Values Normalization with Logic Programming

René Pázman

Softec Ltd., Bratislava, Slovakia,
rene.pazman@softec.sk,
http://rene.pazman.googlepages.com/

Abstract. There are many cases of numeric data in databases representing the same thing that are directly incomparable because of their distinct units. We aim to solve the problem by normalization of the values into same units. The data can be seen as logic statements — especially in ontologies which we use in our project. Utilizing this fact we use logic programming approach to infer the way of values conversion. We explore two common approaches of logic programming: Prolog and ASP and compare the solutions to each other.

1 Introduction

In systems storing some data, which is almost any information system, there are often some data which should be compared, but they are incomparable because they are stored in distinct units. Very frequent example is the case of monetary values — especially in Europe Union, where new members are transforming their currencies into Euro.

The straightforward solution to the unit conversion problem is to gain conversion table for the units used, where coefficients for conversion from any unit to any other unit are held. Such table can be directly used in comparisions, or it can be utilized for storing of normalized values for each source value. E.g., the conversion table for the case of monetary values could consists of some bank's exchange rates. These rates can be used for computing values in one chosen currency, e.g. EUR. Such values can be stored as normalized values and they are directly comparable with each other.

However, there are more complex units, for which possible conversion table could be rather large, for example velocity can be expressed in meters per second, miles per hour etc. Such cases are characterised by values which units are composed of other (basic) units, which is the reason of increase of the conversion table size.

In our project ([6]), we faced the similar problem. We are storing values of salaries of job offers, which are naturally expressed in various units, e.g. EUR per year, or USD per hour. We wanted to compare salary information directly between distinct job offers and thus we needed a method for automatic value conversion into a given normalized unit. We decided to store the normalized values in the original database, which is an RDF ontology of job offers gained from the Web. In order to gain higher flexibility, we did not decide to implement conversions into the chosen normalized unit only, but allow conversions between any two possible units.

In the spirit of the above–mentioned examples, a possible solution should satisfy following requirements:

- Direct conversion coefficients should be held only for atomic units, which are not composed of other units.
- Conversion coefficients need not be held for each pair of atomic units — the conversion table need not be fully pre-computed.
- Units composed of other units are converted using conversions between atomic units.
- The way of particular value conversion should be based on declaratively stated relationships between basic units, and should be dynamically computed.
- The method should be universal — usable for conversion between any comparable units.
- The method should be flexible enough — e.g. change of normalized unit or adding a new unit are easy changes.

2 Method

In order to obtain such conversion functionality, we have considered more approaches of its realization.

We considered methods of automatic data creation at first, by which we could compute and store normalized values. These methods allow us to define simple rules for automatic data addition or change in the moment of creation or change of some other data of a given type. These methods are often provided in the form of triggers or rules on the side of data repository; in the form of inference — or reasoning — rules especially for ontologies.

Our ontology repository is based on Sesame ontology server ([4], [7]). It provides three kinds of reasoning tools: RDF inference, OWL inference ([5]), and Sesame's custom reasoning. Unfortunately, all of these reasoning capabilities were examined as not suitable for our goals expressed as the requirements in the previous section, which was the reason we decided to create our own method of values normalization.

Any data stored in a database, and especially in the ontology, can be viewed as logic statements about entities described by the data. Inferencing new statements from existing statements is the focus of logic programming (LP) approach which we choose as the implementation platform for values normalization. Logic programming is a tool universal enough for the realization of values conversions.

We tested two LP approaches: Prolog ([9]) and Answer Set Programming — ASP ([2]). We created two prototypes — one for each LP approach. The expressive power of Prolog is far above the expressive power of ASP. Thus we choose to create the Prolog program first, and then we tried to convert it into an ASP program.

In the next, we sketches the main parts of rules used within both prototypes. The approach is based on the following groups of rules:

1. Rules for known conversion coefficients between atomic units, e.g.

   ```
   coefficient(day,hour,24).
   coefficient(year,day,365).
   coefficient(usd,skk,25.768).
   coefficient(eur,skk,34.065).
   ```

2. Rules for normalized units definition for each value type, e.g.

   ```
   hasNormalizedUnit(salary,division(eur,year)).
   ```

3. Rules for direct conversion using coefficients for atomic units, e.g.

   ```
   exchangeCoefficient(Unit1,Unit2,Coeff) :-
       coefficient(Unit1,Unit2,Coeff).
   ```

4. Rules for indirect conversion between atomic units using more coefficients, e.g.

   ```
   coefficient(Unit1,Unit2,multiplication(Coeff1,Coeff2)) :-
       coefficient(Unit1,Unit3,Coeff1),
       coefficient(Unit3,Unit2,Coeff2),
       Unit1 != Unit2.
   ```

5. Rules for conversion of compound units, e.g.

   ```
   exchangeCoefficient(
               multiplication(Unit1From,Unit2From),
               multiplication(Unit1To,Unit2To),
               multiplication(Coefficient1, Coefficient2)) :-
       exchangeCoefficient(Unit1From,Unit1To,Coefficient1),
       exchangeCoefficient(Unit2From,Unit2To,Coefficient2).
   ```

6. A rule for the computation of the normalized value:

   ```
   hasNormalizedValue(
               Object,QuantityProperty,
               multiplication(OriginalValue,Coefficient)) :-
       hasProperty(Object,QuantityType,QuantityProperty,
                   OriginalValue,OriginalUnit),
       hasNormalizedUnit(QuantityType,NormalizedUnit),
       exchangeCoefficient(OriginalUnit,NormalizedUnit,Coefficient).
   ```

   Predicate `hasProperty` is defined by those properties of job offers, which should be normalized.

These rules are only schemes of rules used in logic programs. Real rules differ from these ones dependent on the LP approach used. In Prolog, the rules are directly usable with some little changes. The situation is more complicated in ASP. The rule schemes 4 and 5 cannot be directly used in ASP.

The reason is the expressive power of ASP — any ASP solver is basically a propositional logic solver. The tools, which allow to introduce variables, function symbols etc. are called parsers, and they translate a logic program into a program appropriate for ASP solvers. The result of the translation is a grounded propositional program.

In order to allow such translation, an original program must fulfill some specific constraints, and the rule schemes 4 and 5 fail to fulfill them. Such rules generate a processing error, or their processing time is either infinite or too long. The actual constraints depend on the parser used.

Actual rules, which replace the rules 4 and 5 (and also the rule scheme 3) in the ASP prototype are the following:

```
simpleCoefficient0(Unit,Unit,1) :- isAtomicUnit(Unit).
simpleCoefficient0(Unit1,Unit2,Coeff) :-
    coefficient(Unit1,Unit2,Coeff ).
simpleCoefficient0(Unit1,Unit2,division(1,Coeff)) :-
    coefficient(Unit2,Unit1,Coeff).

simpleConvertible0(Unit1,Unit2) :-
    simpleCoefficient0(Unit1,Unit2,Coeff).

simpleCoefficient1(Unit1,Unit2,Coeff) :-
    simpleCoefficient0(Unit1,Unit2,Coeff).
simpleCoefficient1(Unit1,Unit2,multiplication(Coeff1,Coeff2)) :-
    not simpleConvertible0(Unit1,Unit2),
    simpleCoefficient0(Unit1,Unit3,Coeff1),
    simpleCoefficient0(Unit3,Unit2,Coeff2),
    Unit1 != Unit2.

simpleConvertible1(Unit1,Unit2) :-
    simpleConvertible0(Unit1,Unit2).
simpleConvertible1(Unit1,Unit2) :-
    simpleCoefficient1(Unit1,Unit2,Coeff).

simpleCoefficient2(Unit1,Unit2,Coeff) :-
    simpleCoefficient1(Unit1,Unit2,Coeff).
simpleCoefficient2(Unit1,Unit2,multiplication(Coeff1,Coeff2)) :-
    not simpleConvertible1(Unit1,Unit2),
    simpleCoefficient1(Unit1,Unit3,Coeff1),
    simpleCoefficient1(Unit3,Unit2,Coeff2),
    Unit1 != Unit2.
...
```

```
simpleCoefficient(Unit1,Unit2,Coeff) :-
    simpleCoefficient<MAX_NUMBER>(Unit1,Unit2,Coeff).

exchangeCoefficient0(Unit1,Unit2,Coeff) :-
    simpleCoefficient(Unit1,Unit2,Coeff).

exchangeCoefficient1(Unit1,Unit2,Coeff) :-
    exchangeCoefficient0(Unit1,Unit2,Coeff).
exchangeCoefficient1(
            multiplication(Unit1From,Unit2From),
            multiplication(Unit1To,Unit2To),
            multiplication(Coefficient1,Coefficient2)) :-
    exchangeCoefficient0(Unit1From,Unit1To,Coefficient1),
    exchangeCoefficient0(Unit2From,Unit2To,Coefficient2).
exchangeCoefficient1(
            division(Unit1From,Unit2From),
            division(Unit1To,Unit2To),
            division(Coefficient1,Coefficient2)) :-
    exchangeCoefficient0(Unit1From,Unit1To,Coefficient1),
    exchangeCoefficient0(Unit2From,Unit2To,Coefficient2).
...

exchangeCoefficient(Unit1,Unit2,Coeff) :-
    exchangeCoefficient<MAX_NUMBER>(Unit1,Unit2,Coeff).
```

These rules allow the parser to generate a ground program by the leveled definition of some predicates, namely `simpleCoefficient` and `exchangeCoefficient`. The predicate `simpleCoefficient` defines either direct or indirect conversion coefficients between atomic units. It uses the predicate `simpleConvertible` which relates atomic units that are simply convertible in the same level. The predicate `exchangeCoefficient` extends conversion from atomic units to compound units.

Leveled predicates definition allows the parser to generate a ground program for specified level of predicates using the lower level of predicates, e.g. to generate grounding for `simpleCoefficient1` from groundings of `simpleCoefficient0` and `simpleConvertible0`.

The highest level of predicates definition is firmly given — the definition ends at `<MAX_NUMBER>` level.

3 Valinor

The conversion algorithm is implemented as the tool VALINOR (Values Normalization), which is verified in the form of two prototypes — one for Prolog and the other for ASP. The tool automatically normalizes salary values stored in the job offer ontology for those salary values which have not yet been normalized.

The Prolog prototype uses SWI-Prolog ([11]) as the logic programming engine. SWI-Prolog is an open–source Prolog system widely used in research and education as well as for commercial applications.

The other prototype uses ASP solver SMODELS ([8]) and its affiliated parser LPARSE ([10]), open–source tools for computing answer sets of logic programs. We choose the solver and its parser because of their language extensions, namely for support of function symbols the tool uses for compound units.

The tool works — independently to LP approach — in the following steps:

1. The tool downloads actual exchange rates from the web site of NBS (Slovak National Bank — `http://www.nbs.sk/`) and prepares a logic program with the conversion coefficients for currency units (predicate `coefficient`).
2. The tool reads those properties of job offers (salaries) from the ontology repository, which ought to be normalized and prepares a logic program with them (predicate `hasProperty`).
3. The programs prepared in the previous steps are joined with the universal conversion and normalization program. The resulting program is sent to the LP engine (either SWI-Prolog or LPARSE with SMODELS).
4. The results of the computation are the expected normalized values, which are written into the ontology repository afterwards.

The tool is scheduled to work in regular time intervals at our project site. In one run it computes normalized salaries for all job offers which were added from its last run.

4 Conclusion

We developed a simple method for values normalization based on logic programming. It is well suited for conversion of any values for which conversions of its atomic units are known. In our point of view, the principles of the method could also be used in other automatic data conversion scenarios.

We created two prototypes of the method using two approaches of LP — Prolog and ASP. We choose the two LP approaches in order to overcome constraints of ontology languages that are based on descriptive logics.

Both prototypes work with our test data and provide normalized properties we expect. They also satisfy the requirements we stated in the first section. However, there are some important distinctions between them.

It is known that ASP approach is more appropriate than Prolog approach in cases when the whole logic database should be taken into account and when the complexity of problem exceeds polynomial complexity ([1]). As it turned out our problem was not global nor complex. ASP language constraints forced us also to implement more complex rules as in the Prolog case. The rules are much less readable and intuitive. Another serious argument against the ASP solution is that it computes the whole conversion table during parsing, which we wanted to avoid.

Thus Prolog seems to be the right tool for the normalization problem and it seems to fulfill all the requirements, but it has also some shortcomings, which were the reason we tried to implement also the ASP prototype. ASP supports strong negation and constraints checking and guards the consistency of the logic theory represented in the program. Prolog is sensitive to ordering of rules and ordering of literals in the body of rules. Inappropriate ordering of rules leads to infinite loops. Thus Prolog is not truly declarative and in the strict point of view not a knowledge representation language ([3]). However, for the reason of values normalization, the Prolog approach is more suitable.

Acknowledgements

This work was partially supported by the Slovak State Program of Research and Development "Establishing of Information Society" under the contract No. 1025/04.

References

1. Alferes, J. J., Pearce, D.: Semantics of Logic Programs and Non-monotonic Reasoning. Presentation at 12th European Summer School in Logic, Language and Information, ESSLLI'00, Birmingham, UK, August 2000.
2. Anger, Ch., Konczak, K., Linke, T., Schaub, T.: A Glimpse of Answer Set Programming. In: Knstliche Intelligenz, 19:1, pp. 12–17, 2005.
3. Baral, C., Gelfond, M., Scherl, R.: Using answer set programming to answer complex queries. Workshop of Advanced Question Answering for Intelligence (AQUAINT) R&D Program, ARDA 04-06, Boston, USA, May 2004.
4. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: An Architecture for Storing and Querying RDF Data and Schema Information. In: Fensel, D., Hendler, J., Lieberman, H., Wahlster, W. (Eds.): Semantics for the WWW, MIT Press, 2001.
5. Kiryakov, A.: OWLIM: Balancing Between Scalable Repository and Light-weight Reasoner. Presented at the Developer's Track of WWW'06, Edinburgh, Scotland, UK, May 2006.
6. Návrat, P., Bieliková, M., Rozinajová, V.: Acquiring, Organizing and Presenting Information and Knowledge from the Web. In: B. Rachev, A. Smrikarov (Eds.), Proc. of CompSysTech'06, Veliko Turnovo, Bulgaria, June 2006.
7. Pázman, R.: Tvorba nezávislého rozhrania pre ontologickú organizačnú pamäť (in Slovak language). In: Laclavík, M., Budinská, I., Hluchý, L. (Eds.): Proceedings of the 1st Workshop on Intelligent and Knowledge oriented Technologies, WIKT'06, Bratislava, Slovakia, pp. 118–121, November 2006.
8. Simons, P.: Extending and Implementing the Stable Model Semantics. PhD. thesis, Helsinki University of Technology, Helsinki, Finland, April 2000.
9. Sterling, L., Shapiro, E.: The Art of Prolog: advanced programming techniques. MIT Press, Cambridge, MA, USA, MIT Press Series In Logic Programming archive, pp. 427, 1986.
10. Syrjänen, T.: Implementation Of Local Grounding For Logic Programs With Stable Model Semantics. Technical Report B 18, Helsinki University of Technology, Helsinki, Finland, October 1998.

11. Wielemaker, J.: An overview of the SWI-Prolog Programming Environment. In: Mesnard, F., Serebenik, A. (Eds.): Proceedings of the 13th International Workshop on Logic Programming Environments, Katholieke Universiteit Leuven, Heverlee, Belgium, pp. 1–16, December 2003.