Solving, Reasoning, and Programming in Common Logic

Temur Kutsia[1] and Mircea Marin[2]

[1] RISC, Johannes Kepler University Linz, Austria
[2] West University of Timisoara, Romania

Abstract. Common Logic (CL) is a recent ISO standard for exchanging logic-based information between disparate computer systems. Sharing and reasoning upon knowledge represented in CL require equation solving over terms of this language. We study computationally well-behaved fragments of such solving problems and show how they can influence reasoning in CL and transformations of CL expressions.

1 Introduction

Common Logic (CL) is a framework for a family of logic-based languages. Its syntax and semantics has been published as an ISO/IEC International Standard [1]. The purpose of CL is to facilitate knowledge exchange in a heterogeneous network, between disparate computer systems.

Open networks, like the Web, pose new challenges to knowledge representation languages: How can formalized information be represented on such open networks so that it can be identified by interested agents, reasoned upon, and integrated with related information? In [2], CL is presented from this perspective, as a result of representational adaptations of traditional first-order logic motivated by developments in knowledge representation and the (Semantic) Web.

These adaptations lead to a pretty liberal framework. In CL, symbols (called names) do not have fixed arity (variable polyadicity), the language contains sequence markers that stand for finite term sequences, one and the same symbol can play the syntactic roles of both function symbol and predicate (cross-categoricity), terms can be applied to other terms, and the language is type-free. Despite the fact that the syntax looks higher-order, CL admits first-order model theory. The framework permits a variety of different syntactic forms, called dialects. The dialects are expressible in the CL syntax and share the single semantics.

One of the advantages of logic-based knowledge representation frameworks is that they enable the use of automated reasoning techniques to draw useful conclusions from the information. However, there are not many reasoning tools for CL yet. Probably, [3] is the only large-scale attempt to provide a reasoning support (for KIF [4], a predecessor of CL).

CL is not compact, which is, in general, a serious problem for automated reasoning, but the clausal fragment of CL does not suffer from it [2]. One of

the reasons why proving support for CL is not well-developed is the difficulty of unification. Even a simple unification problem that involves sequence markers may have infinitely many unifiers.

In this paper we approach this problem by giving precise characterizations to several terminating fragments of CL unification. Such fragments are important for developing proving and programming methods in languages that conform to the CL Standard. In fact, resolution and factoring rules can be used for proving in clausal fragments of CL with finitary unification. The unification fragment restrictions can be weakened for Horn theories. Moreover, we show how well-moded Horn clauses can model conditional sequence transformation rules. These rules can be used, for instance, to formulate and execute queries in CL. The main computational mechanism for such rule-based programming is matching. We give complexity results for various CL matching fragments.

2 Preliminaries

We introduce Common Logic and its dialects following [2]. This is a less general definition that the one in [1], but it is sufficient for our purposes.

Let $\mathcal{N}$ be a countable set. Its elements are called *names*. They play the role of nonlogical symbols when building CL expressions. Besides $\mathcal{N}$, any Common Logic dialect includes a countable set $\mathcal{X}$ of *sequence variables* (sequence markers in the CL terminology), *connectives* ($\neg, \rightarrow, \leftrightarrow, \vee, \wedge$) and *quantifiers* ($\exists, \forall$). The sets $\mathcal{N}$ and $\mathcal{X}$ are assumed to be disjoint.

In any CL dialect (a language $\mathcal{L}_{\mathcal{N}}$ based on $\mathcal{N}$), *terms* t and sequences of terms and sequence variables (*term sequences*) $\tilde{s}$ are defined by the grammar below, where a is a name and X is a sequence variable:

$$t ::= a \mid t(\tilde{s}) \quad s ::= t \mid X \quad \tilde{s} ::= s_1, \ldots, s_n \quad n \geq 0.$$

The letters $a, b, c, f, g, x, y, z, u$ are used for names, t, l, r for terms, X, Y, Z, U for sequence variables, s, q for a term or a sequence variable, and $\tilde{s}$ and $\tilde{q}$ for term sequences. The terms $f()$ and f are not identified. Terms of the form $t(\tilde{s})$ are called *functional terms.*

A *sentence* is either an atom, a Boolean sentence, or a quantified sentence: $sen ::= atom \mid bool \mid quant$. Each of them is defined by the following productions:

$$\begin{aligned} atom &::= t(\tilde{s}) \mid \doteq(l, r). \\ bool &::= atom \mid \neg(sen) \\ &\quad \mid \rightarrow(sen_1, sen_2) \mid \leftrightarrow(sen_1, sen_2) \\ &\quad \mid \vee(sen_1, \ldots, sen_n) \mid \wedge(sen_1, \ldots, sen_n). \\ quant &::= \exists \tilde{x}.\, sen \mid \forall \tilde{x}.\, sen. \end{aligned}$$

In this definition $n \geq 0$ and $\tilde{x}$ is a finite, nonrepeating sequence of names and sequence variables. In this context, the names that occur in $\tilde{x}$ are called *individual variables.* To distinguish, we reserve the letters x, y, z, u for them. The atoms

$\doteq(l, r)$ are called equational atoms. We will use the standard infix notation for sentences. Note that CL sentences may contain unquantified sequence variables.

Example 1. $\exists x, X.\, x(a)(f(x, X)) \vee f(x, X)$ is a sentence.

A *substitution* is a mapping from names to terms and from sequence variables to term sequences, which is the identity almost everywhere. They are denoted by lower case Greek letters, reserving ε for the identity substitution. The *domain* of a substitution σ is the set $dom(\sigma) = \{\chi \mid \chi \neq \sigma(\chi)\}$. We write substitutions as sets, e.g., $\sigma = \{x \mapsto f(a, X)(f(), f), X \mapsto \langle a, f()(Y)\rangle\}$, with $\{x, X\} = dom(\sigma)$. The $\langle$ and $\rangle$ are used only for readability. The notions of substitution *composition* $\circ$, *extension* to term sequences, and the *restriction* of σ to a set of names or sequence variables S, denoted $\sigma|_S$, are defined in the standard way. σ is *more general* than ϑ on a set of names and sequence variables S, written $\sigma \leq_S \vartheta$, if there exists η such that $(\eta \circ \sigma)|_S = \vartheta|_S$.

2.1 Interpretations

An outstanding feature of Common Logic, when compared with First Order Logic, is complete cross-categoricity: the same name can have occurrences with different interpretations: as constants, as functions, or as relations. This feature is motivated by the anarchic character of knowledge available in open networks such as World Wide Web, where the same name can be used with different meanings in different contexts.

All features of CL are nicely captured at the semantic level by the notion of intensional interpretation, which is a tuple $\mathbf{I} = \langle D, den, fext, rext\rangle$ where

- $den : \mathcal{N} \to D$ provides intensional meaning to the names of the language,
- $fext : D \to \{f \mid f : D^* \to D\}$ yields functional extensions to the functional occurrences of names,
- $rext : D \to \{r \mid r \subseteq D^*\}$ yields relational extensions to the relational occurrences of names,

and D^* is the set of all finite sequences of elements from D, including the empty sequence. A corresponding variable assignment is a mapping $\sigma : N \cup \mathcal{X} \to D^*$ with $N \subseteq \mathcal{N}$ and $\sigma(a) \in D$ for all $a \in N$.

The denotation $d^\sigma_{\mathbf{I}}(expr)$ of an expression *expr* ranging over terms or term sequences of CL is defined w.r.t. an interpretation $\mathbf{I}$ and corresponding variable assignment σ as follows. Terms are interpreted as elements of D, and terms sequences are interpreted as elements of D^*:

$d^\sigma_{\mathbf{I}}(a) := den(a)$ if $a \in \mathcal{N} \setminus N$; $d^\sigma_{\mathbf{I}}(a) = \sigma(a)$ if $a \in N$;
$d^\sigma_{\mathbf{I}}(t(\tilde{s})) := fext(d^\sigma_{\mathbf{I}}(t))(d^\sigma_{\mathbf{I}}(\tilde{s}))$;
$d^\sigma_{\mathbf{I}}(X) := \sigma(X)$ if $X \in \mathcal{X}$;
$d^\sigma_{\mathbf{I}}(s_1, \ldots, s_n) := d^\sigma_{\mathbf{I}}(s_1), \ldots, d^\sigma_{\mathbf{I}}(s_n)$

We write $\models_{\mathbf{I},\sigma} atom$, and say that the atomic sentence *atom* is true in interpretation $\mathbf{I}$ and assignment σ, if

$atom \equiv t(\tilde{s})$ and $d^{\sigma}_{\mathbf{I}}(\tilde{s}) \in rext(d^{\sigma}_{\mathbf{I}}(t))$, or
$atom \equiv s \doteq t$ and $d^{\sigma}_{\mathbf{I}}(s) = d^{\sigma}_{\mathbf{I}}(t)$.

The relation $\models_{\mathbf{I},\sigma} sen$ for an arbitrary sentence sen is defined as expected, in the standard way. Since sentences can contain unquantified sequence variable, we always interpret them w.r.t. a variable assignment.

The notions of Herbrand universe and interpretation from First Order Logic can be defined for Common Logic as well. In our setting, the Herbrand universe consists is the set of all variable-free terms, and the Herbrand base consists of all variable-free atoms which are not equalities. A Herbrand interpretation in CL is any interpretation $\mathbf{H} = \langle H, den, fext, rext \rangle$ with H the set of variable-free terms, den is the identity function on $\mathcal{N}$, and $fext : H \to (H^* \to H)$, $fext(t)(\tilde{s}) := t(\tilde{s})$ for all $\tilde{s} \in H^*$. Thus, like in First Order Logic, the Herbrand interpretations of CL differ only by the assignment of relational extensions to the relational occurrences of names.

Note that the interpretation of equational atoms is the same in all Herbrand interpretations of CL.

2.2 Unification in Common Logic

A *Common Logic unification problem* (CLU problem) is a sentence $\exists \tilde{x}.\, l_1 \doteq r_1 \wedge \cdots \wedge l_n \doteq r_n$. It is assumed that $\tilde{x}$ contains all sequence variables that occur in l's or r's. A *unifier* for it is a substitution σ such that (i) $dom(\sigma)$ does not contain names and sequence variables that occur in the problem but not in $\tilde{x}$, and (ii) $\sigma(l_i) = \sigma(r_i)$ for each $1 \leq i \leq n$. Below we write CLU problems in the form of a set of (unordered) equations $\{l_1 \stackrel{?}{=} r_1, \ldots, l_n \stackrel{?}{=} r_n\}$. A problem is *solvable* if it has a unifier. A *complete set of unifiers* of a CLU problem Γ is a set of substitutions Σ such that (i) every $\sigma \in \Sigma$ is a unifier of Γ, (ii) For any unifier ϑ of Γ there exists $\sigma \in \Sigma$ such that $\sigma \leq_{var(\Gamma)} \vartheta$, where $var(\Gamma)$ is the set of individual and sequence variables in Γ. This set is *minimal,* if for all $\sigma_1, \sigma_2 \in \Sigma$, $\sigma_1 \leq_{var(\Gamma)} \sigma_2$ implies $\sigma_1 = \sigma_2$. A CLU problem Γ is unitary (finitary, infinitary) if it has a minimal complete set of unifiers (denoted by $mcsu(\Gamma)$) of cardinality 1 (finite cardinality, infinite cardinality). For any solvable CLU problem there exists a *minimal* complete set of unifiers.

A *fragment* of CLU is a set of CLU problems defined by some syntactic restriction. A fragment is *unitary* if all solvable problems in it are. It is *finitary* if it contains at least one finitary problem and no infinitary problems. It is *infinitary* if it contains at least one infinitary problem. The CLU itself is infinitary: $\{f(a, X) \stackrel{?}{=} f(X, a)\}$ has an infinite $mcsu$: $\{\{X \mapsto \langle\rangle\}, \{X \mapsto a\}, \{X \mapsto \langle a, a\rangle\}, \ldots\}$. The challenge is to find "computationally interesting" finitary or unitary fragments.

3 Solving

A *state* is a pair $\langle \Gamma, \vartheta \rangle$ of a CLU problem and a substitution. The CLU procedure is described by a set of state transformations of the form $\langle \Gamma \cup \{l \stackrel{?}{=} r\}, \vartheta \rangle \rightsquigarrow$

$\langle \sigma(\Gamma \cup \Gamma'), \sigma \circ \vartheta \rangle$. Each of the transformations is characterized by a rule of the form $l \stackrel{?}{=} r \rightsquigarrow \Gamma'$ and a substitution σ. The rules are:

Simplification rules:

S1. $t \stackrel{?}{=} t \rightsquigarrow \emptyset$.

S2. $t(l, \tilde{s}) \stackrel{?}{=} t(r, \tilde{q}) \rightsquigarrow \{l \stackrel{?}{=} r, t(\tilde{s}) \stackrel{?}{=} t(\tilde{q})\}$,
where $(l, \tilde{s}) \neq (r, \tilde{q})$.

S3. $t(X, \tilde{s}) \stackrel{?}{=} t(X, \tilde{q}) \rightsquigarrow \{t(\tilde{s}) \stackrel{?}{=} t(\tilde{q})\}$,
where $\tilde{s} \neq \tilde{q}$.

S4. $t'(\tilde{s}) \stackrel{?}{=} t''(\tilde{q}) \rightsquigarrow \{t' \stackrel{?}{=} t'', f(\tilde{s}) \stackrel{?}{=} f(\tilde{q})\}$,
where $t' \neq t''$.

In all cases $\sigma = \varepsilon$. Both t' and t'' are functional terms. The name f is chosen arbitrarily.

Projection rule:

P. $t(X, \tilde{s}) \stackrel{?}{=} t(\tilde{q}) \rightsquigarrow \{t(\tilde{s}) \stackrel{?}{=} t(\tilde{q})\}$,
where $t(X, \tilde{s}) \neq t(\tilde{q})$.

In this case $\sigma = \{X \mapsto \langle\rangle\}$.

Variable Elimination rules:

VE1. $x \stackrel{?}{=} t \rightsquigarrow \emptyset$.

VE2. $x(\tilde{s}) \stackrel{?}{=} t(\tilde{q}) \rightsquigarrow \{f(\tilde{s}) \stackrel{?}{=} f(\tilde{q})\}$.

In both cases $\sigma = \{x \mapsto t\}$, where x does not occur in t. The name f is chosen arbitrarily.

VE3. $t(X, \tilde{s}) \stackrel{?}{=} t(q, \tilde{q}) \rightsquigarrow \{t(X', \tilde{s}) \stackrel{?}{=} t(\tilde{q})\}$,

where $\sigma = \{X \mapsto \langle q, X' \rangle\}$, X does not occur in q, and X' is fresh.

To solve a problem Γ, we start with $\langle \Gamma, \varepsilon \rangle$ and apply these transformations nondeterministically on the selected equation. At each step, only one equation is selected. If we get $\langle \emptyset, \vartheta \rangle$, then we say that the procedure computes ϑ. The set of all computed substitutions for a problem Γ is denoted by $comp(\Gamma)$. This set can be infinite. It can be enumerated by a fair application of the procedure rules (e.g., during a breadth-first construction of the derivation tree for $\langle \Gamma, \varepsilon \rangle$).

Example 2. Let $\Gamma = \{x(Y)(X, f) \stackrel{?}{=} y(f(x), Z)(f, X)\}$ be a unification problem. Then

$$comp(\Gamma) = \{\{x \mapsto y, Y \mapsto f(y), Z \mapsto \langle\rangle, X \mapsto \langle\rangle\},$$
$$\{x \mapsto y, Y \mapsto f(y, Z), X \mapsto \langle\rangle\},$$

$$\{x \mapsto y, Y \mapsto f(y), Z \mapsto \langle\rangle, X \mapsto f\},$$
$$\{x \mapsto y, Y \mapsto f(y, Z), X \mapsto f\},$$
$$\{x \mapsto y, Y \mapsto f(y), Z \mapsto \langle\rangle, X \mapsto \langle f, f\rangle\},$$
$$\{x \mapsto y, Y \mapsto f(y, Z), X \mapsto \langle f, f\rangle\}, \ldots\}.$$

Example 3. $comp(\{x() \stackrel{?}{=} x(Y)\}) = \{\{Y \mapsto \langle\rangle\}\}$. $comp(\{f(x) \stackrel{?}{=} g(a)\}) = comp(\{x(a) \stackrel{?}{=} f(x)(a)\}) = comp(\{x() \stackrel{?}{=} x\}) = \emptyset$.

The CLU rules given above are adapted from the sequence unification (SU) rules [5]. SU has more features, such as sequence functions, or fixed arity symbols together with the variadic ones. But it does not allow a term to be applied to other terms. For instance, the CL term $f(a)(X, g(x, b()))$ is not a valid expression there. In principle, we could encode it into the syntax accepted by the SU, introducing a new variadic application symbol @ and writing $@(@(f, a), X, @(g, x, @(b)))$. We chose not to do it, because such an encoding destroys the syntactic form of the fragments we will be dealing with and makes characterization of terminating cases more cumbersome. Therefore, we write rules directly into the CL syntax. On the other hand, it is easy to see that the encoding preserves solvability: a CLU problem is solvable iff its encoded SU version is solvable. Therefore, from the decidability, soundness and completeness of sequence unification [5–7] we can get the same results for the CLU procedure:

Theorem 1. *CLU is decidable, infinitary, and admits a complete unification procedure (described above).*

On the other hand, as one can see from Example 2, the procedure as we described it is not minimal: It may compute two unifiers such that one is more general than the other (maybe even the same unifiers several times). It is possible to get an (almost) minimal procedure if we require $\stackrel{?}{=}$ to be ordered, add orientation rules, and put additional constraints on the application of variable elimination rules, similar to the ones in [5]. The reason why we have not chosen that approach is that we wanted to keep the procedure simple, with the minimal number of rules. It is sufficient for our purposes.

One can notice that the procedure as it is described above does not necessarily terminate for a Γ even if $comp(\Gamma)$ is finite. For instance, $\Gamma = \{f(X, a) \stackrel{?}{=} f(a, X, a)\}$ is such an instance, where $comp(\Gamma) = \emptyset$, but the procedure does not stop. A way out of this situation is to tailor the decision algorithm into the CLU procedure and apply the transformation rules only to solvable problems. In this way, the procedure terminates for Γ iff $comp(\Gamma)$ (and, hence, $mcsu(\Gamma)$) is finite.

Decidability of CLU (via decidability of SU) is based on decidability of word unification with regular constraints (WURC) and, hence, is at least as difficult as WURC that is PSPACE-complete [8]. The fact that CLU is infinitary restricts its usability further. A way out is to identify better-behaved fragments of CLU.[3]

[3] Another way is to impose restrictions on the unifiers, obtaining well-behaved *variants*. An example of such a variant is the restriction of instantiation lengths for sequence variables [3]. We do not consider variants in this paper.

We consider only fragments with sequence variables. Otherwise, the problem becomes unitary.

3.1 Terminating Fragments of CLU

The CLU procedure terminates if it can recognize unsolvable problems and the set of unifiers it computes is finite. (We do not consider infinite but finitely representable sets of unifiers here.) Because of the high complexity of the decision procedure, we consider here finitary fragments whose solvability can be decided with more light-weight methods. In all of them, the decidability test in the CLU procedure is replaced by implicit failure that arises when no rule can be applied to the problem. A fragment is *terminating* if the CLU procedure (or its relevant modification) terminates on it.

Linear Fragment (L-CLU) A term (atom, unification problem) is called *linear*, if no variable occurs in it more than once.

Example 4. The unification problem $\{f(X, f(Y, Z)) \stackrel{?}{=} f(a, b, f(x, c))\}$ is linear, while $\{f(f(a, X), f(X, a)) \stackrel{?}{=} f(x, x)\}$ is not.

Inspecting the CLU rules, it is easy to see that no rule duplicates occurrences of a variable in linear problems. It indicates that the CLU procedure preserves linearity. As a side effect, there is no need to perform occurrence checks.

Theorem 2. *The L-CLU fragment is terminating.*

Proof. We define the size of a term or a sequence variable:

- $size(t) = 1$ if t is a name or a sequence variable.
- $size(t(s_1, \ldots, s_n)) = 1 + \sum_{i=1}^{n} size(s_i) + size(t)$.

Then, the complexity measure of a CLU problem Γ is defined as a tuple $\langle n, M \rangle$, where n is the number of variables in Γ and M is the multiset of sizes of equations in Γ. The measures are ordered lexicographically, where the first components are compared by the standard ordering on the naturals and the second components are compared by the multiset extension of the standard ordering. Each rule of the CLU procedure strictly decreases the size of unification problems: For the rule VE3 this observation is based on the fact that $size(t') > size(f)$ and $size(t'') > size(f)$ (since t' and t'' are functional terms). For the other rules it is straightforward to check this property. Since the measure ordering is well-founded, we obtain termination of the procedure.

L-CLU is finitary. For the problem $\{f(X, f(Y, Z)) \stackrel{?}{=} f(a, b, f(x, c))\}$ above, *mcsu* consists of three elements: $\{X \mapsto (a, b), Y \mapsto (), Z \mapsto (x, c)\}$, $\{X \mapsto (a, b), Y \mapsto x, Z \mapsto c\}$, and $\{X \mapsto (a, b), Y \mapsto (x, c), Z \mapsto ()\}$.

KIF Fragment (KIF-CLU) The restriction on sequence variables in this fragment originates from KIF [4]. Sequence variables occupy only the last argument position in each subterm where they occur. Linearity is not required. With a slight modification of the rules, we avoid branching in unification derivations, obtaining a single most general unifier (mgu) for solvable problems. The modification affects the Projection and VE3 rules:

- The Projection rule is applied only when $\tilde{q}$ is the empty sequence.
- In the VE3 rule, substitute X with the entire $\langle q, \tilde{q}\rangle$.

The procedure terminates, because the rules strictly reduce the complexity measure defined in the proof of Theorem 2. This modification basically corresponds to the algorithm in [9].

Example 5. The CLU problem $\{f(a, X)(g(x, y, X), Y) \stackrel{?}{=} f(x, a, Y)(g(a, Z), U)\}$ is in the KIF-fragment, with an mgu $\{x \mapsto a, X \mapsto \langle a, U\rangle, Z \mapsto \langle y, a, U\rangle, Y \mapsto U\}$.

Inverse KIF (I-KIF-CLU) Here sequence variables occupy only the first argument position in each subterm where they occur. Changing the order in the Simplification and Variable Elimination rules for KIF, starting it from the last argument, we obtain a unification algorithm for I-KIF-CLU.

Unique Postfix Fragment (U-Post-CLU) Given a term $r = t(\tilde{s_1}, X, \tilde{s_2})$, we call the sequence $\tilde{s_2}$ *a postfix of* X *in* r. Note that the same variable may have several postfixes in the same term. For instance, the postfixes of X in $f(a, X, b, X, g(X))$ are $\langle b, X, g(X)\rangle$ and $\langle g(X)\rangle$. A CLU problem Γ is called *unique-postfix,* if each sequence variable X occurring in Γ has the same postfix in all subterms it occurs. Therefore, we can speak of *the postfix* of X in Γ.

Example 6. An example of a U-Post-CLU problem is $\{f(a, X, f(Y, b), y)(Z) \stackrel{?}{=} x(x, g(X, f(Y, b), y)),\ f(U, a) \stackrel{?}{=} f(f(Y, b), X, f(Y, b), y)\}$. On the other hand, terms like $f(a, X, b, X, g(X))$ can not occur in U-Post-CLU problems. There can be no cyclic dependence between sequence variable occurrences in postfixes.

The unification procedure for the U-Post-CLU fragment is obtained from the CLU procedure by replacing the rule VE3 with two new ones:

VE3': $t(X, \tilde{s}) \stackrel{?}{=} t(\tilde{q}, X, \tilde{s}) \rightsquigarrow \{t() \stackrel{?}{=} t(\tilde{q})\}$,
where $\sigma = \varepsilon$.

VE3": $t(X, \tilde{s}) \stackrel{?}{=} t(q, \tilde{q}) \rightsquigarrow \{t(X', \tilde{s}) \stackrel{?}{=} t(\tilde{q})\}$,
where $\sigma = \{X \mapsto \langle q, X'\rangle\}$, provided that X does not occur in q, $\tilde{q}$ does not contain X as an element, and X' is fresh.

The condition "$\tilde{q}$ does not contain X as an element" means that $\tilde{q}$ does not have a form $\ldots, X, \ldots$. VE3' and VE3" are exhaustive and mutually exclusive for the equations of the form $f(X, \tilde{s}) \stackrel{?}{=} f(\tilde{q})$. They do not violate soundness of CLU.

The U-Post-CLU procedure does not preserve the unique postfix property: VE3" transforms $\{f(X,\tilde{s}) \stackrel{?}{=} f(Y,\tilde{q})\} \cup \Gamma'$ into $\sigma(\{f(X',\tilde{s}) \stackrel{?}{=} f(\tilde{q})\} \cup \Gamma')$ with $\sigma = \{X \mapsto \langle Y, X'\rangle\}$. Hence, occurrences of Y in Γ' now appear in $\sigma(\Gamma')$ with the postfix $\sigma(\tilde{q})$. Besides, there may be new occurrences of Y in $\sigma(\Gamma')$ (and, maybe, also deeper in $\sigma(\tilde{q})$) with the postfix $\langle X', \tilde{s}\rangle$. However, if we manage to solve $\sigma(\{f(X',\tilde{s}) \stackrel{?}{=} f(\tilde{q})\})$ say, with ϑ, then $\vartheta(\sigma(\Gamma'))$ will be again postfix-unique, because $\vartheta(\sigma(\tilde{q})) = \vartheta(\sigma(\langle X', \tilde{s}\rangle)) = \vartheta(\langle X', \tilde{s}\rangle)$. This observation suggests the idea of imposing the following control: After applying VE3", split the obtained CLU problem into two: $\Delta_1 = \sigma(\{f(X',\tilde{s}) \stackrel{?}{=} f(\tilde{q})\})$ and $\Delta_2 = \sigma(\Gamma')$, where Δ_1 is again postfix-unique. We solve Δ_1 and apply its solution ϑ to Δ_2. Then $\vartheta(\Delta_2)$ is U-Post-CLU and we try to solve it.

We show that such a control leads to termination. In fact, we should show that the sequence of transformations of Δ_1 terminates and the measure of $\vartheta(\Delta_2)$ (defined in the sense of Theorem 2) is strictly smaller than the measure of Γ. If Δ_1 is transformed with the VE3" rule, then the result is split again, and so on. This process can not continue infinitely, because VE3" strictly reduces the number of arguments of $t(q,\tilde{q})$ (since $\tilde{q}$ does not contain X as an element). An alternative to VE3" is the projection rule, which reduces the number of variables. The other rules strictly reduce the measure. Therefore, after finite number of steps (in the course of transforming Δ_1) we stop either with an unsolvable problem, which immediately leads to termination, or return a solution ϑ of Δ_1. Hence, we arrive at $\vartheta(\Delta_2)$, which is postfix-unique and contains fewer variables compared to Γ, i.e., its measure is strictly smaller than the measure of Γ. It implies

Theorem 3. *The U-Post-CLU fragment is terminating.*

The considered control does not affect completeness.

Note that the splitting control is crucial for termination here. Otherwise, we may transform an (unsolvable) U-Post-CLU problem into its own variant:

$$\begin{array}{l}\{f(X,x) \stackrel{?}{=} f(a,Y,b), f(X,x) \stackrel{?}{=} f(Y,b)\} \rightsquigarrow_{\text{VE3"},\{X \mapsto \langle a,X'\rangle\}} \\ \{f(X',x) \stackrel{?}{=} f(Y,b), f(a,X',x) \stackrel{?}{=} f(Y,b)\} \rightsquigarrow_{\text{VE3"},\{Y \mapsto \langle a,Y'\rangle\}} \\ \{f(X',x) \stackrel{?}{=} f(a,Y',b), f(X',x) \stackrel{?}{=} f(Y',b)\}.\end{array}$$

With splitting, we first follow transformations of $\{f(X,x) \stackrel{?}{=} f(a,Y,b)\}$, obtaining its solution $\vartheta = \{X \mapsto \langle a, Y\rangle, x \mapsto b\}$. Then we would continue with $\vartheta(\{f(X,x) \stackrel{?}{=} f(Y,b)\}) = \{f(a,Y,b) = f(Y,b)\}$, transforming it with VE3' into $\{f(a) \stackrel{?}{=} f()\}$, which gives failure.

Unique Prefix Fragment (U-Pre-CLU) If in the definition of U-Post-CLU we change $\tilde{s_2}$ to $\tilde{s_1}$, we obtain the U-Pre-CLU fragment. To prove termination, we may modify the new rules and the arguments in the proof for the U-Post-CLU fragment as we did it for the I-KIF fragment.

Unique Variables in One Side (UV-CLU) These problems have the form $\Gamma = \{l_1 \stackrel{?}{=} r_1, \ldots, l_m \stackrel{?}{=} r_m\}$, where each variable occurring in $l_1, \ldots, l_m$ is unique in Γ. Simplification and Projection rules preserve this restriction. In the Variable

Elimination rules, σ can not affect the l's in Γ and, hence, the UV-CLU property is still preserved. We define complexity measure as a triple $\langle n, M_l, M_r \rangle$, where n is the number of distinct variables in Γ, M_l is the multiset of sizes of the l's, and M_r is the multiset of sizes of the r's. The measures are compared lexicographically. The rule VE3 does not increase n and either strictly reduces M_l (if $t(X, \tilde{s})$ in the rule corresponds to an r) or does not increase n and M_l and strictly reduces M_r (if $t(X, \tilde{s})$ corresponds to an l). It is not hard to see that the other rules strictly reduce the measure. Hence, UV-CLU is terminating.

Matching Fragment (CLM) In matching equations, one side does not contain variables (is ground). We write $l \ll^? r$ for a matching equation with r ground. The CLU rules can be simplified for them: No occurrence check; Projection and VE3 rules collapse into one rule:

$$\begin{aligned} \text{PVE}: \quad & t(X, \tilde{s}) \ll^? t(\tilde{q_1}, \tilde{q_2}) \rightsquigarrow \{t(\tilde{s}) \ll^? t(\tilde{q_2})\} \\ & \text{with } \sigma = \{X \mapsto \tilde{q_1}\}. \end{aligned}$$

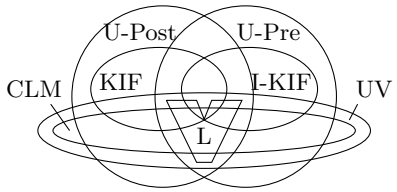
The rules preserve matching problems.

Theorem 4. *The CLM fragment is terminating.*

Proof. For CLM problems, each rule strictly reduces the complexity measure defined in the proof of Theorem 2.

Note that CLM is a special case of UV, because every matching problem $\{t_1 \ll^? t'_1, \ldots, t_n \ll^? t'_n\}$ can be expressed as the UV-problem $\{t'_1 \stackrel{?}{=} t_1, \ldots, t'_n \stackrel{?}{=} t_n\}$.

Relationships between the Terminating Fragments These are illustrated by the diagram below where the fragments are indicated by the V-like shape (L), four ellipses (KIF, I-KIF, CLM, and UV), and two circles (U-Post and U-Pre):



KIF and I-KIF are unitary, while the others are finitary.

4 Reasoning

Difficulty of CLU suggests basically two ways of extending classical results in resolution/superposition based theorem proving for the clausal fragment of Common Logic. One is to keep unification problems in constraints (as discussed, e.g., in [10] and described in more details in [11]), using incomplete light-weight methods to detect their unsatisfiability, and at the end, when only constraints are left,

employ the full-scale decision procedure to check if the proof succeeded or not. In this way we do not need to impose any syntactical restrictions on the CL clauses involved in the deduction.

The other approach avoids the expensive decidability test. This is achieved by restricting clauses to guarantee finitary unification. Also here we can choose between computing the unifiers and keeping the unification problems in constraints (whose decidability is easier than in the general case). In this paper we stick to the approach of computing unifiers.

Note that, in the theorem proving context, ordering is an important ingredient of calculi, which helps to prune the search space significantly. However, we do not consider ordering relations here. See, e.g., [12] for an example of an ordering that takes into account sequence variables.

A clause is a sentence $\forall \tilde{x}.\, L_1 \vee \cdots \vee L_n$, where each L is a *literal* (an atom or its negation). Following the standard convention, clauses are written without the quantifiers, but they are assumed to be there to identify the variables. We recall the rules for resolution-based theorem proving without equality (see, e.g., [10]). They remain valid for the clausal fragment of CL without equality we are dealing with in this section.

$$\text{Binary resolution:}\quad \frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma}.$$

$$\text{Factoring:}\quad \frac{C \vee A \vee B}{(C \vee A)\sigma}, \qquad \frac{C \vee \neg A \vee \neg B}{(C \vee \neg A)\sigma}.$$

In both rules, A and B are atomic sentences and $\sigma \in \mathit{mcsu}(\{A \stackrel{?}{=} B\})$. In the resolution rule, the given clauses are assumed to be variable disjoint.

We should introduce restrictions on the form of clauses to guarantee that the *mcsu* involved in the rules is always finite. The restrictions are based on the finitary fragments from the previous section. Let each clause satisfy the same syntactic criterion that defines one of the following fragments: L, KIF, I-KIF, U-Post, U-Pre. Then the resolution and factoring rules preserve these properties and the *mcsu*'s in the rules are finite. Hence, we obtain

Theorem 5. *Let S be a set of CL clauses. If each clause in S satisfies the same syntactic criterion that defines either L, KIF, I-KIF, U-Post, or U-Pre fragment, then binary resolution and factoring are finitely branching rules and their combination forms a refutationally complete proving method for S.*

We could think of putting these restrictions not on the entire clauses, but on each literal. That would certainly increase the classes covered but, unfortunately, even the linear literals make the *mcsu* infinite: Consider factoring the clause $p(a, X) \vee p(X, a)$. However, looking into the problem carefully, one may notice that, for the binary resolution rule the *mcsu* is still finite even under this new restriction.

This new observation leads us to look for a form of clauses for which the binary resolution rule alone is refutationally complete. As it has been shown in [10, 13], Horn clauses (clauses with at most one positive literal) have such a

property. These papers also indicate that the order of negative literal selection does not matter (free selection). It remains valid in our case as well. Summarizing all the observations, we obtain

Theorem 6. *Let S be a set of CL Horn clauses. If each literal in S satisfies the same syntactic criterion that defines either L, KIF, I-KIF, U-Post, or U-Pre fragment, then binary resolution (with free selection) is a finitely branching, refutationally complete proving method for S.*

We can relax the conditions of the last theorem, removing all the restrictions from the negative literals in Horn clauses, but requiring that the positive ones to be linear:

Theorem 7. *Let S be a set of CL Horn clauses. If each positive literal occurring in the clauses in S is linear, then binary resolution (with free selection) is a finitely branching, refutationally complete proving method for S.*

Proof. The finite branching property follows from the fact that the CLU problem $\{A \stackrel{?}{=} B\}$ in the binary resolution rule satisfies the UV restriction: A is linear and disjoint from B. Refutational completeness can be proved as in the standard first-order case [10, Theorem 7.2].

A consequence of this result is the completeness of SLD-resolution for Common Logic under relatively liberal restrictions. It, like Theorem 6, can be used for logic programming in CL that itself can be seen as a generalization of logic programming in HiLog [14].

Finitary unification introduces extra backtracking points in programming. Hence, from this point of view, CL programs that obey the KIF and I-KIF restrictions behave exactly like the usual (sequence variable free) logic programs, because these restrictions lead to unitary unification algorithms.

5 Programming

Theorems 6 and 7 provide a basis for logic programming in Common Logic. Now, we take advantage of finitary matching in CL and demonstrate how it can be used in rule-based programming with sequence transformations that we can model as well-moded CL programs. Such rules can be used, for instance, for a CL conforming query language.

The logic programs that we consider in this section are constructed from atoms of the form $t(\mathsf{f}(\tilde{s}), \mathsf{f}(\tilde{q}))$, where f is a name whose occurrence in these designated positions gets a special treatment: It is used as a kind of "wrapper" around sequences, to distinguish the "input" arguments $\tilde{s}$ from the "output" ones $\tilde{q}$. As a syntactic sugar, we write $t(\mathsf{f}(\tilde{s}), \mathsf{f}(\tilde{q}))$ as $t :: \tilde{s} \Longrightarrow \tilde{q}$. We build clauses just from the atoms of this form. For simplicity, here we consider Horn clauses only, i.e., we deal with definite programs. Queries are negative Horn clauses. Following the usual conventions, we write clauses as $A \leftarrow B_1, \ldots, B_n$ and queries as $\leftarrow B_1, \ldots, B_n$.

The standard definition of well-modedness [15] is adapted to our queries and clauses:

- A query $t_1 :: \tilde{s}_1 \Longrightarrow \tilde{q}_1, \ldots, t_n :: \tilde{s}_n \Longrightarrow \tilde{q}_n$ is well-moded, if $var(t_i, \tilde{s}_i) \subseteq \cup_{j=1}^{i-1} var(\tilde{q}_j)$ for all $1 \leq i \leq n$.
- A clause $t_0 :: \tilde{q}_0 \Longrightarrow \tilde{s}_{n+1} \leftarrow t_1 :: \tilde{s}_1 \Longrightarrow \tilde{q}_1, \ldots, t_n :: \tilde{s}_n \Longrightarrow \tilde{q}_n$ is well-moded if $var(t_i, \tilde{s}_i) \subseteq \cup_{j=0}^{i-1} var(t_0, \tilde{q}_j)$ for all $1 \leq i \leq n+1$.

The operational semantics is given by the following two inference rules, resolution and identity factoring: (The name id used in them is interpreted as the identity.[4])

$$\text{Resolution: } \frac{\leftarrow t :: \tilde{s} \Longrightarrow \tilde{q}, Q}{\leftarrow (Body, \mathsf{id} :: \tilde{s}' \Longrightarrow \tilde{q}, Q)\sigma},$$

if $t' :: \tilde{q}' \Longrightarrow \tilde{s}' \leftarrow Body$ is a fresh variant of some program clause, t and $\tilde{s}$ are ground, and σ is a matcher of the CLM problem $\{t' \ll^? t,\ \mathsf{f}(\tilde{q}') \ll^? \mathsf{f}(\tilde{s})\}$.

$$\text{Identity factoring: } \frac{\leftarrow \mathsf{id} :: \tilde{s} \Longrightarrow \tilde{q}, Q}{\leftarrow Q\sigma},$$

where $\tilde{q}$ is ground and σ is a matcher of $\{\mathsf{f}(\tilde{s}) \ll^? \mathsf{f}(\tilde{q})\}$.

Since CLM is finitary, both rules introduce branching points by selecting σ nondeterministically. As a decision problem, CLM is NP-complete. Membership in NP is easy. Therefore, we prove here NP-hardness. It can be shown by reduction from the NP-hard positive 1-IN-3-SAT [16], which is the usual method for similar problems (see, e.g., [7,17]). A positive 1-IN-3-SAT problem is given by a set of clauses $\{C_1, \ldots, C_n\}$ where each clause C_i contains exactly three positive literals $p_i^1 \vee p_i^2 \vee p_i^3$ from a set of literals $p_1, \ldots, p_m$. A truth assignment solves the problem if it maps exactly one literal from each clause to true. For each p_i, we introduce a sequence variable X_i. We choose two function symbols c and t to represent, respectively, clause and truth. The given positive 1-IN-3-SAT problem is encoded as the following CLM problem:

$$\{c(X_1^1, X_1^2, X_1^3) \ll^? c(t), \ldots, c(X_n^1, X_n^2, X_n^3) \ll^? c(t)\}$$

This encoding is polynomial and preserves solvability in both directions: p_i^j is true iff X_i^j is instantiated with t and p_i^j is false iff X_i^j is instantiated with the empty sequence. Hence, the matching problem is solvable iff the corresponding 1-IN-3-SAT problem is solvable. It implies that CLM is NP-hard. Hence, we proved the theorem:

Theorem 8. *CLM is NP-complete.*

The encoding shows that, in fact, individual variables are not needed to show NP-completeness of CLM. We can make the result stronger, showing that

[4] More precisely, the relational extension of the denotation of id contains all pairs of the same elements over the domain of discourse.

CLM is NP-complete even if no sequence variable occurs in it more than twice. For this, we use the encoding of positive 1-IN-3-SAT into two kinds of CLM equations. The first one is almost the same as above, the only difference is that each occurrence of the literal p_i is encoded with a fresh sequence variable. The second kind of equations has the form $f(Y_i^1, f(g(X_i^1), \ldots, g(X_i^{n_i})), Y_i^2) \ll^? f(f(g(t), \ldots, g(t)), f(g(), \ldots, g()))$, in which n_i is the number of occurrences of the literal p_i in the given positive 1-IN-3-SAT problem, X_i^j is the variable that encodes the j'th occurrence of the literal p_i in the first kind of equation, and $g(t)$'s and $g()$'s occur n_i times in the right hand side. It guarantees that all the variables $X_i^1, \ldots, X_i^{n_i}$ are instantiated at the same time either with t or with the empty sequence. This corresponds to the fact that all occurrences of p_i in the 1-IN-3-SAT problem are either true or false at the same time. We will have at most m such equations, since there are m literals in the given 1-IN-3-SAT problem: $\{p_1, \ldots, p_m\}$. An equation is written only for those p's which occur more than once in the given problem. Its size linearly depends on the number of occurrences of the corresponding literal in the 1-IN-3-SAT problem. Hence, the encoding is polynomial and each sequence variable occurs in this encoding at most twice: Once in the first kind and at most once in the second kind of equation. It shows that the following theorem holds:

Theorem 9. *CLM with at most two occurrences of sequence variables and with no occurrences of individual variables is NP-complete.*

Moreover, we can show that the time complexity of linear CLM is $O(n^3)$. Let $t \ll^? s$ be a linear CLM problem. Let F be the set of names in s. Let t' be the term obtained from t by replacing all occurrences of individual variables with a symbol X, and all occurrences of sequence variables with a symbol Y. We construct the CFG $G = (N, S, P)$ with nonterminals $N = \{S, X\}$, start nonterminal S, and set of productions

$$P = \{S \to t'\} \cup \{X \to X(Y)\} \cup \{X \to f \mid f \in F\} \cup \\ \{Y \to \epsilon\} \cup \{Y \to X, Y\}.$$

Note that the nonterminal X is intended to generate bindings for individual variables, and the nonterminal Y is intended to generate bindings for sequence variables. Since t is a linear term, we have $t \ll^? s$ if and only if $s \in L(G)$. Obviously, the size of this CFG language membership problem is $O(n)$, where n is the size of the corresponding matching problem. Since recognition and parsing of context-free languages can be achieved in $O(n^3)$ time (see [18]), we proved the theorem:

Theorem 10. *The time complexity of linear CLM is in $O(n^3)$.*

Finally, we note that, KIF and I-KIF fragments of CLM are in P and conjecture that the unique prefix and postfix ones are also such. As a counting problem (computing the cardinality of the minimal complete set of matchers, see [19]), #CLM is #P-complete.

Example 7. We can program first-order term rewriting in the rule-based style just described:

$$rw(z) :: x \Longrightarrow y \leftarrow z :: x \Longrightarrow y.$$
$$rw(z) :: u(X, x, Y) \Longrightarrow u(X, y, Y) \leftarrow rw(z) :: x \Longrightarrow y.$$

Assuming that we have a rule $r :: f(x, y) \Longrightarrow x$ in the program, we can compute values for the variable x in the goal $rw(r) :: f(f(a, b), f(b, c)) \Longrightarrow x$. These are $f(a, b)$, $f(a, f(b, c))$, and $f(f(a, b), b)$. This example illustrates how one can select a subterm at arbitrary depth.

With sequence variables, one can compactly formulate quite complex, even incomplete queries. Terms in the functional position can encode strategies and their combinations as they appear in rule-based calculi such as ρ-calculus [20] and ρLog [21]. The expressive power can be further increased by introducing the concept of negation as failure. It will allow, for instance, to encode the strategy for computing normal forms.

6 Discussion

Common Logic has evolved from first-order logic. Its evolution has been motivated by the recent developments in the use of the Web for representing, sharing, and reasoning upon knowledge. A tool that adopts CL for these tasks will encounter some kind of equation solving problem over CL expressions. Variable polyadicity, cross-categoricity, and especially the use of sequence variables make CL equation solving quite a difficult task. To make it practically useful, one has to identify fragments of CL expressions that admit "easy" solving methods. To the best of our knowledge, this problem has not been addressed systematically from this perspective.

In this paper, we aimed at filling this gap, studying computationally well-behaved fragments of CL equation solving. These fragments, in fact, cover important practical cases.

CLM is a very important one, because it can form the basis of (rewrite or transformation) rule-based programming for CL. It can be used for important operations over CL expressions such as transformation, extraction, and querying. Unrestricted CL matching is NP-complete, but we know many examples of successful use of NP-complete matching algorithms in computation. Some such instances are associative-commutative matching which is used, for instance, in Maude [22], associative matching used in Tom [23], and sequence matching used in Mathematica [24]. Moreover, if we combine CLM with the other restrictions (e.g., linearity or KIF) we obtain polynomial fragments with a great potential for rule-based querying and transforming expressions written in CL syntax.

The practical value of the UV fragment can be illustrated on the basis of Theorem 7, in the context of Horn clause programming. It basically says that one does not have to bother about any syntactic restrictions in the bodies of such clauses, provided that the clause heads are linear.

The other fragments play an important role in representing knowledge in CL and, subsequently, reasoning over it. We have analyzed the form of CL expressions with sequence variables in the SUMO and its domain ontologies.[5] The analysis revealed the fact that the fragments considered in our paper are, actually, the most typical ones: 97.6% of the formulas with sequence variables fall in one of those categories. Sequence variables occur in SUMO, geography, and distributed computing ontologies 85 times (without counting occurrences in the quantifier prefix) in 42 formulas. Out of them, 1 formula (with 2 occurrences of sequence variables) has a shape that would lead to U-Post-CLU or U-Pre-CLU problem, 4 formulas (6 occurrences) classify for I-KIF, and 36 formulas (75 occurrences) for KIF. Such a dominance of the KIF fragment is not surprising, because these ontologies originally have been based on KIF (a predecessor of CL). Only recently they have been translated to CL.[6]

We think that our work can be useful for ontology developers in CL. Ontology development in CL is still in its early stage. Based on the fragments discussed here, ontology developers can employ sequence variables more freely (using more well-behaved fragments instead of concentrating only on KIF) and still have ontologies amenable to reasoning tools.

Our work can be useful for the authors of reasoning tools as well. The technique of sequence variable expansion [25] (in the original terminology, row variable expansion) used for dealing with sequence variables is incomplete. It adds new formulas and changes semantics. From each formula with n sequence variables, m^n new formulas are generated, where m is a predefined number. The new formulas are obtained by replacing each sequence variable by $0, 1, \ldots, m-1$ fresh individual variables. Our results can make this technique obsolete in the vast majority of practical cases, as the above mentioned analysis of ontologies showed. The reasoners that incorporate algorithms for terminating fragments of CLU will need to perform the expansion very rarely and will not loose completeness for this reason. It will make them more valuable for ontology consistency checking: If reasoning using the row variable expansion technique does not reveal inconsistency, one can not say whether it really is consistent or not because of incompleteness of the technique. The more ontologies written in CL appear, the more will be the demand for corresponding reasoners, and algorithms for terminating solving fragments will be useful there.

Acknowledgment

This work has been partially supported by the Austrian Science Fund (FWF) under the project P 24087-N18, and by the EC FP7-ICT Project SPRERS 246839.

[5] http://www.ontologyportal.org/
[6] http://www.kojeware.com/sumo-cl.clif

References

1. Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In Robinson and Voronkov [26], pages 19–99.
2. Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on Java. In Franz Baader, editor, *RTA*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.
3. Weidong Chen, Michael Kifer, and David Scott Warren. HiLog: a foundation for higher-order logic programming. *JLP*, 15(3):187–230, 1993.
4. Horatiu Cirstea and Claude Kirchner. The rewriting calculus - parts I and II. *Logic Journal of the IGPL*, 9(3), 2001.
5. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *TCS*, 285(2):187–243, 2002.
6. Piotr Dembinski and Jan Maluszynski. And-parallelism with intelligent backtracking for annotated logic programs. In *SLP*, pages 29–38, 1985.
7. Michael R. Genesereth and Richard E. Fikes. Knowledge Interchange Format. Version 3.0. Reference Manual. Technical Report KSL-92-86, Comp. Sci. Department, Stanford University, June 1992.
8. Miki Hermann and Phokion G. Kolaitis. The complexity of counting problems in equational matching. *JSC*, 20(3):343–362, 1995.
9. Ian Horrocks and Andrei Voronkov. Reasoning support for expressive ontology languages using a theorem prover. In Jürgen Dix and Stephen J. Hegner, editors, *FoIKS*, volume 3861 of *LNCS*, pages 201–218. Springer, 2006.
10. ISO/IEC. Information technology—Common Logic (CL): A framework for a family of logic-based languages. International Standard ISO/IEC 24707, 2007. Available online at http://standards.iso.org/ittf/PubliclyAvailableStandards/-c039175_ISO_IEC_24707_2007(E).zip.
11. Temur Kutsia. Theorem proving with sequence variables and flexible arity symbols. In Matthias Baaz and Andrei Voronkov, editors, *LPAR*, volume 2514 of *LNCS*, pages 278–291. Springer, 2002.
12. Temur Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In Jacques Calmet, Belaid Benhamou, Olga Caprotti, Laurent Henocque, and Volker Sorge, editors, *AISC*, volume 2385 of *LNCS*, pages 290–304. Springer, 2002.
13. Temur Kutsia. Equational prover of Theorema. In Robert Nieuwenhuis, editor, *RTA*, volume 2706 of *LNCS*, pages 367–379. Springer, 2003.
14. Temur Kutsia. Solving equations with sequence variables and sequence functions. *JSC*, 42(3):352–388, 2007.
15. Temur Kutsia, Jordi Levy, and Mateu Villaret. On the relation between context and sequence unification. *JSC*, 45(1):74–95, 2010.
16. Christopher Lynch. Oriented equational logic programming is complete. *JSC*, 23(1):23–45, 1997.
17. Mircea Marin and Temur Kutsia. Foundations of the rule-based system ρLog. *J. Applied Non-Classical Logics*, 16(1-2):151–168, 2006.
18. Christopher Menzel. Knowledge representation, the World Wide Web, and the evolution of logic. *Synthese*, 182(2):269–295, 2011.
19. Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Robinson and Voronkov [26], pages 371–443.

20. Adam Pease and Christoph Benzmüller. Sigma: An integrated development environment for formal ontology. *AI Commun. (Special Issue on Intelligent Engineering Techniques for Knowledge Bases)*, 2012. In print.
21. Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004.
22. John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.
23. Thomas J. Schaefer. The complexity of satisfiability problems. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *STOC*, pages 216–226. ACM, 1978.
24. M. Schmidt-Schauß and J. Stuber. The complexity of linear and stratified context matching problems. *Theory of Computing Syst.*, 37(6):717–740, 2004.
25. Stephen Wolfram. *The Mathematica Book*. Wolfram Media, 5th edition, 2003.
26. D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.