Continuous Knowledge Integration for Community Resources

Unpublished working document as of November 18, 2013. Do not cite.

Ralf Lämmel and Thomas Schmorleiz and Andrei Varanovich

University of Koblenz-Landau, Software Languages Team

Abstract. Community resources such as wikis, forums, and open online textbooks are important knowledge resources, each one with its specific vocabulary, in fact, its specific knowledge model. This document describes general principles for the integration of software knowledge with a case study on functional programming textbooks such that the wiki of the software chrestomathy 101 hosts the integrated knowledge eventually. Knowledge integration is meant to be continuous in that, for example, more and more resources may be integrated over time and monitoring is applied so that the effective use of the integrated vocabulary is measured. A deliberate limitation of our approach is its assumption of vocabularies of limited size—suitable for manual validation, thereby leading to knowledge that is readily useful, for example, in teaching. We have made the underlying framework (i.e., 101integrate) as well as all data related to the case study publicly available.

1 Introduction

Software knowledge is available from many resources. We are specifically interested in (open online) community resources; think of Wikipedia, more domain-specific wikis, possibly open and online textbooks, forums like StackOverflow, the Apple Knowledge Base, or the support forums by Microsoft.

Each resource uses its specific vocabulary, in fact, its specific knowledge model. This hampers effective use of such distributed, complementary resources. For instance, consider the situation of a student who learns Haskell by consulting a textbook, HaskellWiki, and Wikipedia. Those three resources use different terms and different means of organization; these resources lack effective integration.

We address the kind of integration scenario as just described. To this end, we propose the notion of *continuous knowledge integration* (CKI). It contains elements of knowledge extraction ('mining') and maintenance ('monitoring'). We describe the principles of CKI for the integration of resources with software knowledge. Further, we describe a case study on functional programming textbooks such that the wiki of the software chrestomathy 101companies (or just

'101') [4][1] gets to host the integrated knowledge eventually. This work has resulted in the framework 101integrate, which we made publicly available including all data of the case study.[2]

Knowledge integration is meant to be continuous in that, for example, more and more resources may be integrated over time and monitoring is applied so that the effective use of the integrated vocabulary is measured. A deliberate limitation of our approach is its assumption of vocabularies of limited size—suitable for manual validation, thereby leading to knowledge that is readily useful, for example, for teaching. Here we note that large-scale integration of vocabularies (taxonomies or ontologies) is useful for indexing and searching and linking, but it simply generates too much data, if the goal is to maintain a sizable knowledge base with expected human-authored and -validated elements per term. Knowledge consumers are also saved from being overwhelmed in this way.

Road-map §2 motivates CKI in the context of 101. §3 lists general principles of CKI. §4–§7 applies CKI in a case study mainly concerned with integrating four Haskell textbooks into the knowledge base of 101. §8 concludes the document.

2 Motivation: 101kb

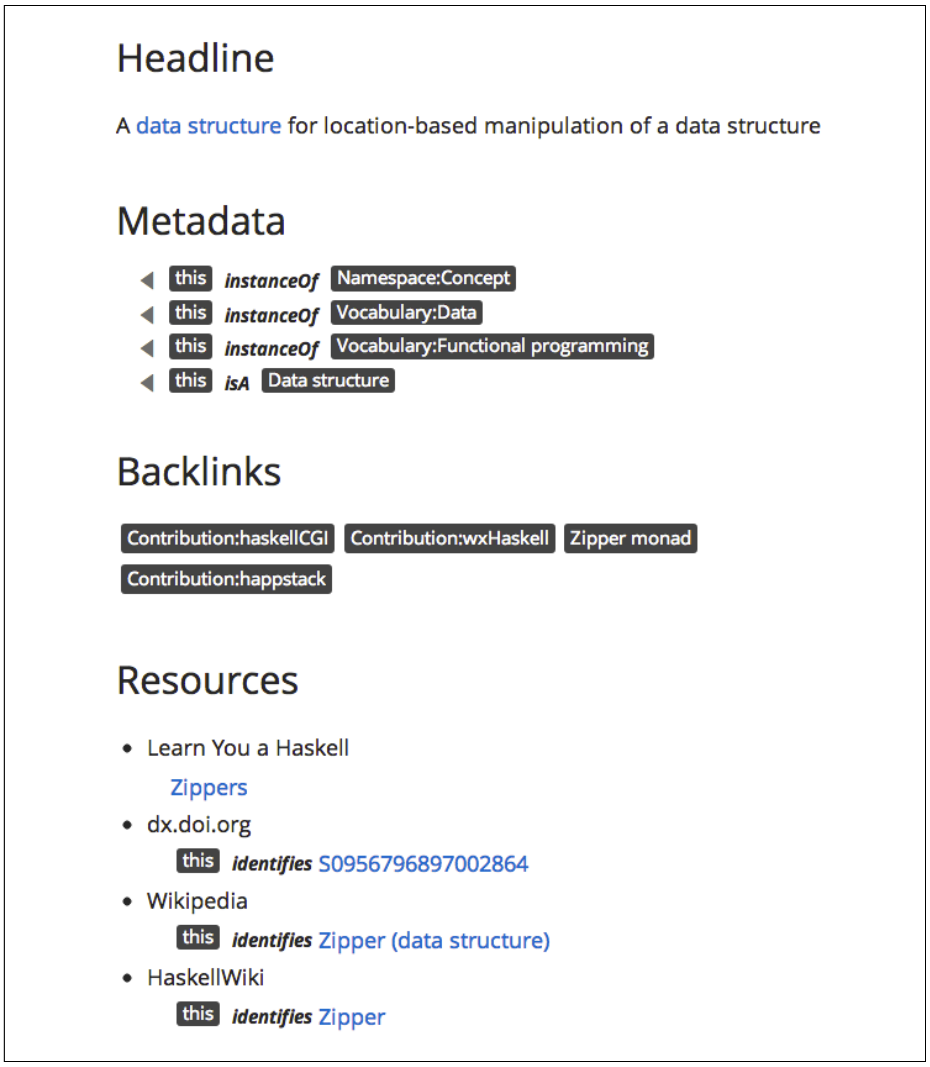
In the terminology of knowledge representation and integration, the software chrestomathy 101 [4] and specifically its wiki (i.e., 101wiki) can be viewed as a knowledge base; we also use the name 101kb, thus. That is, 101kb contains for *general software knowledge* with categories for software *concepts* (e.g., 'object composition' or 'unit testing'), software *languages* (e.g., 'Haskell', 'XML', and 'SQL'), and software *technologies* (e.g., 'javac', 'hibernate', or 'ant'). Further, 101kb contains more *specific, illustrative software knowledge* in terms of documentation for many implementations of the 101system—a Human Resources Management System. These implementations are also referred to as *contributions* because they constitute the central means of contributing to the the community resource 101.

Consider Figure 1 for some illustration of knowledge available through 101kb. Knowledge about the software concept 'Zipper' is shown. The *Headline* section explains the concept in informal terms. The *Metadata* section contains classification-related or ontological knowledge. In particular:

– 'this' (i.e., 'Zipper') is an instance of the namespace for software concepts.
– 'this' is a member of the vocabularies 'data' and 'functional programming'.
– 'this' is classified as a data structure.

The *Backlinks* section readily reports on other wiki pages mentioning the 'Zipper' concept. Specifically, three contribution pages are listed—these are documentations of small software systems in the chrestomathy which make use of

[1] http://101companies.org/
[2] https://github.com/101companies/101integrate



Headline

A data structure for location-based manipulation of a data structure

Metadata

◀ this *instanceOf* Namespace:Concept
◀ this *instanceOf* Vocabulary:Data
◀ this *instanceOf* Vocabulary:Functional programming
◀ this *isA* Data structure

Backlinks

Contribution:haskellCGI Contribution:wxHaskell Zipper monad
Contribution:happstack

Resources

- Learn You a Haskell
  Zippers
- dx.doi.org
  this *identifies* S0956796897002864
- Wikipedia
  this *identifies* Zipper (data structure)
- HaskellWiki
  this *identifies* Zipper

Fig. 1. The software concept 'Zipper' as rendered on the 101wiki.

a zipper or are inspired by the concept. Another mention arises from the page for the software concept 'Zipper monad' which essentially provides a specific implementation of the concept.

Ultimately, the *Resources* section links to external resources. The 'Learn You a Haskell' resource is the online available textbook of the same name [8]; the link takes one right to the book's chapter on zippers. Further, there is a DOI-based reference for the seminal paper on zippers [5]. Finally, there are also links to zipper-related pages on Wikipedia and HaskellWiki. Except for the textbook link, all the other resource links are marked with the predicate 'identifies', which, in the ontology of 101, implies that these resources are judged as being dedicated to the relevant concept as opposed to merely providing relevant information, in which case a weaker predicate 'linksTo' would be used. Textbook links are currently not classified in this manner.

The integration challenge

- What existing vocabulary (e.g., software concepts) to promote in the 101kb, given that each additional term requires validation and effort in integrating the term into the ontology of 101 and using the term, e.g., in the documentation of contributions?
- How to assess the quality of resource usage in the 101kb, when we assume that mere reification of a term on the 101wiki is not sufficient, but instead each integrated term should be referenced in some meaningful way? Specifically, the documentation of contributions should reference software concepts.

In the following sections, we address this challenge through a notion of continuous knowledge integration. This notion is not in any way specific to the 101 context; the principles are generally valid for the integration of community resources.

3 Principles of continuous knowledge integration

A systematic and reproducible approach to continous knowledge integration (CKI) relies on several principles which we set up in this section. One can think of these principles also as a small system of design patterns for CKI solutions. (For what it matters, we are also inspired by the principles of 'continous integration' in software engineering.[3]) The CKI principles can be organized in groups (in fact, 'phases') as follows:

Integration =		
	Selection	— Select a resource
	+ **Extraction**	— Extract knowledge from the resource
	+ **Import**	— Import knowledge into the knowledge base
	+ **Maintenance**	— Maintain resource integration

Along these phases, different kinds of expertise are reuqired, giving rise to roles of those involved in CKI—expertise regarding the technical characteristics of the resource, the conceptual characteristics (actual content) of the resource, text mining [10], the knowledge base used for integration, and possibly others.

3.1 Principles for *selecting* a resource

These principles codify actions to be taken when a resource is selected for integration. In fact, these principles may help in deciding whether a resource should be selected and what the level of integration may be along the process to come.

[3] `http://en.wikipedia.org/wiki/Continuous_integration` (Last visited 18 June 2013)

Profiling "*Profile resource under integration.*"

– Is the resource *online* or *offline*?
– Is it *open* or *closed* (so that one can access it openly or only with credentials)?
– Is it *positionable* (so that one can navigate to some context of the resource)?
– Is it *controllable* (so that one can add information to it, e.g., backlinks)?

Licensing "*Establish license for resource under integration.'*"

– What are the terms of *analyzing* the resource algorithmically?
– What are the terms of *publishing* results of any analysis?
– What are the terms of *controlling* the resource for adding information?

Expertise "*Identify experts for resource under integration.*"

– Who can implement mining algorithms for the resource?
– Who can (in conceptual terms) validate the extracted knowledge?
– Who can (in practical terms) position or control the resource?

3.2 Principles for *extracting* knowledge from the resource

These principles codify actions to be taken when data is extracted from a resource. Such extraction consists of steps for content normalization, actual mining to retrieve a vocabulary (or generally other kinds of knowledge), validation, and reporting.

Normalization "*Implement a normalizer for the resource.*"
The resource is mapped from its resource-specific representation to a more standardized format (e.g., stemmed raw text) to which mining algorithms could be applied. Along normalization, the structure of the resource (e.g., in terms of book chapters) is to be preserved so that the origin of terms is carried along.

Mining "*Extract knowledge algorithmically.*"
In this paper, we focus on vocabularly mining, but taxonomy or ontology mining could also be of interest. With such focus on vocabulary mining, a list of candidate terms is to be generated algorithmically. The list shall be of a manageable size to enable manual validation and other human-based effort. Mining may leverage, for example, text-mining techniques that are picked in a resource-specific manner. Multiple criteria for mining could be used, e.g., vocabulary mining based on absolute popularity versus inverse document frequency.

Validation "*Validate extracted knowledge.*"
An expert may assess the correctness of the extracted knowledge (e.g., terms) to account for effects due the involved mining techniques and their specific application (e.g., stemming issues, anomalies of content, configuration of thresholds). In this manner, some extraction results may be marked for exlusion and manual additions may be noted as well. Preferably, such validation is performed independently (at least partially) by multiple experts, thereby enabling measurement of the accuracy of validation.

Reporting "*Produce a detailed mining report for scrutiny.*"
The report describes the parameters and the results of the applied mining techniques as well as the results of validation so that the extraction is transparent and can be further validated by others (specifically also by the authors of the resource under integration) and also used as an accessible experience in performing integration for other resources.

3.3 Principles for *importing* the resource into the knowledge base

Mapping "*Map extracted terms to existing terms.*"
Those extracted terms with an existing counterpart in the knowledge base may be readily mapped. The term used by the integrated resource may of course differ from the term used in the knowledge base. Expertise regarding both the integrated resource and the knowledge base is required in devising the mapping. As with validation (see above), mapping should be preferably performed independently (at least partially) by multiple experts, thereby enabling measurement of the accuracy of mapping.

Reification "*Reify new entities in the knowledge base.*"
There are likely to be extracted terms, which do not yet exist in the knowledge base. Thus, they must be reified in the knowledge base. Some standard procedure may apply here. For instance, a normalized name may be assigned to the term; it may be injected into existing categories; it may be connected to standard resources such as Wikipedia.

Positioning "*Establish positioning access to the resource.*"
Given a confirmed term and an origin (such as a chapter or specific paragraph), we must be able to render the resource with the relevant origin used for positioning. For instance, we may use URL-based positioning for an online resource. Positioning is a prerequisite for executing the links established by mapping and reificiation. Positioning may also be helpful for validation and mapping (see above).

Publishing "*Publish a discoverable mapping.*"
An integrated resource is published when the exploration of the knowledge base (e.g., by a website) is expected to reveal links to the integrated resource in a discoverable manner—also subject to infrastructure support including positioning (see above).

3.4 Principles for *maintaining* resource integration

Organization "*Organize imported knowledge.*"
The mere mapping of extracted terms needs to be complemented by efforts to advance or revise the taxonomy or ontology of the knowledge base to better account for the new terms. For instance, new categories may need to be created so that they can be populated with extracted terms.

Monitoring "*Monitor knowledge base.*"
When a vocabulary or other knowledge was extracted from a resource and mapped to a knowledge base, then a reasonable expectation is that the knowledge is used (referenced) systematically. This expectation should be monitored on the grounds of appropriate automated queries (e.g., for vocabularly usage) to provide feedback to knowledge integrators and content authors.

Testing "*Automate regression testing for resource.*"
Evolution of the resource may break resource integration in different ways. First, all algorithmic steps (notably normalization and mining) may break because of format changes. Second, extraction results may deviate. Some changes may be tolerated by the architecture, but this is not sensible when the new results are no longer covered by the existing mapping. Third, positioning access may break. Thus, regression testing should address these problems.

4 Selection of Haskell textbooks as resources

In the following few sections, we report on a case study in addressing the CKI principles in the context of enriching the 101kb by vocabulary which is extracted from selected, popular textbooks on the functional programming Haskell.

We begin with the selection of the resources (see the principles of §3.1). We apply continous knowledge integration to these popular textbooks on functional programming in Haskell:

CRAFT [11] "Haskell: The Craft of Functional Programming"
PIH [6] "Programming in Haskell"
RWH [9] "Real World Haskell"
LYAH [8] "Learn You a Haskell"

Selection was influenced by the assumed objective to cover Haskell in the 101kb specifically in the context of teaching functional programming at the introductory level. Some of the more powerful programming techniques such as functors or monads should be covered also.

We determined that two popular textbooks were available online with open access: [9,8]. (We are not aware of any other Haskell textbooks with this profile.) So we favored these two books as we intended to produce a good experience in properly connecting to textbooks from the 101wiki, especially for the benefit of students in the corresponding programming course.[4]

We decided to select more resources to increase diversity and to actually study the contributions of different resources in a meaningful way. Thus, we selected the offline resource [6] because it is an established introductory text with which we were also familiar through teaching. Further, we selected the offline resource [11] because it is also an established text on Haskell; its mathematical or logical approach was thought of as being complementary to the other texts.

[4] `http://101companies.org/wiki/Course:Lambdas_in_Koblenz`

In both cases, we were able to negotiate terms with the authors such that we could indeed access and analyze offline sources for the books and publish the extracted vocabularies including some additional reporting information.

5 Vocabulary extraction from Haskell textbooks

We continue with the extraction of the knowledge (in fact, vocabulary; see the principles of §3.2) for the case study. Modulo up-front normalization, we performed extraction in a uniform way for all textbooks. We required access to a book's index to focus vocabulary extraction on terms readily indexed. Accordingly, we matched preprocessed index entries from available textbooks with the book's content so that candidates terms were identified with the help of text summarization techniques [1] including inverse document frequency [7] to take into account distribution of terms over chapters. Processing index and content of each books involves data cleaning, stemming, and ranking. This process is largely automated by the 101integrate framework; we mention human intervention when it occurs.

5.1 Normalization

Books PIH and CRAFT were available to us through their LaTeX sources. Chapter structure and index entries were 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. W.l.o.g., merely for convenience, index entries were extracted from the indices for the ebooks.

Index normalization We started with a raw list of index entries which we processed as follows. Duplicates and special characters and symbols were automatically removed, e.g., $\Rightarrow$ and $\neq$. Subentries (such as 'associativity: using with monads') were also removed automatically.

101integrate leverages the Natural Language Toolkit (NLTK [3]) for stemming. To this end, each raw term is split into words (using NTLK functionality), each word is stemmed, and the resulting words are joined back together. For instance, singular and plural forms of a term are grouped to the singular form in this manner. As the result, one obtains a normalized set of terms that is free of trivial redundancy.

In fact, prior to standard stemming, specific, pattern-based normalizations were performed to account for the special nature of the index entries at hand. These patterns were authored by us and then automatically applied. For brevity, we skip such custom stemming here, as its impact on the final results turned out to be minimal.

Finally, many raw terms were removed automatically by applying WordCount[5] as a stop list, i.e., a list of "common English words". This step was

[5] http://www.wordcount.org/main.php

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

prepared by reviewing raw terms (from all books) sorted by their ranks. We decided that Rank 90 is a suitable 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.

Content normalization We did not attempt vocabulary extraction for the source code because it would add some additional challenges. Thus, we identified markup (e.g., LaTeX environments or the <pre> tag of HTML) used for code samples and we set up rules with `101integrate` to remove all code automatically prior to matching.

Further, all LaTeX and HTML markup was removed automatically prior to matching. The content was also further normalized by steps as we explained for the index entries: case conversion, pattern-based normalization, stemming, and exclusion based on the stop-list. We excluded *introduction*-, *preface*-, and *conclusion*-like chapters manually.

5.2 Mining

The normalized index terms were to be matched with the normalized content of the book. Here, we note that the index entries from all books were 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 §6.

Terms of chapter profiles At this stage of the process, we are supposed to favor terms of a raw vocabulary for a given book. To this end, we adopted simple techniques of text summarization. That is, we selected validated index terms as candidate terms, if they occur 'frequently enough' in the book's content, but they are not scattered over 'too many' chapters of the book, subject to appropriate thresholds. Specifically, we excluded 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 favored 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.

Term	Getting started with Haskell and GHCi	Basic types and definitions	Designing and writing programs	Data types tuples and lists	Programming with lists	Defining functions over lists	Playing the game IO in Haskell	Reasoning about programs	Generalization patterns of computation	Higher order functions	Developing higher order programs	Overloading type classes and type checking	Algebraic types	Case study Huffman codes	Abstract data types	Lazy programming	Programming with monads	Domain Specific Languages	Time and space behaviour
action							·												
algebraic													•						
algebraic type													•						
base case								·											
bool												•							
calculation																•			·
code														•					
coding														·					
command	·																		
complexity																			•
constructor				·									•						
database				·															
design											·								
eq												•							
equality												•							
evaluation																•			
file	·																		
filter									·										
float		•																	
folding									·										
foldr									·										
GHCi	•																		
guard		•																	
head						·													
I/O							•										·		
induction								•											
infinite list																•			
IO							·										•		
local			·																
map									·	•									
maximum			·																
model							·												
module	•				·									●					
monad																	●	·	
operator		•								•									
package					•														
parser																•			
partial										•									
partial application										•									
pattern matching						·													
picture					●						·							•	
prelude					·														
proof								●											
queue															•				
random																		·	
recursion			•			·													
regular expression											·							·	
set															●				
state																	•		
strict																			·
testing			·					·											
text						·					·								
tree													•	•	•				·
tuple				·															
type checking												•							

Fig. 2. Chapter profiling for CRAFT

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

Fig. 3. Terms excluded from the chapter profile of CRAFT

Figure 2 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 candidate term (per row) and associates a chapter with it (per column). To provide additional insight, the size of the bullet represents matching frequency over candidate terms: ●: the 5 most frequent candidate terms; •: $\geq$ median; ∙: $<$ median.

Popular terms Due to the nature of the chapter profiling algorithm, some general (functional) programming terms, such as *function*, are not selected since they are used throughout the books. Arguably, we would also want extract such terms on the grounds of an alternative mining technique.

Thus, we picked candidates for popular terms per book as follows. We ordered matched terms by frequency, while we excluded terms that are readily in the chapter profile of the book. Then, we decided that popular terms are those that have >10 % frequency of the topmost term's frequency. Such threshold decisions are based on manual inspection of term lists produced by 101integrate.

5.3 Validation

Terms of chapter profiles were validated as follows. Manual inspection was performed to exclude 'uninteresting' terms. 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 3 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" [11]. Thus, the term is excluded. Of course, 101kb includes the concept of a 'model', but in a more specific sense than present in the book at hand.

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. Term metrics

Popular candidate terms were validated manually. Note that we have already excluded popular terms of common English earlier on in the process. Most candidate terms could be confirmed by validation. 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.

Table 2 summarizes metrics for found terms. In particular, 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 having to do with different views on essential or interesting concepts of (functional) programming. In case of doubt, we resolved disagreement by being inclusive.

As part of the validation process, we also evaluated whether the terms per chapter sufficiently capture the concepts conveyed by the title. Thus, we added so-called *title terms* for chapters, when this was 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 was thus added.

6 Vocabulary import into the 101kb

We continue the description of the case study with the discussion of the actual import of the extracted vocabulary into the 101kb (see the principles of §3.3). We systematically reify all validated terms in the 101kb and link the integrated resources with 101. Again, this process is largely automated by 101integrate and also benefits from infrastructural support of 101wiki, as it was illustrated in §2.

6.1 Mapping and reification

We review validated terms in the context of their contributing chapters and the existing terms in the 101kb to suggest a suitable mapping. (In practice, such manual work on mapping is intertwined with validation, as discussed earlier.)

If the relevant term is not yet available on 101kb, then it needs to be reified. Remember the illustration of the 'Zipper' concept in §2. *Metadata* and non-textbook *resources* were authored when reification was required because of the

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

Fig. 4. Mapping for the first few terms of CRAFT

term's occurrence in the vocabulary that was extracted from one textbook of the study [8].

Let us also consider an example where the mapping needs to account for terminology differences. The term 'class' appears as a term of CRAFT's chapter profile; see Figure 2. 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, which is the context assumed by 101, even though 'class' may be sufficiently clear in the narrow context of functional programming with Haskell.

Figure 4 shows the first few mapping entries from validated to 101 terms for CRAFT. For what it matters, 101integrate processes such mappings by means of a designated CSV file per integrated resource.

While it may be relatively simple to agree on whether or not a validated term should be reified, it may be harder to agree on the specific term to be used in the 101kb. There may be several reasonable candidates and points of views. Thus, domain-specific portals are consulted for resolution. We consulted HaskellWiki as well as Wikipedia, which also organizes functional programming knowledge. Also, we realized that there is a strong need for naming conventions for 101kb.

6.2 Comparison of the resources

Mapping also enables a sensible comparison of the vocabularies obtained from the different resources. We take the view that a resource (a textbook) is characterized in a distinguished manner, relatively to all other sources, by the terms that it uniquely contributes to the combined vocabulary. An expert may judge whether these unique contributions are meaningful.

Figure 5 lists the unique terms contributed by each textbook of the study; 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'.

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*

Fig. 5. Comparison of the different Haskell textbooks

– 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 vocabularies.

6.3 Positioning

In the case study, we developed positioning access for the open online textbooks. Positioning is used on the 101wiki to connect wiki pages to textbook paragraphs; see Figure 1 again for illustration.

6.4 Publishing

101integrate maintains all mappings and the origins of terms in integrated resources with positioning access enabled. The framework publishes the mappings

```
[ { "fullName": "Programming in Haskell", "name": "PIH",
    "isLinkable": false, "error": "missing mapping" },
  { "fullName": "Real World Haskell", "name": "RWH",
     "isLinkable": true, "error": "missing mapping"},
  { "fullName": "Haskell: The Craft of Functional Programming", "name": "Craft",
    "isLinkable": false, "error": "missing mapping"},
  { "fullName": "Learn You a Haskell", "name": "LYAH",
    "isLinkable": true,
    "primary": [
      {"chapter": "Zippers", "full": "http://learnyouahaskell.com/zippers"}],
    "secondary": []}
]
```

Fig. 6. Response of 101integrate's publishing service for 'Zipper'.

and origins effectively through a designated web service; see Figure 6 for the service's response for the term 'Zipper'.[6]. The JSON response lists all applicable resources and states whether or not the current term is mapped for each of the resources. It is also stated whether the resource is 'linkable' (i.e., whether positioning is supported in open online manner). An actual link is listed for one of the textbooks. The key 'primary' deals with terms of the chapter profiles; the key 'secondary' deals with popular terms; see §5. The 101wiki uses the service to retrieve resource links and render them as shown in Figure 1.

7 Maintenance of the 101kb

We complete the description of the case study with the discussion of maintaining resource integration (see the principles of §3.4).

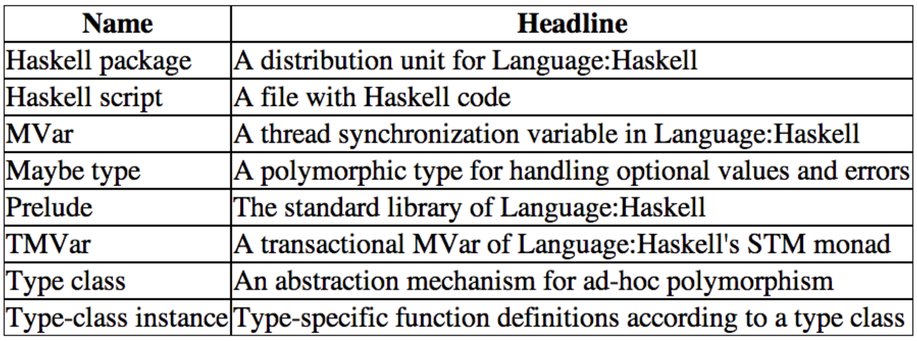
7.1 Organization of the 101kb

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.*

[6] Request URL: `http://worker.101companies.org/services/termResources/Zipper.json`

The introduction of these vocabularies and their assignment to specific terms is (deliberately) a manual process, which is informed by the review of all available sources including Wikipedia and HaskellWiki. 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

Fig. 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.

7.2 Monitoring the 101kb

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 documentation of 101 contributions. Accordingly, we should make sure that the extracted terms are eventually referenced—specifically by contributions. We consider it an important methodological aspect of CKI to keep on monitoring vocabulary usage.

Figure 8 summarizes vocabulary usage for the textbooks at hand and all Haskell-based contributions of 101. Terms are listed vertically and ordered by the number of referring contributions. Contributions are listed horizontally and ordered by the number of directly referenced terms. We cut off the listing of contributions when contributions have less than 3 direct references. We cut off the listing of terms for the first term with zero referring contributions; see 'Query' at the bottom. (Such views are computed by 101integrate.) The big bullets indicate proper references indeed, whereas the small bullets report on indirect references. For instance, several contributions refer (directly) to 'Zipper' as we already noticed in Figure 1.

Contribution/Term	haskellStarter (10/25)	haskellParsec (8/13)	haskellList (7/18)	hxtPickler (6/8)	monoidal (5/16)	haskellLambda (5/15)	haskellEngineer (4/17)	haskellComposition (4/12)	writerMonad (4/12)	haskellVariation (4/10)	wxHaskell (4/8)	haskellDB (4/7)	dph (4/4)	haskellTree (3/11)	nonmonadic (3/11)	happstack (3/10)	hxt (3/9)	mvar (3/9)	tmvar (3/8)	haskellCGI (3/6)	haskellData (3/6)	haskellRecord (3/6)	hdbc (3/3)
Language:Haskell (30/0)	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Technology:GHC (17/13)	•	●	●	•	●	●	●	●	●	●	●	●	●	●	●	●	•	•	•	●	•	•	•
Technology:GHCi (16/14)	●	•	•	●	•	•	•	•	•	•	●	●	●	•	•	•	●	●	●	•	●	●	●
Algebraic data type (7/23)	●	●	•	●	•	•	•	●	•	●	•	•	•	•	•	•	•	•	•	•	●	●	•
Monad (3/13)	•	●	•	•	•	•	•	•	●	•					●		•		•				
Zipper (3/1)											●					●			•	●			
Data constructor (2/14)	•	•		•	•			●		●					•						•	•	
Map function (2/11)	•	•	●		●	•	•	•	•	•				•	•								
Anonymous function (2/10)	•	•	●		•	●	•	•	•	•					•								
MVar (2/8)		•							•		•	•				•	•	●	●				
Fold function (2/2)			●		●	•								•									
Recursion (2/2)	●		●			•	•																
Higher-order function (2/1)			●		•	●																	
Database (2/0)												●											●
Language:XML (2/0)				●													●						
Technology:Parsec (1/17)	•	●	•		•	•	•	•	•	•	•	•			•	•	•	•					
Module (1/11)	•		•			•	●	•				•	•					•	•		•	•	
Monoid (1/11)	•	•	•		●	•	•	•	•	•					•								
Parsing (1/9)		●		•					•		•	•				•	•	•					
Writer monad (1/9)		•							●		•	•			•	•	•	•					
Local scope (1/7)	•		•			•	●	•													•	•	
Type class (1/6)		•		●	•				•					•	•		•						
Tuple (1/3)	●						•	•		•													
Prelude module (1/2)	•		•										●										
Float (1/1)	●						•																
Function application (1/1)	●						•																
Lambda abstraction (1/1)			•			●																	
Monad transformer (1/1)		●																					
Parser combinator (1/1)		●																					
Pattern matching (1/1)	●						•																
Pure function (1/1)	●						•																
String (1/1)	●						•																
Type-class instance (1/1)				●	•																		
Functor (1/0)														●									
Query (0/26)	•		•	•	•	•	•	•		•	•	•	•	•		•	•	•	•	•	•	•	•

Fig. 8. Usage of the vocabulary from the integrated textbooks in Haskell-based contributions of 101 (with the less referenced terms and the less referring contributions not shown).

There are clearly many more terms without contributions directly referring to them. In fact, there are even terms without any reference. This status may be viewed as an action item to add insight to the 101kb so that all neglected terms are properly referenced eventually. This sort of monitoring gives rise to a designated role of 'knowledge integrators' such that they commit to the resolution of action items for under-referenced terms. Likewise, authors of the documentation for contributions receive feedback from such a view, as they are reminded of the fact that they perhaps do not mention enough meaningful (functional) programming concepts.

8 Conclusion

We have described and demonstrated continuous knowledge integration (CKI) as being focused on community resources for software knowledge. Our case study shows that CKI is suitable, specifically, for the consolidation of technical vocabulary from community resources such as textbooks and wikis. The resulting vocabulary is readily helpful in teaching programming and the documentation of programs.

The semi-automatic characteristics of CKI imply that all involved authors remain closely familiar with the vocabulary and the mapping along the process; also, knowledge consumers are not overwhelmed by overly sized / inclusive vocabularies. Continuity of integration relies, for example, on special means of monitoring term usage, which provides actionable feedback both to knowledge integrators and knowledge consumers.

Our realization of CKI relies on a framework 101integrate for all automated steps and 101wiki as the wiki (or knowledge base, say 101kb) that integrates software knowledge. The 101wiki can be used by both knowledge consumers and knowledge integrators. The achieved level of linking support combined with some other features of 101 such as its use of an ontology for knowledge organization aspires to the notion of a knowledge integration environment [2].

In future work, we will continue validation and generalization of 101integrate so that the process becomes technically totally painless and knowledge integrators can focus on the intellectual dimension of CKI. In §3, we also touched upon the issue of 'controlling resources', e.g., by adding rich links to an integrated resource entity to link back to the integrated knowledge base. For instance, we hope to experiment with this idea for community resources such as GitHub and StackOverflow.

Acknowledgment We are grateful to Graham Hutton and Simon Thompson for sharing the sources of the Haskell books [6,11] with us for the purpose of this research. We are also very grateful to several people who helped on the technical part of this work—specifically, Kevin Klein and Martin Leinberger.

References

1. Aggarwal, C.C., Zhai, C. (eds.): Mining Text Data. Springer (2012)
2. Bell, P., Davis, E.A., Linn, M.C.: The knowledge integration environment: theory and design. In: The first international conference on Computer support for collaborative learning. pp. 14–21. CSCL '95 (1995)

3. Bird, S., Loper, E., Klein, E.: Natural Language Processing with Python. O'Reilly Media Inc. (2009)
4. Favre, J.M., Lämmel, R., Schmorleiz, T., Varanovich, A.: *101companies*: a community project on software technologies and software languages. In: Proc. of TOOLS 2012. LNCS, vol. 7304, pp. 59–74. Springer (2012)
5. Huet, G.: The Zipper. J. Funct. Program. 7(5), 549–554 (1997)
6. Hutton, G.: Programming in Haskell. Cambridge University Press (2007), http://www.cs.nott.ac.uk/ gmh/book.html
7. Jones, K.S.: A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation 28, 1121 (1972)
8. Lipovaca, M.: Learn You a Haskell for Great Good! no starch press (2011), http://learnyouahaskell.com/
9. O'Sullivan, B., Stewart, D., Goerzen, J.: Real World Haskell. O'Reilly Media (2008), `http://book.realworldhaskell.org/`
10. Rajman, M., BESANON, R., Besancon, R.: Text mining: Natural language techniques and text mining applications. In: In Proceedings of the 7 th IFIP Working Conference on Database Semantics (DS-7). Chapam. pp. 7–10. Hall (1997)
11. Thompson, S.: Haskell: The Craft of Functional Programming (3rd edition). Addison-Wesley (2011), `http://www.haskellcraft.com/craft3e/Home.html`