Declarative Constraint Programming with Definitional Trees

Rafael del Vado Vírseda*

Dpto. de Sistemas Informáticos y Programación,
Universidad Complutense de Madrid, Spain
rdelvado@sip.ucm.es

Abstract. The new generic scheme $CFLP(\mathcal{D})$ has been recently proposed in [14] as a logical and semantic framework for lazy Constraint Functional Logic Programming over a parametrically given constraint domain $\mathcal{D}$. Further, [15] presented a Constrained Lazy Narrowing Calculus $CLNC(\mathcal{D})$ as a convenient computation mechanism for solving goals for $CFLP(\mathcal{D})$-programs, which was proved sound and strongly complete with respect to $CFLP(\mathcal{D})$'s semantics. Now, in order to provide a formal foundation for an efficient implementation of goal solving methods in existing systems such as *Curry* [8] and $\mathcal{TOY}$ [13,6], this paper enriches the $CFLP(\mathcal{D})$ framework by presenting an optimization of the $CLNC(\mathcal{D})$ calculus by means of definitional trees to efficiently control the computation. We prove that this new Constrained Demanded Narrowing Calculus $CDNC(\mathcal{D})$ preserves the soundness and completeness properties of $CLNC(\mathcal{D})$ and maintains the good properties shown for needed and demand-driven narrowing strategies [4,11,17].

1 Introduction

The effort to combine the main lines of research in multiparadigm declarative programming, namely *Constraint Logic Programming* (CLP) [10] and *Functional Logic Programming* (FLP) [7], in a unified and suitable framework called *Constrained Functional Logic Programming* ($CFLP$), arose around 1990 and has grown in the last years. Recently, a new generic scheme called $CFLP(\mathcal{D})$ has been proposed in [14] as a logical and semantic framework for lazy Constraint Functional Logic Programming over a parametrically given constraint domain $\mathcal{D}$, which provides a clean and rigorous declarative semantics for $CFLP$ languages as in the $CLP(\mathcal{D})$ scheme, but overcoming some limitations of older $CFLP$ schemes [12,16]. In this setting, $CFLP(\mathcal{D})$-programs are presented as sets of constrained rewrite rules that define the behavior of possible higher-order and/or non-deterministic lazy functions over $\mathcal{D}$. The main novelties in [14] were a new formalization of constraint domains for $CFLP$ and a new *Constraint ReWriting Logic* $CRWL(\mathcal{D})$ parameterized by a constraint domain $\mathcal{D}$, which provides a logical characterization of program semantics. Further, [15] has extended [14] with a

* The work of this author has been partially supported by the Spanish National Project MELODIAS (TIC2002-01167).

B. Gramlich (Ed.): FroCoS 2005, LNAI 3717, pp. 184–199, 2005.

suitable operational semantics, which relies on a new formal notion of constraint solver and a new *Constrained Lazy Narrowing Calculus* $CLNC(\mathcal{D})$ for solving goals for $CFLP(\mathcal{D})$-programs, which can be proved sound and strongly complete w.r.t. $CRWL(\mathcal{D})$'s semantics. These properties qualify $CLNC(\mathcal{D})$ as a convenient computation mechanism for declarative constraint programming languages.

However, efficiency is a major concern for the implementation of $CFLP(\mathcal{D})$ systems, since non-deterministic computations often generate huge search spaces with their associated overheads both in terms of time and space. In the field of functional logic programming languages using lazy narrowing as operational model, *needed narrowing strategies* [4,2,9] and *demand-driven narrowing strategies* [11,17] are known to provide a sound and complete goal solving mechanism while avoiding unneeded computation steps. These strategies are based on *definitional trees*, first introduced in [1], and they have led to efficient implementations of lazy narrowing in existing systems such as *Curry* [8] and $\mathcal{TOY}$ [13,6].

Although *Curry* and $\mathcal{TOY}$ support constraint programming over a few specific domains, general results on the combination of demand/needed lazy narrowing with goal solving are still missing. The aim of the present paper is to provide such results. More precisely, this paper uses definitional trees to design a combination of the *Constrained Lazy Narrowing Calculus* $CLNC(\mathcal{D})$ from [15] and the *Demand-driven Narrowing Calculus* DNC from [17], yielding a new *Constrained Demanded Narrowing Calculus* $CDNC(\mathcal{D})$ over a parametrically given constraint domain $\mathcal{D}$ which can be proved sound and strongly complete w.r.t. $CRWL(\mathcal{D})$'s semantics, contracts needed positions, and maintains the efficiency properties shown for existing demand/needed narrowing strategies.

The organization of this paper is as follow: Section 2 is devoted to summarize the presentation of the $CFLP(\mathcal{D})$ scheme [14,15] and the technical preliminaries need to formalize the notion of definitional tree. Section 3 introduces a refined representation of definitional trees that deals properly with constraints and defines the subclass of $CFLP(\mathcal{D})$-programs used in this work. In Section 4 we give a formal presentation of the calculus $CDNC(\mathcal{D})$, proving soundness and completeness results and sketching optimality. Finally, some conclusions and plans for future work are drawn in Section 5.

2 The Generic Scheme $CFLP(\mathcal{D})$

In this section we introduce some technical preliminaries regarding the basis of the $CFLP(\mathcal{D})$ scheme [14,15] for lazy Constraint Functional Logic Programming over a parametrically given constraint domain $\mathcal{D}$. We will use this scheme as the logical and semantic framework to define our declarative constraint programs and our new Constrained Demanded Narrowing Calculus with definitional trees.

2.1 Expressions and Patterns

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

We assume a *universal signature* $\Sigma = \langle DC, FS \rangle$, where $DC = \bigcup_{n\in\mathbb{N}} DC^n$ and $FS = \bigcup_{n\in\mathbb{N}} FS^n$ are families of countably infinite and mutually disjoint sets of *data constructors* resp. *evaluable function symbols*, each one with an associated arity. Evaluable functions can be further classified into domain dependent *primitive functions* $PF^n \subseteq FS^n$ and user *defined functions* $DF^n = FS^n \setminus PF^n$ for each $n \in \mathbb{N}$. We write $\Sigma_\bot$ for the result of extending DC^0 with the special symbol $\bot$, intended to denote an undefined data value. As notational conventions, we use $c, d \in DC$, $f, g \in FS$ and $h \in DC \cup FS$, and we define the *arity* of $h \in DC^n \cup FS^n$ as $ar(h) = n$. We also assume that 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 $\mathcal{V}$ of *variables* $X, Y, \ldots$ and a set $\mathcal{U}$ of *primitive elements* $u, v, \ldots$ (as e.g. the set $\mathbb{R}$ of the real numbers) mutually disjoint and disjoint from $\Sigma_\bot$. *Partial expressions* $e \in Exp_\bot(\mathcal{U})$ have the following syntax:

$$e ::= \bot \mid u \mid X \mid h \mid (e\, e_1)$$

where $u \in \mathcal{U}$, $X \in \mathcal{V}$, $h \in DC \cup FS$. Following a usual convention, we assume that application associates to the left, and we use the notation $(e\, \overline{e}_n)$ to abbreviate $(e\, e_1 \ldots e_n)$. The set of variables occurring in e is written $var(e)$. An expression e is called *linear* iff there is no $X \in var(e)$ having more than one occurrence in e. The following classification of expressions is also useful: $(X\, \overline{e}_m)$, with $X \in \mathcal{V}$ and $m \geq 0$, is called a *flexible expression*, while $u \in \mathcal{U}$ and $(h\, \overline{e}_m)$ with $h \in DC \cup FS$ are called *rigid expressions*. Moreover, a rigid expression $(h\, \overline{e}_m)$ is called *active* iff $h \in FS$ and $m \geq ar(h)$, and *passive* otherwise. The occurrence of a symbol is *passive* iff is a primite element $u \in \mathcal{U}$ or is the root symbol h of a passive expression (a symbol, as used in this sense, is called a *passive symbol*). Some interesting subsets of $Exp_\bot(\mathcal{U})$ are: $GExp_\bot(\mathcal{U})$, the set of the *ground* expressions e such that $var(e) = \emptyset$ and $Exp(\mathcal{U})$, the set of the *total* expressions e with no occurrences of $\bot$. Another important subclass of expressions is the set of *partial patterns* $s, t \in Pat_\bot(\mathcal{U})$, whose syntax is defined as follows:

$$t ::= \bot \mid u \mid X \mid c\, \overline{t}_m \mid f\, \overline{t}_m$$

where $u \in \mathcal{U}$, $X \in \mathcal{V}$, $c \in DC^n$, $m \leq n$, $f \in FS^n$, $m < n$. The subsets $Pat(\mathcal{U})$, $GPat_\bot(\mathcal{U}) \subseteq Pat_\bot(\mathcal{U})$ consisting of the *total* and *ground* patterns, respectively, are defined in the natural way. We define the *information ordering* $\sqsubseteq$ as the least partial ordering over $Pat_\bot(\mathcal{U})$ satisfying the following properties: $\bot \sqsubseteq t$ for all $t \in Pat_\bot(\mathcal{U})$, and $(h\overline{t}_m) \sqsubseteq (h\overline{t'}_m)$ whenever these two expressions are patterns and $t_i \sqsubseteq t'_i$ for all $1 \leq i \leq m$.

2.2 Substitutions

As usual, we define *substitutions* $\sigma \in Sub_\bot(\mathcal{U})$ as mappings $\sigma : \mathcal{V} \to Pat_\bot(\mathcal{U})$ extended to $\sigma : Exp_\bot(\mathcal{U}) \to Exp_\bot(\mathcal{U})$ in the natural way. By convention, we write ε for the identity substitution, $e\sigma$ instead of $\sigma(e)$, and $\sigma\theta$ for the composition of σ and θ, such that $e(\sigma\theta) = (e\sigma)\theta$ for any $e \in Exp_\bot(\mathcal{U})$. We define the *domain* and the *variable range* of a substitution in the usual way, namely: $dom(\sigma) = \{X \in \mathcal{V} \mid \sigma(X) \neq X\}$ and $ran(\sigma) = \bigcup_{X\in\, dom(\sigma)} var(\sigma(X))$. As

usual, a substitution σ such that $dom(\sigma) \cap ran(\sigma) = \emptyset$ is called *idempotent*. For any set of variables $\mathcal{X} \subseteq \mathcal{V}$ we define the *restriction* $\sigma \restriction_{\mathcal{X}}$ as the substitution σ' such that $dom(\sigma') = \mathcal{X}$ and $\sigma'(X) = \sigma(X)$ for all $X \in \mathcal{X}$. We use the notation $\sigma =_{\mathcal{X}} \theta$ to indicate that $\sigma \restriction_{\mathcal{X}} = \theta \restriction_{\mathcal{X}}$, and we abbreviate $\sigma =_{\mathcal{V} \setminus \mathcal{X}} \theta$ as $\sigma =_{\setminus \mathcal{X}} \theta$. An expression e' is an *instance* of e if there is a substitution σ with $e' = e\sigma$. In this case we write $e \preceq e'$. An expression e' is a *variant* of e if $e \preceq e'$ and $e' \preceq e$.

2.3 Positions

To manipulate expressions and patterns we give the following definitions. An *occurrence* or *position* is a sequence p of positive integers identifying a subexpression in an expression. For every expression e, the set $Pos(e)$ of *positions* in e is inductively defined as follow: the empty sequence denoted by ϵ, identifies e itself. For every expression of the form $h\overline{e}_m$, the sequence $i \cdot q$, where i is a positive integer not greater than n and q is a position, identifies the subexpression of e_i at q. The subexpression of e at p is denoted by $e|_p$ and the result of *replacing* $e|_p$ with e' in e is denoted by $e[e']_p$. The expression $p \cdot q$ denotes the position resulting from the concatenation of the positions p and q. If e is a linear expression, $pos(X, e)$ will be used for the position of the variable X occurring in e.

2.4 Constraints over a Given Constraint Domain

Now we are ready to give a short summary of the generic $CFLP(\mathcal{D})$ scheme by introducing the essential notions of constraint domain $\mathcal{D}$, constraints and constraint solver over $\mathcal{D}$ which are needed for this work. Additional explanations and examples can be found in [14].

Definition 1 (constraint domain). *A* constraint domain *is any structure* $\mathcal{D} = \langle D_{\mathcal{U}}, \{p^{\mathcal{D}} \mid p \in PF\}, Solve^{\mathcal{D}} \rangle$ *such that the carrier set* $D_{\mathcal{U}} = GPat_{\bot}(\mathcal{U})$ *coincides with the set of ground patterns for some set of primitive elements* $\mathcal{U}$*, the interpretation* $p^{\mathcal{D}} \subseteq D_{\mathcal{U}}^n \times D_{\mathcal{U}}$ *of each* $p \in PF^n$ *(we use the notation* $p^{\mathcal{D}}\overline{t}_n \rightarrow t$ *to indicate that* $(\overline{t}_n, t) \in p^{\mathcal{D}}$*) satisfies* monotonicity*,* antimonotonicity *and* radicality *requirements (see [14] for details) and* $Solve^{\mathcal{D}}$ *is a* constraint solver*, whose expected behavior will be explained in* Definition 4 *below.*

Assuming an arbitrarily fixed constraint domain $\mathcal{D}$ built over a certain set of primitive elements $\mathcal{U}$ (as e.g. the set $\mathbb{R}$ of the real numbers), we will now define the syntax and semantics of constraints over $\mathcal{D}$ used in this work.

Definition 2 (constraints over a constraint domain).

1. Primitive Constraints *have the syntactic form* $p\overline{t}_n \rightarrow\!! \; t$*, with* $p \in PF^n$*,* $\overline{t}_n \in Pat_{\bot}(\mathcal{U})$ *and* $t \in Pat(\mathcal{U})$ *(for example, addition constraints* $X + Y \rightarrow\!! \; R$ *or comparison constraints* $X > 0 \rightarrow\!! \; true$ *over* $\mathbb{R}$*). The special constants* $\triangle$ *and* $\blacktriangle$ *are also primitive constraints.*
2. Constraints *have the syntactic form* $p\,\overline{e}_n \rightarrow\!! \, t$*, with* $p \in PF^n$*,* $\overline{e}_n \in Exp_{\bot}(\mathcal{U})$ *and* $t \in Pat(\mathcal{U})$ *(i.e. possibly including occurrences of user defined function symbols). The special constants* $\triangle$ *and* $\blacktriangle$ *are also constraints.*

In the sequel we use the notation $PCon_\perp(\mathcal{D})$ for the set of all the primitive constraints π over $\mathcal{D}$ and we reserve the capital letters Π and S for sets of primitive constraints, often interpreted as conjunctions. The semantics of primitive constraints depends on the notion of solution, presented in the next definition.

Definition 3 (primitive semantic notions).

1. *The set of* valuations *over $\mathcal{D}$ is defined as the set of ground substitutions $Val_\perp(\mathcal{D}) = GSub_\perp(\mathcal{U})$. The set of* solutions *of $\pi \in PCon_\perp(\mathcal{D})$ is a subset $Sol_\mathcal{D}(\pi) \subseteq Val_\perp(\mathcal{D})$ defined as follows: $Sol_\mathcal{D}(\triangle) = Val_\perp(\mathcal{D})$, $Sol_\mathcal{D}(\blacktriangle) = \emptyset$ and $Sol_\mathcal{D}(p\,\overline{t}_n \rightarrow\!!\, t) = \{\eta \in Val_\perp(\mathcal{D}) \mid t\eta$ is total and $p^\mathcal{D}\,\overline{t}_n\eta \rightarrow t\eta\}$. The set of solutions of $\Pi \subseteq PCon_\perp(\mathcal{D})$ is defined as $Sol_\mathcal{D}(\Pi) = \bigcap_{\pi\in\Pi} Sol_\mathcal{D}(\pi)$.*
2. *Π is called* satisfiable *in $\mathcal{D}$ (in symbols, $Sat_\mathcal{D}(\Pi)$) iff $Sol_\mathcal{D}(\Pi) \neq \emptyset$. Otherwise Π is called* unsatisfiable *(in symbols, $Unsat_\mathcal{D}(\Pi)$).*
3. *π is a* consequence *of Π in $\mathcal{D}$ (in symbols, $\Pi \models_\mathcal{D} \pi$) iff $Sol_\mathcal{D}(\Pi) \subseteq Sol_\mathcal{D}(\pi)$. In particular, $p\,\overline{t}_n \rightarrow\!!\, t$ is a consequence of Π in $\mathcal{D}$ (in symbols, $\Pi \models_\mathcal{D} p\,\overline{t}_n \rightarrow\!!\, t$) iff $p^\mathcal{D}\,\overline{t}_n\eta \rightarrow t\eta$ with $t\eta$ total holds for all $\eta \in Sol_\mathcal{D}(\Pi)$.*

Finally, we describe the behavior of the constraint solver $Solve^\mathcal{D}$ introduced in *Definition 1* as the basis of our new goal solving mechanism. This notion of solver was first introduced in [15] w.r.t the semantics given in the previous definition.

Definition 4 (constraint solver over $\mathcal{D}$).

1. *We say that a variable $X \in \mathcal{V}$ is* demanded *by a set of total primitive constraints $\Pi \subseteq PCon(\mathcal{D})$ iff $\mu(X) \neq \perp$ holds for every $\mu \in Sol_\mathcal{D}(\Pi)$. We write $dvar_\mathcal{D}(\Pi)$ for the set of the variables demanded by Π. For practical constraint domains, $dvar_\mathcal{D}(\Pi)$ is expected to be computable (see [15]).*
2. *A* constraint solver *over a constraint domain $\mathcal{D}$ is formalized by a function $Solve^\mathcal{D}$ expecting as parameters a finite set $S \subseteq PCon(\mathcal{D})$ of total primitive constraints (called the* constraint store*) and a finite set of variables $\chi \subseteq \mathcal{V}$ (called the set of* protected variables*). The solver is expected to return a finite disjunction of alternatives $Solve^\mathcal{D}(S, \chi) = \bigvee_{i=1}^k (S_i \,\square\, \sigma_i)$, where $\square$ must be interpreted as conjunction, and satisfying the following requirements:*
   (a) *Each $S_i \subseteq PCon(\mathcal{D})$ is in χ-*solved form *(i.e. $Solve^\mathcal{D}(S_i, \chi) = S_i \,\square\, \varepsilon$). Furthermore, either all the protected variables disappear or some protected variable becomes demanded (i.e. $var(S_i) \cap \chi = \emptyset$ or else $dvar_\mathcal{D}(S_i) \cap \chi \neq \emptyset$).*
   (b) *Each $\sigma_i \in Sub(\mathcal{U})$ is an idempotent total substitution that cannot bind protected variables (i.e. $dom(\sigma_i) \cap var(S_i) = \emptyset$ and $\chi \cap (dom(\sigma_i) \cup ran(\sigma_i)) = \emptyset$).*
   (c) *No solution is lost by the constraint solver and the solution space associated to each alternative is included in the one of the input constraint store (i.e. $Sol_\mathcal{D}(S) = \bigcup_{i=1}^k Sol_\mathcal{D}(S_i \,\square\, \sigma_i)$).*

   In the case $k = 0$, $\bigvee_{i=1}^k (S_i \,\square\, \sigma_i)$ is understood as $\blacktriangle$. In this case, $Sol_\mathcal{D}(S) \subseteq Sol_\mathcal{D}(\blacktriangle) = \emptyset$ means failure detection. More details on the working of constraint solvers will be given in Section 4.

Example 1 (The constraint domain $\mathcal{H}_{seq}$). We consider the simple constraint domain $\mathcal{H}_{seq}$, analogous to the extension of the Herbrand Domain with equality and disequality constraints, built over an empty set of primitive elements and having the strict equality *seq* as its only primitive, interpreted to behave as follows: $seq^{\mathcal{H}_{seq}}\ t\ t \rightarrow true$ for all total $t \in GPat(\emptyset)$; $seq^{\mathcal{H}_{seq}}\ t\ s \rightarrow false$ for all $t, s \in GPat_{\perp}(\emptyset)$ such that t, s have no common upper bound w.r.t. the information ordering $\sqsubseteq$; $seq^{\mathcal{H}_{seq}}\ t\ s \rightarrow \perp$ otherwise. In the sequel, $t == s$ abbreviates the equality constraint $seq\ t\ s \rightarrow! true$ and $t /= s$ abbreviates the disequality constraint $seq\ t\ s \rightarrow! false$. A constraint solver $Solve^{\mathcal{H}_{seq}}$ for this domain can be found in [15]. Further examples of other interesting constraint domains known for their practical value in constraint programming (real numbers, finite domains, etc) can be found in [14]. All the results of this paper are valid for any arbitrary constraint domain $\mathcal{D}$ satisfying *Definition 1.*

3 *CFLP*($\mathcal{D}$)-Programs with Definitional Trees

The class of so-called $COISS(\mathcal{D})$-programs with constraints and definitional trees used in this work is a proper subclass of the generic $CFLP(\mathcal{D})$-programs presented in [14,15]. In this section we discuss $COISS(\mathcal{D})$-programs and their intended semantics.

3.1 Overlapping Definitional Trees with Constraints

In the sequel we assume an arbitrarily fixed constraint domain $\mathcal{D}$ built over a set of primitive elements $\mathcal{U}$. In this setting, *$CFLP(\mathcal{D})$-programs* are presented as sets of constrained rewrite rules that define the behavior of possibly higher-order and/or non-deterministic lazy functions over $\mathcal{D}$, called *program rules*. More precisely, a program rule R for $f \in DF^n$ has the form $R : f\,\overline{t}_n \rightarrow r \Leftarrow P \,\square\, C$ (abbreviated as $f\,\overline{t}_n \rightarrow r$ if P and C are both empty) and is required to satisfy the three conditions listed below:

1. The *left-hand side* $f\,\overline{t}_n$ is a linear expression, and for all $1 \leq i \leq n$, $t_i \in Pat(\mathcal{U})$ are total patterns. The *right-hand side* $r \in Exp(\mathcal{U})$ is also total.
2. P is a finite sequence of so-called *productions* of the form $e_i \rightarrow R_i$ $(1 \leq i \leq k)$, intended to be interpreted as conjunction of local definitions, and fulfilling the following two *admissibility conditions*:
   (a) For all $1 \leq i \leq k$, $e_i \in Exp(\mathcal{U})$ is a total expression, R_i is a different variable, and $R_i \notin var(f\,\overline{t}_n)$.
   (b) It is possible to reorder the productions of P in the form $P \equiv e_1 \rightarrow R_1, \ldots, e_k \rightarrow R_k$ where $R_j \notin var(e_i)$ for all $1 \leq i \leq j \leq k$ (*no cycles*).
3. C is a finite set of total constraints, also intended to be interpreted as conjunction, and possibly including occurrences of defined function symbols.

Example 2. The following $CFLP(\mathcal{D})$-program can be used over the constraint domain $\mathcal{H}_{seq}$ presented in *Example 1.* We use the constructors $0 \in DC^0$, $s \in DC^1$, and a Prolog-like syntax for list constructors (i.e. [ ] denotes the empty list

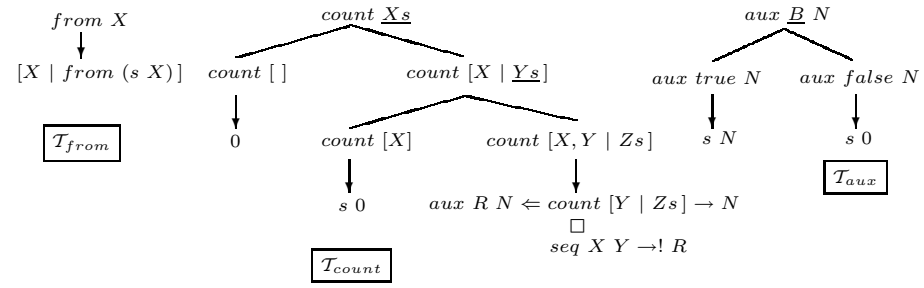

Fig. 1. Constrained definitional trees for *from*, *count* and *aux*

and $[X|Xs]$ denotes a non-empty list consisting of a first element X and a remaining list Xs). More examples of $CFLP(\mathcal{D})$-programs can be found in [14,15].

$from\ X \rightarrow [X \mid from\ (s\ X)]$ % increasing infinite list *from* starting value X

$count\ [\,] \rightarrow 0$ % *counting* consecutive and repetitive values from
$count\ [X] \rightarrow s\ 0$ % the head of a list until finding a different value
$count\ [X,Y|Zs] \rightarrow aux\ R\ N \Leftarrow count\ [Y|Zs] \rightarrow N\ \square\ seq\ X\ Y \rightarrow!\ R$

$aux\ true\ N \rightarrow s\ N$
$aux\ false\ N \rightarrow s\ 0$

The class of $COISS(\mathcal{D})$-programs is defined as the subclass of $CFLP(\mathcal{D})$-programs whose defining rules can be organized in a hierarchical structure called *definitional tree* [1]. More precisely, we choose to reformulate the notions presented in [2,3,17] about *overlapping definitional trees* and *conditional overlapping inductively sequential systems*, including now constrained rules over a generic constraint domain $\mathcal{D}$.

Definition 5 (definitional trees with constraints).
Let $\mathcal{P}$ be a $CFLP(\mathcal{D})$-program over a given constraint domain $\mathcal{D}$. A call pattern *is any linear pattern of the form $f\overline{t}_n$, where $f \in DF^n$ and $\overline{t}_n \in Pat_\perp(\mathcal{U})$. $\mathcal{T}$ is a* constrained Definitional Tree over $\mathcal{D}$ *($cDT(\mathcal{D})$ for short)* with call pattern τ *iff its depth is finite and one of the following cases holds:*

- $\mathcal{T} \equiv \underline{rule}(\tau \rightarrow r_1 \Leftarrow P_1 \,\square\, C_1 | \ldots | r_m \Leftarrow P_m \,\square\, C_m)$, *where $\tau \rightarrow r_i \Leftarrow P_i \,\square\, C_i$ for all $1 \leq i \leq m$ is a variant of a program rule in $\mathcal{P}$.*
- $\mathcal{T} \equiv \underline{case}(\tau, X, [\mathcal{T}_1, \ldots, \mathcal{T}_k])$, *where X is a variable in τ, $h_1, \ldots, h_k$ $(k > 0)$ are pairwise different passive symbols of $\mathcal{P}$, and for all $1 \leq i \leq k$, $\mathcal{T}_i$ is a $cDT(\mathcal{D})$ with call pattern $\tau\sigma_i$, where $\sigma_i = \{X \mapsto h_i\overline{Y}_{m_i}\}$ with $\overline{Y}_{m_i}$ new distinct variables such that $h_i\overline{Y}_{m_i} \in Pat(\mathcal{U})$.*

We represent a $cDT(\mathcal{D})$ $\mathcal{T}$ with call pattern τ using the notation $\mathcal{T}_\tau$. A $cDT(\mathcal{D})$ of a function symbol *$f \in DF^n$ defined by $\mathcal{P}$ is a $cDT(\mathcal{D})$ $\mathcal{T}$ with call pattern $f\overline{X}_n$, where $\overline{X}_n$ are new variables. We represent it using the notation $\mathcal{T}_f$.*

Definition 6 ($COISS(\mathcal{D})$-programs).

1. *A function symbol $f \in DF^n$ is called* constrained overlapping inductively sequential *w.r.t. a $CFLP(\mathcal{D})$-program $\mathcal{P}$ iff there exists a $cDT(\mathcal{D})$ $\mathcal{T}_f$ of f such that the collection of all the program rules $\tau \to r_i \Leftarrow P_i \square C_i$ $(1 \leq i \leq m)$ obtained from the different nodes $\underline{rule}(\tau \to r_1 \Leftarrow P_1 \square C_1 | \ldots | r_m \Leftarrow P_m \square C_m)$ occurring in $\mathcal{T}_f$ equals, up to variants, the collection of all the program rules in $\mathcal{P}$ whose left hand side has the root symbol f.*
2. *A $CFLP(\mathcal{D})$-program $\mathcal{P}$ is called a* Constrained Overlapping Inductively Sequential System over $\mathcal{D}$ *(shortly, $COISS(\mathcal{D})$) iff each function defined by $\mathcal{P}$ is constrained overlapping inductively sequential.*

As a concrete example, we consider the $CFLP(\mathcal{H}_{seq})$-program given in *Example 2*. From the definitional trees illustrated by the pictures given in *Figure 1*, it is easy to check that this program is a $COISS(\mathcal{H}_{seq})$. For example, the defined function symbol *count* has the following definitional tree $\mathcal{T}_{count}$:

$\underline{case}\ (count\ Xs, Xs, [$
$\quad \underline{rule}\ (count\ [\,]\ \to\ 0),$
$\quad \underline{case}\ (count\ [X|Ys], Ys, [$
$\quad\quad \underline{rule}\ (count\ [X] \to s\ 0),$
$\quad\quad \underline{rule}\ (count\ [X, Y|Zs]\ \to\ aux\ R\ N\ \Leftarrow\ count\ [Y|Zs]\ \to\ N\ \square\ seq\ X\ Y\ \to!\ R)\ ]\ )\ ]\ )$

3.2 The Constraint ReWriting Logic *CRWL*($\mathcal{D}$)

The *Constraint ReWriting Logic* $CRWL(\mathcal{D})$, parameterized by a constraint domain $\mathcal{D}$ over a set of primitive elements $\mathcal{U}$ and formalized by means of a constrained rewriting calculus, was introduced in [14] in order to provide a declarative semantic for $CFLP(\mathcal{D})$-programs. Now, in order to use $CRWL(\mathcal{D})$ as a logical framework for the semantics of $COISS(\mathcal{D})$-programs, we must first introduce the two possible kinds of constrained statements (*c-statements*) that we intend to derive from a given $COISS(\mathcal{D})$-program:

- *c-productions* of the form $e \to t \Leftarrow \Pi$, where $e \in Exp_\bot(\mathcal{U})$, $t \in Pat_\bot(\mathcal{U})$ and $\Pi \subseteq PCon_\bot(\mathcal{D})$.
- *c-constraints* of the form $p\,\overline{e}_n \to! t \Leftarrow \Pi$, with $p \in PF^n$, $\overline{e}_n \in Exp_\bot(\mathcal{U})$, $t \in Pat(\mathcal{U})$ and $\Pi \subseteq PCon_\bot(\mathcal{D})$.

The purpose of the calculus $CRWL(\mathcal{D})$ is to infer the semantic validity of arbitrary c-statements φ from the program rules in $\mathcal{P}$. We write $\mathcal{P} \vdash_{\mathcal{D}} \varphi$ to indicate that the c-statement φ can be derived from $\mathcal{P}$ in the constrained rewriting calculus $CRWL(\mathcal{D})$ using the set of inference rules given in [14,15]. Useful properties and correctness results relating $CRWL(\mathcal{D})$-derivability to a suitable model-theoretic semantics are also given in [14].

4 Constrained Lazy Narrowing with Definitional Trees

In this section we present a *Constrained Demanded Narrowing Calculus* (shortly, $CDNC(\mathcal{D})$) over $COISS(\mathcal{D})$-programs. For our discussion of this new calculus with definitional trees we are going to combine the ideas and techniques underlying the $CLNC(\mathcal{D})$ calculus in [15] (with generic constraints but no definitional trees) and the DNC calculus in [17] (with definitional trees but no generic constraints). The general idea is to ensure the computation of answers from goals which are correct with respect to $CRWL(\mathcal{D})$'s semantics, while using definitional trees in a similar way to [4] to ensure that all the constrained lazy narrowing steps performed during the computation are needed ones. We give first a precise definition for the class of admissible goals and answers that we are going to use.

4.1 Admissible Goals and Answers

A *goal* for a $COISS(\mathcal{D})$-program must have the form $G \equiv \exists\overline{U}.P \,\square\, C \,\square\, S \,\square\, \sigma$, where the symbol $\square$ must be interpreted as conjunction, and:

- $\overline{U} =_{def} evar(G)$ is the set of so-called *existential variables* of the goal G. These are intermediate variables, whose bindings may be partial patterns.
- $P \equiv e_1 \to R_1, \ldots, e_n \to R_n$ is a finite conjunction of productions where each R_i is a distinct variable and e_i is an expression or a pair of the form $< \tau, \mathcal{T} >$, where τ is an instance of the pattern in the root of a $cDT(\mathcal{D})$ $\mathcal{T}$. Those productions $e \to R$ whose left hand side e is simply an expression are called *suspensions*, while those whose left hand side is of the form $< \tau, \mathcal{T} >$ are called *demanded productions*. The set of *produced variables* of G is defined as $pvar(P) =_{def} \{R_1, \ldots, R_n\}$ and we define the *production relation* $X \gg_P Y$ iff there is some $1 \leq i \leq n$ such that $X \in var(e_i)$ and $Y \equiv R_i$.
- $C \equiv \delta_1, \ldots, \delta_k$ is a finite conjunction of total constraints (possibly including occurrences of defined function symbols).
- $S \equiv \pi_1, \ldots, \pi_l$ is a finite conjunction of total primitive constraints, called *constraint store*.
- σ is an idempotent substitution called *answer substitution* such that $dom(\sigma) \cap var(P \,\square\, C \,\square\, S) = \emptyset$.

Additionally, any *admissible goal* must satisfy the same admissibility conditions given in [15] about produced variables plus a new admissibility condition for definitional trees:

DT *For each demanded production* $< \tau, \mathcal{T} > \to R$ *in* P*, the variable* R *is demanded (i.e.* $R \in dvar_{\mathcal{D}}(G)$ *in the sense of* Definition 7*), and the variables in* $\mathcal{T}$ *not occur in other place of the goal.*

Similarly to [17,15], $CDNC(\mathcal{D})$ uses a notion of *demanded variable* to deal with lazy evaluation, but now in this work w.r.t. a constraint store, higher-order and definitional trees.

Definition 7. *Let* $G \equiv \exists\overline{U}.\ P \,\square\, C \,\square\, S \,\square\, \sigma$ *be an admissible goal for a given* $COISS(\mathcal{D})$*-program and* $X \in var(G)$*. We say that* X *is a* demanded variable *in* G *iff one of the following cases holds:*

- $X \in dvar_{\mathcal{D}}(S)$ *(see item* 1 *in* Definition 4*).*
- *there exists some suspension* $(X\overline{a}_k \to R) \in P$ *such that* $k > 0$ *and* R *is a demanded variable in* G*.*
- *there exists some demanded production* $(< e,\ \underline{case}\ (\tau, Y,\ [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R) \in P$ *such that* $X = e|_{pos(Y,\tau)}$ *and* R *is a demanded variable in* G*.*

We write $dvar_{\mathcal{D}}(G)$ *(or more precisely* $dvar_{\mathcal{D}}(P \,\square\, S)$*) for the set of demanded variables in the goal* G*.*

An admissible goal $G \equiv \exists\overline{U}.\ P \,\square\, C \,\square\, S \,\square\, \sigma$ is called a *solved goal* iff P and C are empty and S is in $\emptyset$-solved form in the sense of item 2.(a) in *Definition 4*. An *initial goal* can be any admissible goal.

Definition 8. *An* answer *for an admissible goal* $G \equiv \exists\overline{U}.\ P \,\square\, C \,\square\, S \,\square\, \sigma$ *and a given* $COISS(\mathcal{D})$*-program* $\mathcal{P}$*, must have the form* $\Pi \,\square\, \theta$*, where* $\Pi \subseteq PCon(\mathcal{D})$ *is a finite conjunction of total primitive constraints,* $\theta \in Sub_{\perp}(\mathcal{U})$ *is an idempotent substitution such that* $dom(\theta) \cap var(\Pi) = \emptyset$*, and there is some substitution* $\theta' =_{\setminus evar(G)} \theta$ *fulfilling the following three conditions:*

- $\mathcal{P} \vdash_{\mathcal{D}} (P \,\square\, C)\theta' \Leftarrow \Pi$ *in* $CRWL(\mathcal{D})$ *(for demanded productions* $(< \tau, \mathcal{T} > \to R) \in P$*, we consider just* $\mathcal{P} \vdash_{\mathcal{D}} \tau\theta' \to \theta'(R) \Leftarrow \Pi$*, because definitional trees are only used to control the computation),*
- $\Pi \models_{\mathcal{D}} S\theta'$ *(i.e.,* $Sol_{\mathcal{D}}(\Pi) \subseteq Sol_{\mathcal{D}}(S\theta')$ *according to item* 3 *in* Definition 3*),*
- $X\theta' \equiv t\theta'$ *for each binding* $X \mapsto t \in \sigma$*, abbreviated as* $\theta' \in Sol(\sigma)$*.*

We write $Ans_{\mathcal{P}}(G)$ *for the set of all answers for* G*. An answer* $\Pi \,\square\, \theta \in Ans_{\mathcal{P}}(G)$ *is called* trivial *iff* $Unsat_{\mathcal{D}}(\Pi)$ *and* non-trivial *otherwise. Moreover, we can relate goals and answers by extending the notion of solution introduced in* Definition 3*.*

Definition 9. *Let* $G \equiv \exists\overline{U}.\ P \,\square\, C \,\square\, S \,\square\, \sigma$ *be an admissible goal for a given* $COISS(\mathcal{D})$*-program* $\mathcal{P}$*. We say that a valuation* $\mu \in Val_{\perp}(\mathcal{D})$ *is a* solution *of* G *iff* $(\emptyset \,\square\, \mu) \in Ans_{\mathcal{P}}(G)$*. We write* $Sol_{\mathcal{P}}(G)$ *for the set of all solutions for* G*. Analogously, we define the set of solutions for an answer* $\Pi \,\square\, \theta$ *as* $Sol_{\mathcal{D}}(\Pi \,\square\, \theta) =_{def} \{\mu \in Val_{\perp}(\mathcal{D}) \mid \mu \in Sol_{\mathcal{D}}(\Pi) \cap Sol(\theta)\}$*.*

The next new result is useful to prove the main properties about $CDNC(\mathcal{D})$ and shows that $CRWL(\mathcal{D})$'s semantics does not accept an undefined value for demanded variables, identifying demanded and needed computation steps in derivations.

Lemma 1 (Demand Lemma).
If $\Pi \,\square\, \theta$ *is a non-trivial answer of an admissible goal* G *for a* $COISS(\mathcal{D})$*-program and* $X \in dvar_{\mathcal{D}}(G)$ *then* $\theta'(X) \neq \perp$ *for all* $\theta' =_{\setminus evar(G)} \theta$ *given by* Definition 8*.*

Proof. Let $\Pi \,\square\, \theta \in Ans_{\mathcal{P}}(G)$ non-trivial (i.e., $Sol_{\mathcal{D}}(\Pi) \neq \emptyset$). By *Definition 8*, there exits $\Pi \,\square\, \theta' \in Ans_{\mathcal{P}}(G)$ such that $\theta =_{\setminus evar(G)} \theta'$. We prove that $\theta'(X) \neq \perp$ reasoning by induction on the order $\gg_P^+$ (the transitive closure of the production relation (see [15] for details) is well-founded due to the property of non-cycles between produced variables in admissible goals). We consider three cases for $X \in dvar_{\mathcal{D}}(G)$ according to *Definition 7*:

- $X \in dvar_{\mathcal{D}}(S)$:
  We suppose that $\theta'(X) = \bot$ and let $\mu \in Sol_{\mathcal{D}}(\Pi)$. By *Definition 8*, $\Pi \models_{\mathcal{D}} S\theta'$, and then $\theta'\mu \in Sol_{\mathcal{D}}(S)$ and $\mu(\theta'(X)) = \mu(\bot) = \bot$. However, since $X \in dvar_{\mathcal{D}}(S)$ and according to item 1 in *Definition 4*, it follows that $\mu(\theta'(X)) \neq \bot$. Therefore, $\theta'(X) \neq \bot$.
- $(X\overline{a}_k \to R) \in P$ with $k > 0$ and $R \in dvar_{\mathcal{D}}(G)$:
  Since $X \gg^+_P R$ and $R \in dvar_{\mathcal{D}}(G)$, by *induction hypothesis* $\theta'(R) \neq \bot$. By *Definition 8*, $\mathcal{P} \vdash_{\mathcal{D}} \theta'(X)\overline{a_k\theta'} \to \theta'(R) \Leftarrow \Pi$. However, since $k > 0$, it is only possible in $CRWL(\mathcal{D})$ if $\theta'(X) \neq \bot$.
- $X = e|_{pos(Y,\tau)}$, $(< e, \underline{case}(\tau, Y, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R) \in P$ and $R \in dvar_{\mathcal{D}}(G)$:
  In this case, $e = f\overline{e}_n$ with $f \in DF^n$. Since $X = e|_{pos(Y,\tau)}$, it follows that $X \gg^+_P R$. Moreover, since $R \in dvar_{\mathcal{D}}(G)$, by *induction hypothesis* $\theta'(R) \neq \bot$. By *Definition 8* and the rule $\mathbf{DF}_{\mathcal{P}}$ of the $CRWL(\mathcal{D})$ calculus [14], $\mathcal{P} \vdash_{\mathcal{D}} f\ \overline{e_n\theta'} \to \theta'(R) \Leftarrow \Pi$ using $(f\overline{t'}_n \to r \Leftarrow P \,\square\, C) \in [\mathcal{P}]_\bot$ and deductions $\mathcal{P} \vdash_{\mathcal{D}} e_i\theta' \to t'_i \Leftarrow \Pi$ for all $1 \leq i \leq n$. On the other hand, $pos(Y, \tau) = i \cdot p$ with $1 \leq i \leq n$ and $p \in Pos(e_i)$ because $\tau \preceq e$. Due to the form of the definitional tree $\underline{case}$ (see *Definition 5*), t'_i has a passive symbol h_j $(1 \leq j \leq k)$ in the position p. Moreover, there must be only passive symbols above h_j in t'_i. Hence, $t'_i \neq \bot$. Moreover, since $\tau \preceq e$ with $X = e_i|_p$, there must be the same passive symbols and in the same order above $\theta'(X)$ in the position p of $e_i\theta'$. It follows that $\mathcal{P} \vdash_{\mathcal{D}} e_i\theta' \to t'_i \Leftarrow \Pi$ applies the $CRWL(\mathcal{D})$-rule of decomposition **DC** in the $CRWL(\mathcal{D})$ calculus [14] to yield $\mathcal{P} \vdash_{\mathcal{D}} \theta'(X) \to h_j \ldots \Leftarrow \Pi$. We conclude that $\theta'(X) \neq \bot$. □

4.2 The $CDNC(\mathcal{D})$ Calculus

The calculus $CDNC(\mathcal{D})$ consists of a set of transformation rules for admissible goals. Each transformation takes the form $G \Vdash G'$, specifying one of the possible ways of performing one step of goal solving. We write $G \Vdash_R G'$ to indicate that $G \Vdash G'$ by means of the $CDNC(\mathcal{D})$ transformation rule R. Derivations are sequences of $\Vdash$-steps. As in the case of constrained SLD derivations for $CLP(\mathcal{D})$ programs [10], successful derivations will eventually end with a solved goal. Failing derivations (ending with an obviously inconsistent goal ■) and infinite derivations are also possible. Similarly to [15,17], all the goal transformation rules are applied by viewing P and C as sets, rather than sequences.

- The goal transformation rules concerning suspensions $e \to R$ (see *Figure 2*) are designed with the aim of modelling the behavior of constrained lazy narrowing with *sharing* as in the $CLNC(\mathcal{D})$ calculus [15], involving primitive functions, possibly higher-order defined functions and functional variables.
- The goal transformation rules for demanded productions $< e, \mathcal{T} > \to R$ (see *Figure 3*) encode the *needed narrowing strategy* guided by the tree $\mathcal{T}$, in a vein similar to [9,17]: if $\mathcal{T}$ is a $\underline{rule}$ tree, then the transformation **RRA** chooses one of the available rules for rewriting e, introducing appropriate suspensions and constraints in the new goal so that lazy evaluation is ensured. If $\mathcal{T}$ is a $\underline{case}$ tree, one of the transformations **CSS**, **DI** or **DN** can be

SS Simple Suspension $\exists X, \overline{U}.\ t \to X, P \sqcap C \sqcap S \sqcap \sigma \vdash_{\mathbf{SS}}$
$\exists \overline{U}.\ (P \sqcap C \sqcap S)\sigma_0 \sqcap \sigma$ if $t \in Pat(\mathcal{U})$ and $\sigma_0 = \{X \mapsto t\}$.

IM Imitation $\exists X, \overline{U}.\ h\overline{e}_m \to X, P \sqcap C \sqcap S \sqcap \sigma \vdash_{\mathbf{IM}}$
$\exists \overline{X}_m, \overline{U}.\ (\overline{e_m \to X_m}, P \sqcap C \sqcap S)\sigma_0 \sqcap \sigma$
if $h\overline{e}_m \notin Pat(\mathcal{U})$ is passive, $X \in dvar_{\mathcal{D}}(P \sqcap S)$ and $\sigma_0 = \{X \mapsto h\overline{X}_m\}$ with $\overline{X}_m$ new variables such that $h\overline{X}_m \in Pat(\mathcal{U})$.

EL Elimination $\exists X, \overline{U}.\ e \to X, P \sqcap C \sqcap S \sqcap \sigma \vdash_{\mathbf{EL}}$
$\exists \overline{U}.\ P \sqcap C \sqcap S \sqcap \sigma$ if $X \notin var(P \sqcap C \sqcap S \sqcap \sigma)$.

PF Primitive Function $\exists X, \overline{U}.\ p\overline{e}_n \to X, P \sqcap C \sqcap S \sqcap \sigma \vdash_{\mathbf{PF}}$
$\exists \overline{X}_q, X, \overline{U}.\ \overline{e_q \to X_q}, P \sqcap C \sqcap p\overline{t}_n \to! X, S \sqcap \sigma$
if $p \in PF^n$, $X \in dvar_{\mathcal{D}}(P \sqcap S)$, and $\overline{X}_q$ are new variables ($0 \le q \le n$ is the number of $e_i \notin Pat(\mathcal{U})$) such that $t_i \equiv X_j$ ($0 \le j \le q$) if $e_i \notin Pat_{\perp}(\mathcal{U})$ and $t_i \equiv e_i$ otherwise for each $1 \le i \le n$.

DT Definitional Tree $\exists X, \overline{U}.\ f\overline{e}_n \to X, P \sqcap C \sqcap S \sqcap \sigma \vdash_{\mathbf{DT_1}}$
$\exists X, \overline{U}.\ < f\overline{e}_n, \mathcal{T}_{f\overline{X}_n} > \to X, P \sqcap C \sqcap S \sqcap \sigma$
$\exists X, \overline{U}.\ f\overline{e}_n\overline{a}_k \to X, P \sqcap C \sqcap S \sqcap \sigma \vdash_{\mathbf{DT_2}}$
$\exists X, X', \overline{U}.\ < f\overline{e}_n, \mathcal{T}_{f\overline{X}_n} > \to X', X'\overline{a}_k \to X, P \sqcap C \sqcap S \sqcap \sigma$
if $f \in DF^n (k > 0)$, $X \in dvar_{\mathcal{D}}(P \sqcap S)$, and both X' and all variables in $\mathcal{T}_{f\overline{X}_n}$ are new variables.

FV Functional Variable $\exists X, \overline{U}.\ F\overline{e}_q \to X, P \sqcap C \sqcap S \sqcap \sigma \vdash_{\mathbf{FV}}$
$\exists \overline{X}_p, X, \overline{U}.\ (h\overline{X}_p\overline{e}_q \to X, P \sqcap C \sqcap S)\sigma_0 \sqcap \sigma\sigma_0$
if $F \notin pvar(P), q > 0$, $X \in dvar_{\mathcal{D}}(P \sqcap S)$, $\sigma_0 = \{F \mapsto h\overline{X}_p\}$ and $\overline{X}_p$ are new variables such that $h\overline{X}_p \in Pat(\mathcal{U})$.

Fig. 2. $CDNC(\mathcal{D})$-rules for suspensions

applied, according to the kind of symbol occurring in e at the case-distinction position. Otherwise, we fail using **CC** or the computation must be delayed.
– The goal transformation rules concerning constraints (see *Figure 4*) are designed to combine (primitive or user defined) constraints with the action of a constraint solver that fulfill the requirements given in *Definition 4*. Failure rule **SF** is used for failure detection in constraint solving.

The following simple example of goal solving is intended to illustrate the main properties of the $CDNC(\mathcal{D})$ calculus. At each goal transformation step, we underline which subgoal is selected.

Example 3. We compute all the answers from the user defined constraint N /= s (*count* (*from* M)) using the $COISS(\mathcal{H}_{seq})$-program given in *Example 2* and the definitional trees given in *Figure 1*. This example illustrates the use of productions to achieve the effect of a *demand-driven* evaluation with infinite lists and the use of definitional trees for ensuring the efficient choice of demanded redexes.

$\sqcap$ $\underline{N /= s\ (count\ (from\ M))}$ $\sqcap \sqcap\ \varepsilon \vdash_{\mathbf{AC}}$ (*non-primitive constraint*)
$\exists L.\ \underline{s\ (count\ (from\ M)) \to L}$ $\sqcap \sqcap\ N /= L\ \sqcap \vdash_{\mathbf{IM}\{\mathbf{L} \mapsto \mathbf{s\ K}\}}$ (*L is necessary and demanded*)
$\exists K.\ count\ (from\ M) \to K$ $\sqcap \sqcap$ $\underline{N /= s\ K}$ $\sqcap \vdash_{\mathbf{CS}\{\mathbf{K}\}}$ (*K is necessary but not demanded*)

CSS Case Selection

$\exists R, \overline{U}.\ < e, \underline{case}(\tau, X, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) >\!\!\rightarrow R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{CSS}}$
$\exists R, \overline{U}.\ < e, \mathcal{T}_i >\!\!\rightarrow R, P \,\square\, C \,\square\, S \,\square\, \sigma$

if $e|_{pos(X,\tau)} = h_i \ldots$, with $1 \leq i \leq k$ given by e, where h_i is the passive symbol associated to $\mathcal{T}_i$.

DI Demand Instantiation

$\exists R, \overline{U}.\ < e, \underline{case}(\tau, X, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) >\!\!\rightarrow R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{DI}}$
$\exists \overline{Y}_{m_i}, R, \overline{U}.\ (< e, \mathcal{T}_i >\!\!\rightarrow R, P \,\square\, C \,\square\, S)\sigma_0 \,\square\, \sigma\sigma_0$

if $e|_{pos(X,\tau)} = Y$, $Y \notin pvar(P)$, $\sigma_0 = \{Y \mapsto h_i \overline{Y}_{m_i}\}$ with h_i $(1 \leq i \leq k)$ the passive symbol associated to $\mathcal{T}_i$ and $\overline{Y}_{m_i}$ are new variables.

DN Demand Narrowing

$\exists R, \overline{U}.\ < e, \underline{case}(\tau, X, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) >\!\!\rightarrow R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{DN}}$
$\exists R', R, \overline{U}.\ e|_{pos(X,\tau)} \rightarrow R',$
$< e[R']_{pos(X,\tau)}, \underline{case}(\tau, X, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) >\!\!\rightarrow R, P \,\square\, C \,\square\, S \,\square\, \sigma$

if $e|_{pos(X,\tau)} = g \ldots$ with $g \in FS$ active (primitive or defined function symbol) and R' new variable.

RRA Rewrite Rule Application

$\exists R, \overline{U}.\ < e, \underline{rule}(\tau \rightarrow r_1 \Leftarrow P_1 \,\square\, C_1 | \ldots | r_k \Leftarrow P_k \,\square\, C_k) >\!\!\rightarrow R, P \,\square\ C \,\square$
$S \,\square\, \sigma \Vdash_{\mathbf{RRA}}$
$\exists \overline{X}, R, \overline{U}.\ \sigma_f(R_1) \rightarrow R_1, \ldots, \sigma_f(R_m) \rightarrow R_m, r_i\sigma_c \rightarrow R, P_i\sigma_c, P \,\square\, C_i\sigma_c,$
$C \,\square\, S \,\square\, \sigma$

- $\sigma_0 = \sigma_c \uplus \sigma_f$ with $dom(\sigma_0) = var(\tau)$ and $\tau\sigma_0 = e$.
- $\sigma_c =_{def} \sigma \restriction_{dom_c(\sigma_0)}$, where $dom_c(\sigma_0) = \{X \in dom(\sigma_0) \mid \sigma_0(X) \in Pat(\mathcal{U})\}$.
- $\sigma_f =_{def} \sigma \restriction_{dom_f(\sigma_0)}$, where $dom_f(\sigma_0) = \{X \in dom(\sigma_0) \mid \sigma_0(X) \notin Pat(\mathcal{U})\}$ $= \{R_1, \ldots, R_m\}$.
- $\overline{X} \equiv var(\tau \rightarrow r_i \Leftarrow P_i \,\square\, C_i) \backslash dom_c(\sigma_0)$.

CC Case non-Cover

$\exists R, \overline{U}.\ < e, \underline{case}(\tau, X, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) >\!\!\rightarrow R, P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{CC}} \blacksquare$

if $e|_{pos(X,\tau)} = h \ldots$ is a passive symbol and $h \notin \{h_1, \ldots, h_k\}$, where h_i is the passive symbol associated to $\mathcal{T}_i$ $(1 \leq i \leq k)$.

Fig. 3. $CDNC(\mathcal{D})$-rules for demanded productions and failure detection

At this point, a constraint solver over $\mathcal{H}_{seq}$ (see for example [15]) gives two possible alternatives: $Solve^{\mathcal{H}_{seq}}(\{N \mathrel{/=} s\ K\}, \{K\}) = (\square\ \{N \mapsto 0\}) \vee (\{N' \mathrel{/=} K\} \,\square\, \{N \mapsto s\ N'\})$, and there are two possible continuations of the computation, where the use of definitional trees $\mathcal{T}_{from}$, $\mathcal{T}_{count}$ and $\mathcal{T}_{aux}$ in demanded productions guides and avoids *don't know* choices of program rules and failure computations w.r.t. the previous calculus $CLNC(\mathcal{D})$ [15]:

$\exists K.\ \underline{count\ (from\ M) \rightarrow K}\ \square\ \square\ \square\ \{N \mapsto 0\} \Vdash_{\mathbf{EL}}$ (*K is now unnecessary!*)
$\square\ \square\ \square\ \{N \mapsto 0\}$ **computed answer:** $S_1 \,\square\, \sigma_1 \equiv \square\ \{N \mapsto 0\}$

CS Constraint Solving $\exists\overline{U}.\ P \sqcap C \sqcap S \sqcap \sigma \vdash_{\mathbf{CS}\{\chi\}} \exists\overline{Y}_i, \overline{U}.\ (P \sqcap C)\sigma_i \sqcap S_i \sqcap \sigma\sigma_i$

if $\chi = pvar(P)$, S is not χ-solved, $Solve^{\mathcal{D}}(S,\chi) = \bigvee_{i=1}^{k}(S_i \sqcap \sigma_i)$, and $\overline{Y}_i$ are the new variables introduced by the solver in $S_i \sqcap \sigma_i$, for each $1 \leq i \leq k$.

AC Atomic Constraint $\exists\overline{U}.\ P \sqcap\ p\overline{e}_n \rightarrow\!!\ t, C \sqcap S \sqcap \sigma \vdash_{\mathbf{AC}} \exists\overline{X}_q, \overline{U}.\ \overline{e_q \rightarrow X_q}, P \sqcap C \sqcap\ p\overline{t}_n \rightarrow\!!\ t, S \sqcap \sigma$

if $p \in PF^n$, $p\overline{e}_n \rightarrow\!!\ t$ is a constraint, $\overline{X}_q$ are new variables ($0 \leq q \leq n$ is the number of $e_i \notin Pat_\perp(\mathcal{U})$) such that $t_i \equiv X_j$ $(0 \leq j \leq q)$ if $e_i \notin Pat_\perp(\mathcal{U})$ and $t_i \equiv e_i$ otherwise for each $1 \leq i \leq n$.

SF Solving Failure $\exists\overline{U}.\ P \sqcap C \sqcap S \sqcap \sigma \vdash_{\mathbf{SF}\{\chi\}} \blacksquare$

if $\chi = pvar(P)$, S is not χ-solved, and $Solve^{\mathcal{D}}(S,\chi) = \blacktriangle$.

Fig. 4. $CLNC(\mathcal{D})$-rules for constraint solving and failure detection

$\exists K, N'.\ \underline{count\ (from\ M) \rightarrow K} \sqcap \sqcap\ N' /= K \sqcap\ \{N \mapsto s\ N'\} \vdash_{\mathbf{DT}}$ (*K is now demanded!*)

$\exists K, N'.\ \underline{< count\ (from\ M),\ \mathcal{T}_{count} >\!\!\rightarrow K} \sqcap \sqcap\ N' /= K \sqcap\ \{N \mapsto s\ N'\} \vdash_{\mathbf{DN}}$

$\exists K', K, N'.\ \underline{from\ M \rightarrow K'}, < count\ K',\ \mathcal{T}_{count} >\!\!\rightarrow K \sqcap \sqcap\ N' /= K \sqcap\ \{N \mapsto s\ N'\} \vdash_{\mathbf{DT}}$

$\exists K', K, N'.\ \underline{< from\ M,\ \mathcal{T}_{from} >\!\!\rightarrow K'}, < count\ K', \mathcal{T}_{count} >\!\!\rightarrow K \sqcap \sqcap$ (*K′ is demanded*)
$N' /= K \sqcap\ \{N \mapsto s\ N'\} \vdash_{\mathbf{RRA}}$ (*'from' rule application*)

$\exists K', K, N'.\ \underline{[M|from\ (s\ M)] \rightarrow K'}, \underline{< count\ K',\ \mathcal{T}_{count} >\!\!\rightarrow K} \sqcap \sqcap$
$N' /= K \sqcap\ \{N \mapsto s\ N'\} \vdash^{3}_{\mathbf{IM}\{\mathbf{K'} \mapsto [\mathbf{A}|\mathbf{As}]\},\ \mathbf{SS}\{\mathbf{A} \mapsto \mathbf{M}\},\ \mathbf{CSS}}$

$\exists As, K, N'.\ \underline{from\ (s\ M) \rightarrow As}, < count\ [M|As],\ \mathcal{T}_{count\ [\cdot|\cdot]} >\!\!\rightarrow K \sqcap \sqcap$
$N'/= K \sqcap\ \{N \mapsto s\ N'\} \vdash_{\mathbf{DT}}$ (*As is demanded by the definitional tree*)

$\exists As, K, N'.\ \underline{< from\ (s\ M),\ \mathcal{T}_{from} >\!\!\rightarrow As}, < count\ [M|As],\ \mathcal{T}_{count\ [\cdot|\cdot]} >\!\!\rightarrow K \sqcap \sqcap$
$N'/= K \sqcap\ \{N \mapsto s\ N'\} \vdash_{\mathbf{RRA}}$ (*'from' rule application*)

$\exists As, K, N'.\ \underline{[s\ M|from\ (s\ (s\ M))] \rightarrow As}, \underline{< count\ [M|As],\ \mathcal{T}_{count\ [\cdot|\cdot]} >\!\!\rightarrow K} \sqcap \sqcap$
$N' /= K \sqcap\ \{N \mapsto s\ N'\} \vdash^{3}_{\mathbf{IM}\{\mathbf{As} \mapsto [\mathbf{B}|\mathbf{Bs}]\},\ \mathbf{SS}\{\mathbf{B} \mapsto \mathbf{s\ M}\},\ \mathbf{CSS}}$

$\exists Bs, K, N'.\ from\ (s\ (s\ M)) \rightarrow Bs, \underline{< count\ [M, s\ M|Bs],\ \mathcal{T}_{count\ [\cdot,\cdot|\cdot]} >\!\!\rightarrow K} \sqcap \sqcap$
$N'/= K \sqcap\ \{N \mapsto s\ N'\} \vdash_{\mathbf{RRA}}$ (*'count' rule application*)

$\exists R, N'', Bs, K, N'.\ from\ (s\ (s\ M)) \rightarrow Bs, \underline{aux\ R\ N'' \rightarrow K}, count\ [s\ M|Bs] \rightarrow N'' \sqcap$
$seq\ M\ (s\ M) \rightarrow\!!\ R \sqcap N'/= K \sqcap\ \{N \mapsto s\ N'\} \vdash_{\mathbf{DT}}$ (Bs, N'' *are not demanded*)

$\exists R, N'', Bs, K, N'.from\ (s\ (s\ M)) \rightarrow Bs, \underline{< aux\ R\ N'', \mathcal{T}_{aux} >\!\!\rightarrow K}, count\ [s\ M|Bs] \rightarrow N'' \sqcap$
$seq\ M\ (s\ M) \rightarrow\!!\ R \sqcap\ N' /= K \sqcap\ \{N \mapsto s\ N'\} \vdash_{\mathbf{DI}\{\mathbf{R} \mapsto \mathbf{false}\}}$

$\exists N'', Bs, K, N'.\ from\ (s\ (s\ M)) \rightarrow Bs, \underline{< aux\ false\ N'',\ \mathcal{T}_{aux,false} >\!\!\rightarrow K},$
$count\ [s\ M|Bs] \rightarrow N''\ \sqcap M/= s\ M \sqcap N'/= K \sqcap \{N \mapsto s\ N'\} \vdash^{2}_{\mathbf{RRA},\mathbf{SS}\{\mathbf{K} \mapsto \mathbf{s\ 0}\}}$

$\exists N'', Bs, N'.\ \underline{from\ (s\ (s\ M)) \rightarrow Bs}, \underline{count\ [s\ M|Bs] \rightarrow N''} \sqcap\ M /= s\ M \sqcap$
$N' /= s\ 0 \sqcap\ \{N \mapsto s\ N'\} \vdash^{2}_{\mathbf{EL}}$ (N'' *is unnecessary and then Bs is unnecessary!*)

$\exists N'. \sqcap \underline{M/= s\ M} \sqcap N'/= s\ 0 \sqcap \{N \mapsto s\ N'\} \vdash^{2}_{\mathbf{AC},\mathbf{CS}\{\}}$ (*we have again two possibilities*)
$\exists N'. \sqcap \sqcap\ N' /= s\ 0 \sqcap\ \{M \mapsto 0, N \mapsto s\ N'\}$
computed answer: $S_2 \sqcap \sigma_2 \equiv N' /= s\ 0 \sqcap\ \{M \mapsto 0, N \mapsto s\ N'\}$
$\exists M', N'. \sqcap \sqcap\ M' /= M, N' /= s\ 0 \sqcap\ \{M \mapsto s\ M', N \mapsto s\ N'\}$
computed answer: $S_3 \sqcap \sigma_3 \equiv M' /= M, N' /= s\ 0 \sqcap\ \{M \mapsto s\ M',\ N \mapsto s\ N'\}$

4.3 Properties of the $CDNC(\mathcal{D})$ Calculus

In this last subsection, the relationship between the logic $CRWL(\mathcal{D})$ and our goal solving mechanism $CDNC(\mathcal{D})$ is established in the main results of the paper, namely *soundness* and *completeness* of the $CDNC(\mathcal{D})$ calculus w.r.t. $CFLP(\mathcal{D})$'s semantics. To prove both properties we use techniques similar to those used for the $CLNC(\mathcal{D})$ calculus presented in [15]. The following soundness result ensures that computed answers for a goal G are indeed correct answers.

Theorem 1 (Soundness of $CDNC(\mathcal{D})$).
If G_0 is an initial goal and $G_0 \vdash^*_{CDNC(\mathcal{D})} G_n$, where $G_n \equiv \exists \overline{U}.\ \square\ \square\, S \,\square\, \sigma$ is a solved goal, then $S \,\square\, \sigma \in Ans_{\mathcal{P}}(G_0)$.

Completeness of $CDNC(\mathcal{D})$ is based on the following idea: whenever $\Pi \,\square\, \theta \in Ans_{\mathcal{P}}(G)$ 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 $\Pi \,\square\, \theta$". The following completeness result reveals that $CDNC(\mathcal{D})$ is *strongly complete*, i.e. the local choice of the goal transformation rule applied at each step can be a *don't care* choice.

Theorem 2 (Completeness of $CDNC(\mathcal{D})$).
Let G_0 an initial admissible goal and $\Pi_0 \,\square\ \theta_0 \in Ans_{\mathcal{P}}(G_0)$ non-trivial. Then there exist a finite number of derivations ending in solved goals $G_0 \vdash^*_{CDNC(\mathcal{D})} G_i$ $(1 \leq i \leq k)$ such that $Sol_{\mathcal{D}}(\Pi_0 \,\square\ \theta_0) \subseteq \bigcup_{i=1}^{k} Sol_{\mathcal{P}}(G_i)$.

Finally, from the viewpoint of efficiency, and according to our *Demand Lemma*, definitional trees in demanded productions are used for ensuring only needed narrowing steps in the line of [4,2,17]. Then, computations in $CDNC(\mathcal{D})$ are in essence needed narrowing derivations modulo non-deterministic choices between overlapping and constrained program rules. Therefore, our efficient mechanism maintains the optimality properties shown in [4,2,17] guiding (and avoiding) *don't know* choices of constrained program rules by means of definitional trees.

5 Conclusions

We have presented an effective computational model for the integration of constraint logic and functional logic programming by means of a new Constrained Demanded Narrowing Calculus $CDNC(\mathcal{D})$ parameterized by a constraint domain $\mathcal{D}$ and using definitional trees to guide the choice of demanded redexes. We have proved soundness and completeness of the new narrowing calculus, and we have argued that the use of definitional trees leads to efficiency improvements w.r.t. our previous calculus $CLNC(\mathcal{D})$ [15]. These properties renders $CDNC(\mathcal{D})$ adequate as a concrete specification for the implementation of the computational behavior in existing $CFLP(\mathcal{D})$ systems such as *Curry* [8] and $\mathcal{TOY}$ [13,6].

In the near future, we plan to investigate both improvements and applications of the $CFLP(\mathcal{D})$ scheme. Since $CFLP(\mathcal{D})$ assumes only free data constructors, planned improvements include enriching the scheme with algebraic data constructors. Planned applications will focus on practical instances of the $CFLP(\mathcal{D})$

scheme, supporting arithmetic constraints over the real numbers and finite domain ($\mathcal{FD}$) constraints. In particular, we plan to investigate practical constraint solving methods and applications of our resulting language in $\mathcal{TOY}(\mathcal{FD})$ [6].

Last but not least, we are working on *declarative debugging* techniques for $CFLP(\mathcal{D})$-programs, following previous work for functional logic programs [5].

References

1. S. Antoy. Definitional trees. In *Proc. Int. Conf. on Algebraic and Logic Programming (ALP'92)*, volume 632 of Springer LNCS, pp. 143–157, 1992.
2. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of ALP'97*, pages 16-30. Springer LNCS 1298, 1997.
3. S. Antoy. Constructor-based conditional narrowing. In *Proc. PPDP'01*, ACM Press, pp. 199-206, 2001.
4. S. Antoy, R. Echahed, M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4): 776-822, 2000.
5. R. Caballero, M. Rodríguez-Artalejo. $\mathcal{DDT}$: A Declarative Debugging Tool for Functional Logic Languages. In *Proc. of the 7th International Symposium on FLOPS'04*, volume 2998 of Springer LNCS, pp. 70–84, 2004.
6. A. J. Fernández, M. T. Hortalá-González and F. Sáenz Pérez. $\mathcal{TOY}(\mathcal{FD})$. *User's Manual*, October 27, 2003. System available at `http://www.lcc.uma.es/~afdez/cflpfd/`.
7. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming* 19&20, pp. 583–628, 1994.
8. M. Hanus (ed.), *Curry: an Integrated Functional Logic Language*, Version 0.8, April 15, 2003. `http://www-i2.informatik.uni-kiel.de/~curry/`.
9. M. Hanus, C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33-75, 1999.
10. J. Jaffar, M.J. Maher, K. Marriott and P.J. Stuckey. The Semantics of Constraint Logic Programs. *Journal of Logic Programming*, 37 (1-3) pp. 1–46, 1998.
11. R. Loogen, F.J. López-Fraguas, M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In Proc. *Int. Symp. on PLILP'93*, volume 714 of Springer LNCS pp. 184–200, 1993.
12. F.J. López-Fraguas. A General Scheme for Constraint Functional Logic Programming. In *Proc. Int. Conf. on ALP'92*, Springer LNCS 632, pp. 213–227, 1992.
13. F.J. López-Fraguas, J. Sánchez-Hernández. *$\mathcal{TOY}$: A Multiparadigm Declarative System*. Proc. RTA'99, Springer LNCS 1631, pp 244–247, 1999. System available at `http://toy.sourceforge.net`.
14. F.J. López-Fraguas, M. Rodríguez-Artalejo and R. del Vado-Vírseda. Constraint Functional Logic Programming Revisited. In *International Workshop on Rewriting Logic and its Applications WRLA'04*, Elsevier ENTCS series, vol. 117, pp. 5–50, 2005.
15. F.J. López-Fraguas, M. Rodríguez-Artalejo and R. del Vado-Vírseda. A lazy narrowing calculus for declarative constraint programming. In *Prof of the 6th International Conference on PPDP'04*, ACM Press, pp. 43–54, 2004.
16. M. Marin, T. Ida and T. Suzuki. Cooperative Constraint Functional Logic Programming. In *Int. Symposium on IPSE'2000*, pp. 223–230, November 1–2, 2000.
17. R. del Vado Vírseda. A Demand-driven Narrowing Calculus with Overlapping Definitional Trees. *5th Int. Conference on PPDP'03*, ACM Press, pp. 213–227, 2003.