

Learning Semantic Descriptions of Web Information Sources*

Mark James Carman and Craig A. Knoblock

University of Southern California

Information Sciences Institute

4676 Admiralty Way,

Marina del Rey, CA 90292

{carman, knoblock}@isi.edu

Abstract

The Internet is full of information sources providing various types of data from weather forecasts to travel deals. These sources can be accessed via web-forms, Web Services or RSS feeds. In order to make automated use of these sources, one needs to first model them semantically. Writing semantic descriptions for web sources is both tedious and error prone. In this paper we investigate the problem of automatically generating such models. We introduce a framework for learning Datalog definitions for web sources, in which we actively invoke sources and compare the data they produce with that of known sources of information. We perform an inductive search through the space of plausible source definitions in order to learn the best possible semantic model for each new source. The paper includes an empirical evaluation demonstrating the effectiveness of our approach on real-world web sources.

1 Introduction

We are interested in making use of the vast amounts of information available as services on the Internet. In order to make this information available for structured querying, we must first model the sources providing it. Writing source descriptions by hand is a laborious process. Given that different services often provide similar or overlapping data, it should be possible to use knowledge of previously modeled services to learn descriptions for newly discovered ones.

When presented with a new source of information (such as a Web Service), the first step in the process of modeling the source is to determine what type of data it requires as

*This research is based upon work supported in part by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010, in part by the National Science Foundation under Award No. IIS-0324955, and in part by the Air Force Office of Scientific Research under grant number FA9550-04-1-0105. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

input and what type of data it produces as output. In previous work [Heß and Kushmerick, 2003; Lerman *et al.*, 2006], researchers have addressed the problem of classifying the attributes of a service into semantic types (such as *zipcode*). Once the semantic types for the inputs are known, we can invoke the service, but are still not able to make use of the data it returns. To do that, we need also to know how the output attributes relate to the input. For example, a weather service may return a *temperature* value when queried with a *zipcode*. The service is not very useful, until we know whether the temperature being returned is the current temperature, the predicted high temperature for tomorrow, or the average temperature for this time of year. These three possibilities can be described by Datalog rules as follows: (Note that the $\$$ -symbol is used to distinguish the input attributes of a source.)

```
1 source($zip, temp) :- currentTemp(zip, temp).
2 source($zip, temp) :- forecast(zip, temp).
3 source($zip, temp) :- averageTemp(zip, temp).
```

The expressions state that the input *zipcode* is related to the output temperature according to domain relation called *currentTemp*, *forecast*, and *averageTemp* respectively, each of which is defined in some domain ontology. In this paper we describe a system capable of inducing such definitions automatically. The system leverages what it knows about the domain, namely the ontology and a set of known sources, to learn a definition for a newly discovered source.

1.1 An Example

We introduce the problem of inducing definitions for online sources by way of an example. In the example we have four semantic types, namely: *zipcode*, *distance*, *latitude* and *longitude*. We also have three known sources of information, each of which has a definition in Datalog. The first source, aptly named *source1*, takes in a *zipcode* and returns the latitude and longitude coordinates of its centroid. The second calculates the great circle distance between two pairs of coordinates, while the third converts a distance from kilometers into miles. Definitions for the sources are as follows:

```
source1($zip, lat, long):- centroid(zip, lat, long).
source2($lat1, $long1, $lat2, $long2, dist):-
  greatCircleDist(lat1, long1, lat2, long2, dist).
source3($dist1, dist2):- convertKm2Mi(dist1, dist2).
```

The goal in this example is to learn a definition for a new service, called *source4*, that has just been discovered on the

Internet. This new service takes in two zipcodes as input and returns a distance value as output:¹

```
source4($zip, $zip, distance)
```

The system described in this paper takes this type signature as well as the definitions for the known sources and searches for an appropriate definition for the new source. The definition discovered in this case would be the following conjunction of calls to the known sources:

```
source4($zip1, $zip2, dist):-
  source1(zip1, lat1, long1),
  source1(zip2, lat2, long2),
  source2(lat1, long1, lat2, long2, dist2),
  source3(dist2, dist).
```

This definition states that the output distance can be calculated from the input zipcodes, by first giving those zipcodes to source1, calculating the distance between the resulting coordinates using source2, and then converting the distance into miles using source3. To test whether this source definition is correct the system must invoke the new source and the definition to see if the values generated agree with each other. The following table shows such a test:

\$zip1	\$zip2	dist (actual)	dist (predicted)
80210	90266	842.37	843.65
60601	15201	410.31	410.83
10005	35555	899.50	899.21

In the table, the input zipcodes have been selected randomly from a set of examples, and the output from the source and the definition are shown side by side. Since the output values are quite similar, once the system has seen a sufficient number of examples, it can be confident that it has found the correct semantic definition for the new source.

The definition above was written in terms of the source predicates, but could just as easily have been written in terms of the domain relations. To do so, one needs to replace each source predicate by its definition as follows:

```
source4($zip1, $zip2, dist):-
  centroid(zip1, lat1, long1),
  centroid(zip2, lat2, long2),
  greatCircleDist(lat1, long1, lat2, long2, dist2),
  convertKm2Mi(dist1, dist2).
```

Written in this way, the new definition for source4 makes sense at an intuitive level: The source is simply calculating the distance in miles between the centroids of the zipcodes.

2 Problem Formulation

We are interested in learning definitions for sources by invoking them and comparing the output they produce with that of known sources of information. We formulate the problem as a tuple $\langle T, R, S, s^* \rangle$, where T is a set of *semantic data-types*, R is a set of *domain relations*, S is a set of known *sources*, and s^* is the *new source*. Each of the semantic types comes with a set of example values and a function for checking equality

¹The assignment of semantic types to the inputs and outputs of a service can be performed automatically as described in [Lerman et al., 2006]. In general, sources will output relations rather than singleton values.

between values of the type. The set of relations R may include interpreted predicates, such as \leq . Each source $s \in S$ is associated with a type signature, a binding constraint (that distinguishes input from output) and a view definition, that is a conjunctive query over the relations in R . The new source to be modeled s^* , is described in the same way, except that its view definition is unknown. The solution to the *Source Definition Induction Problem* is a definition for this source.

By describing sources using the powerful language of conjunctive queries, we are able to model most information sources on the Internet (as sequential compositions of simple functionality). We do not deal with languages involving more complicated constructs such as *aggregation*, *union* or *negation* because the resulting search space would be prohibitively large. Finally, we assume an open-world semantics, meaning that sources may be incomplete with respect to their definitions (they may not return all the tuples implied by their definition). This fact complicates the induction problem and is addressed in section 3.4.

3 Algorithm

The algorithm used to search for and test candidate definitions takes as input a type signature for the new source (also called the *target predicate*). The space of candidate definitions is then enumerated in a best-first manner, in a similar way to top-down Inductive Logic Programming (ILP) systems like FOIL [Cameron-Jones and Quinlan, 1994]. Each candidate produced is tested to see if the data it returns is in some way similar to the target:

```

1 Invoke target with set of random inputs;
2 Add empty clause to queue;
3 while queue  $\neq \emptyset$  do
4    $v \leftarrow$  best definition from queue;
5   forall  $v' \in \text{expand}(v)$  do
6     if  $\text{eval}(v') \geq \text{eval}(v)$  then
7       insert  $v'$  into queue;
8     end
9   end
10 end
```

Algorithm 1: Best-First Search Algorithm

3.1 Invoking the Source

The first step in our algorithm is to generate a set of tuples that will represent the target predicate during the induction process. In other words, we try to invoke the new source to sample some data. Doing this without biasing the induction process is not trivial. The system first tries to invoke the source with random combinations of input values taken from the examples of each type. Many sources have implicit restrictions on the combination of input values. For example, a geocoding service which takes a *number*, *street*, and *zipcode* as input, may only return an output if the address actually exists. In such cases, randomly combining values to form input tuples is unlikely to result in any successful invocations. After failing to invoke the source a number of times, the system will try to generate examples from other sources whose output contains the required combination of attribute types. Fre-

quency distributions can also be associated with the example values of each semantic type, such that common constants (like *Ford*) can be chosen more frequently than less common ones (like *Ferrari*).

3.2 Generating Candidates

Once the system has assembled a representative set of tuples for the new source, it starts generating candidate definitions by performing a top-down best-first search through the space of conjunctions of source predicates. In other words, it begins with a very simple source definition and builds ever more complicated definitions by adding one literal (source predicate) at a time to the end of the best definition found so far. It keeps doing this until the data produced by the definition matches that produced by the source being modeled. For example, consider a newly discovered source that takes in a zipcode and a distance, and returns all the zipcodes that lie within that radius (along with their respective distances). The target predicate representing the source is:

```
source5($zip1, $dist1, zip2, dist2)
```

Now assume we have one known source, namely *source4* from the previous example:

```
source4($zip1, $zip2, dist)
```

and we also have the interpreted predicate:

```
≤ ($dist1, $dist2)
```

The search for a definition for *source5* might then proceed as follows. The first definition generated is the empty clause:

```
source5($_, $_, _, _).
```

The null character (`_`) represents a *don't care* variable, which means that none of the inputs or outputs have any restrictions placed on their values. Literals (source predicates) are then added one at a time to refine this definition.² Doing so produces the following candidate definitions, among others:

```
source5($zip1, $dist1, _, _) :- source4(zip1, _, dist1).
source5($zip1, $_, zip2, _) :- source4(zip1, zip2, _).
source5($_, $dist1, _, dist2) :- ≤(dist1, dist2).
```

Note that the semantic types in the signature of the target predicate limit greatly the number of candidate definitions produced. The system checks each of these candidates in turn, selecting the best one for further expansion. Assuming that the first of the three scores the highest, it would be expanded to form more complicated candidates, such as:

```
source5($zip1, $dist1, _, dist2) :-
  source4(zip1, _, dist1), ≤(dist1, dist2).
```

The size of the search space is highly dependent on the arity of the sources. Sources with multiple attributes of the same type make for an exponential number of possible definitions at each expansion step. To limit the search in such cases, we first generate candidates with a minimal number of join variables in the final literal and progressively constrain the best performing definitions (by further equating variables).

The rationale for performing search over the source predicates rather than the domain predicates is that if the search were performed over the latter an additional *query reformulation* step would be required each time a definition is tested.

²Prior to adding the first literal, the system checks if any output echoes an input value, e.g. `source5($zip1, $_, zip1, _)`.

In practice, the fact that definitions for the known sources may contain multiple literals means that many different conjunctions of domain predicates will reformulate to the same conjunction of source predicates, resulting in a much larger search space. For this reason, we perform the search over the source predicates and rely on post-processing to remove redundant literals from the unfolding of the definition produced.

3.3 Limiting the Search Space

The search space generated by this top-down search algorithm may be very large even for a small number of sources. As the number of sources available increases, the search space becomes so large that techniques for limiting it must be used. We employ some standard (and other not so standard) ILP techniques for limiting this space:

1. Maximum clause length
2. Maximum predicate repetition
3. Maximum existential quantification level
4. Definitions must be executable
5. No repetition of variables allowed within a literal

Such limitations are often referred to as *inductive search bias* or *language bias* [Nédellec *et al.*, 1996]. The first restriction limits the length of the definitions produced, while the second limits the number of times the same source predicate can appear in a given candidate. The third restricts the complexity of the definitions by reducing the number of literals that do not contain variables from the head of the clause.³ The fourth requires that source definitions can be executed from left to right, i.e., that the inputs of each source appear in the head of the clause or in one of the literals to the left of that literal. Finally, we disallow definitions in which the same variable appears multiple times in the same literal (in the body of the clause). For example, the following definition which returns the distance between a zipcode and itself, would not be generated, because `zip1` appears twice in the last literal:

```
source5($zip1, $_, _, dist2) :- source4(zip1, zip1, dist2).
```

Such definitions occur rarely in practice, thus it makes sense to exclude them, thereby greatly reducing the search space.

3.4 Comparing Candidates

We proceed to the problem of evaluating candidate definitions. The basic idea is to compare the output produced by the source with that produced by the definition on the same input. The more similar the tuples produced, the higher the score for the candidate. We then average the score over different input tuples to see how well the candidate describes the source overall. In the motivating example, a single output tuple (distance value) was produced for every input tuple (pair of zipcodes). In general, multiple output tuples may be produced by a source (as was the case for *source5*). Thus the system needs to compare the set of output tuples produced by the target with those produced by the definition to see if any of the tuples are the same. Since both the new source and the known sources can be incomplete, the two sets may simply

³The existential quantification level of a literal is the shortest path from that literal (via join variables) to the head of the clause.

overlap, even if the candidate definition correctly describes the new source. Assuming that we can count the number of tuples that are the same, we can use the *Jaccard similarity* to measure how well the candidate hypothesis describes the data returned by the new source:

$$eval(v) = \frac{1}{|I|} \sum_{i \in I} \frac{|O_s(i) \cap O_v(i)|}{|O_s(i) \cup O_v(i)|}$$

Here I denotes the set of input tuples used to test the new source s . $O_s(i)$ denotes the set of tuples returned by the source when invoked with input tuple i . $O_v(i)$ is the corresponding set returned by the candidate definition v . If we view this hypothesis testing as an information retrieval task, we can consider *recall* to be the number of common tuples divided by the number of tuples produced by the source, and *precision* to be the common tuples divided by the tuples produced by the definition. The Jaccard similarity takes both precision and recall into account in a single score.

The table below provides examples of the score for different output tuples. The first three rows of the table show inputs for which the predicted and actual output tuples overlap. In the fourth row, the definition produced a tuple, while the source didn't, so the definition was penalised. In the last row, the definition correctly predicted that no tuples would be output from the source. Our score function is undefined at this point. From a certain perspective the definition should score well here because it has correctly predicted that no tuples would be returned for that input, but giving a high score to a definition when it produces no tuples can be dangerous. Doing so may cause overly constrained definitions that can generate very few output tuples to score well, while less constrained definitions that are better at predicting the output tuples on average can score poorly. To prevent this from happening, we simply ignore inputs for which the definition correctly predicts zero tuples. (This is the same as setting the score for this case to be the average for the other cases.) After ignoring the last row, the overall score for this definition is calculated to be 0.46.

input tuple i	actual output $O_s(i)$	predicted output $O_v(i)$	Jaccard similarity
$\langle a, b \rangle$	$\{\langle x, y \rangle, \langle x, z \rangle\}$	$\{\langle x, y \rangle\}$	1/2
$\langle c, d \rangle$	$\{\langle x, w \rangle, \langle x, z \rangle\}$	$\{\langle x, w \rangle, \langle x, y \rangle\}$	1/3
$\langle e, f \rangle$	$\{\langle x, w \rangle, \langle x, y \rangle\}$	$\{\langle x, w \rangle, \langle x, y \rangle\}$	1
$\langle g, h \rangle$	\emptyset	$\{\langle x, y \rangle\}$	0
$\langle i, j \rangle$	\emptyset	\emptyset	#undef!

3.5 Approximate Matches Between Constants

When deciding whether the two tuples produced by the target and the definition are the same, we must allow for some flexibility in the values they contain. In the motivating example for instance, the distance values returned did not match exactly, but were “sufficiently similar” to be accepted as the same. For certain nominal types, like *zipcode*, it makes sense to check equality using exact string matches. For numeric types like *temperature*, an error bound (like $\pm 0.5^\circ C$) or a percentage error (such as $\pm 1\%$) may be more reasonable. For strings like *company name*, edit distances such as the JaroWinkler score do a better job at distinguishing strings

representing the same entity from those representing different ones. (See [Bilenko *et al.*, 2003] for a discussion of string matching techniques.) In other cases a simple procedure might be available to check equality for a given type, so that values like “Monday” and “Mon” are equated. The actual *equality procedure* used will depend on the semantic type and we assume in this work that such a procedure is given in the problem definition. We note that the procedure need not be 100% accurate, but only provide a sufficient level of accuracy to guide the system toward the correct definition. Indeed, equality rules could even be generated offline by training a machine learning classifier.

3.6 Scoring Partial Definitions

As the search proceeds toward the correct definition, many semi-complete (unsafe) definitions will be generated. These definitions do not produce values for all attributes of the target predicate but only a subset of them. For example, the following definition produces only one of the two output attributes returned by the source:

```
source5($zip1, $dist1, zip2, _) :-
    source4(zip1, zip2, dist1).
```

This presents a problem, because our score is only defined over sets of tuples containing *all* of the output attributes of the new source. One solution might be to wait until the definitions become sufficiently long as to produce all outputs before comparing them to see which one best describes the new source. There are two reasons why we wouldn't want to do that: Firstly, the space of complete (safe) definitions is too large to enumerate, and thus we need to compare partial definitions so as to guide the search toward the correct definition. Secondly, the best definition that the system can generate may well be a partial one, as the set of known sources may not be sufficient to completely model the source.

We can compute the score over the projection of the source tuples on the attributes produced by the definition, but then we are giving an unfair advantage to definitions that do not produce all of the source's outputs. That is because it is far easier to correctly produce a subset of the output attributes than to produce all of them. So we need to penalise such definitions accordingly. We do this by first calculating the size of the domain of each of the missing attributes. In the example above, the missing attribute is a distance value. Since distance is a continuous variable, we approximate the size of its domain using $(max - min)/accuracy$, where *accuracy* is the error-bound on distance values. (This cardinality calculation may be specific to each semantic type.) Armed with the domain size, we penalise the score by scaling the number of tuples returned by the definition according to the size of the domains of all output attributes not generated by it. In essence, we are saying that all possible values for these extra attributes have been “allowed” by this definition. (This technique is similar to that used for learning without explicit negative examples in [Zelle *et al.*, 1995].)

4 Experiments

We tested the system on 25 different problems (target predicates) corresponding to *real* services from five domains. The

methodology for choosing services was simply to use any service that was publicly available, free of charge, worked, and didn't require website wrapping software. The domain model used in the experiments was the same for each problem and included 70 semantic types, ranging from common ones like *zipcode* to more specific types such as stock *ticker* symbols. It also contained 36 relations that were used to model 35 different publicly available services. These known sources provided some of the same functionality as the targets.

In order to induce definitions for each problem, the new source (and each candidate) was invoked at least 20 times using random inputs. To ensure that the search terminated, the number of iterations of the algorithm was limited to 30, and a search time limit of 20 minutes was imposed. The inductive search bias used during the experiments was: {max. clause length: 7, predicate repetition limit: 2, max. existential quantification level: 5, candidate must be executable, max. variable occurrence per literal: 1}. An accuracy bound of $\pm 1\%$ was used to determine equality between *distance*, *speed*, *temperature* and *price* values, while an error bound of ± 0.002 degrees was used for *latitude* and *longitude*. The JaroWinkler score with a threshold of 0.85 was used for strings such as *company*, *hotel* and *airport* names. A hand-written procedure was used for matching *dates*.

4.1 Results

Overall the system performed very well and was able to learn the intended definition (albeit missing certain attributes) in 19 of the 25 problems. Some of the more interesting definitions learnt by the system are shown below:

- 1 GetDistanceBetweenZipCodes(\$zip0, \$zip1, dis2):-
 GetCentroid(zip0, lat1, lon2),
 GetCentroid(zip1, lat4, lon5),
 GetDistance(lat1, lon2, lat4, lon5, dis10),
 ConvertKm2Mi(dis10, dis2).
- 2 USGSElevation(\$lat0, \$lon1, dis2):-
 ConvertFt2M(dis2, dis1), Altitude(lat0, lon1, dis1).
- 3 GetQuote(\$tic0, pri1, dat2, tim3, pri4, pri5, pri6, pri7,
 cou8, -, pri10, -, -, pri13, -, com15) :-
 YahooFinance(tic0, pri1, dat2, tim3, pri4, pri5, pri6,
 pri7, cou8),
 GetCompanyName(tic0, com15, -, -),
 Add(pri5, pri13, pri10), Add(pri4, pri10, pri1).
- 4 YahooWeather(\$zip0, cit1, sta2, -, lat4, lon5, day6, dat7,
 tem8, tem9, sky10) :-
 WeatherForecast(cit1, sta2, -, lat4, lon5, -, day6, dat7,
 tem9, tem8, -, -, sky10, -, -),
 GetCityState(zip0, cit1, sta2).
- 5 YahooHotel(\$zip0, \$-, hot2, str3, cit4, sta5, -, -, -, -) :-
 HotelsByZip(zip0, hot2, str3, cit4, sta5, -).
- 6 YahooAutos(\$zip0, \$mak1, dat2, yea3, mod4, -, -, pri7, -) :-
 GoogleBaseCars(zip0, mak1, -, mod4, pri7, -, -, yea3),
 ConvertTime(dat2, -, dat10, -, -),
 GetCurrentTime(-, -, dat10, -).

The first definition calculates the distance in miles between two zipcodes and is the same as in our original example (*source4*). The second source provided USGS elevation data in feet, which was found to be sufficiently similar to known altitude data in meters. The third source provided stock quote

information, and a definition was learnt involving a similar service from Yahoo. For this source, the system discovered that the current price was the sum of the previous day's close and today's change. The fourth definition is for a weather forecast service, and a definition was learnt in terms of another forecast service. (The system distinguished high from low and forecast from current temperatures.) The fifth source provided information about nearby hotels. Certain attributes of this source (like the hotel's *url* and *phone* number) could not be learnt, because none of the known sources provided them. Nonetheless, the definition learnt is useful as is. The last source was a classified used-car listing from Yahoo that took a zipcode and car manufacturer as input. The system discovered that there was some overlap between the cars (make, model and price) listed on that source and those listed on another site provided by Google.

Domain	Problems # (#Attr.)	Candidates # (#Lit.)	Precis.	Recall
geospatial	9 (5.7)	136 (1.9)	100%	84%
financial	2 (11.5)	1606 (4.5)	56%	63%
weather	8 (11.8)	368 (2.9)	91%	62%
hotels	4 (8.5)	43 (1.3)	90%	60%
cars	2 (8.5)	68 (2.5)	50%	50%

The table above shows for each domain, the number of problems tested, the average number of attributes per problem (in parentheses), the average number of candidates generated prior to the winning definition, and the average number of literals per definition found (in parentheses). The last two columns give the average precision and recall values, where precision is the ratio of correctly generated attributes (of the new source) to all of the attributes generated, and recall is the ratio of correctly generated attributes, to all of the attributes that *should have been* generated. These values indicate the quality of the definitions produced. Ideally, we would like to have 100% precision (no errors in the definitions) and high recall (most of the attributes being generated). That was the case for the 9 geospatial problems. One reason for the particularly good performance on this domain was the low number of attributes per problem, resulting in smaller search spaces. As would be expected, the number of candidates generated was higher for problems with many attributes (financial and weather domains). In general, precision was very high, except for a small number of problems (in the financial and cars domains). Overall the system performed extremely well, generating definitions with a precision of 88% and recall of 69%.

5 Related Work

Early work on the problem of learning semantic definitions for Internet sources was performed by [Perkowitz and Etzioni, 1995], who defined the *category translation problem*. That problem can be seen as a simplification of the source induction problem, where the known sources have no binding constraints or definitions and provide data that does not change over time. Furthermore, they assume that the new source takes a single value as input and returns a single tuple as output. To find solutions to this problem, the authors too used a form of inductive search based on an extension of the FOIL algorithm [Cameron-Jones and Quinlan, 1994].

More recently, there has been some work on classifying web services into different domains [Heß and Kushmerick, 2003] and on clustering similar services together [Dong *et al.*, 2004]. This work is closely related, but at a more abstract level. Using these techniques one can state that a new service is probably a *weather* service because it is similar to other weather services. This knowledge is very useful for service discovery, but not sufficient for automating service integration. In our work we learn more expressive descriptions of web services, namely view definitions that describe how the attributes of a service relate to one another.

The schema integration system CLIO [Yan *et al.*, 2001] helps users build queries that map data from a source to a target schema. If we view this source schema as the set of known sources, and the target schema as a new source, then our problems are similar. In CLIO, the integration rules are generated *semi-automatically* with some help from the user.

The iMAP system [Dhamanka *et al.*, 2004] tries to discover complex (many-to-one) mappings between attributes of a source and target schema. It uses a set of *special purpose searchers* to find different types of mappings. Our system uses a general ILP-based framework to search for many-to-many mappings. Since our system can perform a similar task to iMAP, we tested it on the hardest problem used to evaluate iMAP. The problem involved aligning data from two online cricket databases. Our system, despite being designed to handle a more general task, was able to achieve 77% precision and 66% recall, which is comparable to the performance (accuracy) range of 50-86% reported for iMAP on complex matches with overlapping data.

Finally, the Semantic Web community have developed standards [Martin *et al.*, 2004; Roman *et al.*, 2005] for annotating sources with semantic information. Our work complements theirs by providing a way to automatically generate semantic information, rather than relying on service providers to create it manually. The Datalog-based representation used in this paper (and widely adopted in information integration systems [Levy, 2000]) can be converted to the Description Logic-based representations used in the Semantic Web.

6 Discussion

In this paper we presented a completely automatic approach to learning definitions for online information sources. This approach exploits definitions of sources that have either been given to the system or learned previously. The resulting framework is a significant advance over prior approaches that have focused on learning only the input and outputs of services. One of the most important applications of this work is to learn semantic definitions for data integration systems [Levy, 2000]. Such systems require an accurate definition in order to exploit and integrate available sources of data.

Our results demonstrate that we are able to learn definitions with a moderate size domain model and set of known sources. We intend to scale the technique to much larger problems. We note that the system is already applicable for many domain-specific problems. For example, in the area of geospatial data integration where the domain model is naturally limited, the technique can be applied as is.

There are a number of future directions for this work that will allow the system to be applied more broadly. These include (1) introducing constants into the modeling language, (2) developing additional heuristics to direct the search toward the best definition, (3) developing a robust termination condition for halting the search, (4) introducing hierarchy into the semantic types, and (5) introducing functional and inclusion dependencies into the definition of the domain relations.

Acknowledgements: We thank José Luis Ambite, Kristina Lerman, Snehal Thakkar, and Matt Michelson for useful discussions regarding this work.

References

- [Bilenko *et al.*, 2003] Mikhail Bilenko, Raymond J. Mooney, William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.
- [Cameron-Jones and Quinlan, 1994] R. Mike Cameron-Jones and J. Ross Quinlan. Efficient top-down induction of logic programs. *SIGART Bull.*, 5(1):33–42, 1994.
- [Dhamanka *et al.*, 2004] R. Dhamanka, Y. Lee, A. Doan, A. Halevy, and P. Domingos. *imap*: Discovering complex semantic matches between database schemas. In *Proceedings of SIGMOD'04*, 2004.
- [Dong *et al.*, 2004] X. Dong, A. Y. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *VLDB*, 2004.
- [Heß and Kushmerick, 2003] Andreas Heß and Nicholas Kushmerick. Learning to attach semantic metadata to web services. In *2nd International Semantic Web Conference (ISWC)*, 2003.
- [Lerman *et al.*, 2006] Kristina Lerman, Anon Plangprasopchok, and Craig A. Knoblock. Automatically labeling data used by web services. In *Proceedings of AAAI'06*, 2006.
- [Levy, 2000] Alon Y. Levy. Logic-based techniques in data integration. In Jack Minker, editor, *Logic-Based Artificial Intelligence*. Kluwer Publishers, November 2000.
- [Martin *et al.*, 2004] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing semantics to web services: The owl-s approach. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, 2004.
- [Nédellec *et al.*, 1996] C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, 1996.
- [Perkowitz and Etzioni, 1995] M. Perkowitz and O. Etzioni. Category translation: Learning to understand information on the internet. In *IJCAI-95*, 1995.
- [Roman *et al.*, 2005] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005.
- [Yan *et al.*, 2001] Ling Ling Yan, René J. Miller, Laura M. Haas, and Ronald Fagin. Data-driven understanding and refinement of schema mappings. In *SIGMOD'01*, 2001.
- [Zelle *et al.*, 1995] J. M. Zelle, C. A. Thompson, M. E. Califf, and R. J. Mooney. Inducing logic programs without explicit negative examples. In *Proceedings of the Fifth International Workshop on Inductive Logic Programming*, 1995.