Серия «Математика»
Том 1 (2007), № 1, С. 188—204

Онлайн-доступ к журналу:
http://isu.ru/izvestia

ИЗВЕСТИЯ
Иркутского
государственного
университета

УДК 510.62: 004.82

Logic programing in knowledge domains

A. V. Mantsivoda, V. A. Lipovchenko, A. A. Malykh
({andrei, lip, malykh}@baikal.ru)
Irkutsk State University, Irkutsk

Abstract. We propose an approach to combining logic programming and knowledge representation paradigms. This approach is based on the conception of description terms. LP and KR are integrated in such a way that their underlying logics are carefully separated. A core idea here is to push the KR techniques on the functional level. On the LP level the knowledge base is considered as a constraint store, in which special propagation methods are ruling. A $\mathcal{NCC}$ calculus that handles description terms is developed as an underlying inference system for propagation. On the basis of this formalism, a constraint logic programming language integrating both LP and KR approaches is designed.

Keywords: logic programing, description logics, description term, constraint logic programing, naming constraints calculus.

Introduction

In this paper we propose an approach to integration of the logic programming and knowledge representation paradigms. Knowledge representation techniques are popular nowadays. Some large and ambitious projects incorporate KR as a significant component. For instance, in the Semantic Web [2] KR is used for describing knowledge domains, information resources, and developing metadata. KR formalisms are mostly based on description logics (DL) [1], which offer flexible tools for knowledge representation. Many efforts have been made to develop efficient inference systems and automated solvers for DLs [9][7][6][1].

Though description logics are elegant and useful, logic programming (LP) also has a number of strong means for knowledge management. In particular, LP is very well tuned to explicit object handling, whereas DLs are more focused on general knowledge manipulation (*e.g.*, solving classification problems in knowledge bases). Hence the idea of incorporation of LP tools in DLs looks quite attractive [5][3]. But here we have a serious

obstacle. Both DLs and LP are based on *constructive* logical systems. But since they came from the different subsets of the first order logic, their constructive properties have *different* origins. Therefore if we try to mix up LP and DL within a generalized logical system we have to sacrifice a lot [5]. Moreover, the fragile nature of DLs becomes unprotected against a number of tough LP-methods, such as negation as failure, and this can bring either additional restrictions, or additional logical problems (see the discussion [8][10]).

In [11] we have introduced an approach to knowledge representation, which is based on the notion of a description term. This approach develops and refines the paradigm of semantic programming [4]. The main idea behind description terms is to move knowledge representation techniques from the logical level to the functional level of terms and objects. As for integration of LP and DL styles, this idea allows us not to mix up the two styles within a joined logical formalism, but keep them separated while preserving tight interconnections between them.

1. The General Scheme

Let Δ be a knowledge domain. We introduce a special kind of terms called *description terms*. A description term t is interpreted as a description of some element (object) $d \in \Delta$. This description includes information about *classes* (concepts) and *properties* (rôles), which characterize d. A description can be incomplete, if it contains only partial data about d. Moreover, one term can describe many objects (for instance, if a term says only that the person's name is John, this term describes all Johns in Δ). Thus, t must be interpreted as the set of those objects $d \in \Delta$, which t describes, that is $t^I \subseteq \Delta$, or more precisely $t^I = \{d \mid t \stackrel{\wedge}{=} d, d \in \Delta\}$. Here $\mathbf{I}$ is an interpretation, $t \stackrel{\wedge}{=} d$ means that d is described by t. Since data in terms is incomplete, it is useful to be able to compare them. We say that t_1 is *approximated* by t_2 ($t_1 \gg t_2$) if all information in t_2 is contained also in t_1. Note that if $t_1 \gg t_2$ then $t_1^I \subseteq t_2^I$, that is, the more precise information t_1 has, the less number of objects it describes. Also it is natural to have an operation, which merges data from a couple of terms t_1 and t_2 within one term. We call this operation *amalgamation* and define it as the least upper bound of t_1 and t_2 with respect to the partial order $\gg$: a term t is the amalgam of t_1 and t_2 ($t = t_1 \sqcap t_2$) if $t \gg t_1$, $t \gg t_2$ and for any t_0, such that $t_0 \gg t_1$ and $t_0 \gg t_2$, we have $t_0 \gg t$. Evidently, for soundness of the whole idea, it is necessary to have $(t_1 \sqcap t_2)^I = t_1^I \cap t_2^I$.

Note that standard logic programming terms (LP-terms) can be regarded as a particular case of description terms. First, LP-terms can contain partial and incomplete information (if they are non-ground and have occurrences of unbound variables). Second, during computation the information

in terms becomes more and more precise due to substitutions of variables. Third, given two LP-terms t_1 and t_2, we can compare information saved in them, as well as amalgamate this information. Both the comparison and amalgamation can be performed thanks to the most general unifier. We put $t_1 \gg t_2$ if $t_1 = MGU(t_1, t_2)$ and $t = t_1 \sqcap t_2$ if $t = MGU(t_1, t_2)$. As the knowledge domain Δ we can take an Herbrand universe H. Then t describes any $t' \in H$ such that $t' = MGU(t, t')$, that is $t^I = \{t' \mid t' \in H, t' = MGU(t, t')\}$.

This consideration partially justifies the step of incorporation of description terms in logic programming: we substitute there LP-terms by description terms. The main inference rule in this case remains almost the same, but with slight generalization:

$$\text{(LPd)} \qquad \frac{p(t_1, \ldots, t_k), p_2, \ldots, p_n \qquad p(t'_1, \ldots, t'_k) : -r_1, \ldots, r_m}{(r_1, \ldots, r_m, p_2, \ldots, p_n)\theta}$$

if $t_i\Theta \gg t'_i\Theta$ for $i = \overline{1,k}$. Here t_i and t'_i are description terms, the left premise is the goal, the right premise is a rule, r_i and p_i are atoms. The LPd-rule is applied if there exists a substitution $\Theta = \{x_1/t_1, \ldots, x_k/t_k\}$, where t_i are description terms, such that all $t'_i\Theta$ approximate corresponding $t_i\Theta$.

The next thing we should do is to establish in our system the knowledge base (KB), which keeps knowledge about the domain Δ. We organize KB in the form of a constraint store, which consists of *naming constraints*. A naming constraint is an expression of the form

$$id :: t$$

where id is the name (identifier) of an object in Δ, and t is its description. This constraint means that an object d of Δ named id is described by t. $id :: t$ is true in an interpretation I, if $id^I \in t^I$. It is convenient to have two types of names in our system. The first type is *unique* names, which uniquely define objects (that is, if $id_1 \neq id_2$ then $id_1^I \neq id_2^I$). Unique names correspond to the conception of names in description logics, which are also unique. The other type is *temporary* names. For instance, in order to manipulate information about 'someone who's killed John' we assign this unknown person a new temporary name, which allows concentration of data about this criminal using naming constraints. The core difference between these two types of names is that temporary names do not preserve uniqueness, that is, two different temporary names can identify the same object. In the LPd-rule both the unique and temporary names play the rôle of constants, but in checking $t_i \gg t'_i$ naming constraints can be involved to reflect the context, in which the rule is applied.

We also need to incorporate *axioms*, which describe the knowledge domain as a whole. In description logics axioms have the form of inclusions

or equivalences, which are satisfied by any element of Δ. For instance, the axiom $Man \sqsubseteq Person$, which is equivalent to $\neg Man \sqcup Person$, means that any element of Δ is either not-a-man or a person. The *dual nature* of description terms helps us to introduce axioms in our scheme. On the one hand, in naming constraints description terms describe single objects. On the other hand, since description terms are interpreted as subsets of Δ, we can use them to describe also the sets of objects. So, as axioms we use those description terms, which describe *all* objects of Δ, that is, $t^I = \Delta$.

Application of axioms depends on the inference system and propagation strategies, which are ruling in the naming constraint store. In the general scheme we do not specify this inference system, because different entailments can play this role. For instance, in the sequel we develop an implementation of the general scheme and suggest some new inference system based on modification of the resolution principle [13]. But it is also possible to exploit modifications of tableau algorithms [1].

2. Description Terms

In this section we introduce description terms, which are the basic notion of this paper. Let $\Re = \langle M_1, \ldots, M_k; \Omega \rangle$ be a data type model, where M_i are sorts (data types) and Ω is a signature. We assume that all elements of $\Re$ are distinguished. For any M_i we introduce the minimal set of constants $\bar{M}_i$ such that for any $\hat{m} \in M_i$ two constants m and $\sim m$ belong to $\bar{M}_i$. The set of all constants corresponding to elements of $M = M_1 \cup \ldots \cup M_k$ is denoted by $\bar{M} = \bar{M}_1 \cup \ldots \cup \bar{M}_k$.

The language of description terms has the following basic components:

1. The set of constants $\bar{M}$;
2. The set of sort names $S = \{s_1, \ldots, s_k, \sim s_1, \ldots, \sim s_k\}$ of the model $\Re$.
3. The 'top' and 'bottom' constants $\top$, $\bot$;
4. The infinite set of unique names $ID = \{id_1, id_2, \ldots\}$;
5. The infinite set of temporary names $DX = \{dx_1, dx_2, \ldots\}$;
6. The set of attribute (rôle) names $Attr = \{p_1, p_2, \ldots, p_k\}$;
7. The set of atomic concepts (class names) and negative concepts: $CN = \{c_1, c_2, \ldots, c_p, \sim c_1, \sim c_2, \ldots, \sim c_p\}$.

We denote $IX = ID \cup DX$, and use ix (possibly with indices) to denote names from IX.

Now we can define the set $T_{\Re}$ of description terms.

Definition 1. *1. If $t \in CN \cup IX \cup \{\bot, \top\}$ then t is an atomic description term.*

2. If $p \in Attr$ and $r \in \bar{M} \cup IX \cup S \cup T_{\Re}$, then $p : r$ and $p * r$ are atomic description terms.

3. If $t_1, t_2, \ldots, t_n$ are atomic description terms then $(t_1; t_2; \ldots; t_n)$ is a description term. If $n = 0$ then $(t_1; \ldots; t_n) = \bot$.

4. If $t_1, t_2, \ldots, t_n$ are description terms then $(t_1, t_2, \ldots, t_n)$ is a description term. If $n = 0$ then $(t_1, \ldots, t_n) = \top$.

As a rule we omit parentheses supposing $((t_1, t_2), t_3) = (t_1, (t_2, t_3)) = t_1, t_2, t_3$ and $((t_1; t_2), t_3) = t_1; t_2, t_3$, and $(t_1, (t_2; t_3)) = t_1, t_2; t_3$. Also we do not distinguish the order: $t_1, t_2 = t_2, t_1$ and $t_1; t_2 = t_2; t_1$. Note that the term $(t_1, t_2); t_3$ is impossible.

Table I. The semantics of description terms

Terms	Described objects
c	$c^I \subseteq \Delta$
$\sim c$	$\Delta \setminus c^I$
$t_1, \ldots, t_n$	$t_1^I \cap \ldots \cap t_n^I$
$t_1; \ldots; t_n$	$t_1^I \cup \ldots \cup t_n^I$
s_i	$s_i^I = \bar{M}_i$
$p : t$	$\{x \mid x \in \Delta, \exists y \in t^I : \langle x, y\rangle \in p^I\}$
$p * t$	$\{x \mid x \in \Delta, \forall y \in \Delta : \langle x, y\rangle \in p^I \rightarrow y \in t^I\}$
$p : s_i$	$\{x \mid x \in \Delta, \exists y \in \bar{M}_i : \langle x, y\rangle \in p^I\}$
$p * s_i$	$\{x \mid x \in \Delta, \forall y : \langle x, y\rangle \in p^I \rightarrow y \in \bar{M}_i\}$
$p : m$	$\{x \mid x \in \Delta, \langle x, m\rangle \in p^I\}$
$p * m$	$\{x \mid x \in \Delta, \forall y : \langle x, y\rangle \in p^I \rightarrow y = m\}$
ix	$ix^I \in \Delta$
$\top$	$\top^I = \Delta$
$\bot$	$\bot^I = \emptyset$

The denotational semantics of description terms is summarized in table I. This semantics is significantly the same as that in DLs. It is important, because we want to be in touch with DLs all the time. We use the lightweight syntax (the comma for conjunction and the semicolon for disjunction) to follow LP traditions (though note that ';' has greater priority, *i.e.* $t_1; t_2, t_3$ is understood as $(t_1; t_2), t_3$). Besides, the lightweight syntax seems more readable and attractive, but, of course, this is a matter of taste. And we stress again that descriptions are understood as *terms*, which do not have the logical values of true and false, but are interpreted as sets of objects.

Definition 2. *A naming constraint is an expression of the form $ix :: t$, where $ix \in IX$ and $t \in T_{\Re}$. The denotational semantics of naming constraints is defined as follows: $\mathbf{I} \models ix :: t$ iff $ix^{\mathbf{I}} \in t^{\mathbf{I}}$.*

Here are examples of description terms and naming constraints:

Example 1. `Id-john :: gardener;mechanic, hasSpouse:Id-mary, hasChild*~male Id-mary :: sex:"f" , studies-at:university`

John is either a gardener or a mechanic, his spouse is Mary, and all his children (if any) are not boys. Mary is female, she studies at some university.

The version of description terms we consider in this section correlates with the description logic $\mathcal{ALC}$ [1]. The language of this DL includes atomic concepts A_i, universal concept $\top$, bottom concept $\bot$, full negation over arbitrary concepts $\neg C$, universal quantification $\forall R.C$. Traditionally, we explicitly include in the language disjunction and full existential quantification, because they are expressed in $\mathcal{ALC}$: $C \sqcup D \equiv \neg(\neg C \sqcap \neg D)$ and $\exists R.C \equiv \neg\forall R.\neg C$.

To work with $\mathcal{ALC}$ and description terms simultaneously, we introduce the notion of synonyms: every atomic concept A of $\mathcal{ALC}$ is the synonym for an exactly one $a \in CN$, which is in its turn the synonym of A, and this is denoted $a = \mathrm{syn}(A)$. An interpretation I preserves synonyms if $a^I = \{x \mid I \models A(x)\}$ for a ny $a = \mathrm{syn}(A)$. Let I be an interpretation preserving synonyms. An $\mathcal{ALCC}$–concept F and a description term t are equivalent in $\mathbf{I}$, if $\{x \mid I \models F(x)\} = t^{\mathbf{I}}$. A concept and a description term are equivalent if they are equivalent in any interpretation preserving synonyms. It is easy to show that for any $\mathcal{ALCC}$–concept and axiom F there exists an equivalent description term. For this we should transform F into 'incomplete' conjunctive normal form (negation is pushed to atomic concepts, but disjunctions and conjunctions are not pushed through quantifiers), using the standard transformation rules: $C \sqsubseteq D \to \neg C \sqcup D$, $C \equiv D \to (\neg C \sqcup D) \sqcap (\neg D \sqcup C)$, $\neg(A \sqcap B) \to \neg A \sqcup \neg B$, $\neg(A \sqcup B) \to \neg A \sqcap \neg B$, $\neg\exists R.A \to \forall R.\neg A$, $\neg\forall R.A \to \exists R.\neg A$, $C \sqcup (A \sqcap B) \to (C \sqcup A) \sqcap (C \sqcup B)$, $(A \sqcap B) \sqcup C \to (A \sqcup C) \sqcap (B \sqcup C)$.

For concepts in conjunctive normal form transformation to description terms is straightforward: $trans(A) = a$, $trans(\neg A) = {\sim}a$, $trans(A \sqcap B) = trans(A), trans(B)$, $trans(A \sqcup B) = trans(A); trans(B)$, $trans(\exists P.A) = p : trans(A)$, $trans(\forall P.A) = p * trans(A)$, where $a = \mathrm{syn}(A)$.

Proposition 1. *For any $\mathcal{ALCC}$–concept/axiom F there exists $t \in T_{\Re}$, such that F and t are equivalent.*

Proof is straightforward and based on the properties of cnf and the semantics of description terms.

Thus, any DL–axiom F can be transformed into an equivalent description term t, and $t^I = \top^I$ for I such that $I \models F$. By definition, the general form of description terms is

$$t = (t_1^1; \ldots; t_{n_1}^1), \ldots, (t_1^m; \ldots; t_{n_m}^m)$$

where t_i^j are atomic. Evidently, if $t^I = \top^I$ then $(t_1^j; \ldots; t_{n_j}^j)^I = \top^I$ for any $j = \overline{1,m}$. That is, every $t^j = t_1^j; \ldots; t_{n_j}^j$ is also an axiom. For some reasons we prefer to use as axioms separate t^j instead of heavyweight $t^1, \ldots, t^m$. Thus we transform any DL–axiom into the set of separate disjunctive description terms of the form $t_1^j; \ldots; t_{n_j}^j$.

Note that an axiom $t_1; \ldots; t_n$ stratifies the domain Δ into n (not necessarily disjoint) segments (that is, $t_1^{\mathbf{I}} \cup \ldots \cup t_n^{\mathbf{I}} = \top^{\mathbf{I}} = \Delta$), and any object $d \in \Delta$ must belong to at least one of them.

3. A Calculus for Naming Constraints

In this section we consider syntactic methods for handling naming constraints. We start with introduction of the two key operations acting on description terms — approximation and amalgamation. Approximation checks whether a description term contains all data saved in another term. Since we plan to use approximation in the core of our system (in particular, in the rule LPd), then approximation must be (1) syntactically definable, and (2) algorithmically lite (as much as possible).

Definition 3. *Approximation is denoted by $\gg$ and defined as follows:*

1. $f \gg f$, *for any* $f \in CN \cup IX$;
2. $p : f \gg p : g$ *and* $p * f \gg p * g$, *if one of the following holds:*
   a) $f, g \in \bar{M} \cup S$ *and* $f = g$;
   b) $g = s_i \in S$ *and* $f \in \bar{M}_i$;
   c) $g = {\sim}m_i \in S$ *and* $f = s_j$, $j \neq i$;
   d) $f, g \in T_{\Re}$ *and* $f \gg g$;
3. $t_1, \ldots, t_n \gg t'_1, \ldots, t'_m$, *if* $\forall i \in \{1..m\}, \exists j \in \{1..n\} : t_j \gg t'_i$;
4. $t_1; \ldots; t_n \gg t'_1; \ldots; t'_m$, *if* $\forall j \in \{1..n\}, \exists i \in \{1..m\} : t_j \gg t'_i$;
5. $\bot \gg t$ *for any* $t \in T_{\Re}$;
6. $t \gg \top$ *for any* $t \in T_{\Re}$.

This definition is simple and natural, though a little bit awkward due to its syntactic nature. $t_1 \gg t_2$ can be informally characterized as 't_1 is *evidently* more precise than t_2' or 't_1^I is evidently subsumed by t_2^I'.

Proposition 2. *If* $t_1 \gg t_2$ *then* $t_1^{\mathbf{I}} \subseteq t_2^{\mathbf{I}}$ *in any interpretation* $\mathbf{I}$. *In particular, if* $ix \gg t$ *then* $\models ix :: t$.

Proposition 3. *(1)* $t \gg t$. *(2) If* $t_1 \gg t_2$ *and* $t_2 \gg t_3$ *then* $t_1 \gg t_3$.

The last proposition shows that $\gg$ is a partial order. Let us introduce the operation of amalgamation $t_1 \sqcap t_2$, which is very simple for this kind of description terms:

Definition 4. *We define* $t_1 \sqcap t_2 = (t_1, t_2)$.

The following proposition shows that this definition of amalgamation is correct w.r.t. approximation:

Definition 5. $t_1 \sqcap t_2$ *is the least upper bound of* t_1 *and* t_2 *w.r.t. the partial order* $\gg$. *For any interpretation* I, $(t_1 \sqcap t_2)^I = t_1^I \cap t_2^I$.

Now let us introduce an inference system for naming constraints (for brevity we denote it $\mathcal{NCC}$ – the naming constraints calculus). In this system we again rely on duality of description terms. Being terms for the outside world, inside they must have logical behaviour, which is described in the inference system represented in table II.

Table II. Naming constraints calculus

(Res) $\dfrac{ix :: (c; t_1), (\sim c; t_2), t}{ix :: t_1; t_2}$	(Any) $\dfrac{ix_1 :: (p : t; t_1), (p * t'; t_2), t_3}{ix_1 :: p : (t, t'); t_1; t_2}$
(Ax) $\dfrac{\mathrm{Axiom}(t_1; \ldots; t_k)}{ix :: t_1; \ldots; t_k}$	(Ama) $\dfrac{ix :: t_1 \qquad ix :: t_2}{ix :: t_1, t_2}$
(Ex) $\dfrac{ix :: (p : t; t_1), t_2}{ix :: (p : dx; t_1), t_2 \qquad dx :: t}$	(Subst) $\dfrac{ix_1 :: \bot \qquad ix_2 :: t[ix_1]}{ix_2 :: t[\bot]}$
(False) $\dfrac{ix :: t}{ix :: \bot}$, if $t \gg \bot$	(Perm) $\dfrac{ix_1 :: ix_2, t}{ix_2 :: ix_1, t}$
$c, \sim c \in CN$, $t, t_1, \ldots, t_n \in T_{\Re}$, $p \in$ Attr, $dx \in DX$, $ix, ix_1, ix_2 \in IX$, dx in (Ex) is a new temporary name	

NCC is based on the resolution principle, though the tableau algorithms [1][9] are also definitely applicable to naming constraints. We believe that resolution reasoning is quite appropriate for knowledge handling, though there are serious obstacles like undecidability of the general resolution scheme. In [14] it was shown encouragingly enough that Vampire (a well known general purpose reasoning system based on the resolution principle) [12] managed to successfully solve difficult classification problems in real-life ontologies.

To escape additional boring transformations we do not distinguish in NCC $\bot$ and $(p : \bot)$, $\bot$ and $(\bot, t)$, t and $(\bot; t)$.

The basic rules of NCC are (Res) and (Any), which handle concepts and rôles (attributes), respectively. Both are based on the general resolution scheme. (Res) introduces resolution on concepts. (Any) is an analogue of resolution working with rôles. Unlike resolution, in (Any) all subformulas survive. This is because the current version of NCC can not handle contents of attributes and thus can not prepare necessary contrary pairs. So let us consider a particular case of (Any) to demonstrate similarities with resolution:

$$\text{(Any1)}\ \frac{ix_1 :: (p : c; t_1), (p * {\sim}c; t_2)}{ix_1 :: t_1; t_2}$$

First we convert the premise of (Any1) into an equivalent FOL formula:

$$(\exists y.(p(ix_1, y) \wedge c(y)) \vee t_1(ix_1)) \wedge (\forall z.(p(ix_1, z) \rightarrow \neg c(z)) \vee t_2(ix_1))$$

in which the concepts c, t_1 and t_2 are represented by unary predicate symbols (suppose for a moment that t_1 and t_2 are atomic), and the rôle p is established by the binary predicate symbol. After Scolemization and transformations we have three clauses

$$p(ix_1, f(ix_1)) \vee t_1(ix_1) \text{ and } c(f(ix_1)) \vee t_1(ix_1) \text{ and } \neg p(x, z) \vee \neg c(z) \vee t_2(ix_1)$$

where the term $f(ix_1)$ represents scolemized y. We apply binary resolution twice and then factoring to get desired $t_1(ix_1) \vee t_2(ix_1)$, which has the same meaning as the conclusion of (Any1) $ix_1 :: t_1; t_2$. Thus, (Any1) is a 'package' of resolution applications oriented on handling attributes.

The resolution principle is specialized in NCC for two reasons. First, NCC–rules are oriented to reflect specific features of concept and rôle relations. Second, we make entailments more compact and sensitive to specific strategies in knowledge domains. We do not need the general resolution scheme because we handle only a very restricted class of formulas. Also it is forbidden to apply the resolution rules to axioms beyond constraints. We are trying to carefully restrict and polish the wild and undirected nature of the resolution principle in order to find a trade-off between completeness, decidability and efficiency.

Now let us consider the other rules of NCC. (Ax) is intended for application of axioms. In DLs axiom application has great impact, since careless use of axioms is able to bring about a combinatorial blow-up. For instance, the early versions of tableau algorithms applied all axioms to any newly generated element, and that resulted in combinatorially hopeless situations. The most efficient DL-solvers [7][6] use special techniques (such as from [9]), which allow substantial improvements. We do not consider here strategies of application of (Ax), though undoubtedly, this is the one of the most interesting problems. The simplest heuristics we are ready to suggest here

is to apply (Ax) only to those axioms, which can be immediately involved in application of one of the resolution rules, though sometimes it can bring incompleteness.

(Ama) amalgamates two naming constraints of the same object. (Ex) pushes attribute values on the outer level for further manipulations. It can be informally described as 'suppose, there exists dx such that t'. (Subst) handles the situation when some object does not exist. Then its occurrences in other constraints can be substituted by $\bot$. (Id) and (False) catch inconsistency. (Id) supports uniqueness of names from *ID*. (False) catches false within description terms. (Perm) handles names describing the same object (in cases when (Id) is not applicable).

A constraint store $\mathcal{CS}$ is a finite set of naming constraints and axioms. In our system constraint stores play the role of knowledge bases.

Definition 6. *I is an interpretation of a constraint store $\mathcal{CS}$ in a knowledge domain Δ, if for any axiom $a \in \mathcal{CS}$, $a^I = \Delta$ and for any naming constraint $ix :: t \in \mathcal{CS}$, $ix^I \in t^I$. $\mathcal{CS}$ is consistent if there exist its interpretation, and inconsistent otherwise.*

A NCC–sequence is a sequence of constraint stores $\mathcal{CS}_1, \ldots, \mathcal{CS}_n$, such that $\mathcal{CS}_{i+1} = \mathcal{CS}_i \cup \{C\}$, where C is obtained from elements of $\mathcal{CS}_i$ by application of some NCC-rule. We say that a naming constraint C is entailed from a constraint store $\mathcal{CS}$ ($\mathcal{CS} \vdash C$), if there exists a NCC–sequence $\mathcal{CS}, \mathcal{CS}_1, \ldots, \mathcal{CS}_n$, such that $C \in \mathcal{CS}_n$. The name $ix \in IX$ is *initial* in a NCC–sequence $\mathcal{CS}, \mathcal{CS}_1, \ldots, \mathcal{CS}_n$ if $ix :: t \in \mathcal{CS}$ for some t. Then $ix :: t$ is called *a premise*.

Proposition 4. *If $\mathcal{CS} \vdash ix :: \bot$ for some initial ix, then $\mathcal{CS}$ is inconsistent.*

Proposition 4 justifies the refutation procedure based on NCC.

In the end of this section we return to the definition of approximation and augment it with extra rules concerning the context, in which approximation is performed. For instance, if we check $ix \gg person$ and $ix :: person \in \mathcal{CS}$, then we can use this naming constraint to prove approximation. But such name unfolding must be done with care because if we have $person \gg ix$ and $ix :: person \in \mathcal{CS}$ then unfolding is dangerous. Later $ix :: gardener$ may appear in $\mathcal{CS}$, and after (Ama) we have false $person \gg person, gardener$. This problem arises due to incompleteness of information saved in naming constraints. Entailment in NCC is monotone and information on an object ix can be only augmented. Thus we can safely apply unfolding in the left argument of approximation. But unfolding in the right argument is unsafe. So, we have:

7. $ix \gg t$, if $ix :: t_1 \in \mathcal{CS}$ and $t_1 \gg t$.

In particular, in an application of (LPd) during examination of $t_i \gg t'_i$, only name occurrences in the actual parameter t_i can be unfolded. And this is natural because t'_i plays the role of a pattern. To preserve soundness we also have to impose some restrictions on the application of the substitution Θ in (LPd).

Unfolding can bring problems with loops when, for instance, $tom :: hasSpouse : ann \in \mathcal{CS}$ and $ann :: hasSpouse : tom \in \mathcal{CS}$. To handle loops we use:

8. *If during examination of $ix \gg t$, $ix \gg t$ appears again in some branch, then the examination in this branch stops.*

This means that if we find a loop, but there are no other obstacles, then $ix \gg t$ is proved. This rule corresponds to the greatest fixed point semantics.

4. clp(K)

In this section we demonstrate how the approach based on description terms can be integrated into logic programming. We do this in the form of a constraint logic programming language, and therefore denote this integration as clp(K), where K stands for 'knowledge'. Starting with Prolog we

1. substitute ordinary terms by description terms, and to syntactically distinguish description terms put them in curly brackets;
2. replace the standard inference LP-rule with LPd;
3. establish the constraint store containing axioms and naming constraints, and introduce in it some propagation scheme based on NCC;
4. introduce two new built-ins: `axiom/1` (posting an axiom in the constraint store) and `::/2` (posting a naming constraint in the constraint store);
5. introduce special built-ins for retrieval and retraction of information in the constraint store (not considered here).

To illustrate behaviour of clp(K) let us consider an example from Greek mythology, which is popular in the DL community. This is the story about Oedipus, who killed his father and married his mother Iocaste. Oedipus and Iocaste had children, and Polyneikis was among them. Polyneikis also had children, among them Thersandros. It is also known that Thersandros was not a patricide.

The following clp(K) rule defines naming constraints, which describe these ancient circumstances:

```
myth :-
      iocaste :: {hasChild : oedipus, hasChild : polyneikes},
      oedipus :: {patricide, hasChild : polyneikes},
      polyneikes :: {hasChild : thersandros},
      thersandros :: {~patricide}.
```

The question we want to answer is whether Iokaste has a child that is a patricide and that himself has a child, which is not a patricide. Easy to check that it is always true, but in different models this is not the same child, because everything depends on whether Polyneikis is a patricide or not. The person we are interested in can be described by the description term `{patricide, hasChild : ~patricide}`. To prove existence of the person, we will refute the negation of this term. In correspondence with Proposition 4, this means that an inconsistent naming constraint $ix :: \bot$, where ix is initial, must be entailed from the constraint store. So, we have the following quest:

```
?- myth,iocaste::{hasChild*(~patricide;hasChild*patricide)}.
```

Table III. Solving the Greek myth problem

A	Iocaste :: hasChild : Oedipus, hasChild : Polyneikes	(*Premise*)
B	Oedipus :: patricide, hasChild : Polyneikes	(*Premise*)
C	Polyneikes :: hasChild : Thersandros	(*Premise*)
D	Thersandros :: ∼patricide	(*Premise*)
Q	Iocaste :: hasChild * (∼patricide; hasChild * patricide)	(*Quest*)
1	Iocaste :: hasChild : Oedipus, hasChild : Polyneikes, hasChild * (∼patricide; hasChild * patricide)	(*Ama, A, Q*)
2a	Iocaste :: hasChild : (Oedipus, (∼patricide; hasChild * patricide)), hasChild : Polyneikes, hasChild * (∼patricide; hasChild * patricide)	(*Any, 1*)
2b	Oedipus :: ∼patricide; hasChild * patricide	(*Ex', 2a*)
3	Polyneikes :: ∼patricide; hasChild * patricide	(*Any+Ex', 1*)
4	Oedipus :: patricide, hasChild:Polyneikes, ∼patricide; hasChild * patricide *Ama,B,2b*)	
5	Oedipus :: patricide, hasChild : Polyneikes, hasChild * patricide	(*Res,4*)
6	Polyneikes :: patricide	(*Any+Ex',5*)
7	Polyneikes :: hasChild : Thersandros, ∼patricide; hasChild * patricide, patricide	(*Ama,C,3,6*)
8	Polyneikes :: hasChild : Thersandros, hasChild * patricide, patricide	(*Res,7*)
9	Thersandros :: patricide	(*Any+Ex',8*)
10	Thersandros :: patricide, ∼patricide	(*Ama,D,9*)
11	Thersandros :: ⊥	(*Res,10, Thersandros is initial*)

The refutation sequence is established in table III. In order not to overload the proof with temporary names, we use in it the following sim-

plification of (Ex):

$$\text{(Ex')} \qquad \frac{ix :: (p : (ix_1, t); t_1), t_2}{ix :: (p : ix_1; t_1), t_2 \qquad\qquad ix_1 :: t}$$

The clp(K)–tasks like Oedipus' correspond to DL–tasks about objects (in A-box). But we can also solve in clp(K) the analogues of general terminological problems with use of axioms (T-box problems). Let us consider such a problem. We define in ALC the concept of grandparent, which is someone, whose child is a parent.

$$Grandparent \equiv \exists hasChild.Parent$$

We also define the DL-axiom of an ancient:

$$Ancient \equiv Parent \sqcup \exists hasChild.Ancient$$

The task is to prove $Grandparent \sqsubseteq Ancient$, which is a typical example of inclusion problems in DLs. It is well known that this problem is equivalent to refutation of $Grandparent \sqcap \neg Ancient$.

To solve this task in clp(K), we transform axioms into the set of description terms. There are five:

```
relatives :-
          axiom({~grandparent ; hasChild:parent}),
          axiom({grandparent ; hasChild*~parent}),
          axiom({~ancient ; parent ; hasChild:ancient}),
          axiom({~parent ; ancient}),
          axiom({ancient ; hasChild*~ancient}).
```

Then we formulate the quest:

```
?- relatives, c :: grandparent, ~ancient.
```

The refutation procedure for this quest is shown in table IV.

In the tasks about Oedipus and ancients we use logic programming means only for organization of the constraint store. But there are a lot of various types of interplay between the logic programming and knowledge representation layers. For instance, we can use the LP layer as a supervisor for the knowledge base. In the following definition the `apply_axiom` relation applies alternatives of an axiom to an element `ID`. Applies bit by bit, until it finds an alternative, which is consistent with the constraint store:

```
apply_axiom(ID, {D1; D2}) :- ID :: D1.
apply_axiom(ID, {D1; D2}) :- apply_axiom(ID, D2).
```

Table IV. Solving the inclusion problem

A	$\sim$ grandparent ; hasChild:parent	(*Axiom*)
C	grandparent ; hasChild*$\sim$ parent	(*Axiom*)
D	$\sim$ ancient ; parent ; hasChild:ancient	(*Axiom*)
F	$\sim$ parent ; ancient	(*Axiom*)
G	ancient ; hasChild*$\sim$ ancient	(*Axiom*)
Q	c :: grandparent, $\sim$ ancient	(*Quest*)
1	c :: $\sim$ grandparent ; hasChild:parent	(*Ax A*)
2	c :: grandparent, $\sim$ ancient, hasChild:parent	(*Ama, Res Q, 1*)
3a	c :: ancient ; hasChild*$\sim$ ancient	(*Ax G*)
3b	c :: grandparent, $\sim$ ancient, hasChild:parent, hasChild*$\sim$ ancient	(*Ama, Res 2, 3a*)
4a	c :: grandparent, $\sim$ ancient, hasChild: d, hasChild*$\sim$ ancient	(*Ex 3b*)
4b	d :: parent	(*Ex 3b*)
5a	d :: $\sim$ancient	(*Any,Ex' 4a*)
5b	d :: parent, $\sim$ ancient	(*Ama 4b, 5a*)
6	d :: parent, $\sim$ ancient, $\sim$ parent	(*Ax, Ama, Res F, 5b*)
7	d :: $\bot$	(*Res, False 6*)
8	c :: $\bot$	(*Subst, False 7, 4a, c is initial*)

This definition implements the exhausted search over alternatives. In particular, this simple example shows that logic programs can be used to formulate strategies for the knowledge base.

Another possibility is to use description terms solely on the LP level. The following program defines again `grandparent` and `ancient` relations, but this time with LP means:

```
grandparent({hasChild:hasChild : Z}, Z).

ancient({hasChild : Z}, Z).
ancient({hasChild : Y}, Z) :- ancient(Y, Z).
```

Here we use description terms to represent data, but do not use the constraint store. The standard LP-definition of the same relations is

```
grandparent(X, Y) :- haschild(X, Z), haschild(Z, Y).

ancient(X, Y) :- haschild(X, Y).
ancient(X, Y) :- haschild(X, Z), ancient(Y, Z).
```

Let us consider an example:

```
?- john :: {hasChild:peter, hasChild:mary}.
?- mary :: {hasChild:ann}.
?-  ancient(john, ann).
..yes
```

This example shows that since description terms can have multiple occurrences of attributes, so approximation can set choice points. But this is just an analogue of the alternative facts `haschild(john, peter)` and `haschild(john, mary)` in standard logic programs.

In the same way we can describe abstract data types, for instance, `stack`:

```
top({head: Top}, Top).
push(Stack, El, {head:El, tail:Stack}).
pop({tail:Stack}, Stack).
isempty({emptystack}).
```

Moreover, we can give the general description of the type `stack`, which in $\mathcal{ALC}$ looks as follows:

$$Stack \equiv Emptystack \sqcup (\exists Head.\top \sqcap \exists Tail.Stack)$$

Its analogue in clp(K) is:

```
dtype_stack :-
          axiom({~stack; emptystack; head:object}),
          axiom({~stack; emptystack; tail:stack}),
          axiom({stack; ~emptystack}),
          axiom({stack; head*null, tail*~stack}).
```

Here concepts `object` and `null` are the equivalents of $\top$ and $\bot$, respectively. Note that `head*null` describes exactly those objects, which do not have the attribute `head`. Basing on these axioms we can check, for instance, whether an object `X` is a stack:

```
isStack(X) :- dtype_stack, c :: {~stack, X}, !, fail.
isStack(_).
```

This definition is most likely impractical but illustrates the idea. Its first rule uses negation as failure, but it does not affect the knowledge management engine because the logic programming and knowledge representation layers are separated.

Note that the last examples show also that description terms can suggest yet another approach to integration of object oriented and logic programming styles, in which objects are inhabitants of the naming constraint store. But this is just a preliminary idea which requires careful considerations.

5. Conclusion and Future Work

In this paper we consider an approach to amalgamation of the logic programming and knowledge representation paradigms. This amalgamation is

based on handling knowledge in the form of description terms and naming constraints. A special calculus NCC is proposed that works with naming constraints. NCC includes inference rules, which are modifications and specializations of the resolution principle. Description terms allow integration of LP and KR in such a way that each style has its own layer. The underling logics of LP and KR do not affect each other, and that results in more logical freedom on each level. The amalgamation of LP and KR schemes is established in the form of clp(K), in which the knowledge base plays the role of a constraint store.

The paper gives only basic definitions and results. A lot of interesting questions are left aside. In particular, there are many questions concerning the calculus NCC. This calculus is still under development, and a trade-off between completeness, decidability and efficiency is looked for. The other interesting problem is adaptation of the standard tableau algorithm as a clp(K) propagation formalism, and analysis of NCC behaviour in comparison with tableau algorithms. There are a lot of problems with the language design including interaction between logic variables and description terms. And, of course, implementation issues are also very exciting.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The description logic handbook: theory, implementation, and applications. Cambridge University Press, 2003.
2. T. Berners-Lee, J. Hendler, O. Lassila. The Semantic Web. Scientific American, May, 2001.
3. Boley, H., Tabet, S., Wagner, G. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In Proc. Semantic Web Working Symposium (SWWS'01), 381–401. Stanford University, July/August 2001.
4. S.S. Goncharov, Yu.L.Ershov, D.I.Sviridenko. Semantic programming, Information processing, Proc. IFIP 10-th World Comput. Congress, Dublin, v.10, 1986, 1093-1100.
5. Grosof, B.N., Horrocks, I., Volz, R. Description Logic Programs: Combining Logic Programs with Description Logic. Proc. of the Twelfth International World Wide Web, May 2003, ACM, 48–57.
6. V. Haarslev and R. Moeller. RACER System Description. In Proc. of IJCAR'2001, Lecture Notes in AI, 2083, 701–705, Springer, 2001
7. I. Horrocks. Using an expressive description logic: FaCT or fiction? In A. G. Cohn, L. Schubert, and S. C. Shapiro, eds, Proceedings of KR'98, 636–647. Morgan Kaufmann Publishers, 1998.
8. I. Horrocks, B. Parsia, P. Patel-Schneider, and J. Hendler. Semantic web architecture: Stack or two towers? Principles and Practice of Semantic Web Reasoning (PPSWR 2005), Lecture Notes in Computer Science, 3703, 37-41, Springer, 2005.
9. I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, eds., Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning

(LPAR'99), 1705, Lecture Notes in Artificial Intelligence, 161–180. Springer-Verlag, 1999.
10. M. Kifer, J. de Bruijn, H. Boley, D. Fensel. A Realistic Architecture for the Semantic Web. Rules and Rule Markup Languages for the Semantic Web, First International Conference, RuleML 2005, Lecture Notes in Computer Science, 3791, 17–29, Springer, 2005.
11. A.V. Mantsivoda. Semantic programming for semantic web. Invited Talk. Proc. 9th Asian Logic Conference, August 2005, 17-21.
12. A Riazanov and A. Voronkov. The Design and Implementation of Vampire. AI Communications, 15(2-3):91–110, 2002.
13. A. Robinson and A. Voronkov, editors. Handbook of Automated Reasoning. Elsevier, 2001.
14. D. Tsarkov, A. Riazanov, S. Bechhofer, and I. Horrocks. Using Vampire to reason with OWL. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, Proc. of the 2004 International Semantic Web Conference (ISWC 2004), 3298, Lecture Notes in Computer Science, 471–485. Springer, 2004.

В. А. Липовченко, А. В. Манцивода, А. А. Малых
Логическое программирование в областях знаний

Аннотация. В работе предлагается подход, интегрирующий парадигмы логического программирования и представления знаний. Этот подход базируется на концепции дескриптивных термов. Логическое программирование и представление знаний объединены таким образом, что лежащие в их основе логики аккуратно разделены. Ключевая идея здесь — сдвинуть формализм представления знаний на функциональный уровень. На уровне логического программирования база знаний рассматривается как совокупность ограничений, в котором специальные работают методы распространения ограничений. Создано исчисление NCC, работающее с дескриптивными термами, которое является базовой системой вывода для распространения ограничений. На основе данного формализма формируется язык логического программирования в ограничениях, интегрирующий подходы логического программирования и систем обработки знаний.