CLP(\(H\)): Constraint Logic Programming for Hedges

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

MÁRIO FLORIDO
DCC-FC and LIACC, University of 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 Timișoara, Romania (e-mail: mmarin@info.uvt.ro)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

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.

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),

This is an extended version of a paper presented at the Twelfth 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.
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 Sect. 3. We call this language CLP(\textit{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(\textit{H}) can be seen also as a generalization of CLP over strings CLP(\textit{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 π can be instantiated with a hedge of minimal length 1 and maximal length 3, we can write a disjunction π ≜ x ∨ π ≜ (x₁, x₂) ∨ π ≜ (x₁, x₂, x₃), where the lower case x’s are term variables.

Flexibility and the expressive power of CLP(\textit{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 ground expressions at some point. This effectively reduces constraint solving to hedge matching (Kutsia and Marin 2005a; 2005b), 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(\textit{H}) programs which give rise to well-moded or KIF constraints.\footnote{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 \textit{L}_{\textit{\Lambda}} uses unitary unification for higher-order patterns instead of undecidable higher-order unification.}
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.

2 Preliminaries

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

- $\mathcal{V}_T$: term variables, denoted by $x, y, z, \ldots$,
- $\mathcal{V}_H$: hedge variables, denoted by $\mathfrak{x}, \mathfrak{y}, \mathfrak{z}, \ldots$,
- $\mathcal{V}_F$: function variables, denoted by $X, Y, Z, \ldots$,
- $\mathcal{F}_u$: unranked unordered function symbols, denoted by $f, g, h, \ldots$,
- $\mathcal{F}_o$: unranked ordered function symbols, denoted by $f_0, g_0, h_0, \ldots$,
- $\mathcal{P}$: 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 $\equiv$, and the unranked membership predicate in.
- Regular operators: $\epsilon, \cdot, +, \ast$.
- Logical connectives and quantifiers: $\neg, \lor, \land, \rightarrow, \leftrightarrow, \exists, \forall$.
- Auxiliary symbols: parentheses and the comma.

Function symbols, denoted by $f, g, h, \ldots$, are elements of the set $\mathcal{F} = \mathcal{F}_u \cup \mathcal{F}_o$. A variable is an element of the set $\mathcal{V} = \mathcal{V}_T \cup \mathcal{V}_H \cup \mathcal{V}_F$. 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:

\[
\begin{align*}
t & ::= x \mid f(H) \mid X(H) & & \text{Term} \\
T & ::= t_1, \ldots, t_n \quad (n \geq 0) & & \text{Term sequence} \\
h & ::= t \mid \mathfrak{t} & & \text{Hedge element} \\
H & ::= h_1, \ldots, h_n \quad (n \geq 0) & & \text{Hedge}
\end{align*}
\]

We denote the set of terms by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and the set of ground (i.e., variable-free) terms by $\mathcal{T}(\mathcal{F})$. 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 $\epsilon$. 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., \((f(a), \pi, b)\) instead of \(f(a), \pi, b\). For hedges \(H = (h_1, \ldots, h_n)\) and \(H' = (h_1', \ldots, h_m')\), the notation \((H, H')\) stands for the hedge \((h_1, \ldots, h_n, h_1', \ldots, h_m')\).

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 \(p \in \mathcal{P}\) is an \(n\)-ary predicate symbol. Atoms are denoted by \(A\).

Regular hedge expressions \(R\) are defined inductively:

\[
R ::= \text{eps} \mid (R \cdot R) \mid R + R \mid R^* \mid f(R)
\]

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

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 \(\text{var}(O)\). We let \(\exists_v N\) denote the formula \(\exists v_1 \cdots \exists v_n N\), where \(V = \{v_1, \ldots, v_n\} \subseteq V\). \(\exists_v N\) denotes \(\exists_{\text{var}(N) \setminus V} N\). We write \(\exists N\) (resp. \(\forall N\)) for the existential (resp. universal) closure of \(N\). We refer to a language over the alphabet \(A\) as \(\mathcal{L}(A)\).

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 \(\sigma\), we write \(e \sigma\) for the instance of \(e\) under \(\sigma\). This is a standard operation that replaces in \(e\) each free occurrence of a variable \(v\) by its image under \(\sigma\), i.e., by \(\sigma(v)\). If needed, bound variables are renamed to avoid variable capture. For instance, for the constraint \(C = \forall x.f(X(a, \pi), \pi) = f(g(\pi, a, b, x), b, x)\) and the substitution \(\sigma = \{X \mapsto g, \pi \mapsto (b, x), \pi \mapsto e, x \mapsto f(c)\}\), we have \(C \sigma = \forall x.f(g(a, b, x), b, x) = f(g(a, z), b, z)\). A substitution \(\sigma\) 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 \(\forall (L_1 \land \cdots \land L_n \rightarrow A)\), \(n \geq 0\), 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 \(\exists (L_1 \land \cdots \land L_n)\), \(n \geq 0\), 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 \(\mathcal{K} = c_1 \land \cdots \land c_n\), if there is a \(c_i\), \(1 \leq i \leq n\), such that

- the variable is \(x\), \(c_i = x \equiv t\), and \(x\) occurs neither in \(t\) nor elsewhere in \(\mathcal{K}\), or
- the variable is \(\pi\), \(c_i = \pi \equiv H\), and \(\pi\) occurs neither in \(H\) nor elsewhere in \(\mathcal{K}\), or
- the variable is \(X\), \(c_i = X \equiv F\) and \(X\) occurs neither in \(F\) nor elsewhere in \(\mathcal{K}\), or
- the variable is \(x\), \(c_i = x \equiv f(R)\) and \(x\) does not occur in membership constraints elsewhere in \(\mathcal{K}\), or
- the variable is \(\pi\), \(c_i = \pi \equiv R\), \(\pi\) does not occur in membership constraints elsewhere in \(\mathcal{K}\), and \(R\) has the form \(R_1 \cdot R_2\) or \(R_1^*\).
In this case we also say that $c_i$ is solved in $K$. Moreover, $K$ is called solved if for any $1 \leq i \leq n$, $c_i$ is solved in it. $K$ is called partially solved, if for any $1 \leq i \leq n$, $c_i$ is solved in $K$, or has one of the following forms:

- Membership atom:
  - $f_u(H_1, \bar{x}, H_2)$ in $f_u(R)$.
  - $(\bar{x}, H)$ in $R$ where $H \neq \epsilon$ and $R$ has the form $R_1 \cdot R_2$ or $R_1^*$.

- Equation:
  - $(\bar{x}, H_1) \doteq (\bar{y}, H_2)$ where $\bar{x} \neq \bar{y}$, $H_1 \neq \epsilon$ and $H_2 \neq \epsilon$.
  - $(\bar{x}, H_1) \doteq (T, \bar{y}, H_2)$ where $\bar{x} \not\in \text{var}(T)$, $H_1 \neq \epsilon$, and $T \neq \epsilon$. The variables $\bar{x}$ and $\bar{y}$ are not necessarily distinct.
  - $f_u(H_1, \bar{x}, H_2) \doteq f_u(H_3, \bar{y}, H_4)$ where $(H_1, \bar{x}, H_2)$ and $(H_3, \bar{y}, H_4)$ 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

$$\text{rewrite}(x, y) \leftarrow \text{rule}(x, y)$$

and the recursive case

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

where $x, y$ are term variables, $\bar{x}, \bar{y}$ 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

$$\text{rule}(X(\bar{x}_1, x_2), X(\bar{y})) \leftarrow \bar{x}_1 \doteq f(a^*) \cdot b^*, \bar{x}_1 \doteq (x, \bar{x}), \bar{y} \doteq (x, f(\bar{x})),$$

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

\(^2\) In the notation defined in the previous section, strictly speaking, we need to write this constraint as $f(a(\text{eps})^*) \cdot b(\text{eps})^*$. However, for brevity and clarity of the presentation we omit eps here.
With this program, the goal \( \leftarrow \text{rewrite}(f(f(f(a,a), b)), x) \) 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 \( \leftarrow \text{rewrite}(f(f(a,a), b), y) \) together with the constraint \( x \overset{\tau}{=} f(y) \) for \( x \). The next transformation is performed by the base case of the \text{rewrite} predicate, resulting into the goal \( \leftarrow \text{rule}(f(f(a,a), b), y) \).

This goal is then transformed by the \text{rule} clause, which gives the constraint \( X(\overline{x}_1, \overline{x}_2) \overset{\tau}{=} f(f(a,a), b) \land y = X(\overline{y}) \land \overline{x}_1 \in f(a^* \cdot b^* \land \overline{x}_1 \overset{\tau}{=} (x', \overline{x}) \land \overline{y} \overset{\tau}{=} (x', f(\overline{x})) \land x \overset{\tau}{=} f(y) \). This constraint has two solutions, depending whether \( \overline{x}_1 \) equals \( f(a,a) \) or to \( (f(a,a), b) \). From one we get \( x \overset{\tau}{=} f(f(f(a,a), f)) \), and from the other \( x \overset{\tau}{=} f(f(f(a,a), f(b))) \). These solutions give the above mentioned answers.

### Example 2

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

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

To implement this definition in CLP(H), we use the predicate \text{rpo} between two terms, and four helper predicates: \text{rpo}_{\text{all}} to implement the comparison \( s >_\text{rpo} t_i \) for all \( i \); \text{prec} to implement the comparison depending on the precedence; \text{ext} to implement the comparison with respect to an extension of \( >_{\text{rpo}} \); and \text{status} to give the status of a function symbol. The predicate \text{lex} implements \( >_{\text{rpo}} \) and \text{mul} implements \( >_{\text{rpo}} \). 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. \( >_{\text{rpo}} \) requires four clauses, since there are four alternatives in the definition:

1. \( \text{rpo}(X(\overline{x}, x, \overline{y}), x) \),
   \( \text{rpo}(X(\overline{x}, x, \overline{y}), y) \leftarrow \text{rpo}(x, y) \).  
2a. \( \text{rpo}(X(\overline{x}), Y(\overline{y})) \leftarrow \text{rpo}_{\text{all}}(X(\overline{x}), (\overline{y})), \text{prec}(X, Y) \).
2b. \( \text{rpo}(X(\overline{x}), X(\overline{y})) \leftarrow \text{rpo}_{\text{all}}(X(\overline{x}), (\overline{y})), \text{ext}(X(\overline{x}), X(\overline{y})) \).

\text{rpo}_{\text{all}} is implemented with recursion:

\( \text{rpo}_{\text{all}}(x, \langle \rangle) \),
\( \text{rpo}_{\text{all}}(x, (y, \overline{y})) \leftarrow \text{rpo}(x, y), \text{rpo}_{\text{all}}(x, (\overline{y})) \).

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

\( \text{ext}(X(\overline{x}), X(\overline{y})) \leftarrow \text{status}(X, \text{lex}), \text{lex}((\overline{x}), (\overline{y})) \).
\( \text{ext}(X(\overline{x}), X(\overline{y})) \leftarrow \text{status}(X, \text{mul}), \text{mul}((\overline{x}), (\overline{y})) \).
status can be given as a set of facts, lex needs one clause, and mul requires three:

\[
\begin{align*}
\text{lex}(\langle x, y \rangle, \langle y, z \rangle) & \leftarrow \text{rpo}(x, y). \\
\text{mul}(\{x, y\}) & \leftarrow. \\
\text{mul}(\{x, y\}) & \leftarrow \text{mul}(\{y\}). \\
\text{mul}(\{x, y\}) & \leftarrow \text{rpo}(x, y), \text{mul}(\{x, y\}).
\end{align*}
\]

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 \( \epsilon \). Given a sequence \( s = (s_1, s_2, \ldots, s_n) \in S^n \), we denote by \( \text{perm}(s) \) 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 \( \mathcal{G} \) for a language \( \mathcal{L}(A) \) 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 \( f \in F \) to a function \( I(f) : D^n \rightarrow D \), and each \( n \)-ary predicate symbol \( p \in P \) to an \( n \)-ary relation \( I(p) \subseteq D^n \). Moreover, if \( f \in F_0 \) then \( I(f)(s) = I(f)(s') \) for all \( s, s' \in D^* \) and \( s' \in \text{perm}(s) \).

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 \( \mathcal{G} = \langle D, I \rangle \) and variable assignment \( \sigma \) is shown below. The interpretations \( [H]_{\mathcal{G}, \sigma} \) of hedges (including terms) is defined as follows:

\[
\begin{align*}
[v]_{\mathcal{G}, \sigma} & := \sigma(v), \quad \text{where } v \in \mathcal{V}_T \cup \mathcal{V}_H. \\
[I(H)]_{\mathcal{G}, \sigma} & := I(f)([H]_{\mathcal{G}, \sigma}). \\
[X(H)]_{\mathcal{G}, \sigma} & := \sigma(X)([H]_{\mathcal{G}, \sigma}). \\
[h_1, \ldots, h_n]_{\mathcal{G}, \sigma} & := (h_1)_{\mathcal{G}, \sigma}, \ldots, (h_n)_{\mathcal{G}, \sigma}.
\end{align*}
\]

Note that terms are interpreted as elements of \( D \) and hedges as elements of \( D^* \). We may omit \( \sigma \) and write simply \( [E]_{\mathcal{G}} \) for the interpretation of a ground expression \( E \). The interpretation of regular expressions is defined as follows:

\[
\begin{align*}
[\epsilon]_{\mathcal{G}} & := \{ \epsilon \}. \\
[I(R)]_{\mathcal{G}} & := \{ I(f)(H) \mid H \in [R]_{\mathcal{G}} \}. \\
[R_1 + R_2]_{\mathcal{G}} & := [R_1]_{\mathcal{G}} \cup [R_2]_{\mathcal{G}}. \\
[R_1 \cdot R_2]_{\mathcal{G}} & := \{(H_1, H_2) \mid H_1 \in [R_1]_{\mathcal{G}}, H_2 \in [R_2]_{\mathcal{G}}\}. \\
[R^*]_{\mathcal{G}} & := [R]_{\mathcal{G}}.
\end{align*}
\]

Primitive constraints are interpreted with respect to a structure \( \mathcal{G} \) and variable assignment \( \sigma \) as follows:

\[
\mathcal{G} \models_{\sigma} t_1 \doteq t_2 \text{ iff } [t_1]_{\mathcal{G}, \sigma} = [t_2]_{\mathcal{G}, \sigma}.
\]
\[ S|\sigma H \in R \text{ iff } [H]_{\sigma,\sigma} \in [R]_{\sigma}. \]

\[ S|\sigma p(t_1, \ldots, t_n) \text{ iff } I(p)([t_1]_{\sigma,\sigma}, \ldots, [t_n]_{\sigma,\sigma}). \]

The notions \( S|\sigma N \) for validity of an arbitrary formula \( N \) in \( S \), and \( |\sigma N \) for validity of \( N \) in any structure are defined in the standard way.

An intended structure is a structure \( I \) with the carrier set \( \mathcal{T}(F) \) and interpretations \( I \) defined for every \( f \in F \) by \( I(f)(H) := f(H) \). Thus, intended structures identify terms and hedges by themselves. Also, if \( R \) is any regular hedge expression then \( [R]_I \) is the same in all intended structures, and will be denoted by \([R]\). Other remarkable properties of intended structures \( I \) are: Variable assignments are substitutions, \( I|\sigma t_1 \equiv t_2 \) iff \( t_1 \sigma \equiv t_2 \), and \( I|\sigma H \in R \) iff \( H \sigma \in [R] \).

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 \( (t_1, \ldots, t_n) \in \mathcal{T}(F)^n \). 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.

As usual, we will write \( P|\sigma = G \) 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 \( \text{lm}(P) \).
2. If \( G \) is a goal then \( P|\sigma = G \) iff \( \text{lm}(P) \) is a model of \( G \).

A ground substitution \( \sigma \) is an intended solution (or simply solution) of a constraint \( C \) if \( I|\sigma \models C \sigma \) for all intended structures \( I \).

**Theorem 1**

If the constraint \( C \) is solved, then \( I|\sigma \models \exists C \) holds for all intended structures \( I \).

### 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 \( K_1 \lor \cdots \lor K_n \), where \( K \)'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., (Dovier et al. 2000; Comon 1998)).

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 \( F_1() \equiv F_2() \) and \( f_a(H_1) \equiv f_a(H_2) \) we write \( F_1 \equiv F_2 \) and \( H_1 \equiv H_2 \) respectively. The symmetric closure of the relation \( \equiv \) 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 \( L \land K \), it is intended to act on an entire conjunct of the DNF, modulo associativity and commutativity of \( \land \). These assumptions guarantee that the constraint obtained after each rule application is again in DNF.
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 \( \text{Log} \).

\[
\begin{align*}
N \land N & \leadsto N \quad \text{false} \land N \leadsto \text{false} \\
N \lor N & \leadsto N \quad \text{true} \land N \leadsto N \\
\text{false} & \land N \leadsto \text{false} \\
\text{true} & \land N \leadsto \text{false} \\
\text{true} & \lor N \leadsto N \\
H = H & \leadsto \text{true}
\end{align*}
\]

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 \( \text{Fail} \).

\[
\begin{align*}
(F1) \quad x & \simeq (H_1, F(H), H_2) \leadsto \text{false}, \text{ if } x \in \text{var}(H). \\
(F2) \quad \pi & \simeq (H_1, t, H_2) \leadsto \text{false}, \text{ if } \pi \in \text{var}(H_1, t, H_2). \\
(F3) \quad f_1(H_1) & \simeq f_2(H_2) \leadsto \text{false}, \text{ if } f_1 \neq f_2. \\
(F4) \quad \epsilon & \simeq (H_1, t, H_2) \leadsto \text{false}. \\
(F5) \quad f_1(H) & \text{ in } f_2(R) \leadsto \text{false}, \text{ if } f_1 \neq f_2. \\
(F6) \quad \epsilon & \text{ in } R \leadsto \text{false}, \text{ if } \epsilon \notin [R]. \\
(F7) \quad (H_1, t, H_2) & \text{ in } \text{eps} \leadsto \text{false}
\end{align*}
\]

Decomposition Rules.

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

\[
\begin{align*}
(D1) \quad f_u(H) & \simeq f_u(T) \land \kappa \leadsto \bigvee_{T' \in \text{perm}(T)} (H \doteq T' \land \kappa), \text{ where } H \text{ and } T \text{ are disjoint.} \\
(D2) \quad (t_1, H_1) & \simeq (t_2, H_2) \leadsto t_1 \doteq t_2 \land H_1 \doteq H_2, \text{ where } H_1 \neq \epsilon \text{ or } H_2 \neq \epsilon.
\end{align*}
\]
Deletion Rules.

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

\((\text{Del}1)\) \(x, H_1 \simeq x, H_2 \Rightarrow H_1 \doteq H_2.\)

\((\text{Del}2)\) \(f_u(H_1, h, H_2) \simeq f_u(H_3, h, H_4) \Rightarrow f_u(H_1, H_2) \doteq f_u(H_3, H_4).\)

\((\text{Del}3)\) \(x \simeq (H_1, x, H_2) \Rightarrow H_1 \doteq \epsilon \land H_2 \doteq \epsilon, \text{ if } H_1 \neq \epsilon.\)

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:

\((\text{E1})\) \(x \simeq t \land K \Rightarrow x \doteq t \land K\vartheta,\)

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

\((\text{E2})\) \(x \simeq H \land K \Rightarrow x \doteq H \land K\vartheta,\)

where \(x \notin \text{var}(H), x \in \text{var}(K),\) and \(\vartheta = \{x \mapsto H\}.\) If \(H = \overline{y}\) for some \(y,\) then in addition it is required that \(\overline{y} \in \text{var}(K).\)

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 \(\overline{\pi}\) can contain it.

\((\text{E3})\) \((\overline{\pi}, H) \simeq T \land K \Rightarrow \bigvee_{T=(T_1, T_2)} \left(\overline{\pi} \doteq T_1 \land H\vartheta \doteq T_2 \land K\vartheta\right),\)

where \(\overline{\pi} \notin \text{var}(T), \vartheta = \{\overline{\pi} \mapsto T_1\},\) and \(H \neq \epsilon.\)

\((\text{E4})\) \((\overline{\pi}, H_1) \simeq (T, t, H_2) \land K \Rightarrow \bigvee_{T=(T_1, T_2)} \left(\overline{\pi} \doteq T_1 \land H_1\vartheta \doteq (T_2, t, H_2)\vartheta \land K\vartheta\right)\)

where \(\overline{\pi} \notin \text{var}(T), \overline{\pi} \in \text{var}(t), \vartheta = \{\overline{\pi} \mapsto T_1\},\) and \(H_1 \neq \epsilon.\)

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

\((\text{E5})\) \(X \simeq F \land K \Rightarrow X \doteq F \land K\vartheta,\)

where \(X \neq F, X \in \text{var}(K),\) and \(\vartheta = \{X \mapsto F\}.\) If \(F\) is a function variable, then in addition it is required that \(F \in \text{var}(K).\)
These equations involve an extension of concatenation

\[ X(H_1) \approx F(H_2) \land K \simeq X \approx F \land F(H_1)\vartheta \approx F(H_2)\vartheta \land K\vartheta. \]

where \( X \neq F \), \( \vartheta = \{X \mapsto F\} \), and \( H_1 \neq \epsilon \) or \( H_2 \neq \epsilon \).

\[ X(H_1) \approx X(H_2) \land K \approx \bigvee_{f \in \mathcal{F}} (X \approx f \land f(H_1)\vartheta \approx f(H_2)\vartheta \land K\vartheta), \]

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 \( \mathcal{F} \) guarantees that the disjunction in (E7) is finite.

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, 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 \( \lf(R) \), is a finite set of pairs \((f(R_1), R_2)\), which is defined recursively as follows:

\[
\lf(\epsilon) = \emptyset.
\]

\[
\lf(f(R)) = \{(f(R), \epsilon)\}.
\]

\[
\lf(R_1 + R_2) = \lf(R_1) \cup \lf(R_2).
\]

\[
\lf(R_1 \cdot R_2) = \lf(R_1) \circ R_2, \text{ if } \epsilon \notin [R_1].
\]

\[
\lf(R_1 \cdot R_2) = \lf(R_1) \circ R_2 \cup \lf(R_2), \text{ if } \epsilon \in [R_1].
\]

\[
\lf(R^*) = \lf(R) \circ R^*.
\]

These equations involve an extension of concatenation \( \circ \) that acts on a linear form and a regular expression and returns a linear form. It is defined as \( l \circ \epsilon = l \), and \( l \circ R = \{(f(R_1), R_2) \mid (f(R_1), R_2) \in l, R_2 \neq \epsilon\} \cup \{(f(R_1), R) \mid (f(R_1), \epsilon) \in l\} \), if \( R \neq \epsilon \).

The linear form \( \lf(R) \) of a regular expression \( R \) has the property (Antimirov 1996)\(^3\):

\[
[R] \setminus \{\epsilon\} = \bigcup_{f(R_1), R_2 \in \lf(R)} [f(R_1) \cdot R_2],
\]

which justifies its use in the rule M2 below.

The first group of membership rules looks as follows:

\[ (\pi_1, \ldots, \pi_n) \in \epsilon \land K \land \bigwedge_{i=1}^{n} \pi_i = \epsilon \land K\vartheta, \]

where \( \vartheta = \{\pi_1 \mapsto \epsilon, \ldots, \pi_n \mapsto \epsilon\} \), \( n > 0 \).

\(^3\) 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.
B. Dundua, M. Florido, T. Kutsia, and M. Marin

\[(M2)\] \((t, H) \in R \land K \leadsto \bigvee_{(f(R_1), R_2) \in f(R)} \big(t \in f(R_1) \land H \in R_2 \land K\big),\]
where \(H \neq \epsilon\) and \(R \neq \text{eps}\).

\[(M3)\] \((\mathcal{T}, H) \in f(R) \land K \leadsto (\mathcal{T} \in f(R) \land H = \epsilon \land K) \lor (\mathcal{T} = \epsilon \land H \in f(R) \land K),\]
where \(H \neq \epsilon\).

\[(M4)\] \(t \in R^* \leadsto t \in R\).

\[(M5)\] \(t \in R_1 \cdot R_2 \land K \leadsto (t \in R_1 \land \epsilon \in R_2 \land K) \lor (\epsilon \in R_1 \land t \in R_2 \land K)\).

\[(M6)\] \(t \in R_1 + R_2 \land K \leadsto (t \in R_1 \land K) \lor (t \in R_2 \land K)\).

\[(M7)\] \((\mathcal{T}, H) \in R_1 + R_2 \land K \leadsto (\mathcal{T}, H) \in R_1 \land K) \lor (\mathcal{T}, H) \in R_2 \land K\).

\[(M8)\] \(v \in R_1 \lor v \in R_2 \leadsto v \in R,\)
where \(v \in \mathcal{V}_T \cup \mathcal{V}_H, [R] = [R_1] \cap [R_2]\), and neither \(v \in R_1\) nor \(v \in R_2\) can be transformed by the other rules.

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

\[
\text{step} := \text{first(Log, Fail, Del, Dec, Elim, Memb)}.
\]

When applied to a constraint, \text{step} transforms it by the \textit{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 \textit{normal form} with respect to \text{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:

\[
\text{solve} := \text{compose}(\text{dnf}, \text{NF}(\text{step})).
\]

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.

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

**Lemma 1**

If \( \text{step}(C) = D \), then \( \mathfrak{I} \models C \iff \exists \text{var}(C) \mathfrak{I} \models D \) for all intended structures \( \mathfrak{I} \).

**Theorem 3**

If \( \text{solve}(C) = D \), then \( \mathfrak{I} \models C \iff \exists \text{var}(C) \mathfrak{I} \models D \) for all intended structures \( \mathfrak{I} \), and \( D \) 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 \( \langle G \parallel C \rangle \), where \( G \) is the sequence of literals and \( C = K_1 \lor \cdots \lor K_n \), where \( K \)'s are conjunctions of true, false, and primitive constraints. The **definition of an atom** \( p(t_1, \ldots, t_m) \) in program \( P \), \( \text{defn}_P(p(t_1, \ldots, t_m)) \), 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 \( \text{defn}_P \) each time returns fresh variants.

A state \( \langle L_1, \ldots, L_n \parallel C \rangle \) can be reduced with respect to \( P \) as follows: Select a literal \( L_i \). Then:

- If \( L_i \) is a primitive constraint and \( \text{solve}(C \land L_i) \neq \text{false} \), then it is reduced to \( \langle L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n \parallel \text{solve}(C \land L_i) \rangle \).
- If \( L_i \) is a primitive constraint and \( \text{solve}(C \land L_i) = \text{false} \), then it is reduced to \( \langle \square \parallel \text{false} \rangle \).
- If \( L_i \) is an atom and \( \text{defn}_P(L_i) = \emptyset \), then it is reduced to \( \langle \square \parallel \text{false} \rangle \).

A derivation from a state \( S \) in a program \( P \) is a finite or infinite sequence of states \( S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_n \rightarrow \cdots \) 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 \( \langle G \parallel \text{true} \rangle \). The length of a (finite) derivation of the form \( S_0 \rightarrow S_1 \rightarrow \cdots \rightarrow S_n \) is \( n \). A
derivation is finished if the last goal cannot be reduced, that is, if its last state is of the form \( \langle \Box | \| \mathcal{C} \rangle \) where \( \mathcal{C} \) is partially solved or false. If \( \mathcal{C} \) 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.

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 \( m_p : \{1,\ldots,n\} \rightarrow \{i, o\} \). If \( m_p(i) = i \) (resp. \( m_p(i) = o \)) then the position \( i \) is called an input (resp. output) position of \( p \). The predicates in and \( \ddot{=} \) have only output positions. For a literal \( L = p(t_1,\ldots,t_n) \) (where \( p \) can be also in or \( \ddot{=} \)), we denote by \( \text{invar}(L) \) and \( \text{outvar}(L) \) the sets of variables occurring in terms in the input and output positions of \( p \).

If a predicate is used with different modes \( m^k_{p_1},\ldots,m^k_{p_n} \) in the program, we may consider each \( p_{m^k_p} \) 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 \( \text{invar}(\text{true}) := \emptyset, \text{outvar}(\text{true}) := \emptyset, \text{invar}(\text{false}) := \emptyset, \) and \( \text{outvar}(\text{false}) := \emptyset \).

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

1. For all \( 1 \leq i \leq n \), \( \text{invar}(E_i) \subseteq \bigcup_{j=1}^{i-1} \text{outvar}(E_j) \).
2. If for some \( 1 \leq i \leq n \), \( E_i \) is \( t_1 \ddot{=} t_2 \), then \( \text{var}(t_1) \subseteq \bigcup_{j=1}^{i-1} \text{outvar}(E_j) \) or \( \text{var}(t_2) \subseteq \bigcup_{j=1}^{i-1} \text{outvar}(E_j) \).
3. If for some \( 1 \leq i \leq n \), \( E_i \) is a membership atom, then the inclusion \( \text{var}(E_i) \subseteq \bigcup_{j=1}^{i-1} \text{outvar}(E_j) \) 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 \( \langle L_1,\ldots,L_n \parallel \mathcal{K}_1 \lor \cdots \lor \mathcal{K}_m \rangle \) is well-moded, where \( \mathcal{K} \)'s are conjunctions of true, false, and primitive constraints, if the formula \( \langle L_1 \land \cdots \land L_n \land \mathcal{K}_1 \rangle \lor \cdots \lor \langle L_1 \land \cdots \land L_n \land \mathcal{K}_m \rangle \) is well-moded.

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

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

A program is well-moded if all its clauses are well-moded.
Example 3
In Example 1, if in the user-defined binary predicates \textit{rewrite} and \textit{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:

- 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.

To prove these statements, some technical lemmas are needed.

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 \( K_1 \) and \( K_2 \) be two arbitrary (possibly empty) conjunctions of extended literals such that the conjunction \( K_1 \land K_2 \land v \doteq e \) is well-moded. Let \( \theta = \{ v \mapsto e \} \) be a substitution. Then \( K_1 \land K_2 \theta \land v \doteq e \) is also well-moded.

The next lemma states that reduction with respect to a well-moded program preserves well-modedness of states:

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

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

The following theorem shows that satisfiable well-moded constraints can be completely solved:

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

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

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

\[
C \rightsquigarrow (x \doteq \epsilon \land (a, \overline{y}) \doteq (a, b, a, c, c) \land f (x, a, x) \doteq f (\overline{y}, \overline{x}) \land \overline{y} \text{ in } c \text{(eps)}^*)
\]

\[
\land (x \doteq a \land (a, \overline{y}) \doteq (b, a, c, c) \land f (z, a, x) \doteq f (\overline{y}, \overline{x}) \land \overline{y} \text{ in } c \text{(eps)}^*)
\]

\[
\land (x \doteq (a, b) \land (a, \overline{y}) \doteq (a, c, c) \land f (x, a, x) \doteq f (\overline{y}, \overline{x}) \land \overline{y} \text{ in } c \text{(eps)}^*)
\]

\[
\ldots
\]

\[
\land (x \doteq (a, b, a, c) \land (a, \overline{y}) \doteq \epsilon \land f (z, a, x) \doteq f (\overline{y}, \overline{x}) \land \overline{y} \text{ in } c \text{(eps)}^*)
\]

\[
\rightsquigarrow^+ (x \doteq \epsilon \land \overline{y} \doteq (b, a, c, c) \land f (x, a, x) \doteq f (\overline{y}, \overline{x}) \land \overline{y} \text{ in } c \text{(eps)}^*)
\]
\( \forall (x, y) \models (a, b) \land \models (c, c) \land f(x, a, x) \models f(y, x) \land \models c(\text{eps})^+ \)

\( \models (x, y) \models (b, a, c, c) \land f(x, a, x) \models f(b, a, c, c) \land \models (b, a, c, c) \models c(\text{eps})^+ \)

\( \models (x, y) \models (c, c) \land f(x, a, x) \models f(b, a, c, c) \land \models (b, a, c, c) \models c(\text{eps})^+ \)

\( \models (x, y) \models (c, c) \land f(x, a, x) \models f(c, c, a, b) \land \models (c, c) \models c(\text{eps})^+ \)

\( \models (x, y) \models (c, c) \land f(x, a, x) \models f(c, c, a, b) \land \models (c, c) \models c(\text{eps})^+ \)

\( \models (x, y) \models (c, c) \land f(x, a, x) \models f(c, c, a, b) \land \models (c, c) \models c(\text{eps})^+ \)

\( \models (x, y) \models (c, c) \land f(x, a, x) \models f(c, c, a, b) \land \models (c, c) \models c(\text{eps})^+ \)

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 \( (G \parallel C) \rightarrow \cdots \rightarrow (\square \parallel C) \) be a finished derivation with respect to a well-moded CLP(H) program, starting from a well-moded goal \( G \). If \( C \neq \text{false} \), then \( C \) is solved.

### 8 Programs in the KIF Form

Knowledge Interchange Format, shortly 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:

\[
t_\kappa := x \mid f_\kappa(H_\kappa) \mid f_{\kappa_1}(t_{\kappa_1}, \ldots, t_{\kappa_n}) \mid X(t_{\kappa_1}, \ldots, t_{\kappa_n}) \quad (n \geq 0) \quad \text{KIF Term}
\]

\[
H_\kappa := t_{\kappa_1}, \ldots, t_{\kappa_n} \mid t_{\kappa_1}, \ldots, t_{\kappa_n}, \kappa \quad (n \geq 0) \quad \text{KIF Hedge}
\]

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 \( f_\kappa(x, f_\kappa(a, x), f_\kappa(x, b, \kappa)) \) and \( f_\kappa(a, x, b) \) are in the KIF form, while \( f_\kappa(x, a, \kappa) \) and \( f_\kappa(x, a, \kappa), f_\kappa(x, b, \kappa) \) 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 \( \kappa \) 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.

**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. 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 \texttt{solve} will be a KIF constraint.

It leads us to the following result:

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

We illustrate now how to solve a simple KIF constraint:

Example 5
Let \(C = f(x, \overline{x}) \equiv f(\overline{y}, a, \overline{y}) \land \overline{x} \in a(\text{eps})^* \land \overline{y} \in a(\text{eps}) \cdot a(b(\text{eps})^*)^*\). Then solve performs the following derivation:

\[
C \rightsquigarrow x \equiv g(\overline{y}) \land \overline{x} \equiv (a, \overline{y}) \land \overline{x} \in a(\text{eps})^* \land \overline{y} \in a(\text{eps}) \cdot a(b(\text{eps})^*)^*
\]

\[
\rightsquigarrow x \equiv g(\overline{y}) \land \overline{x} \equiv (a, \overline{y}) \land (a, \overline{y}) \in a(\text{eps})^* \land \overline{y} \in a(\text{eps}) \cdot a(b(\text{eps})^*)^*
\]

\[
\rightsquigarrow x \equiv g(\overline{y}) \land \overline{x} \equiv (a, \overline{y}) \land \overline{y} \in a(\text{eps}) \cdot a(\text{eps})^*
\]

The obtained constraint is solved.

A state \(\langle L_1, \ldots, L_n \parallel K_1 \lor \cdots \lor K_m \rangle\) is in the KIF form (KIF state), if the formula \((L_1 \land \cdots \land L_n \land K_1) \lor \cdots \lor (L_1 \land \cdots \land L_n \land K_m)\) 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 \texttt{solve} computes KIF constraints. Therefore, we can establish the following theorem:

Theorem 7
Let \(\langle G \parallel \text{true} \rangle \rightsquigarrow \cdots \rightsquigarrow \langle \Box \parallel C' \rangle\) be a finished derivation with respect to a KIF program, starting from a KIF goal \(G\). If \(C' \neq \text{false}\), then \(C'\) 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:

\[
\text{append\_dl}(x_1\cdot x_2, x_2\cdot x_3, x_1\cdot x_3),
\]
where the hyphen is a function symbol and \(x_1, x_2, x_3\) are term variables. The KIF goal

\[
\text{append\_dl}(f_1(a, b, \overline{x})\cdot f_2(\overline{x}), f_2(c, d, e, \overline{y})\cdot f_3(\overline{y}), x\cdot f_3)
\]
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:

\[
\begin{align*}
& (\text{append\_dl}(f_1(a, b, \overline{x})\cdot f_2(\overline{x}), f_2(c, d, e, \overline{y})\cdot f_3(\overline{y}), x\cdot f_3) \parallel \text{true}) \\
\implies & (x_1\cdot x_2 \parallel f_1(a, b, \overline{x})\cdot f_2(\overline{x}), x_2\cdot x_3 \parallel f_2(c, d, e, \overline{y})\cdot f_3(\overline{y}), x_1\cdot x_3 \parallel x\cdot f_3 \parallel \text{true}) \\
\implies & (x_2\cdot x_3 \parallel f_2(c, d, e, \overline{y})\cdot f_3(\overline{y}), x_1\cdot x_3 \parallel x\cdot f_3 \parallel x_1 \parallel f_1(a, b, \overline{x}) \land x_2 \parallel f_2(\overline{x})) \\
\implies & (x_1\cdot x_3 \parallel x\cdot f_3 \parallel x_1 \parallel f_1(a, b, c, d, e, \overline{y}) \land x_2 \parallel f_2(c, d, e, \overline{y}) \land x_3 \parallel f_3(\overline{y}) \land \overline{x} \parallel (c, d, e, \overline{y})) \\
\implies & (\text{false}) \\
\implies & x_1 \parallel f_1(a, b, c, d, e) \land x_2 \parallel f_2(c, d, e) \land x_3 \parallel f_3 \land \overline{x} \parallel (c, d, e) \land \overline{y} \parallel \epsilon \land \\
x \parallel f_1(a, b, c, d, e)).
\end{align*}
\]

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 or 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.

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.

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

References

Antimirov, V. M. 1996. Partial derivatives of regular expressions and finite automaton
rewriting on java. In Term Rewriting and Applications, 18th International Conference,
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
Coelho, J. and Florido, M. 2004. CLP(Flex): constraint logic programming applied to
XML processing. In On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and
ODBASE, OTM Confederated International Conferences, Agia Napa, Cyprus, October 25-29,
2004, Proceedings, Part II, R. Meersman and Z. Tari, Eds. Lecture Notes in Computer Science,
Coelho, J. and Florido, M. 2006. VeriFLog: A constraint logic programming approach to
verification of website content. In Advanced Web and Network Technologies, and Applications,
APWeb 2006 International Workshops: XRA, IWSN, MEGA, and ICSE, Harbin, China,
Coelho, J. and Florido, M. 2007. XCentric: logic programming for XML processing. In 9th
ACM International Workshop on Web Information and Data Management (WIDM 2007),
Comon, H. 1998. Completion of rewrite systems with membership constraints. Part II: constraint
Dembinski, P. and Maluszynski, J. 1985. And-parallelism with intelligent backtracking for
annotated logic programs. In Proceedings of the 1985 Symposium on Logic Programming,
Dundua, B., Florido, M., Kutsia, T., and Marin, M. 2014. Constraint logic programming for
hedges: A semantic reconstruction. In Functional and Logic Programming - 12th International
Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014, Proceedings, M. Codish and


Appendix A Proofs

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

Proof
Since $C$ is solved, each disjunct $K$ in it has a form $v_1 = e_1 \land \ldots \land v_n = e_n \land v'_1$ in $R_1 \land \ldots \land v'_m$ in $R_m$ 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 $[R_j] \neq \emptyset$ for all $1 \leq j \leq m$. Note that while $v_i$'s do not occur anywhere else in $K$, it still might be the case that some $v'_j$, $1 \leq j \leq m$, occurs in some $e_k$, $1 \leq k \leq n$.

Let $e'_j$ be an element of $[R_j]$ for all $1 \leq j \leq m$. Assume that for each $1 \leq i \leq n$, the substitution $\sigma'_i$ is a grounding substitution for $e_i$ with the property that $v'_i \sigma'_i = e'_j$ 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 $K$. Therefore, $\mathfrak{I} \models \exists C$ holds. \qed

Theorem 2 (Termination of $\text{solve}$)
$\text{solve}$ terminates on any quantifier-free constraint.

Proof
We need to show that $\text{NF}(\text{step})$ terminates for any quantifier-free constraint in DNF. We define a complexity measure $\text{cm}(C)$ for such constraints, and show that $\text{cm}(C') < \text{cm}(C)$ holds whenever $C' = \text{step}(C)$.

For a hedge $H$ (resp., for a regular expression $R$), we denote by $\text{size}(H)$ (resp., by $\text{size}(R)$) its denotational length, e.g., $\text{size}(e) = 0$, $\text{size}(\text{eps}) = 1$, $\text{size}(f(f(a)), \pi) = 4$, and $\text{size}(f(f(a \cdot b'))) = 6$.

The complexity measure $\text{cm}(K)$ of a conjunction of primitive constraints $K$ 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 $K$.
- $M_1 := \{ \text{size}(H) \mid H \in R \in K, H \neq \text{eps} \}$.
- $N_2$ is the number of primitive constraints in the form $\pi$ in $R$ in $K$.
- $M_2 := \{ \text{size}(R) \mid H \in R \in K \}$.
- $M_3 := \{ \text{size}(t_1) + \text{size}(t_2) \mid t_1 \vdash t_2 \in K \}$.

The complexity measure $\text{cm}(C)$ of a constraint $C = K_1 \lor \cdots \lor K_n$ is defined as $\{ \text{cm}(K_1), \ldots, \text{cm}(K_n) \}$.

Measures are compared by the multiset extension of the lexicographic ordering on tuples. The components that are natural numbers ($N_1$ and $N_2$) are, of course, compared by the standard ordering on naturals. The multiset components $M_1$, $M_2$, and $M_3$ are compared by the multiset extension of the standard ordering on the naturals.

The strict part of the ordering on measures is obviously well-founded. The Log rules strictly reduces it. For the other rules, the table below shows which rule reduces which component of the measure. The symbols $>$ and $\geq$ indicate the strict and non-strict decrease, respectively. It implies the termination of the algorithm $\text{solve}$. 


If \( \text{step}(C) = D \), then \( \mathcal{I} \models (C \leftrightarrow \overline{\text{var}(C)}D) \) for all intended structures \( \mathcal{I} \).

**Proof**

By case distinction on the inference rules of the solver, selected by the strategy first in the application of \text{step}. We illustrate here two cases, when the selected rules are (E3) and (M2). For the other rules the lemma can be shown similarly.

In (E3), \( C \) has a disjunct \( K = (\pi, H) \vdash T \land K' \) with \( \pi \not\in \text{var}(T) \), and \( D \) is the result of replacing \( K \) in \( C \) with the disjunction \( C' = \bigvee_{T_{=}(T_1, T_2)} (\pi \vdash T_1 \land H \theta \vdash T_2 \land K' \theta) \) where \( \theta = \{ \pi \mapsto T_1 \} \). Therefore, it is sufficient to show that \( \mathcal{I} \models (\forall v \in \text{var}(T, H, T) . \text{if } v = \pi \text{, equality trivially holds. If } v \neq \pi \text{, we have } \pi \theta \sigma = T_1 \pi = \pi \sigma) \). Hence, \( \mathcal{I} \models (\forall v \in \text{var}(T, H, T)) . \text{if } v = \pi \text{, equality trivially holds. If } v \neq \pi \text{, we have } \pi \theta \sigma = T_1 \pi = \pi \sigma). \)

Now, let the selected rule be (M2). In this case \( C \) has a disjunct \( K = (t, H) \) in \( R \land K' \) with \( H \neq \epsilon \) and \( R \neq \text{eps} \). Then \( D \) is the result of replacing \( K \) in \( C \) with \( C' = \bigvee_{t_{=}(R_1, R_2)} (t \in f(R_1) \land H \text{ in } R_2 \land K'). \) Therefore, to show \( \mathcal{I} \models (C \leftrightarrow \overline{\text{var}(C)}D) \), it is enough to show that \( \mathcal{I} \models (\forall K \in \overline{\text{var}(C)}C') \). Since \( \text{var}(C') = \text{var}(K) \), this amounts to showing that for all ground substitutions \( \sigma \) of \( \text{var}(K) \) we have \( \mathcal{I} \models (\forall \pi \sigma, H \sigma \in T \land K \land K') \). Hence, \( \mathcal{I} \models (\forall \pi \sigma, H \sigma) = T \land K \land K' \). It finishes the proof for (E3).
Theorem 3
If \textbf{solve}(C) = D, then \[ J \models \forall (C \leftrightarrow \exists_{\text{var}}(C)D) \] for all intended structures \( J \), and \( D \) is either partially solved or the false constraint.

Proof
We assume without loss of generality that \( C \) is in DNF. \[ J \models \forall (C \leftrightarrow \exists_{\text{var}}(C)D) \] follows from Lemma 1 and the following property: If \( J \models \forall (C_1 \leftrightarrow \exists_{\text{var}}(C_1)C_2) \) and \( J \models \forall (C_2 \leftrightarrow \exists_{\text{var}}(C_2)C_3) \), then \( J \models \forall (C_1 \leftrightarrow \exists_{\text{var}}(C_1)C_3) \). The property itself relies on the fact that \( J \models \forall (\exists_{\text{var}}(C_1) \exists_{\text{var}}(C_2)C_3 \leftrightarrow \exists_{\text{var}}(C_1)C_3) \), which holds because all variables introduced by the rules of the solver in \( C_3 \) are fresh not only for \( C_2 \), but also for \( C_1 \).

As for the partially solved constraint, by the definition of \textbf{solve} and Theorem 2, \( D \) 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 \( D \). But this contradicts the fact that \( D \) is in a normal form. Hence, \( D \) is partially solved.

\( \square \)

Lemma 2
Let \( v \models e \) be an equation, where \( v \) is a variable and \( e \) is the corresponding expression such that \( v \) does not occur in \( e \). Let \( K_1 \) and \( K_2 \) be two arbitrary (possibly empty) conjunctions of extended literals such that the conjunction \( K_1 \land K_2 \land v \models e \) is well-moded.

Let \( \theta = \{ v \mapsto e \} \) be a substitution. Then \( K_1 \land K_2 \theta \land v \models e \) is also well-moded.

Proof
The point in this lemma is that it does not matter how \( K_1 \) and \( K_2 \) are chosen. We consider two cases. First, when \( v \models e \) is the leftmost literal containing \( v \) in a well-moded sequence corresponding to \( K_1 \land K_2 \land v \models e \) and, second, when this is not the case.

Case 1. Let \( \tilde{E}_1, v \models e, \tilde{E}_2 \) be a well-moded sequence corresponding to \( K_1 \land K_2 \land v \models e \), such that \( \tilde{E}_1 \) does not contain \( v \). Note that there is no assumption (apart from what guarantees well-modedness of \( K_1 \land K_2 \land v \models e \)) on the appearance of literals in \( \tilde{E}_1 \) and \( \tilde{E}_2 \). They may contain literals from \( K_1 \) only, from \( K_2 \) only, or from both \( K_1 \) and \( K_2 \).

Well-modedness of \( \tilde{E}_1, v \models e, \tilde{E}_2 \) requires the variables of \( e \) to appear in \( \tilde{E}_1 \). Consider the sequence \( \tilde{E}_1, v \models e, \tilde{E}_2[\theta] \), where the notation \( \tilde{E}[\theta] \) stands for such an instance of \( \tilde{E} \) in which \( \theta \) affects only literals from \( K_2 \). Then \( \tilde{E}_1, v \models e \) is well-moded and it can be safely extended by \( \tilde{E}_2[\theta] \) without violating well-modedness, because the variables in \( v \models e \) still precede (in the well-moded sequence) the literals from \( \tilde{E}_2[\theta] \), and the relative order of the other variables (in the well-moded sequence) does not change. Hence, \( \tilde{E}_1, v \models e, \tilde{E}_2[\theta] \) is a well-moded sequence that corresponds to \( K_1 \land K_2 \theta \land v \models e \).

Case 2. Let \( \tilde{E}_1, L, \tilde{E}_2, v \models e, \tilde{E}_3 \) be a well-moded sequence corresponding to \( K_1 \land K_2 \land v \models e \), where \( L \) is the leftmost literal that contains \( v \) in an output position. Again, we make no assumption on literal appearances in the subsequences of the sequence. Then \( \tilde{E}_1, L, v \models e, \tilde{E}_2, \tilde{E}_3 \) is also a well-moded sequence (corresponding to \( K_1 \land K_2 \land v \models e \)), because \( v \) still appears in an output position in \( L \) left to \( v \models e \), the variables in \( e \) still precede literals from \( \tilde{E}_3 \), and the relative order of the other variables does not change. For literals in \( \tilde{E}_2 \) that contain variables from \( e \) such a reordering does not matter.

Note that \( v \) does not appear in \( \tilde{E}_1 \); if it were there in some literal in an output position, then \( L \) would not be the leftmost such literal. If it were there in some literal \( L' \) in an
We distinguish two cases, depending whether $\theta$ affects $L$. Therefore, we consider the other rules in more detail. We denote the disjunct state, we have to show that state well-modedness is preserved by each rule of the solver.

Let $\hat{E}_1, L[\theta], v \doteq e, \hat{E}_2[\theta], \hat{E}_3[\theta]$ be a sequence of all literals taken from $K_1 \land K_2 \land v \doteq e$. We distinguish two cases, depending whether $\theta$ affects $L$ or not.

**$\theta$ affects $L$.** Then it replaces $v$ in $L$ with $e$, i.e., $L[\theta] = L\theta$. Then the variables of $e$ appear in output positions in $L\theta$ and, hence, placing $v \doteq e$ after $L\theta$ in the sequence would not destroy well-modedness. As for the $L\theta$ itself, we have two alternatives:

1. $L\theta$ is an equation, say $s \doteq t\theta$, obtained from $L = (s \doteq t)$ by replacing occurrences of $v$ in $t$ by $e$. In this case, by well-modedness of $\hat{E}_1, L, v \doteq e, \hat{E}_2, \hat{E}_3$, variables of $s$ appear in $\hat{E}_1$ and $s$ does not contain $v$. Then the same property is maintained in $\hat{E}_1, L\theta, v \doteq e, \hat{E}_2[\theta], \hat{E}_3[\theta]$, since $s$ remains in $L\theta$ and $\hat{E}_1$ does not change.

2. $L\theta$ is an atom. Then replacing $v$ by $e$ in an output position of $L$, which gives $L\theta$, does not affect well-modedness.

Hence, we got that $\hat{E}_1, L, v \doteq e$ is well-moded. Now we can safely extend this sequence with $\hat{E}_2[\theta], \hat{E}_3[\theta]$, because variables in new occurrences of $e$ in $\hat{E}_2[\theta], \hat{E}_3[\theta]$ are preceded by $v \doteq e$, and the relative order of the other variables does not change. Hence, the sequence $\hat{E}_1, L\theta, v \doteq e, \hat{E}_2[\theta], \hat{E}_3[\theta]$ is well-moded.

**$\theta$ does not affect $L$.** Then $L[\theta] = L$, the sequence $\hat{E}_1, L, v \doteq e$ is well-moded and it can be safely extended with $\hat{E}_2[\theta], \hat{E}_3[\theta]$, obtaining the well-moded sequence $\hat{E}_1, L, v \doteq e, \hat{E}_2[\theta], \hat{E}_3[\theta]$.

Hence, we showed also in Case 2 that there exists a well-moded sequence of literals, namely, $\hat{E}_1, L[\theta], v \doteq e, \hat{E}_2[\theta], \hat{E}_3[\theta]$, that corresponds to $K_1 \land K_2 \theta \land v \doteq e$. Hence, $K_1 \land K_2 \theta \land v \doteq e$ is well-moded.

**Lemma 3**

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

**Proof**

Let $G = L_1, \ldots, L_i, \ldots, L_m, C = K_1 \lor \cdots \lor K_m$, and $\langle G \parallel C \rangle$ be a well-moded state. We will use the notation $\hat{G}$ for the conjunction of all literals in $G$, i.e., $\hat{G} = L_1 \land \cdots \land L_i \land \cdots \land L_m$. Assume that $L_i$ is the selected literal in reduction that gives $\langle G' \parallel C' \rangle$ from $\langle G \parallel C \rangle$. We consider four possible cases, according to the definition of operational semantics:

**Case 1.** Let $L_i$ be a primitive constraint and $C' \neq \text{false}$. Let $D$ denote the DNF of $C \land L_i$.

In order to prove that $\langle G' \parallel C' \rangle$ is well-moded, by the definition of $\text{solve}$, it is sufficient to prove that $\langle G' \parallel \text{step}(D) \rangle$ is well-moded. Since, obviously, $\langle G' \parallel D \rangle$ is a well-moded state, we have to show that state well-modedness is preserved by each rule of the solver.

Since $C' \neq \text{false}$, the step is not performed by any of the failure rules of the solver. For the rules M1–M8, M11–M12, D1, and D2, it is pretty easy to verify that $\langle G' \parallel \text{step}(D) \rangle$ is well-moded. Therefore, we consider the other rules in more detail. We denote the disjunct of $D$ on which the rule is applied by $K_D$. The cases below are distinguished by the rules:
Del. Here the same variable is removed from both sides of the selected equation. Assume $s \doteq t, 2$ is a well-moded sequence corresponding to $G' \land K_D$, and $s \doteq t$ is the selected equation affected by one of the deletion rules. Well-modedness of $1, s \doteq t, 2$ requires that the variable deleted at this step from $s \doteq t$ should occur in an output position in some other literal in $1$. Let $s' \doteq t'$ be the equation obtained by the deletion step from $s \doteq t$. Then $1, s' \doteq t', 2$ is again well-moded, which implies that $G' \land \text{step}(K_D)$ is well-moded and, therefore, that $\langle G' \parallel \text{step}(D) \rangle$ is well-moded.

M9. Let $G' \land K_D$ be represented as $G' \land \pi$ in $f(R) \land K'$, where $\pi$ in $f(R)$ is the membership atom affected by the rule. Note that then $G' \land \pi \doteq x \land x$ in $f(R) \land K'$ is also well-moded. Applying Lemma 2, we get that $\hat{G}' \land \pi \doteq x \land x$ in $f(R) \land K' \theta$ is also well-moded. Applying Lemma 2, we get that $\hat{G}' \land \pi \doteq x \land x$ in $f(R) \land \theta \doteq f \land f'$ is well-moded, where $\theta = \{ \pi \rightarrow x \}$. Then we get well-modedness of $G' \land \text{step}(K_D)$, which implies well-modedness of $\langle G' \parallel \text{step}(D) \rangle$.

M10. Let $G' \land K_D$ be represented as $G' \land X(H)$ in $f(R) \land K'$, where $X(H)$ in $f(R)$ is the membership atom affected by the rule. Note that then $G' \land X(H)$ in $f(R) \land X \doteq f \land K'$ is also well-moded. Applying Lemma 2, we get that $\hat{G}' \land X(H) \theta \doteq f \land f' \land K' \theta$ is well-moded, where $\theta = \{ X \rightarrow f \}$. But it means that $\hat{G}' \land \text{step}(K_D)$ is well-moded, which implies that $\langle G' \parallel \text{step}(D) \rangle$ is well-moded.

E1, E2. For these rules, well-modedness of $G' \land \text{step}(K_D)$ is a direct consequence of Lemma 2.

E3. Let $G' \land K_D$ be represented as $\hat{G}' \land (\pi, H_1) \simeq H_2 \land K'$, where $(\pi, H_1) \simeq H_2$ is the equation affected by the rule and $\pi \notin \text{var}(H_2)$. Then $G' \land \pi \doteq H' \land H_1 \doteq H'' \land K'$ is also well-moded for some $H'$ and $H''$ with $(H', H'') = H_2$. Applying Lemma 2, we get that $G' \land \pi \doteq H' \land H_1 \theta \doteq H'' \land K' \theta$ is well-moded, where $\theta = \{ \pi \rightarrow H' \}$. Since $H'$ and $H''$ were arbitrary, it implies that $\hat{G}' \land \text{step}(K_D)$ and, therefore, $\langle G' \parallel \text{step}(D) \rangle$ is well-moded.

E4. Similar to the case of the rule E3.

Case 2. Let $L_i$ be a primitive constraint and $C' = \text{false}$, where $C' = \text{solve}(C \land L_i)$. Then by the operational semantics we have $G' = \Box$ and the theorem trivially holds, since the state $\langle G \parallel \text{false} \rangle$ is well-moded.

Case 3. Let $L_i$ be an atom $p(t_1, \ldots, t_k, \ldots, t_l)$. Assume that $Pr$ contains a clause of the form $p(r_1, \ldots, r_k, \ldots, r_l) \leftarrow B$, where $B$ denotes the body of the clause. Assume also that for the predicate $p$, the set $\{1, \ldots, k\}$ is the set of the input positions and $\{k + 1, \ldots, l\}$ is the set of the output ones. Then we have

$$G = L_1, \ldots, L_{i-1}, p(t_1, \ldots, t_k, \ldots, t_l), L_{i+1}, \ldots, L_n,$$

$$G' = L_1, \ldots, L_{i-1}, t_1 \doteq r_1, \ldots, t_k \doteq r_k, \ldots, t_l \doteq r_l, B, L_{i+1}, \ldots, L_n,$$

$$C' = C = K_1 \lor \cdots \lor K_m.$$

From well-modedness of the state $\langle G \parallel C \rangle$ we know that for all $1 \leq j \leq m$, the literals from $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$ and $K_j$ can be reordered in two sequences of literals $j$ and $j'$ in such a way that the sequence $\frac{1}{2}, p(t_1, \ldots, t_k, \ldots, t_l), \frac{1}{2}$ is well-moded. Then we have $\text{var}(t_1, \ldots, t_k) \subseteq \text{outvar}(j)$. Therefore, we obtain that the sequence

$$t_1 \doteq r_1, \ldots, t_k \doteq r_k,$$

is well-moded for all $1 \leq j \leq m$.

From well-modedness of $p(r_1, \ldots, r_k, \ldots, r_l) \leftarrow B$ we know that $\text{var}(r_k+1, \ldots, r_l) \subseteq$
outvar(B) ∪ var(r1, . . . , rk). By item 1 of the definition of program well-modedness, the literals of B can be put into a well-moded sequence, written, say, as B1, . . . , Bq, such that for each 1 ≤ u ≤ q and v ∈ invar(Bu) we have v ∈ outvar(Bu′) for some u′ < u, or v ∈ var(r1, . . . , rk). From then we can say that the sequence
\[ t_1 \doteq r_1, \ldots, t_k \doteq r_k, B_1, \ldots, B_q, t_{k+1} \doteq r_{k+1}, \ldots, t_l \doteq r_l \]  
(A2)
is well-moded.

From (A1) and (A2), by the definition of well-modedness, we can conclude that
\[ \frac{1}{j}, t_1 \doteq r_1, \ldots, t_k \doteq r_k, B_1, \ldots, B_q, t_{k+1} \doteq r_{k+1}, \ldots, t_l \doteq r_l \]  
(A3)
is well-moded for all 1 ≤ j ≤ m. By construction, the literals in (A3) are exactly those from G′ ∨ K_j for 1 ≤ j ≤ m. It means that (G′ ∨ K_j) is well-moded for all 1 ≤ j ≤ m, which implies that (G′ ∨ C′) is well-moded.

Case 4. If defn_p(L_i) = {}, then G′ = \{\}, C′ = false, and the theorem trivially holds. □

**Corollary 1**

If C is a well-moded constraint, then solve(C) is also well-moded.

**Proof**

By the definition of well-modedness, since C is well-moded, the state (a ⊧ a ∨ C) is also well-moded, where a is an arbitrary function symbol. By the operational semantics, we have the reduction (a ⊧ a ∨ C) → (□ || solve(a ⊧ a ∧ C)). By Lemma 3, we get that (□ || solve(a ⊧ a ∧ C)) is also well-moded and, hence, solve(a ⊧ a ∧ C) is well-moded. By the definition of solve and the rules of the solver, it is straightforward to see that solve(a ⊧ a ∧ C) = solve(C). Hence, solve(C) is well-moded. □

**Theorem 4**

Let C be a well-moded constraint and solve(C) = C′, where C′ ̸= false. Then C′ is solved.

**Proof**

By the Corollary 1, the constraint C′ is well-moded. If C′ is true then it is already solved. Consider the case when C′ is not false. Let C′ = K_1 ∨ · · · ∨ K_m. Since C′ ̸= false, by the Theorem 3 C′ is partially solved. It means that each K_j, 1 ≤ j ≤ m, is partially solved and well-moded. By definition, K_j is well-moded if there exists a permutation of its literals c_1, . . . , c_n which satisfies the well-modedness property. Assume c_1, . . . , c_{i−1} are solved. By this assumption and the definition of well-modedness, each of c_1, . . . , c_{i−1} is an equation whose one side is a variable that occurs neither in its other side nor in any other primitive constraint. Then well-modedness of K_j guarantees that the other sides of these equations are ground terms. Assume by contradiction that c_i is partially solved, but not solved. If c_i is a membership constraint, well-modedness of K_j implies that c_i does not contain variables and, therefore, can not be partially solved. Now let c_i be an equation. Since all variables in c_1, . . . , c_{i−1} are solved, they can not appear in c_i. From this fact and well-modedness of K_j, c_i should have at least one ground side. But then it can not be partially solved. The obtained contradiction shows that C′ is solved. □
Theorem 5
Let \( \langle G \parallel \text{true} \rangle \leadsto \cdots \leadsto \langle \square \parallel C \rangle \) be a finished derivation with respect to a well-moded CLP(H) program, starting from a well-moded goal \( G \). If \( C \neq \text{false} \), then \( C \) is solved.

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

To prove this statement, we use induction on the length \( n \) of the derivation. When \( n = 0 \), then \( C' = \text{true} \) 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 \( \langle G \parallel \text{true} \rangle \leadsto \cdots \leadsto \langle G_n \parallel C_n \rangle \leadsto \langle G_{n+1} \parallel C_{n+1} \rangle \). Assume that \( G_{n+1} \) that is either \( \square \) or consists only of propositional constants. According to the operational semantics, there are two possibilities how the last step is made:

1. \( G_n \) has a form (modulo permutation) \( L, p_1, \ldots, p_m, m \geq 0 \), where \( L \) is primitive constraint, the \( p_i \)'s are propositional constants, \( G_{n+1} = p_1, \ldots, p_m \), and \( C_{n+1} = \text{solve}(C_n \land L) \).
2. \( G_n \) has a form (modulo permutation) \( q, p_1, \ldots, p_m, m \geq 0 \), where \( q \) and \( p_i \)'s are propositional constants, the program contains a clause \( q \leftarrow q_1, \ldots, q_k, k \geq 0 \), where all \( q_i \), \( 1 \leq i \leq k \), are propositional constants, \( G_{n+1} = q_1, \ldots, q_k, p_1, \ldots, p_m \), and \( C_{n+1} = C_n \).

In the first case, by the \( n \)-fold application of Lemma 3 we get that \( \langle G_n \parallel C_n \rangle \) is well-moded. Since the \( p_i \)'s have no influence on well-modedness (they are just propositional constants), \( C_n \land L \) is well-moded and hence it is solvable. By Theorem 4 we get that if \( C_{n+1} = \text{solve}(C_n \land L) \neq \text{false} \), then \( C_{n+1} \) is solved.

In the second case, since \( G_n \) consists of propositional constants only, by the induction hypothesis we have that if \( C_n \) is not \( \text{false} \), then it is solved. But \( C_n = C_{n+1} \). It finishes the proof. \( \square \)

Lemma 4
Any partially solved KIF constraint is solved.

Proof
Let \( K \) be a partially solved conjunction of primitive constraints. Then, by the definition, each primitive constraint \( c \) from \( K \) should be either solved in \( K \), or should have one of the following forms:

- Membership atom:
  - \( f_u(H_1, \overline{x}, H_2) \) in \( f_u(R) \).
  - \( \langle \overline{x}, H \rangle \) in \( R \) where \( H \neq \epsilon \) and \( R \) has the form \( R_1 \cdot R_2 \) or \( R_1^* \).

- Equation:
  - \( \langle \overline{x}, H_1 \rangle \doteq (\overline{y}, H_2) \) where \( \overline{x} \neq \overline{y}, H_1 \neq \epsilon \) and \( H_2 \neq \epsilon \).
  - \( \langle \overline{x}, H_1 \rangle \doteq (T, \overline{y}, H_2) \), where \( \overline{x} \notin \text{var}(T) \), \( H_1 \neq \epsilon \), and \( T \neq \epsilon \). The variables \( \overline{x} \) and \( \overline{y} \) are not necessarily distinct.
  - \( f_u(H_1, \overline{x}, H_2) \doteq f_u(H_3, \overline{y}, H_4) \) where \( (H_1, \overline{x}, H_2) \) and \( (H_3, \overline{y}, H_4) \) are disjoint.
However, \(c\) is also a KIF constraint. By the definition of KIF form, none of the above mentioned forms for membership atoms and equations are permitted. Hence, \(c\) is solved in \(K\) and, therefore, \(K\) is solved. It implies the lemma.

**Theorem 6**

Let \(C\) be a KIF constraint and \(\text{solve}(C) = C'\), where \(C' \neq \text{false}\). Then \(C'\) is solved.

**Proof**

By Theorem 3, \(C'\) should be in a partially solved form. It is also in the KIF form, as we noted above. Then, by Lemma 4, \(C'\) is solved.

**Theorem 7**

Let \(\langle G \parallel \text{true} \rangle \rightarrow \cdots \rightarrow \langle \Box \parallel C' \rangle\) be a finished derivation with respect to a KIF program, starting from a KIF goal \(G\). If \(C' \neq \text{false}\), then \(C'\) is solved.

**Proof**

Since the reduction preserves KIF states, \(C'\) is in the KIF form. Since the derivation is finished and \(C' \neq \text{false}\), by the definition of finished derivation, \(C'\) is partially solved. By Lemma 4, we conclude that \(C'\) is solved.