User–Defined Rules Made Simple with Functional Programming

Sava Mintchev

Baring Asset Management, 155 Bishopsgate, London EC2M 3XY, UK
sava.mintchev@barings.com

Abstract. To be successful, any new business information system must address the needs of business users, and have a short 'time–to–value'. Depending on the requirements, the appropriate tools and techniques would vary. Sometimes a good way to meet the needs of business users is by providing them with a domain–specific language (DSL) in which they can model their problems or seek solutions.

In this paper, we discusses our experience of an industrial project for the development of a corporate information system. A small DSL has been created using the Haskell functional language. The DSL has given business users the required degree of flexibility and control. The development was completed on time, and has confirmed Haskell's expressive power and the high performance of its compiled code. We also argue that Haskell is relevant to parallel Big Data processing, and to Decision Modelling applications.

Keywords: data analysis, integration, domain–specific language, business rules, decision modelling, functional programming, Haskell.

1 Introduction

Much of the potential value of Big Data is hidden in the insights which can be gleaned from it. To realise this value, IT specialists need to collaborate with domain experts, in order to devise, continuously apply and fine-tune the most appropriate data analysis methods. One of the main challenges of Big Data today is to make such multi-disciplinary collaboration as effective and productive as possible.

In a recent survey [17], business executives were asked why companies may be holding out on using Big Data. The top 3 answers were:

1. Need more education on how Big Data solves business problems (62% of respondents)
2. Need Big Data solutions to better address the needs of business users (53%)
3. Need better time–to–value for Big Data (47%)

In other words, executives want solutions to *specific business problems* and needs; and they want a quick return on investment in Big Data projects.

W. Abramowicz and A. Kokkinaki (Eds.): BIS 2014, LNBIP 176, pp. 229–240, 2014.

The same survey also addresses technology issues. When listing reasons for seeking commercial alternatives to Hadoop 2.0 (the popular open–source Big Data processing framework), most respondents cite "simplified data integration".

Business users are experts in their own field; they are not trained data analysts, computer programmers, statisticians, or experts in machine learning. Many business users would probably point to Excel as their tool of choice; but Excel on its own is not a tool for big data storage or analysis. Many other popular end-user tools and underlying technologies are also inappropriate ([6], [7]). The challenge for IT is not only to provide adequate technology, but also to help bridge the gap between business users and that technology.

In this paper we discuss the experience of addressing such challenges in a recent project for development of a data analysis system at Barings[1]. The business problem is summarised first, followed by an outline of the system design. Specific attention is given in subsequent sections to the approach to providing business user-definable rules for data analysis.

2 The Business Problem

The aim of the project was to deliver a new system for evaluation of the company's products (mutual investment funds) relative to each other, and within the wider universe of funds offered by other providers. The evaluation is based on the characteristics of funds, their historic performance, as well as market / sector trends.

Previously, such analysis had been carried out partially, manually, using external data providers' reporting and BI tools, and Excel (see Fig. 1). There were a number of problems:

- manual, inefficient and error prone processes
- inability to combine data from multiple sources to achieve deeper, multi-dimensional analysis
- lack of scalability - could be applied for a few funds only
- data and calculation maintenance headaches

In order to perform the required analysis fully, over 10^6 rows of data need to be processed. While such a volume is not truly "Big Data" by modern standards, it is sufficiently large to make analysis in Excel impractical.

A new, automated, scalable and flexible solution was needed, and a project was started. A traditional "waterfall" approach was applied to this project. In the initial feasibility stage, requirements were gathered by a business consultant working together with representatives of the relevant business areas: product development, investment management, sales, performance measurement. Then

[1] Baring Asset Management provides investment management services in developed and emerging markets to clients worldwide. The company operates from 11 countries, and has around 100 investment professionals, covering equity, bond and alternative asset classes. It is a subsidiary of MassMutual, a leading diversified financial services organisation.

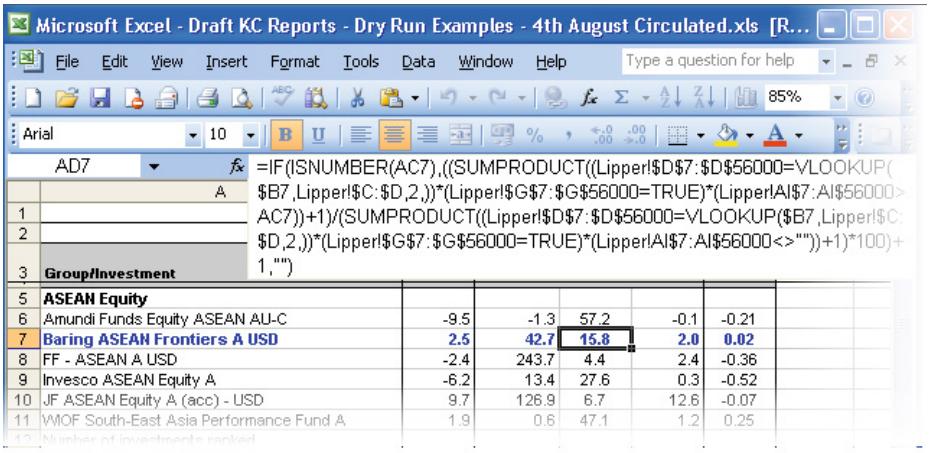

Fig. 1. Old pre-project Excel fragment

a search was conducted in the marketplace for existing solutions (software packages or external services) fitting the requirements. As no appropriate off-the-shelf solution was found, the project was scheduled for in-house design and development.

3 System Design

The new system has had to provide a range of functionality including data integration, calculation of analytics, report production, and user interface. It was decided at the outset that the system would be built of independent modules, all using a dedicated shared database. Each module could utilise a different technology. In this way we have been able to choose the most appropriate technology for each part, and utilise our existing investment in infrastructure, development tools and expertise. The modular system structure is illustrated in Fig 2.

The web-based **User Interface** controls the execution of all other system modules. The **ETL** (Extract–Transform–Load) module processes data from multiple sources: two external data providers (fund returns, flows, sales *etc.*), an internal performance system, and data spreadsheets. It transforms incoming data, resolves dependencies between different data sources, and populates the common data store, housed in a relational database management system (**RDBMS**). Both the User Interface and ETL modules have been created using the webMethods suite from Software AG, in line with our existing integration strategy [11].

The **Analytics Calculation** engine applies the methodology (as agreed by the business) for calculating statistical measures and analytics based on fund data: relative return, risk, momentum, track record, saleability, sales productivity, *etc.* This module has an Object–Relational mapping layer (Hybernate) [12], and is exposed as a webMethods service. The **Score Calculation** module is

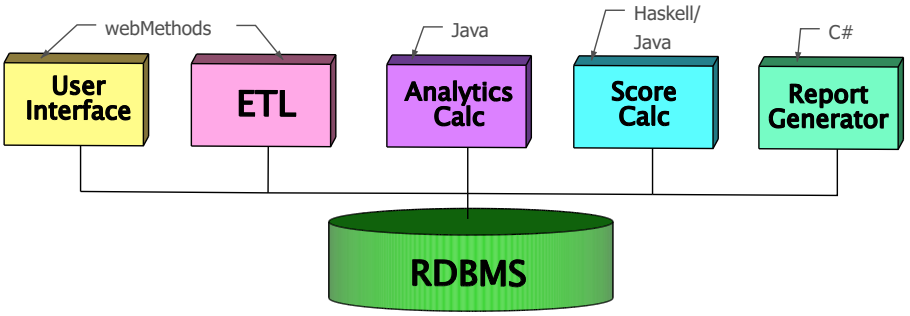

Fig. 2. System structure

discussed in the next section. The **Report Generator** creates a set of reports in strictly predefined Excel formats.

4 User-Definable Rules

An important requirement for the new system has been to enable business users to formulate and maintain *scoring rules*. The rules are used to formally rate each product – investment fund – against other funds within a given category. Each rule shows how to calculate a new value from underlying product characteristics, calculated analytics, and the output of other rules.

The following fragment shows a simplified representation of 3 such rules (`BAND`, `PERF_HL` and `PERF`) defined in terms of underlying product characteristics (`mstarRating`) and other rules defined elsewhere (`MD_HL` – *market demand*, `perfMDscoreSum` – *performance plus market demand score*, `KCGRANK1Y` – *key comp group rank over 1 year*, `QRANK1Y` – *quartile rank*, `REL_RETURN` – *relative return*):

```
BAND
  | PERF_HL  >= 2 && MD_HL >= 1 && perfMDscoreSum >= 3
              = 2
  | PERF_HL  >= 1 && MD_HL >= 0 && perfMDscoreSum >= 2
              = 1
  | otherwise = 0

PERF_HL
  | PERF >= 3.80    = 2
  | PERF >= 2.35    = 1
  | otherwise       = 0

PERF = mstarRating * 0.25 + KCGRANK1Y * 0.25 + QRANK1Y * 0.25 +
       REL_RETURN * 0.25
```

The first two rules represent step functions, while the third is a weighted average. The names (`BAND` *etc*) are cryptic, but come from the domain experts (business users) themselves.

To meet the business requirements, we have had to give end users the means to formulate their own scoring rules, and provide a rule evaluation module. The fragment of scoring rules can be seen as code in a functional programming language – Haskell; and in principle, a Haskell interpreter or compiler can be used for evaluating such rules.

From a different viewpoint, scoring rules can be seen as statements in a simple domain-specific language (DSL). An interpreter for such a language can easily be created in a lazy functional programming language like Haskell.

4.1 Representation of Scoring Rules

The sample rule fragment from Sect. 4 uses Haskell syntax, but this is not how our business analysts and users had envisaged their scoring rules to be formulated. In the Business Requirements document, different types of rule were presented in different tabular formats. So in the example from Sect. 4 , the `BAND` and `PERF_HL` rules were in a table called "conditional" rules, while the `PERF` rule was in another table called "scoring factor" rules.

A proposal for representing scoring rules in Haskell was put forward to Business consultants and users, offering them greater expressive power. It was declined, on the grounds that rules defined in fixed-format tables would result in "*the right balance between flexibility and change control*". In other words, business users would have the freedom to define new rules within the predetermined formats, and any changes to those formats would be made by developers in the IT department.

This leads us to consider how the sample rules from Sect. 4 can be represented in Haskell data types. The declarations can be quite straightforward[2]:

```
data ScoringRule = SR Id [ScoringRuleAlt]

data ScoringRuleAlt
    = ConditionalRule [Condition] Double [Note]
    | ScoringFactor [FactorWeight]
```

where `SR` is a *constructor*, and `Id` is a data type of rule identifiers. A rule has a list of alternatives (`ScoringRuleAlt`), corresponding to *guards* in Haskell syntax. The two alternative constructors (`ConditionalRule` and `ScoringFactor`) represent the two types of rule from Sect. 4.

The numeric values which rules are applied to, as well as the results of rule evaluation can be stored in a type of `ScoreValue`:

```
data ScoreValue = SV Id Double [Note]
```

[2] The code given here is simplified for clarity.

4.2 Scoring Rule Evaluation

We mentioned in Sect.4 two possible approaches to the evaluation of scoring rules:

1. Use a standard Haskell compiler or interpreter (*in which case scoring rules would be written in Haskell*)
2. Write an interpreter (*of a small language which scoring rules would be written in*)

Given that business users would not write their scoring rules in Haskell, the first approach loses its attraction: in order to use a standard compiler or interpreter, we would first have to read and pretty-print the scoring rules in Haskell syntax. We have decided against such an approach because:

a) In the context of our organisation, it is undesirable for a development tool (*e.g.* a Haskell compiler) to be used in a Production environment (*i.e.* beyond the development or test environments);
b) Using a standard compiler, it would be more difficult to provide execution trace and error messages which are clear and meaningful to a business user.

So we have decided to write an interpreter for scoring rules. This is quite easy to do in a lazy functional language:

```
calcScores :: [ScoringRule]->[ScoreValue]->[ScoreValue]
calcScores scoringRules scoreValues =
  let newScoreValues = map (calcScoringRule scoreValues') scoringRules
      scoreValues' = scoreValues ++ newScoreValues
  in newScoreValues
```

The `calcScores` function gets a list of rules (`scoringRules`), and a list of pre-calculated values. It returns a list of new score values, by evaluating every rule (`map (calcScoringRule...)`) in the context of all values (`scoreValues'`) – both pre-calculated values (`scoreValues`), and the results of rule evaluation (`newScoreValues`).

The function which evaluates a single rule has the following type signature:

```
calcScoringRule :: [ScoreValue] -> ScoringRule -> ScoreValue
calcScoringRule scoreValues scoringRule =
    SV (getScoringRuleId scoringRule) val notes
```

where `val` (definition is omitted for brevity) is the floating point number to which `scoringRule` evaluates; `notes` are associated with the applicable alternative of the rule; and `getScoringRuleId` is a de-constructor:

```
getScoringRuleId :: ScoringRule -> Id
getScoringRuleId (SR id ruleAlts) = id
```

4.3 The Joy of Laziness

An interpreter for a simple language can be written in many languages, and it is worthwhile considering what difference Haskell makes. Because scoring rules

can be given in any order, and one rule can refer to the results of other rules, an interpreter needs to evaluate rules in an appropriate sequence. One way of achieving this would be to apply (albeit simple) dependency analysis to the set of scoring rules.

The `calcScores` function from Sect.4.2 does not involve dependency analysis. The complete function declaration is perhaps the simplest way of stating what the rule evaluator is; it can be seen as a formal specification.

Moreover, the lazy evaluation semantics of Haskell ensures that `calcScores` is also an *executable* specification. In the `calcScores` function, using the result (`newScoreValues`) in its declaration only makes sense because of lazy evaluation. To see that, consider the declaration of `calcScoringRule`: it returns a `ScoreValue` with an identifier which comes straight from the rule being evaluated. Therefore the list of new score values (the result of `calcScores`) can safely be unwound, and the identifier of each element can be inspected.

4.4 Polymorphism and Higher–Order Functions

The `calcScores` function from Sect. 4.2 is meant for evaluating a specific set of rules - those described by the `ScoringRule` data type. Using the features of Haskell, it is straightforward to turn that function into a "generic" evaluator (`calcScoresGen`) for other sets of rules:

```
calcScoresGen :: ([a] -> b -> a) -> [b] -> [a] -> [a]
calcScoresGen calcScoringRuleX scoringRules scoreValues =
  let newScoreValues = map (calcScoringRuleX scoreValues') scoringRules
      scoreValues' = scoreValues ++ newScoreValues
  in newScoreValues
```

The only difference compared to `calcScores` is that we have added the function `calcScoringRuleX` of type `([a] -> b -> a)` as an argument to `calcScoresGen`. Here "`a`" can be any type of "score value", and "`b`" is an arbitrary type of "scoring rule".

Of course, in our example the definition of `calcScoresGen` is extremely simple; but one can imagine a more sophisticated function which, by virtue of polymorphism and higher–order functions, is still applicable to different data types, representing different domain-specific languages.

Hopefully the reader can see that the approach of using a DSL interpreted in a functional language has rather bigger potential relevance than the simple example of scoring rules might suggest.

5 Implementation

A simple rule evaluator along the lines of `calcScores` from Sect. 4.2 could be written in Haskell in hours; in our case, it was done incrementally, on and off within a week. The source code totalled under 500 lines.

Another few days were spent on a database access layer, and on exposing the rule evaluator as a service. In total, the development and unit test of the score

calculator took 8 man–days effort over a period of 3 weeks (interleaved with other projects). This represents just under 10% of the development effort for the whole project.

5.1 Program Compilation

The scoring rule evaluator was developed in Haskell, and compiled using the Glasgow Haskell Compiler (GHC) [10]. We have also produced a *naive*, unoptimised translation into Java which preserves the lazy evaluation semantics of the original Haskell code.

Table 1 shows the lines of code in the original programs (.hs), the Java translation (.java), and GHC-generated C code (.hc).

Table 1. Score Calculator – Lines of code

Module	**Description**	**LOC**		
		.hs	**.java**	**.hc**
ScoreDefs	*Data type declarations, access methods*	115	710	6,500
ScoreParse	*Parsing functions*	125	1,100	3,050
ScorePrint	*Pretty-printing*	50	390	1,300
ScoreCalc	*Interpreter for rules*	180	1,500	4,730
	Total	470	3,700	15,580

5.2 Integration with Other Modules

Haskell provides interoperability with other languages via the Foreign Function Interface (FFI) and is implemented in GHC for C/C++. There are additional tools developed by members of the Haskell community to facilitate interfacing to other languages, including Java. There are also packages (HDBC and others) for connecting to database servers. In the simplest case, a process started by an OS command can use standard Haskell IO for data communication via files or pipes. The Network library in Haskell can also be used.

The scoring rule evaluator is a stand–alone module in an application built in other languages. As part of the project, we would not have been able to evaluate different approaches; in this case, the simplest route of standard Haskell IO would suffice. However, given that we have translated the Haskell program into Java code, we could quite easily write additional Java code which links to the translated program. We have used this approach to call functions in the (translated) Haskell program from other Java code in two ways:

- Use database access in Java when calling translated Haskell functions;
- Expose a translated Haskell function as a Java service in webMethods

5.3 Experimental Results

The execution time of the scoring rule evaluator is shown in Table 2 for different size test data sets.

Table 2. Score Calculator – Run time

Number of scores	**GHC code run time** (*sec*)
10^3	0.02
10^4	0.1
10^5	1
10^6	10

Tests were run on a 2.4 GHz Intel Core 2 Duo processor machine under Mac OS. Haskell was compiled with "ghc -O".

6 Discussion

The work was completed successfully. Haskell's strict static type checking helped ensure that there was not much that could go wrong during testing. The maturity of Haskell tools – in particular of the Glasgow Haskell Compiler, GHC – has been apparent and reassuring, and the performance of compiled code has been impressive. The GHC compiler has been continuously developed and improved for over 20 years, and incorporates vast amounts of research in functional programming language implementation.

Since project completion in 2012, the system has been in active use. In our experience of other projects, subsequent enhancement requests are quite common soon after the go-live date, and further development is often needed. This has not been the case for this project. The flexibility which the new system provides to business users (to define their own rules using a simple DSL) has made further development unnecessary to date.

The claimed "simplicity" of the approach is twofold. First, business users and analysts have been able to design their own rules, without having to worry about implementing them (e.g. in Excel, or in a BI tool). Business users therefore envisaged a simple solution which followed their preferred data analysis methodology, and gave them the desired flexibility. Second, from an IT development perspective, the choice of a functional language – Haskell – made it relatively easy to create a DSL for user–defined rules. In Sect 4 we have tried to show how the expressive power of Haskell made the implementation simple.

A drawback of this approach is the reliance on specific programming language skills; there are not nearly as many Haskell programmers as there are Java or SQL programmers. Another drawback comes from the need to integrate a module written in Haskell with the rest of the system which uses different technologies.

6.1 Related Work

A review of modern business intelligence technology is presented in [2]. The system discussed in this paper has elements of a typical BI architecture, like Extract–Transform–Load (ETL) tools, Relational DBMS, Analytics calculation engine, Front-end and reporting applications. In this project there was not a requirement for online analytical operations (filtering, aggregation, drill-down, pivoting) or advanced visualisation, as described in [14]. On the other hand we had very specific reporting requirements, fixed in terms of content and layout. It was a conscious design decision not to employ an OLAP server, thereby foregoing the potential benefits of powerful BI tools. However we do have OLAP capability elsewhere in our IT architecture [11], *e.g.* for financial management information.

The use of DSLs for facilitating the collaboration between between domain experts and IT developers has been well established [5]. The design and implementation of DSLs embedded in Haskell has also been an area of productive research and development [1]. Our implementation is not 'embedded'; it is comparatively simple and straightforward, but has nonetheless helped to meet a real business requirement.

In the realm of business rules and decision support, there has been recent progress towards bridging the gap between Business and IT [15]. The *Decision Model and Notation* (DMN) specification has been submitted to the OMG, and has just been adopted and published in draft. The specification's main goal is to define an industry standard notation for decision management and business rules which is understandable by business users, analysts, and IT developers. The new specification is related to the Business Process Model and Notation (BPMN) OMG standard, in that BPMN decision tasks can be modelled with DMN. The DMN specification concerns a special type of business rules, and as such is more concise and application–focused than the Semantics of Business Vocabulary and Business Rules (SBVR) specification. DMN covers both modelling and execution aspects. We think that it would be appropriate to consider the implementation of DMN's *Friendly Enough Expression Language* (FEEL) – which is free of side effects – as a DSL in a functional language such as Haskell.

How relevant is a pure functional language like Haskell to the challenges of Big Data? It can be quite relevant, due to its support for parallel and concurrent programming [9], and the high performance of compiled code. With respect to concurrent programming, researchers have found that "*applications built with GHC enjoy solid multicore performance and can handle hundreds of thousands of concurrent network connections.*" [3]. But it is the parallel programming capability which makes Haskell particularly relevant to Big Data processing. In [8], the author reverse–engineers Google's MapReduce programming model [4], and creates a formal executable specification in Haskell. The specification is then refined to model parallel execution opportunities. The author also considers Google's domain–specific language Sawzall from a functional specification perspective.

It is said that writing MapReduce routines in languages like Java, Python or Ruby is rather more difficult than writing relational database queries in SQL

[16]. There are different approaches for making this task easier [2]. With its expressive power and support for parallel programming, Haskell could prove to be a good tool for MapReduce programming.

7 Conclusion and Future Work

In this paper we have discussed our experience in addressing some of the challenges associated with Big Data processing: data integration, and involvement of business users. We have applied a modular approach, employing different technologies and languages which are appropriate for the different stages in data processing. We have focused in particular on the use of a small domain-specific language (DSL) with an interpreter written in Haskell, a pure functional programming language.

The functional language development described here, while self-contained and completed successfully, is only a modest beginning for future work. The same approach can be adopted in other cases where there is a requirement for evaluating declarative rules. There are other possible applications of a similar rule engine in the context of the investment management industry, for example:

- Scoring of securities in equity research / quantitative analysis;
- Ensuring compliance with various regulatory rules and client–specific mandate restrictions;
- Calculation of client fees and rebates – the methods of fee calculation can vary from client to client;
- Master Data Management functionality – flexible rules for market data validation and "golden copy" construction

For wider industrial use, stability of the programming language and compilers is extremely important. Haskell is now a mature language with a stable specification and compliant implementations, most notably GHC. It is backed and developed by a wide community of researchers and industry practitioners [13]. Some of its features have been adopted by other languages, thereby becoming more familiar; and it is taught at an increasing number of universities. With its high–performance parallel implementation, Haskell can be a serious contender for effective Big Data processing.

Acknowledgments. The author is grateful to Stuart Holden, Stefan Holes, Stephen Schwartz, Mohanad Al-Bazi and other colleagues who have worked on this project.

References

1. Augustsson, L., Mansell, H., Sittampalam, G.: Paradise: a two-stage DSL embedded in Haskell. In: ACM Sigplan Notices, vol. 43, pp. 225–228. ACM (2008)
2. Chaudhuri, S., Dayal, U., Narasayya, V.: An overview of Business Intelligence technology. Communications of the ACM 54(8), 88–98 (2011)

3. Collins, G., Beardsley, D.: The Snap framework: A web toolkit for Haskell. IEEE Internet Computing 15(1), 84–87 (2011)
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
5. Ghosh, D.: DSL for the Uninitiated. Communications of the ACM 54(7), 44–50 (2011)
6. Jacobs, A.: The pathologies of Big Data. Communications of the ACM 52(8), 36–44 (2009)
7. Labrinidis, A., Jagadish, H.V.: Challenges and opportunities with Big Data. Proceedings of the VLDB Endowment 5(12), 2032–2033 (2012)
8. Lämmel, R.: Google's mapreduce programming modelrevisited. Science of Computer Programming 70(1), 1–30 (2008)
9. Marlow, S.: Parallel and concurrent programming in Haskell. In: Zsók, V., Horváth, Z., Plasmeijer, R. (eds.) CEFP. LNCS, vol. 7241, pp. 339–401. Springer, Heidelberg (2012)
10. Marlow, S., Jones, S.P.: The Glasgow Haskell Compiler. In: Brown, A., Wilson, G. (eds.) The Architecture of Open Source Applications, vol. II (2012), http://www.aosabook.org
11. Mintchev, S.: Open it for business: Transforming information system infrastructure with a commercial BPM suite. In: Abramowicz, W. (ed.) BIS 2011. Lecture Notes in Business Information Processing, vol. 87, pp. 230–241. Springer, Heidelberg (2011)
12. O'Neil, E.J.: Object/Relational Mapping 2008: Hibernate and the Entity Data Model (edm). In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1351–1356. ACM (2008)
13. O'Sullivan, B., Stewart, D.B., Goerzen, J.: Real World Haskell. O'Reilly Media (2009), http://book.realworldhaskell.org
14. Risi, M., Sessa, M., Tucci, M., Tortora, G.: CoDe modeling of graph composition for Data Warehouse report visualization. IEEE Transactions on Knowledge and Data Engineering 26(3), 563–576 (2014)
15. Taylor, J., Fish, A., Vanthienen, J., Vincent, P.: Emerging standards in decision modeling. In: Intelligent BPM Systems: Impact and Opportunity. BPM and Workflow Handbook Series, Future Strategies Inc., Lighthouse Pt (2013)
16. Teplow, D.: The database emperor has no clothes: Hadoops inherent advantages over RDBMS in the Big Data era. Business Intelligence Journal 18, 36–39 (2013), http://tdwi.org
17. uSamp. 2013 Big Data in Business Study. Study fielded by uSamp (United Sample Inc.), commissioned by 1010data (December 2013), http://info.1010data.com/Whitepaper-2013BigDataStudy.html