $E \models^*_\Sigma s \approx t$:

[G1] $E \vdash_\Sigma t \approx t$

[G2] $\dfrac{E \vdash_\Sigma s \approx t}{E \vdash_\Sigma t \approx s}$

[G3] $\dfrac{E \vdash_\Sigma s \approx t \quad E \vdash_\Sigma t \approx u}{E \vdash_\Sigma s \approx u}$

[G4] $$\frac{E \vdash_\Sigma s_1 \approx t_1 \quad \ldots \quad E \vdash_\Sigma s_n \approx t_n}{E \vdash_\Sigma f(\overline{s_n}) \approx f(\overline{t_n})}$$
if $f : \overline{\tau_n} \to \tau$ is an operator of appropriate type

[G5] $$\frac{E \vdash_\Sigma s \approx t}{E \vdash_\Sigma s\theta \approx t\theta} \text{ where } \theta \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$$

---

and showed that $Alg(\Sigma)/_{\sim E} \models_\Sigma s \approx t$ iff $E \vdash_\Sigma s \approx t$. Based on this theoretical result, we can prove an entailment $E \vdash_\Sigma s \approx t$ by using the *search* function with the inference rules (G1)-(G5). This method is very nondeterministic, and thus too inefficient for computation purposes: a better method is the one based on demodulation. The demodulation calculus consists of two deduction rules: [G1] and [dem], where

[dem] $$\frac{E \vdash_\Sigma s \simeq t}{E \vdash_\Sigma s[r\theta]_p \simeq t}$$

where $(l \simeq r) \in E$, $p \in \mathcal{P}os(s)$, and $s|_p = l\theta$. Here, $s \simeq t$ stands for $s \approx t$ or $t \approx s$. Note that a [dem]-deduction step requires the computation of a substitution θ such that $s|_p = l\theta$. A substitution θ for which $\mathcal{D}(\theta) \subseteq \mathcal{V}(t)$ and $s = t\theta$ is called *matcher* of t with s. The existence of matchers is decidable; moreover, if a matcher exists then it is unique and computable.

A demodulation refutation is a sequence of demodulation steps

$$s \approx t \overset{\mathrm{DM}}{\Longrightarrow}_{[\mathrm{dem}]} s_1 \approx t_1 \overset{\mathrm{DM}}{\Longrightarrow}_{[\mathrm{dem}]} \ldots \overset{\mathrm{DM}}{\Longrightarrow}_{[\mathrm{dem}]} u \approx u \overset{\mathrm{DM}}{\Longrightarrow}_{[\mathrm{G1}]} \Box.$$

Such a refutation corresponds to the following proof by demodulation of $E \vdash_\Sigma s \approx t$:

$$\{E \vdash_\Sigma s \approx t\} \overset{\mathrm{DM}_s}{\Longrightarrow} E \vdash_\Sigma s_1 \approx t_1 \overset{\mathrm{DM}_s}{\Longrightarrow} \ldots \overset{\mathrm{DM}_s}{\Longrightarrow} \{E \vdash_\Sigma u \approx u\} \overset{\mathrm{DM}_s}{\Longrightarrow} \{\}.$$

Demodulation is a sound and complete proof calculus, i.e. $E \vdash_\Sigma s \approx t$ if and only if there exists a proof by demodulation $s \approx t \overset{\mathrm{DM}}{\Longrightarrow}^* \Box$.

Unification

Unification is concerned with realizing entailments of the form $\emptyset \vdash_\Sigma \exists G$, where G is a sentence of the form $(s_1 \approx t_1) \wedge \ldots \wedge (s_n \approx t_n)$, abbreviated $\bigwedge_{i=1}^n (s_i \approx t_i)$. Because $\mathcal{T}(\mathcal{F})$ is initial in $Alg(\Sigma)$, we have that $\emptyset \models_\emptyset^* \exists G$ iff $\mathcal{T}(\mathcal{F}) \models \exists G$ iff there exists a ground substitution $\theta \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ such that $\mathcal{T}(\mathcal{F}) \models_\Sigma G\theta$, or equivalently, that $s_i\theta \approx t_i\theta$ for all $i \in \{1, \ldots, n\}$.

We call *unifier* of $\bigwedge_{i=1}^n (s_i \approx t_i)$ any substitution $\theta \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ such that $s_i\theta = t_i\theta$ for all $i \in \{1, \ldots, n\}$, and denote by $\mathcal{U}(G)$ the set of unifiers of G. It is not difficult to see that $\mathcal{U}(G)$ is an ideal of $\mathcal{S}ubst(\mathcal{F}, \mathcal{V})$, i.e. if $\theta \in \mathcal{U}(G)$ and $\gamma \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ then $\theta\gamma \in \mathcal{U}(G)$. This suggests to define the following relation on $\mathcal{S}ubst(\mathcal{F}, \mathcal{V})$:

Let $V \subseteq \mathcal{V}$ and $\theta_1, \theta_2 \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$. We say that θ_1 *V-subsumes* θ_2, notation $\theta_1 \leq^V \theta_2$ if there exists $\gamma \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ such that $\theta_2 = \theta_1\gamma\ [V]$. V is omitted when $V = \mathcal{V}$.

It is easy to see that $\leq^{\mathcal{V}(G)}$ is a quasi-order on $\mathcal{U}(G)$ and that

$$\theta \in \mathcal{U}(G) \wedge \theta \leq^{\mathcal{V}(G)} \gamma \Rightarrow \gamma \in \mathcal{U}(G).$$

We denote by $<^V$ the partial order induced by $\leq^V$. An important theoretical result is that $<^V$ is well-founded. A minimal element of $\mathcal{U}(G)$ with respect to $<^{\mathcal{V}(G)}$ is called *most general unifier of* G, and most general unifiers are unique modulo renaming. We write $\theta = mgu(G)$ to express the fact that θ is a most general unifier of G.

Proving that $\emptyset \vdash_\Sigma \exists G$ amounts to proving that G has a most general unifier. Most general unifiers can be computed with the inference rules of the calculus UN shown in Fig. 2.1. The expression $s \simeq t$ stands for $s \approx t$

$$[\text{del}]\ \frac{\emptyset \vdash_\Sigma \exists(G_1 \wedge (t \approx t) \wedge G_2)}{\emptyset \vdash_\Sigma \exists(G_1 \wedge G_2)}$$

$$[\text{v}]\ \frac{\emptyset \vdash_\Sigma \exists(G_1 \wedge (X \simeq t) \wedge G_2)}{\emptyset \vdash_\Sigma \exists(G_1 \wedge G_2)\theta} \text{ where } X \notin \mathcal{V}(t) \text{ and } \theta = \{X \mapsto t\}$$

$$[\text{dec}]\ \frac{\emptyset \vdash_\Sigma \exists(G_1 \wedge f(\overline{s_n}) \approx f(\overline{t_n}) \wedge G_2)}{\emptyset \vdash_\Sigma \exists(G_1 \wedge \bigwedge_{i=1}^n (s_i \approx t_i) \wedge G_2)}$$

Fig. 2.1: The calculus UN: inference rules for unification

or $t \approx s$. A convenient representation of an unification step resulted from the application of a deduction rule $\dfrac{\emptyset \vdash_\Sigma \exists G}{\emptyset \vdash_\Sigma \exists G'}$ with label $\alpha \in \{[\text{dec}],[\text{del}]\}$ and computed substitution θ is[1] $G \overset{\text{UN}}{\Longrightarrow}_{\alpha,\theta} G'$. The subscript α is omitted when irrelevant. A UN-refutation is a sequence

$$G \overset{\text{UN}}{\Longrightarrow}_{\theta_1} G_1 \overset{\text{UN}}{\Longrightarrow}_{\theta_2} \ldots \overset{\text{UN}}{\Longrightarrow}_{\theta_n} \square$$

abbreviated $G \overset{\text{UN}}{\Longrightarrow}{}^*_\theta \square$ where $\theta := \theta_1 \ldots \theta_n$. It is well known that $\emptyset \vdash_\Sigma \exists G$ iff there PU-refutation $G \overset{\text{UN}}{\Longrightarrow}{}^*_\theta \square$. Moreover, $\theta = mgu(G)$.

Paramodulation

Paramodulation is an effective proof subcalculus which realizes entailment relations of the form $E \vdash_\Sigma \exists G$ where E is a set of equations and G is of the

[1] the substitution computed upon a [dec] or [del] inference step is assumed to be ε

form $(s_1 \approx t_1) \wedge \ldots \wedge (s_n \approx t_n)$, abbreviated $\bigwedge_{i=1}^{n}(s_i \approx t_i)$. Such a sentence G is called a *goal*.

According to Birkhoff's theoretical result, proving $E \vdash_\Sigma \exists G$ amounts to proving that there exists a substitution $\theta : \mathcal{V}(G) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $E \vdash_\Sigma G\theta$. In the literature, such a substitution θ is called *E-unifier* or *E-solution* of G. We denote by $\mathcal{U}_E(G)$ the set of unifiers of a goal G.

The set $\mathcal{U}_E(G)$ is an ideal of $\mathcal{S}ubst(\mathcal{F}, \mathcal{V})$, i.e. if $\theta \in \mathcal{U}_E(G)$ and $\gamma \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ then $\theta\gamma \in \mathcal{U}_E(G)$. It is easy to see that $\leq^{\mathcal{V}(G)}$ is a quasi-order on $\mathcal{U}_E(G)$.

For programming purposes it is desirable to compute a *complete* set of E-unifiers of G, i.e. a set $c\mathcal{U}_E(G) \subseteq \mathcal{U}_E(G)$ which satisfies:

- if $\gamma \in \mathcal{U}_E(G)$ then there exists $\theta \in \mathcal{U}_E(G)$ such that $\theta \leq^{\mathcal{V}(G)} \gamma$.

Paramodulation is an effective proof subcalculus which generates proofs from which a complete set of E-unifiers can be extracted. It consists of only two inference rules:

$$[\text{u}] \quad \frac{E \vdash_\Sigma \exists(G_1 \wedge (s \approx t) \wedge G_2)}{E \vdash_\Sigma \exists(G_1\theta \wedge G_2\theta)} \text{ if } \theta = mgu(s,t),$$

$$[\text{pm}] \quad \frac{E \vdash_\Sigma \exists(G_1 \wedge (s \simeq t) \wedge G_2)}{E \vdash_\Sigma \exists(G_1\theta \wedge ((s[r]_p)\theta \simeq t\theta) \wedge G_2\theta)}$$

where $l \simeq r$ is a fresh variant of an equation in E, $p \in \mathcal{P}os(s)$, and $s|_p = l\theta$. Given $E \subseteq \mathcal{E}q(\mathcal{F}, \mathcal{V})$, a paramodulation step (PM-step for short) is an expression of the form $G \overset{\text{PM}}{\Longrightarrow}_\theta G'$ where $\dfrac{G}{G'}$ is either a [u]-step with computed substitution θ or a [pm]-step with computed substitution θ.

Note that performing a [u]-step requires the computation of an mgu : this can be achieved with the unification calculus. Thus the unification calculus is a subcalculus of paramodulation. A proof by paramodulation of the entailment $E \vdash_\Sigma \exists G$ is a sequence of steps:

$$G \overset{\text{PM}}{\Longrightarrow}_{\theta_1} G_1 \overset{\text{PM}}{\Longrightarrow}_{\theta_2} \ldots \overset{\text{PM}}{\Longrightarrow}_{\theta_n} \Box,$$

abbreviated $G \overset{\text{PM}}{\Longrightarrow}{}^*_\theta \Box$ where $\theta := \theta_1 \ldots \theta_n$. The main properties of paramodulation are:

Soundness if $G \overset{\text{PM}}{\Longrightarrow}{}^*_\theta \Box$ then $\theta \in \mathcal{U}_E(G)$, and

Completeness if $\gamma \in \mathcal{U}_E(G)$ then there exists $G \overset{\text{PM}}{\Longrightarrow}{}^*_\theta \Box$ with $\theta \leq^{\mathcal{V}(G)} \gamma$.

Soundness guarantees that paramodulation is an effective proof subcalculus which realizes entailments of the form $E \vdash_\Sigma \exists \bigwedge_{i=1}^{n}(s_i \approx t_i)$, and completeness ensures its appropriateness for computing a complete set of E-unifiers.

Term Rewriting

The demodulation deduction rule is inconvenient for an operational semantics because of the high nondeterminism due to the choice of the axiom to be used and of the term position where demodulation takes place. Term rewriting is a specialization of demodulation which is designed to reduce these sources of nondeterminism.

Term rewriting is based on the idea of replacing equals by equals. Following this idea, equations between terms are oriented into rewrite rules. For instance, the equations $0+X \approx X$ and $X+succ(Y) \approx succ(X+Y)$ form an algebraic specification of the function $+$ assuming the term constructors 0 and *succ*. We can orient these equations and obtain two (rewrite) rules: $0+X \to X$ and $X+succ(Y) \to succ(X+Y)$. With orientation, we gain an operational model: reduction. We can reduce a term with the rules of $+$, e.g.:

$$a + succ(0+b) \to a + succ(b) \to succ(a+b).$$

Formally, a *rewrite rule* is an oriented equation written in the form $l \to r$ with $l, r \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $l \notin \mathcal{V}$ and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. A *term rewriting system* (TRS for short) is a finite set of rewrite rules. The *rewriting relation* $\to_{\mathcal{R}} \subseteq \mathcal{T}(\mathcal{F},\mathcal{V}) \times \mathcal{T}(\mathcal{F},\mathcal{V})$ with respect to a TRS $\mathcal{R}$ is defined as follows:

$$s \to_{\mathcal{R}} s[r\theta]_p \text{ if } s|_p = l\theta \text{ for some } p \in \mathcal{P}os_{\mathcal{F}}(s) \text{ and } (l \to r) \in \mathcal{R}.$$

The term $l\theta$ is called a *redex* and we say *s rewrites to t by contracting redex $l\theta$*. The expression $s \to_{\mathcal{R}} t$ is called a *rewrite step*. Alternatively, we can write $s \to_{p,\sigma,l\to r} t$ or $s \to_{p,l\to r} t$ when we want to make explicit the position, substitution and variant of the rewrite rule involved in performing the rewrite step. When no confusion can arise, we will omit to specify $\mathcal{R}$ as prefix or subscript.

Notice that upon rewriting in the presence of a TRS axiomatization, the axioms are applied in only one direction. The rewrite rules can be interpreted as partial definitions of the head symbols of their left-hand sides. That is, in the presence of a term rewriting system $\mathcal{R}$ we can regard the set of operators $\mathcal{F}$ as the disjoint union of two sets of symbols: the set $\mathcal{F}_d{:=}\{f \in \mathcal{F} \mid f = \mathrm{root}(l) \text{ for some rule } l \to r \in \mathcal{R}\}$ of *defined symbols*, and the set $\mathcal{F}_c{:=}\mathcal{F}\setminus\mathcal{F}_d$ of *constructors*. Constructors are injective, i.e. if $c \in \mathcal{F}_c$ then $c(\overline{s_n}) = c(\overline{t_n})$ iff $s_i = t_i$ for all $1 \le i \le n$, and different constructors build different terms, i.e. if $c, d \in \mathcal{F}_c$ with $c \neq d$ then $c(\overline{s_m}) \neq d(\overline{t_n})$.

A term s is an *$\mathcal{R}$-normal form*, or *$\mathcal{R}$-normalized*, if there is no term t with $s \to_{\mathcal{R}} t$. A term *s has an $\mathcal{R}$-normal form* if there exists an $\mathcal{R}$-normal form t such that $s \to^*_{\mathcal{R}} t$. If s has a unique $\mathcal{R}$-normal form then we denote this unique normal form by $s{\downarrow_{\mathcal{R}}}$. A term t is *$\mathcal{R}$-normalized* if there is no rewrite step $t \to_{\mathcal{R}} t'$. A substitution θ is *$\mathcal{R}$-normalized* if $X\theta$ is $\mathcal{R}$-normalized for all $X \in \mathcal{D}(\theta)$.

Two terms s and t are *$\mathcal{R}$-joinable*, notation $s \downarrow_{\mathcal{R}} t$, if there exists a term u such that $s \to_{\mathcal{R}}^* u$ and $t \to_{\mathcal{R}}^* u$.

Term rewriting systems are convenient axiomatizations of equational theories. The most important properties of term rewriting systems which are useful in defining effective proof subcalculi for the entailment system of equational logic are confluence and termination. A TRS $\mathcal{R}$ is *terminating* if there are no infinite reduction derivations $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} t_3 \to_{\mathcal{R}} \cdots$ $\mathcal{R}$ is *confluent* if for all terms s, t_1, t_2 with $s \to_{\mathcal{R}}^* t_1$ and $s \to_{\mathcal{R}}^* t_2$ we have $t_1 \downarrow_{\mathcal{R}} t_2$. Note that termination implies the existence of normal forms, whereas confluence implies uniqueness of normal forms.

Proving entailments of the form $E \vdash_{\Sigma} s \approx t$ is greatly simplified if we know that E is the axiomatization of a theory which can be presented with a confluent and terminating TRS $\mathcal{R}$: we compute the (unique) normal forms $s{\downarrow_{\mathcal{R}}}$ and $t{\downarrow_{\mathcal{R}}}$ and compare them syntactically. Because of the confluence property, the computation of a normal form is a deterministic process since both the choice of the rewriting position and the choice of of the rewrite rule are don't care nondeterministic. A *rewriting proof* of an entailment $\mathcal{R} \vdash_{\Sigma} s \approx t$ can be represented by a sequence of steps

$$s \approx t \to_{\mathcal{R}} s_1 \approx t_1 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} s_n{\downarrow_{\mathcal{R}}} \approx t_n{\downarrow_{\mathcal{R}}}$$

abbreviated $s \approx t \to_{\mathcal{R}}^* s_n{\downarrow_{\mathcal{R}}} \approx t{\downarrow_{\mathcal{R}}}$, where

$$s \approx t \to_{\mathcal{R}} s' \approx t' :\Leftrightarrow (s = s' \wedge t \to_{\mathcal{R}} t') \vee (s \to_{\mathcal{R}} s' \wedge t = t')$$

Term rewriting can be used as operational semantics of functional programming languages. A functional program is a set $\mathcal{R}$ of confluent and terminating rewrite rules. Typical queries in functional logic programming are: $\mathcal{R} \vdash_{\Sigma} s \approx t$ and $\mathcal{R} \vdash_{\Sigma} \exists X.X \approx t$. Queries of the first type are proven as shown above, whereas queries of the second type are proven by computing $u = t{\downarrow_{\mathcal{R}}}$ and binding X to u. In such languages, reduction of a term to normal form is called *evaluation*.

Due to its simplicity, term rewriting became a research field of its own, where essential properties such as termination and confluence are being deeply investigated [Klo90, Klo92]. One important application is the following: for a given a theory $T = \langle \Sigma, E \rangle$ find a presentation $\langle \Sigma, \mathcal{R} \rangle$ with $\mathcal{R}$ a confluent and terminating TRS. If such a presentation exists, the entailment relation $E \vdash_{\Sigma} s \approx t$ is equivalent with $\mathcal{R} \vdash_{\Sigma} s \approx t$, which can be realized with the proof subcalculus that has been outlined here.

Term rewriting is also at the core of well developed applications of theorem proving, program synthesis via completion and algebraic specifications.

Among the other properties of TRSs which are relevant for equational deduction, we recall left-linearity, right-linearity, orthogonality and constructorness. A rewrite rule $l \to r$ is *left-linear* (*right linear*) if l (r) does not contain multiple occurrences of the same variable. A TRS $\mathcal{R}$ is *left-linear* (*right-linear*) if it consists only of left-linear (right-linear) rewrite rules. In order to define orthogonal TRSs we first introduce the notion of *critical pair.*

Definition 17 (critical pair) *Let $\mathcal{R}$ be a TRS and $l_1 \to r_1$ and $l_2 \to r_2$ be variants without common variables such that there exists $p \in \mathcal{P}os_{\mathcal{F}}(l_1)$ and a most general unifier θ of $(l_1)|_p$ and l_2. The pair $(l_1[r_2]_p\sigma, r_1\sigma)$ of reducts of $l_1\sigma$ is a* critical pair *of $\mathcal{R}$.*

Definition 18 (orthogonal TRS) *A TRS $\mathcal{R}$ is* orthogonal *if it is left-linear and has no critical pairs.*

Definition 19 (constructor TRS) *A TRS $\mathcal{R}$ is* constructor *if it consists of rewrite rules of the form $f(\overline{l_n}) \to t$ with $\overline{l_n} \in \mathcal{T}(\mathcal{F}_c, \mathcal{V})$.*

Narrowing

We already described how the restriction of axioms to confluent and terminating TRSs greatly simplifies the design of an effective proof subcalculus for entailments of the form $\mathcal{R} \vdash_\Sigma s \approx t$. In this subsection we describe how this restriction can make the realization of deductions of the form $\mathcal{R} \vdash_\Sigma \exists \bigwedge_{i=1}^n (s_i \approx t_i)$ much more efficient than with paramodulation.

A syntactic object G' is a *fresh variant* of a syntactic object G if:

(i) $\mathcal{V}(G')$ do not occur in any syntactic object constructed so far, and

(ii) there exists a renaming θ such that $G' = G\theta$.

For instance, the rewrite rule $R' = Z + succ(W) \to succ(Z+W)$ is a fresh variant of the rule $R = X + succ(Y) \to succ(X+Y)$ because $\mathcal{V}(R) \cap \mathcal{V}(R') = \{X, Y\} \cap \{Z, W\} = \emptyset$ and $R' = R\theta$ where $\theta = \{X \mapsto Z, Y \mapsto W\}$ is a renaming substitution.

Narrowing, proposed originally as a theorem proving tool [Fay79, Hul80], is an effective proof calculus for entailments of the form

$$\mathcal{R} \vdash_\Sigma \exists \bigwedge_{i=1}^{n} (s_i \approx t_i)$$

where $\mathcal{R}$ is a confluent term rewriting system. It can be viewed as a specialization of paramodulation which is a natural combination of term rewriting and unification. The narrowing relation defined below was introduced by Hullot [Hul80].

Definition 20 (narrowing) *We say that a term s is* narrowable *into a term t if there exist a position $p \in \mathcal{P}os_{\mathcal{F}}(s)$, a fresh variant $l \to r$ of a rewrite rule with $\mathcal{V}(l) \cap \mathcal{V}(s) = \emptyset$, and a substitution θ such that*

- *θ is a most general unifier of $s|_p$ and l, and*
- *$t = (s[r]_p)\theta$.*

We write $s \rightsquigarrow_{p,l \to r,\theta} t$ or simply $s \rightsquigarrow_\theta t$. The relation $\rightsquigarrow$ is called narrowing.

When confusions may arise, we underline the subterm or side of the equation to which narrowing is applied. We write $s \rightsquigarrow^*_\theta t$ if there exists a narrowing derivation

$$s = t_1 \rightsquigarrow_{\theta_1} \cdots \rightsquigarrow_{\theta_n} t_n = t$$

with $\theta = \theta_1 \ldots \theta_n$. If $n = 0$ then $\theta = \varepsilon$.

The narrowing relation can be used to define an effective proof subcalculus which realizes the entailment relation $\mathcal{R} \vdash_\Sigma \exists G$ where G is a sentence of the form $\bigwedge_{i=1}^n (s_i \approx t_i)$. The calculus is called NC and it consists of two inference rules: [u] and [n], where the syntactic unification rule [u] is the same as for paramodulation, and the narrowing rule [n] is defined by

$$[\mathrm{n}]\ \frac{\mathcal{R} \vdash_\Sigma \exists(G \wedge (s \simeq t) \wedge G')}{\mathcal{R} \vdash \exists(G\theta \wedge (s' \simeq t\theta) \wedge G'\theta)} \text{ if } s \rightsquigarrow_\theta s'.$$

Here $s \simeq t$ stands for $s \approx t$ or $t \approx s$. A narrowing step (NC-step for short) is an expression of the form $G \overset{\mathrm{NC}}{\Longrightarrow}_\theta G'$ where $\dfrac{G}{G'}$ is an instance of an inference rule of NC with computed substitution θ.

An NC-*derivation* is a sequence of NC-steps

$$G_0 \overset{\mathrm{NC}}{\Longrightarrow}_{\theta_1} G_1 \overset{\mathrm{NC}}{\Longrightarrow}_{\theta_2} \ldots \overset{\mathrm{NC}}{\Longrightarrow}_{\theta_n} G_n$$

abbreviated $G \overset{\mathrm{NC}}{\Longrightarrow}{}^*_\theta G_n$ where $\theta = \theta_1 \ldots \theta_n$. An NC-*refutation* is an NC-derivation of the form $G \overset{\mathrm{NC}}{\Longrightarrow}{}^*_\theta \Box$. The most important properties of the calculus NC are:

Soundness If $G \overset{\mathrm{NC}}{\Longrightarrow}{}^*_\theta \Box$ then $\theta \in \mathcal{U}_\mathcal{R}(G)$.

Completeness If $\gamma \in \mathcal{U}^n_\mathcal{R}(G)$ then there exists an NC-refutation $G \overset{\mathrm{NC}}{\Longrightarrow}{}^*_\theta \Box$ such that $\theta \leq^{\mathcal{V}(G)} \gamma$.

Similar to a paramodulation proof, a narrowing proof of $\mathcal{R} \vdash_\Sigma \exists G$ is an NC-refutation $G \overset{\mathrm{NC}}{\Longrightarrow}{}^*_\theta \Box$. We denote by $Ans^{\mathrm{NC}}_\mathcal{R}(G)$ the set of substitutions computed upon NC-refutations, i.e. $Ans^{\mathrm{NC}}_\mathcal{R}(G) := \{\theta \mid G \overset{\mathrm{NC}}{\Longrightarrow}{}^*_\theta \Box\}$.

Note that the completeness property of narrowing differs from the completeness property of paramodulation. Whereas paramodulation guarantees that *any* $\mathcal{R}$*-unifier* of a goal is subsumed by a computed substitution, narrowing can only guarantee that *any* $\mathcal{R}$*-normalized* $\mathcal{R}$*-unifier* is subsumed by a computed substitution.

Chapter 3

Functional Logic Programming

Functional logic programming [Han97] is a declarative programing style which offers features from functional programming and logic programming. Each paradigm complements the other; e.g., logic programming contributes partial data structures whereas functional programming contributes infinite data structures. Functional evaluation is more efficient and offers better control. Logic evaluation supports existential variables and inversion. Early research in this area has been concentrated on the definition and improvement of appropriate execution principles for functional logic languages. In recent years efficient implementations of these execution principles have been developed.

The fundamental problem of functional logic programming is how to execute a program which may lead to the evaluation of a functional expression containing uninstantiated logical variables. There are currently two operational principles to deal with this problem:

Narrowing. Narrowing solves this problem by making guesses on the values of these variables for a seamless integration of logic and functional computations.

Residuation which delays the evaluation of expressions containing uninstantiated variables.

The rest of this chapter is structured as follows. In Sect. 3.1 we give a description of the requirements which must be satisfied by a functional logic programming language. Section 3.2 describes the lazy narrowing calculus as operational semantics of functional logic programming. Section 3.3 gives a short account to two extensions of first-order lazy narrowing which are of interest in improving the expressive power of functional logic programming: lazy narrowing with conditional term rewriting systems, and higher-order lazy narrowing.

3.1 Preliminaries

A program in a functional logic programming language is a theory $P = \langle \Sigma, \mathcal{R} \rangle$ axiomatized with a confluent term rewriting system $\mathcal{R}$. When specifying a program P, the user has a specific model M_P in mind, which is called in the literature *standard model* or *intended model* of the theory. After specifying the program, the user can ask questions about his/her program. These questions are called *queries*, and belong to a specified class $quer(\Sigma)$ of sentences in the language of logic of P. In the case of functional logic programming, the intended model is $M_P = \mathcal{T}(\mathcal{F})/_{\sim^{\mathcal{R}}}$ where $\sim^{\mathcal{R}}$ is the congruence relation induced by $\mathcal{R}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and the queries are existential closures of the form $\exists G$ where G is a conjunction of equations $\bigwedge_{i=1}^{n}(s_i \approx t_i)$. Since M_P is an initial algebra in $Alg(\Sigma, \mathcal{R})$ we have that $M_P \models_\Sigma \exists G$ if and only if $\mathcal{R} \models_\Sigma^* \exists G$.

When the user submits a query $\exists G$, if it is the case that $M_P \vdash_\Sigma \exists G$, the system will return a set $Ans_{\mathcal{R}}(G)$ of substitutions for the existentially quantified variables of the query as its associated *answers*.

The most satisfactory way of exploiting provability as a method of settling facts about the intended model of a functional logic program is to guarantee the following property:

Query completeness For any program $P = \langle \Sigma, \mathcal{R} \rangle$ and query $\exists G \in quer(\Sigma)$:

$$M_P \models_\Sigma \exists G \Leftrightarrow \forall \gamma \in \mathcal{U}_{\mathcal{R}}(G)\ \exists \theta \in Ans_{\mathcal{R}}(G)\ .\ \theta \leq \gamma.$$

The restriction to $\mathcal{R}$-normalized solutions is reasonable in functional logic programming, where we are usually interested in computing *value* bindings for the existentially quantified variables of the goal. From the functional point of view, a value is an $\mathcal{R}$-irreducible term. Therefore, the query completeness requirement is often weakened by replacing $\mathcal{U}_{\mathcal{R}}(G)$ with $\mathcal{U}_{\mathcal{R}}^n(G)$, where $\mathcal{U}_{\mathcal{R}}^n(G)$ is the set of $\mathcal{R}$-normalized solutions of G. Henceforth, whenever we will refer to completeness we will understand the weakened requirement for query completeness.

We have already mentioned narrowing as an effective proof subcalculus which generates proofs for entailments of the form $\mathcal{R} \vdash_\Sigma \exists \bigwedge_{i=1}^{n}(s_i \approx t_i)$. Any NC-refutation $G \stackrel{\text{NC}}{\Longrightarrow}{}_\theta^* \Box$ yields as answer the substitution $\theta\restriction_{\mathcal{V}(G)}$, thus

$Ans^{\rm NC}_{\mathcal{R}}(G) = \{\theta\lceil_{\mathcal{V}(G)} \mid \langle G, \varepsilon\rangle \stackrel{\rm NC}{\Longrightarrow}{}^{*} \langle \Box, \theta\rangle\}$. The soundness and completeness properties of narrowing ensure that

- $Ans^{\rm NC}_{\mathcal{R}}(G) \subseteq \mathcal{U}_{\mathcal{R}}(G)$, and
- for any $\gamma \in \mathcal{U}^{n}_{\mathcal{R}}(G)$ there exists a substitution $\theta \in Ans^{\rm NC}_{\mathcal{R}}(G)$ such that $\theta \leq^{\mathcal{V}(G)} \gamma$. (i.e., query completeness holds.)

Most of the functional logic programming languages available nowadays have an operational semantics based on narrowing or lazy narrowing. Narrowing makes use of mgu computations to decide whether an instance of a subterm can be rewritten. Lazy narrowing overcomes the mgu computation by integrating it in its deduction rules.

3.2 Lazy Narrowing

The main disadvantage of NC is the costly operation of term unification. Lazy narrowing calculi avoid the unification operation by decomposing the inference rule of NC into a smaller number of more primitive inference rules [Höl89, Sny91]. The main disadvantage of these calculi is that the search space of proofs for the entailment relation becomes larger because of the nondeterminism between the inference rules which can be applied to the selected equation.

Preliminaries

It is customary to describe the inference rules of a lazy narrowing calculus $\mathcal{C}$ by expressions of the form $\dfrac{G, e, G'}{G''\theta}$ where G, G', G'' are sequences of equations, e is an equation, and θ is a substitution. The corresponding sequent style representation is

$$\frac{\mathcal{R} \vdash_{\Sigma} \exists(\bigwedge_{e' \in G} e' \wedge e \wedge \bigwedge_{e' \in G'} e')}{\mathcal{R} \vdash_{\Sigma} \exists([\theta] \wedge \bigwedge_{e' \in G''} e'\theta)}$$

where $[\theta] := \bigwedge_{X \in \mathcal{D}(\theta)}(X \approx X\theta)$. θ is called the substitution *computed* upon the application of the inference rule.

A $\mathcal{C}$*-step* is an expression of the form $G \stackrel{\mathcal{C}}{\Longrightarrow}_{\theta} G'$ where $\dfrac{G}{G'}$ is an inference rule of $\mathcal{C}$ with computed substitution θ. Additional subscripts may be provided, depending on the calculus under consideration.

A $\mathcal{C}$*-derivation* is a sequence of $\mathcal{C}$-steps $G_0 \stackrel{\mathcal{C}}{\Longrightarrow}_{\theta_1} G_1 \stackrel{\mathcal{C}}{\Longrightarrow}_{\theta_2} \ldots \stackrel{\mathcal{C}}{\Longrightarrow}_{\theta_n} G_n$, abbreviated $G_0 \stackrel{\mathcal{C}}{\Longrightarrow}{}^{*}_{\theta} G_n$, where $\theta = \theta_1 \ldots \theta_n$. We denote by $|\Pi|$ the length of a $\mathcal{C}$-derivation Π, i.e. the number of $\mathcal{C}$-steps of Π.

According to our convention of notation, a $\mathcal{C}$-step $\overline{e_m} \overset{\mathcal{C}}{\Longrightarrow}_\theta \overline{e'_n}$ corresponds to a deduction step $\mathcal{R} \vdash_\Sigma \exists(\bigwedge_{i=1}^m e_i) \overset{\mathcal{C}_s}{\Longrightarrow} \exists(\bigwedge_{j=1}^n e'_j)$.

We write $s \simeq t$ for $s \approx t$ or $t \approx s$. It is assumed that the inference rules of the calculi described in this thesis preserve the orientation of $\simeq$.

The Calculus LNC

Our presentation of the lazy narrowing calculus LNC follows [MOI96]. The calculus LNC consists of the inference rules shown in Figure 3.1. The

---

[o] *outermost narrowing*

$$\frac{G, f(\overline{s_n}) \simeq t, G'}{G, \overline{s_n \approx l_n}, r \simeq t, G'}$$

if there exists a fresh variant $f(\overline{l_n}) \to r$ of a rewrite rule in $\mathcal{R}$,

[i] *imitation*

$$\frac{G, f(\overline{s_n}) \simeq X, G'}{(G, \overline{s_n \approx X_n}, G')\theta}$$

if $\theta = \{X \mapsto f(\overline{X_n})\}$ with $X_1, \ldots, X_n$ fresh variables,

[d] *decomposition*

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

[v] *variable elimination*

$$\frac{G, s \simeq X, G'}{(G, G')\theta}$$

if $X \notin \mathcal{V}(s)$ and $\theta = \{X \mapsto s\}$,

[t] *removal of trivial equations*

$$\frac{G, X \approx X, G'}{G, G'}$$

Fig. 3.1: The inference rules of LNC.

equations $\overline{s_n \approx l_n}$ created by the outermost narrowing inference rule are called *parameter-passing* equations.

Note that none of the inference rules of LNC requires mgu computations.

An LNC-*refutation* is an LNC-derivation of the form $G_0 \overset{\text{LNC}}{\Longrightarrow}{}^*_\theta \Box$. Additional subscripts may be specified, such as the label of the selected inference rule or the rewrite rule variant used in the application of the [o]-rule.

LNC is a suitable model of computation for functional logic programming because of the following properties:

Soundness If $G \stackrel{\text{LNC}}{\Longrightarrow}{}^*_\theta \square$ then $\theta \in \mathcal{U}_\mathcal{R}(G)$.

Completeness If $\gamma \in \mathcal{U}^n_\mathcal{R}(G)$ then there exists an LNC-refutation $G \stackrel{\text{LNC}}{\Longrightarrow}{}^*_\theta \square$ such that $\theta \leq^{\mathcal{V}(G)} \gamma$.

Soundness implies that $Ans^{\text{LNC}}_\mathcal{R}(G) \subseteq \mathcal{U}_\mathcal{R}(G)$, and completeness implies that $Ans^{\text{LNC}}_\mathcal{R}(G)$ is a complete set of normalized $\mathcal{R}$-unifiers.

There are three sources of nondeterminism in the usage of LNC:

1. selection of equation in the current goal,
2. selection of inference rule to be applied to the selected equation, and
3. selection of rewrite rule to be used upon applying the [o]-inference rule.

The second source of nondeterminism does not appear in NC because NC has only one inference rule. It is also well known that NC is *strongly complete*, in the sense that the first source of non-determinism is *don't care*, i.e. we are free to chose any equation without influencing the soundness and completeness results. In contrast, LNC is not strongly complete, as one can see from the example below. The terms underlined in the example are the ones selected in the corresponding inference steps.

Example 1 *Consider the TRS*

$$\mathcal{R} = \{f(X) \to g(h(X), X), g(X, X) \to a, b \to h(b)\}$$

and the goal $G = f(b) \approx a$. *Confluence of* $\mathcal{R}$ *can be proved by induction on the structure of terms and some case analysis. The normalized empty substitution* ε *is a solution of* G *because*

$$\underline{f(b)} \approx a \to_\mathcal{R} g(h(b), \underline{b}) \approx a \to_\mathcal{R} \underline{g(h(b), h(b))} \approx a \to_\mathcal{R} a \approx a$$

By adopting the selection function sel_{right} *which always selects the rightmost equation in a goal, we can only compute infinite* LNC*-derivations*

$$\begin{aligned}
&\underline{f(b) \approx a} \stackrel{\text{LNC}}{\Longrightarrow}_{[\text{o}],\text{f}(\text{X}_1)\to\text{g}(\text{h}(\text{X}_1),\text{X}_1)} b \approx X_1, \underline{g(h(X_1), X_1)} \approx a\\
&\stackrel{\text{LNC}}{\Longrightarrow}_{[\text{o}],\text{g}(\text{X}_2,\text{X}_2)\to\text{a}} b \approx X_1, h(X_1) \approx X_2, \underline{X_1 \approx X_2}, \underline{a \approx a}\\
&\stackrel{\text{LNC}}{\Longrightarrow}_{[\text{d}]} \stackrel{\text{LNC}}{\Longrightarrow}_{[\text{v}],\theta_1} b \approx X_2, \underline{h(X_2) \approx X_2}\\
&\stackrel{\text{LNC}}{\Longrightarrow}_{[\text{i}],\theta_2} b \approx X_2, \underline{h(X_3) \approx X_3}\\
&\stackrel{\text{LNC}}{\Longrightarrow}_{[\text{i}],\theta_3} \cdots
\end{aligned}$$

where $\theta_1 = \{X_1 \mapsto X_2\}, \theta_2 = \{X_2 \mapsto h(X_3)\}, \theta_3 = \{X_3 \mapsto h(X_4)\}, \ldots$ *We see that we can not generate any* LNC*-refutation starting from* G *if we use the selection function* sel_{right}. *Thus* LNC *with selection function* sel_{right} *is not complete. Hence* LNC *is* not *strongly complete.*

Because of the properties mentioned before, NC seems to be a better model of computation than LNC. In [MOI96] it is shown that the first source of nondeterminism can be dropped if we adopt a leftmost equation selection function. This means that in each LNC-step we select the leftmost equation which appears in the current goal.

Example 2 *Consider* $\mathcal{R} = \{0 + X \to X, s(X) + Y \to s(X + Y)\}$ *and the goal* $G = X + Y \approx s(0)$. *It is easy to see that* $\mathcal{R}$ *is confluent and that the only* $\mathcal{R}$*-normalized solutions of* G *are* $\{X \mapsto 0, Y \mapsto s(0)\}$ *and* $\{X \mapsto s(0), Y \mapsto 0\}$. *Both solutions can be computed with* LNC*-refutations:*

$$\begin{aligned} G &\overset{\text{LNC}}{\Longrightarrow}_{[o],0+X_1\to X_1} \underline{X \approx 0}, Y \approx X_1, X_1 \approx s(0) \\ &\overset{\text{LNC}}{\Longrightarrow}_{[v],\{X\mapsto 0\}} \underline{Y \approx X_1}, X_1 \approx s(0) \overset{\text{LNC}}{\Longrightarrow}_{[v],\{Y\to X_1\}} \underline{X_1 \approx s(0)} \\ &\overset{\text{LNC}}{\Longrightarrow}_{[v],\{X_1\mapsto s(0)\}} \square \end{aligned}$$

$$\begin{aligned} G &\overset{\text{LNC}}{\Longrightarrow}_{[o],s(X_1)+Y_1\to s(X_1+Y_1)} \underline{X \approx s(X_1)}, \underline{Y \approx Y_1}, s(X_1 + Y_1) \approx s(0) \\ &\overset{\text{LNC}}{\Longrightarrow}{}^{2}_{[v],\{X\mapsto s(X_1),Y\mapsto Y_1\}} \underline{s(X_1 + Y_1) \approx s(0)} \\ &\overset{\text{LNC}}{\Longrightarrow}_{[d]} \underline{X_1 + Y_1 \approx 0} \overset{\text{LNC}}{\Longrightarrow}_{[o],0+X_2\to X_2} \underline{X_1 \approx 0}, \underline{Y_1 \approx X_2}, \underline{X_2 \approx 0} \\ &\overset{\text{LNC}}{\Longrightarrow}{}^{3}_{[v],\{X_1\mapsto 0,Y_1\mapsto X_2,X_2\mapsto 0\}} \square \end{aligned}$$

The nondeterminism of LNC due to the selection of the inference rule to be applied to the selected equation is depicted in Fig. 3.2. Practical

	root(t)		
root(s)	$\mathcal{V}$	$\mathcal{F}_c$	$\mathcal{F}_d$
$\mathcal{V}$	[v],[t]	[v],[i]	[v],[i],[o]
$\mathcal{F}_c$	[v],[i]	[d]	[o]
$\mathcal{F}_d$	[v],[i],[o]	[o]	[d],[o]

Fig. 3.2: LNC: nondeterminism between the inference rules for selected equation $s \approx t$

refinements of LNC which reduce the nondeterminism between its inference rules are given in [MO98]. These refinements fall in one of the following two categories:

1. they are realized by imposing suitable restrictions on the underlying term rewrite system. Among the restrictions proposed in [MO98] we recall: left-linear, orthogonal, and constructor term rewriting systems,

2. they are realized for particular classes of equations. Successful deterministic refinements of the calculus LNC have been obtained for:

   - equations which descend from parameter-passing equations,
   - equations which descend directly from equations in the initial goal,
   - equations with strict semantics. By adding equations with strict semantics to the family of equational sentences, we actually restrict the class of solutions of a goal: we say θ is a solution of an equation $s \approx t$ with strict semantics iff $s\theta \rightarrow_{\mathcal{R}} u$ and $t\theta \rightarrow^*_{\mathcal{R}} u$ with $u \in \mathcal{T}(\mathcal{F}_c, \mathcal{V})$. Alternatively, we say that θ is a strict solution of $s \approx t$.

     θ is a strict solution of a goal G iff it is strict solution of any equation in G.

It is shown that by restricting the goals to conjunctions of equations with strict semantics, and the functional logic program to a left-linear confluent constructor system, all the nondeterminism due to the selection of the inference rule to be applied next can be eliminated. The calculus resulted in this way is called $\mathrm{LNC_d}$.

3.3 Extensions

Two extensions of the first-order lazy narrowing calculus are of particular interest in the design of a powerful operational principle of functional logic programming:

1. lazy narrowing for conditional TRS,

2. higher-order lazy narrowing.

3.3.1 Lazy Conditional Narrowing

The first extension adds to functional logic programs the expressive power of conditional term rewrite systems. Formally, a conditional term rewriting system (CTRS for short) over a signature $\mathcal{F}$ is a set $\mathcal{R}$ of (conditional) rewrite rules of the form $l \rightarrow r \Leftarrow c$ where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $l \notin \mathcal{V}$, and the conditional part c is a (possibly empty) sequence $\overline{s_n \approx t_n}$ of equations. If $c = \Box$ then we may simply write $l \to r$ instead. CTRSs are classified according to the distribution of variables in rewrite rules as follows [MH94]:

- a 1-CTRS $\mathcal{R}$ contains no extra variables, i.e. $\mathcal{V}(r) \cup \mathcal{V}(c) \subseteq \mathcal{V}(l)$ for every $l \to r \Leftarrow c \in \mathcal{R}$,
- a 2-CTRS $\mathcal{R}$ may contain extra variables in the condition only, i.e. $\mathcal{V}(r) \subseteq \mathcal{V}(l)$ for every $l \to r \Leftarrow c \in \mathcal{R}$,
- a 3-CTRS $\mathcal{R}$ may also have extra variables in the right hand side provided these occur in the corresponding conditions. Formally, $\mathcal{V}(r) \subseteq \mathcal{V}(l) \cup \mathcal{V}(c)$ for every conditional rewrite rule $l \to r \Leftarrow c \in \mathcal{R}$.

The rewrite relation $\to_{\mathcal{R}}$ associated with a conditional term rewriting system $\mathcal{R}$ is obtained by interpreting the equality signs in the conditional part of a rewrite rule as joinability. Formally, $\mathcal{R}$ is the smallest (w.r.t. inclusion) rewrite relation with the property that $l\sigma \to r\sigma$ whenever there exist a variant $l \to r \Leftarrow c$ of a conditional rewrite rule in $\mathcal{R}$ and a substitution σ such that $s'\sigma \downarrow t'\sigma$ for every equation $s' \approx t'$ in c. An inductive definition of $\to_{\mathcal{R}}$ is given below.

Definition 21 (conditional term rewriting) *Let $\mathcal{R}$ be a CTRS. We inductively define the TRSs $\mathcal{R}_n$ as follows:*

$$\mathcal{R}_0 = \{x \approx x \to \mathtt{true}\}$$
$$\mathcal{R}_{n+1} = \{l\sigma \to r\sigma \mid l \to r \Leftarrow c \in \mathcal{R} \text{ and } e\sigma \to^*_{\mathcal{R}_n} \mathtt{true} \text{ for all } e \text{ in } c\}$$

We define $s \to_{\mathcal{R}} t$ if and only if $s \to_{\mathcal{R}_n} t$ for some $n \geq 0$. The minimum such n is called the depth *of the rewrite step $s \to_{\mathcal{R}} t$.*

With this understanding, all the notions defined for TRSs extend to CTRSs.

The extension of lazy narrowing to CTRS has been stated by Hamada *et al.* [HMS99], who propose a new calculus called LCNC. LCNC is obtained from LNC by replacing the outermost narrowing rule [o] with the inference rule

[o] *outermost conditional narrowing*

$$\frac{G, f(\overline{s_n}) \simeq t, G'}{G, \overline{s_n \approx l_n}, r \simeq t, c, G'}$$

if $f(\overline{l_n}) \to r \Leftarrow c$ is a fresh variant of a conditional rewrite rule in $\mathcal{R}$.

Thus the only difference between LCNC and LNC is in the outermost narrowing rule: in LCNC we add the conditional part of the applied conditional rewrite rule to the new goal.

The following definition is useful in stating the completeness results known so far for LCNC.

Definition 22 (level-confluent CTRS) *A CTRS $\mathcal{R}$ is* level-confluent *if every term rewrite system $\mathcal{R}_n$ is confluent.*

It has been shown [HMS99] that LCNC is complete for arbitrary confluent CTRSs without extra variables with respect to normalized solutions and for terminating and level-confluent CTRSs without any restrictions on the distribution of variables in the conditional rewrite rules.

3.3.2 Higher-order Lazy Narrowing

The design of a higher-order lazy narrowing calculus is one of the most challenging problems for the functional logic programming community. While reduction has long been studied and deep theories related to reduction have been amassed in literature, enough theories of narrowing of comparable richness and depth have not been developed yet. This is particularly so in higher-order theories for narrowing.

Recently, proposals of higher-order lazy narrowing calculi started to appear in an attempt to define a suitable operational semantics for a functional logic programming language enhanced with higher-order constructs, such as functional variables and λ-abstractions.

We mention here two directions of research:

1. One direction of research tries to lift the first-order lazy narrowing calculus LNC and its deterministic refinements to higher-order logic by preserving as much as possible the main properties of LNC which makes it attractive for computational purposes. We mention here the calculus HLNC proposed by Suzuki *et al.* [SNI97], which aims at combining LNC with β-reduction. It is shown that the combination yields a sound and complete calculus for a particular class of higher-order term rewriting systems called TRS_λ. Unfortunately, this class of term rewriting systems is too restricted to make HLNC of much practical interest.

2. Another approach is the one followed by Prehofer, which proposes a higher-order lazy narrowing calculus called LN [Pre98] for solving systems of oriented equations. The calculus LN is based on the higher-order rewrite system of Nipkow [NP98]. The main disadvantages of

LN are the high nondeterminism due to the inference rules to be applied to a selected equation, and the selection of the rewrite rule to be used upon performing outermost narrowing of terms with variable at root position. Proposals to reduce these sources of nondeterminism have already been proposed [Pre98], but the results are still too weak to make LN of much practical interest.

Chapter 4

Lazy Narrowing for Applicative Term Rewrite Systems

In this chapter we introduce our first lazy narrowing calculus extended with higher-order constructs. The results presented here are revised versions of earlier results reported in [MMIY99].

4.1 Introduction

A straightforward way to use lazy narrowing for higher-order equational reasoning is via applicative term rewriting systems. The following example illustrates the expressiveness of such a term rewriting system.

Example 3 *Consider the term rewriting system $\mathcal{R}$:*

$$\begin{array}{ll}
\quad\texttt{plus}(0, Y) & \rightarrow Y \\
\quad\texttt{plus}(\texttt{S}(X), Y) & \rightarrow \texttt{S}(\texttt{plus}(X, Y)) \\
\quad\texttt{double}(X) & \rightarrow \texttt{plus}(X, X) \\
\texttt{map}(F, [\,]) & \rightarrow [\,] \\
\texttt{map}(F, [X \mid Y]) & \rightarrow [F(X) \mid \texttt{map}(F, Y)] \\
\texttt{compose}(F, G, X) & \rightarrow F(G(X))
\end{array}$$

Here $[\,]$ *is syntactic sugar for the empty list* `nil`*, and* $[\ h \mid t\ ]$ *stands for the list* $\texttt{cons}(h, t)$ *with head* h *and tail* t*.*

The functions `map` *and* `compose` *are higher-order. For instance, solving the goal*

$$G = \mathtt{map}(F, [\mathtt{S}(0), 0, \mathtt{S}(0)] \approx [\mathtt{S}(\mathtt{S}(\mathtt{S}(0))), \mathtt{S}(0), \mathtt{S}(\mathtt{S}(\mathtt{S}(0)))]$$

means finding substitutions like $\{F \mapsto \lambda x.\mathtt{compose}(\mathtt{S}, \mathtt{double}, x)\}$ *which can not be done with LNC.*

We can make use of LNC if we encode G *and* $\mathcal{R}$ *with applicative terms. Such a transformation yields the applicative term rewriting system* $\mathcal{R}'$*:*

$$\begin{array}{ll}
\mathtt{ap}(\mathtt{ap}(\mathtt{plus}, 0), Y) & \to Y \\
\mathtt{ap}(\mathtt{ap}(\mathtt{plus}, \mathtt{S}(X)), Y) & \to \mathtt{ap}(\mathtt{S}, \mathtt{ap}(\mathtt{ap}(\mathtt{plus}, X), Y)) \\
\mathtt{ap}(\mathtt{double}, X) & \to \mathtt{ap}(\mathtt{ap}(\mathtt{plus}, X), X)
\end{array}$$

$$\begin{array}{ll}
\mathtt{ap}(\mathtt{ap}(\mathtt{map}, F), \mathtt{nil}) & \to \mathtt{nil} \\
\mathtt{ap}(\mathtt{ap}(\mathtt{map}, F), \mathtt{ap}(\mathtt{ap}(\mathtt{cons}, X), Y)) & \to \mathtt{ap}(\mathtt{ap}(\mathtt{cons}, \mathtt{ap}(F, X)), \\
& \quad \mathtt{ap}(\mathtt{ap}(\mathtt{map}, F), Y)) \\
\mathtt{ap}(\mathtt{ap}(\mathtt{ap}(\mathtt{compose}, F), G), X) & \to \mathtt{ap}(\mathtt{ap}(F, G), X).
\end{array}$$

and a corresponding applicative goal G'*. The translation encodes each term* $t = f(t_1, t_2, \ldots, t_n)$ *by* $t^a := \mathtt{ap}(\cdots \mathtt{ap}(f, t_1^a), \cdots), t_n^a)$*, where* t_i^a *are the applicative encodings of* t_i*. The special symbol* `ap` *is the only function symbol used, all the other function symbols are interpreted as mere constants.*

The LNC calculus can now be used to solve the translated goal.

Obviously, LNC is very inefficient for applicative term rewrite systems: while just one inference step extracts all arguments of a first-order term, for applicative term rewriting systems we need one inference step for each argument. In other words, we have to extract all arguments step by step before we even know whether, e.g., we are using the appropriate rewrite rule, even in the case that the head-symbol is a known function.

Starting from this observation, we define a new calculus which we call LNCA (**LNC** for **A**pplicative term rewriting systems). The main advantage of LNCA over LNC is the specialization of the [o]-rule: if the selected term in an [o]-step has a function symbol f at leftmost-innermost position then we consider only [o]-steps with respect to rewrite rules which that define f, i.e. have f at the leftmost-innermost position of the lhs. In this way the nondeterministic application of an [o]-step is drastically reduced.

This chapter is organized as follows. In Section 4.2 we introduce the basic notions and notations for applicative term rewriting systems. Section 4.3 introduces our calculus LNCA for applicative term rewriting systems and contains a detailed proof of its soundness. The completeness of LNCA is given in Section 4.4. The proof is based on a thorough analysis of the structure and properties of a class of complete LNC-refutations (Subsection 4.4.2) and on the design of a lifting process of such a refutation into an LNCA-refutation. Finally, in Section 4.5 we draw some conclusions about the usefulness of LNCA and how to reduce further its high nondeterminism.

4.2 Preliminaries

We assume given a signature $\mathcal{F} \cup \{\mathtt{ap}\}$ of function symbols and a countable set $\mathcal{V}$ of variables. We distinguish the symbol `ap` with $\mathrm{ar}(\mathtt{ap}) = 2$. An *applicative term* is a term built from variables, the binary function symbol `ap` and the symbols of $\mathcal{F}$ interpreted as constants. Thus, the applicative representation of a term $f(\overline{t_n}) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is $\mathtt{ap}(\ldots \mathtt{ap}(f, t_1) \ldots, t_n)$. Given two applicative terms s and t, the juxtaposition $s\ t$ denotes the applicative term $\mathtt{ap}(s, t)$. Parentheses are omitted under the convention of association to the left, so $(\mathtt{plus}\ (\mathtt{S}\ 0))\ 0$ and $\mathtt{plus}\ (\mathtt{S}\ 0)\ 0$ denote the same applicative term. The *head symbol* of an applicative term is the symbol that occurs at the leftmost innermost position. The notions of subterm, position, substitution and replacement defined for first-order terms are carried over to applicative terms.

In the sequel we denote head symbols which are either in $\mathcal{F}$ or $\mathcal{V}$ by the possibly subscripted letters a, b, c, variables by x, y, z, symbols from $\mathcal{F}$ by f, g, h, arbitrary applicative terms by l, r, s, t, u, v and integral numbers by i, j, k.

Definition 23 (applicative TRS) *An* applicative rewrite rule *is a pair $l \to r$ between two applicative terms such that*

- *l is of the form $f\ l_1\ \ldots\ l_n$,*
- *$f \in \mathcal{F}$ with* $\mathrm{ar}(f) = n$,
- *$\mathcal{V}(l) \supseteq \mathcal{V}(r)$.*

An applicative term rewriting system *(ATRS for short) is a set of applicative rewrite rules.*

We abbreviate an applicative term $a\ t_1\ \ldots\ t_n$ to $a\ \overline{t_n}$, and to a if $n = 0$. By the same convention, $b\ \overline{s_m}\ \overline{t_n}$ stands for $b\ s_1\ \ldots\ s_m\ t_1\ \ldots\ t_n$ and $c\ \overline{t_{i,j}}$ for $c\ t_i\ \ldots\ t_j$.

All the other notions and definitions defined in Sect. 2.4 for first-order equational logic are lifted to the applicative case by replacing the notion of term with that of applicative term.

4.3 Inference Rules

One reason why LNC does not handle applicative terms efficiently is because the rewrite rule used in an [o]-step is determined by the outermost symbol of an applicative term. This symbol is almost always the binary symbol `ap` which does not impose any restriction on the choice of the rewrite rule to be considered. We overcome this problem by specializing the inference rules of LNC to look at the head symbol rather than at the outermost symbol of the term under consideration and, if that symbol is a function symbol, to choose only the rewrite rules which define it.

LNCA consists of the inference rules [of], [ov], [if], [iv], [vf], [vv], [df], [dv] shown below.

[of] *outermost narrowing for head-function terms*

$$\frac{f\ \overline{s_m}\ \overline{t_n} \simeq t, G}{\overline{s_m \approx u_m}, r\ \overline{t_n} \approx t, G}$$

if $f\ \overline{u_m} \to r$ is a fresh variant of a rewrite rule in $\mathcal{R}$.

[ov] *outermost narrowing for head-variable terms*

$$\frac{x\ \overline{s_m}\ \overline{t_n} \simeq t, G}{(\overline{s_m \approx v_m}, r\ \overline{t_n} \approx t, G)\theta}$$

if there exists a fresh variant $f\ \overline{u_k}\ \overline{v_m} \to r$ of a rewrite rule in $\mathcal{R}$, $m > 0$, and $\theta = \{x \mapsto f\ \overline{u_k}\}$.

[if] *imitation for head-function terms*

$$\frac{f\ \overline{s_m}\ \overline{t_n} \approx x\ \overline{u_n}, G}{(\overline{s_m \approx x_m}, \overline{t_n \approx u_n}, G)\theta} \qquad \frac{x\ \overline{u_n} \approx f\ \overline{s_m}\ \overline{t_n}, G}{(\overline{s_m \approx x_m}, \overline{u_n \approx t_n}, G)\theta}$$

if $m > 0$, $\theta = \{x \mapsto f\ \overline{x_m}\}$ with $x_1, \ldots, x_m$ fresh variables.

[iv] *imitation for head-variable terms*

$$\frac{y\ \overline{s_m}\ \overline{t_n} \approx x\ \overline{u_n}, G}{(\overline{s_m \approx x_m}, \overline{t_n \approx u_n}, G)\theta} \qquad \frac{x\ \overline{u_n} \approx y\ \overline{s_m}\ \overline{t_n}, G}{(\overline{s_m \approx x_m}, \overline{u_n \approx t_n}, G)\theta}$$

if $m > 0$, $x \neq y$ and $\theta = \{x \mapsto y\ \overline{x_m}\}$ with $x_1, \ldots, x_m$ fresh variables.

[df] *decomposition for head-function terms*

$$\frac{f\ \overline{s_n} \approx f\ \overline{t_n}, G}{\overline{s_n \approx t_n}, G}$$

[dv] *decomposition for head-variable terms*

$$\frac{x\ \overline{s_n} \approx x\ \overline{t_n}, G}{\overline{s_n \approx t_n}, G}$$

[vf] *variable-elimination for head-function terms*

$$\frac{f\ \overline{s_m}\ \overline{t_n} \approx x\ \overline{u_n}, G}{(\overline{t_n \approx u_n}, G)\theta} \quad \frac{x\ \overline{u_n} \approx f\ \overline{s_m}\ \overline{t_n},, G}{(\overline{u_n \approx t_n}, G)\theta}$$

if $x \notin \mathcal{V}(f\ \overline{s_m})$ and $\theta = \{x \mapsto f\ \overline{s_m}\}$.

[vv] *variable-elimination for head-variable terms*

$$\frac{y\ \overline{s_m}\ \overline{t_n} \approx x\ \overline{u_n}, G}{(\overline{t_n \approx u_n}, G)\theta} \quad \frac{x\ \overline{u_n} \approx y\ \overline{s_m}\ \overline{t_n}, G}{(\overline{u_n \approx t_n}, G)\theta}$$

if $x \notin \mathcal{V}(y\ \overline{s_m})$ and $\theta = \{x \mapsto y\ \overline{s_m}\}$.

We write $G_1 \Rrightarrow_{[\alpha],\theta} G_2$ to denote an LNCA-step corresponding to an inference rule α with $\alpha \in \{[\text{of}], [\text{ov}], [\text{if}], [\text{iv}], [\text{df}], [\text{dv}], [\text{vf}], [\text{vv}]\}$, upper goal G_1, lower goal G_2, and computed substitution θ. We assume $\theta = \varepsilon$ when $\alpha \in \{[\text{of}], [\text{df}], [\text{dv}]\}$. A goal G is a solved form (in LNCA) if $G = \square$.

We denote by $\mathcal{LNCA}$ the class of LNCA-refutations. A [V]-step is either a [vf]-step or a [vv]-step. An [I]-step is either an [if]-step or an [iv]-step. An [o]-step is either an [of] or an [ov]-step.

The soundness of LNCA is stated in the following theorem.

Theorem 1 (soundness) *Let $\mathcal{R}$ be a confluent ATRS and G a goal. If there exists an LNCA-refutation $A : G \Rrightarrow^*_\theta \square$ then θ is a solution of G.*

Proof. The proof is performed in two steps. We first prove that for every LNCA-step of the form $G_1 \Rrightarrow_\theta G_2$ then the following property holds: if θ' is a solution of G_2 then $\theta\theta'$ is a solution of G_1. Next, we prove by induction on the length of the LNCA-refutation that the LNCA calculus is sound.

Let $G_1 \Rrightarrow_\theta G_2$ be an arbitrary LNCA-step and θ' a solution of G_2. We prove that $\theta\theta'$ is a solution of G_1. The proof is by case distinction on the nature of the step from G_1 to G_2. Let $G_1 \Rrightarrow_\theta G_2$ and θ' be a solution of G_2.

- Assume

$$G_1 = f\ \overline{s_m}\ \overline{t_n} \simeq t, G \Rrightarrow_{[\text{of}],\varepsilon} G_2 = \overline{s_m \approx u_m}, r\ \overline{t_n} \approx t, G$$

where $f\ \overline{u_m} \to r$ is a fresh variant of some rule in $\mathcal{R}$. Then $\theta = \varepsilon$ and $\theta\theta' = \theta'$. We have to prove that θ' is a solution of G_1. Since θ' is a solution of G_2, the following conditions hold: (1) $\forall i \in \{1, \ldots, m\}.s_i\theta' \leftrightarrow^*_\mathcal{R} u_i\theta'$, (2) $(r\ \overline{t_n})\theta' \leftrightarrow^*_\mathcal{R} t\theta'$, and (3) θ' is a solution of G. Because of (3), we only have to prove that θ' is a solution of the equation $f\ \overline{s_m}\ \overline{t_n} \simeq t$. We note that:

$$(f\ \overline{s_m}\ \overline{t_n})\theta' \overset{(1)}{\longleftrightarrow}{}^*_{\mathcal{R}} (f\ \overline{u_m}\ \overline{t_n})\theta' \to_{\{f\ \overline{u_n} \to r\}} (r\ \overline{t_n})\theta' \overset{(2)}{\longleftrightarrow}{}^*_{\mathcal{R}} t\theta'$$

and hence θ' is a solution of the equation $f\ \overline{s_m}\ \overline{t_n} \simeq t$.

- Assume $G_1 = x\ \overline{s_m}\ \overline{t_n} \simeq t, G \Rrightarrow_{[\mathrm{ov}],\theta} G_2 = (\overline{s_m \approx v_m}, r\ \overline{t_n} \approx t, G)\theta$ where $f\ \overline{u_k}\ \overline{v_m} \to r$ is a fresh variant of a rewrite rule in $\mathcal{R}$, $m > 0$ and $\theta = \{x \mapsto f\ \overline{u_k}\}$. Then (1) $\forall i \in \{1, \ldots, m\}.s_i\theta\theta' \leftrightarrow^*_{\mathcal{R}} v_i\theta\theta'$, (2) $(r\ t_n)\theta\theta' \leftrightarrow^*_{\mathcal{R}} t\theta\theta'$, (3) $\theta\theta'$ is a solution of G. Because of (3), we only have to prove that $\theta\theta'$ is a solution of the equation $x\ \overline{s_m}\ \overline{t_n} \simeq t$. We note that:

$$(x\ \overline{s_m}\ \overline{t_n})\theta\theta' = (f\ \overline{u_k}\ \overline{s_m}\ \overline{t_n})\theta\theta' \overset{\text{by }(1)}{\longleftrightarrow}{}^*_{\mathcal{R}} (f\ \overline{u_k}\ \overline{v_m}\ \overline{t_n})\theta\theta' \to_{\{f\ \overline{u_n}\ \overline{v_m} \to r\}}$$
$$(r\ \overline{t_n})\theta\theta' \overset{(2)}{\longleftrightarrow}{}^*_{\mathcal{R}} t\theta\theta'$$

and hence $\theta\theta'$ is a solution of the equation $x\ \overline{s_m}\ \overline{t_n} \simeq t$.

- Assume $G_1 = a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}, G \Rrightarrow_{\alpha,\theta} G_2 = (\overline{s_m \approx x_m}, \overline{t_n \simeq u_n}, G)\theta$, where $a \in \mathcal{V} \cup \mathcal{F}$, $\alpha \in \{[\mathrm{if}], [\mathrm{iv}]\}$, $\theta = \{x \mapsto a\ \overline{x_m}\}$, with $x_1, \ldots, x_m$ fresh variables. Then we have: (1) $\forall i \in \{1, \ldots, m\}.s_i\theta\theta' \leftrightarrow^*_{\mathcal{R}} x_i\theta\theta'$, (2) $\forall j \in \{1, \ldots, n\}.t_j\theta\theta' \leftrightarrow^*_{\mathcal{R}} u_j\theta\theta'$, (3) $\theta\theta'$ is a solution of G. Because of (3), we only have to prove that $\theta\theta'$ is a solution of $a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}$. Since:

$$(x\ \overline{u_n})\theta\theta' = (a\ \overline{x_m}\ \overline{u_n})\theta\theta' \overset{\text{by }(1)}{\longleftrightarrow}{}^*_{\mathcal{R}} (a\ \overline{s_m}\ \overline{u_n})\theta\theta' \overset{\text{by }(2)}{\longleftrightarrow}{}^*_{\mathcal{R}} (a\ \overline{s_m}\ \overline{t_n})\theta\theta'$$

the substitution $\theta\theta'$ is a solution of the equation $a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}$.

- Assume $G_1 = a\ \overline{s_n} \approx a\ \overline{t_n}, G \Rrightarrow_{\alpha,\varepsilon} G_2 = (\overline{s_n \approx t_n}, G$ where $a \in \mathcal{V} \cup \mathcal{F}$ and $\alpha \in \{[\mathrm{df}], [\mathrm{dv}]\}$. In this case $\theta = \varepsilon$, $\theta\theta' = \theta'$, and we have: (1) $\forall i \in \{1, \ldots, n\}.s_i\theta' \leftrightarrow^*_{\mathcal{R}} t_i\theta'$, (2) θ' is a solution of G. Because of (2), we only have to prove that θ' is a solution of $a\ \overline{s_n} \approx a\ \overline{t_n}$, which is obvious because of property (1).

- Assume $G_1 = a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}, G \Rrightarrow_{\alpha,\theta} G_2 = (\overline{t_n \simeq u_n}, G)\theta$ where $a \in \mathcal{V} \cup \mathcal{F}$, $\alpha \in \{[\mathrm{vf}], [\mathrm{vv}]\}$, $x \notin \mathcal{V}(f\ \overline{s_m})$ and $\theta = \{x \mapsto a\ \overline{s_m}\}$. Then: (1) $\forall i \in \{1, \ldots, n\}.t_i\theta\theta' \leftrightarrow^*_{\mathcal{R}} u_i\theta\theta'$, (2) $\theta\theta'$ is a solution of G. Because of (2), we only have to prove that $\theta\theta'$ is a solution of the equation $f\ \overline{s_m}\ \overline{t_n} \approx x\ \overline{u_n}$. We have:

$$(x\ \overline{u_n})\theta\theta' = (a\ \overline{s_m}\ \overline{u_n})\theta\theta' \overset{\text{by }(1)}{\longleftrightarrow}{}^*_{\mathcal{R}} (a\ \overline{s_m}\ \overline{t_n})\theta\theta'$$

We prove now that LNCA is sound. The proof is by induction on the length of the LNCA-refutation. First we prove that all LNCA-refutations of length 1 are sound. The only possible LNCA-refutations of length 1 are the following:

$$\begin{array}{ll} f \approx f \Rrightarrow_{[\mathrm{df}],\varepsilon} \Box, & \\ x \approx x \Rrightarrow_{[\mathrm{dv}],\varepsilon} \Box, & \\ f\ \overline{s_m} \simeq x \Rrightarrow_{[\mathrm{vf}],\theta} \Box, & \text{where } x \notin \mathcal{V}(f\ \overline{s_m}) \text{ and } \theta = \{x \mapsto f\ \overline{s_m}\} \\ y\ \overline{s_m} \simeq x \Rrightarrow_{[\mathrm{vv}],\theta} \Box, & \text{where } x \notin \mathcal{V}(y\ \overline{s_m}) \text{ and } \theta = \{x \mapsto y\ \overline{s_m}\} \end{array}$$

Obviously, all these LNCA-refutations are sound.

Assume now that $A : G_1 \Rrightarrow_\sigma^+ \Box$ is an LNCA-refutation of length $|A| > 1$. Then we can write $A : G_1 \Rrightarrow_\theta G_2 \Rrightarrow_{\theta'}^+ \Box$ where $\theta\theta' = \sigma$. We want to prove that σ is a solution of G_1. By the induction hypothesis for $A_{>1}$, θ' is a solution of G_2. According to our first proof step, this implies that $\theta\theta'$ is a solution of G_1. Thus, $\sigma = \theta\theta'$ is a solution of G_1. □

4.4 Completeness

In this section we prove the completeness of LNCA for confluent ATRSs with respect to normalized substitutions. Subsection 4.4.2 contains an analysis of the structure of LNC-refutations generated from normal NC-refutations. Based on this analysis, we introduce the class of well-formed LNC-refutations and prove the completeness of LNC with respect to this class. Subsection 4.4.3 is concerned with the study of LNC-refutations for ATRSs. In Subsection 4.4.4 we state some properties of well-formed LNC-refutations for ATRSs. Finally, in Subsection 4.4.5 we prove the completeness of our calculus.

4.4.1 Preliminaries

We first recall some well known theoretical results that are relevant to our analysis of LNCA.

We denote by $\mathcal{LNC}$ the class of LNC-refutations. We say that an NC-derivation $\Pi : G \rightsquigarrow_\theta^* \top$ is *normal* if it respects the leftmost equation selection strategy and for every representation of Π in the form

$$\Pi : G = G_1, s \simeq t, G_2 \rightsquigarrow_{\theta_1}^* \top, (s \simeq t, G_2)\theta_1 \rightsquigarrow_{\theta_2}^* \top$$

the substitution $\theta_2\restriction_{\mathcal{V}(s\theta_1)}$ is normalized.

We denote by $\mathcal{NC}$ the class of normal NC-refutations. The following result was proven in [MH94]:

Theorem 2 *For every normalized solution θ of a goal G there exists $\Pi \in \mathcal{NC}$ such that $\Pi : G \rightsquigarrow^*_{\theta'} \top$ and $\theta' \leq \theta\ [\mathcal{V}(G)]$.*

Lemma 2 *There exists a well-founded order $\ll\ \subseteq \mathcal{NC} \times \mathcal{NC}$ such that:*

$$\begin{array}{l} \forall \Pi : G = e, G' \rightsquigarrow^+_{\theta} \top \in \mathcal{NC}.\exists \langle \Psi_1, \Pi_1 \rangle. \\ \quad \Psi_1 : G \Rightarrow_{\sigma} G_1 \wedge \Pi_1 : G_1 \rightsquigarrow^*_{\theta'} \top \in \mathcal{NC} \wedge \Pi_1 \ll \Pi \wedge Rel(\Pi, \Psi_1, \Pi_1) \\ \quad \wedge \sigma\theta' \leq \theta\ [\mathcal{V}(G)] \end{array}$$

where $Rel(\Pi, \Psi_1, \Pi_1)$ is defined as follows:

1. *The descendants of G' are narrowed in Π at the same positions and in the same order as the descendants of $G'\sigma$ in Π_1.*

2. *If Ψ_1 is a $[\mathrm{d}]$-step:*

   $$\Psi_1 : G = f(\overline{s_n}) \approx f(\overline{t_n}), G' \Rightarrow_{[\mathrm{d}]} \overline{s_n \approx t_n}, G'$$

   then $i \cdot j \cdot p$ is a narrowing position to a descendant of e in Π iff $i \cdot p$ is a narrowing position to a descendant of $s_j \approx t_j$ in Π_1.

3. *If Ψ_1 is an $[\mathrm{o}]$-step:*

   $$G = \underline{f(\overline{s_n})} \simeq t, G' \Rightarrow_{[o],k,f(\overline{l_n}) \to r} \overline{s_n \approx l_n}, r \simeq t, G'$$

   then:

   (a) Π *narrows a descendant of e at position k.*

   (b) Π_1 *does not narrow descendants of $s_i \approx l_i$ at positions in the rhs.*

   (c) *$1 \cdot p$ is a narrowing position to a descendant of $s_j \approx l_j$ in Π_1 iff $k \cdot j \cdot p$ is a narrowing position to a descendant of $f(\overline{s_n}) \simeq t$ in Π.*

   (d) *$2 \cdot p$ is a narrowing position to a descendant of $r \approx t$ in Π iff $(3-k) \cdot p$ is a narrowing position to a descendant of $f(\overline{s_n}) \simeq t$ in Π.*

4. *If Ψ_1 is an $[\mathrm{i}]$-step:*

   $$G = f(\overline{s_n}) \simeq x, G' \Rightarrow_{[i],k,\sigma=\{x \mapsto f(\overline{x_n})\}} \overline{s_n\sigma \approx x_n}, G'\sigma$$

   then:

   (a) Π *starts with an NC-step at a position of the form $k \cdot j \cdot p$ with $1 \leq j \leq n$.*

(b) $i \cdot j \cdot p$ is a narrowing position to a descendant of e in Π iff $i' \cdot p$ is a narrowing position to a descendant of $s_j\sigma \approx x_j$ in Π_1, where $i' = i$ if $k = 1$ and $i' = 3 - i$ if $k = 2$.

5. *If Ψ_1 is a [v]-step then Π starts with an NC-step at root position.*

Corollary 1 *Let $\Pi : G \leadsto_\theta^+ \top \in \mathcal{NC}$. Then the successive applications of Lemma 2, starting from Π, yield an LNC-refutation $\Psi : G \Rightarrow_{\theta'}^+ \Box \in \mathcal{LNC}$ such that $\theta' \leq \theta$ $[\mathcal{V}(G)]$.*

Proof. The result of successive applications of Lemma 2, starting from Π, is depicted in the figure below:

$$\begin{array}{ll}
\Pi = \Pi_0 : & G_0 = G \leadsto_{\theta_0 = \theta}^+ \top \\
& \Downarrow_{\sigma_0} \\
\Pi_1 : & G_1 \leadsto_{\theta_1}^+ \top \\
& \vdots \\
\Pi_i : & G_i \leadsto_{\theta_i}^+ \top \\
& \Downarrow_{\sigma_i} \\
\Pi_{i+1} : & G_{i+1} \leadsto_{\theta_{i+1}}^+ \top \\
& \Downarrow_{\sigma_{i+1}} \\
& \vdots
\end{array}$$

Since $\forall i.\Pi_{i+1} \ll \Pi_i$, this process will eventually terminate with an NC-refutation $\Pi_{n+1} : G_{n+1} = \Box \leadsto_{\theta_{n+1}=\varepsilon}^0 \top$. The LNC-refutation generated in this way is:

$$\Psi : G_0 \Rightarrow_{\sigma_0} G_1 \Rightarrow_{\sigma_1} \cdots \Rightarrow_{\sigma_n} G_{n+1} = \Box.$$

According to Lemma 2, $\forall i \in \{1, \ldots, n\}.\sigma_i\theta_{i+1} \leq \theta_i$ $[\mathcal{V}(G_i)]$. Then $\theta' = \sigma_0\sigma_1 \ldots \sigma_n = \sigma_0\sigma_1 \ldots \sigma_n\theta_{n+1} \leq \sigma_0\sigma_1 \ldots \sigma_{n-1}\theta_n \leq \ldots \leq \sigma_0\theta_1 \leq \theta_0 = \theta$ $[\mathcal{V}(G)]$. $\Box$

4.4.2 Well-formed LNC-refutations

We first introduce some useful notations. Let: $\Pi : G \leadsto^* \top \in \mathcal{NC}$ and $e \in G$. We define:

$\mathcal{P}(e, \Pi)$	the property that narrowing is never applied at positions of the rhs of a descendant of e in Π.
$E_p(\Pi)$	the longest prefix of G such that $\forall e \in E_p(\Pi).\mathcal{P}(e, \Pi)$.
$\psi(\Pi)$	the LNC-step constructed from Π as shown in Lemma 2.
$\pi(\Pi)$	the NC-refutation constructed from Π as shown in Lemma 2.
$\Psi(\Pi)$	the LNC-refutation constructed from Π as described in Corollary 1.

First we prove the following lemma:

Lemma 3 *Let* $\Pi : G_0 = \underline{e}, G' \rightsquigarrow_{\theta}^{+} \top \in \mathcal{NC}$ *and assume* $\pi(\Pi) : G_1 \rightsquigarrow_{\theta'}^{*} \top$. *Then the following conditions hold:*

1. *If* $\psi(\Pi)$ *is an* [i]*-step:*

$$G_0 = f(\overline{s_n}) \simeq x, G' \Rightarrow_{\sigma=\{x\mapsto f(\overline{x_n})\}} G_1 = (\overline{s_n \approx x_n}, G')\sigma$$

then in $\pi(\Pi)$ *narrowing is applied to at least one of the descendants of the equations* $\overline{s_n\sigma \approx x_n}$ *at a position of the lhs.*

2. *If* $E_p(\Pi) \neq \square$ *and* $\psi(\Pi) : G_0 = E_p(\Pi), G'_i \Rightarrow_\sigma G_1 = (G', G'_i)\sigma$ *then:*
   (a) If $\psi(\Pi)$ *is an* [i]*-step then it is applied to the lhs of* e,
   (b) $G'\sigma = E_p(\pi(\Pi))$,
   (c) if $\psi(\Pi)$ *is an* [o]*-step then it is applied to the lhs of* e.

Proof.

1. If $\psi(\Pi)$ is an [i]-step:

$$\psi(\Pi) : G_0 = f(\overline{s_n}) \simeq x, G' \Rightarrow_{[i],k,\sigma=\{x\mapsto f(\overline{x_n})\}} G_1 = \overline{s_n\sigma \approx x_n}, G'\sigma$$

then $\pi(\Pi) : \overline{s_n\sigma \approx x_n}, G'\sigma \rightsquigarrow^* \top$. According to Lemma 2, 4.(a), the first NC-step of Π is applied at a position of the form $k\cdot j\cdot p$ where $1 \leq j \leq n$. Then, according to Lemma 2, 4.(b), $\pi(\Pi)$ narrows a descendant of $s_j\sigma \approx x_j$ at position $1\cdot p$, which is a position of the lhs.

2. Since $E_p(\Pi) \neq \square$ there exists $e \in E_p(\Pi)$.
(a) Assume $\psi(\Pi)$ is an [i]-step. Then e is of the form $x \simeq f(\overline{s_n})$ with $x \in \mathcal{V}$ and $n > 0$. We want to prove that the rhs of e is x. If this is not the case then $e = x \approx f(\overline{s_n})$. By Lemma 2, 4.(a), Π starts with an NC-step at a position of the form $2\cdot j\cdot p$ in e where $1 \leq j \leq n$. Since $e \in E_p(\Pi)$, this case is impossible and therefore we must have $e = f(\overline{s_n}) \approx x$.
(b) Let $e' \in G'\sigma$. We have to prove that $e' \in E_p(\pi(\Pi))$, i.e. that the property $\mathcal{P}(e', \pi(\Pi))$ holds. We distinguish two cases:

(b1) e' is a descendant of e in $\psi(\Pi)$. Then $\psi(\Pi)$ is an [o]-, [d]- or [i]-step. If $\psi(\Pi)$ is an [o]-step then it is applied to the lhs since, by Lemma 2, 3.(a), the existence of an [o]-step to the rhs would imply $e \notin E_p(\Pi)$. Therefore, we can write:

$$\psi(\Pi) : \quad \underline{f(\overline{s_n})} \approx t, G' \Rightarrow_{[o], f(\overline{l_n})\to r} \overline{s_n \approx l_n}, r \approx t, G'$$

such that $e' = r \approx t$. If property $\mathcal{P}(r \approx t, \pi(\Pi))$ does not hold then there is a narrowing position to a descendant of $r \approx t$ in $\pi(\Pi))$ of the form $2{\cdot}p$. From Lemma 2, 3.(d) results the existence of a narrowing position of the form $2{\cdot}p$ to a descendant of e in Π. Since this contradicts the condition $e \in E_p(\Pi)$, we deduce that property $\mathcal{P}(e', \pi(\Pi))$ holds.

If $\psi(\Pi)$ is a [d]-step:

$$f(\overline{s_n}) \approx f(\overline{t_n}), G' \Rightarrow_{[\mathrm{d}]} \overline{s_n \approx t_n}, G'$$

then $e' = s_j \approx t_j$ for some $j \in \{1, \ldots, n\}$. Because $e \in E_p(\Pi)$, Π does not perform narrowing at positions of the form $2{\cdot}j{\cdot}p$ to descendants of e. From Lemma 2, 3.(b) we deduce that $\pi(\Pi)$ does not narrow descendants of $s_j \approx t_j$ at positions of the rhs. Thus, $\mathcal{P}(s_i \approx t_i, \pi(\Pi))$ holds.

If $\psi(\Pi)$ is an [i]-step then, according to 2.(a) of this lemma, e is of the form $f(\overline{s_n}) \approx x$ with $x \in \mathcal{V}$ and $n > 0$. In this case we can write:

$$f(\overline{s_n}) \approx x, G' \Rightarrow_{[i], \sigma = \{x \mapsto f(\overline{x_n})\}} \overline{s_n\sigma \approx x_n}, G'\sigma$$

and assume $e' = s_j\sigma \approx x_j$ for some $j \in \{1, \ldots, n\}$. We want to prove that $\pi(\Pi)$ does not narrow descendants of e' at positions of the rhs. If narrowing is applied to a descendant of e' at a position of the rhs then from Lemma 2, 4.(b) we deduce that narrowing is applied to a descendant of e at a position of the rhs. This contradicts our assumption that $e \in E_p(\Pi)$. Therefore, $\mathcal{P}(e', \pi(\Pi))$ must hold.

(b2) e' is not an LNC-descendant of e in $\psi(\Pi)$. Then e' is either a parameter-passing equation of e or an LNC-descendant of some $e'' \in E_p(\Pi) \cap G'$. The case when e' is a parameter-passing equation of e is covered by Lemma 2, 3.(b). The other case is an immediate consequence of Lemma 2, 1.

Hence $e' \in G'$ implies $e' \in E_p(\sigma(\Pi))$.

If $e' \notin G'$ then e' is a one-step descendant of an equation $\overline{e} \notin E_p(\Pi)$. Then narrowing is applied to the rhs of a descendant of e' in Π and by Lemma 2, 1. narrowing is applied to the rhs of e' in $\pi(\Pi)$. Thus, $e' \notin E_p(\sigma(\Pi))$.

(c) Assume $\psi(\Pi)$ is an [o]-step. From $e \in E_p(\Pi)$ and Lemma 2 3.(a) we deduce that $\psi(\Pi)$ is applied to the lhs of e. □

Lemma 4 *Let* $\Pi : G \rightsquigarrow_{\theta}^{+} \top \in \mathcal{NC}$.

1. *If an* [i]*-step is applied to a descendant* e' *of an equation* $e \in E_p(\Pi)$ *in* $\Psi(\Pi)$ *then it is applied to the lhs of* e'.

2. *If an* [o]*-step is applied to a descendant* e' *of an equation* $e \in E_p(\Pi)$ *in* $\Psi(\Pi)$ *then it is applied to the lhs of* e'.

Proof. In the proof we will make use of the following notations:

$\mathcal{P}_{[\mathrm{i}]}(\Pi)$: If an [i]-step is applied to a descendant e' of an equation $e \in E_p(\Pi)$ in $\Psi(\Pi)$ then it is applied to the lhs of e'.
$\mathcal{P}_{[\mathrm{o}]}(\Pi)$: If an [o]-step is applied to a descendant e' of an equation $e \in E_p(\Pi)$ in $\Psi(\Pi)$ then it is applied to the lhs of e'.

We prove by induction with respect to the order $\ll$ on $\mathcal{NC}$ that the properties $\mathcal{P}_{[\mathrm{o}]}(\Pi)$ and $\mathcal{P}_{[\mathrm{i}]}(\Pi)$ hold.

Let $\Pi_1 = \pi(\Pi)$. Because $\Pi_1 \ll \Pi$, from the induction hypothesis we get that $\mathcal{P}_{[\mathrm{i}]}(\Pi_1)$ and $\mathcal{P}_{[o]}(\Pi_1)$ hold. According to Lemma 3, 2.(b), all one-step descendants of equations of $E_p(\Pi)$ are in $E_p(\Pi_1)$. Then, by the induction hypothesis for Π_1, all [o]-steps to descendants of equations of $E_p(\Pi)$ in $\Psi(\Pi_1) = \Psi(\Pi)_{>1}$ are applied to the lhs. Moreover, if $\psi(\Pi)$ is an [o]-step then, by Lemma 3, 2.(c), $\psi(\Pi)$ is applied to the lhs of e. We conclude that $\mathcal{P}_{[\mathrm{o}]}(\Pi)$ holds.

It remains to prove that $\mathcal{P}_{[\mathrm{i}]}(\Pi)$ holds. Assume $e \in E_p(\Pi)$ such that an [i]-step is applied to a descendant of e in $\Psi(\Pi)$. We distinguish two cases:
(i) $\Psi(\Pi)$ starts with an [i]-step to e. Then, by Lemma 3, 2.(a), [i] is applied to the lhs of e.
(ii) $[i]$ is applied in $\Psi(\Pi_1)$ to a descendant of an immediate descendant e' of e in Π. According to Lemma 3, 2.(b), we have $e' \in E_p(\Pi_1)$ and the result follows from the induction hypothesis applied to Π_1. □

Definition 24 *Let* $\Psi \in \mathcal{LNC}$. *We define:*

$\mathcal{P}_{[\mathrm{o}]}(\Psi)$: *if an* [o]*-step is applied to a descendant* e *of a parameter-passing equation then it is applied to the lhs of* e.
$\mathcal{P}_{[\mathrm{i}]}(\Psi)$: *if an* [i]*-step is applied to a descendant* e *of a parameter-passing equation then it is applied to the lhs of* e.

The following theorem summarizes the main properties of LNC-refutations obtained by lifting normal NC-refutations.

Theorem 3 *Let* $\Pi : G \rightsquigarrow_{\theta}^{*} \top \in \mathcal{NC}$ *and* $\Psi_0 = \Psi(\Pi)$. *Then* Ψ_0 *satisfies the following properties:*

1. *If Ψ_0 contains a sub-refutation Ψ' that starts with an* [i]*-step:*

$$\Psi' : f(\overline{s_n}) \simeq x, G' \Rightarrow_{[\mathrm{i}],\sigma=\{x \mapsto f(\overline{x_n})\}} \overline{s_n\sigma \approx x_n}, G'\sigma \Rightarrow^*_{\theta'} \square$$

   then:

   (a) *The first step is of Ψ' not directly followed by n* [v]*-steps.*

   (b) *If $x \in \mathcal{V}(\overline{s_n})$ then $x\sigma\theta'$ is normalized.*

2. *The properties $\mathcal{P}_{[\mathrm{i}]}(\Psi_0)$ and $\mathcal{P}_{[\mathrm{o}]}(\Psi_0)$ hold.*

Proof. If $|\Pi| = 0$ then there is nothing left to prove. Otherwise, we can write $\Pi : G \Rightarrow^+_\theta \square$. By Corollary 1 we have $\Psi_0 : G \Rightarrow^*_{\theta'} \square$ where $\theta' \leq \theta\ [\mathcal{V}(G)]$.

Assume that Ψ_0 contains a sub-refutation Ψ' that starts with an [i]-step. Then we have the following situation:

$$\begin{array}{lll}
\Pi_0 = \Pi : & G_0 = G & \rightsquigarrow^+ \top \\
 & \Downarrow^* & \\
\Pi_k = \pi(\Pi_{k-1}) : & G_k = f(\overline{s_n}) \simeq x, G' & \rightsquigarrow^+ \top \\
 & \Downarrow_{[\mathrm{i}],\sigma} & \\
\Pi_{k+1} = \pi(\Pi_k) : & G_{k+1} = \overline{s_n\sigma \approx x_n}, G'\sigma & \rightsquigarrow^+ \top \\
 & \Downarrow^* & \\
 & \square & \rightsquigarrow^0_\varepsilon \top
\end{array}$$

where $\sigma = \{x \mapsto f(\overline{x_n})\}$. Since $\Pi_k \in \mathcal{NC}$, according to Lemma 3, 1., narrowing is applied in Π_{k+1} to at least one of the descendants of the equations $\overline{s_n\sigma \approx x_n}$ at a position of the lhs. Suppose $s_i\sigma \approx x_i$ is narrowed at a position of the lhs.

Assume now that the first step of Ψ' is followed by n [v]-steps. Then the construction of $\Psi'_{>1}$ is as depicted below.

$$\begin{array}{lll}
\Pi_{k+1} : & G_{k+1} = \overline{s_n\sigma \approx x_n}, G'\sigma & \rightsquigarrow^+ \top \\
 & \Downarrow^{i-1}_{[\mathrm{v}]} & \\
\Pi_{k+i} : & G_{k+i} = \overline{s_{i,n}\sigma_i \approx x_{i,n}}, G'\sigma_i & \rightsquigarrow^+ \top \\
 & \Downarrow^{n-i+1}_{[\mathrm{v}]} & \\
\Pi_{k+n+1} & G_{k+n+1} = G'\sigma_n & \rightsquigarrow^+ \top \\
 & \Downarrow^* & \\
 & \square &
\end{array}$$

where $\sigma_1 = \sigma\{x_1 \mapsto s_1\sigma\}$, ..., $\sigma_n = \sigma_{n-1}\{x_n \mapsto s_n\sigma_{n-1}\}$. According to Lemma 2, 1., the descendants of the equation $s_i\sigma \approx x_i$ are narrowed at a position of the lhs in Π_{k+1}, ... , Π_{k+i}. Since $\psi(\Pi_{k+i})$ is a [v]-step, from

Lemma 2, 5. we deduce that Π_{k+i} starts with a narrowing step at root position. This contradiction proves the validity of condition 1.(a).

We next prove condition 1.(b). Assume that $x \in \mathcal{V}(\overline{s_n})$. We want to prove that $x\sigma\theta'_1$ is normalized. By Lemma 2, 4.(a), Π_k starts with a step at non-root position. Since Π_k is normal, $\theta \restriction_{\mathcal{V}(f(\overline{s_n}))}$ is normalized. In particular, $x\theta$ is a normal form. Since $\theta' \leq \theta\ [\mathcal{V}(G)]$ and $x \in \mathcal{V}(G)$, we deduce that $x\theta$ is an instance of $x\theta'$, and therefore $x\theta'$ is normalized.

We prove now that $\mathcal{P}_{[\mathrm{o}]}(\Psi(\Pi))$ and $\mathcal{P}_{[\mathrm{i}]}(\Psi(\Pi))$ hold. Let e be a parameter-passing equation in $\Psi(\Pi)$. Then the construction of $\Psi(\Pi)$ from Π looks as follows:

$$
\begin{array}{lll}
\Pi_0 = \Pi : & G_0 = G & \leadsto^{+} \top \\
 & \Downarrow^{*} & \\
\Pi_k = \pi(\Pi_{k-1}) : & G_k = f(\overline{s_n}) \simeq t, G' & \leadsto^{+} \top \\
 & \Downarrow_{[\mathrm{o}], f(\overline{l_n}) \to r} & \\
\Pi_{k+1} = \pi(\Pi_k) : & G_{k+1} = s_1 \approx l_1, \ldots, \underbrace{s_i \approx l_i}_{e}, \ldots, s_n \approx l_n, r \approx t, G' & \leadsto^{+} \top \\
 & \Downarrow^{*} & \\
 & \square & \leadsto^{0}_{\varepsilon} \top
\end{array}
$$

By Lemma 2, 3.(b) we have that $s_1 \approx l_1, \ldots, s_n \approx l_n \in E_p(\Pi_{k+1})$. In particular, $e \in E_p(\Pi_{k+1})$. From Lemma 4, 1. for $\Pi_{k+1} \in \mathcal{NC}$ we know that if an [i]-step is applied to a descendant e' of e in $\Psi(\Pi)_{>k+1} = \Psi(\Pi_{k+1})$ then it is applied to the lhs. Hence $\mathcal{P}_{[\mathrm{i}]}(\Psi(\Pi))$ holds. Also, from Lemma 4, 2. for $\Pi_{k+1} \in \mathcal{NC}$ we know that if an [o]-step is applied to a descendant e' of e in $\Psi_{>k+1} = \Psi(\Pi_{k+1})$ then it is applied to the lhs of e'. Hence $\mathcal{P}_{[\mathrm{o}]}(\Psi(\Pi))$ holds. □

It is now appropriate to characterize the LNC-refutations generated by Ψ from normal NC-refutations.

Definition 25 (Well-formed LNC-refutation) $\Psi \in \mathcal{LNC}$ *is* well-formed *if it satisfies the following properties:*

1. *If Ψ contains a sub-refutation that starts with an* [i]*-step*

   $$\Psi' : f(\overline{s_n}) \simeq x, G' \Rightarrow_{[i],\sigma=\{x \mapsto f(\overline{x_n})\}} \overline{s_n\sigma \approx x_n}, G'\sigma \Rightarrow^{*}_{\theta'} \square$$

   then:

   (a) *the first step of Ψ' is not directly followed by n* [v]*-steps.*

   (b) *if $x \in \mathcal{V}(\overline{s_n})$ then $x\sigma\theta'$ is normalized.*

2. *Properties $\mathcal{P}_{[\mathrm{i}]}(\Psi)$ and $\mathcal{P}_{[\mathrm{o}]}(\Psi)$ hold.*

We denote by $\mathcal{WF}$ the class of well-formed LNC-refutations. An immediate consequence of Theorem 2 and Theorem 4 is:

Corollary 2 *For every normalized solution θ of G there exists $\Psi : G \Rightarrow^*_{\theta'} \square \in \mathcal{WF}$ with $\theta' \leq \theta$ $[\mathcal{V}(G)]$.*

At the end of this subsection we state some useful properties of well-formed LNC-refutations.

Lemma 5 *If $\Psi \in \mathcal{WF}$ then $\Psi_{>i}$ of Ψ is well-formed for all $1 \leq i \leq |\Psi|$.*

Lemma 6 *Let $\Psi \in \mathcal{WF}$ such that $\Psi_{>k} : \overline{s_n \approx u_n}, G' \Rightarrow^*_\theta \square$, where $\overline{s_n \approx u_n}$ are descendants of parameter-passing equations. Then*

$$\forall 1 \leq i \leq n.(s_i\theta \rightarrow^* u_i\theta) \tag{4.1}$$

<u>**Proof.**</u> Let $\Phi = \Psi_{>k}$. The proof is by induction on $|\Phi|$. If $|\Phi| = 1$ then $n = 1$ and Ψ consists of a [d]-, [v]- or [t]-step. In each of these cases, property (4.1) holds. Assume now $|\Phi| > 1$. We distinguish the following cases:

- Φ starts with a [v]- or a [t]-step. Then $s_1\theta = u_1\theta$. From the induction hypothesis for $\Phi_{>1}$ we have $s_i\theta \rightarrow^* u_i\theta$ if $2 \leq i \leq n$.

- Φ starts with an [o]-step to the lhs. Then $s = f(\overline{s'_k})$ and:

  $$\Phi_{>1} : \overline{s_n \approx u_n}, G' \Rightarrow_{[\mathrm{o}], f(\overline{u'_k}) \to r} \overline{s'_k \approx u'_k}, r \approx u_1, \overline{s_{2,n} \approx u_{2,n}}, G' \Rightarrow^*_\theta \square.$$

  From the induction hypothesis we have $s'_i\theta \rightarrow^* u'_i\theta$ $(1 \leq i \leq k)$, $r\theta \rightarrow^*_\theta u_1\theta$, $s_j\theta \rightarrow^*_\theta u_j\theta$ $(2 \leq j \leq n)$. It remains to prove that $s_1\theta \rightarrow^*_\theta u_1\theta$, which is obvious because $s_1\theta = f(\overline{s'_k\theta}) \rightarrow^* f(\overline{u'_k})\theta \rightarrow r\theta \rightarrow^* u_1\theta$.

- Φ starts with a [d]-step. Then $s_1 = f(\overline{s'_\ell}), u_1 = f(\overline{u'_\ell})$, and:

  $$\Phi : f(\overline{s'_\ell}) \approx f(\overline{u'_\ell}), \overline{s_{2,n} \approx u_{2,n}}, G' \Rightarrow_{[\mathrm{d}]} \overline{s'_\ell \approx u'_\ell}, \overline{s_{2,n} \approx u_{2,n}}, G' \Rightarrow^*_\theta \square.$$

  From the induction hypothesis we have $s'_i\theta \rightarrow^* u'_i\theta$ $(1 \leq i \leq \ell)$ and $s_j\theta \rightarrow^* u_j\theta$ $(2 \leq j \leq n)$. It remains to prove that $s_1\theta \rightarrow^*_\theta u_1\theta$, which is obvious because $s_1\theta = f(\overline{s'_\ell\theta}) \rightarrow^* f(\overline{u'_\ell})\theta = u_1\theta$.

- Φ starts with an [i]-step. By Lemma 3, 2.(a), Φ is of the form

  $$f(\overline{s'_\ell}) \approx u_1, \overline{s_{2,n} \approx u_{2,n}}, G' \Rightarrow_{[\mathrm{i}],\sigma_1} \overline{s'_\ell\sigma_1 \approx x_\ell}, \overline{s_{2,n}\sigma_1 \approx u_{2,n}}, G'\sigma_1 \Rightarrow^*_{\theta'} \square$$

  with $\sigma_1 = \{u_1 \mapsto f(\overline{x_n})\}$. From the induction hypothesis we have $s'_i\theta = s'_i\sigma_1\theta' \rightarrow^* x_i\theta'$ $(1 \leq i \leq \ell)$ and $s_j\theta = s'_j\sigma_1\theta' \rightarrow^* u_j\sigma_1\theta = u_j\theta$ $(2 \leq j \leq n)$. It remains to prove that $s_1\theta \rightarrow^*_\theta u_1\theta$, which is obvious because $s_1\theta = f(\overline{s'_\ell\theta}) \rightarrow^* f(\overline{x_\ell\theta'}) = f(\overline{x_\ell})\theta' = u_1\sigma_1\theta' = u_1\theta$. $\square$

Note that property 1. of well-formedness is not necessary to prove Lemma 6.

Corollary 3 *If* $\Psi : \underline{s} \simeq t, G' \Rightarrow_{[\mathrm{o}]}\Rightarrow^*_\theta \Box \in \mathcal{WF}$ *then* $s\theta$ *is not a normal form.*

<u>Proof.</u> Ψ can be written as:

$$f(\overline{s_n}) \simeq t, G' \Rightarrow_{[\mathrm{o}], f(\overline{l_n}) \to r}, \overline{s_n \approx l_n}, r \approx t, G'.$$

By Lemma 6 we have $\forall i \in \{1, \ldots, n\}. s_i\theta \to^* l_i\theta$. This implies:

$$s\theta = f(\overline{s_n})\theta \to^* f(\overline{l_n})\theta \to r\theta$$

and hence $s\theta$ is reducible. $\Box$

Lemma 7 *Let* $\Psi : G_1, s \approx t, G_2 \Rightarrow^*_\theta \Box \in \mathcal{WF}$ *such that* $s \approx t$ *is the* n*-th equation in the initial goal of* Ψ*. We denote by*

$$\phi_{\mathrm{swap}}(\Psi, n) : G_1, t \approx s, G_2 \Rightarrow^*_\theta \Box$$

the LNC-refutation obtained from Ψ *by performing the same inference steps in the same order at corresponding positions. Then* $\phi_{\mathrm{swap}}(\Psi, n)$ *is well-formed.*

<u>Proof.</u> From the construction of $\phi_{\mathrm{swap}}(\Psi, n)$ we see that $\phi_{\mathrm{swap}}(\Psi, n)$ verifies condition 1. of well-formedness. The validity of condition 2. of well-formedness for $\phi_{\mathrm{swap}}(\Psi, n)$ follows from its validity for Ψ and the observation that, due to the asymmetry of the [o]-inference rule, the descendants of parameter-passing equations are identical in Ψ and $\phi_{\mathrm{swap}}(\Psi, n)$. $\Box$

In the sequel we confine our attention to the case of ATRSs.

4.4.3 LNC-refutations for ATRSs

In this subsection we analyze the structure of LNC-refutations for the particular case of ATRSs. We first introduce the notions of immediate a-descendant and a-descendant of an equation.

Definition 26 (immediate a-descendant) *Let* $A : G = e, G' \Rightarrow G''$ *be an LNC inference step.*

- *If* $G = s_1\ s_2 \simeq t, G' \Rightarrow_{[\mathrm{o}], l_1\ l_2 \to r} G'' = s_1 \approx l_1, s_2 \approx l_2, r \approx t, G'$ *then* $s_1 \approx l_1$ *in* G'' *is the only immediate a-descendant of* e*.*
- *If* $G = f \simeq t, G' \Rightarrow_{[\mathrm{o}], f \to r} G'' = r \approx t, G'$ *then there is no immediate a-descendant of* e*.*

- *If* $G = s_1\ s_2 \simeq x, G' \Rightarrow_{[\mathrm{i}],\sigma=\{x\mapsto x_1\ x_2\}} G'' = s_1\sigma \approx x_1, s_2\sigma \approx x_2, G'\sigma$ *then* $s_1\sigma \approx x_1$ *in* G'' *is the only immediate a-descendant of* e.

- *If* $G = s_1\ s_2 \approx t_1\ t_2, G' \Rightarrow_{[\mathrm{d}]} G'' = s_1 \approx t_1, s_2 \approx t_2, G'$ *then* $s_1 \approx t_1$ *in* G'' *is the only immediate a-descendant of* e.

- *If* A *is a* [v]- *or a* [t]-*step then* e *has no immediate a-descendants.*

Definition 27 (a-descendant) *The relation of* a-descendant *is the reflexive-transitive closure of the relation of immediate a-descendant.*

Note the difference between the notions of a-descendant and descendant.

Lemma 8 *Let* $\Psi : G = s \approx t, G' \Rightarrow^*_\theta \Box$. *If the first* [o]-*step of* Ψ *is applied to an a-descendant of* $s \approx t$ *then there exists* $\Psi' \in \{\Psi, \phi_{\mathrm{swap}}(\Psi, 1)\}$ *such that:*

(i) all [i]-*steps before the first* [o]-*step in* Ψ' *are applied to the left-hand side,*

(ii) the first [o]-*step of* Ψ' *is applied to the lhs of an a-descendant of* $s \approx t$.

<u>Proof.</u> A simple case analysis reveals that if an [o]-step is applied to an a-descendant of $s \approx t$ then A starts with $m \geq 0$ [d]-steps, followed by $p \geq 0$ [i]-steps, followed by an [o]-step.

If $p = 0$ then we can write Ψ in the form:

$$\Psi : G = a\ \overline{u_n}\ \overline{s_m} \simeq x\ t_m, G' \Rightarrow^m_{[\mathrm{d}]} \underline{a\ \overline{u_n}} \simeq x, \overline{s_m \simeq t_m}, G' \Rightarrow_{[\mathrm{o}],k,l\to r} \Rightarrow^*_{\theta_2} \Box.$$

Then $\Psi' = \Psi$ if $k = 1$ and $\phi_{\mathrm{swap}}(\Psi, 1)$ if $k = 2$ obviously satisfies conditions (i)-(ii).

If $p > 0$ then we can write:

$$\begin{aligned}\Psi :\quad & G = a\ \overline{u_n}\ \overline{s_m} \simeq x\ \overline{t_m}, G' \Rightarrow^m_{[\mathrm{d}]} \underline{a\ \overline{u_n}} \simeq x, \overline{s_m \simeq t_m}, G' \\ & \Rightarrow^p_{[i],k,\theta_1} \underline{(a\ \overline{u_{n-p}}} \approx x_{n-p}, \overline{u_{n-p+1,n} \approx x_{n-p+1,n}}, \\ & \overline{s_m \approx t_m}, G')\theta_1 \Rightarrow_{[\mathrm{o}],k,l\to r} \Rightarrow^*_{\theta_2} \Box.\end{aligned}$$

If the first [i]-step is to the lhs (i.e., $k = 1$) then we can take $\Psi' = \Psi$, otherwise we can take $\Psi' = \phi_{\mathrm{swap}}(\Psi, 1)$. $\Box$

Lemma 9 *Let* $G = f\ \overline{s_m} \approx g\ \overline{t_n}, G'$ *such that* $f \neq g$ *or* $m \neq n$. *Then for every* $A : G \Rightarrow^* \Box \in \mathcal{LNC}$ *there exists an application of an [o]-step to an a-descendant of* $f\ \overline{s_m} \approx g\ \overline{t_n}$.

Proof. By induction on $n+m$. Obviously, A starts with an [o]-step or with a [d]-step. If A starts with an [o]-step then there is nothing more to prove. Assume now that A starts with a [d]-step. If $m=0$ then the only possibility is $n=0$ and $g=f$. Since this contradicts our hypothesis, we must have $m>0$. By a similar argument we infer that $n>0$ and therefore A can be written as:

$$A : G \Rightarrow_{[\mathrm{d}]} f\ \overline{s_{m-1}} \approx g\ \overline{t_{n-1}}, s_m \approx t_n, G'.$$

We can now apply the induction hypothesis to $A_{>1}$ and get the desired result. □

Lemma 10 *Let A be an LNC-refutation $f\ \overline{s_n} \approx f\ \overline{t_n}, G' \Rightarrow^*_\theta \square$. If there are no* [o]*-steps applied to a-descendants of $f\ \overline{s_n} \approx f\ \overline{t_n}$ in A, then A is of the form:*

$$A : f\ \overline{s_n} \approx f\ \overline{t_n}, G' \Rightarrow^{n+1}_{[\mathrm{d}]} \overline{s_n \approx t_n}, G' \Rightarrow \square.$$

Proof. By induction on n. If $n=0$ then the first step must be [d] and we are done. Suppose $n>0$. Then A starts with a [d]-step. Therefore A can be written as $A : f\ \overline{s_n} \approx f\ \overline{t_n}, G' \Rightarrow_{[\mathrm{d}]} f\ \overline{s_{n-1}} \approx f\ \overline{t_{n-1}}, s_n \approx t_n, G' \Rightarrow^*_\theta \square$ and the conclusion follows from the induction hypothesis for $A_{>1}$. □

Lemma 11 *Let $G = x\ \overline{s_n} \simeq g\ \overline{t_m}, G'$ such that $m < n$. Then for every LNC-refutation $A : G \Rightarrow^* \square$ there exists an application of an* [o]*-step to an a-descendant of $x\ \overline{s_n} \simeq g\ \overline{t_m}$.*

Proof. By induction on $n+m>0$. Since $0 \le m < n$, A starts either with an [o]-step or with a [d]-step. If A starts with an [o]-step then we are done. If not, then A starts with a [d]-step:

$$A : G \Rightarrow_{[\mathrm{d}]} x\ \overline{s_{n-1}} \simeq g\ \overline{t_{m-1}}, s_n \simeq t_m, G' \Rightarrow^* \square.$$

From the induction hypothesis for $A_{>1}$ we infer the existence of an [o]-step which is applied to an a-descendant of the immediate a-descendant of $x\ \overline{s_n} \simeq g\ \overline{t_m}$ in $A_{>1}$, and therefore to an a-descendant of $x\ \overline{s_n} \simeq g\ \overline{t_m}$ in A. □

Lemma 12 *Let $A : G = f\ \overline{s_m} \approx g\ \overline{t_n}, G' \Rightarrow^* \square$ such that $m < \mathrm{ar}(f)$ and $n < \mathrm{ar}(g)$. Then $m=n$, $f=g$, and $G \Rightarrow^{m+1}_{[\mathrm{d}]} \overline{s_m \approx t_m}, G' \Rightarrow^* \square$.*

Proof. By induction on $|A|$. If $|A|=1$ then A must consist of only a [d]-step. This implies $f=g$ and $m=n=0$. Assume now that $|A|>1$. We distinguish three cases for the first step in A:

1. $G \Rightarrow_{[\mathrm{d}]} f\ \overline{s_{m-1}} \approx g\ \overline{t_{n-1}}, s_m \approx t_n, G' \Rightarrow^* \Box$.
   From the induction hypothesis for $A_{>1}$ we get $f = g$ and $m-1 = n-1$, and we are done.

2. A starts with an [o]-step. Note that we can not have $m = 0$ in this case because there are no rewrite rules in $\mathcal{R}$ with lhs f. Hence $m > 0$ and we can assume that the first step of A is:
   $\underline{f\ \overline{s_m}} \approx g\ \overline{t_n}, G' \Rightarrow_{[\mathrm{o}],h\ \overline{l_k} \to r} f\ \overline{s_{m-1}} \approx h\ \overline{l_{k-1}}, s_m \approx l_k, r \approx g\ \overline{t_n}, G' \Rightarrow^* \Box$
   where $k = \mathrm{ar}(h)$. From the induction hypothesis for $A_{>1}$ we get $f = h$ and $m-1 = k-1$. This implies $\mathrm{ar}(f) = \mathrm{ar}(h) = k = m$. This case is impossible because we assume that $m < \mathrm{ar}(f)$.

3. $f\ \overline{s_m} \approx \underline{g\ \overline{t_n}}, G' \Rightarrow_{[\mathrm{o}],h\ \overline{l_k} \to r} g\ \overline{t_{n-1}} \approx h\ \overline{l_{k-1}}, t_n \approx l_k, r \approx f\ s_m, G' \Rightarrow^* \Box$.
   This case is also impossible and the proof similar to the previous one.

$\Box$

Lemma 13 *Let $A : G = f\ \overline{s_m} \approx t, G' \Rightarrow^* \Box$ such that it contains an* [o]*-step which is applied to an a-descendant of $f\ \overline{s_m} \approx t$. If the first* [o]*-step to an a-descendant of $f\ \overline{s_m} \approx t$ is applied to the lhs then $m \geq \mathrm{ar}(f)$.*

Proof. By induction on $|A|$. If $|A| = 0$ there is nothing more to prove. If $|A| = 1$ then A consists of a [d]- or a [v]-step and the lemma trivially holds. Otherwise we distinguish the following cases for the first step in A:

1. A starts with a [v]-step. Then there are no more a-descendants left.

2. A starts with an [o]-step to the lhs. If $m = 0$ then A this case is possible only if $\mathrm{ar}(f) = 0$ and then we are done. Otherwise $m > 0$ and we have:

   $$A : f\ \overline{s_m} \approx t, G' \Rightarrow_{[\mathrm{o}],h\ \overline{l_k} \to r} f\ \overline{s_{m-1}} \approx h\ \overline{l_{k-1}}, s_m \approx l_k, r \approx t, G' \Rightarrow^* \Box$$

   where $\mathrm{ar}(h) = k > 0$. If $m < \mathrm{ar}(f)$, then by Lemma 12 we must have $h = f$ and $m - 1 = k - 1$. But this implies $m = k = \mathrm{ar}(h) = \mathrm{ar}(f)$, which contradicts the assumption that $m < \mathrm{ar}(f)$. Thus, $m \geq \mathrm{ar}(f)$.

3. A starts with a [d]-step. If $m = 0$ then $t = f$ and there are no a-descendants left. If $m > 0$ then A can be written as:

   $$A : f\ \overline{s_m} \approx g\ \overline{t_n}, G' \Rightarrow_{[\mathrm{d}]} f\ \overline{s_{m-1}} \approx g\ \overline{t_{n-1}}, s_m \approx t_n, G' \Rightarrow^* \Box.$$

   From the induction hypothesis for $A_{>1}$ we deduce $m - 1 \geq \mathrm{ar}(f)$, and hence $m \geq \mathrm{ar}(f)$.

4. A starts with an [i]-step. Then $m > 0$ and A has the form:

   $$A : f\ \overline{s_m} \approx x, G' \Rightarrow_{[\mathrm{i}],\sigma=\{x\mapsto x_1\ x_2\}} (f\ \overline{s_{m-1}} \approx x_1, s_m \approx x_2, G')\sigma \Rightarrow^* \Box.$$

   The induction hypothesis for $A_{>1}$ yields immediately the desired result. $\Box$

Lemma 14 *Let* $A : \underline{f\ \overline{s_n}} \approx t, G' \Rightarrow_{[\mathrm{o}],l\to r} \Rightarrow^* \Box$ *where* $n = \mathrm{ar}(f)$. *Then* l *has the form* $f\ \overline{l_n}$.

Proof. If $n = 0$ then $l = f$. Otherwise l is of the form $h\ \overline{l_k}$ with $\mathrm{ar}(h) = k > 0$ and A is of the form

$$A : f\ \overline{s_n} \approx t, G' \Rightarrow_{[\mathrm{o}],h\ \overline{l_k}\to r} f\ \overline{s_{n-1}} \approx h\ \overline{l_{k-1}}, s_n \approx l_k, r \approx t, G' \Rightarrow^* \Box.$$

From Lemma 12 for $A_{>1}$ we have $f = h$ and $n = k$. $\Box$

Lemma 15 *Let* $A : \underline{x\ \overline{s_n}} \approx t, G' \Rightarrow_{[o],f\ \overline{l_k}\to r} \Rightarrow^* \Box$ *such that* $n > 0$ *and* [o] *is never applied to an a-descendant of* $x\ \overline{s_n} \approx t$ *in* $A_{>1}$ *.Then* $k \geq n$.

Proof. Without loss of generality, A can be written as:

$$\begin{array}{ll} \underline{x\ \overline{s_n}} \approx t, G' & \Rightarrow_{[\mathrm{o}],f\ \overline{l_k}\to r} x\ \overline{s_{n-1}} \approx f\ \overline{l_{k-1}}, s_n \approx l_k, r \approx t, G' \\ & \Rightarrow^{n-1}_{[\mathrm{d}]} x \approx f\ \overline{l_j}, s_1 \approx l_{j+1}, \ldots, s_n \approx l_k, r \approx t, G' \Rightarrow^* \Box \end{array}$$

such that $j \geq 0$ and $k = j + n$. Then $k \geq n$. $\Box$

Lemma 16 *If* $A \in \mathcal{LNC}$ *is of the form*

$$A : G = a\ \overline{t_n} \approx x, G' \Rightarrow_{[\mathrm{v}],\sigma=\{x\mapsto a\ \overline{t_n}\}} G'\sigma \Rightarrow^*_\theta \Box$$

where $n > 0$ *then there exists a refutation:*

$$\begin{array}{rl} A' : G = a\ \overline{t_n} \approx x, G' \Rightarrow_{[\mathrm{i}],\sigma_1=\{x\mapsto x_1\ x_2\}} & (a\ \overline{t_{n-1}} \approx x_1, t_n \approx x_2, G')\sigma_1 \\ \Rightarrow_{[\mathrm{v}],\sigma_2=\{x_1\mapsto a\ \overline{t_{n-1}}\}} & (t_n \approx x_2, G')\sigma_1\sigma_2 \\ \Rrightarrow_{[\mathrm{V}],\sigma_3=\{x_2\mapsto t_n\}} & G'\sigma_1\sigma_2\sigma_3 \Rightarrow^*_\theta \Box. \end{array}$$

such that $A_{>1} = A'_{>3}$ *and* $\sigma_1\sigma_2\sigma_3\restriction_{\mathcal{V}(G)} = \sigma$.

Proof. From the applicability of a [v]-inference step to the equation $a\ \overline{t_n} \approx x$ we infer that $x \notin \mathcal{V}(a\ \overline{t_n})$. Let x_1, x_2 be fresh variables and $\sigma_3 = \{x_2 \mapsto t_n\}$. Because x, x_1, $x_2 \notin \mathcal{V}(a\ \overline{t_n})$ we can construct an LNC-derivation:

$$\begin{array}{ll} A' : G & \Rightarrow_{[\mathrm{i}],\sigma_1} a\ \overline{t_{n-1}} \approx x_1, t_n \approx x_2, G'\sigma_1 \\ & \Rightarrow_{[\mathrm{v}],\sigma_2} t_n \approx x_2, G'\sigma_1\sigma_2 \Rightarrow_{[\mathrm{v}],\sigma_3} G'\sigma_1\sigma_2\sigma_3 \end{array}$$

Let $G_1 = t_n \approx x_2, G'\sigma_1\sigma_2$. Note that we can apply a [V]-step (with $n = 0$) to G_1:

$$G_1 \Rrightarrow_{[\mathrm{V}],\sigma_3} G'\sigma_1\sigma_2\sigma_3$$

We have $\sigma_1\sigma_2\sigma_3 = \{x \mapsto a\ \overline{t_n}, x_1 \mapsto a\ \overline{t_{n-1}}, x_2 \mapsto t_n\}$. Then $\sigma_1\sigma_2\sigma_3 \restriction_{\mathcal{V}(G)} = \sigma$ because $x_1, x_2 \notin \mathcal{V}(G)$. Since $\mathcal{V}(G') \subseteq \mathcal{V}(G)$, we have $G'\sigma_1\sigma_2\sigma_3 = G'\sigma$. Therefore, we can replace the second [v]-step of A' with a [V]-step and obtain the (mixed) refutation A'. □

4.4.4 Well-formed LNC-refutations for ATRSs

Lemma 17 *Let $A : G = a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}, G' \Rightarrow^*_\theta \square \in \mathcal{WF}$. If the* [o]*-inference rule is never applied to an a-descendant of $a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}$ then there exists an LNCA-derivation $B : G \Rrightarrow^*_\sigma G_1$ and $A' : G_1 \Rightarrow^*_{\theta'} \square \in \mathcal{WF}$ such that $|A'| < |A|$ and $\theta = \sigma\theta'$ $[\mathcal{V}(G)]$.*

<u>**Proof.**</u> Because the [o]-inference rule is never applied to a-descendants of $a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}$ of G, the first n LNC-steps of A must be [d]-steps. Hence, A can be written as:

$$A : G = a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}, G' \Rightarrow^n_{[\mathrm{d}]} a\ \overline{s_m} \simeq x, \overline{t_n \simeq u_n}, G' \Rightarrow^*_\theta \square.$$

We distinguish the following situations:

(1) <u>$a = x$.</u> We prove that in this case we must have $m = 0$. Assume $m \neq 0$. Then the only possibility is to start $A_{>n}$ with an [i]-step. We notice that in this case all the subsequent LNC-steps are [i]-steps and A is non-terminating. Therefore $m = 0$ and the first step of $A_{>n}$ is a [t]-step:

$$A_{>n} : x \approx x, \overline{t_n \simeq u_n}, G' \Rightarrow_{[t]} \overline{t_n \simeq u_n}, G' \Rightarrow^*_\theta \square.$$

In this case we can replace the first $n+1$ LNC-steps of A with a [dv]-step:

$$B : G \Rrightarrow_{[dv]} \overline{t_n \simeq u_n}, G' \Rightarrow^*_\theta \square.$$

We can take $A' = A_{>n+1}$ with $\theta' = \theta$, $\sigma = \varepsilon$.

(2) <u>$a \neq x$.</u> Then the only possibility is to start $A_{>n}$ with a sequence of i [i]-steps, where $0 \leq i \leq m$, followed by a [v]-step. There are two possibilities:

(2a) <u>$i = 0$.</u> In this case $A_{>n}$ can be written as:

$$a\ \overline{s_m} \simeq x, \overline{t_n \simeq u_n}, G' \Rightarrow_{[\mathrm{v}],\sigma} (\overline{t_n \simeq u_n}, G')\sigma \Rightarrow_{\theta'} \square$$

with $\sigma = \{x \mapsto a\ \overline{s_m}\}$. This implies that $x \notin \mathcal{V}(a\ \overline{s_m})$ and therefore we can perform the [V]-step:

$$B : G \Rrightarrow_{[\mathrm{V}],\sigma} (\overline{t_n \simeq u_n}, G')\sigma$$

In this case we can choose $G_1 = (\overline{t_n \simeq u_n}, G')\sigma$ and $A' = A_{>n+1}$.

(2b) $\underline{i > 0.}$ In this case $A_{>n}$ can be written as:

$$
\begin{array}{l}
a\ \overline{s_m} \simeq x, \overline{t_n \simeq u_n}, G' \\
\Rightarrow^{i}_{[\mathrm{i}],\sigma_1\ldots\sigma_i} \quad (a\ \overline{s_{m-i}} \approx x'_{m-i+1}, \overline{s_{m-i+1,m} \approx x_{m-i+1,m}}, \overline{t_n \simeq u_n}, \\
\qquad\qquad\quad G')\sigma_1 \ldots \sigma_i \\
\Rightarrow_{[\mathrm{v}],\sigma'_i} G_1 = (\overline{s_{m-i+1,m} \approx x_{m-i+1,m}}, \overline{t_n \simeq u_n}, G')\sigma_1 \ldots \sigma_i \sigma'_i \Rightarrow^*_{\theta'} \square
\end{array}
$$

where $\sigma_1 = \{x \mapsto x'_m\ x_m\}, \ldots, \sigma_i = \{x'_{m-i+2} \mapsto x'_{m-i+1}\ x_{m-i+1}\}$ and $\sigma'_i = \{x'_{m-i+1} \mapsto a\ s_{m-i}\}$ with $x_{m-i+1}, x'_{m-i+1}, \ldots, x_m, x'_m \in \mathcal{V} - \mathcal{V}(G_1)$ fresh variables. By applying Lemma 16 $m-i$ times to the first [v]-step of A, we obtain:

$$
\begin{array}{lll}
A'' : & G & \Rightarrow^{n}_{[\mathrm{d}]} a\ \overline{s_m} \simeq x, \overline{t_n \simeq u_n}, G' \\
& & \Rightarrow^{i+(m-i)}_{[\mathrm{i}],\sigma_1\ldots\sigma_m} (a \approx x'_1, \overline{s_m \approx x_m}, \overline{t_n \simeq u_n}, G')\sigma_1 \ldots \sigma_m \\
& & \Rightarrow_{[\mathrm{v}],\sigma'_m} (\overline{s_m \approx x_m}, \overline{t_n \simeq u_n}, G')\sigma_1 \ldots \sigma_m \sigma'_m \\
& & \Rrightarrow^{m-i}_{[\mathrm{V}],\sigma'} G_1 = (\overline{s_{m-i+1,m} \approx x_{m-i+1,m}}, \overline{t_n \simeq u_n}, G')\sigma_1 \ldots \sigma_i \sigma'_i \\
& & \Rightarrow_{\theta'} \square
\end{array}
$$

where $\sigma' = \{x_1 \mapsto s_1, \ldots, x_{m-i} \mapsto s_{m-i}\}$, $\sigma_1 \ldots \sigma_m \sigma'_m \sigma' \restriction_{\mathcal{V}(G)} = \sigma_1 \ldots \sigma_i \sigma'_i$, and $A''_{>n+m+1+m-i} = A_{>n+i+1}$. We have $\sigma_1 \ldots \sigma_m \sigma'_m \restriction_{\mathcal{V}(G)} = \{x \mapsto a\ \overline{x_n}\}$. We let $\rho = \{x \mapsto a\ \overline{x_n}\}$ and the LNCA-step:

$$
a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}, G' \Rrightarrow_{[\mathrm{I}],\rho} (\overline{s_m \approx x_m}, \overline{t_n \simeq u_n}, G')\rho
$$

replace the first $n+m+1$ LNC-steps of A''. Now we can choose:

$$
B : G \Rrightarrow_{[\mathrm{I}],\rho} \Rrightarrow^{m-i}_{[\mathrm{V}],\sigma'} G_1
$$

and $A' = A_{>n+i+1}$. By 2., we have $\theta = \rho\sigma'\theta'$ $[\mathcal{V}(G)]$. $\square$

The following lemma is of importance when lifting a well-formed LNC-refutation to an LNCA-refutation requires the introduction of an [i]-step.

Lemma 18 *Let $A : G = (G'_1, r \approx x_1, s \approx x_2, G'_2)\sigma \Rightarrow^*_{\theta} \square$, be a well-formed LNCA-refutation where $\sigma = \{x \mapsto x_1\ x_2\}$ and such that:*

(i) $x_1, x_2 \in \mathcal{V} - \mathcal{V}(r, s, x, G'_1, G'_2)$,

(ii) if $x \in \mathcal{V}(G'_1, r, s)$ then $x\sigma\theta$ is normalized.

Then there exists $A' \in \mathcal{WF}$ of the form: $A' : G' = (G'_1, r\ s \approx x, G'_2) \Rightarrow^*_{\theta'} \square$ such that $\sigma\theta = \theta'$ and $|A'| \le |A| + 1$.

<u>**Proof.**</u> We distinguish two cases:

1. A has a sub-refutation $(x_1\ x_2 \simeq t, G_1, r \approx x_1, s \approx x_2, G'_2)\sigma\theta_1 \Rightarrow^* \square$ which does not start with a [v]- or an [o]-step applied to $t\sigma\theta_1$,

2. A does not have such a sub-refutation.

First we prove case 1. Let

$$A'' = A_{>i_1} : G_{i_1} = (x_1\ x_2 \simeq t, G_1, r \approx x_1, s \approx x_2, G'_2)\sigma\theta_1 \Rightarrow^* \square$$

be the longest sub-refutation of A which does not start with an [o]- or a [v]-step applied to $t\sigma\theta_1$. Then obviously $x \in \mathcal{V}(G'_1)$ and, according to hypothesis (ii), $x\sigma\theta$ is normalized. This implies that $(x_1\ x_2)\theta$ is a normal form.

We notice that the only way a term of the form $x_1\ x_2$ is decomposed in the sub-derivation $B : G \Rightarrow^{i_1}_{\theta_1} G_{i_1}$ of A is by applying a [d]-, [i]- or [o]-step to an equation of the form $x_1\ x_2 \simeq v$ where v is any term. From the definition of A'' we deduce that such steps do not appear in B and therefore the following conditions hold:

- $x_1, x_2 \notin \mathcal{D}(\theta_1)$,
- if x_1 and x_2 appear in $\mathcal{I}(\theta_1)$ then they appear in subterms of the form $x_1\ x_2$.

As a consequence $(x_1\ x_2)\sigma\theta_1 = x_1\ x_2$. Because $A \in \mathcal{WF}$, from Corollary 3 we obtain by contraposition that A'' does not start with an [o]-step applied to $x_1\ x_2$. If A'' starts with an [i]-step then we must have $t\sigma\theta_1 = z \in \mathcal{V}$ and A'' is of the form:

$$\begin{aligned} &A'' : x_1\ x_2 \simeq z, G'' \Rightarrow_{[\mathrm{i}],\sigma'=\{z\mapsto y_1\ y_2\}} x_1 \approx y_1, x_2 \approx y_2, G''\sigma' \\ &\Rightarrow^2_{[\mathrm{v}],\sigma''} G''\sigma'\sigma'' \Rightarrow^* \square \end{aligned}$$

with $G'' = (G_1, r \approx x_1, s \approx x_2, G'_2)\sigma\theta_1$. Since this contradicts the assumption that A'' is well-formed, we have that A'' does not start with an [i]-step. Hence the next step of A must be a [d]-step. In this case $t\sigma\theta_1 = v_1\ v_2$ for some terms v_1, v_2 and G'_1 is of the form $G''_0, x \simeq t, G''_1$ such that we can write:

$$\begin{aligned} A : G = \;& (G''_0, x \simeq t, G''_1, r \approx x_1, s \approx x_2, G'_2)\sigma \\ \Rightarrow^{i_1}_{\theta_1}\;& G_{i_1} = x_1\ x_2 \simeq v_1\ v_2, (G''_1, r \approx x_1, s \approx x_2, G'_2)\sigma\theta_1 \\ \Rightarrow_{[\mathrm{d}]}\;& x_1 \simeq v_1, x_2 \simeq v_2, (G''_1, r \approx x_1, s \approx x_2, G'_2)\sigma\theta_1 \\ \Rightarrow^{i_2}_{\theta_2}\;& (G''_1, r \approx x_1, s \approx x_2, G'_2)\sigma\theta_1\theta_2 \\ \Rightarrow^{i_3}_{\theta_3}\;& G_{i_1+i_2+i_3+1} = (r \approx x_1, s \approx x_2, G'_2)\sigma\theta_1\theta_2\theta_3 \\ \Rightarrow^{i_4}_{\theta_4}\;& \square. \end{aligned}$$

Starting from A, we construct A' as follows. Let $B_4 \in \mathcal{LNC}$ be of the form:

$$B_4 : (r\ s \approx x_1\ x_2, G'_2)\sigma\theta_1\theta_2\theta_3 \Rightarrow_{[\mathrm{d}]} G_{i_1+i_2+i_3+1} \Rightarrow^{i_4}_{\theta_4} \Box$$

such that $(B_4)_{>1} = A_{>(i_1+i_2+i_3+1)}$. Then $B_4 \in \mathcal{WF}$ because $A_{>(i_1+i_2+i_3+1)} \in \mathcal{WF}$. Let $B'_3 \in \mathcal{LNC}$ be of the form:

$$B'_3 : x_1 \simeq v_1, x_2 \simeq v_2, (G''_1, r\ s \approx x_1\ x_2, G'_2)\sigma\theta_1 \Rightarrow^{i_2+i_3+i_4+1}_{\theta_2\theta_3\theta_4} \Box$$

such that:

- The first $i_2 + i_3$ steps of B'_3 coincide with the first $i_2 + i_3$ steps of $A_{>i_1+1}$,
- $(B'_3)_{>(i_2+i_3)} = B_4$.

Then $B'_3 \in \mathcal{WF}$. From B'_3 we construct

$$B_3 : v_1 \approx x_1, v_2 \approx x_2, (G''_1, r\ s \approx x_1\ x_2, E_2)\sigma\theta_1 \Rightarrow^{i_2+i_3+i_4+1}_{\theta_2\theta_3\theta_4} \Box$$

by permuting, if necessary, the sides of the first two equations and applying the same inference steps in the same order at corresponding positions. Since $B'_3 \in \mathcal{WF}$, from Lemma 7 we obtain that $B_3 \in \mathcal{WF}$.

Since $x_1, x_2 \notin G'_1$ we have that $x_1, x_2 \notin G''_1$. We already noticed that $x_1, x_2 \notin \mathcal{D}(\theta_1)$ and if x_1 and x_2 appear in $\mathcal{I}(\theta_1)$ then they appear in subterms of the form $x_1\ x_2$. Therefore we can remove all occurrences of x_1 and x_2 from $\mathcal{I}(\theta_1)$ by replacing all the occurrences of $x_1\ x_2$ by x. Assume that by this transformation we obtain δ_1 from θ_1. Because $\sigma\theta_1 = \delta_1\sigma$ we can consider the LNC refutation

$$\begin{aligned} B'_2 :\ & G'' = (x \approx t, G''_1, r\ s \approx x, G'_2)\delta_1 \\ & \Rightarrow_{[\mathrm{i}],\sigma} v_1 \approx x_1, v_2 \approx x_2, (G'_1, r\ s \approx x_1\ x_2, G'_2)\sigma\theta_1 \\ & \Rightarrow^{i_2+i_3+i_4+1}_{\theta_2\theta_3\theta_4} \Box \end{aligned}$$

where $(B'_2)_{>1} = B_3$. Then $x\delta_1\sigma\theta_2\theta_3\theta_4 = x\sigma\theta_1\theta_2\theta_3\theta_4 = x\sigma\theta$ is normalized. The only case when $B'_2 \notin \mathcal{WF}$ is where the first [i]-step is followed by two [v]-steps. In this case we define B_2 as the LNC-refutation obtained from B'_2 by replacing the first three steps by a [v]-step. Otherwise we assume $B_2 = B'_2$. Then $B_2 \in \mathcal{WF}$ and $|B_2| \leq i_2 + i_3 + i_4 + 2$. We finally define:

$$\begin{aligned} A' : G' = (G''_1, r\ s \approx x, G'_2) \ & \Rightarrow^{i_1}_{\delta_1} G'' = (x \approx v_1\ v_2, G''_1, r\ s \approx x, G'_2)\delta_1 \\ & \Rightarrow^{\leq i_2+i_3+i_4+2}_{\sigma\theta_2\theta_3\theta_4} \Box \end{aligned}$$

where the first i_1 steps coincide with those of A and are applied in the same order at the same positions, and $A'_{>i_1} = B_2$. Then $A' \in \mathcal{WF}$ and $|A'| = i_1 + |B_2| \leq i_1 + i_2 + i_3 + i_4 + 2 = |A| + 1$.

We now prove case 2. In this case A can be written as:

$$A : G = (G'_1, r \approx x_1, s \approx x_2, G'_2)\sigma \Rightarrow^{i_1}_{\theta_1} G_{i_1} = (r \approx x_1, s \approx x_2, G'_2)\sigma\theta_1 \Rightarrow^{i_2}_{\theta_2} \square$$

such that $x_1, x_2 \notin \mathcal{D}(\sigma\theta_1)$. From A we construct the LNC-refutation B as follows:

$$\begin{array}{llll} A: & G \Rightarrow^{i_1}_{\theta_1} & (r \approx x_1, s \approx x_2, G'_2)\sigma\theta_1 & \Rightarrow^{i_2}_{\theta_2} \square \\ & & \Uparrow_{[\mathrm{i}],\sigma} & \\ B: & G \Rightarrow^{i_1}_{\delta_1} & G'_{i_1} = (r\ s \approx x, G'_2)\delta_1 & \end{array}$$

where δ_1 is defined like in case 1, the first i_1 steps of B coincide with the first i_1 steps of A and are applied in the same order and at the same positions, and $B_{>i_1+1} = A_{>i_1}$. If $A_{>i_1}$ starts with two [v]-steps then we define A' to be B in which the sequence $\Rightarrow_{[\mathrm{i}]}\Rightarrow_{[\mathrm{v}]}\Rightarrow_{[\mathrm{v}]}$ of steps to G'_{i_1} is replaced by a [v]-step. Otherwise $A' = B$. Then $A' \in \mathcal{WF}$ and $|A'| \leq i_1 + i_2 + 1 = |A| + 1$. $\square$

Lemma 19 *Let $A : f\ \overline{s_n} \approx t, G \Rightarrow^*_\theta \square \in \mathcal{WF}$ such that:*

(i) There exists a first [o]*-step of A which is applied to the lhs of an a-descendant of $f\ \overline{s_n} \approx t$,*

(ii) All [i]*-steps before the first* [o]*-step are applied to the lhs.*

Then there exists a fresh variant $f\ \overline{u_m} \to r$ of a rewrite rule such that:

(a) The last [o]*-step to an a-descendant of $f\ \overline{s_n} \approx t$ is of the form*

$$\begin{array}{ll} A_1: & (f\ \overline{s_m} \approx t')\sigma, G' \\ & \Rightarrow_{f\ \overline{u_m} \to r} (f\ \overline{s_{m-1}} \approx f\ \overline{u_{m-1}}, s_m \approx u_m, r\ \overline{s_{m+1,n}} \approx t')\sigma, G' \end{array}$$

where $m \leq n$,

(b) There exists $A' \in \mathcal{WF}$ of the form

$$A' : \overline{s_m \approx u_m}, r\ \overline{s_{m+1,n}} \approx t, G \Rightarrow^*_\theta \square$$

such that $|A'| < |A|$.

Proof. We first prove (a). Because of (ii), the a-descendant e' of $f\ \overline{s_m} \approx t$ to which the first [o]-step is applied is of the form $e' = (f\ \overline{s_p} \approx t'')\sigma'$ where $p \leq n$. Since the a-descendants of e' are descendants of parameter-passing equations and the conditions $\mathcal{P}_{[\mathrm{o}]}(A)$ and $\mathcal{P}_{[\mathrm{i}]}(A)$ of well-formedness hold, we deduce that the a-descendant of e to which the last [o]-step is applied is of the form $(f\ \overline{s_m} \approx t')\sigma$ where $m \leq p \leq n$. It remains to prove that the rewrite rule variant employed in this last [o]-step is of the form $f\ \overline{u_m} \to r$.

By Lemma 14, it suffices to prove that $m = \mathrm{ar}(f)$. By Lemma 13 for the well-formed sub-refutation A'' of A starting with A_1 we have $m \geq \mathrm{ar}(f)$. If $m = 0$ then also $\mathrm{ar}(f) = 0$ and there is nothing more to prove. If $m > 0$ then we can write A'' as follows:

$$\begin{aligned}(f\ \overline{s_m} \approx t')\sigma, G' \ &\Rightarrow_{h\ \overline{u_k} \to r} (f\ \overline{s_{m-1}} \approx h\ \overline{u_{k-1}}, s_m \approx u_k, r\ \overline{s_{m+1,n}} \approx t')\sigma, G' \\ &\Rightarrow^* \Box\end{aligned}$$

where $k = \mathrm{ar}(h) > 0$. Since $A''_{>1}$ does not contain [o]-steps applied to a-descendants of $f\ \overline{s_{m-1}} \approx h\ \overline{u_{k-1}}$, we can apply Lemma 9 to $A''_{>1}$ and obtain by contraposition that $f = h$ and $m-1 = k-1$. Hence $\mathrm{ar}(f) = \mathrm{ar}(h) = k = m$.

We prove now (b). We prove by induction on $n - m$ the existence a well-formed LNC-refutation

$$A' : \overline{s_m \approx u_m}, r\ \overline{s_{m+1,n}} \approx t, G \Rightarrow^*_\theta \Box$$

which in addition to $|A'| < |A|$ it also satisfies the condition:

$\mathcal{C}(A, A')$: If $A : e, G \Rightarrow^* \Box$ then for every $e' \in G$ the following implication holds: if [o] is never applied to the rhs of the descendants of e' in A then [o] is never applied to the rhs of the descendants of e' in A'.

This condition is used in the proof of case 2., where we construct an LNC-refutation with a new parameter-passing equation.

Case I. Assume $n = m$. Then we distinguish two subcases:

(a) $\underline{m = 0}$. Because property (a) holds, A is of the form $A : \underline{f} \approx t, G \Rightarrow_{[o], f \to r} r \approx t, G \Rightarrow^*_\theta \Box$ and we can take $A' = A_{>1}$. Obviously, $\mathcal{C}(A_{>1}, A')$ implies $\mathcal{C}(A, A')$.

(b) $\underline{m > 0}$. Because of property (a), the first LNC-step of A coincides with the last [o]-step to an a-descendant of $f\ \overline{s_m} \approx t$. Therefore, we can write:

$$A : \underline{f\ \overline{s_m}} \approx t, G \Rightarrow_{[o], f\ \overline{u_m} \to r} f\ \overline{s_{m-1}} \approx f\ \overline{u_{m-1}}, s_m \approx u_m, r \approx t, G \Rightarrow^*_\theta \Box.$$

From Lemma 12 we know that $A_{>1}$ contains a sub-refutation:

$$A' : \overline{s_m \approx u_m}, r \approx t, G \Rightarrow^*_\theta \Box$$

such that $|A'| \leq |A_{>1}| < |A|$ and $\mathcal{C}(A_{>1}, A')$. Also, $\mathcal{C}(A_{>1}, A')$ implies $\mathcal{C}(A, A')$.

Case II. Assume $n > m$. We distinguish the following situations:

1. A starts with a [d]-step. Then $t = t_1\ t_2$ for some terms t_1, t_2 and we have:

$$A : f\ \overline{s_n} \approx t_1\ t_2, G \Rightarrow_{[\mathrm{d}]} f\ s_{n-1} \approx t_1, s_n \approx t_2, G \Rightarrow^*_\theta \square.$$

Since $A_{>1}$ has properties (i) and (ii), from the induction hypothesis we infer the existence of $B \in \mathcal{WF}$ of the form:

$$B : G'_1 = (\overline{s_m \approx u_m}, r\ \overline{s_{m+1,n-1}} \approx t_1, s_n \approx t_2, G) \Rightarrow^*_\theta \square$$

such that $|B| < |A_{>1}|$ and $\mathcal{C}(A_{>1}, B)$ holds. Also, $\mathcal{C}(A_{>1}, B)$ implies $\mathcal{C}(A, B)$. Let $B_{>i}$ be the sub-refutation of B such that

$$B_{>i} : G_2 = (r\ \overline{s_{m+1,n-1}} \approx t_1, s_n \approx t_2, G)\theta_1 \Rightarrow^j_{\theta_2} \square.$$

We construct the LNC-refutation

$$\begin{aligned} A' :\ & G_1 = \overline{s_m \approx u_m}, r\ s_{m+1,n} \approx t_1\ t_2, G \\ & \Rightarrow^i_{\theta_1} G'_2 = (r\ \overline{s_{m+1,n}} \approx t_1\ t_2, G)\theta_1 \\ & \Rightarrow_{[\mathrm{d}]} G_2 = (r\ \overline{s_{m+1,n-1}} \approx t_1, s_n \approx t_2, G)\theta_1 \Rightarrow^j_{\theta_2} \square \end{aligned}$$

where the first i steps of A' coincide with those of B and $A'_{>(i+1)} = B_{>i}$. Then $A' \in \mathcal{WF}$ and $|A'| = i + j + 1 = |B| + 1 < |A_{>1}| + 1 = |A|$. Since $B \in \mathcal{WF}$, we deduce that $A' \in \mathcal{WF}$. Moreover, $\mathcal{C}(A, B)$ implies $\mathcal{C}(A, A')$.

2. A starts with an [o]-step to the lhs. Then A is of the form:

$$A : f\ \overline{s_n} \approx t, G \Rightarrow_{[\mathrm{o}], h\ \overline{v_k} \to r'} f\ \overline{s_{n-1}} \approx h\ \overline{v_{k-1}}, s_n \approx v_k, r' \approx t, G \Rightarrow^*_\theta \square$$

where $k = \mathrm{ar}(h) > 0$. From the induction hypothesis for $A_{>1}$, there exists $B \in \mathcal{WF}$ of the form

$$B : G'_1 = \overline{s_m \approx u_m}, r\ \overline{s_{m+1,n-1}} \approx h\ \overline{v_{k-1}}, s_n \approx v_k, r' \approx t, G \Rightarrow^*_\theta \square$$

such that $|B| < |A_{>1}|$ and $\mathcal{C}(A_{>1}, B)$. From the validity of property $\mathcal{P}_{[\mathrm{o}]}(A_{>1})$ we deduce that [o] is never applied to the rhs of descendants of the equation $t_n \approx v_k$ in $A_{>1}$. From $\mathcal{C}(A_{>1}, B)$ we have that [o] is never applied to the rhs of the descendants of $t_n \approx v_k$ in B. Let $B_{>i}$ be the sub-refutation of B such that $B_{>i} : G_2 = (r\ \overline{s_{n-1}} \approx h\ \overline{v_{k-1}}, s_n \approx v_k, r' \approx t, G)\theta_1 \Rightarrow^j_{\theta_2} \square$. We construct the LNC-refutation:

$$\begin{aligned} A' :\ & G_1 = \overline{s_m \approx u_m}, r\ \overline{s_{m+1,n}} \approx t, G \\ & \Rightarrow^i_{\theta_1} G'_2 = (r\ \overline{s_{m+1,n}} \approx t, G)\theta_1 \\ & \Rightarrow_{[o], h\ \overline{v_k} \to r'} G_2 = (r\ \overline{s_{m+1,n-1}} \approx h\ \overline{v_{k-1}}, s_n \approx v_k, r' \approx t, G)\theta_1 \\ & \Rightarrow^j_{\theta_2} \square \end{aligned}$$

where the first i steps of A' coincide with those of B and $A'_{>(i+1)} = B_{>i}$.

Then $|A'| = |B|+1 < |A_{>1}|+1 = |A|$. From our previous remark that no [o]-steps are applied to the rhs of descendants of $t_n \approx v_k$ we deduce $A' \in \mathcal{WF}$. From the construction of A' and the fact that $\mathcal{C}(A_{>1}, B)$ holds we infer that $\mathcal{C}(A, A')$ holds too.

3. A starts with an [i]-step to the lhs. Then:

$$A : f\ \overline{s_n} \approx x, G \Rightarrow_{[\mathrm{i}],\sigma=\{x\mapsto x_1\ x_2\}} (f\ \overline{s_{n-1}} \approx x_1, s_n \approx x_2, G)\sigma \Rightarrow^*_{\theta'} \Box.$$

An application of the induction hypothesis to $A_{>1}$ reveals the existence of a $B \in \mathcal{WF}$ of the form:

$$B : (\overline{s_m \approx u_m}, r\ \overline{s_{m+1,n-1}} \approx x_1, s_n \approx x_2, G)\sigma \Rightarrow^*_{\theta'} \Box$$

such that $|B| < |A_{>1}|$ and $\mathcal{C}(A_{>1}, B)$. We distinguish two subcases:

(a) $x \in \mathcal{V}(f\ \overline{s_n})$. Then $x\sigma\theta'$ is normalized because $A \in \mathcal{WF}$. We can now apply Lemma 18 to B and obtain $A' \in \mathcal{WF}$ of the form:

$$A' : \overline{s_m \approx u_m}, r\ \overline{s_{m+1,n}} \approx x, G \Rightarrow_{\theta''} \Box$$

such that $\sigma\theta' = \theta''$ and $|A'| \leq |B| + 1$. But $\sigma\theta' = \theta$ and hence $\theta'' = \theta$. Also, $|A'| \leq |B| + 1 < |A_{>1}| + 1 = |A|$. From the construction of A' from B given in the proof of Lemma 18 results that if [o] is never applied to the rhs of descendants of $e \in G$ in B then [o] is never applied to the rhs of descendants of e in A'. This observation together with $\mathcal{C}(A_{>1}, B)$ implies $\mathcal{C}(A, A')$.

(b) $x \notin \mathcal{V}(f\ \overline{s_n})$. Then $x \notin \mathcal{V}(\overline{s_m \approx u_m}, r\ \overline{s_{m+1,n-1}}, s_n)$ and we can again apply Lemma 18 to construct from B the desired $A' \in \mathcal{WF}$ with property $\mathcal{C}(A, A')$. $\Box$

Lemma 20 *Let $A \in \mathcal{WF}$ be of the form*

$$A : G = x\ \overline{s_n} \approx t, G' \Rightarrow^*_\theta \Box \tag{4.2}$$

such that there exists a first [o]*-step of A which is applied to the lhs of an a-descendant of $x\ \overline{s_n} \approx t$ and all the [i]-steps which precede it are applied to the lhs. Then there exists a fresh variant $f\ \overline{u_k}\ \overline{v_m} \to r$ of a rewrite rule such that:*

(a) $0 < m \leq n$,

(b) *The last* [o]*-step to an a-descendant of* $x\ \overline{s_n} \approx t$ *is of the form*

$$\begin{array}{ll} A_1: & (x\ \overline{s_m} \approx t')\theta_1, G'' \\ & \Rightarrow_{[\mathrm{o}], f\ \overline{u_k}\ \overline{v_m} \to r} (f\ s_{m-1} \approx f\ \overline{u_k}\ \overline{v_{m-1}}, s_m \approx v_m, r\ \overline{s_{m+1,n}} \approx t')\theta_1, G'' \end{array}$$

(c) *There exists a* $A' \in \mathcal{WF}$ *of the form:*

$$A' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n}} \approx t, G')\sigma \Rightarrow^*_{\theta'} \Box \tag{4.3}$$

with $\sigma = \{x \mapsto f\ \overline{u_k}\}$ *such that* $\sigma\theta' = \theta$ *and* $|A'| < |A|$.

Proof. We notice that the a-descendants of parameter-passing equations are parameter-passing equations. Since the first [o]-step to an a-descendant of $x\ \overline{s_n} \approx t$ is applied to the lhs then, because of property $\mathcal{P}_{[\mathrm{o}]}(A)$, all the [o]-steps to a-descendants of $x\ \overline{s_n} \approx t$ are applied to the lhs. Also, because of property $\mathcal{P}_{[\mathrm{i}]}(A)$, all $[i]$ steps between the first and the last [o]-step to an a-descendant of $x\ \overline{s_n} \approx t$ are applied to the lhs. Therefore, the last a-descendant of $x\ \overline{s_n} \approx t$ to which an [o]-step is applied is of the form $\underline{(x\ \overline{s_m})\theta_1} \approx t'\theta_1$ with $m \leq n$.

We prove now that $m > 0$. Assume that $m = 0$. Then from the applicability of an [o]-step to the lhs of $(x\ \overline{s_m})\theta_1 \approx t'\theta_1$ we deduce that $x \in \mathcal{D}(\theta_1)$. Also, by Corollary 3, the term $x\theta_1$ is reducible. Since this contradicts property 1.(b) of well-formedness for A, we must have $m > 0$. From Lemma 15 we deduce that the variant of the rewrite rule employed in the [o]-step to the lhs of $(x\ \overline{s_m})\theta_1 \approx t'\theta_1$ can be written as $f\ \overline{u_k}\ \overline{v_m} \to r$. Thus, conditions (a) and (b) hold.

We prove now that condition (c) also holds. Consider the sub-refutation A''' of A that starts with an [o]-step applied to the lhs of the a-descendant $(x\ \overline{s_m} \approx t')\theta_1$ of $x\ \overline{s_n} \approx t$. This refutation starts from a goal of the form $G_2 = (\underline{x\ \overline{s_m}} \approx t')\theta_1, G''$. Since $A''' \in \mathcal{WF}$, it is of the form:

$$G_2 \Rightarrow_{[\mathrm{o}], f\ \overline{u_k}\ \overline{v_m} \to r} (x\theta_1)\ \overline{s'_{m-1}} \approx f\ \overline{u_k}\ \overline{v_{m-1}}, s'_m \approx v_m, r \approx t'\theta_1, G'' \Rightarrow^* \Box$$

where $s'_i = s_i\theta_1$ for $1 \leq i \leq m$. Since the first step of this sub-refutation is also the last one to an a-descendant of $x\ \overline{s_n} \approx t$, we deduce that the following $m-1$ steps must be [d]-steps. Therefore we can write:

$$\begin{array}{l} A : G \Rightarrow^*_{\theta_1} G_2 = (\underline{x\ \overline{s_m}} \approx t')\theta_1, G'' \\ \Rightarrow_{[\mathrm{o}], f\ \overline{u_k}\ \overline{v_m} \to r} \underline{(x\theta_1)\ \overline{s'_{m-1}}} \approx f\ \overline{u_k}\ \overline{v_{m-1}}, s'_m \approx v_m, r \approx t'\theta_1, G'' \\ \Rightarrow^{m-1}_{[\mathrm{d}]} x\theta_1 \approx f\ \overline{u_k}, \boxed{s'_1 \approx v_1}, \ldots, \boxed{s'_m \approx v_m}, r \approx t'\theta_1, G'' \Rightarrow^* \Box. \end{array} \tag{4.4}$$

The equations displayed within boxes are descendants of the parameter-passing equations generated from the equation $(x\ \overline{s_m} \approx t')\theta_1 \in G_2$. They are used in specifying the property $\mathcal{C}_2(A, A')$ defined below.

We now prove by induction on $|A|$ the existence of $A' \in \mathcal{WF}$. In the proof we make use of the property that the condition $\mathcal{C}(A, A') = \mathcal{C}_1(A, A') \wedge \mathcal{C}_2(A, A')$ holds in each induction step. Here, $\mathcal{C}_1(A, A')$ and $\mathcal{C}_2(A, A')$ are defined as follows:

Let A and A' be the LNC-refutations under consideration of the forms given in (4.2) and (4.3). Then:

$\mathcal{C}_1(A, A')$: for every $e' \in G'$, if [o] is never applied to the rhs of descendants of e' in A then [o] is never applied to the rhs of descendants of e' in A'

$\mathcal{C}_2(A, A')$: assuming A is written in the form (4.4) described above then for every $i \in \{1, \ldots, m\}$ the following implication holds: if [v] is the only LNC-step applied to a descendant of $s'_i \approx v_i$ in A then [v] is the only LNC-step applied to a descendant of $s_i \approx v_i$ in A'.

First we note that because of assumption (ii) A can not start with a [v]-step or a [t]-step. We distinguish three possibilities for the first LNC-step of A:
Case I. A starts with an [i]-step. Then $t \in \mathcal{V}$. Because of assumption (ii) we must have $n > 0$. In this case A can be written as:

$$A : G \Rightarrow_{[i],\sigma_0} (t\sigma_0)\ \overline{s''_{n-1}} \approx x_1, s''_n \approx x_2, G'\sigma_0 \Rightarrow^* \square$$

where $\sigma_0 = \{t \mapsto x_1\ x_2\}$ with $x_1, x_2 \in \mathcal{V}$ fresh variables, $s''_i = s_i\sigma_0$ for $1 \leq i \leq n$. We further distinguish two subcases:

1. $t = x$. Then A is of the form

$$A : G \Rightarrow_{[\mathrm{i}],\sigma_0} x_1\ x_2\ \overline{s''_{n-1}} \approx x_1, s''_n \approx x_2, G'\sigma_0 \Rightarrow^*_{\theta''} \square$$

where $\sigma_0\theta'' = \theta$, and the sub-refutation A''' of A can be written as:

$$\begin{aligned} A''' :\ & G_2 = (\underline{x_1\ x_2\ \overline{s_m}} \approx t')\theta_1, G'' \\ & \Rightarrow_{[\mathrm{o}], f\ \overline{u_k}\ \overline{v_m} \to r}\ (x_1\ x_2\ \overline{s_{m-1}} \approx f\ \overline{u_k}\ \overline{v_{m-1}}, \\ & \qquad s_m \approx v_m, r \approx t')\theta_1, G'' \Rightarrow^* \square. \end{aligned}$$

The equation $(x_1\ x_2\ \overline{s_m} \approx t')\theta_1$ is an a-descendant of $x_1\ x_2\ \overline{s''_{n-1}} \approx x_1$ obtained by applying a sequence of [d]-, [i]- and [o]-steps. Since [i]- and [o]-steps of this sequence are applied only to the lhs, we have $\mathcal{V}(x_1\ x_2\ \overline{s''_{n-1}} \approx x_1) \cap \mathcal{D}(\theta_1) = \{x_1\}$ and $\mathcal{V}(x_1\ x_2\ \overline{s''_{n-1}} \approx x_1) \cap \mathcal{I}(\theta_1) = \emptyset$. Therefore $x_2 \notin \mathcal{D}(\theta_1) \cup \mathcal{V}(x_1\theta_1)$. From Lemma 15 we deduce that

$k \geq 1$. Since $A''' \in \mathcal{WF}$, we can write it in the form:

$$\begin{array}{l} G_2 = (\underline{x_1\ x_2\ \overline{s_m}} \approx t')\theta_1, G'' \\ \Rightarrow_{[\mathrm{o}], f\ \overline{u_k}\ \overline{v_m}\to r}\ x_1\ x_2\ \overline{s'_{m-1}} \approx f\ \overline{u_k}\ \overline{v_{m-1}}, s'_m \approx v_m, r \approx t'\theta_1, G'' \\ \Rightarrow^m_{[\mathrm{d}]}\ \ x_1\theta_1 \approx f\ \overline{u_{k-1}}, \boxed{x_2 \approx u_k}, \boxed{s'_1 \approx v_1}, \ldots, \boxed{s'_m \approx v_m}, \\ \qquad r \approx t'\theta_1, G'' \Rightarrow^* \square \end{array}$$

where the equations displayed within boxes are descendants of the parameter-passing equations generated from the equation $(x_1\ x_2\ \overline{s_m} \approx t')\theta_1 \in G_2$. Since $x_2 \notin \mathcal{V}(x_1\theta_1 \approx f\ \overline{u_{k-1}})$, we can further write

$$\begin{array}{ll} A'''_{>m+1}: & x_1\theta_1 \approx f\ \overline{u_{k-1}}, (x_2 \approx u_k, \overline{s_m \approx v_m}, r \approx t')\theta_1, G'' \\ & \Rightarrow^*_{\theta_2} x_2 \approx u_k\theta_2, (\overline{s_m \approx v_m}, r \approx t')\theta_1\theta_2, G''\theta_2 \Rightarrow_{[\mathrm{v}]}\Rightarrow^* \square. \end{array}$$

We can now apply the the induction hypothesis to $A_{>1} \in \mathcal{WF}$ and deduce the existence of a well-formed LNC-refutation A'' of the form:

$$A'' : (x_2 \approx u_k, \overline{s_m \approx v_m}, r\ \overline{s_{m+1,n-1}} \approx x_1, s_n \approx x_2, G')\sigma_0\sigma_1 \Rightarrow^*_{\theta'''} \square$$

where $\sigma_1 = \{x_1 \mapsto f\ \overline{u_{k-1}}\}$, $|A''| < |A_{>1}|$, and $\mathcal{C}(A_{>1}, A'')$, and $\sigma_1\theta''' = \theta''$. From the assumption $\mathcal{C}_2(A_{>1}, A'')$ and the observation that [v] is the only inference step applied to an a-descendant of $x_2 \approx u_k$ in A we conclude that the first LNC-step to $(x_2 \approx u_k)\sigma_0\sigma_1 = x_2 \approx u_k$ must be a [v]-step. Hence:

$$A''_{>1} : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n-1}} \approx x_1, s_n \approx x_2, G')\sigma_0\sigma_1\{x_2 \mapsto u_k\} \Rightarrow^* \square.$$

Note that $\sigma_0\sigma_1\{x_2 \mapsto u_k\} = \{x \mapsto f\ \overline{u_k}, x_1 \mapsto f\ u_{k-1}, x_2 \mapsto u_k\}$. Because $\sigma = (\sigma_0\sigma_1\{x_2 \mapsto u_k\})\restriction_{\mathcal{V}(\overline{s_m\approx v_m}, r\ \overline{s_{m+1,n-1}}, G')}$, we can write:

$$\begin{array}{ll} A''_{>1}: & (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n-1}} \approx f\ \overline{u_{k-1}}, s_n \approx u_k, G')\sigma \\ & \Rightarrow^i_{\tau_1} (r\ \overline{s_{m+1,n-1}} \approx f\ \overline{u_{k-1}}, s_n \approx u_k, G')\sigma\tau_1 \Rightarrow^*_{\tau_2} \square. \end{array}$$

We perform the following construction of A' from $A''_{>1}$:

$$\begin{array}{ll} A': & (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n}} \approx x, G')\sigma \Rightarrow^i_{\tau_1} (r\ t_n \approx f\ \overline{u_k}, G')\sigma\tau_1 \\ & \Rightarrow_{[\mathrm{d}]} (r\ \overline{s_{m+1,n-1}} \approx f\ \overline{u_{k-1}}, s_n \approx u_k, G')\sigma\tau_1 \Rightarrow^*_{\tau_2} \square \end{array}$$

where the first i LNC-steps of A' coincide with the first i LNC-steps of $A''_{>1}$ and $A'_{>i+1} = A''_{>i+1}$. Then A' is well-formed and satisfies the requirements of our lemma. The validity of $\mathcal{C}(A, A')$ results from the way in which A' is constructed from A'' and from the property $\mathcal{C}(A_{>1}, A'')$.

2. $t \neq x$. Then A is of the form

$$A : G \Rightarrow_{[\mathrm{i}],\sigma_0} x\ \overline{s''_{n-1}} \approx x_1, s''_n \approx x_2, G'\sigma_0 \Rightarrow^*_{\theta''} \Box$$

where $\sigma_0\theta'' = \theta$. By the induction hypothesis for $A_{>1}$ there exists a $A'' \in \mathcal{WF}$ of the form:

$$A'' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n-1}} \approx x_1, s_n \approx x_2, G')\sigma_0\sigma \Rightarrow^*_{\theta_0} \Box$$

such that $\sigma\theta_0 = \theta''$, $|A''| < |A_{>1}|$ and $\mathcal{C}(A_{>1}, A'')$. Since $\sigma_0\sigma = \sigma\sigma_0$, we can write A'' in the form:

$$A'' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n-1}} \approx x_1, s_n \approx x_2, G')\sigma\sigma_0 \Rightarrow^*_{\theta_0} \Box.$$

An application of Lemma 18 to A'' yields the desired $A' \in \mathcal{WF}$.

Case II. A starts with an [o]-step. We distinguish two cases:

- $n = m$. Then this step is also the last [o]-step to an a-descendant of $x\ \overline{s_m} \approx t$. We have then:

$$\begin{array}{ll} A : x\ \overline{s_m} \approx t, G' & \\ \Rightarrow_{[\mathrm{o}], f\ \overline{u_k}\ \overline{v_m} \to r} & x\ \overline{s_{m-1}} \approx f\ \overline{u_k}\ \overline{v_{m-1}}, s_m \approx v_m, r \approx t, G' \\ \Rightarrow^{m-1}_{[\mathrm{d}]} & x \approx f\ \overline{u_k}, \overline{s_m \approx v_m}, r \approx t, G' \\ \Rightarrow_{[\mathrm{v}],\sigma=\{x \mapsto f\ \overline{u_k}\}} & (\overline{s_m \approx v_m}, r \approx t, G')\sigma \Rightarrow^*_{\theta'} \Box \end{array}$$

  and we can choose $A' = A_{>m+1}$.

- $n > m$. In this case we have:

$$A : x\ \overline{s_n} \approx t, G' \Rightarrow_{[o], l_1\ l_2 \to r'} x\ \overline{s_{n-1}} \approx l_1, s_n \approx l_2, r' \approx t, G' \Rightarrow^*_{\sigma'} \Box.$$

  By the induction hypothesis for $A_{>1}$ we infer the existence of $A'' \in \mathcal{WF}$ of the form:

$$A'' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n-1}} \approx l_1, s_n \approx l_2, r' \approx t, G')\sigma \Rightarrow^*_{\theta'} \Box$$

  such that $\sigma\theta' = \theta$, $|A''| < |A_{>1}|$ and $\mathcal{C}(A_{>1}, A'')$. Let $i_1 \geq 0$ such that:

$$\begin{array}{l} A'' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n-1}} \approx l_1, s_n \approx l_2, r' \approx t, G')\sigma \\ \Rightarrow^{i_1}_{\sigma'} (r\ \overline{s_{m+1,n-1}} \approx l_1, t_n \approx l_2, r' \approx t, G')\sigma\sigma' \Rightarrow^* \Box. \end{array}$$

  We construct:

$$\begin{array}{l} A' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n}} \approx t, G')\sigma \Rightarrow^{i_1}_{\sigma'} (r\ \overline{s_{m+1,n}} \approx t, G')\sigma\sigma' \\ \Rightarrow_{[\mathrm{o}], l_1\ l_2 \to r'} (r\ \overline{s_{m+1,n-1}} \approx l_1, s_n \approx l_2, r' \approx t, G')\sigma\sigma' \Rightarrow^* \Box \end{array}$$

such that the first i_1 steps of A' coincide with the first i_1 steps of A'' and $A'_{>i_1+1} = A''_{>i_1}$. We notice that $|A'| = |A''| + 1 < |A_{>1}| + 1 = |A|$. We must show that $A' \in \mathcal{WF}$. Because $A'' \in \mathcal{WF}$, we only have to show that in A' there are no [o]-steps applied to the rhs of descendants of the parameter-passing equation $(t_n \approx l_2)\sigma\sigma'$. We note that $(s_n \approx l_2)\sigma$ is a parameter-passing equation in $A_{>1}$ and therefore [o]-steps are never applied to the rhs of the descendants of $(t_n \approx l_2)\sigma$. From $\mathcal{C}(A_{>1}, A'')$ we infer that in A'' [o]-steps are never applied to the rhs of the descendants of $(s_n \approx l_2)\sigma$. From the construction of A' it is easily seen that also in A' [o] is never applied to the rhs of $(s_n \approx l_2)\sigma\sigma'$.

The validity of $\mathcal{C}(A, A')$ results from $\mathcal{C}(A_{>1}, A'')$ and the construction of A' from A''.

Case III. A starts with a [d]-step. Then $n > 0$, $t = t_1\ t_2$ for some terms t_1, t_2 and:

$$A : x\ s_m \approx t_1\ t_2, G' \Rightarrow_{[\mathrm{d}]} x\ \overline{s_{n-1}} \approx t_1, s_n \approx t_2, G' \Rightarrow^*_\theta \Box$$

and we can apply the induction hypothesis to $A_{>1}$ and obtain a well-formed LNC-refutation:

$$A'' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n-1}} \approx t_1, s_n \approx t_2, G')\sigma \Rightarrow_{\theta'} \Box$$

with $\sigma\theta' = \theta$, $|A'| < |A_{>1}|$ and $\mathcal{C}(A_{>1}, A'')$. Let $i_1 \geq 0$ such that:

$$\begin{aligned} &A'' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n-1}} \approx t_1, s_n \approx t_2, r' \approx t, G')\sigma \\ &\Rightarrow^{i_1} (r\ \overline{s_{n-1}} \approx t_1, s_n \approx t_2, G')\sigma\sigma' \Rightarrow^* \Box. \end{aligned}$$

Then we define A' as follows:

$$\begin{aligned} &A' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n}} \approx t, G')\sigma \Rightarrow^{i_1}_{\sigma'} (r\ \overline{s_{m+1,n}} \approx t_1\ t_2, G')\sigma\sigma' \\ &\Rightarrow_{[\mathrm{d}]} (r\ \overline{s_{m+1,n-1}} \approx t_1, s_n \approx t_2, G')\sigma\sigma' \Rightarrow^* \Box \end{aligned}$$

where the first i_1 steps of A' coincide with the first i_1 steps of A'' and $A'_{>i_1+1} = A''_{>i_1}$. Then $A' \in \mathcal{WF}$ and $|A'| = |A''| + 1 < |A_{>1}| + 1 = |A|$. Also, property $\mathcal{C}(A, A')$ follows from $\mathcal{C}(A_{>1}, A'')$ and the construction of A' from A''. $\Box$

4.4.5 The Completeness Theorem

Lemma 21 *Let $\mathcal{R}$ be a confluent ATRS and G be a goal. For every well-formed LNC-refutation $A : G \Rightarrow^*_\theta \Box$ there exists an LNCA-derivation B : $G \Rrightarrow^*_\sigma G_1$ and a well-formed LNC-refutation $A' : G_1 \Rightarrow^*_{\theta'} \Box$ such that $\sigma\theta' = \theta\ [\mathcal{V}(G)]$ and $|A'| < |A|$.*

Proof. Let $G = s \approx t, G'$. We distinguish the following cases:

(1) No [o]-steps are applied to a-descendants of $s \approx t$ in A. We have to consider the following cases:

(1a) $s = f\ \overline{s_m}$ and $t = g\ \overline{t_n}$. According to Lemma 9, we must have $f = g$ and $n = m$. According to Lemma 10, there exists a $A' \in \mathcal{WF}$ of the form $A' : \overline{s_n \approx t_n}, G' \Rightarrow_\theta^* \square$ such that $|A'| < |A|$. Then, for:

$$B : G = \underline{f\ s_n \approx f\ t_n}, G' \Rightarrow_{[\mathrm{df}]} G_1$$
$$G_1 = \overline{s_n \approx t_n}, G',$$
$$\theta' = \theta,\ \sigma = \varepsilon$$

the conclusion of Lemma 21 holds.

(1b) $s \approx t$ is of the form $a\ \overline{s_m} \simeq x\ \overline{t_n}$ with $m \geq n$. This case is covered by Lemma 17.

(1c) Otherwise, $s \approx t$ must be of the form $f\ \overline{s_m} \simeq x\ \overline{u_n}$ with $m < n$. According to Lemma 11, there exists an [o]-step in A which is applied to an a-descendant of $s \approx t$. Since we assumed the contrary, this case is impossible.

(2) The first [o]-step is applied to the lhs of an a-descendant of $s \approx t$. By Lemma 8, there exists $A'' \in \{A, \phi_{\mathrm{swap}}(A, 1)\}$ such that all [i]-steps before the first [o]-step are applied to the lhs and the first [o]-step is applied to the lhs. According to Lemma 7, $A'' \in \mathcal{WF}$. Assume:

$$A'' : \underline{a\ \overline{s_n}} \approx r', G' \Rightarrow_\theta^* \square$$

where $a \in \mathcal{F} \cup \mathcal{V}$. We distinguish two cases:

(a) $a = f \in \mathcal{F}$. Let $m = \mathrm{ar}(f)$. From Lemma 19 we infer the existence of a fresh variant $f\ \overline{u_m} \to r$ of a rewrite rule with $m \leq n$ and of an $A' \in \mathcal{WF}$ of the form

$$A' : \overline{s_m \approx u_m}, r\ \overline{s_{m+1,n}} \approx t, G' \Rightarrow_\theta^* \square$$

such that $|A'| < |A|$. We note that we can choose B to be:

$$B : f\ \overline{s_n} \simeq r', G' \Rightarrow_{[\mathrm{of}], f\ \overline{u_m} \to r} \overline{s_m \approx u_m}, r\ \overline{s_{m+1,n}} \approx r', G'$$

(b) $a = x \in \mathcal{V}$. By Lemma 20 we can assume the existence of a fresh variant $f\ \overline{u_k}\ \overline{v_m} \to r$ of a rewrite rule such that $0 < m \leq n$ and of an $A' \in \mathcal{WF}$ of the form:

$$A' : (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n}} \approx t, G')\sigma \Rightarrow_{\theta'}^* \square$$

with $\sigma = \{x \mapsto f\ \overline{u_k}\}$ such that $\sigma\theta' = \theta$ and $|A'| < |A|$. We can now consider the $[ov]$-step of LNCA:

$$\begin{aligned} B:\quad & G = (x\ \overline{s_n} \simeq r', G') \\ & \Rrightarrow_{[ov],\sigma,f\ \overline{u_k}\ \overline{v_m}\to r} G_1 = (\overline{s_m \approx v_m}, r\ \overline{s_{m+1,n}} \approx r', G')\sigma. \end{aligned}$$

(3) The first [o]-step is applied to the rhs of an a-descendant of $s \approx t$. Then obviously the first [o]-step is not preceded by [i]-steps, and in $\phi_{\text{swap}}(A,1)$ the first [o]-step is applied to the lhs. This case reduces to case (2). $\Box$

Theorem 4 (completeness theorem) *Let $\mathcal{R}$ be a confluent ATRS and G a goal. For every normalized solution θ of G there exists a successful LNCA-derivation $A : G \Rrightarrow^*_{\theta'} \Box$ such that $\theta' \leq \theta$ $[\mathcal{V}(G)]$.*

Proof. By Corollary 2 and induction on $|A|$ using Lemma 21. $\Box$

4.5 Conclusion

The completeness proof of LNCA for confluent ATRSs is by induction on the length of well-formed LNC-refutations: any well-formed LNC-refutation can be replaced with a sequence of LNCA-steps followed by a shorter well-formed LNCA-refutation, such that the computed answer is the same. Schematically, the lifting process of a well-formed LNC-refutation into an LNCA-refutation is depicted in the figure below. Notice that the length of

situation of Lemma 21	well-formed LNC-refutation $G = s \approx t, G' \Rightarrow^N \Box$	partially lifted refutation $G \Rrightarrow^* G'' \Rightarrow^* \Box$
(1a)	$f\ \overline{s_n} \approx f\ \overline{t_n}, G' \Rightarrow^N \Box$	$G \Rrightarrow_{[\mathrm{df}]} \overline{s_n \approx t_n}, G' \Rightarrow^{N-1} \Box$
(1b)	$x\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}, G' \Rightarrow^N \Box$	$G \Rrightarrow_{[\mathrm{dv}]} G'' \Rightarrow^{N-1-n} \Box$
	$a\ \overline{s_m}\ \overline{t_n} \simeq x\ \overline{u_n}, G' \Rightarrow^N \Box$ with $a \neq x$	$G \Rrightarrow_{[\mathrm{V}]} G'' \Rightarrow^{N-1} \Box$ or $G \Rrightarrow_{[\mathrm{I}]}\Rrightarrow^{m-i}_{[\mathrm{V}]} G'' \Rightarrow^{N-i-1} \Box$
(2a)	$f\ \overline{s_n} \simeq r', G' \Rightarrow^N \Box$	$G \Rrightarrow_{[\mathrm{of}]} G'' \Rightarrow^{<N-1} \Box$
(2b)	$x\ \overline{s_n} \simeq r', G' \Rightarrow^N \Box$	$G \Rrightarrow_{[\mathrm{of}]} G'' \Rightarrow^{<N-1} \Box$
(3)	similar to cases (2a), (2b)	

Fig. 4.1: Lifting of a well-formed LNC-refutation to an LNCA-refutation

the LNCA-refutation corresponding to a well-formed LNC-refutation may be longer (case (1b) of Lemma 21). This means that LNC may sometimes be more efficient than LNCA if we compare them in terms of refutation length. Intuitively, the advantage of LNCA versus LNC is the reduction of the width of the search space of solutions because of the inference rule [of] of LNCA which is a specialization of the [o]-inference rule of LNC.

A possible direction of further research is to analyze how the deterministic refinements of LNC to LNC_d can be carried over to LNCA. By following a similar approach we expect to find a deterministic calculus for applicative term rewriting systems.

Chapter 5

Lazy Narrowing for Higher-Order Pattern Rewrite Systems

In this chapter we study the possibility to extend the equational reasoning capabilities of lazy narrowing to languages of simply-typed λ-terms. The expressive power of the higher-order constructs available in λ-calculus is widely recognized by the equational logic community. In particular, higher-order term rewriting supports symbolic computation with complex structures, and the underlying kernel of the language of computer algebra *Mathematica*™ [Wol96] is based on a particular version of higher-order rewriting. In [Pre98], Prehofer develops a powerful higher-order lazy narrowing calculus LN which is based on the notion of higher-order term rewriting introduced by Nipkow [Nip91, Nip93, NP98]. For alternative notions of higher-order term rewriting we refer the reader to [Klo90, Klo92, vO94, Wol93]. The advantage of adopting Nipkow's rewrite relation instead of the others proposed so far stems from its similarity with first-order term rewriting, but still it is quite general. Because of this fact, several notions and results from first-order term rewriting and narrowing can be lifted to the higher-order case.

Motivation

Our experience with the with the first-order lazy narrowing calculi proposed by the members of the SCORE group during recent years [NI95, Suz95, MOI96, MO98] helped us to detect many similarities between first-order lazy narrowing calculus LNC and LN even though they are based on different theoretical frameworks. Therefore we considered worthwhile to try to extend to LN the deterministic refinements of the calculus LNC which make it suitable for practical applications.

Another reason for our research was that this area of research is still at its infancy. The practical utility of higher-order constructs in declarative programming has already been explored in logic programming languages such as λ-Prolog [MN86], but useful and powerful functional logic programming languages are still missing. This is mainly so because of the lack of an efficient operational principle as it is lazy narrowing in the first-order logic case. Proposals for higher-order lazy narrowing calculi have started to appear [Pre98, SNI97] but unfortunately they are of little practical interest either because they are too restricted or too nondeterministic.

Our investigation draws on two sources: the calculus LN, and the deterministic refinements of the first-order lazy narrowing calculus LNC. The outcome is a series of lazy narrowing calculi equipped with suitable equation selection strategies, each of them tailored for a particular class of higher-order term rewriting systems and satisfying certain completeness properties. These calculi are more deterministic than LN, and thus more suitable for solving complex problems. Most of the calculi proposed by us are the outcome of our attempt to lift to LN the deterministic refinements of the calculus LNC.

Before setting up the theoretical framework of our work, we mention some of the main reasons why the higher-order setting of lazy narrowing more difficult than first-order logic:

- The terms of the language are simply typed λ-terms, which are identified modulo the α, β and η conversion rules of the λ-calculus.
- The narrowing and rewriting operations require a correct treatment of the locally bound variables,
- Both pattern-matching and unification of simply-typed λ-terms are highly intractable operations.

To overcome the complications mentioned above, we adopt the following restrictions:

1. The terms of our language are represented in long $\beta\eta$-normal form.

   Motivation: the long $\beta\eta$-normal form representation is convenient because of the following well known theoretical result [HS86]: two simply-typed λ-terms are $\alpha\beta\eta$-convertible iff their long $\beta\eta$-normal forms are α-convertible. Another important theoretical result is that

α-convertible λ-terms can be identified at the syntactical level [dB72]. Thus, this representation of simply typed λ-terms simplifies the test of $\alpha\beta\eta$-convertibility.

2. The equational theories are axiomatized with confluent pattern rewrite systems, i.e. sets of rewrite rules between λ-terms of base type whose left-hand sides are patterns.

   Motivation: patterns [Mil91] are elements of a restricted class of simply-typed λ-terms which has the following important property: pattern unification is unitary. As a consequence, matching with a pattern is unitary. This property simplifies the formal treatment of term rewriting [Nip91, Nip93], making it more similar to the first-order case. The restriction to rules of base type is motivated by the Nipkow's observation [Nip91] that term rewriting with rules of non-base type becomes problematic if we want to work with terms in β-normal form. To illustrate, consider the rewrite rule of non-base type $\lambda x.f(G(x), x) \to \lambda x.g(x)$ and the λ-term $t = f(g(a), a)$. The λ-term t is β-equivalent to $(\lambda x.f((g(x), x)))(a)$. If β-reduction is not implicit then we can rewrite t to $(\lambda x.g(x))(a)$ which is equivalent by β-reduction to $g(a)$. However, the representation of terms in $\beta\eta$-normal form prohibits the reduction of t. These problematic situation is eliminated if we restrict the rewrite rules to be of base type.

Structure of the chapter

The rest of this chapter is structured as follows. In Sect. 5.1 we introduce the main concepts, notions and theoretical results that are relevant to our study: the algebra of simply-typed λ-terms, the notions of preunification and higher-order term rewriting, and the equational logic which supports reasoning in theories axiomatized with pattern rewrite systems. A brief account to the main properties of the lazy narrowing calculi for pattern rewrite systems is given in Sect. 5.2. In Sect. 5.3 we introduce $\mathrm{LN}_{\mathrm{ff}}$, our first calculus for pattern rewrite systems, and analyze its main properties. In Sect. 5.4 we introduce the refinement LN_1 which addresses the most critical source of nondeterminism of the calculus $\mathrm{LN}_{\mathrm{ff}}$: outermost narrowing at variable position. This refinement is realized by adopting a suitable equation selection strategy and a criterion which eliminates the application of outermost narrowing at variable position for certain equations, in a way that makes LN_1 sound and complete. The following sections describe deterministic refinements of the calculus $\mathrm{LN}_{\mathrm{ff}}$. In Sect. 5.5 we address the higher-order counterpart of the eager variable elimination problem, and define a method to reduce the nondeterminism of solving certain parameter-passing equations without losing completeness of LN_1. Starting from Sect. 5.7 we define deterministic refinements of the calculus LN_1 for left-linear pattern rewrite systems. In Sect. 5.7 we define a criterion to detect and eliminate equations which do not contribute to the computation of a solution and propose the calculus LN_2 with strategy $\mathcal{S}_c$ to realize it. Lazy narrowing with constructor PRSs is analyzed in Sect. 5.8: the outcome is the calculus LN_3 with strategy $\mathcal{S}_c$, which is a deterministic refinement of LN_2 with strategy $\mathcal{S}_c$. Equations with strict semantics are introduced in Sect. 5.9 and a suitable subcalculus is defined for solving them. An approach to extend the calculus LN_{ff} to conditional rewriting is outlined in Sect. 5.10. Finally, in Sect. 5.11 we draw some conclusions and outline directions of further research.

The results presented in this chapter are further developments of the results reported in [MIS99c].

5.1 Preliminaries

This section introduces basic definitions and results related to higher-order equational reasoning.

5.1.1 The Language

In this subsection we introduce the language of simply-typed λ-terms.

We assume given a set S_0 of sorts called *base types*. The set S of types is defined as the inductive closure of S_0 under $\{\rightarrow\}$ where $\rightarrow$ is the function type constructor. We assume that $\rightarrow$ is right associative and that $\tau, \tau', \tau_1, \tau_2, \ldots$ range over S.

A *simply-typed signature* is a tuple $\Sigma = \langle S_0, \mathcal{F}\rangle$ where S_0 is a set of base types and $\mathcal{F} = \{\mathcal{F}_\tau\}_{\tau \in S}$ is an S-sorted set of symbols such that $\mathcal{F}_\tau \neq \emptyset$ for all $\tau \in S$. We assume given an S-sorted set $\{\mathcal{V}_\tau\}_{\tau\in S}$ with $\mathcal{V}_\tau$ countably infinite for any $\tau \in S$. The elements of $\mathcal{V}$ are called *variables*. The sets $\mathcal{F}$ and $\mathcal{V}$ are assumed to satisfy the conditions $\mathcal{F}_\tau \cap \mathcal{F}_{\tau'} = \mathcal{V}_\tau \cap \mathcal{V}'_\tau = \mathcal{F}_\tau \cap \mathcal{V}_\tau = \mathcal{F}_\tau \cap \mathcal{V}_{\tau'} = \emptyset$ for any $\tau \neq \tau'$. We define $opns(\Sigma) := \mathcal{F}$ and $sorts(\Sigma) := S$.

The following definition introduces the class of simply-typed λ-terms and their associated types.

Definition 28 (simply typed λ-term) *A* type judgement *stating that a λ-term t is of type τ is written as $t : \tau$. The following inference rules inductively define the set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of* simply typed λ-terms *and their associated type:*

$$\frac{x \in \mathcal{V}_\tau}{x : \tau} \qquad \frac{f \in \mathcal{F}_\tau}{f : \tau} \qquad \frac{s : \tau_1 \rightarrow \tau_2 \qquad t : \tau_1}{(s\ t) : \tau_2} \qquad \frac{x : \tau_1 \qquad s : \tau_2}{(\lambda x.s) : \tau_1 \rightarrow \tau_2}.$$

We denote by $\mathcal{T}(\mathcal{F},\mathcal{V})$ the class of simply typed λ-terms, by type(t) the type of a simply typed λ-term t, and write $t : \overline{\tau_n} \to \tau$ whenever t is a simply-typed λ-term with type $\tau_1 \to \ldots \to \tau_n \to \tau$.

In the sequel by term we understand a simply-typed λ-term and adopt the following naming conventions:

- X, Y, Z, H, possibly primed and/or with a subscript, are free variables,
- x, y, z, possibly primed and/or with a subscript, are bound variables,
- f, g are function symbols,
- h denotes a free variable or function symbol,
- v denotes a bound variable or function symbol,
- s, t, l, r, possibly primed and/or with subscript, are terms.

Given two λ-terms s, t and a variable x, we denote the *abstraction of s over x* by $\lambda x.s$, and the *application of s on t* by $(s\ t)$. We will use n-fold abstraction and application, written as $\lambda\overline{x_n}.s$ for $\lambda x_1 \ldots x_n.s$ and $s(\overline{t_n})$ for $((\ldots(s\ t_1)\ldots)\ t_n)$ respectively. Free and bound variables of a term t will be denoted by $\mathcal{V}(t)$ and $\mathcal{BV}(t)$, respectively.

The conversion rules in λ-calculus are:

$$R_\alpha : \lambda x.t \to \lambda y.(t\{x \mapsto y\}) \text{ if } y \in \mathcal{V} - \mathcal{V}(t) \qquad (\alpha\text{-conversion})$$

$$R_\beta : (\lambda x.s)\ t \to s\{x \mapsto t\} \qquad (\beta\text{-conversion})$$

$$R_\eta : \lambda x.(t\ x) \to_\eta t \text{ if } x \in \mathcal{V} - \mathcal{V}(t) \qquad (\eta\text{-conversion})$$

A simply typed λ-term of the form $(\lambda x.s)\ t$ is called *β-redex.* We denote by $=_\alpha$ the congruence relation induced by the α-conversion rule on $\mathcal{T}(\mathcal{F},\mathcal{V})$. In the sequel we will identify syntactically two α-convertible λ-terms[1], and thus write $s = t$ whenever $s =_\alpha t$.

In the sequel we denote by $\to_\phi$ the reduction relation induced by R_ϕ on $\mathcal{T}(\mathcal{F},\mathcal{V})$. ($\phi \in \{\alpha,\beta,\eta\}$.) The *$\beta\eta$-reduction relation* $\to_{\beta\eta}$ is defined by $s \to_{\beta\eta} t$ if $s \to_\beta t$ or $s \to_\eta t$. For every w-reduction relation ($w \in \{\alpha,\beta,\eta\}^*$) we define the corresponding w-conversion relation $=_w$ as the reflexive, symmetric and transitive closure of $\to_w$. It is well known [Bar84] that the reduction relations $\to_\beta$, $\to_\eta$, and $\to_{\beta\eta}$ are confluent and terminating. We denote by $t{\downarrow_\beta}$, $t{\downarrow_\eta}$ and $t{\downarrow_{\beta\eta}}$ the corresponding normal forms of a term t.

[1]The identification of two α-convertible λ-terms can be achieved by comparing their de Bruijn [dB72] encodings.

A term t in β-normal form can be uniquely written as $\lambda\overline{x_n}.h(\overline{t_n})$ where $h \in \mathcal{V} \cup \mathcal{F}$. We call h the *root of* t and denote it by $\mathrm{root}(t)$. The application of a ϕ-conversion rule ($\phi \in \{\beta, \eta\}$) in the other direction is called *ϕ-expansion.*

Definition 29 (η-expanded form) *Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{V}) = \lambda\overline{x_n}.v(\overline{s_m})$ be a β-normal form, where $v = \mathrm{root}(t)$. The η-expanded form of t, written $t{\uparrow_\eta}$, is defined by*

$$t{\uparrow_\eta} := \lambda\overline{x_{n+k}}.v(\overline{s_m{\uparrow_\eta}}, x_{n+1}{\uparrow_\eta}, \ldots, x_{n+k}{\uparrow_\eta})$$

where $t : \overline{\tau_{n+k}} \to \tau$ and $x_{n+1}, \ldots, x_{n+k}$ are fresh variables.

Given a term t, we call $t{\downarrow_\beta}{\uparrow_\eta}$ the *long $\beta\eta$-normal form* of t, also written as $t{\updownarrow_\beta^\eta}$. It is well known [HS86] that $s =_{\alpha\beta\eta} t$ iff $s{\updownarrow_\beta^\eta} =_\alpha t{\updownarrow_\beta^\eta}$.

We will in general assume that terms are in long $\beta\eta$-normal form. For brevity, we write variables in η-normal form, e.g. write X instead of $\lambda\overline{x_k}.X(\overline{x_k})$ whenever $X \in \mathcal{V} \setminus \{x_1, \ldots, x_k\}$. We assume that the transformation of a λ-term t into long $\beta\eta$-normal form is an implicit operation.

A λ-term t is *flex* if $\mathrm{root}(t) \in \mathcal{V}(t)$. A *rigid* λ-term is a λ-term which is not flex. A *flex/flex* equation is an equation between flex terms. The notions of *flex/rigid*, *rigid/flex* and *rigid/rigid* equation are defined similarly.

Like in the first-order case, the basic operations on λ-terms are substitution and replacement. Similar to the first-order case, we denote by $\mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ the set of substitutions from $\mathcal{V}$ to $\mathcal{T}_\lambda(\Sigma, \mathcal{V})$.

The result $t\theta$ of applying a substitution θ on a term t is defined as follows:

- $X\theta = \theta(X)$ if $X \in \mathcal{V}$,
- $f\theta = f$ if $f \in opns(\Sigma)$,
- $(s\ t)\theta = (s\theta\ t\theta)$,
- $(\lambda x.s)\theta = \lambda x.(s\theta\restriction_{\mathcal{D}(\theta)-\{x\}})$.

Definition 30 (position) *The set $\mathcal{P}os(t)$ of* positions in a λ-term t *is the set of sequences of natural numbers defined inductively as follows:*

$$\mathcal{P}os(t) := \begin{cases} \{\epsilon\} & \text{if } t \in \mathcal{V}, \\ \{\epsilon\} \cup \bigcup_{i=1}^{n} i{\cdot}q \mid q \in \mathcal{P}os(t_i)\} & \text{if } t = v(\overline{t_n}), \\ \{\epsilon\} \cup \{1{\cdot}p \mid p \in \mathcal{P}os(\lambda\overline{x_{2,n}}.s)\} & \text{if } t = \lambda\overline{x_n}.s. \end{cases}$$

Definition 31 (subterm) *Let t be a λ-term and $p \in \mathcal{P}os(t)$. The subterm of t at position p, denoted by $t|_p$, is defined by:*

$$t|_p := \begin{cases} t & \textit{if } t = \varepsilon, \\ t_i|_q & \textit{if } t = v(\overline{t_n}) \textit{ and } p = i{\cdot}q, \\ (\lambda\overline{x_{2,n}}.s)|_q & \textit{if } t = \lambda\overline{x_n}.s \textit{ and } p = 1{\cdot}q. \end{cases}$$

If p is a position in a term t then $\mathcal{BV}(t,p)$ denotes the set of all λ-abstracted variables on the path to p in s.

Example 4 *If $t = \lambda x_1, x_2.f(\lambda x_3.g(x_1,x_3), b)$ and $p = 111 \in \mathcal{P}os(t)$ then $t|_p = \lambda x_3.g(x_1,x_3)$ and $\mathcal{BV}(t,p) = \{x_1, x_2\}$.*

All the other definitions and notations pertaining to substitution, position, replacement, equational theory, and provability are carried over from the first-order case to the higher-order case.

5.1.2 Higher-order Unification

Similar to first-order unification, higher-order unification is concerned with solving problems of the following type: given a goal $\overline{s_n \approx t_n}$, find a representation of the set of unifiers

$$\mathcal{U}(G) := \{\theta \in \mathcal{S}ubst(\mathcal{F},\mathcal{V}) \mid \forall i \in \{1,\ldots,n\}.s_i\theta = t_i\theta\}.$$

Whereas in the first order case there exists a most general unifier $\theta = mgu(G)$, i.e. a substitution $\theta \in \mathcal{U}(G)$ such that $\theta \leq^{\mathcal{V}(G)} \gamma$ for all $\gamma \in \mathcal{U}(G)$, the situation is different in the higher-order case. Huèt has shown [Huè76] that there exist higher-order equations $s \approx t \in \mathcal{E}q(\mathcal{F},\mathcal{V})$ which do not have most general unifier: there may exist an infinite chain of unifiers $\theta_1, \ldots, \theta_n \in \mathcal{U}(s \approx t)$ such that

$$\theta_1 >^{\mathcal{V}(G)} \theta_2 >^{\mathcal{V}(G)} \theta_3 >^{\mathcal{V}(G)} \ldots$$

For such goals it is not possible to specify a complete set of unifiers of G, i.e. a set $c\mathcal{U}(G) \subseteq \mathcal{U}(G)$ such that

$$\forall\gamma \in \mathcal{U}(G)\ \exists\theta \in c\mathcal{U}(G).\theta \leq^{\mathcal{V}(G)} \gamma.$$

A practical way to remedy this situation is to weaken the requirement of an unification calculus to compute a complete set of unifiers, and to ask for computations of so-called *preunifiers*.

The main idea of preunification is to avoid solving flex/flex equations, since an attempt to solve them may never end.

[t] *Deletion*

$$\frac{G, t \approx t, G'}{G, G'}$$

[d] *Decomposition*

$$\frac{G, \lambda\overline{x}.v(\overline{s_n}) \approx \lambda\overline{x}.v(\overline{t_n}), G'}{G, \overline{\lambda\overline{x}.s_n \approx \lambda\overline{x}.t_n}, G'}$$

[v] *Variable elimination*

$$\frac{G, \lambda\overline{x}.X(\overline{x}) \simeq \lambda\overline{x}.t, G'}{(G, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{x}.t\}$ if $X \notin \mathcal{V}(\lambda\overline{x}.t)$

[i] *Imitation*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \simeq \lambda\overline{x}.f(\overline{t_m}), G'}{(G, \overline{\lambda\overline{x}.H_m(\overline{s_n}) \simeq \lambda\overline{x}.t_m}, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_n}.f(\overline{H_m(\overline{y_n})})\}$ and $\overline{H_m}$ are new variables of appropriate types

[p] *Projection*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \simeq \lambda\overline{x}.t), G'}{(G, \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}) \simeq \lambda\overline{x}.t), G')\theta}$$

where $1 \leq i \leq n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are new variables.

Fig. 5.1: System PT for higher-order preunification

We present in Fig. 5.1 a set of inference rules for preunification which is a version of the system PT for preunification proposed by Snyder and Gallier [SG89]. Note that the substitution θ involved in the inference rules [i] and [p] is not written in long $\beta\eta$-normal form. The long $\beta\eta$-normal form of the substitution computed for [i] can be written as

$$\{X \mapsto \lambda\overline{y_n}.f(\overline{\lambda\overline{z_{j_m}}.H_m(\overline{y_n}, \overline{z_{j_m}})})\},$$

and that of the substitution computed for [p] can be written as

$$\{X \mapsto \lambda\overline{y_n}.f(\overline{\lambda\overline{z_{j_p}}.H_m(\overline{y_n}, \overline{z_{j_p}})})\}.$$

The substitution computed upon an imitation step is called *imitation binding*, and the one computed upon a projection step is called *projection binding*. A *partial binding* is an imitation or a projection binding.

The notions of step and derivation for the calculus PT are defined similarly to LNC. A PT-refutation is a PT-derivation of the form $G \overset{\text{PT}}{\Longrightarrow}{}^*_\theta F$ where F is a flex/flex goal.

Definition 32 (preunifier) *A preunifier of a goal G is a substitution θ for which there exists a flex/flex goal F such that $\theta\gamma' \in \mathcal{U}(G)$ for any $\gamma' \in \mathcal{U}(F)$.*

The soundness and completeness properties of PT are stated below.

soundness If $G \overset{\text{PT}}{\Longrightarrow}{}^*_\theta F$ is a PT-refutation and $\gamma \in \mathcal{U}(F)$ then $\theta\gamma \in \mathcal{U}(G)$.

completeness If $\gamma \in \mathcal{U}(G)$ then there exist a PT-refutation $G \overset{\text{PT}}{\Longrightarrow}{}^*_\theta F$ with $\gamma = \theta\gamma'$ $[\mathcal{V}(G)]$ for some $\gamma' \in \mathcal{U}(F)$.

Thus, the set $Ans^{\text{PT}}(G) := \{\theta \mid G \overset{\text{PT}}{\Longrightarrow}{}^*_\theta F \text{ is a PT-refutation}\}$ is a set of preunifiers of G. An important theoretical result which is relevant in this thesis is the next lemma.

Lemma 22 (Lemma 4.1.6. in [Pre98]) *For any flex/rigid equation $e = \lambda\overline{x}.X(\overline{s_n}) \approx \lambda\overline{x}.t$ with $\theta \in \mathcal{U}(e)$ there exists a PT-step $e \overset{\text{PT}}{\Longrightarrow}_\alpha G'$ with $\alpha \in \{[\text{i}], [\text{p}]\}$ and $\gamma' \in \mathcal{U}(G')$ such that*

- $\mathcal{D}(\gamma') = (\mathcal{D}(\gamma) \setminus \{X\}) \cup \mathcal{R}ng(\theta)$,
- $X\gamma = X\theta\gamma'$, *and*
- $\gamma = \gamma'$ $[\mathcal{D}(\gamma) \setminus \{X\}]$.

We note that PT is an effective proof calculus which realizes the entailment $\emptyset \vdash_\Sigma \exists G$. Each PT-refutation $G \overset{\text{PT}}{\Longrightarrow}_{\theta_1} G_1 \ldots \overset{\text{PT}}{\Longrightarrow}_{\theta_n} G_n = F$ with F a flex/flex goal corresponds to a sequent proof:

$$\exists \bigwedge_{e\in G} e \overset{\text{PT}_s}{\Longrightarrow} \exists(\bigwedge_{e\in G_1} e) \overset{\text{PT}_s}{\Longrightarrow} \ldots \overset{\text{PT}_s}{\Longrightarrow} \exists(\bigwedge_{e\in F} e).$$

It is easy to see that $\emptyset \vdash_\Sigma \exists(\bigwedge_{e\in F} e)$ iff $\emptyset \vdash_\Sigma \exists \bigwedge_{e\in G} e$. This is a consequence of the following observations:

- $\emptyset \vdash_\Sigma \exists(\bigwedge_{e\in F} e)$ holds because F is flex/flex goal. Thus $\emptyset \vdash_\Sigma \exists \bigwedge_{e\in G} e \Rightarrow \emptyset \vdash_\Sigma \exists(\bigwedge_{e\in F} e \wedge [\theta])$.
- $\emptyset \vdash_\Sigma \exists(\bigwedge_{e\in F} e \Rightarrow \emptyset \vdash_\Sigma \exists \bigwedge_{e\in G} e$ (by soundness of PT).

Huèt [Huè76] has shown that the calculus PT is strongly complete, i.e. that completeness of PT does not depend on how equations are selected.

5.1.3 Higher-order Term Rewriting

Our definitions for higher-order TRSs are inspired from [Pre98]. In order to simplify the notion of rewriting, we will restrict the rewrite rules to be of base type only.

Definition 33 (general higher-order rewrite system) *A* rewrite rule *is a pair (l, r) of terms written as $l \to r$, such that*

- *l is not η-equivalent to a free variable,*
- *l and r are long $\beta\eta$-normal forms of the same base type, and*
- *$\mathcal{V}(r) \subseteq \mathcal{V}(l)$.*

A general higher-order rewrite system *(GHRS) is a set of rewrite rules.*

Since we work with terms in long $\beta\eta$-normal form only, we can always write a rewrite rule in the form $f(\overline{l_n}) \to r$ where $f \in \mathcal{F}$. This allows us to distinguish between defined symbols and constructors in the same way as in first-order term rewriting.

A generalization of the first-order term rewriting relation must take into consideration the presence of bound variables in the subterms that are rewritten. Note that a subterm $s|_p$ may contain free variables which were bound in s. To get a formal handle of these variables, we introduce the notion of *lifter*.

Definition 34 (lifter) *Let t be a term and $W \subset \mathcal{V}$. An $\overline{x_k}$-*lifter *of t away from W is a substitution $\theta = \{X \mapsto \rho(X)(\overline{x_k}) \mid X \in \mathcal{V}(t)\}$ where ρ is a renaming such that $\mathcal{D}(\rho) \subseteq \mathcal{V}(t)$, $\mathcal{R}ng(\rho) \cap W = \emptyset$ and $\rho(X) : \overline{\tau_k} \to \tau$ if $x_1 : \tau_1, \ldots, x_k : \tau_k$ and $X : \tau$.*

For simplicity, we will always assume that W contains all the variables used so far and leave W implicit. A term t is *$\overline{x}$-lifted* if an $\overline{x}$-lifter has been applied to t. Similarly, we say a rewrite rule $l \to r$ is *$\overline{x}$-lifted* if l and r are $\overline{x}$-lifted.

Definition 35 (rewrite step) *Let $\mathcal{R}$ be a GHRS and s a term. A* rewrite step *from s is a relation $s \to_{p,\theta,l\to r} t$ such that*

- *$\{x_1, \ldots, x_k\} = \mathcal{BV}(s, p)$ and $l \to r$ is an $\overline{x_k}$-lifter of a rule in $\mathcal{R}$, and*
- *$(\lambda\overline{x}.s)|_p = \lambda\overline{x}.(l\theta)$ and $t = \lambda\overline{x}.s[r\theta]_p$.*

We will often omit the parameters p, θ and $l \to r$ of a rewrite step $s \to_{p,\theta,l\to r} t$ and write instead $s \to_{\mathcal{R}} t$. If the GHRS $\mathcal{R}$ is understood from the context then we may omit $\mathcal{R}$ as well.

Example 5 *Consider the term* $s = \lambda x.f(x)$ *and the GHRS* $\mathcal{R} = \{f(X) \to g(X)\}$. *By choosing the position* $p = 1$ *we have* $\{x\} = \mathcal{BV}(s,p)$. *An* x*-lifter of the rule* $f(X) \to X$ *is* $f(X_1(x)) \to g(X_1(x))$. *Then* $s|_p = f(x) = f(X_1(x))\theta$ *where* $\theta = \{X_1 \mapsto \lambda y.y\}$, *and therefore we can perform the rewrite step*

$$s \to_{1,\theta,f(X_1(x))\to g(X_1(x))} \lambda x.s[g(X_1(x))\theta]_1 = (\lambda x.f(x)).[g(x)]_1 = \lambda x.g(x)$$

Notice that whereas first-order term rewriting is closed under substitutions, higher-order term rewriting is not: reducibility of s does not imply reducibility of $s\theta$. (E.g., if $\mathcal{R} = \{a \to b\}$ then the term $t = \lambda x.X(a,b)$ is reducible, but the term $t\{X \mapsto \lambda x,y.y\} = \lambda x.b$ is not.) However, the reflexive, transitive closure of $\to_{\mathcal{R}}$ is stable, that is, if $s \to^*_{\mathcal{R}} t$ then $s\theta \to^*_{\mathcal{R}} t\theta$. Also, a higher-order rewriting step is parameterized w.r.t. the substitution θ. In the first-order case, θ was omitted because of the uniqueness property of a matcher. Unfortunately, the uniqueness property of a matcher does not hold for simply typed λ-terms, as one can see in the example below.

Example 6 *Let* $a \in \mathcal{F}$ *with* $\mathrm{ar}(a) = 0$, $X \in \mathcal{V}$, $s = F(a)$, $t = a$. *Then both* $\theta_2 = \{X \mapsto \lambda x.x\}$ *and* $\theta_2 = \{X \mapsto \lambda x.a\}$ *are matchers of* s *with* t *because* $s\theta_1 = s\theta_2 = t$.

To ensure the existence of a unique matcher, we restrict the matching problem to a particular class of λ-terms.

Definition 36 (pattern) *A* relaxed higher-order pattern *is a simply typed* λ*-term* s *in* β*-normal form with the property that all free variables in* s *have only bound variables as arguments, i.e. if* $X(\overline{t_n})$ *is a subterm of* s, *then all* $t_i{\downarrow_\eta}$ *are bound variables. A* (higher-order) pattern *is a relaxed pattern where the arguments to free variables are distinct bound variables.*

Patterns behave like first-order terms in many respects. For instance, two patterns have a unique mgu modulo renaming [Nip91]. Since ground λ-terms are patterns, matchers of patterns with ground terms are unique, and thus term rewriting with rewrite rules having patterns to the left-hand side is similar to first-order term rewriting.

Definition 37 (PRS) *A* pattern rewrite system *(PRS for short) is a GHRS where all rewrite rules have patterns to the left hand side.*

In the sequel, if not stated otherwise, we assume that $\mathcal{R}$ is a PRS.

A suitable calculus for pattern unification is PU, which consists of the inference rules of PT plus the inference rules [ffs] and [ffd] shown below.

[ffs] *flex/flex same*

$$\frac{G, \lambda\overline{x}.X(\overline{y_m}) \approx \lambda\overline{x}.X(\overline{y'_m}), G'}{G\theta, G'\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.H(\overline{z_p})\}$, $\{\overline{z_p}\} = \{y_i \mid (1 \leq i \leq n) \wedge (y_i = y'_i)\}$ and H is a fresh variable

[ffd] *flex/flex different*

$$\frac{G, \lambda\overline{x}.X(\overline{y_m}) \approx \lambda\overline{x}.Y(\overline{y'_n}), G'}{G\theta, G'\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.H(\overline{z_p}), Y \mapsto \lambda\overline{y'_m}.H(\overline{z_p})\}$, $\{\overline{z_p}\} = \{\overline{y_m}\} \cap \{\overline{y'_n}\}$ and H is a fresh variable.

Theorem 5 (Miller [Mil91]) *If* $\mathcal{U}(G) \neq \emptyset$ *then there exists a* PU-*refutation* $G \overset{\text{PU}}{\Longrightarrow}{}^{*}_{\theta}\, \theta$. *Moreover,* $\theta = mgu(G)$.

A rewrite rule $(l \to r) \in \mathcal{R}$ is *left (right) linear* if l (r) does not contain multiple occurrences of the same free variable.

All the other notions defined for TRSs are extended to PRSs in the obvious way.

5.1.4 Higher-order Equational Logic

A PRS $\mathcal{R}$ induces an equivalence relation $\sim^{\mathcal{R}}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Similar to the first-order case, there exists [Pre98] a set of deduction rules which realize the entailment $\mathcal{R} \vdash s \approx t$ iff $s \sim^{\mathcal{R}} t$.

It is known [Wol93] that there exists an institution $\langle sen(\Sigma), \underline{Mod}(\Sigma), \models_{\Sigma}\rangle$ such that $\mathcal{R} \models^{*}_{\Sigma} s \approx t$ iff $s \sim^{\mathcal{R}} t$.

The following result of Nipkow [Nip91] states the correspondence between higher-order equational provability and higher-order term rewriting.

Theorem 6 *If* $\mathcal{R}$ *is a PRS then* $\mathcal{R} \vdash_{\Sigma} s \approx t$ *iff* $s\updownarrow^{\eta}_{\beta} \leftrightarrow^{*}_{\mathcal{R}} t\updownarrow^{\eta}_{\beta}$.

According to this theorem, we can decide the validity of an equation $s \approx t$ in the equational theory generated by a confluent PRS $\mathcal{R}$ by showing that there exists a λ-term u such that $s \to^{*} u$ and $t \to^{*}_{\mathcal{R}} u$. Like in the first-order case, we can represent such a rewrite proof by a sequence of steps

$$s \approx t \to_{\mathcal{R}} s_1 \approx t_1 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} u \approx u$$

where $s \approx t \to_{\mathcal{R}} s' \approx t' :\Leftrightarrow (s = s' \wedge t \to_{\mathcal{R}} t') \vee (s \to_{\mathcal{R}} s' \wedge t = t')$. The construction of a rewrite proof for the entailment $\mathcal{R} \vdash_{\Sigma} s \approx t$ requires the search for a suitable sequence of rewrite steps from $s \approx t$ to $u \approx u$. The choice of the rewrite step is a source of nondeterminism which is

- *don't care* if $\mathcal{R}$ is confluent and terminating, and
- *don't know* if $\mathcal{R}$ is only confluent.

5.2 Higher-order Lazy Narrowing for Pattern Rewrite Systems

In the first-order case, the lazy narrowing calculus LNC is an effective proof subcalculus which realizes entailments of the form $\mathcal{R} \vdash_\Sigma \exists \bigwedge_{i=1}^{n}(s_i \approx t_i)$. An LNC-proof corresponds to an LNC-refutation

$$\overline{s_n \approx t_n} \stackrel{\text{LNC}}{\Longrightarrow}{}^{*}_{\theta} \Box$$

which guarantees that $\mathcal{R} \vdash_\Sigma \bigwedge_{i=1}^{n}(s_i\theta \approx t_i\theta)$ or, equivalently, that $\theta \in \mathcal{U}_\mathcal{R}(\overline{s_n \approx t_n})$. For programming purposes it is desirable to have a calculus which computes a complete set of normalized $\mathcal{R}$-unifiers, and LNC is right on place: the set $Ans_{\mathcal{R}}^{\text{LNC}}(G) := \{\theta\restriction_{\mathcal{V}}(G) \mid G \stackrel{\text{LNC}}{\Longrightarrow}{}^{*}_{\theta} \Box\}$ is a subset of $\mathcal{U}_\mathcal{R}(G)$ which is complete for $\mathcal{U}_\mathcal{R}^{n}(G)$.

A higher-order lazy narrowing calculus should preserve as much as possible the properties of LNC, lifted in an appropriate way to a higher-order equational logic. But what are these properties? What are the higher-order counterparts of soundness and completeness of a lazy narrowing calculus for equational theories axiomatized with confluent PRSs in the simply typed λ-calculus?

First we take a look at the higher-order counterparts of the notions of unifier and $\mathcal{R}$-unifier.

For the calculi presented in this chapter we found useful to extend the syntax domain of equations with a new construct: oriented equations. An equation is a pair of terms (s, t) of the same type, written as $s \approx t$ if it is unoriented, and as $s \rhd t$ if it is oriented. An *$\mathcal{R}$-unifier* of an oriented equation $s \rhd t$ is a substitution θ such that $s\theta \to_{\mathcal{R}}^{*} t\theta$. An *$\mathcal{R}$-unifier* of an unoriented equation $s \approx t$ is a substitution θ such that $s\theta{\downarrow_{\mathcal{R}}}t\theta$. A *goal* is a logical conjunction of oriented and/or unoriented equations. A substitution θ is an $\mathcal{R}$-unifier of a goal $\overline{e_n}$ if it is an $\mathcal{R}$-unifier of any of its component equations e_i $(1 \leq i \leq n)$. Alternatively, we say that θ is *solution* of $\overline{e_n}$. Like in the first-order case, we write $\mathcal{U}_\mathcal{R}(G) := \{\theta \mid \theta \in \mathcal{U}_\mathcal{R}(e) \text{ for all } e \in G\}$ for the set of $\mathcal{R}$-unifiers of a goal G, and $\mathcal{U}_\mathcal{R}^{n}(G) := \{\theta \in \mathcal{U}_\mathcal{R}(G) \mid \theta \text{ is } \mathcal{R}\text{-normalized}\}$ for the set of $\mathcal{R}$-normalized $\mathcal{R}$-unifiers of G.

We have already mentioned that the computation of a unifier of two λ-terms is highly intractable: instead, we demand only the capability to compute preunifiers (e.g., by using the calculus PT.) Since the computation of higher-order unifiers is not practical, the computation of $\mathcal{R}$-unifiers is even more unpractical. Therefore, we weaken the requirement of a calculus to compute $\mathcal{R}$-unifiers to the requirement to compute *$\mathcal{R}$-preunifiers.*

Definition 38 ($\mathcal{R}$-preunifier) *An* $\mathcal{R}$-preunifier *of a goal G is a substitution θ for which there exists a flex/flex goal F such that* $\forall\gamma \in \mathcal{U}_{\mathcal{R}}(F).\theta\gamma \in \mathcal{U}_{\mathcal{R}}(G)$.

The notions of $\mathcal{C}$-step and $\mathcal{C}$-derivation for a given a higher-order calculus $\mathcal{C}$ are defined like in the first-order case.

A *$\mathcal{C}$-refutation* is a $\mathcal{C}$-derivation of the form $G \overset{\mathcal{C}}{\Longrightarrow}{}^*_\theta F$ such that F is a flex/flex goal and there is no $\mathcal{C}$-step starting with F. The superscript $\mathcal{C}$ will be omitted when the calculus is understood from the context.

The notions of soundness and completeness are generalized to a higher-order calculus $\mathcal{C}$ as follows:

- $\mathcal{C}$ is *sound* if for any $\mathcal{C}$-refutation $G \Rightarrow^*_\theta F$ and $\mathcal{R}$-unifier $\gamma \in \mathcal{U}_{\mathcal{R}}(F)$, we have $(\theta\gamma)\lceil_{\mathcal{V}(G)} \in \mathcal{U}_{\mathcal{R}}(G)$
- $\mathcal{C}$ is *complete* if for any goal G with unifier $\gamma \in \mathcal{U}^n_{\mathcal{R}}(G)$ there exists a $\mathcal{C}$-refutation $G \overset{\mathcal{C}}{\Longrightarrow}{}^*_\theta F$ with F a flex/flex goal, such that $\gamma = \theta\gamma'$ $[\mathcal{V}(G)]$ for some $\gamma' \in \mathcal{U}_{\mathcal{R}}(F)$.

We define $Ans^{\mathcal{C}}_{\mathcal{R}}(G) := \{\langle F, \theta\rangle \mid G \overset{\mathcal{C}}{\Longrightarrow}{}^*_\theta F \text{ is a } \mathcal{C}\text{-refutation}\}$.

Most often, the lazy narrowing calculi are not strongly complete. This means that the order in which the equations are selected affects the completeness of the calculus. One possibility to overcome the loss of completeness is to define a suitable equation selection strategy. Intuitively, an equation selection strategy defines a criterion that rules out the selection of all equations whose selection could generate the loss of completeness. In general, such a criterion is dependent on the "history" of the $\mathcal{C}$-derivation up to the current goal:

$$G_0 \overset{\mathcal{C}}{\Longrightarrow}_{\theta_1} G_1 \overset{\mathcal{C}}{\Longrightarrow}_{\theta_2} \ldots \overset{\mathcal{C}}{\Longrightarrow}_{\theta_{\mathrm{N}}} G_N.$$

We denote by $\mathcal{H}is_{\mathrm{LN_{ff}}}$ the set of all $\mathcal{C}$-derivations, and by $\mathcal{H}is_{\mathrm{LN_{ff}}}(G_N)$ the set of all $\mathcal{C}$-derivations which end with G_N.

In the sequel we give the formal notions related to equation selection functions and equation selection strategies.

Definition 39 (selection function) *Let $\mathcal{C}$ be a calculus. A* selection function *for $\mathcal{C}$ is a function*

$$sel : \mathcal{H}is_{\mathrm{LN_{ff}}} \to \mathcal{E}q(\mathcal{F}, \mathcal{V}) \cup \{\perp\}$$

which, when applied to a $\mathcal{C}$-derivation $\Pi : G \overset{\mathcal{C}}{\Longrightarrow}{}^*_\theta G'$ yields either $\perp$, or an equation $sel(\Pi) \in G'$. In addition, we assume that

- *if $sel(\Pi) = e \in G'$ then there exists a $\mathcal{C}$-step $\pi : G' \overset{\mathcal{C}}{\Longrightarrow}_{\theta'} G''$ in which the selected equation is e.*

The symbol $\perp$ denotes undefinedness. If $\Pi \in \mathcal{H}is_{\mathrm{LN_{ff}}}(G')$ and $sel(\Pi) = e \in G'$ then e is called the equation selected with sel in G'.

Our definition of selection function is very similar to the notion of selection rule in first-order logic programming [Apt90]. The main difference is that in our setting the selection function may be undefined for nonempty goals as well. We adopt a weaker condition than in logic programming because we want to avoid solving certain higher-order equations for which the computation of a solution is highly intractable.

We will often write $\mathcal{C}$-steps in the form $G, \underline{e}, G' \overset{\mathcal{C}}{\Longrightarrow}_\theta G''$ to express the fact that e is the equation selected in the goal G, e, G' upon the $\mathcal{C}$-step.

Definition 40 (strategy) *Let $\mathcal{C}$ be a calculus. A* strategy *for $\mathcal{C}$ is a nonempty set of equation selection functions for $\mathcal{C}$.*

Let A be a strategy for a calculus $\mathcal{C}$ and $\Pi : G \overset{\mathcal{C}}{\Longrightarrow}{}^*_\theta G' \in \mathcal{H}is_{\mathrm{LN_{ff}}}(G')$. A $\langle \mathcal{C}, A\rangle$*-step* for Π is a $\mathcal{C}$-step

$$\underbrace{G_1, \underline{e}, G_2}_{G'} \overset{\mathcal{C}}{\Longrightarrow}{}^*_{\theta'} G''$$

such that $e = sel(\Pi)$ for some $sel \in A$. A $\langle \mathcal{C}, A\rangle$*-derivation* is a $\mathcal{C}$-derivation

$$G_0 \overset{\mathcal{C}}{\Longrightarrow}_{\theta_1} G_1 \overset{\mathcal{C}}{\Longrightarrow}_{\theta_2} \ldots \overset{\mathcal{C}}{\Longrightarrow}_{\theta_N} G_N$$

such that for any $i \in \{1, \ldots, N\}$, $G_{i-1} \overset{\mathcal{C}}{\Longrightarrow}_{\theta_i} G_i$ is a $\langle \mathcal{C}, A\rangle$-step for the $\mathcal{C}$-subderivation $G_0 \overset{\mathcal{C}}{\Longrightarrow}{}^*_{\theta_1 \ldots \theta_i} G_i$.

A $\langle \mathcal{C}, A\rangle$*-refutation* is a $\langle \mathcal{C}, A\rangle$-derivation of the form $\Pi : G \overset{\mathcal{C}}{\Longrightarrow}{}^*_\theta G'$ such that $sel(\Pi) = \perp$ for all $sel \in A$.

A calculus $\mathcal{C}$ with a strategy A is *complete* if for any substitution $\gamma \in \mathcal{U}^n_{\mathcal{R}}(G)$ there exists a $\langle C, A\rangle$-refutation $G \overset{\mathcal{C}}{\Longrightarrow}{}^*_\theta G'$ and $\gamma' \in \mathcal{U}_{\mathcal{R}}(G')$ such that $\gamma = \theta\gamma'$ $[\mathcal{V}(G)]$ for some $\gamma' \in \mathcal{U}_{\mathcal{R}}(G')$. $\mathcal{C}$ is *strongly complete* if it is complete with any strategy.

5.3 The Calculus $\mathrm{LN}_{\mathrm{ff}}$

We represent $\mathrm{LN}_{\mathrm{ff}}$ as the disjoint combination of two subcalculi: $\mathrm{LN}_{\mathrm{ff}}^{\triangleright}$ and $\mathrm{LN}_{\mathrm{ff}}^{\approx}$. The subcalculus $\mathrm{LN}_{\mathrm{ff}}^{\triangleright}$ consists of the inference rules for selected equations which are oriented, and $\mathrm{LN}_{\mathrm{ff}}^{\approx}$ consists of the inference rules for selected equations which are unoriented.

The subcalculus $\mathrm{LN}_{\mathrm{ff}}^{\triangleright}$

The inference rules of $\mathrm{LN}_{\mathrm{ff}}^{\triangleright}$ are shown in Fig. 5.2. The design $\mathrm{LN}_{\mathrm{ff}}^{\triangleright}$ is inspired by the calculus LN, which was proposed for computing solutions of oriented goals. The notable difference is that LN does not contain the inference rules $[\mathrm{ffs}]_{\triangleright}$ and $[\mathrm{ffd}]_{\triangleright}$ for higher-order pattern unification. $[\mathrm{ffs}]_{\triangleright}$ and $[\mathrm{ffd}]_{\triangleright}$ have been added to improve the computing power.

The subcalculus $\mathrm{LN}_{\mathrm{ff}}^{\approx}$

The subcalculus $\mathrm{LN}_{\mathrm{ff}}^{\approx}$ can be viewed a generalization of the subcalculus $\mathrm{LN}_{\mathrm{ff}}^{\triangleright}$ to the case when the selected equation is unoriented. Its inference rules are shown in Fig. 5.3.

5.3.1 Main Properties

In general, the humble addition of new inference rules to a calculus affects its main properties, such as soundness and completeness. We already mentioned that the calculus $\mathrm{LN}_{\mathrm{ff}}$ was inspired by the calculus LN by adding the missing inference rules which are necessary for performing full pattern unification. With this addition we can perform further inference steps on a flex pattern/flex pattern equation, thereby computing more concrete solutions than with LN, but we may lose some of the important properties of LN such as soundness, completeness, and strong completeness.

Soundness

We first observe that $\mathrm{LN}_{\mathrm{ff}}$ is a sound calculus.

Lemma 23 *Let $\Pi : G \Rightarrow_{\theta}^{*} G'$ be an $\mathrm{LN}_{\mathrm{ff}}$-derivation. If $\gamma \in \mathcal{U}_{\mathcal{R}}(G')$ then $\theta\gamma \in \mathcal{U}_{\mathcal{R}}(G)$.*

Proof. The proof is by induction on the length of the $\mathrm{LN}_{\mathrm{ff}}$-derivation. The case when $|\Pi| = 0$ is trivial. Assume $|\Pi| > 0$. Then we can write

$$\Pi : G \Rightarrow_{\theta_1} G'' \Rightarrow_{\alpha,\theta_2} G'$$

$[\mathrm{del}]_{\rhd}$ *deletion*

$$\frac{G, t \rhd t, G'}{G, G'}$$

$[\mathrm{d}]_{\rhd}$ *decomposition*

$$\frac{G, \lambda\overline{x}.v(\overline{s_n}) \rhd \lambda\overline{x}.v(\overline{t_n}), G'}{G, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.t_n}, G'}$$

$[\mathrm{i}]_{\rhd}$ *imitation*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \rhd \lambda\overline{x}.g(\overline{t_m}), G'}{(G, \overline{\lambda\overline{x}.H_m(\overline{s_n}) \rhd \lambda\overline{x}.t_m}, G')\theta} \quad \frac{G, \lambda\overline{x}.g(\overline{t_m}) \rhd \lambda\overline{x}.X(\overline{s_n}), G'}{(G, \overline{\lambda\overline{x}.t_m \rhd \lambda\overline{x}.H_m(\overline{s_n})}, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{x_n}.g(\overline{H_m(\overline{x_n})})\}$ and $\overline{H_m}$ are fresh variables.

$[\mathrm{p}]_{\rhd}$ *projection*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \rhd \lambda\overline{x}.t, G'}{(G, \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}) \rhd \lambda\overline{x}.t, G')\theta} \qquad \frac{G, \lambda\overline{x}.t \rhd \lambda\overline{x}.X(\overline{s_n})), G'}{(G, \lambda\overline{x}.t \rhd \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}), G')\theta}$$

where $1 \le i \le n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are fresh variables.

$[\mathrm{on}]_{\rhd}$ *outermost narrowing at nonvariable position*

$$\frac{G, \lambda\overline{x}.f(\overline{s_n}) \rhd \lambda\overline{x}.t, G'}{G, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \rhd \lambda\overline{x}.t, G'}$$

if $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule.

$[\mathrm{ov}]_{\rhd}$ *outermost narrowing at variable position*

$$\frac{G, \lambda\overline{x}.X(\overline{s_m}) \rhd \lambda\overline{x}.t, G'}{(G, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \rhd \lambda\overline{x}.t, G')\theta}$$

if $\lambda\overline{x}.t$ is rigid, $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule and $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types.

$[\mathrm{ffs}]_{\rhd}$ *flex/flex same*

$$\frac{G, \lambda\overline{x}.X(\overline{y_n}) \rhd \lambda\overline{x}.X(\overline{y'_n}), G'}{(G, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{y_i \mid 1 \le i \le n \text{ and } y_i = y'_i\}$.

$[\mathrm{ffd}]_{\rhd}$ *flex/flex different*

$$\frac{G, \lambda\overline{x}.X(\overline{y_m}) \rhd \lambda\overline{x}.Y(\overline{y'_n}), G'}{(G, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.H(\overline{z_p}), Y \mapsto \lambda\overline{y'_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{\overline{y_m}\} \cap \{\overline{y'_n}\}$.

Fig. 5.2: The calculus $\mathrm{LN}^{\rhd}_{\mathrm{ff}}$: inference rules

$[\mathrm{del}]_{\approx}$ *deletion*

$$\frac{G, t \approx t, G'}{G, G'}$$

$[\mathrm{d}]_{\approx}$ *decomposition*

$$\frac{G, \lambda\overline{x}.v(\overline{s_n}) \approx \lambda\overline{x}.v(\overline{t_n}), G'}{G, \overline{\lambda\overline{x}.s_n \approx \lambda\overline{x}.t_n}, G'}$$

$[\mathrm{i}]_{\approx}$ *imitation*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \approx \lambda\overline{x}.g(\overline{t_m}), G'}{(G, \overline{\lambda\overline{x}.H_m(\overline{s_n}) \approx \lambda\overline{x}.t_m}, G')\theta} \qquad \frac{G, \lambda\overline{x}.g(\overline{t_m}) \approx \lambda\overline{x}.X(\overline{s_n}), G'}{(G, \overline{\lambda\overline{x}.t_m \approx \lambda\overline{x}.H_m(\overline{s_n})}, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{x_n}.g(\overline{H_m(\overline{x_n})})\}$ and $\overline{H_m}$ are fresh variables.

$[\mathrm{p}]_{\approx}$ *projection*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \approx \lambda\overline{x}.t, G'}{(G, \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}) \approx \lambda\overline{x}.t, G')\theta} \qquad \frac{G, \lambda\overline{x}.t \approx \lambda\overline{x}.X(\overline{s_n})), G'}{(G, \lambda\overline{x}.t \approx \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}), G')\theta}$$

where $1 \leq i \leq n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are fresh variables.

$[\mathrm{on}]_{\approx}$ *outermost narrowing at nonvariable position*

$$\frac{G, \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t, G'}{G, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G'}$$

if $\cong \in \{\approx, \approx^{-1}\}$, $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule.

$[\mathrm{ov}]_{\approx}$ *outermost narrowing at variable position*

$$\frac{G, \lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.t, G'}{(G, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G')\theta}$$

if $\cong \in \{\approx, \approx^{-1}\}$, $\lambda\overline{x}.t$ is rigid, $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule and $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types.

$[\mathrm{ffs}]_{\approx}$ *flex/flex same*

$$\frac{G, \lambda\overline{x}.X(\overline{y_n}) \approx \lambda\overline{x}.X(\overline{y'_n}), G'}{(G, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{y_i \mid 1 \leq i \leq n \text{ and } y_i = y'_i\}$.

$[\mathrm{ffd}]_{\approx}$ *flex/flex different*

$$\frac{G, \lambda\overline{x}.X(\overline{y_m}) \approx \lambda\overline{x}.Y(\overline{y'_n}), G'}{(G, G')\theta}$$

where $\theta = \{X \mapsto \lambda\overline{y_m}.H(\overline{z_p}), Y \mapsto \lambda\overline{y'_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{\overline{y_m}\} \cap \{\overline{y'_n}\}$.

Fig. 5.3: The calculus $\mathrm{LN}_{\mathrm{ff}}^{\approx}$: inference rules

where $\theta = \theta_1\theta_2$ and $\alpha \in \{[\mathrm{d}], [\mathrm{on}], [\mathrm{ov}], [\mathrm{i}], [\mathrm{p}], [\mathrm{ffs}], [\mathrm{ffd}]\}$. Assume that $\gamma \in \mathcal{U}_{\mathcal{R}}(G')$. We want to prove that $\theta\gamma \in \mathcal{U}_{\mathcal{R}}(G)$.

We first prove that $\theta_2\gamma \in \mathcal{U}_{\mathcal{R}}(G'')$. This proof is by case distinction on α.

If $\alpha \in \{[\mathrm{d}], [\mathrm{i}]\}$ then we can assume that

$$\begin{aligned} G''\theta_2 &= G_1, \lambda\overline{x}.v(\overline{s_n}) \cong \lambda\overline{x}.v(\overline{t_n}), G_2, \\ G' &= G_1, \overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n}, G_2 \end{aligned}$$

where $\cong \in \{\approx, \approx^{-1}, \rhd\}$.

- If $\alpha \in \{[\mathrm{d}]_\rhd, [\mathrm{i}]_\rhd\}$ then $\gamma \in \mathcal{U}_{\mathcal{R}}(G')$ implies $\gamma \in \mathcal{U}_{\mathcal{R}}(\overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.t_n}) \cap \mathcal{U}_{\mathcal{R}}(G_1) \cap \mathcal{U}_{\mathcal{R}}(G_2)$. From $\gamma \in \mathcal{U}_{\mathcal{R}}(\overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.t_n})$ we deduce that $\lambda\overline{x}.s_i\gamma \to^*_{\mathcal{R}} \lambda\overline{x}.t_i\gamma$ for all $i \in \{1,\ldots,n\}$, and thus $\lambda\overline{x}.v(\overline{s_n}) \to^*_{\mathcal{R}} \lambda\overline{x}.v(\overline{t_n})$, i.e. $\gamma \in \mathcal{U}_{\mathcal{R}}(\lambda\overline{x}.v(\overline{s_n}) \rhd \lambda\overline{x}.v(\overline{t_n}))$. We conclude that $\gamma \in \mathcal{U}_{\mathcal{R}}(G_1) \cap \mathcal{U}_{\mathcal{R}}(\lambda\overline{x}.v(\overline{s_n}) \rhd \lambda\overline{x}.v(\overline{t_n})) \cap \mathcal{U}_{\mathcal{R}}(G_2)$, i.e. $\gamma \in \mathcal{U}_{\mathcal{R}}(G''\theta_2)$. This yields $\theta_2\gamma \in \mathcal{U}_{\mathcal{R}}(G'')$.

- If $\alpha \in \{[\mathrm{d}]_\approx, [\mathrm{i}]_\approx\}$ then $\gamma \in \mathcal{U}_{\mathcal{R}}(G')$ implies $\gamma \in \mathcal{U}_{\mathcal{R}}(\overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n}) \cap \mathcal{U}_{\mathcal{R}}(G_1) \cap \mathcal{U}_{\mathcal{R}}(G_2)$. From $\gamma \in \mathcal{U}_{\mathcal{R}}(\overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n})$ we deduce that $\lambda\overline{x}.s_i\gamma \leftrightarrow^*_{\mathcal{R}} \lambda\overline{x}.t_i\gamma$ for all $i \in \{1,\ldots,n\}$, and thus $\lambda\overline{x}.v(\overline{s_n}) \leftrightarrow^*_{\mathcal{R}} \lambda\overline{x}.v(\overline{t_n})$, i.e. $\gamma \in \mathcal{U}_{\mathcal{R}}(\lambda\overline{x}.v(\overline{s_n}) \cong \lambda\overline{x}.v(\overline{t_n}))$. We conclude that $\gamma \in \mathcal{U}_{\mathcal{R}}(G_1) \cap \mathcal{U}_{\mathcal{R}}(\lambda\overline{x}.v(\overline{s_n}) \cong \lambda\overline{x}.v(\overline{t_n})) \cap \mathcal{U}_{\mathcal{R}}(G_2)$, i.e. $\gamma \in \mathcal{U}_{\mathcal{R}}(G''\theta_2)$. This yields $\theta_2\gamma \in \mathcal{U}_{\mathcal{R}}(G'')$.

If $\alpha \in \{[\mathrm{on}], [\mathrm{ov}]\}$ then we can assume that

$$\begin{aligned} G''\theta_2 &= G_1, \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t, G_2 \\ G' &= G_1, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G_2 \end{aligned}$$

where $\cong \in \{\approx, \approx^{-1}, \rhd\}$, $f(\overline{l_n}) \to r$ is an $\overline{x}$-lifted variant of a rewrite rule in $\mathcal{R}$. From $\gamma \in \mathcal{U}_{\mathcal{R}}(G')$ we obtain

(1) $\gamma \in \mathcal{U}_{\mathcal{R}}(G_1) \cap \mathcal{U}_{\mathcal{R}}(G_2)$,
(2) $\lambda\overline{x}.s_i\gamma \to^*_{\mathcal{R}} \lambda\overline{x}.l_i\gamma$ for all $i \in \{1,\ldots,n\}$, and
(3) $\lambda\overline{x}.r\gamma \to^*_{\mathcal{R}} \lambda\overline{x}.t\gamma$ if $\alpha \in \{[\mathrm{on}]_\rhd, [\mathrm{ov}]_\rhd\}$,
and $\lambda\overline{x}.r\gamma \leftrightarrow^*_{\mathcal{R}} \lambda\overline{x}.t\gamma$ if $\alpha \in \{[\mathrm{on}]_\approx, [\mathrm{ov}]_\approx\}$.

Then $\lambda\overline{x}.f(\overline{s_n\gamma}) \overset{(2)}{\to}{}^*_{\mathcal{R}} \lambda\overline{x}.f(\overline{l_n\gamma}) \to_{\mathcal{R}} \lambda\overline{x}.r\gamma \overset{(3)}{\to}{}^*_{\mathcal{R}} \lambda\overline{x}.t\gamma$ if $\alpha \in \{[\mathrm{on}]_\rhd, [\mathrm{ov}]_\rhd\}$, and $\lambda\overline{x}.f(\overline{s_n\gamma}) \overset{(2)}{\to}{}^*_{\mathcal{R}} \lambda\overline{x}.f(\overline{l_n\gamma}) \to_{\mathcal{R}} \lambda\overline{x}.r\gamma \overset{(3)}{\leftrightarrow}{}^*_{\mathcal{R}} \lambda\overline{x}.t\gamma$ if $\alpha \in \{[\mathrm{on}]_\approx, [\mathrm{ov}]_\approx\}$.

Thus, $\gamma \in \mathcal{U}_{\mathcal{R}}(G''\theta_2)$, and therefore $\theta_2\gamma \in \mathcal{U}_{\mathcal{R}}(G'')$.

If $\alpha = [\mathrm{p}]$ then $G' = G''\theta_2$, which yields $\sigma\gamma \in \mathcal{U}_{\mathcal{R}}(G'')$.

If $\alpha \in \{[\text{ffs}],[\text{ffd}]\}$ then we can assume that $G''\theta_2 = G_1, \lambda\overline{x}.t \cong \lambda\overline{x}.t, G_2$ with $\cong \in \{\approx, \rhd\}$ and $G' = G_1, G_2$. Obviously, $\gamma \in \mathcal{U}_{\mathcal{R}}(G')$ implies $\gamma \in \mathcal{U}_{\mathcal{R}}(G''\theta_2)$, and thus $\theta_2\gamma \in \mathcal{U}_{\mathcal{R}}(G'')$.

The case $\alpha = [\text{del}]$ is trivial.

We have shown that $\theta_2\gamma \in \mathcal{U}_{\mathcal{R}}(G'')$. From the induction hypothesis for the LN_{ff}-derivation $G \Rightarrow^*_{\theta_1} G''$ results that $\theta_1\theta_2\gamma (= \theta\gamma) \in \mathcal{U}_{\mathcal{R}}(G)$. □

Complete Strategies

We first note that LN_{ff} is not strongly complete.

Example 7 *Assume* $\mathcal{R} = \{f(X) \to c(X,X), a \to b\}$ *and the goal*

$$G = f(a) \rhd c(a,X), b \rhd X.$$

It can be verified that $\mathcal{R}$ *is confluent and that* $\gamma = \{X \mapsto b\}$ *is an* $\mathcal{R}$*-unifier of* G_0*. Consider the* LN_{ff}*-derivation*

$$\begin{aligned}
\Pi: \quad & G = \underline{f(a) \rhd c(a,X)}, b \rhd X \\
& \Rightarrow_{[\text{on}], f(X_1) \to c(X_1,X_1)} G_1 = a \rhd X_1, \underline{c(X_1,X_1) \rhd c(a,X)}, b \rhd X \\
& \Rightarrow_{[\text{d}]} G_2 = a \rhd X_1, X_1 \rhd a, \underline{X_1 \rhd X}, b \rhd X \\
& \Rightarrow_{[\text{ffd}]\{X_1 \mapsto H, X_2 \mapsto H\}} G_3 = \underline{a \rhd H}, H \rhd a, b \rhd H \\
& \Rightarrow_{[\text{i}], \{H \mapsto a\}} \underline{a \rhd a}, b \rhd a \Rightarrow_{[\text{del}]} b \rhd a.
\end{aligned}$$

Π *is a maximal* LN_{ff}*-derivation, and in each step we can choose only one inference rule.* Π *computes the substitution* $\theta = \{X_1 \mapsto a, X_2 \mapsto a, H \mapsto a\}$*, and there is no* $\gamma' \in \mathcal{U}_{\mathcal{R}}(b \rhd a)$ *such that* $\gamma = \theta\gamma'$*. (Actually, it can be shown that* $\mathcal{U}_{\mathcal{R}}(b \rhd a) = \emptyset$*.) Thus* LN_{ff} *is not strongly complete.*

Note that θ *can be computed with* LN_{ff} *if we select any other equation of* G_2 *except the third one. Thus, this example does not refute the completeness of* LN_{ff}.

This example illustrates that LN_{ff} is not strongly complete. This does not imply that LN_{ff} is incomplete.

We recall that the first-order lazy narrowing calculus LNC also lacks strong completeness, but it was proven that LNC is complete for the computation of normalized solutions if it is adopted the leftmost equation selection function. We will try to do a similar thing with our lazy narrowing calculus LN_{ff}, i.e., to define equation selection functions which make LN_{ff} complete for the computation of normalized solutions.

We first introduce some terminology that will be used in stating our results.

Definition 41 *Let $G = \overline{e_n}$ be a goal. We define* $\mathrm{Repr}(G)$ *as the set of triples of the form $\langle G, \theta, \overline{R_n}\rangle$ with $\theta \in \mathcal{U}_{\mathcal{R}}(G)$ and $\overline{R_n}$ a sequence of rewrite proofs of the fact that $\theta \in \mathcal{U}_{\mathcal{R}}(e_i)$, for $i \in \{1, \ldots, n\}$. More precisely, for each $i \in \{1, \ldots, n\}$ we have $R_i : s_i\theta \approx t_i\theta \rightarrow^*_{\mathcal{R}} u_i \approx u_i$ if $e_i = s_i \approx t_i$, and $R_i : s_i\theta \rhd t_i\theta \rightarrow^*_{\mathcal{R}} t_i\theta \rhd t_i\theta$ if $e_i = s_i \rhd t_i$. In the last case, only the term to the left-hand side of the equation is rewritten.*

We denote by $|R|$ the length of a rewrite proof R. If $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ then $|t|$ denotes the *size* of t, i.e.

$$|t| := \begin{cases} |s| & \text{if } t = \lambda x.s, \\ |t_1| + |t_2| & \text{if } t = (t_1\ t_2), \\ 1 & \text{if } t \in \mathcal{V} \cup \mathcal{F}. \end{cases}$$

We also define the size of an equation by $|s \approx t| := |s \rhd t| := |s| + |t|$, and consider the following well-founded orderings on the set of triples introduced in Definition 41:

- $\langle \overline{e_n}, \theta, \overline{R_n}\rangle >_A \langle \overline{e'_m}, \theta', \overline{R'_m}\rangle$ if $\{|R_1|, \ldots, |R_n|\} >_{mul} \{|R'_1|, \ldots, |R'_m|\}$,
- $\langle G, \theta, \overline{R}\rangle >_B \langle G', \theta', \overline{R'}\rangle$ if $\{|t| \mid t \in \mathcal{I}(\theta)\} >_{\mathrm{mul}} \{|t'| \mid t' \in \mathcal{I}(\theta')\}$,
- $\langle \overline{e_n}, \theta, \overline{R_n}\rangle >_C \langle \overline{e_m}, \theta', \overline{R'_m}\rangle$ if $\{|e_1|, \ldots, |e_n|\} >_{mul} \{|e'_1|, \ldots, |e'_m|\}$,
- $\succ$ is the lexicographic combination of $>_A, >_B, >_C$.

Definition 42 (critical variable) *The set of* critical variables *of an equation $e \in \mathcal{E}q(\mathcal{F}, \mathcal{V})$ is*

$$\mathcal{V}_c(e) := \begin{cases} \mathcal{V}(s) & \text{if } e = s \rhd t \\ \mathcal{V}(s) \cup \mathcal{V}(t) & \text{if } e = s \approx t \end{cases}$$

In order to define suitable equation selection functions for $\mathrm{LN}_{\mathrm{ff}}$ we will add more structure to the concept of $\mathrm{LN}_{\mathrm{ff}}$-step.

Let $\pi : G \Rightarrow_{\alpha,\theta} G'$ be an $\mathrm{LN}_{\mathrm{ff}}$-step and $e \in G$ the equation upon π from G. The *descendants* and the *linear descendants* of e in G' w.r.t. π, denoted by $\mathrm{desc}_\pi(e)$ and $\mathrm{ldesc}_\pi(e)$ respectively, are the equations in G' defined as shown in the table below.

α	e	$\mathrm{desc}_\pi(e)$	
		$\mathrm{ldesc}_\pi(e)$	
[on]	$\lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t$	$\lambda\overline{x}.r \cong \lambda\overline{x}.t$	$\overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}$,
[ov]	$\lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.t$	$\lambda\overline{x}.r \cong \lambda\overline{x}.t\theta$	$\overline{\lambda\overline{x}.H_n(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_n}$
[dec]	$\lambda\overline{x}.v(\overline{s_n} \cong \lambda\overline{x}.v(\overline{t_n})$	$\overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n}$	-
[i]	$\lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.f(\overline{t_m})$	$\overline{\lambda\overline{x}.H_m(\overline{s_n\theta}) \cong \lambda\overline{x}.t_m\theta}$	-
[p]	$\lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.t$	$\lambda\overline{x}.(s_i\theta)(\overline{H_p(\overline{s_n\theta})}) \cong \lambda\overline{x}.t\theta$	-
[del]	$\lambda\overline{x}.t \cong \lambda\overline{x}.t$	-	-
[ffs]	$\lambda\overline{x}.X(\overline{y_n}) \cong \lambda\overline{x}.X(\overline{y'_n})$	-	-
[ffd]	$\lambda\overline{x}.X(\overline{y_m}) \cong \lambda\overline{x}.X(\overline{y'_n})$	-	-

If $e \in G$ is not selected upon the $\mathrm{LN_{ff}}$-step $\pi : G \Rightarrow_\theta G'$ then $\mathrm{desc}_\pi(e) = \mathrm{ldesc}_\pi(e) = e\theta$. The notions of descendant and linear descendant of a subgoal G'' of G in G' w.r.t. an $\mathrm{LN_{ff}}$-derivation $\Pi : G \Rightarrow^*_\theta G'$ are defined inductively in the obvious way.

We associate with every $\mathrm{LN_{ff}}$-derivation $\Pi : G \Rightarrow^*_\theta G'$ a *precursor function* prec_Π which maps the equations of G' to subgoals of G'. For any $e \in G$, the equations in $\mathrm{prec}_\Pi(e)$ are called the *precursors of e in G w.r.t.* Π. Formally, $\mathrm{prec}_\Pi(e)$ is defined as follows:

Definition 43 (precursor) *If* $\Pi : G \Rightarrow^0 G$ *is an empty* $\mathrm{LN_{ff}}$*-derivation then* $\mathrm{prec}_\Pi(e) = \Box$ *for all* $e \in G$.

If $\Pi : G \Rightarrow^* G_1, e, G_2 \Rightarrow_\alpha G' = G'_1, e', G'_2$ *is a nonempty* $\mathrm{LN_{ff}}$*-derivation such that* $e' \in \mathrm{desc}(e)$ *then*

$$\mathrm{prec}_\Pi(e') = \begin{cases} \mathrm{desc}_\pi(\mathrm{prec}_{\Pi'}(e), e) \setminus e' & \text{if } \alpha \in \{[\mathrm{on}], [\mathrm{ov}]\},\ e \text{ is selected} \\ & \text{upon } \pi \text{ from } G \text{ and } e' = \mathrm{ldesc}_\pi(e) \\ \mathrm{desc}_\pi(\mathrm{prec}_{\Pi'}(e)) & \text{otherwise.} \end{cases}$$

where

- π *is the last* $\mathrm{LN_{ff}}$*-step of* Π*, and* Π' *is the* $\mathrm{LN_{ff}}$*-subderivation of* Π *without* π*,*
- $\mathrm{desc}_\Pi(\overline{e_n}) := \mathrm{desc}_\Pi(e_1), \ldots, \mathrm{desc}_\Pi(e_n)$ *whenever* $\Pi : G \Rightarrow G'$ *is an* $\mathrm{LN_{ff}}$*-derivation and* $\overline{e_n}$ *is a subgoal of* G'*,*
- $G \setminus e$ *denotes the goal obtained by removing the equation* e *from* G*.*

For the sake of simplicity we drop the subscript and simply write $\mathrm{prec}(e)$ instead of $\mathrm{prec}_\Pi(e)$ if Π is understood from the context.

We first prove the following three technical lemmata.

Lemma 24 *Let* $G = \overline{e_n}$ *with* $e_k \in G$ *a non-flex/flex equation. Then for any* $\langle \overline{e_n}, \gamma, \overline{R_n} \rangle \in \mathrm{Repr}(G)$ *there exists an* $\mathrm{LN_{ff}}$*-step with selected equation* e_k

$$\pi : \overline{e_{k-1}}, \underline{e_k}, \overline{e_{k+1,n}} \Rightarrow_{\alpha,\theta} G'$$

and a triple $\langle G', \gamma', \overline{R'} \rangle \in \mathrm{Repr}(G')$ *such that:*

(a) $\langle G, \gamma, \overline{R_n} \rangle \succ \langle G', \gamma', \overline{R'} \rangle$*, and*

(b) $\gamma = \theta\gamma'\ [\mathcal{V}(G)]$.

<u>**Proof.**</u> Since e_k is not a flex/flex equation, we can write $e_k = s \cong t$ with $\cong \in \{\approx, \approx^{-1}, \rhd\}$ and either s or t is a rigid term. Then the rewrite proof R_k corresponding to e_k is of the form

- $s\gamma \rhd t\gamma \rightarrow^*_{\mathcal{R}} t\gamma \rhd t\gamma$ if $e_k = s \rhd t$, or
- $s\gamma \approx t\gamma \rightarrow^*_{\mathcal{R}} u \approx u$ if $e_k = s \approx t$.

We distinguish two cases, depending on whether R_k has length 0 or not.

If $|R_k| = 0$ then $s\gamma = t\gamma$. Then it can be shown (cf. the proof of Thm. 4.1.7. in [Pre98]) that there exists an $\mathrm{LN_{ff}}$-step $G \Rightarrow_{\alpha,\theta} G'$ with $\alpha \in \{[\mathrm{del}], [\mathrm{dec}], [\mathrm{i}], [\mathrm{p}]\}$ and a triple $\langle G', \gamma', \overline{R'}\rangle \in \mathrm{Repr}(G')$ such that $\gamma = \theta\gamma'\ [\mathcal{V}(G)]$ and either:

- $\langle G, \gamma, \overline{\mathcal{R}_n}\rangle >_{\mathrm{B}} \langle G', \gamma', \overline{R'}\rangle$, or
- $\langle G, \gamma, \overline{R_n}\rangle =_{\mathrm{B}} \langle G', \gamma', \overline{R'}\rangle$ and $\langle G, \gamma, \overline{R_n}\rangle >_{\mathrm{C}} \langle G', \gamma', \overline{R'}\rangle$.

From the hypothesis $|R_k| = 0$ we learn that $\langle G, \gamma, \overline{R_n}\rangle =_{\mathrm{A}} \langle G', \gamma', \overline{R'}\rangle$. Thus $\langle G, \gamma, \overline{R_n}\rangle \succ \langle G', \gamma', \overline{R'}\rangle$.

If $|R_k| > 0$ then we distinguish two cases, depending on whether R_k has rewrite steps at the root position of a side of some equation in the derivation.

Subcase 1. If R_k has no such rewrite steps then we can assume that $s\theta = \lambda\overline{x}.g(\overline{s_p})$, $t\theta = \lambda\overline{x}.g(\overline{t_p})$ such that R_k only rewrites $\overline{s_p}$ and $\overline{t_p}$. If s and t are both rigid then we can apply a decomposition step $\pi : G \Rightarrow_{[\mathrm{d}],\varepsilon} G'$ and determine $\langle G', \gamma, \overline{R'}\rangle \in \mathrm{Repr}(G')$ such that

$\langle G, \gamma, \overline{R_n}\rangle \geq_{\mathrm{A}} \langle G', \gamma, \overline{R'}\rangle, \langle G, \gamma, \overline{R_n}\rangle =_{\mathrm{B}} \langle G', \gamma, \overline{R'}\rangle, \langle G, \gamma, \overline{R_n}\rangle >_{\mathrm{C}} \langle G', \gamma, \overline{R'}\rangle.$

Thus $\langle G, \gamma, \overline{R_n}\rangle \succ \langle G', \gamma, \overline{R'}\rangle$. If either s or t is flex, then suppose s is flex. (The case when t is flex is similar.) We can write $s = \lambda\overline{x}.X(\overline{u})$ with $X \in \mathcal{V}(s)$. Then (cf. Lemma 22) we can perform an α-step

$$\pi : G = \overline{e_{k-1}}, \underline{\lambda\overline{x}.X(\overline{u}) \cong t}, \overline{e_{k+1,n}} \Rightarrow_{\alpha,\theta} G'$$

with $\alpha \in \{[\mathrm{i}], [\mathrm{p}]\}$, for which there exists a substitution γ' such that

(i) $\mathcal{D}(\gamma') = (\mathcal{D}(\gamma) \setminus \{X\}) \cup \mathcal{R}ng(\theta)$,

(ii) $X\gamma = X\theta\gamma'$,

(iii) $\gamma = \gamma'\ [\mathcal{D}(\gamma) \setminus \{X\}]$.

From (i)-(iii) we conclude that $\{|t| \mid t \in \mathcal{I}(\gamma)\} >_{mul} \{|t'| \mid t' \in \mathcal{I}(\gamma')\}$. Then we can determine $\overline{R'}$ such that $\langle G, \gamma, \overline{R_n}\rangle \geq_{\mathrm{A}} \langle G', \gamma', \overline{R'}\rangle$ and $\langle G, \gamma, \overline{R_n}\rangle >_{\mathrm{B}} \langle G', \gamma', \overline{R'}\rangle$. Hence $\langle G, \gamma, \overline{R_n}\rangle \succ \langle G', \gamma', \overline{R'}\rangle$.

Subcase 2. If R_k has rewrite steps at the root position of a side of some equation then we consider the first of these and write:

$$s\theta \cong t\theta \rightarrow^*_{\mathcal{R}} \lambda\overline{x}.f(\overline{u_m}) \cong t' \rightarrow_{f(\overline{l_m})\to r,\delta} \lambda\overline{x}.r\delta \cong t' \rightarrow^*_{\mathcal{R}} u \cong u.$$

Since $f(\overline{l_m}) \to r$ is a fresh variant of a rewrite rule and $\mathcal{D}(\delta) \subseteq \mathcal{V}(f(\overline{l_m}))$, we conclude that $\gamma' := \gamma \cup \delta$ is a well-defined substitution and that $\gamma = \gamma'$ $[\mathcal{V}(G)]$. We denote by R''_k the rewrite subderivation of R_k which ends with $\lambda\overline{x}.r\delta \cong t'$, and with R'_{m+1} the rewrite subproof of R_k which starts with $\lambda\overline{x}.r\delta \cong \lambda\overline{x}.t'$. We observe that we must have $s\theta = \lambda\overline{x}.f(\overline{s_m})$ and that we can extract from R''_k a sequence of rewrite proofs

$$\overline{R'_m} := \overline{\lambda\overline{x}.s_m \rhd \lambda\overline{x}.u_m \to^*_{\mathcal{R}} \lambda\overline{x}.u_m \rhd \lambda\overline{x}.u_m}$$

such that $|R'_i| < |R''_k|$ for all $i \in \{1, \ldots, m\}$. We distinguish two subcases, depending on whether s is rigid or not. If s is rigid then $s = \lambda\overline{x}.f(\overline{s'_m})$, and we can perform the LN$_{\text{ff}}$-step

$$\begin{aligned}\pi: \quad & G = \overline{e_{k-1}}, \lambda\overline{x}.f(\overline{s'_m}) \cong t, \overline{e_{k+1,n}} \\ & \Rightarrow_{[\text{on}], f(\overline{l_m}) \to r} G' = \overline{e_{k-1}}, \overline{\lambda\overline{x}.s'_m \rhd \lambda\overline{x}.l_m}, \lambda\overline{x}.r \cong t, \overline{e_{k+1,n}}.\end{aligned}$$

It is easy to see that $\gamma' \in \mathcal{U}_{\mathcal{R}}(G')$ and that $\langle G', \gamma', \overline{R'}\rangle \in \text{Repr}(G')$ where $\overline{R'} := \overline{R_{k-1}}, \overline{R'_m}, R'_{m+1}, \overline{R_{k+1,n}}$. Even more, we have that $\langle G, \gamma, \overline{R_n}\rangle >_{\text{A}} \langle G', \gamma', \overline{R'}\rangle$, and therefore $\langle G, \gamma, \overline{R_n}\rangle \succ \langle G', \gamma', \overline{R'}\rangle$.

If s is a flex term then $s = \lambda\overline{x}.X(\overline{s'_p})$ and $s\theta = \lambda\overline{x}.f(\overline{s''_m})$. This implies (cf. Lemma 22) the existence of an LN$_{\text{ff}}$-step

$$\pi : G = \overline{e_{k-1}}, \underline{\lambda\overline{x}.X(\overline{s'_p}) \cong t}, \overline{e_{k+1,n}} \Rightarrow_{[i],\theta} G'$$

for which there exists a substitution γ' such that

- $\mathcal{D}(\gamma') = (\mathcal{D}(\gamma) \setminus \{X\}) \cup \mathcal{R}ng(\theta)$,
- $X\gamma = X\theta\gamma'$,
- $\gamma = \gamma'$ $[\mathcal{D}(\gamma) \setminus \{X\}]$.

Then we can determine $\langle G', \gamma', \overline{R'}\rangle \in \text{Repr}(G')$ such that

$$\langle G, \gamma, \overline{R_n}\rangle \geq_{\text{A}} \langle G', \gamma', \overline{R'}\rangle, \langle G, \gamma, \mathcal{R}\rangle >_{\text{B}} \langle G', \gamma', \overline{R'}\rangle,$$

and hence $\langle G, \gamma, \overline{R_n}\rangle \succ \langle G', \gamma', \overline{R'}\rangle$. □

Lemma 25 *Let $G = \overline{e_n}$ with $e_k = t \cong t$, $\cong \in \{\approx, \rhd\}$ and $\gamma \in \mathcal{U}_{\mathcal{R}}(G)$. Then for any triple $\langle \overline{e_n}, \gamma, \overline{R_n}\rangle \in \text{Repr}(G)$ we have that $\langle G, \gamma, \overline{R_n}\rangle \succ \langle G', \gamma, \overline{R'_{n-1}}\rangle$, where $G' = \overline{e_{k-1}}, \overline{e_{k+1,n}}$ and $\overline{R'_{n-1}} = \overline{R_{k-1}}, \overline{R_{k+1,n}}$.*

Proof. Obvious.

Lemma 26 *If $G = G_1, \underline{e}, G_2 \Rightarrow_{\alpha,\theta} G'$ is an* LN$_{\text{ff}}$*-step with $\alpha \in \{[\text{ffs}], [\text{ffd}]\}$ and $\gamma\restriction_{\mathcal{V}_c(e)}$ is normalized then for any $\langle G, \gamma, \overline{R}\rangle \in \text{Repr}(G)$ there exists $\langle G', \gamma', \overline{R'}\rangle \in \text{Repr}(G')$ such that:*

(a) $\langle G, \gamma, \overline{R}\rangle \succ \langle G', \gamma', \overline{R'}\rangle$, *and*

(b) $\gamma = \theta\gamma'\ [\mathcal{V}(G)]$.

Proof. The proof is by case distinction on the syntactic structure of e.

1. Assume e is of the form $\lambda\overline{x}.X(\overline{y_m}) \rhd \lambda\overline{x}.Y(\overline{y'_n})$. Because $\gamma \in \mathcal{U}_{\mathcal{R}}(e)$, we have $\lambda\overline{x}.X(\overline{y_m})\gamma \to^*_{\mathcal{R}} \lambda\overline{x}.Y(y'_n)\gamma$. Also, $X\gamma$ is $\mathcal{R}$-normalized because $\mathcal{V}_c(e) = \{X\}$, and therefore $\lambda\overline{x}.X(\overline{y_m})\gamma_0$ is $\mathcal{R}$-normalized too. Then $\lambda\overline{x}.X(\overline{y_m})\gamma \to^*_{\mathcal{R}} \lambda\overline{x}.Y(y'_n)\gamma$ yields $\lambda\overline{x}.X(\overline{y_m})\gamma = \lambda\overline{x}.Y(y'_n)\gamma$, i.e. $\gamma \in \mathcal{U}(e)$. Since $\theta \in mgu(e)$, there exists a substitution γ' such that $\gamma = \theta\gamma'\ [\mathcal{V} \setminus \{X, Y\}]$.

   Because $G\gamma$ and $G'\gamma'$ differ only by a trivial equation, we conclude that $\gamma' \in \mathcal{U}_{\mathcal{R}}(G')$. Then $\langle G', \gamma', \overline{R'}\rangle \in \mathrm{Repr}(G')$ where $\overline{R'}$ is obtained from $\overline{R}$ by removing the rewrite derivation of length 0 corresponding to e. Obviously, $\langle G, \gamma, \overline{R}\rangle \succ \langle G', \gamma', \overline{R'}\rangle$.

2. The case when $e = \lambda\overline{x}.X(\overline{y_m}) \rhd \lambda\overline{x}.X(\overline{y'_m})$ can be proved in a similar way. $\square$

Corollary 1 *Lemmata 24 and 25 imply that the subcalculus* LN_r *of* $\mathrm{LN_{ff}}$ *obtained by removing the inference rules* [ffs] *and* [ffd] *is strongly complete. Also, the completeness property of* LN_r *holds for arbitrary* $\mathcal{R}$*-unifiers: the restriction to* $\mathcal{R}$*-normalized* $\mathcal{R}$*-unifiers is not necessary.*

We are ready now to define our first equation selection strategy for the calculus $\mathrm{LN_{ff}}$.

Definition 44 (strategy $\mathcal{S}_0$) *The strategy* $\mathcal{S}_0$ *for* $\mathrm{LN_{ff}}$*-derivations is the set of selection functions*

$$sel : \mathcal{H}is_{\mathrm{LN_{ff}}} \to \mathcal{E}q(\mathcal{F}, \mathcal{V}) \cup \{\bot\}$$

such that for any $\mathrm{LN_{ff}}$*-derivation* $\Pi \in \mathcal{H}is_{\mathrm{LN_{ff}}}(G')$ *and any equation* $e = s \cong t \in G'$ *with* $\cong \in \{\approx, \rhd\}$ *the following conditions holds:*

(c1) $(s \cong t) = sel(\Pi)$ *only if*

1. $s = t$ *or*
2. $\mathrm{prec}_\Pi(s \cong t) = \square$ *if* s, t *are patterns.*

Note that since it sel is selection function for $\mathrm{LN_{ff}}$, there must exist an $\mathrm{LN_{ff}}$-step $\pi : G' \Rightarrow G''$ upon which the selected equation is e.

In the rest of this subsection we prove that the calculus $\mathrm{LN_{ff}}$ with strategy $\mathcal{S}_0$ is complete.

We first prove an auxiliary lemma which will be used in the proof of completeness of $\mathrm{LN_{ff}}$ with strategy $\mathcal{S}_0$.

Lemma 27 *Let* $\Pi : G_0 \Rightarrow_{\theta_1} \ldots \Rightarrow_{\theta_N} G_N$ *be an* $\langle \mathrm{LN_{ff}}, \mathcal{S}_0 \rangle$*-derivation with:*

(a) $\gamma_0 \in \mathcal{U}^n_{\mathcal{R}}(G_0)$,

(b) for all $i \in \{0, \ldots, N-1\}$ *there exists* $\gamma_i \in \mathcal{U}_{\mathcal{R}}(G_i)$ *such that* $\gamma_i = \theta_{i+1}\gamma_{i+1}\ [\mathcal{V}(G_i)]$, *and*

(c) $e_N \in G_N$ *with* $\mathrm{prec}(e_N) = \Box$.

Then $\gamma_N \restriction_{\mathcal{V}_c(e)}$ *is* $\mathcal{R}$*-normalized.*

<u>Proof.</u> Let e_i be the equation in G_i from which e_N descends and $\delta_i = \theta_{i+1} \ldots \theta_N$ for $i \in \{0, \ldots, N\}$. We prove by induction on i a stronger result: $\gamma_N \restriction_{\mathcal{V}_c(e_i\delta_i)}$ is $\mathcal{R}$-normalized for any $i \in \{1, \ldots, N\}$. Since $\mathcal{V}_c(e_N\delta_N) = \mathcal{V}_c(e_N)$, this result implies that $\gamma_N \restriction_{\mathcal{V}_c(e_N)}$ is $\mathcal{R}$-normalized.

We first introduce the notion of linear ancestor. We say that e' is an *linear ancestor* of e if e' is a descendant of a linear descendant of e'.

Let $\pi_i : G_i \Rightarrow_{\alpha_i, \theta_i} G_{i+1}$ be the $(i+1)$-th $\mathrm{LN_{ff}}$-step of Π.

If $i = 0$ then $\delta_0 = \varepsilon$ and $\gamma_N \restriction_{\mathcal{V}_c(e_0\delta_0)}$ is $\mathcal{R}$-normalized because $\delta_0\gamma_N \restriction_{\mathcal{V}_c(e_0)} = \theta_1 \ldots \theta_N \gamma_N \restriction_{\mathcal{V}_c(e_0)} \overset{(b)}{=} \gamma_0 \restriction_{\mathcal{V}_c(e_0)}$ and γ_0 is $\mathcal{R}$-normalized.

We next show that $\theta_N \restriction_{\mathcal{V}_c(e_{i+1}\delta_{i+1})}$ is $\mathcal{R}$-normalized if $\theta_N \restriction_{\mathcal{V}_c(e_i\delta_i)}$ is $\mathcal{R}$-normalized.

Suppose e_i is not a linear ancestor of e_N. We show that $\mathcal{V}_c(e_{i+1}) \subseteq \mathcal{V}_c(e_i\theta_{i+1})$ by the following case distinction.

(a) $\alpha_i \in \{[\mathrm{on}], [\mathrm{ov}]\}$. Since e_i is not a linear ancestor, we have that e_{i+1} is a parameter-passing equation created by π_i and therefore $\mathcal{V}_c(e_{i+1}) \subseteq \mathcal{V}_c(e_i\theta_{i+1})$.

(b) $\alpha_i \notin \{[\mathrm{on}], [\mathrm{ov}]\}$. A simple analysis by case distinction on π_i reveals that $\mathcal{V}_c(e_{i+1}) \subseteq \mathcal{V}_c(e_i\theta_{i+1})$.

The induction hypothesis yields that $\theta_N \restriction_{\mathcal{V}_c(e_i\delta_i)}$ is $\mathcal{R}$-normalized. Because $\mathcal{V}_c(e_i\delta_i) = \mathcal{V}_c(e_i\theta_{i+1})\delta_{i+1} \supseteq \mathcal{V}_c(e_{i+1})\delta_{i+1} = \mathcal{V}_c(e_{i+1}\delta_{i+1})$, we conclude that $\theta_N \restriction_{\mathcal{V}_c(e_{i+1}\delta_{i+1})}$ is $\mathcal{R}$-normalized.

Suppose e_i is a linear ancestor of e_N. Then we can write $e_i = \lambda\overline{x}.s' \cong \lambda\overline{x}.t'$, $e_i\theta_{i+1} = \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t$ with $\cong \in \{\approx, \approx^{-1}, \rhd\}$, and

$$\begin{array}{lll}\Pi : & G_0 \Rightarrow^i_{\theta_1 \ldots \theta_i} & G_i = G'_i, \underline{e_i}, G''_i \\ & \Rightarrow_{\alpha, f(\overline{l_n}) \to r, \theta_{i+1}} & G_{i+1} = G'_i\theta_{i+1}, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G''_i\theta_{i+1} \\ & \Rightarrow^*_{\theta_{i+2} \ldots \delta_N} & G_N = G'_N, e_N, G''_N.\end{array}$$

with $\alpha \in \{[\mathrm{on}], [\mathrm{ov}]\}$. Since $\mathrm{prec}(e_N) = \Box$ and $\mathrm{LN_{ff}}$ is sound, we have

$$\lambda\overline{x}.s_k\delta_{i+1} \to^* \lambda\overline{x}.l_k\delta_{i+1}.$$

Then $\lambda\overline{x}.s'\delta_i = \lambda\overline{x}.f(\overline{s_n})\delta_{i+1} = \lambda\overline{x}.f(\overline{s_n\delta_{i+1}}) \to^* \lambda\overline{x}.f(\overline{l_n\delta_{i+1}}) \to \lambda\overline{x}.r\delta_{i+1}$, and therefore $\mathcal{V}_c(e_i\delta_i) = \mathcal{V}_c(\lambda\overline{x}.f(\overline{s_n})\delta_{i+1}) \supseteq \mathcal{V}_c(\lambda\overline{x}.r\delta_{i+1}) = \mathcal{V}_c(e_{i+1}\delta_{i+1})$. Since $\gamma_N\restriction_{\mathcal{V}_c(e_i\delta_i)}$ is $\mathcal{R}$-normalized because of the induction hypothesis, we conclude that $\gamma_N\restriction_{\mathcal{V}_c(e_{i+1}\delta_{i+1})}$ is $\mathcal{R}$-normalized, too. □

Lemma 28 (completeness of $\mathbf{LN_{ff}}$ with strategy $\mathcal{S}_0$) *Let G be a goal with $\mathcal{R}$-normalized solution θ. Then there exists an $\langle \mathrm{LN_{ff}}, \mathcal{S}_0\rangle$-refutation $\Pi : G = G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow_{\theta_2} \ldots \Rightarrow_{\theta_N} G_N = F$ such that:*

(a) $\theta = \theta_1 \ldots \theta_N\gamma\ [\mathcal{V}(G)]$ for some $\gamma \in \mathcal{U}_{\mathcal{R}}(F)$,

(b) for any $e \in G_i$ with $\mathrm{prec}(e) = \Box$, the substitution $\theta_{i+1}\ldots\theta_N\gamma\restriction_{\mathcal{V}_c(e)}$ is $\mathcal{R}$-normalized.

Proof. We prove by induction a stronger result, namely that there exists an $\langle \mathrm{LN_{ff}}, \mathcal{S}_0\rangle$-refutation $\Pi : G = G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow_{\theta_2} \ldots \Rightarrow_{\theta_N} G_N = F$ and a sequence of triples

$$\langle G_0, \gamma_0, \overline{R^0}\rangle \in \mathrm{Repr}(G_0), \ldots, \langle G_N, \gamma_N, \overline{R^N}\rangle \in \mathrm{Repr}(G_N)$$

with $\gamma_i = (\theta_{i+1}\ldots\theta_N\gamma)\restriction_{\mathcal{V}(G_i)}$ $(i = 0, \ldots, N)$, such that conditions (a), (b) and

(c) $\langle G_i, \gamma_i, \overline{R^i}\rangle \succ \langle G_{i+1}, \gamma_{i+1}, \overline{R^{i+1}}\rangle$ for all $i \in \{0, \ldots, N-1\}$.

hold. The proof is constructive. Assume that we succeeded to find an $\langle \mathrm{LN_{ff}}, \mathcal{S}_0\rangle$-derivation

$$\Pi_k : G = G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow_{\theta_2} \ldots \Rightarrow_{\theta_k} G_k$$

and a sequence of triples

$$\langle G_0, \gamma_0, \overline{R^0}\rangle \in \mathrm{Repr}(G_0), \ldots, \langle G_k, \gamma_k, \overline{R^k}\rangle \in \mathrm{Repr}(G_k)$$

for which the following conditions hold:

$P_1(k)$**:** $\theta = \theta_1 \ldots \theta_k\gamma_k\ [\mathcal{V}(G)]$ for some $\gamma_k \in \mathcal{U}_{\mathcal{R}}(G_k)$,

$P_2(k)$**:** for any $e \in G_i$ $(i = 0, \ldots, k)$ with $\mathrm{prec}_{\Pi_i}(e) = \Box$, the substitution $\gamma_i\restriction_{\mathcal{V}_c(e)}$ is $\mathcal{R}$-normalized, and

$P_3(k)$**:** $\langle G_i, \gamma_i, \overline{R^i}\rangle \succ \langle G_{i+1}, \gamma_{i+1}, \overline{R^{i+1}}\rangle$ for all $i \in \{0, \ldots, k-1\}$

If $\mathit{sel}(\Pi_k) = \bot$ for all $\mathit{sel} \in \mathcal{S}_0$ then Π_k is an $\langle \mathrm{LN_{ff}}, \mathcal{S}_0\rangle$-refutation, and we can choose $\Pi = \Pi_k$. Otherwise Π_k is not an $\langle \mathrm{LN_{ff}}, \mathcal{S}_0\rangle$-refutation and we can write $G_k = G', s \cong t, G''$ such that $s \cong t = \mathit{sel}(\Pi_k)$ for some $\mathit{sel} \in \mathcal{S}_0$.

We prove by case distinction on the syntactic structure of the equation $s \cong t$ that there exists an $\langle \mathrm{LN_{ff}}, \mathcal{S}_0 \rangle$-step

$$\pi_k : G_k = G', \underline{s \cong t}, G'' \Rightarrow_{\alpha,\theta_{k+1}} G_{k+1}$$

and a triple $\langle G_{k+1}, \gamma_{k+1}, \overline{R^{k+1}} \rangle \in \mathrm{Repr}(G_{k+1})$ such that the conditions $P_1(k+1)$, $P_2(k+1)$ and $P_3(k+1)$ hold.

(A) If $s \cong t$ is not a flex/flex equation or a flex/flex equation between identical terms, then by Lemmata 24 and 25, there exists an $\langle \mathrm{LN_{ff}}, \mathcal{S}_0 \rangle$-step $\pi_k : G_k = G'_k, \underline{s \cong t}, G''_k \Rightarrow_{\alpha,\theta_{k+1}} G_{k+1}$ for which the conditions
(1) $\langle G_k, \gamma_k, \overline{R^k} \rangle \succ \langle G_{k+1}, \gamma_{k+1}, \overline{R^{k+1}} \rangle$, and
(2) $\gamma_k = \theta_{k+1}\gamma_{k+1}\ [\mathcal{V}(G_k)]$
hold. Then $P_1(k) \wedge (2) \Rightarrow P_1(k+1)$, and $P_3(k) \wedge (1) \Rightarrow P_3(k+1)$. From Lemma 27 we obtain that that $P_2(k+1)$ holds too.

(B) Assume $s \cong t$ is a flex/flex equation with $s \neq t$. Then s, t are flex patterns and $\mathrm{prec}_{G_k}(s \cong t) = \Box$. From $P_2(k)$ we know that $\gamma_k \restriction_{\mathcal{V}_c(s \cong t)}$ is $\mathcal{R}$-normalized. By Lemma 26 we can perform an $\langle \mathrm{LN_{ff}}, \mathcal{S}_0 \rangle$-step

$$\pi_k : G_k = G'_k, \underline{s \cong t}, G''_k \Rightarrow_{\alpha,\theta_{k+1}} G_{k+1}$$

with $\alpha \in \{[\mathrm{ffs}], [\mathrm{ffd}]\}$ and
(1) $\langle G_k, \gamma_k, \overline{R^k} \rangle \succ \langle G_{k+1}, \gamma_{k+1}, \overline{R^{k+1}} \rangle$,
(2) $\gamma_k = \theta_{k+1}\gamma_{k+1}\ [\mathcal{V}(G_k)]$.
Then $P_1(k) \wedge (2) \Rightarrow P_1(k+1)$, and $P_3(k) \wedge (1) \Rightarrow P_3(k+1)$. From Lemma 27 we obtain that that $P_2(k+1)$ holds too.

Because $\succ$ is a well founded order, the sequence

$$\langle G_0, \gamma_0, \overline{R^0} \rangle \succ \langle G_1, \gamma_1, \overline{R^1} \rangle \succ \ldots$$

will eventually terminate with a triple $\langle G_N, \gamma_N, \overline{R^N} \rangle \in \mathrm{Repr}(G_N)$. Then

$$\Pi : G_0 \Rightarrow_{\theta_1} \ldots \Rightarrow_{\theta_N} G_N$$

is an $\langle \mathrm{LN_{ff}}, \mathcal{S}_0 \rangle$-refutation for which the conditions (a) and (b) hold. $\Box$

The calculus $\mathrm{LN_{ff}}$ with strategy $\mathcal{S}_0$ suffers of high nondeterminism between its inference rules. This nondeterminism is shown in Figs. 5.4, 5.5. We embellished the labels of the inference rules as follows:

- a label with subscript 1 denotes the corresponding inference rule for a flex/rigid equation, whereas a label with subscript 2 denotes the corresponding inference rule for a rigid/flex equation

$_{\mathrm{root}(s)}\backslash^{\mathrm{root}(t)}$	$\mathcal{V}(t)$	F_d	$F_c \cup \mathcal{BV}(t)$
$\mathcal{V}(s)$	$[\mathrm{del}]^1/[\mathrm{ffs}]^2/[\mathrm{ffd}]^2$	$[\mathrm{i}]_1^1, [\mathrm{p}]_1^1, [\mathrm{ov}]_1^1, [\mathrm{on}]_2^1$	$[\mathrm{i}]_1^1, [\mathrm{p}]_1^1, [\mathrm{ov}]_1^1$
F_d	$[\mathrm{i}]_2^1, [\mathrm{p}]_2^1, [\mathrm{ov}]_2^1, [\mathrm{on}]_1^1$	$[\mathrm{del}]^1/([\mathrm{on}]_1^2, [\mathrm{on}]_2^2, [\mathrm{d}]^2)$	$[\mathrm{on}]^1$
$F_c \cup \mathcal{BV}(s)$	$[\mathrm{i}]_2^1, [\mathrm{p}]_2^1, [\mathrm{ov}]_2^1$	$\times$	$[\mathrm{del}]^1/[\mathrm{d}]^2$

Fig. 5.4: Inference rules of $\mathrm{LN}_{\mathrm{ff}}$ for equation $s \approx t$ selected with $sel \in \mathcal{S}_0$

$_{\mathrm{root}(s)}\backslash^{\mathrm{root}(t)}$	$\mathcal{V}(t)$	F_d	$F_c \cup \mathcal{BV}(t)$
$\mathcal{V}(s)$	$[\mathrm{del}]^1/[\mathrm{ffs}]/[\mathrm{ffd}]$	$[\mathrm{i}]_1, [\mathrm{p}]_1, [\mathrm{ov}]_1$	$[\mathrm{i}]_1, [\mathrm{p}]_1, [\mathrm{ov}]_1$
F_d	$[\mathrm{del}]^1/([\mathrm{i}]_2, [\mathrm{p}]_2, [\mathrm{ov}]_2^1, [\mathrm{on}]_1)$	$[\mathrm{on}]_1, [\mathrm{d}]$	$[\mathrm{on}]_1$
$F_c \cup \mathcal{BV}(s)$	$[\mathrm{i}]_2, [\mathrm{p}]_2$	$\times$	$[\mathrm{del}]^1/[\mathrm{d}]$

Fig. 5.5: Inference rules of $\mathrm{LN}_{\mathrm{ff}}$ for equation $s \rhd t$ selected with $sel \in \mathcal{S}_0$

- superscripts denote the priority of applying an inference rule. The highest priority is 1. [del] is a rule with highest priority.

In the case of lazy narrowing calculi this nondeterminism is usually ***don't know***, i.e. in order to guarantee completeness we usually have to consider all the possible choices. For example, for a flex/rigid equation $s \approx t$ with $\mathrm{root}(t) \in \mathcal{F}_d$ we have to consider the inference rules $[\mathrm{i}]_1$, $[\mathrm{p}]_1$, $[\mathrm{ov}]_1$, $[\mathrm{on}]_2$. One exception is the rule [del], which can be applied deterministically (because of Lemma 25).

In particular, $\mathrm{LN}_{\mathrm{ff}}$ has high don't know nondeterminism between the inference rules for flex/rigid equations. There are at least three inference rules that have to be considered for flex/rigid equations: $[\mathrm{ov}]_1$, $[\mathrm{i}]_1$ and $[\mathrm{p}]_1$. In particular, it is desirable to avoid $[\mathrm{ov}]_1$ because it does not restrict in any way the rewrite rule to be used.

Since $\mathrm{LN}_{\mathrm{ff}}$ with strategy $\mathcal{S}_0$ is complete regardless of the order of selecting the equations which are not flex/flex (Lemma 24), we can further refine the strategy $\mathcal{S}_0$ to avoid solving flex/rigid equations as much as possible.

We propose to use selection functions which satisfy (c1) and

(c2) a flex/rigid equation $e \in G$ with $\mathrm{prec}(e) \neq \Box$ is selected only if all the other equations of G are either flex/flex, or flex/rigid with precursors.

We denote this class of selection functions with $\mathcal{S}_n$. Since $\mathcal{S}_n \subseteq \mathcal{S}_0$, any $\langle \mathrm{LN}_{\mathrm{ff}}, \mathcal{S}_n \rangle$-refutation is also an $\langle \mathrm{LN}_{\mathrm{ff}}, \mathcal{S}_0 \rangle$-refutation. Note that an $\langle \mathrm{LN}_{\mathrm{ff}}, \mathcal{S}_0 \rangle$-refutation is of the form $G \Rightarrow_\theta^* F$ where F is a flex/flex goal without equations of the form $e = s \rhd t$ with s, t flex patterns and $\mathrm{prec}_F(e) = \Box$.

By obeying (c2) we delay the selection of a flex/rigid equation with precursors as much as possible because we want to avoid the nondeterminism of applying the rules $[\mathrm{i}]_1$, $[\mathrm{p}]_1$ and [ov]. We may avoid this nondeterminism if by solving the other equations first, we can transform a flex pattern/rigid equation with precursors into a flex/rigid equation without precursors, or into a rigid/rigid equation.

Like in the first-order case, it is useful to distinguish the descendants of parameter-passing equations from the other equations. In this paper, when we want to emphasize that an equation $s \rhd t$ is a (descendant of a) parameter-passing equation, we write $s \blacktriangleright t$ instead.

5.4 Outermost Narrowing at Variable Position

Our object of study in this section is the calculus $\mathrm{LN_{ff}}$ with strategy $\mathcal{S}_n$. By adopting the strategy $\mathcal{S}_n$ we delay the selection of flex/rigid equations as much as possible. We investigate restrictions under which the inference rule [ov] can be eliminated for flex/rigid selected equations, without losing completeness.

Lemma 29 *Let $G_0 = G, e, G'$ be a goal and $e = \lambda\overline{x}.X(\overline{y}) \rhd \lambda\overline{x}.t$ with $\lambda\overline{x}.t$ a rigid term and $\lambda\overline{x}.X(\overline{y})$ a pattern. Then for any triple $\langle G_0, \gamma_0, \overline{R}\rangle \in \mathrm{Repr}(G_0)$ such that $\gamma_0\restriction_{\mathcal{V}_c(e)}$ is $\mathcal{R}$-normalized there exists an $\langle \mathrm{LN_{ff}}, \mathcal{S}_n\rangle$-step $\pi : G_0 = G, \underline{e}, G' \Rightarrow_{\alpha,\theta} G_1$ with $\alpha \neq [\mathrm{ov}]$ and a triple $\langle G_1, \gamma_1, \overline{R'}\rangle \in \mathrm{Repr}(G_1)$ such that:*

(a) $\langle G_0, \gamma_0, \overline{R}\rangle \succ \langle G_1, \gamma_1, \overline{R'}\rangle$, *and*

(b) $\gamma_0 = \theta\gamma_1\ [\mathcal{V}(G_0)]$.

<u>**Proof.**</u> By Lemma 24, there exists an $\mathrm{LN_{ff}}$-step

$$\pi : G_0 = G, \underline{e}, G' \Rightarrow_{\alpha,\theta} G_1$$

which satisfies the conditions (a) and (b). Then π is an $\langle \mathrm{LN_{ff}}, \mathcal{S}_n\rangle$-step as well. From the assumption that $\gamma_0\restriction_{\mathcal{V}_c(e)}$ is $\mathcal{R}$-normalized we deduce that $X\gamma_0$ is $\mathcal{R}$-irreducible, and thus $\lambda\overline{x}.X(\overline{y})\gamma_0$ is also $\mathcal{R}$-irreducible. From the proof of Lemma 24 for this case results that $\alpha \neq [\mathrm{ov}]$. $\square$

We define LN_1 as the calculus obtained from $\mathrm{LN_{ff}}$ by modifying the side conditions of the inference rule [ov] as follows:

[ov] *outermost narrowing at variable position*

$$\frac{G, \lambda\overline{x}.X(\overline{s_m}) \rhd \lambda\overline{x}.t, G'}{(G, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \rhd \lambda\overline{x}.l_n}, \lambda\overline{x}.r \rhd \lambda\overline{x}.t, G')\theta}$$

if

- $\lambda\overline{x}.X(\overline{s_m})$ is a flex-pattern only if $\mathrm{prec}(\lambda\overline{x}.X(\overline{s_m}) \rhd \lambda\overline{x}.t) \neq \Box$,
- $\lambda\overline{x}.t$ is rigid,
- $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule, and
- $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types.

Lemma 30 (completeness of LN_1 with strategy $\mathcal{S}_n$) *Let G be a goal and $\gamma \in \mathcal{U}^n_{\mathcal{R}}(G)$. Then there exists an $\langle \mathrm{LN}_1, \mathcal{S}_n\rangle$-refutation $G \Rightarrow^*_\theta F$ such that $\gamma = \theta\gamma'$ $[\mathcal{V}(G)]$ for some $\gamma' \in \mathcal{U}_{\mathcal{R}}(F)$.*

Proof. Similar to the proof of Lemma 28, but we use Lemma 29 instead of Lemma 24.

5.5 Eager Variable Elimination

In this subsection we address the nondeterminism of LN_1 with strategy $\mathcal{S}_n$ due to the selection of the inference rule to be applied to rigid/flex descendants of parameter-passing equations. More precisely, we want to reduce the nondeterminism between the inference rules of LN_1 with strategy $\mathcal{S}_n$ for selected equations of the form $\lambda\overline{x}.f(\overline{s_n}) \blacktriangleright \lambda\overline{x}.X(\overline{y})$ where $f \in \mathcal{F}_d$ and $\lambda\overline{x}.X(\overline{y})$ is a flex pattern. This way of reducing the nondeterminism of LN_1 was inspired by the eager variable elimination problem of the calculus LNC. In the first-order case it is shown that by applying the inference rule

$$\text{[v]} \qquad \frac{G, s \blacktriangleright X, G'}{(G, G')\{X \mapsto s\}} \text{ if } X \notin \mathcal{V}(s)$$

prior to other applicable inference rules is a complete method (with respect to the equation selection function sel_{left}) for orthogonal TRSs.

An attempt to generalize this result to the calculus LN_1 with strategy $\mathcal{S}_0$ for orthogonal PRSs raises the following questions:

1. Can we generalize to orthogonal PRSs the essential properties of orthogonal TRS?
2. What it the higher-order version of a first-order equation of the form $s \blacktriangleright X$ with $X \notin \mathcal{V}(s)$?

3. What is the counterpart of the [v]-rule of LNC in the calculus LN_1?

We give here a list of possible answers.

1. The crucial property that makes the eager variable elimination method complete for orthogonal TRSs is the validity of the standardization theorem for orthogonal TRSs [MOI96]. Roughly speaking, the standardization theorem states that if $\mathcal{R}$ is orthogonal and $s \to_{\mathcal{R}}^{*} t$ then there exists an outside-in reduction derivation from s to t. Recently, van Oostrom [vO96], succeeded to prove the standardization theorem for orthogonal PRS.

2. Intuitively, the higher-order counterpart of a first-order variable is a higher-order pattern. From this point of view, the higher-order eager variable elimination problem should address the nondeterminism between the inference rules which are applicable to selected equations of the form $\lambda\overline{x}.f(\overline{s_n}) \blacktriangleright \lambda\overline{x}.X(\overline{y})$ with $f \in \mathcal{F}_d$.

3. LN_1 has no variable elimination rule. We observe that we can simulate a variable elimination step with a finite sequence of [i]- and [p]-steps. With this understanding, higher-order eager variable elimination addresses the possibility to eliminate the application of $[\text{on}]_{\triangleright}$ to flex/rigid descendants of parameter-passing equations.

In the rest of this subsection we will prove that if $\mathcal{R}$ is an orthogonal PRS then the application of $[\text{on}]_{\triangleright}$ can be dropped for selected descendants of parameter-passing equations without influencing the completeness property of LN_1 with strategy $\mathcal{S}_n$.

Preliminaries

We will generalize the first-order eager variable elimination method to LN_1 with the help of outside-in reduction derivations.

Definition 45 (eager variable elimination) *An* $\langle \text{LN}_1, \mathcal{S}_n\rangle$*-derivation* Π respects the eager variable elimination method *if* [on] *is never applied to rigid/flex equations of the form* $\lambda\overline{x}.f(\overline{s_n}) \blacktriangleright \lambda x.X(\overline{y})$ *with* $f \in \mathcal{F}_d$.

The notion of outside-in reduction derivations for orthogonal PRSs is carried over from that for first order TRSs [Suz96].

Definition 46 (outside-in reduction derivation) *Let* $\mathcal{R}$ *be an orthogonal PRS. A* $\mathcal{R}$*-reduction derivation of equations is called* outside-in *if every subderivation* $oe \to_p e_0 \to_{p_1} \cdots \to_{p_n} e_n \to_{q,l\to r} e'$ *satisfies the following condition: if* $p > q > \varepsilon$ *and all* p_i $(1 \le i \le n)$ *are disjoint from* p *then* p/q *is above or disjoint from any free variable position in* l.

The only difference from the first-order case given in [MO98] is the presence of bound variables below the free variables. The definition above states that the flex subterms of a higher-order pattern, called *binding holes* by Oostrom [vO96], are regarded as mere variables.

Theorem 7 (Oostrom [vO96]) *For any rewrite derivation $s \to_{\mathcal{R}}^{*} t$ by an orthogonal PRS $\mathcal{R}$, there exists an outside-in rewrite derivation from s to t.* $\square$

Completeness

We follow the same line of reasoning as in [MO98] to show that the eager variable elimination method for parameter-passing equations preserves completeness of LN_1 with strategy $\mathcal{S}_n$: first we introduce a property of rewrite derivations which holds for any outside-in rewrite derivation. Next we show that this property is preserved by Lemmata 24, 25 and 29. This result motivates the possibility to inhibit the application of $[\mathrm{on}]_{\triangleright}$ to equations of the form $\lambda\overline{x}.f(\overline{s_n}) \blacktriangleright \lambda\overline{x}.X(\overline{y})$ with $f \in \mathcal{F}_d$.

First we introduce a class of restricted outside-in reduction derivations.

Definition 47 (property $\mathcal{P}_{HO}$) *Let $\mathcal{R}$ be an orthogonal PRS and*

$$R : s\theta \rhd t\theta \to_{\mathcal{R}}^{*} t\theta \rhd t\theta$$

an outside-in $\mathcal{R}$-rewrite derivation. Then we say R has property $\mathcal{P}_{HO}$ *if every reduction step in it satisfies the following condition: if a position $1{\cdot}p$ is rewritten in the reduction step and later steps do not take place above $1{\cdot}p$, then p is above or disjoint from any free variable position in t.*

Lemma 31 *Let $G = \overline{e_n}$ be a goal, $\Pi \in \mathcal{H}is_{\mathrm{LN}_1}(G)$ and $\langle G, \gamma, \overline{R_n}\rangle \in \mathrm{Repr}(G)$ such that any R_i is an outside-in rewrite derivation and R_i has property $\mathcal{P}_{HO}$ if e_i is a parameter-passing equation. Suppose $sel(\Pi) = e_k$ for some selection function $sel \in \mathcal{S}_n$. Then there exists an $\langle \mathrm{LN}_1, \mathcal{S}_n\rangle$-step $\pi : \overline{e_{k-1}}, \underline{e_k}, \overline{e_{k+1,n}} \Rightarrow_{\theta} G' = \overline{e'_m}$ and $\langle G', \gamma', \overline{R'}\rangle \in \mathrm{Repr}(G_1)$ such that*

(a) $\langle G, \gamma, \overline{R_n}\rangle \succ \langle G', \gamma', \overline{R'_m}\rangle$,

(b) $\gamma = \theta\gamma'$ $[\mathcal{V}(G)]$, *and*

(c) if $j \in \{1, \ldots, m\}$ such that e'_j is a parameter-passing equation then R'_j has property $\mathcal{P}_{HO}$.

The proof is done by an easy but tedious case analysis on the transformations described in the proofs of Lemmata 24, 25 and 29.

Lemma 32 *If $\mathcal{R}$ is an orthogonal PRS then the eager variable elimination method preserves the completeness of the calculus* LN_1 *with strategy* $\mathcal{S}_n$ *for* $\mathcal{R}$*-normalized solutions.*

Proof. Let$\gamma \in \mathcal{U}^n_{\mathcal{R}}(G)$. Because $\mathcal{R}$ is orthogonal, there exists $\langle G, \gamma, \overline{R} \rangle \in \mathrm{Repr}(G)$ such that $\overline{R}$ is a sequence of outside-in $\mathcal{R}$-derivations. A successive application of Lemma 31 yields a sequence

$$\langle G, \gamma, \overline{R} \rangle = \langle G_0, \gamma_0, \overline{R^0} \rangle \succ \langle G_1, \gamma_1, \overline{R^1} \rangle \succ \ldots \succ \langle G_n, \gamma_n, \overline{R^n} \rangle \succ \ldots$$

and a corresponding $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-derivation

$$G = G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow_{\theta_2} \ldots \Rightarrow_{\theta_n} G_n \Rightarrow_{\theta_{n+1}} \cdots$$

such that for any $i \geq 0$

(a) $\langle G_i, \gamma_i, \overline{R^i} \rangle \succ \langle G_{i+1}, \gamma_{i+1}, \overline{R^{i+1}} \rangle$,

(b) $\gamma_i = \theta_{i+1}\gamma_{i+1}\ [\mathcal{V}(G_i)]$, and

(c) any $R'_j \in \overline{R^i}$ has property $\mathcal{P}_{HO}$ if the corresponding equation $e'_j \in G_i$ is a parameter-passing equation.

Because $\succ$ is well-founded, the sequence of triples will eventually terminate with a triple $\langle G_N, \theta_N, \overline{R^N} \rangle$ where G_N is a flex/flex goal such that $sel(G_N) = \perp$ for all $sel \in \mathcal{S}_n$. Correspondingly, we obtain an $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-refutation

$$\Pi : G = G_0 \Rightarrow_{\theta_1} G_1 \Rightarrow_{\theta_2} \ldots \Rightarrow_{\theta_n} G_N.$$

From conditions (b) for every $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-step, we deduce that $\gamma = \theta\gamma_N\ [\mathcal{V}(G)]$ where $\theta := \theta_1 \ldots \theta_N$.

Assume $e = \lambda\overline{x}.f(\overline{s_n}) \blacktriangleright \lambda\overline{x}.X(\overline{y}) \in G_i$ is a rigid/flex parameter-passing equation selected in Π, with $f \in \mathcal{F}_d$. According to our construction of Π, the rewrite derivation corresponding to e in $\overline{R^i}$ has property $\mathcal{P}_{HO}$. This implies that no rewrite step in $\mathcal{R}_i$ takes place to the root of the left-hand side. From the proof of Lemma 24, subcase 2 results by contraposition that the $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-step applied to G_i is not [on]. Thus Π respects the eager variable elimination method.

Hence we can the adopt the eager variable elimination method in the calculus LN_1 with strategy $\mathcal{S}_n$ without losing completeness. We denote the newly obtained calculus by LN_1^{ev}. □

5.6 Lazy Narrowing for Left-Linear Pattern Rewrite Systems

In the sequel we assume that $\mathcal{R}$ is a left-linear PRS. The following lemma captures some of the essential properties of $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-derivations for left-linear PRSs. It is the counterpart for LN_1 with strategy $\mathcal{S}_n$ of Lemma 3.1 in [MO98] for LNC with strategy $\mathcal{S}_{\mathrm{left}}$.

Lemma 33 *Let $\mathcal{R}$ be a left-linear PRS and $\Pi\colon G \Rightarrow^* G', s \blacktriangleright t, G''$ an $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-derivation. Then:*

(1) $(\mathcal{V}(G', s) \cup \mathcal{V}(G\theta)) \cap \mathcal{V}(t) = \emptyset$,

(2) For any equation $e \in G''$, if $\mathcal{V}(e) \cap \mathcal{V}(t) \neq \emptyset$ then $s \blacktriangleright t \in \mathrm{prec}(e)$, and

(3) t is a linear pattern.

Proof. We will make use of the following general property of an LN_1-step:

(*) If $G \Rightarrow_\sigma G'$ is an LN_1-step then σ is a linear pattern substitution with $\mathcal{R}ng(\sigma)$ a set of fresh variables.

Because $s \blacktriangleright t$ is a descendant of a parameter-passing equation, Π can be written as:

$$\Pi : G \Rightarrow^*_{\theta'} G_0 \Rightarrow_{\alpha,\sigma} G_1 = (G'_1, e, G''_1) \Rightarrow^* G', s \blacktriangleright t, G''$$

where

- $\alpha \in \{[\mathrm{on}], [\mathrm{ov}]\}$,
- e is a parameter-passing equation generated by the $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-step $\pi : G_0 = (G'_0, \underline{e_0}, G''_0) \Rightarrow_{\alpha,\sigma} G_1$ of Π (i.e., $e \in \mathrm{desc}_\pi(e_0)$ but $e \neq \mathrm{ldesc}_\pi(e_0)$)

Let $\Pi' : G_1 \Rightarrow^* G', s \blacktriangleright t, G''$ be the $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-subrefutation of π starting from G_1.

The proof proceeds by induction on the length $|\Pi'|$ of Π'.

Case 1. If $|\Pi'| = 0$ then either

$$\begin{aligned}\pi :\ & G_1 = (G'_1, \lambda\overline{x}.f(\overline{s_n}) \cong u, G''_1) \Rightarrow_{[\mathrm{on}], f(\overline{l_n}) \to r} \\ & \underbrace{G'_1, \overline{\lambda\overline{x}.s_p \blacktriangleright \lambda\overline{x}.l_p}}_{G'}, \underbrace{\lambda\overline{x}.s_{p+1} \blacktriangleright \lambda\overline{x}.l_{p+1}}_{s \blacktriangleright t}, G''\end{aligned}$$

or

$$\pi : \quad G_1 = (G_1', \lambda\overline{x}.X(\overline{s_m}) \cong u, G_1'') \Rightarrow_{[\mathrm{ov}],\sigma,f(\overline{l_n})\to r} \underbrace{G_1'\sigma, \overline{\lambda\overline{x}.H_p(\overline{s_m\sigma}) \blacktriangleright \lambda\overline{x}.l_p}}_{G'}, \underbrace{\lambda\overline{x}.H_{p+1}(\overline{s_m\sigma}) \blacktriangleright \lambda\overline{x}.l_{p+1}}_{s\blacktriangleright t}, G''$$

with $p < n$ and $\sigma = \{X \mapsto \lambda\overline{x_m}.f(\overline{H_n(\overline{x_m})})\}$. Because $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted left-linear rule, we learn that in the first situation the term $t(= \lambda\overline{x}.l_{p+1})$ is a linear pattern which has no variables in common with G_1, $\overline{\lambda\overline{x}.l_p}$, $\overline{\lambda\overline{x}.s_p}$ and $\mathcal{V}(G\theta)$. In the second situation we can furthermore make use of (*) to infer that t has no variables in common with $G_1\sigma, \overline{\lambda\overline{x}.H_p(\overline{s_m\sigma}) \blacktriangleright \lambda\overline{x}.l_p}$, s and $\mathcal{V}(G\theta)$. Thus (1) holds in both situations. (3) is an immediate consequence of the fact that $\lambda\overline{x}.f(\overline{l_n})$ is a linear pattern (because $\mathcal{R}$ is a left-linear PRS). Note that the only equation of G'' which may have variables in common with t is $\lambda\overline{x}.r \cong u$. Then $s \blacktriangleright t \in \mathrm{prec}(\lambda\overline{x}.r \cong u)$ and (2) holds, too.

Case 2. Suppose $|\Pi'| > 0$. Then we can write Π as

$$G \Rightarrow^*_{\theta'} \underbrace{G_0 \Rightarrow_{\alpha,\sigma} G_1}_{\pi} = \underbrace{(G_1', e, G_1'') \Rightarrow^*_{\theta''} G_2', s' \blacktriangleright t', G_2'' \Rightarrow_{\alpha',\sigma'} G', s \blacktriangleright t, G''}_{\Pi'}$$

such that $s \blacktriangleright t$ descends from $s' \blacktriangleright t'$. Let $G_2 = (G_2', s' \blacktriangleright t', G_2'')$ and $\theta_1 = \theta'\sigma\theta''$. From the induction hypothesis we have

(4) $(\mathcal{V}(G_2', s') \cup \mathcal{V}(G\theta_1)) \cap \mathcal{V}(t') = \emptyset$,

(5) if $e'' \in G_2''$ with $\mathcal{V}(e'') \cap \mathcal{V}(t') \neq \emptyset$ then $s'\blacktriangleright t' \in \mathrm{prec}(e'')$, and

(6) t' is a linear pattern.

Let π' be the last step of Π'.

Subcase 2.1 Assume $s' \blacktriangleright t'$ is selected in π'. Then α' can be:

[on]/[ov]: In this case $s' = \lambda\overline{x}.h(\overline{s_m'})$ with $h \in \mathcal{V}(s') \cup \mathcal{F}_d$. Assume the variant of the $\overline{x}$-lifted rule used in the last step is $f(\overline{l_n}) \to r$. Then $G' = G_2'\sigma', \overline{\lambda\overline{x}.s_p'' \blacktriangleright \lambda\overline{x}.l_p}$ for some $p \leq n$, where either

- $\overline{s_m''} = \overline{s_m'}$, $\sigma' = \varepsilon$ and $n = m$ if $h = f$, or
- $\overline{s_n''} = \overline{H_n(\overline{s_m'\sigma'})}$ and $\sigma' = \{h \mapsto \lambda\overline{x_m}.f(\overline{H_n(\overline{x_m})})\}$ if $h \in \mathcal{V}(s')$.

Assume first that $p < n$. Then $s \blacktriangleright t = \lambda\overline{x}.s_{p+1}'' \blacktriangleright \lambda\overline{x}.l_{p+1}$ and the only equation in G'' which may have variables in common with $t(= \lambda\overline{x}.l_{p+1})$ is $\lambda\overline{x}.r \blacktriangleright t'\sigma'$. In this case (2) holds because, by the definition of precursor, $s \blacktriangleright t \in \mathrm{prec}(\lambda\overline{x}.r \blacktriangleright t'\sigma')$. The term $t(=$

$\lambda\overline{x}.l_{p+1})$ is a linear pattern because $\mathcal{R}$ is left-linear, and thus (3) holds. To prove (1) we note that:

$(\mathcal{V}(G',s) \cup \mathcal{V}(G\theta)) \cap \mathcal{V}(t) = (\mathcal{V}(G'_2\sigma', \overline{\lambda\overline{x}.s''_p \blacktriangleright \lambda\overline{x}.l_p}) \cup \mathcal{V}(G\theta_1\sigma')) \cap \mathcal{V}(\lambda\overline{x}.l_{p+1}) \subseteq ((\mathcal{V}(G'_1, s') \cup \mathcal{V}(G\theta_1) \cup \mathcal{R}ng(\sigma')) \cap \mathcal{V}(\lambda\overline{x}.l_{p+1})) \cup (\mathcal{V}(\overline{\lambda\overline{x}.l_p}) \cap \mathcal{V}(\lambda\overline{x}.l_{p+1})) = \emptyset \cup \emptyset = \emptyset$. Thus (1) holds as well.

If $p = n$ then $s \blacktriangleright t = \lambda\overline{x}.r \blacktriangleright t'\sigma'$ and $G' = G'_2\sigma', \overline{\lambda\overline{x}.s''_n \blacktriangleright \lambda\overline{x}.l_n}$. In this case (3) follows from (6), (*) and (7) $|\mathcal{D}(\sigma')| \leq 1$.

For proving (1) we note that $\mathcal{V}(G',s) \cap \mathcal{V}(t) = \mathcal{V}(G'_2\sigma', \overline{\lambda\overline{x}.s''_p \blacktriangleright \lambda\overline{x}.l_p}, \lambda\overline{x}.r) \cap \mathcal{V}(t'\sigma') \subseteq (\mathcal{V}(G'_2\sigma', \lambda\overline{x}.s'\sigma') \cap \mathcal{V}(t'\sigma')) \cup (\mathcal{V}(\overline{\lambda\overline{x}.l_n}, \lambda\overline{x}.r) \cap \mathcal{V}(t'\sigma')) = \emptyset$ because $\mathcal{V}(G'_2\sigma', \lambda\overline{x}.s'\sigma') \cap \mathcal{V}(t\sigma') \overset{(4,*,7)}{=} \emptyset$ and $\mathcal{V}(\overline{\lambda\overline{x}.l_n}, \lambda\overline{x}.r) \cap \mathcal{V}(t') = \emptyset$. Moreover, $\mathcal{V}(G\theta) \cap \mathcal{V}(t) = \mathcal{V}(G\theta_1\sigma') \cap \mathcal{V}(t'\sigma') \overset{(4,*,7)}{=} \emptyset$. Thus (1) holds.

For proving (2) assume that $e' \in G''$ satisfies $\mathcal{V}(e') \cap \mathcal{V}(t) \neq \emptyset$. Then e' descends from some equation $e'' \in G''_2$ and $e' = e''\sigma'$. This implies that $\mathcal{V}(e''\sigma') \cap \mathcal{V}(t'\sigma) \neq \emptyset$, and thus $\mathcal{V}(e'') \cap \mathcal{V}(t') \neq \emptyset$. From (5) we conclude that $s' \blacktriangleright t' \in \text{prec}(e'')$, and (2) follows from the definition of precursor.

[d] Then $s' = \lambda\overline{x}.v(\overline{s_k}), t' = \lambda\overline{x}.v(\overline{t_k})$ with $v \in \{\overline{x}\} \cup \mathcal{F}$, and $s \blacktriangleright t$ is a descendant of the form $\lambda\overline{x}.s_p \blacktriangleright \lambda\overline{x}.t_p$ with $1 \leq p \leq k$. (1) follows from (4) and (6), and (3) from (6). For proving (2), assume $e' \in G''$ such that $\mathcal{V}(e') \cap \mathcal{V}(t) \neq \emptyset$. We note that e' is not a descendant of $s' \blacktriangleright t'$ because $\mathcal{V}(\lambda\overline{x}.t_p) \cap \mathcal{V}(\lambda\overline{x}.s_j \blacktriangleright \lambda\overline{x}.t_j) \overset{(4,6)}{=} \emptyset$ for any $j > p$. Therefore e' descends from some equation $e'' \in G''_2$ and $\mathcal{V}(e'') \cap \mathcal{V}(t) \neq \emptyset$. By (5), $s' \blacktriangleright t' \in \text{prec}(e'')$, which implies (2).

[i] Because of (4), $\mathcal{V}(t') \cap \mathcal{V}(G'_2, s') = \emptyset$,and thus π' is either

$$G'_2, \lambda\overline{x}.X(\overline{s'_q}) \blacktriangleright \lambda\overline{x}.h(\overline{t_k}), G''_2 \Rightarrow_{[\mathrm{i}],\sigma'} G'_2\sigma', \overline{\lambda\overline{x}.H_k(\overline{s'_q\sigma'}) \blacktriangleright \lambda\overline{x}.t_k}, G''_2\sigma'$$

or

$$G'_2, \lambda\overline{x}.h(\overline{t_k}) \blacktriangleright \lambda\overline{x}.X(\overline{y_q}), G''_2 \Rightarrow_{[\mathrm{i}],\sigma'} G'_2, \overline{\lambda\overline{x}.t_k \blacktriangleright \lambda\overline{x}.H_k(\overline{y_q})}, G''_2\sigma'$$

where $X \in \mathcal{V} \setminus \{\overline{x}\}$, $\sigma' = \{X \mapsto \lambda\overline{x_q}.h(\overline{H_k(\overline{x_q})})\}$, $h \in \mathcal{F}$, and $\overline{y_q}$ is a sequence of distinct bound variables.

In the first sub-case we have $s \blacktriangleright t = \lambda\overline{x}.H_p(\overline{s'_q\sigma'}) \blacktriangleright \lambda\overline{x}.t_p$ for some $p \in \{1,\ldots, k\}$ and $G' = (G'_2\sigma', \overline{\lambda\overline{x}.H_{p-1}(\overline{s'_q\sigma'}) \blacktriangleright \lambda\overline{x}.t_{p-1}})$. Then (3) follows from (6). To prove (1) we observe that $\mathcal{V}(G',s) \cap \mathcal{V}(t) = \mathcal{V}(G'_2\sigma', \overline{\lambda\overline{x}.H_{p-1}(\overline{s'_q\sigma'}) \blacktriangleright \lambda\overline{x}.t_{p-1}}) \cap \mathcal{V}(\lambda\overline{x}.t_p) \subseteq (\mathcal{V}(G'_2\sigma', s'\sigma') \cap \mathcal{V}(t'))$

$\cup\ (\mathcal{V}(\overline{\lambda\overline{x}.t_p}) \cap \mathcal{V}(\lambda\overline{x}.t_{p+1})) \overset{(6)}{=} \mathcal{V}(G_2'\sigma', s'\sigma') \cap \mathcal{V}(t') \subseteq (\mathcal{V}(G_2', s') \cup \mathcal{R}ng(\sigma')) \cap \mathcal{V}(t') \overset{(4,*)}{=} \emptyset$ and $\mathcal{V}(G\theta) \cap \mathcal{V}(t) \subseteq (\mathcal{V}(G\theta_1) \cup \mathcal{R}ng(\sigma')) \cap \mathcal{V}(t') \overset{(4,*)}{=} \emptyset$.

For proving (2) note that $\mathcal{V}(t) \cap \mathcal{V}(\lambda\overline{x}.H_j(\overline{s_q'\sigma}) \blacktriangleright \lambda\overline{x}.t_j) \overset{(4,6,*)}{=} \emptyset$ for all $j > p$. Thus, if $e' \in G''$ satisfies $\mathcal{V}(e') \cap \mathcal{V}(t) \neq \emptyset$ then e' descends from an equation $e'' \in G_2''$. This implies that $e' = e''\sigma'$, and hence $\emptyset \neq \mathcal{V}(e') \cap \mathcal{V}(t) = \mathcal{V}(e''\sigma') \cap \mathcal{V}(t') = \mathcal{V}(e''\sigma') \cap \mathcal{V}(t') = (\mathcal{V}(e''\sigma') \cap \mathcal{V}(t'\sigma')$. This implies that $\mathcal{V}(e'') \cap \mathcal{V}(t') \neq \emptyset$. By (5) we get $s' \blacktriangleright t' \in \mathrm{prec}(e'')$, and (2) follows from the definition of precursor.

In the second sub-case we have $s \blacktriangleright t = \lambda\overline{x}.t_p \blacktriangleright \lambda\overline{x}.H_p(\overline{y_q})$ for some $p \in \{1, \ldots, k\}$ and $G' = G_2', \overline{\lambda\overline{x}.t_{p-1} \blacktriangleright \lambda\overline{x}.H_{p-1}(\overline{y_q})}$. Then (3) holds trivially and (1) results from the observation that $\mathcal{V}(G', s) \cap \mathcal{V}(t) \overset{(*)}{=} \emptyset$ and $\mathcal{V}(G\theta) \cap \mathcal{V}(t) \subseteq \mathcal{V}(G\theta_1\sigma') \cap \mathcal{V}(t'\sigma') \overset{(4,*)}{=} \emptyset$. For (2), assume that $e' \in G''$ satisfies $\mathcal{V}(e') \cap \mathcal{V}(t) \neq \emptyset$. Note that $\mathcal{V}(\lambda\overline{x}.t_j \blacktriangleright \lambda\overline{x}.H_j(\overline{y_q})) \cap \mathcal{V}(t) = \emptyset$ for all $j > p$. Therefore e' descends from some equation $e'' \in G_2'$. This implies that $e' = e''\sigma'$. Because $\emptyset \neq \mathcal{V}(e') \cap \mathcal{V}(t') = \mathcal{V}(e''\sigma') \cap \mathcal{V}(t\sigma')$, we have $\mathcal{V}(e'') \cap \mathcal{V}(t') \neq \emptyset$, and by (5) we deduce that $s' \blacktriangleright t' \in \mathrm{prec}(e'')$. Therefore (2) holds, too.

[p] Then $s' \blacktriangleright t' = \lambda\overline{x}.X(\overline{s_m}) \blacktriangleright u$ or $s' \blacktriangleright t' = u \blacktriangleright \lambda\overline{x}.X(\overline{s_m})$ with u rigid, and $s \blacktriangleright t = (s' \blacktriangleright t')\sigma'$ with $\sigma' = \{X \mapsto \lambda\overline{x_m}.x_j(\overline{H_p(\overline{x_m})})\}$ for some $1 \leq j \leq m$. It is easy to see that under the additional fact that $|\mathcal{D}(\sigma')| = 1$, the following logical implications hold: (4)$\wedge$(*) $\Rightarrow$ (1), (5)$\wedge$(*) $\Rightarrow$ (2), and (6)$\wedge$(*) $\Rightarrow$ (3).

Subcase 2.2 Assume $s' \blacktriangleright t'$ is not selected in π'. Then $s \blacktriangleright t = s'\sigma' \blacktriangleright t'\sigma'$. We distinguish two situations, depending on whether α' is [ffd] or not. If $\alpha' =$ [ffd] then it selects a flex/flex equation $e' = \lambda\overline{y}.X(\overline{y'}) \rhd \lambda\overline{y}.Y(\overline{y''})$ in G_2, where $\overline{y'}$ and $\overline{y''}$ are sequences of distinct bound variables and $\mathrm{prec}(e') = \Box$. There are two possibilities:

1. $e' \in G_2'$; then (4) implies that $\{X, Y\} \cap \mathcal{V}(t') = \emptyset$.

2. $e' \in G_2''$; then $s' \blacktriangleright t' \notin \mathrm{prec}(e')$ (because $\mathrm{prec}(e') = \Box$) $\overset{(5)}{\Rightarrow} \{X, Y\} \cap \mathcal{V}(t') = \emptyset$.

In both situations we have $\mathcal{D}(\sigma') \cap \mathcal{V}(t') = \{X, Y\} \cap \mathcal{V}(t') = \emptyset$. We label this equality with (7). Then $t = t'\sigma' \overset{(7)}{=} t'$, and (3) follows from (6). Also (1) holds because $\mathcal{V}(G', s) \cap \mathcal{V}(t) \subseteq \mathcal{V}(G_2'\sigma', s'\sigma) \cap \mathcal{V}(t') \overset{(7)}{=} (\mathcal{V}(G_2', s') \cup \mathcal{R}ng(\sigma')) \cap \mathcal{V}(t') \overset{(4,*)}{=} \emptyset$ and $\mathcal{V}(G\theta) \cap \mathcal{V}(t) \subseteq (\mathcal{V}(G\theta_1) \cup \mathcal{R}ng(\sigma')) \cap \mathcal{V}(t') \overset{(4,*)}{=} \emptyset$.

For (2), let $e'_1 \in G''$ such that $\mathcal{V}(e'_1) \cap \mathcal{V}(t) \neq \emptyset$. Proving (2) amounts to proving that $s \blacktriangleright t \in \mathrm{prec}(e'_1)$. Let $e''_1 \in G''_2$ such that e'_1 descends from e''_1. Note that $\mathcal{V}(e''_1) \cap \mathcal{V}(t') = \emptyset$ is possible only if $X \in \mathcal{V}(t'), Y \in \mathcal{V}(e'_1)$, or $Y \in \mathcal{V}(t'), X \in \mathcal{V}(e'_1)$, which contradicts (7). Thus $\mathcal{V}(e''_1) \cap \mathcal{V}(t') \neq \emptyset \overset{(5)}{\Rightarrow} s' \blacktriangleright t' \in \mathrm{prec}(e''_1) \Rightarrow (2)$.

We assume now that $\alpha' \neq$ [ffd]. Then $|\mathcal{D}(\sigma')| \leq 1$. We label this property with (8). Note that $\mathcal{V}(G\theta_1) \cap \mathcal{V}(t') = \emptyset$ because of (4), and therefore either $\mathcal{V}(G\theta) \cap \mathcal{V}(t) \subseteq \mathcal{V}(G\theta_1) \cap (\mathcal{V}(t') \cup \mathcal{R}ng(\sigma')) \overset{(4,*)}{=} \emptyset$ if $\mathcal{D}(\sigma') \subseteq \mathcal{V}(t')$, or $\mathcal{V}(G\theta) \cap \mathcal{V}(t) \subseteq (\mathcal{V}(G\theta_1) \cup \mathcal{R}ng(\sigma')) \cap \mathcal{V}(t') \overset{(4,*)}{=} \emptyset$ otherwise. Thus $\mathcal{V}(G\theta)) \cap \mathcal{V}(t) = \emptyset$, and therefore, for proving (1), it remains to show that $\mathcal{V}(t) \cap \mathcal{V}(G', s) = \emptyset$. If π' is not [on] or [ov] applied to some equation of G'_2 then $\mathcal{V}(G', s) \cap \mathcal{V}(t) \subseteq \mathcal{V}(G'_1\sigma', s'\sigma') \cap \mathcal{V}(t'\sigma') \overset{(*,4,8)}{=} \emptyset$. Otherwise, $\mathcal{V}(G', s) \subseteq \mathcal{V}(G'_1\sigma', s'\sigma') \cup V$ where V is the set of fresh variables introduced by the rule variant used in the narrowing step. Then $(\mathcal{V}(G', s) \cup V) \cap \mathcal{V}(t) \subseteq \mathcal{V}(G'_1\sigma', s'\sigma' \cup V) \cap \mathcal{V}(t'\sigma') \overset{(*,4,8)}{=} \emptyset$. Hence (1) holds in both situations. (3) follows from (6), (*) and (8). For (2), assume that $e' \in G''$ such that $\mathcal{V}(e') \cap \mathcal{V}(s \blacktriangleright t) \neq \emptyset$ and let e'' be the equation in G''_2 from which e' descends. Then $\mathcal{V}(e') \subseteq \mathcal{V}(e''\sigma')$ if π' is not [on] or [ov] applied to e'', or $\mathcal{V}(e') \subseteq \mathcal{V}(e''\sigma') \cup V$ otherwise, where V are the fresh variables introduced by the rule variant of the narrowing step. In both cases we obtain $\emptyset \neq \mathcal{V}(e') \cap \mathcal{V}(s \blacktriangleright t) \subseteq \mathcal{V}(e''\sigma') \cap \mathcal{V}(s'\sigma \blacktriangleright t'\sigma)$, which yields $\mathcal{V}(s' \blacktriangleright t') \cap \mathcal{V}(e'') \neq \emptyset$. From (5) we get that $s' \blacktriangleright t' \in \mathrm{prec}(e'')$, which yields (2). $\square$

5.7 Redundant Equations

Upon computations of $\mathcal{R}$-normalized solutions with $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-refutations, the calculus may generate new equations which are redundant, i.e. solving them does not contribute to the computation of an $\mathcal{R}$-normalized solution. This behaviour is illustrated in the example below.

Example 8 *Let $\mathcal{R} = \{f(X_1, X_2) \to X_1\}$ and $G = f(X, f(f(3))) \rhd Y$. G has the $\mathcal{R}$-normalized solution $\theta = \{X \mapsto H_2, Y \mapsto H_2\}$ which can be computed with an $\langle \mathrm{LN}_1, \mathcal{S}_n \rangle$-refutation as follows:*

$$\begin{aligned} G \quad & \Rightarrow_{[\mathrm{on}]} G_1 = X_1 \blacktriangleright X, f(f(3)) \blacktriangleright X_2, X_1 \rhd Y \\ & \Rightarrow_{[\mathrm{ffd}],\sigma} G_2 = f(f(3)) \blacktriangleright X_2, H_1 \rhd Y \Rightarrow^3_{[\mathrm{i}]_2,\sigma_2} G_3 = H_1 \rhd Y \Rightarrow_{[\mathrm{ffd}],\sigma_3} \square \end{aligned}$$

where $\sigma_1 = \{X \mapsto H_2, X_1 \mapsto H_2\}$, $\sigma_3 = \{H_1 \mapsto H_2, Y \mapsto H_2\}$, and the subderivation $G_2 \Rightarrow^3_{[\mathrm{i}]_2,\sigma_2} G_3$ selects only descendants of $f(f(3)) \blacktriangleright X_2$.

We see that solving this equation does not contribute to the computation of θ*, and thus it can be eliminated from* G_2 *without affecting the computed solution. The calculus* LNC [MO98] *generates a derivation that eliminates this equation by performing a* [v]*-step which binds* X_2 *to* $f(f(3))$*, but it also generates unnecessary derivations that apply* [on] *or [i] to this equation.*

The following definition formalizes our notion of redundant equation.

Definition 48 (redundant equation) *Let* $G \Rightarrow^* G'$ *be an* $\langle \mathrm{LN}_1, \mathcal{S}_n\rangle$*-derivation and* $e = \lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.X(\overline{y})$ *an equation in* G' *with* $\mathcal{V}(\lambda\overline{y}.s) \cap \{\overline{x}\} = \emptyset$*. Then* e *is* redundant *if* $X \notin \mathcal{V}(G' \setminus e, \lambda\overline{x}.s)$.

Note that according to this definition, an equation of the form $\lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.X(\overline{y})$ with $\lambda\overline{y}.s \cap \{\overline{x}\} \neq \emptyset$ is *not* redundant. For example, the equation $\lambda x, y.Y(x, y) \blacktriangleright \lambda x, y.X(x)$ is not redundant. Intuitively, this equation is not redundant because it contains the information that the left-hand side $\lambda x, y.Y(x, y)$ must be reduced to a term which does not depend on y.

The elimination rule for redundant equations is

$$[\mathrm{rm}] \qquad \frac{G, e, G'}{G, G'} \text{ if } e \text{ is redundant.}$$

In the sequel we study the soundness and completeness of the calculus $\mathrm{LN}_1 \cup \{[\mathrm{rm}]\}$.

Lemma 34 *If* $\Pi : G_0 \Rightarrow^*_\theta G_1 \Rightarrow_{[\mathrm{rm}]} G_2$ *is an* $\langle \mathrm{LN}_1, \mathcal{S}_n\rangle$*-derivation then*

$$\{\theta\gamma\restriction_{\mathcal{V}(G_0)} \mid \gamma \in \mathcal{U}_{\mathcal{R}}(G_1)\} = \{\theta\gamma'\restriction_{\mathcal{V}(G_0)} \mid \gamma' \in \mathcal{U}_{\mathcal{R}}(G_2)\}.$$

<u>Proof.</u> Assume the last step of Π is

$$\pi : G_1 = G, \underline{\lambda\overline{x}.s(\overline{y}) \blacktriangleright \lambda\overline{x}.X(\overline{y})}, G' \Rightarrow_{[\mathrm{rm}]} G_2 = G', G''.$$

Let $A_1 = \{\theta\gamma\restriction_{\mathcal{V}(G_0)} \mid \gamma \in \mathcal{U}_{\mathcal{R}}(G_1)\}$, $A_2 = \{\theta\gamma'\restriction_{\mathcal{V}(G_0)} \mid \gamma' \in \mathcal{U}_{\mathcal{R}}(G_2)\}$, and $V = \mathcal{V}(G\theta)$. Obviously, $\mathcal{U}_{\mathcal{R}}(G_1) \subseteq \mathcal{U}_{\mathcal{R}}(G_2)$, and thus $A_1 \subseteq A_2$.

Let $\delta \in A_2$. Then there exists $\gamma' \in \mathcal{U}_{\mathcal{R}}(G_2)$ such that $\delta = \theta\gamma'$. From the shapes of G_1 and G_2 results that $\delta\restriction_{\mathcal{V}\setminus\{X\}} \cup \{X \mapsto \lambda\overline{x}.s\gamma'\}$ is a well-defined substitution and $\delta \in \mathcal{U}_{\mathcal{R}}(G_1)$. From Lemma 33.(1) results that $X \notin V$. Therefore, $\theta\gamma'\restriction_{\mathcal{V}(G_0)} = \theta\delta\restriction_{\mathcal{V}(G_0)}$, and thus $A_2 \subseteq A_1$.

Hence $A_1 = A_2$. □

Lemma 35 *Let* $G = \overline{e_n}$ *be a goal,* $\langle G, \gamma, \overline{R_n}\rangle \in \mathrm{Repr}(G)$*,* $e_k \in G$ *a redundant equation,* $G' = (\overline{e_{k-1}}, \overline{e_{k+1,n}})$*, and* $\overline{R'_{n-1}} = (\overline{R_{k-1}}, \overline{R_{k+1,n}})$.

Then we can perform the [rm]*-step* $G \Rightarrow_{[\mathrm{rm}],\varepsilon} G'$*, and we have*

$$\langle G, \theta, \overline{R_n}\rangle \succ \langle G', \theta, \overline{R'_{n-1}}\rangle.$$

Proof. Obvious. □

Lemmata 34 and 35 imply that we preserve the soundness and completeness properties of LN_1 with strategy $\mathcal{S}_n$ if we extend $\mathcal{S}_n$ to a strategy which can select redundant equations, and refine LN_1 to a calculus which applies rule [rm] to selected equations which are redundant. We call the new strategy $\mathcal{S}_c$ and the refined calculus LN_2. The definition of the strategy $\mathcal{S}_c$ can be obtained from the definition of $\mathcal{S}_n$ by replacing condition (c1) with

(c1') a flex/flex equation is selected only if it is redundant or (c1) holds.

The nondeterminism between the inference rules of LN_2 with strategy $\mathcal{S}_c$ is shown in Figs. 5.6 and 5.7. The superscripts attached to the labels in the table indicate the priority of applying the corresponding inference rule. These priorities are established by imposing additional side conditions to the corresponding inference rules of LN_1. E.g., for equations of the form $e = \lambda\overline{x}.v(\overline{s_n}) \rhd \lambda\overline{x}.X(\overline{t_n})$ with $v \in \{\overline{x}\} \cup \mathcal{F}_c$, checks whether the preconditions to apply rule [rm] are satisfied. If yes, then [rm] is applied deterministically. Otherwise the selected equation is not redundant and the inference rules $[\mathrm{i}]_2$, $[\mathrm{p}]_2$ of LN_2 are applied nondeterministically.

${}_{\mathrm{root}(s)}\backslash^{\mathrm{root}(t)}$	$\mathcal{V}(t)$	F_d	$F_c \cup \mathcal{BV}(t)$
$\mathcal{V}(s)$	$[\mathrm{del}]^1/[\mathrm{rm}]^2/$ $[\mathrm{ffs}]^3/[\mathrm{ffd}]^3$	$[\mathrm{i}]_1^1, [\mathrm{p}]_1^1,$ $[\mathrm{ov}]_1^1, [\mathrm{on}]_2^1$	$[\mathrm{i}]_1^1, [\mathrm{p}]_1^1, [\mathrm{ov}]_1^1$
F_d	$[\mathrm{rm}]^1/$ $([\mathrm{i}]_2^2, [\mathrm{p}]_2^2, [\mathrm{on}]_1^2)$	$[\mathrm{del}]^1/$ $([\mathrm{on}]_1^2, [\mathrm{on}]_2^2, [\mathrm{d}]^2)$	$[\mathrm{on}]_1^1$
$F_c \cup \mathcal{BV}(s)$	$[\mathrm{rm}]^1/$ $([\mathrm{i}]_2^2, [\mathrm{p}]_2^2, [\mathrm{ov}]_2^2)$	$\times$	$[\mathrm{del}]^1/[\mathrm{d}]^2$

Fig. 5.6: Inference rules of LN_2 for equation $s \approx t$ selected by $sel \in \mathcal{S}_c$

${}_{\mathrm{root}(s)}\backslash^{\mathrm{root}(t)}$	$\mathcal{V}(t)$	F_d	$F_c \cup \mathcal{BV}(t)$
$\mathcal{V}(s)$	$[\mathrm{del}]^1/[\mathrm{rm}]^2/[\mathrm{ffs}]^3/[\mathrm{ffd}]^3$	$[\mathrm{i}]_1^1, [\mathrm{p}]_1^1, [\mathrm{ov}]^1$	$[\mathrm{i}]_1^1, [\mathrm{p}]_1^1, [\mathrm{ov}]^1$
F_d	$[\mathrm{rm}]^1/([\mathrm{i}]_2^2, [\mathrm{p}]_2^2, [\mathrm{on}]^2)$	$[\mathrm{del}]^1/([\mathrm{on}]^2, [\mathrm{d}]^2)$	$[\mathrm{on}]$
$F_c \cup \mathcal{BV}(s)$	$[\mathrm{rm}]^1/([\mathrm{i}]_2^2, [\mathrm{p}]_2^2)$	$\times$	$[\mathrm{del}]^1/[\mathrm{d}]^2$

Fig. 5.7: Inference rules of LN_2 for equation $s \rhd t$ selected by $sel \in \mathcal{S}_c$

In the sequel we investigate the calculus LN_2 with strategy $\mathcal{S}_c$.

5.8 Lazy Narrowing for Left-Linear Constructor Pattern Rewrite Systems

In this section we study the possibility to reduce the nondeterminism between the inference rules of LN_2 with strategy $\mathcal{S}_n$ for confluent left-linear constructor fully-extended PRSs.

The restriction to left-linear constructor rewrite systems is quite customary in functional logic programming. In the first-order case it was shown [MO98] that for left-linear constructor PRS we can completely eliminate the nondeterminism on the selection of inference rules for solving flex/rigid descendants of parameter-passing equations without losing completeness because all descendants of parameter-passing equations have constructor terms to the right-hand side. Unfortunately, LN_2 with strategy $\mathcal{S}_n$ does not have this property, as we can see from the following example.

Example 9 *Consider the left-linear constructor PRS $\mathcal{R} = \{f(X) \to X\}$ and the goal $G = f(Y(X)) \rhd a$. Then any $\langle \mathrm{LN}_2, \mathcal{S}_n\rangle$-derivation starts with the following $\langle \mathrm{LN}_2, \mathcal{S}_n\rangle$-subderivation*

$$\begin{aligned}\underline{f(Y(X)) \rhd a} &\Rightarrow_{[\mathrm{on}]} G = Y(X) \blacktriangleright X_1, \underline{X_1 \rhd a}\\ &\Rightarrow_{[\mathrm{ov}],\sigma=\{X_1 \mapsto f(H)\}} G' = \underline{Y(X) \blacktriangleright f(H)}, H(X) \blacktriangleright X_2, X_2 \rhd a\end{aligned}$$

The application of [ov] *in the second inference step introduces the defined symbol f in the right-hand side of the leftmost parameter-passing equation of G'.* □

It is easy to see that such undesirable parameter-passing equations are created as the result of an [ov]- or [i]-step. Thus, we should invent a method to avoid performing [ov]- or [i]-steps in such problematic situations, but without losing completeness. The following lemma gives the basic idea for our attempt.

Lemma 36 *Let $e = \lambda\overline{z}.s(\overline{y}) \rhd \lambda\overline{z}.X(\overline{y})$, $e' = \lambda\overline{x}.X(\overline{t}) \trianglerighteq \lambda\overline{x}.u$ be equations such that $\lambda\overline{x}.u$ is rigid, $\mathcal{V}(s) \cap \{\overline{z}\} = \emptyset$, $X \notin \mathcal{V}(s, \lambda\overline{x}.u)$ and $V = \mathcal{V} \setminus \{X\}$. Assume that $e'' = \lambda\overline{x}.s(\overline{t}) \trianglerighteq \lambda\overline{x}.u$ is the equation obtained from e' by replacing $\lambda\overline{x}.X(\overline{t})$ with $\lambda\overline{x}.s(\overline{t})$. Then*

$$\{\theta\restriction_V \mid \theta \in \mathcal{U}_{\mathcal{R}}(e, e')\} = \{\theta'\restriction_V \mid \theta' \in \mathcal{U}_{\mathcal{R}}(e, e'')\}.$$

Proof. We show that the left-hand side is a subset of the right-hand side, and vice versa.
($\subseteq$) Let $\theta \in \mathcal{U}_{\mathcal{R}}(e, e')$. It is sufficient to prove that $\theta' \in \mathcal{U}_{\mathcal{R}}(e'')$. By definition, $\theta \in \mathcal{U}_{\mathcal{R}}(e, e')$ iff

(1) $\lambda\overline{z}.s\theta(\overline{y}) \rightarrow^*_{\mathcal{R}} \lambda\overline{z}.(X\theta)(\overline{y})$ and
(2) $\theta \in \mathcal{U}_{\mathcal{R}}(e')$.

Because $\lambda\overline{x}.(s\theta)(\overline{t\theta}) \overset{(1)}{\rightarrow}{}^*_{\mathcal{R}} \lambda\overline{x}.(X\theta)(\overline{t\theta})$, we have that $\theta \in \mathcal{U}_{\mathcal{R}}(\lambda\overline{x}.s(\overline{t}) \rhd \lambda\overline{x}.X(\overline{t}))$. From (2) we know that $\theta \in \mathcal{U}_{\mathcal{R}}(\lambda\overline{x}.X(\overline{t}) \trianglerighteq \lambda\overline{x}.u)$. Then $\theta \in \mathcal{U}_{\mathcal{R}}(\lambda\overline{x}.s(\overline{t}) \trianglerighteq \lambda\overline{x}.u)$, i.e. $\theta \in \mathcal{U}_{\mathcal{R}}(e'')$.

($\supseteq$) Assume $\sigma = \theta\lceil_V$ where $\theta \in \mathcal{U}_{\mathcal{R}}(e, e'')$. Then
(3) $\lambda\overline{z}.s\theta(\overline{y}) \rightarrow^*_{\mathcal{R}} \lambda\overline{z}.(X\theta)(\overline{y})$ and
(4) $\theta \in \mathcal{U}_{\mathcal{R}}(e'')$.
We want to prove that there exists a substitution θ' such that
(5) $\lambda\overline{z}.s\theta'(\overline{y}) \rightarrow^*_{\mathcal{R}} \lambda\overline{z}.(X\theta')(\overline{y})$,
(6) $\theta' \in \mathcal{U}_{\mathcal{R}}(\lambda\overline{x}.X(\overline{t}) \trianglerighteq \lambda\overline{x}.u)$, and
(7) $\theta = \theta'\ [V]$.
Let $\theta' = \theta\lceil_V \cup \{X \mapsto \lambda\overline{y}.s\theta(\overline{y})\}$ which gives (7) as an immediate consequence. Since $X \notin \mathcal{V}(s)$ we have $\lambda\overline{z}.s\theta'(\overline{y}) = \lambda\overline{z}.s\theta(\overline{y}) = \lambda\overline{z}.(X\theta')(\overline{y})$. Thus (5) holds and (8) $\lambda\overline{z}.X\theta'(\overline{y}) \rightarrow^*_{\mathcal{R}} \lambda\overline{z}.X\theta(\overline{y})$ from (3). For (6) we observe that if $e'' = \lambda\overline{x}.X(\overline{t}) \rhd \lambda\overline{x}.u$ then

$$\lambda\overline{x}.(X\theta')(\overline{t\theta'}) = \lambda\overline{x}.s\theta(\overline{t\theta'}) \overset{(8)}{\rightarrow}{}^*_{\mathcal{R}} \lambda\overline{x}.s\theta(\overline{t\theta}) \overset{(4)}{\rightarrow}{}^*_{\mathcal{R}} \lambda\overline{x}.u\theta = \lambda\overline{x}.u\theta',$$

and if $e'' = \lambda\overline{x}.X(\overline{t}) \simeq \lambda\overline{x}.u$ then

$$\lambda\overline{x}.(X\theta')(\overline{t\theta'}) = \lambda\overline{x}.s\theta(\overline{t\theta'}) \overset{(8)}{\rightarrow}{}^*_{\mathcal{R}} \lambda\overline{x}.s\theta(\overline{t\theta}) \overset{(4)}{\leftrightarrow}{}^*_{\mathcal{R}} \lambda\overline{x}.u\theta = \lambda\overline{x}.u\theta'.$$

Thus (6) holds in both situations. □

Remark 1 *In Lemma 36 we require $\mathcal{V}(s) \cap \{\overline{z}\} = \emptyset$. Without this restriction the case "$\supseteq$" in the proof of Lemma 36 does not hold.*

Lemma 36 suggests the following new inference rule:

$$[\mathrm{c}]\ \frac{G, \lambda\overline{z}.s(\overline{y}) \blacktriangleright \lambda\overline{z}.X(\overline{y}), G', \lambda\overline{x}.X(\overline{t}) \trianglerighteq \lambda\overline{x}.u, G''}{G, \lambda\overline{z}.s(\overline{y}) \blacktriangleright \lambda\overline{z}.X(\overline{y}), G', \lambda\overline{x}.s(\overline{t}) \trianglerighteq \lambda\overline{x}.u, G''}$$
if $\lambda\overline{x}.u$ is rigid and $\mathcal{V}(s) \cap \{\overline{z}\} = \emptyset$.

The equation selected by the [c]-rule is $\lambda\overline{x}.X(\overline{t}) \trianglerighteq \lambda\overline{x}.u$ and its descendant is defined to be $\lambda\overline{x}.s(\overline{t}) \trianglerighteq \lambda\overline{x}.u$. The notions of precursor and descendant are carried over to the [c]-rule in the natural way. We denote by LN_3 the calculus obtained from LN_2 by adding the rule [c] and by applying it instead of [ov] and $[\mathrm{i}]_1$ whenever possible.

Remark 2 *The statements of Lemma 33 hold for the calculus* LN_3 *with strategy* $\mathcal{S}_c$ *as well.*

The soundness of the calculus LN_3 is an immediate consequence of the following lemma:

Lemma 37 *If* $\Pi : G_0 \Rightarrow_\theta^* G_1 \Rightarrow_{[\mathrm{c}]} G_2$ *is an* $\langle \mathrm{LN}_3, \mathcal{S}_n \rangle$*-derivation then*

$$\{\theta\gamma\restriction_{\mathcal{V}(G_0)} \mid \gamma \in \mathcal{U}_\mathcal{R}(G_1)\} = \{\theta\gamma'\restriction_{\mathcal{V}(G_0)} \mid \gamma' \in \mathcal{U}_\mathcal{R}(G_2)\}.$$

Proof. Assume the equation selected in the last $\langle \mathrm{LN}_3, \mathcal{S}_n \rangle$-step of Π is $\lambda\overline{x}.X(\overline{t}) \trianglerighteq \lambda\overline{x}.u$. Then there exists a parameter-passing equation $e' = \lambda\overline{z}.s \blacktriangleright \lambda\overline{z}.X(\overline{y}) \in \mathrm{prec}_{G_1}(e)$. From Remark 2 we learn that $X \notin \mathcal{V}(G\theta)$. Then Lemma 37 follows from Lemma 36. □

Assume
$$G_0 \Rightarrow_\theta^* G_1 = G', \underbrace{\lambda\overline{x}.X(\overline{t}) \trianglerighteq \lambda\overline{x}.u}_{e}, G''$$
is an $\langle \mathrm{LN}_3, \mathcal{S}_c \rangle$-derivation and $e = sel(G_1)$ for some $sel \in \mathcal{S}_c$. From Remark 2 and condition (c2) of strategy $\mathcal{S}_c$ we deduce that either: (a) there exists a parameter-passing equation $e' = \lambda\overline{z}.s \blacktriangleright \lambda\overline{z}.X(\overline{y}) \in \mathrm{prec}_{G_1}(e)$, or (b) all the equations $s' \blacktriangleright t' \in G \setminus e$ satisfy the condition $X \notin \mathcal{V}(t')$. The problematic situation is (a): in this case it is desirable to be able to apply rule [c] instead of [ov] or [i]. Unfortunately, there is a side condition which can prohibit the application of [c], namely if e' satisfies the condition $\mathcal{V}(s) \cap \{\overline{z}\} \neq \emptyset$. We avoid such situations by imposing an additional condition on the left-linear constructor PRS as well.

Definition 49 (full extension) *A term is* fully-extended *if every free variable in the term has all the bound variables in the current scope as arguments. A PRS consisting of rewrite rules with fully-extended left-hand sides is called* fully-extended*.*

The notion of fully-extended PRSs was first introduced by Prehofer [Pre98], though he had a different motivation than ours. Hereafter, we simply refer to fully-extended PRSs as EPRSs.

Lemma 38 *If* $\mathcal{R}$ *is a left-linear EPRS then the right-hand sides of the parameter-passing equations generated in* $\langle \mathrm{LN}_3, \mathcal{S}_c \rangle$*-derivations are fully-extended.*

Here we don't show the detailed proof but only note that the right-hand sides of the parameter-passing equations generated by [ov] and [on] are fully-extended and that [i] and [p] instantiate variables with fully-extended terms. [ffd] and [ffs] are the only rules which may instantiate variables with non-fully-extended terms; however, they never instantiate the variables in the right-hand sides of the parameter-passing equations. This follows from Remark 2.

The following lemma is the counterpart for LN_3 of Lemma 3.1.(2) in [MO98] for LNC.

Lemma 39 *Let $\mathcal{R}$ be a left-linear constructor EPRS and $\Pi\colon G \Rightarrow_\theta^* G', s \blacktriangleright t, G''$ an $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-derivation. Then t is a linear pattern constructor term.*

Lemma 39 explains why the calculus LN_3 with strategy $\mathcal{F}_c$ is more deterministic than LN_2 with strategy $\mathcal{S}_c$: we can completely remove column 2 from Fig. 5.7 for descendants of parameter-passing equations (i.e., equations of the form $s \blacktriangleright t$).

We have already seen why LN_2 with strategy $\mathcal{S}_c$ is sound. Now we explain why LN_3 with strategy $\mathcal{S}_c$ is complete.

Lemma 40 *Let $\mathcal{R}$ be a confluent left-linear constructor EPRS. If $\gamma \in \mathcal{U}_\mathcal{R}(G)$ is normalized then there exists an $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-refutation $\Pi : G \Rightarrow_\theta^* F$ such that $\theta\gamma' = \gamma\ [\mathcal{V}(G)]$ for some solution $\gamma \in \mathcal{U}_\mathcal{R}(F)$.*

Proof. (Sketch) Let γ be a normalized solution of G. We will prove that there exists an $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-refutation $\Pi : G \Rightarrow_\theta^* F$ with $\theta\eta = \gamma\ [\mathcal{V}(G)]$ for some solution η of F. The construction of Π is depicted in the diagram below. in Fig. 5.8. Here $\alpha_1, \ldots, \alpha_n \in \{$[i],[ov]$\}$. We prove by induction on

$$
\begin{array}{lllllllll}
\exists\Pi_1 : G \Rightarrow_{\theta_1}^* & G_1 & & & & & & & \Rightarrow_{\alpha_1}\Rightarrow^* F_1 \\
 & \Downarrow_{[c]}^+ & & & & & & & \\
\exists\Pi_2 : & G_1' & \Rightarrow_{\theta_2}^+ & G_2 & & & & & \Rightarrow_{\alpha_2}\Rightarrow^* F_2 \\
 & & & \Downarrow_{[c]}^+ & & & & & \\
 & & & & \ddots & & & & \\
 & & & & & \Downarrow_{[c]}^+ & & & \\
\exists\Pi_n : & & & & & G_{n-1}' & \Rightarrow_{\theta_n}^+ & G_n & \Rightarrow_{\alpha_n}\Rightarrow^* F_n
\end{array}
$$

Fig. 5.8: Construction of an $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-refutation $\Pi : G \Rightarrow^* G_n$ for a given $\mathcal{R}$-normalized solution $\gamma \in \mathcal{U}_\mathcal{R}(G)$

n that the following conditions hold:

(a) the LN_2-derivation $G \Rightarrow_{\theta_1\ldots\theta_n}^* G_n$ shown in the diagram is an $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-derivation,

(b) for any $i \leq n$ there exists $\gamma_i \in \mathcal{U}_\mathcal{R}(G_i)$ such that $\theta_1 \ldots \theta_i\gamma_i = \gamma\ [\mathcal{V}(G)]$,

(c) $|\gamma_1| > \ldots > |\gamma_n|$ where $|\gamma_i| := \{|X\gamma_i| \mid X \in \mathcal{V}(G\theta_1 \ldots \theta_i)\}$.

The construction starts with an $\langle \mathrm{LN}_2, \mathcal{S}_c\rangle$-refutation $\Pi_1 : G \Rightarrow^*_\delta F_1$ such that:

- $\gamma = \delta_1\gamma''_1$ for some $\gamma''_1 \in \mathcal{U}_\mathcal{R}(F_1)$,
- [ov] or [i] is applied to an equation e selected in some intermediate goal G'' of Π_1 only if prec(e) is a non-empty flex/flex goal and all the non-flex/flex equations of G'' are flex/rigid with precursors.

From the completeness of LN_2 with strategy $\mathcal{S}_c$, such an $\langle \mathrm{LN}_2, \mathcal{S}_c\rangle$-refutation exists. Let G_1 be the first goal in Π_1 where an equation to which [c] can be applied is selected. If such a goal does not exist then Π_1 is an $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-refutation and we choose $\Pi = \Pi_1$, $F = F_1$. Otherwise we cut out the subrefutation originating in G_1, keep the subderivation $G \Rightarrow^*_{\theta_1} G_1$, and start applying all the possible [c]-steps on the equation selected by Π_1 from G_1. Then $G \Rightarrow^*_\theta G_1$ is an $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-derivation and $\gamma = \theta_1\gamma_1\ [\mathcal{V}(G)]$ for some $\gamma_1 \in \mathcal{U}_\mathcal{R}(G_1)$. Suppose

- $e = \lambda\overline{x}.X(\overline{t}) \trianglerighteq u$ is the flex/rigid equation selected from G_1,
- $e' = \lambda\overline{x}.X'(\overline{t'}) \trianglerighteq u$ is the result of applying all [c]-steps on e, and
- G'_1 is the corresponding goal.

If the sequence of precursors used in the [c]-steps is $\lambda\overline{x}.X_{n+1}(\ldots) \blacktriangleright \lambda\overline{x}.X_n(\ldots), \ldots, \lambda\overline{x}.X_1(\ldots) \blacktriangleright \lambda\overline{x}.X(\overline{x})$ then Lemma 36 yields that $\gamma'_1 = \gamma_1\ [\mathcal{V}(G_1) \setminus \{\overline{X_n}\}]$ for some $\gamma'_1 \in \mathcal{U}_\mathcal{R}(G'_1)$. From Remark 2 we know that $\{\overline{X_n}\} \cap \mathcal{V}(G\theta_1) = \emptyset$, which implies that $\theta_1\gamma_1 = \theta_1\gamma'_1\ [\mathcal{V}(G)]$.

Now, we can select e' and perform an LN_2-step which is also an LN_3-step. We start with such an LN_3-step and construct an $\langle \mathrm{LN}_1, \mathcal{S}_c\rangle$-refutation $\Pi_2 : G'_1 \Rightarrow^+_{\theta_2} G_2 \Rightarrow_{\alpha_2,\sigma_1}\Rightarrow^*_{\delta_2} F_2$ with $\alpha_2 \in \{[\mathrm{i}],[\mathrm{ov}]\}$ in the same way as we constructed Π_1. Again, we cut out the subrefutation (if any) of Π_2 which starts with the LN_2-step that violates strategy $\mathcal{S}_c$, retain the $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-derivation $G'_1 \Rightarrow^+_{\theta_2} G_2$ and determine $\gamma_2 \in \mathcal{U}_\mathcal{R}(G_2)$. It is important to see that we obtain an $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-derivation $G'_1 \Rightarrow^+_{\theta_2} G_2$ with $\emptyset \neq \mathcal{D}(\sigma_1) \subseteq \mathcal{D}(\theta_2) \subseteq \mathcal{V}(G\theta_1)$, which explains why $|\gamma_1| > |\gamma_2|$.

Since there is no infinite descending chain $|\gamma_1| > |\gamma_2| > \ldots$, our construction will eventually terminate and yield the desired $\langle \mathrm{LN}_3, \mathcal{S}_c\rangle$-refutation. □

5.9 Strict Equality

In functional logic programming it is customary to consider two expressions equal if they reduce to the same constructor term [MO98]. This so-called *strict equality* can be integrated into our lazy narrowing calculi if we distinguish between the following types of equations:

1. unoriented equations
   (a) with nonstrict semantics, denoted by $s \approx t$,
   (b) with strict semantics, denoted by $s \doteq t$.
2. oriented equations
   (a) with strict semantics, denoted by $s \gg t$
   (b) with nonstrict semantics
      i. linear descendant of initial equation, denoted by $s \rhd t$
      ii. descendant of parameter-passing equation, denoted by $s \blacktriangleright t$.

An $\mathcal{R}$-unifier of of an equation with strict semantics is an $\mathcal{R}$-unifier of its nonstrict counterpart.

Definition 50 (strict solution) *A* strict solution *of a nonstrict equation e is an $\mathcal{R}$-unifier of e.*

A strict solution *of a strict equation $s \doteq t$ is a solution $\theta \in \mathcal{U}_{\mathcal{R}}(s \approx t)$ such that there exists $u \in \mathcal{T}(\mathcal{F}_c, \mathcal{V})$ for which $s\theta \rightarrow^*_{\mathcal{R}} u$ and $t\theta \rightarrow^*_{\mathcal{R}} u$.*

A strict solution *of a strict equation $s \gg t$ is a solution $\theta \in \mathcal{U}_{\mathcal{R}}(s \gg t)$ such that $t\theta \in \mathcal{T}(\mathcal{F}_c, \mathcal{V})$.*

In this section we look for efficient calculi to compute normalizes strict solutions of goals consisting of equations of type 1.(a), 1.(b), 2.(a), 2.(b).i.

We start with the calculus LN_2 with strategy $\mathcal{S}_c$ introduced in Sect. 5.7, and regard is as the disjoint union of two subcalculi:

- LN_2^{ns} for non-strict equations, and
- LN_2^{s} for strict equations.

In the sequel we analyze how the subcalculus LN_2^{s} can be specialized to efficiently solve strict equations. The following lemma resumes the results of our analysis.

Lemma 41 *Let $\mathcal{R}$ be a left-linear PRS, $G = G_1, e, G_2$ be a goal and γ be an $\mathcal{R}$-normalized solution of G. If e can be selected with strategy $\mathcal{S}_c$ then there exists an $\langle \mathrm{LN}_2, \mathcal{S}_c \rangle$-step $\pi : G \Rightarrow_\theta G'$ and a substitution $\gamma' \in \mathcal{U}_{\mathcal{R}}(G')$ such that*

(1) e is the equation selected by π,

(2) $\gamma = \theta\gamma'$ $[\mathcal{V}(G)]$,

(3) if $e = s \gg t$ *then*

(3.1) $\mathrm{root}(t) \notin \mathcal{F}_d$,

(3.2) if $\mathrm{root}(t) \in \mathcal{F}_d$ *then* π *is an* [on]*-step.*

Proof.
(1)&(2) We know from the completeness of LN_2 with strategy $\mathcal{S}_c$ that if we interpret strict equations as oriented equations then there exist an $\langle \mathrm{LN}_2, \mathcal{S}_c\rangle$-step $\pi : G \Rightarrow_\theta G'$ and a substitution $\gamma' \in \mathcal{U}_\mathcal{R}(G')$ which satisfy (1) and (2). We only have to show that if γ is a strict solution of $e' = s \rhd t \in G$ then γ' is a strict solution of the linear descendants of e' in G'. This can be shown by case distinction on the type of π.
(3) For (3.1), note that if $\mathrm{root}(t) \in \mathcal{F}_d$ then $t\gamma \notin \mathcal{T}(\mathcal{F}_c, \mathcal{V})$ which contradicts with our assumption on γ. Therefore we must have $\mathrm{root}(t) \notin \mathcal{F}_d$. For (3.2), we observe that if $\mathrm{root}(s) \in \mathcal{F}_d$ then any rewrite derivation $s\gamma \to^*_\mathcal{R} t\gamma$ must contain a rewrite step at the root position. Then we can choose π to be an [on]-step. □

This lemma suggests to define the specialization of LN_2^s consisting of the inference rules $[\mathrm{i}]^s$, $[\mathrm{p}]^s$, $[\mathrm{ov}]^s$, $[\mathrm{on}]^s$, $[\mathrm{ffs}]^s$, $[\mathrm{ffd}]^s$, $[\mathrm{d}]^s$ shown in Fig. 5.9.

We call LN_4 the calculus resulted from LN_2 by replacing the subcalculus LN_2^s with the one depicted in Fig. 5.9.

Because the calculus LN_4 is a specialization of LN_2, it results that LN_4 with strategy $\mathcal{S}_c$ is sound: whenever $G \Rightarrow^*_\theta F$ is an $\langle \mathrm{LN}_4, \mathcal{S}_c\rangle$-refutation and $\gamma \in \mathcal{U}_\mathcal{R}(F)$, we have that $\theta\gamma \in \mathcal{U}_\mathcal{R}(G)$.

Furthermore, the calculus LN_4 with strategy $\mathcal{S}_c$ is also complete for the computation of the strict $\mathcal{R}$-normalized solutions of a goal. From Lemma 41.(1)-(2) we learn that LN_4 with strategy $\mathcal{S}_c$ is a complete calculus for the computation of $\mathcal{R}$-normalized solutions. Even more, Lemma 41.(3) allows us to consider only $\langle \mathrm{LN}_4, \mathcal{S}_c\rangle$-refutations for which conditions (3.1) and (3.2) hold. Therefore, we can reduce the nondeterminism between the inference rules of LN_4 with strategy $\mathcal{S}_c$ for a selected equation $s \gg t$ as shown in Fig. 5.10.

5.10 Conditional Pattern Rewrite Systems

In this section we outline the possibility to extend the calculus LN_1 to the case of conditional pattern rewrite systems.

[d]^s *decomposition*

$$\frac{G, \lambda\overline{x}.v(\overline{s_n}) \cong \lambda\overline{x}.v(\overline{t_n}), G'}{G, \overline{\lambda\overline{x}.s_n \gg \lambda\overline{x}.t_n}, G'}$$

if $v \in \mathcal{F}_c \cup \{\overline{x}\}$ and $\cong \in \{\doteq, \gg\}$

[i]^s *imitation*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.g(\overline{t_m}), G'}{(G, \overline{\lambda\overline{x}.H_m(\overline{s_n}) \cong \lambda\overline{x}.t_m}, G')\theta}$$

where $\cong \in \{\doteq, \doteq^{-1}, \gg, \gg^{-1}\}$, $g \in \mathcal{F}_c$, $\theta = \{X \mapsto \lambda\overline{x_n}.g(\overline{H_m(\overline{x_n})})\}$ and $\overline{H_m}$ are fresh variables.

[p]^s *projection*

$$\frac{G, \lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.t, G'}{(G, \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n})}) \cong \lambda\overline{x}.t, G')\theta}$$

where $\cong \in \{\doteq, \doteq^{-1}, \gg, \gg^{-1}\}$, $1 \leq i \leq n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are fresh variables.

[ov]^s *outermost narrowing at variable position*

$$\frac{G, \lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.v(\overline{t}), G'}{(G, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \blacktriangleright \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.v(\overline{t}), G')\theta}$$

if $\cong \in \{\gg, \doteq, \doteq^{-1}\}$, $v \in \underline{\{\overline{x}\}} \cup \mathcal{F}_c$, $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule, $\theta = \{X \mapsto \lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types, and $\overline{s_m}$ are distinct bound variables only if the selected equation has precursors.

[on]^s *outermost narrowing at nonvariable position*

$$\frac{G, \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t, G'}{G, \overline{\lambda\overline{x}.s_n \blacktriangleright \lambda\overline{x}.l_n}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G'}$$

if $\cong \in \{\doteq, \doteq^{-1}, \gg\}$ and $f(\overline{l_n}) \to r$ is a fresh variant of an $\overline{x}$-lifted rule.

[ffs]^s *flex/flex same*

$$\frac{G, \lambda\overline{x}.X(\overline{y_n}) \cong \lambda\overline{x}.X(\overline{y'_n}), G'}{(G, G')\theta}$$

where $\cong \in \{\doteq, \gg\}$, $\theta = \{X \mapsto \lambda\overline{y_n}.H(\overline{z_p})\}$ with $\overline{y_n} \neq \overline{y'_n}$ and $\{\overline{z_p}\} = \{y_i \mid 1 \leq i \leq n \text{ and } y_i = y'_i\}$.

[ffd]^s *flex/flex different*

$$\frac{G, \lambda\overline{x}.X(\overline{y_m}) \cong \lambda\overline{x}.Y(\overline{y'_n}), G'}{(G, G')\theta}$$

where $\cong \in \{\doteq, \gg\}$, $\theta = \{X \mapsto \lambda\overline{y_m}.H(\overline{z_p}), Y \mapsto \lambda\overline{y'_n}.H(\overline{z_p})\}$ with $\{\overline{z_p}\} = \{\overline{y_m}\} \cap \{\overline{y'_n}\}$.

Fig. 5.9: Inference rules for strict equations

$_{\mathrm{root}(s)}\backslash^{\mathrm{root}(t)}$	$\mathcal{V}(t)$	F_d	$F_c \cup \mathcal{BV}(t)$
$\mathcal{V}(s)$	$[\mathrm{ffs}]^s/[\mathrm{ffd}]^s$	$\times$	$[\mathrm{i}]^s_1, [\mathrm{p}]^s_1, [\mathrm{ov}]^s$
F_d	$[\mathrm{on}]^s$	$\times$	$[\mathrm{on}]^s$
$F_c \cup \mathcal{BV}(s)$	$[\mathrm{i}]^s_2, [\mathrm{p}]^s_2$	$\times$	$[\mathrm{d}]^s$

Fig. 5.10: LN_4: Inference rules for equation $s \gg t$ selected with $sel \in \mathcal{S}_c$

$_{\mathrm{root}(s)}\backslash^{\mathrm{root}(t)}$	$\mathcal{V}(t)$	F_d	$F_c \cup \mathcal{BV}(t)$
$\mathcal{V}(s)$	$[\mathrm{ffs}]^s/[\mathrm{ffd}]^s$	$[\mathrm{on}]^s_2$	$[\mathrm{i}]^s_1, [\mathrm{p}]^s_1, [\mathrm{ov}]^s_1$
F_d	$[\mathrm{on}]^s_1$	$[\mathrm{on}]^s_1$	$[\mathrm{on}]^s_1$
$F_c \cup \mathcal{BV}(s)$	$[\mathrm{i}]^s_2, [\mathrm{p}]^s_2, [\mathrm{ov}]^s_2$	$[\mathrm{on}]^s_2$	$[\mathrm{d}]^s$

Fig. 5.11: LN_4: Inference rules for equation $s \doteq t$ selected with $sel \in \mathcal{S}_c$

Definition 51 (conditional PRS) *A* conditional PRS *is a set of conditional rewrite rules of the form* $f(\overline{l_n}) \to r \Leftarrow G$ *where*

- $f(\overline{l_n}), r$ *are terms of the same base type,*
- G *is a goal consisting of oriented and/or unoriented equations, and*
- $\mathcal{V}(r) \subseteq \mathcal{V}(f(\overline{l_n})) \cup G$.

Higher-order conditional rewriting with respect to a conditional PRS $\mathcal{R}$ is defined similarly to conditional term rewriting (see Ch. 3, Def. 21), by using the notion of higher-order rewriting given in Def. 35, and by interpreting $s \approx t$ as $s \downarrow_{\mathcal{R}} t$ and $s \rhd t$ as $s \to^*_{\mathcal{R}} t$.

The conditional counterpart of the calculus $\mathrm{LN}_{\mathrm{ff}}$ is the calculus $\mathrm{CLN}_{\mathrm{ff}}$ obtained by extending the inference rules for outermost narrowing to the conditional case, as follows:

[on] *outermost narrowing at nonvariable position*

$$\frac{G, \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t, G'}{G, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, \overline{\lambda\overline{x}.u_p \cong_p \lambda\overline{x}.v_p}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G'}$$

if $\cong \in \{\approx, \approx^{-1}, \rhd\}$ and $f(\overline{l_n}) \to r \Leftarrow \overline{u_p \cong_p v_p}$ is a fresh variant of an $\overline{x}$-lifted rule with $\cong_1, \ldots, \cong_p \in \{\approx, \rhd\}$.

[ov] *outermost narrowing at variable position*

$$\frac{G, \lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.t, G'}{(G, \overline{\lambda\overline{x}.H_n(\overline{s_m}) \rhd \lambda\overline{x}.l_n}, \overline{\lambda\overline{x}.u_p \cong_p \lambda\overline{x}.v_p}, \lambda\overline{x}.r \cong \lambda\overline{x}.t, G')\theta}$$

if $\cong\in\{\approx,\approx^{-1},\rhd\}$, $\lambda\overline{x}.t$ is rigid and $f(\overline{l_n})\to r\Leftarrow\overline{u_p\cong_p v_p}$ is a fresh variant of an $\overline{x}$-lifted rule and $\theta=\{X\mapsto\lambda\overline{y_m}.f(\overline{H_n(\overline{y_m})})\}$ with $\overline{H_n}$ fresh variables of appropriate types and $\cong_1,\ldots,\cong_p\in\{\approx,\rhd\}$.

We generalize the notions of descendant and linear descendant of an equation selected in an [on]- or [ov]-step as shown below.

α	e	$\mathrm{desc}_\pi(e)$	
		$\mathrm{ldesc}_\pi(e)$	
[on]	$\lambda\overline{x}.f(\overline{s_n})\cong\lambda\overline{x}.t$	$\overline{\lambda\overline{x}.u_p\cong_p\lambda\overline{x}.v_p}$ $\lambda\overline{x}.r\cong\lambda\overline{x}.t$	$\overline{\lambda\overline{x}.s_n\rhd\lambda\overline{x}.l_n}$
[ov]	$\lambda\overline{x}.X(\overline{s_m})\cong\lambda\overline{x}.t$	$\overline{\lambda\overline{x}.u_p\cong_p\lambda\overline{x}.v_p}$ $\lambda\overline{x}.r\cong\lambda\overline{x}.t$	$\overline{\lambda\overline{x}.H_n(\overline{s_m\theta})\rhd\lambda\overline{x}.l_n}$

The notion of precursor is generalized as follows:

Definition 52 (precursor) *If* $\Pi: G\Rightarrow^0 G$ *is an empty* $\mathrm{CLN}_{\mathrm{ff}}$*-derivation then* $\mathrm{prec}_\Pi(e)=\Box$ *for all* $e\in G$.

If $\Pi: G\Rightarrow^* G_1,e,G_2\Rightarrow_\alpha G'=G_1',e',G_2'$ *is a nonempty* $\mathrm{CLN}_{\mathrm{ff}}$*-derivation such that* $e'\in\mathrm{desc}(e)$, π *is the last* $\mathrm{CLN}_{\mathrm{ff}}$*-step of* Π, *and* Π' *is the* $\mathrm{CLN}_{\mathrm{ff}}$*-subderivation of* Π *without* π *then:*

- *if* $\alpha=[\mathrm{on}]$ *and* $e=\lambda\overline{x}.f(\overline{s_n})\cong\lambda\overline{x}.t$ *then*
  - $\mathrm{prec}_\Pi(e'):=G'',\overline{\lambda\overline{x}.s_n\blacktriangleright\lambda\overline{x}.l_n},\overline{\lambda\overline{x}.u_p\cong_p\lambda\overline{x}.v_p}$ *if* $e'=\lambda\overline{x}.r\cong\lambda\overline{x}.t$
  - $\mathrm{prec}_\Pi(e')=G'',\overline{\lambda\overline{x}.s_n\blacktriangleright\lambda\overline{x}.l_n}$ *if* $e'=\lambda\overline{x}.u_k\cong_k\lambda\overline{x}.v_k$ *for some* $k\in\{1,\ldots,p\}$,
  - $\mathrm{prec}_\Pi(e')=G''$ *if* $e'=\lambda\overline{x}.s_k\blacktriangleright\lambda\overline{x}.l_k$ *for some* $k\in\{1,\ldots,n\}$,
- *if* $\alpha=[\mathrm{ov}]$ *and* $e=\lambda\overline{x}.X(\overline{s_m})\cong\lambda\overline{x}.t\theta$ *then*
  - $\mathrm{prec}_\Pi(e')=G'',\overline{\lambda\overline{x}.H_n(\overline{s_m\theta})\blacktriangleright\lambda\overline{x}.l_n},\overline{\lambda\overline{x}.u_p\cong_p\lambda\overline{x}.v_p}$ *if* $e'=\lambda\overline{x}.r\cong\lambda\overline{x}.t\theta$
  - $\mathrm{prec}_\Pi(e')=G'',\overline{\lambda\overline{x}.H_n(s_m\theta)\blacktriangleright\lambda\overline{x}.l_n}$ *if* $e'=\lambda\overline{x}.u_k\cong_k\lambda\overline{x}.v_k$ *for some* $k\in\{1,\ldots,p\}$,
  - $\mathrm{prec}_\Pi(e')=G''$ *if* $e'=\lambda\overline{x}.H_k(\overline{s_m\theta})\blacktriangleright\lambda\overline{x}.l_k$ *for some* $k\in\{1,\ldots,n\}$,
- G'', *otherwise*

where $G'':=\mathrm{desc}_\pi(\mathrm{prec}_{\Pi'}(e))\setminus e$.

The strategies $\mathcal{S}_0$, $\mathcal{S}_n$ and $\mathcal{S}_c$ are defined like in the unconditional case (Def. 43, pp. 98). By a similar reasoning it can be shown that the calculus $\mathrm{CLN}_{\mathrm{ff}}$ is sound and complete if we adopt the equation selection strategy $\mathcal{S}_0$ or $\mathcal{S}_n$.

It can be shown that some of the refinements towards more determinism of the calculus $\mathrm{LN}_{\mathrm{ff}}$ with strategy $\mathcal{S}_n$ can be generalized to the calculus $\mathrm{CLN}_{\mathrm{ff}}$ with strategy $\mathcal{S}_n$. Two possible refinements of LCN_1 with strategy $\mathcal{S}_n$ are:

1. LCN_1 with strategy $\mathcal{S}_n$: the conditional counterpart of the calculus LN_1 with strategy $\mathcal{S}_n$ (Sect. 5.4),

2. LCN_2 with strategy $\mathcal{S}_c$: the conditional counterpart of the calculus LN_2 with strategy $\mathcal{S}_c$.

We claim without proof that

- LCN_1 with strategy $\mathcal{S}_n$ is sound and complete if $\mathcal{R}$ is confluent,
- LCN_2 with strategy $\mathcal{S}_c$ is sound and complete if $\mathcal{R}$ is left-linear and confluent.

5.11 Conclusion

A summary of the lazy narrowing calculi proposed in this chapter and of the refinements achieved with them is shown in Fig. 5.12.

We want to emphasize that these refinements can be effectively used to drive the computation of a functional logic program. The calculus LN_1 with strategy $\mathcal{S}_n$ has been integrated into the functional logic component of the distributed constraint functional logic system CFLP [MIS99b], and the integration of the other refinements presented in this chapter is under way.

Fig. 5.13 depicts the dependencies among the lazy narrowing calculi that inspired our research. The calculi written with slanted boldfaced fonts are those proposed by us. The equation selection strategies corresponding to the lazy narrowing calculi are mentioned in parentheses.

Note that all calculi proposed by us are not restricted to terminating term rewriting systems. The restriction to terminating rewrite systems is quite strong in functional logic programming. Our concern was to design calculi which can generate a complete set of $\mathcal{R}$-normalized $\mathcal{R}$-unifiers with respect to (various classes of) confluent PRSs.

Another extension which adds expressive power to functional logic programs are conditional pattern rewrite systems. A proposal for a sound and

	Properties	Strategy	PRS	Deterministic refinements
$\mathrm{LN_{ff}}$	sound (Lemma 23)			
$\mathrm{LN_1}$	sound,complete (Lemma 30)	$\mathcal{S}_0, \mathcal{S}_n$	confluent	-[ov] for $\lambda\overline{x}.X(\overline{y}) \cong \lambda\overline{x}.t$ without precursors
LN_1^{ev}	sound,complete (Lemma 32)	$\mathcal{S}_n$	orthogonal	-[on] for $\lambda\overline{x}.f(\overline{s}) \blacktriangleright \lambda\overline{x}.X(\overline{y})$ with $f \in \mathcal{F}_d$
$\mathrm{LN_2}$	sound,complete (Lemma 34, 35)	$\mathcal{S}_c$	left-linear confluent	[rm] for redundant equations
$\mathrm{LN_3}$	sound,complete (Lemma 37,40)	$\mathcal{S}_c$	left-linear confluent fully extended	$\not\exists$ equations $\lambda\overline{x}.s \blacktriangleright \lambda\overline{x}.f(\overline{t})$ with $f \in \mathcal{F}_d$
$\mathrm{LN_4}$	sound,complete (Lemma 41)	$\mathcal{S}_c$	left-linear confluent	Efficient subcalculus for equations with strict semantics (Figs. 5.9–5.11)

Fig. 5.12: Lazy narrowing calculi for pattern rewrite systems

complete lazy narrowing calculus for confluent and terminating conditional PRS is given in [Pre98]. The development of an efficient lazy narrowing calculus for larger classes of conditional PRSs is an extremely challenging direction of research. In Sect. 5.10 we proposed the calculus $\mathrm{CLN_{ff}}$ as the conditional counterpart of $\mathrm{LN_{ff}}$ and mentioned a few theoretical results that can be lifted from $\mathrm{LN_{ff}}$ to $\mathrm{CLN_{ff}}$. We conjecture that $\mathrm{LN_{ff}}$ is a good starting point for the development of an expressive and powerful calculus for equational theories represented by confluent conditional PRSs.

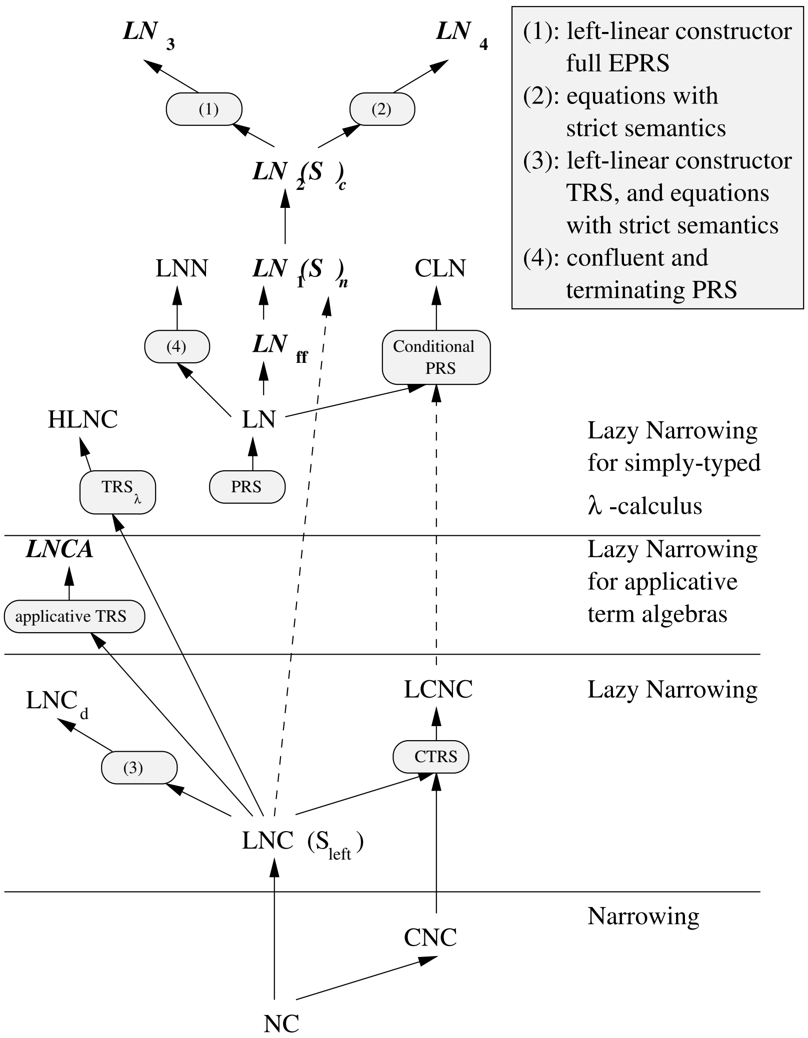


Fig. 5.13: Narrowing calculi: dependency diagram

Chapter 6

Cooperative Constraint Functional Logic Programming

In this chapter we introduce a cooperative constraint functional logic programming scheme which combines features of functional logic programming and cooperative constraint solving. The scheme is intended to combine the advantages of cooperative constraint solving (e.g., efficient and powerful methods to solve large and complex problems) with the features of functional logic programming.

We describe our scheme as an extension of the traditional constraint programming scheme CP($\mathcal{X}$) in two directions:

- support for cooperative constraint solving. The scheme is parameterized with a strategy $\mathcal{S}$ which defines the way how a set of components solvers $\{CS_1, \ldots, CS_n\}$ of the constraint domain $\mathcal{X}$ cooperate upon solving systems of constraints. We abstract this extension in a scheme CP($\mathcal{X}, \mathcal{S}$),

- program construction facilities. We adopt a functional logic programming style to support one's own abstractions by means of user programs. The reduction of a problem containing user defined symbols to a problem that can be solved by the CP($\mathcal{X}, \mathcal{S}$) scheme (i.e., without user-defined constructs) is achieved with a so called constraint lazy narrowing calculus $\mathcal{C}$. Defining an effective operational principle for solving systems of constraints involving user defined operators boils down to the design of a suitable combination of the operational semantics of the scheme $\mathrm{CP}(\mathcal{X},\mathcal{S})$ with the constraint lazy narrowing calculus $\mathcal{C}$. We propose a scheme $\mathrm{CFLP}(\mathcal{X},\mathcal{S},\mathcal{C})$ which is intended to clarify the meaning of the combination of the two mutually dependent operational principles.

In a nutshell, our constraint functional logic programming scheme can be written schematically

$$\mathrm{CFLP}(\mathcal{X},\mathcal{S},\mathcal{C}) = \mathrm{CP}(\mathcal{X},\mathcal{S}) + \mathrm{FLP}(\mathcal{X},\mathcal{C})$$

where

- $\mathcal{X}$ is the underlying constraint domain of the system,
- $\mathcal{S}$ is a strategy that defines the operational semantics of solving systems of constraints over $\mathcal{X}$ with a cooperation of a given collection $\{CS_1, \ldots, CS_n\}$ of constraint solvers,
- $\mathcal{C}$ is a calculus for solving equations containing user defined constructs.

This chapter is structured as follows. In Sect. 6.1 we recall the traditional constraint programming scheme $\mathrm{CP}(\mathcal{X})$.

In Sects. 6.2–6.4 we describe extensions of the traditional constraint programming scheme with some desirable features.

In Sect. 6.2 we address the possibility to extend $\mathrm{CP}(\mathcal{X})$ with additional constraint solving capabilities by providing support for concurrent constraint solving. The main concern of this extension is how to make cooperate different constraint solvers defined over the same constraint domain but with different admissible constraints. We introduce an additional argument $\mathcal{S}$ called *strategy*, which formalizes the mechanism of constraint solving cooperation.

In Sect. 6.3 we discuss another extension which dramatically improves the expressive power of constraint programming: support for defining one's own abstractions—user defined functions—by means of constrained functional logic programs. Such a feature requires the extension of the computational mechanism of (concurrent) constraint programming with a mechanism to solve equations involving user defined function symbols. For handling user defined abstractions we add to the the $\mathrm{CP}(\mathcal{X},\mathcal{S})$ scheme a new parameter $\mathcal{C}$ which denotes a constraint lazy narrowing calculus. The resulted scheme should integrate the advantages of both declarative programming frameworks of $\mathrm{CP}(\mathcal{X},\mathcal{S})$ and functional logic programming based on a calculus $\mathcal{C}$. Therefore we call it concurrent constraint functional logic programming, and denote it by $\mathrm{CFLP}(\mathcal{X},\mathcal{S},\mathcal{C})$. The main concern of concurrent constraint functional logic programming is to define a clear operational semantics that combines the calculus $\mathcal{C}$ with the operational principle of CP($\mathcal{X}, \mathcal{S}$). Since the calculus $\mathcal{C}$ dramatically influences the overall performance of the system, it is required to detect effective calculi. We claim that the design of an effective constrained lazy narrowing calculus can take benefit of the results achieved in FLP by generalizing to CFLP the essential features underlying the deterministic refinements of lazy narrowing. In Chapter 7 we discuss how the design of the underlying calculi of our system `CFLP` was influenced by the lazy narrowing calculi introduced in Chapter 5.

Finally, in Sect. 6.4 we define a distributed model of CFLP($\mathcal{X}, \mathcal{S}, \mathcal{C}$). The distributed model is intended to serve as basis of implementations of constraint functional logic programming systems which make use of constraint solving resources located in a distributed environment, such as heterogeneous network of computers.

6.1 The CP($\mathcal{X}$) Scheme

Constraint programming (CP) is based on the idea of specifying a problem by a set of constraints. A constraint is simply a logical relation among several unknowns (or variables), which take a value in a given domain $\mathcal{X}$. This domain has a well known algebraic structure equipped with natural algebraic operations such as addition or multiplication, and with privileged predicates such as equality and various forms of inequality.

What is essential for the use of constraints for programming and computing purposes is that constraints introduce a uniform framework for manipulating partial information. Every constraint is a piece of information about objects. There are some trivial connectives (mainly only conjunction) provided in the system as the basic mechanism to be more precise.

Programming is viewed as specifying a problem by a set of constraints. The system is equipped with a mechanism (the *solver*) which solves the problem by computing its canonical form as result.

In the sequel we give a formal account to the notions of constraint domain and solver to capture the basic characteristics of constraint programming outlined above.

Constraint Domain

The concept of constraint domain formalizes the idea of constraint. Our presentation is an adaptation of the terminology in [Mon96] to simply-typed signatures.

We assume that a simply-typed signature of a constraint domain can be represented in the form $\Sigma = \langle S_0 \cup \{bool\}, \mathcal{F} \cup \Pi \rangle$ such that:

- $\{\mathtt{true} : bool, \mathtt{false} : bool\} \cup \{\approx_\tau \mid \tau \in S\} \subseteq \Pi$,

- $\mathcal{F} \cap (\Pi \setminus \{\texttt{true}, \texttt{false}\}) = \emptyset$, where S is the inductive closure of $S_0 \cup \{bool\}$ under the function type constructor,
- if $p \in \Pi$ then $p : \overline{\tau_n} \to bool$.

The symbols of Π are called *predicates* and the symbols of $\mathcal{F}$ are regarded as *function symbols.* We define $fcts(\Sigma) := \mathcal{F}$, and assume that any Σ-algebra $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$ corresponding to a simply-typed signature $\Sigma = \langle S_0 \cup \{bool\}, \mathcal{F} \cup \Pi \rangle$ meets the following requirements:

- $A_{bool} = \{true, false\}$ where $\{true, false\}$ is a boolean domain, and
- $\alpha(\texttt{true}) = true, \alpha(\texttt{false}) = false$, $\alpha(\approx_\tau) = =_\tau$ where $=_\tau$ is the equality operator over A_τ.

Definition 53 (simple constraint domain) *A* simple constraint domain *is a quadruple $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ such that:*

- $\Sigma = \langle S_0 \cup \{bool\}, \mathcal{F} \cup \Pi \rangle$ *is a simply-typed signature,*
- $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$ *is a simply-typed Σ-algebra,*
- $\mathcal{V} = \{\mathcal{V}_\tau\}_{\tau \in S}$ *is an S-sorted set of* variables *such that $\mathcal{V}_\tau$ is countably enumerable for any $\tau \in S$ and $\mathcal{V} \cap (\mathcal{F} \cup \Pi) = \emptyset$,*
- *a subset $\Phi \subseteq \mathcal{E}q(\mathcal{F}, \mathcal{V}) \cup \{\texttt{true}, \texttt{false}\}$ with $\texttt{true}, \texttt{false} \in \Phi$. The set Φ is called the set of* basic constraints *of $\mathcal{X}$.*

We use ϕ, possibly with subscript, to range over basic constraints and denote by ∇ the set of all S-sorted $\mathcal{A}$-valuations $v : \mathcal{V} \to |\mathcal{A}|$.

The meaning of a constraint can be given by its set of solutions.

Definition 54 *Given a constraint domain $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ and a constraint $\phi \in \Phi$, a* solution of ϕ *is an $\mathcal{A}$-valuation $v \in \nabla$ such that*

$$\mathcal{A} \models v^*(\phi)$$

where $v^* : \mathcal{T}(fcts(\Sigma), \mathcal{V}) \to |\mathcal{A}|$ denotes the homomorphic extension of v (see Def. 1, pp. 9). We denote by $[\![\phi]\!]^{\mathcal{A}}$ the set of solutions of ϕ.

If V is a finite set of variables, notation $V \subseteq_{\text{fin}} \mathcal{V}$, then we define

$$[\![\phi]\!]^{\mathcal{A}}_V := \{v \mid \exists v' \in [\![\phi]\!]^{\mathcal{A}} \text{ such that } v\restriction_V = v'\restriction_V\}.$$

The notions of interpretation, model and satisfaction relation $\models$ between a simply-typed Σ-algebra and a constraint of a simple system are defined as usual. The notions of satisfiability and validity can be restated as follows:

- ϕ is *satisfiable* if $[\![\phi]\!]^{\mathcal{A}} \neq \emptyset$, otherwise it is *unsatisfiable.*
- ϕ is *valid* if $[\![\phi]\!]^{\mathcal{A}} = \nabla$.

A preorder can be defined over Φ to model the richness of the information contained in a constraint.

Definition 55 (refinement ordering) *Let $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ be a simple constraint domain, $V \subseteq_{\text{fin}} \mathcal{V}$, and $\phi_1, \phi_2 \in \Phi$. The constraint ϕ_1 is a V-*refinement *of ϕ_2, written as $\phi_2 \leq_{|V} \phi_1$ if $[\![\phi_1]\!]^{\mathcal{A}}_{V} \subseteq [\![\phi_2]\!]^{\mathcal{A}}_{V}$. ϕ_1 is a* refinement *of ϕ_2, written as $\phi_2 \leq \phi_1$, if ϕ_1 is a $\mathcal{V}$-refinement of ϕ_2.*

It is straightforward to see that the refinement ordering is a preorder over constraints which satisfies the following properties:

$$
\begin{array}{l}
\phi \leq \phi \\
\texttt{true} \leq \phi \leq \texttt{false} \\
\phi_1 \leq \phi_2 \wedge \phi_2 \leq \phi_3 \Rightarrow \phi_1 \leq \phi_3
\end{array}
$$

We denote by $\equiv$ the equivalence relation induced by the preorder $\leq$ on constraints.

Simple constraint domains provide the formalism for constraint systems. In order to define operators over constraints, we impose a logical structure over Φ by applying logical connectives. The most important connectives are conjunction and disjunction.

Definition 56 (constraint domain) *A constraint domain is a quadruple $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi' \rangle$ where $\langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ is a simple constraint domain and Φ' is the inductive closure of Φ defined as follows:*

$$\Phi \subseteq \Phi',$$

$$\xi_1, \xi_2 \in \Phi' \Rightarrow \xi_1 \wedge \xi_2 \in \Phi',$$

$$\xi_1, \xi_2 \in \Phi' \Rightarrow \xi_1 \vee \xi_2 \in \Phi'.$$

The expression $\xi_1 \wedge \xi_2$ is called the conjunction *of ξ_1 and ξ_2, and the expression $\xi_1 \vee \xi_2$ is called the* disjunction *of ξ_1 and ξ_2.*

The functions $\mathcal{V} : \Phi \to \mathcal{V}$ and $[\![\cdot]\!]^{\mathcal{A}} : \Phi \to 2^{\nabla}$ are extended to Φ' as follows:

- $\mathcal{V}(\xi_1 \wedge \xi_2) := \mathcal{V}(\xi_1 \vee \xi_2) := \mathcal{V}(\xi_1) \cup \mathcal{V}(\xi_2)$,
- $[\![\xi_1 \wedge \xi_2]\!]^{\mathcal{A}} := [\![\xi_1]\!]^{\mathcal{A}} \cap [\![\xi_2]\!]^{\mathcal{A}}$ *and* $[\![\xi_1 \vee \xi_2]\!]^{\mathcal{A}} := [\![\xi_1]\!]^{\mathcal{A}} \cup [\![\xi_2]\!]^{\mathcal{A}}$.

In the sequel we assume that $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ is a constraint domain. We call Φ the *computation domain* of $\mathcal{X}$.

We observe that the following properties of constraint conjunction and disjunction hold:

$$(CS_1) : \begin{cases} \phi_1 \wedge \phi_2 & \equiv & \phi_2 \wedge \phi_1 \\ \phi \wedge \phi & \equiv & \phi \\ (\phi_1 \wedge \phi_2) \wedge \phi_3 & \equiv & \phi_1 \wedge (\phi_2 \wedge \phi_3) \\ \phi \wedge \mathtt{true} & \equiv & \phi \\ \phi \wedge \mathtt{false} & \equiv & \mathtt{false} \end{cases}$$

$$(CS_2) : \begin{cases} \phi_1 \vee \phi_2 & \equiv & \phi_2 \vee \phi_1 \\ \phi \vee \phi & \equiv & \phi \\ \phi \vee \mathtt{false} & \equiv & \phi \\ \phi \vee \mathtt{true} & \equiv & \mathtt{true} \\ (\phi_1 \vee \phi_2) \vee \phi_3 & \equiv & \phi_1 \vee (\phi_2 \vee \phi_3) \\ \phi_1 \wedge (\phi_2 \vee \phi_3) & \equiv & (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3) \end{cases}$$

We denote by $\equiv_s$ the congruence relation induced by (CS_1) and (CS_2) on Φ. In the sequel we will identify any two constraints which are $\equiv_s$-congruent.

The notion of binary constraint conjunction can be extended to the conjunction of a (possibly infinite) set of constraints. From the properties mentioned before results that a conjunction of constraints can be simply represented as a set of constraints. We write $\bigwedge_{i=1}^{n} \phi_i$ for the conjunction $\phi_1 \wedge \ldots \wedge \phi_n$.

Conjunction is the most important operator over constraints, which allows to specify an object by stating several independent properties by using constraints and then putting them together.

The properties of the logical connectives $\wedge$ and $\vee$ allow us to write a constraint $M \in \Phi$ in an equivalent disjunctive normal form

$$\bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} \phi_{i,j}$$

where $\phi_{i,j}$ are primitive constraints. Such a constraint is called *constraint store*; it consists of m disjuncts $\xi_i := \bigwedge_{j=1}^{n_i} \phi_{i,j}$ called *elementary constraint stores*. We denote by $\Phi_\wedge$ the set of elementary constraint stores of Φ, by Φ_p the set of primitive constraints of Φ, and adopt the following conventions of notation:

- M, M_1, M_2, $\ldots$ range over Φ,
- ϕ, ϕ', ϕ_1, ϕ_2, $\ldots$ range over Φ_p,
- ξ, ξ', ξ_1, ξ_2, $\ldots$ range over the set $\Phi_\wedge$ of elementary constraint stores.

Solver

Intuitively, a solver over a constraint domain $\langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ is an algorithm which transforms an elementary constraint store into a constraint which is 'simpler' than ξ but equivalent to ξ in $\mathcal{A}$ (a solver preserves the solutions). Moreover, the repeated application of a solver always terminates and reaches a fixed point called *solved* form or *canonical* form. Usually, the solved form of an elementary constraint store ξ has a syntactic structure from which the set of solutions $[\![\phi]\!]^{\mathcal{A}}$ can be easily derived.

Definition 57 (solver) *A* solver *on a constraint domain* $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ *is a computable function* $CS : \Phi \to \Phi$ *which has the following properties:*

disjunction: $CS(\bigvee_{i=1}^{N} \xi_i) := \bigvee_{i=1}^{N} CS(\xi_i)$,

soundness: $\forall \xi \in \Phi_\wedge,\ \xi \leq CS(\xi)$,

completeness: $\forall \xi \in \Phi_\wedge,\ CS(\xi) \leq \xi$,

termination or fixed-point: $\forall \xi \in \Phi_\wedge, \exists n \in \mathbb{N},\ CS^{n+1}(\xi) = CS^{n}(\xi)$.

We denote by *Solver*($\mathcal{X}$) the set of component solvers over $\mathcal{X}$. In order to ease the integration of the operational principles of cooperative constraint solving and lazy narrowing for functional logic programming, we represent the output $CS(\xi)$ of a constraint solver CS in the form $\bigvee_{k=1}^{N} \langle [\theta_k], \xi_k \rangle$ where

- $\theta_k \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ is an idempotent substitution and $\xi_k \in \Phi_k$ such that $\mathcal{V}(\xi_k) \cap \mathcal{D}(\theta_k) = \emptyset$,
- $[\theta_k] := \bigwedge_{X \in \mathcal{D}(\theta_k)} (X \approx X\theta_k)$,
- $\langle [\theta_k], \xi_k \rangle$ denotes the constraint $[\theta_k] \wedge \xi_k$,
- $CS([\theta_k] \wedge \xi_k) = \bigvee_{j=1}^{N'} \langle [\theta_k \theta'_j], \xi'_j \rangle$ where $\bigvee_{j=1}^{N'} \langle \theta'_j, \xi'_j \rangle = CS(\xi_k)$

for any $k \in \{1, \ldots, N\}$. We use this representation to capture the fact that the constraint solving computational mechanism increases the information contained in a constraint, i.e., the instantiation of logical variables.

A solver induces an ordering on constraints w.r.t. the solver. Intuitively, the result of a solver is smaller than its input, i.e. we can identify a constraint solver with a simplifier.

Definition 58 (solver ordering) *Let* CS *be a solver on the constraint system* $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$. *Then the solver ordering induced by* CS *on* $\Phi_\wedge$ *is defined as follows:* $\xi \leq_{CS} \xi'$ *if* $\exists n \in \mathbb{N}$ *such that* $\bigvee_{k=1}^{N} \langle [\theta_k], \xi_k \rangle = CS^n(\xi)$ *and* $\xi' = \xi_k$ *for some* $k \in \{1, \ldots, N\}$.

The relation $\leq_{CS}$ on Φ is a quasi-ordering, and the related ordering $<_{CS}$ is noetherian.

With our representation, we can regard a solver CS as the set of *simplification rules* $\{\xi \overset{\mathcal{CS}}{\Longrightarrow}_{\theta_k} \xi_k \mid CS(\xi) = \bigvee_{k=1}^{N} \langle [\theta_k], \xi_k \rangle, \xi \not\equiv_s CS(\xi)\}$. Similar to lazy narrowing derivations, a *CS-derivation* is a sequence

$$\xi \overset{\mathcal{CS}}{\Longrightarrow}_{\theta_1} \xi_1 \overset{\mathcal{CS}}{\Longrightarrow}_{\theta_2} \ldots \overset{\mathcal{CS}}{\Longrightarrow}_{\theta_n} \xi_n$$

of CS-simplification steps, abbreviated $\xi \overset{\mathcal{C}}{\Longrightarrow}{}^{*}_{\theta} \xi_n$ where $\theta = \theta_1 \ldots \theta_n$. A *CS-refutation* is a CS-derivation $\xi \overset{\mathcal{CS}}{\Longrightarrow}{}^{*}_{\theta} \xi_n$ such that there is no CS-step starting with ξ_n.

Computing the CS-canonical form of an elementary constraint store ξ is realized by generating all the CS-refutations $\xi \overset{\mathcal{CS}}{\Longrightarrow}{}^{*}_{\theta_k} \xi_k$, $1 \leq k \leq N$. Then the canonical form of ξ is the disjunction of the elements of the set $CF_{CS}(\xi) := \{\langle \theta_k, \xi_k \rangle \mid k \in \{1, \ldots, n\}\}$.

6.1.1 Extensions

The main drawbacks of the traditional constraint programming scheme are:

1. *Lack of program construction*: the only means to describe a problem is by using the language of constraints of the underlying constraint domain. A constraint program represents the content of a constraint store. There is no means of abstraction or mechanism of programming to specify the dynamic construction of a constraint store.

2. *Fixed computational domain*: for efficiency reasons, the underlying constraint solver of a constraint programming language is restricted to some classes of constraints, i.e. Φ is a strict subset of the set of constraints supported by the language. This means that the constraint solver is in general *incomplete*, i.e. unable to solve all constraints supported by the language. The design of a complete solver is a hard and lengthy task which usually yields an inefficient solver.

Recent years have witnessed a growing interest to improve the traditional CP scheme by eliminating these drawbacks. The following two sections give an account to the ongoing research which addresses these problems.

6.2 The CP($\mathcal{X}, \mathcal{S}$) Scheme

In this section we introduce a formalism that supports the second improvement of the traditional CP($\mathcal{X}$) scheme: the possibility to extend the computation domain of a constraint domain and to improve the solving capabilities by replacing the built-in constraint solver with a cooperation of constraint solvers.

6.2.1 State of the Art

The main idea of cooperative constraint programming is to avoid the design of a general-purpose constraint solver by replacing it with a collection of specialized constraint solvers equipped with a suitable mechanism of cooperation that supports solving complex problems that none of the individual solvers can handle alone. Thus, the effort of designing an efficient and powerful constraint solver (such as a quantifier elimination solver over real closed fields) is shifted to designing a strategy for cooperating constraint solvers. The importance and advantages of adopting this approach are widely recognized and drafts for proposals of cooperative constraint solving systems started to appear.

Most of the systems proposed so far (such as the cooperative architecture of Marti-Rueher [MR95] or `CoSAc` [Mon96]) are focused on the easy integration, cooperation, and reusability of various constraint solvers. In general, the design of such an integrated system requires a good knowledge of the basic components (constraint manager, communication manager, etc.) in order to add new solvers or to replace integrated solvers.

Monfroy [Mon96] proposes `BALI` (Binding Architecture for SoLver Integration), a domain independent environment for executing solver collaborations. In his view, 'solver collaboration' means 'solver combination' and 'solver cooperation', where 'solver combination' focuses on building a solver for the union of theories, and 'solver cooperation' is concerned with communication problems between solvers defined over the same constraint domain but with possibly different admissible constraints.

6.2.2 Enrichment

Enrichments are the means to extend the set of constraints accepted by a solver.

Definition 59 (constraint domain enrichment) *Let*
$\mathcal{X} = \langle\langle S_0 \cup \{bool\}, \mathcal{F} \cup \Pi\rangle, \mathcal{A}, \mathcal{V}, \Phi\rangle$
$\mathcal{X}' = \langle\langle S_0' \cup \{bool\}, \mathcal{F}' \cup \Pi'\rangle, \mathcal{A}', \mathcal{V}', \Phi'\rangle$
be two constraint domains. Then $\mathcal{X}'$ is an enrichment *of $\mathcal{X}$ if:*

1. *$S_0 = S_0'$, $\mathcal{F} \subseteq \mathcal{F}'$ and $\Pi = \Pi'$*
2. *$|\mathcal{A}| = |\mathcal{A}'|$ and for all $g \in \mathcal{F} \cup \Pi$, $g^{\mathcal{A}'} = g^{\mathcal{A}}$ (i.e., the interpretation of g is the same in $\mathcal{A}$ and $\mathcal{A}'$)*

3. $\mathcal{V} = \mathcal{V}'$ *and* $\Phi \subseteq \Phi'$.

Thus, the enrichment of a constraint domain $\mathcal{X}$ consists of some additional functions defined on the original domain.

We now extend the class of constraints a solver *CS* can manipulate. The new constraints belong to an enrichment of the constraint domain of *CS*.

Definition 60 (*CS*-admissible primitive constraint) *Let* $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ *be a constraint system, CS be a solver on* $\mathcal{X}' = \langle \Sigma', \mathcal{A}', \mathcal{V}', \Phi' \rangle$ *and* $\phi \in \Phi_p$. *Then* ϕ *is a CS-admissible primitive constraint over* $\mathcal{X}$ *if* $\phi \in \Phi'$ *and* $\mathcal{X}$ *is an enrichment of* $\mathcal{X}'$.

In order to extend the notion of *CS*-admissible primitive constraint to all kind of constraints, we introduce the following definition.

Definition 61 (*CS*-admissible part of a constraint) *Let* $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ *be a constraint domain, CS be a solver over* $\mathcal{X}' = \langle \Sigma', \mathcal{A}', \mathcal{V}', \Phi' \rangle$ *and* $S \in \Phi_\wedge$. *Then the CS-*admissible part of ξ over $\mathcal{X}$, *denoted by* $adm_{CS,\mathcal{X}}(\xi)$, *is recursively defined as follows:*

- $adm_{CS,\mathcal{X}}(\phi \wedge \xi) := \phi \wedge adm_{CS,\mathcal{X}}(\xi)$ *if* ϕ *is a CS-admissible primitive constraint over* $\mathcal{X}$,
- $adm_{CS,\mathcal{X}}(\phi \wedge \xi) := adm_{CS,\mathcal{X}}(\xi)$ *if* ϕ *is not a CS-admissible primitive constraint over* $\mathcal{X}$,
- $adm_{CS,\mathcal{X}}(\texttt{true}) = \texttt{true}$, $adm_{CS,\mathcal{X}}(\texttt{false}) = \texttt{false}$.

We denote by $\overline{adm}_{CS,\mathcal{X}}(\xi)$ *the CS-nonadmissible part of* ξ.

We extend a constraint solver by allowing it to manipulate constraints that are not defined over its own constraint domain.

Definition 62 (solver enrichment) *Let CS be a solver over* $\mathcal{X}'$ *and* $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ *be an enrichment of* $\mathcal{X}'$. *Then the* enrichment of *CS over* $\mathcal{X}$ *is defined by*

$$CS^e(\xi) := \bigvee_{k=1}^{N} \langle [\theta_k], \xi_k \wedge \xi'' \theta_k \rangle$$

where $adm_{CS,\mathcal{X}'}(\xi) = \xi'$, $\overline{adm}_{CS,\mathcal{X}'}(\xi) = \xi''$ *and* $CS(\xi') = \bigvee_{k=1}^{N} \langle [\theta_k], \xi_k \rangle$.

The following lemma is not hard to prove.

Lemma 42 *If CS is a solver over a constraint domain* $\mathcal{X}$ *then the solver enrichment* CS^e *is a solver over* $\mathcal{X}$.

For the sake of simplicity, we will drop the superscript ^{e} and write *CS* instead of CS^e.

6.2.3 Solver Cooperation

Solver cooperation is a formalism which enables to combine solving techniques over constraint domains in an attempt to yield a solver for the union of the individual constraint domains.

Definition 63 (solver cooperation) *Let $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ be a constraint domain and $CS_1, \ldots, CS_n$ be n solvers defined respectively over $\mathcal{X}_1, \ldots, \mathcal{X}_n$. Then $CS_1, \ldots, CS_n$* cooperate over *$\mathcal{X}$, notation $\langle \overline{CS_n} \rangle_{\mathcal{X}}$, if $\mathcal{X}$ is an enrichment of $\mathcal{X}_i$ for all $i \in \{1, \ldots, n\}$.*

In other words, a solver cooperation for a constraint domain $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ is a set of constraint solvers, each of them being defined over a certain subset of Φ. We call the individual solvers of a cooperation *component constraint solvers.*

In order to solve constraints in Φ with a solver cooperation $\langle \overline{CS_n} \rangle$, we have to clarify how the individual constraint solvers cooperate. The main ingredients of describing mechanisms for solver cooperation are the fixed point operator, the special solvers, and the concept of strategy for solver cooperation.

A powerful formalism to describe solver cooperation is the notion of strategy.

The fixed point operator

Assume $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ is a constraint domain. The termination property of the set $\mathit{Solver}(\mathcal{X})$ guarantees the following property:

$$\forall CS \in \mathit{Solver}(\mathcal{X}). \exists m \in \mathbb{N}. \forall \xi \in \Phi_{\wedge}. CS^{m+1}(\xi) = CS^{m}(\xi)$$

where $CS^0 := Id$, $CS^{n+1} := CS \circ CS^n$,

As a consequence, we can define the operator $\mathrm{fp} : \mathit{Solver}(\mathcal{X}) \to \mathit{Solver}(\mathcal{X})$, $\mathrm{fp}(CS) = CS^m$ where $m \in \mathbb{N}$ is the least natural number such that

$$\forall CS \in \mathit{Solver}(\mathcal{X}). \forall \xi \in \Phi_{\wedge}. CS^{m+1}(\xi) = CS^{m}(\xi).$$

The operator $\mathrm{fp} : \mathit{Solver}(\mathcal{X}) \to \mathit{Solver}(\mathcal{X})$ is called the *fixed point operator.*

Special solvers

A suitable formalism for solver cooperation should take into account the preprocessing operations that transform constraints into a suitable input to the individual constraint solvers. Such operations include translations of constraints into a suitable syntactic form, syntactic purification operations, etc. A convenient way to describe these operations is to regard them as special constraint solvers.

To illustrate, we consider the variable abstraction solver which performs syntactic purification in an attempt to transform non-admissible constraints into admissible ones.

Example 10 *Let $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ be a constraint domain with $\Sigma = \langle S_0 \cup \{bool\}, \mathcal{F} \cup \Pi \rangle$ such that:*

- $S_0 := \{complList, compl\}$, $|\mathcal{A}| := \{A_\tau\}_{\tau \in \{compl, complList, bool\}}$ *with:*

  A_{compl} :=*the domain* $\mathbb{C}$ *of complex numbers, and*
  $A_{complList}$:=*the domain* $List_{\mathbb{C}}$ *of lists of complex numbers.*

- $\mathcal{F}$ *contains the operators* $+$, $-$, $\cdot$, $\sqrt{\ }$, *cons, car, cdr, and the constant symbols with the usual interpretations, the constant symbols of the set* $\mathbb{C}$ *of complex numbers, and the empty list* $[\,]$.

- $\Pi := \{\approx_{\mathbb{C}}, \approx_{List_{\mathbb{C}}}, \mathtt{true}, \mathtt{false}\}$.

- $\Phi := \mathcal{E}q(\mathcal{F}, \mathcal{V}) \cup \{\mathtt{true}, \mathtt{false}\}$.

Let CS_1 be a solver over the constraint domain $\mathcal{X}_1 = \langle \Sigma_1, \mathcal{A}_1, \mathcal{V}, \Phi_1 \rangle$ with $\Sigma_1 = \langle S_0 \cup \{bool\}, \mathcal{F}_1 \cup \Pi_1 \rangle$, where:

- $\mathcal{F}_1 := \mathcal{F} \setminus \{+, -, \cdot, \sqrt{\ }\}$

- $\Pi_1 := \{\approx_{complList}, \mathtt{true}, \mathtt{false}\}$,

- $\Phi_1 := \mathcal{E}q(\mathcal{F}_1, \mathcal{V}) \cup \{\mathtt{true}, \mathtt{false}\}$

and let CS_2 be a solver over the constraint domain $\mathcal{X}_2 = \langle \Sigma_2, \mathcal{A}_2, \mathcal{V}, \Phi_2 \rangle$ with $\Sigma_2 = \langle S_0 \cup \{bool\}, \mathcal{F}_2 \cup \Pi_2 \rangle$, where:

- $\mathcal{F}_2 := \mathcal{F} \setminus \{cons, car, [\,]\}$ *with the same interpretation as in* $\mathcal{X}$,

- $\Pi_2 := \{\approx_{compl}, \mathtt{true}, \mathtt{false}\}$,

- *The equations in Φ_2 are those between multivariate polynomials with complex coefficients.*

Then $\mathcal{X}$ is an enrichment of $\mathcal{X}_1$ and $\mathcal{X}_2$ and thus, CS_1 and CS_2 can cooperate to solve constraints in Φ, such as

$$\xi : [X + Y + Z, X \cdot X + Y \cdot Y + Z \cdot Z] \approx [7, 9].$$

The constraint ξ is neither CS_1-admissible nor CS_2-admissible. But ξ_1 can be transformed into a conjunction of admissible constraints by replacing the polynomial expressions inside ξ with fresh variables:

$$\xi' : [X_1, X_2] \approx [7, 9] \wedge X_1 \approx X + Y + Z \wedge X_2 \approx X \cdot X + Y \cdot Y + Z \cdot Z.$$

The first equation of ξ' is CS_1-admissible, whereas the last two equations of ξ' are CS_2-admissible.

Definition 64 (aliens, pure and impure constraint) *Let $\Sigma = \langle S_0 \cup \{bool\}, \mathcal{F} \cup \Pi\rangle$ be a simply-typed signature, $\mathcal{X}_1 = \langle \Sigma_1, \mathcal{A}_1, \mathcal{V}, \Phi_1\rangle$, $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi\rangle$ be constraint domains such that $\mathcal{X}$ is an extension of $\mathcal{X}_1$.*

A Σ_1-pure *term (resp. constraint) is a Σ_1-term (resp. Σ_1-formula). A* Σ_1-alien *subterm of a Σ-term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a subterm $t|_p$ of t such that:*

- $\mathrm{root}(t|_q) \in \mathit{fcts}(\Sigma_1) \cup \mathcal{V}$ *for all* $q < p$,
- $\exists q < p$ *such that* $\mathrm{root}(t|_q) \in \mathit{fcts}(\Sigma_1)$, *and*
- $\mathrm{root}(t|_p) \in \mathcal{F} \setminus \mathit{fcts}(\Sigma_1)$.

The set of Σ_1-alien terms in an elementary constraint $\phi \in \Phi_\wedge$ is denoted by $Alien_{\Sigma_1}(\phi)$. An elementary constraint $\xi \in \Phi_\wedge$ is Σ_1-impure *if $Alien(\xi) \neq \emptyset$.*

When we describe a component solver CS_k on $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi\rangle$, it is convenient to express a constraint from $\Phi_\wedge$ as a pair $\langle \xi, \xi'\rangle_{\Sigma_k}$, where ξ is Σ_k-pure and ξ' is Σ_k-impure, and $\langle \xi, \xi'\rangle$ means the conjunction $\xi \wedge \xi'$. The subscript Σ_k will be omitted when it is understood from the context.

With this representation, the notion of CS_k-simplification step over Φ_k is extended to CS_k-simplification step over Φ in the following way:

$$\langle \xi, \xi'\rangle_{\Sigma_k} \stackrel{CS_k}{\Longrightarrow}_\theta \xi'' \wedge (\xi'\theta)$$

if $\xi \stackrel{CS_k^e}{\Longrightarrow}_\theta \xi''$. Accordingly, the characteristic properties of the component constraint solver CS_k can be restated as follows:

soundness: if $\langle \xi, \xi'\rangle_{\Sigma_k} \stackrel{CS_k}{\Longrightarrow}_\theta \xi'' \wedge \xi'\theta$ and $\gamma \in [\![\xi'' \wedge \xi'\theta]\!]^{\mathcal{A}}$ then $(\theta\gamma)\restriction_{\mathcal{V}(\xi,\xi')} \in [\![\xi' \wedge \xi'']\!]^{\mathcal{A}}$

completeness: for any $\gamma \in [\![\xi \wedge \xi']\!]^{\mathcal{A}}$, if $\xi \wedge \xi'$ is not a CS_k-canonical form then there exists a simplification step $\langle \xi, \xi'\rangle_{\Sigma_k} \stackrel{CS_k}{\Longrightarrow}_\theta \xi'' \wedge \xi'\theta$ with $\gamma = \theta\gamma'$ $[\mathcal{D}(\gamma)]$ for some $\gamma' \in [\![\xi'' \wedge \xi'\theta]\!]^{\mathcal{A}}$

termination: there are no infinite simplification derivations

$$\xi = \xi_0 \stackrel{CS_k}{\Longrightarrow}_{\theta_1} \xi_2 \stackrel{CS_k}{\Longrightarrow}_{\theta_2} \cdots$$

Definition 65 (variable abstraction solver) *Let $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ be a constraint domain and $\langle \overline{CS_n} \rangle_{\mathcal{X}}$ a solver cooperation such that CS_k is a solver over $\mathcal{X}_k = \langle \Sigma_k, \mathcal{A}_k, \mathcal{V}, \Phi_k \rangle$ $(k = 1, \ldots, n)$. The* abstraction solver `Abs` over $\mathcal{X}$ *is characterized by simplification rules of the form:*

$$s \simeq t \wedge \xi \rightarrow_{\varepsilon}^{\text{Abs}} s[X(\overline{y_n})]_q \simeq t \wedge (\lambda \overline{y_n}.X(\overline{y_n}) \approx \lambda \overline{y_n}.s|_q) \wedge \xi$$

such that

- $\text{root}(s) \in \Sigma_k$ *for some* $k \in \{1, \ldots, n\}$,
- $\{\overline{y_n}\} = \mathcal{BV}(s, q)$,
- $s|_q \in Alien_{\Sigma_k}(s \simeq t)$,
- *X is a fresh variable of appropriate type.*

The variable abstraction solver transforms impure constraints into pure ones by adding fresh variables of appropriate types. We call such variables *abstraction* variables.

It can be shown that the variable abstraction solver is a solver over $\mathcal{X}$ and that if $\xi \in \Phi$ then the `Abs`-canonical form of $\xi_\wedge$ is a disjunction of conjunctions of pure formulas.

Strategies for solver cooperation

A strategy for solver cooperation describes the way in which the component solvers $\langle \overline{CS_n} \rangle_{\langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle}$ of a solver cooperation cooperate upon solving systems of constraints in Φ.

Cooperation primitives (or *combinators*) [Mon96] provide a powerful formalism to specify cooperation strategies. Each cooperation primitive formalizes a different mechanism for solver cooperation. We recall here the cooperation primitives which are most commonly used: sequentiality, parallelism, and concurrency.

Sequentiality is the most straightforward way of combining solvers: it enables a constraint solver to act on a constraint store which is the result of another constraint solver.

Definition 66 (sequential composition) *Let $\langle CS_1, CS_2 \rangle_{\mathcal{X}}$ be a solver cooperation over a constraint domain $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$. The sequential composition of the solvers CS_1, CS_2 is the function $CS_1; CS_2 : \Phi_\wedge \to \Phi$ which is defined by*

$$CS_1; CS_2(\xi) := CS_2(CS_1(\xi))$$

for any $\xi \in \Phi_\wedge$.

The parallel combinator formalizes the parallel execution of several component solvers. The result of such a cooperation is the conjunction of the results of each solver.

Definition 67 (parallel combinator) *Let $\langle \overline{CS_n} \rangle_{\mathcal{X}}$ be a solver cooperation. The* parallel combination *over $\mathcal{X}$ of $\overline{CS_n}$ is the function $CS_1 \| \ldots \| CS_n : \Phi_\wedge \to \Phi$ defined by:*

$$CS_1 \| \ldots \| CS_n(\xi) := \bigwedge_{k=1}^{n} CS_k(\xi).$$

The principal use of the concurrency combinator is the following: given a solver cooperation $\langle \overline{CS_n} \rangle_{\mathcal{X}}$, the concurrent combination of $\overline{CS_n}$ enables the concurrent execution of the solvers and keeps the result of the most efficient one. This primitive is of interest when the solvers are time consuming (e.g., Gröbner basis computations or solvers of Diophantine equations.)

Definition 68 (concurrency combinator) *Let $\langle \overline{CS_n} \rangle_{\mathcal{X}}$ be a solver cooperation. A* concurrent combination *of $\overline{CS_n}$ is a function $CS_1 ? \ldots ? CS_n : \Phi_\wedge \to \Phi$ such that*

$$\forall \xi \in \Phi_\wedge, \exists k \in \{1, \ldots, n\}, CS_1 ? \ldots ? CS_n(\xi) = CS_k(\xi)$$

We describe a strategy for a solver cooperation $\langle \overline{CS_n} \rangle_{\mathcal{X}}$ as an expression built over the union of the set $\{\overline{CS_n}\}$ with a set Id_{sp} of identifiers of special solvers (e.g., $Id_{sp} = \{\mathtt{Abs}\}$), by applying the fixed-point operator fp and the cooperation combinators ;, $\|$ and ?.

For example, one could define the strategy $\mathcal{S}_{\mathrm{fp}}(\overline{CS_n}) := \mathrm{fp}(\mathtt{Abs}); \mathrm{fp}(\overline{CS_n})$. This strategy was deeply investigated by Hong [Hon94, Hon92a], and is frequently used in the area of cooperative constraint solving.

The main difficulty in defining a suitable strategy $\mathcal{S}$ for a solver cooperation $\langle \overline{CS_n} \rangle_{\mathcal{X}}$ is to find reasonable restrictions on the cooperation that guarantee that $\mathcal{S}(\overline{CS_n})$ is a constraint solver. In general, the main problem is to ensure termination. For example, sequential composition $CS_1; CS_2$ is not necessarily a constraint solver: whereas it can be shown that $CS_1; CS_2$ is a sound and complete function, it may be the case that it does not have the termination property.

Proving the termination property of a sequential combination $CS_1; CS_2$ of component solvers is usually done by showing that there exists a quasi-ordering $\leq$ such that $\leq_{CS_1} \subseteq \leq$ and $\leq_{CS_2} \subseteq \leq$.

The scheme

The scheme CP($\mathcal{X}, \mathcal{S}$) differs from CP($\mathcal{X}$) in two respects:

1. whereas CP($\mathcal{X}$) has implicit a constraint solver, the scheme CP($\mathcal{X}, \mathcal{S}$) keeps implicit a solver cooperation $\langle \overline{CS_n} \rangle_{\mathcal{X}}$,
2. a second parameter is provided to make explicit the strategy which describes the way how the component solvers $CS_1, \dots, CS_n$ cooperate.

6.3 The CFLP($\mathcal{X}, \mathcal{S}, \mathcal{C}$) Scheme

In general, when modeling complex problems that require constraint solving capabilities, the user wants to have a mechanism which supports the specification of his own abstractions. This additional functionality can be obtained by means of user-level programs. That is, a constraint programming system should provide a mechanism to extend the expressive power of the underlying language with new function and/or predicate symbols defined in terms of the function and/or predicate symbols already recognized by the system. This view of programming suggests to define a scheme that extends the constraint programming scheme with a mechanism to enrich the underlying constraint domain $\mathcal{X}$ with user defined function symbols.

The rest of this section is structured as follows. Subsection 6.3.1 gives an overview of the state of the art on declarative constraint programming. In Subsect. 6.3.2 we describe our framework which supports the extension of the constraint programming scheme with user defined function symbols over a constraint domain.

6.3.1 State of the Art

The most popular framework which supports the integration of constraint solving capabilities with programming is constraint logic programming (CLP) introduced by Jaffar and Lassez [JL87, JM87]. In this framework, the scheme CLP($\mathcal{X}$) defines a language based on Horn clausal logic which is augmented with the capacity of solving constraints in a particular constraint domain $\mathcal{X}$. We now briefly describe here the CLP languages that have received substantial development effort.

CLP($\mathcal{R}$) [JMSY90] has linear arithmetic constraints and computes over the domain of real numbers. Nonlinear constraints are ignored (delayed) until they become effectively linear. CHIP [DvHS$^+$88] and Prolog III [Col90] compute over several domains. Both compute over boolean domains: Prolog III over the 2-valued boolean algebra and CHIP over a larger boolean algebra that contains symbolic values. Both CHIP and Prolog III perform linear arithmetic over the rational numbers. CHIP also performs linear arithmetic over bounded subsets of integers (known as *finite domains*), and

Prolog III also computes over a domain of strings. CAL [ASS$^+$88] defines a family of CLP languages over algebraic, boolean, and linear algebraic domains. The constraint solving techniques include Gröbner bases [Buc85] and the Simplex method to solve linear algebraic constraints. In the prototype system RISC-CLP($\mathcal{R}$) [Hon92b] non-linear constraints are fully involved in the computation. Solving them is achieved with two algebraic methods: Partial Cylindrical Algebraic Decomposition and Gröbner bases. RISC-CLP(CF) supports constraints over the domain of complex functions, such as functional equalities, differential equations etc. Constraint solving is realized by iterating several solving methods such as Laplace transformation, non-linear equation solving, etc.

The first proposal for a constraint functional logic programming scheme (CFLP) seems to be [DGP91b, DGP91a], where a scheme for defining constrained functional logic languages over arbitrary constraint domains is presented. Roughly speaking, they define CFLP($\mathcal{X}$) as CLP(FP($\mathcal{X}$)), that is, as an instance of CLP($\mathcal{X}$) where the base structure $\mathcal{X}$ is enriched with new functions defined by means of a functional language, and where the constraints include equations between FP($\mathcal{X}$) expressions, to be solved by narrowing. In [LF92, LF94] López proposes a CFLP($\mathcal{X}$) scheme in which programs are built as sets of constrained rewrite rules in untyped first-order logic and nondeterministic functions are disallowed. Similarly to CLP, CFLP($\mathcal{X}$) programs have a least model which is a conservative extension of the constraint domain $\mathcal{X}$; a calculus called *constraint lazy narrowing* is proposed as a sound and complete operational mechanism for solving CFLP goals. Due to the presence of defined functions, the design of a CFLP scheme is much more challenging than that of CLP. Firstly, it is necessary to make the constraint solving and the lazy narrowing operational principles mutually dependent. Secondly, the equations can not be treated as constraints as in constraint programming, but as joinability and/or reducibility statements whose meaning has to be clarified. López does this by adopting a cpo-based semantics, where he computes over a Scott domain and all the functions and/or predicates are continuous.

Another logical framework for the design of a declarative programming language is CRWL (Constructor-based conditional ReWriting Logic) [GMGRA96, GMGRA97, GMGRA99]. This scheme supports nondeterministic functions whose combination with lazy evaluations turns out to be helpful. An experimental system called $\mathcal{TOY}$ [LFSH99] was implemented based on the CRWL paradigm and extended recently with support for algebraic datatypes and higher-order functions.

Recently, some work has been done to combine CFLP with CRWL. The paper [ASLFRA99] proposes a first-order CFLP language, called SETA, whose domain is a combination of primitive and user-defined domains.

SETA's primitive domain is given by a constraint domain $\mathcal{X}$, as in CLP($\mathcal{X}$). In addition, the user can define his own datatypes using a polymorphic type constructor for multisets and arbitrary free type constructors.

6.3.2 Constraint Functional Logic Programming

We employ the functional logic programming style for extending the expressive power of CP($\mathcal{X}$, $\mathcal{S}$) with new function symbols. We call our scheme CFLP($\mathcal{X}$, $\mathcal{S}$, $\mathcal{C}$), and can describe it as

$$\mathrm{CFLP}(\mathcal{X},\mathcal{S},\mathcal{C}) = \mathrm{CP}(\mathcal{X},\mathcal{S}) + \mathrm{FLP}(\mathcal{C})$$

where

- $\mathcal{X}$ is a constraint domain equipped with a solver cooperation $\langle\overline{CS_n}\rangle_{\mathcal{X}}$,
- $\mathcal{S}$ is a strategy that describes for the solver cooperation $\langle\overline{CS_n}\rangle_{\mathcal{X}}$,
- $\mathcal{C}$ is a calculus for solving systems of equations involving function symbols defined by the user. Such a calculus is called a *constraint lazy narrowing calculus*, and it is the result of generalizing a lazy narrowing calculus to act over equational goals constructed over a signature which, besides defined function symbols and constructors, contains additional operators whose meaning is given by the algebra of the underlying constraint domain (so called *external* operators).

We regard a system based on the CFLP($\mathcal{X}$, $\mathcal{S}$, $\mathcal{C}$) scheme as a combination of two subsystems:

1. a constraint solving subsystem which is in charge to solve the constraints of the goal. This subsystem is an implementation of the scheme CP($\mathcal{X}$, $\mathcal{S}$) determined by the explicit parameters $\mathcal{X}$ and $\mathcal{S}$ and the implicit solver cooperation $\langle\overline{CS_n}\rangle_{\mathcal{X}}$,
2. a functional logic subsystem which handles the user defined abstractions in the goal. The operational semantics of this component is described with the calculus $\mathcal{C}$.

These two operational principles are mutually dependent, and the main problem is to find a sound way to integrate them.

In the rest of this section we formalize the basic ideas outlined above.

The Language

We assume given

- a constraint domain $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ where
  - $\Sigma = \langle S_0 \cup \{bool\}, \mathcal{F}_e \cup \Pi \rangle$ is a simply-typed signature and S is the inductive closure of $S_0 \cup \{bool\}$ under the function type constructor. The symbols in $\mathcal{F}_e$ are called *external symbols*,
  - $\mathcal{A} = \langle \{A_\tau\}_{\tau \in S}, \alpha \rangle$,
  - $\Pi = \{\approx_\tau : \tau, \tau \to bool, \rhd_\tau : \tau, \tau \to bool \mid \tau \in S\}$
    $\cup \{\mathtt{true} : bool, \mathtt{false} : bool\}$,
- a finite set $\mathcal{F}_d$ of simply-typed symbols called *defined symbols*,
- a finite set $\mathcal{F}_c$ of simply-typed symbols called *constructors*

such that $\mathcal{F}_c$, $\mathcal{F}_d$, $\mathcal{F}_e \setminus \{\mathtt{true}, \mathtt{false}\}$, Π and $\mathcal{V}$ are mutually disjoint.

We define $\mathcal{F} := \mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{F}_e$, $\Sigma' := \langle S, \mathcal{F} \cup \Pi \rangle$, and the set Φ' as the closure of the set $\Phi_p \cup \mathcal{E}q(\mathcal{F}, \mathcal{V})$ under conjunction and disjunction.

Definition 69 (constrained rewrite rule) *A* constrained rewrite rule *over a constraint domain $\mathcal{X} = \langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ is an expression of the form*

$$f(\overline{l_n}) \to r \Leftarrow E$$

where

- *$f(\overline{l_n}), r \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ are terms of the same base type,*
- *$f \in \mathcal{F}_d$ and $f(\overline{l_n})$ is a pattern,*
- *$E \in \mathcal{E}q(\mathcal{F}, \mathcal{V})^*$ is a sequence of equations representing their conjunction, such that $\mathcal{V}(r) \subseteq \mathcal{V}(f(\overline{l_n})) \cup \mathcal{V}(E)$.*

The term $f(\overline{l_n})$is called the left-hand side*, r the* right-hand *side, and E the* conditional part *of the constrained rewrite rule. If $E = \Box$ then we simply write $f(\overline{l_n}) \to r$.*

Constrained rewrite rules provide the means to define one's own functions over a given constraint domain.

Definition 70 (program) *A* CFLP program *is a set of constrained rewrite rules.*

Definition 71 (equation) *An* unoriented equation *is an expression of the form $s \approx t$ where s, t are terms of the same type. An* oriented equation *is an expression of the form $s \rhd t$ where s, t are terms of the same type.*

Definition 72 (strict equation) *An* unoriented strict equation *is an expression of the form $s \doteq t$ where s, t are terms of the same type. An* oriented strict equation *is an expression of the form $s \gg t$ where s, t are terms of the same type. A* strict equation *is either an unoriented strict equation or an oriented strict equation.*

Strict equations are also called *equations with strict semantics*. We denote the set of equations and strict equations by $\mathcal{E}q(\mathcal{F}, \mathcal{V})$.

Definition 73 (constrained rewriting) *The* constrained rewriting relation *induced by a CFLP program $\mathcal{R}$ is the binary relation on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ defined as follows:*

$$\lambda\overline{x}.s \to_{\mathcal{R}} \lambda\overline{x}.t$$

if there exist

- *a position $p \in \mathcal{P}os(\lambda\overline{x}.s)$ with $\{\overline{y}\} = \mathcal{BV}(\lambda\overline{x}.s, p)$,*
- *$f \in \mathcal{F}_d$ such that $(\lambda\overline{x}.s)|_p = f(\overline{s_n})$,*
- *an $\overline{y}$-lifted fresh variant of a rule $f(\overline{l_n}) \to r \Leftarrow E$, and*
- *a substitution $\theta \in \mathcal{T}(\mathcal{F}, \mathcal{V})$*

such that

(a) $\lambda\overline{x}.t = (\lambda\overline{x}.s)[r\theta]_p$, and $\lambda\overline{x}.s_k = \lambda\overline{x}.l_k\theta$ for all $k \in \{1, \dots, n\}$.

(b) If $\lambda\overline{y}.s' \approx \lambda\overline{y}.t' \in \{E\theta\}$ then $\lambda\overline{y}.s' \downarrow_{\mathcal{R}} \lambda\overline{y}.t'$ where $\downarrow_{\mathcal{R}}$ is the joinability relation induced by $\to_{\mathcal{R}}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$

(c) If $\lambda\overline{y}.s' \doteq \lambda\overline{y}.t' \in \{E\theta\}$ then $\exists \lambda\overline{x}.u \in \mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$ such that $\lambda\overline{y}.s' \to^*_{\mathcal{R}} \lambda\overline{x}.u$ and $\lambda\overline{y}.t' \to^*_{\mathcal{R}} \lambda\overline{x}.u$,

(d) If $\lambda\overline{y}.s' \rhd \lambda\overline{y}.t' \in \{E\theta\}$ then $\lambda\overline{y}.s' \to^*_{\mathcal{R}} \lambda\overline{y}.t'$

(e) If $\lambda\overline{y}.s' \gg \lambda\overline{y}.t' \in \{E\theta\}$ then $\lambda\overline{x}.t' \in \mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$ and $\lambda\overline{y}.s' \to^*_{\mathcal{R}} \lambda\overline{y}.t'$

Thus we interpret $\rhd$ as $\to_{\mathcal{R}}$-reducibility, $\approx$ as $\to_{\mathcal{R}}$-joinability, and $\gg$ and $\doteq$ as their strict counterparts.

Given a CFLP program $\mathcal{R}$ over a constraint domain $\mathcal{X}$, one can ask questions about the properties that hold in the intended model of $\mathcal{R}$. Such questions are called *queries*.

Definition 74 (query) *A* query *(or* goal*) is a formula* $\bigwedge_{k=1}^{N} e_k$ *where* $e_1, \ldots, e_N \in \mathcal{E}q(\mathcal{F}, \mathcal{V})$.

An answer to a query G is a substitution θ such that $G\theta$ holds in the rewrite logic associated with the given program $\mathcal{R}$. Proving that $\theta \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ is an answer of G is done by *constrained rewriting.*

Definition 75 (solution) *A* solution *(or* answer*) of a goal* $G = \bigwedge_{k=1}^{N} e_k$ *with respect to a CFLP program* $\mathcal{R}$ *is a substitution* $\theta \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$ *such that for any equation* $e \in \{e_k \mid 1 \leq k \leq N\}$ *we have*

- $\lambda\overline{x}.s\theta \downarrow_{\mathcal{R}} \lambda\overline{x}.t\theta$ *if* e *is of the form* $\lambda\overline{x}.s \approx \lambda\overline{x}.t$,
- *there exists a term* $\lambda\overline{x}.u \in \mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$ *such that* $\lambda\overline{x}.s\theta \rightarrow_{\mathcal{R}} \lambda\overline{x}.u$ *and* $\lambda\overline{x}.t\theta \rightarrow_{\mathcal{R}} \lambda\overline{x}.u$ *if* e *is of the form* $\lambda\overline{x}.s \doteq \lambda\overline{x}.t$,
- $\lambda\overline{x}.s\theta \rightarrow^*_{\mathcal{R}} \lambda\overline{x}.t\theta$ *if* e *is of the form* $\lambda\overline{x}.s \rhd \lambda\overline{x}.t$,
- $\lambda\overline{x}.s\theta \rightarrow^*_{\mathcal{R}} \lambda\overline{x}.t\theta$ *and* $\lambda\overline{x}.t\theta \in \mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$ *if* e *is of the form* $\lambda\overline{x}.s \gg \lambda\overline{x}.t$.

We denote by $Ans_{\mathcal{R}}(G)$ *the set of solutions of a goal* G *with respect to* $\mathcal{R}$.

Solving a goal G with respect to a CFLP program $\mathcal{R}$ amounts to computing a representation of $Ans_{\mathcal{R}}(G)$. Our main concern is to define an efficient calculus to solve CFLP goals. For this purpose we employ a calculus called *constrained lazy narrowing.*

Constrained lazy narrowing

We employ a calculus for solving CFLP goals which is based on a top-down execution methodology similar to lazy narrowing: it incrementally binds the logical variables in the goal by performing so called *constrained lazy narrowing steps.* Constrained lazy narrowing is the natural extension of lazy narrowing from functional logic programming with inference rules to solve constraints, i.e. equations between terms built without user defined function symbols. The constraints generated during constraint narrowing are collected into a constraint store and solved with a CP($\mathcal{X}$, $\mathcal{S}$) scheme.

In order to capture the incremental computation of a solution, we describe the constrained lazy narrowing calculus as a set of inference rules on triples of the form $\langle \theta \mid \{\overline{e_m}\} \mid \xi \rangle$ where the components of the triple have the following meaning:

- θ is the (partial) solution computed so far,

- $\{\overline{e_m}\}$ is the sequence of equations of the current goal which are still relevant for the functional logic component of the CFLP system,
- ξ is the set of constraints collected so far. For solving systems of constraints we employ a CP$(\mathcal{X}, \mathcal{S})$ scheme where $\mathcal{S}$ is a strategy for a cooperation $\langle\overline{CS_n}\rangle_{\mathcal{X}}$ which defines a constraint solver over $\mathcal{X}$.

Such triples are called *states.* A state $\langle\theta \mid \{\overline{e_m}\} \mid \xi\rangle$ denotes the goal

$$([\theta] \wedge \bigwedge_{e \in \{\overline{e_m}\} \cup \xi} e).$$

The sequence $\{\overline{e_m}\}$ is called the *functional logic store of the state*, and ξ is called the *constraint store* of the state.

The inference rules below define our constraint lazy narrowing calculus as a natural extension of the lazy narrowing calculus CLN$_{\text{ff}}$ described in Sect. 5.10. The symbols E, E', E_1, E_2 stand for arbitrary sequences of equations.

[d] *decomposition*

$$\frac{\langle\gamma \mid \{E_1, \lambda\overline{x}.g(\overline{s_n}) \cong \lambda\overline{x}.g(\overline{t_n}), E_2\} \mid \xi\rangle}{\langle\gamma \mid \{E_1, \overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n}, E_2\} \mid \xi\rangle}$$

if $\cong \in \{\approx, \doteq, \rhd, \gg\}$ and $g \in \mathcal{F}_c$

$$\frac{\langle\gamma \mid \{E_1, \lambda\overline{x}.g(\overline{s_n}) \cong \lambda\overline{x}.g(\overline{t_n}), E_2\} \mid \xi\rangle}{\langle\gamma \mid \{E_1, \overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n}, E_2\} \mid \xi\rangle}$$

if $\cong \in \{\approx, \rhd\}$ and $g \in \mathcal{F}_d$

[on] *constrained narrowing at nonvariable position*

$$\frac{\langle\gamma \mid \{E_1, \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t, E_2\} \mid \xi\rangle}{\langle\gamma \mid \{E_1, \overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}, E, \lambda\overline{x}.r \cong \lambda\overline{x}.t, E_2\} \mid \xi\rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \doteq, \doteq^{-1}, \rhd, \gg\}$ and $f(\overline{l_n}) \to r \Leftarrow E$ is a fresh $\overline{x}$-lifted variant of a rule in $\mathcal{R}$

[ov] *constrained narrowing at variable position*

$$\frac{\langle\gamma \mid \{E_1, \lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.t, E_2\} \mid \xi\rangle}{\langle\gamma\theta \mid \{E_1\theta, \overline{\lambda\overline{x}.H_n(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_n}, E, \lambda\overline{x}.r \cong \lambda\overline{x}.t\theta, E_2\theta\} \mid \xi\theta\rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \doteq, \rhd, \gg\}$, $\lambda\overline{x}.t$ is a rigid term, $f(\overline{l_n}) \to r \Leftarrow E$ is a fresh $\overline{x}$-lifted variant of a rule in $\mathcal{R}$ and $\theta = \{X \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$ with $\overline{H_m}$ fresh variables of appropriate types.

[del] *deletion*

$$\frac{\langle \gamma \mid \{E_1, \lambda\overline{x}.t \approx \lambda\overline{x}.t, E_2\} \mid \xi\rangle}{\langle \gamma \mid \{E_1, E_2\} \mid \xi\rangle} \quad \frac{\langle \gamma \mid \{E_1, \lambda\overline{x}.t \rhd \lambda\overline{x}.t, E_2\} \mid \xi\rangle}{\langle \gamma \mid \{E_1, E_2\} \mid \xi\rangle}$$

[i] *imitation*

$$\frac{\langle \gamma \mid \{E_1, \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.X(\overline{y}), E_2\} \mid \xi\rangle}{\langle \gamma\theta \mid \{E_1\theta, \overline{\lambda\overline{x}.s_n\theta \cong \lambda\overline{x}.X_n}, E_2\theta\} \mid \xi\theta\rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \doteq, \doteq^{-1}, \rhd, \rhd^{-1}, \gg, \gg^{-1}\}$, $\theta = \{X \mapsto \lambda\overline{y}.f(\overline{X_n(\overline{y})})\}$ where $f \in \mathcal{F}_c \cup \mathcal{F}_d$ and $\overline{X_n}$ are fresh variables of appropriate types.

[p] *projection*

$$\frac{\langle \gamma \mid \{E_1, \lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.t, E_2\} \mid \xi\rangle}{\langle \gamma\theta \mid \{E_1\theta, \lambda\overline{x}.s_i(\overline{H_p(\overline{s_n\theta})}) \cong \lambda\overline{x}.t\theta, E_2\theta\} \mid \xi\theta\rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \doteq, \doteq^{-1}, \rhd, \rhd^{-1}, \gg, \gg^{-1}\}$, $1 \leq i \leq n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are fresh variables.

[va] *variable abstraction*

$$\frac{\langle \gamma \mid \{E_1, \lambda\overline{x}.s \cong \lambda\overline{x}.t, E_2\} \mid \xi\rangle}{\langle \gamma \mid \{E_1, \lambda\overline{x}.s[X(\overline{y})]_q \cong \lambda\overline{x}.t, \lambda\overline{y}.s|_p \cong \lambda\overline{y}.X(\overline{y}), E_2\} \mid \xi\rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \doteq, \doteq^{-1}, \rhd, \rhd^{-1}, \gg, \gg^{-1}\}$, $\mathrm{root}(\lambda\overline{x}.s) \in \mathcal{F}_e$, $\{\overline{y}\} = \mathcal{BV}(\lambda\overline{x}.s, p)$ and $(\lambda\overline{x}.s)|_p \in \mathit{Alien}(\lambda\overline{x}.s)$

[ff] *flex/flex equations*

$$\frac{\langle \gamma \mid \{E_1, \lambda\overline{x}.X(\overline{y_n}) \cong \lambda\overline{x}.X(\overline{y'_n}), E_2\} \mid \xi\rangle}{\langle \gamma\theta \mid \{E_1\theta, E_2\theta\} \mid \xi\theta\rangle}$$

where $\cong \in \{\approx, \rhd, \gg, \doteq\}$, $\theta = \{X \mapsto \lambda\overline{y_n}.Z(\overline{z})\}$ with $\{\overline{z}\} = \{y_i \mid y_i = y'_i\}$ and Z is a fresh variable of appropriate type.

$$\frac{\langle \gamma \mid \{E_1, \lambda\overline{x}.X(\overline{y_n}) \cong \lambda\overline{x}.Y(\overline{y'_n}), E_2\} \mid \xi\rangle}{\langle \gamma\theta \mid \{E_1\theta, E_2\theta\} \mid \xi\theta\rangle}$$

where $\cong \in \{\approx, \rhd, \gg, \doteq\}$, $\theta = \{X \mapsto \lambda\overline{y_n}.Z(\overline{z}), Y \mapsto \lambda\overline{y'_m}.Z(\overline{z})\}$ with $\{\overline{z}\} = \{\overline{y_n}\} \cap \{\overline{y'_m}\}$ and Z is a fresh variable of appropriate type.

[cp+] *constraint propagation*

$$\frac{\langle \gamma \mid \{E_1, e, E_2\} \mid \xi\rangle}{\langle \gamma \mid \{E_1, E_2\} \mid \xi \cup \{e\}\rangle}$$

if $e \in \mathcal{E}q(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$.

[cp−]

$$\frac{\langle \gamma \mid \{E_1\} \mid \xi \cup \{e\}\rangle}{\langle \gamma \mid \{e, E_1\} \mid \xi\rangle}$$

if $e \notin \mathcal{E}q(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$.

[cs] *constraint solve*

$$\frac{\langle \gamma \mid \{E\} \mid \xi\rangle}{\langle \gamma\theta \mid \{E\theta\} \mid \xi'\rangle}$$

if $\xi \in 2^{\mathcal{E}q(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})}$ and $\langle \theta, \xi'\rangle \in CF_{\mathcal{S}(\overline{CS_n})}(\xi)$.

The inference rules [cp+] and [cp−] abstract away the cooperation between the functional logic component and the constraint solving subsystem; the [cs] rule encapsulates the computation performed by the a constraint solving subsystem that implements the scheme CP$(\mathcal{X}, \mathcal{S})$, and the rest of the rules describe the computation steps performed by the functional logic subsystem.

The computation of solutions of a given goal $\bigwedge_{k=1}^{m} e_k$ proceeds by generating so called *constrained lazy narrowing refutations*, i.e. constrained lazy narrowing derivations of the form: $\langle \varepsilon \mid \{\overline{e_m}\} \mid \emptyset\rangle \Rightarrow^* \langle \gamma \mid \Box \mid \xi\rangle$ with $\bigwedge_{\phi\in\xi} \phi$ an $\mathcal{S}(\overline{CS_n})$-canonical form. Obviously, the constrained lazy narrowing calculus is sound, i.e., if $G = \bigwedge_{k=1}^{n} e_k$ is a goal and $\langle \varepsilon \mid \{\overline{e_m}\} \mid \emptyset\rangle \Rightarrow^* \langle \gamma \mid \Box \mid \xi\rangle$ and $\theta \in [\![\bigwedge_{\phi\in\xi} \phi]\!]^{\mathcal{A}}$ then $\gamma\theta \in Ans_{\mathcal{R}}(G)$.

The constrained lazy narrowing calculus inherits from the calculus CLN$_{\rm ff}$ the sources of high nondeterminism that make it inconvenient for practical applications. Therefore, it is useful to identify acceptable restrictions of the CFLP programs and goals, such that our constrained lazy conditional narrowing calculus (CLCNC for short) can be refined towards more deterministic versions. The deterministic refinements are achieved by imposing a suitable selection strategy of the equation from the functional logic store and suitable priorities between the inference rules of CLCNC which can be applied to the selected equation.

Thus, we regard the argument $\mathcal{C}$ of the CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$ scheme as a deterministic refinement of CLCNC, characterized by a suitable equation selection strategy and a suitable priorities for the inference rules applicable to the selected equation.

Similar to the design of the deterministic refinements of the calculus LN$_{\rm ff}$ presented in Chapter 5, the design of an effective constrained lazy narrowing calculus $\mathcal{C}$ is determined by the following considerations:

1. The properties of the CFLP program. The most important property is confluence, i.e. if $\lambda\overline{x}.s \leftrightarrow^*_{\mathcal{R}} \lambda\overline{x}.t$ then $\lambda\overline{x}.s \downarrow_{\mathcal{R}} \lambda\overline{x}.t$. Other important properties are such as left-linearity, orthogonality, etc.,

2. The class of solutions which we want to compute. In functional logic programming we are usually interested in computing $\mathcal{R}$-normalized substitutions. In Chapter 5 we have seen how this restriction can be used to make deterministic refinements of the calculus LN_1, .e.g, to restrict the application of outermost narrowing at variable position. A similar restriction of the constrained narrowing at variable position can be achieved by adopting a similar notion of $\mathcal{R}$-normalized solution and generalizing the notion of precursor to constrained lazy narrowing,

3. The semantics of equations. For example, by distinguishing equations with strict semantics, it is possible to define a more efficient calculus.

To describe the nondeterminism which is inherent to the computations performed with the calculus $\mathcal{C}$, we introduce the concept of *configuration.*

Definition 76 (configuration) A configuration *is a sequence of* n *states* $\overline{A_n}$*, which represents their logical disjunction.*

The initial configuration of the system is $\langle\varepsilon \mid \{\overline{e_m}\} \mid \emptyset\rangle$ where $\bigwedge_{k=1}^{m} e_m$ is the initial goal.

If we want to find all the solutions of a goal $\bigwedge_{k=1}^{m} e_k$ which can be computed with $\mathcal{C}$, we enumerate all the $\mathcal{C}$-refutations starting with $\langle\varepsilon \mid \{\overline{e_m}\} \mid \emptyset\rangle$ in accordance with a suitable search strategy. We adopt the breadth-first strategy defined by the following inference rules:

[cfg] *inference rules for configurations*

$$\frac{A, \overline{A_n}}{\overline{A_n}, \overline{A'_k}} \tag{6.1}$$

if A is $\mathcal{C}$-reducible and $\langle\gamma \mid \{\overline{e_m}\} \mid \xi\rangle \stackrel{\mathcal{C}}{\Longrightarrow} A'_1, \ldots, A \stackrel{\mathcal{C}}{\Longrightarrow} A'_k$ are all the possible $\mathcal{C}$-steps starting with A.

$$\frac{A, \overline{A_n}}{\overline{A_n}, A} \tag{6.2}$$

if A is $\mathcal{C}$-irreducible and there is $j > 1$ such that A_j is $\mathcal{C}$-reducible.

The state A is called the state *selected* in the [cfg]-step.

6.4 A Distributed Model of CFLP($\mathcal{X},\mathcal{S},\mathcal{C}$)

In Fig. 6.1 we illustrate the overall architecture of a distributed system based on the CFLP($\mathcal{X},\mathcal{S},\mathcal{C}$) scheme.

The system consists of two subsystems:

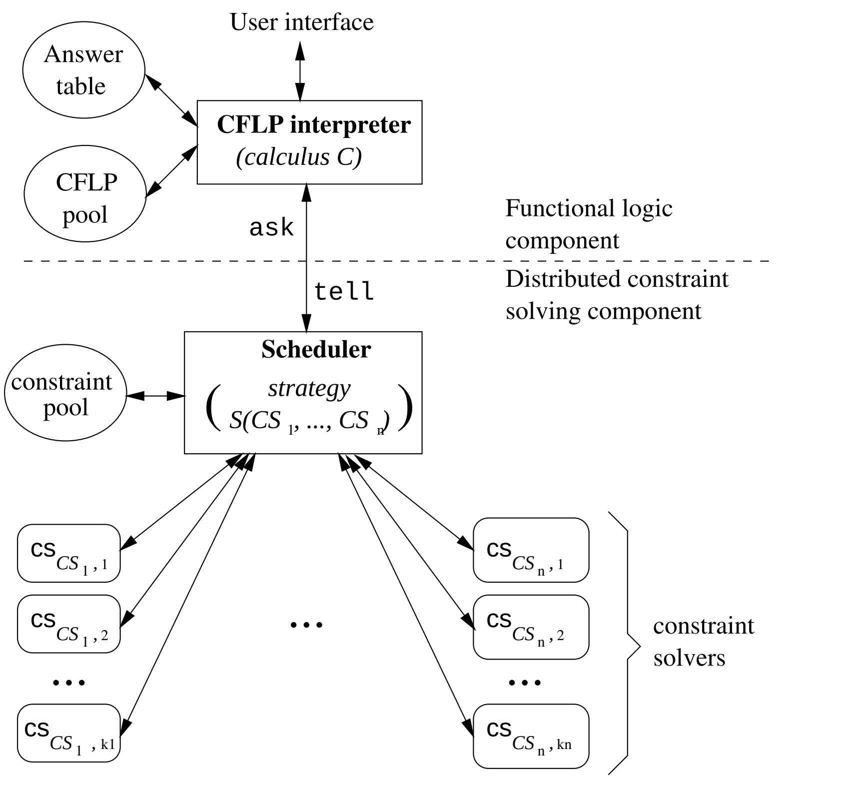


Fig. 6.1: CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$: distributed architecture

1. a CFLP interpreter based on a constrained lazy narrowing calculus $\mathcal{C}$,

2. a distributed constraint solving subsystem which implements a scheme CP($\mathcal{X}, \mathcal{S}$) for a cooperation $\langle \overline{CS_n} \rangle_{\mathcal{X}}$ such that $\mathcal{S}(\overline{CS_n})$ is a component solver on $\mathcal{X}$.

The constraint solving subsystem consists of:

1. a number of constraint solving resources which implement the constraint solving capabilities of the components of $\langle \overline{CS_n} \rangle_{\mathcal{X}}$. Every resource is an implementation of a constraint solver, and resources may be located in a distributed environment, such as a computer network. We denote by $\overline{\mathtt{cs}_{CS_i,k_i}}$ the sequence of available resources $\mathtt{cs}_{CS_i,1}, \ldots, \mathtt{cs}_{CS_i,k_i}$ which implement the solving capability of CS_i ($1 \le i \le n$).

2. a scheduler, which

(a) allocates/deallocates the constraint solving resources of the system,

(b) realizes the [cs]-steps of the CFLP interpreter by coordinating the constraint solving process of the individual constraint stores in accordance with the strategy $\mathcal{S}$.

The two components—CFLP interpreter and scheduler—communicate asynchronously via `ask/tell` messages sent over a bidirectional link connection denoted by *lnk*. The connection *lnk* is a pair $\langle lnk_1, lnk_2 \rangle$ of message queues, where lnk_1 is used for passing messages from the CFLP interpreter to the scheduler, and lnk_2 is used for passing messages in the other direction. We adopt a producer/consumer communication protocol and make use of the following primitive operations for communication:

- **LinkWrite**(*lnk*, *msg*) : send the message `msg` over lnk_{3-i},

- **LinkRead**(*lnk*) : consume the message which was sent first over lnk_i (if any),

- **LinkReadyQ**(*lnk*) : yields True if lnk_i contains a message that can be consumed, and False otherwise

where $i = 2$ if the operation is performed by the CFLP interpreter, and $i = 1$ if it is performed by the scheduler.

6.4.1 The CFLP Interpreter

The CFLP interpreter acts on the configuration of the CFLP system which is stored in the CFLP pool. The operational semantics of the interpreter is given by the calculus $\mathcal{C}$.

We first give an informal description of the operational semantics of the [cs]-steps of the CFLP interpreter in our distributed model. Upon [cs]-steps on a state $A = \langle \gamma \mid \{\overline{e_n}\} \mid \xi \rangle$, the CFLP interpreter requires the result of computing $CF_{\mathcal{S}(\overline{CS_n})}(\xi)$. To get this result, the CFLP interpreter does the following operations:

1. it sends to the scheduler a query to solve the system of constraints ξ via a message $\texttt{ask}(\xi, m)$ with $m \in \mathbb{N}$. To guarantee a safe communication interpreter/scheduler, we assume that m is a message identifier, i.e. if $\texttt{ask}(\xi_1, m_1)$ and $\texttt{ask}(\xi_2, m_2)$ are distinct messages sent by the interpreter to the scheduler then $m_1 \neq m_2$

2. it suspends the evaluation of the corresponding goal until the distributed constraint subsystem starts returning the result

3. it maintains a table **Answer** in which it stores the messages returned by the scheduler. The entry **Answer**(m) contains the list of messages retrieved so far as replies to the message $\texttt{ask}(\xi, m)$.

In our model, we assume that the set $CF_{\mathcal{S}(\overline{CS_n})}(\xi)$ is computed incrementally and its components are returned to the CFLP interpreter via `tell` messages. The `ask` messages are sent to the distributed constraint solving subsystem via **SendMsg** calls (see Fig. 6.2). If the interpreter has sent a message $\texttt{ask}(\xi, m)$ to the scheduler and $CF_{\mathcal{S}(\overline{CS_n})}(\xi) = \{\langle \theta_k, \xi_k \rangle \mid 1 \leq k \leq N\}$ then the scheduler is supposed to send back the sequence of messages $\{\texttt{tell}(\theta_k, \xi_k, m) \mid 1 \leq k \leq N\}$, followed by a message $\texttt{done}(m)$ to signal that the computation of $CF_{\mathcal{S}(\overline{CS_n})}(\xi)$ has finished.

We describe now the operational semantics of the CFLP interpreter for the distributed model outlined above.

1. we extend the syntax domain of states with expressions of the form $\langle \gamma \mid \{E, \texttt{wait}(\xi, m), E'\} \mid \emptyset \rangle$ and $\langle \gamma \mid \texttt{false} \mid \xi \rangle$.

   A state of the form $\langle \gamma \mid \{E, \texttt{wait}(\xi, m), E'\} \mid \emptyset \rangle$ is called a `wait`-state, and it is the result of applying a [cs]-step to $\langle \gamma \mid \{E\} \mid \xi \rangle$. Its presence in the CFLP pool indicates that the CFLP interpreter has suspended the evaluation of $\langle \gamma \mid \{E_1\} \mid \xi \rangle$ until the distributed constraint subsystem will return (an element of) $CF_{\mathcal{S}(\overline{CS_n})}(\xi)$. The `wait`-states are considered to be reducible states.

```
procedure SendMsg(in ask(ξ, m))
 Inputs          ξ    : set of constraints
                 m    : message identifier
 Globals       lnk    : link connection interpreter/scheduler
            Answer    : table of messages received from the scheduler
Answer(m) ← {};
LinkWrite(lnk, ask(ξ, m));
return True
```

Fig. 6.2: Procedure: **SendMsg**

A state of the form $\langle\gamma \mid \texttt{false} \mid \xi\rangle$ is called a *failure state*, and it denotes a final state which does not contribute to the computation of any solution.

2. we extend the calculus $\mathcal{C}$ with inference rules for `wait`-states. Whenever a state $\langle\gamma \mid \{E, \texttt{wait}(\xi, m), E'\} \mid \emptyset\rangle$ is selected in a [cfg]-step, the CFLP interpreter calls the function **GetMsg**(m) (see Fig. 6.3) in an attempt to retrieve the elements of $CF_{\mathcal{S}(\overline{CS_n})}(\xi)$ computed so far and generates the new configuration accordingly.

```
procedure GetMsg(in m)
 Inputs          ξ    : set of constraints
                 m    : message identifier
 Globals       lnk    : link connection between interpreter and scheduler
            Answer    : table of messages received from the scheduler

while LinkReadyQ(lnk)
      msg ← LinkRead(lnk);
      if msg == tell(θ, ξ, m')
        Answer(m') ← Append(Answer(m'), msg)
      elseif msg == done(m')
        Answer(m') ← Append(Answer(m'), msg)
res ← Answer(m);
Answer(m) ← {};
return res
```

Fig. 6.3: Procedure: **GetMsg**

Inference rules

The inference rule [cs] for our distributed model of CFLP becomes

[cs] *constraint solve*

$$\frac{\langle \gamma \mid \{E\} \mid \xi \rangle}{\langle \gamma \mid \{\mathtt{wait}(\xi, m), E\} \mid \emptyset \rangle}$$

if $\mathbf{SendMsg}(\mathit{lnk}, \mathtt{ask}(\xi, m))$.

In addition, we have the following inference rules for configurations:

$[\mathrm{cfg}]_1$ *inference rules for* `wait`*-configurations*

$$\frac{\langle \gamma \mid \{E, \mathtt{wait}(\xi, m), E'\} \mid \emptyset \rangle, \overline{A_p}}{\overline{A_p}, \overline{\langle \gamma\theta_N \mid \{E_1\theta_N\} \mid \xi_N \rangle}, \langle \gamma \mid \{E, \mathtt{wait}(\xi, m), E'\} \mid \emptyset \rangle} \tag{6.3}$$

if $\{\overline{\mathtt{tell}(\theta_N, \xi_N, m)}\} = \mathbf{GetMsg}(m)$.

$$\frac{\langle \gamma \mid \{E, \mathtt{wait}(\xi, m), E'\} \mid \emptyset \rangle, \overline{A_p}}{\overline{A_p}, \overline{\langle \gamma\theta_N \mid \{E\theta_N, E'\theta_N\} \mid \xi_N \rangle}, \langle \gamma \mid \mathtt{false} \mid \emptyset \rangle} \tag{6.4}$$

if $\{\overline{\mathtt{tell}(\theta_N, \xi_N, m)}, \mathtt{done}(m)\} = \mathbf{GetMsg}(m)$.

$[\mathrm{cfg}]_2$ *inference rules for other configurations*

$$\frac{A, \overline{A_n}}{\overline{A_n}, \overline{A'_k}} \tag{6.5}$$

if A is non-`wait`-state which is $\mathcal{C}$-reducible and $\langle \gamma \mid \{\overline{e_m}\} \mid \xi \rangle \overset{\mathcal{C}}{\Longrightarrow} A'_1$, $\ldots$, $A \overset{\mathcal{C}}{\Longrightarrow} A'_k$ are all the possible $\mathcal{C}$-steps starting with A.

$$\frac{A, \overline{A_n}}{\overline{A_n}, A} \tag{6.6}$$

if A is $\mathcal{C}$-irreducible and there is $j > 1$ such that A_j is $\mathcal{C}$-reducible.

6.4.2 The Scheduler

The scheduler implements a strategy $\mathcal{S}$ of a solver cooperation $\langle \overline{CS_n} \rangle_{\mathcal{X}}$ for solving the constraints received from the CFLP interpreter via `ask` messages. It makes use of a local area of memory called *constraint pool*, in which it maintains a dynamic data structure **CTree**, called *constraint tree*.

CTree is a tree whose nodes are expressions of the form $\langle\gamma,\xi\rangle_m$ where $\gamma \in \mathcal{S}ubst(\mathcal{F},\mathcal{V})$, ξ is a set of constraints, and m is a positive integer index. Whenever an $\mathtt{ask}(\xi,m)$ message is received from the CFLP interpreter, the scheduler adds the node $\langle\varepsilon,\xi\rangle_m$ as son of the root of the constrained tree. The branches of the constraint tree are labeled with solver identifiers which are either from $\{\overline{CS_n}\}$ or from a set Id_{spec} of special solver identifiers (e.g., $Id_{spec} = \{\mathtt{Abs}\}$). Branches are labeled in such a way that if $\langle\gamma_0,\xi_0\rangle_m \xrightarrow{c_1} \langle\gamma_1,\xi_1\rangle_m \xrightarrow{c_2} \dots \xrightarrow{c_p} \langle\gamma_p,\xi_p\rangle_m$ is a path in **CTree** starting from a son $\langle\gamma_0,\xi_0\rangle_m$ of the root, then $\langle\gamma_0,\xi_0\rangle \Rightarrow^{c_1;c_2;\dots;c_p} \langle\gamma_p,\xi_p\rangle$ is a $\mathcal{S}(\overline{CS_n})$-subderivation.

Solving the constraints received from the CFLP interpreter is achieved by expanding the nodes of the constrained tree. The expansion of a node $P = \langle\gamma,\xi\rangle_m$ is realized by applying to ξ the solver whose identifier labels the branches of **CTree** starting from P. If this identifier is c and $c(\xi) = \bigvee_{k=1}^{N}\langle\theta_k,\xi_k\rangle$ then we create the nodes $\langle\gamma\theta_1,\xi_1\rangle_m, \dots, \langle\gamma\theta_N,\xi_N\rangle_m$ as sons of P, and label the newly created branches with c.

The expansion of a node $\langle\gamma,\xi\rangle_m$ stops when ξ is an $\mathcal{S}(\overline{CS_n})$-canonical form. In this case the node is called *final*, and we denote it by $\langle\!\langle\gamma,\xi\rangle\!\rangle_m$.

A c-step with $c = CS_i \in \{\overline{CS_n}\}$ can be performed only when there is available a resource $\mathtt{cs}_{CS_i,j}$. The scheduler is required to implement a fair scheduling algorithm of the resources available, i.e. whenever the content of node is not a $\mathcal{S}(\overline{CS_n})$-canonical form and its expansion requires the application of a CS_i-solver, then a resource from $\{\mathtt{cs}_{CS_i,j} \mid 1 \le j \le k_i\}$ will be eventually allocated for expanding that node.

If $\langle\!\langle\theta,\xi\rangle\!\rangle_m$ is a final node then $\langle\theta,\xi\rangle$ is an answer to the query received from the CFLP interpreter via the message $\mathtt{ask}$ with tag m. To model [cs]-steps in our distributed model we need a means to transmit the content of a final node of **CTree** back to the CFLP interpreter.

The procedure which describes the operations performed by the scheduler on **CTree** is shown in Fig. 6.4. We describe it with help of the following auxiliary functions.

GetSolverId(`in` P) yields the identifier of the solver required to expand node P.

Alloc(`in` ***id***, `in` $\langle\gamma,\xi\rangle_m$, `inout` **AllocTbl**, `inout` **state**) checks whether it is available a resource *cs* for *id*. If yes, then

1. allocate *cs* and update **AllocTbl** accordingly,
2. send ξ to be solved by *cs*,
3. $\mathbf{state}(\langle\gamma,\xi\rangle_m) \leftarrow \langle$"*wait*", *id*, *cs*$\rangle$.

```
procedure Schedule(in 𝒮)
Inputs                 𝒮   : strategy for a cooperation ⟨CS_n‾⟩_𝒳
Globals              lnk   : stream of messages from the CFLP interpreter
Locals             CTree   : constraint tree
    {cs‾_{CS_i,k_i}}_{1≤i≤n}  :solving resources of the cooperation ⟨CS_n‾⟩_𝒳
                   state   : table that associates to every node P of CTree
                             one of the following states
                             "idle": P waits to be expanded,
                             ⟨"wait", id, cs⟩: P is currently being expanded by
                             a resource cs} for the solver id ∈ {CS_n‾,
                AllocTbl   : allocation table for constraint solving resources,
                       M   : list of tags of ask queries to be answered
(Setup)
CTree ← the tree consisting of the node P_0 := ⟨ε, {}⟩_0;
state(P_0) ← "done"; M ← [ ];
while True do
  if LinkReadyQ(lnk)
    ask(γ', ξ', m')←LinkRead(lnk);
    Insert ⟨γ', ξ'⟩_{m'} in CTree as son of P_0;
    M ← Append(M, m'); M' ← [ ];
  if M ≠ [ ]
    m ← First(M); M ← Rest(m); IsDone ← True
    for each node P(= ⟨γ, ξ⟩_m) of CTree with state(P) ≠ "done" do
        IsDone ← False;
        if P == ⟨⟨γ, ξ⟩⟩_m
          LinkWrite(lnk, tell(γ, ξ, m));
          state(P) ← "done"
        elseif state(P) == "idle"
          id ← GetSolverId(P);
          if id ∈ Id_sp
            PTree ← SpecialExpand(P, id);
            CTree ← CTree[P ← PTree]
          else
            Alloc(id, P, AllocTbl, state);
        elseif state(P) == ⟨"wait", id, cs⟩
          {⟨θ_N, ξ_N⟩‾} ← GetAnswers(cs);
          Add the nodes {P_i | P_i = ⟨γθ_i, ξ_i⟩_m, 1 ≤ i ≤ N} as sons of P
          and label the corresponding branches with id;
          P_1 ← "idle"; ... ; P_N ← "idle";
          if IsOver(cs)
            Dealloc(cs, AllocTbl); state(P) ← "done"
    if IsDone LinkWrite(lnk, done(m))
    else M ← Append(M, m)
```

Fig. 6.4: Procedure: **Schedule**

Dealloc(`in` $\boldsymbol{cs}$, `inout` **AllocTbl**) deallocates the resource cs and updates the table **AllocTbl** accordingly.

SpecialExpand(`in` $\langle\gamma,\xi\rangle_m$, `in` $\boldsymbol{cs}$, `inout` **state**) performs the following operations:

1. compute $cs(\xi) = \bigvee_{k=1}^{N} \langle\theta_k, \xi_k\rangle$,
2. construct the tree T with root $P := \langle\gamma,\xi\rangle_m$, sons $\overline{P_N := \langle\gamma\theta_N, \xi_N\rangle_m}$ and branches labeled with cs,
3. $\mathbf{state}(P_1) \leftarrow$ "*idle*", ... ,$\mathbf{state}(P_N) \leftarrow$ "*idle*", $\mathbf{state}(P) \leftarrow$ "*done*",
4. return T

GetAnswers(`in` $\boldsymbol{cs}$**)** returns the list of answers computed so far by cs

IsOver($\boldsymbol{cs}$**)** yields True if cs has finished the computation and False otherwise.

Note that the scheduler generates an OR-tree whose expansion is realized with strategy $\mathcal{S}$. Upon generation, if a node $P := \langle\gamma,\xi\rangle_m \in$ **CTree** is not final then:

- if $\mathbf{state}(P)$ = "*done*" then P has N sons $\langle\gamma_1,\xi_1\rangle_m, \ldots, \langle\gamma_N,\xi_N\rangle_m$ such that $[\gamma]\wedge\xi \equiv \bigvee_{k=1}^{N}([\gamma_k]\wedge\xi_k)$. This means that P is an OR-node of **CTree**.
- if $\mathbf{state}(P) = \langle$"*wait*", *id*, *cs*$\rangle$ and P has N sons $\langle\gamma_1,\xi_1\rangle_m, \ldots, \langle\gamma_N,\xi_N\rangle_m$ then we only know for sure that $[\gamma]\wedge\xi \leq \bigvee_{k=1}^{N}([\gamma_k]\wedge\xi_k)$. P becomes an OR-node when its state changes to "*done*".

Constraint solving resources

The constraint solving resources are implementations of the solvers of a cooperation $\langle\overline{CS_n}\rangle_{\mathcal{X}}$. They are in charge of solving the systems of constraints received from the scheduler.

It may be the case that the computation performed by a particular constraint solver may be complex and time consuming. In our distributed model, we assume that a constraint solver cs computes the elements

$$\{\langle\theta_k,\xi_k\rangle \mid 1 \leq k \leq N\}$$

of a disjunction $cs(\xi)$ one after the other, and makes them accessible to the scheduler (via **GetAnswer** calls) as soon as possible.

Chapter 7

The CFLP System

In this chapter we describe the design and implementation of an experimental system called `CFLP` [MS98, MIS99b, MIS99a] which is based on an instance of the scheme CFLP$(\mathcal{X}, \mathcal{S}, \mathcal{C})$ outlined in the previous chapter. The system is implemented completely in *Mathematica*™ [Wol96] and uses the *MathLink* protocol [Miy99, Wol96] for interprocess communication. *Mathematica*™ is a computer algebra system whose kernel is based on a particular version of higher-order term rewriting, and therefore it is very suitable for doing symbolic computations with complex structures in mathematics and programming. In addition, *Mathematica*™ is delivered with powerful packages for solving symbolic constraints, such as methods for solving differential and partial differential equations, Simplex algorithm for systems of linear equations, Gröbner basis methods for systems of polynomial equations, etc.

The characteristic features of our experimental system are the following:

- $\mathcal{X}$ is a constraint domain $\langle \Sigma, \mathcal{A}, \mathcal{V}, \Phi \rangle$ whose computation domain Φ allows to express systems of linear and polynomial equations, differential equations, and partial differential equations. It is based on a solver cooperation $\langle \overline{CS_4} \rangle_{\mathcal{X}}$ where

  1. CS_1 is a solver which can solve algebraic equations which involve invertible functions, and represents the solutions in terms of formal inverse functions.
  2. CS_2 can solve systems of equations between multivariate polynomials over the domain of complex numbers,
  3. CS_3 is a solver for systems of differential equations over algebraic extensions of $\mathbb{C}$,

4. CS_4 is a solver for partial differential equations over algebraic extensions of $\mathbb{C}$,

- the strategy $\mathcal{S}$ for solver cooperation is fp(Abs); fp(CS_1; ... ; CS_n),
- The calculus $\mathcal{C}$ of the functional logic component of CFLP can be selected by the user from a collection of constrained lazy narrowing calculi which are extensions of first- and higher-order lazy narrowing calculi investigated by us
- The language of terms of the system is is an extension with λ-abstractions of the language of terms in *Mathematica*™, and thus it closely resembles the standard mathematical representation.
- The user can adjust the distributed constraint solving subsystem of CFLP by specifying the constraint solving resources used upon the computations.

The rest of this chapter is structured as follows. In Sect. 7.1 we define the language of CFLP. In Sect. 7.2 we describe the operational semantics of the CFLP interpreter. The distributed constraint solving subsystem of CFLP is described in Sect. 7.3. Finally, in Sect. 7.4 we describe the user interface of the system.

7.1 The Language

In this section we describe the main syntax constructs of CFLP.

The language of terms of CFLP consists of simply-typed λ-terms in long $\beta\eta$-normal form, built over a simply typed signature $\Sigma = \langle S_0, \mathcal{F}\rangle$ with $\mathcal{F} = \mathcal{F}_c \cup \mathcal{F}_d \cup \mathcal{F}_e\rangle$. We adopt the usual *Mathematica* notation for terms, extended with the construct $\lambda[\{x_1, \ldots, x_n\}, s]$ for the λ-term $\lambda\overline{x_n}.s$.

An *equation* is an element of the set $\mathcal{E}q(\mathcal{F}, \mathcal{V})$ of expressions of the form

$$s \approx t \mid s \rhd t \mid s \doteq t \mid s \gg t$$

where $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ are simply typed λ-terms of the same type. The semantical distinction between the equational constructs is as follows:

- $s \approx t$ is an unoriented equation. A solution of $s \approx t$ is a substitution θ such that $s\theta \downarrow_{\mathcal{R}} s\theta$,
- $s \rhd t$ is an oriented equation. A solution of $s \rhd t$ is a substitution θ such that $s\theta \to^*_{\mathcal{R}} t\theta$,

- $s \doteq t$ is an unoriented strict equation. A solution of $s \doteq t$ is a substitution θ such that $s\theta \rightarrow^*_{\mathcal{R}} u$ and $t\theta \rightarrow^*_{\mathcal{R}} u$ for some term $u \in \mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$,
- $s \gg t$ is an oriented strict equation. A solution of $s \gg t$ is a substitution θ such that $s\theta \rightarrow^*_{\mathcal{R}} t\theta$ and $t\theta \in \mathcal{T}(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$.

A *goal* in `CFLP` is an expression of the syntax domain

$$e \mid \{G_1, \ldots, G_n\} \mid G_1 \vee \ldots \vee G_n \mid G_1 \| \ldots \| G_n \mid \texttt{true} \mid \texttt{false} \mid \circledast$$

where $e \in \mathcal{E}q(\mathcal{F}, \mathcal{V})$ and $G_1, \ldots, G_n$ are goals.

- $\{G_1, \ldots, G_n\}$ denotes a *sequential-AND goal.* The subgoals $G_1, \ldots,$ G_n are solved sequentially, in an order established by a particular goal selection strategy $\mathcal{S}_{goal}$ which is specific to the calculus $\mathcal{C}$ of the `CFLP` interpreter,
- $G_1 \| \ldots \| G_n$ denotes a *sequential-OR goal*: the subgoals $G_1, \ldots, G_n$ are solved one after the other, starting with G_1 and ending with G_n,
- $G_1 \vee \ldots \vee G_n$ denotes a *parallel-OR goal*: the subgoals $G_1, \ldots, G_n$ are solved with a fair strategy, i.e. a strategy that guarantees that every subgoal will be eventually solved,
- `true` denotes the goal whose set of solutions is ∇,
- `false` denotes the goal with no solutions, i.e. $[\![\texttt{false}]\!]^{\mathcal{A}} = \emptyset$,
- the goal $\circledast$ has the same denotational semantics as `true`, but a different operational semantics: when the interpreter acts on $\circledast$, it simply discards it and afterwards it performs a [cs]-step. $\circledast$ provides the user with a means to control the cooperation between interpreter and distributed constraint subsystem.

A `CFLP` *program* is a set of rewrite rules of the form $f(\overline{l_n}) \rightarrow r \Leftarrow G$ where $f \in \mathcal{F}_d$, $f(\overline{l_n})$, r are CFLP terms of the same base type, and G is a `CFLP` goal.

7.2 The Interpreter

The `CFLP` interpreter implements the calculus $\mathcal{C}$ of the scheme. The `CFLP` system provides the user with the liberty to choose a suitable calculus from a set of built-in ones. The right choice depends on certain properties of the program that describes the problem which we want to solve.

The current implementation of `CFLP` has built-in the following constrained lazy narrowing calculi:

LNCP : the default constrained lazy narrowing calculus used upon a `CFLP` session, obtained by specializing the calculus $\mathrm{CLN}_{\mathrm{ff}}$. The calculus is in general incomplete because it does not perform outermost narrowing at variable position,

$\mathbf{LN}_4$: an implementation of the calculus LN_4 with strategy $\mathcal{S}_n$ outlined in Sect. 5.9,

LCNC, $\mathbf{LNC}_d$: implementations of the constrained versions of the first-order lazy narrowing calculi LCNC, LNC, LNC_d described in Sect. 3.2.

7.2.1 Notions and Notation

In this subsection we introduce some preliminary definitions that are used in describing of the operational semantics of the `CFLP` interpreter by a set of inference rules.

In the design of the built in calculi of the interpreter we took into account a generalization of the notion of precursor introduced in Def. 43, pp. 98. Because `CFLP` goals have a recursive structure, we introduce the concept of precursor of a `CFLP` goal, and define it as a `CFLP` goal. Initially, all the subgoals of the initial goal have no precursor. We keep track of the precursors of equations at runtime by introducing a new construct:

$$\mathrm{prec}(G_1, G_2)$$

where G_1, G_2 are goals. This construct stands for the sequential-AND goal $\{G_1, G_2\}$ such that G_1 is the precursor of G_2.

A *runtime goal* is an expression of the syntax domain

$$\begin{array}{ll} e & \mid \mathrm{prec}(G_1, G_2) \mid \{G_1, \ldots, G_N\} \mid G_1 \vee \ldots \vee G_N \mid G_1 \| \ldots \| G_N \\ & \mid \mathrm{wait}(\xi, m) \mid \mathrm{tell}(\gamma, G, \xi) \mid \mathtt{true} \mid \mathtt{false} \mid \circledast \end{array}$$

where $e \in \mathcal{E}q(\mathcal{F}, \mathcal{V})$, $\xi \in 2^{\mathcal{E}q(\mathcal{F}_e \cup \mathcal{F}_c, \mathcal{V})}$, $m \in \mathbb{N}$ and $G, G_1, \ldots, G_N$ are goals in `CFLP`.

A *state* in `CFLP` is an expression of the form $\langle \gamma \mid G \mid \xi \rangle$ where $\gamma \in \mathcal{S}ubst(\mathcal{F}, \mathcal{V})$, G is a runtime goal, and $\xi \in 2^{\mathcal{E}q(\mathcal{F},\mathcal{V})}$. An *equational state* in `CFLP` is a state of the form $\langle \gamma \mid e \mid \xi \rangle$ with $e \in \mathcal{E}q(\mathcal{F}, \mathcal{V})$. An *elementary state* in `CFLP` is a state of the form $\langle \gamma \mid e \mid \xi \rangle$ with $e \in \mathcal{E}q(\mathcal{F}, \mathcal{V}) \cup \{\circledast\}$.

An *configuration* in `CFLP` is either

- a state, or
- a logical disjunction of configurations. There are two constructs for the logical disjunction of the configurations $A_1, \ldots, A_N$:

– $A_1 \| \dots \| A_N$, and
– $A_1 \vee \dots \vee A_N$, abbreviated $\bigvee_{k=1}^{N} A_k$.

The first construct denotes a *sequential-OR disjunction* of the configurations $A_1, \dots, A_N$, whereas the second one denotes a *parallel-OR disjunction* of the configurations $A_1, \dots, A_N$.

The configurations of a sequential-OR disjunction are reduced to final configurations sequentially, from left to right. The configurations of a parallel-OR disjunction are reduced simultaneously to final configurations, by means of a round-robin reduction strategy.

7.2.2 Constrained Lazy Narrowing Calculi

We define a constrained lazy narrowing calculus $\mathcal{C}$ by a system of inference rules that can be defined in a bottom-up way as follows:

1. inference rules for elementary states,
2. inference rules for non-elementary states,
3. inference rules for configurations,
4. simplification rules for configurations,
5. simplification rules for goals.

The built-in calculi of `CFLP` differ only at the level of inference rules for elementary states and by the goal selection functions associated with them.

In the rest of this subsection we give the inference rules which are common to all constrained lazy narrowing calculi.

Inference rules for non-elementary states

[AND] *sequential-AND*

$$\frac{\langle \gamma \mid \{\overline{G_p}, G, \overline{G'_q}\} \mid \xi \rangle}{\langle \gamma\theta \mid \{\overline{G_p\theta}, G', \overline{G'_q\theta}\} \mid \xi' \rangle}$$

if $G = sel_{goal}(\{\overline{G_p}, G, \overline{G'_q}\})$ and $\langle \varepsilon \mid G \mid \xi \rangle \Rightarrow_\alpha \langle \theta \mid G' \mid \xi' \rangle$ is an α-step with $\alpha \in \{$[AND], $[\text{prec}]_1$, $[\text{prec}]_2$, [OTH]$\}$,

$[\text{prec}]_1$ *precursor goal*

$$\frac{\langle \gamma \mid \text{prec}(G_1, G_2) \mid \xi \rangle}{\langle \gamma\theta \mid \text{prec}(G'_1, G_2\theta) \mid \xi' \rangle}$$

if $sel_{goal}(\text{prec}(G_1, G_2)) = G_1$ and $\langle \varepsilon \mid G_1 \mid \xi \rangle \Rightarrow_\alpha \langle \theta \mid G'_1 \mid \xi' \rangle$ is an α-step with $\alpha \in \{$[AND], $[\text{prec}]_1$, $[\text{prec}]_2$, [OTH]$\}$,

[prec]$_2$ *goal with precursor*

$$\frac{\langle \gamma \mid \mathrm{prec}(G_1, G_2) \mid \xi \rangle}{\langle \gamma\theta \mid \mathrm{prec}(G_1\theta, G_2') \mid \xi' \rangle}$$

if $sel_{goal}(\mathrm{prec}(G_1, G_2)) = G_2$ and $\langle \varepsilon \mid G_2 \mid \xi \rangle \Rightarrow_\alpha \langle \theta \mid G_2' \mid \xi' \rangle$ is an α-step with $\alpha \in \{$[AND], [prec]$_1$, [prec]$_2$, [OTH]$\}$.

[OTH] *otherwise*

$$\frac{\langle \gamma \mid G \mid \xi \rangle}{G'}$$

if $\langle \gamma \mid G \mid \xi \rangle$ is an elementary state and $\langle \gamma \mid G \mid \xi \rangle \Rightarrow_\alpha G'$ is an α-step of $\mathcal{C}$ for elementary states which satisfies its restrictions.

Inference rules for configurations

[$\vee$] *parallel-OR*

$$\frac{\bigvee_{p=1}^{M} A_p \vee A \vee \bigvee_{q=1}^{N} A_q'}{\bigvee_{q=1}^{M} A_q' \vee \bigvee_{p=1}^{M} A_p \vee A'}$$

if $A_1, \ldots, A_p$ are final configurations, A is not a final configuration and $A \Rightarrow_\alpha A'$ is an α-step with $\alpha \in \{[\vee], [\|], [\mathrm{G}]\}$,

[$\|$] *sequential-OR*

$$\frac{A_1 \| \ldots \| A_M \| A \| A_1' \| \ldots \| A_N'}{A' \| A_1' \| \ldots \| A_N' \| A_1 \| \ldots \| A_N}$$

if $A_1, \ldots, A_p$ are final configurations, A is not a final configuration and $A \Rightarrow_\alpha A'$ is an α-step with $\alpha \in \{[\vee], [\|], [\mathrm{G}]\}$,

[G] *state*

$$\frac{\langle \gamma \mid G \mid \xi \rangle}{\bigvee_{k=1}^{N} A_k}$$

if $\langle \gamma \mid G \mid \xi \rangle$ is not a final state, $\langle \gamma \mid G \mid \xi \rangle \Rightarrow_\alpha A_1, \ldots, \langle \gamma \mid G \mid \xi \rangle \Rightarrow_\alpha A_N$ are all the nondeterministic α-steps of $\mathcal{C}$ with $\alpha \in \{$[AND], [prec]$_1$, [prec]$_2$, [OTH]$\}$.

Simplification rules for configurations

$$\begin{array}{lcl}
\bigvee_{k=1}^{1} A_k & = & A_1 \\
\langle \gamma \mid G_1 \vee \ldots \vee G_m \mid \xi \rangle & = & \bigvee_{k=1}^{m} \langle \gamma \mid G_k \mid \xi \rangle \\
\langle \gamma \mid \mathrm{tell}(\theta, G, \xi) \mid \emptyset \rangle & = & \langle \gamma\theta \mid G \mid \xi \rangle \\
\langle \gamma \mid G_1 \| \ldots \| G_m \mid \xi \rangle & = & \langle \gamma \mid G_1 \mid \xi \rangle \| \ldots \| \langle \gamma \mid G_m \mid \xi \rangle \\
\langle \gamma \mid \mathtt{false} \mid \xi \rangle \vee \bigvee_{k=1}^{n} A_k & = & \bigvee_{k=1}^{n} A_k
\end{array}$$

It is easy to see that the simplification rules for configurations define a canonical simplifier. We assume that the configurations are implicitly reduced to the canonical form defined by simplification rules for configurations.

Simplification rules for goals

$$
\begin{array}{lcl}
\{G\} & = & G \\
\{\overline{G_p}, \texttt{true}, \overline{G'_q}\} & = & \{\overline{G_p}, \overline{G'_q}\} \\
\{\overline{G_p}, \texttt{false}, \overline{G'_q}\} & = & \texttt{false} \\
\{\overline{G_p}, \{\overline{G'_q}\}, \overline{G''_r}\} & = & \{\overline{G_p}, \overline{G'_q}, \overline{G''_r}\} \\
\{G_1 \vee \ldots \vee G_N, \overline{G'_m}\} & = & \bigvee_{k=1}^{N} \{G_k, \overline{G'_m}\} \\
\{G_1 \| \ldots \| G_N, \overline{G'_m}\} & = & \{G_1, \overline{G'_m}\} \| \ldots \| \{G_N, \overline{G'_m}\} \\
\{G_1, \mathrm{ask}(\theta, G, \xi), G_2\} & = & \mathrm{ask}(\{G_1\theta, G, G_2\theta\}, \xi)
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{prec}(\texttt{true}, G) & = & G \\
\mathrm{prec}(\texttt{false}, G) & = & \texttt{false} \\
\mathrm{prec}(G, \texttt{false}) & = & \texttt{false} \\
\mathrm{prec}(G_1 \vee \ldots \vee G_N, G) & = & \bigvee_{k=1}^{N} \mathrm{prec}(G_k, G) \\
\mathrm{prec}(G, G_1 \vee \ldots \vee G_N) & = & \bigvee_{k=1}^{N} \mathrm{prec}(G, G_k) \\
\mathrm{prec}(G_1 \| \ldots \| G_N, G) & = & \mathrm{prec}(G_1, G) \| \ldots \| \mathrm{prec}(G_N, G) \\
\mathrm{prec}(G, G_1 \| \ldots \| G_N) & = & \mathrm{prec}(G, G_1) \| \ldots \| \mathrm{prec}(G, G_N)
\end{array}
$$

The siplification rules for goals define a goal canonical simplifier. We assume that the reduction of a goal to its canonical form is an implicit operation.

Remarks

The simplification rules for `CFLP` goals and configurations are designed in such a way that the simplified configurations consist of elementary states of the form $\langle \gamma \mid G \mid \xi \rangle$ with G a runtime goal which is free of sequential-OR and parallel-OR operators. Moreover, G does not contain occurrences of subgoals of the form $\mathrm{tell}(\theta, G', \xi')$: they always vanish during simplification.

7.2.3 The Calculus LNCP

LNCP is the default calculus of the interpreter. In order to characterize it, we state its inference rules for elementary states, the restrictions which limit the nondeterminism between its inference rules for equational states, and the goal selection function.

Inference rules for elementary states

[f] *failure detection*

$$\frac{\langle \gamma \mid \lambda\overline{x}.s \gg \lambda\overline{x}.f(\overline{s_n}) \mid \xi \rangle}{\langle \gamma \mid \texttt{false} \mid \xi \rangle}$$

if $f \in \mathcal{F}_d$,

$$\frac{\langle \gamma \mid \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.g(\overline{t_n}) \mid \xi \rangle}{\langle \gamma \mid \texttt{false} \mid \xi \rangle}$$

if $f, g \in \mathcal{F}_c$, $f \neq g$, and $\cong \in \{\approx, \rhd, \doteq, \gg\}$,

$$\frac{\langle \gamma \mid e \mid \xi \rangle}{\langle \gamma \mid \texttt{false} \mid \xi \rangle}$$

if $\xi \notin 2^{\mathcal{E}q(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})}$ (that is, ξ contains an equation $e' \notin \mathcal{E}q(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$) and $e \in \mathcal{E}q(\mathcal{F}, \mathcal{V})$

[d] *decomposition*

$$\frac{\langle \gamma \mid \lambda\overline{x}.g(\overline{s_n}) \cong \lambda\overline{x}.g(\overline{t_n}) \mid \xi \rangle}{\langle \gamma \mid \{\overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n}\} \mid \xi \rangle}$$

if $g \in \mathcal{F}_c$ and $\cong \in \{\approx, \rhd, \doteq, \gg\}$,

$$\frac{\langle \gamma \mid \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.f(\overline{t_n}) \mid \xi \rangle}{\langle \gamma \mid \{\overline{\lambda\overline{x}.s_n \cong \lambda\overline{x}.t_n}\} \mid \xi \rangle}$$

if $f \in \mathcal{F}_d$ and $\cong \in \{\approx, \rhd\}$,

[del] *deletion*

$$\frac{\langle \gamma \mid \lambda\overline{x}.t \approx \lambda\overline{x}.t \mid \xi \rangle}{\langle \gamma \mid \texttt{true} \mid \xi \rangle} \qquad \frac{\langle \gamma \mid \lambda\overline{x}.t \rhd \lambda\overline{x}.t \mid \xi \rangle}{\langle \gamma \mid \texttt{true} \mid \xi \rangle}$$

[on] *constrained narrowing at nonvariable position*

$$\frac{\langle \gamma \mid \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.t \mid \xi \rangle}{\langle \gamma \mid \mathrm{prec}(\mathrm{prec}(\{\overline{\lambda\overline{x}.s_n \rhd \lambda\overline{x}.l_n}\}, G), \lambda\overline{x}.r \cong \lambda\overline{x}.t) \mid \xi \rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \doteq, \doteq^{-1}, \rhd, \gg\}$ and $f(\overline{l_n}) \to r \Leftarrow G$ is a fresh $\overline{x}$-lifted variant of a rule in $\mathcal{R}$,

[ov] *constrainted narrowing at variable position*

$$\frac{\langle \gamma \mid \lambda\overline{x}.X(\overline{s_m}) \cong \lambda\overline{x}.t \mid \xi \rangle}{\langle \gamma\theta \mid \mathrm{prec}(\mathrm{prec}(\{\overline{\lambda\overline{x}.H_n(\overline{s_m\theta}) \rhd \lambda\overline{x}.l_n}\}, G), \lambda\overline{x}.r \cong \lambda\overline{x}.t) \mid \xi\theta \rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \doteq, \rhd, \gg\}$, $\lambda\overline{x}.t$ is a rigid term, $f(\overline{l_n}) \to r \Leftarrow G$ is a fresh $\overline{x}$-lifted variant of a rule in $\mathcal{R}$ and $\theta = \{X \mapsto \lambda\overline{x_n}.f(\overline{H_m(\overline{x_n})})\}$ with $\overline{H_m}$ fresh variables of appropriate types,

[i] *imitation*

$$\frac{\langle \gamma \mid \lambda\overline{x}.f(\overline{s_n}) \cong \lambda\overline{x}.X(\overline{y}) \mid \xi \rangle}{\langle \gamma\theta \mid \overline{\lambda\overline{x}.s_n\theta \cong \lambda\overline{x}.X_n} \mid \xi\theta \rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \rhd, \rhd^{-1}, \gg, \gg^{-1}, \doteq, \doteq^{-1}\}$, $\theta = \{X \mapsto \lambda\overline{y}.f(\overline{X_n(\overline{y})})\}$ where $f \in \mathcal{F}_c \cup \mathcal{F}_d$ and $\overline{X_n}$ are fresh variables of appropriate types.

[p] *projection*

$$\frac{\langle \gamma \mid \lambda\overline{x}.X(\overline{s_n}) \cong \lambda\overline{x}.t \mid \xi \rangle}{\langle \gamma\theta \mid \texttt{true} \mid \xi\theta \rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \doteq, \doteq^{-1}, \rhd, \rhd^{-1}, \gg, \gg^{-1}\}$, $1 \leq i \leq n$, $\lambda\overline{x}.t$ is rigid, $\theta = \{X \mapsto \lambda\overline{y_n}.y_i(\overline{H_p(\overline{y_n})})\}$, $y_i : \overline{\tau_p} \to \tau$, and $\overline{H_p : \tau_p}$ are fresh variables.

[va] *variable abstraction*

$$\frac{\langle \gamma \mid \lambda\overline{x}.s \cong \lambda\overline{x}.t \mid \xi \rangle}{\langle \gamma \mid \lambda\overline{x}.s[X(\overline{y})]_q \cong \lambda\overline{x}.t, \lambda\overline{y}.s|_p \cong \lambda\overline{y}.X(\overline{y}) \mid \xi \rangle}$$

if $\cong \in \{\approx, \approx^{-1}, \rhd, \rhd^{-1}, \gg, \gg^{-1}, \doteq, \doteq^{-1}\}$, $\mathrm{root}(\lambda\overline{x}.s) \in \mathcal{F}_e$, $\{\overline{y}\} = \mathcal{BV}(\lambda\overline{x}.s, p)$ and $(\lambda\overline{x}.s)|_p \in \mathit{Alien}(\lambda\overline{x}.s)$

[ff] *flex/flex equations*

$$\frac{\langle \gamma \mid \lambda\overline{x}.X(\overline{y_n}) \cong \lambda\overline{x}.X(\overline{y'_n}) \mid \xi \rangle}{\langle \gamma\theta \mid \texttt{true} \mid \xi\theta \rangle}$$

where $\cong \in \{\approx, \rhd, \gg, \doteq\}$, $\theta = \{X \mapsto \lambda\overline{y_n}.Z(\overline{z})\}$ with $\{\overline{z}\} = \{y_i \mid y_i = y'_i\}$ and Z is a fresh variable of appropriate type.

$$\frac{\langle \gamma \mid \lambda\overline{x}.X(\overline{y_n}) \cong \lambda\overline{x}.Y(\overline{y'_m}) \mid \xi \rangle}{\langle \gamma\theta \mid \texttt{true} \mid \xi\theta \rangle}$$

where $\cong \in \{\approx, \rhd, \gg, \doteq\}$, $\theta = \{X \mapsto \lambda\overline{y_n}.Z(\overline{z}), Y \mapsto \lambda\overline{y'_m}.Z(\overline{z})\}$ with $\{\overline{z}\} = \{\overline{y_n}\} \cap \{\overline{y'_m}\}$ and Z is a fresh variable of appropriate type.

[cp+] *constraint propagation*

$$\frac{\langle \gamma \mid e \mid \xi \rangle}{\langle \gamma \mid \texttt{true} \mid \xi \cup \{e\} \rangle}$$

if $e \in \mathcal{E}q(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$ is a constraint.

[cs] *constraint solve*

$$\frac{\langle \gamma \mid \circledast \mid \xi \rangle}{\langle \gamma \mid \mathrm{wait}(\xi, m) \mid \emptyset \rangle}$$

if $\xi \in 2^{\mathcal{E}q(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})}$ and $\mathbf{SendMsg}(\mathit{lnk}, \mathtt{ask}(\xi, m))$

$$\frac{\langle \gamma \mid \Box \mid \xi \rangle}{\langle \gamma \mid \mathrm{wait}(\xi, m) \mid \emptyset \rangle} \quad \frac{\langle \gamma \mid \mathtt{true} \mid \xi \rangle}{\langle \gamma \mid \mathrm{wait}(\xi, m) \mid \emptyset \rangle}$$

if $\xi \in 2^{\mathcal{E}q(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})}$, $\bigwedge_{e \in \xi} e$ is not a $\mathcal{S}(\overline{CS}_n)$-canonical form and **Send-Msg**$(\mathit{lnk}, \mathtt{ask}(\xi, m))$,

[wait] *wait*

$$\frac{\langle \gamma \mid \mathrm{wait}(\xi, m) \mid \emptyset \rangle}{\langle \gamma \mid \bigvee_{k=1}^{N} \mathrm{tell}(\theta_k, \mathtt{true}, \xi_k) \vee \mathrm{wait}(\xi, m) \mid \emptyset \rangle}$$

if $\{\overline{\mathtt{tell}(\theta_N, \xi_N, m)}\} = \mathbf{GetMsg}(m)$

$$\frac{\langle \gamma \mid \mathrm{wait}(\xi, m) \mid \emptyset \rangle}{\langle \gamma \mid \bigvee_{k=1}^{N} \mathrm{tell}(\theta_k, \mathtt{true}, \xi_k) \vee \mathtt{false} \mid \emptyset \rangle}$$

if $\{\overline{\mathtt{tell}(\theta_N, \xi_N, m)}, \mathtt{done}(m)\} = \mathbf{GetMsg}(m)$.

In CFLP we eliminate the nondeterminism between [cp+] and the other inference rules of LNCP applicable to a selected equation $s \cong t \in \mathcal{E}q(\mathcal{F}_c \cup \mathcal{F}_e, \mathcal{V})$ by defining a suitable criterion to decide whether $s \cong t$ is a constraint or not. Our constraint decision procedure does the following tests:

- If $s \cong t$ is a rigid/rigid equation then it is not a constraint,
- Otherwise $s \cong t$ is a constraint only if $(\mathrm{root}(s) \cup \mathrm{root}(t)) \cap \mathcal{V}$ is a *constraint* variable, i.e. a variable whose range of values is determined by solving a system of constraints. In CFLP we keep track of the constraint variables in the runtime goal as follows:
  - we provide a means to declare the constraint variables in the initial runtime goal (see Sect. 7.4),
  - the variable abstractions introduced during [va]-steps are constraint variables,
  - if X is a constraint variable and X is bound to a term t then $\mathcal{V}(t)$ are constraint variables.

Note that there is no nondeterminism between the inference rules of `CFLP` for configurations. Also, there is no nondeterminism between the inference rules of `CFLP` for non-elementary states. The nondeterminism of LNCP is due to the various alternatives to realize an [OTH]-step. These alternatives are determined by the syntactic structure of the runtime goal of the elementary state and by the restrictions of the calculus LNCP.

Restrictions

The restrictions of the calculus LNCP specify the inference rules that are considered for reducing an equational state and the priorities of applying them. We describe the restrictions of LNCP in tabular form. (See Figs. 7.1, 7.2, 7.3, 7.4.)

$_{\mathrm{root}(s)}\backslash^{\mathrm{root}(t)}$	$\mathcal{V}(t)$	$\mathcal{F}_d$	$\mathcal{F}_c \cup \mathcal{BV}(t)$	$\mathcal{F}_e$
$\mathcal{V}(s)$	$[\mathrm{f}]^1/[\mathrm{ff}]^2/[\mathrm{cp}]^3$	$[\mathrm{f}]^1/[\mathrm{on}]^2_2$	$[\mathrm{f}]^1/([\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$
$\mathcal{F}_d$	$[\mathrm{f}]^1/[\mathrm{on}]^2_1$	$[\mathrm{f}]^1/[\mathrm{on}]^2_1$	$[\mathrm{f}]^1/[\mathrm{on}]^2_1$	$[\mathrm{f}]^1/[\mathrm{on}]^2_1$
$\mathcal{F}_c \cup \mathcal{BV}(s)$	$[\mathrm{f}]^1/([\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/[\mathrm{on}]^2_2$	$[\mathrm{f}]^1/[\mathrm{d}]^2$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$
$\mathcal{F}_e$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$	$[\mathrm{f}]^1/[\mathrm{on}]^2_2$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$

Fig. 7.1: LNCP: inference rules for equational state $\langle \gamma \mid \lambda\overline{x}.s \doteq \lambda\overline{x}.t \mid \xi \rangle$

$_{\mathrm{root}(s)}\backslash^{\mathrm{root}(t)}$	$\mathcal{V}(t)$	$\mathcal{F}_d$	$\mathcal{F}_c \cup \mathcal{BV}(t)$	$\mathcal{F}_e$
$\mathcal{V}(s)$	$[\mathrm{f}]^1/[\mathrm{ff}]^2/[\mathrm{cp}]^3$	[f]	$[\mathrm{f}]^1/([\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$
$\mathcal{F}_d$	$[\mathrm{f}]^1/[\mathrm{on}]^2$	[f]	$[\mathrm{f}]^1/[\mathrm{on}]^2$	$[\mathrm{f}]^1/[\mathrm{on}]^2$
$\mathcal{F}_c \cup \mathcal{BV}(s)$	$[\mathrm{f}]^1/([\mathrm{i}]^2, [\mathrm{p}]^2)$	[f]	$[\mathrm{f}]^1/[\mathrm{d}]^2$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$
$\mathcal{F}_e$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$	[f]	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$

Fig. 7.2: LNCP: inference rules for equational state $\langle \gamma \mid \lambda\overline{x}.s \gg \lambda\overline{x}.t \mid \xi \rangle$

The superscripts of the labels used in the table indicate the priorities of applying the corresponding inference rules.

The inference rule [f] is specified with highest priority in all the entries of the tables for equational states. This means that whenever [f] can be applied, only [f] is applied. Also, no outermost narrowing at variable position is performed for solving oriented strict equations. This decision may affect the completeness of the computation, but it drastically reduces the search space for solutions.

Note that the right place to impose the priorities depicted in Figs. 7.1, 7.2, 7.3, 7.4 is in the precondition of the inference rule [OTH].

root(s) \ root(t)	$\mathcal{V}(t)$	$\mathcal{F}_d$	$\mathcal{F}_c \cup \mathcal{BV}(t)$	$\mathcal{F}_e$
$\mathcal{V}(s)$	$[\mathrm{f}]^1/[\mathrm{del}]^2/[\mathrm{ff}]^3/[\mathrm{cp}]^4$	$[\mathrm{f}]^1/([\mathrm{on}]^2, [\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/([\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$
$\mathcal{F}_d$	$[\mathrm{f}]^1/([\mathrm{on}]^2, [\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/[\mathrm{del}]/([\mathrm{on}]^3_1, [\mathrm{on}]^3_2, [\mathrm{d}]^3)$	$[\mathrm{f}]^1/[\mathrm{on}]^2$	$[\mathrm{f}]^1/[\mathrm{on}]^2$
$\mathcal{F}_c \cup \mathcal{BV}(s)$	$[\mathrm{f}]^1/([\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/[\mathrm{on}]^2$	$[\mathrm{f}]^1/[\mathrm{d}]^2$	$[\mathrm{f}]^1//([\mathrm{cp}]^2, [\mathrm{va}]^2)$
$\mathcal{F}_e$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$	$[\mathrm{f}]^1/[\mathrm{on}]^2$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$

Fig. 7.3: LNCP: inference rules for equational state $\langle \gamma \mid \lambda\overline{x}.s \approx \lambda\overline{x}.t \mid \xi \rangle$

root(s) \ root(t)	$\mathcal{V}(t)$	$\mathcal{F}_d$	$\mathcal{F}_c \cup \mathcal{BV}(t)$	$\mathcal{F}_e$
$\mathcal{V}(s)$	$[\mathrm{f}]^1/[\mathrm{del}]^2/[\mathrm{ff}]^3/[\mathrm{cp}]^4$	$[\mathrm{f}]^1/([\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/([\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$
$\mathcal{F}_d$	$[\mathrm{f}]^1/([\mathrm{on}]^2, [\mathrm{i}]^2, [\mathrm{p}]^2)$	$[\mathrm{f}]^1/[\mathrm{del}]/([\mathrm{on}]^3_1, [\mathrm{on}]^3_2, [\mathrm{d}]^3)$	$[\mathrm{f}]^1/[\mathrm{on}]^2$	$[\mathrm{f}]^1/[\mathrm{on}]^2$
$\mathcal{F}_c \cup \mathcal{BV}(s)$	$[\mathrm{f}]^1/([\mathrm{i}]^2, [\mathrm{p}]^2)$	[f]	$[\mathrm{f}]^1/[\mathrm{d}]^2$	$[\mathrm{f}]^1//([\mathrm{cp}]^2, [\mathrm{va}]^2)$
$\mathcal{F}_e$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$	[f]	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$	$[\mathrm{f}]^1/([\mathrm{cp}]^2, [\mathrm{va}]^2)$

Fig. 7.4: LNCP: inference rules for equational state $\langle \gamma \mid \lambda\overline{x}.s \rhd \lambda\overline{x}.t \mid \xi \rangle$

The goal selection function

The goal selection function depends on the history of the LNCP-derivation which is captured in the syntactic structure of the runtime goal.

First, we define two subsets of the set of inference rules of LNCP for elementary states: the set $\mathcal{R}^{prec}$ of inference rules for equations with precursors, and the set $\mathcal{R}^{noprec}$ of inference rules for equations without precursors.

$$\mathcal{R}^{noprec} = \{[\mathrm{f}], [\mathrm{d}], [\mathrm{del}], [\mathrm{on}], [\mathrm{i}], [\mathrm{p}], [\mathrm{va}], [\mathrm{ff}], [\mathrm{cp+}], [\circledast], [\mathrm{wait}]\},$$
$$\mathcal{R}^{prec} = \mathcal{R}^{noprec} \setminus \{[\mathrm{ff}]\}.$$

For a given runtime goal G we define the partial function

$$Fc(G,\alpha) := \begin{cases} Fc(G_k,\alpha) & \text{if } G=\{G_1,\ldots,G_n\},\ Fc(G_i,\alpha)=\perp \\ & \text{for all } i<k \text{ and } Fc(G_k,\alpha)\neq\perp, \\ Fc(G_1,\alpha) & \text{if } G=\mathrm{prec}(G_1,G_2) \text{ and } Fc(G_1,\alpha)\neq\perp, \\ Fc(G_2,\mathit{prec}) & \text{if } G=\mathrm{prec}(G_1,G_2),\ Fc(G_1\alpha)=\perp, \\ & \text{and } Fc(G_2,\mathit{prec})\neq\perp, \\ \langle G,R\rangle & \text{if } G\in\mathcal{E}q(\mathcal{F},\mathcal{V})\cup\{\circledast\} \\ & \cup\{\mathtt{wait}(\xi,m)\mid \xi\in 2^{\mathcal{E}q(\mathcal{F}_c\cup\mathcal{F}_e,\mathcal{V})}, m\in\mathbb{N}\} \\ & \text{and } R\neq\emptyset, \text{ where } R \text{ is the set of inference} \\ & \text{rules in } \mathcal{R}^\alpha \text{ with highest priority} \\ & \text{which are applicable to } u\cong v, \\ \perp & \text{otherwise} \end{cases}$$

where $\alpha \in \{\mathit{prec}, \mathit{noprec}\}$. The result $Fc(G)$ is $\perp$ if G is a final goal, or a pair of the form $\langle e,\mathcal{R}\rangle$ where e is the equational subgoal which is selected, and R is the set of inference rules that are applied nondeterministically upon [OTH] steps.

The goal selection function sel_{goal} is defined by:

$$sel_{goal}(G) := \begin{cases} \perp & \text{if } Fc(G,\mathit{noprec})=\perp, \\ e & \text{if } Fc(G,\mathit{noprec})=\langle e,R\rangle. \end{cases}$$

From this definition results that the inference rule [ff] is never applied to flex/flex equations with precursors.

7.2.4 The Calculus LCN_2

LCN_2 with strategy $\mathcal{S}_c$ is a suitable choice for solving problems that can be described with `CFLP` programs which are left-linear confluent fully-extended conditional PRSs, and neither the program nor the goal contain external operators.

The `CFLP` implementation of LCN_2 performs the inference rules of $\mathrm{LNCP}\cup\{[\mathrm{rm}]\}$, except [va], [cp+], [cs] and [wait] which are not needed in pure functional logic programming. Therefore we can characterize LCN_2 by stating its inference rules for equational states. We group these inference rules into two categories:

$\mathcal{R}_s$: inference rules for strict equations,

$\mathcal{R}_{ns}$: inference rules for nonstrict equations.

Restrictions

The restrictions of the calculus LCN_2 are the ones described in Chapter 5, Sect. 5.9 for PRSs, but lifted to conditional PRSs. We describe here the way how these restrictions are implemented in CFLP.

For each of the sets $\mathcal{R}_s$ and $\mathcal{R}_{ns}$ we consider two versions, one for equations with precursors and another one for equations without precursors. The sets of inference rules for equations without precursors are:

- $\mathcal{R}_s^{noprec} = \mathcal{R}_s$ if the selected equation is strict,
- $\mathcal{R}_{ns}^{noprec} = \{$[f], [d], [on], [ov]^{restr}, [i], [p], [ff]$\}$ if the selected equation is non-strict.

and the sets of inference rules for equations with precursors are:

- $\mathcal{R}_s^{prec} = \mathcal{R}_s$,
- $\mathcal{R}_{ns}^{prec} = \{$[f], [d], [on], [ov], [i], [p]$\}$ for the subcalculus $\mathrm{LCN}_2^{\approx}$.

We express the restrictions on the inference rules applicable on the selected equation of a runtime goal G with the partial function $Fc(G, noprec)$. The call $Fc(G, noprec)$ yields $\perp$ if G is a final goal; otherwise, it returns the pair $\langle e, R\rangle$ with e the selected equation and R the set of inference rules which are applied nondeterministically during an [OTH]-step to solve the selected equation. The function Fc is defined as follows:

$$Fc(G,\alpha) := \begin{cases} Fc(G_k,\alpha) & \text{if } G = \{G_1,\dots,G_n\},\ Fc(G_i,\alpha) = \perp \\ & \text{for all } i < k \text{ and } Fc(G_k,\alpha) \neq \perp, \\ Fc(G_1,\alpha) & \text{if } G = \mathrm{prec}(G_1,G_2) \text{ and } Fc(G_1,\alpha) \neq \perp, \\ Fc(G_2,prec) & \text{if } G = \mathrm{prec}(G_1,G_2),\ Fc(G_1\alpha) = \perp, \\ & \text{and } Fc(G_2,prec) \neq \perp, \\ \langle u \cong v, R\rangle & \text{if } u \cong v \in \mathcal{E}q(\mathcal{F},\mathcal{V}) \text{ and } R \neq \emptyset, \text{ where } R \text{ is} \\ & \text{the set of inference rules in } \mathcal{R}_\beta^\alpha \text{ with} \\ & \text{highest priority which are applicable to } u \cong v, \\ \perp & \text{otherwise} \end{cases}$$

where $\alpha \in \{prec,\ noprec\}$.

The goal selection function

The goal selection function sel_{goal} is defined by:

$$sel_{goal}(G) := \begin{cases} \perp & \text{if } Fc(G, noprec) = \perp, \\ e & \text{if } Fc(G, noprec) = \langle e, R\rangle. \end{cases}$$

Note that $sel_{goal} \in \mathcal{S}_n$, and thus the calculus LN_4 with selection function sel_{goal} is sound and complete for `CFLP` programs represented by left-linear confluent fully-extended pattern rewrite systems.

7.2.5 Other Calculi

The other built-in calculi of the `CFLP` interpreter are implemented in a similar way as the calculi described before. In general, the built-in calculi correspond to implementations of different definitions of the function Fc and use the goal selection function of the calculus LNCP.

7.3 The Distributed Constraint Solving Subsystem

In the current implementation of `CFLP`, the constraint solving resources of the system are built on top of the constraint solving capabilities provided by *Mathematica*. Four types solvers for solving constraints over the domain of real and complex numbers have been integrated into the system:

- a solver for systems of equations between multivariate polynomials over the domain of complex numbers.
- a solver for systems of differential equations over algebraic extensions of $\mathbb{C}$,
- a solver for partial differential equations over algebraic extensions of $\mathbb{C}$,
- a variable elimination solver which can solve algebraic equations which involve invertible functions.

All solvers are implemented by separate *Mathematica* processes executing in parallel and communicating with the constraint scheduler via *MathLink* connections. There are two types of CFLP constraint solvers:

- Local constraint solvers. These solvers run as subsidiary *Mathematica* kernel processes of the `CFLP` constraint scheduler and run on the same machine with the `CFLP` interpreter.
- Shared constraint solvers. A shared constraint solver is started on the local or a remote machine from outside a `CFLP` session and can be connected later to more schedulers, which may run on possibly different machines. This means that we may have the situation depicted in Fig. 7.5.

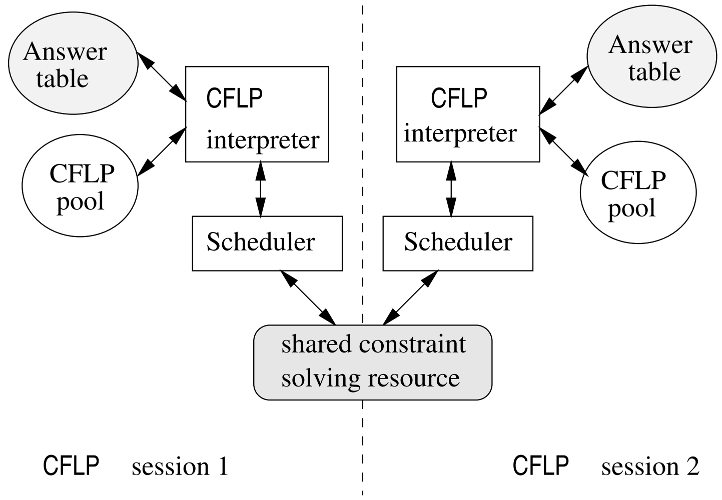


Fig. 7.5: CFLP: shared constraint solving architecture

During a CFLP session, the distributed constraint solving subsystem can be configured by specifying:

- the number and type of the local constraint solving resources,

- the remote machines which provide shared constraint solving resources. The scheduler component of a CFLP session maintains a list of machine names which are intended to provide shared constraint solving resources. Whenever the user modifies this list upon the system configuration, the CFLP scheduler does the following operations:

  1. it disables its connections to the shared constraint solving resources running on machines which are not specified in the current list of machine names,

  2. it establishes *MathLink* connections to the constraint solving resources available on the newly specified machines.

The facility to use shared constraint resources is currently supported only on Unix-like platforms.

7.4 The User Interface

CFLP is delivered as a collection of packages that can be loaded upon a *Mathematica* session. By loading the package TSolve.m and executing the command

StartCFLP[ ]

the user of a *Mathematica* session starts the processes that implement the CFLP system and gets access to its constraint solving capabilities.

Access to CFLP is provided via the function **TSolve**. The syntax call is:

TSolve[*goal*, *opts*]

or

TSolve[*goal*, *vars*, *opts*] where

- *goal* is the variable annotated goal to be solved in CFLP syntax, and
- *opts* are *Mathematica* options specific to the **TSolve** call,
- *vars* is a list of symbols which are interpreted as logical variables in the goal.

The following options are recognized by the **TSolve** function:

Rules the list of variable annotated rules of the CFLP program,

Constructor the list of constructor symbols used in the specifications of *goal* and of the CFLP program,

DefinedSymbol the list of defined symbols introduced by the CFLP program. This option is now obsolete: upon a **TSolve** call, all the symbols defined in the program specified with the **Rules** option are regarded as defined symbols,

NSolution the maximum number of solutions to be computed upon a **TSolve** call,

ConstraintVars the constraint variables in the initial goal.

Upon a **TSolve** call, the variable symbols of the goal and CFLP program are identified as follows:

- the logical variables in *goal* are identified by putting an overbar above to at least one occurrence of each variable in *goal*,

- the variables in a `CFLP` rule are identified by underlining at least one occurrence of each variable in that rule.

In addition, the variables, defined symbols, and constructor symbols may be type annotated. A type annotation of a symbol v is an expression of the form $v : \tau$ where τ is a type expression. A type expressions can be:

- Basic type. The following basic types are recognized in `CFLP`: `Bool` for booleans, `Float` for real numbers, `Compl` for complex numbers, `Strings` for strings in *Mathematica* representation, `Rat` for rationals, `Int` for integers, *Symb* for symbols in *Mathematica* representation, and $\Box$ for the void type.
- Type variable. Type variables are denoted by undefined *Mathematica* symbols.
- List type. The expression $\mathrm{TyList}(\tau)$ denotes the type of lists with elements of type τ.
- Function type. The expression $\tau_1 \times \ldots \times \tau_n \to \tau$ denotes the type of functions which take arguments of types $\tau_1, \ldots, \tau_n$ and return a result of type τ.

The constructor symbols may be type annotated in the list of constructors specified with the **Constructor** option, and the defined symbols may be type annotated in the list of defined symbols specified with the **DefinedSymbol** option. The logical variables of a goal can type annotated in the places of their underlined occurrences in the goal. The variables of a `CFLP` rule can be type annotated in the places of their overlined occurrences in the rule.

Type annotations are meaningful if we enable a polymorphic type checker which verifies the type consistency of the `CFLP` goal and program. If the polymorphic type checker is disabled, then the type annotations are simply ignored. For example, in the call

$$
\begin{array}{l}
\mathbf{TSolve}[\mathtt{f}\ [x, \overline{x : \mathrm{Float}}] \approx \{x\}], \\
\qquad \mathbf{DefinedSymbol} \to \{\mathtt{f} : \mathrm{Float} \times \mathrm{Float} \to \mathrm{TyList}[\mathrm{Float}]\}, \\
\qquad \mathbf{Rules} \to \{\mathtt{f}[\underline{x}, \underline{y : \mathrm{Float}}] \to \{y^2 + 2\ x\}\}\ ]
\end{array}
$$

the `CFLP` goal is $\mathtt{f}[x, x] \approx x$ with logical real variable x, and the `CFLP` program consists of the rule

$$\mathbf{f}[x, y] \to \{y^2 + 2\ x\}\}$$

where x, y are universally quantified variables over the domain of reals. In addition, it is known that f is a defined function symbol of two real arguments which returns as result a list of reals.

The defined symbols can also be declared with the `CFLP` command **DefinedSymbol**; instead of specifying the option

$$\textbf{DefinedSymbol} \to \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$$

inside a **TSolve** call, the user can first give the command

$$\textbf{DefinedSymbol}[f_1 : \tau_1, \ldots, f_n : \tau_n]$$

and next perform the **TSolve** call without the option **DefinedSymbol**.

In a similar way, the system `CFLP` provides command counterparts for the options **Constructor** and **Rules**.

Support for typing in expressions in `CFLP` syntax is provided via a palette which is displayed when the `CFLP` system is started. In addition, the palette contains control buttons which allow the user to

- configure the `CFLP` session,
- enable/disable type-checking the `CFLP` goal and program,
- interrupt a non-terminating **TSolve** call,
- end the `CFLP` session.

Chapter 8

Examples

In this section we present examples which illustrate the constraint solving capabilities of the CFLP system.

8.1 Program Calculation

Consider the problem of writing a program to check whether a list of numbers is *steep* [HT99]. A list is said to be steep if each element of the list is greater than the average of the elements that follow it. A straightforward CFLP program to solve this problem is

$$\begin{array}{rcl}
\mathtt{steep}[\{\}] & \to & \text{True}, \\
\mathtt{steep}[[\underline{n} \mid \underline{ns}]] & \to & n > \mathtt{avg}[ns] \land \mathtt{steep}[ns], \\
\mathtt{avg}[\underline{ns}] & \to & \mathtt{sum}[ns]/\mathtt{length}[ns], \\
\mathtt{sum}[\{\}] & \to & 0 \\
\mathtt{sum}[[\underline{n} \mid \underline{ns}]] & \to & n + \mathtt{sum}[ns], \\
\mathtt{length}[\{\}] & \to & 0, \\
\mathtt{length}[[\underline{n} \mid \underline{ns}]] & \to & 1 + \mathtt{length}[ns]
\end{array} \tag{8.1}$$

where

- $\land$: Bool $\times$ Bool $\to$ Bool, $>$: Rat $\times$ Rat $\to$ Rat, $/$: Int $\times$ Int $\to$ Rat are external operators,
- `steep`, , `avg`, `sum`, `length` are the defined symbols which are specified in the CFLP program.

This program is modular and easy to read. Unfortunately it is very inefficient (of quadratic complexity) due to the repeated applications of `avg` to the sublists. It can be shown that an efficient program with linear complexity exists.

$$\texttt{steepOpt}[\underline{ns}] \quad\rightarrow\quad \texttt{sel-c1}[\texttt{steepAux}[ns]]$$

where sel-c1, sel-c2 and sel-c3 are the selectors of the ternary constructor c, and

$$\begin{array}{rcl} \texttt{steepAux}[\{\}] & \rightarrow & \texttt{c}[\mathrm{True}, 0, 0], \\ \texttt{steepAux}[[\underline{n} \mid \underline{ns}]] & \rightarrow & \texttt{c}[\ \ (n > \texttt{sel-c2}[ns]/\texttt{sel-c3}[ns]) \wedge \texttt{sel-c1}[ns], \\ & & \quad n + \texttt{sel-c2}[ns], \\ & & \quad 1 + \texttt{sel-c3}[ns]] \end{array}$$

It is desirable to have a means to automatically compute the efficient version steepOpt of the steep function from its readable version. This kind of program transformation is based on the theory of constructive algorithmics [Bir87, Bac95], and can be described by a series of applications of calculational rules that describe program properties.

The computation of steepOpt from steep can be realized with the following *fusion* calculational rule [Bir89]:

$$\frac{\begin{array}{rcl} f[e] & = & e' \\ f[g[\underline{a}, \underline{r}]] & = & h[\underline{a}, \underline{r}] \end{array}}{f[\texttt{foldr}[g, e, \underline{x}]] \;=\; \texttt{foldr}[h, e', \underline{x}]} \tag{8.2}$$

where $\texttt{foldr} : (\alpha \times \beta \rightarrow \beta) \times \beta \times \mathrm{TyList}[\alpha] \rightarrow \beta$ is defined as usual:

$$\begin{array}{rcl} \texttt{foldr}[\underline{g}, \underline{e}, \{\}] & \rightarrow & e, \\ \texttt{foldr}[\underline{g}, \underline{e}, [\underline{n} \mid \underline{ns}]] & \rightarrow & g[n, \texttt{foldr}[g, e, ns]]. \end{array}$$

The computation proceeds as follows:

1. consider $\texttt{steepAux}[\underline{ns}] = \texttt{c}[\texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]]$ where c is a ternary constructor operator,

2. to make rule (8.2) applicable, we choose $f = \texttt{steepAux}$, $g = \texttt{Cons}$ (the CFLP list constructor), $e = \{\}$ and $e' = \texttt{steepAux}[e] = \texttt{c}[\mathrm{True}, 0, 0]$,

3. compute h such that $\texttt{steepAux}[[n \mid ns]] = h[n, \texttt{steepAux}[ns]]$ for all naturals n and lists of natural numbers ns. More specifically, we look for $h2, h2, h3$ such that

$$\begin{array}{rl} \texttt{steepAux}[[n \mid ns]] = \texttt{c}[& h1[n, \texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]], \\ & h2[n, \texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]], \\ & h3[n, \texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]]], \end{array}$$

4. compute $\texttt{steepOpt} = \lambda[\{ns\}, \texttt{sel-c1}[\texttt{foldr}[h, e', ns]]]$.

Now we show how this translation can be achieved with the CFLP system. We consider the CFLP program

$$
\begin{aligned}
\text{Prog} := \{\ & \texttt{steep}[\{\}] \to \text{True},\\
& \texttt{steep}[[\underline{n} \mid \underline{ns}]] \to n > \texttt{avg}[ns] \wedge \texttt{steep}[ns],\\
& \texttt{avg}[\underline{ns}] \to \texttt{sum}[ns]/\texttt{length}[ns],\\
& \texttt{sum}[\{\}] \to 0\\
& \texttt{sum}[[\underline{n} \mid \underline{ns}]] \to n + \texttt{sum}[ns],\\
& \texttt{length}[\{\}] \to 0,\\
& \texttt{length}[[\underline{n} \mid \underline{ns}]] \to 1 + \texttt{length}[ns],\\
& \texttt{steepAux}[\underline{ns}] \to \texttt{c}[\texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]]\};
\end{aligned}
$$

and the CFLP goal:

$$
\begin{aligned}
G :=\ & \lambda[\{n, ns\}, \texttt{steepAux}[[n \mid ns]]]\\
& \approx\\
& \lambda[\{n, ns\}, \texttt{c}[\ \overline{h1}[n, \texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]],\\
& \qquad\qquad\quad \overline{h2}[n, \texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]],\\
& \qquad\qquad\quad \overline{h3}[n, \texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]]]];
\end{aligned}
$$

Note the way how the λ-abstraction construct is used to express an equational formula of the form $\forall\overline{x}.(s \approx t)$ in higher-order logic: we write $\lambda\overline{x}.s \approx \lambda\overline{x}.t$. In particular, G denotes the equational formula

$$
\begin{aligned}
& \exists h1, h2, h3.\ \forall n, ns.\\
& \qquad \texttt{steepAux}[[n \mid ns]] \approx \texttt{c}[\ h1[n, \texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]],\\
& \qquad\qquad\qquad\qquad\qquad\qquad h2[n, \texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]],\\
& \qquad\qquad\qquad\qquad\qquad\qquad h3[n, \texttt{steep}[ns], \texttt{sum}[ns], \texttt{length}[ns]]].
\end{aligned}
$$

By performing the command:

TSolve[G,
  Rules $\to$ Prog,
  Constructor $\to$ $\{\texttt{c}, \text{Plus}, \text{Greater}, \text{And}, \text{Times}, \text{Power}\}$]

the system responds:

$$
\begin{aligned}
\{\{\ & h1 \to \lambda[\{z\$125, z\$126, z\$127, z\$128\}, z\$125 > \frac{z\$127}{z\$128}\ \&\&\ z\$126],\\
& h2 \to \lambda[\{z\$409, z\$410, z\$411, z\$412\}, z\$409 + z\$411],\\
& h3 \to \lambda[\{z\$533, z\$534, z\$535, z\$536\}, 1 + z\$536]\}\}
\end{aligned}
$$

The computation of steepOpt from steepAux is now straightforward.

Remarks

The application of the fusion rule for computing the efficient version of the `steep` function requires the capability to perform higher-order term rewriting and unification of higher-order patterns. Both operations are supported in `CFLP`.

Note that upon the **TSolve** call we declare the *Mathematica* operators `And`, `Greater`, `Plus`, `Times`, and `Power` as constructors. By default, these operators are external symbols, and the calculus LNCP will generate corresponding constraints which are submitted to the distributed constraint solver. For this particular problem we wish to treat these operators as constructors, and therefore we declare them as such.

8.2 Electrical Circuit Modeling

Our second example shows how electric circuit layouts can be described and computed with `CFLP`. This example illustrates the utility of simply-typed λ-terms for describing

- the behaviour in time of electrical circuits,
- the recursive construction of complex circuit layouts.

In addition, the computation which describes the composition of electrical circuits is a complex task which requires the cooperation of different constraint solvers.

We first define the `spec` function which describes the behavior in time t of an electrical component as a function of the current $i[t]$ and voltage $v[t]$ in the circuit. The `CFLP` rules of `spec` correspond to a recursive definition, where we base case describes the behavior of elementary circuits such as resistor, capacitor, and inductor, and the inductive case describes the behavior of serial and parallel connections of electrical components. The `CFLP` program is given below. We don't explain the underlying electronic laws since it should be easy to read them off from the program.

(* declare the `CFLP` constructors *)
Constructor[`res`, `ind`, `cap`, `comp`, `serial`, `parallel`];
(* specify the `CFLP` program *)
Prog:= {
$\quad \texttt{spec}[\texttt{res}[\underline{r : \text{Float}}], \underline{v : \text{Float} \to \text{Float}}, \underline{i : \text{Float} \to \text{Float}}] \to \text{True} \Leftarrow$
$\qquad \lambda[\{t\}, v[t]] \approx \lambda[\{t\}, r * i[t]],$
$\quad \texttt{spec}[\texttt{ind}[\underline{l : \text{Float}}], \underline{v : \text{Float} \to \text{Float}}, \underline{i : \text{Float} \to \text{Float}}] \to \text{True} \Leftarrow$
$\qquad \lambda[\{t\}, v[t]] \approx \lambda[\{t\}, l * i'[t]],$

$$
\begin{aligned}
&\texttt{spec}[\texttt{cap}[\underline{c : \mathrm{Float}}], \underline{v : \mathrm{Float} \to \mathrm{Float}}, \underline{i : \mathrm{Float} \to \mathrm{Float}}] \to \mathrm{True} \Leftarrow \\
&\qquad \lambda[\{t\}, i[t]] \approx \lambda[\{t\}, c * v'[t]], \\
&\texttt{spec}[\texttt{serial}[\{\}], \lambda[\{t\}, 0], \underline{i : \mathrm{Float} \to \mathrm{Float}}] \to \mathrm{True}, \\
&\texttt{spec}[\texttt{serial}[[\underline{comp} \mid \underline{comps}]], \underline{v}, \underline{i}] \to \mathrm{True} \Leftarrow \\
&\qquad \{\texttt{spec}[comp, v1, i] \approx \mathrm{True}, \texttt{spec}[\texttt{serial}[comps], v2, i] \approx \mathrm{True}, \\
&\qquad \lambda[\{t\}, v[t]] \approx \lambda[\{t\}, \underline{v1}[t] + \underline{v2}[t]]\}, \\
&\texttt{spec}[\texttt{parallel}[\{\}], \underline{v : \mathrm{Float} \to \mathrm{Float}}, \lambda[\{t\}, 0]] \to \mathrm{True}, \\
&\texttt{spec}[\texttt{parallel}[[\underline{comp} \mid \underline{comps}]], \underline{v}, \underline{i}] \to \mathrm{True} \Leftarrow \\
&\qquad \{\texttt{spec}[comp, v, \underline{i1}] \approx \mathrm{True}, \texttt{spec}[\texttt{parallel}[comps], v, \underline{i2}] \approx \mathrm{True}, \\
&\qquad \lambda[\{t\}, i[t]] \approx \lambda[\{t\}, \underline{i1}[t] + \underline{i2}[t]]\}\}
\end{aligned}
$$

Note the usage of the list construct in the recursive specification of serial and parallel connections of electrical components. The `CFLP` system recognizes the following list constructions:

- $\{t_1, \ldots, t_n\}$: the list consisting of components $t_1, \ldots, t_n$,
- $[\ h \mid tl\ ]$: `CFLP` list in Prolog-style notation; it denotes the list with head h and tail tl.

8.2.1 RLC Circuit

Consider the problem of finding the behavior in time of the current in an electrical circuit consisting of a serial connection of a resistor with $R = 2$, an inductor with $L = 1$ and a capacitor with $C = 1/2$ (see Fig. 8.1), under the additional conditions that the voltage is a constant in time and the current was 0 at time 0.

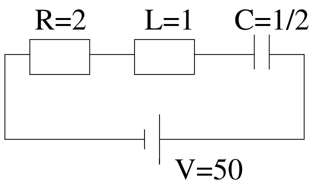


Fig. 8.1: RLC circuit

In this case, the goal which we want to solve is

$$G := \{\texttt{spec}[\texttt{serial}[\{\texttt{res}[2], \texttt{ind}[1], \texttt{cap}[1/2]\}],\ \lambda[\{t\}, 50], \bar{i}] \approx \mathrm{True},\ i[0] \approx 0\}$$

The logical variable (that is, the existentially quantified variable) of the goal is i. Note the usage of the expression $\lambda[\{t\}, 50]$ for specifying that the voltage is constant (50 Ω) in time.

Now we can ask the CFLP system to solve the problem by computing the solutions of G :

$$\textbf{TSolve}[\mathrm{G}, \textbf{Rules} \to \mathrm{Prog}, \textbf{ConstrVariables} \to \{i\}]$$

The system responds:

$$\{\{i \to \lambda[\{t\}, -\mathrm{c1\$5}\ \ e^{-t}\ \mathrm{Sin}[t]\}\}$$

Note that the computed answer is a parametric solution, since c4 is a variable.

The CFLP system allows the user to trace the computation steps performed by the interpreter and by the distributed constraint solving subsystem. The trace of the execution of the interpreter shows how the initial configuration $\langle \varepsilon \mid G \mid \{\}\rangle$ is reduced by the CFLP interpreter to a state of the form $\langle \varepsilon \mid \circledast \mid \xi\rangle$ where

$$\begin{aligned}\xi = \ & \{\mathrm{v\$348} \approx \lambda[\{t\}, i'[t]], i \approx \lambda[\{t\}, \mathrm{v\$381}'[t]/2],\\ & \lambda[\{t\}, V] \approx \lambda[\{t\}, 2\ i[t] + \mathrm{v\$348}[t] + \mathrm{v\$381}[t]], i[0] \approx 0\}\end{aligned}$$

The system of constraints ξ is sent to the constraint scheduler which coordinates the instances of the constraint solvers CS_1, CS_2, CS_3, CS_4 to solve it. A trace of the computation performed by the distributed constraint subsystem reveals how the answer is computed.

1. An instance of CS_1 acts on $\langle \varepsilon \mid \xi\rangle$ and yields $\langle \theta_1 \mid \xi_1\rangle$ where
   - $\theta_1 = \{\mathrm{v\$348} \to \lambda[\{t\}, \mathrm{v\$381}''[t]/2], i \to \lambda[\{t\}, \mathrm{v\$381}'[t]/2]\}$,
   - $\xi_1 = \ \{\lambda[\{t\}, V - \mathrm{v\$381}[t] - \mathrm{v\$381}'[t] - \mathrm{v\$381}''[t]/2] \approx \lambda[\{t\}, 0], \mathrm{v\$381}'[0] \approx 0\}$,
   - $\mathrm{v\$348}, \mathrm{v\$381}$ are fresh variables introduced upon LNCP steps.
2. An instance of CS_3 is applied to $\langle \theta_1, \xi_1\rangle$ (note that CS_2 is not applicable to ξ_2) and yields $\langle \theta_1\theta_2, \xi_2\rangle$ where
   - $\theta_2 = \{\mathrm{v\$381} \to \lambda[\{t\}, V + c1\$5\ e^{-t}\ \mathrm{Cos}[t] - c1\$4\ e^{-t}\ \mathrm{Sin}[t]]\}$,
   - $\xi_2 = \{c1\$5 - c1\$4 \approx 0\}$
3. An instance of CS_1 is applied to $\langle \theta_2, \xi_2\rangle$ and yields $\langle \theta_1\theta_2\theta_3, \{\}\rangle$ where $\theta_3 = \{c1\$4 \to -c1\$5\}$,
4. The CFLP scheduler returns the result to the CFLP interpreter, such that the following [G]-step is performed:

$$\langle \varepsilon \mid \circledast \mid \xi\rangle \Rightarrow \langle \theta_1\theta_2\theta_3 \mid \Box \mid \emptyset\rangle.$$

In this way the system reaches a final configuration, from which is generated the answer $(\theta_1\theta_2\theta_3)\restriction_{\mathcal{V}(G)}$, that is $\{i \to \lambda[\{t\}, -\mathrm{c1\$5}\ e^{-t}\ \mathrm{Sin}[t]\}$.

If we supply more information, such as the value of the current at time $t = 1$:

$$\mathbf{TSolve}[\{\mathrm{G}, \bar{i}[1] \approx 1\}, \mathbf{Rules} \to \mathrm{Prog}, \mathbf{ConstrVariables} \to \{i\}]$$

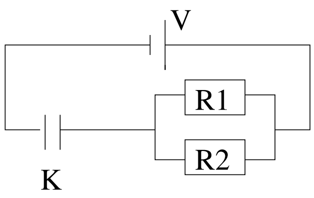
then the system is able to uniquely determine the current as a function of time:

$$\{\{i \to \lambda[\{t\}, \frac{1}{2}\ e^{1-t}\ \mathrm{Csc}[1]\ \mathrm{Sin}[t]\}\}.$$

8.2.2 A Problem of Circuit Design

The following example illustrates how OR-parallelism is treated in `CFLP`.

Let an electric circuit be given with a resistor $R1$ of $0.1M\Omega$ connected in parallel with a resistor $R2$ of $0.4M\Omega$, and a capacitor K in serial connection with the two resistors. Also, there is a kit of electrical components in which capacitors of $1\mu F$, $3\mu F$, $5\mu F$, $20\mu F$ and $50\mu F$ are available. We want to know which capacitor to use in our circuit such that the time until the voltage of the capacitor reaches 98% of the final voltage is less than 1s. To



solve this problem, we employ a slightly changed version of the program Prog used in solving the previous problem. In order to make the voltage of a capacitor explicit, we specify it in the following way:

$\mathtt{spec}[\mathtt{cap}[\underline{c}, \underline{v}], v, \underline{i}] \to \mathrm{True} \Leftarrow$
$\quad \lambda[\{t\}, i[t]] \approx \lambda[\{t\}, c * v'[t]]$

Let Prog1 be the program corresponding to this slight change. Now, the `CFLP` goal corresponding to our problem is:

$\mathrm{G} := \{k \approx 10^{-6} \vee k \approx 3 * 10^{-6} \vee k \approx 2 * 10^{-5} \vee k \approx 5 * 10^{-5},$
$\quad \mathtt{spec}[\mathtt{serial}[$
$\quad\quad \{\mathtt{cap}[\bar{k}, V1], \mathtt{parallel}[\{\mathtt{res}[10^5], \mathtt{res}[4 * 10^5]\}]\}],$

$$\begin{aligned}&\lambda[\{t\},\mathrm{V}],i] \doteq \mathrm{True},\\ \circledast,\ & V1[0] \approx 0,\\ \circledast,\ & (V1[1] > 0.98 * \mathrm{V}) \approx \mathrm{True}\}\end{aligned}$$

Note the usage of $\circledast$ to explicitly invoke the distributed constraint solving subsystem to solve the constraints accumulated so far, and the use of the parallel-OR construct to specify the components of our electrical kit. The logical variable used in the specification of G are:

i**:** $i[t]$ denotes the current at time t in the circuit,

k**:** the characteristic value of the capacitor,

$V1$**:** $V1[t]$ denotes the voltage of the capacitor at time t.

The call

TSolve[G, $\{V1, i\}$,
Rules $\rightarrow$ Prog1,
ConstrVariables $\rightarrow$ $\{i\}$,
Constructor $\rightarrow$ $\{\mathrm{V}\}$]

yields the answer:

$$\left\{\left\{k \rightarrow \frac{1}{1000000}\right\}, \left\{k \mapsto \frac{3}{1000000}\right\}\right\}$$

Note that `CFLP` yields answers only for the variables which are annotated in G, which in this case is k only. In this case, the initial configuration is $A = \langle \varepsilon \mid \mathrm{G} \mid \emptyset \rangle$, which is simplified immediately to $\bigvee_{i=1}^{4} A_i$ where A_i are states of the form $\langle \varepsilon \mid \mathrm{G}_i \mid \emptyset \rangle$ witth G1, G2, G3, G4 sequential-AND goals which differ only by the first equation. Assume that the distributed constraint solving subsystem of `CFLP` makes use of 3 instances of CS_1 and 2 instances of CS_3. A possible scenario of the evolution in time of the computation performed by `CFLP` upon solving the goal G is as shown in Fig. 8.2. Note that in the time interval $[t_1, t_2]$ there is no resource available to process the state which descended from A_4, and in the time interval $[t_2, t_4]$ there is no resource available to process the states which descended from A_3 and A_4. The computation performed in the time interval $[t_5, t_6]$ detects the inconsistency of the constraints represented in the descendants of A_3 and A_4.

The solving effort can be reduced if we take into account the operational semantics of `CFLP`. For instance, we can solve our problem by considering the goal

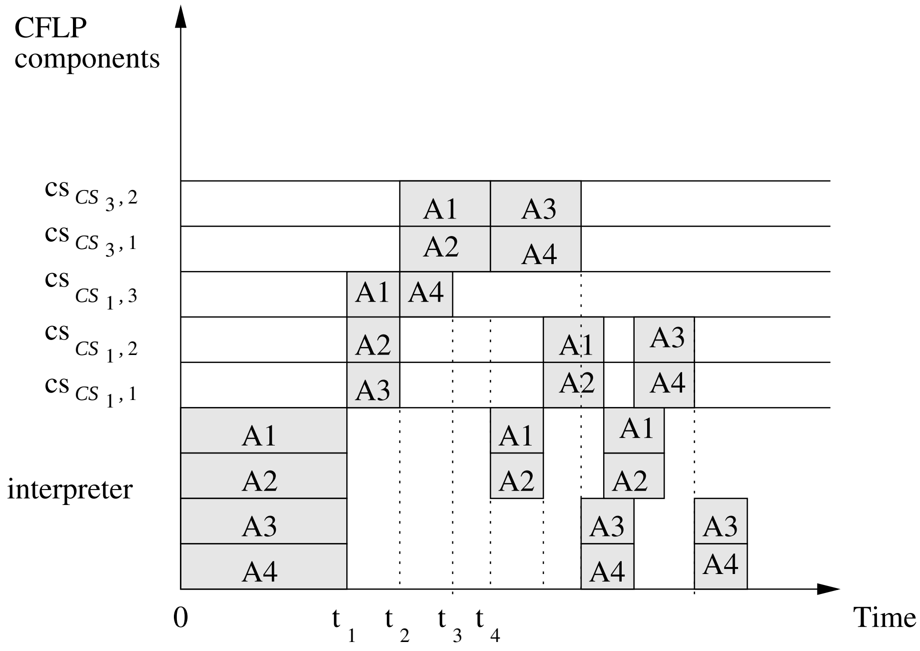


Fig. 8.2: `CFLP`: time complexity of overall computation

$$
\begin{aligned}
\mathrm{G1} := \{&\mathtt{spec}[\mathtt{serial}[\\
&\qquad \{\mathtt{cap}[\overline{k}, V1], \mathtt{parallel}[\{\mathtt{res}[10^5], \mathtt{res}[4*10^5]\}]\}],\\
&\qquad \lambda[\{t\}, V], i] \doteq \mathrm{True},\\
&\circledast, V1[0] \approx 0,\\
&k \approx 10^{-6} \vee k \approx 3*10^{-6} \vee k \approx 2*10^{-5} \vee k \approx 5*10^{-5},\\
&\circledast, (V1[1] > 0.98*V) \approx \mathrm{True}\}
\end{aligned}
$$

The call

TSolve[G1, $\{V1, i\}$,
  Rules $\rightarrow$ Prog,
  ConstrVariables $\rightarrow$ $\{i\}$,
  Constructor $\rightarrow$ {V}]

yields the same answer, but the computation consumes fewer resources. The reason is that `CFLP` first computes a solution for $V1$ which is generic for any characteristic value k of the capacitor. This computation requires only 2 constraint solving resources instead of 8. Next, k is bound to 4 possible values (parallel-OR nondeterminism) and `CFLP` filters out the inadmissible values of k.

8.2.3 Computation of Circuit Specifications

Consider the problem of finding the specifications of all the electrical circuits that can be built with a kit of given electrical components. We encode the specification of an electrical circuit as a list with 3 elements

$$\{comp, i[t], v[t]\}$$

where

- *comp* is a constructor term that describes the geometric structure of the electrical circuit. For example, the constructor term

  $$\texttt{serial}[\texttt{res}[\mathrm{R1}], \texttt{parallel}[\{\texttt{cap}[\mathrm{C1}], \texttt{res}[\mathrm{R2}], \texttt{ind}[\mathrm{L}]\}]]$$

  describes a serial connection of 3 electrical components: a resistor R1, a parallel connection of a capacitor C1 with a resistor R2, and an inductor L,

- $i[t]$ describes the behaviour in time of the current in the electrical circuit,

- $v[t]$ describes the behaviour in time of the voltage in the electrical circuit.

We represent the kit of available circuit components with a list, e.g.

$$\{\texttt{res}[10^5], \texttt{res}[4 * 10^5], \texttt{cap}[10^{-6}]\}$$

describes a kit consisting of two resistors with resistances of $0.1\mathrm{M}\Omega$ and $0.4\mathrm{M}\Omega$ and a capacitor with capacity of $1\mu\mathrm{F}$.

First we provide a predicate `MkComp`[*kit*, *comp*] which holds if *comp* is a circuit made of the components of the kit of electrical components *kit*. `MkComp` is defined in terms of two predicates which are mutually recursive: `MkSerialComp` and `MkParallelComp`. The meaning of these predicates can be easily read off from the `CFLP` rewrite rules given below. The auxiliary function `split` is provided for partitioning the electrical components of the kit into two subsets, which are used for constructing two electrical circuits which are connected in serial or in parallel.

$$
\begin{array}{l}
\texttt{MkComp}[\underline{kit}, \underline{comp}] \to \texttt{MkSerialComp}[kit, comp], \\
\texttt{MkComp}[\underline{kit}, \underline{comp}] \to \texttt{MkParallelComp}[kit, comp], \\
\texttt{MkSerialComp}[\underline{comp}, comp] \to \text{True}, \\
\texttt{MkParallelComp}[\underline{comp}, comp] \to \text{True}, \\
\texttt{MkSerialComp}[\underline{L}, comp] \to \text{True} \Leftarrow \{ \\
\quad \texttt{split}[L, \underline{L1}, [\underline{H} \mid \underline{T}]] \approx \text{True} \\
\quad \{\texttt{MkSerialComp}[L1, \underline{comp1}] \approx \text{True}, \\
\texttt{MkParallelComp}[[H \mid T], \underline{comp2}] \approx \text{True}\} || \\
\quad \{\texttt{MkParallelComp}[L1, \underline{comp1}] \approx \text{True}, \\
\quad \texttt{MkParallelComp}[[H \mid T], \underline{comp2}] \approx \text{True}\} || \\
\quad \{\texttt{MkParallelComp}[L1, \underline{comp1}] \approx \text{True}, \\
\quad \texttt{MkSerialComp}[[H \mid T], \underline{comp2}] \approx \text{True}\}, \\
\quad comp \approx \texttt{serial}[comp1, comp2]\}, \\
\texttt{MkParallelComp}[\underline{L}, comp] \to \text{True} \Leftarrow \{ \\
\quad \texttt{split}[L, \underline{L1}, [\underline{H} \mid \underline{T}]] \approx \text{True} \\
\quad \{\texttt{MkSerialComp}[L1, \underline{comp1}] \approx \text{True}, \\
\quad \texttt{MkParallelComp}[[H \mid T], \underline{comp2}] \approx \text{True}\} || \\
\quad \{\texttt{MkSerialComp}[L1, \underline{comp1}] \approx \text{True}, \\
\quad \texttt{MkSerialComp}[[H \mid T], \underline{comp2}] \approx \text{True}\} || \\
\quad \{\texttt{MkParallelComp}[L1, \underline{comp1}] \approx \text{True}, \\
\texttt{MkSerialComp}[[H \mid T], \underline{comp2}] \approx \text{True}\}, \\
\quad comp \approx \texttt{parallel}[comp1, comp2]\}, \texttt{split}[\{\underline{comp}\}, comp, \{\}] \to \text{True}, \\
\texttt{split}[\{\underline{comp}\}, \{\}, comp] \to \text{True}, \\
\texttt{split}[[\underline{comp1}, \underline{comp2} \mid \underline{T}], \underline{l1}, \underline{l2}] \to \text{True} \Leftarrow \\
\quad \{\texttt{split}[[comp1 \mid T], \underline{L1}, \underline{L2}] \approx \text{True}, \\
\quad \{l1 \approx [comp2 \mid L1], l2 \approx L_2\} || \{l1 \approx L1, l2 \approx [comp2 \mid L_2]\}\}
\end{array}
$$

A circuit consisting of only one component can be regarded either as a serial connection of one component, or as a parallel connection of one component. Note that any electrical circuit can be described either as a serial connection of two sub-circuits where one of them is a parallel connection, or as a parallel connection of two sub-circuits where one of them is a serial connection. This is the reason why we do not generate

- serial connections of serial connections, or
- parallel connections of parallel connections.

The predicate $\texttt{MkSerialComp}[L, comp]$ describes the following process of constructing a non-trivial serial connection of electric circuits with components from L:

1. First, L is split into nonempty two sublists $L1$ and $L2$,

2. $L1$ is used for constructing a component *comp1*, and $+2$ is used for constructing a component $L2$, such that at least one of the components *comp1* and *comp2* is a parallel component. Note how the sequential-OR construct is used in the condition part of `MkSerialComp` to specify this condition. Note that we could have used the parallel-OR construct instead,

3. the component *comp* is built up by connecting *comp1* and *comp2* in serial.

In a similar way is defined the construction of a non-trivial serial connection of electric circuits.

Let Prog1 be the program obtained by extending Prog with the defining rules of the predicates `MkComp`, `MkSerialComp`, `MkParallelComp` and `split`. This program can be used to generate all the possible configurations of circuits built with a given set of electrical components. For instance, the call

$$\begin{array}{l}\textbf{TSolve}[\{kit \approx \{\texttt{res}[R1], \texttt{res}[R2], \texttt{res}[R3]\},\\ \qquad\qquad \texttt{MkComp}[kit, \overline{comp}] \approx \texttt{True}\},\\ \qquad\quad \{kit\},\\ \qquad\quad \textbf{Rules} \to \mathrm{Prog1}, \textbf{Constructor} \to \{R1, R2, R3\}]\end{array}$$

yields as result a list of bindings of *comp* to all possible electrical circuit configurations built with 3 resistors $R1$, $R2$, $R3$:

$$\begin{array}{l}\{\{comp \to \texttt{serial}[\{\texttt{serial}[\{\texttt{res}[R3], \texttt{res}[R2]\}], \texttt{res}[R1]\}]\},\\ \ \ comp \to \texttt{parallel}[\{\texttt{serial}[\{\texttt{res}[R3], \texttt{res}[R2]\}], \texttt{res}[R1]\}]\},\\ \ \ comp \to \texttt{serial}[\{\texttt{serial}[\{\texttt{res}[R2], \texttt{res}[R3]\}], \texttt{res}[R1]\}]\},\\ \ \ comp \to \texttt{parallel}[\{\texttt{serial}[\{\texttt{res}[R2], \texttt{res}[R3]\}], \texttt{res}[R1]\}]\},\\ \ \ \ldots\}\end{array}$$

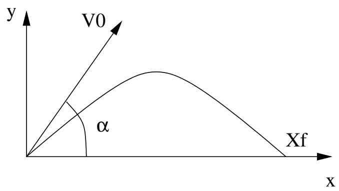
To compute the specifications of the electrical circuits consisting of two resistors $R1$ and $R2$, one inductor $L1$, and one capacitor $C1$, we can call

$$\begin{array}{l}\textbf{TSolve}[\{kit \approx \{\texttt{res}[R1], \texttt{res}[R2], \texttt{ind}[L1], texttttcap[C1]\},\\ \qquad\qquad \texttt{MkComp}[kit, comp] \approx \texttt{True},\\ \qquad\quad \texttt{spec}[comp, \lambda[\{t\}, V], \lambda[\{t\}, i[t]]] \approx \mathrm{True},\\ \qquad\quad \overline{compspec} \approx \{comp, \lambda[\{t\}, V], \lambda[\{t\}, i[t]]\}\},\\ \qquad\quad \{kit, comp, i\},\\ \qquad\quad \textbf{Rules} \to \mathrm{Prog1}, \textbf{Constructor} \to \{R1, R2, L1, C1, V\}]\end{array}$$

Upon the call we assume that the voltage is a constant in time. We express this fact by describing the voltage function with the λ-term $\lambda[\{t\}, V]$ where V is an arbitrary value. By declaring V as a constructor, the `CFLP` system will treat it as a symbolic constant. The call resumes with a list of bindings of *complist* to the specifications of circuits built with resistors $R1$, $R2$, inductor $L1$ and capacitor $C1$. At page 199 we illustrate the first 3 answers returned by `CFLP`.

8.3 A Ballistic Problem

This problem from [MR94] consists in finding the falling point of an object.



The object is launched with initial speed $V0$ and incidence angle α with the ground. The problem is to determine the falling point on the earth of an object, knowing its initial speed $V0 = 500\text{m/s}$ and the initial incidence $\alpha = 45^o$.

We denote by Tf the falling time, by Vx the horizontal projection of $V0$, and by Vy the vertical projection of $V0$. Then the movement of the object can be described in terms of its coordinates $(x[t], y[t])$ at moment t as follows:

$$\begin{aligned}
&\mathrm{Vx} = V0 * \cos(\alpha),\\
&\mathrm{Vy} = V0 * \sin(\alpha),\\
&x[t] = \mathrm{Vx} * t,\\
&y[t] = \mathrm{Vy} * t - 981 * t^2/200.
\end{aligned}$$

To solve this problem, we specify the following `CFLP` program:

$$\begin{array}{rl}
\mathrm{Prog} := \{\texttt{ballistic} & [\underline{\mathrm{V}}, \underline{\alpha}] \to \{\lambda[\{t\}, \underline{x}[t]], \lambda[\{t\}, \underline{y}[t]]\} \Leftarrow \{\\
& \mathrm{Vx} \approx \mathrm{V} * \mathrm{Cos}[\alpha], \mathrm{Vy} \approx \mathrm{V} * \mathrm{Sin}[\alpha],\\
& \lambda[\{t\}, x[\mathrm{t}]] \approx \lambda[\{t\}.\mathrm{Vx} * \mathrm{t}],\\
& \lambda[\{t\} y[\mathrm{t}]] \approx \lambda[\{t\}, \mathrm{Vy} * \mathrm{t} - 981 * t^2/200]\},\\
\texttt{falling} & [\underline{\mathrm{V0}}, \underline{\alpha}] \to \underline{\mathrm{Xf}} \Leftarrow \{\\
& \texttt{ballistic}[\mathrm{V0}, \alpha, \underline{\mathrm{Tf}}] \approx \{\lambda[\{t\}, \underline{x}[t]], \lambda[\{t\}, \overline{y}[t]]\},\\
& x[\underline{\mathrm{Tf}}] \approx \overline{\mathrm{Xf}}, y[\underline{\mathrm{Tf}}] \approx 0\}\}
\end{array}$$

and the goal

$$G := \texttt{falling}[500, \mathrm{Pi}/4] \approx \overline{\mathrm{Xf}}$$

The call

$$\mathbf{TSolve}[G, \mathbf{Rules} \to \mathrm{Prog}, \mathbf{ConstrVariables} \to \{\mathrm{Xf}\}]$$

will yield, apart form the degenerated solution $\{\mathrm{Xf} \to 0\}$, the solution $\left\{\mathrm{Xf} \to \dfrac{25000000}{981}\right\}$. In this case, the initial problem is reduced to solving a system of equations that is solved with the solvers CS_1 and CS_2.

$$
\Big\{\mathtt{compspec} \to \Big\{\mathtt{parallel[}
\{\mathtt{serial[\{serial[\{cap[C1],\ ind[L1]\}],\ res[R2]\}],\ res[R1]\}],\ \lambda[\{t\},\ V]},
\lambda\Big[\{t\},\ \mathtt{C1}\left(\frac{1}{2\sqrt{\mathtt{C1}}\ \mathtt{L1}}\left(\mathtt{c1\$1}\ e^{\frac{\left(-\sqrt{\mathtt{C1}}\ \mathtt{R2}+\sqrt{-4\ \mathtt{L1}+\mathtt{C1}\ \mathtt{R2}^2}\right) t}{2\sqrt{\mathtt{C1}}\ \mathtt{L1}}}\left(-\sqrt{\mathtt{C1}}\ \mathtt{R2}+\sqrt{-4\ \mathtt{L1}+\mathtt{C1}\ \mathtt{R2}^2}\right)\right) - \frac{1}{2\ \mathtt{C1}\ \mathtt{L1}}\left(\mathtt{c1\$2}\ e^{-\frac{\left(\mathtt{C1}\ \mathtt{R2}+\sqrt{\mathtt{C1}}\ \sqrt{-4\ \mathtt{L1}+\mathtt{C1}\ \mathtt{R2}^2}\right) t}{2\ \mathtt{C1}\ \mathtt{L1}}}\left(\mathtt{C1}\ \mathtt{R2}+\sqrt{\mathtt{C1}}\ \sqrt{-4\ \mathtt{L1}+\mathtt{C1}\ \mathtt{R2}^2}\right)\right)\right)+\frac{\mathtt{V}}{\mathtt{R1}}\Big]\Big]\Big\}\Big\}
$$

$$
\Big\{\mathtt{compspec} \to \Big\{\mathtt{serial[\{serial[\{serial[\{cap[C1],\ ind[L1]\}],\ res[R2]\}],\ res[R1]\}]},
\lambda[\{t\},\ V],\ \lambda\Big[\{t\},\ \mathtt{C1}\left(-\frac{1}{2\ \mathtt{C1}\ \mathtt{L1}}\left(\mathtt{c1\$3}\ e^{-\frac{\left(\mathtt{C1}\ \mathtt{R1}+\mathtt{C1}\ \mathtt{R2}-\sqrt{-4\ \mathtt{C1}\ \mathtt{L1}+(-\mathtt{C1}\ \mathtt{R1}-\mathtt{C1}\ \mathtt{R2})^2}\right) t}{2\ \mathtt{C1}\ \mathtt{L1}}}
\left(\mathtt{C1}\ \mathtt{R1}+\mathtt{C1}\ \mathtt{R2}-\sqrt{-4\ \mathtt{C1}\ \mathtt{L1}+(-\mathtt{C1}\ \mathtt{R1}-\mathtt{C1}\ \mathtt{R2})^2}\right)\right) -
\frac{1}{2\ \mathtt{C1}\ \mathtt{L1}}\left(\mathtt{c1\$4}\ e^{-\frac{\left(\mathtt{C1}\ \mathtt{R1}+\mathtt{C1}\ \mathtt{R2}+\sqrt{-4\ \mathtt{C1}\ \mathtt{L1}+(-\mathtt{C1}\ \mathtt{R1}-\mathtt{C1}\ \mathtt{R2})^2}\right) t}{2\ \mathtt{C1}\ \mathtt{L1}}}
\left(\mathtt{C1}\ \mathtt{R1}+\mathtt{C1}\ \mathtt{R2}+\sqrt{-4\ \mathtt{C1}\ \mathtt{L1}+(-\mathtt{C1}\ \mathtt{R1}-\mathtt{C1}\ \mathtt{R2})^2}\right)\right)\right)\Big]\Big]\Big\}\Big\}
$$

$$
\Big\{\mathtt{compspec} \to
\Big\{\mathtt{parallel[\{serial[\{serial[\{ind[L1],\ cap[C1]\}],\ res[R2]\}],\ res[R1]\}],\ \lambda[\{t\},}
\mathtt{V]},\ \lambda\Big[\{t\},\ \mathtt{C1}\left(\frac{\mathtt{c1\$5}\ e^{\frac{\left(-\sqrt{\mathtt{C1}}\ \mathtt{R2}+\sqrt{-4\ \mathtt{L1}+\mathtt{C1}\ \mathtt{R2}^2}\right) t}{2\sqrt{\mathtt{C1}}\ \mathtt{L1}}}\left(-\sqrt{\mathtt{C1}}\ \mathtt{R2}+\sqrt{-4\ \mathtt{L1}+\mathtt{C1}\ \mathtt{R2}^2}\right)}{2\sqrt{\mathtt{C1}}\ \mathtt{L1}} - \frac{1}{2\ \mathtt{C1}\ \mathtt{L1}}
\left(\mathtt{c1\$6}\ e^{-\frac{\left(\mathtt{C1}\ \mathtt{R2}+\sqrt{\mathtt{C1}}\ \sqrt{-4\ \mathtt{L1}+\mathtt{C1}\ \mathtt{R2}^2}\right) t}{2\ \mathtt{C1}\ \mathtt{L1}}}\left(\mathtt{C1}\ \mathtt{R2}+\sqrt{\mathtt{C1}}\ \sqrt{-4\ \mathtt{L1}+\mathtt{C1}\ \mathtt{R2}^2}\right)\right)\right)+\frac{\mathtt{V}}{\mathtt{R1}}\Big]\Big]\Big\}\Big\}
$$

$$
\Big\{\mathtt{compspec} \to \Big\{\mathtt{serial[\{serial[\{serial[\{ind[L1],\ cap[C1]\}],\ res[R2]\}],\ res[R1]\}]},
\lambda[\{t\},\ V],\ \lambda\Big[\{t\},\ \mathtt{C1}\left(-\frac{1}{2\ \mathtt{C1}\ \mathtt{L1}}\left(\mathtt{c1\$7}\ e^{-\frac{\left(\mathtt{C1}\ \mathtt{R1}+\mathtt{C1}\ \mathtt{R2}-\sqrt{-4\ \mathtt{C1}\ \mathtt{L1}+(-\mathtt{C1}\ \mathtt{R1}-\mathtt{C1}\ \mathtt{R2})^2}\right) t}{2\ \mathtt{C1}\ \mathtt{L1}}}
\left(\mathtt{C1}\ \mathtt{R1}+\mathtt{C1}\ \mathtt{R2}-\sqrt{-4\ \mathtt{C1}\ \mathtt{L1}+(-\mathtt{C1}\ \mathtt{R1}-\mathtt{C1}\ \mathtt{R2})^2}\right)\right) -
\frac{1}{2\ \mathtt{C1}\ \mathtt{L1}}\left(\mathtt{c1\$8}\ e^{-\frac{\left(\mathtt{C1}\ \mathtt{R1}+\mathtt{C1}\ \mathtt{R2}+\sqrt{-4\ \mathtt{C1}\ \mathtt{L1}+(-\mathtt{C1}\ \mathtt{R1}-\mathtt{C1}\ \mathtt{R2})^2}\right) t}{2\ \mathtt{C1}\ \mathtt{L1}}}
\left(\mathtt{C1}\ \mathtt{R1}+\mathtt{C1}\ \mathtt{R2}+\sqrt{-4\ \mathtt{C1}\ \mathtt{L1}+(-\mathtt{C1}\ \mathtt{R1}-\mathtt{C1}\ \mathtt{R2})^2}\right)\right)\right)\Big]\Big]\Big\}\Big\}
$$

Chapter 9

Conclusion

This thesis introduces a scheme for concurrent constraint functional logic programming and describes an instance which is a distributed implementation in a network of computers. The scheme is based on an operational semantics which combines the principles of cooperative constraint solving over a constraint domain with the principle of higher-order lazy narrowing for conditional pattern rewrite systems.

Higher-order constructs are a desirable feature in functional logic programming and, by extension, in constraint functional logic programming too. Therefore, a large part of the thesis was concerned with the design of a powerful higher-order lazy narrowing calculus. For the needs of functional logic programming, we addressed two extensions:

- lazy narrowing with ATRSs in applicative term algebras, and
- lazy narrowing with PRSs and conditional PRSs in simply-typed term algebras

and proposed various refinements to make them more efficient.

In the design of our cooperative constraint functional logic programming scheme we took into consideration the second extension of the functional logic programming style, i.e. lazy narrowing with conditional PRSs in simply-typed term algebras.

We want to emphasize that the scheme outlined in this thesis is, by no means, the end of the story. A careful analysis of the properties integrated calculus is still missing, but it is very likely that theoretical results obtained in pure functional logic programming can be generalized to our CFLP model. A very interesting direction of research is to find a good characterization of the notion of precursor in CFLP. Keeping track of the precursors of a subgoal is more difficult because of the possibility to transfer precursors in the constraint store during [cp+]-steps.

The intention of our experimental system `CFLP` is to prove the suitability of the distributed model for cooperative constraint solving described in Sect. 6.4. We didn't focus on the design of an efficient implementation, which is another promising direction of future research.

Bibliography

[AKP93] H. Ait-Kaci and A. Podelski. Towards a Meaning of LIFE. Technical Report PRL-RR-11, PRL, 1993.

[Apt90] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter Formal Models and Semantics, pages 493–574. Elsevier Science Publishers B.V., 1990.

[ASLFRA99] P. Arenas-Sánchez, F.J. López-Fraguas, and M. Rodriguez-Artalejo. Functional plus Logic Programming with Built-in and Symbolic Constraints. In *PPDP'99*, volume 1702 of *LNCS*, pages 152–169. Springer, 1999.

[ASS+88] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *Proceedings of the International Conference on Fifth Generation of Computer Systems*, pages 263–276, 1988.

[Bac95] R. Backhouse. The Calculational Method. In *Special Issue on the Calculational Method, Information Processing Letters*, pages 53–121, 1995.

[Bar84] H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 90. North Holland, second edition, 1984.

[Bir35] G. Birkhoff. On the structure of abstract algebras. In *Proc. Cambr. Philos. Soc. 31*, pages 433–454, 1935.

[Bir87] R. Bird. An Introduction to the Theory of Lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[Bir89] R. Bird. An Constructive Functional Programming. In *STOP Summer School on Constructive Algorithmics*, Abeland, 1989.

[Buc85] B. Buchberger. *Multidimensional Systems Theory*, chapter Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory, pages 184–232. D. Reidel Publishing Company, 1985.

[Col90] A. Colmerauer. Prolog iii reference and users manual. version 1.1. Technical report, Marseilles, 1990.

[dB72] N.G. de Bruijn. *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*, volume 34 of *Indagationes Mathematicae*, pages 381–392. 1972.

[DGP91a] J. Darlington, Y.K. Guo, and H. Pull. A New Perspective on the Integration og Functional and Logic Languages. Technical report, Imperial College, September 1991.

[DGP91b] J. Darlington, Y.K. Guo, and H. Pull. Introducing constraint Functional Logic Programming. Technical report, Imperial College, February 1991.

[DvHS+88] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, December 1988.

[Fay79] M.J. Fay. First-Order Unification in an Equational Theory. In *Proceedings of Workshop on Automated Deduction (CADE'1979)*, pages 161–177. Academic Press, 1979.

[GMGRA96] J.C. Gónzalez-Moreno, M.T. Hortalá Gónzalez, and M. Ródriguez-Artalejo. A Rewriting Logic for Declarative Programming. In *ESOP'96*, volume 1058 of *LNCS*, pages 156–172, Linköping, 1996. Springer.

[GMGRA97] J.C. Gónzalez-Moreno, M.T. Hortalá Gónzalez, and M. Ródriguez-Artalejo. A Higher-Order Rewriting Logic for Functional Logic Programming. In *Proceedings of International Conference on Logic Programming*, pages 153–167, Leuven, 1997. MIT Press.

[GMGRA99] J.C. Gónzalez-Moreno, M.T. Hortalá Gónzalez, and M. Ródriguez-Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[HAK+00] M. Hanus, S. Antoy, H. Kuchen, F.J. López-Fraguas, W. Lux, J.J.M. Navarro, and F. Steiner. Curry: An Integrated Functional Logic Language (Version 0.7 of February 2, 2000). Technical report, 2000.

[Han97] M. Hanus. A Unified Model for Functional and Logic Programming. In *Proc. of the 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 80–93, Paris, 1997.

[HMS99] M. Hamada, A. Middeldorp, and T. Suzuki. Completeness Results for a Lazy Conditional Narrowing Calculus. In *DMTCS/CATS'99*, pages 217–231, Auckland, 1999. Springer-Verlag Singapore.

[Höl89] S. Hölldobler. Foundations of Equational Logic Programming. volume 353 of *LNAI*. Springer-Verlag, 1989.

[Hon92a] H. Hong. CLP(CF): Constraint Logic Programming over Complex Functions. Technical Report 94-09, RISC-Linz, Castle of Hagenberg, Austria, 1992.

[Hon92b] H. Hong. Non-linear Constraints Solving over Real Numbers in Constraint Logic Programming (Introducing RISC-CLP). Technical Report 92-08, RISC-Linz, Castle of Hagenberg, Austria, 1992.

[Hon94] H. Hong. Confluency of Cooperative Constraint Solvers. Technical Report 94-08, RISC-Linz, Castle of Hagenberg, Austria, 1994.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to Combinatorics and λ-Calculus.* Cambridge University Press, 1986.

[HS95] M. Hanus and A. Schwab. Alf user's manual. Technical report, 1995.

[HT99] Z. Hu and M. Takeichi. Calculation Carrying Programs. Technical report, University of Tokyo, September 1999.

[Huè76] G. Huèt. *Résolution d'équations dans les languages d'ordre 1,2, ... ω*. PhD thesis, University Paris-7, 1976.

[Hul80] J.-M. Hullot. Canonical Forms and Unification. In *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *LNCS*, pages 318–334. Springer, 1980.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM symposium on Principles of Programming Languages*, volume 87, pages 111–119, Munich, January 1987. ACM Press.

[JM87] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In J.-L. Lassez, editor, *Proceedings of the 4th ICLP*, volume 87, pages 196–218, Cambridge, MA, January 1987. MIT Press.

[JMSY90] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP($\mathcal{R}$) Language and System. Technical report, T.J. Watson Research Centre, IBM Research Division, Yorktown Heights, 1990.

[Klo90] J.W. Klop. Term Rewriting systems: from Church-Rosser to Knuth-Bendix and beyond. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *LNCS*, pages 350–369, Warwick, 1990.

[Klo92] J.W. Klop. Term rewriting systems. In T.S.E. Maibaum S. Abramsky, D.M. Gabbay, editor, *Handbook of Logic in Computer Science*, volume 2, chapter Formal Models and Semantics, pages 2–116. Elsevier Science Publishers B.V., Oxford University Press, 1992.

[Leu93] H.-F. Leung. *Distributed Constraint Logic Programming*. World Scientific, 1993.

[LF92] F.J. López-Fraguas. A General Scheme for Constraint Functional Logic Programming. In G. Levi H. Kirchner, editor, *Algebraic and Logic Programming*, volume 632 of *LNCS*, pages 213–227, Berlin, 1992.

[LF94] F.J. López-Fraguas. *Programación Funcional y Lógica con Restricciones*. PhD thesis, Univ. Complutense Madrid, 1994. In Spanish.

[LFSH99] F.J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. volume 1631 of *LNCS*, pages 244–247, 1999.

[Loo95] R. Loogen. *Integration funktionaler und logischer Programiersprachen*. Oldenbourg Verlag, 1995.

[Mes89] J. Meseguer. General Logics. Technical Report SRI-CSL-89-5, SRI International, March 1989.

[MG85] J. Meseguer and J. Goguen. Initiality, induction and computability. In Maurice Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541, Amsterdam, the Netherlands, July 1985. Cambridge University Press.

[MH94] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:497–536, 1991.

[MIS99a] M. Marin, T. Ida, and W. Schreiner. A Distributed System for Solving Equational Constraints Based on Lazy Narrowing Calculi. In *JSSST Workshop on Programming and Programming Languages (PPL'99)*, March 1999.

[MIS99b] M. Marin, T. Ida, and W. Schreiner. CFLP: a Mathematica Implementation of a Distributed Constraint Solving System. In *Third International Mathematical Symposium (IMS'99)*, Hagenberg, Austria, August 23-25 1999. Computational Mechanics Publications, WIT Press, Southampton, UK.

[MIS99c] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In A. Middeldorp and T. Sato, editors, *FLOPS'99*, volume 1722 of *LNCS*, pages 225–240. Springer-Verlag, 1999.

[Miy99] C. Miyaji. MathLink*: Network Programming using* Mathematica. Cambridge University Press, 1999.

[MMIY99] M. Marin, A. Middeldorp, T. Ida, and T. Yanagi. LNCA: A Lazy Narrowing Calculus for Applicative Term Rewriting

Systems. Technical Report ISE-TR-99-158, Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Japan, 1999.

[MN86] D.A. Miller and G. Nadathur. Higher-order logic programming. In *3rd International Conference on Logic Programming*, pages 448–462, London, March 1986.

[MO98] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[MOI96] A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.

[Mon96] E. Monfroy. *Solver Collaboration for Constraint Logic Programming*. PhD thesis, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine, 1996.

[MR94] P. Marti and M. Rueher. A Cooperative Scheme for Solving Constraints over The Reals. In H. Hong, editor, *Proceedings of PASCO'94: First Parallel Symbolic Computation Symposium*, volume 5, pages 284–293. World Scientific, 1994.

[MR95] P. Marti and M. Rueher. A Distributed Cooperating Constraint Solving System. *International Journal on Artificial Intelligence Tools*, 40(1&2):93–113, 1995.

[MS98] M. Marin and W. Schreiner. CFLP: A Distributed Constraint Solving System for Functional Logic Programming. In P. Kacsuk and G. Kotsis, editors, *DAPSYS'98 Workshop on Distributed and Parallel Systems*, pages 133–136, Budapest, Hungary, September 28-30 1998.

[Nai91] L. Naish. Adding equations to NU-Prolog. Number 528 in LNCS, pages 15–26, Passau, Germany, 1991.

[NI95] K. Nakahara and T. Ida. A complete narrowing calculus for higher-order functional logic programming. In *Proceedings of Seventh International Conference on Programming Languages: Implementations, Logics, and Programming 95 (PLILP'95)*, volume 982 of *LNCS*, pages 97–114, 1995.

[Nip91] T. Nipkow. Higher-order critical pairs. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 342–349, Amsterdam, the Netherlands, July 1991. IEEE Computer Society Press.

[Nip93] T. Nipkow. Functional unification of higher-order patterns. In *Proceedings of 8th IEEE Symposium on Logic in Computer Science*, pages 64–74, 1993.

[NP98] T. Nipkow and C. Prehofer. Higher-order rewriting and equational reasoning. In P. Schmitt W. Bibel, editor, *Automated Deduction - A Basis for Applications*, volume 1, chapter Formal Models and Semantics, pages 399–430. Kluwer, 1998.

[Pie91] B.C. Pierce. *Basic Category Theory for Computer Scientists.* Foundations of Computing. MIT Press, 1991.

[Pre98] C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming.* Foundations of Computing. Birkäuser Boston, 1998.

[Rue95] M. Rueher. An architecture for cooperating constraint solvers on reals. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*, pages 231–250. Springer-Verlag, 1995.

[SG89] W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. volume 29 of *Journal of Logic Programming*, pages 17–64. Elsevier, 1996.

[Smo95] G. Smolka. The oz programming model. Computer Science Today. Springer, 1995.

[SNI97] T. Suzuki, K. Nakagawa, and T. Ida. Higher-Order Lazy Narrowing Calculus: A Computation Model for a Higher-order Functional Logic Language. In *Proceedings of Sixth International Joint Conference, ALP '97 - HOA '97*, volume 1298 of *LNCS*, pages 99–113, Southampton, 1997.

[Sny91] W. Snyder. *A Proof Theory for General Unification.* Birkhäuser, 1991.

[Suz95] T. Suzuki. Completeness of Narrowing for Orthogonal Conditional Rewrite Systems. In *Proceedings of Fuji International Workshop on Functional and Programming*, pages 63–77, 1995.

[Suz96] T. Suzuki. Standardization Theorem Revisited. In *Proceedings of 5th International Conference, ALP'96*, volume 1139 of *LNCS*, pages 122–134, 1996.

[vO94] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1994.

[vO96] V. van Oostrom. Higher-order Families. In *International Conference on Rewriting Techniques and Applications*, LNCS, 1996.

[Wol93] D.A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[Wol96] S. Wolfram. *The Mathematica Book*. Third Edition. Wolfram Media and Cambridge University Press, 1996.