An Embedded Domain Specific Language for Pattern Mining: a First Attempt

Romuald Thion

Université Claude Bernard Lyon 1
CNRS, LIRIS, UMR 5205, France
romuald.thion@liris.cnrs.fr

Abstract. Logical query languages for pattern mining and their denotational semantics formally define what are interesting patterns in relational databases. The functional programming language Haskell provides an elegant framework to write compilers and interpreters for recursively-defined languages with denotational semantics. In particular, it is especially good at embedding domain specific languages.
This paper reports the experience feedback on a first attempt to build an embedded DSL for a logical pattern mining language. Our objective is to study whether or not an embedded DSL is a good candidate for rapid prototyping of new pattern mining tasks. Interestingly, we obtained a quite clean and concise proof-of-concept program. Using the DSL, an end-user programmer may easily define new patterns an run experiments on sample datasets before further deeper studies. Several questions on the usability and the efficiency of the approach arise from this attempt.

1 Introduction

A Domain Specific Language (DSL) is usually a concise language designed for computational tasks dedicated to a specific application domain. The general purpose functional programming language Haskell [Has99] is notably used for the implementation of DSLs [LM99]. Haskell can be used to write compilers or (interactive) interpreters for DSLs, or alternatively, it can be used to *embed* the DSL, meaning that Haskell acts as an host language: the DSL uses the syntax of Haskell together with code from a small library of functions that defines the constructs and operators [Jon08]. The embedding of a DSL with denotational semantics into Haskell can be divided into three steps. The first one is to turn the inductive definition of the language, usually given as a context-free grammar in Backus–Naur form, into an inductive Haskell datatype. The second step is to define a datatype for the semantic domain in which expressions are evaluated. The key "trick" of this step is to think about Haskell's function as *real mathematical functions*. This approximation is admissible because of the so called *referential transparency property* of Haskell: there are no states, so a function will always return the same value for a given input. The third step is to turn the inductive definition of the "semantic brackets" $[\![\cdot]\!]$ into a function that maps an expression to its interpretation. This definition will typically *pattern matches* the syntactic constructs and build the interpretation inductively.

2 A DSL for the $\mathcal{RL}$ Logical Language

The logical query language called $\mathcal{RL}$ has been defined to express different kinds of patterns to be found in a relation [AFP+11]. Intuitively, $\mathcal{RL}$ expressions capture the semantics of *predicates* in the sense of Mannila and Toivonen [Man97]. One may have selected another language, for instance [CLNP06]. We discuss this point in the Section 3. The syntax of the $\mathcal{RL}$ language is the following: an expression is either an *atomic comparison*, a *logical combination* of expressions or a *quantified expression*. The $\mathcal{RL}$ language includes two different kinds of variables with their respective universal quantifiers: tuple variables ("$TVar$") that are to be instantiated to real tuples from a relation and attribute variables ("$AVar$") to its attributes. A universally quantified formula is true if it is true for every possible choice of tuple or variable respectively. The inductive definition of the $\mathcal{RL}$ language is written in Haskell and rendered using the `lhs2TeX` software:

$$
\begin{aligned}
&\textbf{data}\ \mathcal{RL}_a = (\mathcal{A}_a) \\
&\quad \mid (\wedge)\ (\mathcal{RL}_a)\ (\mathcal{RL}_a) \\
&\quad \mid (\vee)\ (\mathcal{RL}_a)\ (\mathcal{RL}_a) \\
&\quad \mid (\Rightarrow)\ (\mathcal{RL}_a)\ (\mathcal{RL}_a) \\
&\quad \mid \neg\ (\mathcal{RL}_a) \\
&\quad \mid \forall \mathit{AVar}.\ (\mathcal{RL}_a) \\
&\quad \mid \forall \mathit{TVar}.\ (\mathcal{RL}_a)
\end{aligned}
$$

To write a valid $\mathcal{RL}$ expression amounts to define a valid Haskell expression of type $\mathcal{RL}_a$. As $\mathcal{RL}$ is embedded into Haskell, it is possible to use the full power of the host language to build complex $\mathcal{RL}$ expressions. For example, the following definition is for Functional Dependencies (FD) written in $\mathcal{RL}$:

$$
\begin{aligned}
&\mathit{fl}, \mathit{fr}, f_1, f_1' :: \mathcal{RL}_{\mathit{Char}} \\
&\mathit{fl} = \forall \texttt{"A"} \in \texttt{"X"}.\ (((\texttt{"t1"}, \texttt{"A"}) = (\texttt{"t2"}, \texttt{"A"}))) \\
&\mathit{fr} = \forall \texttt{"B"} \in \texttt{"Y"}.\ (((\texttt{"t1"}, \texttt{"B"}) = (\texttt{"t2"}, \texttt{"B"}))) \\
&f_1 = \mathit{fl} \Rightarrow \mathit{fr} \\
&f_1' = (\forall \texttt{"t1"}.\ (\forall \texttt{"t2"}.\ f_1))
\end{aligned}
$$

The definition of FDs corresponds to "f_1'" is quite close to the more usual $\forall t_1.\forall t_2.(\forall A \in X.t_1[A] = t_2[A] \Rightarrow \forall B \in Y.t_1[B] = t_2[B])$. This formula capture the semantics of the FD $X \rightarrow Y$. Clearly, some syntactic sugar and some helpers could be integrated to ease the writing of such expressions. Typically, the expression "$\forall A \in X.\phi$" is already a shorthand for "$\forall A.X(A) \Rightarrow \phi$".

The informal semantics of $\mathcal{RL}$ states that expressions are to be interpreted to true or to false according to an interpretation structured made of a concrete relation, a mapping from "$TVar$"s to concrete tuples from the relation and a mapping from "$AVar$"s to concrete attributes. This structure is captured by following Haskell datatype: $\textbf{type}\ \mathit{RLInter}\ a = (\mathit{RLStructa}, \mathit{TVar} \rightarrow a, \mathit{AVar} \rightarrow a)$.

The type *RLStruct* is actually a direct and naive implementation of a relation into Haskell: according to the untyped and named perspective of the relational model, a tuple is a map from a finite set of attributes to a domain and an instance is a finite set of tuples. This naive choice is enough to provide a minimal yet usable environment to execute the interpreter, however it rises the question on how to efficiently combine the $\mathcal{RL}$ interpreter with a real data source. This point is to be discussed in the Section 3.

The semantic brackets function "$[\![\phi]\!]_{st}$" tells if the $\mathcal{RL}$ formula ϕ is evaluated to true under the given interpretation st. The interpretation of atomic values is not provided for brevity and the logical connectives are straightforward.

$$
\begin{aligned}
&[\![\cdot]\!]_{\cdot} :: (Eq\ a) \Rightarrow (RLInter\ a) \rightarrow (\mathcal{RL}_a) \rightarrow \mathbb{B} \\
&[\![(a)]\!]_{st} && = [\![a]\!]^{\mathcal{A}}_{st} \\
&[\![(x \wedge y)]\!]_{st} && = ([\![x]\!]_{st}) \wedge ([\![y]\!]_{st}) \\
&[\![(x \vee y)]\!]_{st} && = ([\![\varphi]\!]_{st}) \textbf{ where} \\
& && \quad \varphi = \neg\ ((\neg\ x) \wedge (\neg\ y)) \\
&[\![(x \Rightarrow y)]\!]_{st} && = ([\![\varphi]\!]_{st}) \textbf{ where} \\
& && \quad \varphi = (\neg\ x) \vee y \\
&[\![(\neg\ x)]\!]_{st} && = \neg\ ([\![x]\!]_{st}) \\
&[\![(\forall a.\ x)]\!]_{((R,r,\Sigma),(\sigma^1,\sigma^2))} && = foldr\ (\wedge)\ \textbf{tt}\ bs \textbf{ where} \\
& && \quad fs = [\lambda z \rightarrow \textbf{if}\ (z \Leftrightarrow a)\ \textbf{then}\ y\ \textbf{else}\ (\sigma^2\ z) \mid y \leftarrow (\ R)] \\
& && \quad ss = [((R, r, \Sigma), (\sigma^1, \sigma^2)) \mid \sigma^2 \leftarrow fs] \\
& && \quad bs = [[\![x]\!]_s \mid s \leftarrow ss] \\
&[\![(\forall t.\ x)]\!]_{((R,r,\Sigma),(\sigma^1,\sigma^2))} && = foldr\ (\wedge)\ \textbf{tt}\ bs \textbf{ where} \\
& && \quad ...
\end{aligned}
$$

The most interesting case is "$[\![\forall a.x]\!]_{st}$" that captures the essence of the $\mathcal{RL}$ language. First of all, a list "fs" is built: it contains all the possible extensions of σ^2, that is, a member of "fs" is a mapping that assigns the value y to the attribute variable a and $\sigma^2(z)$ to the others. This definition uses a *lambda abstraction* "$\lambda x \rightarrow f$" for "$x \mapsto f(x)$" and a *set builder notation* "$[f\,x \,|x \leftarrow xs]$" that has to be read as "$\{f(x) \mid x \in xs\}$". Secondly, the list of states "ss" is built from the "fs" list: each member of "ss" is a structure based upon "st" with its last component replaced by a new one picked from "fs". Then, the evaluation function on the formula x is run on all states to produce a list of boolean values. Finally, if all these evaluations return true, then the formula "$\forall a.x$" is evaluated to true using the "$foldr$" function. The case for tuple variables is quite similar and is omitted. All told, the definition of the semantic brackets is basically a rewriting of the definitions given in [AFP+11] in a concrete functional programming language.

Assume that one would like to add a new feature to the language, for instance a speculative bounded quantifier "$|t| \geq n.x$" that would tell if the formula x holds true for *at least* n different extensions of σ^1. What the designer has in mind is a first try toward the integration of support into the $\mathcal{RL}$ language. To do so, the programmer has to add the corresponding new constructor to the $\mathcal{RL}_a$ type and the corresponding "$[\![|t| \geq n.x]\!]_{st}$" case using an *ad hoc* function that would replaces the $foldr$ used in the the "$[\![\forall a.x]\!]_{st}$" case.

The objective of the $\mathcal{RL}$ language is to allow developers to rapidly prototype new innovative patterns. The last step toward a full embedding of $\mathcal{RL}$ is to write the query evaluation engine that takes a relation, a formula ϕ and returns the set of assignments of "$AVar$"s such that formula holds true. For the sake of clarity, we restrict ourselves to $\mathcal{RL}$ formulas that uses exactly two attributes variables, say X and Y. The answer to a query ϕ with exactly two variables is the set $\{(\sigma^2(X), \sigma^2(Y)) \mid [\![\phi]\!]_{((R,r,\Sigma),(\sigma,\sigma^2))}$ for all $\sigma\}$. For the $\mathcal{RL}$ expression "$f1'$" capturing FD given earlier, the result of this evaluation is the set of FDs

$X \rightarrow Y$ that hold in the relation. This function has been implemented as well but it is not provided here for the sake of brevity.

3 Discussion

Embedding a recursively defined language with formal denotational semantics into Haskell, or another language with similar features, is made easy for the following reasons: the syntax and the building blocks of Haskell are designed to be close to the mathematical notation and the referential transparency property allows to mathematically reason on programs. However this approach rises (at least) two questions: "*is it possible to do the same with other non-denotational semantics?*" (e.g., operational semantics) and "*how to deal with standard imperative style algorithmic?*". A direct answer to the first question would be "*yes*": Haskell is as expressive as any other language and some frameworks for operational semantics exist. However, the implementation of an operational semantics would come at a price: the link between the formal definition and the concrete code would not be so straightforward and Haskell's features would not be so handy anymore. The second question is subtler. Our proof-of-concept uses naive implementations for both the interpretation function and the query evaluation[1]. Whereas the code sticks quite closely to the formal definitions it is far from being efficient: clever evaluation strategies for query evaluation and conceptual optimizations are not integrated. For instance, some restrictions of the $\mathcal{RL}$ language enjoying good algorithmic properties and amenable to level-wise research strategy have been identified but are not implemented yet. Algorithmic traditionally uses a mix of mathematical operations and imperative operations (while loop, assignments, references, mutable array). Whereas Haskell provides an artillery for the first, it almost completely lacks the second. So a preliminary answer to the second question would be "*no*". Firstly, we argue that this difficulty can be mitigated: at the price of a lack of purity, some languages like OCaml can mix imperative and functional statements. Alternatively, it is possible to use foreign code, for instance an imperative implementation of an algorithm, but mostly on a black-box manner. Secondly, we argue that the efficiency is not a problem at the early stages of the development process, where the objective is more likely to demonstrate feasibility and interest. The core of the proof-of-concept for the $\mathcal{RL}$ language is about 100 lines of code long, this prototype basically "turns the formal definitions into executable code". A domain expert can already use the interpreter to run experiments on small instances to validate his intuition or try some new ideas. As a conclusion, we think that the DSL approach might be interesting to bridge the gap between the formal analysis of a declarative language for pattern mining and its efficient software implementation: once a developer has released a proof-of-concept library for the embedded DSL, the next step is include domain experts in the process to play with the new toy and to check whether if it is worth to go further into deeper studies and more expensive software development.

[1] `http://liris.cnrs.fr/romuald.thion/files/PKDD_LML_13/`

References

AFP+11. M. Agier, C. Froidevaux, J-M. Petit, Y. Renaud, and J. Wijsen. On Armstrong-compliant logical query languages. In *4th International Workshop on Logic in Databases (LID 2011)*, 2011.

CLNP06. T. Calders, L. Lakshmanan, R. Ng, and J. Paredaens. Expressive power of an algebra for data mining. *ACM TODS*, 31:1169–1214, 2006.

Ell04. C. Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 45–56, 2004.

Has99. The Haskell 98 report, 1999. `http://www.haskell.org/onlinereport/`.

HPvD09. F. Hermans, M. Pinzger, and A. van Deursen. Domain-specific languages in practice: A user study on the success factors. In *ACM/IEEE MODELS 2009*, volume 5795 of *LNCS*, pages 423–437, 2009.

Jon08. M. Jones. Experience report: playing the DSL card. *SIGPLAN Notices*, 43(9):87–90, 2008.

KG02. C. Koen and P. Gordon. An embedded language approach to teaching hardware compilation. *SIGPLAN Notices*, 37(12):35–46, 2002.

LM99. D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd conference on DSL (PLAN 1999)*, pages 109–122. ACM, 1999.

Man97. Heikki Mannila. Methods and problems in data mining. In *ICDT*, pages 41–55, 1997.

A Domain Specific Languages

Actually, DSLs are quite common: OpenGL for 3D graphics, PostScript for printed documents, lex and yacc for parser as well as LaTeX for structured documents can be seen as DSLs for their respective domains. The formal definition of a DSL leads to a carefully crafted abstraction of a subset of the application domain, with the following benefits: it is easy to understand and to use by domain expert and it formalizes routines and common practices. Once the subset of the application domain has been identified and formalized, the expert writes his tasks as DSL programs. This leads to smaller programs, because a carefully crafted DSL eliminates most of the boilerplate that one would have to write in a generic programming language before any interesting task. As such DSL programs are smaller, they can be written more quickly and they are easier to maintain. This is particularly true when the DSL is itself a functional language. Last but not least, DSL programs are amenable to formal analysis and proofs. For instance, a program can be transformed into an equivalent but more efficient one using program equivalence relation at optimization time.

The interested reader may read for instance a user study on the success factors on DSLs in practice [HPvD09]. Examples of DSLs embedded into Haskell includes a functional language for graphics processors [Ell04] and a DSL for hardware compilation [KG02]. Many more examples can be found at `http://www.haskell.org/haskellwiki/Research_papers/Domain_specific_languages`.

B Reviews from LML 2013

The paper has been submitted to `http://dtai.cs.kuleuven.be/lml/`. It has been rejected.

———————— REVIEW 1 ————————

PAPER: 4
TITLE: An Embedded Domain Specific Language for Pattern Mining: a First Attempt [Extended Abstract]
AUTHORS: Romuald Thion
OVERALL EVALUATION: -1 (weak reject)
REVIEWER'S CONFIDENCE: 3 (medium)

———— REVIEW ————

The paper explains how a Domain Specific Language (DSL) for the logical query language called RL can be embedded intro the Haskell programming language.

The paper is 5 pages long (and not presented as an extended abstract). Apart from the abstract, there is no mention of any concrete machine learning and data mining problems in the paper which makes it disconnected from the aim of the workshop. The additional space could have been used to make this link to the workshop a lot more explicit.

The author is probably an expert in Haskell and RL but the few pages used to explain the translation from one to the other are technical and not enough detailed and illustrated for a non expert user to assess the elegance of the language and its possible use for data mining/machine learning.

———————— REVIEW 2 ————————

PAPER: 4
TITLE: An Embedded Domain Specific Language for Pattern Mining: a First Attempt [Extended Abstract]
AUTHORS: Romuald Thion
OVERALL EVALUATION: 1 (weak accept)
REVIEWER'S CONFIDENCE: 4 (high)

———— REVIEW ————

The topic of domain specific languages for data mining is a nice one. Assuming the RL language is a pattern mining language, the paper is on topic. I believe that it is of interest, even though the work seems very preliminary.

At present the paper just shows that RL can be turned into a DSL embedded in Haskell. The paper should clearly show the pattern mining functionality of RL. It then needs to clearly demonstrate the utility of the DSL approach. Illustrative examples are needed in both cases.

——————— REVIEW 3 ———————

PAPER: 4
TITLE: An Embedded Domain Specific Language for Pattern Mining: a First Attempt [Extended Abstract]
AUTHORS: Romuald Thion
OVERALL EVALUATION: -1 (weak reject)
REVIEWER'S CONFIDENCE: 3 (medium)

——— REVIEW ———

The author reports on his experience in building a domain specific language for pattern mining using Haskell. Though interesting, the paper looks like a short essay, presenting almost no relationships to existing formalisms, like description logics, and their realizations. In particular, examples in page 3 look like realization of (a fragment?) of the first-order predicate logic. What are advantages of Haskell for doing that?

Detailed comments:

p.1 Abstract an -> and

an host -> a host

so called -> so-called

p.2 Delete "the" in "the Section 3"

This formula captureS

by following -> by the following

The explanation of the interpretation (below the list of formulas) is not clear enough!

line -9: replaces -> replace