A Concurrent Operational Semantics for Constraint Functional Logic Programming [*]

Rafael del Vado Vírseda, Fernando Pérez Morente, and Marcos Miguel García Toledo

Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid
C. Profesor José García Santesmases, s/n. 28040 Madrid, Spain
`rdelvado@sip.ucm.es fperezmo@fdi.ucm.es mmgarciat@fdi.ucm.es`

Abstract

In this paper we describe a sound and complete concurrent operational semantics for constraint functional logic programming languages which allows to model declarative applications in which the interaction between demand-driven narrowing and constraint solving helps to prune the search space, leading to shorter goal derivations. We encode concurrency into the generic *CFLP*($\mathcal{D}$) scheme, a uniform foundation for the operational semantics of constraint functional logic programming systems parameterized by a constraint solver over the given domain $\mathcal{D}$. In this concurrent version of the *CFLP*($\mathcal{D}$) scheme, goal solving processes can be executed concurrently and cooperate together to perform their specific tasks via demand-driven narrowing and declarative residuation guided by constrained definitional trees, constraint solving, and communication by synchronization on logical variables.

1998 ACM Subject Classification D.1.6 Logic Programming; D.3.3 Language Constructs and Features: Constraints, functions and concurrent programming structures.

Keywords and phrases Constraint logic programming, concurrent logic programming, functional logic programming.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.154

1 Introduction

Multiparadigm logic programming languages and systems [2, 6, 8, 11] aim to integrate the most important declarative programming paradigms, namely *functional programming* (demand-driven rewriting strategies, higher-order facilities, etc.) and *(constraint) logic programming* (goal solving, logical variables, computation with constraints, etc.). The endeavor to extend this declarative combined logic paradigm to a practical language suitable for concurrent executions has stimulated much research over the last two decades, resulting in a large variety of proposals [3, 6, 8]. The aim of this research area is the development of *concurrent functional and constraint logic programming* systems [2, 8] that maintain the balance between expressiveness and declarative reading: abstraction, computations as proofs, amenability to meta-programming, etc. However, the interactions between all these different features are complex, so the design and implementation of a sound and complete theoretical framework of concurrent and constrained multiparadigm logic programming systems is non-trivial.
A common feature of the various approaches is the attempt to define declarative operational models for concurrency within the *Constraint Logic Programming scheme CLP*($\mathcal{D}$) [7], which

[*] This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465), and GPD (UCM-BSCH-GR35/10-A-910502).

replaces the basic computational model of logic programming (i.e., *SLD-resolution* with syntactic *unification*) by *constraint solving* over some *constraint domain* $\mathcal{D}$ (e.g., the integer or the real numbers). The $CLP(\mathcal{D})$ scheme can be generalized into the framework of *concurrent constraint programming* [10] to accommodate a simple and powerful model of declarative concurrent computation based on a *global store*, represented by a constraint on the values that variables can assume. All goal solving processes of the system share this common store, and instead of "reading" and "writing" the values of variables, processes may *ask* (check if a constraint is entailed by the store) and *tell* (augment the store with a new constraint).

The $CFLP(\mathcal{D})$ scheme [9] elegantly captures the fundamental ideas behind the multi-paradigm logic systems, generalizing the $CLP(\mathcal{D})$ scheme to provide uniform foundations for the semantics of functional and constraint logic programming languages. The efficient operational semantics relies on *demand-driven narrowing with definitional trees* [12], a combination of syntactic unification and demand-driven rewriting, parameterized by a constraint solver over the given domain $\mathcal{D}$, which is sound and complete with respect to a declarative semantics formalized by a *constraint rewriting logic* [9], and uses a hierarchical structure called *definitional tree* to efficiently control the computation. The current version of the constraint functional logic system $\mathcal{TOY}$ [11] has been designed to efficiently implement the $CFLP(\mathcal{D})$ scheme. However, concurrency is not supported in the $CFLP(\mathcal{D})$ execution model and its efficient implementation in the $\mathcal{TOY}$ system, although declarative forms of concurrency do exist for similar (but less expressive) approaches [3, 6, 8].

Despite those concurrent extensions, we are not aware of any implementation backed by theoretical results concerning the combination of sound and complete demand-driven narrowing with definitional trees and constraint solving to provide a more powerful declarative integration of concurrent programming techniques. The development of these practical techniques is essential for the implementation of concurrent multiparadigm logic programming systems, since they allow further optimizations related to the synchronization and communication of constraint solving mechanisms that can considerably reduce the search space generated by narrowing.

The aim of this paper is to provide a well-founded concurrent operational semantics that has the potential to be at the basis of more efficient implementations of constraint functional logic programming languages than the current ones [4]. The main contribution of this work is the hybrid operational combination between constraint solving and demand-driven narrowing guided by definitional trees, to show that this concurrent operational model allows constraint solving to efficiently reduce the search space generated by narrowing.

The rest of this paper is organized as follows. Section 2 introduces our approach by presenting an example of declarative concurrency in $CFLP(\mathcal{FD})$, a concrete instance of constraint functional logic programming over the finite domain $\mathcal{FD}$ of integer numbers. In Sections 3 and 4 we introduce and enrich the presentation of the generic $CFLP(\mathcal{D})$ scheme [9] underlying the implementation of the $\mathcal{TOY}$ system [11], now with concurrent features. Finally, Section 5 summarizes some conclusions and plans for future work.

2 An Example of Concurrent Execution in $CFLP(\mathcal{FD})$

For a first impression of our proposal of a concurrent operational model in constraint functional logic programming, we consider the following *conditional* ($\Leftarrow$) *rewriting rules* ($\rightarrow$) with constraints over the *constraint finite domain* $\mathcal{FD}$ of integers defining a simple $CFLP(\mathcal{FD})$-program to compute *Fibonacci numbers*:

$$\begin{array}{lll} \mathit{fib}\,(0) & \rightarrow & 1 \\ \mathit{fib}\,(1) & \rightarrow & 1 \\ \mathit{fib}\,(N) & \rightarrow & \mathit{fib}\,(N-1) + \mathit{fib}\,(N-2) \Leftarrow \quad N \geq 2 \end{array}$$

From this program, we want to compute all the values for the variable X from the *user-defined constraint* $\mathit{fib}\,(X) \leq 2$ (i.e., the values 0, 1 and 2 for X). We propose a concurrent operational semantics to improve the efficiency of the sequential $\mathcal{TOY}(\mathcal{FD})$ system [11] implementing $CFLP(\mathcal{FD})$. This enhanced operational semantics begins separating (we use the symbol '$\Box$' for this purpose) the initial goal $\mathit{fib}\,(X) \leq 2$ into the evaluation of a *function call* $\mathit{fib}\,(X) \rightarrow R$ and a solved *constraint store* with a *primitive constraint* $R \leq 2$, introducing a logical variable R for communication and synchronization between both parts:

$$\mathit{fib}\,(X) \rightarrow R \;\Box\; R \leq 2$$

The new system tries to concurrently evaluate both parts: the function call $\mathit{fib}\,(X) \rightarrow R$ by *demand-driven narrowing* [12] (i.e., a combination of *lazy rewriting* '$\rightarrow$' and *unification* '$\mapsto$' by substitutions) for each of the three (variable-renamed) program rules, and the primitive constraint $R \leq 2$ by the $\mathcal{FD}$*-constraint solver* of *SICStus Prolog* underlying $\mathcal{TOY}(\mathcal{FD})$ [11]:

1. **First program rule:** $\mathit{fib}\,(0) \rightarrow 1$

   In order to evaluate the function call $\mathit{fib}\,(X) \rightarrow R$, the first program rule can be applied to instantiate the argument X to 0 (indicated in the goal by the separation symbol '$\Box$' and the unification substitution $\{X \mapsto 0\}$) and to store the corresponding rewriting result 1 in the logical variable R:

   $$1 \rightarrow R \;\Box\; R \leq 2 \;\Box\; \{X \mapsto 0\}$$

   Now, we can reduce R to 1 and apply the accumulated substitution $\{R \mapsto 1, X \mapsto 0\}$ to instantiate the constraint $R \leq 2$. Then, the $\mathcal{FD}$-constraint solver checks the satisfiability of the instantiated store $1 \leq 2$. Thus, the first answer computed by constrained demand-driven narrowing is $\{X \mapsto 0\}$:

   $\Box\; 1 \leq 2 \;\Box\; \{R \mapsto 1, X \mapsto 0\} \Rightarrow \boxed{\sigma_1 = \{X \mapsto 0\}}$ (*First computed answer*)

2. **Second program rule:** $\mathit{fib}\,(1) \rightarrow 1$

   Concurrently, X can be also instantiated to 1 in our computational model, and then the second program rule can be applied to compute the second answer:

   $1 \rightarrow R \;\Box\; R \leq 2 \;\Box\; \{X \mapsto 1\}$
   $\Box\; 1 \leq 2 \;\Box\; \{R \mapsto 1, X \mapsto 1\} \Rightarrow \boxed{\sigma_2 = \{X \mapsto 1\}}$ (*Second computed answer*)

3. **Third program rule:** $\mathit{fib}\,(X) \rightarrow \mathit{fib}\,(X-1) + \mathit{fib}\,(X-2) \Leftarrow X \geq 2$

   Also concurrently, a variable-renamed variant of the third program rule can be applied to the goal, resulting in the evaluation of two new *fib* function calls:

   $$\mathit{fib}\,(X-1) + \mathit{fib}\,(X-2) \rightarrow R \;\Box\; X \geq 2,\, R \leq 2$$
   $$\mathit{fib}\,(X-1) \rightarrow R_1,\, \mathit{fib}\,(X-2) \rightarrow R_2 \;\Box\; X \geq 2,\, R_1 + R_2 = R,\, R \leq 2$$

   In this third case (3), our enhanced version of the $\mathcal{TOY}(\mathcal{FD})$ system explores concurrently two possible ways to efficiently compute more answers, according to the two possible flows of communication and synchronization (i.e., instantiation of the common logical variables X, R_1, R_2 and R) between the mechanisms of demand-driven narrowing and constraint solving, differing in their length (and therefore in efficiency) due to the different concurrent interleavings of both computational mechanisms.

3.1 From narrowing to constraint solving: In this first case, closer to the sequential execution of the $\mathcal{TOY}(\mathcal{FD})$ system [11], our operational model evaluates concurrently the function calls *fib* $(X - 1)$ and *fib* $(X - 2)$ (or equivalently, the flattened and standardized forms *fib* (N_1) and *fib* (N_2) with new constraints $N_1 = X - 1$ and $N_2 = X - 2$, respectively, in the common constraint store) by applying again a combination of demand-driven narrowing and constraint solving. For example, the system can compute the value $\{X \mapsto 2\}$ for X applying concurrently the second and the first program rules, respectively, to compute $\{N_1 \mapsto 1\}$ and $\{N_2 \mapsto 0\}$, and then applying the constraint solver to $1 = X - 1$ and $0 = X - 2$. Then, the corresponding result 1 will be stored in the logical variables R_1 and R_2:

$1 \to R_1, 1 \to R_2 \,\Box\, X \geq 2, R_1 + R_2 = R,\ R \leq 2, 1 = X-1, 0 = X-2 \,\Box\, \{N_1 \mapsto 1, N_2 \mapsto 0\}$
$1 \to R_1, 1 \to R_2 \,\Box\, R_1 + R_2 = R,\ R \leq 2 \,\Box\, \{N_1 \mapsto 1, N_2 \mapsto 0\ X \mapsto 2\}$

To ensure the consistency of this evaluation process by the demand-driven narrowing computation, our concurrent operational model has to protect (or *suspend*) the evaluation of variables R_1 and R_2 from the action of the constraint solver in favour of an evaluation only by narrowing to compute $\{R_1 \mapsto 1, R_2 \mapsto 1\}$ from $1 \to R_1$ and $1 \to R_2$. Analogously, since we want to compute values $\{N_1 \mapsto 1, N_2 \mapsto 0\}$ for the variables N_1 and N_2 by narrowing, we also need to protect both variables from the action of the constraint solver (this is the so-called *flex* narrowing option in this work). Finally, since both processes are synchronized by sharing the common constraint store that contains $R_1 + R_2 = R$, $R \leq 2$, and we have computed by narrowing the values $\{R_1 \mapsto 1, R_2 \mapsto 1\}$, the constraint solver can compute now the substitution $\{R \mapsto 2\}$ and offer to the user the third computed answer $\{X \mapsto 2\}$:

$\Box\, 1 + 1 = R,\ R \leq 2 \,\Box\, \{N_1 \mapsto 1, N_2 \mapsto 0, X \mapsto 2, R_1 \mapsto 1, R_2 \mapsto 1\}$
$\Box\, 2 \leq 2 \,\Box\, \{N_1 \mapsto 1, N_2 \mapsto 0, X \mapsto 2, R_1 \mapsto 1, R_2 \mapsto 1, R \mapsto 2\} \Rightarrow \boxed{\sigma_3 = \{X \mapsto 2\}}$

At this point, the narrowing computation in $\mathcal{TOY}(\mathcal{FD})$ performs an infinite and useless "trial and error" generation of other possible values for X to find new possible answers. For example, alternatively applying the first and second program rules it is possible to compute other values for N_1 and N_2 due to the concurrent evaluation of *fib* (N_1) and *fib* (N_2): $\{N_1 \mapsto 0, N_2 \mapsto 0\}$, $\{N_1 \mapsto 0, N_2 \mapsto 1\}$ or $\{N_1 \mapsto 1, N_2 \mapsto 1\}$. All of these concurrent processes only obtain inconsistent values for X from $N_1 = X - 1$ and $N_2 = X - 2$ and must be discarded. Moreover, for each application of the third program rule, we have to evaluate two new function calls *fib* in order to infinitely compute concrete values for R_1 and R_2, and to check that each concrete instance of the constraint store $R_1 + R_2 = R$, $R \leq 2$ fails. How can our concurrent operational model efficiently help to prevent this infinite and useless search space generated by narrowing? This is the main idea of our paper:

3.2 From constraint solving to narrowing: In this case, our concurrent operational model needs to protect (or *suspend*) variables N_1 and N_2, now from the narrowing action (this is the so-called *rigid* narrowing or *residuation* option in this work). Then, as an important difference with respect to (3.1), the solver allows to solve the constraint $X \geq 2$ to generate and assign directly to the variables X, $N_1 = X - 1$ and $N_2 = X - 2$ only correct integer values: $\{X \mapsto 2, N_1 \mapsto 1, N_2 \mapsto 0\}$, $\{X \mapsto 3, N_1 \mapsto 2, N_2 \mapsto 1\}$, $\{X \mapsto 4, N_1 \mapsto 3, N_2 \mapsto 2\}$, etc. For each of these possible values, the system creates a process and awakes simple concurrent applications of *rewriting* (instead of expensive "trial and error" applications of narrowing as we have seen in (3.1)). For example, for the values $\{X \mapsto 2, N_1 \mapsto 1, N_2 \mapsto 0\}$ the concurrent system computes the same third answer $\{X \mapsto 2\}$ in less time. For any other value $X \geq 3$, this process is free to use, concurrently, efficient $\mathcal{FD}$-constraint solving techniques [1, 4] to add directly to the constraint store $R_1 + R_2 > 2$. Then, the solver fails checking the extended common constraint store $R_1 + R_2 > 2$, $R_1 + R_2 = R$, $R \leq 2$ and stops the generation of more values for X. Moreover, since the goal solving processes share the same constraint store, the (3.2) option kills automatically all the remaining active processes in the (3.1) option, avoiding the generation of an infinite and useless narrowing computation. In conclusion, in this case constraint solving has helped to efficiently compute the last answer, and at the same time has reduced the search space generated by narrowing.

3 Concurrent Constraint Functional Logic Programming

In this section we give a revised summary of the generic $\mathit{CFLP}(\mathcal{D})$ scheme [9] underlying our proposal of a concurrent system for multiparadigm logic programming.

3.1 Expressions, Patterns, and Constraints

A *signature* is a tuple $\Sigma = \langle DC, FS\rangle$ 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* and *evaluable function symbols*. Evaluable functions can be further classified into domain dependent *primitive functions* $PF^n \subseteq FS^n$ (e.g., $+, \leq\, \in PF^2$) and user *defined functions* $DF^n = FS^n \setminus PF^n$ for each arity $n \in \mathbb{N}$ (e.g., $\mathit{fib} \in DF^1$). We also assume a countably infinite set Var of *variables* $X, Y, \ldots$ and a set $\mathcal{U}$ of *primitive elements* $u, v, \ldots$ (as e.g., the set $\mathbb{Z}$ of integer numbers).

Expressions $e, e' \in \mathit{Exp}(\mathcal{U})$ have the syntax $e ::= \bot \mid u \mid X \mid h \mid (e\,e')$, where $\bot$ is a special symbol in DC^0 to denote an undefined data value, $u \in \mathcal{U}$, $X \in \mathit{Var}$, and $h \in DC \cup FS$. The following classification of expressions is useful: $X\,\overline{e}_m$ with $X \in \mathit{Var}$ 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* if and only if $h \in FS^n$ and $m \geq n$, and *passive* otherwise. The occurrence of a symbol is *passive* if and only if is a primite element $u \in \mathcal{U}$ or is the root symbol h of a passive expression (a symbol used in this sense is called a *passive symbol*). Another class of expressions are *Patterns* $s, t \in \mathit{Pat}(\mathcal{U})$, built as $t ::= \bot \mid u \mid X \mid c\,\overline{t}_m \mid f\,\overline{t}_m$, where $c \in DC^n$ $(m \leq n)$ and $f \in FS^n$ $(m < n)$.

For every expression e, the set of *positions* in e is inductively defined as follows: the empty sequence identifies e itself, and for every expression of the form $h\overline{e}_m$, the sequence $i \cdot q$, where i is a positive integer not greater than m 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 $e[e']_p$. If e is a *linear* expression (without repeated variable occurrences), $\mathit{pos}(X, e)$ will be used for the position of the variable X occurring in e. *Substitutions* $\sigma \in \mathit{Sub}(\mathcal{U})$ are mappings $\sigma : \mathcal{V} \to \mathit{Pat}(\mathcal{U})$ extended homomorphically to $\sigma : \mathit{Exp}(\mathcal{U}) \to \mathit{Exp}(\mathcal{U})$. We define the *domain* $\mathit{Dom}(\sigma)$ of a substitution σ as the collection of variables that are not mapped to themselves.

A *constraint domain* $\mathcal{D}$ provides a set of specific data elements $u \in \mathcal{U}$ along with certain primitive functions $p \in PF$ operating on them. For example, the *constraint finite domain* $\mathcal{FD}$ [4, 9] can be formalized as a structure with carrier set consisting of patterns built from the symbols in a signature Σ and the set of primitive elements $\mathbb{Z}$. Symbols in Σ are intended to represent data constructors (e.g., the list constructors), domain specific primitive functions (e.g., addition and multiplication over $\mathbb{Z}$), and user defined functions. *Constraints* have the syntactic form $p\,\overline{e}_n$, with $p \in PF^n$ a primitive relational symbol and $\overline{e}_n \in \mathit{Exp}(\mathcal{U})$ (e.g., $\mathit{fib}\,(X) \leq 2$, $X \geq 2$ or $R_1 + R_2 = R$ in infix notation).

3.2 Programs and Constrained Definitional Trees

In the sequel, we assume an arbitrarily fixed constraint domain $\mathcal{D}$ built over a set of primitive elements $\mathcal{U}$. In this setting, a *program* is a set of constrained rewrite rules that defines 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 $f\,\bar{t}_n \to r \Leftarrow P \,\square\, C$ (abbreviated as $f\,\bar{t}_n \to r$ if P and C are both empty; see Section 2) and is required to satisfy:

- The left-hand side $f\,\bar{t}_n$ is a linear expression with $\bar{t}_n \in Pat(\mathcal{U})$, and the right-hand side $r \in Exp(\mathcal{U})$.
- P is a finite sequence of so-called *productions* of the form $e_i \to R_i$ $(1 \leq i \leq k)$, intended to be interpreted as a conjunction of local definitions with no cycles [9]. For all $1 \leq i \leq k$, $e_i \in Exp(\mathcal{U})$, and $R_i \notin Var(f\,\bar{t}_n)$ are different variables.
- C is a finite set of constraints, also intended to be interpreted as a conjunction, and possibly including occurrences of user-defined function symbols.

$\mathcal{T}_\tau$ is a *constrained Definitional Tree* over $\mathcal{D}$ ($cDT(\mathcal{D})$ for short) with *call pattern* τ (a linear pattern of the form $f\bar{t}_n$, where $f \in DF^n$ and $\bar{t}_n \in Pat(\mathcal{U})$) if its depth is finite and one of the following cases holds for the rules of a program $\mathcal{P}$:

- $\mathcal{T}_\tau \equiv \underline{rule}(\tau \to r_1 \Leftarrow P_1 \,\square\, C_1 \,\|\, \ldots \,\|\, r_m \Leftarrow P_m \,\square\, C_m)$, where $\tau \to r_i \Leftarrow P_i \,\square\, C_i$ for all $1 \leq i \leq m$ are variants of overlapping program rules in $\mathcal{P}$.
- $\mathcal{T}_\tau \equiv \underline{case}(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k])$, where X is a variable in τ, $op \in \{\mathit{flex}, \mathit{rigid}, \mathit{flex/rigid}\}$, $h_1, \ldots, h_k$ $(k > 0)$ 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 $h_i\overline{Y}_{m_i} \in Pat(\mathcal{U})$.

A $\mathcal{T}_f$ *of a function symbol* $f \in DF^n$ defined by a program $\mathcal{P}$ is a $cDT(\mathcal{D})$ with call pattern $f\overline{X}_n$, where $\overline{X}_n$ are new variables, and the collection of all the program rules obtained from the different $\underline{rule}$ nodes equals, up to variants, the collection of all the program rules defining f in $\mathcal{P}$.

3.3 Goals and Answers

A *goal* G for a program has the general form $P \,\square\, C \,\square\, S \,\square\, \sigma$, where the separation symbol '$\square$' must be interpreted as a conjunction, and:

- $P \equiv e_1 \to R_1, \ldots, e_n \to R_n$ is a finite conjunction of so-called *productions*, where each R_i is a distinct variable and e_i is an expression (we call these productions *suspensions*), or a pair of the form $< \tau, \mathcal{T} >$ with τ an instance of the call pattern in the root of a $cDT(\mathcal{D})$ $\mathcal{T}$ (we call these productions *demanded productions*). The set of *produced variables* is $PVar(P) =_{def} \{R_1, \ldots, R_n\}$ (e.g., R, R_1 and R_2 in Section 2).
- $C \equiv \delta_1, \ldots, \delta_k$ is a finite conjunction of constraints (possibly including user-defined function symbols; e.g., $\mathit{fib}\,(X) \leq 2$ in the initial goal of Section 2).
- $S \equiv \pi_1, \ldots, \pi_l$ is a finite conjunction of *primitive* constraints (i.e., constraints with only pattern arguments; e.g., $R \leq 2$), called *constraint store*.
- $\sigma \in Sub(\mathcal{U})$ is an idempotent substitution called *answer substitution* such that $Dom(\sigma) \cap Var(P \,\square\, C \,\square\, S) = \emptyset$.

A *solved goal* is a goal $\square \,\square\, S \,\square\, \sigma$ in which P and C are empty, and identifies an *answer* $S \,\square\, \sigma$ (or simply σ, as we have seen in Section 2). We say that $X \in Var(G)$ is a *demanded variable* in G if and only if one of the following cases holds:

1. Any substitution that is a solution of S cannot bind X to the undefined value $\perp$ (shortly, $X \in DVar_{\mathcal{D}}(S)$). For example, $R \in DVar_{\mathcal{FD}}(R \leq 3)$.
2. There exists a suspension $(X\overline{a}_k \rightarrow R) \in P$ such that $k > 0$ and R is a demanded variable in G (this case is only necessary to deal with higher-order [9]).
3. There exists a demanded production $(< e, \underline{case}\,(\tau, Y,\ op,\ [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \rightarrow R) \in P$ such that $X = e|_{pos(Y,\tau)}$ and R is a demanded variable in G (see e.g., N_1 and N_2 in (3.1) and (3.2)). If *op* in the branch node is of type *flex*, the variable X is called a *flex variable* (e.g., N_1 and N_2 in (3.1)). Otherwise, the variable X is called a *rigid variable* (e.g., N_1 and N_2 in (3.2)).

4 A Concurrent Operational Semantics for $CFLP(\mathcal{D})$

In this section we present a set of *concurrent goal transformation rules* of the form $G \Vdash_{\mathbf{R}} \|_{i=1}^{k}\ G_i$, specifying all the possible *concurrent evaluations* ($\|_{i=1}^{k}$) of subgoals G_i obtained by applying a rule $\mathbf{R}$ of goal solving ($\Vdash_{\mathbf{R}}$) to a goal G in our concurrent operational semantics for the $CFLP(\mathcal{D})$ scheme. All these rules (formally presented in Figures 1 and 2) have been implemented in the $\mathcal{TOY}$ system [11] and are implicitly applied in our running example of Section 2. We refer the reader to that section for detailed examples illustrating the application of all these rules. We write $G \Vdash^{*} \|_{i=1}^{k}\ G_i$ to represent *concurrent derivations*, given by the successive application ($\Vdash^{*}$) of concurrent goal transformation rules from G. For example, the concurrent derivation $G \Vdash^{*} G_1' \parallel G_{21} \parallel G_{22}$ represents the concurrent goal transformation steps $G \Vdash G_1 \parallel G_2$ with $G_1 \Vdash G_1'$ and $G_2 \Vdash G_{21} \parallel G_{22}$.

Each time a goal G contains the conjunction of two or more atomic statements that could be concurrently evaluated (e.g., two or more productions), our operational model creates concurrent goal solving processes, each of one consisting of an atomic statement from G, together with the necessary information for an adequate and consistent demand-driven evaluation applying a concurrent goal transformation rule (i.e., the sets of produced, demanded, rigid and flex variables). Moreover, for synchronization and in order to properly combine the possible computed answers from subgoal processes, as well as the cases in which processes remain *suspended* (indicated by the symbol $\circlearrowleft$) or *fail* (indicated by the symbol $\blacksquare$), new subgoals must share the constraint store of the main goal G.

4.1 Concurrent Demand-Driven Narrowing and Residuation

We start with a suspension $e \rightarrow R$ representing the computation of a function call, for example $fib\,(X) \rightarrow R$, where e has a user-defined function symbol f in the root (e.g., *fib*) and R is a demanded variable (e.g., by the constraint store $R \leq 2$). Then, the rule **DT** (see Figure 1) is applicable, awakening the suspension $e \rightarrow R$, decorating e with an appropriate $cDT(\mathcal{D})$ $\mathcal{T}_f$ (e.g., $\mathcal{T}_{fib}$ given in Section 3), and introducing a new demanded production $< e, \mathcal{T}_f > \rightarrow R$ into the goal. If the function call is not demanded (i.e., R is not a demanded variable), this computation remains suspended ($\circlearrowleft$) until the variable R disappears from the goal (and then the suspension can be eliminated) or R becomes demanded. The goal transformation rules for demanded productions $< e, \mathcal{T}_f > \rightarrow R$ encode the *demand-driven narrowing strategy* [12] guided by the constrained definitional tree $\mathcal{T}_f$, now in a concurrent setting:

- If $\mathcal{T}_f$ is a $\underline{rule}$ tree, then the transformation **RRA** can be concurrently applied ($\|_{i=1}^{k}$) for each of the k available overlapping program rules for rewriting e, introducing appropriate suspensions and constraints into the new subgoals so that a demand-driven evaluation can be ensured.

DT Definitional Tree

$$f\overline{e}_n \to R, P \Box C \Box S \Box \sigma \Vdash_{\mathbf{DT}} \left\{ \begin{array}{ll} < f\overline{e}_n, \mathcal{T}_{f\overline{X}_n} > \to R, P \Box C \Box S \Box \sigma & \text{if } R \in DVar_{\mathcal{D}}(P \Box S) \\ \circlearrowleft & \text{if } R \notin DVar_{\mathcal{D}}(P \Box S) \end{array} \right\}$$

if $f \in DF^n$, and all variables in $\mathcal{T}_{f\overline{X}_n}$ are new variables.

RRA Rewrite Rule Application

$$< f\overline{e}_n, \underline{rule}\,(f\overline{t}_n \to r_1 \Leftarrow P_1 \Box C_1 \,\|\, \ldots \,\|\, r_k \Leftarrow P_k \Box C_k) > \to R, P \Box C \Box S \Box \sigma \Vdash_{\mathbf{RRA}}$$
$$\|_{i=1}^{k}\ \overline{e_n \to t_n}, r_i \to R, P_i, P \Box C_i, C \Box S \Box \sigma$$

CSS Case Selection

$$< e, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \Box C \Box S \Box \sigma \Vdash_{\mathbf{CSS}} < e, \mathcal{T}_i > \to R, P \Box C \Box S \Box \sigma$$

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

CC Case non-Cover

$$< e, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \Box C \Box S \Box \sigma \Vdash_{\mathbf{CC}} \blacksquare$$

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

DN Demand Narrowing

$$< e, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \Box C \Box S \Box \sigma \Vdash_{\mathbf{DN}}$$
$$e|_{pos(X,\tau)} \to R', < e[R']_{pos(X,\tau)}, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \Box C \Box S \Box \sigma$$

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

DP Demand Produced Variable

$$< e, \underline{case}\,(\tau, X, op, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \Box C \Box S \Box \sigma \Vdash_{\mathbf{DP}} \circlearrowleft$$

if $e|_{pos(X,\tau)} = Y$ with $Y \in PVar(P)$.

DR Demand Residuation

$$< e, \underline{case}\,(\tau, X, rigid, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \Box C \Box S \Box \sigma \Vdash_{\mathbf{DR}} \circlearrowleft$$

if $e|_{pos(X,\tau)} = Y$ with $Y \notin PVar(P)$.

DI Demand Instantiation

$$< e, \underline{case}\,(\tau, X, flex, [\mathcal{T}_1, \ldots, \mathcal{T}_k]) > \to R, P \Box C \Box S \Box \sigma \Vdash_{\mathbf{DI}}$$
$$\|_{i=1}^{k}\ (< e, \mathcal{T}_i > \to R, P \Box C \Box S)\sigma_i \Box \sigma\sigma_i$$

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

Figure 1 Rules for concurrency in constrained demand-driven narrowing and residuation.

- If $\mathcal{T}_f$ is a $\underline{case}$ tree, one of the transformations **CSS**, **CC**, **DN**, **DP**, **DR** or **DI** must be applied, according to the kind of symbol h occurring in e at the case-distinction position $pos(X, \tau)$:
  - If h is a passive symbol h_i, then **CSS** selects the appropriate subtree $\mathcal{T}_i$ (otherwise **CC** fails $\blacksquare$).
  - If h is an active primitive or defined function symbol g, then **DN** introduces a new demanded suspension in the goal to evaluate $e|_{pos(X,\tau)}$.
  - If h is a produced variable Y, the goal must remain suspended ($\circlearrowleft$) using **DP** until a concurrent process of the computation evaluates Y.
  - If Y is a non-produced variable, there are two possibilities:
    - ∗ If the branch node has the option *rigid* (or *flex/rigid*), we must suspend the evaluation ($\circlearrowleft$) using **DR** until the variable has been bound, for example, by the action of the constraint solver (as we have seen in (3.2) for N_1 and N_2, and we

AC Atomic Constraint

$$P \,\square\, p\overline{e}_n, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{AC}} \|_{i=1}^{n}\; e_i \to X_i, P \,\square\, C \,\square\, p\overline{X}_n, S \,\square\, \sigma$$

if $p \in PF^n$, $p\overline{e}_n$ is a constraint, and $\overline{X}_n$ are new variables.

CS Constraint Solving

$$P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{CS}\{\chi\}} \|_{i=1}^{k} \left\{ \begin{array}{ll} (P \,\square\, C)\sigma_i \,\square\, S_i \,\square\, \sigma\sigma_i & \text{if (i) or (ii) or (iii) in Section 4.2} \\ \circlearrowleft & \text{otherwise} \end{array} \right\}$$

if $Solver^{\mathcal{D}}(S, \chi) = \bigvee_{i=1}^{k}(S_i \,\square\, \sigma_i)$ with $\chi =_{def} PVar(P) \cup FVar(P)$.

SF Solving Failure

$$P \,\square\, C \,\square\, S \,\square\, \sigma \Vdash_{\mathbf{SF}\{\chi\}} \blacksquare \qquad \text{if } Solver^{\mathcal{D}}(S, \chi) = \mathit{fail}.$$

Figure 2 Rules for concurrent constraint solving.

will formalize in the next subsection). This case corresponds to the computational principle of declarative *residuation* [6].

∗ If the branch node has the option *flex* (or *flex/rigid*), then **DI** selects concurrently ($\|_{i=1}^{k}$) each subtree $\mathcal{T}_i$, generating an appropriate binding σ_i for Y (as e.g., for N_1 and N_2 in (3.1)).

4.2 Concurrent Constraint Solving

The goal transformation rules concerning *concurrent constraint solving* (see Figure 2) are designed to concurrently combine the evaluation of (primitive or user-defined) constraints with the action of a constraint solver over the given domain. The first rule **AC** evaluates non-primitive constraints $p\overline{e}_n$ (e.g., $\mathit{fib}\,(X) \leq 2$) by performing a concurrent evaluation ($\|_{i=1}^{n}$) of their arguments e_i in suspensions $e_i \to X_i$, and introducing a flattened primitive constraint $p\overline{X}_n$ into the common constraint store, with new logical variables $\overline{X}_n$ for the communication and synchronization among all these concurrent goal solving processes.

For the evaluation of primitive constraints in a constraint domain $\mathcal{D}$ we postulate a *constraint solver* of the form $Solver^{\mathcal{D}}(S, \chi)$, which can reduce any given finite conjunction of primitive constraints S representing the constraint store of the goal into an equivalent simpler solved form. The constraint solver needs to take proper care of a selected set of so-called *critical* (or *protected*) *variables* $\chi =_{def} PVar(P) \cup FVar(P)$ occurring in S to ensure a correct demand-driven evaluation (see variables R_1, R_2 and N_1, N_2 in (3.1) and (3.2) of Section 2). We require that any solver invocation returns a finite disjunction of k simpler solved form alternatives $S_i \,\square\, \sigma_i$. Then, the rule **CS** describes the possible concurrent ($\|_{i=1}^{k}$) evaluations of a single goal by a solver's invocation for each possible alternative solved form computed by the constraint solver. To avoid deadlock situations, we require solvers to have the ability to compute and discriminate a distinction of the following cases and situations for each concurrent solved form alternative (illustrated by (3.1) and (3.2) in Section 2):

(i) A suspended production ($\circlearrowleft$) (e.g., suspended by the **DT** rule) with a non-demanded critical variable at the right-hand side may be now demanded (and then activated) by the new constraint store S_i of some alternative $S_i \,\square\, \sigma_i$ (formally, $DVar_{\mathcal{D}}(S_i) \cap \chi \neq \emptyset$), or

(ii) A suspended demanded production ($\circlearrowleft$) (for example, suspended by the **DR** rule) could be activated by applying σ_i to instantiate a *rigid* and not produced variable in this production (i.e., $Dom(\sigma_i) \cap RVar(P) \neq \emptyset$), or

(iii) A suspended production ($\circlearrowleft$) could be irrelevant for the new constraint store S_i (i.e., $Var(S_i) \cap \chi = \emptyset$) and then has to be eliminated.

For any other situation, the corresponding goal solving process must be suspended (⟳) by the action of the constraint solver. Additionally, the failure rule **SF** is used for failure detection (■) in the constraint solving process.

4.3 Soundness and Completeness

We conclude this section with the main theoretical result of the paper ensuring *soundness* and *completeness* for concurrent *CFLP*($\mathcal{D}$)-derivations w.r.t. the declarative semantics of the *CFLP*($\mathcal{D}$) scheme formalized in [5, 9] by means of a *Constraint Rewriting Logic CRWL*($\mathcal{D}$).

▶ **Theorem 1** (Soundness and Completeness). *Let $S \,\square\, \sigma$ be an answer of G.*

(a) **Soundness:** *If $G \Vdash^* \|_{i=1}^{k} G_i$ is a concurrent derivation from G of a finite number k of goals G_i, for each $G_i \equiv \square \,\square\, S_i \,\square\, \sigma_i$ a solved goal, $S_i \,\square\, \sigma_i$ is an answer of the initial goal G. Formally, $Sol_{\mathcal{D}}\,(G_i) \subseteq Sol_{\mathcal{P}}\,(G)$.*

(b) **Completeness:** *There exists a concurrent derivation $G \Vdash^* \|_{i=1}^{k} G_i$, ending with a finite number k of solved goals $G_i \equiv \square \,\square\, S_i \,\square\, \sigma_i$, that covers all the solutions of the initial answer $S \,\square\, \sigma$. Formally, $Sol_{\mathcal{P}}\,(G) \subseteq \bigcup_{i=1}^{k} Sol_{\mathcal{D}}\,(G_i)$.*

5 Conclusions and Future Work

The set of transformation rules presented in Section 4 provides a *sound* and *complete* operational model to describe a concurrent *CFLP*($\mathcal{D}$) scheme as a novel generalization of the classical *CLP*($\mathcal{D}$) scheme useful for concurrent functional and constraint logic programming.

We are currently investigating other practical instances of constraint domains (e.g., linear and non-linear arithmetic constraints over real numbers) and the cooperative integration of more efficient constraint solving methods into our concurrent system (e.g., based on the *ILOG CP* technology [1] or using declarative modeling languages such as *OPL*).

References

1 I. Castiñeiras and F. Sáenz Pérez. *Integrating ILOG CP Technology into $\mathcal{TOY}$*. In Proc. WFLP'09, pages 27-43, 2009.

2 *Curry*. Available at `http://www-ps.informatik.uni-kiel.de/currywiki/`.

3 R. Echahed and W. Serwe. *Defining Actions in Concurrent Declarative Programming*. In Electr. Notes Theor. Comput. Sci. 64, pages 176-194, 2002.

4 A. J. Fernández et. al. *Constraint functional logic programming over finite domains*. Journal of TPLP 7(5), pages 537–582, 2007.

5 F.J. López Fraguas, M. Rodríguez Artalejo, and R. del Vado. *A Lazy Narrowing Calculus for Declarative Constraint Programming*. In PPDP'04, pages. 43–54, 2004.

6 M. Hanus. *Multiparadigm Declarative Languages*. ICLP'07, pages 45-75, 2007.

7 J. Jaffar and J.L. Lassez. *Constraint logic programming*. POPL'87, pages 111-119, 1987.

8 M. Marin, T. Ida, and W. Schreiner. *CFLP: A Mathematica Implementation of a Distributed Constraint Solving System*. The Math. Journal 8(2), pages 287-300, 2001.

9 F.J. López, M. Rodríguez, and R. del Vado. *A new generic scheme for functional logic programming with constraints*. Journal of HOSC 20 (1-2), pages 73-122, 2007.

10 V. Saraswat and M. Rinard. *Concurrent constraint programming*. POPL'90, pages 232-245.

11 *$\mathcal{TOY}$: A Constraint Functional Logic System*. Available at `toy.sourceforge.net`.

12 R. del Vado. *A demand-driven narrowing calculus with overlapping definitional Trees*. In PPDP 2003, ACM, pages 253-263, 2003.