TLP **16** (2): 141–162, 2016. © Cambridge University Press 2015
doi:10.1017/S1471068415000071 First published online 16 April 2015

CLP(H): *Constraint logic programming for hedges**

BESIK DUNDUA
VIAM, Tbilisi State University, Tbilisi, Georgia and LIACC, University of Porto, Porto, Portugal
(*e-mail:* bdundua@gmail.com)

MÁRIO FLORIDO
DCC-FC and LIACC, University of Porto, Porto, Portugal
(*e-mail:* amf@dcc.fc.up.pt)

TEMUR KUTSIA
RISC, Johannes Kepler University, Linz, Austria
(*e-mail:* kutsia@risc.jku.at)

MIRCEA MARIN
West University of Timişoara, Timişoara, Romania
(*e-mail:* mmarin@info.uvt.ro)

submitted 28 September 2014; revised 07 January 2015; accepted 23 February 2015

Abstract

CLP(H) is an instantiation of the general constraint logic programming scheme with the constraint domain of hedges. Hedges are finite sequences of unranked terms, built over variadic function symbols and three kinds of variables: for terms, for hedges, and for function symbols. Constraints involve equations between unranked terms and atoms for regular hedge language membership. We study algebraic semantics of CLP(H) programs, define a sound, terminating, and incomplete constraint solver, investigate two fragments of constraints for which the solver returns a complete set of solutions, and describe classes of programs that generate such constraints.

KEYWORDS: constraint logic programming, constraint solving, hedges.

1 Introduction

Hedges are finite sequences of unranked terms. These are terms in which function symbols do not have a fixed arity: the same symbol may have a different number of arguments in different places. Manipulation of such expressions has been intensively studied in recent years in the context of XML processing, rewriting, automated reasoning, knowledge representation, just to name a few.

* This research has been partially supported by LIACC through Programa de Financiamento Plurianual of the Fundação para a Ciência e Tecnologia (FCT), by the FCT fellowship (ref. SFRH/BD/62058/2009), by the Austrian Science Fund (FWF) under the project SToUT (P 24087-N18), and the by Rustaveli Science Foundation under the grants DI/16/4-120/11 and FR/611/4-102/12.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

When working with unranked terms, variables that can be instantiated with hedges (hedge variables) are a pragmatic necessity. In (pattern-based) programming, hedge variables help to write neat, compact code. Using them, for instance, one can extract duplicates from a list with just one line of a program. Several languages and formalisms operate on unranked terms and hedges. The programming language of Mathematica (Wolfram 2003) is based on hedge pattern matching. Languages such as Tom (Balland *et al.* 2007), Maude (Clavel *et al.* 2007), ASF+SDF (van den Brand *et al.* 2001) provide capabilities similar to hedge matching (via associative functions). ρLog (Marin and Kutsia 2006) extends logic programming with hedge transformation rules, see also (Marin and Kutsia 2003). XDuce (Hosoya and Pierce 2003) enriches untyped hedge matching with regular expression types. The Constraint Logic Programming schema has been extended to work with hedges in CLP(Flex) (Coelho and Florido 2004), which is a basis for the XML processing language XCentric (Coelho and Florido 2007) and a Web site verification language VeriFLog (Coelho and Florido 2006).

The goal of this paper is to describe a precise semantics of constraint logic programs over hedges. We consider positive CLP programs with two kinds of primitive constraints: equations between hedges, and membership in a hedge regular language. Function symbols are unranked. Predicate symbols have a fixed arity. Terms may contain three kinds of variables: for terms (term variables), for hedges (hedge variables), and for function symbols (function variables). Moreover, we may have function symbols whose argument order does not matter (unordered symbols): a kind of generalization of the commutativity property to unranked terms. As it turns out, such a language is very flexible and permits to write short, yet quite clear and intuitive code: one can see examples in Section 3. We call this language CLP(H), for CLP over hedges. It generalizes CLP(Flex) with function variables, unordered functions, and membership constraints. Hence, as a special case, our paper describes the semantics of CLP(Flex). Moreover, as hedges generalize strings, CLP(H) can be seen also as a generalization of CLP over strings CLP($\mathscr{S}$) (Rajasekar 1994), string processing features of Prolog III (Colmerauer 1990), and CLP over regular sets of strings CLP(Σ^*) (Walinsky 1989).

Note that some of these languages allow an explicit size factor for string variables, restricting the length of strings they can be instantiated with. We do not have size factors, but can express this information easily with constraints. For instance, to indicate the fact that a hedge variable $\overline{x}$ can be instantiated with a hedge of minimal length 1 and maximal length 3, we can write a disjunction $\overline{x} \doteq x \vee \overline{x} \doteq (x_1, x_2) \vee \overline{x} \doteq (x_1, x_2, x_3)$, where the lower case x's are term variables.

Flexibility and the expressive power of CLP(H) has its price: equational constraints with hedge variables, in general, may have infinitely many solutions (Kutsia 2004, 2007). Therefore, any complete equational constraint solving procedure with hedge variables is nonterminating. The solver we describe in this paper is sound and terminating, hence incomplete for arbitrary constraints. However, there are fragments of constraints for which it is complete, i.e., computes all solutions. One such fragment is so called well-moded fragment, where variables in one side of equations (or in the left-hand side of the membership atom) are guaranteed to be instantiated with

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

ground expressions at some point. This effectively reduces constraint solving to hedge matching (Kutsia and Marin 2005a,b), plus some early failure detection rules. Another fragment for which the solver is complete is named after the Knowledge Interchange Format (KIF; Genesereth and Fikes 1992), where hedge variables are permitted only in the last argument positions. We identify forms of CLP(H) programs which give rise to well-moded or KIF constraints.[1]

We can easily model lists with ordered function symbols and multisets with the help of unordered ones. In fact, since we may have several such symbols, we can directly model colored multisets. Constraint solving over lists, sets, and multisets has been intensively studied, see, e.g., (Dovier *et al.* 2008) and references there, and the CLP schema can be extended to accommodate them. In our case, an advantage of using hedge variables in such terms is that hedge variables can give immediate access to collections of subterms via unification. It is very handy in programming.

This paper is an extended and revised version of Dundua *et al.* (2014). It is organized as follows: after establishing the terminology in Section 2, we give two motivating examples in Section 3 to illustrate CLP(H). The algebraic semantics is studied in Section 4. The constraint solver is introduced in Section 5. The operational semantics of CLP(H) is described in Section 6. In Sections 7 and 8, we introduce the well-moded and KIF fragments, respectively. Section 9 contains concluding remarks. The proofs can be found in the online appendix.

2 Preliminaries

For common notation and definitions, we mostly follow (Jaffar *et al.* 1998). The alphabet $\mathscr{A}$ consists of the following pairwise disjoint sets of symbols:

- $\mathscr{V}_{\mathsf{T}}$: term variables, denoted by $x, y, z, \ldots$,
- $\mathscr{V}_{\mathsf{H}}$: hedge variables, denoted by $\overline{x}, \overline{y}, \overline{z}, \ldots$,
- $\mathscr{V}_{\mathsf{F}}$: function variables, denoted by $X, Y, Z, \ldots$,
- $\mathscr{F}_{\mathsf{u}}$: unranked unordered function symbols, denoted by $f_{\mathsf{u}}, g_{\mathsf{u}}, h_{\mathsf{u}}, \ldots$,
- $\mathscr{F}_{\mathsf{o}}$: unranked ordered function symbols, denoted by $f_{\mathsf{o}}, g_{\mathsf{o}}, h_{\mathsf{o}}, \ldots$,
- $\mathscr{P}$: ranked predicate symbols, denoted by $p, q, \ldots$.

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

- The propositional constants true and false, the binary equality predicate $\doteq$, and the unranked membership predicate in.
- Regular operators: eps, $\cdot$, $+$, $*$.
- Logical connectives and quantifiers: $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$, $\exists$, $\forall$.
- Auxiliary symbols: parentheses and the comma.

[1] Conceptually, such an approach can be seen to be similar to, e.g., Miller's approach to higher-order logic programming (Miller 1991), where the fragment L_{λ} uses unitary unification for higher-order patterns instead of undecidable higher-order unification.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

Function symbols, denoted by $f, g, h, \ldots$, are elements of the set $\mathscr{F} = \mathscr{F}_{\mathsf{u}} \cup \mathscr{F}_{\mathsf{o}}$. A *variable* is an element of the set $\mathscr{V} = \mathscr{V}_{\mathsf{T}} \cup \mathscr{V}_{\mathsf{H}} \cup \mathscr{V}_{\mathsf{F}}$. A *functor*, denoted by F, is a common name for a function symbol or a function variable.

We define *terms, hedges,* and other syntactic categories over $\mathscr{A}$ as follows:

$$\begin{array}{rll} t & ::= x \mid f(H) \mid X(H) & \text{Term} \\ T & ::= t_1, \ldots, t_n \quad (n \geqslant 0) & \text{Term sequence} \\ h & ::= t \mid \overline{x} & \text{Hedge element} \\ H & ::= h_1, \ldots, h_n \quad (n \geqslant 0) & \text{Hedge} \end{array}$$

We denote the set of terms by $\mathscr{T}(\mathscr{F}, \mathscr{V})$ and the set of ground (i.e., variable-free) terms by $\mathscr{T}(\mathscr{F})$. Besides the letter t, we use also r and s to denote terms.

We make a couple of conventions to improve readability. The empty hedge is written as ϵ. The terms of the form $a(\epsilon)$ and $X(\epsilon)$ are abbreviated as a and X, respectively. We put parentheses around hedges, writing, e.g., $(f(a), \overline{x}, b)$ instead of $f(a), \overline{x}, b$. For hedges $H = (h_1, \ldots, h_n)$ and $H' = (h'_1, \ldots, h'_{n'})$, the notation (H, H') stands for the hedge $(h_1, \ldots, h_n, h'_1, \ldots, h'_{n'})$.

Two hedges are *disjoint* if they do not share a common element. For instance, $(f(a), x, b)$ and $(f(x), f(b, f(a)))$ are disjoint, whereas $(f(a), x, b)$ and $(f(b), f(a))$ are not, because $f(a)$ is their common element.

An *atom* is a formula of the form $p(t_1, \ldots, t_n)$, where $p \in \mathscr{P}$ is an n-ary predicate symbol. Atoms are denoted by A.

Regular hedge expressions R are defined inductively:

$$\mathsf{R} ::= \mathsf{eps} \mid (\mathsf{R} \cdot \mathsf{R}) \mid \mathsf{R} + \mathsf{R} \mid \mathsf{R}^* \mid f(\mathsf{R})$$

where the dot $\cdot$ stands for concatenation, $+$ for choice, and $*$ for repetition. *Primitive constraints* are either term equalities $\doteq (t_1, t_2)$ or membership for hedges $\mathsf{in}(H, \mathsf{R})$. They are written in infix notation, such as $t_1 \doteq t_2$, and H in R.

A *literal* L is an atom or a primitive constraint. *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 $\exists v_1 \cdots \exists v_n N$, where $V = \{v_1, \ldots, v_n\} \subset \mathscr{V}$. $\overline{\exists}_V N$ denotes $\exists_{var(N) \setminus V} N$. We write $\exists N$ (resp. $\forall N$) for the existential (resp. universal) closure of N. We refer to a language over the alphabet $\mathscr{A}$ as $\mathscr{L}(\mathscr{A})$.

A *substitution* is a mapping from term variables to terms, from hedge variables to hedges, and from function variables to functors, such that all but finitely many variables are mapped to themselves. We use lower case Greek letter to denote them.

For an expression (i.e., a term, hedge, functor, literal, or a formula) e and a substitution σ, we write $e\sigma$ for the *instance* of e under σ. This is a standard operation that replaces in e each free occurrence of a variable v by its image under σ, i.e., by $\sigma(v)$. If needed, bound variables are renamed to avoid variable capture. For instance, for the constraint $\mathscr{C} = \forall x. f(X(a, \overline{x}), \overline{x}) \doteq f(g(\overline{y}, a, b, x), b, x)$ and the substitution $\sigma = \{X \mapsto g, \overline{x} \mapsto (b, x), \overline{y} \mapsto \epsilon, x \mapsto f(c)\}$, we have $\mathscr{C}\sigma = \forall z. f(g(a, b, x), b, x) \doteq$

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

$f(g(a,b,z),b,z)$. A substitution σ is *grounding* for an expression e if $e\sigma$ is a ground expression.

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

We say a variable is *solved* in a conjunction of primitive constraints $\mathscr{K} = \mathbf{c}_1 \wedge \cdots \wedge \mathbf{c}_n$, if there is a $\mathbf{c}_i$, $1 \leqslant i \leqslant n$, such that

- the variable is x, $\mathbf{c}_i = x \doteq t$, and x occurs neither in t nor elsewhere in $\mathscr{K}$, or
- the variable is $\overline{x}$, $\mathbf{c}_i = \overline{x} \doteq H$, and $\overline{x}$ occurs neither in H nor elsewhere in $\mathscr{K}$, or
- the variable is X, $\mathbf{c}_i = X \doteq F$ and X occurs neither in F nor elsewhere in $\mathscr{K}$, or
- the variable is x, $\mathbf{c}_i = x \ \mathsf{in}\ f(\mathsf{R})$ and x does not occur in membership constraints elsewhere in $\mathscr{K}$, or
- the variable is $\overline{x}$, $\mathbf{c}_i = \overline{x} \ \mathsf{in}\ \mathsf{R}$, $\overline{x}$ does not occur in membership constraints elsewhere in $\mathscr{K}$, and R has the form $\mathsf{R}_1 \cdot \mathsf{R}_2$ or R_1^*.

In this case, we also say that $\mathbf{c}_i$ is *solved in* $\mathscr{K}$. Moreover, $\mathscr{K}$ is called *solved* if for any $1 \leqslant i \leqslant n$, $\mathbf{c}_i$ is solved in it. $\mathscr{K}$ is *partially solved*, if for any $1 \leqslant i \leqslant n$, $\mathbf{c}_i$ is solved in $\mathscr{K}$, or has one of the following forms:

- Membership atom:
  - $f_{\mathsf{u}}(H_1, \overline{x}, H_2) \ \mathsf{in}\ f_{\mathsf{u}}(\mathsf{R})$.
  - $(\overline{x}, H) \ \mathsf{in}\ \mathsf{R}$ where $H \neq \epsilon$ and R has the form $\mathsf{R}_1 \cdot \mathsf{R}_2$ or R_1^*.
- Equation:
  - $(\overline{x}, H_1) \doteq (\overline{y}, H_2)$ where $\overline{x} \neq \overline{y}$, $H_1 \neq \epsilon$ and $H_2 \neq \epsilon$.
  - $(\overline{x}, H_1) \doteq (T, \overline{y}, H_2)$, where $\overline{x} \notin var(T)$, $H_1 \neq \epsilon$, and $T \neq \epsilon$. The variables $\overline{x}$ and $\overline{y}$ are not necessarily distinct.
  - $f_{\mathsf{u}}(H_1, \overline{x}, H_2) \doteq f_{\mathsf{u}}(H_3, \overline{y}, H_4)$ where $(H_1, \overline{x}, H_2)$ and $(H_3, \overline{y}, H_4)$ are disjoint.

A constraint is *solved*, if it is either true or a non-empty quantifier-free disjunction of solved conjunctions. A constraint is *partially solved*, if it is either true or a non-empty quantifier-free disjunction of partially solved conjunctions.

3 Motivating examples

In this section we illustrate the expressive power of CLP(H) by two examples: the rewriting of terms from some regular hedge language and an implementation of the recursive path ordering with status.

Example 1

The general rewriting mechanism can be implemented with two CLP(H) clauses: the base case

$$rewrite(x, y) \leftarrow rule(x, y)$$

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

and the recursive case

$$rewrite(X(\overline{x}, x, \overline{y}), X(\overline{x}, y, \overline{y})) \leftarrow rewrite(x, y),$$

where x, y are term variables, $\overline{x}, \overline{y}$ are hedge variables, and X is a function variable. It is assumed that there are clauses which define the *rule* predicate. The base case says that a term x can be rewritten to y if there is a rule which does it. The recursive case rewrites a nondeterministically selected subterm x of the input term to y, leaving the context around it unchanged. Applying the base case before the recursive case gives the outermost strategy of rewriting, while the other way around implements the innermost one.

An example of the definition of the *rule* predicate is

$$rule(X(\overline{x}_1, \overline{x}_2), X(\overline{y})) \leftarrow \ \overline{x}_1 \ \mathsf{in}\ f(a^*) \cdot b^*,\ \overline{x}_1 \doteq (x, \overline{z}),\ \overline{y} \doteq (x, f(\overline{z})),$$

where the constraint[2] $\overline{x}_1$ in $f(a^*) \cdot b^*$ requires $\overline{x}_1$ to be instantiated by hedges from the language generated by the regular hedge expression $f(a^*) \cdot b^*$ (that is, from the language $\{f, f(a), f(a,a), \ldots, (f,b), (f(a),b), \ldots, (f(a,\ldots,a), b, \ldots, b), \ldots\}$).

With this program, the goal $\leftarrow rewrite(f(f(f(a,a),b)), x)$ has two answer substitutions: $\{x \mapsto f(f(f(a,a),f))\}$ and $\{x \mapsto f(f(f(a,a),f(b)))\}$. To obtain them, the goal is first transformed by the recursive clause, leading to the new goal $\leftarrow rewrite(f(f(a,a),b), y)$ together with the constraint $x \doteq f(y)$ for x. The next transformation is performed by the base case of the *rewrite* predicate, resulting into the goal $\leftarrow rule(f(f(a,a),b), y)$. This goal is then transformed by the *rule* clause, which gives the constraint $X(\overline{x}_1, \overline{x}_2) \doteq f(f(a,a),b) \wedge y \doteq X(\overline{y}) \wedge \overline{x}_1$ in $f(a^*) \cdot b^* \wedge \overline{x}_1 \doteq (x', \overline{z}) \wedge \overline{y} \doteq (x', f(\overline{z})) \wedge x \doteq f(y)$. This constraint has two solutions, depending whether $\overline{x}_1$ equals $f(a,a)$ or to $(f(a,a),b)$. From one we get $x \doteq f(f(f(a,a),f))$, and from the other $x \doteq f(f(f(a,a),f(b)))$. These solutions give the above mentioned answers.

Example 2

The recursive path ordering (rpo) $>_{\mathrm{rpo}}$ is a well-known term ordering (Dershowitz 1982) used to prove termination of rewriting systems. Its definition is based on a precedence order $\succ$ on function symbols, and on extensions of $>_{\mathrm{rpo}}$ from terms to tuples of terms. There are two kinds of extensions: lexicographic $>_{\mathrm{rpo}}^{lex}$, when terms in tuples are compared from left to right, and multiset $>_{\mathrm{rpo}}^{mul}$, when terms in tuples are compared disregarding the order. The status function τ assigns to each function symbol either *lex* or *mul* status. Then for all (ranked) terms s, t, we define $s >_{\mathrm{rpo}} t$, if $s = f(s_1, \ldots, s_m)$ and

(1) either $s_i = t$ or $s_i >_{\mathrm{rpo}} t$ for some s_i, $1 \leqslant i \leqslant m$, or
(2) $t = g(t_1, \ldots, t_n)$, $s >_{\mathrm{rpo}} t_i$ for all $i, 1 \leqslant i \leqslant n$, and either
  (a) $f \succ g$, or (b) $f = g$ and $(s_1, \ldots, s_n) >_{\mathrm{rpo}}^{\tau(f)} (t_1, \ldots, t_n)$.

To implement this definition in CLP(H), we use the predicate *rpo* for $>_{\mathrm{rpo}}$ between two terms, and four helper predicates: *rpo_all* to implement the comparison $s >_{\mathrm{rpo}} t_i$

[2] In the notation defined in the previous section, strictly speaking, we need to write this constraint as $f(a(\mathtt{eps})^*) \cdot b(\mathtt{eps})^*$. However, for brevity and clarity of the presentation we omit eps here.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

for all i; *prec* to implement the comparison depending on the precedence; *ext* to implement the comparison with respect to an extension of $>_{\mathrm{rpo}}$; and *status* to give the status of a function symbol. The predicate *lex* implements $>_{\mathrm{rpo}}^{lex}$ and *mul* implements $>_{\mathrm{rpo}}^{mul}$. The symbol $\langle\rangle$ is an unranked function symbol, and $\{\}$ is an unordered unranked function symbol. As one can see, the implementation is rather straightforward and closely follows the definition. $>_{\mathrm{rpo}}$ requires four clauses, since there are four alternatives in the definition:

$$
\begin{aligned}
&1. \quad rpo(X(\overline{x}, x, \overline{y}), x).\\
&\qquad rpo(X(\overline{x}, x, \overline{y}), y) \leftarrow rpo(x, y).\\
&2a. \quad rpo(X(\overline{x}), Y(\overline{y})) \leftarrow rpo\_all(X(\overline{x}), \langle \overline{y} \rangle), prec(X, Y).\\
&2b. \quad rpo(X(\overline{x}), X(\overline{y})) \leftarrow rpo\_all(X(\overline{x}), \langle \overline{y} \rangle), ext(X(\overline{x}), X(\overline{y})).
\end{aligned}
$$

rpo_all is implemented with recursion:

$$
\begin{aligned}
&rpo\_all(x, \langle\,\rangle).\\
&rpo\_all(x, \langle y, \overline{y} \rangle) \leftarrow rpo(x, y), rpo\_all(x, \langle \overline{y} \rangle).
\end{aligned}
$$

The definition of *prec* as an ordering on finitely many function symbols is straightforward. More interesting is the definition of *ext*:

$$
\begin{aligned}
&ext(X(\overline{x}), X(\overline{y})) \leftarrow status(X, lex), lex(\langle \overline{x} \rangle, \langle \overline{y} \rangle).\\
&ext(X(\overline{x}), X(\overline{y})) \leftarrow status(X, mul), mul(\{\overline{x}\}, \{\overline{y}\}).
\end{aligned}
$$

status can be given as a set of facts, *lex* needs one clause, and *mul* requires three:

$$
\begin{aligned}
&lex(\langle \overline{x}, x, \overline{y} \rangle, \langle \overline{x}, y, \overline{z} \rangle) \leftarrow rpo(x, y).\\
&mul(\{x, \overline{x}\}, \{\}).\\
&mul(\{x, \overline{x}\}, \{x, \overline{y}\}) \leftarrow mul(\{\overline{x}\}, \{\overline{y}\}).\\
&mul(\{x, \overline{x}\}, \{y, \overline{y}\}) \leftarrow rpo(x, y),\ mul(\{x, \overline{x}\}, \{\overline{y}\}).
\end{aligned}
$$

That's all. This example illustrates the benefits of all three kinds of variables we have and unordered function symbols.

4 Algebraic semantics

For a given set S, we denote by S^* the set of finite, possibly empty, sequences of elements of S, and by S^n the set of sequences of length n of elements of S. The empty sequence of symbols from any set S is denoted by ϵ. Given a sequence $s = (s_1, s_2, \ldots, s_n) \in S^n$, we denote by $perm(s)$ the set of sequences $\{(s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(n)}) \mid \pi$ is a permutation of $\{1, 2, \ldots, n\}\}$.

A *structure* $\mathfrak{S}$ for a language $\mathscr{L}(\mathscr{A})$ is a tuple $\langle D, I \rangle$ made of a non-empty carrier set of *individuals* and an interpretation function I that maps each function symbol $f \in \mathscr{F}$ to a function $I(f) : D^* \to D$, and each n-ary predicate symbol $p \in \mathscr{P}$ to an n-ary relation $I(p) \subseteq D^n$. Moreover, if $f \in \mathscr{F}_{\mathsf{u}}$ 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

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

domain $\mathscr{V}$ that maps term variables to elements of D, hedge variable to elements of D^*, and function variables to functions from D^* to D.

The interpretations of our syntactic categories w.r.t. a structure $\mathfrak{S} = \langle D, I\rangle$ and variable assignment σ is shown below. The interpretations $[\![H]\!]_{\mathfrak{S},\sigma}$ of hedges (including terms) is defined as follows:

$$
\begin{aligned}
&[\![v]\!]_{\mathfrak{S},\sigma} := \sigma(v), \text{ where } v \in \mathscr{V}_{\mathsf{T}} \cup \mathscr{V}_{\mathsf{H}}.\\
&[\![f(H)]\!]_{\mathfrak{S},\sigma} := I(f)([\![H]\!]_{\mathfrak{S},\sigma}).\\
&[\![X(H)]\!]_{\mathfrak{S},\sigma} := \sigma(X)([\![H]\!]_{\mathfrak{S},\sigma}).\\
&[\![(h_1,\ldots,h_n)]\!]_{\mathfrak{S},\sigma} := ([\![h_1]\!]_{\mathfrak{S},\sigma},\ldots,[\![h_n]\!]_{\mathfrak{S},\sigma}).
\end{aligned}
$$

Note that terms are interpreted as elements of D and hedges as elements of D^*. We may omit σ and write simply $[\![E]\!]_{\mathfrak{S}}$ for the interpretation of a ground expression E. The interpretation of regular expressions is defined as follows:

$$
\begin{aligned}
&[\![\mathsf{eps}]\!]_{\mathfrak{S}} := \{\epsilon\}.\\
&[\![f(\mathsf{R})]\!]_{\mathfrak{S}} := \{I(f)(H) \mid H \in [\![\mathsf{R}]\!]_{\mathfrak{S}}\}.\\
&[\![\mathsf{R}_1 + \mathsf{R}_2]\!]_{\mathfrak{S}} := [\![\mathsf{R}_1]\!]_{\mathfrak{S}} \cup [\![\mathsf{R}_2]\!]_{\mathfrak{S}}.\\
&[\![\mathsf{R}_1 \cdot \mathsf{R}_2]\!]_{\mathfrak{S}} := \{(H_1,H_2) \mid H_1 \in [\![\mathsf{R}_1]\!]_{\mathfrak{S}}, H_2 \in [\![\mathsf{R}_2]\!]_{\mathfrak{S}}\}.\\
&[\![\mathsf{R}^*]\!]_{\mathfrak{S}} := [\![\mathsf{R}]\!]^*_{\mathfrak{S}}.
\end{aligned}
$$

Primitive constraints are interpreted with respect to a structure $\mathfrak{S}$ and variable assignment σ as follows:

$$
\begin{aligned}
&\mathfrak{S} \models_\sigma t_1 \doteq t_2 \text{ iff } [\![t_1]\!]_{\mathfrak{S},\sigma} = [\![t_2]\!]_{\mathfrak{S},\sigma}.\\
&\mathfrak{S} \models_\sigma H \text{ in } \mathsf{R} \text{ iff } [\![H]\!]_{\mathfrak{S},\sigma} \in [\![\mathsf{R}]\!]_{\mathfrak{S}}.\\
&\mathfrak{S} \models_\sigma p(t_1,\ldots,t_n) \text{ iff } I(p)([\![t_1]\!]_{\mathfrak{S},\sigma},\ldots,[\![t_n]\!]_{\mathfrak{S},\sigma}).
\end{aligned}
$$

The notions $\mathfrak{S} \models N$ for validity of an arbitrary formula N in $\mathfrak{S}$, and $\models N$ for validity of N in any structure are defined in the standard way.

An *intended structure* is a structure $\mathfrak{I}$ with the carrier set $\mathscr{T}(\mathscr{F})$ and interpretations I defined for every $f \in \mathscr{F}$ by $I(f)(H) := f(H)$. Thus, intended structures identify terms and hedges by themselves. Also, if R is any regular hedge expression then $[\![\mathsf{R}]\!]_{\mathfrak{I}}$ is the same in all intended structures, and will be denoted by $[\![\mathsf{R}]\!]$. Other remarkable properties of intended structures $\mathfrak{I}$ are: variable assignments are substitutions, $\mathfrak{I} \models_\vartheta t_1 \doteq t_2$ iff $t_1\vartheta = t_2\vartheta$, and $\mathfrak{I} \models_\vartheta H \text{ in } \mathsf{R}$ iff $H\vartheta \in [\![\mathsf{R}]\!]$.

Given a program P, its Herbrand base $\mathscr{B}_P$ is, naturally, the set of all atoms $p(t_1,\ldots,t_n)$, where p is an n-ary user-defined predicate in P and $(t_1,\ldots,t_n) \in \mathscr{T}(\mathscr{F})^n$. Then an intended interpretation of P corresponds uniquely to a subset of $\mathscr{B}_P$. An *intended model* of P is an intended interpretation of P that is its model.

As usual, we will write $P \models G$ if G is a goal which holds in every model of P. Since our programs consist of positive clauses, the following facts hold:

(1) Every program P has a least intended model, which we denote by $lm(P)$.
(2) If G is a goal then $P \models G$ iff $lm(P)$ is a model of G.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

A ground substitution ϑ is an *intended solution* (or simply *solution*) of a constraint $\mathscr{C}$ if $\mathfrak{I} \models \mathscr{C}\vartheta$ for all intended structures $\mathfrak{I}$.

Theorem 1
If the constraint $\mathscr{C}$ is solved, then $\mathfrak{I} \models \exists\mathscr{C}$ holds for all intended structures $\mathfrak{I}$.

5 Solver

In this section, we present a constraint solver for quantifier-free constraints in DNF. It is based on rules, transforming a constraint in *disjunctive normal form* (DNF) into a constraint in DNF. We say a constraint is in DNF, if it has a form $\mathscr{K}_1 \vee \cdots \vee \mathscr{K}_n$, where $\mathscr{K}$'s are conjunctions of true, false, and primitive constraints. The number of rules is not small (as it is usual for such kind of solvers, cf., e.g., (Comon 1998; Dovier *et al.* 2000)). To make their comprehension easier, we group them so that similar ones are collected together in subsections. Within each subsection, for better readability, the rule groups are put between horizontal lines.

Before going into the details, we introduce a more conventional way of writing expressions, some kind of syntactic sugar, that should make reading easier. Instead of $F_1() \doteq F_2()$ and $f_o(H_1) \doteq f_o(H_2)$, we write $F_1 \doteq F_2$ and $H_1 \doteq H_2$, respectively. The symmetric closure of the relation $\doteq$ is denoted by $\simeq$. The rules are applied in any context, i.e., they behave as rewrite rules. Moreover, when a rule applies to a conjunction of the form $L \wedge \mathscr{K}$, it is intended to act on an entire conjunct of the DNF, modulo associativity and commutativity of $\wedge$. These assumptions guarantee that the constraint obtained after each rule application is again in DNF.

5.1 Rules

5.1.1 Logical rules

There are eight logical rules which are applied at any depth in constraints, modulo associativity and commutativity of disjunction and conjunction. N stands for any formula. We denote the whole set of rules by Log.

$N \wedge N \rightsquigarrow N$	$N \vee N \rightsquigarrow N$
$\mathsf{false} \wedge N \rightsquigarrow \mathsf{false}$	$\mathsf{false} \vee N \rightsquigarrow N$
$\mathsf{true} \wedge N \rightsquigarrow N$	$\mathsf{true} \vee N \rightsquigarrow \mathsf{true}$
$H \doteq H \rightsquigarrow \mathsf{true}$	$\epsilon\ \mathsf{in}\ \mathsf{R} \rightsquigarrow \mathsf{true},\ \text{if } \epsilon \in [\![\mathsf{R}]\!]$

5.1.2 Failure rules

The first two rules perform occurrence check, rules (F3) and (F5) detect function symbol clash, and rules (F4), (F6), (F7) detect inconsistent primitive constraints. We denote the set of rules (F1)–(F7) by Fail.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

(F1) $x \simeq (H_1, F(H), H_2) \rightsquigarrow \mathsf{false}$, if $x \in var(H)$.

(F2) $\overline{x} \simeq (H_1, t, H_2) \rightsquigarrow \mathsf{false}$, if $\overline{x} \in var(H_1, t, H_2)$.

(F3) $f_1(H_1) \simeq f_2(H_2) \rightsquigarrow \mathsf{false}$, if $f_1 \neq f_2$.

(F4) $\epsilon \simeq (H_1, t, H_2) \rightsquigarrow \mathsf{false}$.

(F5) $f_1(H)\ \mathsf{in}\ f_2(\mathsf{R}) \rightsquigarrow \mathsf{false}$, if $f_1 \neq f_2$.

(F6) $\epsilon\ \mathsf{in}\ \mathsf{R} \rightsquigarrow \mathsf{false}$, if $\epsilon \notin [\![\mathsf{R}]\!]$.

(F7) $(H_1, t, H_2)\ \mathsf{in}\ \mathsf{eps} \rightsquigarrow \mathsf{false}$.

5.1.3 Decomposition rules

The set of these rules is denoted by Dec. They operate on a conjunction of literals and give back either a conjunction of literals again, or a constraint in DNF.

(D1) $f_{\mathsf{u}}(H) \simeq f_{\mathsf{u}}(T) \wedge \mathscr{K} \rightsquigarrow \bigvee_{T' \in perm(T)} \big(H \doteq T' \wedge \mathscr{K}\big)$,

where H and T are disjoint.

(D2) $(t_1, H_1) \simeq (t_2, H_2) \rightsquigarrow t_1 \doteq t_2 \wedge H_1 \doteq H_2$, where $H_1 \neq \epsilon$ or $H_2 \neq \epsilon$.

5.1.4 Deletion rules

These rules delete identical terms or hedge variables from both sides of an equation. We denote this set of rules by Del.

(Del1) $(\overline{x}, H_1) \simeq (\overline{x}, H_2) \rightsquigarrow H_1 \doteq H_2$.

(Del2) $f_{\mathsf{u}}(H_1, h, H_2) \simeq f_{\mathsf{u}}(H_3, h, H_4) \rightsquigarrow f_{\mathsf{u}}(H_1, H_2) \doteq f_{\mathsf{u}}(H_3, H_4)$.

(Del3) $\overline{x} \simeq (H_1, \overline{x}, H_2) \rightsquigarrow H_1 \doteq \epsilon \wedge H_2 \doteq \epsilon$, if $H_1 \neq \epsilon$.

5.1.5 Variable elimination rules

These rules eliminate variables from the given constraint keeping only a solved equation for them. They apply to disjuncts. The first two rules replace a variable with the corresponding expression, provided that the occurrence check fails.

(E1) $x \simeq t \wedge \mathscr{K} \rightsquigarrow x \doteq t \wedge \mathscr{K}\vartheta$,

where $x \notin var(t)$, $x \in var(\mathscr{K})$ and $\vartheta = \{x \mapsto t\}$. If t is a variable then in addition it is required that $t \in var(\mathscr{K})$.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

(E2) $\overline{x} \simeq H \wedge \mathscr{K} \rightsquigarrow \overline{x} \doteq H \wedge \mathscr{K}\vartheta$,
where $\overline{x} \notin var(H)$, $\overline{x} \in var(\mathscr{K})$, and $\vartheta = \{\overline{x} \mapsto H\}$. If $H = \overline{y}$ for some $\overline{y}$, then in addition it is required that $\overline{y} \in var(\mathscr{K})$.

The next two rules (E3) and (E4) assign to a variable an initial part of the hedge in the other side of the selected equation. The hedge has to be a sequence of terms T in the first rule. The disjunction in the rule is over all possible splits of T. In the second rule, only a split of the prefix T of the hedge is relevant and the disjunction is over all such possible splits of T. The rest is blocked by the term t due to occurrence check: no instantiation of $\overline{x}$ can contain it.

(E3) $(\overline{x}, H) \simeq T \wedge \mathscr{K} \rightsquigarrow \bigvee_{T=(T_1,T_2)} \Big(\overline{x} \doteq T_1 \wedge H\vartheta \doteq T_2 \wedge \mathscr{K}\vartheta\Big)$,
where $\overline{x} \notin var(T)$, $\vartheta = \{\overline{x} \mapsto T_1\}$, and $H \neq \epsilon$.

(E4) $(\overline{x}, H_1) \simeq (T, t, H_2) \wedge \mathscr{K} \rightsquigarrow \bigvee_{T=(T_1,T_2)} \Big(\overline{x} \doteq T_1 \wedge H_1\vartheta \doteq (T_2, t, H_2)\vartheta \wedge \mathscr{K}\vartheta\Big)$
where $\overline{x} \notin var(T)$, $\overline{x} \in var(t)$, $\vartheta = \{\overline{x} \mapsto T_1\}$, and $H_1 \neq \epsilon$.

Finally, there are three rules for function variable elimination. Their behavior is standard:

(E5) $X \simeq F \wedge \mathscr{K} \rightsquigarrow X \doteq F \wedge \mathscr{K}\vartheta$,
where $X \neq F$, $X \in var(\mathscr{K})$, and $\vartheta = \{X \mapsto F\}$. If F is a function variable, then in addition it is required that $F \in var(\mathscr{K})$.

(E6) $X(H_1) \simeq F(H_2) \wedge \mathscr{K} \rightsquigarrow X \doteq F \wedge F(H_1)\vartheta \doteq F(H_2)\vartheta \wedge \mathscr{K}\vartheta$.
where $X \neq F$, $\vartheta = \{X \mapsto F\}$, and $H_1 \neq \epsilon$ or $H_2 \neq \epsilon$.

(E7) $X(H_1) \simeq X(H_2) \wedge \mathscr{K} \rightsquigarrow \bigvee_{f \in \mathscr{F}} \Big(X \doteq f \wedge f(H_1)\vartheta \doteq f(H_2)\vartheta \wedge \mathscr{K}\vartheta\Big)$,
where $\vartheta = \{X \mapsto f\}$, and $H_1 \neq H_2$.

We denote the set of rules (E1)–(E7) by Elim. Note that the assumption of finiteness of $\mathscr{F}$ guarantees that the disjunction in (E7) is finite.

5.1.6 Membership rules

The membership rules apply to disjuncts of constraints in DNF, to preserve the DNF structure. They provide the membership check, if the hedge H in the membership atom H in R is ground. Nonground hedges require more special treatment as one can see.

To solve membership constraints for hedges of the form (t, H) with t a term, we rely on the possibility to compute the linear form of a regular expression, that is,

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

to express it as a finite sum of concatenations of regular hedge expressions that identify all plausible membership constraints for t and H. Formally, the *linear form* of a regular expression R, denoted $lf(\mathsf{R})$, is a finite set of pairs $(f(\mathsf{R}_1), \mathsf{R}_2)$, which is defined recursively as follows:

$$
\begin{aligned}
&lf(\mathtt{eps}) = \emptyset.\\
&lf(f(\mathsf{R})) = \{(f(\mathsf{R}), \mathtt{eps})\}.\\
&lf(\mathsf{R}_1 + \mathsf{R}_2) = lf(\mathsf{R}_1) \cup lf(\mathsf{R}_2).\\
&lf(\mathsf{R}_1 \cdot \mathsf{R}_2) = lf(\mathsf{R}_1) \odot \mathsf{R}_2, \text{ if } \epsilon \notin [\![\mathsf{R}_1]\!].\\
&lf(\mathsf{R}_1 \cdot \mathsf{R}_2) = lf(\mathsf{R}_1) \odot \mathsf{R}_2 \cup lf(\mathsf{R}_2), \text{ if } \epsilon \in [\![\mathsf{R}_1]\!].\\
&lf(\mathsf{R}^*) = lf(\mathsf{R}) \odot \mathsf{R}^*.
\end{aligned}
$$

These equations involve an extension of concatenation $\odot$ that acts on a linear form and a regular expression and returns a linear form. It is defined as $l \odot \mathtt{eps} = l$, and $l \odot \mathsf{R} = \{(f(\mathsf{R}_1), \mathsf{R}_2 \cdot \mathsf{R}) \mid (f(\mathsf{R}_1), \mathsf{R}_2) \in l, \mathsf{R}_2 \neq \mathtt{eps}\} \cup \{(f(\mathsf{R}_1), \mathsf{R}) \mid (f(\mathsf{R}_1), \mathtt{eps}) \in l\}$, if $\mathsf{R} \neq \mathtt{eps}$.

The linear form $lf(\mathsf{R})$ of a regular expression R has the property (Antimirov 1996):[3]

$$
[\![\mathsf{R}]\!] \setminus \{\epsilon\} = \bigcup_{(f(\mathsf{R}_1),\mathsf{R}_2)\in lf(\mathsf{R})} [\![f(\mathsf{R}_1) \cdot \mathsf{R}_2]\!], \tag{LF}
$$

which justifies its use in the rule M2 below.

The first group of membership rules looks as follows:

(M1) $(\overline{x}_1, \ldots, \overline{x}_n)$ in eps $\wedge\, \mathscr{K} \rightsquigarrow \wedge_{i=1}^{n}\, \overline{x}_i \doteq \epsilon \wedge \mathscr{K}\vartheta$,
where $\vartheta = \{\overline{x}_1 \mapsto \epsilon, \ldots, \overline{x}_n \mapsto \epsilon\}, n > 0$.

(M2) (t, H) in R $\wedge\, \mathscr{K} \rightsquigarrow \bigvee_{(f(\mathsf{R}_1),\mathsf{R}_2)\in lf(\mathsf{R})} \Big(t \text{ in } f(\mathsf{R}_1) \wedge H \text{ in } \mathsf{R}_2 \wedge \mathscr{K}\Big)$,
where $H \neq \epsilon$ and $\mathsf{R} \neq \mathtt{eps}$.

(M3) $(\overline{x}, H)$ in $f(\mathsf{R}) \wedge \mathscr{K} \rightsquigarrow$
$\Big(\overline{x} \text{ in } f(\mathsf{R}) \wedge H \doteq \epsilon \wedge \mathscr{K}\Big) \vee \Big(\overline{x} \doteq \epsilon \wedge H \text{ in } f(\mathsf{R}) \wedge \mathscr{K}\Big)$,
where $H \neq \epsilon$.

(M4) t in $\mathsf{R}^* \rightsquigarrow t$ in R.

(M5) t in $\mathsf{R}_1 \cdot \mathsf{R}_2 \wedge \mathscr{K} \rightsquigarrow \Big(t \text{ in } \mathsf{R}_1 \wedge \epsilon \text{ in } \mathsf{R}_2 \wedge \mathscr{K}\Big) \vee \Big(\epsilon \text{ in } \mathsf{R}_1 \wedge t \text{ in } \mathsf{R}_2 \wedge \mathscr{K}\Big)$.

(M6) t in $\mathsf{R}_1 + \mathsf{R}_2 \wedge \mathscr{K} \rightsquigarrow \Big(t \text{ in } \mathsf{R}_1 \wedge \mathscr{K}\Big) \vee \Big(t \text{ in } \mathsf{R}_2 \wedge \mathscr{K}\Big)$.

(M7) $(\overline{x}, H)$ in $\mathsf{R}_1 + \mathsf{R}_2 \wedge \mathscr{K} \rightsquigarrow \Big((\overline{x}, H) \text{ in } \mathsf{R}_1 \wedge \mathscr{K}\Big) \vee \Big((\overline{x}, H) \text{ in } \mathsf{R}_2 \wedge \mathscr{K}\Big)$.

(M8) v in $\mathsf{R}_1 \wedge v$ in $\mathsf{R}_2 \rightsquigarrow v$ in R,

[3] In Antimirov (1996), this property has been formulated for word regular expressions, but it straightforwardly extends to regular hedge expressions we use in this paper.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

where $v \in \mathscr{V}_{\mathsf{T}} \cup \mathscr{V}_{\mathsf{H}}$, $[\![\mathsf{R}]\!] = [\![\mathsf{R}_1]\!] \cap [\![\mathsf{R}_2]\!]$, and neither v in R_1 nor v in R_2 can be transformed by the other rules.

Next, we have rules which constrain singleton hedges to be in a term language. They proceed by the straightforward matching or decomposition of the structure. Note that in (M12), we require the arguments of the unordered function symbol to be terms. (M10) and (M9) do not distinguish whether f is ordered or unordered:

(M9) $\overline{x}$ in $f(\mathsf{R}) \wedge \mathscr{K} \rightsquigarrow \overline{x} \doteq x \wedge x$ in $f(\mathsf{R}) \wedge \mathscr{K}\{\overline{x} \mapsto x\}$, where x is fresh.

(M10) $X(H)$ in $f(\mathsf{R}) \wedge \mathscr{K} \rightsquigarrow X \doteq f \wedge f(H)\{X \mapsto f\}$ in $f(\mathsf{R}) \wedge \mathscr{K}\{X \mapsto f\}$.

(M11) $f_{\mathsf{o}}(H)$ in $f_{\mathsf{o}}(\mathsf{R}) \rightsquigarrow H$ in R.

(M12) $f_{\mathsf{u}}(T)$ in $f_{\mathsf{u}}(\mathsf{R}) \wedge \mathscr{K} \rightsquigarrow \bigvee_{T' \in perm(T)} \left(T' \text{ in } \mathsf{R} \wedge \mathscr{K}\right)$.

We denote the set of rules (M1)–(M12) by Memb.

5.2 *The constraint solving algorithm*

In this section, we present an algorithm that converts a constraint with respect to the rules specified in Section 5.1 into a partially solved one. First, we define the rewrite step

$$\mathsf{step} := \mathsf{first}(\mathsf{Log}, \mathsf{Fail}, \mathsf{Del}, \mathsf{Dec}, \mathsf{Elim}, \mathsf{Memb}).$$

When applied to a constraint, step transforms it by the *first* applicable rule of the solver, looking successively into the sets Log, Fail, Del, Dec, Elim, and Memb. If none of them apply, then the constraint is said to be in a *normal form* with respect to step.

The constraint solving algorithm implements the strategy solve defined as a repeated application of the rewrite step, aiming at the computation of a normal form with respect to step. But it also makes sure that the constraint, passed to step, is in DNF:

$$\mathsf{solve} := \mathsf{compose}(\mathsf{dnf}, \mathsf{NF}(\mathsf{step})).$$

Hence, solve takes a quantifier-free constraint, transforms it into its equivalent constraint in DNF (the strategy dnf in the definition stands for the algorithm that does it), and then repeatedly applies step to the obtained constraint in DNF as long as possible. It remains to show that this definition yields an algorithm, which amounts to proving that the strategy NF(step) indeed produces a constraint to which none of the rules from Log, Fail, Del, Dec, Elim, and Memb apply. The termination theorem states exactly this.

Theorem 2 (*Termination of* solve)
solve terminates on any quantifier-free constraint.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

With the next two statements we show that the solver reduces a constraint to its equivalent constraint:

Lemma 1
If $\mathsf{step}(\mathscr{C}) = \mathscr{D}$, then $\mathfrak{I} \models \forall\left(\mathscr{C} \leftrightarrow \overline{\exists}_{var(\mathscr{C})}\mathscr{D}\right)$ for all intended structures $\mathfrak{I}$.

Theorem 3
If $\mathsf{solve}(\mathscr{C}) = \mathscr{D}$, then $\mathfrak{I} \models \forall\left(\mathscr{C} \leftrightarrow \overline{\exists}_{var(\mathscr{C})}\mathscr{D}\right)$ for all intended structures $\mathfrak{I}$, and $\mathscr{D}$ is either partially solved or the false constraint.

6 Operational semantics of CLP(H)

In this section, we describe the operational semantics of CLP(H), following the approach for the CLP schema given in (Jaffar *et al.* 1998). A *state* is a pair $\langle G \parallel \mathscr{C}\rangle$, where G is the sequence of literals and $\mathscr{C} = \mathscr{K}_1 \vee \cdots \vee \mathscr{K}_n$, where $\mathscr{K}$'s are conjunctions of true, false, and primitive constraints. The *definition of an atom* $p(t_1,\ldots,t_m)$ *in program* P, $defn_{\mathrm{Pr}}(p(t_1,\ldots,t_m))$, is the set of rules in Pr such that the head of each rule has a form $p(r_1,\ldots,r_m)$. We assume that $defn_{\mathrm{Pr}}$ each time returns fresh variants.

A state $\langle L_1,\ldots,L_n \parallel \mathscr{C}\rangle$ can be *reduced with respect to* P as follows: Select a literal L_i. Then

- If L_i is a primitive constraint and $\mathsf{solve}(\mathscr{C} \wedge L_i) \neq \mathsf{false}$, then it is reduced to $\langle L_1,\ldots,L_{i-1},L_{i+1},\ldots,L_n \parallel \mathsf{solve}(\mathscr{C} \wedge L_i)\rangle$.
- If L_i is a primitive constraint and $\mathsf{solve}(\mathscr{C} \wedge L_i) = \mathsf{false}$, then it is reduced to $\langle \Box \parallel \mathsf{false}\rangle$.
- If L_i is an atom $p(t_1,\ldots,t_m)$, then it is reduced to

$$\langle L_1,\ldots,L_{i-1},t_1 \doteq r_1,\ldots,t_m \doteq r_m,B,L_{i+1},\ldots,L_n \parallel \mathscr{C}\rangle$$

  for some $(p(r_1,\ldots,r_m) \leftarrow B) \in defn_{\mathrm{Pr}}(L_i)$.
- If L_i is an atom and $defn_{\mathrm{Pr}}(L_i) = \emptyset$, then it is reduced to $\langle \Box \parallel \mathsf{false}\rangle$.

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

Naturally, it is interesting to find syntactic restrictions for programs guaranteeing that non-failed finished derivations produce a solved constraint instead of a partially solved one. In the next two sections, we consider such restrictions, leading to well-moded and KIF style CLP(H) programs that have the desired property.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

7 Well-moded programs

The concept of well-modedness is due to Dembinski and Maluszynski (1985). A *mode* for an n-ary predicate symbol p is a function $m_p : \{1,\ldots,n\} \longrightarrow \{\mathsf{i},\mathsf{o}\}$. If $m_p(i) = \mathsf{i}$ (resp. $m_p(i) = \mathsf{o}$) then the position i is called an *input* (resp. *output*) *position* of p. The predicates in and $\doteq$ have only output positions. For a literal $L = p(t_1,\ldots,t_n)$ (where p can be also in or $\doteq$), we denote by $invar(L)$ and $outvar(L)$ the sets of variables occurring in terms in the input and output positions of p.

If a predicate is used with different modes $m_p^1,\ldots,m_p^k$ in the program, we may consider each $p_{m_p^i}$ as a separate predicate. Therefore, we can assume without loss of generality that every predicate has exactly one mode (cf., e.g., Ganzinger and Waldmann (1992)).

An *extended literal* E is either a literal, true, or false. We define $invar(\mathsf{true}) := \emptyset$, $outvar(\mathsf{true}) := \emptyset$, $invar(\mathsf{false}) := \emptyset$, and $outvar(\mathsf{false}) := \emptyset$.

A *sequence of extended literals* $E_1,\ldots,E_n$ *is well-moded* if the following hold:

(1) For all $1 \leqslant i \leqslant n$, $invar(E_i) \subseteq \bigcup_{j=1}^{i-1} outvar(E_j)$.
(2) If for some $1 \leqslant i \leqslant n$, E_i is $t_1 \doteq t_2$, then $var(t_1) \subseteq \bigcup_{j=1}^{i-1} outvar(E_j)$ or $var(t_2) \subseteq \bigcup_{j=1}^{i-1} outvar(E_j)$.
(3) If for some $1 \leqslant i \leqslant n$, E_i is a membership atom, then the inclusion $var(E_i) \subseteq \bigcup_{j=1}^{i-1} outvar(E_j)$ holds.

A *conjunction of extended literals* G *is well-moded* if there exists a well-moded sequence of extended literals $E_1,\ldots,E_n$ such that $G = \bigwedge_{i=1}^{n} E_i$ modulo associativity and commutativity of conjunction. A *formula in DNF is well-moded* if each of its disjuncts is. A *state* $\langle L_1,\ldots,L_n \parallel \mathscr{K}_1 \vee \cdots \vee \mathscr{K}_m\rangle$ *is well-moded*, where $\mathscr{K}$'s are conjunctions of true, false, and primitive constraints, if the formula $(L_1 \wedge \cdots \wedge L_n \wedge \mathscr{K}_1) \vee \cdots \vee (L_1 \wedge \cdots \wedge L_n \wedge \mathscr{K}_m)$ is well-moded.

A *clause* $A \leftarrow L_1,\ldots,L_n$ *is well-moded* if the following hold:

(1) For all $1 \leqslant i \leqslant n$, $invar(L_i) \subseteq \bigcup_{j=1}^{i-1} outvar(L_j) \cup invar(A)$.
(2) $outvar(A) \subseteq \bigcup_{j=1}^{n} outvar(L_j) \cup invar(A)$.
(3) If for some $1 \leqslant i \leqslant n$, L_i is $t_1 \doteq t_2$, then $var(t_1) \subseteq \bigcup_{j=1}^{i-1} outvar(L_j) \cup invar(A)$ or $var(t_2) \subseteq \bigcup_{j=1}^{i-1} outvar(L_j) \cup invar(A)$.
(4) If for some $1 \leqslant i \leqslant n$, L_i is a membership atom, then $outvar(L_i) \subseteq \bigcup_{j=1}^{i-1} outvar(L_j) \cup invar(A)$.

A *program is well-moded* if all its clauses are well-moded.

Example 3
In Example 1, if in the user-defined binary predicates *rewrite* and *rule* the first argument is the input position and the second argument is the output position, then it is easy to see that the program is well-moded. In Example 2, for well-modedness we need to define both positions in the user-defined predicates to be the input ones.

In the rest of this section, we investigate the behavior of well-moded programs. Before going into the details, we briefly summarize two main results as follows:

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

- The solver can completely solve satisfiable well-moded constraints (instead of partial solutions computed in the general case). See Theorem 4.
- Any finished derivation from a well-moded goal with respect to a well-moded program either ends with a completely solved constraint, or fails. See Theorem 5.

These statements depend on some technical results we give below.

Lemma 2
Let $v \doteq e$ be an equation, where v is a variable and e is the corresponding expression such that v does not occur in e. Let $\mathscr{K}_1$ and $\mathscr{K}_2$ be two arbitrary (possibly empty) conjunctions of extended literals such that the conjunction $\mathscr{K}_1 \wedge \mathscr{K}_2 \wedge v \doteq e$ is well-moded. Let $\theta = \{v \mapsto e\}$ be a substitution. Then $\mathscr{K}_1 \wedge \mathscr{K}_2\theta \wedge v \doteq e$ is also well-moded.

The next lemma states an important result: reduction with respect to a well-moded program preserves well-modedness of states. (Its proof uses Lemma 2. See the online appendix of the paper for the details.)

Lemma 3
Let Pr be a well-moded CLP(H) program and $\langle \mathsf{G} \parallel \mathscr{C} \rangle$ be a well-moded state. If $\langle \mathsf{G} \parallel \mathscr{C} \rangle \rightarrowtail \langle \mathsf{G}' \parallel \mathscr{C}' \rangle$ is a reduction using clauses in Pr, then $\langle \mathsf{G}' \parallel \mathscr{C}' \rangle$ is also a well-moded state.

Corollary 1
If $\mathscr{C}$ is a well-moded constraint, then $\mathsf{solve}(\mathscr{C})$ is also well-moded.

From Corollary 1, Theorem 3, and the definition of well-modedness we can show that satisfiable well-moded constraints can be completely solved.

Theorem 4
Let $\mathscr{C}$ be a well-moded constraint and $\mathsf{solve}(\mathscr{C}) = \mathscr{C}'$, where $\mathscr{C}' \neq \mathsf{false}$. Then $\mathscr{C}'$ is solved.

We illustrate how to solve a simple well-moded constraint:

Example 4
Let $\mathscr{C} = f(\overline{x}, a, \overline{y}) \doteq f(a, b, a, c, c) \wedge f(\overline{z}, a, x) \doteq f(\overline{y}, \overline{x}) \wedge \overline{y}$ in $c(\mathsf{eps})^*$. Then solve performs the following derivation (some steps are contracted):

$$
\begin{aligned}
\mathscr{C} \rightsquigarrow \quad & \big(\overline{x} \doteq \epsilon \wedge (a, \overline{y}) \doteq (a, b, a, c, c) \wedge f(\overline{z}, a, x) \doteq f(\overline{y}, \overline{x}) \wedge \overline{y} \text{ in } c(\mathsf{eps})^*\big) \\
& \vee \big(\overline{x} \doteq a \wedge (a, \overline{y}) \doteq (b, a, c, c) \wedge f(\overline{z}, a, x) \doteq f(\overline{y}, \overline{x}) \wedge \overline{y} \text{ in } c(\mathsf{eps})^*\big) \\
& \vee \big(\overline{x} \doteq (a, b) \wedge (a, \overline{y}) \doteq (a, c, c) \wedge f(\overline{z}, a, x) \doteq f(\overline{y}, \overline{x}) \wedge \overline{y} \text{ in } c(\mathsf{eps})^*\big) \\
& \quad \cdots \\
& \vee \big(\overline{x} \doteq (a, b, a, c, c) \wedge (a, \overline{y}) \doteq \epsilon \wedge f(\overline{z}, a, x) \doteq f(\overline{y}, \overline{x}) \wedge \overline{y} \text{ in } c(\mathsf{eps})^*\big) \\
\rightsquigarrow^+ \quad & \big(\overline{x} \doteq \epsilon \wedge \overline{y} \doteq (b, a, c, c) \wedge f(\overline{z}, a, x) \doteq f(\overline{y}, \overline{x}) \wedge \overline{y} \text{ in } c(\mathsf{eps})^*\big) \\
& \vee \big(\overline{x} \doteq (a, b) \wedge \overline{y} \doteq (c, c) \wedge f(\overline{z}, a, x) \doteq f(\overline{y}, \overline{x}) \wedge \overline{y} \text{ in } c(\mathsf{eps})^*\big) \\
\rightsquigarrow \quad & \big(\overline{x} \doteq \epsilon \wedge \overline{y} \doteq (b, a, c, c) \wedge f(\overline{z}, a, x) \doteq f(b, a, c, c, \overline{x}) \wedge (b, a, c, c) \text{ in } c(\mathsf{eps})^*\big)
\end{aligned}
$$

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

$$
\begin{aligned}
& \vee\big(\overline{x} \doteq (a,b) \wedge \overline{y} \doteq (c,c) \wedge f(\overline{z},a,x) \doteq f(\overline{y},\overline{x}) \wedge \overline{y} \text{ in } c(\mathsf{eps})^*\big) \\
\rightsquigarrow \quad & \big(\overline{x} \doteq \epsilon \wedge \overline{y} \doteq (b,a,c,c) \wedge f(\overline{z},a,x) \doteq f(b,a,c,c,\overline{x}) \\
& \qquad \wedge b \text{ in } c(\mathsf{eps}) \wedge (a,c,c) \text{ in } c(\mathsf{eps})^*\big) \\
& \vee\big(\overline{x} \doteq (a,b) \wedge \overline{y} \doteq (c,c) \wedge f(\overline{z},a,x) \doteq f(\overline{y},\overline{x}) \wedge \overline{y} \text{ in } c(\mathsf{eps})^*\big) \\
\rightsquigarrow \quad & \big(\overline{x} \doteq (a,b) \wedge \overline{y} \doteq (c,c) \wedge f(\overline{z},a,x) \doteq f(\overline{y},\overline{x}) \wedge \overline{y} \text{ in } c(\mathsf{eps})^*\big) \\
\rightsquigarrow^+ \quad & \big(\overline{x} \doteq (a,b) \wedge \overline{y} \doteq (c,c) \wedge f(\overline{z},a,x) \doteq f(c,c,a,b) \wedge (c,c) \text{ in } c(\mathsf{eps})^*\big) \\
\rightsquigarrow^+ \quad & \big(\overline{x} \doteq (a,b) \wedge \overline{y} \doteq (c,c) \wedge f(\overline{z},a,x) \doteq f(c,c,a,b)\big) \\
\rightsquigarrow^+ \quad & \big(\overline{x} \doteq (a,b) \wedge \overline{y} \doteq (c,c) \wedge \overline{z} \doteq (c,c) \wedge x \doteq b\big).
\end{aligned}
$$

The obtained constraint is solved.

The next theorem is the main result for well-moded CLP(H) programs. It states that any finished derivation from a well-moded goal leads to a solved constraint or to a failure.

Theorem 5
Let $\langle \mathsf{G} \parallel \mathsf{true}\rangle \rightarrowtail \cdots \rightarrowtail \langle \square \parallel \mathscr{C}\rangle$ be a finished derivation with respect to a well-moded CLP(H) program, starting from a well-moded goal G. If $\mathscr{C} \neq \mathsf{false}$, then $\mathscr{C}$ is solved.

8 Programs in the KIF form

KIF (Genesereth and Fikes 1992), is a computer-oriented language for the interchange of knowledge among disparate programs. It permits variadic syntax and hedge variables, under the restriction that such variables are only the last arguments of subterms they appear in. Such a fragment has some good computation properties, e.g., unification is unitary (Kutsia 2003). The special form of programs and constraints considered in this section originates from this restriction.

Terms and hedges in the KIF form or, shortly, *KIF terms* and *KIF hedges*, are defined by the following grammar:

$$
\begin{aligned}
t_\kappa &::= x \mid f_{\mathsf{o}}(H_\kappa) \mid f_{\mathsf{u}}(t_{\kappa 1},\ldots,t_{\kappa n}) \mid X(t_{\kappa 1},\ldots,t_{\kappa n}) \quad (n \geqslant 0) && \text{KIF Term} \\
H_\kappa &::= t_{\kappa 1},\ldots,t_{\kappa n} \mid t_{\kappa 1},\ldots,t_{\kappa n},\overline{x} \quad (n \geqslant 0) && \text{KIF Hedge}
\end{aligned}
$$

That means that a term is in the KIF form if hedge variables occur only below ordered function symbols as the last arguments. For example, the terms $f_{\mathsf{o}}(x, f_{\mathsf{o}}(a,\overline{x}), f_{\mathsf{u}}(x,b),\overline{x})$ and $f_{\mathsf{o}}(a,x,b)$ are in the KIF form, while $f_{\mathsf{o}}(\overline{x},a,\overline{x})$ and $f_{\mathsf{u}}(x, f_{\mathsf{o}}(a,\overline{x}), f_{\mathsf{u}}(x,b),\overline{x})$ are not.

If the language does not contain unordered function symbols, then we permit hedge variables under function variables, again in the last position, i.e., of the form $X(H_\kappa)$.

In this section we consider only KIF terms. Therefore, the subscript κ will be omitted.

KIF equations and *KIF atoms* are constructed from KIF terms. In a *KIF membership atom* H in R, the hedge H is a KIF hedge.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

KIF formulas are constructed from KIF primitive constraints and KIF atoms. This special form guarantees that the solver does not need to use all the rules. Simply inspecting them, we can see that Del1, E3, E4, and M3 are not used. In Del3, it is guaranteed that H_2 will be always empty, and in M1 the n will be equal to 1.

Similarly to the well-moded restriction above, our interest to the KIF fragment is justified by its two important properties that characterize the KIF constraint solving and derivation of KIF goals:

- The solver can completely solve satisfiable KIF constraints (instead of partial solutions computed in the general case). See Theorem 6.
- Any finished derivation from a KIF goal with respect to a KIF program either ends with a completely solved constraint, or fails. See Theorem 7.

Their proofs are easier than the ones of the corresponding statements for well-moded programs. (To compare, see the online appendix.) This is largely due to the following lemma:

Lemma 4
Any partially solved KIF constraint is solved.

One can see that no solving rule inserts a term or a hedge variable after the last argument of subterms in constraints. That means, KIF constraints are again transformed into KIF constraints. Hence, the constraint computed by solve will be a KIF constraint. It leads us to the following result:

Theorem 6
Let $\mathscr{C}$ be a KIF constraint and $\mathsf{solve}(\mathscr{C}) = \mathscr{C}'$, where $\mathscr{C}' \neq \mathsf{false}$. Then $\mathscr{C}'$ is solved.

We illustrate now how to solve a simple KIF constraint:

Example 5
Let $\mathscr{C} = f(x, \overline{x}) \doteq f(g(\overline{y}), a, \overline{y}) \wedge \overline{x}\ \mathsf{in}\ a(\mathsf{eps})^* \wedge \overline{y}\ \mathsf{in}\ a(\mathsf{eps}) \cdot a(b(\mathsf{eps})^*)^*$. Then solve performs the following derivation:

$$\begin{aligned}
\mathscr{C} &\rightsquigarrow x \doteq g(\overline{y}) \wedge \overline{x} \doteq (a, \overline{y}) \wedge \overline{x}\ \mathsf{in}\ a(\mathsf{eps})^* \wedge \overline{y}\ \mathsf{in}\ a(\mathsf{eps}) \cdot a(b(\mathsf{eps})^*)^* \\
&\rightsquigarrow x \doteq g(\overline{y}) \wedge \overline{x} \doteq (a, \overline{y}) \wedge (a, \overline{y})\ \mathsf{in}\ a(\mathsf{eps})^* \wedge \overline{y}\ \mathsf{in}\ a(\mathsf{eps}) \cdot a(b(\mathsf{eps})^*)^* \\
&\rightsquigarrow x \doteq g(\overline{y}) \wedge \overline{x} \doteq (a, \overline{y}) \wedge \overline{y}\ \mathsf{in}\ a(\mathsf{eps})^* \wedge \overline{y}\ \mathsf{in}\ a(\mathsf{eps}) \cdot a(b(\mathsf{eps})^*)^* \\
&\rightsquigarrow x \doteq g(\overline{y}) \wedge \overline{x} \doteq (a, \overline{y}) \wedge \overline{y}\ \mathsf{in}\ a(\mathsf{eps}) \cdot a(\mathsf{eps})^*
\end{aligned}$$

The obtained constraint is solved.

A *state* $\langle L_1, \ldots, L_n \parallel \mathscr{K}_1 \vee \cdots \vee \mathscr{K}_m \rangle$ *is in the KIF form* (*KIF state*), if the formula $(L_1 \wedge \cdots \wedge L_n \wedge \mathscr{K}_1) \vee \cdots \vee (L_1 \wedge \cdots \wedge L_n \wedge \mathscr{K}_m)$ is a KIF formula.

KIF clauses are constructed from KIF atoms and literals. *KIF programs* are sets of KIF clauses. It is not hard to check that each reduction step (with respect to a KIF program) in the operational semantics preserves KIF states: It follows from the definition of the operational semantics and the fact that solve computes KIF constraints. Therefore, we can establish the following theorem:

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

Theorem 7
Let $\langle \mathsf{G} \parallel \mathsf{true} \rangle \rightarrowtail \cdots \rightarrowtail \langle \Box \parallel \mathscr{C}' \rangle$ be a finished derivation with respect to a KIF program, starting from a KIF goal G. If $\mathscr{C}' \neq \mathsf{false}$, then $\mathscr{C}'$ is solved.

Example 6
The well-known technique of appending two difference lists can be used in CLP(H) for a more general task: to combine arguments of arbitrary two terms. The program remains the same as in the standard logic programming

$$append\_dl(x_1\text{-}x_2,\ x_2\text{-}x_3,\ x_1\text{-}x_3),$$

where the hyphen is a function symbol and x_1, x_2, x_3 are term variables. The KIF goal

$$append\_dl(f_1(a,b,\overline{x})\text{-}f_2(\overline{x}),\ f_2(c,d,e,\overline{y})\text{-}f_3(\overline{y}),\ x\text{-}f_3)$$

can be used to append to the arguments of $f_1(a,b)$ the arguments of $f_2(c,d,e)$, obtaining $f_1(a,b,c,d,e)$. Note that the terms may have different heads. The derivation proceeds as follows:

$$\begin{aligned}
& \langle append_dl(f_1(a,b,\overline{x})\text{-}f_2(\overline{x}),\ f_2(c,d,e,\overline{y})\text{-}f_3(\overline{y}),\ x\text{-}f_3) \parallel \mathsf{true} \rangle \\
\rightarrowtail\ & \langle x_1\text{-}x_2 \doteq f_1(a,b,\overline{x})\text{-}f_2(\overline{x}),\ x_2\text{-}x_3 \doteq f_2(c,d,e,\overline{y})\text{-}f_3(\overline{y}),\ x_1\text{-}x_3 \doteq x\text{-}f_3 \parallel \mathsf{true} \rangle \\
\rightarrowtail\ & \langle x_2\text{-}x_3 \doteq f_2(c,d,e,\overline{y})\text{-}f_3(\overline{y}),\ x_1\text{-}x_3 \doteq x\text{-}f_3 \parallel x_1 \doteq f_1(a,b,\overline{x}) \wedge x_2 \doteq f_2(\overline{x}) \rangle \\
\rightarrowtail\ & \langle x_1\text{-}x_3 \doteq x\text{-}f_3 \parallel \\
& \qquad x_1 \doteq f_1(a,b,c,d,e,\overline{y}) \wedge x_2 \doteq f_2(c,d,e,\overline{y}) \wedge x_3 \doteq f_3(\overline{y}) \wedge \overline{x} \doteq (c,d,e,\overline{y}) \rangle \\
\rightarrowtail\ & \langle \Box \parallel \\
& \qquad x_1 \doteq f_1(a,b,c,d,e) \wedge x_2 \doteq f_2(c,d,e) \wedge x_3 \doteq f_3 \wedge \overline{x} \doteq (c,d,e) \wedge \overline{y} \doteq \epsilon \wedge \\
& \qquad x \doteq f_1(a,b,c,d,e) \rangle.
\end{aligned}$$

The constraint in the final state is solved.

9 Conclusion

Solving equational and membership constraints over hedges is not an easy task: the problem is infinitary and any procedure that explicitly computes all solutions is non-terminating. The solver that we presented in this paper is not complete, but it is terminating. It solves constraints partially and tries to detect failure as early as it can.

Incorporating the solver into the CLP schema gives CLP(H): constraint logic programming for hedges. We defined algebraic semantics for it and used it to characterized the constraint solver: the output of the solver (which is either partially solved of false) is equivalent to the input constraint in all intended structures.

The fact that the solver, in general, returns a partially solved result (when it does not fail), naturally raises the question: Are there some interesting fragments of constraints that the solver can completely solve? We give a positive answer to this question, defining well-moded and KIF constraints and showing their complete solvability.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

It immediately poses the next question: Can one characterize CLP(H) programs that generate only well-moded or KIF constraints only? We show that by extending the notions of well-modedness and KIF form to programs, we get the desired fragments. Any finished derivation of a goal for such fragments gives a definite answer: Either the goal fails, or a solved constraint is returned.

The constraints we consider in this paper are positive, but at least the well-moded programs can be easily enriched with the negation. Well-modedness guarantees that the eventual test for disequality or non-membership in constraints will be performed on ground hedges, which can be effectively decided.

Acknowledgements

This is an extended version of a paper presented at the 12th International Symposium on Functional and Logic Programming (FLOPS 2014), invited as a rapid publication in TPLP. The authors acknowledge the assistance of the conference chairs Michael Codish and Eijiro Sumii.

References

Antimirov, V. M. 1996. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science 155,* 2, 291–319.

Balland, E., Brauner, P., Kopetz, R., Moreau, P. and Reilles, A. 2007. Tom: Piggybacking rewriting on java. In *Proc. Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26–28, 2007*, F. Baader, Ed. Lecture Notes in Computer Science, vol. 4533. Springer, 36–47.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. L. Eds. 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science, vol. 4350. Springer.

Coelho, J. and Florido, M. 2004. CLP(Flex): Constraint logic programming applied to XML processing. In *Proc. On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences, Agia Napa, Cyprus, October 25–29 Part II*, R. Meersman and Z. Tari, Eds. Lecture Notes in Computer Science, vol. 3291. Springer, 1098–1112.

Coelho, J. and Florido, M. 2006. VeriFLog: A constraint logic programming approach to verification of website content. In *Proc. Advanced Web and Network Technologies, and Applications, APWeb 2006 International Workshops: XRA, IWSN, MEGA, and ICSE, Harbin, China, January 16–18, 2006*, H. T. Shen, J. Li, M. Li, J. Ni and W. Wang, Eds. Lecture Notes in Computer Science, vol. 3842. Springer, 148–156.

Coelho, J. and Florido, M. 2007. XCentric: Logic programming for XML processing. In *Proc. of 9th ACM International Workshop on Web Information and Data Management (WIDM 2007), Lisbon, Portugal, November 9*, I. Fundulaki and N. Polyzotis, Eds. ACM, 1–8.

Colmerauer, A. 1990. An introduction to Prolog III. *Communications of the ACM 33,* 7, 69–90.

Comon, H. 1998. Completion of rewrite systems with membership constraints. Part II: constraint solving. *Journal of Symbolic Computation 25,* 4, 421–453.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

Dembinski, P. and Maluszynski, J. 1985. And-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the 1985 Symposium on Logic Programming, Boston, Massachusetts, USA, July 15–18*, IEEE-CS, 29–38.

Dershowitz, N. 1982. Orderings for term-rewriting systems. *Theoretical Computer Science 17*, 279–301.

Dovier, A., Piazza, C., Pontelli, E. and Rossi, G. 2000. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems 22,* 5, 861–931.

Dovier, A., Piazza, C. and Rossi, G. 2008. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. *ACM Transactions on Computational Logic 9,* 3, 1–30.

Dundua, B., Florido, M., Kutsia, T. and Marin, M. 2014. Constraint logic programming for hedges: A semantic reconstruction. In *Proc. of Functional and Logic Programming – 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4–6*, M. Codish and E. Sumii, Eds. Lecture Notes in Computer Science, vol. 8475. Springer, 285–301.

Ganzinger, H. and Waldmann, U. 1992. Termination proofs of well-moded logic programs via conditional rewrite systems. In *Proc. of Conditional Term Rewriting Systems, 3rd International Workshop, CTRS-92, Pont-à-Mousson, France, July 8–10*, M. Rusinowitch and J. Remy, Eds. Lecture Notes in Computer Science, vol. 656. Springer, 430–437.

Genesereth, M. R. and Fikes, R. E. 1992. *Knowledge Interchange Format, Version 3.0 Reference Manual*. Tech. Rep. Logic-92-1, Stanford University, Stanford, CA, USA.

Hosoya, H. and Pierce, B. C. 2003. Regular expression pattern matching for XML. *Journal of Functional Programming 13,* 6, 961–1004.

Jaffar, J., Maher, M. J., Marriott, K. and Stuckey, P. J. 1998. The semantics of constraint logic programs. *Journal of Logic Programming 37,* 1–3, 1–46.

Kutsia, T. 2003. Equational prover of Theorema. In *Proc. of Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9–11*, R. Nieuwenhuis, Ed. Lecture Notes in Computer Science, vol. 2706. Springer, 367–379.

Kutsia, T. 2004. Solving equations involving sequence variables and sequence functions. In *Proc of Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004, Linz, Austria, September 22–24*, B. Buchberger and J. A. Campbell, Eds. Lecture Notes in Computer Science, vol. 3249. Springer, 157–170.

Kutsia, T. 2007. Solving equations with sequence variables and sequence functions. *Journal of Symbolic Computation 42,* 3, 352–388.

Kutsia, T. and Marin, M. 2005a. Can context sequence matching be used for querying XML? In *Proc. of the 19th International Workshop on Unification UNIF'05*, L. Vigneron, Ed. Nara, Japan, 77–92.

Kutsia, T. and Marin, M. 2005b. Matching with regular constraints. In *Proc. of Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2–6*, G. Sutcliffe and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 3835. Springer, 215–229.

Marin, M. and Kutsia, T. 2003. On the implementation of a rule-based programming system and some of its applications. In *Proc. of the 4th International Workshop on the Implementation of Logics (WIL'03)*, B. Konev and R. Schmidt, Eds. Almaty, Kazakhstan, 55–68.

Marin, M. and Kutsia, T. 2006. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics 16,* 1-2, 151–168.

Miller, D. 1991. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation 1,* 4, 497–536.

Rajasekar, A. 1994. Constraint logic programming on strings: Theory and applications. In *Logic Programming, Proc. of the 1994 International Symposium, Ithaca, New York, USA, November 13–17*, M. Bruynooghe, Ed. MIT Press, 681.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press

van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P. A., Scheerder, J., Vinju, J. J., Visser, E. and Visser, J. 2001. The ASF+SDF meta-environment: A component-based language development environment. *Electronic Notes in Theoretical Computer Science 44,* 2, 3–8.

Walinsky, C. 1989. CLP(Σ^*): Constraint logic programming with regular sets. In *Logic Programming, Proc. of the 6th International Conference, Lisbon, Portugal, June 19–23*, G. Levi and M. Martelli, Eds. MIT Press, 181–196.

Wolfram, S. 2003. *The Mathematica Book*, 5th ed. Wolfram-Media.

https://doi.org/10.1017/S1471068415000071 Published online by Cambridge University Press