Linked Lambdas

Vocabulary Integration in the Functional Programming Context

Ralf Lämmel   Thomas Schmorleiz   Andrei Varanovich

University of Koblenz-Landau, Software Languages Team, `http://softlang.wikidot.com/`

Abstract

There exist various knowledge resources for functional programming: textbooks, wikis, course material, collections of program samples, etc. We describe an approach for integrating functional programming resources in terms of their underlying vocabularies. An important requirement is here that the integrated vocabulary is of manageable size and sufficiently validated and structured to be immediately useful, for example, for teaching on programming and documentation of programs. We apply the approach to four Haskell textbooks, the Haskell Wiki, and relevant content from Wikipedia. We have made the underlying data and tools available openly.

Keywords Vocabulary integration. Vocabulary mining. Functional Programming. Text summarization. Linked Data. Haskell. Knowledge integration.

1. Introduction

Suppose you want to read about a specific programming subject, e.g., 'monads'. You can consult textbooks, wikis, blogs, papers, samples, and others. You may use a web-search engine to locate resources on the subject of interest.

Suppose you want to understand subjects in the context of a vocabulary, e.g., monads in a functional programming context. You can consult tables of contents, glossaries, or indices provided by individual resources. The corresponding vocabularies may be hard to use because of size and lack of guiding structure. For instance, the index of a textbook may easily have 1k entries which are simply ordered alphabetically. Now, suppose you event want to understand the mapping between vocabularies of different resources.

We describe an approach for integrating functional programming resources in terms of the underlying vocabularies. An important requirement is here that the integrated vocabulary is of manageable size and sufficiently validated and structured to be immediately useful, for example, for teaching on programming and documentation of programs.

Figure 1 summarizes the context and the *contribution* of this paper. We integrate four Haskell textbooks [11, 14, 17, 20] as well as the *Haskell Wiki*[1] and relevant content from *Wikipedia*[2]. We

[1] `http://www.haskell.org/haskellwiki`

[2] `http://en.wikipedia.org`

[Copyright notice will appear here once 'preprint' option is removed.]

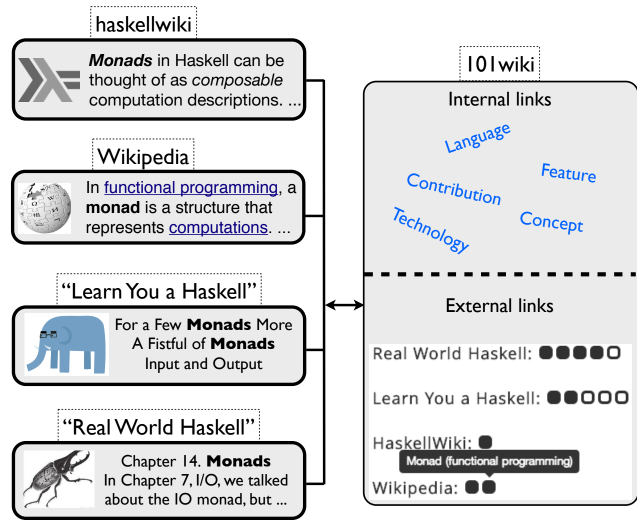

Figure 1. Illustration of knowledge integration for the '*Monad*' concept according to different sources.

use *101wiki*[3] (the semantic wiki of the 101companies Project [8]) for integrating the sources, providing navigation across sources, and demonstrating the use of the integrated vocabulary for the documentation of Haskell programs.

> When consulting the *101wiki* page for 'monad', internal and external links to various resources are rendered. In particular, there are external links to witness integration of resources. The figure shows the moment when the user hovers over a 'resource symbol' related to Wikipedia's monads (with 'Functional programming' added for disambiguation).

Overall, *101wiki* employs *Linked Data*[4] principles for knowledge representation and access.

A key *challenge* of this work is indeed the heterogeneity and complexity of the involved sources, which requires methodological and algorithmic precautions to actually obtain a vocabulary that is useful, say, for teaching and documentation. To this end, we control mining thresholds, we leverage queries (visualizations) on vocabulary data for decision making and quality assurance, and we perform non-automated validation steps along the way.

The underlying data and tools are available openly from the paper's website.[5] By presenting the Haskell case in detail and making everything available openly, we intend to encourage others to integrate further resources on programming and software development.

[3] `http://101companies.org/wiki/`

[4] `http://linkeddata.org/`

[5] `http://softlang.uni-koblenz.de/LinkedLambdas`

Road-map §2 motivates vocabulary mining and knowledge integration for lambdas in the context of the 101companies Project. §3 describes a semi-automated process for mining 'favorable' terms from textbooks. §4 describes a semi-automated, feedback-oriented process to derive 'integrated' terms from 'favorable' terms and then to 'promote' those integrated terms. §5 discusses related work. section 6 concludes the paper.

2. 101 ways to total and cut salaries

The 101companies Project [8] is a software chrestomathy that collects, organizes, and documents implementations of an (HRMS-like) software system for the sake of illustrating, studying and understanding software technologies, languages, and concepts. By now, there are some 150 implementations (also referred to as 'contributions'); they exercise many dozens of languages and technologies; they implement varying features. For instance, basic features of the 101companies system are concerned with totaling and cutting salaries of employees in a company.

- Contribution:haskellStarter: Contribution with small language footprint.
- Contribution:haskellCabal: Packaging, unit testing, and modularization.
- Contribution:haskellList: List processing with map and friends.
- Contribution:haskellWhere: Local scope with where clauses.
- Contribution:haskellLambda: Anonymous functions.
- Contribution:haskellComposition: Use of recursive data types.
- Contribution:haskellVariation: Use of multiple constructors per type.
- Contribution:monoidal: Use of monoids for implementing queries.
- Contribution:nonmonadic: Demonstration of non-monadic style.
- Contribution:writerMonad: Demonstration of monadic style.
- Contribution:haskellParsec: Parsing based on parser combinator library.
- Contribution:haskellSyb: Generic programming.

Figure 2. The theme '*Haskell introduction*'.

It happens that 101companies also aggregates (currently) 28 Haskell-based contributions, which are, in turn, organized in several *themes*, i.e., collections of contributions. For instance, there is the theme '*Haskell introduction*', which is used, for example, in an introductory course on functional programming; see Figure 2 for a summary of the contributions in this theme.

It should be clear that such a large scale effort on organizing knowledge on programming languages and software engineering requires organization principles for the involved vocabulary, documentation, and code. Over the two years period of working on the project, we have observed that contributors (undergraduate students and experienced developers, for example) consistently face the challenge of making good judgement on devising essential vocabulary for documenting and tagging contributions.

In this paper, we show how to systematically perform vocabulary mining and integration in such a context. We use 101wiki for the actual realization of the integrated vocabulary. In principle. our approach is neither specific to the Haskell language nor to the 101companies Project.

3. Vocabulary mining

We match suitable index entries from available textbooks nontrivially with the book's content so that 'favorable' terms are identified with the help of text summarization techniques [1] including inverse document frequency [12].

3.1 Methodology

- We require access to the content, the chapter structure, and the index of a textbook, thereby enabling the application of mining techniques to index and content.

Book	Original Index Entries	Sub-entries	Final Entries
CRAFT	1088	534	696
PIH	1468	210	191
RWH	1244	346	1049
LYAH	1241	691	170

Table 1. Index metrics

- We process a textbook's index to obtain entries ('candidate terms') suitable for matching. In particular, we perform data cleaning, stemming, and ranking.
- We match candidate terms with the textbook's content, which is prepared also by data cleaning and stemming. We track origins of matches for integration purposes as exercised in Figure 1.
- We select candidate terms as 'favorable' terms based on frequency of matching, distribution over chapters, and inverse document frequency, subject to the identification of thresholds.

3.2 Data access

We apply vocabulary integration to these popular textbooks on functional programming in Haskell:

CRAFT [20] "Haskell: The Craft of Functional Programming"

PIH [11] "Programming in Haskell"

RWH [17] "Real World Haskell"

LYAH [14] "Learn You a Haskell"

Books PIH and CRAFT were available to us through their LaTeX sources. Chapter structure and index entries are easily discoverable on the grounds of LaTeX markup. Books LYAH and RWH are accessible online and available as ebooks in HTML. Chapter structure is discoverable from the page sets for the books. Index entries are extracted from the indices for the ebooks.

3.3 Index processing

We start with a raw list of index entries which we process as follows. Duplicates are removed. Special characters and symbols are removed (e.g., $\Rightarrow$ and $\neq$). Subentries (such as 'associativity: using with monads') are removed.

We apply the Natural Language Toolkit (NLTK [5]) for stemming. To this end, we split each candidate term into words (using NTLK functionality), stem each word, and then join back together the resulting words. As the result, we obtain a normalized set of terms that is free of trivial redundancy.

In fact, prior to standard stemming, we also perform specific, pattern-based normalizations that pay attention to the special nature of the index entries at hand. For brevity, we skip such custom stemming here.

Finally, we remove candidate terms by applying WordCount[6] as a stop list, i.e., a list of "common English words". To this end, we review candidate terms (from all books) sorted by their ranks. Rank 90 is defined as the barrier for inclusion as the majority of terms with higher ranks turned out to be clearly specific to programming.

Table 1 summarizes metrics for index entries.

3.4 Content processing

We do not include vocabulary extraction from the source code because it would add some additional challenges. Thus, we identify markup (e.g., LaTeX environments or the <pre> tag of HTML) used for code samples and remove all code prior to matching.

We remove all LaTeX and HTML markup prior to matching. We also exclude *introduction*-, *preface*-, and *conclusion*-like chapters. We also apply case conversion, pattern-based normalization,

[6] `http://www.wordcount.org/main.php`

Book	Chapter profiling	Title terms	Popular terms
CRAFT	52/55	2	12
PIH	25/31	3	6
RWH	65/72	4	8
LYAH	34/38	0	4

Table 2. Favorable term metrics

stemming, and exclusion based on the stop-list in the same way as it was explained for index processing.

Ultimately, the index terms are matched with the prepared content of the book. Here, we note that the index entries from all books are united in one list of terms so that terms of any book are also searched in all other books. Further, we note that this process is obviously biased towards shorter terms (in terms of numbers of words). When mapping matched terms, we reconstruct longer terms in some cases; see §4.

3.5 Chapter profiling

We are ready to favor terms of the raw vocabulary for the given book. To this end, we adopt simple techniques of text summarization.

We select candidate terms as favorable terms, if they occur frequently enough, but they are not scattered over too many chapters of the book. Specifically, we exclude all terms which are exercised by more than 25% of chapters while applying a threshold of 3 occurrences per chapter for counting a chapter as exercising a term. Among the remaining terms, we favor the top-5 most frequent terms for every chapter. We refer to the resulting set of terms with associations to the chapters as the *chapter profile* of a book. This approach comes with the intended bias to select relatively popular terms which are not used, though, universally throughout the book.

Figure 3 shows the chapter profile for one of the Haskell textbooks. There is one row per term and one column per chapter. A bullet in a cell indicates a favorable term (per row) and associates a chapter with it (per column). To provide additional insight, the size of the bullet represents matching frequency over favorable terms: ●: the 5 most frequent favorable terms; •: $\geq$ median; ·: $<$ median.

3.6 Validation

All terms from the chapter profile are manually validated so that 'uninteresting' terms are excluded. The idea is here to focus on terms that concern functional programming, Haskell programming, or generally programming. There are these reasons for some other terms to show up:

- An illustrative term due to the specific examples in the book; this could be a concept or the name of a function or a type.
- A 'general English' term rather than a term related to programming, which may happen because 'common programming English' is not necessarily identical with 'common English' (which we have already suppressed on the grounds of WordCount; see above).

Figure 4 shows the excluded terms for CRAFT. For instance, the term 'model' contributes to the profile of the CRAFT chapter 'Playing the game IO in Haskell'. Inspection does not support any repeated programming-related use of the term. Instead, the term is used in the general English sense of "We can model a tournament by this type definition" [20]. Thus, the term is excluded.

Table 2 summarizes metrics for found favorable terms. We discuss title terms and popular terms below, but the column for *chapter profiling* shows how many terms make it beyond validation.

All such validation was performed redundantly in the sense that two out four books were validated independently by two researchers, when using validation results for the other two books for up-front calibration. There were very few cases of disagreement

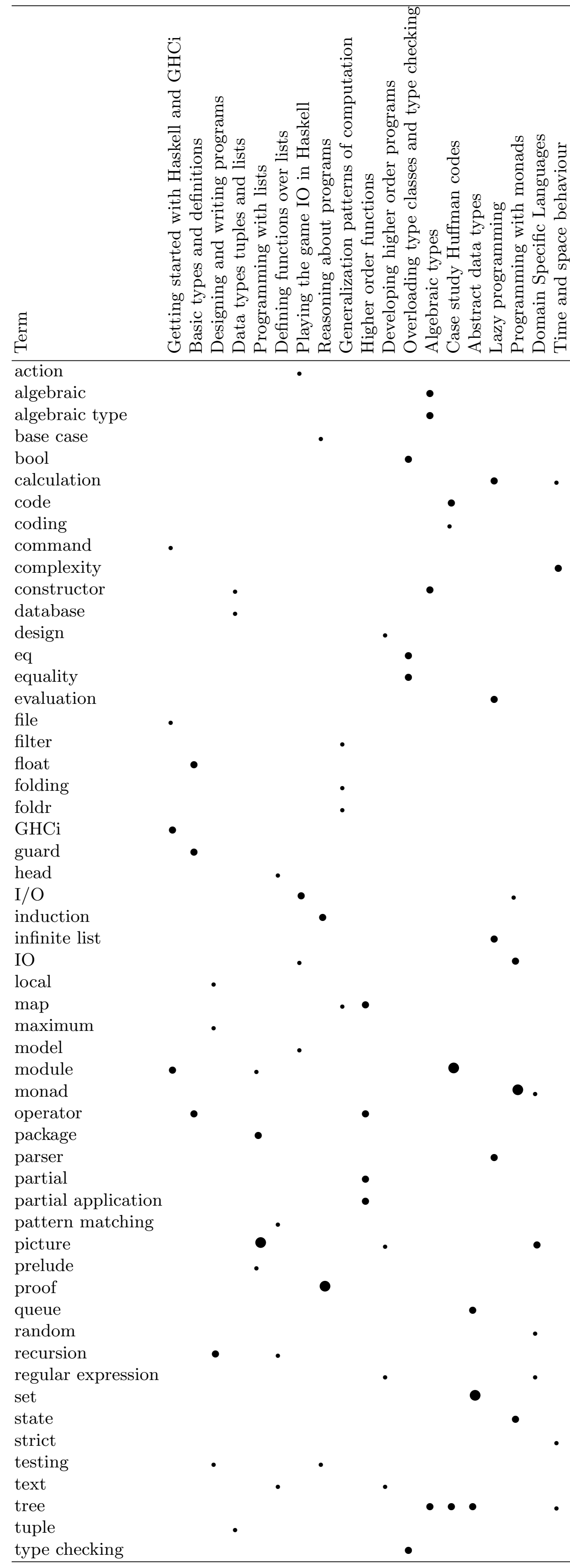

Figure 3. Chapter profiling for CRAFT

- maximum: *illustrative term*
- model: *English term*
- picture: *illustrative term*

Figure 4. Terms excluded from the chapter profile of CRAFT

having to do with different views on essential or interesting concepts of (functional) programming.

3.7 Title terms

As part of the validation process, we also evaluate whether the terms per chapter sufficiently capture the concepts conveyed by the title. Thus, we add a so-called *title term* for chapters, when this is not the case. This was only necessary for very few chapters; see Table 2. For instance, CRAFT's chapter 'Reasoning about programs' has 'induction', 'proof', and 'testing' per profile, but another central term, 'Equational reasoning', is missing, which is hence added.

3.8 Popular terms

Due to the nature of the chapter profiling algorithm, some general (functional) programming terms, such as *function*, are not selected. They are used throughout the books and they deserve to be added to the raw vocabulary.

We pick popular terms per book as follows. We order matched terms by frequency, while we exclude terms that are readily in the chapter profile of the book. Then, we review all terms that have >10 % frequency of the topmost term's frequency. Note that we have already excluded popular terms of common English earlier on in the process. Most terms were favorable and uninteresting terms were excluded as before. It is not surprising that the different books agree on the popular terms to a good extent. For instance, all four books have 'function' and 'list' among the top-3 popular terms.

4. Vocabulary integration

We describe a semi-automated, feedback-oriented process to derive 'integrated' terms from 'favorable' terms and then to 'promote' those integrated terms in a given context such as the chrestomathy of 101 in our case.

4.1 Methodology

- We map 'favorable' terms to 'integrated' terms, thereby eliminating effects of data mining and source-specific impact.
- We review the contributions to the vocabulary for the different sources (i.e., textbooks).
- We 'promote' integrated terms by setting them up as a linked data resources (e.g., on the *101wiki*).
- We organize sub-vocabularies to better measure and understand the vocabulary at hand.
- We exercise the vocabulary and monitor its usage within some context (such as the chrestomathy of 101).
- We devise means of discoverability for the achieved knowledge integration such as links for navigation between sources, as it was illustrated in §1.

4.2 Term mapping

We review favorable terms in the context of their contributing chapters to propose suitable integrated terms. In some cases, the relevant chapters have to be searched to disambiguate the term. In practice, such manual work on mapping acquisition is intertwined with the effort on validating favorable terms; see §3.6.

- action → *Action*
- algebraic type → *Algebraic data type*
- base case → *Base case*
- bool → *Boolean*
- calculation → *Calculation*
- class → *Type class*
- code → *Code*
- coding → *Programming*
- ...

Figure 5. Mapping for the first few terms of CRAFT

Terms in CRAFT only: *Local scope*, *Value*, *Complexity*, *Proof*, *Calculation*, *Equational reasoning*, *Head*, *Equality*, *Programming*, *Queue*, *Argument*, *Result*, *Base case*, *Partial application*, *Program*, *Tuple*, *Set*, *Program design*, *Type checking*, *Higher-order function*, *Name*, *Algebraic data type*, *Infinite list*, *Float*

Terms in PIH only: *Haskell script*, *too generic term*, *Equation*, *Function application*, *Parser combinator*, *Identity element*, *Declaration*, *Function definition*, *Product function*, *Lambda abstraction*

Terms in RWH only: *Foreign function interface*, *Predicate*, *Operator precedence*, *Polymorphism*, *Thread*, *Performance*, *MVar*, *Profiling*, *TCP*, *Directory*, *Property*, *Loop*, *Technology:Parsec*, *Parsing*, *Monad transformer*, *Pointer*, *Technology:HPC*, *Type system*, *User interface*, *Language:XML*, *Core*, *Technology:Glade*, *Exception*, *Error*, *Process*, *Type signature*, *Type definition*, *Program optimization*, *Data type*, *Technology:GHC*, *Pure function*, *Association list*, *Query*, *Output*, *UDP*, *Table*

Terms in LYAH only: *Fmap function*, *Accumulator*, *type-class instance*, *Functor*, *Data structure*, *Monadic value*, *Import*, *Factorial*, *Zipper*, *Condition*, *Expression*, *Sum function*, *Applicative functor*

Terms in more than one book: *Monoid*, *Character*, *Type-class instance*, *Bit*, *List comprehension*, *Testing*, *Fold function*, *Operator*, *Lazy evaluation*, *Recursion*, *I/O system*, *Number*, *State*, *Input*, *Haskell package*, *Type*, *String*, *Type class*, *Random number*, *Tree*, *Command*, *Parser*, *Filter function*, *Code*, *Data constructor*, *Pattern*, *Integer*, *Database*, *Catamorphism*, *Evaluation strategy*, *Action*, *Technology:GHCi*, *Text*, *Tail*, *Regular expression*, *Map function*, *Language:Haskell*, *Induction*, *Function*, *Pattern matching*, *Prelude*, *Stack*, *Eager evaluation*, *List*, *Maybe type*, *Monad*, *Module*, *Guard*, *Boolean*, *File*

Figure 6. Comparison of the different Haskell textbooks

The following example concerns a term of CRAFT's chapter profile; see Figure 3. The term 'class' contributes to the profile of the chapter 'Overloading type classes and type checking'. Thus, 'class' is mapped to 'type class'. The term 'class' would be overly ambiguous in a broader context of programming, even though it may be sufficiently clear in the narrow context of functional programming with Haskell.

Figure 5 shows the first few mapping entries from favorable to integrated terms for CRAFT. (For what it matters, we use a CSV format to maintain these mappings for each book.)

While it may be relatively simple to agree on whether or not a (candidate) favorable term should be included, as we illustrated in §3.6, it may be harder to agree on the 'integrated' term, as there may be several reasonable candidates and points of views. Thus, domain-specific portals are consulted for resolution. We consult Haskell Wiki as well as Wikipedia, which also organizes functional programming knowledge.

4.3 Contributions per source

Mapping also enables a sensible comparison of the vocabularies obtained for the different sources. We assume that a source (a text-

book) is individually characterized, relatively to all other sources, by the terms that it uniquely contributes.

Figure 6 lists the unique terms contributed by each textbook; at the bottom, all remaining ('non-unique') terms are listed. We make a few observations:

- CRAFT contributes terms related profoundly to formal or mathematical areas of functional programming such as 'Proof' and 'Calculation'.
- PIH contributes the fewest terms and much of them are concerned with basic functional programming concepts such as 'Function application' and 'Function definition'.
- RWH contributes the most terms, overall, and it mentions several technologies, whereas the other books do not.
- LYAH contributes terms related to advanced functional programming concepts, such as zippers and applicative functors, which do not make it into the chapter profile of the other books.

Clearly, the books complement each other in terms of their contributions, which supports any effort towards integration.

4.4 Term promotion

We register each term, obtained so far on *101wiki*. A very short description, also referred to as headline, is assigned. Integrated textbooks with online availability are immediately linked to the term—including specific links to relevant chapters and paragraphs. We also link to resources on Wikipedia, Haskell Wiki, and others.

We distinguish three major categories of terms:

- *Software languages* (prefix 'Language')
- *Software technologies* (prefix 'Technology')
- *Software concepts* (empty prefix)

(Most terms correspond indeed to (software) concepts; see again Figure 6.) Consider, for example, the *Zipper* concept contributed by the LYAH textbook. We assign the following metadata to it on *101wiki*:

- *Headline*: A data structure for location-based manipulation of a data structure.
- A link to the *Wikipedia* counterpart: *Zipper (data structure)*.
- A link to the *Haskell Wiki* counterpart: *Zipper*.
- A *DOI* link to the '*The Zipper*' [10]: *10.1017/S0956796897002864*.

All three linked resources are marked as primary resources, using a designated semantic property, which means that we capture the fact that those resources are not simply related to the concept at hand, but they 'define' or 'represent' it.

4.5 Sub-vocabularies

We are interested in better understanding the nature of the concepts at hand. To this end, we classify concepts (non-disjointly) according to several (sub-) vocabularies of which we list the more important ones here:

- *Haskell*: Concepts that are effectively Haskell-specific, e.g., *TMVar* and *Haskell package*.
- *Functional programming*: Concepts broadly associated with functional programming, e.g., *Map function* or *Infinite lists*.
- *Programming*: Concepts associated with programming in general, e.g., *Process* and *Error*.
- *Data*: Concepts focused on data structures, data types, data management, et al., e.g., *Queue* and *Char*.
- *Programming theory*: Concepts associated with mathematical or formal treatment of programs, e.g., *Induction*.

The introduction of these vocabularies and their assignment to specific terms is (currently) a manual process, which is informed by the review of all available sources including Wikipedia and Haskell Wiki. Concepts may be inserted into multiple vocabularies.

Name	Headline
Haskell package	A distribution unit for Language:Haskell
Haskell script	A file with Haskell code
MVar	A thread synchronization variable in Language:Haskell
Maybe type	A polymorphic type for handling optional values and errors
Prelude	The standard library of Language:Haskell
TMVar	A transactional MVar of Language:Haskell's STM monad
Type class	An abstraction mechanism for ad-hoc polymorphism
Type-class instance	Type-specific function definitions according to a type class

Figure 7. The obtained Haskell vocabulary

As far as the four textbooks are concerned, the most popular vocabularies are (in decreasing order of popularity) *Programming*, *Data*, and *Functional programming*. The remaining vocabularies are less frequented. For instance, the *Haskell* vocabulary contains only a few concepts, listed in Figure 7, which essentially means that the books operate at a higher level of abstraction, as opposed to any sort of very Haskell-specific level.

4.6 Term monitoring

Clearly, we were motivated to carry out vocabulary mining and integration because we simply wanted to use somewhat objective means to determine established terms for use in the chrestomathy of 101, specifically on *101wiki*. Accordingly, the discovered terms are increasingly used. We consider it an important methodological aspect to keep on monitoring vocabulary usage.

Figure 8 summarizes vocabulary usage for the textbooks at hand and for one specific theme of Haskell-based contributions only. The terms are listed vertically in the order of frequency of usage by the contributions. The contributions are listed horizontally in the order of number of terms referenced. The big bullets indicate proper references indeed, whereas the small bullets report on indirect references. For instance, contributions *nonmonadic*, *writerMonad*, and *haskellParser* all properly reference the concept *Monad*, albeit for different reasons.

The other Haskell themes cover many additional terms. In fact, this sort of monitoring has been used in the past to improve the documentation of existing contributions and to devise additional contributions to cover important concepts.

5. Related work

When compared to rich related work on vocabulary mining [19], we aim at the informed (hence, semi-automated) consultation of multiple technical, readily indexed sources for the sake of deriving a consolidated and manageable vocabulary with confirmed links to key sources such as Wikipedia.

Vocabulary integration (mapping) is established in the context of ontology matching [7, 16], while vocabularies of substantial size may need to be matched largely automatically, whereas limiting size and non-automated validation are important in our context.

In the context of programming or software reverse engineering, vocabularies are often mined from source code [9, 13] or perhaps other programmer-provided artifacts. Our work specifically uses textbooks for data mining. We do not use the source code in the books—even though this may be a relevant future work topic.

In the context, of lexicon/vocabulary mining from source code [3, 4, 6], the notion of lexicon comparison has also been studied (e.g., comparison of terms extracted from comments versus

Contribution/Term	haskellStarter (9/12)	haskellParsec (7/6)	haskellList (6/17)	haskellVariation (5/12)	monoidal (5/10)	haskellComposition (4/10)	haskellLambda (4/10)	haskellCabal (3/13)	haskellWhere (3/8)	writerMonad (3/6)	nonmonadic (3/4)	haskellSyb (2/6)
Language:Haskell (12/0)	●	●	●	●	●	●	●	●	●	●	●	●
Technology:GHC (11/1)	•	●	●	●	●	●	●	●	●	●	●	●
Algebraic data type (4/8)	●	●	•	●	•	●	•	•	•	•	•	•
Monad (3/0)		●								●	●	
Data constructor (2/5)	•	•		●	•	●					•	•
Fold function (2/5)			●	•	●	•	•	•	•			
Higher-order function (2/5)			●	•	•	•	●	•	•			
Map function (2/5)			●	•	●	•	•	•	•			
Pattern matching (2/4)	●	•	•	●		•		•				
Technology:GHCi (1/11)	●	•	•	•	•	•	•	•	•	•	•	•
Sum function (1/5)			●	•		•	•	•	•			
Module (1/4)	•		•				•	●	•			
Tuple (1/4)	●		•	•		•		•				
Function definition (1/2)	●		•					•				
Local scope (1/2)			•				•		●			
Parsing (1/2)		●								•		•
Pure function (1/2)	●		•					•				
Recursion (1/2)	●		•					•				
Type definition (1/2)	●		•					•				
Lambda abstraction (1/1)			•				●					
Monoid (1/1)					●					•		
Parser combinator (1/0)		●										

• • •

Figure 8. Term usage in the *Haskell introduction* theme.

those extracted from program identifiers). In contrast, we are interested in obtaining a consolidated vocabulary, which then provides a good foundation in the documentation of programs.

Our work enters the territory of taxonomy development, in particular, because of the distinction of (sub-) vocabularies and non-trivial criteria for including and excluding relevant terms. In the context of taxonomy development, the importance of reviewing domain literature and the reuse of existing large-scale categories (taxonomies) such as Wikipedia is established [15, 18].

6. Conclusion

We have described a form of vocabulary integration, which helps with the consolidation of technical vocabulary on the grounds of textbooks and wikis for use in teaching programming and the documentation of programs. The semi-automatic characteristics of the method imply that vocabulary 'integrators' and vocabulary users remain closely familiar with the vocabulary along its continuous management; these parties are further supported by queries (visualizations) on vocabulary data regarding vocabulary mining, integration, and usage.

101wiki (in its current beta version) truly integrates all sources as we illustrated in Figure 1 in §1. Such integration is directly discoverable by the *101wiki* user via the external link area on each wiki page. This level of linking support combined with some other wiki features suggest that 101wiki can be viewed as a simple knowledge integration environment [2].

In future work, we will report on the use and the management of the integrated functional programming vocabulary in a BSc level introductory course on functional programming. We also plan to further advance our developed infrastructure for vocabulary integration, which is already openly available, so that others can leverage it with ease.

Acknowledgment We are grateful to Graham Hutton and Simon Thompson for sharing the sources of the Haskell books [11, 20] with us for the purpose of this research. We are also very grateful for several people who helped on the technical part of this work—specifically, Kevin Klein and Martin Leinberger.

References

[1] C. C. Aggarwal and C. Zhai, editors. *Mining Text Data*. Springer, 2012.

[2] P. Bell, E. A. Davis, and M. C. Linn. The knowledge integration environment: theory and design. In *The first international conference on Computer support for collaborative learning*, CSCL '95, pages 14–21, 1995.

[3] L. R. Biggers and N. A. Kraft. Quantifying the similiarities between source code lexicons. In *Proceedings of the 49th Annual Southeast Regional Conference, 2011*, pages 80–85. ACM, 2011.

[4] L. R. Biggers, B. P. Eddy, N. A. Kraft, and L. H. Etzkorn. Toward a metrics suite for source code lexicons. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011*, pages 492–495. IEEE, 2011.

[5] S. Bird, E. Loper, and E. Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[6] B. P. Eddy and N. A. Kraft. Toward an understanding of the relationship between the identifier and comment lexicons. In *Proceedings of the 49th Annual Southeast Regional Conference, 2011*, pages 342–343. ACM, 2011.

[7] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, 2007.

[8] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich. *101companies*: a community project on software technologies and software languages. In *Proceedings of TOOLS 2012*, volume 7304 of *LNCS*, pages 59–74. Springer, 2012.

[9] S. Haiduc and A. Marcus. On the use of domain terms in source code. In *ICPC*, pages 113–122, 2008.

[10] G. Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[11] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007. http://www.cs.nott.ac.uk/ gmh/book.html.

[12] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:1121, 1972.

[13] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information & Software Technology*, 49(3):230–243, 2007.

[14] M. Lipovaca. *Learn You a Haskell for Great Good!* no starch press, 2011. http://learnyouahaskell.com/.

[15] R. C. Nickerson, J. Muntermann, and U. Varshney. Taxonomy development in information systems: A literature survey and problem statement. In *AMCIS*, page 125, 2010.

[16] B. Omelayenko. Integrating Vocabularies: Discovering and Representing Vocabulary Maps. In *International Semantic Web Conference*, volume 2342 of *LNCS*, pages 206–220. Springer, 2002.

[17] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2008. `http://book.realworldhaskell.org/`.

[18] S. P. Ponzetto and M. Strube. Taxonomy induction based on a collaboratively built knowledge repository. *Artificial Intelligence*, 175(9-10): 1737–1756, 2011. ISSN 0004-3702.

[19] M. Sperettta and S. Gauch. Using Text Mining to Enrich the Vocabulary of Domain Ontologies. In *Proceedings of the 2008 IEEE / WIC / ACM International Conference on Web Intelligence, WI 2008*, pages 549–552. IEEE, 2008.

[20] S. Thompson. *Haskell: The Craft of Functional Programming (3rd edition)*. Addison-Wesley, 2011. `http://www.haskellcraft.com/craft3e/Home.html`.