Journal of Artificial Intelligence Research 30 (2007) 1-50 Submitted 11/06; published 9/07

Learning Semantic Definitions of Online Information Sources

Mark James Carman MARK@BRADIPO.NET
Craig A. Knoblock KNOBLOCK@ISI.EDU
University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

Abstract

The Internet contains a very large number of information sources providing many types of data from weather forecasts to travel deals and financial information. These sources can be accessed via Web-forms, Web Services, RSS feeds and so on. In order to make automated use of these sources, we need to model them semantically, but writing semantic descriptions for Web Services 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 of Web sources. In order to learn these definitions, our system actively invokes the sources and compares the data they produce with that of known sources of information. It then performs an inductive logic search through the space of plausible source definitions in order to learn the best possible semantic model for each new source. In this paper we perform an empirical evaluation of the system using real-world Web sources. The evaluation demonstrates the effectiveness of the approach, showing that we can automatically learn complex models for real sources in reasonable time. We also compare our system with a complex schema matching system, showing that our approach can handle the kinds of problems tackled by the latter.

1. Introduction

Recent years have seen an explosion in the quantity and variety of information available online. One can find shopping data (prices and availability of goods), geospatial data (weather forecasts, housing information), travel data (flight pricing and status), financial data (exchange rates and stock quotes), and that just scratches the surface of what is available. The aim of this work is to make use of that vast store of information.

As the amount of information has increased, so too has its reuse across Web portals and applications. Developers have realised the importance of managing content separately from presentation, leading to the development of XML as a self-describing data format. Content in XML is far easier to manipulate than HTML, simplifying integration across different sources. Standards have also emerged for providing programmatic access to data (like SOAP and REST) so that developers can easily build programs (called Mash-Ups) that combine content from different sites in real-time. Many portals now provide such access to their data and some even provide syntactic definitions (in WSDL) of the input and output these data sources expect. Missing, however, are semantic descriptions of what each source does, which is required in order to support automated data integration.

1.1 Structured Querying

Given all the structured sources available, we would like to combine the data *dynamically* to answer specific user requests (as opposed to statically in the case of Mash-Ups). Dynamic data requests can be expressed as queries such as those shown below. Such queries may require access to multiple (publicly available) data sources and combine information in ways that were not envisaged by its producers.

1. *Tourism:* Get prices and availability for all three star hotels within 100 kilometers of Trento, Italy that lie within 1 kilometer of a ski resort that has over 1 meter of snow.
2. *Transportation:* Determine the time that I need to leave work in order to catch a bus to the airport to pick up my brother who is arriving on Qantas flight 205.
3. *Disaster Prevention:* Find phone numbers for all people living within 1 mile of the coast and below 200 feet of elevation.

It is clear even from this small set of examples just how powerful the ability to combine data from disparate sources can be. In order to give these queries to an automated system, we must first express them formally in a query language such as SQL or Datalog (Ullman, 1989). In Datalog the first query might look as follows:

$\texttt{q(hotel, price)} \text{ :-}$
$\quad \texttt{accommodation(hotel}, \textit{3*}, \texttt{address)},\ \texttt{available(hotel}, \textit{today}, \texttt{price)},$
$\quad \texttt{distance(address}, \langle \textit{Trento,Italy} \rangle, \texttt{dist1)},\ \texttt{dist1} < \textit{100km},$
$\quad \texttt{skiResort(resort, loc1)},\ \texttt{distance(address, loc1, dist2)},$
$\quad \texttt{dist2} < \textit{1km},\ \texttt{snowCondiditions(resort}, \textit{today}, \texttt{height)},\ \texttt{height} > \textit{1m}.$

The expression states that hotel and price pairs are generated by looking up three star hotels in a relational table called *accommodation*, then checking the price for tomorrow night in a table called *available*. The address of each hotel is used to calculate the *distance* to Trento, which must be less than 100 kilometers. The query also checks that there is a *skiResort* within 1 kilometer of the hotel, and that the *snowConditions* for today show more than 1 meter of snow.

1.2 Mediators

A system capable of generating a plan to answer such a query is called an Information Mediator (Wiederhold, 1992). In order to generate a plan, mediators look for sources that are relevant to the query. In this case, relevant sources might be:

1. *The Italian Tourism Website:* to find all hotels near *'Trento, Italy'*.
2. *A Ski Search Engine:* to find ski resorts near each hotel.
3. *A Weather Provider:* to find out how much snow has fallen at each ski resort.

For a mediator to know which sources are relevant, it needs to know what information each source provides. While XML defines the *syntax* (formatting) used by a source, the *semantics* (intended meaning) of the information the source provides must be defined separately. This can be done using Local-as-View (LAV) source definitions in Datalog (Levy, 2000). Essentially, source definitions describe queries that if given to a mediator, will return the same data as the source provides. Example definitions are shown below. The first states that the source *hotelSearch* takes four values as input (inputs are prefixed by the \$-symbol), and returns a list of hotels which lie within the given distance of the input location. For each hotel it also returns the address as well as the price for a room on the given date. (Note that the source only provides information for hotels in Italy.)

```
hotelSearch($location, $distance, $rating, $date, hotel, address, price) :-
  country(location, Italy), accommodation(hotel, rating, address),
  available(hotel, date, price), distance(address, location, dist1),
  dist1 < distance.
findSkiResorts($address, $distance, resort, location) :-
  skiResort(resort, location), distance(address, location, dist1),
  dist1 < distance.
getSkiConditions($resort, $date, height) :-
  snowCondiditions(resort, date, height).
```

In order to generate a plan for answering the query, a mediator performs a process called query reformulation (Levy, 2000), whereby it transforms the query into a new query over (in terms of) the relevant information sources.[1] (A source is relevant if it refers to the same relations as the query.) The resulting plan in this case is shown below.

```
q(hotel, price) :-
  hotelSearch(⟨Trento,Italy⟩, 100km, 3*, today, hotel, address, price),
  findSkiResorts(address, 1km, resort, location),
  getSkiConditions(resort, today, height), height > 1m.
```

In this work, the questions of interest are: where do all the definitions for these information sources come from and more precisely, what happens when we want to add new sources to the system? Is it possible to generate these source definitions automatically?

1.3 Discovering New Sources

In the example above, the mediator knows of a set of relevant sources to use to successfully answer the query. If instead, one of those sources is missing or doesn't have the desired scope (e.g. *getSkiConditions* doesn't provide data for Trento), then the mediator first needs to discover a source providing that information. As the number and variety of information sources increase, we will undoubtedly rely on automated methods for discovering them and annotating them with semantic descriptions. In order to discover relevant sources, a system might inspect a service registry[2] (such as those defined in UDDI), or perform keyword-based search over a Web index (such as Google or del.icio.us). The research community has looked at the problem of discovering relevant services, developing techniques for classifying services into different domains (such as *weather* and *flights*) using service metadata (Heß & Kushmerick, 2003) and clustering similar services together to improve keyword-based search (Dong, Halevy, Madhavan, Nemes, & Zhang, 2004). These techniques, although useful, are not sufficient for automating service integration.

1. The complexity of query reformulation is known to be exponential, although efficient algorithms for performing it do exist (Pottinger & Halevy, 2001).
2. Note that technically, a *service* is different from a *source*. A service is an interface providing access to multiple operations, each of which may provide information. If an operation does not affect the "state of the world" (e.g. by charging somebody's credit card), then we call it an *information source*. For this paper, however, we use the term *service* to refer only to *information sources*.

1.4 Labeling Service Inputs and Outputs

Once a relevant service is discovered, the problem shifts to modeling it semantically. Modeling sources by hand can be laborious, so automating the process makes sense. Since different services often provide similar or overlapping data, it should be possible to use knowledge of previously modeled services to learn descriptions for new ones.

The first step in modeling a source is to determine what type of data it requires as input and produces as output. This is done by assigning *semantic types* (like *zipcode*, *telephone_number*, *temperature*, and so on) to the attributes of a service. Semantic types restrict the possible values of an attribute to a subset of the corresponding primitive type. The research community has investigated automating the assignment process by viewing it as a classification problem (Heß & Kushmerick, 2003). In their system, Heß and Kushmerick trained a Support Vector Machine (SVM) on metadata describing different sources. The system, given a source such as the following:

`getWeather($zip, temp)`

uses the labels `getWeather`, `zip` and `temp` (and any other available metadata) to assign types to the input and output attributes, e.g.: `zip`→*zipcode*, `temp`→*temperature*. Note that additional metadata is often useful for distinguishing between possible assignments. (If, for example, the name of the operation had been `listEmployees`, then `temp` may have referred to a temporary *employee* rather than a *temperature*.)

In subsequent work, researchers developed a more comprehensive system that used both metadata and output data to classify service attributes (Lerman, Plangprasopchok, & Knoblock, 2006). In that system, a Logistic Regression based classifier first assigns semantic types to input parameters. Examples of those input types are then used to invoke the service, and the output is given to a pattern-language based classifier, that assigns types to the output parameters. The authors argue that classification based on both data and metadata is far more accurate than that based on metadata alone. Using an example, it is easy to see why. Consider the following tuples produced by our *getWeather* source:

⟨90292, 25°C⟩, ⟨10274, 15°C⟩, ⟨60610, 18°C⟩, ...

Given the data, the classifier can be certain that `temp` really does refers to a *temperature*, and indeed can even assign it to a more specific type, *temperatureC* (in Celsius).

While the problem of determining the semantic types of a service's attributes is very interesting, and there is room for improvement on current techniques, we assume for the purposes of this work that it has already been solved.

1.5 Generating a Definition

Once we know the parameter types, we can invoke the service, but we are still unable to make use of the data it returns. To do that, we need also to know how the output attributes relate to the input (i.e. a definition for the source). For example, for the *getWeather* service we need to 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. Such relationships can be described by the following definitions:

`getWeather($zip, temp)` :- `currentTemp(zip, temp)`.
`getWeather($zip, temp)` :- `forecast(zip,` *tomorrow*`, temp)`.
`getWeather($zip, temp)` :- `averageTemp(zip,` *today*`, temp)`.

The relations used in these definitions would be defined in a *domain ontology* (or *schema*). In this paper we describe a system capable of learning which if any of these definitions is the correct. The system leverages what it knows about the domain, i.e. the domain ontology and a set of known information sources, to learn what it does not know, namely the relationship between the attributes of a newly discovered source.

1.6 Outline

This paper presents a comprehensive treatment of methods for learning semantic descriptions of Web information sources. It extends our previous work on the subject (Carman & Knoblock, 2007) by presenting detailed descriptions of the methods for both enumerating the search space and evaluating the individual candidate definitions. We provide additional details regarding the evaluation methodology and the results generated.

The paper is structured as follows. We start with an example to motivate the source induction problem and then formulate the problem concisely. We discuss our approach to learning definitions for sources in terms of other known sources of information (section 3). We give details of the search procedure for generating candidate definitions (section 4) and of the evaluation procedure for scoring candidates during search (section 5). We then describe extensions to the basic algorithm (section 6) before discussing the evaluation setup and the experiments (section 7), which demonstrate the capabilities of our system. Finally, we contrast our approach with prior work.

2. Problem

We now describe in detail the problem of learning definitions for newly discovered services. We start with a concrete example of what is meant by learning a source definition. In the example there are four types of data (semantic types), namely: *zipcodes*, *distances*, *latitudes* and *longitudes*. There are also three known sources of information. Each of the sources has a definition in Datalog as shown below. The first service, aptly named *source1*, takes in a zipcode and returns the latitude and longitude coordinates of its centroid. The second service calculates the great circle distance (the shortest distance over the earth's surface) between two pairs of coordinates, and the third converts a distance from kilometres into miles by multiplying the input by the constant 1.609.

```
source1($zip, lat, long) :- centroid(zip, lat, long).
source2($lat1, $long1, $lat2, $long2, dist) :-
        greatCircleDist(lat1, long1, lat2, long2, dist).
source3($dist1, dist2) :- multiply(dist1, 1.609, dist2).
```

The goal in this example is to learn a definition for a newly discovered service, called *source4*. This service takes two zipcodes as input and returns a distance value as output:

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

The system we will describe uses this type signature (input and output type information) to search for an appropriate definition for the source. The definition discovered in this case might be the following conjunction of calls to the individual sources:

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

The definition states that source's output distance can be calculated from the input zipcodes, by giving those zipcodes to source1, taking the resulting coordinates and calculating the distance between them using source2, and then converting that distance into miles using source3. To test whether this definition is correct, the system must invoke both 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 source.

The definition above is given in terms of the source relations, but could also have been written in terms of the domain relations (the relations used to define sources 1 to 3). To convert the definition into that form, one simply needs to replace each source relation by its definition as follows:

$\texttt{source4}(\$\texttt{zip1}, \$\texttt{zip2}, \texttt{dist})\text{:-}$
$\quad\texttt{centroid}(\texttt{zip1}, \texttt{lat1}, \texttt{long1}),\ \texttt{centroid}(\texttt{zip2}, \texttt{lat2}, \texttt{long2}),$
$\quad\texttt{greatCircleDist}(\texttt{lat1}, \texttt{long1}, \texttt{lat2}, \texttt{long2}, \texttt{dist2}),\ \texttt{multiply}(\texttt{dist1}, \mathit{1.609}, \texttt{dist2}).$

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

2.1 Problem Formulation

Having given an example of the *Source Definition Induction Problem*, we now describe the problem more formally, but before doing so, we introduce some concepts and notation. (We note that our focus in this paper is on learning definitions for information-providing, as opposed to "world-altering" services.)

- The *domain* of a *semantic data-type* t, denoted $\mathcal{D}[t]$, is the (possibly infinite) set of constant values $\{c_1, c_2, ...\}$, which constitute the set of values for variables of that type. For example $\mathcal{D}[\texttt{zipcode}] = \{\texttt{90210}, \texttt{90292}, ...\}$
- An *attribute* is a pair $\langle$*label*, *semantic data-type*$\rangle$, e.g. $\langle\texttt{zip1}, \texttt{zipcode}\rangle$. The type of attribute a is denoted *type*(a) and the corresponding domain $\mathcal{D}[\mathit{type}(a)]$ is abbreviated to $\mathcal{D}[a]$.
- A *scheme* is an ordered (finite) set of attributes $\langle a_1, ..., a_n\rangle$ with unique labels, where n is referred to as the *arity* of the scheme. An example scheme might be $\langle\texttt{zip1} : \texttt{zipcode}, \texttt{zip2} : \texttt{zipcode}, \texttt{dist} : \texttt{distance}\rangle$. The *domain* of a scheme A, denoted $\mathcal{D}[A]$, is the Cartesian product of the domains of the attributes in the scheme $\{\mathcal{D}[a_1] \times ... \times \mathcal{D}[a_n]\}$, $a_i \in A$.
- A *tuple* over a scheme A is an element from the set $\mathcal{D}[A]$. A tuple can be represented by a set of name-value pairs, such as $\{\texttt{zip1} = \texttt{90210}, \texttt{zip2} = \texttt{90292}, \texttt{dist} = \texttt{8.15}\}$

- A *relation* is a named scheme, such as $\mathtt{airDistance}(\mathtt{zip1}, \mathtt{zip2}, \mathtt{dist})$. Multiple relations may share the same scheme.
- An *extension* of a relation r, denoted $\mathcal{E}[r]$, is a subset of the tuples in $\mathcal{D}[r]$. For example, $\mathcal{E}[\mathtt{airDistance}]$ might be a table containing the distance between all zipcodes in California. (Note that the extension of a relation may only contain distinct tuples.)
- A *database instance* over a set of relations R, denoted $\mathcal{I}[R]$, is a set of extensions $\{\mathcal{E}[r_1], ..., \mathcal{E}[r_n]\}$, one for each relation $r \in R$.
- A *query language* $\mathcal{L}$ is a formal language for constructing queries over a set of relations. We denote the set of all queries that can be written using the language $\mathcal{L}$ over the set of relations R returning tuples conforming to a scheme A as $\mathcal{L}_{R,A}$.
- The *result set* produced by the execution of a query $q \in \mathcal{L}_{R,A}$ on a database instance $\mathcal{I}[R]$ is denoted $\mathcal{E}_{\mathcal{I}}[q]$.
- A *source* is a relation s, with a binding pattern $\beta_s \subseteq s$, which distinguishes *input attributes* from *output attributes*. (The output attributes of a source are denoted by the complement of the binding pattern[3], $\beta_s^c \equiv s \backslash \beta_s$.)
- A *view definition* for a source s is a query v_s written in some query language $\mathcal{L}_{R,s}$.

The *Source Definition Induction Problem* is defined as a tuple:

$$\langle T, R, \mathcal{L}, S, V, s^* \rangle$$

where T is a set of *semantic data-types*, R is a set of *relations*, $\mathcal{L}$ is a *query language*, S is a set of known *sources*, V is a set of *view definitions* (one for each known *source*), and s^* is the new *source* (also referred to as the *target*).

Each semantic type $t \in T$ must be provided with a set of examples values $E_t \subseteq \mathcal{D}[t]$. (We do not require the entire set $\mathcal{D}[t]$, because the domain of many types may be partially unknown or too large to be enumerated.) In addition, a predicate $eq_t(t, t)$ is available for checking equality between values of each semantic type to handle the case where multiple serialisations of a variable represent the same value.

Each relation $r \in R$ is referred to as a *global relation* or *domain predicate* and its extension is virtual, meaning that the extension can only be generated by inspecting every relevant data source. The set of relations R may include some interpreted predicates, such as $\leq$, whose extension is defined and not virtual.

The language $\mathcal{L}$ used for constructing queries could be any query language including SQL and XQuery (the XML Query Language). In this paper we will use a form of Datalog.

Each source $s \in S$ has an extension $\mathcal{E}[s]$ which is the complete set of tuples that can be produced by the source (at a given moment in time). We require that the corresponding view definition $v_s \in V$ (written in $\mathcal{L}_{R,s}$) is consistent with the source, such that: $\mathcal{E}[s] \subseteq \mathcal{E}_{\mathcal{I}}[v_s]$, (where $\mathcal{I}[R]$ is the current virtual database instance over the global relations). Note that we do not require equivalence, because some sources may provide incomplete data.

The view definition for the source to be modeled s^* is unknown. The solution to the *Source Definition Induction Problem* is a view definition $v^* \in \mathcal{L}_{R,s^*}$ for the source s^* such that $\mathcal{E}[s^*] \subseteq \mathcal{E}_{\mathcal{I}}[v^*]$, and there is no other view definition $v' \in \mathcal{L}_{R,s^*}$ that better describes (provides a tighter definition for) the source s^*, i.e.:

$$\neg \exists v' \in \mathcal{L}_{R,s^*} \; s.t. \; \mathcal{E}[s^*] \subseteq \mathcal{E}_{\mathcal{I}}[v'] \wedge |\mathcal{E}_{\mathcal{I}}[v']| < |\mathcal{E}_{\mathcal{I}}[v^*]|$$

3. The '\'-symbol denotes *set difference*.

Given limited available computation and bandwidth, we note that it may not be possible to guarantee that this optimality condition holds for a particular solution; thus in this paper we will simply strive to find the best solution possible.

2.2 Implicit Assumptions

A number of assumptions are implicit in the problem formulation. The first is that there exists a system capable of discovering new sources and more importantly classifying (to good accuracy) the semantic types of their input and output. Systems capable of doing this were discussed in section 1.4.

The second assumption has to do with the representation of each source as a relational view definition. Most sources on the Internet provide tree structured XML data. It may not always be obvious how best to flatten that data into a set of relational tuples, while preserving the intended meaning of the data. Consider a travel booking site which returns a set of flight options each with a ticket number, a price and a list of flight segments that constitute the itinerary. One possibility for converting this data into a set of tuples would be to break each ticket up into individual flight segment tuples (thereby obscuring the relationship between price and the number of flight segments). Another would be to create one very long tuple for each ticket with room for a number of flight segments (thereby creating tuples with many null values). In this case, it is not obvious which, if either, of these options is to be preferred. Most online data sources can, however, be modeled quite naturally as relational sources; and by first tackling the relational problem, we can develop techniques that can later be applied to the more difficult semi-structured case.

A third assumption is that the set of domain relations suffices for describing the source to be modeled. For instance, consider the case where the domain model only contains relations describing financial data, and the new source provides weather forecasts. Obviously, the system would be unable to find an adequate description of the behavior of the source and would not learn a model of it. From a practical perspective, this limitation is not a big problem, since a user can only request data (write queries to a mediator) using the relations in the domain model anyway. (Thus any source which cannot be described using those relations would not be needed to answer user requests.) In other words, the onus is on the domain modeler to model sufficient relations so as to be able to describe the types of queries a user should be able to pose to the system and consequently, the types of sources that should be available. That said, an interesting avenue for future research would be to investigate the problem of automating (at least in part) the process of expanding the scope of the domain model (by adding attributes to relations, or inventing new ones), based on the types of sources discovered.

2.3 Problem Discussion

A number of questions arise from the problem formulation, the first being where the domain model comes from. In principle, the set of semantic types and relations could come from many places. It could be taken from standard data models for the different domains, or it might just be the simplest model possible that aptly describes the set of known sources. The domain model may then evolve over time as sources are discovered for which no appropriate model can be found. Somewhat related is the question of how specific the semantic types ought to be. For example, is it sufficient to have one semantic type *distance* or should one distinguish between *distance_in_meters* and *distance_in_feet*? Generally speaking, a semantic type should be created for each attribute that is syntactically dissimilar to all other attributes. For example, a *phone_number* and a *zipcode* have very different syntax, thus operations which accept one of the types as input are unlikely to accept the other. In practice, one might create a new semantic type whenever a trained classifier can recognise the type based on its syntax alone. In general, the more semantic types there are, the harder the job of the system classifying the attributes, and the easier the job of the system tasked with learning a definition for the source.

Another question to be considered is where the definitions for the known sources come from. Initially such definitions would need to be written by hand. As the system learns definitions for new sources, they too would be added to the set of known sources, making it possible to learn ever more complicated definitions.

In order for the system to learn a definition for a new source, it must be able to invoke that source and thus needs examples of the input types. The more representative the set of examples available, the more efficient and accurate the learning process will be. An initial set of examples will need to be provided by the domain modeler. Then, as the system learns over time, it will generate a large number of examples of different semantic types (as output from various sources), which can be retained for future use.

Information Integration research has reached a point where mediator technology[4] is becoming mature and practical. The need to involve a human in the writing of source definitions is, however, the Achilles' Heel of such systems. The gains in flexibility that come with the ability to dynamically reformulate user queries are often partially offset by the time and skill required to write definitions when incorporating new sources. Thus a system capable of learning definitions automatically could greatly enhance the viability of mediator technology. This motivation alone seems sufficient for pursuing the problem.

3. Approach

The approach we take to learning semantic models for information sources on the Web is twofold. Firstly, we choose to model sources using the powerful language of conjunctive queries. Secondly, we leverage the set of known sources in order to learn a definition for the new one. In this section we discuss these aspects in more detail.

3.1 Modeling Language

The source definition language $\mathcal{L}$ is the hypothesis language in which new definitions will need to be learnt. As is often the case in machine learning, we are faced with a trade-off with respect to the expressiveness of this language. If the hypothesis language is too simple, then we may not be able to model *real* services using it. On the other hand, if the language is overly complex, then the space of possible hypotheses will be so large that learning will not be feasible. The language we choose is that of conjunctive queries in Datalog, which is

4. Influential Information Integration systems include TSIMMIS (Garcia-Molina, Hammer, Ireland, Papakonstantinou, Ullman, & Widom, 1995), SIMS (Arens, Knoblock, & Shen, 1996), InfoMaster (Duschka, 1997), and Ariadne (Knoblock, Minton, Ambite, Ashish, Muslea, Philpot, & Tejada, 2001).

a highly expressive relational query language. In this section we argue why a less expressive language is not sufficient for our purposes.

Researchers interested in the problem of assigning semantics to Web Services (Heß & Kushmerick, 2003) have investigated the problem of using Machine Learning techniques to classify services (based on metadata characteristics) into different semantic domains, such as *weather* and *flights*, and the operations they provide into different classes of operation, such as *weatherForecast* and *flightStatus*. From a relational perspective, we can consider the different classes of operations as relations. For instance, consider the definition below:

$\texttt{source}(\$\texttt{zip}, \texttt{temp})$:- $\texttt{weatherForecast}(\texttt{zip}, \textit{tomorrow}, \texttt{temp})$.

The source provides weather data by selecting tuples from a relation called *weatherForecast*, which has the desired zipcode and date equal to *tomorrow*. This query is referred to as a *select-project* query because its evaluation can be performed using the relational operators *selection* and *projection*. So far so good, we have been able to use a simple classifier to learn a simple definition for a source. The limitation imposed by this restricted (select-project) modeling language becomes obvious, however, when we consider slightly more complicated sources. Consider a source that provides the temperature in Fahrenheit as well as Celsius. In order to model such a source using a select-project query, we would require that the *weatherForecast* relation be extended with a new attribute as follows:

$\texttt{source}(\$\texttt{zip}, \texttt{tempC}, \texttt{tempF})$:- $\texttt{weatherForecast}(\texttt{zip}, \textit{tomorrow}, \texttt{tempC}, \texttt{tempF})$.

The more attributes that could conceivably be returned by a weather forecast operation (such as dewpoint, humidity, temperature in Kelvin, latitude, etc.), the longer the relation will need to be to cover them all. Better, in this case, would be to introduce a second relation *convertCtoF* that makes explicit the relationship between the temperature values. If, in addition, the source limits its output to zipcodes in California, a reasonable definition for the source might be:

$\texttt{source}(\$\texttt{zip}, \texttt{tempC}, \texttt{tempF})$:-
  $\texttt{weatherForecast}(\texttt{zip}, \textit{tomorrow}, \texttt{tempC})$, $\texttt{convertCtoF}(\texttt{tempC}, \texttt{tempF})$,
  $\texttt{state}(\texttt{zip}, \textit{California})$.

This definition is no longer expressed in the language of select-project queries, because it now involves multiple relations and joins across them. Thus from this simple example, we see that modeling services using simple select-project queries is not sufficient for our purposes. What we need are *select-project-join* queries, also referred to as conjunctive queries.[5] The reader has already been introduced to examples of conjunctive queries throughout the previous sections. Conjunctive queries form a subset of the logical query language Datalog and can be described more formally as follows:

> A *conjunctive query* over a set of relations R is an expression of the form:
> $q(X_0)$:- $r_1(X_1),\ r_2(X_2),\ ...,\ r_l(X_l)$.
> where each $r_i \in R$ is a relation and X_i is an ordered set of variable names of size $arity(r_i)$.[6] Each conjunct $r_i(X_i)$ is referred to as a *literal*. The set of variables in the query, denoted $vars(q) = \bigcup_{i=0}^{l} X_i$, consists of *distinguished* variables X_0 (from the head of the query), and *existential* variables $vars(q)\backslash X_0$, (which

5. Evaluating a *select-project-join* query requires additional relational operators: *natural join* and *rename*.
6. Note that a *conjunctive query* can also be expressed in *first order logic* as follows:
$\forall X_0\ \exists Y\ s.t.\ r_1(X_1) \wedge r_2(X_2) \wedge ... \wedge r_l(X_l)\ \rightarrow\ q(X_0)$ where $X_0 \cup Y = \bigcup_{i=1}^{l} X_i$

only appear in the body). A conjunctive query is said to be *safe* if all the distinguished variables appear in the body, i.e. $X_0 \subseteq \bigcup_{i=1}^{l} X_i$.

3.2 More Expressive Languages

Modeling sources using conjunctive queries implies that aggregate operators like *MIN* and *ORDER* cannot be used in source definitions. The functionality of most sources can be described without such operators. Some sources can only be described poorly, however. Consider a hotel search service that returns the 20 closest hotels to a given location:

`hotelSearch($loc, hotel, dist)` :-
  `accommodation(hotel, loc1), distance(loc, loc1, dist)`.

According to the definition, the source should return all hotels regardless of distance. One cannot express the fact that only the closest hotels will be returned. The reason for not including aggregate operators in the hypothesis language is that the search space associated with learning definitions is prohibitively large. (Thus we leave aggregate operators to future work as discussed in section 9.2.)

Similarly, source definitions cannot contain disjunction, which rules out *union* and *recursive* queries. Again, this simplifying assumption holds for most information sources and greatly reduces the search space. It means however, that a weather service providing forecasts only for cities in the US and Canada would be modeled as:

`s($city, temp)` :- `forecast(city, country,` *tomorrow*, `temp)`.

Since the definition does not restrict the domain of the *country* attribute, when confronted with a request for the forecast in Australia, a mediator would proceed to call the service, oblivious to any restriction on that attribute.

We also do not allow negation in the queries because source definitions very rarely require it, so including it would needlessly complicate the search. In those rare cases where the negation of a particular predicate is useful for describing certain types of sources, the negated predicate can be included (as a distinct predicate) in the search. For instance, we might use "$\geq$" to describe a source, even though strictly speaking it is the negation of "$<$".

3.3 Leveraging Known Sources

Our approach to the problem of discovering semantic definitions for new services is to leverage the set of known sources when learning a new definition. Broadly speaking, we do this by invoking the known sources (in a methodical manner) to see if any combination of the information they provide matches the information provided by the new source. From a practical perspective, this means in order to model a newly discovered source semantically, we require some overlap in the data being produced by the new source and the set of known sources. One way to understand this is to consider a new source producing weather data. If none of the known sources produce any weather information, then there is no way for the system to learn whether the new source is producing historical weather data, weather forecasts - or even that it is describing weather at all. (In principle, one could try to guess what the service is doing based on the type signature alone, but there would be no guarantee that the definition was correct, making it of little use to a mediator.) Given this *overlapping data requirement*, one might claim that there is little benefit in incorporating new sources. We detail some of the reasons why this is not the case below.

The most obvious benefit of learning definitions for new sources is redundancy. If the system is able to learn that one source provides exactly the same information as a currently available source, then if the latter suddenly becomes unavailable, the former can be used in its place. For example, if a mediator knows of one weather source providing current conditions and learns that a second source provides the same or similar data, then if the first goes down for whatever reason (perhaps because an access quota has been reached), weather data can still be accessed from the second.

The second and perhaps more interesting reason for wanting to learn a definition for a new source is that the new source may provide data which lies outside the scope of (or simply was not present in) the data provided by the other sources. For example, consider a weather service which provides temperature values for zipcodes in the United States. Then consider a second source that provides weather forecasts for cities worldwide. If the system can use the first source to learn a definition for the second, the amount of information available for querying increases greatly.

Binding constraints on a service can make accessing certain types of information difficult or inefficient. In this case, discovering a new source providing the same or similar data but with a different binding pattern may improve performance. For example, consider a hotel search service that accepts a zipcode and returns a set of hotels along with their star rating:

`hotelSearch`(`$zip`, `hotel`, `rating`, `street`, `city`, `state`):-
  `accommodation`(`hotel`, `rating`, `street`, `city`, `state`, `zip`).

Now consider a simple query for the names and addresses of all five star hotels in California:

`q`(`hotel`, `street`, `city`, `zip`):- `accommodation`(`hotel`, *5**, `street`, `city`, *California*, `zip`).

Answering this query would require thousands of calls to the known source, one for every zipcode in California, and a mediator could only answer the query if there was another source providing those zipcodes. In contrast, if the system had learnt a definition for a new source which provides exactly the same data but with a different binding pattern (such as the one below), then answering the query would require only one call to a source:

`hotelsByState`(`$state`, `$rating`, `hotel`, `street`, `city`, `zip`):-
  `accommodation`(`hotel`, `rating`, `street`, `city`, `state`, `zip`).

Often the functionality of a complex source can be described in terms of a composition of the functionality provided by other simpler services. For instance, consider the motivating example from section 2, in which the functionality provided by the new source was to calculate the distance in miles between two zipcodes. The same functionality could be achieved by performing four different calls to the available sources. In that case, the definition learnt by the system meant that any query regarding the distance between zipcodes could be handled more efficiently. In general, by learning definitions for more complicated sources in terms of simpler ones, the system can benefit from computation, optimisation and caching abilities of services providing complex functionality.

Finally, the newly discovered service may be faster to access than the known sources providing similar data. For instance, consider a geocoding service that takes in an address and returns the latitude and longitude coordinates of the location. Because of the variety in algorithms used to calculate the coordinates, it's not unreasonable for some geocoding services to take a very long time (upwards of one second) to return a result. If the system were able to discover a new source providing the same geocoding functionality, but using a faster algorithm, then it could locate and display many more addresses on a map in the same amount of time.

4. Inducing Definitions

In this section we describe an algorithm for generating candidate definitions for a newly discovered source. The algorithm forms the first phase in a generate and test methodology for learning source definitions. We defer discussion of the testing phase to later in the paper. We start by briefly discussing work on relational rule learning and then describe how our algorithm builds upon these ideas.

4.1 Inductive Logic Programming

The language of conjunctive queries is a restricted form of first-order logic. In the Machine Learning community, systems capable of learning models using first-order representations are referred to as Inductive Logic Programming (ILP) systems or relational rule learners. Because of the expressiveness of the modeling language, the complexity of learning is much higher than for propositional rule learners (also called attribute-value learners), which form the bulk of Machine Learning algorithms. Given our relational modeling of services, many of the techniques developed in ILP should also apply to our problem.

The First Order Inductive Learner (FOIL) is a well known ILP search algorithm (Cameron-Jones & Quinlan, 1994). It is capable of learning first-order rules to describe a target predicate, which is represented by a set of positive examples (tuples over the target relation, denoted E^+) and optionally also a set of negative examples (E^-). The search for a viable definition in FOIL starts with an empty clause[7] and progressively adds literals to the body (antecedent) of the rule, thereby making the rule more specific. This process continues until the definition (denoted h) covers only positive examples and no negative examples:

$$E^+ \cap \mathcal{E}_\mathcal{I}[h] \neq \emptyset \quad \text{and} \quad E^- \cap \mathcal{E}_\mathcal{I}[h] = \emptyset$$

Usually a set of rules is learnt in this manner by removing the positive examples covered by the first rule and repeating the process. (The set of rules is then interpreted as a union query.) Search in FOIL is performed in a greedy best-first manner, guided by an information gain-based heuristic. Many extensions to the basic algorithm exist, most notably those that combine declarative background knowledge in the search process such as FOCL (Pazzani & Kibler, 1992). Such systems are categorised as performing a *top down* search because they start from an empty clause (the most general rule possible) and progressively specialize the clause. Bottom up approaches, on the other hand, such as GOLEM (Muggleton & Feng, 1990), perform a specific to general search starting from the positive examples of the target.

4.2 Search

We now describe the actual search procedure we use to generate candidate definitions for a new source. The procedure is based on the top-down search strategy used in FOIL. The algorithm takes as input a type signature for the new source and uses it to seed the search for

7. We use the terms *clause* and *query* interchangeably to refer to a *conjunctive query* in Datalog. An empty clause is a query without any literals in the body (right side) of the clause.

```
    input  : A predicate signature $s^*$
    output: The best scoring view definition $v_{best}$
1  invoke target with set of random inputs;
2  $v_{best} \leftarrow$ empty clause $s^*$;
3  add $v_{best}$ to empty $queue$;
4  while $queue \neq \emptyset \ \wedge\ time() < timeout \ \wedge\ i{+}{+} < limit$ do
5      $v_0 \leftarrow$ best definition from $queue$;
6      forall $v_1 \in expand(v_0)$ do
7          insert $v_1$ into $queue$;
8          while $\Delta eval(v_1) > 0$ do
9              forall $v_2 \in constrain(v_1)$ do
10                 insert $v_2$ into $queue$;
11                 if $eval(v_2) \geq eval(v_1)$ then $v_1 \leftarrow v_2$;
12             end
13         end
14         if $eval(v_1) \geq eval(v_{best})$ then $v_{best} \leftarrow v_1$;
15     end
16 end
17 return $v_{best}$;
```

Algorithm 1: Best-first search through the space of candidate source definitions.

candidate definitions. (We will refer to the new source relation as the *target predicate* and the set of known source relations as *source predicates*.) The space of candidate definitions is enumerated in a best-first manner, with each candidate tested to see if the data it returns is similar to the target. Pseudo-code describing the procedure is given in Algorithm 1.[8]

The first step in the algorithm is to invoke the new source with a representative set of input tuples to generate examples of output tuples that characterise the functionality of the source. This set of invocations must include positive examples (invocations for which output tuples were produced) and if possible, also negative tuples (inputs for which no output was returned). The algorithm's ability to induce the correct definition for a source depends greatly on the number of positive examples available. Thus a minimum number of positive invocations of the source is imposed, meaning that the algorithm may have to invoke the source repeatedly using different inputs until sufficient positive invocations can be recorded. Selecting appropriate input values so as to successfully invoke a service is easier said than done. We defer discussion of the issues and difficulties involved in successfully invoking the new source to section 6.1, and assume for the moment that the induction system is able to generate a table of values that represent its functionality.

The next step in the algorithm is to initialise the search by adding an empty clause to the queue of definitions to expand. The rest of the algorithm is simply a best-first search procedure. At each iteration the highest scoring but not yet expanded definition (denoted v_0) is removed from the queue and expanded by adding a new predicate to the end of the

8. The implementation of this algorithm used in the experiments of section 7.3 is available at: http://www.isi.edu/publications/licensed-sw/eidos/index.html

clause (see the next section for an example). Each candidate generated (denoted v_1) is added to the queue. The algorithm then progressively constrains the candidate by binding variables of the newly added predicate, (see section 4.4). The *eval* function (see section 5.3) evaluates the quality of each candidate produced. The procedure stops constraining the candidate when the change in the evaluation function ($\Delta eval$) drops to zero. It then compares v_1 to the previous best candidate v_{best} and updates the latter accordingly.

In principle the algorithm should terminate when the "perfect" candidate definition is discovered - one that produces exactly the same data as the target. In practice that never occurs because the sources are incomplete (don't perfectly overlap with each other) and are noisy. Instead the algorithm terminates when either the queue becomes empty, a time limit is reached or a maximum number of iterations has been performed.

4.3 An Example

We now run through an example of the process of generating candidate definitions. Consider a newly discovered source, which takes in a zipcode and a distance, and returns all the zipcodes that lie within the given radius (along with their respective distances). The target predicate representing this source is:

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

Now assume that there are two known sources. The first is the source for which the definition was learnt in the example from section 2, namely:

```
source4($zip1, $zip2, dist):-
  centroid(zip1, lat1, long1), centroid(zip2, lat2, long2),
  greatCircleDist(lat1, long1, lat2, long2, dist2), multiply(dist1, 1.609, dist2).
```

The second source isn't actually a source but an interpreted predicate:

$\leq$(dist1, dist2).

The search for a definition for the new source might then proceed as follows. The first definition to be generated is the empty clause:

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

The null character (_) represents the fact that none of the inputs or outputs have any restrictions placed on their values. Prior to adding the first literal (source predicate), the system will check whether any output attributes echo the input values. In this case, given the semantic types, two possibilities need to be checked:

```
source5($zip1, $_, zip1, _).
source5($_, $dist1, _, dist1).
```

Assuming neither of these possibilities is true (i.e. improves the score), then literals will be added one at a time to refine the definition. A literal is a source predicate with an assignment of variable names to its attributes. A new definition must be created for every possible literal that includes at least one variable already present in the clause. (For the moment we ignore the issue of binding constraints on the sources being added.) Thus many candidate definitions would be generated, including the following:

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

Note that the semantic types in the type signature of the target predicate limit greatly the number of candidate definitions that can be produced. The system then evaluates each of these candidates in turn, selecting the best one for further expansion. Assuming the first of the three has the best score, it would be expanded by adding another literal, forming more complicated candidates such as the following:

$\texttt{source5(\$zip1,\$dist1,\_,dist2) :- source4(\$zip1,\$\_,dist1)}, \quad \leq\texttt{(dist1,dist2)}$.

This process continues until the system discovers a definition which perfectly describes the source, or is forced to backtrack because no literal improves the score.

4.4 Iterative Expansion

The sources used in the previous example all had relatively low arity. On the Internet this is rarely the case, with many sources producing a large number of attributes of the same type. This is a problem because it causes an exponential number of definitions to be possible at each expansion step. Consider for instance a stock price service, which provides the current, high, low, market-opening and market-closing prices for a given ticker symbol. The type signature for that service would be:

`stockprice($ticker, price, price, price, price, price)`

If the definition to which this predicate is to be added already contains k distinct price variables, then the number of ways in which the price attributes of the new relation can be assigned variable names is $\sum_{i=0}^{5} \binom{5}{i} k^i$, which is prohibitively large even for moderate k.[9] To limit the search space in the case of such high-arity predicates, we first generate candidates with a minimal number of bound variables in the new literal and progressively constrain the best performing of these definitions within each expansion. (High arity predicates are handled in a similar fashion in FOIL, Quinlan and Cameron-Jones, 1993.) For example, consider using the source above to learn a definition for a new source with signature:

`source6($ticker, price, price)`

We start by adding literals to an empty definition as before. This time though, instead of generating a literal for every possible assignment of variable names to the attributes of each relation, we generate only the simplest assignments such that all of the binding constraints are met. (This is the *expand* procedure referred to in Algorithm 1.) In this example, the ticker symbol input of the *stockprice* source would need to be bound, generating a single definition:

`source6($tic, _, _) :- stockprice($tic, _, _, _, _, _).`

This definition would be evaluated, and then more constrained definitions are generated by equating a variable from the same literal to other variables from the clause. (This is the *constrain* procedure from Algorithm 1.) Two such definitions are shown below:

`source6($tic, pri1, _) :- stockprice($tic, pri1, _, _, _, _).`
`source6($tic, pri1, _) :- stockprice($tic, _, pri1, _, _, _).`

The best of these definitions would then be selected and constrained further, generating definitions such as:

9. Intuitively, one can assign variable names to i attributes using k labels in k^i different ways. One can choose i of the 5 attributes in $\binom{5}{i}$ ways, and one can do that for $i = 0, 1, .., 5$. See Weber, Tausend, and Stahl (1995) for a detailed discussion of the size of the hypothesis space in ILP.

```
source6($tic, pri1, pri2) :- stockprice($tic, _, pri1, pri2, _, _).
source6($tic, pri1, pri2) :- stockprice($tic, _, pri1, _, pri2, _).
```

In this way, the best scoring literal can be found without the need to iterate over all of the possible assignments of variables to attributes.

4.5 Domain Predicates vs. Source Predicates

In the examples of sections 4.3 and 4.4, the decision to perform search over the source predicates rather than the domain predicates was made in an arbitrary fashion.[10] In this section we justify that decision. If one were to perform the search over the domain predicates rather than the source predicates, then testing each definition would require an additional *query reformulation* step. For example, consider the following candidate definition for *source*5 containing the domain predicate *centroid*:

```
source5($zip1, $_, _, _) :- centroid(zip1, _, _).
```

In order to evaluate this candidate, the system would need to first treat the definition as a query and reformulate it into a set of rewritings (that together form a union query) over the various sources as follows:

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

This union query can then be executed against the available sources (in this case just *source*4) to see what tuples the candidate definition returns. In practice however, if the definitions for the known sources contain multiple literals (as they normally do) and the domain relations are of high-arity (as they often are), then the search over the space of conjunctions of domain predicates is often much larger than the corresponding search over the space of conjunctions of source predicates. This is because multiple conjunctions of domain predicates (candidate definitions) end up reformulating to the same conjunction of source predicates (union queries). For example, consider the following candidate definitions written in terms of the domain predicates:

```
source5($zip1, $_, _, _) :- centroid(zip1, lat, _), greatCircleDist(lat, _, _, _, _).
source5($zip1, $_, _, _) :- centroid(zip1, _, lon), greatCircleDist(_, lon, _, _, _).
source5($zip1, $_, _, _) :- centroid(zip1, lat, lon), greatCircleDist(lat, lon, _, _, _).
```

All three candidates would reformulate to the same query over the sources (shown below), and thus are indistinguishable given the sources available.

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

In general, the number of candidate definitions that map to the same reformulation can be exponential in the number of hidden variables present in the definitions of the known sources. For this reason, we simplify the problem and search the space of conjunctions of source predicates. In some sense, performing the search over the source predicates can be seen as introducing a "similarity heuristic" which focuses the search toward definitions with similar structure to the definitions of the available sources. We note that the definitions produced can (and will) later be converted to queries over the global predicates by unfolding and

10. A note on the difference between *domain*, *source* and *interpreted* predicates. Domain predicates are invented by a domain expert for use in modeling a particular information domain. They define a common schema that can be used for describing information from different sources. Source predicates represent the sources available in the system. Interpreted predicates, (such as "≤"), are a special type of domain predicate, that can be treated as source predicates, since their meaning is interpreted (understood).

possibly tightening them to remove redundancies. We will discuss the process of tightening the unfoldings in section 6.6.

4.6 Limiting the Search

The search space generated by this top-down search algorithm may be very large even for a small number of sources. The use of semantic types limits greatly the ways in which variables within each definition can be equated (aka the join paths) and thus goes a long way to reduce the size of the search space. Despite this reduction, 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. Such limitations are often referred to as *inductive search bias* or *language bias* (Nédellec, Rouveirol, Adé, Bergadano, & Tausend, 1996).

An obvious way to limit the search is to restrict the number of source predicates that can occur in a definition. Whenever a definition reaches the maximum length, backtracking can be performed, allowing the search to escape from local minima that may result from greedy enumeration. The assumption here is that shorter definitions are more probable than longer ones, which makes sense since service providers are likely to provide data in the simplest form possible. Moreover, the simpler the definition learnt, the more useful it will be to a mediator, so we decide to trade completeness (the ability to express longer definitions) for improved accuracy over shorter definitions.

The second restriction placed on candidate definitions is to limit the number of times the same source predicate appears in a given candidate. This makes sense because the definitions of real services tend not to contain many repeated predicates. Intuitively, this is because most services provide raw data without performing many calculations upon it. Repeated use of the same predicate in a definition is more useful for describing some form of calculation than raw data itself. (Exceptions to this rule exist, for example predicates representing unit conversion functionality such as Fahrenheit to Celsius, may necessarily occur multiple times in the definition of a source.)

The third restriction limits the complexity of the definitions generated by reducing the number of literals that do not contain variables from the head of the clause. Specifically, it limits the *level of existential quantification* (sometimes also referred to as the *depth*, Muggleton and Feng, 1990) of each variable in a clause. This level is defined to be zero for all *distinguished* variables (those appearing in the head of the clause). For *existential* variables it is defined recursively as one plus the lowest level of any variable appearing in the same literal. For example, the candidate definition shown below has a maximum existential quantification level of three because the shortest path from the last literal to the head literal (via join variables) passes through two other literals:

$\texttt{source5}(\$\texttt{zip1}, \$\_, \_, \_) \text{ :- } \texttt{source4}(\$\texttt{zip1}, \$\_, \texttt{d1}), \texttt{source3}(\$\texttt{d1}, \texttt{d2}), \texttt{source3}(\$\texttt{d2}, \_).$

The effect of this bias is to concentrate the search around simpler but highly connected definitions, where each literal is closely linked to the input and output of the source.

The fourth restriction placed on source definitions is that they are executable. More specifically, it should be possible to execute them from left to right, meaning that the inputs of each source appear either in the target predicate (head of the clause) or in one of the literals to the left of that literal. For example, of the two candidate definitions shown below, only the first is executable. The second definition is not, because `zip2` is used as input for *source4* in the first literal, without first being bound to a value in the head of the clause:

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

This restriction serves two purposes. Firstly, like the other biases, it limits the size of the search space. Secondly, it makes it easier to evaluate the definitions produced. In theory, one could still evaluate the second definition above by generating lots of input values for `zip2`, but that would require a lot of invocations for minimal gain.

The last restriction reduces the search space by limiting the number of times the same variable can appear in any given literal in the body of the clause. Definitions in which the same variable appears multiple times in a given literal, such as in the following example which returns the distance between a zipcode and itself, are not very common in practice:

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

Explicitly preventing such definitions from being generated makes sense because sources requiring them are so rare, that it is better to reduce the search space exponentially by ignoring them, than to explicitly check for them each time.

5. Scoring Definitions

We now proceed to the problem of evaluating the candidate definitions generated during search. The basic idea is to compare the output produced by the source with the output produced by the definition on the same input. The more similar the set of tuples produced, the higher the score for the candidate. The score is then averaged over a set of different input tuples to see how well the candidate definition describes the new source.

In the motivating example of section 2, the source for which a definition was being learnt (the definition is repeated below) only produced one output tuple ⟨`dist`⟩ for every input tuple ⟨`zip1, zip2`⟩:

```
source4($zip1, $zip2, dist):-
  centroid(zip1, lat1, long1), centroid(zip2, lat2, long2),
  greatCircleDist(lat1, long1, lat2, long2, dist2),
  multiply(dist1, 1.6093, dist2).
```

This fact made it simple to compare the output of the service with the output of the induced definition. In general however, the source to be modeled (and the candidate definitions modeling it) may produce multiple output tuples for each input tuple. Take for example *source5* from section 4.3, which produces the set of output tuples ⟨`zip2, dist2`⟩ containing all the zipcodes which lie within a given radius of the input zipcode ⟨`zip1, dist1`⟩. In such cases, the system needs to compare a set of output tuples with the set produced by the definition to see if any of the tuples are the same. Since both the new source and the existing sources may not be complete, the two sets may simply 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 need a measure that tells us how well a candidate hypothesis describes the data returned by a source. One such measure is the following:

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

where s is the new source, v is a candidate source definition, and $I \subseteq \mathcal{D}[\beta_s]$ is the set of input tuples used to test the source (β_s is the set of *input attributes* of source s). $O_s(i)$ denotes the set of tuples returned by the new source when invoked with input tuple i. $O_v(i)$ is the corresponding set returned by the candidate definition. Using relational projection (π) and selection (σ) operators and the notation introduced in section 2.1, these sets can be written as follows. (Note that β_s^c represents the *output attributes* of s.)

$$O_s(i) \equiv \pi_{\beta_s^c}(\sigma_{\beta_s=i}(\mathcal{E}[s])) \quad \text{and} \quad O_v(i) \equiv \pi_{\beta_s^c}(\sigma_{\beta_s=i}(\mathcal{E}_{\mathcal{I}}[v]))$$

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 number of common tuples divided by the number of tuples produced by the definition. The above measure takes both precision and recall into account by calculating the average *Jaccard similarity* between the sets. The table below gives an example of how this score is calculated for each input tuple.

input tuple $i \in I$	*actual* output tuples $O_s(i)$	*predicted* output tuples $O_v(i)$	Jaccard similarity for tuple i
$\langle a, b\rangle$	$\{\langle x, y\rangle, \langle x, z\rangle\}$	$\{\langle x, y\rangle\}$	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!

The first two rows of the table show inputs for which the predicted and actual output tuples overlap. In the third row, the definition produces exactly the same set of tuples as the source being modeled and thus gets the maximum score. 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 by 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 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. At the same time, less constrained definitions that are better at predicting the output tuples on average may score poorly. For example, consider a source which returns weather forecasts for zipcodes in Los Angeles:

source($zip, temp) :- forecast(zip, *tomorrow*, temp), UScity(zip, *Los Angeles*).

Now consider two candidate definitions for the source. The first returns the temperature for a zipcode, while the second returns the temperature only if it is below $0°C$:

v_1($zip, temp) :- forecast(zip, *tomorrow*, temp).

v_2($zip, temp) :- forecast(zip, *tomorrow*, temp), temp < *0° C*.

Assume that the source and candidates are invoked using 20 different randomly selected zipcodes. For most zipcodes, the source will not return any output, because the zipcode will lie outside of Los Angeles. The first candidate will likely return output for all zipcodes, while the second candidate would, like the source, only rarely produce any output. This is because the temperature in most zipcodes will be greater than zero, and has nothing to do with whether or not the zipcode is in Los Angeles. If we score definitions highly when they correctly produce no output, the system would erroneously prefer the second candidate over the first (because the latter often produces no output). To prevent that from happening, we simply ignore inputs for which the definition correctly predicts zero tuples. This is the same as setting the score to be the average of the other values.

Returning our attention to the table, after ignoring the last row, the overall score for this definition would be calculated as 0.46.

5.1 Partial Definitions

As the search proceeds toward the correct definition for the service, many semi-complete (unsafe) definitions will be generated. These definitions will not produce values for all attributes of the target tuple but only a subset of them. For example, the candidate:

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

produces only one of the two output attributes produced by the source. 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, however, two reasons why this would not make sense:

- The space of safe definitions is too large to enumerate, and thus we need to compare partial definitions to guide the search toward the correct definition.
- 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.

The simplest way to compute a score for a partial definition is to compute the same function as before, but instead of using the raw source tuples, projecting them over the subset of attributes that are produced by the definition. This revised score is shown below. (Note that the projection is over $v\backslash\beta_s$, which denotes the subset of output attributes of s which are produced by the view definition v. Note also that the projection is not distinct, i.e. multiple instances of the same tuple may be produced.)

$$score_2(s, v, I) = \frac{1}{|I|}\sum_{i\in I}\frac{|\pi_{v\backslash\beta_s}(O_s(i)) \cap O_v(i)|}{|\pi_{v\backslash\beta_s}(O_s(i)) \cup O_v(i)|}$$

This revised score is not very useful however, as it gives an unfair advantage to definitions that do not produce all of the output attributes of the source. This is because it is far easier to correctly produce a subset of the output attributes than to produce all of them. Consider for example the two source definitions shown below. The two definitions are identical except that the second returns the output distance value *dist2*, while the first does not:

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

Since the two are identical, the projection over the subset will in this case return the same number of tuples. This means that both definitions would get the same score although the second definition is clearly better than the first since it produces all of the required outputs.

We need to be able to penalise partial definitions in some way for the attributes they don't produce. One way to do this is to first calculate the size of the domain $|\mathcal{D}[a]|$ of each of the missing attributes. In the example above, the missing attribute is the distance value. Since distance is a continuous value, calculating the size of its domain is not obvious. We can approximate the size of its domain by:

$$|\mathcal{D}[distance]| \approx \frac{max - min}{accuracy}$$

where *accuracy* is the error-bound on distance values. (We will discuss error-bounds further in section 5.4.) Note that the cardinality calculation may be specific to each semantic type. Armed with the domain size, we can penalise the score for the definition by dividing it by the product of the size of the domains of all output attributes not generated by the definition. In essence, we are saying that all possible values for these extra attributes have been "allowed" by this definition. This technique is similar to techniques used for learning without explicit negative examples (Zelle, Thompson, Califf, & Mooney, 1995).

The set of missing output attributes is given by the expression $\beta_s^c \backslash v$, thus the penalty for missing attributes is just the size of the domain of tuples of that scheme, i.e.:

$$penalty = |\mathcal{D}[\beta_s^c \backslash v]|$$

Using this penalty value we can calculate a new score, which takes into account the missing attributes. Simply dividing the projected score by the penalty would not adhere to the intended meaning of compensating for the missing attribute values, and thus may skew the results. Instead, we derive a new score by introducing the concept of *typed dom predicates* as follows:

> A *dom predicate* for a semantic data-type t, denoted dom_t, is a single arity relation whose extension is set to be the domain of the datatype, i.e. $\mathcal{E}[dom_t] = \mathcal{D}[t]$. Similarly, a dom predicate for a scheme A, denoted dom_A, is a relation over A whose extension is $\mathcal{E}[dom_A] = \mathcal{D}[A]$.

Dom predicates were introduced by Duschka to handle the problem of *query reformulation* in the presence of sources with *binding constraints* (Duschka, 1997). (In that work the predicates were not typed, although typing would have resulted in a more efficient algorithm.) Here we use them to convert a partial definition v into a safe (complete) definition v'. We can do this simply by adding a dom predicate to the end of the view definition that generates values for the missing attributes. For the example above, v' would be:

$\texttt{source5}(\$\texttt{zip1}, \$\texttt{dist1}, \texttt{zip2}, x)$:-
  $\texttt{source4}(\$\texttt{zip1}, \$\texttt{zip2}, \texttt{dist2}),\ \leq(\texttt{dist2}, \texttt{dist1}),\ \texttt{dom}_{\texttt{distance}}(x)$.

where x is a new variable of type *distance*. The new view definition v' is safe, because all the variables in the head of the clause also appear in the body. In general, we can turn any unsafe view definition v into a safe definition v' by appending a dom predicate $dom_{\beta_s^c \backslash v}(x_1, ..., x_n)$, where each x_i is a distinguished variable (from the head of the clause) corresponding to an output attribute of v' that wasn't bound in v. Now we can use this complete definition to calculate the score as before:

$$score_3(s, v, I) = score(s, v', I) = \frac{1}{|I|} \sum_{i \in I} \frac{|O_s(i) \cap O_{v'}(i)|}{|O_s(i) \cup O_{v'}(i)|}$$

which can be rewritten (by expanding the denominator) as follows:

$$score_3(s,v,I) = \frac{1}{|I|}\sum_{i\in I}\frac{|O_s(i)\cap O_{v'}(i)|}{|O_s(i)| + |O_{v'}(i)| - |O_s(i)\cap O_{v'}(i)|}$$

We can then remove the references to v' from this equation by considering:

$$O_{v'}(i) = O_v(i)\times \mathcal{E}[dom_{\beta_s^c\backslash v}] = O_v(i)\times \mathcal{D}[\beta_s^c\backslash v]$$

Thus the size of the set is given by $|O_{v'}(i)| = |O_v(i)||\mathcal{D}[\beta_s^c\backslash v]|$ and the size of the intersection can be calculated by taking the projection over the output attributes produced by v:

$$|O_s(i)\cap O_{v'}(i)| = |\pi_{v\backslash\beta_s}(O_s(i)\cap O_v(i)\times\mathcal{D}[\beta_s^c\backslash v])| = |\pi_{v\backslash\beta_s}(O_s(i))\cap O_v(i)|$$

Substituting these cardinalities into the score function given above, we arrive at the following equation for the penalised score:

$$score_3(s,v,I) = \frac{1}{|I|}\sum_{i\in I}\frac{|\pi_{v\backslash\beta_s}(O_s(i))\cap O_v(i)|}{|O_s(i)| + |O_v(i)||\mathcal{D}[\beta_s^c\backslash v]| - |\pi_{v\backslash\beta_s}(O_s(i))\cap O_v(i)|}$$

5.2 Binding Constraints

Some of the candidate definitions generated during the search may have different binding constraints from the target predicate. For instance in the partial definition shown below, the variable *zip2* is an output of the target source, but an input to *source4*:

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

From a logical perspective, in order to test this definition correctly, we need to invoke *source4* with every possible value from the domain of zipcodes. Doing this is not practical for two reasons: firstly, the system may not have a complete list of zipcodes at its disposal. Secondly and far more importantly, invoking *source4* with thousands of different zipcodes would take a very long time and would probably result in the system being blocked from further use of the service. So instead of invoking the same source thousands of times, we approximate the score for this definition by sampling from the domain of zipcodes and invoking the source using the sampled values. We then compensate for this sampling by scaling (certain components of) the score by the ratio of the sampled zipcodes to the entire domain. Considering the example above, if we randomly choose a sample (denoted $\delta[zipcode]$) of say 20 values from the domain of zipcodes, then the set of tuples returned by the definition will need to be scaled by a factor of $|\mathcal{D}[zipcode]|/20$.

A more general equation for computing the scaling factor (denoted SF) is shown below. Note that the sampling may need to be performed over a set of attributes. (Here $\beta_v\backslash\beta_s$ denotes the input attributes of v which are outputs of s.)

$$SF = \frac{|\mathcal{D}[\beta_v\backslash\beta_s]|}{|\delta[\beta_v\backslash\beta_s]|}$$

We now calculate the effect of this scaling factor on the overall score as follows. We denote the set of tuples returned by the definition given the sampled input as $\tilde{O}_v(i)$. This value when scaled will approximate the set of tuples that would have been returned had the definition been invoked with all the possible values for the additional input attributes:

$$|O_v(i)| \approx |\tilde{O}_v(i)| * SF$$

Assuming the sampling is performed randomly over the domain of possible values, the intersection between the tuples produced by the source and the definition should scale in the same way. Thus the only factor not affected by the scaling in the score defined previously is $|O_s(i)|$. If we divide throughout by the scaling factor we have a new scoring function:

$$score_4(s, v, I) = \frac{1}{|I|} \sum_{i \in I} \frac{|\pi_{v \backslash \beta_s}(O_s(i)) \cap \tilde{O}_v(i)|}{|O_s(i)|/SF + |\tilde{O}_v(i)||\mathcal{D}[\beta_s^c \backslash v]| - |\pi_{v \backslash \beta_s}(O_s(i)) \cap \tilde{O}_v(i)|}$$

The problem with this approach is that often the sampled set of values is too small and as a result it does not intersect with the set of values returned by the source, even though a larger sample would have intersected in some way. Thus our sampling introduces unfair distortions into the score for certain definitions, causing them to perform poorly. For example, consider again *source5* and assume that for scalability purposes, the service places a limit on the maximum value for the input radius *dist1*. (This makes sense, as otherwise the user could set the input radius to cover the entire US, and a tuple for every possible zipcode would need to be returned.) Now consider the sampling performed above. If we randomly choose only 20 zipcodes from the set of all possible zipcodes, the chance of the sample containing a zipcode which lies within a 300 mile radius of a particular zipcode (in the middle of the desert) is very low. Moreover, even if one pair of zipcodes (out of 20) results in a successful invocation, this will not be sufficient for learning a good definition for the service.

So to get around this problem we bias the sample such that, whenever possible, half of the values are taken from positive examples of the target (those tuples returned by the new source) and half are taken from negative examples (those tuples not returned by the source). By sampling from both positive and negative tuples, we guarantee that the approximation generated will be as accurate as possible given the limited sample size. We denote the set of positive and negative samples as $\delta^+[\beta_v \backslash \beta_s]$ and $\delta^-[\beta_v \backslash \beta_s]$, and use these values to define *positive* and *total* scaling factors as shown below. (The numerator for the positive values is different from before, as these values have been taken from the output of the new source.)

$$SF^+ = \frac{|\pi_{\beta_v \backslash \beta_s}(\pi_{v \backslash \beta_s}(O_s(i)))|}{|\delta^+[\beta_v \backslash \beta_s]|}$$

The total scaling factor is the same value as before, but calculated slightly differently:

$$SF = \frac{|\mathcal{D}[\beta_v \backslash \beta_s]|}{|\delta^+[\beta_v \backslash \beta_s]| + |\delta^-[\beta_v \backslash \beta_s]|}$$

The score can then be approximated accordingly by taking into account these new scaling factors. The intersection now needs to be scaled using the positive scaling factor:

$$|\pi_{v \backslash \beta_s}(O_s(i)) \cap O_v(i)| \approx |\pi_{v \backslash \beta_s}(O_s(i)) \cap \tilde{O}_v(i)| * SF^+$$

This new scaling results in a new function for evaluating the quality of a view definition:

$$score_5(s, v, I) = \frac{1}{|I|} \sum_{i \in I} \frac{|\pi_{v \backslash \beta_s}(O_s(i)) \cap \tilde{O}_v(i)| * SF^+}{|O_s(i)| + |\tilde{O}_v(i)||\mathcal{D}[\beta_s^c \backslash v]| * SF - |\pi_{v \backslash \beta_s}(O_s(i)) \cap \tilde{O}_v(i)| * SF^+}$$

5.3 Favouring Shorter Definitions

Now that we have derived a score for comparing the data that a source and candidate produce, we can define the evaluation function *eval* used in Algorithm 1. As mentioned in section 4.6, shorter definitions for the target source should be preferred over longer and possibly less accurate ones. In accordance with this principle, we scale the score by the length of the definition, so as to favour shorter definitions as follows:

$$eval(v) = \omega^{length(v)} * score_5(s, v, I)$$

Here $length(v)$ is the length of the clause and $\omega < 1$ is a weighting factor. Setting the weighting factor to be a little less than 1 (such as 0.95) helps to remove logically redundant definitions, which can sometimes be hard to detect, but often return almost exactly the same score as their shorter equivalent. We will discuss the problem of generating non-redundant clauses in section 6.3.

5.4 Approximating Equality

Until now, we have ignored the problem of deciding whether two tuples produced by the target source and the definition are the same. Since different sources may serialize data in different ways and at different levels of accuracy, we must allow for some flexibility in the values that the tuples contain. For instance, in the example from section 2, the distance values returned by the source and definition did not match exactly, but were "sufficiently similar" to be accepted as the same value.

For numeric types like *temperature* or *distance* it makes sense to use an error bound (like $\pm 0.5°C$) or a percentage error (such as $\pm 1\%$) to decide if two values can be considered the same. This is because the sensing equipment (in the case of *temperature*) or the algorithm (in the case of *distance*) will have some error bound associated with the values it produces. We require that an error bound for each numeric type be provided in the problem specification. (Ideally, these bounds would be learnt automatically from examples.)

For certain nominal types like *company* names, where values like ⟨*IBM Corporation*⟩ and ⟨*International Business Machines Corp.*⟩ represent the same value, simplistic equality checking using exact or substring matches is not sufficient for deciding whether two values correspond to the same entity. In this case, string edit distances such as the JaroWinkler score do better at distinguishing strings representing the same entity from those representing different ones (Bilenko, Mooney, Cohen, Ravikumar, & Fienberg, 2003). A machine learning classifier could be trained on a set of such examples to learn which of the available string edit distances best distinguishes values of that type and what threshold to set for accepting a pair as a match. We require that this pair of similarity metric and threshold (or any combinations of metrics) be provided in the problem specification.

In other cases, enumerated types like *months* of the year might be associated with a simple equality checking procedure, so that values like ⟨*January*⟩, ⟨*Jan*⟩ and ⟨1⟩ can be found equal. 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 source description. Indeed, the equality rules could also be generated offline by training a classifier.

Complex types such as *date* present a bigger problem when one considers the range of possible serializations, including values like ⟨5/4/2006⟩ or ⟨Thu, 4 May 2006⟩ or ⟨2006-05-04⟩. In such cases specialized functions are not only required to check equality between values but also to break the complex types up into their constituent parts (in this case *day*, *month* and *year*). The latter would form part of the domain model.

In some cases, deciding whether two values of the same type can be considered equal depends not only on the type, but also on the relations they are used in. Consider the two relations shown below. The first provides the latitude and longitude coordinates of the centroid for a zipcode, while the second returns the coordinates for a particular address:

`centroid(zipcode, latitude, longitude)`

`geocode(number, street, zipcode, latitude, longitude)`

Given the different ways of calculating the centroid of a zipcode (including using the center of mass or the center of population density) an error bound of 500 meters might make sense for equating latitude and longitude coordinates. For a geocoding service, on the other hand, an error bound of 50 meters may be more reasonable. In general, such error bounds should be associated with the set of global relations, instead of just the semantic types, and could be learnt accordingly. When the relations contain multiple attributes, then the problem of deciding whether two tuples refer to the same entity is called *record linkage* (Winkler, 1999). An entire field of research is devoted to tackling this problem. Due to the complexity of the problem and the variety of techniques that have been developed to handle it, we do not investigate it further here.

6. Extensions

In this section we discuss extensions to the basic algorithm needed for handling *real* data sources, as well as ways to reduce the size of the hypothesis space and improve the quality of the definitions produced.

6.1 Generating Inputs

The first step in the source induction algorithm is to generate a set of tuples which will represent the target relation during the induction process. In other words, the system must try to invoke the new source to gather some example data. Doing this without biasing the induction process is easier said than done. The simplest approach to generating input values is to select constants at random from the set of examples given in the problem specification. The problem with this approach is that in some cases the new source will not produce any output for the selected inputs. Instead the system may need to select values according to some distribution over the domain of values in order for the source to invoke correctly. For example, consider a source providing posts of used cars for sale in a certain area. The source takes the make of the car as input, and returns car details:

`usedCars($make, model, year, price, phone)`

Although there are over a hundred different car manufacturers in the world, only a few of them produce the bulk of the cars. Thus invoking the source with values like *Ferrari*, *Lotus* and *Aston Martin* will be less likely to return any tuples, when compared with more common brands such as *Ford* and *Toyota* (unless the source is only providing data for sports cars of course). If a distribution over possible values is available, the system can first try the more common values, or more generally, it can choose values from that set according to the distribution. In this particular example, it might not be too difficult to query the source with a complete set of car manufacturers until one of the invocations returns some data. In general, the set of examples may be very large (such as the 40,000+ zipcodes in the US) and the number of "interesting" values in that set (the ones likely to return results) may be very small, in which case taking advantage of prior knowledge about the distribution of possible values makes sense. It should be noted also that during execution the system will receive a lot of output data from the different sources it accesses. This data can be recorded to generate distributions over possible values for the different types.

The problem of generating viable input data for a new source becomes yet more difficult if the input required is not a single value but a tuple of values. In this case the system can first try to invoke the source with random combinations of attribute values from the examples of each type. Invoking some sources (such as *source5*) is easy because there is no explicit restriction on the combination of input values:

`source5`(`$zip`, `$distance`, `zip`, `distance`)

In other cases, such as a geocoding service the combination of possible input values is highly restricted:

`USGeocoder`(`$number`, `$street`, `$zipcode`, `latitude`, `longitude`)

Randomly selecting input values independently of one another is unlikely to result in any successful invocations. (In order for the invocation to succeed, the randomly generated tuple must correspond to an address that actually exists.) In such cases, after failing to invoke the source a number of times, the system can try to invoke other sources (such as the hotel lookup service below), which produce tuples containing the required attribute types:

`HotelSearch`(`$city`, `hotel`, `number`, `street`, `zipcode`)

In general, this process of invoking sources to generate input for other sources can be chained until a set of viable inputs is generated.

We note that the problem of synthesizing viable input data is itself a difficult and interesting research problem. Our combined approach of utilizing value distributions and invoking alternative services performs well in our experiments (see section 7.3), but an area of future work is to develop a more general solution.

6.2 Dealing with Sources

In order to minimise source accesses, which can be very expensive in terms of both time and bandwidth, all requests to the individual sources are cached in a local relational database. This implementation means that there is an implicit assumption in this work that the output produced by the services is constant for the duration of the induction process. This could be problematic if the service being modeled provides (near) real-time data with an update frequency of less than the time it takes to induce a definition. For a weather prediction service, updated hourly, this may not present much of a problem, since the difference between predicted temperatures may vary only slightly from one update to the next. For a real-time flight status service providing the coordinates of a given aircraft every five minutes, the caching may be problematic as the location of the plane will vary greatly if it takes, for example, one hour to induce a definition. In theory one could test for such variation systematically by periodically invoking the same source with a previously successful input tuple to see if the output has changed, and update the caching policy accordingly.

6.3 Logical Optimisations

Evaluating definitions can be expensive both in terms of time (waiting for sources to return data) and computation (calculating joins over large tables). Thus it makes sense to check each candidate for redundancy before evaluating it. To decide which definitions are redundant, we use the concept of *query containment*:

> A query $q_1 \in \mathcal{L}_{R,A}$ is said to be *contained* in another query $q_2 \in \mathcal{L}_{R,A}$ if for any database instance $\mathcal{I}$, the set of tuples returned by the first query is a subset of those returned by the second, i.e. $\forall_{\mathcal{I}}\ \mathcal{E}_{\mathcal{I}}[q_1] \subseteq \mathcal{E}_{\mathcal{I}}[q_2]$. We denote containment by $q_1 \sqsubseteq q_2$. Two queries are considered *logically equivalent* if $q_1 \sqsubseteq q_2 \wedge q_2 \sqsubseteq q_1$.

For the conjunctive queries learnt in this paper, testing for query containment reduces to the problem of finding a *containment mapping* (Chandra & Merlin, 1977).[11] We can use this test to discover logically equivalent definitions such as the following, (which contain a reordering of the same literals):

```
source($zip, temp):- getCentroid($zip, lat, lon), getConditions($lat, $lon, temp).
source($zip, temp):- getConditions($lat, $lon, temp), getCentroid($zip, lat, lon).
```

Such equivalence checking can be performed efficiently if a canonical ordering of predicate and variable names is chosen a priori. Whenever logically equivalent definitions are discovered, the search can backtrack, thereby avoiding entire sub-trees of equivalent clauses. Similarly, we can test for and skip logically redundant clauses such as the following (which is equivalent to a shorter definition without the second literal):

```
source($zip, _):- getCentroid($zip, lat, long), getCentroid($zip, lat, _).
```

Again, such redundancy checking can be performed efficiently (Levy, Mendelzon, Sagiv, & Srivastava, 1995) resulting in little computational overhead during search.

6.4 Functional Sources

More information may be known about the functionality of certain sources than is expressed by their source definitions. For example, sources like *Multiply* and *Concatenate*, which are implemented locally, will be known to be complete. (A source is considered *complete*, if it returns *all* of the tuples implied by its definition, i.e. $\mathcal{E}[s] = \mathcal{E}_{\mathcal{I}}[v]$.) Whenever such information is available, the induction system can take advantage of it to improve search efficiency. To explain how, we define a class of sources that we call *functional sources*, which are complete and for any input tuple return exactly one output tuple. This is slightly more restrictive than the standard ILP concept of *determinate literals* (Cameron-Jones & Quinlan, 1994), which for every input tuple return *at most* one output tuple. *Multiply* and *Concatenate* are both examples of functional sources. The system takes advantage of the fact that functional sources place no restrictions on their input. Whenever a functional source is added to a candidate definition, the score for that definition doesn't change providing all the source's inputs and none of its outputs are bound. (The set of tuples returned by the new

11. If the queries contain interpreted predicates, then containment testing is a little more involved (Afrati, Li, & Mitra, 2004).

definition is the same as before, but with a few new attributes corresponding to the outputs of the source.) Thus the new definition does not need to be evaluated, but can be added to the queue (of definitions to expand) as is, which becomes particularly advantageous when a source's input arity is high.

6.5 Constants

Constants are often used in source descriptions to define the scope of a service. Consider a weather service that provides reports for zipcodes only in California:

calWeather(\$zip, \$date, temp) :- forecast(zip, date, temp), USstate(zip, *California*).

If a mediator receives a query asking for the forecast for Chicago, it will know that this source is not relevant to the query since Chicago is not in California. Although constants in source descriptions can be very useful, simply introducing them into the hypothesis language could cause the search space to grow prohibitively. (For states, the branching factor would be 50, while for zipcodes it would be in excess of 40,000.) Obviously a generate and test methodology does not make sense when the domain of the semantic type is large. Alternatively, one can explicitly check for repeated values in the tuples returned by the new source (i.e. for constants in the head of the clause) and in the join of the source and definition relations (i.e. for constants in the body of the clause). For example, in the definition below the join of the source relation ⟨zip, date, temp⟩ with the definition relation ⟨zip, date, temp, state⟩ would produce only tuples with state equal to *California*. So that constant could be added to the definition.

source(\$zip, \$date, temp) :- forecast(zip, date, temp), USstate(zip, state).

More complicated detection procedures would be required for discovering constants in interpreted predicates (i.e. range restrictions over numeric attributes).

6.6 Post-Processing

After a definition has been learnt for a new source, it may be possible to tighten that definition by removing logical redundancies from its unfolding. Consider the following definition involving calls to two hotel sources, one to check availability and the other to check its rating:

```
source($hotel, address, rating):-
 HotelAvailability($hotel, address, price), HotelRating($hotel, rating, address).
```

The unfolding of that definition (in terms of the definitions of the hotel sources) contains two references to an *accommodation* relation:

```
source($hotel, address, rating):-
  accommodation(hotel, _, address), available(hotel, today, price),
  accommodation(hotel, rating, address).
```

The first literal is redundant and can be removed from the unfolding. In general, the same rules used to discover redundancy in candidate definitions can be used to remove redundant literals from the unfolding. Moreover, since this post-processing step needs to be performed only once, time can be spent searching for more complicated forms of redundancy.

7. Evaluation

In this section we describe our evaluation of the source induction algorithm. We first describe the experimental setup used and then the experiments performed. Finally, we compare the induction algorithm with a particular complex schema matching system.

7.1 Experimental Setup

The source induction algorithm defined in this paper was implemented in a system called EIDOS, which stands for *Efficiently Inducing Definitions for Online Sources*. EIDOS implements the techniques and optimisations discussed in sections 4 through 6. (Certain extensions from section 6 were only partially implemented: the implementation currently checks for constants only in the head of the clause and does not perform any tightening of the definitions.) All code was written in Java and a MySQL database was used for caching the results of source invocations.

EIDOS was tested on 25 different problems involving real services from several domains including hotels, financial data, weather and cars. The domain model used was *the same* for each problem and included over 70 different semantic types, ranging from common ones like *zipcode* to more specific types such as stock *ticker* symbols. The data model also contained 36 relations (excluding interpreted predicates), which were used to model 33 different services. All of the modeled services are publicly available information sources. We note here that the decision to use the same set of known sources for each problem (regardless of the domain) was important in order to make sure that the tests were realistic. This decision made the problem more difficult than the standard schema matching/mapping scenario in which the source schema is chosen, because it provides data that is known *a priori* to be relevant to the output schema.

In order to give a better sense of the problem setting and the complexity of the known sources available, we list ten below (ordered by arity). Due to space limitations we don't show the complete list nor their definitions, only the input/output types for each source. Note that all the sources share the semantic types *latitude* and *longitude*, which means that the search space associated with these sources alone is very large.

```
1  WeatherConditions($city,state,country,latitude,longitude,time,time,timeoffset,
    datetime,temperatureF,sky,pressureIn,direction,speedMph,humidity,temperatureF)
2  WeatherForecast($city,state,country,latitude,longitude,timeoffset,day,date,
    temperatureF,temperatureF,time,time,sky,direction,speedMph,humidity)
3  GetDistance($latitude,$longitude,$latitude,$longitude,distanceKm)
4  USGeocoder($street,$zipcode,city,state,latitude,longitude)
5  ConvertDMS($latitudeDMS,$longitudeDMS,latitude,longitude)
6  USGSEarthquakes(decimal,timestamp,latitude,longitude)
7  GetAirportCoords($iata,airport,latitude,longitude)
8  CountryCode($latitude,$longitude,countryAbbr)
9  GetCentroid($zipcode,latitude,longitude)
10 Altitude($latitude,$longitude,distanceM)
```

In order to induce definitions for each problem, the source (and each candidate definition) was invoked at least 20 times using random inputs. Whenever possible, the system attempted to generate 10 positive examples of the source (invocations for which the source returned some tuples) and 10 negative examples (inputs which produced no output). To ensure that the search terminated, the number of iterations of the algorithm including backtracking steps was limited to 30. A search time limit of 20 minutes was also imposed. The inductive search bias used during the experiments is shown below, and a weighting factor (defined in section 5.3) of 0.9 was used to direct the search toward shorter definitions.

Search Bias
Maximum clause length = 7
Maximum predicate repetition = 2
Maximum variable level = 5
Executable candidates only
No variable repetition within a literal

In the experiments, different procedures were used to decide equality between values of the same type as discussed in section 5.4. Some of the equality procedures used for different types are listed below. The accuracy bounds and thresholds used were chosen to maximize overall performance of the learning algorithm. (In practice, a meta-learning algorithm could be used to determine the best accuracy bounds for different attribute types.) For all semantic types not listed below, substring matching (checking if one string contained the other) was used to test equality between values.

Types	Equality Procedure
latitudes, longitudes	accuracy bound of ± 0.002
distances, speeds, temperatures, prices	accuracy bound of $\pm 1\%$
humidity, pressure, degrees	accuracy bound of ± 1.0
decimals	accuracy bound of ± 0.1
companies, hotels, airports	JaroWinkler score ≥ 0.85
dates	specialised equality procedure

The experiments were run on a dual core 3.2 GHz Pentium 4 with 4 GB of RAM (although memory was not a limiting factor in any of the tests). The system was running Windows 2003 Server, Java Runtime Environment 1.5 and MySQL 5.0.

7.2 Evaluation Criteria

In order to evaluate the induction system, one would like to compare for each problem the definition generated by the system with the *ideal definition* for that source (denoted v_{best} and v^* respectively). In other words, we would like an evaluation function, which rates the quality of each definition produced with respect to a hand-written definition for the source (i.e. $quality : v_{best} \times v^* \rightarrow [0, 1]$). The problem with this is twofold. Firstly, it is not obvious how to define such a similarity function over conjunctive queries and many different possibilities exist (see Markov and Marinchev, 2000, for a particular example). Secondly, working out the best definition by hand, while taking into account the limitations of the domain model and the fact that the available sources are noisy, incomplete, possibly less accurate, and even serialise data in different ways, may be extremely difficult, if even possible. So in order to evaluate each of the discovered definitions, we instead count the number of *correctly generated* attributes in each definition. An attribute is said to be *correctly generated*, if:

- it is an input, and the definition correctly restricts the domain of possible values for that attribute, or
- it is an output, and the definition correctly predicts its value for given input tuples.

Consider the following definition that takes two input values and returns the difference and its square root (providing the difference is positive):

$\texttt{source(\$A, \$B, C, D) :- sum(B, C, A), product(D, D, C), A} \geq \texttt{B.}$

and imagine that the induction system managed to learn only that the source returns the difference between the input and the output, i.e.:

`source($A, $B, C, _) :- sum(B, C, A).`

We say that the first input attribute A is not *correctly generated* as it is an input and is not constrained with respect to the input attribute B (the inequality is missing). The input B is deemed correctly generated as it is present in the `sum` relation (only one input is penalised for the missing inequality). The output C is deemed *correctly generated* with respect to the inputs, and the missing attribute D isn't generated at all. (Note that if the ordering of variables in the `sum` relation had been different, say `sum(A, B, C)`, then C would have been generated, but not *correctly generated*.)

Given a definition for correctly generated attributes, one can define expressions for *precision* and *recall* over the attributes contained in a source definition. We define *precision* to be the ratio of correctly generated attributes to the total number of attributes generated by a definition, i.e.:

$$precision = \frac{\text{\# of correctly generated attributes}}{\text{total \# of generated attributes}}$$

We define *recall* to be the ratio of generated attributes to the total number of attributes that *would have been generated* by the ideal definition, given the sources available. (In some cases no sources are available to generate values for an attribute in which case, that attribute is not included in the count.)

$$recall = \frac{\text{\# of correctly generated attributes}}{\text{total \# of attributes that should have been generated}}$$

Note that we defined *precision* and *recall* at the schema level in terms of the attributes involved in a source definition. They could also be defined at the data level in terms of the tuples being returned by the source and the definition. Indeed, the Jaccard similarity used to score candidate definitions is a combination of data-level precision and recall values. The reason for choosing schema level metrics in our evaluation is that they better reflect the semantic correctness of the learnt definition, in so far as they are independent of the completeness (amount of overlap) between the known and target sources.

Returning to our example above, the precision for the simple definition learnt would be 2/3 and the recall would be 2/4. Note that, if the `product` relation had not been available in our domain model (in which case attribute D could *never* have been generated), recall would have been higher at 2/3.

7.3 Experiments

The definitions learnt by the system are described below. Overall the system performed very well and was able to learn the intended definition (ignoring missing join variables and superfluous literals) in 19 out of the 25 problems.

7.3.1 Geospatial Sources

The first set of problems involved nine geospatial data sources providing a variety of location based information. The definitions learnt by the system are listed below. They are reported in terms of the source predicates rather than the domain relations (i.e. the unfoldings) because the corresponding definitions are much shorter. This makes it easier to understand how well the search algorithm is performing.

```
1  GetInfoByZip($zip0,cit1,sta2,_,tim4) :-
    GetTimezone($sta2,tim4,_,_), GetCityState($zip0,cit1,sta2).
2  GetInfoByState($sta0,cit1,zip2,_,tim4) :-
    GetTimezone($sta0,tim4,_,_), GetCityState($zip2,cit1,sta0).
3  GetDistanceBetweenZipCodes($zip0,$zip1,dis2) :-
    GetCentroid($zip0,lat1,lon2), GetCentroid($zip1,lat4,lon5),
    GetDistance($lat1,$lon2,$lat4,$lon5,dis10), ConvertKm2Mi($dis10,dis2).
4  GetZipCodesWithin($_,$dis1,_,dis3) :-
    <(dis3,dis1).
5  YahooGeocoder($str0,$zip1,cit2,sta3,_,lat5,lon6) :-
    USGeocoder($str0,$zip1,cit2,sta3,lat5,lon6).
6  GetCenter($zip0,lat1,lon2,cit3,sta4) :-
    WeatherConditions($cit3,sta4,_,lat1,lon2,_,_,_,_,_,_,_,_,_,_,_),
    GetZipcode($cit3,$sta4,zip0).
7  Earthquakes($_,$_,$_,$_,lat4,lon5,_,dec7,_) :-
    USGSEarthquakes(dec7,_,lat4,lon5).
8  USGSElevation($lat0,$lon1,dis2) :-
    ConvertFt2M($dis2,dis1), Altitude($lat0,$lon1,dis1).
9  CountryInfo($cou0,cou1,cit2,_,_,cur5,_,_,_,_) :-
    GetCountryName($cou0,cou1), GoCurrency(cur5,cou0,_),
    WeatherConditions($cit2,_,cou1,_,_,_,_,_,_,_,_,_,_,_,_,_).
```

The first two sources provide information about zipcodes, such as the name of the city, the state and the timezone. They differ in their binding constraints, with the first taking a zipcode as input, and the second taking a state. The second source returns many output tuples per input value, making it harder to learn the definition, even though the two sources provide logically the same information. The induced definitions are the best possible given the known sources available. (None of them provided the missing output attribute, a telephone area-code.) The third source calculates the distance in miles between two zipcodes, (it is the same as *source4* from section 2). The correct definition was learnt for this source, but for the next source, which returned zipcodes within a given radius, a reasonable definition could not be learnt within the time limit.[12] Ignoring binding constraints, the intended

12. The recall for this problem is 1/4 because the input attribute *dis1* is determined to be the only *correctly generated* attribute (it is constrained with respect to the output attribute *dis3*), while all four attributes should have been generated (the output attribute *dis3* is not generated by the $<$ predicate). The precision is 1/1 because *dis1* is the only generated attribute, and it is correctly generated.

definition was the same as the third, but with a restriction that the output distance be less than the input distance. Thus it would have been far easier for eidos to learn a definition for the fourth source in terms of the third. Indeed, when the new definition for the third source was added to the set of known sources, the system was able to learn the following:

```
4' GetZipCodesWithin($zip0,$dis1,zip2,dis3) :-
     GetDistanceBetweenZipCodes($zip0,$zip2,dis3), <(dis3,dis1).
```

The ability of the system to improve its learning ability over time as the set of known sources increases is a key benefit of the approach.

Source five is a geocoding service provided by Yahoo. (Geocoding services map addresses to latitude and longitude coordinates.) eidos learnt that the same functionality was provided by a service called *USGeocoder*. Source six is a simple service providing the latitude/longitude coordinates and the city and state for a given zipcode. Interestingly, the system learnt that the source's coordinates were better predicted by a weather conditions service (discussed in section 7.3.3), than by the *GetCentroid* source from the third definition. Note that when the new source definition is unfolded it will contain extraneous predicates related to weather information.[13] The additional predicates do not interfere with the usefulness of the definition, however, as a query reformulation algorithm will still use the source to answer the same queries regardless. (Thus precision and recall scores are not affected.) A post-processing step to remove extraneous predicates is possible, but would require additional information in the domain model.[14] The seventh source provided earthquake data within a bounding box which it took as input. In this case, the system discovered that the source was indeed providing earthquake data, (*lat4* and *lon5* are the coordinates of the earthquake and *dec7* is its magnitude). It didn't manage, however, to work out how the input coordinates related to the output. The next source provided elevation data in feet, which was found to be sufficiently similar to known altitude data in metres. Finally, the system learnt a definition for a source providing information about countries such as the currency used, and the name of the capital city. Since known sources were not available to provide this information, the system ended up learning that weather reports were available for the capital of each country.

Problem	# Candidates	# Invocations	Time (s)	log_{10}(Score)	Precision	Recall
1	25	5068	85	-1.36	4/4	4/4
2	24	9804	914	-1.08	4/4	4/4
3	888	11136	449	-0.75	3/3	3/3
4	3	11176	25	0.25	1/1	1/4
5	50	13148	324	-0.45	6/6	6/6
6	40	15162	283	-7.61	5/5	5/5
7	11	14877	18	-6.87	3/3	3/9
8	15	177	72	-8.58	3/3	3/3
9	176	28784	559	-5.77	4/4	4/4

13. The unfolding is shown below. The *conditions* predicate could be removed without affecting its meaning:
`GetCenter($zip0, lat1, lon2, cit3, sta4):- municipality(cit3, sta4, zip2, tim3), country(_, cou5, _), northAmerica(cou5), centroid(zip2, lat1, lon2), conditions(lat1, lon2, _, _, _, _, _, _, _, _, _), timezone(tim3, _, _), municipality(cit3, sta4, zip0, _).`

14. In particular, universally quantified knowledge would be needed in the domain model, e.g.:
$\forall \texttt{lat}, \texttt{lon}\ \exists x_1, ..., x_{11}$ s.t. conditions(lat, long, $x_1, ..., x_{11}$)

The table shows some details regarding the search performed to learn each of the definitions listed above. For each problem, it shows the number of candidates generated prior to the winning definition, along with the time and number of source invocations required to learn the definition. (The last two values should be interpreted with caution as they are highly dependent on the delay in accessing sources, and on the caching of data in the system.) The scores shown in the fifth column are a normalised version of the scoring function used to compare the definitions during search. (Normalisation involved removing the penalty applied for missing outputs.) The scores can be very small, so the logarithm of the values is shown (hence the negative values).[15] These scores can be interpreted as the confidence the system has in the definitions produced. The closer the value is to zero, the better the definition's ability to produce the same tuples as the source. We see that the system was far more confident about the definitions one through five, than the latter ones.[16] The last two columns give the precision and recall value for each problem. The average precision for these problems was 100%. (Note that a high precision value is to be expected, given that the induction algorithm relies on finding matching tuples between the source and definition.) The average recall for the geospatial problems was also very high at 84%.

7.3.2 Financial Sources

Two sources were tested that provided financial data. The definitions generated by EIDOS for these sources are shown below.

```
10 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,$pri10,pri1).
11 YahooExchangeRate($_,$cur1,pri2,dat3,_,pri5,pri6) :-
    GetCurrentTime(_,_,dat3,_), GoCurrency(cur1,cou5,pri2),
    GoCurrency(_,cou5,pri5), Add($pri2,$pri5,pri12), Add($pri2,$pri6,pri12).
```

The first financial service provided stock quote information, and the system learnt that the source returned exactly the same information as a stock market service provided by Yahoo. It was also able to work out that the previous day's close plus today's change was equal to the current price. The second source provided the rate of exchange between the currencies given as input. In this case, the system did not fare well. It was unable to learn the intended result, which involved calculating the exchange rate by taking the ratio of the values for the first and second currency.

Problem	# Candidates	# Invocations	Time (s)	log_{10}(Score)	Precision	Recall
10	2844	16671	387	-8.13	12/13	12/12
11	367	16749	282	-9.84	1/5	1/4

Details regarding the search spaces for the two problems are shown above. The average precision and recall for these problems were much lower at 56% and 63% respectively, because the system was unable to learn the intended definition in the second problem.

15. The positive value for problem 4 results from an approximation error.
16. Low scores and perfect precision and recall (problems 6, 8 and 9) indicate very little overlap between the target and the known sources. The fact that the system learns the correct definition in such cases is testimony to the robustness of the approach.

7.3.3 Weather Sources

On the Internet, there are two types of weather information services, those that provide forecasts for coming days, and those that provide details of current weather conditions. In the experiments, a pair of such services provided by *Weather.com* were used to learn definitions for a number of other weather sources. The first set of definitions, which correspond to sources that provide current weather conditions, are listed below:

```
12 NOAAWeather($ica0,air1,_,_,sky4,tem5,hum6,dir7,spe8,_,pre10,tem11,_,_) :-
    GetAirportInfo($ica0,_,air1,cit3,_,_),
    WeatherForecast($cit3,_,_,_,_,_,_,_,_,_,_,_,sky4,dir7,_,_),
    WeatherConditions($cit3,_,_,_,_,_,_,_,_,tem5,sky4,pre33,_,spe8,hum6,tem11),
    ConvertIn2mb($pre33,pre10).
13 WunderGround($sta0,$cit1,tem2,_,_,pre5,pre6,sky7,dir8,spe9,spe10) :-
    WeatherConditions($cit1,sta0,_,_,_,_,_,_,dat8,tem9,sky7,pre5,dir8,spe13,_,tem2),
    WeatherForecast($cit1,sta0,_,_,_,_,_,_,tem24,_,_,_,_,_,spe10,_),
    ConvertIn2mb($pre5,pre6),<(tem9,tem24),ConvertTime($dat8,_,_,_,_),<(spe9,spe13).
14 WeatherBugLive($_,cit1,sta2,zip3,tem4,_,_,dir7,_,_) :-
    WeatherConditions($cit1,sta2,_,_,_,_,_,_,_,tem4,_,_,dir7,_,_,_),
    GetZipcode($cit1,$sta2,zip3).
15 WeatherFeed($cit0,$_,tem2,_,sky4,tem5,_,_,pre8,lat9,_,_) :-
    WeatherConditions($cit0,_,_,lat9,_,_,_,_,_,_,sky4,pre8,dir12,_,_,tem5),
    WeatherForecast($cit0,_,_,_,_,_,_,_,_,tem2,_,_,_,dir12,_,_).
16 WeatherByICAO($ica0,air1,cou2,lat3,lon4,_,dis6,_,sky8,_,_,_,_) :-
    Altitude($lat3,$lon4,dis6), GetAirportInfo($ica0,_,air1,cit6,_,cou8),
    WeatherForecast($cit6,_,cou8,_,_,_,_,_,_,_,_,_,sky8,_,_,_),
    GetCountryName($cou2,cou8).
17 WeatherByLatLon($_,$_,_,_,_,lat5,lon6,_,dis8,_,_,_,_,_,_) :-
    Altitude($lat5,$lon6,dis8).
```

In the first problem, the system learnt that source 12 provided current conditions at airports, by checking the weather report for the cities in which each airport was located. This particular problem demonstrates some of the advantages of learning definitions for new sources described in section 3.3. Once the definition has been learnt, if a mediator receives a request for the current conditions at an airport, it can generate an answer for that query by executing a single call to the newly modeled source, (without needing to find a nearby city). The system performed well on the next three sources (13 to 15) learning definitions which cover most of the attributes of each. On the last two problems, the system did not perform as well. In the case of source 16, the system spent most of its time learning which attributes of the airport were being returned (such as its country, coordinates, elevation, etc.). In the last case, the system was only able to learn that the source was returning some coordinates along with their elevation. We note here that different sources may provide data at different levels of accuracy. Thus the fact that the system is unable to learn a definition for a particular source could simply mean that the data being returned by that source wasn't sufficiently accurate for the system to label it a match.

In addition to current weather feeds, the system was run on two problems involving weather forecast feeds. It did very well on the first problem, matching all bar one of the attributes (the country) and finding that the order of the high and low temperatures was inverted. It did well also for the second problem, learning a definition for the source that produced most of the output attributes.

```
18 YahooWeather($zip0,cit1,sta2,_,lat4,lon5,day6,dat7,tem8,tem9,sky10) :-
    WeatherForecast($cit1,sta2,_,lat4,lon5,_,day6,dat7,tem9,tem8,_,_,sky10,_,_,_),
    GetCityState($zip0,cit1,sta2).
19 WeatherBugForecast($_,cit1,sta2,_,day4,sky5,tem6,_) :-
    WeatherForecast($cit1,sta2,_,_,_,tim5,day4,_,tem6,_,tim10,_,sky5,_,_,_),
    WeatherConditions($cit1,_,_,_,_,tim10,_,tim5,_,_,_,_,_,_,_,_).
```

Details regarding the number of candidates generated in order to learn definitions for the different weather sources are shown below. The average precision of the definitions produced was 91%, while the average recall was 62%.

Problem	# Candidates	# Invocations	Time (s)	log_{10}(Score)	Precision	Recall
12	277	579	233	-2.92	8/9	8/11
13	1989	426	605	-6.35	6/9	6/10
14	98	2499	930	-13.37	5/5	5/8
15	199	754	292	-6.48	5/6	5/10
16	102	946	484	-29.69	6/7	6/9
17	45	7669	1026	-26.71	3/3	3/13
18	119	13876	759	-5.74	10/10	10/11
19	116	14857	1217	-12.56	5/5	5/7

7.3.4 Hotel Sources

Definitions were also learnt for sources providing hotel information from Yahoo, Google and the US Fire Administration. These definitions are shown below.

```
20 USFireHotelsByCity($cit0,_,_,sta3,zip4,cou5,_) :-
    HotelsByZip($zip4,_,_,cit0,sta3,cou5).
21 USFireHotelsByZip($zip0,_,_,cit3,sta4,cou5,_) :-
    HotelsByZip($zip0,_,_,cit3,sta4,cou5).
22 YahooHotel($zip0,$_,hot2,str3,cit4,sta5,_,_,_,_,_) :-
    HotelsByZip($zip0,hot2,str3,cit4,sta5,_).
23 GoogleBaseHotels($zip0,_,cit2,sta3,_,_,lat6,lon7,_) :-
    WeatherConditions($cit2,sta3,_,lat6,lon7,_,_,_,_,_,_,_,_,_,_,_),
    GetZipcode($cit2,$sta3,zip0).
```

The system performed well on three out of the four problems. It was unable in the time allocated to discover a definition for the hotel attributes (name, street, latitude and longitude) returned by the Google Web Service. The average precision for these problems was 90% while the average recall was 60%.

Problem	# Candidates	# Invocations	Time (s)	log_{10}(Score)	Precision	Recall
20	16	448	48	-4.00	4/4	4/6
21	16	1894	5	-2.56	4/4	4/6
22	43	3137	282	-2.81	5/5	5/9
23	95	4931	1161	-7.50	3/5	3/6

7.3.5 Cars and Traffic Sources

The last problems on which the system was tested were a pair of traffic related Web Services. The first service, provided by Yahoo, reported live traffic data (such as accidents and construction work) within a given radius of the input zipcode. No known sources were available which provided such information, so not surprisingly, the system was unable to learn a definition for the traffic related attributes of that source. (Instead, the system discovered a relationship between the input *zipcode* and the output *longitude* that wasn't correct, so precision for this problem was zero.)

```
24 YahooTraffic($zip0,$_,_,lat3,lon4,_,_,_) :-
    GetCentroid($zip0,_,lon4), CountryCode($lat3,$lon4,_).
25 YahooAutos($zip0,$mak1,dat2,yea3,mod4,_,_,pri7,_) :-
    GoogleBaseCars($zip0,$mak1,_,mod4,pri7,_,_,yea3),
    ConvertTime($dat2,_,dat10,_,_), GetCurrentTime(_,_,dat10,_).
```

The second problem involved a classified used-car listing from Yahoo that took a zipcode and car manufacturer as input. EIDOS was able to learn a good definition for that source, taking advantage of the fact that some of the same cars (defined by their make, model, year and price) were also listed for sale on Google's classified car listing.

Problem	# Candidates	# Invocations	Time (s)	log_{10}(Score)	Precision	Recall
24	81	29974	1065	-11.21	0/3	0/4
25	55	405	815	-5.29	6/6	6/6

Since the system failed on the first problem (it found some incorrect/non-general relationships between different attributes), but succeeded on the second problem to find the best possible definition, the average precision and recall for these problems were both 50%.

7.3.6 Overall Results

Across the 25 problems, EIDOS managed to generate definitions with high accuracy (average precision was 88%) and a large number of attributes (average recall was 69%). These results are promising, especially considering that all problems involved real data sources with in some cases very small overlap between the data produced by the target and that provided by the known sources (as evidenced by low logarithmic scores). In addition to minimal overlap, many sources provided incomplete tuples (i.e. tuples containing multiple "NULL" or "N/A" values) as well as erroneous or inaccurate data, making the problem all the more difficult. The high average precision and recall lead us to believe that the Jaccard measure is doing a good job of distinguishing correct from incorrect definitions in the presence of data sources that are both noisy (inconsistent) and incomplete (missing tuples and values).

Comparing the different domains, one can see that the system performed better on problems with fewer input and output attributes (such as the geospatial problems), which was to be expected given that the resulting search space is much smaller.

7.4 Empirical Comparison

Having demonstrated the effectiveness of eidos in learning definitions for real information services, we now show that the system is capable of handling the same problems as a well-known complex schema matching system.

The iMAP system (Dhamanka, Lee, Doan, Halevy, & Domingos, 2004) (discussed in section 8.4) is a schema matcher that can learn complex (many-to-one) mappings between the concepts of a source and a target schema. It uses a set of *special purpose searchers* to learn different types of mappings. The eidos system, on the other hand, uses a generic search algorithm to solve a comparable problem. Since the two systems can be made to perform a similar task, we show that eidos is capable of running on one of the problem domains used in the evaluation of iMAP. We chose the particular domain of online cricket databases because it is the only one used in the evaluation that involved aligning data from two independent data sources. (All other problems involved generating synthetic data by splitting a single database into a source and target schema, which would not have been as interesting for eidos.)

Player statistics from two online cricket databases (cricketbase.com and cricinfo.com) were used in the experiments. Since neither of the sources provided programmatic access to their data, the statistics data was extracted from HTML pages and inserted into a relational database. The extraction process involved flattening the data into a relational model and a small amount of data cleaning. (The resulting tables are similar but not necessarily exactly the same as those used in the iMAP experiments.) The data from the two websites was used to create three data sources representing each website. The three sources representing cricinfo.com were then used to learn definitions for the three sources representing cricketbase.com. Other known sources were available to the system, including functionality for splitting apart comma-separated lists, adding and multiplying numbers, and so on. The definitions learnt to describe the cricketbase services are shown below:

```
1 CricbasePlayers($cou0,nam1,_,dat3,_,unk5,unk6) :-
   CricinfoPlayer($nam1,dat3,_,_,_,lis5,nam6,unk5,unk6), contains($lis5,cou0),
   CricinfoTest($nam6,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_).
2 CricbaseTest($_,nam1,cou2,_,_,_,cou6,cou7,dec8,cou9,_,_,_,cou13,cou14,dec15,_,
  dec17,cou18,_,_) :-
   CricinfoTest($nam1,_,_,cou2,cou6,cou18,cou7,_,dec8,dec15,_,cou9,_,_,_,cou14,
   cou13,_,_,_,_,dec17).
3 CricbaseODI($_,nam1,cou2,_,_,_,cou6,cou7,dec8,cou9,cou10,_,cou12,cou13,_,dec15,
  dec16,dec17,cou18,cou19,_) :-
   CricinfoODI($nam1,_,_,cou2,cou6,_,cou10,_,dec8,_,cou7,cou18,cou19,cou9,_,_,
   cou13,cou12,dec15,_,dec16,dec17).
```

The first source provided player profiles by country. The second and third sources provided detailed player statistics for two different types of cricket (Test and One-Day-International respectively). The system easily found the best definition for the first source. The definition involved looking for the player's country in a list of teams that he played for. eidos did not perform quite as well on the second and third problems. There were two reasons for this. Firstly, the arity of these sources was much higher with many instances of the same semantic type (*count* and *decimal*), making the space of possible alignments much larger.

(Because of the large search space, a longer timeout of 40 minutes was used.) Secondly, a high frequency of null values (the constant "N/A") in the data for some of the fields confused the algorithm, and made it harder for it to discover overlapping tuples with all of the desired attributes.

Problem	# Candidates	# Invocations	Time (s)	log_{10}(Score)	Precision	Recall
1	199	3762	432	-3.95	5/5	5/5
2	1162	1517	1319	-4.70	8/11	8/16
3	3114	4299	2127	-6.28	8/14	8/16

Details of the search performed to learn the definitions are shown above. The average precision for these problems was 77% while the average recall was lower at 66%. These values are comparable to the quality of the matchings reported for iMAP.[17] These results are very good, considering that EIDOS searches in the space of many-to-many correspondences, (trying to define the set of target attributes contemporaneously), while iMAP searches the spaces of one-to-one and many-to-one correspondences. Moreover, EIDOS first invokes the target source to generate representative data (a task not performed by iMAP) and then performs a *generic* search for reasonable definitions without relying on specialised search algorithms for different types of attributes (as is done in iMAP).

8. Related Work

In this section we describe how the work in this paper relates to research performed by the Machine Learning, Database and the Semantic Web communities. Before doing that, we describe some early work performed by the Artificial Intelligence community. We also discuss how our algorithm differs from standard ILP techniques, and in particular why a direct application of such techniques was not possible for our problem.

8.1 An Early Approach

The first work concerned with learning models for describing operations available on the Internet was performed (in the pre-XML era) on a problem called *category translation* (Perkowitz & Etzioni, 1995; Perkowitz, Doorenbos, Etzioni, & Weld, 1997). This problem consisted of an incomplete internal world model and an external information source with the goal being to characterize the information source in terms of the world model. The world model consisted of a set of objects O, where each object $o \in O$ belonged to a certain *category* (e.g. people) and was associated with a set of attributes $\langle a_1(o), ..., a_n(o)\rangle$, made up of strings and other objects. A simple relational interpretation of this world model would consider each category to be a relation, and each object to be a tuple. The information source, meanwhile, was an operation that took in a single value as input and returned a single tuple as output. The *category translation problem* can be viewed as a simplification of the *source definition induction problem*, whereby:

17. The actual values for precision and recall in the cricket domain are not quoted, but an accuracy range of 68-92% for simple matches (one-to-one correspondences between source and target fields) and 50-86% for complex matches (many-to-one correspondences) across synthetic and real problems was given.

- The extensions of the global relations are explicit. (There is one source per global relation, and it doesn't have binding constraints, i.e. $R = S$.)
- The information provided by the sources does not change over time.
- The new source takes a single value as input and returns a single tuple as output.

In order to find solutions to instances of the category translation problem, the authors employed a variant of relational path-finding (Richards & Mooney, 1992), which is an extension on the FOIL algorithm, to learn models of the external source. The technique described in this paper for solving instances of the source induction problem is similar in that it too is based on a FOIL-like inductive search algorithm.

8.2 Direct Application of ILP techniques

Researchers became interested in the field of Inductive Logic Programming in the early nineties, resulting in a number of different ILP systems being developed including FOIL (Cameron-Jones & Quinlan, 1994), PROGOL (Muggleton, 1995) and ALEPH[18]. Ideally, one would like to apply such "off-the-shelf" ILP systems to the source definition induction problem. A number of issues, however, limit the direct applicability of these systems. The issues can be summarised as follows:

- Extensions of the global relations are virtual.
- Sources may be incomplete with respect to their definitions.
- Explicit negative examples of the target are not available.
- Sources may serialise constants in different ways.

The first issue has to do with the fact that all ILP systems assume that there is an extensional definition of the target predicate and extensional (or in some cases intentional) definitions of the (source) predicates that will be used in the definition for the target. In other words, they assume that tables already exist in some relational database to represent both the new source and the known sources. In our case, we need to generate such tables by first invoking the services with relevant inputs. One could envisage invoking each of the sources with every possible input and using the resulting tables to perform induction. Such a direct approach would not be feasible for two reasons. Firstly, a complete set of possible input values may not be known to the system. Secondly, even if it is possible to generate a complete set of viable inputs to a service, it may not be practical to query the source with such a large set of tuples. Consider *source4* from section 2, which calculates the distance in miles between two zipcodes. Given that there are over 40,000 zipcodes in the US, generating an extensional representation of this source would require performing more than a billion invocations! Performing such a large number of invocations does not make sense when a small number of example invocations would suffice for characterising the functionality of the source. In this paper we have developed an efficient algorithm that only queries the sources as needed in order to evaluate individual candidate definitions.

The second issue regarding the incompleteness of the sources causes a problem when a candidate is to be evaluated. Since the set of tuples returned by each known source may only

18. See the Aleph Manual by Ashwin Srinivasan, which is available at: http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html

be a subset of those implied by its own definition, so too will be the set of tuples returned by the candidate hypothesis when executed against those sources. This means that when the system tries to evaluate a hypothesis by comparing those tuples with the output of the new source, it cannot be sure that a tuple which is produced by the new source but not by the hypothesis is in fact not logically implied by it. This fact is taken into account in our evaluation function for scoring candidate definitions, discussed in section 5.

The third issue regarding the lack of explicit negative examples for the target predicate also affects the evaluation of candidate hypotheses. The classic approach to dealing with this problem is to assume a closed world, in which all tuples (over the head relation) which are not explicitly declared to be positive must be negative. Since the new source may in fact be incomplete with respect to the best possible definition for it, this assumption does not necessarily hold. In other words, just because a particular tuple is produced when the candidate definition is executed and that same tuple is not returned by the new source does not necessarily mean that the candidate definition is incorrect.

The fourth issue has to do with the fact that the data provided by different sources may need to be reconciled, in the sense that different serialisations of (strings representing) the same value (such as "Monday" and "Mon" for instance) must be recognized. Since ILP systems have been designed to operate over a single database containing multiple tables, the issue of heterogeneity in the data is not handled by current systems. In section 5.4 we discussed how this heterogeneity is resolved in our system.

8.3 Machine Learning Approaches

Since the advent of services on the Internet, researchers have been investigating ways to model them automatically. Primarily, interest has centered on using machine learning techniques to classify the input and output types of a service, so as to facilitate service discovery. Heß & Kushmerick proposed using a Support Vector Machine to classify the input and output attributes into different semantic types based on metadata in interface descriptions (Heß & Kushmerick, 2003, 2004). Their notion of semantic types (such as *zipcode*) as opposed to syntactic types (like *integer*) went some way toward defining the functionality that a source provides. Recently, other researchers (Lerman et al., 2006) proposed the use of logistic regression for assigning semantic types to input parameters based on metadata, and a pattern language for assigning semantic types to the output parameters based on the data the source produces. This work on classifying input and output attributes of a service to semantic types forms a prerequisite for the work in this article. For the purposes of this paper, we have assumed that this problem has been solved.

In addition to classifying the input/output attributes of services, Heß & Kushmerick investigated the idea of classifying the services themselves into different service types. More precisely, they used the same classification techniques to assign service interfaces to different semantic domains (such as *weather* and *flights*) and the operations that each interface provides to different classes of operation (such as *weatherForecast* and *flightStatus*). The resulting source description (hypothesis) language is limited to select-project queries, which are not sufficiently expressive to describe many of the sources available on the Internet. According to that approach, since every operation must be characterized by a particular operation class, operations that provide overlapping (non-identical) functionality would need to be assigned different classes as would operations which provide composed functionality (such as, for example, an operation that provides both weather and flight data). The need for an exhaustive set of operation classes (and accompanying training data) is a major limitation of that approach, not shared by the work described in this paper, which relies on a more expressive language for describing service operations.

One way to eliminate the need for a predefined set of operation types is to use unsupervised clustering techniques to generate the (operation) classes automatically from examples (WSDL documents). This idea was implemented in a system called Woogle (Dong et al., 2004). The system clustered service interfaces together using a similarity score based on the co-occurrence of metadata labels. It then took advantage of the clusters produced to improve keyword-based search for Web Services. An advantage of this unsupervised approach is that no labeled training data is required, which can be time-consuming to generate. Such clustering approaches, however, while useful for service discovery, suffer the same limitations as the previous approach when it comes to expressiveness.

8.4 Database Approaches

The database community has long been interested in the problem of integrating data from disparate sources. Specifically, in the areas of data warehousing (Widom, 1995) and information integration (Wiederhold, 1996), researchers are interested in resolving semantic heterogeneity which exists between different databases so that the data can be combined or accessed via a single interface. The *schema mapping problem* is the problem of determining a mapping between the relations contained in a *source schema* and a particular relation in a *target schema*. A mapping defines a transformation which can be used to populate the target relation with data from the source schema. Mappings may be arbitrarily complex procedures, but in general they will be declarative queries in SQL or Datalog. The complexity of these queries makes the *schema mapping problem* far more difficult than the highly investigated *schema matching problem* (Rahm & Bernstein, 2001), which involves finding 1-to-1 correspondences between fields of a source and target schema.

The source definition induction problem can be viewed as a type of schema mapping problem, in which the known sources define the source schema and the unknown source specifies the target relation. In order to solve a schema mapping problem, one typically takes advantage of all available auxiliary information (including source and target data instances, labels from the respective schemas, and so on). Such problems are generally simpler, however, because the data (the extensions of the relations) in the source and target schema are usually explicitly available. In source induction, that data is hidden behind a service interface, which has binding constraints, and the data itself can be extremely large or even (in the case of sources providing mathematical functions) infinite. Thus making the problem considerably more difficult.

The schema integration system CLIO (Yan, Miller, Haas, & Fagin, 2001) helps users build SQL queries that map data from a source to a target schema. In CLIO, foreign keys and instance data are used to generate integration rules semi-automatically. Since CLIO relies heavily on user involvement, it does not make sense to compare it directly with the automated system developed in this paper.

Another closely related problem is that of complex schema matching, the goal of which is to discover complex (many-to-one) mappings between two relational tables or XML schemas. This problem is far more complicated than basic (one-to-one) schema matching because:

- The space of possible correspondences between the relations is no longer the Cartesian product of the source and target relations, but the powerset of the source relation times the target relation.
- Many-to-one mappings require a mapping function, which can be simple like *concatenate(x,y,z)*, or an arbitrarily complex formula such as $z = x^2 + y$.

The iMAP system (Dhamanka et al., 2004) tries to learn such many-to-one mappings between the concepts of a set of source relations and a target relation. It uses a set of *special purpose searchers* to learn different types of mappings (such as mathematical expressions, unit conversions and time/date manipulations). It then uses a meta-heuristic to control the search being performed by the different special purpose searchers. If one views both the source schema and the functions available for use in the mappings (such as *concatenate(x,y,z)*, *add(x,y,z)*, etc.) as the set of known sources in the *source definition induction problem*, then the complex schema matching and source induction problems are somewhat similar. The main differences between the problems are:

- The data associated with the source schema is explicit (and static) in complex schema matching, while it is hidden (and dynamic) in source induction.
- In general, the set of known sources in a source induction problem will be much larger (and the data they provide may be less consistent), than the set of mapping functions and source relations in a complex schema matching problem.

In this paper we develop a general framework for handling the source induction problem. Since iMAP provides functionality which is similar to that of our system, we perform a simple empirical comparison in section 7.4.

8.5 Semantic Web Approach

The stated goal of the Semantic Web (Berners-Lee, Hendler, & Lassila, 2001) is to enable machine understanding of Web resources. This is done by annotating those resources with *semantically meaningful* metadata. Thus the work described in this paper is very much in line with the Semantic Web, in so far as we are attempting to discover semantically meaningful definitions for online information sources. De facto standards for annotating services with *semantic markup* have been around for a number of years. These standards provide service owners with a metadata language for adding declarative statements to service interface descriptions in an attempt to describe the semantics of each service in terms of the functionality (e.g. a *book purchase* operation) or data (e.g. a *weather forecast*) that it provides. Work on these languages is related to this article from two perspectives:

- It can be viewed as an *alternative approach* to gaining knowledge as to the semantics of a newly discovered source (providing it has semantic metadata associated with it).
- Semantic Web Service annotation languages can be seen as a *target language* for the semantic descriptions learnt in this paper.

If a Web Service is already semantically annotated, heterogeneity may still exist between the ontology used by the service provider and that used by the consumer, in which case the learning capabilities described in this paper may be required to reconcile those differences. More importantly, we are interested in the vast number of sources for which semantic markup is currently unavailable. The work in this article complements that of the Semantic Web community by providing a way of automatically annotating sources with semantic information; thereby relieving service providers of the burden of manually annotating their services. Once learnt, Datalog source definitions can be converted to Description Logic-based representations such as is used in OWL-S (Martin, Paolucci, McIlraith, Burstein, McDermott, McGuinness, Parsia, Payne, Sabou, Solanki, Srinivasan, & Sycara, 2004) and WSMO (Roman, Keller, Lausen, de Bruijn, Lara, Stollberg, Polleres, Feier, Bussler, & Fensel, 2005). The reason we use Datalog in this paper (rather than Description Logics) is that most mediator-based integration systems rely on it as a representation language.

9. Discussion

In this paper we have presented a completely automatic approach to learning definitions for online services. Our approach exploits the definition 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 inputs and outputs or the class of a service. We have demonstrated empirically the viability of the approach.

The key contribution of this article is a procedure for learning semantic definitions for online information services that is:

- *Fully automated*: Definitions are learnt in a completely automated manner without the need for any user intervention.
- *More expressive*: The query language for defining sources is that of conjunctive queries, which is far more expressive than previous attribute-value approaches.
- *Sufficiently robust*: The procedure is able to learn definitions in the presence of noisy and incomplete data, and thus is sufficiently robust to handle real data sources.
- *Data-access efficient*: The procedure samples data from live sources, invoking them sparingly and only as required, making it highly efficient in terms of source accesses.
- *Evolving*: The procedure's ability to learn definitions improves over time as each new definition is learnt and added to the set of known sources.

9.1 Application Scenarios

There are a number of different application scenarios for a system that is capable of learning definitions for online sources. They generally involve providing semantic definitions to data integration systems, which then exploit and integrate the available sources.

The most obvious application for our work would be a system (depicted on the left side of Figure 1) that crawls the Web, searching for information sources. Upon finding a source, the system would use a classifier to assign semantic types to it, followed by the inductive learner to generate a definition for it. The definition could then be used to annotate the source for the Semantic Web, or by a mediator for answering queries. Importantly, this entire process could run with minimal user involvement.

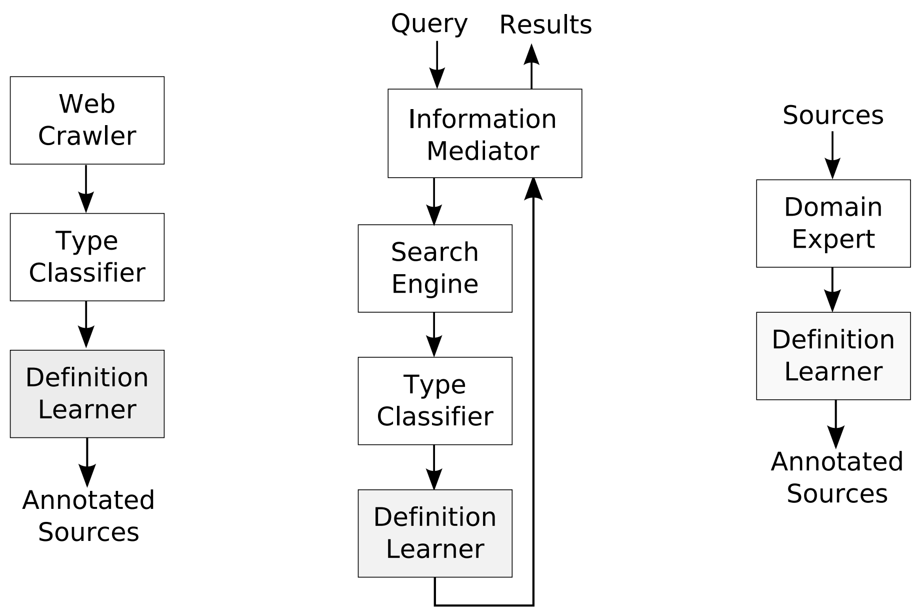

Figure 1: Architecture diagrams for three different application scenarios.

A more challenging application scenario (shown in the center of Figure 1) would involve real-time service discovery. Consider the case where a mediator is unable to answer a particular query because the desired information lies out of scope of the sources available. A search is then performed based on the "missing conjuncts" (relation names and constants) from the query using a specialised Web Service search engine, such as Woogle (Dong et al., 2004). The services returned would be annotated with semantic types and, if possible, semantic definitions. After the definitions are provided to the mediator, it would complete the query processing and return an answer to the user. This scenario may seem a little far-fetched until one considers a specific example: imagine a user interacting with a geospatial browser (an online atlas). If the user turns on a particular information layer, such as ski resorts, but no source is available for the current field of view (of, for instance, Italy), then no results would be displayed. In the background a search could be performed and a new source discovered, which provides ski resorts all over Europe. The relevant data could then be displayed, with the user unaware that a search has been performed.

Perhaps the most likely application scenario (to the right of Figure 1) for a source induction system would be a mixed initiative one. In this case a human would annotate the different operations of a service interface with semantic definitions. At the same time, the system would attempt to induce definitions for the remaining operations, and prompt the user with suggestions for them. In this scenario the classifier may not be needed, since attributes of the same name in the different operations would likely have the same semantic type. Moreover, since the definitions learnt by the system may in some cases contain erroneous or superfluous predicates, the user could also be involved in a process of checking and improving the definitions discovered.

9.2 Opportunities for Further Research

A number of future directions for this work will allow these techniques to be applied more broadly. We now discuss two such directions, improving the search algorithm and extending the query language.

As the number of known sources grows, so too will the search space, and it will be necessary to develop additional heuristics to better direct the search toward the best definition. Many heuristic techniques have been developed in the ILP community and some may be applicable to the source induction problem. More pressing perhaps is the need to develop a robust termination condition for halting the search once a "sufficiently good" definition has been discovered. As the number of available sources increases, the simple timeout used in the experiments will be ineffective as certain (more complicated) definitions will necessarily take longer to learn than others.

Another way to increase the applicability of this work is to extend the query language so that it better describes the sources available. Often online sources do not return a complete set of results but rather cut off the list at some maximum cardinality. For example the *YahooHotel* source described in section 7.3.4 returns a maximum of 20 hotels near a given location, and orders them according to distance. In this case, recognising the specific ordering on the tuples produced would be very useful to a mediator. A second useful extension to the query language would be the ability to describe sources using the procedural construct *if-then-else*. This construct is needed to describe the behaviour of some sources on certain inputs. For example, consider the *YahooGeocoder* from section 7.3.1, which takes as input a tuple containing a street name, number, and zipcode. If the geocoder is unable to locate the corresponding address in its database (because it doesn't exist), instead of returning no tuples, it returns the centroid of the zipcode. Describing such behavior is only possible using procedural constructs.

Acknowledgments

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. The U.S. Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copyright annotation thereon. 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.

References

Afrati, F. N., Li, C., & Mitra, P. (2004). On containment of conjunctive queries using arithmetic comparisions. In *9th International Conference on Extending Database Technology (EDBT 2004)* Heraklion-Crete, Greece.

Arens, Y., Knoblock, C. A., & Shen, W.-M. (1996). Query reformulation for dynamic information integration. *Journal of Intelligent Information Systems - Special Issue on Intelligent Information Integration*, *6*(2/3), 99–130.

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The semantic web. *Scientific American*, *284*(5), 34–43.

Bilenko, M., Mooney, R. J., Cohen, W. W., Ravikumar, P., & Fienberg, S. E. (2003). Adaptive name matching in information integration.. *IEEE Intelligent Systems*, *18*(5), 16–23.

Cameron-Jones, R. M., & Quinlan, J. R. (1994). Efficient top-down induction of logic programs. *SIGART Bulletin*, *5*(1), 33–42.

Carman, M. J., & Knoblock, C. A. (2007). Learning semantic descriptions of web information sources. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)* Hyderabad, India.

Chandra, A. K., & Merlin, P. M. (1977). Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the 9th ACM Symposium on Theory of Computing (STOC)*, pp. 77–90 Boulder, Colorado.

Dhamanka, R., Lee, Y., Doan, A., Halevy, A., & Domingos, P. (2004). imap: Discovering complex semantic matches between database schemas. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*.

Dong, X., Halevy, A. Y., Madhavan, J., Nemes, E., & Zhang, J. (2004). Simlarity search for web services. In *Proceedings of VLDB*.

Duschka, O. M. (1997). *Query Planning and Optimization in Information Integration*. Ph.D. thesis, Department of Computer Science, Stanford University.

Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., & Widom, J. (1995). Integrating and accessing heterogeneous information sources in tsimmis. In *Proceedings of the AAAI Symposium on Information Gathering, pp. 61-64*.

Heß, A., & Kushmerick, N. (2003). Learning to attach semantic metadata to web services. In *2nd International Semantic Web Conference (ISWC)*.

Heß, A., & Kushmerick, N. (2004). Iterative ensemble classification for relational data: A case study of semantic web services. In *15th European Conference on Machine Learning (ECML2004)* Pisa, Italy. Springer.

Knoblock, C. A., Minton, S., Ambite, J. L., Ashish, N., Muslea, I., Philpot, A., & Tejada, S. (2001). The ariadne approach to web-based information integration. *International Journal of Cooperative Information Systems*, *10*(1-2), 145–169.

Lerman, K., Plangprasopchok, A., & Knoblock, C. A. (2006). Automatically labeling data used by web services. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*.

Levy, A. Y. (2000). Logic-based techniques in data integration. In Minker, J. (Ed.), *Logic-Based Artificial Intelligence*. Kluwer Publishers.

Levy, A. Y., Mendelzon, A. O., Sagiv, Y., & Srivastava, D. (1995). Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 95–104 San Jose, Calif.

Markov, Z., & Marinchev, I. (2000). Metric-based inductive learning using semantic height functions. In *Proceedings of the 11th European Conference on Machine Learning (ECML 2000)*. Springer.

Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., & Sycara, K. (2004). 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)*.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, *13*(3-4), 245–286.

Nédellec, C., Rouveirol, C., Adé, H., Bergadano, F., & Tausend, B. (1996). Declarative bias in ILP. In De Raedt, L. (Ed.), *Advances in Inductive Logic Programming*, pp. 82–103. IOS Press.

Pazzani, M. J., & Kibler, D. F. (1992). The utility of knowledge in inductive learning. *Machine Learning*, *9*, 57–94.

Perkowitz, M., & Etzioni, O. (1995). Category translation: Learning to understand information on the internet. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*.

Perkowitz, M., Doorenbos, R. B., Etzioni, O., & Weld, D. S. (1997). Learning to understand information on the internet: An example-based approach. *Journal of Intelligent Information Systems*, *8*(2), 133–153.

Pottinger, R., & Halevy, A. Y. (2001). Minicon: A scalable algorithm for answering queries using views. *VLDB Journal*, *10*(2-3).

Quinlan, J. R., & Cameron-Jones, R. M. (1993). FOIL: A midterm report. In *Machine Learning: ECML-93, European Conference on Machine Learning, Proceedings*, Vol. 667, pp. 3–20. Springer-Verlag.

Rahm, E., & Bernstein, P. (2001). A survey of approaches to automatic schema matching. *VLDB Journal*, *10*(4).

Richards, B. L., & Mooney, R. J. (1992). Learning relations by pathfinding. In *National Conference on Artificial Intelligence*, pp. 50–55.

Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., & Fensel, D. (2005). Web service modeling ontology. *Applied Ontology*, *1*(1), 77–106.

Ullman, J. D. (1989). *Principles of Database and Knowledge-Base Systems*, Vol. 2. Computer Science Press, Rockville, Maryland.

Weber, I., Tausend, B., & Stahl, I. (1995). Language series revisited: The complexity of hypothesis spaces in ILP. In *Proceedings of the 8th European Conference on Machine Learning*, Vol. 912, pp. 360–363. Springer-Verlag.

Widom, J. (1995). Research problems in data warehousing. In *CIKM '95: Proceedings of the fourth International Conference on Information and Knowledge Management*, pp. 25–30. ACM Press.

Wiederhold, G. (1992). Mediators in the architecture of future information systems. *Computer*, *25*(3), 38–49.

Wiederhold, G. (Ed.). (1996). *Intelligent Integration of Information.* Kluwer Academic Publishers, Boston MA.

Winkler, W. (1999). The state of record linkage and current research problems. Tech. rep., Statistical Research Division, U.S. Bureau of the Census, Washington, DC.

Yan, L. L., Miller, R. J., Haas, L. M., & Fagin, R. (2001). Data-driven understanding and refinement of schema mappings. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of data.*

Zelle, J. M., Thompson, C. A., Califf, M. E., & Mooney, R. J. (1995). Inducing logic programs without explicit negative examples. In *Proceedings of the Fifth International Workshop on Inductive Logic Programming.*