ELSEVIER

Artificial Intelligence 138 (2002) 117–147

Artificial Intelligence

www.elsevier.com/locate/artint

Preference Logic Grammars: Fixed point semantics and application to data standardization ☆

Baoqiu Cui [a,*,1], Terrance Swift [b]

[a] *IBM Silicon Valley Lab, San Jose, CA 95123, USA*
[b] *Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, USA*

Received 15 May 2000

Abstract

The addition of preferences to normal logic programs is a convenient way to represent many aspects of default reasoning. If the derivation of an atom A_1 is preferred to that of an atom A_2, a preference rule can be defined so that A_2 is derived only if A_1 is not. Although such situations can be modelled directly using default negation, it is often easier to define preference rules than it is to add negation to the bodies of rules. As first noted by Govindarajan et al. [Proc. Internat. Conf. on Logic Programming, 1995, pp. 731–746], for certain grammars, it may be easier to disambiguate parses using preferences than by enforcing disambiguation in the grammar rules themselves. In this paper we define a general fixed-point semantics for preference logic programs based on an embedding into the well-founded semantics, and discuss its features and relation to previous preference logic semantics. We then study how preference logic grammars are used in *data standardization*, the commercially important process of extracting useful information from poorly structured textual data. This process includes correcting misspellings and truncations that occur in data, extraction of relevant information via parsing, and correcting inconsistencies in the extracted information. The declarativity of Prolog offers natural advantages for data standardization, and a commercial standardizer has been implemented using Prolog. However, we show that the use of preference logic grammars allow construction of a much more powerful and declarative commercial standardizer, and discuss in detail how the use of the non-monotonic construct of preferences leads to improved commercial software.
© 2002 Elsevier Science B.V. All rights reserved.

☆ A preliminary version of this article appeared in Proceedings of the 1999 International Conference on Logic Programming and Non-Monotonic Reasoning, by Baoqiu Cui, Terrance Swift and David S. Warren. This paper does not reflect the official positions of the US Defense Logistics Agency or the US Customs Service.

* Corresponding author.
E-mail addresses: bcui@us.ibm.com (B. Cui), tswift@cs.sunysb.edu (T. Swift).

[1] The work presented here was largely done while the author was affiliated with the Computer Science Department, SUNY at Stony Brook.

0004-3702/02/$ – see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0004-3702(02)00185-6

Keywords: Logic Programming; Non-monotonic reasoning; Reasoning with preferences; XSB; Natural language processing

1. Introduction

Horn clauses have proven remarkably useful for parsing when their syntactic variant, *Definite Clause Grammars* (*DCGs*) [9], are employed. DCGs are commonly used to construct LL parsers in Prolog—but in Prologs that include tabling, such as XSB [19] or YAP [14], DCGs can also implement the more powerful class of LR parsers. Even LR parsers, however, can prove cumbersome for implementing grammars that contain ambiguities. While LR grammars can be written to deterministically parse potential ambiguities, the determinism comes at a cost of the conciseness of the grammar. This problem is especially important for natural language applications where ambiguities often occur and which may require a high degree of maintenance when a grammar written for one corpus of text is re-applied to a new corpus.

As proposed in [6,7] a natural way to resolve grammatical ambiguities is to declare preferences for one parse over another by adding *preference clauses* to DCGs. The resulting framework is called *Preference Logic Grammars* (*PLGs*), which can be taken as a grammatical form of Preference Logic Programs (PLPs). The following example from [7] illustrates their use in a simple example from programming languages.

Example 1.1. An imperative programming language containing an if-then-else construct must specify how to disambiguate "dangling-else" statements. In other words, it should determine whether the sequence of tokens:

if cond$_1$ then if cond$_2$ then assign$_1$ else assign$_2$

should be parsed as

if cond$_1$ then (if cond$_2$ then assign$_1$ else assign$_2$)

or as

if cond$_1$ then (if cond$_2$ then assign$_1$) else assign$_2$

The following grammar fragment is ambiguous, and gives both possible parses (see, e.g., [16] for an introduction to DCGs).

```
ifstmt(if(C,T)) -->
       [if],cond(C),[then],stmtseq(T).
ifstmt(if(C,T,E)) -->
       [if],cond(C),[then],stmtseq(T),
       [else],stmtseq(E).
```

This grammar can be disambiguated via the following *preference rule*, which indicates that the first argument of the head is preferred to the second—i.e., that the "else-statement" is to be associated with the nearest conditional statement:

```
    prefer(ifstmt(if(C,if(C1,T,E))),
           ifstmt(if(C,if(C1,T),E))).
```

When a Preference Logic Grammar is evaluated several possible parses are constructed through its grammar rules; however only the preferred parses, as specified by clauses of *prefer/2*, are retained. From a practical point of view, adding preferences to DCGs (or Horn clauses) can have a striking benefit for an important commercial application of natural language analysis called *data standardization* [18]. The problem of data standardization is to extract meaningful, standardized information from formatted textual strings. For instance, data standardization might seek to extract street address or telephone information from unstructured strings contained in a relational database, EDI field or XML page.

Example 1.2. To take a simple but concrete example of data standardization, a relational database may contain the following (misspelled) textual string:[2]

```
TO THE ORDR OF ZZZ AUTOPARTS INC 129 WASHING
TON EL SEGUNDO
```

A *name and address standardizer* might extract the company name, address, city and postal zip code all in a standard format, as the following record indicates:

Name:	'ZZZ AUTOPARTS'
Title:	'INC'
OrganizationFlag:	yes
Street:	'129 WASHINGTON'
POBOX:	
City:	'EL SEGUNDO'
State:	'CA'
Zip:	'90245'

Data standardization thus relies on parsing to extract the company name, street number and so on from the string; on techniques to infer missing information to provide the proper zip code for the string; on facilities to correct badly entered information to correct the street name; and on a detailed knowledge of organization and personal names to understand that the phrase `TO THE ORDR OF` is a preamble, and not part of a company name. At the same time, data standardization does not require techniques for understanding deep linguistic structures, and is performed over a relatively narrow semantic domain.

Since nearly any large organization must maintain data about names and addresses (of suppliers, customers, etc.), name and address standardization is of great commercial significance. Because standardization relies on rule-based reasoning both to parse and to infer missing data, standardization is a natural application area for logic programming.

[2] All examples are from commercial data, with minor changes to protect the privacy of the entities involved.

Indeed, a sophisticated Prolog standardizer was first described in [18] and has been extensively used in a variety of commercial applications (see Section 3 for details). Name and address standardization, however, can be highly ambiguous since, among other reasons, the domain of names and addresses has many proper names, and good lexicons are difficult to obtain for this domain. Because of these ambiguities, standardization is a natural application for Preference Logic Grammars in particular. In a sustained re-implementation of the Prolog standardizer mentioned above it was found that the use of Preference Logic Grammars dramatically decreased code size, and by extension maintenance cost.

The development of a commercial-grade standardizer using Preference Logic Grammars required several steps beyond the standareizer's re-implementation. First, an implementation of Preference Logic Grammars was required that was not only provably correct, but that could efficiently standardize organizational databases consisting of tens of millions of records. In addition, as Example 1.1 shows, it is convenient to allow preferences to be atom-based rather than rule-based, as parse trees are often represented as logical atoms. However, it is not straightforward how such a correct and efficient implementation can be derived using the original semantics of Preference Logic Grammars as given in [6]. This semantics is based on Kripke-like structures that, as discussed below, do not easily relate tostable model semantics or to well-founded semantics, both of which have efficient implementations. Other preference logic formalisms in the literature also have important differences from the semantics defined here (see Section 4).

Thus, before considering the application of Preference Logic Grammars, Section 2 discusses in detail a fixed-point semantics and implementation. The new semantics embeds a more general form of Preference Logic Grammars, called Preference Logic Programs, into the well-founded semantics (WFS) [20]. The new semantics offers the advantage that Preference Logic Programs can be easily transformed so that they are directly executable by a tabling engine. In addition, the new semantics allows preferences to be computed dynamically and allows unrestricted use of negation in the bodies of Preference Logic Programs. We show that even when Preference Logic Programs have no negation in their bodies, their embedding may create a non-stratified normal program, and in fact correspond to a subclass of normal program called *monophonic*. Finally, we show that the new semantics coincides with the semantics of [6] for what we term *simple* Preference Logic Programs.

Using these results on semantics and implementation, Section 3 describes the use of Preference Logic Grammars in data standardization. We discuss the problem of data standardization, describe in detail the Prolog-based standardizer and the Preference Logic Grammar standardizer that replaced it, and present detailed code comparisons. Finally, in Section 4, we review related work in preference logics and in data standardization.

1.1. Terminology

Throughout this paper, we assume the standard terminology of logic programming as can be found in, e.g. [8], with a few extensions that we note here. A program P is defined over a language $\mathcal{L}_P$ of predicate symbols, function symbols, and variables. Because Preference Logic Programs have features that resemble meta-programming, we will sometimes equate predicate and function symbols that have the same name and arity. $\mathcal{H}_P$ is the Herbrand base of $\mathcal{L}_P$. A goal to P is simply an atom in $\mathcal{L}_P$. If M is a model of a program, we denote M restricted to atoms in a set S as $M|_S$. A ground atom A_1 *depends* on a ground atom A_2 if there is a path from A_1 to A_2 in the static atom dependency graph of P as defined in, e.g., [10].

Definition 1.1. A Preference Grammar (PLG) $[P, \mathit{Pref}]$ is a set *Pref* of preference rules (*or* preferences) of the form

$$\mathit{prefer}(\mathit{Term}_1, \mathit{Term}_2) \text{ :– } \mathit{Body}$$

along with a set P of definite grammar clauses (see, e.g., [16]), such that atoms of the form *prefer/2* do not occur in the bodies of clauses in P or *Pref*. If $\mathit{prefer}(\mathit{Term}_1, \mathit{Term}_2)$ is the head of a preference rule, then an atom in $\mathcal{L}_{[P,\mathit{Pref}]}$ that unifies with Term_1 or Term_2 is a *preference atom*.

If P is a set of normal rules rather than a set of definite grammar clauses, then $[P, \mathit{Pref}]$ is called a *Preference Logic Program* (*PLP*). If P and *Pref* are both sets of definite rules $[P, \mathit{Pref}]$ is a definite PLP.

Preference Logic Grammars can be straightforwardly transformed into Preference Logic Programs in a manner similar to the way in which Definite Clause Grammars can be transformed into Definite Programs (see [7] for a formal definition of this transformation). Because of the ease of transformation we discuss both PLGs and PLPs in this paper depending on the context: PLGs are used for standardization examples while their semantics is developed using PLPs.

Both the semantics of Section 2 and its implementation are quite general, and allow the derivation of preferences to depend on preference atoms themselves. Nonetheless, it is sometimes useful to consider special cases of PLPs in which preferences are static.

Definition 1.2. Let $[P, \mathit{Pref}]$ be a PLP. A *derived atom* in $[P, \mathit{Pref}]$ is an atom that depends on a preference atom but is not a preference atom itself. A *core atom* is an atom that is neither a preference atom nor a derived atom. Accordingly, a clause whose head is a core atom is a *core clause*, and the set of core clauses in $[P, \mathit{Pref}]$ is denoted P_C. Derived clauses are defined similarly, and the set of derived clauses is denoted P_D.

Preferences in $[P, \mathit{Pref}]$ are *static* if all atoms in the bodies of rules in *Pref* are core atoms, i.e., if atoms of the form *prefer/2* are core atoms. If preferences are static, the minimal model of $(P_C \cup \mathit{Pref})$ is called the *canonical sub-model* of $[P, \mathit{Pref}]$.

Example 1.3. Let $\mathcal{P}_1 = [P_1, \mathit{Pref}_1]$ be the definite preference logic program

```
prefer(p(a),p(d)).          prefer(p(b),p(d)) :- b(1).

b(1).

p(a) :- p(d).                p(b).                p(d).
p(e) :- b(2).

d(a) :- p(a).
```

$p(a)$ is a preference atom, $b(1)$ a core atom, and $d(a)$ a derived atom. Since all atoms in the bodies of preference rules are core atoms, P_1 has static preferences. Its canonical sub-model has $\{b(1), \mathit{prefer}(p(a), p(d)), \mathit{prefer}(p(b), p(d))\}$ true, and $\{b(2)\}$ false.

The restrictions on static preferences are not uncommon when adding preferences or priorities to logical formalisms. See, e.g. [2], for a discussion. Because static preferences are part of a canonical sub-model, we can define a relation $<_{pref}$ between atoms such that $A_1 <_{pref} A_2$ if A_2 is transitively preferred to A_1 (using the relation *prefer/2* in a canonical sub-model). We say that static preferences as captured by the relation $<_{pref}$ are *well-behaved* if it is a strict partial order and is well-founded in the sense that there is no infinite chain of atoms $A_1 <_{pref} A_2 <_{pref} \cdots$.

2. A fixed-point semantics for Preference Logic Programs

As described in [6], PLPs are based on a possible-worlds semantics. While this semantics is described further in Appendix B, we present a short example of the possible worlds semantics of PLPs in order to give its flavor and to motivate the fixed-point semantics whose development is the main concern of this section.

Example 2.1. Consider the program $\mathcal{P}_2$:

```
prefer(p(1),p(2)).
prefer(p(2),p(1)).

p(1) :- r.
p(2) :- r.

r.
```

whose preferences are static. r is a core atom, while $p(1)$ and $p(2)$ are preference atoms. In [6] a canonical sub-model (Definition 1.2) of P_2 is first obtained, which contains the true atoms $\{r, \mathit{prefer}(p(1), p(2)), \mathit{prefer}(p(2), p(1))\}$. The canonical sub-model is then extended to a model, which is a Kripke-like structure $(M. \leqslant)$ in which M is a set of (2-valued) extensions of I to preference and derived atoms. For $W_1, W_2 \in M$, $W_1 \leqslant_{sp} W_2$ iff there is an atom A_1 in W_1 and an atom A_2 in W_2 such that $A_1 <_{pref} A_2$, and a world W is strongly optimal if there is no world W' different than W such that $W <_{sp} W'$ (see Definition B.2).

The possible-worlds model for P_2 contains four worlds, whose true atoms are:

$$M_1 = \{\mathit{prefer}(p(1), p(2)), \mathit{prefer}(p(2), p(1)), r\},$$

$$M_2 = \{\mathit{prefer}(p(1), p(2)), \mathit{prefer}(p(2), p(1)), r, p(1)\},$$

$$M_3 = \{\mathit{prefer}(p(1), p(2)), \mathit{prefer}(p(2), p(1)), r, p(2)\},$$

$$M_4 = \{\mathit{prefer}(p(1), p(2)), \mathit{prefer}(p(2), p(1)), r, p(1), p(2)\}.$$

By the description above, it can be seen that there is a single strongly optimal world: M_1.

As can be seen from Example 2.1, the possible worlds semantics may not be amenable to practical computation since all extensions may have to be taken with respect to preference and derived atoms and these extensions compared to determine strong optimality.

In this section, we present an alternate semantics for PLPs based on an embedding into normal programs which are then evaluated under WFS. We refer to this semantics as the *normal embedding* semantics, which has the following advantages:

- Once a PLP has been transformed into a normal program, P_{norm}, it can be evaluated by a fixed-point engine for WFS such as XSB. Further, the evaluation has in the worst case a complexity quadratic in the size of P_{norm}.
- The normal embedding semantics of PLPs can be easily compared to other semantics based on the well-founded or stable semantics. In Section 2.2 we show that definite PLPs (Definition 1.1) correspond to the class of *monophonic* normal programs. The relationship of the normal embedding semantics with other preference logic formalisms based on stable models or the well-founded semantics is discussed in Section 4.
- The normal embedding semantics allows somewhat more general forms of PLPs than the possible worlds semantics. In particular, the new semantics allow dynamic preference rules and unrestricted use of (ground) negation in the bodies of all rules.
- Finally, Appendix B shows that the normal embedding semantics coincides with the possible-worlds semantics for the class of *simple* PLPs.

2.1. Embedding PLPs into well-founded normal programs

We now formally define the semantics of Preference Logic Programs via a transformation into normal logic programs. The following definition is a reformulation of an embedding first introduced in [4].

Definition 2.1. Let $[P, Pref]$ be a PLP, and A a preference atom in $[P, Pref]$. A is *potentially overridden* (*potentially preferred*) if A unifies with some A_2 (A_1) such that $prefer(A_1, A_2)$ is the head of a preference rule in $[P, Pref]$.

Next, assume that $\mathcal{L}_{[P,Pref]}$ does not contain the predicate symbols *overridden/2, pnot/2,* or *trans_prefer/2*. The *normal embedding* of $[P, Pref]$, $[P, Pref]_{norm}$, is the smallest program containing

(1) The rules r' defined as follows. Let r be a rule

$$H \;:\!-\; A_1, \ldots, A_n, not\ B_1, \ldots, not\ B_m$$

in $P \cup Pref$.

(a) If H is potentially overridden, then

$$r' = H \;:\!-\; A_1, \ldots, A_n, B'_1, \ldots, B'_m, not\ overridden(H).$$

(b) Otherwise,

$$r' = H \;:\!-\; A_1, \ldots, A_n, B'_1, \ldots, B'_m.$$

In either case, for $0 \leqslant i \leqslant m$, $B'_i = pnot(H, B_i)$ if H is potentially preferred and B_i is potentially overridden, and $B'_i = B_i$ otherwise.

(2) The *auxiliary rules*

$$overridden(A_1) \text{ :– } trans\_prefer(A_2, A_1), A_2.$$

$$trans\_prefer(A_1, A_2) \text{ :– } prefer(A_1, A_2).$$
$$trans\_prefer(A_1, A_2) \text{ :– } trans\_prefer(A_1, A_3), prefer(A_3, A_2).$$

$$pnot(A_1, A_2) \text{ :– } trans\_prefer(A_1, A_2).$$
$$pnot(A_1, A_2) \text{ :– } not\ A_2.$$

The well-founded model [20] of $[P, Pref]_{norm}$, $WFM([P, Pref]_{norm})$, is the *normal embedding model* of $[P, Pref]$.

Example 2.2. Consider the following preference program which may be taken to have the flavor of a default logic.

```
prefer(a,not_a).

b :- not not_b, a._
a :- not not_a.
not_a :- not a.
```

The preference rule *prefer*(a, *not_a*) causes a to be potentially preferred and *not_a* to be potentially overridden. Accordingly rules with head *not_a* will have the literal *not overridden*(a) added to their bodies. Likewise, since a is potentially preferred, negative literals in the bodies of rules for a may need to be rewritten to use *pnot/2*. The embedded form is:

```
prefer(a,not_a).

b :- not not_b, a.
a :- pnot(a,not_a).
not_a :- not a, not overridden(not_a).
```

along with auxiliary rules. The normal embedding model has $\{b, a, prefer(a, not\_a)\}$ true and $\{not\_a, not\_b\}$ false.

In the normal embedding preference atoms are treated as any other atoms in $\mathcal{L}_{[P,Pref]}$. If an atom A is not unifiable with an argument in the head of a preference rule, A will be neither potentially preferred nor potentially overridden, and rules for A will be unaffected. As a corollary, if the set of preference rules in a PLP $[P, Pref]$ is empty, the normal embedding will have no effect on P beyond adding the auxiliary rules. Definition 2.1 allows preferences to be dynamic in the sense that their truth-value may depend on the truth value of other parts of the program, including other preference atoms. In addition, preferences can be declared on preferences themselves.

It is immediate from Definition 2.1 that for any atom A, if $\mathit{prefer}(A, A)$ is true, then A must be either false or undefined in $\mathit{WFM}([P, \mathit{Pref}]_{\mathit{norm}})$.[3]

The main purpose for Definition 2.1 is so that PLPs (and PLGs) can be computed efficiently by an engine for the well-founded semantics without requiring grounding (except to avoid floundered negation). We note in passing that while three-valued preferences and preferences among preferences have not yet proven useful for data standardization they can be useful for psychiatric diagnosis (see Section 4).

2.2. Definite preference logic

In a PLP, $[P, \mathit{Pref}]$, if P and Pref are both definite programs then $[P, \mathit{Pref}]$ is a definite PLP. Note that by Definition 1.1, it is straightforward to transform a PLG into a definite PLP. Definite PLPs can be taken as a generalization of DCGs, and are of particular interest for grammar processing.

While Definition 2.1 provides a general semantics for PLPs, it is natural to ask whether this semantics can be simplified for PLGs that are deemed to be "typical". For instance, since PLGs can be transformed into definite PLPs, if suitable restrictions of definite PLPs are two-valued, a simpler semantics might be possible. Unfortunately, Example 2.3 shows that this is not easily done.

Example 2.3. Consider the following PLP $[P_3, \mathit{Pref}_3]$:

```
prefer(p(a),p(b)).

p(a) :- p(b).
p(b).
```

which is a definite PLP whose preferences are static (Definition 1.2) and well-behaved. Both $p(a)$ and $p(b)$, are undefined in $\mathit{WFM}([P_3, \mathit{Pref}_3]_{\mathit{norm}})$.

We now characterize definite PLPs by their relationship with normal programs.

Definition 2.2. Let P be a ground program. Then P is *monophonic* if each atom A in P is defined by a (possibly empty) set of rules, having the form

$$\begin{array}{l} A \;\text{:–}\; A_{1,1}, \ldots, A_{1,n_1}, \mathit{not}\ A_1, \ldots, \mathit{not}\ A_p \\ \vdots \qquad\qquad\qquad\qquad\qquad \vdots \\ A \;\text{:–}\; A_{m,1}, \ldots, A_{m,n_q}, \mathit{not}\ A_1, \ldots, \mathit{not}\ A_p \end{array}$$

where A_i, $A_{i,j}$ are all atoms, $m, p, q \geqslant 0$, and $(\forall k, 0 \leqslant k \leqslant q)\ n_k \geqslant 0$.

In other words, a program is monophonic if each rule defining an atom A has the same set of negative literals as all other rules defining A. Next, we define a transformation that embeds monophonic normal programs into preference logic programs.

[3] A preprocessor implementing the normal embedding, along with several examples, can be obtained from http://www.cs.sunysb.edu/~tswift.

Definition 2.3. Let P be a monophonic program, in which the predicate *prefer/2* is assumed not to occur. Then the *preferred embedding of* P, $[P_{def}, P_{Pref}]$, is defined as follows:

- Let A be an atom in P defined by the rules:

$$\begin{array}{lll} A & :- & A_{1,1}, \ldots, A_{1,n_1}, \mathit{not}\ B_1, \ldots, \mathit{not}\ B_p \\ \vdots & & \vdots \\ A & :- & A_{m,1}, \ldots, A_{m,n_q}, \mathit{not}\ B_1, \ldots, \mathit{not}\ B_p \end{array}$$

  where $m > 0$. Then P_{def} contains the rules

$$\begin{array}{lll} A & :- & A_{1,1}, \ldots, A_{1,n_1} \\ \vdots & & \vdots \\ A & :- & A_{m,1}, \ldots, A_{m,n_q} \end{array}$$

  P_{Pref} contains the rules

$$\mathit{prefer}(B_i, A), \quad 1 \leqslant i \leqslant p.$$

- Neither P_{def} nor P_{Pref} contains any other rules.

The following result provides an equivalence between definite Preference Logic Programs to their monophonic normal counterparts.

Theorem 2.1.

(1) *Let $[P_1, \mathit{Pref}]$ be a definite PLP. Then $[P_1, \mathit{Pref}]_{norm}$ is a monophonic program.*
(2) *Let P be a monophonic program. Then $[P_{def}, P_{Pref}]$ is a definite PLP, and*

$$\mathit{WFM}(P) = \mathit{WFM}([P_{def}, P_{pref}]_{norm})|_{\mathcal{A}}$$

where $\mathcal{A}$ is the set of atoms in $\mathcal{L}_P$.

The proof can be found in Appendix A.

Theorem 2.1 indicates that the normal embedding model for definite PLPs (or PLGs) need not be two-valued. This topic is pursued further in Appendix B, which shows that a class of PLPs called *simple* have a two-valued normal embedding model. However, because PLGs may not be two-valued, their complexity is not linear in the size of a grammar. More precisely, suppose $\mathcal{P} = [P, \mathit{Pref}]$ is a ground PLG. Then evaluation of a query to $\mathcal{P}$ has worst-case complexity $|P_{norm}| \times |\mathit{size}(P_{norm})|$, where $|\mathcal{P}_{norm}|$ is the number of (ground) atoms in $\mathcal{P}_{norm}$ and $|\mathit{size}(\mathcal{P}_{norm})|$ is the size of $\mathcal{P}_{norm}$. This is somewhat worse than the complexity of traditional natural language methods, such as Earley parsing, which is linear in $|\mathit{size}(P)|$, the size of the grammar and which, if the underlying grammar is transformed to Chomsky Normal Form, can be made quadratic in the number of productions of the grammar for a given input sentence $|P|$ (see, e.g., [15] for more information).

3. Data standardization and Preference Logic Grammars

We now present a detailed study of how Preference Logic Grammars have been used to improve a name and address data standardizer written in Prolog. For convenience, we refer to the Prolog standardizer that does not use preference logic grammars as the *Prolog standardizer*, and to the version rewritten to include PLGs s the *preference logic standardizer*. We begin by describing the architecture and functionality of the Prolog standardizer before turning to the preference logic standardizer.

3.1. Prolog standardizer architecture

The Prolog standardizer described in this section has for several years been central to information processing by the US Customs Service and has standardized in real-time *all* manifest records for cargo shipments arriving into the US, and *all* customs forms filed for imports into the US. Within these records, name and address standardization is performed on entities of interest, such as manufacturers, importers, and consignees. On a somewhat smaller scale, the Prolog standardizer has been used to match supplier information for the Defense Logistics Agency against Web-based catalog information; and has been used to correlate customer information collected by different departments of a large investment bank. Further details on the architecture and application of this standardizer may be found in [12,18].

As indicated in the introduction, the input of the Prolog standardizer is a textual string, while the output is a structure consisting of standardized elements. Accordingly the architecture of the Prolog standardizer consists of four stages:

- An initial *tokenization phase* which converts the free text record into a stream of tokens.
- A *bottom–up parsing phase* which corrects spelling of tokens and is responsible for grouping designated token sequences into new tokens, sometimes called *supertokens*.
- A frame-oriented *top–down parsing phase* implemented using Prolog Definite Clause Grammars.
- A final *post-processing phase* which corrects badly parsed entities and handles inconsistent or missing data.

We discuss each of the last three phases in turn.

3.1.1. The bottom–up parsing phase

The bottom–up parse is responsible for simple correction and grouping of tokens when the correction or grouping does not depend on encountering the tokens within a particular context. The bottom–up parse performs several functions:

- *Explicit Translation*. For instance, translating keywords in foreign languages such as `'AEROPORTO'` to `'AIRPORT'`;
- *Correcting Misspellings* such as correcting `'WISCONSON'` to `'WISCONSIN'`;

- *Supertokenization* of sequences of tokens. One example of this is grouping the sequence `'SALT','LAKE','CITY'` into `'SALT LAKE CITY'` a town in Utah. If these tokens were not grouped, later stages of the parser might inadvertently recognize `'LAKE CITY'`, a town in Pennsylvania, as the city field.
- *Correcting Line Breaks* such as correcting `'WASHING', | , 'TON'` to `'WASHINGTON'`, where | denotes a line-break and/or a carriage return.

Example 3.1. Continuing from Example 1.2, after undergoing tokenization, the string

```
'TO THE ORDR OF ZZZ AUTOPARTS INC 129 WASHING |
 TON EL SEGUNDO'
```

is represented internally in Prolog list syntax as

```
['TO','THE','ORDR','OF','ZZZ','AUTOPARTS','INC',
  integer('129'),'WASHING','|','TON','EL','SEGUNDO']
```

After undergoing the bottom–up parse, the list would have the form

```
['ORDER OF','ZZZ','AUTOPARTS','INC',integer('129'),
 'WASHINGTON','EL SEGUNDO']
```

where the token sequence `'TO','THE','ORDR','OF'` is transformed to `'ORDER OF'`; `'WASHING','|','TON'` to `'WASHINGTON'`; and `'EL','SEGUNDO'` to `'EL SEGUNDO'`.

The bottom–up parser is implemented as a series of list processing routines that takes the output of the raw tokenization, and successively applies grouping and correction steps.

The code for the bottom–up parsing phase consists of simple list processing routines, most of which are automatically generated by declaring keywords such as cities, provinces, and so on (see [18] for details), or by user-defined tables of transformations. Such keyword or table specification need not be done by a Prolog programmer, and in fact the Prolog standardizer consists of over 26,000 bottom–up parsing rules that have been written or generated to correct misspellings, line breaks, or to perform supertokenization. Thus, the use of a restricted bottom–up parsing phase allows the standardizer to be more cheaply maintainable, even though the transformations themselves could also be performed in the top–down parsing phase.

3.1.2. The top–down parsing phase

The top–down parsing phase is structured as a recursive-descent LL(K) parser and coded using DCGs. The goal of the top–down parser is to fill up an *entity frame* represented by the Prolog term:

$$\begin{aligned}&\mathit{frame}(\mathit{Rel\_type}, \mathit{entity}(\mathit{Name}, \mathit{Title}, \mathit{Organization\_flag}),\\ &\qquad \mathit{address}(\mathit{Room}, \mathit{Building}, \mathit{Street}, \mathit{PoBox}, \mathit{City}, \mathit{State}, \mathit{Country}, \mathit{Zip}),\\ &\qquad \mathit{Telephone}, \mathit{Attention\_name}, \mathit{Other\_frames})\end{aligned}$$

In addition to information about personal or organizational names and addresses, information is extracted about telephone numbers, faxes, and specific parties in an organization whose attention called out in the text. Textual information may contain multiple names and addresses when one company acts on behalf of, or does business as another. If more than one entity is designated in a string, the frames nest through the *Other_frames* field; the type of relation, if any, between a nested company and its "parent" is denoted in the *Rel_type* field of the nested company.

Example 3.2. Continuing from Example 3.1, the top–down parsing phase takes as input the token sequence:

```
['ORDR OF','ZZZ','AUTOPARTS','INC',integer('129'),
 'WASHINGTON','EL SEGUNDO']
```

and produces an entity frame, abstractly represented as

Name:	`['ZZZ','AUTOPARTS']`
Title:	`['INC']`
Organization Flag:	`yes`
Street:	`['129','WASHINGTON','EL SEGUNDO']`

in which empty frame elements are omitted. The top–down parsing phase correctly identifies the organization, but over-parses the street address to improperly include the token `'EL SEGUNDO'`. This mistake will be corrected in the post-processing phase.

The Prolog standardizer contains many different rules for recognizing various forms of organizations, personal names, and street addresses. In the example above, the name is parsed by a rule that checks for an optional personal or organization prefix, (such as `'ORDR OF'`, or `'MR'`), followed by a sequence of tokens, (`'ZZZ','AUTOPARTS'`). Consuming tokens for an organization name is ended by recognizing a token that begins a production for organization termination (`'INC'` in this case), or a token that begins some other grammatical element, such as an integer that may begin a street address. Similarly, in the above example the street address is parsed by a rule that recognizes an integer followed by a sequence of tokens and street terminator. A mistake occurs, however since the input string contains no street terminator (e.g., `'STREET'`, `'CALZADA'`, `'CALLE'`, and so on), nor does it contain a sequence of tokens that unambiguously belong to another frame element(s), such as a city–state pair. In other words, top–down street address parsing would have worked correctly for the sequence `integer('129'),'WASHINGTON','STREET'` or for `integer('129'),'WASHINGTON','EL SEGUNDO','CA'`.

Top–down parsing code is quite elaborate: there are 724 DCG rules with a total of 2082 lines of code in the Prolog standardizer. These rules are necessary in part to handle a wide variety of address forms for different languages: for instance, abbreviations for post office boxes are recognized for several different languages, and street addresses in different languages require radically different grammar productions. However, many rules are also necessary to handle the ambiguity of underlying grammatical elements when tokens may be misspelled or when information may be missing. While databases exist for some of these elements such as organizations and their addresses, they are difficult to obtain, and in any case are of limited use when input strings are corrupted. Accordingly, many of the DCG rules make use of contextual information to decide into which part of the frame unfamiliar tokens (such as `'ZZZ'`) or ambiguous tokens (such as `'WASHINGTON'`) should be placed. This, the parser first enters a state where unfamiliar tokens are taken to be part of an organization or personal name. Based on recognizing name terminators or initiators or new elements, it moves into other states. In Example 3.2, when the organization terminator is consumed, the parser enters a state, *address*, in which many different types of information, such as room-number, post-office box, town, state, city, zipcode and others can all be recognized. After consuming the integer `129`, the parser switches to a state in which unknown tokens are tentatively taken to be part of a street address. This switch of context to a tentative street address production may not always be the best choice, since other address elements, such as room numbers, may also begin with an integer. We call such a choice of how to interpret default or unknown tokens within the top–down parse a *recognition heuristic*. As with the improper recognition of the city `'EL SEGUNDO'` in the street address in Example 3.2, parsing mistakes that cannot be solved using recognition heuristics or by other top–down parsing means are fixed in the post-processing phase of the Prolog Standardizer.

3.1.3. The post-processing phase

Post-processing has two purposes:

- To correct mistakes in parsing arising from ambiguities in the input string, such as seen in Example 3.2, or from mistakes in the recognition heuristic.
- To add missing address elements to the entity frame when these elements can be unambiguously inferred.

Example 3.3. Given as input the frame depicted in Example 3.2, a post-processor can apply the rule that if a frame has an empty city field, and has a street address whose last token is a city name (as indicated by an underlying knowledge base of cities), then the token should be moved from the street address to the city field, producing the frame:

Name :	`['ZZZ', 'AUTOPARTS']`
Title :	`['INC']`
Organization Flag :	`yes`
Street :	`['129', 'WASHINGTON']`
City :	`['EL SEGUNDO']`

More information can be added to this latter frame. According to the knowledge base of cities, there is a single city in the United States called `'EL SEGUNDO'`, and this city has a unique 5-digit zipcode. Accordingly, the frame can be transformed to

Name:	`['ZZZ','AUTOPARTS']`
Title:	`['INC']`
Organization Flag:	`yes`
Street:	`['129','WASHINGTON']`
City:	`['EL SEGUNDO']`
State:	`['CA']`
Zip:	`['90245']`

Corresponding to the frame presented in Example 1.2.

Correcting top–down parses. The post-processor consists of a series of rules that transform parsing information in parsing frames. The rule to pull the city from the street address has the somewhat simplified form:

```
post_process_name(frame(Rel_type,
                        entity(Name,Title,
                               Organization_flag),
                        address(Room,Building,Street,
                                PoBox,City,State,
                                Country,Zip),
                        Telephone,Attention_name,
                        Other_frames),
                  frame(Rel_type,
                        entity(Name,Title,
                               Organization_flag),
                        address(Room,Building,Street,
                                PoBox,New_city,State,
                                Country,Zip),
                        Telephone,Attention_name,
                        Other_frames) ) :-
        is_empty(City),
        length(Street,Length),Length >= 3,
        last_token(Street,New_city,New_street),
        city(New_city),
        \+ street_terminator(New_city).
```

The rule first checks whether the city field of the input parse is empty. If so, it checks to see that the street field contains at least 3 tokens, then checks that the last token is a city, and finally checks to see that the city name is not also a street terminator. Curiously, according to postal records there is a town named `'STREET'` in Maryland, a `'BROADWAY'` New Jersey, and several other towns whose names are similarly ambiguous.

Inferring information and checking consistency. The post-processing phase is also responsible for applying consistency checks on city, state, country and zipcode elements, and for inferring missing data when possible. These consistency checks are based largely on the following fact bases:

- 42,000 United States cities with their states, and 5-digit zip codes;
- The 500 largest Canadian cities with their provinces;
- 10,000 additional city–country pairs.

Depending on the knowledge base used, the post-processing phase can check the validity of various locations. If the standardizer does not recognize a valid location it attempts to correct the spelling of the city name using a more aggressive algorithm than permitted in earlier stages. To take a concrete example, if the city name in the parsed output is `PITSBURG`, the zipcode is `15123`, and the country is `US`, we determine that the city corresponding to zipcode `15123` is `PITTSBURGH`. To make this transformation, the standardizer checks whether the string-edit distance is less than a predefined threshold (which is a function of the string length) and corrects the city if so. Related algorithms are used for non-US cities.

3.1.4. Discussion of the Prolog standardizer

The Prolog standardizer uses many different aspects of Prolog. The simple syntax of Prolog is useful when generating bottom–up parsing rules; Prolog's DCGs are used in the top–down parsing phase; and the declarativity of Prolog is essential to maintaining the many DCG and post-processing rules. Furthermore, the efficient loading and indexing capabilities of modern Prologs are used by the knowledge base and parsing rules. As a result, the Prolog standardizer is a successful commercial product that would have been difficult if not impossible to produce without logic programming techniques. At the same time, the Prolog standardizer has severe limitations. Recognition heuristics—such as when to recognize a street address as opposed to a room number as discussed in Section 3.1.2—may be embedded deeply in DCG or post-processing code, and these heuristics may need to be changed when moving from one textual corpus to another. Recognition heuristics lead to a large number of contexts in the top–down parser—i.e., where unknown tokens are recognized as part of a name, various address elements and so on. These contexts may depend on what has already been parsed: an unknown token may be assumed to be part of a street name if a street name has not already been parsed; otherwise the token may be part of, say, a city name. Similarly, a short integer may be taken tentatively to begin a street address if no such address has been found; otherwise it may be taken to be part of a room or suite number. This leads to a large number of contexts for the top–down parser, so that in addition to recognition heuristics being difficult to adjust for a new corpus, they must be adjusted in many places within the top–down parser, and in the post-processing phase, as in Example 3.3. The result is that maintaining the Prolog standardizer needs extended attention from a skilled Prolog programmer, making it too expensive for many commercial applications.

3.2. Architecture of the preference logic standardizer

The code complexity of the Prolog standardizer arises from the fact that many grammatical elements cannot always be parsed unambiguously. The top–down parsing phase of the Prolog standardizer is deterministic (using a certain amount of look-ahead) but the cost of this determinancy is that a large number of contexts are required as well as a significant post-processing phase. The preference logic standardizer makes use of the technique in Example 1.1: a simple, but ambiguous grammar is defined, and ambiguity is resolved using preference rules. Using this technique the preference logic standardizer effectively moves most heuristics to preference rules, leading to simpler parsing and post-processing rules. As a result, the preference logic standardizer has replaced the Prolog standardizer for new commercial applications. The preference logic standardizer has been extensively used to match supplier information for the Defense Logistics Agency against Web-based catalog information, and against data from supply depots for the various armed services. It has also been used to clean customer data for a large pharmaceutical company.

The functionality of the preference logic standardizer is essentially that of the Prolog standardizer: recognition and standardization of different forms of organizational and personal names and addresses. The architecture for the preference logic standardizer consists of code for the same four phases as the Prolog standardizer. Conceptually, the tokenization and bottom–up parsing phases are the same for the two standardizers. The final two phases of preference logic standardization are:

- *A preference logic parsing* phase, which creates sets of name and address elements, each of which sets corresponds to an entity frame.
- A *post-processing phase* which infers missing data.

In the preference logic standardizer, the post-processing phase does not correct misparsed data, but only infers missing city, state, country or zipcode information, as done in Example 3.3, or marks data for these fields as inconsistent. Thus the post-processing code for the preference logic standardizer is essentially the same as that used by the Prolog standardizer to infer missing data, and is not discussed further in this section. Rather, we discuss in detail the architecture of code for the preference logic parsing phase.

3.2.1. The preference logic parsing phase

We begin by examining at a high level how preference logic grammars are used on the running standardization example.

Example 3.4. Consider the output of the bottom–up parser as described in Example 3.1:

```
['ORDER OF','ZZZ','AUTOPARTS','INC',integer('129'),
 'WASHINGTON','EL SEGUNDO']
```

Abstractly, the preference logic parsing phase encapsulates the parsing of names and addresses via sets of name and address elements. If the parsing routines succeed with more than one set, despite the preferences defined for them, the parse is ambiguous. Two possible sets of address elements for the above token sequence are such that:

(1) the street address element contains the token sequence `integer('129')`, `'WASHINGTON','EL SEGUNDO'`, and the city element is empty;
(2) the street address element contains the token sequence `integer('129')`, `'WASHINGTON'`, and the city element contains the token `EL SEGUNDO` (which is a valid city).

However the second parse may be preferred to the first since the second contains a greater number of (valid) address elements.

Code for the preference logic parsing phase has been implemented using XSB [19]. Fig. 1 contains some relevant top-level predicates for preference logic address parsing as encoded in XSB syntax. In order to be evaluated by XSB, the code in Fig. 1 would need be pre-processed to undergo the normal embedding (Definition 2.1). When a query is made to the transformed program, the top–down execution strategy of Prolog would be used for program clause resolution, combined with a bottom–up propagation of answers and of success information of tabled negation. This control strategy is evidenced by the cut operator is used in Fig. 1. Just as pratical Prolog programs contain predicates and control operators that are outside of Horn clauses with negation, so practical preference logic programs in XSB may contain cut operators, non-tabled Prolog negation (`\+ /2`), aggregation meta-predicates such as `setof/3` and so on. In particular, the use of the cut is safe in Fig. 1 since there is no tabled predicate in the scope of the cut. Intuitively, the safety arises because the cut affects only program resolution steps for non-tabled predicates and so cannot interfere with bottom–up propagation of answers from tables.

The top-level predicate for address parsing is `address/1`; upon its success its argument is bound to a list of address elements that are preferred via the predicate `prefer/2` in Fig. 1. The predicate `address/1` calls the DCG predicate `scan_address/3`, which produces all possible address parses for the token string. `scan_address/3` is called via the tabled-DCG meta-predicate `tphrase/1`, an XSB analog to the prolog DCG meta-predicate `phrase/1`. A Prolog DCG uses a difference lists to denote the segment of a token sequence that it parses; but copying difference lists into and out of tables can be expensive when tabling is used. Because difference lists for all sub-parses may be copied into and out of a table, a parse that has linear complexity (in the length of the token sequence) for Prolog DCGs may have quadratic space and time complexity if DCGs are tabled naively. Rather than difference lists, tabled DCGs use integers that denote their position in a token sequence, and assert the token sequence itself into Prolog's store. As an example of this translation, the token sequence in Example 3.4 would cause the facts `word(0,'ORDER OF',1)` and `word(1,'ZZZ',2)`, and so on, to be asserted to XSB's store. To make use of this, `tphrase(scan_address(Addr))` would make the call `scan_address(Addr,0,X)` so that parsing would begin on the first token in the store, and upon the success of the call, `X` would be instantiated to the position of the last word used in the parse. Assertion of token sequences to the store is done through XSB library predicates, and is not depicted in Fig. 1, but detailed information of this routine can be found in [19].

The predicate `scan_address/3` is called a *scanning predicate* whose only purpose is to call the DCG predicate `address_element_set/3` on each position of the token sequence. As its name implies, this predicate produces, through backtracking, all possible sets of address elements by calls to the predicate `address_element/4`.[4] As can be seen from this predicate, an address element is simply a sequence of tokens together with the type of the element—whether it is a `street`, `pobox`, and so on. The fact that `address_element/4` is tabled is an important factor in the efficiency of the preference logic standardizer: different sets of address elements can be obtained without having to reparse each element. The algorithm implicit in Fig. 1, is strikingly different from the approach to top–down parsing discussed in Section 3.1.2. The parser does not require different contexts to determine how to interpret unknown or ambiguous tokens, nor does it requrire recognition heuristics embedded in DCG rules. It simply generates possible parses, and leaves most disambiguation to the preference rules.

The preference rules in Fig. 1 are based on general principles. The first rule states that a parses with more different types of address elements are preferred: a set $\mathcal{S}_1$ of address elements is preferred to another set, $\mathcal{S}_2$, if $\mathcal{S}_1$ contains a type of address element that $\mathcal{S}_2$ does not contain, but if the converse is not true. For example, in Example 3.4 a set that contains both a city and a street address is preferred to a set that contains a street address only. The second rule states that longer address elements are preferred: that a set $\mathcal{S}_1$ of address elements is preferred to a set $\mathcal{S}_2$ if they both contain the same types of address elements, but if $\mathcal{S}_1$ contains at least one element of type T_1 that is a super-sequence of an element of type T_1 in $\mathcal{S}_2$. For example, if $\mathcal{S}_1$ were the same except that $\mathcal{S}_1$ contained the street address element `42,'N','SHAVER'`, but $\mathcal{S}_2$ contained the street address element `42,'N'`, $\mathcal{S}_1$ would be preferred to $\mathcal{S}_2$. Of course other preference rules may be needed for a given domain, e.g., to ensure that sets of address elements do not contain two street addresses. This is not usually necessary, in part because preferences can also be used to limit the number of address elements that are parsed.

3.2.1.1. Pruning using preference logic. A simple preference relation on the predicate `preferred_city_state_zip/4` (Fig. 1) ensures that a city, state, zip triple succeeds as an address elements only if it is "optimal" over the entire token sequence. Code for this predicate is shown in Fig. 2 and resembles the creation of address sets in Fig. 1. The predicate `preferred_city_state_zip/3` is called to see if a preferred city, state, zip triple begins at a given position, *p*, in the token sequence. `preferred_city_state_zip/3` in turn calls `scan_city_state_zip/3`, which finds all preferred city, state, zip triples for *any* position in the token sequence. If some preferred city, state, zip triple in fact begins at *p*, `preferred_city_state_zip/3` succeeds, otherwise it fails. The preference rules defined on `scan_city_state_zip/3` compare only the city, state, zip triples themselves and not their positions; further they use a simple weighting function (the specification of which is not shown) to weigh the information in the triples. Note that tabling `scan_city_state_zip/3` ensures that it is executed only for the first position in which it is called; subsequent calls will read preferred triples out of the table store.

[4] In XSB version 2.4 the call to tabled negation, *tnot/1* in Fig. 1, must be replaced by a call to *'sk_not'/1* in order to execute non-ground tabled negation.

```
:- table address/1.
address(Addr):-                                     % Begin with first token
        tphrase(scan_address(Addr)).                  % of input sentence.

scan_address(Addr) -->
        address_element_set(Addr).
scan_address(Addr) -->
        [_],
        scan_address(Addr).

address_element_set([elt(Type,Elt)|Rest])-->
        address_element(Type,Elt),
        address(Rest).
address_element_set([]) --> tnot(address_element_set(_Type,_Elt)).

:- table address_element/4.
address_element(room,Rm) --> room(Rm).
address_element(building,Bld) --> building(Bld).
address_element(street,Str) --> street(Str).
address_element(pobox,PO) --> pobox(PO).
address_element(csz,Csz) --> preferred_city_state_zip(CSZ).
address_element(country,Ctry) --> country(Ctry).

prefer(address(Address1),                % Prefer parses with
       address(Address2)) :-             % more types of elemenets.
       member(elt(Type,_),Address1),\+ member(elt(Type,_),Address2),
       !,
       \+ (member(elt(Type1,_),Address2),\+ member(elt(Type1,_),
                                                  Address1)).
prefer(address(Address1),                % Prefer parses with
       address(Address2)):-              % lengthier elements.
       sort(Address1,Saddress1),
       sort(Address2,Saddress2),
       contained_in(Saddress2,Saddress1).
```

Fig. 1. Parsing an address using tabling.

3.2.1.2. DCG rules for organization names. In the preference logic standardizer, the actual DCG rules to define address elements and names are ambiguous but concise. This is most clearly seen by examining rules for parsing organization names, which are often proper names and cannot easily be informed by a lexicon. An organization parse is represented by a structure of the form *element*(*Rel_Type*, *Name*, *Terminators*). In this case, *Rel_Type* is either the token `org` or some more specific relation-type between organizations such as `'ON BEHALF OF'` or `'DOING BUSINESS AS'`. *Name* is a list of tokens representing the organization name, and *Terminators* are the list of tokens terminating the organization name, such as `'INC'`, `'CO'`, `'GMBH'`, etc. The first two clauses of `org/3` call the predicates `preferred_company/3` and `rel_company/3` and then themselves recursively. A well-structured company is one that has an explicit termination sequence, such as `'INC'` or `'AND','ASSOCIATES'`; a "subsidiary" company is a company along with a parsed relationship type such as `'ON BEHALF`

```
:- table preferred_city_state_zip/3.
preferred_city_state_zip(CSZ,Beg,End):-
        scan_city_state_zip(CSZ,B,E),
        B = Beg, E = End.

:- table scan_city_state_zip/3.
scan_city_state_zip(CSZ) -->
        city_state_zip(CSZ).
scan_city_state_zip(CSZ) -->
        [_],
        scan_city_state_zip(CSZ).

prefer(scan_city_state_zip(CSZ1,_,_),
       scan_city_state_zip(CSZ2,_,_)):-
        weigh_csz(CSZ1,W1),
        weigh_csz(CSZ2,W2),
        W1 >=  W2.
```

Fig. 2. Predicates needed to prune according to city, state, and zip code.

OF'. The last two clauses of org/3 handle cases where relation types and organization terminators may be omitted. They allow the parse to read to a line break (represented by the pipe symbol) or beyond, if necessary. In a manner similar to preference rules for addresses, preference rules for organizations prefer organizations with more specific information, that is with longer lists of organization terminators and with specific relation types to those with shorter lists of organization terminators and less specific organization types (i.e., with the relation type of org). Personal names are parsed with a separate set of rules. Personal names may be well-structured—(beginning with 'MR', 'MS', etc., or ending with, e.g., 'PHD')—or not. Preference rules then are used to determine whether a name belongs to that of an organization or a person: well-structured names (or subsidiary companies) are preferred to non-well structured names. On the other hand, deciding whether a name that is not well-structured belongs to a person or to an organization by default is corpus-dependant and is also performed by preference rules.

Once preferred names and preferred sets of address elements are obtained, they are combined into entity frames, representing parses. Note that the third clause of org/3 can cause a great deal of ambiguity if well-structured companies, or subsidiary companies are not parsed. In this case, preference relations on entity frames together with the preference relations on addresses ensure that only those name parses that combine with preferred address parses are retained in preferred entity frames. Because of the several levels of preference rules, experience shows that the preference logic standardizer generally gives a single parse for names and addresses. In a small percentage of cases (usually less than 1%), ambiguous parses are derived and the results sent to an error file. We note that the ability of the Preference Logic Standardizer to recognize cases where it parses improperly is an advantage over that of the Prolog Standardizer which does not have this ability.

```
:- table org/3.
org([Elem|R]) -->
     well_structured_company(Elem),
     org(R).
org([Elem|Others]) -->
     rel_company(Elem),
     org(Others).
org([elt(org,company(Elem,[]))]) -->
     any_words(Elem).
```

Fig. 3. Ambiguous grammar rules for organization names.

3.2.1.3. Discussion. To summarize, preference rules are used in several places in the name address standardizer:

(1) to determine preferred organization names (defined on the relation `org/3` shown in Fig. 3);
(2) to determine preferred personal names (as discussed above);
(3) to determine preferred names overall by combining preferred organization names and preferred personal names;
(4) to determine preferred address elements (currently only used for `preferred_city_state_zip/3`, shown in Fig. 2);
(5) to determine preferred sets of address elements (shown in Fig. 1);
(6) to determine preferred entity frames (as discussed above).

As will be substantiated in Section 3.3 the address parsing of Fig. 1 leads to much more concise code than used by the DCG-parser of Section 3.1. General DCG rules, along the lines of code in Fig. 3 are written to parse names and address elements, and preferences are declared at various levels. As a result, when migrating from one corpus to another, changes to the code can be minimized, and sometimes limited to preference rules themselves. The ability to factor out heuristics into preference rules makes code maintenance simpler than would be possible if programming were simply done in the well-founded semantics using the normal embedding.

However, standardizer development requires not only a knowledge of Prolog and preference logic, but also knowledge of several specifics of XSB. While tabled DCGs are fairly simple to use in XSB, they have certain differences with Prolog DCGs, in particular since tabled DCGs must have an input sentance asserted to Prolog store. Code must also undergo a preprocessing phase in order to be executable, and preprocessed code (as well as tabled code) can be difficult to trace at times. Furthermore, as evidenced in Fig. 1, the interplay between cuts and tables may need to be understood in XSB in order to program effectively in preference logic. There are other specifics of XSB and its evaluation method required by the standardizer. Of course, all such specifics are documented [19], but learning specifics of XSB, (or of other research-based programming systems) requires a degree of intellectual commitment by the programmer.

Table 1
Code sizes for prolog and XSB standardizers

Function	Clauses	Lines
Tokenization	94	412
Bottom–up parse	26205	26205
Domain information	59150	59150
Control and utilities	727	1345
(Prolog) Top–down	724	2082
(Prolog) Post-processing	604	2838
(PLP) Top–down	198	686
(PLP) Post-processing	7	106

Table 2
Performance of various standardizers

	Prolog Stdzr	PLP Stdzr (no pruning)	PLP Stdzr (pruning)
Records per second	54	14	19

3.3. Comparison of the two standardizers

Table 1 provides insight into the amount of code in each standardizer. Clearly, most code comprises domain information, mostly tables of cities, states, zip codes, countries, and so on; along with rules for the bottom–up parse, which as mentioned in Section 3.1.1 is largely automatically generated based on declarations of keywords. The most elaborate code is in the top–down parse and in the post-processing: each of these sections of code is reduced. Indeed, the post-processing step almost eliminated, consisting only of consistency checks for city, state, zipcode triples in the preference logic standardizer. Thus, while using the new standardizer architecture does not lead to a large reduction in overall standardizer code, it greatly reduces the amount of code needed by later phases of standardization—the code that requires the most programmer maintenance.

Testing on Defense Department data indicates that the PLP standardizer works correctly on about 96–97% of the time, a rate that is virtually identical to the Prolog standardizer.[5] We note that the two standardizers differ slightly in their functionality so that the numbers in each table, should be taken as approximate comparisons. Even with this disclaimer, it can be seen that the PLP standardizer drastically reduces code in the top–down parsing and post processing stages. This is due both to the simple, ambiguous grammatical forms that tabling allows (as illustrated by code in Fig. 3) and to the declarative use of preference rules that are combined with the grammar rather than applied after the entire string has been parsed. Table 2 indicates the performance of the various standardizers in terms of records per second standardized on a PC. While the PLP standardizer is 3 times slower than the Prolog standardizer, the tradeoff of speed for declarativity is beneficial for this

[5] Verification is performed by human analysis of a random sample of data.

application since the costs of maintenance by far outweigh the performance costs as long as the performance costs remain reasonable.

4. Related work

Semantics. The semantics for preferences presented in Section 2 is distinct from semantics for preferences presented previously in the literature. One distinction is that the approach in this paper is atom-based rather than rule-based as with [1–3,5]. While simple transformations can convert atom-based preference formalisms to rule-based formalisms and back, we feel that an atom-based formalism is more natural to specify preference grammars, as atoms directly represent schemas for parse trees. Furthermore, an atom-based formalism avoids ambiguities that can arise when distinct but unifiable non-ground rules are given different priorities (see [2] for a full discussion). Of perhaps greater importance is that the rule-based formalisms [2,5] and the atom-based formalism of [13] are all based on a semantics for stable models or answer sets rather than the well-founded semantics, a point to which we now turn.

Theorem 2.1 indicates that definite PLPs—or PLGs—correspond to monophonic normal programs, so that a semantics for PLGs under the normal embedding of Definition 2.1 must incorporte non-stratified negation. In our opinion, this implies that the semantics should be based either on the well-founded semantics or stable model semantics (or their extensions). Well-founded semantics appears preferable for this application not only because of its lower worst-case asymptotic complexity, but because the program does not need to be grounded for each input string in order to be evaluated. The approach of [1] is based on the well-founded semantics, but its computation is based on a fixed-point operator distinct from that used to compute the well-founded semantics, while our semantics is based on a direct transformation. Nonetheless, we believe the major contribution of this paper lies not in the form of preference logic used, but in the efficient implementation of the preference logic that we have described and in the documentation of its use in a commercial standardizer.

Applications. Gartner et al. [4] discuss the use of preference logic for modelling psychiatric diagnoses according to the American Psychiatric Association's *Diagnostic and Statistical Manual of Mental Disorders, version 4* (*DSM-IV*). We describe in a simplified manner the features of preference logic required for psychiatric diagnosis in order to compare this application to data standardization. First, as described in [4], modelling DSM-IV uses a preference logic program $[P, \mathit{Pref}]$ in which P is non-stratified. This arises from the possibility that a patient may have symptoms such that DSM-IV rules reduce to cases of mutual exclusion. That is, DSM-IV rules may be "reduced" to cases such as:

```
aspergers_disorder :- not autism.
autism :- not aspergers_disorder
```

Under the well-founded semantics it is unknown whether a patient may have *autism* or *aspergers disorder*, so that undefined truth values indicate that information in DSM-IV

Table 3
Comparison of two preference logic applications

Feature	Standardization	Diagnosis
Definite PLP	Yes	No
Static preferences	Yes	No
Well-behaved preferences	Yes	No
Ground source program	No	Yes

is not sufficient for diagnosis. In such a situation, a diagnostician may add preferences to indicate cases in which one of the conflicting diagnoses may be preferred to another, so that the preferences allow a diagnostician to configure DSM-IV for her own purposes without altering the modelling of DSM-IV. [4] also provides examples in which dynamic preferences and preferences that are non well-behaved (Section 1.1) are needed. Table 3 summarizes these differences. The table also indicates that the application of psychiatric diagnosis is simpler than that of standardization in that the source preference logic program can easily be grounded, unlike our standardization examples.

The name and address standardizer presented here is not unique in the commercial world. The most widely used address standardizer is written by the US Postal Service and does not use logic programming techniques. Comparisons between the Postal standardizer and the standardizer of [18] indicate complimentary strengths: the Prolog standardizer is much better at extracting addresses from free text, at parsing the various address components, and at handling foreign addresses. Because it works off of a more complete knowledge base, the Postal standardizer is better at correcting address data once the address components have been identified (e.g., the street, post-office box and city identified). Indeed, these two standardizers have worked together to commercial advantage. Name and address standardization thus provides an example of an important commercial problem for which logic programming techniques offer significant advantages over other existing methods. It should also be noted that the domain of names and addresses is only one domain for which data standardization has commercial importance, and standardizers have been written for other domains, including aircraft part information, transportation records, and cargo descriptions. There are no other known standardizers for these domains.

5. Conclusions

We have described a simple logic for preferences, its efficient implementation, and its successful application to a commercial problem. We believe that such implementation and application efforts are important both for logic programming and non-monotonic reasoning. Commercial organizations are often reluctant to use Prolog for program development, let alone extensions of Prolog that include preferences or other uncommon techniques for knowledge representation. We believe that it is only by developing efficient implementations of these techniques that their research and commercial applications can be discovered and tested—and that it is through such applications that the significance of the knowledge representation techniques will ultimately be judged.

The use of tabling as an implementation platform to combine knowledge representation techniques and parsing formalisms has implications beyond those described in the paper. It is well-known that a number of parsing formalisms besides DCGs can be easily viewed as systems of logical deduction. These include newer formalisms such as tagged-attribute grammars and categorical grammars, that can be implemented using tabled logic programming (see [15] for a survey of these formalisms and their potential implementations). The technique of tabling has also been used to implement a variety of non-monotonic logics in addition to the preference logics described here (see, e.g., [17]). Other combinations of non-monotonic formalisms and parsing techniques may prove fruitful for commercial applications beyond data standardization, such as natural language query evaluation, or text retrieval. Formulating these combinations and testing out their practical usefulness remains an intriguing unexplored area.

Acknowledgements

The authors owe David S. Warren a debt of thanks for help in rewriting the Prolog standardizer to use tabling and preferences. The authors would like to thank Bharat Jayaraman and Kannan Govindarajan for their comments on a prelimiary version of this paper. This work was partially supported by NSF grants CCR-9702581, EIA-97-5998, and INT-96-00598.

Appendix A. Proofs of theorems

Theorem A.1.

(1) *Let $[P_1, Pref]$ be a definite PLP. Then $[P_1, Pref]_{norm}$ is a monophonic program.*
(2) *Let P be a monophonic program. Then $[P_{def}, P_{Pref}]$ is a definite PLP, and*

$$WFM(P) = WFM([P_{def}, P_{pref}]_{norm})|_{\mathcal{A}}$$

where $\mathcal{A}$ is the set of atoms in $\mathcal{L}_P$.

Proof. (1) Immediate from Definitions 2.1 and 2.2.

(2) The first part of the theorem, that if P is monophonic then $[P_{def}, P_{Pref}]$ is a definite preference program is straightforward from Definitions 2.2 and 2.3.

To see the second part, consider the rules defining an atom A in P:

$$\begin{array}{l} A \;:\!-\; A_{1,1}, \ldots, A_{1,n_1}, \mathit{not}\ B_1, \ldots, \mathit{not}\ B_p \\ \vdots \\ A \;:\!-\; A_{m,1}, \ldots, A_{m,n_q}, \mathit{not}\ B_1, \ldots, \mathit{not}\ B_p \end{array}$$

From Definition 2.3, $[P_{def}, P_{Pref}]$ will have rules of the form

$$\begin{array}{l} A \;:\!-\; A_{1,1}, \ldots, A_{1,n_1} \\ \vdots \\ A \;:\!-\; A_{m,1}, \ldots, A_{m,n_q} \end{array}$$

and

$$\mathit{prefer}(B_i, A), \quad \forall i,\ 1 \leqslant i \leqslant p.$$

Next, $[P_{def}, P_{Pref}]_{norm}$ will have the rules

$$\begin{aligned} &A \ :\!-\ A_{1,1}, \ldots, A_{1,n_1}, \mathit{not\ overridden}(A)\\ &\vdots\\ &A \ :\!-\ A_{m,1}, \ldots, A_{m,n_q}, \mathit{not\ overridden}(A)\end{aligned}$$

where *overridden/1* is defined as in Definition 2.1. It is now straightforward to see that *overridden*(A) is true iff the conjunction $not\ A_1, \ldots, not\ A_p$ is true. Thus, for atoms in $\mathcal{L}_P$, the well-founded models of the two program must be the same. □

Appendix B. Simple Preference Logic Programs

The normal embedding semantics of Section 2 provides a convenient fixed point semantics for PLPs in terms of well-understood normal programs. A different approach, however, was originally taken to define Preference Logic Grammars in [6]. First, in [6] PLPs have *static* preferences (Definition 1.2); second, a possible worlds semantics was provided for these PLPs. In this section we review relevant aspects of the possible worlds semantics of [6] and discuss its relation to our fixed-point semantics. We show that the two semantics coincide for the class of *simple* PLPs.[6]

We begin by defining the concept of a *world*. Suppose a PLP, $[P, \mathit{Pref}]$ is definite and has well-behaved preferences. In this case, the canonical sub-model of $[P, \mathit{Pref}]$ (Definition 1.2), together with a set of (true) preference atoms determines a model for $[P, \mathit{Pref}]$. This model can be constructed via the least fixed point of the inference operator T_{P_D} starting from the canonical sub-model of $[P, \mathit{Pref}]$ and the preference atoms. Accordingly, Definition B.1 defines a world of $[P, \mathit{Pref}]$ using two-valued interpretations on preference atoms.

Definition B.1. Let $[P, \mathit{Pref}]$ be a definite *PLP* whose preferences are static, and whose canonical sub-model is I. A two-valued interpretration of preference atoms in P whose true atoms are a subset of the true atoms in the minimal model of P is called a *world*.

Example B.1. In Example 1.3, there are 8 possible worlds, whose true atoms are subsets of $\{p(a), p(b), p(d)\}$. Any interpretation for which $p(e)$ is true is not a world, since $p(e)$ is not true in the minimal model of $\mathcal{P}_1$. Note that the truth of $d(a)$ can be computed based on whether $p(a)$ is true in world or not.

The possible worlds semantics of preference logic programs is based on *strongly optimal worlds*.

[6] Some of the results in this section were presented in a very preliminary form in [4].

Definition B.2. Let $[P, \mathit{Pref}]$ be a definite *PLP* whose preferences are static. A world W_1 is *reflexively preferred* to a world W_2 (denoted $W_2 \leqslant_{sp} W_1$) if there exists a $A_1 \in W_1$, $A_2 \in W_2$ such that $A_2 \leqslant_{sp} A_1$. A world W is *strongly optimal* if for any other world $\mathcal{W}_{pref}$, $W \leqslant_{sp} W_{pref} \Rightarrow W = W_{pref}$.

We now turn to the *optimal subproblem property* of a PLPs. Intuitively a PLP $[P, \mathit{Pref}]$ has this property if the derivation of an optimal preference atom (a preference atom that is maximal with respect to $<_{pref}$) in the minimal model of P does not require non-optimal preference atoms. The program in Example 1.3 does not have the optimal sub-problem property since $p(a)$ is an optimal preferenc atom in the minimal model of P_1, but $p(d)$ is required to derive $p(a)$, and $p(d)$ is not optimal since both $p(b)$ and $p(a)$ are preferred to $p(d)$.

The definition of the optimal subproblem property relies on the notion of a world being supported. A supported world W is analogous to a supported interpretation in logic programming. Its definition below is complicated by the fact that derivation of preference atoms in a world may depend on the truth of derived atoms which in turn may depend on the set of preference atoms in W. A world is first extended to an interpretation I_W of $[P, \mathit{Pref}]$ by unioning to W the least fixed point, starting from W, of the inference operator, $T_{P_C \cup P_D}$, of the derived and core atoms. Next, it is determined whether the preference atoms in W are supported in I_W. Formally:

Definition B.3. Let W be a world for a definite PLP $[P, \mathit{Pref}]$. W is *supported* if for

$$I_W = \mathit{least\_fixed\_point}\big(T_{P_C \cup P_D}(W)\big) \cup W,$$
$$W = T_P(I_W)|_{pref\_atoms},$$

where *pref_atoms* are the set of preference atoms in $T_P(I_W)$.

Let I be an interpretation for $[P, \mathit{Pref}]$. The reduction of I, I_{red}, is obtained by restricting I to preference atoms in $[P, \mathit{Pref}]$, and setting to false all and only those preference atoms $A \in I$ such that $\mathit{prefer}(A_1, A)$ is true in I.

Let M be the minimal model for $(P \cup \mathit{Pref})$ considered as a definite program. A program $[P, \mathit{Pref}]$ has the *optimal subproblem property* if M_{red} is supported.

A definite PLP whose preferences are static and well-behaved and that has the optimal subproblem property is called *simple*.

Example B.2. The minimal model, M, of $\mathcal{P}_1$, considered as a definite program has

$$\big\{\mathit{prefer}\big(p(a), p(d)\big), \mathit{prefer}\big(p(b), p(d)\big), b(1), p(a), p(b), p(d)\big\}$$

true and $\{b(2), p(e)\}$ false. M_{red} has true atoms $\{p(a), p(b)\}$. The interpretation

$$I_{M_{red}} = \mathit{least\_fixed\_point}\big(T_{P_C \cup P_D}(M_{red})\big) \cup M_{red}$$

is also two-valued with true atoms

$$\big\{\mathit{prefer}\big(p(a), p(d)\big), \mathit{prefer}\big(p(b), p(d)\big), b(1), p(a), p(b)\big\}.$$

$T_P(I_{M_{red}})|_{pref\_atoms}$ has true atoms $\{p(b)\}$ capturing the fact that the derivation of $p(a)$ requires a non-optimal atom, $p(d)$, so that $\mathcal{P}_1$ does not have the optimal sub-problem property.

The coincidence of the normal embedding semantics and the possible worlds semantics is summarized by the following theorem.

Theorem B.1. *Let $[P, Pref]$ be a simple PLP, and $[P, Pref]_{norm}$ be the normal embedding of P. Then*

(1) *there is a unique strongly optimal world, $\mathcal{W}$ for $[P, Pref]$;*
(2) *$WFM([P, Pref]_{norm})$ is two-valued;*
(3) *A is true in $WFM([P, Pref]_{norm})$ iff $A \in \mathcal{W}$.*

Proof. (1) Consider the reduction, M_{red}, of the minimal model M_{min} of $(P \cup Pref)$. Clearly by Definition B.1, M_{red} is a world. M_{red} must also be strongly optimal, for suppose there were some other world W such that $M_{red} \leqslant_{sp} W$. Then $\exists A_W \in W, \exists A_M \in M_{red}$ such that $A_M <_{pref} A_W$. It is not the case that $A_W \in M_{red}$ since $A_M \in M_{red}$, and $A_M <_{pref} A_W$ contradicts tha fact that M_{red} is reduced. A_W must be in M_{min} for W to be a world. However, $A_W \notin (M_{min} - M_{red})$. To see this, assume the opposite and note that, since $A_W \notin M_{red}$, there must be some $A \in M_{min}$ such that $A_W <_{pref} A$. Since preferences are well-behaved in $[P, Pref]$, it cannot be the case that $A <_{pref} A$ so that by the definition of reduction, M_{red} must contain A, or some atom preferred to A, contradicting the assumption that $A_W \notin (M_{min} - M_{red})$. Thus, such an A_W cannot exist and M_{red} must be strongly optimal.

The argument that M_{red} is unique is as follows. For there to be a strongly preferred world W different from M, it must be the case that W contains some atom A_W, such that $A_W \notin M_{red}$, and such that for no atom $A_M \in M_{red}$ is it the case that $A_W <_{pref} A_M$ or $A_M <_{pref} A_W$. However, A_W must be in M_{min} by the definition of a world, and must also be in M_{red} by definition of the reduction operator since preferences in $[P, Pref]$ are well-behaved,

(2) To show that $WFM([P, Pref]_{norm})$ is two-valued, we argue that $[P, Pref]_{norm}$ is dynamically stratified [11]. Since any dynamically stratified program has a two-valued well-founded model proving this property will suffice.

Since P is definite, the core atoms will be dynamically stratified. Also note that if preference atoms of P are two-valued, derived atoms in P will also be two-valued. The remainder of the proof considers an arbitrary preference atom A_1, and shows that A_1 must be true or false in $WFM([P, Pref]_{norm})$, i.e., either there is a rule for A_1 all of whose literals are true, or that there must be a false literal in each rule for A_1 (see [11] for a formal definition). This will be shown either by referring to the dynamic stratum of A_1 or, in certain cases, showing that the static atom dependency graph of $[P, Pref]_{norm}$ contains no cycles through negation involving A_1, a condition that implies that A_1 must be true or false.

(a) First, we note that since $Pref$ is static and non-cyclic, $Pref$ will introduce no cycles through negation in $[P, Pref]_{norm}$ between atoms A_1 and A_2 in P that do not depend on each other in P. In other words, if there is no path between A_1 and A_2 in the static atom dependency graph of P, the normal transformation will introduce no cycles through negation involving A_1 or A_2 in the static atom dependency graph of $[P, Pref]_{norm}$.

(b) Next, we consider cases of interactions between preferences and dependencies in P for A_1 and an arbitrary preference atom A_2 in P where we assume A_1 depends on A_2 in P.
   (i) As a first subcase, assume that A_2 is preferred to A_1, but that A_2 does not depend on A_1 in P. In this case, the normal embedding of will not introduce a cycle through negation in the dependency graph of P. We note in passing that in this case in the dependency graph of $[P, \mathit{Pref}]_{norm})$ A_1 will depend both positively on A_2 (since A_1 depends on A_2 in P), and negatively on A_2 since A_2 is preferred to A_1.
   (ii) Next, consider the case in which A_1 depends on A_2 in P (as stated above), A_2 may or may not depend on A_1 in P, and A_1 is preferred to A_2. In this case, if there is a rule for A_2 in P, then there will be a negative cycle in the atom dependency graph of $[P, \mathit{Pref}]_{norm}$, so that dynamic strata must be considered. There are two subcases to consider.
      (A) If A_1 can be derived apart from A_2 in $[P, \mathit{Pref}]_{norm}$, then A_1 a belongs to a lower dynamic stratum than A_2, and A_1 will be true and A_2 false in the well-founded model of $[P, \mathit{Pref}]_{norm}$.
      (B) If A_1 cannot be proved apart from A_2 there are two further subcases to consider. (1) If the derivation of A_2 is failed, A_2 will be false in the well-founded model of $[P, \mathit{Pref}]_{norm}$ and will not contribute to the undefinedness of A_1. (2) If the derivation of A_2 does not fail, then A_1 and A_2 might be undefined in $[P, \mathit{Pref}]$. To show that this cannot happen, we consider whether A_1 is in a M_{red} where M_{red} is the reduction of the minimal model of P (see Definition B.3). We begin by noting that since P is definite and preferences are well-behaved, M_{red} is two-valued. Assume that A_1 is optimal in the minimal model of P. Then if A_1 is not true in M_{red}, then it must be underivable in P. To see that A_1 is not true in M_{red}, consider that A_1 cannot be derived without A_2 (in the context of the present assumption) and since

$$M_{red} = T_P\big(\mathit{least\_fixed\_point}\big(T_{P_C \cup P_D}(M_{red})\big) \cup M_{red}\big)|_{\mathit{pref\_atoms}}$$

      by the optimal subproblem property, A_2 must be in M_{red} as well, but since A_1 is preferred to A_2, A_2 cannot be in M_{red}, leading to a contradiction. Therefore A_1 must be underivable in P, and so will not be undefined in $\mathit{WFM}([P, \mathit{Pref}])_{norm}$. For the case in which A_1 is not optimal in the minimal model of P, there must be some other atom A' optimal in the minimal model of P and preferred to A_1. Since preferences are acyclic, the argument for A' is similar to that of A: A' must either be true, or $[P, \mathit{Pref}]$ will not have the optimal subproblem property.

(3) Note that in part (1) of the proof the unique strontly optimal world $\mathcal{W}$ for $[P, \mathit{Pref}]$ was identified with M_{red}. Using the argument of part (2), the remainder of the proof is a straightforward induction on the dynamic strata of $\mathit{WFM}([P, \mathit{Pref}]_{norm})$ to show that $\mathit{WFM}([P, \mathit{Pref}]_{norm}) = M_{red}$. □

References

[1] G. Brewka, Well-founded semantics for extended logic programs with dynamic preferences, J. Artificial Intelligence Res. 4 (1996) 19–36.
[2] G. Brewka, T. Eiter, Preferred answer sets for extended logic programs, Artificial Intelligence 109 (1999) 297–356. Full version of paper originally published in Proceedings of the 6th Conference on Principles of Knowledge Representation and Reasoning, Trento, Italy, 1998, pp. 86–97.
[3] J. Delgrande, T. Schaub, H. Tompits, Logic programs with compiled preferences, in: Proc. 8th International Workshop on Non-Motonic Reasoning, Breckenridge, CO, 2000.
[4] J. Gartner, T. Swift, A. Tien, L.M. Pereira, C. Damásio, Psychiatric diagnosis from the viewpoint of computational logic, in: J.W. Lloyd et al. (Eds.), Proc. International Conference on Computational Logic, Lecture Notes in Artificial Intelligence, Vol. 1861, Springer, Berlin, 2000, pp. 1362–1376.
[5] M. Gelfond, T.C. Son, Reasoning with prioritized defaults, in: J. Dix, L.M. Pereira, T.C. Przymusinski (Eds.), Logic Programming and Knowledge Representation, LPKR'97, Lecture Notes in Computer Science, Vol. 1471, Springer, Berlin, 1998, pp. 164–223.
[6] K. Govindarajan, B. Jayaraman, S. Mantha, Preference logic programming, in: Proc. International Conference on Logic Programming, Tokyo, 1995, pp. 731–746.
[7] B. Jayaraman, K. Govindarajan, S. Mantha, Preference logic grammars, Computer Languages 24 (1998) 179–196.
[8] J.W. Lloyd, Foundations of Logic Programming, Springer, Berlin, 1984.
[9] F.C.N. Pereira, D.H.D. Warren, Parsing as deduction, in: Proc. 21st Annual Meeting of the Association for Computational Linguistics, Cambridge, MA, 1983, pp. 137–144.
[10] H. Przymusinska, T. Przymusinski, Weakly perfect model semantics for logic programs, in: Proc. Joint International Conference/Symposium on Logic Programming, Seattle, WA, 1988, pp. 1106–1120.
[11] T. Przymusinski, Every logic program has a natural stratification and an iterated least fixed point model, in: ACM Symp. on Principles of Database Systems, Philadelphia, PA, ACM Press, 1989, pp. 11–21.
[12] I.V. Ramakrishnan, A. Roychoudhury, T. Swift, A standardization tool for data warehousing, in: Practical Applications of Prolog, 1997.
[13] C. Sakama, K. Inoue, Represending priorities in logic programs, in: Proc. Joint International Conference/Symposium on Logic Programming, Bonn, Germany, 1996, pp. 82–96.
[14] V. Santos Costa, L. Damas, R. Reis, R. Azevedo, YAP User's Manual, 2000, http://www.ncc.up.pt/~vsc/Yap.
[15] S. Shieber, Y. Schabes, F. Pereira, Principles and implementations of deductive parsing, J. Logic Programming 24 (1995) 3–36.
[16] L. Sterling, E. Shapiro, The Art of Prolog, MIT Press, Cambridge, MA, 1986.
[17] T. Swift, Tabling for non-monotonic programming, Ann. Math. Artificial Intelligence 25 (3–4) (1999) 201–240.
[18] T. Swift, C. Henderson, R. Holberger, J. Murphey, E. Neham, CCTIS: An expert transaction processing system, in: Proc. Sixth Conference on Industrial Applications of Artificial Intelligence, 1994, pp. 131–140.
[19] The XSB Programmer's Manual: Version 2.4, Vols. 1 and 2, 2001, http://xsb.sourceforge.net.
[20] A. van Gelder, K. Ross, J. Schlipf, Unfounded sets and well-founded semantics for general logic programs, J. ACM 38 (3) (1991) 620–650.