Journal of Technical Science & Technologies; ISSN: 2298-0032; e-ISSN: 2346-8270; Volume 5, Issue 2, 2016

Application of CLP(SC) in Membrane Computing

Jemal ANTIDZE*
Besik DUNDUA**
Mikheil RUKHAIA***
Lali TIBUA****

Abstract

Present artcile studies semantics of the constraint logic programming built over sequences and contexts, called CLP(SC). Sequences and contexts are constructed over function symbols and function variables which do not have fixed arity, together with term, sequence, and context variables. For some function symbols, the order of the arguments matter (ordered symbols). For some others, this order is irrelevant (unordered symbols). Term variables stand for single terms, sequence variables for sequences, context variables for contexts, and function variables for function symbols. We have studied the semantics of CLP(SC) and showed its application in membrane computing.

Keywords: Constraint Logic Programming, Sequence Context Matching, Membrane Computing.

Introduction

Unranked terms are built over function symbols which do not have a fixed arity (unranked symbols). They are nearly ubiquitous in XML-related applications (Libkin, 2006). They model variadic procedures used in programming languages (Menzel, 2011; Wand, 1987). Moreover, they appear in rewriting (Jacquemard & Rusinowitch, 2008), knowledge representation (Menzel, 2011; ISO/IEC, 2007), theorem proving (Kutsia, 2002; Kutsia & Buchberger, 2004), and program synthesis (Chasseur & Deville, 1997) just to name a few.

When working with unranked terms, it is a pragmatic necessity to consider variables which can be instantiated by a finite sequences of terms (called sequences). Such variables are referred to as unranked variables or sequence variables. We use the latter term in this paper. An example of an unranked term is $f(\bar{x}, f, x, \bar{y})$, where f is an unranked function symbol, $\bar{x}$ and $\bar{y}$ are sequence variables, and x is a usual individual variable which can be instantiated by a single term. We can match this term, e.g., to the term $f(f,a,f,b)$ in two different ways, with the substitutions $\{\bar{x} \mapsto \varepsilon, x \mapsto a, \bar{y} \mapsto (f,b)\}$ and, $\{\bar{x} \mapsto (f,a)\ x \mapsto b, \bar{y} \mapsto \varepsilon\}$ where ε is the empty sequence and (f,a) is a sequence consisting of two terms f and a.

Terms are considered as singleton sequences and sequences, as term sequences, can be concatenated to each other. In this way, sequences can "grow horizontally" and sequence variables help to explore it, filling gaps between siblings. However, such a concatenation has quite a limited power since it does not affect the depth of sequences, and does not permit them "to grow vertically". To address this problem, Bojańczyk and Walukiewicz (2008) introduced so called forest algebras, where alongside sequences (thereby called forests), context also appears. These are sequences with a single hole in some leaf. Contexts can be composed by putting one of them in the hole of the other. Moreover, context can be applied to a sequence by putting it into the hole, resulting into a sequence. One can introduce context variables to stand for such contexts and function variables to stand for function symbols.

Sometimes the order of arguments in terms does not matter. For ranked terms, the corresponding natural equational theory is given by commutativity axioms for the corresponding function symbols. The counterparts of commutative function symbols in the unranked case are called orderless or unordered symbols (Kutsia, 2002). The pro-

*Assoc. Prof. Dr., Institute of Applied Mathematics, Tbilisi State University, & Faculty of Mathematics and Computer Sciences, Sokhumi State University, Tbilisi, Georgia. E-mail: jeantidze@yahoo.com
**Ph. D., Institute of Applied Mathematics, Tbilisi State University, & Faculty of Computer Technologies and Engineering, International Black Sea University, Tbilisi, Georgia. E-mail: bdundua@gmail.com
***Assoc. Prof. Dr., Institute of Applied Mathematics, Tbilisi State University, & Faculty of Computer Technologies and Engineering, International Black Sea University, Tbilisi, Georgia. E-mail: mrukhaia@ibsu.edu.ge
****Ph. D., Faculty of Exact and Natural Sciences, Tbilisi State University, & School of IT, Engineering and Mathematics, University of Georgia, Tbilisi, Georgia. E-mail: ltibua@gmail.com

gramming language of Mathematica (Wolfram, 2003) is an example of a successful application in programming of both syntactic and equational unranked pattern matching (including unordered matching) algorithms with sequence variables.

Our goal was to combine unranked contexts and sequences in a single framework, permitting both ordered and unordered function symbols, to study constraint solving in such a combined theory, and use it in the Constraint Logic Programming schema. Such a language is rich, possesses powerful means to traverse trees both horizontally and vertically in a single or multiple steps, and allows the user to naturally express data structures (e.g., trees, sequences, multisets) and write code concisely.

We restrict CLP programs over sequences and contexts to be well-moded. The well-modedness restriction is quite natural for conditional rule-based programs. Pp Log (Dundua, Kutsia, & Marin, 2009) is one of such experimental systems, which combines contexts and sequences and requires its rules to be well-moded. Hence, our result also shows how to express Pp Log semantics in terms of CLP.

The paper is organized as follows. In Section 2, the notions and the terminology is introduced. In Section 3, we define the interpretation of various syntactic categories and the declarative semantics of the CLP(SC) language. Section 4 describes the operational semantics of CLP(SC). In Section 5 we show how to model P systems in CLP(SC). Finally, in Section 6 we conclude the paper.

Preliminaries

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

- $\mathcal{V}_T$: term variables, denoted by *x,y,z,...*
- $\mathcal{V}_S$: sequence variables, denoted by *x,y,z,...*
- $\mathcal{V}_F$: function variables, denoted by *X,Y,Z,...*
- $\mathcal{V}_C$: context variables, denoted by $X_\bullet, Y_\bullet, Z_\bullet,\dots$
- F_u: unranked unordered function symbols, denoted by f_u, g_u, h_u,...
- F_o: unranked ordered function symbols, denoted by f_o, g_o, h_o,...
- the constant • (the hole),
- *P* : ranked predicate symbols, denoted by *p,q,...*

The sets of variables are countable, while the $\mathcal{A}$ts of function and predicate symbols are finite. In addition, also contains

- The propositional constants *true* and *false*, the binary equality predicates $\doteq$.
- Logical connectives and quantifiers:
  $\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \exists, \forall.$
- Auxiliary symbols: parentheses and the comma.

Function symbols denoted by *f,g,h,...* are elements of the set $F=F_u \cup F_0$ A variable is an element of the set. $V=V_T \cup V_S \cup V_F \cup V_C$ A functor, denoted by *F*, is a common name for a function symbol or a function variable.

We define inductively terms, sequences, and other syntactic categories over as follows:

$$
\begin{array}{ll}
t := x \mid F(S) \mid X_\bullet(t) & \textit{Term} \\
T := t_1,\dots,t_n \quad (n \ge 0) & \textit{Term sequence} \\
\bar{s} := t \mid \bar{x} & \textit{Sequence element} \\
S := \bar{s}_1,\dots,\bar{s}_n \quad (n \ge 0) & \textit{Sequence} \\
C := \bullet \mid F(S.C.S) \mid X_\bullet(C) & \textit{contexts} \\
W := (S.C.S) \mid S &
\end{array}
$$

We denote the set of terms (resp. contexts) by *T(F,v)* (resp., by *C(F,v)*). The sets of ground (variable-free) terms and contexts are denoted, respectively, by *T(F)* and *C(F)*. For readability, we put parentheses around sequences writing, e.g., $(f(a)\ \bar{x}, b)$ instead of $f(a)\ \bar{x}, b$. The empty sequence is written as ε. Besides the letter *t*, we use also *r* and *s* to denote terms. A context *C* may be applied to a term *t* (resp. context *C`*), written *C[t]* (resp. a context *C[C`]*), and the result is the term (resp. context) obtained from *C* by replacing the hole • with *t* (resp. with *C`*).

Primitive constraints are either term equalities $\doteq(t_1,t_2)$ or context equalities $\doteq(C_1,C_2)$ and are written in infix notation, such as $t_1 \doteq t_2$, and $C_1 \doteq C_2$. Sometimes we write $e_1 \doteq e_2$, as a generic notation for both term equalities and context equalities. We denote primitive constraints by *c*.

A mode for an *n*-ary predicate symbol *p* is a function m_p : $\{1,\dots,n\} \rightarrow \{i,o\}$. If $m_p(i)=i$ (resp. $m_p(i)=o$) then the position *i* is called an *input position* (resp. output position) of *p*. We assume that the predicate $\doteq$ has only output positions.

An *atom* is a formula of the form $p(t_1,\dots t_n)$ where $p \in P$ is an *n*-ary predicate symbol. Atoms are denoted by *A*. A literal *L* is an atom or a primitive constraint. For a literal $L = p(t_1,\dots t_n)$ where *P* can be also $\doteq$, we denote by *invar(L)* and *outvar(L)* the sets of variables occurring in terms in the input and output positions of *p*, respectively. We define *invar(true)=Ø, outvar(true)=Ø, invar(false)=Ø, outvar(false)=Ø*

Formulas are defined as usual. *A constraint* is an arbitrary first-order formula built over *true, false,* and primitive constraints. The set of free variables of a syntactic object *O* is denoted by *var(O)*. We let $\exists_V N$ denote the formula $\exists v_1 \dots \exists v_n N$, where $V=\{v_1,\dots v_n\} \subset v$. $\bar{\exists}_V N$ denotes $\exists_{\mathrm{var}(N)/V} N$. We write $\exists N$ (resp. $\forall N$) for the existential (resp. universal) closure of *N*. We refer to a language over the alphabet *A* as *L(A)*.

A substitution is a mapping from term variables to terms, from sequence variables to sequences, from function variables to functors and from context variables to contexts, such that all but finitely many terms, sequences, and function variables are mapped to themselves and all but finitely many context variables are mapped to themselves applied to the hole. Substitutions extend to terms, sequences, contexts, literals, and conjunction of literals.

A (constraint logic) program is a finite set of *rules* of the form $\forall L_1 \wedge \ldots \wedge L_n \to A$, usually written as $A \leftarrow L_1 \ldots L_n$,where A is an atom and $L_1,...,L_n$ are literals $(n \geq 0)$. A goal is a formula of the form $\exists L_1 \wedge \ldots \wedge L_n$, $n \geq 0$, usually written as $L_1,...,L_n$.

A sequence of literals $L_1,...,L_n$ is called *well-moded* if the following conditions hold:

1. For all $1 \leq i \leq n$, *in var* $(L_i) \subseteq \bigcup_{j=1}^{i-1}$ *out* $\mathrm{var}(L_j)$
2. If for some $1 \leq i \leq n$, L_i is $W_1 \doteq W_2$, then
   $\mathrm{var}(W_1) \subseteq \bigcup_{j=1}^{i-1} out\, \mathrm{var}(L_j)$ or
   $\mathrm{var}(W_2) \subseteq \bigcup_{j=1}^{i-1} out\, \mathrm{var}(L_j)$.

A conjunction of literals G is called *well-moded* if there exists a well-moded sequence of literals $L_1,...,L_n$ such that $G = \wedge_{i=1}^{n} L_i$ modulo associativity and commutativity. *A formula in disjunctive normal form (DNF) is well-moded* if each of its disjuncts is well-moded. A state $\langle L_1, \ldots, L_n \| K_1 \vee \cdots \vee K_n \rangle$ is *well-moded*, where *K*'s are conjunctions of primitive constraints, if the formula $(L_1 \wedge \ldots \wedge L_n \wedge K_1) \vee \cdots \vee (L_1 \wedge \ldots \wedge L_n \wedge K_n)$ is well-moded. A *clause* $A \leftarrow L_1, \cdots, L_n$ is *well-moded* if the following two conditions hold:

1. For all $1 \leq i \leq n$,
   $\dot{in}\ \mathrm{var}(L_i) \subseteq \bigcup_{j=1}^{i-1} out\, \mathrm{var}(L_j) \cup \dot{in}\ \mathrm{var}(A)$.
2. $out\, \mathrm{var}(A) \subseteq \bigcup_{j=1}^{n} out\, \mathrm{var}(L_j) \cup \dot{in}\ \mathrm{var}(A)$.
3. If for some $1 \leq i \leq n$, L_i is the equality $W_1 \doteq W_2$, then $\mathrm{var}(W_1) \subseteq \bigcup_{j=1}^{i-1} out\, \mathrm{var}(L_j) \cup \dot{in}\ \mathrm{var}(A)$ or $\mathrm{var}(W_2) \subseteq \bigcup_{j=1}^{i-1} out\, \mathrm{var}(L_j) \cup \dot{in}\ \mathrm{var}(A)$.

A conjunction of primitive constraints $K = c_1 \wedge \ldots \wedge c_n$ is solved if for $1 \leq i \leq n, c_i$ has the form $v \doteq u$ where $v \in \mathcal{V}$ and does not occur neither in *u* nor elsewhere in *K*. *u* is the corresponding language construct: term, sequence, functor, or context. A constraint is solved, if it is either **true** or a non-empty quantifier-free disjunction of solved conjunctions.

Declarative Semantics

Let *S* and *T* be sets. We denote by *S** the set of finite, possibly empty, sequences of the elements of *S*, where ε is the empty sequence, by S^n the set of sequences of length *n* of the elements of *S*, and by T^S the set of functions from *S* to *T*. Given a sequence $s = (s_1, s_2, \ldots, s_n) \in S^n$, we denote by *perm(s)* the set of sequences $\{(s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(n)}) | \pi$ is a permutation of $\{1,2,\ldots,n\}\}$.

A structure *G* for a language *L(A)* is a tuple *{D,I}* made of a non-empty carrier set of individuals and an interpretation function *I* that maps each function symbol $f \in \mathcal{F}$ to a function $I(f): D^* \to D$ and each *n*-ary predicate symbol $p \in \mathcal{P}$ to an *n*-ary relation $I(p) \subseteq D^n$. Given such a structure, we also define the operation $I_c : (D^* \to D) \to D^* \to D^* \to (D \to D) \to (D \to D)$ by $I_c(\psi)(h_1)(h_2)(\varphi)(d) := \psi(h_1, \varphi(d)\ h_2)$ for all $\psi \in D^{D^*}$; $h_1, h_2 \in D^*$; $d \in D$, and $\varphi \in D^D$. Moreover, if $f \in F_u$ then $I(f)(s) = I(f)(s')$ for all $s \in D^*$ and $s' \in$ *perm(s)*. *A variable assignment* for such a structure is a function with domain that maps term variables to elements of *D*; sequence variable to elements of *D**; function variables to functions from *D** to *D*; and context variables to functions from *D* to *D*.

The interpretations of out syntactic categories w.r.t. a structure *G={D,I}* and variable assignment σ is shown below. The interpretations $\|S\|_G^\sigma$ of sequences (including terms) and $\|C\|_G^\sigma$ of contexts are defined as follows:

$$\|x\|_G^\sigma := \sigma(x)$$

$$\|f(S)\|_G^\sigma := I(f)(\|S\|_G^\sigma)$$

$$\|X(S)\|_G^\sigma := \sigma(X)(\|S\|_G^\sigma)$$

$$\|X_\bullet(t)\|_G^\sigma := \sigma(X_\bullet)(\|t\|_G^\sigma)$$

$$\|\bar{x}\|_G^\sigma := \sigma(\bar{x})$$

$$\|(\bar{s}_{1,\ldots,}\bar{s}_n)\|_G^\sigma := (\|(\bar{s}_1)\|_G^\sigma, \ldots, \|\bar{s}_n\|_G^\sigma)$$

$$\|\bullet\|_G^\sigma := Id_D$$

$$\|f(S_1, C, S_2)\|_G^\sigma := I_c(I(f))(\|S_1\|_G^\sigma)(\|S_2\|_G^\sigma)(\|C\|_G^\sigma)$$

$$\|X(S_1, C, S_2)\|_G^\sigma := I_c(\sigma(X))(\|\overline{S_1}\|_G^\sigma)(\|\overline{S_2}\|_G^\sigma)(\|C\|_G^\sigma)$$

$$\|X_\bullet(C)\|_G^\sigma := \sigma(X_\bullet) \circ \|C\|_G^\sigma$$

where $\circ$ denotes function composition and Id_D is the identity function on *D*.

Note that terms are interpreted as elements of *D*, sequences as elements of *D**, and contexts as elements of D^D. We may omit σ and write simply $\|E\|_G$ for the interpretation of a variable-free (i.e., *ground*) expression *E*.

Primitive constraints are interpreted w.r.t. a structure *G* and variable assignment σ as follows: $G|=_\delta t_1 \doteq t_2$ iff $\|t_1\|_G^\sigma = \|t_2\|_G^\sigma$; $G|=_\delta C_1 \doteq C_2$ iff $\|C_1\|_G^\sigma = \|C_2\|_G^\sigma$; and $G|=_\delta p(t_1, \ldots, t_n)$ iff $I(p)(\|t_1\|_G^\sigma, \ldots, \|t_n\|_G^\sigma)$ holds. The notions *G*|=*N* for validity of an arbitrary formula *N* in *G* and |=*N* for validity of *N* in any structure are defined in the standard way.

From now on we identify every context *C* with the function that maps every term *t* to the term *C[t]*. An *intended structure* is a structure *G* with carrier set *TF* and interpretations *I* defined for every $f \in F$ by $I(f)(S) := f(S)$ It follows that $I_c(I(f))(S_1)(S_2) = f(S_1, \bullet, S_2)$ Thus, intended structures identify terms, sequences and contexts by themselves. Other remarkable properties of intended structures *J* are:

$J|=_\vartheta\ t_1 \doteq t_2$ iff $t_1\vartheta = t_2\vartheta$ and $J|=_\vartheta\ C_1 \doteq C_2$ if $C_1\vartheta = C_2\vartheta$.

Given a *CLP(H)* program *P*, its Herbrand base B_p is, naturally, the set of all atoms $p(t_1,...,t_n)$, where *p* is an *n*-ary user-defined predicate in *P* and $(t_1, \ldots, t_n) \in T(F)^n$. Then an intended interpretation of corresponds uniquely to a subset of B_p. An *intended model P* is an intended interpretation of *P* that is its model. We will write shortly H-structure, H-interpretation, H-model for intended structures, interpretations, and models, respectively.

As usual, we will write *P*|=*G* if *G* a goal which holds in every model of *P*. Since the programs considered by us consist of positive clauses, the following facts hold:

1. Every program *P* has a least H model, which we denote by *lm(P,H).*

2. If *G* is a goal then *P*|=*G* iff *lm(P,H)* is a model of *G*.

A ground substitution ϑ is an H-solution (or simply *solution*) of a constraint *C* if *J*|=*C* for all H-structures J. We denote, *J*|=*C* for all H-structures J by $|=_H C$.

Theorem 1. If the constraint *C* is solved, *J*|=*∃C* holds for all intended structures J.

Operational Semantics

In this section we describe the operational semantics of CLP(SC), by following the approach for CLP schema given in (Jaffar, et. al., 1998). A state is a pair *{G||C}*, where *G* is the sequence of literals and C is a quantifier-free constraint. If *G* is an empty sequence, we write 0 instead. The *definition* of an atom $p(t_1,\ldots,t_m)$ in program $P, defn_p(p(t_1,\ldots,t_m))$ is the set of rules in *P* such that the head of each rule has a form $p(r_1,\ldots,r_m)$. We assume that each time returns fresh variants.

A state $\langle L_1,\ldots,L_n \| C\rangle$ can be reduced as follows: Select a literal L_i. Then:

• If L_i is a primitive constraint literal and $solve(C \wedge L_i) \neq false$ then it is reduced to $\langle L_1,\ldots,L_{i-1},L_{i+1},\ldots,L_n \| solve(C \wedge L_i)\rangle$.

• If L_i is a primitive constraint literal and $solve(C \wedge L_i) = false$, then it is reduced to $\langle O \| false\rangle$.

• If L_i is an atom $p(t_1,\ldots,t_m)$, then it is reduced to $\langle L_1,\ldots,L_{i-1},\ t_1 = r_1,\ldots,t_m = r_m, B\, L_{i+1},\ldots,L_n \| C\rangle$ for some $(p(r_1,\ldots,r_m) \leftarrow B) \in defn_p(L_i)$.

• If L_i is a atom and $defn_p(L_i) = \varnothing$ then it is reduced to $\langle 0 \| false\rangle$.

A *derivation from a state S* in a program *P* is a finite or infinite sequence of states $S_0 \mapsto S_1 \mapsto \ldots \mapsto S_n \mapsto \cdots$ where S_0 is *S* and there is a reduction from each S_{1-1} to S_i, using rules in *P*. *A derivation from a goal G* in a program *P* is a derivation from $\langle G \| true\rangle$. The *length P* of a (finite) derivation of the form $S_0 \mapsto S_1 \mapsto \ldots \mapsto S_n$ is *n*. A derivation is *finished* if the last goal cannot be reduced; that is, if its last state is of the form $\langle 0 \| C\rangle$ where *C* is in partially solved form. If *C* is false, the derivation is said to be *failed*.

In (Beriashvili & Dundua, 2016; Dundua, 2014) we have constructed a constraint solving algorithm which simplifies constraints built over the language given in this paper. In particular, the solving algorithm transforms constraints in DNF into constraints in DNF. We have proved, that the constraint solver is sound, terminated and complete for well-moded constraints. Which means, the solver completely solves any well-moded constraint. Based on this result, we can easily prove theorem stating, that well-modedness is preserved by program derivation steps:

Theorem 2. Let *P* be a well-moded CLP(SC) program and $\langle G \| C\rangle$ be a well-moded state. If $\langle G \| C\rangle \mapsto \langle G' \| C'\rangle$ is a reduction using clauses in *P*, then $\langle G' \| C'\rangle$ is also a well-moded state.

An important result for well-moded programs is that any finished derivation from a well-moded goal leads to a solved constraint or to a failure:

Theorem 3. Let $\langle G \| true\rangle \mapsto \ldots \mapsto \langle 0 \| C'\rangle$ be a finished derivation with respect to a well-moded CLP(SC) program, starting from a well-moded goal *G* If $C' \neq false,$ then C` is solved.

Applications

In this section we discuss an application of CLP(SC) in membrane computing (Paun, 2000).

Membrane computing describes the evolution in time of biological processes modeled with *supercells.* A supercell is a structure consisting of a membrane which contains a finite set of objects from a set *U* (the *universe* of investigation) and a finite number of other supercells. Usually, the structure of objects of *U* is irrelevant; therefore, we can model them as terms of the form $f_0()$ where f_0 is an ordered function symbol. Thus, in our framework $U = \{f_0)(|\ f_0 \in F_0\}$. For computational purposes, no order is imposed on the objects of a supercell, nor on the supercells contained in it. Therefore, we can model the content of a supercell as follows:

• We use term $c(f_0^1,\ldots,f_0^m)$ with $c \in F_u$ to denote a supercell content made of elements $f_0^1,\ldots,f_0^n$.

• We use term $f_u(c(f_0^1,\ldots,f_0^m),sc_1,\ldots,sc_n)$ to denote a supercell with membrane f_u, multiset content $\{f_0^1,\ldots,f_0^m\}$ and supercells $sc_1,\ldots,sc_n$ immediately below the membrane f_u.

For membrane computing we adopt the following syntactic notions:

$sc ::= f_u(c(\eta),sc_1,\ldots,sc_n)\ (n \geq 0)$ supercell

$\eta ::= ()|\ (f_\circ,\eta)$ supercell objects

where $f_u \in F_u \setminus \{c\}$ $c \in F_u$, and $f_\circ \in F_\circ$ Note the restriction $f_u \in F_u \setminus \{c\}$ which allows us to distinguish between the multiset of objects of a supercell, and the supercells immediately below the membrane of a supercell. Another restriction not captured by the grammar of supercells, but implicit in membrane computing, is that the membranes of supercells are distinguishable; thus, no element of $F_u \setminus \{c\}$ appears twice in a supercell. The degree of a supercell is the total number of supercells in that supercell, including itself. Note that the degree of a supercell is the number of elements of there set $\{f_u \in F_u |\ f_u \neq c$ and f_u appears *sc}*.

A *P* system is a rewrite-based computational model that performs computations of biological processes modeled with supercells. Suppose *sc* is a supercell of degree *n* with membranes $f_u^1,\ldots,f_u^n$ and outermost membrane f_u^1 (also known as skin). In essence, a *P* system for this supercell is a collection on *n* systems of evolution rules. We denote the system of evolution rules for membrane f_u^1 by R_i· R_i consists of evolution rules of the form $lhs \rightarrow rhs$ where lhs is a string over *U* and rhs is either of the form rhs' or $rhs'\delta$ where is a string over

$$(U \times here,out\}) \cup (U \times \{un_1,\ldots,in_n\}).$$

For such a rule *r*, we define the following auxiliary operations: $e\lim(r) := (f_\circ^{i1}, \ldots, f_\circ^{im})$ if $f_\circ^{i1}, \ldots, f_\circ^{im}$ is the lhs of *r*; $new(r) := (f_\circ^{j1}, \ldots, f_\circ^{p})$ if $\{f_\circ^{jl} \mid 1 \le l \le p\}$ is the muti-set of all objects paired with here in the rhs of *r*; $add(r) := (f_\circ^{k_1}, \ldots, f_\circ^{k_r})$ if $\{f_\circ^{k_1}, \ldots, f_\circ^{k_r}\}$ is the multiset of all objects paired with out in the rhs of *r*; and $add_i(r) := (f_\circ^{l_1}, \ldots, f_\circ^{l_s})$ if $\{f_\circ^{l_1}, \ldots, f_\circ^{l_s}\}$ is the multiset of all objects paired with in_i in the rhs of *r*.

Rule *R* is intended to rewrite the supercell with membrane f_u^1, inside a supercell of form

$F(c(H_1), f_u^i(c(e\lim(r), H_1), sc_{i1}, \ldots, sc_{iq}, H_2), H_3)$

where $\{i_1, \ldots, i_q\} := \{j \in \{1, \ldots, n\} \mid add_j(r) \neq \varnothing$. This pattern detects the content *c(H₁)* of the parent supercell to which new objects must be added, the sequence *e*lim(*r*) of objects of supercell *i* that must go away, and the supercells $sc_{il}, \ldots, sc_{iq}$ comprised by supercell *i* that acquire objects during this rule application.

We model the computation encoded in an evolution rule $r \in R_i$ of supercell *i* as follows.

If f_u^1 is not skin membrane, we distinguish 2 cases.

1. If *i* has no δ in rhs , we produce

$$F(c(\overline{c}), f_u^i(c(e\lim(r), \overline{c'}), sc_{i1}, \ldots, sc_{iq}, \overline{sc'}), \overline{sc}) \rightarrow$$
$$F(c(\overline{c}, add(r)), f_u^i(c(new(r)\overline{c'}), sc_{i1}', \ldots, sc_{iq}', \overline{sc'}), \overline{sc})$$

where $F \in \nu_F, H$ is a sequence, $\overline{c}, \overline{c'}, \overline{sc}, \overline{sc'} \in \nu_S$ and

$$sc_j = f_u^j(c(\overline{c_j}), \overline{sc_j}), \quad sc_j' = f_u^j(add_j(r)\overline{c_j}), \overline{sc_j}),$$

for all $j \in \{i_1, \ldots i_q\}$, where $\overline{c_j}, \overline{sc_j} \in \nu_S$.

2. If *r* has δ in rhs, we produce

$$F(c(\overline{c}), f_u^i(c(e\lim(r), \overline{c'}), sc_{i1}, \ldots, sc_{iq}, \overline{sc'}), \overline{sc}) \rightarrow$$
$$F(c(new(r), add(r), \overline{cc'}), sc_{i1}', \ldots, sc_{iq}', \overline{sc'}, \overline{sc})$$

where $F \in \nu_F, \overline{c}, \overline{c'}, \overline{sc}, \overline{sc'} \in \nu_S$, and all the terms sc_j, sc_j are like in the previous case for all $j \in \{i_1, \ldots, i_q\}$.

The other case to consider is when f_u^1 is the skin membrane. In this case, δ is not allowed to appear in rhs of *r* and our translation yields the rewrite rule

$$f_u^i(c(old(r), \overline{c}), sc_{i1}, \ldots, sc_{iq}, \overline{sc}) \rightarrow$$
$$f_u^i(c(new(r), \overline{c}, sc_{i1_{i1}}', \ldots, sc_{iq_{i1}}', \overline{sc})$$

The rewrite relation induced by R_i on a supercell can be encoded in CLP(SC) by a predicate *rw_i(x,y)* which takes into account the priorities of the rules of R_i. If $R_i = \{lhs_i \rightarrow rhs_i \mid 1 \le i \le n_i\}$, where rules are enumerated in decreasing order of priorities, we can define the predicate *rw_i(x,y)* which denotes the fact that supercell *x* is rewritten to supercell *y* by applying the rule of R_i with highest priority. The definition of *rw* for R_i consists of n_i facts:

$$rw\_i(X_*(lhs_1), X_*(rhs_1))$$
$$rw\_i(X_*(lhs_2), X_*(rhs_2))$$
$$\ldots$$
$$rw\_i(X_*(lhs_{ni}), X_*(rhs_{n_i}))$$

which are tried top-down. Next we can define *rwNF_i(x,y)* which holds if *y* is the normal form of rewriting *x* with rules, in a manner that always selects the applicable rule with highest priority.

$$rwNF\_i(x,z) \leftarrow rw\_i(x,y) rwNF\_i(y,z).$$
$$rwNF\_i(x,x).$$

Conclusion

We have integrated a constraint solving algorithm for equations over terms and contexts into the constraint logic programming schema. The solving algorithm is sound terminating and complete for well-moded constraints and solves equations built over unranked ordered and unordered function symbols. We have studied the declarative and the operational semantics of the derived programming language CLP(SC) and found its applications in membrane computing.

Acknowledgments

This research has been supported by the Shota Rustaveli National Science Foundation under the grants FR/325/4-120/14 and YS15_2.1.2_70.

References

Beriashvili, M., & Dundua, B. (2016). A constraint solver for equations over sequences and contexts. In B. T. Nguyen, T. V. Do, H. A. L. Thi, & N. T. Nguyen (Eds.), Advanced computational methods for knowledge engineering - proceedings of the 4th international conference on computer science, applied mathematics and applications, ICCSAMA 2016, 2-3 may, 2016, vienna, austria (Vol. 453, pp. 115–127). Springer.

Bojanczyk, M., & Walukiewicz, I. (2008). Forest algebras. In J. Flum, E. Gr¨adel, & T. Wilke (Eds.), Logic and automata (Vol. 2, pp. 107-132). Amsterdam University Press.

Boley, H. (1999). A tight, practical integration of relations and functions (Vol. 1712). Springer.

Chasseur, E., & Deville, Y. (1997). Logic program schemas, constraints, and semi-unification. In N. E. Fuchs (Ed.), Lopstr (Vol. 1463, p. 69-89). Springer.

Dundua, B. (2014). Programming with sequence and context variables:foundations and applications (Unpublished doctoral dissertation). Universidade do Porto.

Dundua, B., Kutsia, T., & Marin, M. (2009). Strategies in prholog. In M. Ferna´ndez (Ed.), Proceedings ninth international workshop on reduction strategies in rewriting and programming, WRS 2009, brasilia, brazil, 28th june 2009. (Vol. 15, pp. 32–43).

ISO/IEC. (2007). Information technology—Common Logic (CL): A framework for a family of logic-based languages. International Standard ISO/IEC 24707 (first ed.).

Jacquemard, F., & Rusinowitch, M. (2008). Closure of hedgeautomata languages by hedge rewriting. In A. Voronkov (Ed.), Rta (Vol. 5117, p. 157-171). Springer.

Jaffar, J., Maher, M. J., Marriott, K., & Stuckey, P. J. (1998). The semantics of constraint logic programs. J. Log. Program., 37(1-3), 1-46.

Kutsia, T. (2002). Theorem proving with sequence variables and flexible arity symbols. In M. Baaz & A. Voronkov (Eds.), Lpar (Vol. 2514, p. 278-291). Springer.

Kutsia, T., & Buchberger, B. (2004). Predicate logic with sequence variables and sequence function symbols. In A. Asperti, G. Bancerek, & A. Trybulec (Eds.), Mkm (Vol. 3119, p. 205-219). Springer.

Libkin, L. (2006). Logics for unranked trees: An overview. Logical Methods in Computer Science, 2(3).

Menzel, C. (2011). Knowledge representation, the world wide web, and the evolution of logic. Synthese, 182(2), 269-295.

Paun, G. (2000). Computing with membranes (P systems): A variant. Int. J. Found. Comput. Sci., 11(1), 167–181.

Wand, M. (1987). Complete type inference for simple objects. In Lics (p. 37-44). IEEE Computer Society.

Wolfram, S. (2003). The mathematica book (Fifth ed.). Wolfram-Media.