Logic Programs for Intelligent Web Search

Bernd Thomas

Universität Koblenz, Institut für Informatik, Rheinau 1, D-56075 Koblenz, Germany
Email: bthomas@uni-koblenz.de

Abstract

We present a general framework for information extraction from web pages based on a special wrapper language, called *token-templates*. By using *token-templates* in conjunction with logic programs we are able to reason about web page contents, search and collect facts and derive new facts from various web pages. We give a formal definition for the semantics of logic programs extended by *token-templates* and define a general answer-complete calculus for these extended programs. These methods and techniques are used to build intelligent mediators and web information systems.
Keywords: intelligent information systems; intelligent information retrieval; WWW; information extraction; template based wrappers; mediators; logic programming; theory reasoning; deductive web databases; softbots.

1 Introduction

Our goal is to develop techniques for the process of information extraction (IE), locating specific pieces of text in a document, from semistructured documents like web pages. In the last few years many techniques have been developed to solve this problem [1, 6, 7, 9, 16], where *wrappers* and *mediators* fulfill the general process to retrieve and integrate information from heterogeneous data sources into one information system.

We focus our work on a special class of wrappers, which extract information from web pages and map it into a relational representation. We call this information extraction process *fact-retrieval*; due to logic programming the extracted information is represented by ground atoms. In this paper we present a general framework for the fact retrieval from arbitrary, semistructured documents based on our special wrapper language, called *token-templates.*

We assume that, normally, the user is guided by his own domain specific knowledge when searching the web, manually extracting information and comparing the found facts. It is very obvious that these user processes involve inference mechanisms like reasoning about the contents of web pages, deducing relations between web pages and using domain specific background knowledge. This user processes can be modeled by deduction, based on a set of rules, e.g. which pages to visit and how to extract facts. The logic programming paradigm allows us to: model background knowledge to guide the web search and the application of extraction templates; reason about web pages and to collect relevant facts. In the context of *wrappers* and *mediators* [16], *token-templates* are used to construct special wrappers to retrieve facts from web pages. Logic programs in addition with token-templates offer a basis to construct wrappers and mediators, which search the web with deductive methods.

2 A Wrapper Language for Semistructured Documents

We assume the reader to be familiar with the concepts of *feature structures* and *unification* The process of fact-retrieval is seperated into two major steps:
1) The source code of a web page as shown in Figure 1 is transformed into a list of tokens (Section 2.1), by using an arbitrary lexical anlayser, like *FLEX* or *LEX*[1]. 2) Extraction templates built from tokens and special operators are applied to the tokenized document. According to the successfull matching of these templates the relevant information is extracted by means of unification techniques and mapped into a relational representation.

```
<P><IMG SRC=img/bmp_priv.gif><B>286 AT</B>,
1 MB Ram, 60 MB FP, 2 Disklaufw., Bigtower,
100 DM, 4 x 1 MB SIMM 30 Pin, 50 DM.
Tel.: 06742/ 5926
<P><IMG SRC=img/bmp_priv.gif><B>Pentium 90</B>
48 MB RAM, Soundblaster AWE 64, DM 650,-.
Tel.: 06743/ 1582
```

Fig. 1. HTML source code of an online advertisment

2.1 The Token

We call a *simple* and acyclic feature structure a *token*, if and only if it has a feature named `type` and no feature value that consists of another feature. A feature value may consist of any constant or variable. We write variables in capital letters and constants quoted if they start with a capital letter. Furthermore, we choose a term notation for feature structures (token), that is different from that proposed by Carpenter in [2]. We do not code the features to a fixed argument position, instead we extend the arguments of the annotated term, by the notation `Feature = Value` (extended term notation). This offers us more flexibility in the handling of features. For example the text `<b>Pentium 90</b>` may be written as the list of tokens:`[token(type=html, tag=b), token(type=word,txt='Pentium'), token(type=wspace,val=blank), token(type=int,val=90), token(type=html_end,tag=b)]`.

2.2 Token Matching

Let us assume, that an arbitrary web page transformed into a token list is given. The key idea is now to recognize a token or a token sequence in this token list. Therefore we need techniques to match a token description with a token. For feature structures a special unification was defined in [12]. For our purposes we need a modified version of this unification, the *token-unification.*

Definition 1 (Token-Unification). *Let* $T_1 = token(m_1 = w_1, m_2 = w_2, \ldots, m_j = w_j)$ $j \geq 1$ *and* $T_2 = token(m'_1 = w'_1, m'_2 = w'_2, \ldots, m'_k = w'_k)$ $k \geq 1$ *be tokens. Let* $A_1 = \bigcup_{i=1}^{j} \{(m_i, w_i)\}$ *and* $A_2 = \bigcup_{i=1}^{k} \{(m'_i, w'_i)\}$ *be the feature-value sets of the tokens* T_1 *and* T_2*. It must hold that* $\{m_1, \ldots, m_j\} \subseteq \{m'_1, \ldots, m'_k\}$*. If this is not the case, we call* T_1 *and* T_2 *not token-unifiable,* $\not\perp$*. Let* $F = \{m \mid (m, w) \in A_1\}$ *be the set of features* T_1 *and* T_2 *have in common. The terms* T'_1 *and* T'_2 *are defined as follows:* $T'_1 := token(m_1 = w_1, \ldots, m_n = w_n)$ *and* $T'_2 := token(m_1 = w'_1, \ldots, m_n = w'_n)$ *with* $m_i \in F \wedge (m_i, w_i) \in A_1 \wedge (m_i, w'_i) \in A_2 \wedge 1 \leq i \leq n \wedge n = |F|$*.* T_1 *and* T_2 *are token-unifiable* iff T'_1 *is unifiable with* T'_2*. The most general unifier (mgU)* σ *of* T_1 *and* T_2 *is the mgU of* T'_1 *and* T'_2 *wrt. the usual definition. We*

[1] This method allows us to apply our techniques not only to *HTML*-documents, but also to any kind of semi-structured text documents.

write $T_1 \sqcup_\sigma T_2$ iff T_1 *is token-unifiable with* T_2 *and* σ *is the mgU of the unification from* T_1' *with* T_2'. ■

The motivation for this directed unification[2] is to interpret the *left* token to be a pattern to match the *right* token. This allows us to set up feature constraints in an easy way, by simply adding a feature to the *left* token. On the other hand we can match a whole class of tokens, if we decrease the feature set of the *left* token to consist only of the *type* feature. For example: *token(tag=X,type=html,href=Y)* $\sqcup_\sigma$ *token(type=html, tag=a,href='http://www.bmw.de')* with $\sigma = [X/a, Y/'http://www.bmw.de']$

For ease of notation we introduce an alternative notation t^k for a token of type k (`type` feature has value k), that is given by k or $k(f_1 = v_1, f_2 = v_2, \ldots, f_n = v_n)$ with f_i and v_i are the features and values of t^k where n is the number of features of t^k. We call this notation *term-pattern* and define a transformation $\mathcal{V}$ on *term-patterns* such that $\mathcal{V}$ transforms the *term-pattern* into the corresponding token. This transformation exchanges the functor of the *term-pattern*, from type k to *token* and adds the argument $type = k$ to the arguments. Now we can define the basic match operation on a *term-pattern* and a token:

Definition 2 (Term-Match). *Let* t^k *be a term-pattern and* T *a token in extended term notation. The term-pattern* t^k *matches the token* T, $t^k \sqsubset_\sigma T$, iff $\mathcal{V}(t^k)$ *is token-unifiable with* T: $t^k \sqsubset_\sigma T \iff \mathcal{V}(t^k) \sqcup_\sigma T$ *where* σ *is mgU.* ■

Example: *html(href=Y)* $\sqsubset_\sigma$ *token(type=html,tag=a,href='http://www.bmw.de')*
with $\sigma = [Y/'http://www.bmw.de']$

2.3 Token Pattern

If we interpret a token to be a special representation of text pieces, the definition of term-matching allows us to recognize certain pieces of text and extract them by the process of unification. This means the found substitution σ contains the extracted information. Therefore we developed the language of *token-pattern*, to define extraction templates, which are used to match match sequences of tokens in a tokenized web page. *Token-patterns* are based on similiar concepts as are regular expressions. The difference is, that additional operators for iteration, exclusion, enumaration, unification and negation are defined on *tokens* and *token-pattern*. Assume that a tokenized document D is given. A match of a *token-pattern* p on D, $p \rhd D$, returns a set of triples (MS, RS, σ), where MS is the matched token sequence, RS is the rest sequence of D and σ is the mgU of the token unifications applied during the matching process. We emphasize, that we compute all matches and do not stop after we have found one successful match, though this can be achieved by the use of the *once* operator.

[2] In the sense that the feature set of the *left* token must be a subset of the feature set of the *right* token.

pattern	semantics
p	If $p \sqsubset_\sigma D(1)$ holds, then the matched sequence is the list containing exactly one element $D(1)$ and *RS* is *D* without the first element. $D(n)$ denotes the n-th element of the sequence D.
$?p_1$	Matches the pattern p_1 once or never. First the match of p_1 then the empty match sequence is enumerated.
$!p_1$	Matches the pattern p_1 once or never. First the empty match sequence then the match of p_1 is enumerated.
$+p_1$	Matches the pattern p_1 arbitrarily often but at least once. Uses a decreasing enumeration order of the matches according to their length, starting with the longest possible match.
$-p_1$	Matches pattern p_1 arbitrarily often but at least once. Uses an increasing enumeration order of the matches according to their length, starting with the shortest possible match.
$*p_1$	Matches the pattern p_1 arbitrarily often. Uses a decreasing enumeration order of the matches according to their length, starting with the longest possible match.
$\#p_1$	Matches the pattern p_1 arbitrarily often. Uses a increasing enumeration order of the matches according to their length, starting with the shortest possible match.
$not(t_1,\ldots,t_n)$	The *not* operator matches exactly one token t in *D*, if no $t_i \in \{t_1,\ldots,t_n\}$ exists, such that $t_i \sqsubset_\sigma t$ holds. The token $t_1 \ldots t_n$ are excluded from the match.
$times(n,t)$	Matches exactly *n* tokens t.
any	Matches an arbitrary token.
$once(p_1)$	The *once* operator 'cuts' the set of matched tokens by p_1 down to the first match of p_1. Useful if we are interested only in the first match and not in all alternative matches defined by p_1.
$X = p_1$	Unification of X and the matched sequence of p_1. Only successful if p_1 is successful and if *MS* of p_1 is unifiable with X.
p_1 *and* p_2	Only if p_1 and p_2 both match successfully, this pattern succeeds. The matched sequence of p_1 and p_2, is the concatenation of *MS* of p_1 and *MS* of p_2.
p_1 *or* p_2	p succeeds if one of the pattern $p1$ or p_2 is successfully matched. The matched sequence of p is either the matched sequence of p_1 or p_2. The *and* operator has higher priority than the *or* operator, (e.g. *a and b or c* $\equiv$ (*a and b*) *or c*)

2.4 Token-Templates

A *token-template* defines a relation between a tokenized document, extraction variables and a *token-pattern*. Extraction variables are those variables used in a *token-pattern*, which are of interest due to their instantiation wrt. to the information we want to extract. They *contain* the extracted information we are interested in.

Definition 3 (Token-Template). *Let p be a token-pattern, D an arbitrary tokenized document and $v_1,\ldots,v_n \in \mathcal{V}ar(p)$ variables in p. For $v_1,\ldots,v_n$ we write v and v_σ for applying the substitution σ to v. A token-template r is defined as follows:*

$$r(D,v,p) := \{(D,v_\sigma,p_\sigma) | (MS,RS,\sigma) \in p \rhd D\}$$

Template definitions are written as: $\underline{template}\ r(D,v_1,\ldots,v_n) := p$
r is called the template name, v_σ is the extraction tupel and $v_1,\ldots,v_n$ are called extraction variables. ■

Figure 2 depicts a *token-template*[3] for the extraction of advertisments as shown in Figure 1.

$\underline{template}\ advertise(Doc, Item, Description) := \#any\ and\ html(tag = img)\ and\ html(tag = b)\ and\ Item = *not(html\_end(tag = b))\ and\ html\_end(tag = b)\ and\ once(Description = *not(html(tag = p)))$

Fig. 2. Token-Template for advertisment extraction

3 Logic Programs and Token-Templates

The basic idea of the integration of *token-templates* and LPs is to extend a logic programm with a set of *token-templates* (extended LPs), that are interpreted as special program clauses. Intuitively *token-templates* provide a set of facts to be used in logic programs. From the implementational point of view these *token-template* predicates may be logical programs or modules that implement the downloading of web pages and the token matching. From the theoretical point of view we consider these template sets to be axiomatizations of a theory, where the calculation of the theory (the facts) are performed by a background reasoner. In the following we refer to *normal logic programs* when we talk about logic programs. We assume the reader to be familiar with the fields of *logic programming* [11] and *theory reasoning* [13].

3.1 Template Theories

A *token-template theory* $\mathcal{T}_T$ is the set of all template ground atoms, that we obtain by applying all templates in T. For example, consider the template set $\{t(D,v,p)\}$. Assume p to be an arbitrary *token-pattern* and v an extraction tupel. A *template theory* for T is given by $\mathcal{T}_{\{t(D,v,p)\}} := \{t(D,v',P')|(D,v',p') \in t(D,v,p)\}$. This interpretation of *token-templates* associates a set of ground unit clauses with a given set of *token-templates.* The formal definition is as follows:

Definition 4 (Template-Theory, $\mathcal{T}_\mathbf{T}-$Interpretation, $\mathcal{T}_\mathbf{T}-$Model). *Let $G(t(D,v,p)) = \{t(D,v',p')|D$ is a ground term and $(D,v',p') \in t(D,v,p)\}$ be the set of facts of a template $t(D,v,p)$ and $T = \{t_1(D,v_1,p_1), t_2(D,v_2,p_2), \ldots, t_n(D,v_n,p_n)\}$ a set of token-templates. Then $\mathcal{T}_T = \bigcup_{i=1}^{n} G(t_i(D,v_i,p_i))$ with $t_i(D,v_i,p_i) \in T$ is the token-template theory for T. Let P be a normal logic program with signature Σ, such that Σ is also a signature for $\mathcal{T}_T$:*

- *A $\Sigma-$Interpretation I is a $\mathcal{T}_T$–$\Sigma-$Interpretation* iff *$I \models \mathcal{T}_T$.*
- *A Herbrand $\mathcal{T}_T$–$\Sigma-$Interpretation is a $\mathcal{T}_T$–$\Sigma-$Interpretation, that is also a herbrand interpretation.*
- *A $\mathcal{T}_T$–$\Sigma-$Interpretation I is a $\mathcal{T}_T-$Model for P* iff *$I \models P$.*
- *Let X be a clause, wrt. Σ. X is a logical $\mathcal{T}_T$–Consequence from P, $P \models_{\mathcal{T}_T} X$,* iff *for all $\mathcal{T}_T$–$\Sigma-$Interpretations I : $I \models P$ follows $I \models X$.*

■

[3] For a more detailed description of the *token-template* language (e.g. recursive templates, code calls) the reader is refered to [15]

The corresponding *template theory* for the token-template depicted in Figure 2 and the document of Figure 1 is the set: {advertise('286 AT', '1 MB Ram, 60 MB FP, 2 Disklaufw., Bigtower, 100 DM, 4 x 1 MB SIMM 30 Pin, 50 DM. Tel.: 06742/ 5926'), advertise('Pentium 90', '48 MB RAM, Soundblaster AWE 64, DM 650,-. Tel.: 06743/ 1582')}

3.2 The $\mathcal{T}_T$–Calculus

So far we have shown when a formula is a logical consequence from an extended logic program. Next the definition of a general calculus for *extended logic programs* based on an arbitrary sound and answer complete calculus is given.

Definition 5 ($\mathcal{T}_T$–Derivation). *Let $\mathcal{K}$ be a sound and answer-complete calculus for normal logic programs and $\vdash$ is the derivation defined by $\mathcal{K}$. Let P be a normal logic program, T a set of token-templates and $\mathcal{T}_T$ the template theory for T. A query $\exists$ Q with calculated substitution σ is $\mathcal{T}_T$–derivable from P, $P \vdash_{\mathcal{T}_T} Q\sigma$,* iff *$Q\sigma$ is derivable from $P \cup \mathcal{T}_T$, $P \cup \mathcal{T}_T \vdash Q\sigma$. A calculus $\mathcal{K}$, with $\mathcal{T}_T$–Derivation is called $\mathcal{T}_T$–Calculus.* ■

Theorem 1 (Soundness $\mathcal{T}_T$–Calculus). *Let $\mathcal{K}$ be a $\mathcal{T}_T$–Calculus and $\vdash_{\mathcal{T}_T}$ the derivation relation defined by $\mathcal{K}$. Let T be a set of token-templates, $\mathcal{T}_T$ the template-theory for T and Q a query for a normal program P. Further let σ be a substitution calculated by $\vdash_{\mathcal{T}_T}$. Then* $\forall Q : P \vdash_{\mathcal{T}_T} Q\sigma \Rightarrow P \models_{\mathcal{T}_T} \forall\, Q_\sigma$ □

Theorem 2 (Completeness $\mathcal{T}_T$–Calculus). *Let $\mathcal{K}$ be $\mathcal{T}_T$–Calculus and $\vdash_{\mathcal{T}_T}$ the derivation relation defined by $\mathcal{K}$. Let T be a set of token-templates, $\mathcal{T}_T$ the template-theory for T and Q a query for a normal program P. Let θ be a correct answer for Q, σ a calculated answer and γ a substitution.*
Then $\forall \theta : P \models_{\mathcal{T}_T} Q\theta \Rightarrow P \vdash_{\mathcal{T}_T} Q\sigma$ *where* $\theta = \sigma\gamma$ □

By Definition 5 we can extend any arbitrary sound and answer-complete first order predicate logic calculus with template theories and use this retrieved information from the WWW for further deductive processes. That means, existing theorem provers or logic programming systems can be used to query the web and reason about its contents. Therefore a calculation procedure is needed to calculate the according template theories. In this paper we neither describe such a procedure in detail nor do we show how the calculation of a template theory can be efficiently integrated in the derivation process of a calculus. The proofs of the soundness and completness theorem are given in [15].

4 Deductive Techniques for Intelligent Web Search

Logic programming and deduction in general offer a wide variety to guide the web search and fact-retrieval process with intelligent methodes and inference processes. The following sections describe some of these techniques.

4.1 Deductive Web Databases

Assume we know two web pages of shoe suppliers, whose product descriptions we want to use as facts in a deductive database. Additionally we are interested in some information about the producer of the product, his address and telephone number that can be retrieved from an additional web page. Therefore we define two *token-templates*, *price_list* and *address*. To simplify notation we leave out the exact *token-pattern* definitions. The following small deductive database allows us to ask for articles and to

derive new facts that provide us with information about the product and the producer. We achieve this by the two rules *article* and *product*, which extract the articles offered in the web pages and will derive new facts about the article and the producer.

$web\_page('shoes\ online', http://www.sh-online.com/offer.html) \leftarrow$
$web\_page('Land\ of\ Shoes', http://www.shland.com/products.html) \leftarrow$
$article(Supplier, Article, Price, ProducerUrl) \leftarrow$
$\quad web\_page(Supplier, Document),\ price\_list(Document, Article, Price, ProducerUrl, Pattern)$
$product(Supplier, Producer, Article, Price, Str, Tel, City) \leftarrow$
$\quad article(Supplier, Article, Price, ProdUrl),\ address(ProdUrl, Supplier, Str, Tel, City, Pattern)$

Example query: Select all products with article name "*Doc Martens*" that cost less than 100: $\leftarrow Article = \text{"}Doc\ Martens\text{"}, product(Supplier, Article, Producer, Price, Str, Tel, City), Price < 100$

4.2 Conceptual Hierachy Guided Search

Many information systems lack the ability to use a conceptual background knowledge to include related topics of interest to their search. Consider the case that a user is interested in computer systems that cost less than 1000 DM. It is obvious that the system should know the common computer types and descriptions (e.g. IBM, Toshiba, Compaq, Pentium, Notebook, Laptop) and how they are conceptually related to each other. Such knowledge will assist a system in performing a successful search. One way to represent such knowledge is by concept description languages or in general by means of knowledge representation techniques. Extended logic programs offer powerful means to set up ontologies. For example a simple relation *is_a* can be used to represent conceptual hierachies to guide the search for information. Consider the following small knowledge base:

$notebook('ThinkPad') \leftarrow \qquad notebook('Satellite') \leftarrow$
$is\_a(notebook, computer) \leftarrow \qquad is\_a(desktop, computer) \leftarrow$
$is\_a(X, notebook) \leftarrow notebook(X)$
$relevant(Q, Z) \leftarrow is\_a(Z, Q)$
$relevant(Q, Z) \leftarrow is\_a(Y, Q), relevant(Y, Z)$

Assume our general query to search for computers less than 1000 DM is splitted into a sub-query like $\leftarrow relevant(computers, X)$ to our small example knowledge base. The query computes a set of answers: $\theta = \{[X/notebook], [X/desktop], [X/'ThinkPad'], [X/'Satellite']\}$. This additional inferred query information θ can be used in two different ways: (1.) Use it to search for new web pages, e.g. by querying standard search engines with elements of θ as search keywords and apply further extended logic programs on the returned candidate pages. (2.) Enhance the information extraction process with the derived information θ by reducing the constraints on special token features in the token-templates to be applied, e.g. *word(txt=Q) or word(txt=*θ_1*) or ... or word(txt=*θ_n*)* where θ_i are computed answers and Q is the original object to search for. That means in general the search is extended with all *known* sub-concepts (e.g. $computer \rightarrow notebook \rightarrow \ldots$) or instances (e.g. $'ThinkPad'$) of the query concept.

4.3 Optimizing Web Retrieval

To avoid fetching of irrelevant web pages and starting a *fact-retrieval* process we know for certain to fail, Levy suggests in [10] the use of source descriptions. For the fact retrieval from the WWW this might offer a great speed up, because due to the network load the fetching of web documents is often very time intensive. We can easily apply

these methods to *extended logic programs*, by the definition of rules, whose body literals define constraints on the head arguments expressing our knowledge about the content of the web pages. The following example illustrates these methods:

$$offer('http://www.autos.de', Price, Country) \leftarrow Price < 40000, Country = 'Deutschland'$$
$$offer('http://www.cars.com', Price, Country) \leftarrow Price > 40000, Country = 'USA'$$
$$product(Car, Price, Country) \leftarrow offer(Url, Price, Country), template\_cars(Url, Car, Price, Country)$$

Assuming we are interested in american cars that cost 50000 dollars, the query *product(C,50000,'USA')* will retrieve the according offers. Because of the additional constraints on the price and the country given in the body of the rule *offer*, the irrelevant web page with german car offers is left out. By simple methods, provided by the logic programming paradigm for free, we are able to guide the search and fact retrieval in the world wide web based on knowledge representation techniques and we are able to speed up the search for relevant information.

5 Related Work and Conclusion

Several web information systems have been developed in the last few years. One class of applications is called *Softbots* [4], which are domain specific automated search tools for the WWW. Such existing systems use either tailored extraction techniques (*Ahoy!*) that are very domain specific or their extraction techniques are based on highly restrictive assumptions about the syntactical structure of a web page (*Shopbot*). Both systems do not follow the concept of a general purpose extraction language like *token-templates*. *Token-templates* are applicable to any kind of semistructured text documents, and hence not restricted to a specific domain.

Systems like *IM* [10] or *W3QS* [8] also provide means to query web information sources. Though Levy et. al. also choose a relational data model to reason about data, and show several techniques for source descriptions or constructing query plans, they leave the problem of information extraction undiscussed in their work. We showed solutions for both the extraction of facts and reasoning by extended logic programs. The *W3QS* system uses a special web query language similiar to the relational database query language *SQL*. Though an additional construction kit for IE processes is given, this seems to be focussed only on the detection of hyper links and their descriptions. The concept of database views for web pages is also introduced, but no information about recursive views is provided, whereas extended logic programs offer these abilities.

Heterogeneous information systems, like *HERMES* [14], *Infomaster* [5] or *TSIMMIS* [3] all use special mediator techniques to access web information sources among other data sources. These systems use their own mediator model (language) to interface with the special data source wrappers. The system *HERMES* for example is based on a declarative logical mediator language and therefore is similiar to our approach using extended logic programs as mediators and *token-templates* as a special wrapper language. The advantage of our presented approach is simply that the above named systems, except *TSIMMIS*, do not incorporate a general purpose wrapper language for text documents.

Different from the template based extraction languages described in [7] and [1] or the underlying language used in the wrapper construction tool by Gruser et. al. [6],

token-templates incorporate the powerful concepts of recursion and *code calls* [15]. These concepts allow the recognition and extraction of arbitrary hierachical syntactic structures and extend the matching process by additional control procedures invoked by code calls. Logic programs used as *code calls* can guide the extraction process with a manifold of AI methods. Our methods presented have been successfully implemented as a search-engine for private advertisments [15] and a meta online-encyclopedia.

References

1. M. E. Califf and R. J. Mooney. Relational Learning of Pattern-Match Rules for Information Extraction. In *Working Papers of the ACL-97 Workshop in Natural Language Learning*, 1997.
2. B. Carpenter. Typed Feature Structures: an Extension of First-order Terms. In *Proceedings of the International Symposium on Logic Programming*, 1991. San Diego.
3. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, Y. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of IPSJ*, 1994. Japan.
4. O. Etzioni. Moving Up the Information Food Chain. *AI Magazine*, 18(2):11–18, Summer 1997.
5. M. R. Genesereth, A. M. Keller, and O. Duschka. Infomaster: An Information Integration System. *Proceedings of ACM SIGMOD Conference*, May 1997.
6. J. Gruser, L. Raschid, M. Vidal, and L. Bright. A wrapper generation toolkit to specify and construct wrappers for web accesible data. Technical report, UMIACS, University of Maryland, 1998.
7. J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *In Proceedings of the Workshop on Management of Semistructured Data*, May 1997.
8. D. Konopnicki and O. Shmueli. W3QS: A query system for the world-wide web. In *Proceedings of VLDB'95*, 1995.
9. N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In M. E. Pollack, editor, *Fifteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 729–735, August 1997. Japan.
10. A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of the 22nd VLDB Conference*, 1996. Mumbai(Bombay), India.
11. J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2 edition, 1987.
12. S. M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI, Leland Stanford Junior University, 1986. CSLI Lecture Notes 4.
13. M. Stickel. Automated Deduction by Theory Resolution. *Journal of Automated Reasoning*, 1:333–355, 1985.
14. V. Subrahmanian, S. Adali, A. Brink, R. Emery, J. J. Lu, A. Rajput, T. J. Rogers, R. Ross, and C. Ward. HERMES: A Heterogeneous Reasoning and Mediator System, 1996. http://www.cs.umd.edu//projects/hermes/overview/paper/index.html.
15. B. Thomas. Intelligent Web Querying with Logic Programs. In J. Dix and S. Hölldobler, editors, *Proceedings of the Workshop on Inference Systems in Knowledge-based Systems, preceding the national German AI conference KI '98, Bremen, Germany*. University of Koblenz, TR 10/98, August 1998.
16. G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.