Constraint Logic Programming for Hedges: A Semantic Reconstruction

Besik Dundua[1,3], Mário Florido[1], Temur Kutsia[2], and Mircea Marin[4]

[1] DCC-FC & LIACC, University of Porto, Portugal
[2] RISC, Johannes Kepler University, Linz, Austria
[3] VIAM, Ivane Javakhishvili Tbilisi State University, Georgia
[4] West University of Timişoara, Romania

Abstract. We describe the semantics of CLP(H): constraint logic programming over 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 give algebraic semantics of CLP(H) programs, define a sound, terminating, and incomplete constraint solver, and describe some fragments of constraints for which the solver returns a complete set of solutions.

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.

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 [21] is based on hedge pattern matching. Languages such as Tom [1], Maude [2], ASF+SDF [19] provide capabilities similar to hedge matching (via associative functions). ρLog [17] extends logic programming with hedge transformation rules. XDuce [13] enriches untyped hedge matching with regular expression types. The Constraint Logic Programming schema has been extended to work with hedges in CLP(Flex) [3], which is a basis for the XML processing language XCentric [5] and a Web site verification language VeriFLog [4].

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 symbol 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 Sect. 2. 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($\mathcal{S}$) [18], string processing features of Prolog III [6], and CLP over regular sets of strings CLP(Σ^*) [20].

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 [15]. 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 ground expressions at some point. This effectively reduces constraint solving to hedge matching (which is known to be NP-complete [16]), plus some early failure detection rules. Another fragment for which the solver is complete is named after the Knowledge Interchange Format, KIF [12], 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.

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., [10] 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.

The paper is organized as follows: We start with motivating examples in Sect. 2. In Sect. 3 we describe the syntax of CLP(H). Sect. 4 is about semantics. The constraint solver is introduced in Sect. 5. The operational semantics of CLP(H) is described in Sect. 6. In Sect. 7, we introduce well-moded and KIF fragments of CLP(H) programs, for which the constraint solver is complete. Due to space restrictions, proofs of technical lemmas are put in the report [11].

2 Motivating Examples

In this section we show how to write programs in CLP(H). For illustration, we chose 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 $rewrite(x, y) \leftarrow rule(x, y)$ 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 symbol 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[5] $\overline{x}_1$ 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 answers: $\{x \mapsto f(f(f(a, a), f))\}$ and $\{x \mapsto f(f(f(a, a), f(b)))\}$.

Example 2. The recursive path ordering (rpo) $>_{\mathrm{rpo}}$ is a well-known term ordering [8] 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, $1 \le i \le m$, or
2. $t = g(t_1, \ldots, t_n)$, $s >_{\mathrm{rpo}} t_i$ for all $i, 1 \le i \le n$, 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 $s >_{\mathrm{rpo}} t_i$ 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

[5] In the notation defined later, strictly speaking, we need to write this constraint as $f(a(\mathtt{eps})^*) \cdot b(\mathtt{eps})^*$, where `eps` is the regular expression for the empty hedge. However, for brevity and clarity of the presentation we omit `eps` here.

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:

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

rpo_all is implemented with recursion:

$$rpo\_all(x, \langle\,\rangle). \qquad rpo\_all(x, \langle y, \overline{y}\rangle) \leftarrow rpo(x, y), rpo\_all(x, \langle\overline{y}\rangle).$$

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

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

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

$$lex(\langle\overline{x}, x, \overline{y}\rangle, \langle\overline{x}, y, \overline{z}\rangle) \leftarrow rpo(x, y).$$
$$mul(\{x, \overline{x}\}, \{\}). \qquad mul(\{x, \overline{x}\}, \{x, \overline{y}\}) \leftarrow mul(\{\overline{x}\}, \{\overline{y}\}).$$
$$mul(\{x, \overline{x}\}, \{y, \overline{y}\}) \leftarrow rpo(x, y),\ mul(\{x, \overline{x}\}, \{\overline{y}\}).$$

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

3 Preliminaries

For common notation and definitions, we mostly follow [14]. The alphabet $\mathcal{A}$ consists of the following pairwise disjoint sets of symbols:

- $\mathcal{V}_{\mathsf{T}}$: term variables, denoted by $x, y, z, \ldots$,
- $\mathcal{V}_{\mathsf{H}}$: hedge variables, denoted by $\overline{x}, \overline{y}, \overline{z}, \ldots$,
- $\mathcal{V}_{\mathsf{F}}$: function variables, denoted by $X, Y, Z, \ldots$,
- $\mathcal{F}_{\mathsf{u}}$: unranked unordered function symbols, denoted by $f_{\mathsf{u}}, g_{\mathsf{u}}, h_{\mathsf{u}}, \ldots$,
- $\mathcal{F}_{\mathsf{o}}$: unranked ordered function symbols, denoted by $f_{\mathsf{o}}, g_{\mathsf{o}}, h_{\mathsf{o}}, \ldots$,
- $\mathcal{P}$: 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, $\mathcal{A}$ also contains

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

– Auxiliary symbols: parentheses and the comma.

Function symbols, denoted by $f, g, h, \ldots$, are elements of the set $\mathcal{F} = \mathcal{F}_{\mathsf{u}} \cup \mathcal{F}_{\mathsf{o}}$. A *variable* is an element of the set $\mathcal{V} = \mathcal{V}_{\mathsf{T}} \cup \mathcal{V}_{\mathsf{H}} \cup \mathcal{V}_{\mathsf{F}}$. 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 $\mathcal{A}$ as follows:

$t ::= x \mid f(H) \mid X(H)$	Term
$T ::= t_1, \ldots, t_n \quad (n \geq 0)$	Term sequence
$h ::= t \mid \overline{x}$	Hedge element
$H ::= h_1, \ldots, h_n \quad (n \geq 0)$	Hedge

We denote the set of terms by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and the set of ground (variable-free) terms by $\mathcal{T}(\mathcal{F})$. For readability, we put parentheses around hedges, writing, e.g., $(f(a), \overline{x}, b)$ instead of $f(a), \overline{x}, b$. The empty hedge is written as ϵ. Besides the letter t, we use also r and s to denote terms. 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.

An *atom* is a formula of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$ 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. Instead of $F_1() \doteq F_2()$ and $f_{\mathsf{o}}(H_1) \doteq f_{\mathsf{o}}(H_2)$ we write $F_1 \doteq F_2$ and $H_1 \doteq H_2$ respectively. We denote the symmetric closure of the relation $\doteq$ by $\simeq$.

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 $V = \{v_1, \ldots, v_n\} \subset \mathcal{V}$. $\overline{\exists}_V N$ 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 $\mathcal{A}$ as $\mathcal{L}(\mathcal{A})$.

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 term, hedge, and function variables are mapped to themselves. Substitutions extend to terms, hedges, literals, conjunction of literals.

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

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

– the variable is x, $\mathbf{c}_i = x \doteq t$, and x occurs neither in t nor elsewhere in $\mathcal{K}$, or

– the variable is $\overline{x}$, $\mathbf{c}_i = \overline{x} \doteq H$, and $\overline{x}$ occurs neither in H nor elsewhere in $\mathcal{K}$, or
– the variable is F, $\mathbf{c}_i = X \doteq F$ and X occurs neither in F nor elsewhere in $\mathcal{K}$, or
– the variable is x, $\mathbf{c}_i = x \ \mathsf{in}\ f(\mathsf{R})$ and x does not occur in membership constraints elsewhere in $\mathcal{K}$, 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 $\mathcal{K}$, 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* $\mathcal{K}$. Moreover, $\mathcal{K}$ is called *solved* if for any $1 \leq i \leq n$, $\mathbf{c}_i$ is solved in it. $\mathcal{K}$ is *partially solved*, if for any $1 \leq i \leq n$, $\mathbf{c}_i$ is solved in $\mathcal{K}$, 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 R has a 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.

4 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 $\{(s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(n)}) \mid \pi \text{ is a permutation of } \{1, 2, \ldots, n\}\}$.

A *structure* $\mathfrak{S}$ for a language $\mathcal{L}(\mathcal{A})$ 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 $f \in \mathcal{F}$ to a function $I(f) : D^* \to D$, and each n-ary predicate symbol $p \in \mathcal{P}$ to an n-ary relation $I(p) \subseteq D^n$. Moreover, if $f \in \mathcal{F}_{\mathsf{u}}$ 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 domain $\mathcal{V}$ 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 ($v \in \mathcal{V}_{\mathsf{T}} \cup \mathcal{V}_{\mathsf{H}}$):

$$[\![(H_1, \ldots, H_n)]\!]_{\mathfrak{S},\sigma} := ([\![H_1]\!]_{\mathfrak{S},\sigma}, \ldots, [\![H_n]\!]_{\mathfrak{S},\sigma}), \qquad [\![v]\!]_{\mathfrak{S},\sigma} := \sigma(v),$$

$$[\![f(H)]\!]_{\mathfrak{S},\sigma} := I(f)([\![H]\!]_{\mathfrak{S},\sigma}), \qquad [\![X(H)]\!]_{\mathfrak{S},\sigma} := \sigma(X)([\![H]\!]_{\mathfrak{S},\sigma}).$$

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:

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

Primitive constraints are interpreted w.r.t. a structure $\mathfrak{S}$ and variable assignment σ as follows: $\mathfrak{S} \models_\sigma t_1 \doteq t_2$ iff $[\![t_1]\!]_{\mathfrak{S},\sigma} = [\![t_2]\!]_{\mathfrak{S},\sigma}$; $\mathfrak{S} \models_\sigma H \ \mathsf{in}\ \mathsf{R}$ iff $[\![H]\!]_{\mathfrak{S},\sigma} \in [\![\mathsf{R}]\!]_{\mathfrak{S}}$; and $\mathfrak{S} \models_\sigma p(t_1, \ldots, t_n)$ iff $I(p)([\![t_1]\!]_{\mathfrak{S},\sigma}, \ldots, [\![t_n]\!]_{\mathfrak{S},\sigma})$ holds. 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 $\mathcal{T}(\mathcal{F})$ and interpretations I defined for every $f \in \mathcal{F}$ 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 $\mathfrak{I} \models_\vartheta H \ \mathsf{in}\ \mathsf{R}$ iff $H\vartheta \in [\![\mathsf{R}]\!]$.

Given a program P, its Herbrand base $\mathcal{B}_P$ 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 $(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F})^n$. Then an intended interpretation of P corresponds uniquely to a subset of $\mathcal{B}_P$. An *intended model* of P is an intended interpretation of P that is its model. We will write shortly $\mathcal{H}$-structure, $\mathcal{H}$-interpretation, $\mathcal{H}$-model for intended structures, interpretations, and models, respectively.

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 $\mathcal{H}$-model, which we denote by $lm(P, \mathcal{H})$.
2. If G is a goal then $P \models G$ iff $lm(P, \mathcal{H})$ is a model of G.

A *partially solved form of a constraint* $\mathcal{C}_1$ is a constraint $\mathcal{C}_2$ such that $\mathcal{C}_2$ is partially solved and $\mathfrak{I} \models \forall\left(\mathcal{C}_1 \leftrightarrow \overline{\exists}_{var(\mathcal{C}_1)}\mathcal{C}_2\right)$ for any $\mathcal{H}$-structure $\mathfrak{I}$.

A ground substitution ϑ is a *$\mathcal{H}$-solution* (or simply *solution*) of a constraint $\mathcal{C}$ if $\mathfrak{I} \models \mathcal{C}\vartheta$ for all $\mathcal{H}$-structures $\mathfrak{I}$. The notation $\models_{\mathcal{H}} \mathcal{C}$ stands for $\mathfrak{I} \models \mathcal{C}$ for all $\mathcal{H}$-structures $\mathfrak{I}$.

Theorem 1. *If the constraint $\mathcal{D}$ is solved, then $\mathfrak{I} \models \exists\mathcal{D}$ holds.*

Proof. Since $\mathcal{D}$ is solved, each disjunct $\mathcal{K}$ in it has a form $v_1 \doteq e_1 \wedge \cdots \wedge v_n \doteq e_n \wedge v'_1 \ \mathsf{in}\ \mathsf{R}_1 \wedge \cdots \wedge v'_m \ \mathsf{in}\ \mathsf{R}_m$ where $m, n \geq 0$, $v_i, v'_j \in \mathcal{V}$ and e_i is an expression corresponding to v_i. Moreover, $v_1, \ldots, v_n, v'_1, \ldots, v'_m$ are distinct and $[\![\mathsf{R}_j]\!] \neq \emptyset$ for all $1 \leq j \leq m$. Assume σ'_i is a grounding substitution for e_i for all $1 \leq i \leq n$, and let e'_j be an element of $[\![\mathsf{R}_j]\!]$ for all $1 \leq j \leq m$. Then $\sigma = \{v_1 \mapsto e_1\sigma'_1, \ldots, v_n \mapsto e_n\sigma'_n, v'_1 \mapsto e'_1, \ldots, v'_m \mapsto e'_m\}$ solves $\mathcal{K}$. Therefore, $\mathfrak{I} \models \exists\mathcal{D}$ holds.

5 Solver

We consider constraints in DNF: $\mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$, where $\mathcal{K}$'s are conjunctions of of true, false, and primitive constraints. The solver defined below transforms a constraint into a partially solved form. The solver is formulated in a rule-based way. The number of rules is not small (as it is usual for such kind of solvers, cf., e.g., [9, 7]). To make their comprehension easier, we group them so that similar ones are collected together in subsections. Within each subsection, for better readability, they are put in frames. In the rules, $\mathcal{K}$ stands for a maximal conjunction of primitive constraints. The rules are applied in any context.

5.1 Rules

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.

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

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

$(\mathsf{F1})$ $x \simeq (H_1, F(H), H_2) \rightsquigarrow \mathsf{false},$ if $x \in var(H)$.
$(\mathsf{F2})$ $\overline{x} \simeq (H_1, t, H_2) \rightsquigarrow \mathsf{false},$ if $\overline{x} \in var(H_1, t, H_2)$.
$(\mathsf{F3})$ $f_1(H_1) \simeq f_2(H_2) \rightsquigarrow \mathsf{false},$ if $f_1 \neq f_2$.
$(\mathsf{F4})$ $\epsilon \simeq (H_1, t, H_2) \rightsquigarrow \mathsf{false}$.
$(\mathsf{F5})$ $f_1(H)\ \mathsf{in}\ f_2(\mathsf{R}) \rightsquigarrow \mathsf{false},$ if $f_1 \neq f_2$.
$(\mathsf{F6})$ $\epsilon\ \mathsf{in}\ \mathsf{R} \rightsquigarrow \mathsf{false},$ if $\epsilon \notin [\![\mathsf{R}]\!]$,
$(\mathsf{F7})$ $(H_1, t, H_2)\ \mathsf{in}\ \mathsf{eps} \rightsquigarrow \mathsf{false}$.

Decomposition Rules. Each of the decomposition rules operates on a conjunction of constraint literals and gives back either a conjunction of constraint literals again, or constraints in DNF. We denote the set of rules $(\mathsf{D1})$ and $(\mathsf{D2})$ by Dec.

$(\mathsf{D1})$ $f_{\mathsf{u}}(H) \simeq f_{\mathsf{u}}(T) \wedge \mathcal{K} \rightsquigarrow \bigvee_{T' \in perm(T)} \big(H \doteq T' \wedge \mathcal{K}\big)$,
where H and T are disjoint.
$(\mathsf{D2})$ $(t_1, H_1) \simeq (t_2, H_2) \rightsquigarrow t_1 \doteq t_2 \wedge H_1 \doteq H_2,$ where $H_1 \neq \epsilon$ or $H_2 \neq \epsilon$.

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 $H_1 \neq \epsilon$.

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

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

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

(E4) $(\overline{x}, H_1) \simeq (T, t, H_2) \wedge \mathcal{K} \rightsquigarrow \bigvee_{T=(T_1,T_2)} \Big(\overline{x} \doteq T_1 \wedge H_1\vartheta \doteq (T_2, t, H_2)\vartheta \wedge \mathcal{K}\vartheta\Big)$
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) $X \simeq F \wedge \mathcal{K} \rightsquigarrow X \doteq F \wedge \mathcal{K}\vartheta$,
where $X \neq F$, $X \in var(\mathcal{K})$, and $\vartheta = \{X \mapsto F\}$. If F is a function variable, then in addition it is required that $F \in var(\mathcal{K})$.

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

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

We denote the set of rules (E1)–(E7) by Elim.

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 term sequences 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, 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 $(f(\mathsf{R}_1), \mathsf{R}_2)$ called *monomials*, which is defined recursively as follows:

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

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 rules are as follows:

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

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

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

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

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

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

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

(M8) v in $\mathsf{R}_1 \wedge v$ in $\mathsf{R}_2 \rightsquigarrow v$ in R, where $v \in \mathcal{V}_\mathsf{T} \cup \mathcal{V}_\mathsf{H}, [\![\mathsf{R}]\!] = [\![\mathsf{R}_1]\!] \cap [\![\mathsf{R}_2]\!]$.

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:

$$
\begin{array}{ll}
\text{(M9)} & \overline{x} \text{ in } f(\mathsf{R}) \wedge \mathcal{K} \rightsquigarrow \overline{x} \doteq x \wedge x \text{ in } f(\mathsf{R}) \wedge \mathcal{K}\{\overline{x} \mapsto x\}, \text{where } x \text{ is fresh.} \\
\text{(M10)} & X(H) \text{ in } f(\mathsf{R}) \wedge \mathcal{K} \rightsquigarrow X \doteq f \wedge f(H)\{X \mapsto f\} \text{ in } f(\mathsf{R}) \wedge \mathcal{K}\{X \mapsto f\}. \\
\text{(M11)} & f_{\mathsf{o}}(H) \text{ in } f_{\mathsf{o}}(\mathsf{R}) \rightsquigarrow H \text{ in } \mathsf{R}. \\
\text{(M12)} & f_{\mathsf{u}}(T) \text{ in } f_{\mathsf{u}}(\mathsf{R}) \wedge \mathcal{K} \rightsquigarrow \bigvee_{T' \in perm(T)} \Big(T' \text{ in } \mathsf{R} \wedge \mathcal{K}\Big).
\end{array}
$$

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

5.2 The Constraint Solving Algorithm

In this section, unless otherwise stated, by a constraint we mean a formula $\mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$, where $\mathcal{K}$'s are conjunctions of true, false, and primitive constraints. 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.

The constraint solving algorithm implements the strategy solve defined as a computation of a normal form with respect to step:

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

That means, step is applied to a constraint repeatedly as long as possible. It remains to show that this definition yields an algorithm, which amounts to proving that a constraint to which none of the rules Log, Fail, Del, Dec, Elim, and Memb applies, is produced by NF(step) for any constraint $\mathcal{C}$.

Theorem 2 (Termination of solve). solve *terminates on any input constraint.*

Proof (Sketch). We define a complexity measure $cm(\mathcal{C})$ for quantifier-free constraints in DNF, and show that $cm(\mathcal{C}') < cm(\mathcal{C})$ holds whenever $\mathcal{C}' = \mathsf{step}(\mathcal{C})$.

For a hedge H (resp. regular expression R), we denote by $size(H)$ (resp. by $size(\mathsf{R})$) its denotational length, e.g., $size(\mathsf{eps}) = 1$, $size(f(f(a)), \overline{x}) = 4$, and $size(f(f(a \cdot b^*))) = 6$. The complexity measure $cm(\mathcal{K})$ of a conjunction of primitive constraints $\mathcal{K}$ is the tuple $\langle N_1, M_1, N_2, M_2, M_3 \rangle$ defined as follows ($\{\!|\}\!|$ stands for a multiset):

- N_1 is the number of unsolved variables in $\mathcal{K}$.
- $M_1 := \{\!|size(H) \mid H \text{ in } \mathsf{R} \in \mathcal{K}, H \neq \epsilon|\!\}$.
- N_2 is the number of primitive constraints in the form v in R where $v \in \mathcal{V}$ plus the number of primitive constraints in the form $\overline{x}$ in R in $\mathcal{K}$.
- $M_2 := \{\!|size(\mathsf{R}) \mid H \text{ in } \mathsf{R} \in \mathcal{K}|\!\}$.

– $M_3 := \{\!|size(t_1) + size(t_2) \mid t_1 \doteq t_2 \in \mathcal{K}|\!\}$.

The complexity measure $cm(\mathcal{C})$ of a constraint $\mathcal{C} = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$ is defined as $\{\!|cm(\mathcal{K}_1), \ldots, cm(\mathcal{K}_n)|\!\}$. Measures are compared by the multiset extension of the lexicographic ordering on tuples. The Log rules strictly reduce the measure. For the other rules, the table below shows which rule reduces which component of the measure, which implies termination of the algorithm solve.

Rule	N_1	M_1	N_2	M_2	M_3
(M1),(M10),(E1)–(E7)	$>$				
(F5),(F7),(M2),(M3), (M11), (M12)	$\geq$	$>$			
(M8), (M9)	$\geq$	$\geq$	$>$		
(F6),(M4)–(M7)	$\geq$	$\geq$	$\geq$	$>$	
(D1), (D2), (F1)–(F4), (Del1)–(Del3)	$\geq$	$\geq$	$\geq$	$\geq$	$>$

The next lemma is needed to prove that the solver reduces a constraint to its equivalent constraint:

Lemma 1. *If* $\mathsf{step}(\mathcal{C}) = \mathcal{D}$, *then* $\models_{\mathcal{H}} \forall\Big(\mathcal{C} \leftrightarrow \overline{\exists}_{var(\mathcal{C})}\mathcal{D}\Big)$.

Theorem 3. *If* $\mathsf{solve}(\mathcal{C}) = \mathcal{D}$, *then* $\models_{\mathcal{H}} \forall\Big(\mathcal{C} \leftrightarrow \overline{\exists}_{var(\mathcal{C})}\mathcal{D}\Big)$ *and* $\mathcal{D}$ *is a partially solved form of* $\mathcal{C}$.

Proof. $\models_{\mathcal{H}} \forall\big(\mathcal{C} \leftrightarrow \overline{\exists}_{var(\mathcal{C})}\mathcal{D}\big)$ follows from Lemma 1 and the following property: If $\models_{\mathcal{H}} \forall\big(\mathcal{C}_1 \leftrightarrow \overline{\exists}_{var(\mathcal{C}_1)}\mathcal{C}_2\big)$ and $\models_{\mathcal{H}} \forall\big(\mathcal{C}_2 \leftrightarrow \overline{\exists}_{var(\mathcal{C}_2)}\mathcal{C}_3\big)$, then $\models_{\mathcal{H}} \forall\big(\mathcal{C}_1 \leftrightarrow \overline{\exists}_{var(\mathcal{C}_1)}\mathcal{C}_3\big)$. The property itself relies on the fact that $\models_{\mathcal{H}} \forall\big(\overline{\exists}_{var(\mathcal{C}_1)}\overline{\exists}_{var(\mathcal{C}_2)}\mathcal{C}_3 \leftrightarrow \overline{\exists}_{var(\mathcal{C}_1)}\mathcal{C}_3\big)$, which holds because all variables introduced by the rules of the solver in $\mathcal{C}_3$ are fresh not only for $\mathcal{C}_2$, but also for $\mathcal{C}_1$.

As for the partially solved form, by the definition of solve and Theorem 2, $\mathcal{D}$ is in a normal form. Assume by contradiction that it is not partially solved. By inspection of the solver rules, based on the definition of partially solved constraints, we can see that there is a rule that applies to $\mathcal{D}$. But this contradicts the fact that $\mathcal{D}$ is in a normal form. Hence, $\mathcal{D}$ is partially solved. By Lemma 1, we conclude that $\mathcal{D}$ is a partially solved form of $\mathcal{C}$.

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 [14]. A *state* is a pair $\langle G \parallel \mathcal{C}\rangle$, where G is the sequence of literals and $\mathcal{C} = \mathcal{K}_1 \vee \cdots \vee \mathcal{K}_n$, where $\mathcal{K}$'s are conjunctions of true, false, and primitive constraints. The *definition of an atom* $p(t_1, \ldots, t_m)$ *in program* P, $defn_P(p(t_1, \ldots, t_m))$, 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 $defn_P$ each time returns fresh variants.

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

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

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

for some $(p(r_1, \ldots, r_m) \leftarrow B) \in \mathit{defn}_P(L_i)$.
– If L_i is a atom and $\mathit{defn}_P(L_i) = \emptyset$, then it is reduced to $\langle \Box \parallel \mathsf{false}\rangle$.

A *derivation from a state* S in a program P 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 P. A *derivation from a goal* G in a program P 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 $\langle \Box \parallel \mathcal{C}\rangle$ where $\mathcal{C}$ is partially solved or false. If $\mathcal{C}$ is false, the derivation is said to be *failed*.

7 Well-Moded and KIF Programs

In this section we consider syntactic restrictions that lead to well-moded and KIF style CLP(H) programs. They are interesting, because the constraints that appear in derivations for such programs can be completely solved by solve.

7.1 Well-Moded Programs

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 $\mathit{invar}(L)$ and $\mathit{outvar}(L)$ the sets of variables occurring in terms in the input and output positions of p.

A sequence of literals $L_1, \ldots, L_n$ is *well-moded* if the following hold:

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

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

1. For all $1 \leq i \leq n$, $invar(L_i) \subseteq \bigcup_{j=1}^{i-1} outvar(L_j) \cup invar(A)$.
2. $outvar(A) \subseteq \bigcup_{j=1}^{n} outvar(L_j) \cup invar(A)$.
3. If for some $1 \leq i \leq n$, L_i is $H_1 \doteq H_2$, then $var(H_1) \subseteq \bigcup_{j=1}^{i-1} outvar(L_j) \cup invar(A)$ or $var(H_2) \subseteq \bigcup_{j=1}^{i-1} outvar(L_j) \cup invar(A)$.
4. If for some $1 \leq i \leq n$, 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 the first argument is the input position and the second argument is the output position in the user-defined predicates, 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.

Well-modedness is preserved by program derivation steps:

Lemma 2. *Let P be a well-moded* CLP(H) *program and $\langle G \parallel \mathcal{C} \rangle$ be a well-moded state. If $\langle G \parallel \mathcal{C} \rangle \rightarrowtail \langle G' \parallel \mathcal{C}' \rangle$ is a reduction using clauses in P, then $\langle G' \parallel \mathcal{C}' \rangle$ is also a well-moded state.*

The solver reduces well-moded constrains either to a solved form of to false:

Lemma 3. *Let $\mathcal{C}$ be a well-moded constraint and $\mathsf{solve}(\mathcal{C}) = \mathcal{C}'$, where $\mathcal{C}' \neq \mathsf{false}$. Then $\mathcal{C}'$ is solved.*

The theorem below is the main theorem 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 4. *Let $\langle G \parallel \mathsf{true} \rangle \rightarrowtail \cdots \rightarrowtail \langle \Box \parallel \mathcal{C}' \rangle$ be a finished derivation with respect to a well-moded* CLP(H) *program, starting from a well-moded goal G. If $\mathcal{C}' \neq \mathsf{false}$, then $\mathcal{C}'$ is solved.*

Proof. We prove a more general statement: Let $\langle G \parallel \mathsf{true} \rangle \rightarrowtail \cdots \rightarrowtail \langle G' \parallel \mathcal{C}' \rangle$ be a derivation with respect to a well-moded program, starting from a well-moded goal G and ending with G' that is either $\Box$ or consists only of atomic formulas without arguments (propositional constants). If $\mathcal{C}' \neq \mathsf{false}$, then $\mathcal{C}'$ is solved.

To prove this statement, we use induction on the length n of the derivation. When $n = 0$, then $\mathcal{C}' = \mathsf{true}$ and it is solved. Assume the statement holds when the derivation length is n, and prove it for the derivation with the length $n + 1$. Let such a derivation be $\langle G \parallel \mathsf{true} \rangle \rightarrowtail \cdots \rightarrowtail \langle G_n \parallel \mathcal{C}_n \rangle \rightarrowtail \langle G_{n+1} \parallel \mathcal{C}_{n+1} \rangle$. There are two possibilities to make the last step:

1. G_n has a form (modulo permutation) $L, p_1, \ldots, p_n$, where L is primitive constraint, the p's are propositional constants, $G_{n+1} = p_1, \ldots, p_n$, and $\mathcal{C}_{n+1} = \mathsf{solve}(\mathcal{C}_n \wedge L)$.
2. G_n has a form (modulo permutation) $q, p_1, \ldots, p_n$, where q and p's are propositional constants, $G_{n+1} = p_1, \ldots, p_n$, and $\mathcal{C}_{n+1} = \mathcal{C}_n$.

In the first case, note that by Lemma 2, $\langle G_n \parallel \mathcal{C}_n \rangle$ is well-moded. Since the p's have no influence on well-modedness (they are just propositional constants), $\mathcal{C}_n \wedge L$ is well-moded. By Lemma 3 we get that if $\mathcal{C}_{n+1} = \mathsf{solve}(\mathcal{C}_n \wedge L) \neq \mathsf{false}$ then $\mathcal{C}_{n+1}$ is solved.

In the second case, since G_n consists of propositional constants only, by the induction hypothesis we have that if $\mathcal{C}_n$ is not false, then it is solved. But $\mathcal{C}_n = \mathcal{C}_{n+1}$. It finishes the proof.

7.2 Programs in the KIF Form

A term is in the *KIF form* (*KIF-term*) if hedge variables occur only below ordered function symbols,[6] and they occupy only the last argument position in each subterm where they appear. For example, the term $f_{\mathsf{o}}(x, f_{\mathsf{o}}(a, \overline{x}), f_{\mathsf{u}}(x, b), \overline{x})$ is in the KIF form, while $f_{\mathsf{o}}(\overline{x}, a, \overline{x})$, $f_{\mathsf{u}}(x, f_{\mathsf{o}}(a, \overline{x}), f_{\mathsf{u}}(x, b), \overline{x})$ are not. A hedge (T, h) is in the KIF form, if T is a sequence of KIF-terms and h is either a KIF-term or a hedge variable.

An atom $p(t_1, \ldots, t_n)$ (including $t_1 \doteq t_2$) is in the KIF form, if each t_i, $1 \leq i \leq n$, is a KIF-term. A membership atom H in R is in the KIF-form, if H is a KIF-hedge. A CLP(H) program is in the KIF form, if it is constructed from literals in the KIF form. Note that the programs in examples 1 and 2 are not KIF programs. One could rewrite them in this form, but the code size would become a bit larger.

The notion of KIF form extends naturally to constraints and states, requiring that all their literals should be in the KIF form. KIF-programs, like well-moded ones discussed above, also show a good behavior. As the lemmas below state, reductions preserve the KIF form and the solver is complete:

Lemma 4. *Let P be a* CLP(H) *program in the KIF form and $\langle G \parallel \mathcal{C} \rangle$ be a KIF-state. If $\langle G \parallel \mathcal{C} \rangle \rightarrowtail \langle G' \parallel \mathcal{C}' \rangle$ is a reduction using clauses in P, then $\langle G' \parallel \mathcal{C}' \rangle$ is also a KIF-state.*

Lemma 5. *Let $\mathcal{C}$ be a KIF-constraint and $\mathsf{solve}(\mathcal{C}) = \mathcal{C}'$, where $\mathcal{C}' \neq \mathsf{false}$. Then $\mathcal{C}'$ is solved.*

We illustrate how to solve a simple KIF constraint:

Example 4. Let $\mathcal{C} = f(x, \overline{x}) \doteq f(g(\overline{y}), a, \overline{y}) \wedge \overline{x}$ in $a(\mathtt{eps})^* \wedge \overline{y}$ in $a(\mathtt{eps}) \cdot a(b(\mathtt{eps})^*)^*$. Then solve performs the following derivation:

$$\begin{aligned}
\mathcal{C} &\rightsquigarrow^2 x \doteq g(\overline{y}) \wedge \overline{x} \doteq (a, \overline{y}) \wedge (a, \overline{y}) \;\mathsf{in}\; a(\mathtt{eps})^* \wedge \overline{y} \;\mathsf{in}\; a(\mathtt{eps}) \cdot a(b(\mathtt{eps})^*)^* \\
&\rightsquigarrow x \doteq g(\overline{y}) \wedge \overline{x} \doteq (a, \overline{y}) \wedge \overline{y} \;\mathsf{in}\; a(\mathtt{eps})^* \wedge \overline{y} \;\mathsf{in}\; a(\mathtt{eps}) \cdot a(b(\mathtt{eps})^*)^* \\
&\rightsquigarrow x \doteq g(\overline{y}) \wedge \overline{x} \doteq (a, \overline{y}) \wedge \overline{y} \;\mathsf{in}\; a(\mathtt{eps}) \cdot a(\mathtt{eps})^*
\end{aligned}$$

The obtained constraint is solved.

[6] If the language does not contain unordered function symbols, then hedge variables are permitted under function symbols as well.

The theorem below is the main theorem for KIF programs and can be proved similarly to the analogous theorem for well-moded programs (Theorem 4). It states that any finished derivation from a KIF-goal with respect to a KIF-program leads to a solved constraint or to a failure:

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

8 Conclusion

We defined a semantics for CLP(H) programs and introduced a solver for positive equational and membership constraints over hedges. The solver, in general, is incomplete. It is natural, since hedge unification is infinitary. We identified two special cases of CLP(H) programs which lead to constraints, for which the solver computes a complete set of solutions.

Acknowledgments

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 Rustaveli Science Foundation under the grants DI/16/4-120/11 and FR/611/4-102/12.

References

1. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In *Proc. RTA'07*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
3. J. Coelho and M. Florido. CLP(Flex): constraint logic programming applied to XML processing. In R. Meersman and Z. Tari, editors, *CoopIS/DOA/ODBASE (2)*, volume 3291 of *LNCS*, pages 1098–1112. Springer, 2004.
4. J. Coelho and M. Florido. VeriFLog: a constraint logic programming approach to verification of website content. In H. T. Shen, J. Li, M. Li, J. Ni, and W. Wang, editors, *APWeb Workshops*, volume 3842 of *LNCS*, pages 148–156. Springer, 2006.
5. J. Coelho and M. Florido. XCentric: logic programming for XML processing. In I. Fundulaki and N. Polyzotis, editors, *WIDM*, pages 1–8. ACM, 2007.
6. A. Colmerauer. An introduction to Prolog III. *Commun. ACM*, 33(7):69–90, 1990.
7. H. Comon. Completion of rewrite systems with membership constraints. Part II: constraint solving. *J. Symb. Comput.*, 25(4):421–453, 1998.

8. N. Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.
9. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, 2000.
10. A. Dovier, C. Piazza, and G. Rossi. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. *ACM Trans. Comput. Log.*, 9(3), 2008.
11. B. Dundua, M. Florido, T. Kutsia, and M. Marin. Constraint logic programming for hedges: A semantic reconstruction. RISC Report Series 14-02, RISC, University of Linz, Austria, 2014.
12. M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Stanford University, Stanford, CA, USA, 1992.
13. H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *J. Funct. Program.*, 13(6):961–1004, 2003.
14. J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *J. Log. Program.*, 37(1-3):1–46, 1998.
15. T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symb. Comput.*, 42(3):352–388, 2007.
16. T. Kutsia and M. Marin. Solving, reasoning, and programming in Common Logic. In *SYNASC*, pages 119–126. IEEE Computer Society, 2012.
17. M. Marin and T. Kutsia. Foundations of the rule-based system ρlog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.
18. A. Rajasekar. Constraint logic programming on strings: Theory and applications. In *SLP*, page 681, 1994.
19. M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In R. Wilhelm, editor, *CC*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer, 2001.
20. C. Walinsky. CLP(Σ^*): constraint logic programming with regular sets. In G. Levi and M. Martelli, editors, *ICLP*, pages 181–196. MIT Press, 1989.
21. S. Wolfram. *The Mathematica book*. Wolfram-Media, fifth edition, 2003.