A Lazy Narrowing Calculus for Declarative Constraint Programming

F. J. López-Fraguas
fraguas@sip.ucm.es

M. Rodríguez-Artalejo
mario@sip.ucm.es

Rafael del Vado Vírseda
rdelvado@sip.ucm.es

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
Avda. Complutense s/n, 28040 Madrid

ABSTRACT
The new generic scheme $CFLP(D)$ has been recently proposed in [24] as a logical and semantic framework for lazy constraint functional logic programming over a parametrically given constraint domain $D$. In this paper we extend such framework with a suitable operational semantics, which relies on a new constrained lazy narrowing calculus for goal solving parameterized by a constraint solver over the given domain $D$. This new calculus is sound and strongly complete w.r.t. the declarative semantics of $CFLP(D)$ programs, which was formalized in [24] by means of a Constraint Rewriting Logic $CRW(L(D))$.

Categories and Subject Descriptors
D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.1.6 [Programming Techniques]: Logic Programming; D.3.2 [Programming Languages]: Language Classifications—Constraint and logic languages; D.3.3 [Programming Languages]: Language Constructs and Features—Constraints

General Terms
Algorithms, Languages, Performance, Theory

Keywords
Functional logic programming languages, constraint logic programming, rewrite systems, narrowing, constraint solvers

1. INTRODUCTION
The idea of Constraint Functional Logic Programming arose around 1990 as an attempt to combine two lines of research in declarative programming, namely Constraint Logic Programming and Functional Logic Programming.

Constraint logic programming was started by a seminal paper published by J. Jaffar and J.L. Lassez in 1987 [16], where the $CLP$ scheme was first introduced. The aim of the scheme was to define a family of constraint logic programming languages $CLP(D)$ parameterized by a constraint domain $D$, in such a way that the well established results on the declarative and operational semantics of logic programs [20, 1] could be lifted to all the $CLP(D)$ languages in an elegant and uniform way. The best updated presentation of the classical $CLP$ semantics can be found in [18]. In the course of time, $CLP$ has become a very successful programming paradigm, supporting a clean combination of logic programming and domain-specific methods for constraint satisfaction, simplification and optimization, and leading to practical applications in various fields [32, 17, 28].

On the other hand, functional logic programming refers to a line of research started in the 1980s and aiming at the integration of the best features of functional programming and logic programming. As far as we know, the first attempt to combine functional and logic languages was done by J.A. Robinson and E.E. Sibert when proposing the language LOGLISP [30]. Some other early proposals for the design of functional + logic languages are described in [9]. A good survey of the operational principles and implementation techniques used for the integration of functions into logic programming can be found in [14]. Narrowing, a natural combination of rewriting and unification, originally proposed as a theorem proving tool, has been used as a goal solving mechanism in functional logic languages such as Curry [15] and TOY [22]. Under various more or less restrictive conditions, several narrowing strategies are known to be complete for goal solving [14, 29].

To our best knowledge, the first attempt of combining constraint logic programming and functional logic programming was the $CFLP(D)$ scheme proposed by J. Darlington, Y.K. Guo and H. Pull [8]. The idea behind this approach can be described by the equation $CFLP(D) = CLP(FP(D))$, intended to mean that a $CFLP$ language over the constraint domain $D$ is viewed as a $CLP$ language over an extended constraint domain $FP(D)$ whose constraints include equations between expressions involving user defined functions, to be solved by narrowing.

The $CFLP(D)$ scheme proposed by F.J. López-Fraguas in [21] aimed at providing a declarative semantics such that
CLP(D) programs could be formally understood as a particular case of CFLP(D) programs. In the classical approach to CLP semantics a constraint domain is viewed as a first order structure D, and constraints are viewed as first order formulas that can be interpreted in D. In [21] programs were built as sets of constrained rewrite rules. In order to support a lazy semantics for the user defined functions, constraint domains D were formalized as continuous structures, with a Scott domain [13] as carrier, and a continuous interpretation of function and predicate symbols. The resulting semantics had many pleasant properties, but also some limitations. In particular, defined functions had to be first order and deterministic, and the use of patterns in function definitions had to be simulated by means of special constraints.

More recently, yet another CFLP scheme has been proposed in the PhD Thesis of M. Marin [25]. This approach introduces CFLP(D, S, L), a family of languages parameterized by a constraint domain D, a strategy S which defines the cooperation of several constraint solvers over D, and a constraint lazy narrowing calculus L for solving constraints involving functions defined by user given constrained rewrite rules. This approach relies on solid work on higher-order lazy narrowing calculi and has been implemented on top of Mathematica [26, 27]. Its main limitation from our viewpoint is the lack of declarative semantics.

In a recent work [24] we have proposed a new generic scheme CFLP(D), intended as a logical and semantic framework for lazy Constraint Functional Logic Programming over parametrically given constraint domains D, which provides a clean and rigorous declarative semantics for CFLP(D) languages as in the CFLP(D) scheme but overcomes the limitations of our older CFLP(D) scheme [21]. CFLP(D) programs are presented as sets of constrained rewrite rules that define the behaviour of possibly higher-order and/or non-deterministic lazy functions over D. The main novelties in [24] were a formalization of constraint domains for CFLP, a new notion of interpretation for CFLP(D) programs, and a new Constraint Rewriting Logic CRW(L(D)) parameterized by a constraint domain, which provides a logical characterization of program semantics.

Our aim in this paper is to formalize an operational semantics for the new generic scheme CFLP(D) proposed in [24]. We present a lazy constrained narrowing calculus CLNC(D) for solving goals for CFLP(D) programs, which can be proved sound and strongly complete w.r.t. CRW(L(D)) semantics. These properties qualify CLNC(D) as a convenient computation mechanism for declarative constraint programming languages.

The reader of this paper is assumed to have some knowledge on the foundations of logic programming [20, 1] and term rewriting [5]. The rest of the paper is organized as follows. The next section is devoted to summarize the presentation of the CFLP(D) scheme [24] and to formalize the notion of a constraint solver over a given constraint domain. In Section 3 we give a formal presentation of the calculus CLNC(D). We discuss the soundness and completeness results in Section 4. Finally, some conclusions and plans for future work are drawn in Section 5.

2. THE GENERIC SCHEME CFLP(D)

In this section we introduce the main features of the CFLP(D) scheme [24], as a basis for the constraint narrowing calculus CLNC(D) presented in the rest of the paper.

2.1 Applicative expressions, patterns and subtitutions

We briefly introduce the syntax of applicative expressions and patterns, which is needed for understanding the construction of constraint domains and constraint solvers.

We assume a universal signature Σ = \( \langle DC, FS \rangle \), where \( \text{DC} = \bigcup_{n \in \mathbb{N}} \text{DC}^n \) and \( \text{FS} = \bigcup_{n \in \mathbb{N}} \text{FS}^n \) are families of countably infinite and mutually disjoint sets of data constructors resp. evaluable function symbols each one with an associated arity. We write \( \Sigma_\perp \) for the result of extending \( \text{DC}^0 \) with the special symbol \( \perp \), intended to denote an undefined data value. As notational conventions, we use \( c, d \in \text{DC} \), \( f, g \in \text{FS} \) and \( h \in \text{DC} \cup \text{FS} \), and we define the arity of \( h \in \text{DC}^n \cup \text{FS}^n \) as \( \text{ar}(h) = n \). We also assume that \( \text{DC}^0 \) includes the three constants true, false and success, which are useful for representing the results returned by various primitive functions. Next we assume a countably infinite set \( V \) of variables \( X, Y, \ldots \) and a set \( U \) of urelements \( u, v, \ldots \), mutually disjoint and disjoin from \( \Sigma_\perp \). Urelements are intended to represent some domain specific set of values, as e.g. the set \( \mathbb{R} \) of real numbers used in the well-known CLP language CLP(R) [19]. Partial expressions \( e \in \text{Exp}_\perp(U) \) have the following syntax:

\[
e ::= \perp \mid u \mid X \mid h \mid (e_1 e_2)
\]

where \( u \in U \), \( X \in V \), \( h \in \text{DC} \cup \text{FS} \). These expressions are usually called applicative, because \( (e_1 e_2) \) stands for the application operation (represented as juxtaposition) which applies the function denoted by \( e \) to the argument denoted by \( e_1 \). Applicative syntax is common in higher order functional languages. The usual first order syntax for expressions can be translated to applicative syntax by means of so-called curried notation. For instance, \( f(X,g(Y)) \) becomes \( (f \; X \; g \; Y) \). Following a usual convention, we assume that expression associates to the left, and we use the notation \( (e \; \pi) \) to abbreviate \( (e \; e_2 \ldots e_n) \). The set of variables occurring in \( e \) is written \( \text{var}(e) \). An expression \( e \) is called linear if there is no \( X \in \text{var}(e) \) having more than one occurrence in \( e \). The following classification of expressions is also useful: \( (X\; \pi_m) \), with \( X \in V \) and \( m \geq 0 \), is called a flexible expression, while \( u \in U \) and \((h \; \pi_n)\) with \( h \in \text{DC} \cup \text{FS} \) are called rigid expressions. Moreover, a rigid expression \( (h \; \pi_n) \) is called active iff \( h \in \text{FS} \) and \( m \geq \text{ar}(h) \), and passive otherwise. Intuitively, reducing an expression at the root makes sense only if the expression is active. Some interesting subsets of \( \text{Exp}_\perp(U) \) are: \( \text{GEExp}_\perp(U) \), the set of the ground expressions \( e \) such that \( \text{var}(e) = \emptyset \); \( \text{Exp}_\perp(U) \), the set of the total expressions \( e \) with no occurrences of \( \perp \); \( \text{GEExp}_\perp(U) \), the set of the ground and total expressions \( \text{GEExp}_\perp(U) \cap \text{Exp}_\perp(U) \). Another important subclass of expressions is the set of partial patterns \( s, t \in \text{Pat}_\perp(U) \), whose syntax is defined as follows:

\[
t ::= \perp \mid u \mid X \mid c \; \pi_m | \pi_m
\]

where \( u \in U \), \( X \in V \), \( c \in \text{DC}^n \), \( m \leq n \), \( f \in \text{FS}^n \), \( m < n \). Note that expressions \( (\pi_m) \) with \( f \in \text{FS}^n \), \( m \geq n \) are not allowed as patterns, because they are potentially evaluable using a primitive or user given definition for function \( f \). Patterns of the form \( (\pi_m) \) with \( f \in \text{FS}^n \), \( m < n \) have been used in functional logic programming [12] as a convenient representation of higher order values. The subsets \( \text{Pat}_\perp(U) \), \( \text{GPat}_\perp(U) \), \( \text{GPat}_\perp(U) \subseteq \text{Pat}_\perp(U) \) consisting of the total, ground and total and ground patterns, respectively, are
defined in the natural way. Following the spirit of denotational semantics [13], we view \( Pat_\bot(U) \) as the set of finite elements of a semantic domain, and we define the information ordering \( \sqsubseteq \) as the least partial ordering over \( Pat_\bot \) satisfying the following properties: \( \sqsubseteq \sqsubseteq \) for all \( t \in Pat_\bot(U) \), and \((ht_m) \sqsubseteq (ht'_m)\) whenever these two expressions are patterns and \( t_i \sqsubseteq t'_i \) for all \( 1 \leq i \leq m \). The sequence \( t_m \sqsubseteq \) will be understood as meaning that \( t_i \subseteq t'_i \) for all \( 1 \leq i \leq m \).

Note that a pattern \( t \in Pat_\bot(U) \) is maximal w.r.t. the information ordering iff it is a total pattern, i.e. \( t \in \mathcal{P}(\bot) \). For some purposes it is useful to extend the information ordering to the set of all partial expressions. This extension is simply defined as the least partial ordering over \( Exp_\bot(U) \) which verifies \( \sqsubseteq \subseteq \) for all \( e \in Exp_\bot(U) \), and \((e_1 \sqsubseteq e'_1) \) whenever \( e \subseteq e' \) and \( e_1 \subseteq e'_1 \). As usual, we define substitutions \( \sigma \in Sub_\bot(U) \) as mappings \( \sigma : V \rightarrow Pat_\bot(U) \) extended to \( \sigma : Exp_\bot(U) \rightarrow Exp_\bot(U) \) in the natural way. Similarly, we consider total substitutions \( \sigma \in Sub(U) \) given by mappings \( \sigma : V \rightarrow Pat(U) \), ground substitutions \( \sigma \in GSub(U) \) given by mappings \( \sigma : V \rightarrow GPat(U) \), and ground total substitutions \( \sigma \in GSub(U) \) given by mappings \( \sigma : V \rightarrow GPat(U) \).

By convention, we write \( \varepsilon \) for the identity substitution, \( \sigma \theta \) instead of \( \sigma(e\theta) \) for the composition of \( \sigma \) and \( \theta \), such that \( \varepsilon(e\theta) = (e\varepsilon)\theta \) for any \( e \in Exp_\bot(U) \). We define the domain and the variable range of a substitution in the usual way, namely: \( dom(\sigma) = \{ X \in V \mid \sigma(X) \neq X \} \) and \( ran(\sigma) = \bigcup_{e \in dom(\sigma)} \var(\sigma(X)) \). As usual, a substitution \( \sigma \) such that \( dom(\sigma) \cap ran(\sigma) = 0 \) is called idempotent. For any set of variables \( X \subseteq V \) we define the restriction \( \sigma | X \) as the substitution \( \sigma' \) such that \( dom(\sigma') = X \) and \( \sigma'(X) = \sigma(X) \) for all \( X \in X \). We use the notation \( \sigma = \tau \theta \) to indicate that \( \sigma | X = \tau | X \), and we abbreviate \( \sigma = \varepsilon(X) \theta \) as \( \sigma = \varepsilon X \theta \). Finally, we consider two different ways of comparing given substitutions \( \sigma, \sigma' \in Sub_\bot(U) \), \( \sigma \) is said to be less particular than \( \sigma' \) over \( X \subseteq V \) (in symbols, \( \sigma \leq_X \sigma' \)) iff \( \sigma \theta = X \sigma' \) for some \( \theta \in Sub_U \). The notation \( \sigma \leq \sigma' \) abbreviates \( \forall X. \sigma \leq_X \sigma' \). is said to be less particular than \( \sigma' \) over \( X \subseteq V \) (in symbols, \( \sigma \leq_X \sigma' \)) iff \( \sigma(X) \subseteq \sigma'(X) \) for all \( X \in X \). The notation \( \sigma \leq \sigma' \) abbreviates \( \forall X. \sigma \leq_X \sigma' \).

2.2 Constraints over a given constraint domain

Intuitively, a constraint domain is expected to provide a set of specific data elements, along with certain primitive functions and predicates operating upon them. The following definition extends the notion of constraint domain \( D \) introduced in [24] by adding a constraint solver:

**Definition 1.** Constraint Domains.

1. A constraint signature is any family \( PF = \bigcup_{n \in \mathbb{N}} PF_n \) of primitive function symbols \( p \), each one with an associated arity, such that \( PF_n \subseteq FS_n \) for each \( n \in \mathbb{N} \).
2. A constraint domain of signature \( PF \) is any structure \( D = \langle D_U, \{ p^D \mid p \in PF \}, \text{Sol}_D \rangle \) such that the carrier set \( D_U = GPat_\bot(U) \) coincides with the set of ground patterns for some set of urelements \( U \), the interpretation \( p^D \) of each \( p \in PF_n \) satisfies the following requirements:
   (a) \( p^D \subseteq D_U \times D_U \), which boils down to \( p^D \subseteq D_U \) in the case \( n = 0 \).
   (b) \( p^D \) behaves monotonically in its arguments and antimonotonically in its result; i.e., if \( p^D \ t_1 \rightarrow t_2 \), \( t_1 \subseteq t'_1 \) for all \( 1 \leq i \leq m \). In the sequel, \( t_m \sqsubseteq \) will be understood as meaning that \( t_i \subseteq t'_i \) for all \( 1 \leq i \leq m \).
   (c) \( p^D \) behaves residually in the following sense: whenever \( p^D \ t_1 \rightarrow t' \) and \( t \neq \bot \), there is some total \( t' \in D_U \) such that \( p^D \ t \rightarrow t' \).

and \( Sol^D \) is a constraint solver, whose expected behaviour will be explained in Definition 5 below.

Assuming an arbitrarily fixed constraint domain \( D \) built over a certain set of urelements \( U \), we will now define the syntax of constraints. In the sequel, we will write \( DF = FS \setminus PF \) for the set of user defined function symbols, and \( DF^\bot = FS^\bot \setminus PF^\bot \) for the set of user defined function symbols of arity \( n \). The following definition distinguishes primitive constraints without any active occurrence of defined function symbols, from user defined constraints that can have such occurrences. For the sake of brevity, we sometimes write simply ‘constraints’ instead of ‘user defined constraints’.

**Definition 2. Syntax of Constraints.**

1. **Atomic Primitive Constraints** have the syntactic form \( p t_i \rightarrow ! t \), with \( p \in PF^\bot, t_i \in Pat_U \) for all \( 1 \leq i \leq n \), and \( t \in Pat(U) \). The special constants \( \Diamond \) and \( \Box \) are also atomic primitive constraints.
2. **Primitive Constraints** are built from atomic primitive constraints by means of logical conjunction \( \land \) and existential quantification \( \exists \).
3. **Atomic Constraints** have the syntactic form \( p t_i \rightarrow ! t \), with \( p \in PF^\bot, t_i \in Exp_\bot(U) \) for all \( 1 \leq i \leq n \), and \( t \in Pat(U) \). The special constants \( \Diamond \) and \( \Box \) are also atomic constraints.
4. **Constraints** are built from atomic constraints by means of conjunction \( \land \) and existential quantification \( \exists \).

In the sequel we use the notations: \( PCon_\bot(D) \) for the set of all the primitive constraints \( \pi \) over \( D \) and \( PCon(D) \) for the set of all the total primitive constraints over \( D \), defined as \( \{ \pi \in PCon_\bot(D) \mid \pi \) has no occurrences of \( \bot \} \). We also write \( DCon_\bot(D) \) for the set of all the user defined constraints \( \delta \) over \( D \), as well as \( DCon(D) \) for the subset of \( DCon_\bot(D) \) consisting of total constraints. We reserve the capital letters II resp. \( C \) for sets of primitive resp. user defined constraints, often interpreted as conjunctions. The semantics of primitive constraints depends on the notion of solution, presented in the next definition.

**Definition 3. Solutions of Primitive Constraints.**

1. The set of valuations resp. total valuations over \( D \) is defined as \( Val_\bot(D) = GSub(U) \) resp. \( Val(D) = GSub(U) \).
2. The set of solutions of \( \pi \in PCon_\bot(D) \) is a subset \( Sol^D(\pi) \subseteq Val_\bot(D) \) recursively defined as follows:
   (a) \( Sol^D(\varepsilon) = Val_\bot(D) \) and \( Sol^D(\Box) = \emptyset \).
   (b) \( Sol^D(p t_i \rightarrow ! t) = \{ \eta \in Val_\bot(D) \mid \eta \) is total and \( p^D \ t \eta \rightarrow t \eta \} \).
   (c) \( Sol^D(\pi_1 \land \pi_2) = Sol^D(\pi_1) \cup Sol^D(\pi_2) \).
(d) \( \text{Sol}_D(\exists X.\pi) = \{ \eta \in V a l_\perp(D) \mid \eta' \in \text{Sol}_D(\pi) \text{ for some } \eta' = _{\exists(X)} \eta \} \).

3. The set of solutions of \( \Pi \subseteq P C o n_\perp(D) \) is defined as \( \text{Sol}_D(\Pi) = \bigcap_{\pi \in \Pi} \text{Sol}_D(\pi) \), corresponding to a logical reading of \( \Pi \) as the conjunction of its members. In particular, \( \text{Sol}_D(\emptyset) = V a l_\perp(D) \), corresponding to the logical reading of an empty conjunction as the identically true constraint \( \emptyset \).

Using the notion of solution, some useful semantic notions related to primitive constraints are easily introduced:

**Definition 4. Primitive Semantic Notions.**
Assuming a finite set \( \Pi \subseteq P C o n_\perp(D) \) of primitive constraints, a primitive constraint \( \pi \in P C o n_\perp(D) \), expressions \( e, e' \in E x p_\perp(U) \), patterns \( t_n, t \in P a t_\perp(U) \), and a primitive function symbol \( p \in P F^\perp \), we define:

1. \( \pi \) is called **satisfiable in** \( D \) (in symbols \( S a t_D(\pi) \)) iff \( \text{Sol}_D(\pi) \neq \emptyset \). Otherwise \( \pi \) is called **unsatisfiable** (in symbols \( U n s a t_D(\pi) \)). Anologically for constraint sets.

2. \( \pi \) is a **consequence** of \( \Pi \) in \( D \) (in symbols, \( \Pi \models_D \pi \)) iff \( \text{Sol}_D(\Pi) \subseteq \text{Sol}_D(\pi) \). In particular, \( \Pi \models_D t \) is a consequence of \( \Pi \) in \( D \) (in symbols, \( \Pi \models_D \pi_{t \leftarrow t} \)) iff \( \text{Sol}_D(\Pi) \models_D t_n \rightarrow t \) for all \( t_n \in \text{Sol}_D(\Pi) \).

3. \( e \subseteq e' \) is a consequence of \( \Pi \) in \( D \) (in symbols, \( \Pi \models_D e \subseteq e' \)) iff \( \forall \eta \; \eta \in \text{Sol}_D(\Pi) \).

At this point we can specify the expected behaviour of the constraint solver \( \text{Solve}^D \) introduced in **Definition 1**. The following definition is inspired in \([21, 2, 23]\). Under the constraint domain \( \mathcal{H}_{\text{seq}} \). We consider the constraint domain \( \mathcal{H}_{\text{seq}} \) built over an empty set of urelements and having the strict equality \( \mathsf{seq} \) as its only primitive, interpreted to behave as follows: \( \mathsf{seq}^{\text{seq}} t \rightarrow t \) for all \( t \in G P a t(\emptyset) \); \( \mathsf{seq}^{\text{seq}} t s \rightarrow \text{false} \) for all \( t, s \in G P a t_\perp(\emptyset) \) such that \( t, s \) have no common upper bound \( \mathsf{w.r.t.} \) the information ordering; \( \mathsf{seq}^{\text{seq}} t s \rightarrow \text{false} \) otherwise. In the sequel, \( t = s \) abbreviates \( \mathsf{seq} t s \rightarrow \text{true} \), \( t \neq s \) abbreviates \( \mathsf{seq} t s \rightarrow \text{false} \) and \( \text{Tot}(t) \rightarrow \text{false} \). A possible constraint solver \( \text{Solve}^D \) can be found in **Appendix A**. The specification of solvers for other useful constraint domains is planned as future work.

### 2.3 CFLP(D)-programs

In the sequel we assume an arbitrarily fixed constraint domain \( D \) built over a set of urelements \( U \). In this setting, **CFLP(D)-Programs** are presented as sets of constrained rewrite rules that define the behaviour of possibly higher order and/or non-deterministic lazy functions over \( D \), called **program rules**. More precisely, a program rule \( R \) for \( f \in D^F^\perp \) has the form \( R : f T_n \rightarrow \varnothing \in D C^D \) and is required to satisfy the conditions listed below:

1. The **left-hand side** \( f T_n \) is a linear expression, and for all \( 1 \leq i \leq n, t_i \in P a t(U) \) are total patterns.

2. The **right-hand side** \( r \in E x p(U) \) is a total expression.

3. \( P \) is a finite sequence of so-called **productions** \( e_i \rightarrow s_i \) \((1 \leq i \leq k)\) also intended to be interpreted as conjunction, and fulfilling the following admissibility **conditions**:

   (a) For all \( 1 \leq i \leq k, e_i \in E x p(U) \) is a total expression, \( s_i \in P a t(U) \) is a total linear pattern, and \( \var{v}_i \cap \var{v}(f T_n) = \emptyset \).

   (b) It is possible to reorder the productions of \( P \) in the form \( P \equiv e_1 \rightarrow s_1, \ldots, e_k \rightarrow s_k \) where \( \var{v}(e_i) \cap \var{v}(s_j) = \emptyset \) for all \( 1 \leq i \leq j \leq k \).

   (c) For all \( 1 \leq i < j \leq k, \var{v}(s_i) \cap \var{v}(s_j) = \emptyset \).

4. \( C \subseteq D C o n(D) \) is a finite set of total constraints, intended to be interpreted as conjunction, and possibly including occurrences of defined function symbols.

A program rule such that \( P \) and \( C \) are both empty can be abbreviated as \( f T_n \rightarrow r \). We note that an equivalent formulation for the admissibility condition 3(b) can be obtained by defining the production relation \( X \rightarrow_P Y \) iff there is some \( 1 \leq i \leq k \) such that \( X \in \var{v}(e_i) \) and \( Y \in \var{v}(s_i) \), and requiring that the transitive closure of \( \rightarrow_P \) must be irreflexive, or equivalently, a strict partial order.

**Example 2.** The following **CFLP(D)-Program** can be used over the constraint domain \( \mathcal{H}_{\text{seq}} \) presented in **Example 1**.
We use the constructors $0 \in DC^0$, $s \in DC^1$, a constructor for pairs (i.e. $(e_1, e_2)$) denotes the pair of a first element $e_1$ and a second element $e_2$) and a Prolog-like syntax for list constructors (i.e. $[\ ]$ denotes the empty list and $[X|Xs]$ denotes a non-empty list consisting of a first element $X$ and a remaining list $Xs$). More examples of CFLP($D$)-programs can be found in [24].

$$\text{from } N \rightarrow [N|\text{from}(s, N)]$$

null $[X|Xs] \rightarrow s \rightarrow 0$

split $([]) \rightarrow ([], [])$

split $[X|Xs] \rightarrow \text{case } R X \text{ Ys } Zs \rightarrow \text{ split } Xs \rightarrow (Ys, Zs)$

$$\text{seq } X s \rightarrow t \rightarrow 0 \rightarrow 1 R$$

case $X Ys Zs \rightarrow (([X|Ys], Zs))$

case $\text{false } X Ys Zs \rightarrow (YS, [X|Zs])$

2.4 The Constraint Rewriting Logic CRWL($D$)

The Constraint Rewriting Logic CRWL($D$), parameterized by a constraint domain $D$, was introduced in [24] in order to provide a declarative semantic for CFLP($D$)-programs. In order to define this logic, we must first introduce some preliminary notions about the constrained statements that we intend to derive from a given CFLP($D$)-program.

**Definition 6. Constrained Statements and $D$-entailment.** Let $D$ be any fixed constraint domain over a set of urelements $U$. In what follows we assume partial patterns $t, t_i \in \text{Pat}_1(U)$, partial expressions $e, e_i \in \text{Exp}_1(U)$, and a finite set $\Pi \subseteq \text{PCom}_1(D)$ of primitive constraints.

1. We consider two possible kinds of constrained statements (c-statements):

   (a) c-productions $e \rightarrow t \Leftarrow t$. A c-production is called trivial if $t$ is unifiable with $\bot$.

   (b) c-atoms $p_{e, e_i} \rightarrow t \Leftarrow \Pi$, with $p \in PH^n$ and $t$ total. A c-atom is called trivial if $\text{Unsat}(\Pi)$.

2. Given two c-statements $\varphi$ and $\varphi'$, we say that $\varphi$ $D$-entails $\varphi'$ (in symbols, $\varphi \vdash_D \varphi'$) if one of the two following cases holds:

   (a) $\varphi = e \rightarrow t \Leftarrow \Pi$ and $\varphi' = e' \rightarrow t' \Leftarrow \Pi'$, and there is some $\sigma \in Sub_1(U)$ such that $\Pi' \vdash_D \Pi \sigma$, $\Pi' \vdash_D e' \equiv e\sigma$, and $\Pi' \vdash_D t' \subseteq t\sigma$.

   (b) $\varphi = p_{e, e_i} \rightarrow t \Leftarrow \Pi$, $\varphi' = p_{e, e_i}^{\sigma} \rightarrow t' \Leftarrow \Pi'$, and there is some $\sigma \in Sub_1(U)$ such that $\Pi' \vdash_D \Pi \sigma$, $\Pi' \vdash_D p_{e, e_i}^{\sigma} \equiv (p_{e, e_i})\sigma$, and $\Pi' \vdash_D t' \subseteq t\sigma$.

The next definition assumes a given CFLP($D$)-program $P$ and uses the notation $[P]_\Pi$ for the set $\{R \theta | R \in P, \theta \in Sub_1(U)\}$ consisting of all the possible instances of the function defining rules belonging to $P$. The purpose of the calculus is to infer the semantic validity of arbitrary c-statements from the program rules in $P$.

**Definition 7. Constrained Rewriting Calculus.** We write $P \vdash_D \varphi$ to indicate that the c-statement $\varphi$ can be derived from $P$ in the constrained rewriting calculus CRWL($D$) using the inference rules given in Figure 1. Some of these rules depend on the semantic notions given in Definition 4

1. Note that $\Pi' \vdash_D t' \subseteq t\sigma$ would be wrong, because $\rightarrow !$ behaves monotonically both in its arguments and in its result. See Definition 3 (b).

and the following semantic notion for productions: $pT_n \rightarrow t$ is a consequence of $\Pi$ in $D$ (in symbols, $\Pi \models_D pT_n \rightarrow t$) iff $p^2T_n \sigma \eta \rightarrow t\eta$ holds for all $\eta \in \text{Sol}_2(\Pi)$.

By convention, we agree that no inference rule of the constrained rewriting calculus is applied in case that some textually previous rule can be used. In particular, no rule except TI can be used to infer a trivial c-statement, and SP is not applied whenever RH is applicable. Moreover, we also agree that the premise $P \square C \Leftarrow \Pi$ in rule DF$_*P$ must be understood as a shorthand for several premises $\alpha \Leftarrow \Pi$, one for each atomic statement $\alpha$ occurring in $P \square C$.

Any derivation in the constrained rewriting calculus can be represented as a proof tree whose nodes are labelled by c-statements, where each node has been inferred from its children by means of the inference rules. In the sequel, we will use the following notations:

1. $T$ is called an easy proof tree iff $T$ makes no use of the inference rules DF$_P$, PF and AC.

2. $| T |$ denotes the restricted size of the proof tree $T$, defined as the number of nodes in $T$ which are inferred with some of the rules DF$_P$, PF or AC. Obviously, $| T | = 0$ iff $T$ is an easy proof tree.

3. $\vdash_D \varphi$ indicates that $P \vdash_D \varphi$ is witnessed by the proof tree $T$.

The next state result state two useful properties of the constrained rewriting calculus. The (rather technical) proof and other properties of CRWL($D$) can be found in [24].

**Lemma 1. Properties of the CRWL($D$) Calculus.**

1. Approximation Property: For any $e \in \text{Exp}_1(U)$, $t \in \text{Pat}_1(U)$: $\Pi \models_D e \equiv t$ iff there is some easy proof tree $T$ such that $T : \vdash_D e \rightarrow t \Leftarrow \Pi$ (derivation from empty program).

2. Entailment Property: $T : \vdash_D \varphi$ and $\varphi \supseteq \varphi'$ implies $T' : \vdash_D \varphi'$ for some proof tree $T'$ such that $| T' | \leq | T |$.

Correctness results relating CRWL($D$)-derivability to a suitable model-theoretic semantics are also given in [24]. More precisely, as argued in [24], CRWL($D$) is sound and complete w.r.t. strong semantics, and sound and ground complete w.r.t. weak semantics, two different classes of semantics.

3. THE CLNC($D$) CALCULUS

This section presents a new Constrained Lazy Narrowing Calculus over a parametrically given constraint domain $D$ (shortly, CLNC($D$)) for solving CFLP($D$)-goals, borrowing ideas and techniques from previous lazy narrowing calculi for FLP [11, 12, 31] and CFLP [21, 2, 3] languages. We give first a precise definition for the class of admissible goals, answers and solutions we are going to work with.

**Definition 8.** A goal for a given CFLP($D$)-program must have the form $G \models_D \square T$. P $\square T \square S \square 0$, where the symbol $\square$ must be interpreted as conjunction, and:

- $\text{evar}(G) =_{e, f} T$ is the set of so-called existential variables of the goal $G$.
- These are intermediate variables, whose bindings in a solution may be partial patterns.
- $\text{fvar}(G) =_{e, f} \text{var}(G) - \text{evar}(G)$ is the set of so-called free variables of the goal $G$. 
Additionally, any admissible goal must satisfy the following admissibility conditions, called goal invariants:

- **LN** Each produced variable is produced only once, i.e. the tuple \( t_1, \ldots, t_n \) must be linear.
- **EX** All the produced variables must be existential, i.e. \( \text{pvar}(P) \subseteq \text{evar}(G) \).
- **NC** The transitive closure of the production relation \( \Rightarrow_P \) (given in Subsection 2.3) must be irreflexive, or equivalently, a strict partial order.
- **SL** No produced variable enters the answer substitution, i.e. \( \text{var}(\sigma) \cap \text{pvar}(P) = \emptyset \).

Similarly to [11, 12, 31], CLNC(D) uses a notion of demanded variable to deal with lazy evaluation, but now w.r.t. a constraint store. Intuitively, productions \( e \rightarrow X \) in \( G \), where \( e \) is not a pattern, do not propagate the binding \( \{X \rightarrow e\} \). Instead, evaluation of \( e \) must be triggered, provided that \( X \) is demanded in \( G \). The result will be shared by all the occurrences of \( X \).

**Definition 9.** Let \( G \equiv \exists \mathcal{T}. \ P \sqcup C \sqcup \sigma \sqcup S \) be an admissible goal for a given CFLP(D)-program and \( X \in \text{var}(G) \).

We say that \( X \) is a demanded variable in \( G \) if \( X \in \text{dvar}_P(S) \) (see Definition 5) or there exists some production \( (X_{\tau_k} \rightarrow t) \in P \) such that, either \( t \notin V \) or else \( k > 0 \) and \( t \) is a demanded variable in \( G \). We write \( \text{dvar}_P(G) \) (or more precisely \( \text{dvar}_P(P \sqcup S) \)) for the set of demanded variables in the goal \( G \).

<table>
<thead>
<tr>
<th>TI Trivial Inference</th>
<th>( \varphi ) if ( \varphi ) is a trivial c-statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR Restricted Reflexivity</td>
<td>( t \rightarrow t \equiv \Pi ) if ( t \in U \cup V ).</td>
</tr>
<tr>
<td>SP Simple Production</td>
<td>( s \rightarrow t \equiv \Pi ) if ( s \in \text{Pat}_\bot(U) ), ( s \in V ) or ( t \in V ), and ( \Pi \vdash_D s \supseteq t ).</td>
</tr>
<tr>
<td>DC Decomposition</td>
<td>( e_1 \rightarrow t_1 \equiv \Pi, \ldots, e_m \rightarrow t_m \equiv \Pi ) if ( h\sigma_m ) is passive.</td>
</tr>
<tr>
<td>IR Inner Reduction</td>
<td>( h\sigma_m \rightarrow X \equiv \Pi ) if ( h\sigma_m ) is passive but not a pattern.</td>
</tr>
<tr>
<td>PF Primitive Function</td>
<td>( e_1 \rightarrow t_1 \equiv \Pi, \ldots, e_n \rightarrow t_n \equiv \Pi ) if ( p \in PF^n ), ( t_i \in \text{Pat}_\bot(U) ) for each ( 1 \leq i \leq n ), and ( \Pi \vdash_D p\tilde{t}_n \rightarrow t ).</td>
</tr>
<tr>
<td>DF ( \mathcal{P} )-Defined Function</td>
<td>( e_1 \rightarrow t_1 \equiv \Pi, \ldots, e_n \rightarrow t_n \equiv \Pi ) if ( f \in \mathcal{D}^n ) ((k &gt; 0)), ((f\tilde{t}<em>n \rightarrow r \in \mathcal{P} \sqcup C) \in [\mathcal{P}]</em>\bot ), ( s \in \text{Pat}_\bot(U) ).</td>
</tr>
<tr>
<td>AC Atomic Constraint</td>
<td>( e_1 \rightarrow t_1 \equiv \Pi, \ldots, e_n \rightarrow t_n \equiv \Pi ) if ( p \in PF^n ), ( t_i \in \text{Pat}_\bot(U) ) for each ( 1 \leq i \leq n ), and ( \Pi \vdash_D p\tilde{t}_n \rightarrow \neg t ).</td>
</tr>
</tbody>
</table>

**Figure 1:** Rules for CRWL(D)-derivability
DC Decomposition \[ \exists \mathcal{T}. \; h_\sigma m \rightarrow h_\tau m, \; P \Box C \Box S \Box \sigma \vdash_{DC} \exists \mathcal{T}. \; e_\tau m \rightarrow \tau m, \; P \Box C \Box S \Box \sigma \] if \( h_\sigma m \) is passive.

SP Simple Production \[ \exists \mathcal{T}. \; X \rightarrow t, \; P \Box C \Box S \Box \sigma \vdash_{SP} \exists \mathcal{T}. \; (P \Box C \Box S)\sigma_0 \square \sigma_0 \] if \( t \notin \mathcal{T} \) and \( \sigma = \{ X \rightarrow t \} \).

IM Imitation \[ \exists \mathcal{T}, \; h_\sigma m \rightarrow X, \; P \Box C \Box S \Box \sigma \vdash_{IM} \exists \mathcal{T}. \; (e_\tau m \rightarrow \tau m, \; P \Box C \Box S)\sigma_0 \square \sigma \] if \( h_\sigma m \notin \text{Pat}(\mathcal{T}) \) is passive, \( X \in \text{var}(P \Box C \Box S \Box \sigma) \) and \( \sigma_0 = \{ X \rightarrow hX \} \) with \( X_\mathcal{T} \) new variables such that \( hX \mathcal{T} \in \text{Pat}(\mathcal{T}) \).

EL Elimination \[ \exists \mathcal{T}. \; e \rightarrow X, \; P \Box C \Box S \Box \sigma \vdash_{EL} \exists \mathcal{T}. \; P \Box C \Box S \Box \sigma \] if \( X \notin \text{var}(P \Box C \Box S \Box \sigma) \).

PF Primitive Function \[ \exists \mathcal{T}. \; p_\tau n \rightarrow t, \; P \Box C \Box S \Box \sigma \vdash_{PF} \exists \mathcal{T}, \; e_\tau m \rightarrow X_\mathcal{T}, \; P \Box C \Box p_\tau n \rightarrow ! t, \; S \Box \sigma \] if \( p \in \text{PF} \), \( t \notin \mathcal{T} \) or \( t \in \text{var}(P \Box S) \), and \( X_\mathcal{T} \) are new variables \((0 \leq q \leq n) \) is the number of \( e_i \notin \text{Pat}(\mathcal{T}) \) such that \( t_i \equiv X_j (0 \leq j \leq q) \) if \( e_i \notin \text{Pat}_i(\mathcal{T}) \) and \( t_i \equiv e_i \) otherwise for each \( 1 \leq i \leq n \).

DF Defined Function \[ \exists \mathcal{T}. \; p_\tau n \rightarrow t, \; P \Box C \Box S \Box \sigma \vdash_{DF} \exists \mathcal{T}, \; e_\tau m \rightarrow \tau m, \; P \Box C \Box S \Box \sigma \] if \( f \in \text{DF}^n (k > 0), \) \( t \notin \mathcal{T} \) or \( t \in \text{var}(P \Box S) \), \( \tau \) is a fresh variant of a rule \( p \) with \( \mathcal{T} = \text{var}(R) \) and \( X \) new variables.

FV Functional Variable \[ \exists \mathcal{T}. \; F_\tau q \rightarrow t, \; P \Box C \Box S \Box \sigma \vdash_{FV} \exists \mathcal{T}, \; (hX_\mathcal{T} p_\tau n \rightarrow t, \; P \Box C \Box S)\sigma_0 \square \sigma_0 \] if \( \{ F \notin \text{var}(P) \}, \; q > 0, \; t \notin \mathcal{T} \) or \( t \in \text{var}(P \Box S) \), \( \sigma_0 = \{ F \rightarrow hX \} \) and \( X_\mathcal{T} \) are new variables such that \( hX \mathcal{T} \in \text{Pat}(\mathcal{T}) \).

Figure 2: CLNC(D)-rules for constrained lazy narrowing

CS Constraint Solving \[ \exists \mathcal{T}. \; P \Box C \Box S \Box \sigma \vdash_{CS(\chi)} \exists \mathcal{T}, \; (P \Box C)\sigma_1 \Box S_1 \Box \sigma_1 \] if \( \chi = \text{var}(P) \), \( S \) is not \( \chi \)-solved, \( \text{Solve}^D(S, \chi) = \bigvee \sigma_1 (S_1 \Box \sigma_1) \), and \( \mathcal{T} \) are the new variables introduced by the solver in \( S_1 \Box \sigma_1 \), for each \( 1 \leq i \leq k \).

AC Atomic Constraint \[ \exists \mathcal{T}. \; P \Box p_\tau n \rightarrow ! t, \; C \Box S \Box \sigma \vdash_{AC} \exists \mathcal{T}, \; e_\tau m \rightarrow X_\mathcal{T}, \; P \Box C \Box p_\tau n \rightarrow ! t, \; S \Box \sigma \] if \( p \in \text{PF} \), \( p_\tau n \rightarrow t \) is an atomic constraint, \( X_\mathcal{T} \) are new variables \((0 \leq q \leq n) \) is the number of \( e_i \notin \text{Pat}_i(\mathcal{T}) \) such that \( t_i \equiv X_j (0 \leq j \leq q) \) if \( e_i \notin \text{Pat}_i(\mathcal{T}) \) and \( t_i \equiv e_i \) otherwise for each \( 1 \leq i \leq n \).

CF Conflict Failure \[ \exists \mathcal{T}. \; h_\sigma m \rightarrow h_\tau q, \; P \Box C \Box S \Box \sigma \vdash_{CF} \] if \( h_\sigma m \) is passive, and \( h \neq h' \) or else \( p \neq q \).

SF Solving Failure \[ \exists \mathcal{T}. \; P \Box C \Box S \Box \sigma \vdash_{SF(\chi)} \] if \( \chi = \text{var}(P) \), \( S \) is not \( \chi \)-solved, and \( \text{Solve}^D(S, \chi) = \bigwedge \).

Figure 3: CLNC(D)-rules for constraint solving and failure detection

An admissible goal \( G \equiv \exists \mathcal{T}. \; P \Box C \Box S \Box \sigma \) is called a solved goal if \( P \) and \( C \) are empty and \( S \) is in \( \emptyset \)-solved form in the sense of Definition 5. An initial goal can be any admissible goal.

Definition 10. An answer for an admissible goal \( G \equiv \exists \mathcal{T}. \; P \Box C \Box S \Box \sigma \) and a given CFLP(D)-program \( P \), must have the form \( \Pi \Box \theta \), where \( \Pi \subseteq \text{PCo}m(D) \) is a finite conjunction of atomic primitive constraints, \( \theta \in \text{Sub}_0(\Pi) \) is an idempotent substitution such that \( \text{dom}(\theta) \cap \text{var}(\Pi) = \emptyset \), and there is some substitution \( \theta' = \theta_{\text{var}(\theta)} \) fulfilling the following conditions:

- \( P \vdash_D (P \Box C)\theta' \subseteq \Pi \),
- \( \Pi \vdash_D S\theta' \),
- \( X' \equiv t\theta' \) for each \( X \rightarrow t \in \sigma \), abbreviated as \( \theta' \in \text{Sol}(\sigma) \).

A witness \( M \) for the fact that \( \Pi \Box \theta \) is an answer of \( G \) is defined as a multiset containing all the CRWL(D)-proofs mentioned above. We write \( \text{Ans}_D(G) \) for the set of all answers for \( G \). An answer \( \Pi \Box \theta \in \text{Ans}_D(P) \) is called trivial if \( \text{Unsat}_D(\Pi) \) and non-trivial otherwise.
Definition 11. Let $G \equiv \exists \exists \exists$. $P \square C \square S \square \sigma$ be an admissible goal for a given $CLP(D)$-program $P$. We say that a valuation $\mu \in Val(D)$ is a solution of $G$ if there is some valuation $\mu' = _{\text{evar}(G)}\mu$ satisfying the following conditions:

- $P \vdash (P \square C)\mu'$,
- $\vdash \exists \exists S \mu'$ (i.e. $\mu' \in Sol(\exists))$,
- $X \mu' \equiv t\mu'$ for each $X \rightarrow t \in \sigma$, abbreviated as $\mu' \in Sol(\sigma)$.

We write $Sol(\exists)$ for the set of all solutions for $G$. Analogously, we define the set of solutions for an answer $\Pi \equiv \exists_\exists \exists \exists \sigma$.

From Definition 10 and Definition 11, it is easy to prove that the notion of solution is a particular case of the notion of a solution. More formally, if $G$ is an admissible goal and $\mu \in Val(D)$ then $\exists \exists \exists \exists \sigma \in Ans(\exists)$. Another useful relationship between answers and solutions is given in the next proposition.

**Proposition 1.** Let $G \equiv \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exists \exist
\[ ∃U, Us, M. \{ Y \mapsto (s Y) \} → [U]\{U\} \triangledown ∎ \]
\[ M = \emptyset ∐ (X → s M) \triangledown_{DC} \]
\[ ∃U, Us, M. Y → U, \{ s Y \} → Us ∎ \]
\[ M = \emptyset ∐ (X → s M) \triangledown_{EL} \]
\[ ∃M. ∘ □ M /= 0 ∐ (X → s M) \)

computed answer: \( S_2[σ^2] ≡ M / 0 ∐ (X → s M) \)

For this example, it is also possible to prove that \( ∎ □ \theta ≡ ∎ ∎ X /= \emptyset \) is a correct answer of \( G_0 \) such that \( Sol_{H\text{seq}}(Π ∎ \theta) \subseteq \bigcup_{j \geq 1} Sol_{H\text{seq}}(S_0 ∎ σ_j) \); but no single computed answer \( S □ σ \) verifies \( Sol_{H\text{seq}}(Π ∎ \theta) ⊆ Sol_{H\text{seq}}(S □ σ) \).
We will see in Theorem 2 that this is true in general.

Example 4. Splitting a list. The next example splits a list with only one element using the \( CFLP(\mathcal{H}_{\text{seq}}) \)-program split given in Example 2.

\[ \square \text{split } [X] = (Xs, Ys) \square \square \varepsilon \vdash AC \]
\[ R_1. \text{split } [X] → R_1 \square □ R_2 = (Xs, Ys) \square \varepsilon \vdash □_{DF} \]
\[ Ys, Zs, R, R_1. \text{case } R X Ys Zs → R_1. \]
\[ \text{split } [X] → [Ys, Zs] \square □ \text{seq } X s 0 → ! R, \]
\[ R_1 = (Xs, Ys) \square \varepsilon \vdash □_{DF} \]
\[ Ys, Zs, R, R_1. \text{case } R X Ys Zs → R_1. \]
\[ [Xs, \varepsilon] → [Ys, Zs] \square □ \text{seq } X s 0 → ! R, \]
\[ R_1 = (Xs, Ys) \square \varepsilon \vdash □_{SP} \]
\[ \text{seq } X s 0 \not= 0 ! R, R_1 = (Xs, Ys) \square \varepsilon \vdash □_{CS(R_1)} \]

Now, the constraint solver over \( H_{\text{seq}} \) (see Appendix A) gives three possible alternatives

\[ \text{Solve}^{H_{\text{seq}}}([Xs, Ys, \{ R_1 \} = ((\{ R_1 \} = (Xs, Ys) \square \{ R → \text{true, } Xs \not= 0 \}) \lor \{ R_1 \} = (Xs, Ys) \square \{ R → \text{false, } Xs \not= 0 \}) \lor \{ R_1 \} = (Xs, Ys, M) \not= 0 \square \{ R → \text{false, } Xs \not= 0 \}) \]

and there are three possible continuations of the computation, each of one with a computed answer associated

1. \[ \exists R_1. \text{case true } 0 ! [ ] \rightarrow R_1 \square □ \]
\[ R_1 = (Xs, Ys) \square \{ Xs \not= 0 \} \vdash □_{DF} \]
\[ R_1. \square \{ 0, 0 \} \rightarrow R_1 \square □ \]
\[ R_1 = (Xs, Ys) \square \{ Xs \not= 0 \} \vdash □_{SP} \]
\[ \square \{ 0, 0 \} \rightarrow [Xs, \varepsilon] \square \{ Xs \not= 0 \} \vdash □_{CS(R_1)} \]
\[ \text{answer: } S_1[σ^1] \equiv \{ Xs \not= 0, Xs \not= 0, Ys \not= [ ] \} \]

2. \[ \exists R_1. \text{case false } 0 ! [ ] \rightarrow R_1 \square □ \]
\[ R_1 = (Xs, Ys) \square \{ Xs \not= 0 \} \vdash □_{DF} \]
\[ R_1. \square \{ 0, 0 \} \rightarrow R_1 \square □ \]
\[ R_1 = (Xs, Ys) \square \{ Xs \not= 0 \} \vdash □_{SP} \]
\[ \square \{ 0, 0 \} \rightarrow [Xs, \varepsilon] \square \{ Xs \not= 0 \} \vdash □_{CS(R_1)} \]
\[ \text{answer: } S_2[σ^2] \equiv \{ Xs \not= 0, Xs \not= 0, Ys \not= [ ] \} \]

3. \[ ∃M. R_1. \text{case false } s M ! [ ] \rightarrow R_1 \square □ \]
\[ R_1 = (Xs, Ys) \not= 0 \square \{ Xs \not= 0 \} \vdash □_{DF} \]
\[ R_1. \square \{ 0, 0 \} \rightarrow R_1 \square □ \]
\[ R_1 = (Xs, Ys) \not= 0 \square \{ Xs \not= 0 \} \vdash □_{SP} \]
\[ \square \{ 0, 0 \} \rightarrow [Xs, \varepsilon] \not= 0 \square \{ Xs \not= 0 \} \vdash □_{CS(R_1)} \]
\[ \text{answer: } S_3[σ^3] \equiv M \not= 0 \square \{ Xs \not= 0, Xs \not= 0, Ys \not= [ ] \} \]

4. PROPERTIES OF \( \text{CLNC}(D) \)

This section presents the main results of the paper, namely soundness and completeness of goal solving in \( \text{CLNC}(D) \) w.r.t. \( CRWL(D) \) semantics. We emphasize the technical difficulty of the Completeness Theorem 2, harder to prove than related results for \( \text{FLP} \) languages [11, 12, 31] and also stronger and more general than previous related results for \( \text{FILT} \) languages [21, 2, 3]. As main differences w.r.t. the constrained lazy narrowing calculus for the \( \text{CFLP}(D, S, L) \) scheme [25], we provide a logical semantics for correct answers (Definition 10) and a formal notion of constraint solver (Definition 5) well suited to that semantics.

Our first result proves correctness of a single transformation step. It says that transformation steps preserve admissibility of goals, fail only in case of unsatisfiable goals and do not introduce new solutions.

**Lemma 2. Correctness Lemma.**

1. The transformation steps preserve admissibility of goals: If \( G \vdash_{\text{CLNC}(D)} G' \) and \( G \) is admissible, then \( G' \) is admissible. Moreover, \( \text{fvar}(G') \subseteq \text{fvar}(G) \).

2. The transformation steps fail only in case of unsatisfiable goals: If \( G \vdash_{\text{CLNC}(D)} G' \) then \( \text{Solve}(G) = 0 \) (or equivalently, \( \text{Ans}_P(G) \) includes only trivial answers).

3. The transformation steps do not introduce new solutions: If \( G \vdash_{\text{CLNC}(D)} G' \) and \( ∎ □ θ \in \text{Ans}_P(G) \) then \( ∎ □ θ \in \text{Ans}_P(G) \).

The following soundness result follows easily from the Correctness Lemma. It ensures that computed answers for a goal \( G \) are indeed correct answers of \( G \).

**Theorem 1. Soundness of \( \text{CLNC}(D) \).**

If \( G_0 \) is an initial goal and \( G_0 \vdash_{\text{CLNC}(D)} G_n \), where \( G_n \triangledown ∎ \not= \emptyset \) is a solved goal, then \( ∎ □ σ \) is a solved goal, then \( ∎ □ σ \in \text{Ans}_P(G_0) \).

**Proof.** From Proposition 1 we get \( S □ σ \in \text{Ans}_P(G_n) \). Now, if we repeatedly backwards apply item 3. of the Correctness Lemma, we obtain \( S □ σ \in \text{Ans}_P(G_0) \).

Completeness of \( \text{CLNC}(D) \) is based on the following idea: whenever \( ∎ □ θ \in \text{Ans}_P(G) \) and \( G \) is not yet solved, there are finitely many local choices for a first computation step \( G \vdash G_j (1 \leq j \leq l) \) so that the new goals \( G_j \) are “closer to be solved” and “cover all the solutions of \( ∎ □ θ \).” This idea is made precise in the next lemma, which relies on a sophisticated well-founded progress ordering. A similar technique was used in [11, 12, 31] to prove completeness of lazy narrowing calculi for \( \text{FLP} \) languages. In the present \( \text{CFLP}(D) \) setting, the solver \text{Solve}^+ must be taken into account. As a consequence, the number \( l \) of local choices can be greater that 1 in general, and the progress ordering is more complicated than those used in [11, 12, 31].

**Lemma 3. Progress Lemma.**

Assume an admissible goal \( G \) not in solved form, and a witnessed non-trivial answer \( M : ∎ □ θ \in \text{Ans}_P(G) \). Then:

1. There is some \( \text{CLNC}(D) \) transformation rule applicable to \( G \).
2. For any CLN(C(D)) rule R applicable to G, there exist l goals Gj with witnessed non-trivial answers Mj : Πj, ∇θj ∈ AnsΠ(Gj) (1 ≤ j ≤ l) such that:

- G ⊢ Gj for each 1 ≤ j ≤ l,
- SolΠ(Π ⊡ θ) ⊆ Uj=1 SolΠ((3, c), Π ⊡ θj),
- (G, M) ⊢ (Gj, Mj) for each 1 ≤ j ≤ l, where ⊢ is the well-founded progress ordering defined in Appendix B.

Proof. (1) If G ∋ □θ ∈ S, then verifies the lexicographic progress ordering. Details are omitted here.

4. CONCLUSIONS AND FUTURE WORK

We have presented a new constrained lazy narrowing calculus CLN(C(D)) parameterized by a constraint domain D, intended as a formal specification of a goal solving procedure for constraint functional logic programs in a recently proposed CCLP(D) scheme [24]. CLN(C(D)) relies on a new formal notion of constraint solver. It is sound and strongly complete w.r.t. the declarative semantics given in [24].

In the near future, we plan to investigate both improvements and applications of the CCLP(D) scheme. Planned improvements include enriching the scheme with algebraic data constructors in the vein of [4] and the optimization of CLN(C(D)) by means of definitional trees, extending the approach of [31]. Planned applications will focus on practical instances of the CCLP(D) scheme, supporting arithmetic constraints over the real numbers and finite domain constraints. In particular, we plan to formalize the work started in [10] as an instance of the CCLP(D) scheme, and to investigate practical constraint solving methods and applications of the resulting language.

Last but not least, we plan to extend the work on declarative debugging of functional logic programs started in [6, 7] to CCLP(D)-programs, considering the diagnosis of both wrong answers and missing answers, and implementing the resulting debugging methods for some practical instances of the CCLP(D) scheme.
6. REFERENCES


A. A CONSTRAINT SOLVER OVER H_SEQ

Definition 12. The Constraint Solver Solve_{H_{seq}}.

We assume the constraint domain H_{seq} described in Example 1. Let \( \chi \subseteq \mathcal{V} \) be a set of protected variables and \( S \subseteq \text{PCon}(H_{seq}) \) a conjunction of atomic primitive constraints over \( H_{seq} \) of the form \( S \equiv \text{seq} t_1 s_1 \rightarrow r_1, \ldots, \text{seq} t_n s_n \rightarrow r_n \), where \( t_i, s_i \in \text{Pat}(\emptyset) \) and \( r_i \in \{\text{true}, \text{false}\} \cup \mathcal{V} \). We define a constraint solver for equality and disequality constraints as follows: \( \text{Solve}_{H_{seq}}(S, \chi) = \bigvee_{i=1}^{k} (S, \varnothing) \Leftrightarrow_{def} S_{\mathcal{E}} \varphi \Leftrightarrow_{\chi} \bigvee_{i=1}^{k} (S, \varnothing) \). If the relation \( \varphi \Leftrightarrow_{\chi} \) defined below denotes one constraint solver step, and \( \varphi \not\equiv_{\chi} \) expresses that \( \varphi \) is not reducible by \( \varphi \Leftrightarrow_{\chi} \).

The following rule system specifies the behaviour of the relation \( \varphi \Leftrightarrow_{\chi} \) between constraint disjunctions of the form \( \bigvee_{i}
( S, \varnothing), \) such that each \( S_i \subseteq \text{PCon}(H_{seq}), \) \( \varnothing \in \text{Sub}(\emptyset), \) \( \chi \cap \text{var}(S_i) = \emptyset \) and \( \text{var}(S_i) \cap \text{dom}(\varnothing) = \emptyset \). When applying the rules for \( \varphi \Leftrightarrow_{\chi} \) we ignore the order of \( S \) and we view \( \varphi \equiv \) as symmetric.

**General rules for \( \varphi \Leftrightarrow_{\chi} \)**

\[ R_0 \ldots \lor S_{i} \quad \Leftrightarrow_{\chi} \quad \varphi \Leftrightarrow_{\chi} \quad R_1 \]

if \( S_{i} \Leftrightarrow_{\chi} \varphi \), \( \chi \cap \text{var}(S_i) = \emptyset \) and \( \text{var}(S_i) \cap \text{dom}(\varnothing) = \emptyset \).

**Rules for strict equality**

\[ R_2 \quad h_{\mathcal{E}} = h_{\mathcal{E}} \quad \Leftrightarrow_{\chi} \quad \chi \cap \text{var}(t) = \emptyset \quad \text{and} \quad \theta = \{X \rightarrow t\}. \]

**Rules for strict disequality**

\[ R_4 \quad h_{\mathcal{E}} \neq h_{\mathcal{E}} \quad \Leftrightarrow_{\chi} \quad 
\]

\[ R_5 \quad h_{\mathcal{E}} \neq h_{\mathcal{E}} \quad \Leftrightarrow_{\chi} \quad \chi \neq \emptyset \quad \text{and} \quad \theta = \{X \rightarrow h_{\mathcal{E}}\}. \]

**Calculation of demanded variables in H_{seq}**

The following rules serve to compute the set \( \text{dvar}_{H_{seq}}(S) \) of demanded variables by a satisfiable set of primitive constraints \( S \subseteq \text{PCon}(H_{seq}) \).

\[ \text{dvar}(\text{seq} t s \rightarrow R, S) = 
\]

\[ \{R\} \cup \text{dvar}_{H_{seq}}(t = s, S_{\theta_1}) \cup \text{dvar}(t = s, S_{\theta_2}) \] if \( R \) is a variable, \( \theta_1 = \{R \rightarrow \text{true}\} \) and \( \theta_2 = \{R \rightarrow \text{false}\}. \]

**B. PROGRESS ORDERING**

Definition 13. Well-founded progress ordering. Let \( \mathcal{P} \) be a CLNC(D)-program, \( G \equiv \exists \mathcal{U} \cup \varnothing \cup C \cup S \) an admissible goal for \( \mathcal{P} \) and \( \mathcal{M} : \exists \mathcal{U} \in \text{Ans}(\mathcal{G}) \) a witnessed answer. We define the following sizes associated to \( G \) and \( \mathcal{M} \):

- The restricted size of the witness \( \mathcal{M} = \{T_1, \ldots, T_n\} \) (represented by \( | \mathcal{M} | \)) is the multiset of natural numbers \( \{\|T_1|, \ldots, |T_n|\} \), where \( |T_i| \) denotes the restricted size of each proof tree \( T_i (1 \leq i \leq n) \), as defined in Subsection 2.4.
- The size \( |G| \) is the number of occurrences in \( G \) of expressions \( F\mathcal{T}_\mathcal{S} \) with \( F \) a variable and \( k > 0 \).
- The size \( |G| \) is the number of occurrences in \( G \) of rigid and passive expressions \( \mathcal{T}_\mathcal{S} \), that are not patterns.
- The size \( |G| \) is the total syntactic size of the right hand sides of productions in \( G \).
- The restricted size of the constraint store (represented by \( |S| \)) is evaluated to 1 if \( S \) is in solved form and 0 in other case.

Over pairs \((G, \mathcal{M})\) we define a well-founded progress ordering \( (G, \mathcal{M}) \succ (G', \mathcal{M}') \) as:

\[ \text{dvar}(\text{seq} t s \rightarrow R, S) = 
\]

\[ \{R\} \cup \text{dvar}_{H_{seq}}(t = s, S_{\theta_1}) \cup \text{dvar}(t = s, S_{\theta_2}) \]

where \( \succ_\text{lex} \) is the lexicographic product of \( \succ_\text{mul} \times \succ_N \times \succ_N \) and \( \succ_\text{mul} \) is the multiset order for multisets over \( N \), and \( \succ_N \) is the usual ordering over \( N \). See [5] for definitions of these notions.

The following table shows the behaviour of the different CLNC(D) transformations w.r.t. the five components of the lexicographic progress ordering.

<table>
<thead>
<tr>
<th>RULE</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DC</td>
<td>(\succ_\text{mul})</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
</tr>
<tr>
<td>SP</td>
<td>(\succ_\text{mul})</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
</tr>
<tr>
<td>IM</td>
<td>(\succ_\text{mul})</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
</tr>
<tr>
<td>EL</td>
<td>(\succ_\text{mul})</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
</tr>
<tr>
<td>PF</td>
<td>(\succ_\text{mul})</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
</tr>
<tr>
<td>DP</td>
<td>(\succ_\text{mul})</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
</tr>
<tr>
<td>FV</td>
<td>(\succ_\text{mul})</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
</tr>
<tr>
<td>CS</td>
<td>(\succ_\text{mul})</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
</tr>
<tr>
<td>AC</td>
<td>(\succ_\text{mul})</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
<td>(\succ_N)</td>
</tr>
</tbody>
</table>

Table 1: progress ordering