Inducing Source Descriptions for Automated Web Service Composition

Mark James Carman and **Craig A. Knoblock**
University of Southern California
Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292
{carman, knoblock}@isi.edu

Abstract

We introduce a framework for learning Local-as-View (LAV) source definitions for the operations provided by a Web Service. These definitions can then be used to *compose* the operations of the Web Service into query execution plans. In order to learn the definition for a new service, we actively invoke the service and compare its output with that of other known services. We combine Inductive Logic Programming (ILP) and Query Reformulation techniques in order to generate and test plausible source definitions for the operations of the service. We test our framework on a real Web Service implementation.

Introduction

New Web Services are being made available on the internet all the time, and while some of them provide completely new functionality, most are slight variations on already existing services. We are interested in the problem of enabling systems to take advantage of these new services without the need for reprogramming. An existing system can only make use of the new service if it has some notion as to what functionality the service provides. Once the system understands this functionality it can compose the service to achieve user-defined goals or incorporate the new service into an already existing workflow. Broadly speaking, there are three approaches to gaining semantic knowledge regarding the functionality of a new service. We call these approaches *standardization*, *semantic markup* and *induction*, and summarise them as follows:

- *Standardisation:* An industry consortium defines a set of standard schemas (possibly in XML Schema) and operations (as WSDL service descriptions) for the information domain. We then require that all service providers use these schemas to describe their data, and those operations to make their data available.
- *Semantic Markup:* Rely on service providers to annotate their services with semantic labels, corresponding to concepts from an ontology, and then employ semantic web techniques to reason about mappings between ontologies which are "understood" by the client, and those that are used by the provider.

- *Induction:* Place no requirements on the service provider, but have the client employ schema matching (Rahm & Bernstein 2001) and service classification techniques (Heß & Kushmerick 2003) to hypothesize what functionality each service might provide.

We follow the third approach, taking the idea one step further by actively querying sources to see if the output agrees with that defined by our model of them. Moreover, we search *not only* for identical services, but try to discern whether a new service has a different scope, or indeed combines the functionality of other known services. We restrict the problem to that of dealing with those operations of a service which only produce information and do not have any affects which "change the state of the world". For example, we are primarily interested in modeling operations which provide access to "flight information" say, but not those which allow the user to "buy a ticket". Information providing Web Services can be composed automatically using query reformulation techniques as described in (Thakkar, Ambite, & Knoblock 2004). We follow the Local-as-View (LAV) (Levy *et al.* 1995) approach to modeling information producing operations (also called sources) in which the information domain is modeled by a set of so-called "domain relations" or "global predicates", and each source is described as a view over the domain relations. The problem of discovering the functionality of a new service can thus be seen as the problem of discovering the LAV source description for each of the operations it provides.

A Motivating Example

Before diving into a description of our source induction framework, we analyse two examples which illustrate more clearly the problem our framework is intended to address. In this first example, we want to determine that the operation provided by a newly discovered service is equivalent to an already known currency exchange-rate operation. The data model for this problem contains two "semantic" data types, called `currency` and `decimal`. Example values for these datatypes, such as $\{\texttt{EUR}, \texttt{USD}, \texttt{AUD}, \texttt{JPY}\}$ and $\{1936.27, 1.30584, 0.531778\}$ are available to the system. The domain description also contains a single global relation, which is defined as:

$$\texttt{exchange}(\texttt{currency}, \texttt{currency}, \texttt{decimal})$$

In addition to the domain model, the problem contains a de-

scription of the services which are known and available to the system. In this case we have a single service which provides the following operation:

LatestRates(country1^b, country2^b, rate):-
exchange(country1, country2, rate)

The superscript b on attributes in the head of a source description indicates that the attribute is an input and needs to be *bound* before invoking the source. Given this background knowledge, the problem is to discover a source description for an operation provided by a newly discovered service. A predicate denoting the "unknown source" is shown below:

RateFinder(fromCountry^b, toCountry^b, rate)

Guided by meta-data similarity (such as the edit distance between the input labels of the known and unknown sources[1]), the system can hypothesize that the input fields of this new service are of type currency. In order to test this hypothesis, it can attempt to invoke the new service using examples of this semantic type as input. Doing so generates tuples as follows:

$\{\langle$EUR, USD, 1.30799$\rangle$, $\langle$USD, EUR, 0.764526$\rangle$, ...$\}$

The fact that tuples are returned by the source means that the classification of the input types is "probably" correct. To be more certain, the system could attempt to invoke the service using examples of the other type, decimal.[2]

Taking into account both the meta-data and the newly generated tuples, the system can classify the output attribute rate to be of type decimal. With the type signature complete, the system now needs to find a definition for the new source. Based on the signature we have two possibilities (labeled $\texttt{def}_1$ and $\texttt{def}_2$) for the definition, both of which involve the predicate exchange:

$\texttt{def}_1$(from^b, to^b, rate):- exchange(from, to, rate)
$\texttt{def}_2$(from^b, to^b, rate):- exchange(to, from, rate)

To test which, if any, of these two source definitions is correct, we need to generate tuples which adhere to each definition and compare these tuples to those produced by the new source. The system can generate tuples for each definition by reformulating the definitions in terms of the known service LatestRates, as follows:

$\texttt{def}_1$(from^b, to^b, rate):- LatestRates(from, to, rate)
$\texttt{def}_2$(from^b, to^b, rate):- LatestRates(to, from, rate)

Invoking the two reformulated definitions using the same inputs as before produces the tuples in the table below:

input	RateFinder	$\texttt{def}_1$	$\texttt{def}_2$
$\langle$EUR, USD$\rangle$	$\langle$1.30799$\rangle$	$\langle$1.30772$\rangle$	$\langle$0.764692$\rangle$
$\langle$USD, EUR$\rangle$	$\langle$0.764526$\rangle$	$\langle$0.764692$\rangle$	$\langle$1.30772$\rangle$
$\langle$EUR, AUD$\rangle$	$\langle$1.68665$\rangle$	$\langle$1.68979$\rangle$	$\langle$0.591789$\rangle$

The tuples produced by $\texttt{def}_1$ are "very similar" although not identical to those produced by RateFinder. Once the system has seen a "sufficient" number of tuples, it will conclude that the best definition for the new service is indeed $\texttt{def}_1$.

[1] See (Heß & Kushmerick 2003) for an approach to matching input and output labels.

[2] The system knows that the examples of decimal are not plausible values for the type currency, because they do not fit a generalised expression for examples of the latter, namely: [A-Z]^3.

A More Complicated Example

We briefly describe a second example which motivates the need for more expressive source descriptions, i.e. descriptions involving joins over different relations. In the example we have five semantic data types, namely: airport, temperature, latitude, longitude, and zip_code. Examples of the first two types are {LAX, JFK, SFO, ORD} and {15F, 12.4F, 70F, 27.8F} respectively. We also have three domain predicates, which are defined as follows:

weather(latitude, longitude, temperature)
zip(latitude, longitude, zip_code)
airport(airport, latitude, longitude)

These predicates are used in the definition of three different sources which are available to the system:

Zip2Temp(zip^b, temp):- zip(X, Y, zip), weather(X, Y, temp)
ZipFind(lat^b, lon^b, zip):- zip(lat, lon, zip)
AirportInfo(iata^b, lat, lon):- airport(iata, lat, lon)

The new service for which we want to discover a definition is denoted: AirportWeather(code^b, temp). By first hypothesizing that the input parameter to the new source is of type airport, the system can invoke the source and produce the following tuples:

$\langle$LAX, 63.0F$\rangle$, $\langle$JFK, 28.9F$\rangle$, $\langle$SFO, 60.1F$\rangle$, $\langle$ORD, 28.0F$\rangle$

As in the previous example, the system compares both the meta-data and data of the output attribute and classifies it to be of type temperature. Having established the type signature, the system can generate plausible definitions for the new service, such as:

$\texttt{def}_1$(code^b, temp):-
airport(code, Y, Z), weather(Y, Z, temp)

The description involves two domain relations, because no single predicate takes both arguments. According to the principle of Occam's Razor the simplest model which explains the data is assumed the most plausible. Here, the simplest model would have been the Cartesian product of the two relations. It is unlikely, however, that somebody would create a service which returns the Cartesian product, when they could provide access to the relations separately. Thus the system starts its search with the more complicated service description, in which attributes of the same type are joined across relations.[3] To test the new source definition, the system reformulates it in terms of the known services as follows:[4]

$\texttt{def}_1$(A, B):-
AirportInfo(A, X, Y), ZipFind(X, Y, Z), Zip2Temp(Z, B)

Now, using the same airport codes, the reformulated query produces the following tuples:

$\langle$LAX, 62F$\rangle$, $\langle$JFK, 28F$\rangle$, $\langle$SFO, 59F$\rangle$, $\langle$ORD, 28F$\rangle$

As in the previous example, the values returned by the reformulated query are "similar" but not identical to those returned by the new source. Thus the system can assume that the definition is probably correct. Notice that the coverage of the new source is different from the query over the other sources, as it can provide weather information about

[3] This simple heuristic provides a kind of inductive search bias (Nédellec *et al.* 1996) for the learning system.

[4] A functional dependency, which holds temperature constant over a zip_code is needed in order to generate this reformulation.

airports outside of the US. This means, that by inducing the definition of a new source using our knowledge of existing sources, we are not only discovering new ways of accessing the same information, but are also expanding the amount of information available for querying!

Problem Definition

We now outline our framework for automatically inducing definitions for the operations of a web service. We first define an instance of the *Source Definition Induction Problem* as a tuple $\langle T, P, S, n \rangle$, where T is a set of *semantic data-types*, P is a set of *domain predicates*, S is a set of *known services*, and n is a *newly discovered service*. In the following sections we define each of the above.

Semantic Data-Types and Domain Predicates

A semantic type $t \in T$ is described by a tuple:

$$\langle parent, regex, Examples, metric \rangle$$

Semantic data-types are arranged in a taxonomy, with $parent$ being the super-type. The regular expression $regex$ defines the *canonical* form of values of this type. We need a reference format since different services may input or output the same value in different formats. For example, one service may output a *USTelephoneNumber* as (213) 129 2333, while another service outputs the same value as 213-129-2333. $Examples$ is a set of possible values for the semantic type. Finally, *metric* is a function which returns similarity scores for pairs of values of the type.

The set of predicates P is made up of global relations (from the mediated schema) and interpreted predicates. Interpreted predicates are predicates for which the system has a local implementation. There will be a number of such predicates available in the system, such as inequality $\leq$ and equality $=$, possibly some mathematical functions (e.g.$+$), string transformations such as $\texttt{concatenate}(\texttt{s1}, \texttt{s2}, \texttt{s3})$, precision operators like $\texttt{round}(\texttt{accuracy}, \texttt{v1}, \texttt{v2})$, and data-type specific transformations such as $\texttt{Celsius2Fahrenheit}(\texttt{t1}, \texttt{t2})$.

Each predicate has typed arguments and may respect some functional dependencies. Functional dependencies express constraints over the data, stating that one or more attributes of a predicate are determined by the values of other attributes. They can be used to define transformations between different representations of the same information. For example the predicate $\texttt{Celsius2Fahrenheit}(\texttt{t1}, \texttt{t2})$ has a functional dependency: $\texttt{t1} \leftrightarrow \texttt{t2}$, (stating that there is only one Fahrenheit value for any given Celsius value and vice versa). Knowledge of functional dependencies can be critical for query reformulation. In some cases functional dependencies may not be given in the problem specification, but may need to be mined from the data.

Service Descriptions

S is a set of known services. Each of the services has been annotated with semantic information in terms of the domain model (i.e. T and P). An annotated service $s \in S$ can be written as follows:

$$\langle name, url, desc, Operations \rangle$$

The labels $name$ and url are identifiers for the service, while $desc$ is a string in natural language describing its functionality. The set of $Operations$ are the information sources provided by the service. Each operation can be described as follows:

$$\langle name, desc, Inputs, Outputs, view \rangle$$

The labels $name$ and $desc$ are strings in natural language describing the operation. Each $input$ and $output$ is a triple $\langle name, type, transformation \rangle$. The $name$ is the meta-data describing the input, which may include a path through a hierarchy of XML schema tags. The $type$ field is the annotation of this input/output with a semantic data-type from T. The $transformation$ is a function for converting values to/from the reference format. In addition to the semantic type annotation, each operation is associated with a $view$ definition which is a conjunctive query over the domain predicates, P.

The new service n is defined in the same way as above, except that its operations are not annotated with semantic types, and it does not have a conjunctive query definition associated with it. The *Source Definition Induction Problem* is to automatically generate the semantic annotation for this new service, given the annotation of the known services.

The Algorithm

We separate the procedure for inducing source descriptions into two phases. In the first phase each of the input and output attributes of the new service are classified with a semantic types. This step involves schema matching techniques (Rahm & Bernstein 2001) as well as the active invocation of services, as in (Johnston & Kushmerick 2004). In the second phase, the new type signature is used to generate candidate hypotheses for the Datalog description of the new source. The space of such hypotheses is searched and candidate hypotheses are tested to see if the data they return agrees with their description. The two phases of the algorithm can be expressed as follows:

Phase I:

1. Generate hypothesis regarding input/output types
2. Attempt to invoke service using examples of input types
3. Repeat until valid hypothesis is found

Phase II:

1. Generate candidate hypotheses for new source definition
2. Reformulate hypotheses as queries over known sources
3. Execute queries and invoke source using randomly selected input tuple[5]
4. Rank hypotheses according to output similarity
5. Discard poorly performing hypotheses
6. Add more complicated hypotheses and select next input
7. Repeat until best score converges on a single hypothesis

[5]Note that, when a source inputs a tuple of cardinality greater than one it can be non-trivial to choose "valid" combinations of input attribute values.

Assumptions Regarding Source Definitions

We make a number of assumptions regarding the definition of the new source. These assumptions can be summarised as follows:

1. Function-free safe Datalog queries (no aggregate operators)
2. No-recursion or disjunction[6] (no union queries)
3. Negation-free (no set difference or $\neq$)
4. Open-world semantics (sources are incomplete)

The first condition states that in order for the semantics of an operation to be learnt it must be expressible in Datalog. The second and third conditions represent simplifying assumptions which hold for most service operations, and which greatly reduce the search space of possible definitions. In contrast, the final condition regarding the incompleteness of the sources actually complicates the problem. This assumption is necessary, however, as it is frequently the case that web services are incomplete with respect to the tuples that fit their definitions.

Comparing Candidate Hypotheses

In the motivating examples, at most one output tuple was produces for every input tuple. This was the case because functional dependencies existed over the predicates which defined the sources. In general, functional dependencies will not necessarily exist, and multiple output tuples will be produced by each source for a given input tuple.

The problem of discovering which tuples from two sources represent the same real world entity is referred to as the duplicate detection problem in the record linkage literature. In general, one would need to employ such techniques to compare the outputs of the new source and the reformulated query to see which (if any) of the tuples are the same. Note that both the new service and the existing sources are assumed to be incomplete. Thus even if the hypothesis regarding the description of the new source is correct, the set of tuples it returns will not necessarily be a subset of those returned by the reformulation! The two sets may simply overlap, which is a problem, given that we are trying to show at this point that the hypothesis is contained in the "true source description". Assuming that we can count the number of tuples that are the same, we need a measure which tells us which of the candidate hypotheses best describes the data returned by the new source. One such measure is the following:

$$Score = \frac{2}{|I|} \sum_{i \in I} \frac{|O_n(i) \cap O_d(i)|}{|O_n(i)| + |O_d(i)|}$$

where I is the set of input tuples used to test the source. $O_n(i)$ denotes the set of tuples returned by the new source when invoked with input tuple i. $O_d(i)$ is the corresponding set returned by the reformulation of definition d. If we view this hypothesis testing as an information retrieval task, we can consider *recall* (resp. *precision*) to be the number of common tuples, divided by the number of tuples produced by the source (reformulated query). Under this interpretation, the above score is the average *F1-Measure* over the input trials.

[6] We plan to investigate definitions involving a limited form of disjunction where source functionality depends on the value of an input attribute of an enumerated type with low cardinality.

Case Study and Experiments

In order to evaluate the feasibility of our proposal for inducing source descriptions, we investigated a use-case in which the user is currently annotating the operations of a new service. Since the different operations of a given service are usually closely related, it may be possible to induce the source description for one operation using the same predicates in the annotated operation. By concentrating on different operations of the same service, we can to a large extent ignore the problems of type matching and data reformating.

The service we investigated provides geospatial data and is called *ZipCodeLookup*[7]. Three of the operations provided by the service are shown below[8]:

1. $\texttt{getDistanceBetweenZipCodes}(\texttt{zip1}^\texttt{b}, \texttt{zip2}^\texttt{b}, \texttt{dist})$
2. $\texttt{getZipCodesWithin}(\texttt{zip1}^\texttt{b}, \texttt{dist1}^\texttt{b}, \texttt{zip2}, \texttt{dist2})$
3. $\texttt{getZipCodeCoords}(\texttt{zip}^\texttt{b}, \texttt{lat}, \texttt{long}, \texttt{latR}, \texttt{longR})$[9]

In our test case, the user is currently annotating the web service description with a source definition for each operation. She decides the best definition for the first operation will be:

$\texttt{getDistanceBetweenZipCodes}(\texttt{zip1}^\texttt{b}, \texttt{zip2}^\texttt{b}, \texttt{dist})$:-
  $\texttt{centroid}(\texttt{zip1}, \texttt{lat1}, \texttt{long1})$,
  $\texttt{centroid}(\texttt{zip2}, \texttt{lat2}, \texttt{long2})$,
  $\texttt{distanceInMiles}(\texttt{lat1}, \texttt{long1}, \texttt{lat2}, \texttt{long2}, \texttt{dist})$

Given this source description the system is now in a position to induce source descriptions for the other operations. It can attempt to "fill in" the missing source descriptions automatically. The system can do this because it has a set of typed predicates, namely `centroid` and `distanceInMiles`, which it can use to build source descriptions for the other operations. The semantic types in this domain are *zipCode*, *distance*, *latitudeInDegrees*, and so on. In addition to the user defined predicates, we assume that the system also has access to a definition for the inequality predicate $\leq$, which is typed to accept real valued attributes, such as *distance* or *latituteInDegrees*. (Because of symmetry, there is no need for the corresponding $\geq$ predicate to be made available.) The system will also have access to a definition for the equality predicate $=$, which can be applied to attributes of the same type.

Generating Plausible Definitions

In order to induce a definition for the source predicate `getZipCodesWithin`, the system first investigates the type signature of the new predicate. Since the signature of this

[7] The WSDL file describing this service can be found at http://www.codebump.com/services/zipcodelookup.asmx?WSDL

[8] The other operations provided by the service are either a) too complicated for inclusion in the current use case, or b) involve "world altering effects" (such as saving information on the server) and cannot therefore be expressed in Datalog.

[9] `lat` and `long` are short for latitude and longitude *in Degrees*, while `latR` and `longR` are short for the same *in Radians*

	Possible Source Description	Cumulative Scores
1	$\texttt{cen}(\texttt{z1}, \texttt{lt1}, \texttt{lg1}), \texttt{cen}(\texttt{z2}, \texttt{lt2}, \texttt{lg2}), \texttt{dIM}(\texttt{lt1}, \texttt{lg1}, \texttt{lt2}, \texttt{lg2}, \texttt{d1}), (\texttt{d2} = \texttt{d1})$	0.00, 0.00, 0.33, ...
2	$\texttt{cen}(\texttt{z1}, \texttt{lt1}, \texttt{lg1}), \texttt{cen}(\texttt{z2}, \texttt{lt2}, \texttt{lg2}), \texttt{dIM}(\texttt{lt1}, \texttt{lg1}, \texttt{lt2}, \texttt{lg2}, \texttt{d1}), (\texttt{d2} \leq \texttt{d1})$	invalid
3	$\texttt{cen}(\texttt{z1}, \texttt{lt1}, \texttt{lg1}), \texttt{cen}(\texttt{z2}, \texttt{lt2}, \texttt{lg2}), \texttt{dIM}(\texttt{lt1}, \texttt{lg1}, \texttt{lt2}, \texttt{lg2}, \texttt{d2}), (\texttt{d2} \leq \texttt{d1})$	0.86, 0.88, 0.59, ...
4	$\texttt{cen}(\texttt{z1}, \texttt{lt1}, \texttt{lg1}), \texttt{cen}(\texttt{z2}, \texttt{lt2}, \texttt{lg2}), \texttt{dIM}(\texttt{lt1}, \texttt{lg1}, \texttt{lt2}, \texttt{lg2}, \texttt{d2}), (\texttt{d1} \leq \texttt{d2})$	0.00, 0.00, 0.00, ...
5	$\texttt{cen}(\texttt{z1}, \texttt{lt1}, \texttt{lg1}), \texttt{cen}(\texttt{z2}, \texttt{lt2}, \texttt{lg2}), \texttt{dIM}(\texttt{lt1}, \texttt{lg1}, \texttt{lt2}, \texttt{lg2}, \texttt{d2}), (\texttt{d1} \leq \#\texttt{d})$	0.80, 0.82, 0.88, ...
6	$\texttt{cen}(\texttt{z1}, \texttt{lt1}, \texttt{lg1}), \texttt{cen}(\texttt{z2}, \texttt{lt2}, \texttt{lg2}), \texttt{dIM}(\texttt{lt1}, \texttt{lg1}, \texttt{lt2}, \texttt{lg2}, \texttt{d2}), (\texttt{lt1} \leq \texttt{d1})$	uncheckable
	$\vdots$	
n	$\texttt{cen}(\texttt{z1}, \texttt{lt1}, \texttt{lg1}), \texttt{cen}(\texttt{z2}, \texttt{lt2}, \texttt{lg2}), \texttt{dIM}(\texttt{lt1}, \texttt{lg1}, \texttt{lt2}, \texttt{lg2}, \texttt{d2}), (\texttt{d2} \leq \texttt{d1}), (\texttt{d1} \leq \#\texttt{d})$	0.86, 0.88, 0.92, ...

Table 1: Inducing Source Descriptions for $\texttt{getZipCodesWithin}(\texttt{z1}^{\texttt{b}}, \texttt{d1}^{\texttt{b}}, \texttt{z2}, \texttt{d2})$

predicate contains all of the types in the signature of the known source predicate `getDistanceBetweenZipCodes`, a valid starting point for generating plausible definitions for the former is the definition of the latter. (It seems a valid assumption that the structure of different sources of the same service will share a similar Datalog structure, such as joins over existential variables like `lt1` and `lg1`.) The system can perform a local search starting from this source description by adding, deleting and altering the predicates it contains. Some plausible source descriptions which result from adding new predicates to the definition of `getDistanceBetweenZipCodes` are shown in table 1 in an order in which they might be generated during search. (The predicate and attribute names have been abbreviated to save space.)

This first definition shown in the table returns all zip codes which lie exactly `d1` miles from zip code `z1`. It also returns the input distance `d1` as the output distance `d2`. (Note, that it is not uncommon for the operations of a service to "echo" an input value as an output value.) A single equality predicate has been added to the initial source description. This predicate was needed to bind a value to the output attribute `d2`.

The second definition has been inserted in the table to show that not all possible predicate additions result in valid definitions. This definition is invalid and would not be generated by the search procedure, because the comparison predicate $\leq$ cannot be used to bind the variable `d2` to a value. Intuitively, if $\leq$ did bind `d2` to a value, it would have to bind it to every possible value less than `d1`, resulting in an infinite set of output tuples.

In contrast, the third definition is valid, because the variable `d2` is already bound to a value (by the predicate `dIM`) before being compared using the inequality predicate. Note that `d1` is an input variable (bound attribute) of the unknown source, so it does not need to be bound by the $\leq$ predicate. According to this definition, the source predicate returns all zip codes which lie within the distance `d1` of the zip `z1`. It also returns the distance to each of the zip codes found. Conversely, the fourth definition returns all zip codes which lie outside of the area `d1` miles from `z1`, along with their respective distances.

The fifth definition involves a comparison containing the symbol $\#\texttt{d}$, which represents a constant of type *distance*. The particular value for this constant will not be chosen until after invoking the reformulated definition, and will be chosen so as to maximise the score for this definition. Choosing the value prior to executing wouldn't make sense, as the continuous variable would mean that arbitrarily many hypotheses of this type would need to be generated.

The sixth definition returns tuples where the latitude attribute `lt1` is less than the distance `d1`. This comparison may seem odd given an intuitive notion as to the inherent usefulness of a given query, but unless told otherwise it makes perfect sense to the search procedure. Depending on the domain, we might want to explicitly rule out such comparisons, thereby providing an "inductive language bias" to the system. In order to check the correctness of each possible definition against the source, we must first reformulate the definition into a Datalog query over the sources available. This new definition cannot be reformulated in terms of the sources available, as the existential variable `lt1` is not accessible in the source `getDistanceBetweenZipCodes`. Thus the hypothesis is "uncheckable" and will be disregarded during the search.

The last definition shown on the table contains two additional predicates. It is in fact contained in both the third and fifth definition, meaning it will only ever return a subset of the tuples returned by those definitions. Contained queries are useful whenever a definition is found to be too general to fit the data, i.e. more tuples are returned by the definition than the unknown source.

Testing the Definitions

We now proceed to the problem of testing the hypotheses generated. The system has the following examples of the type *zipCode*: $\{\texttt{90292}, \texttt{90266}, \texttt{89101}, \texttt{79936}, \texttt{08008}\}$ (We note that the larger and "more representative" the set of initial examples, the more likely the system is to discover the correct definition for the service.) The system doesn't have any examples of the type *distance*, which it needs in order to invoke the unknown source. It can generate examples of this type by executing the known source with (possibly all) combinations of the example zip codes. Doing so produces the following examples: $\{\texttt{6.6}, \texttt{241.9}, \texttt{722.5}, \texttt{2452.8}, ...\}$[10]

[10] Values have been rounded from the 9-figure precision returned by the service.

The system can now proceed to pick an input tuple. In theory the input tuple should be constructed by randomly selecting values from the examples of each type. In practice, since the number of valid input combinations may be low, it could be best to select values that have "appeared" together in a tuple before. Supposing the system were to "randomly" chose the input tuple $\langle 90292, 6.6 \rangle$, a call to the unknown source `getZipCodesWithin` produces 48 tuples. The system now needs to compare this set with that generated by each of the possible definitions. Generating tuples for each definition is not a straightforward task however. Taking the first definition as an example, the system can reformulate this definition in terms of the source available as:

$\texttt{def}_1(\texttt{z1}^\texttt{b}, \texttt{d1}, \texttt{z2}^\texttt{b}, \texttt{d2})$:-
  $\texttt{getDistanceBetweenZipCodes}(\texttt{z1}^\texttt{b}, \texttt{z2}^\texttt{b}, \texttt{d1})$,
  $(\texttt{d2} = \texttt{d1})$

We note that the binding pattern of the reformulated definition is different from that of the unknown source. This means that the system cannot simply give the input tuple $\langle \texttt{z1}, \texttt{d1} \rangle$ to the reformulated definition. Instead, it also needs to bind a value for `z2` each time it calls the reformulation. Since the system is interested in generating all tuples regardless of the value for `z2`, it should invoke the reformulation for *every possible value* of that attribute. This presents us with two problems. Firstly, the system does not have a list of all the values that a variable of type *zipCode* can take. In theory, it could attempt to generate such a list by repeatedly invoking sources with output attributes of this type. Doing so would consume a large amount of bandwidth and may result in the system being blocked from further use of the service.[11] Thus the set of example zip codes "discovered thus far" is assumed to be a sufficiently large domain of values for `z2`. Secondly, this domain of values may be very large, (numbering in the thousands after a few iterations), resulting in a large number of invocations to test each definition. To limit the number of invocations, the system randomly samples values for `z2` from the domain of *zipCodes* and uses the resulting tuples to approximate the score for the definition. (In the experiments, 20 samples were selected for each iteration.)

In this way, each reformulated source definition is invoked using the input tuple $\langle 90292, 6.6 \rangle$ and known values for `z2`. The score for each definition is the first value in the column *Cumulative Scores* of the table. The definitions are then tested with other randomly selected input tuples, producing the rest of the values in the column.

After a sufficient number of input tuples have been tested, the algorithm converges on a definition for the unknown operation. The convergence criterion we use is the statistical significance of the difference between the scores of the best and the second best definitions. The system stops generating input tuples when it becomes 95% confident that the best definition is indeed the better than all of the other definitions for the source. The actual definition found in this case is:

$\texttt{getZipCodesWithin}(\texttt{zip1}^\texttt{b}, \texttt{dist1}^\texttt{b}, \texttt{zip2}, \texttt{dist2})$:-
  $\texttt{centroid}(\texttt{zip1}, \texttt{lat1}, \texttt{long1})$,
  $\texttt{centroid}(\texttt{zip2}, \texttt{lat2}, \texttt{long2})$,
  $\texttt{distanceInMiles}(\texttt{lat1}, \texttt{long1}, \texttt{lat2}, \texttt{long2}, \texttt{dist2})$,
  $(\texttt{dist2} \leq \texttt{dist1})$,
  $(\texttt{dist1} \leq 243.3)$.

The "true" source definition has the range restriction $(\texttt{dist1} \leq 300)$. The accuracy of the definition produced by the system, depends on the number of `zipCode` examples made initially available to the system (as well as the set of distance values generated during execution).

Reversing the Process for the Third Operation

In order to induce a definition for the third source predicate, `getZipCodeCoords`, we must employ a slightly different procedure from that used for the second predicate. Firstly we need to assume that an additional domain predicate, $\texttt{degrees2radians}(\texttt{latD}, \texttt{longD}, \texttt{latR}, \texttt{longR})$, is available as an interpreted predicate in the system. Using this predicate and taking the type signature into account, the system can generate $\texttt{def}_1$ as a plausible definition for the unknown source:

$\texttt{def}_1(\texttt{zip}, \texttt{lat}, \texttt{long}, \texttt{latR}, \texttt{longR})$:-
  $\texttt{centroid}(\texttt{zip}, \texttt{lat}, \texttt{long})$,
  $\texttt{degrees2radians}(\texttt{lat}, \texttt{long}, \texttt{latR}, \texttt{longR})$.

Obviously, we cannot reformulate a query with this definition in terms of the known source predicate. It would appear, therefore, that we cannot check whether this definition is correct or not. We can, however, test the approximate correctness of this operation by inverting our usual approach. Instead of viewing this definition as a query against the sources available, we can see if another source definition can be reformulated in terms of this new definition. In particular, we can reformulate the definition for the known predicate `getDistanceBetweenZipCodes` in terms of $\texttt{def}_1$, as shown below:

$\texttt{getDistanceBetweenZipCodes}(\texttt{zip1}, \texttt{zip2}, \texttt{dist})$:-
  $\texttt{def}_1(\texttt{zip1}, \texttt{lat1}, \texttt{long1}, \texttt{latR1}, \texttt{longR1})$,
  $\texttt{def}_1(\texttt{zip2}, \texttt{lat2}, \texttt{long2}, \texttt{latR2}, \texttt{longR2})$,
  $\texttt{distanceInMiles}(\texttt{lat1}, \texttt{long1}, \texttt{lat2}, \texttt{long2}, \texttt{dist})$.

If we assume, that an implementation for the predicate `distanceInMiles` is made available (via a local routine or another service), then the system can execute this reformulation. By comparing the tuples produced by the reformulation to those produced by the first source, it can check whether the first predicate `centroid` of the new source definition is correct.

The story does not end here, however, as the second predicate in $\texttt{def}_1$ was not needed in order for the above reformulation to hold. Thus the system still needs to check whether the second part of the definition is correct. Since a local definition is available for the interpreted predicate `degrees2radians`, the system can perform the same trick as before, reformulating the known predicate in terms of $\texttt{def}_1$ as follows:

$\texttt{degrees2radians}(\texttt{lat}, \texttt{long}, \texttt{latR}, \texttt{longR})$:-
  $\texttt{def}_1(\texttt{zip}, \texttt{lat}, \texttt{long}, \texttt{latR}, \texttt{longR})$.

Then by invoking the new service using different zip codes as input, it can verify that the relationship between the coordinates is in fact the same as that predicted by the known

[11] Flooding a service with invocations could be misinterpreted as a denial-of-service attack.

predicate, meaning that $\texttt{def}_1$ is indeed the correct definition for the operation.

Related Work

This work is closely related to the category translation problem defined in (Perkowitz & Etzioni 1995). Our approach differs in that we focus on a relational modeling of the sources, and on inducing joins between domain relations, rather than nested function applications. Nonetheless, strategies they apply for choosing the most relevant values to give as input may apply directly to this work. This work also relates to that of (Johnston & Kushmerick 2004), who actively query sources to determine whether schema matching performed over the input/output datatypes was correct. The difference between their work and ours, is that instead of just learning how the input/output types map to some global schema, we are trying to learn the the source description itself, i.e. *how* the input and output *relate to each other* in terms of the domain model.

Our work also relates to the schema integration system CLIO (Yan *et al.* 2001), which helps users build queries that map data from a source to a target schema. If we view the source schema as unknown sources, and the target schema as global relations, then CLIO generates Global-as-View (not LAV) integration rules. In CLIO, the integration rules are not generated automatically, although some prompting from the system is provided. The iMAP system (Dhamanka *et al.* 2004) tries to learn the complex (many-to-one) mappings between the concepts of a source and target schema automatically. It uses a set of "special purpose searchers" to learn different types of mappings. In our work we concentrate on developing a more general framework employing Inductive Logic Programming (ILP) techniques. A second difference between our work and theirs, is that they assume to have instances (sets of tuples) of the source and target schema available. In our work we need also to deal with the problem of generating such data, i.e. deciding when and how to invoke the sources available.

Discussion

In this paper we have introduced a framework for automatically learning definitions for newly discovered services based on knowledge of the services already available. The definitions learnt can then be used to *compose* these services into query execution plans.

Our work builds on ideas from Inductive Logic Programming (ILP), but is complicated by the fact that in contrast to ILP, the extensions of the domain predicates are not directly accessible. Instead, the domain relations must be accessed via the sources available, making query reformulation an integral aspect of the problem. Furthermore, as is generally the case for LAV models, we cannot assume sources to be complete with respect to their source definitions. Thus we must use an approximate measure to compare candidate definitions with one another.

This work is preliminary, and the framework described is missing certain details. One such detail is the size of the search space that needs to be explored before we can apply the convergence criterion. In other words, how many candidate hypotheses need to be generated before we can be confident the "correct" definition is amongst them? Heuristics for guiding the search through the space of source definitions are also important as the search space is very large. Finally, we need to incorporate techniques from record linkage (Michalowski, Thakkar, & Knoblock 2005) for testing whether two sources are producing the same tuples, and grammar induction algorithms (Lerman, Minton, & Knoblock 2003) for learning data-type descriptions from examples, for use in classifying inputs and outputs.

Acknowledgements

We would like to thank Snehal Thakkar, José Luis Ambite and Kristina Lerman for many useful discussions regarding different aspects of this work.

This research is based upon work supported in part by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010, in part by the National Science Foundation under Award No. IIS-0324955, and in part by the Air Force Office of Scientific Research under grant number FA9550-04-1-0105.

The 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

Dhamanka, R.; Lee, Y.; Doan, A.; Halevy, A.; and 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*.

Heß, A., and Kushmerick, N. 2003. Automatically attaching semantic metadata to web services. In *IJCAI-2003 Workshop on Information Integration on the Web*.

Johnston, E., and Kushmerick, N. 2004. Aggregating web services with active invocation and ensembles of string distance metrics. In *EKAW'04*.

Lerman, K.; Minton, S.; and Knoblock, C. 2003. Wrapper maintenance: A machine learning approach. *JAIR* 18:149–181.

Levy, A. Y.; Mendelzon, A. O.; Sagiv, Y.; and Srivastava, D. 1995. Answering queries using views. In *PODS'95*, 95–104.

Michalowski, M.; Thakkar, S.; and Knoblock, C. A. 2005. Automatically utilizing secondary sources to align information across data sources. *AI Magazine, Special Issue on Semantic Integration* 26(1).

Nédellec, C.; Rouveirol, C.; Adé, H.; Bergadano, F.; and Tausend, B. 1996. Declarative bias in ILP. In De Raedt, L.,

ed., *Advances in Inductive Logic Programming*. IOS Press. 82–103.

Perkowitz, M., and Etzioni, O. 1995. Category translation: Learning to understand information on the internet. In *IJCAI-95*.

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

Thakkar, S.; Ambite, J. L.; and Knoblock, C. A. 2004. A data integration approach to automatically composing and optimizing web services. In *In Proceeding of 2004 ICAPS Workshop on Planning and Scheduling for Web and Grid Services*.

Yan, L. L.; Miller, R. J.; Haas, L. M.; and 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*.