The 101haskell Chrestomathy
A Whole Bunch of Learnable Lambdas

Ralf Lämmel  Thomas Schmorleiz  Andrei Varanovich
University of Koblenz-Landau, Software Languages Team, Germany

Abstract
The paper describes the 101haskell chrestomathy—a collection of Haskell programs implementing features of a hypothetical information system in a manner to represent knowledge about functional programming useful for learning (and teaching). The programs are enriched with documentation, metadata, and links to other knowledge resources such as Wikipedia and Haskell textbooks. The underlying ontology is informed by a process of knowledge integration which derives a consolidated vocabulary mainly by text mining and summarization from textbooks. The usefulness of 101haskell for teaching is demonstrated with a functional programming course that is directly based on 101haskell.

Categories and Subject Descriptors H.1.1 [Programming Techniques]: Applicative (Functional) Programming

General Terms Languages, Documentation.

Keywords Program chrestomathy, Software chrestomathy, Functional programming education, Semantic metadata, Knowledge integration, Wikipedia, 101companies, 101haskell, Haskell.

1. Introduction
The 101companies project`s 101` for short, aims at the aggregation of knowledge about programming and software engineering in a manner that is useful for learning (and teaching) in these areas. 101haskell, which is the subject of the present paper, is an elaborate branch of 101; it focuses on functional programming and Haskell, specifically.

Basically, 101haskell is a structured collection of Haskell programs, while 101 covers many other languages. Importantly, the programs are enriched with documentation, metadata, and links to other knowledge resources such as Wikipedia and Haskell textbooks. Also, the programs implement features of a hypothetical information system and they demonstrate different programming domains, concepts, techniques, and technologies.

In fact, 101haskell is an advanced program chrestomathy, i.e., a collection of programs useful for learning. Chrestomathies are widely used in the programming field, and 101haskell’s characteristics enable its direct use in teaching.

Contributions of this paper
• Validation of a chrestomathy’s usefulness in teaching. An introductory functional programming course is layered directly on top of 101haskell. The validation boils down to the argument that the available content, in the intrinsic form (semantic wiki, interlinked source code, interlinked external resources) provides the foundation of a viable course.

• A well-engineered Haskell chrestomathy. 101haskell collects Haskell-based software systems implementing features of a simple information system. Different techniques and technologies are exercised. Software engineering best practices are applied—practices with regard to documentation, testing, modularization, and build management.

• Enrichment of a chrestomathy with an ontology. To this end, metadata is associated with all entities (implemented features, relevant languages, technologies, concepts, external resources, etc.) for classification and other relationships. Such enrichment is applied to 101haskell.

• Knowledge integration for a chrestomathy. A textbook-driven process establishes a consolidated vocabulary for interlinkage across chrestomathy, wikis, and textbooks. The degree of knowledge integration in terms of vocabulary usage can be monitored. Integration is applied to 101haskell, Wikipedia, HaskellWiki, and 4 Haskell textbooks [21, 34, 39, 46].

Our contributions leverage concepts of knowledge integration, applications of ontologies in software engineering, concepts of the Semantic Web and Linked Data, as well as techniques of text mining and summarization. The results of the present paper are expected to be meaningful to other programming paradigms and other areas of computer science.

Road-map of this paper provides background on the notion of chrestomathy, in general, and 101haskell and 101, in particular. sketches an excerpt of 101haskell including some aspects of chrestomathy organization and software engineering. describes enrichment of a chrestomathy with an ontology for 101haskell. describes knowledge integration for 101haskell. characterizes an introductory functional programming course based on 101haskell. Related work is summarized in and the paper is concluded in

See the paper’s website for accompanying material.

http://101companies.org

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2988-0/13/10...$15.00.

http://en.wikipedia.org/wiki/Chrestomathy
http://softlang.uni-koblenz.de/101haskell

http://101companies.org
Relationship to our previous work

An early overview of 101 appeared in [14]. 101haskell did not properly exist at that point; semantic enrichment [4] and knowledge integration [5] were developed only later. Further papers, [13] [15] [28], relate, to some extent, to 101: techniques in software reverse engineering and software architecture are validated with the help of 101. These techniques are orthogonal to what is described in the present paper and either irrelevant or unexplored for 101haskell at this point.

2. Background on 101haskell and 101

We quickly recall the chrestomathy notion in philology, linguistics, programming, and software engineering. We position the ‘101 project as a software chrestomathy. Ultimately, we position 101haskell as the Haskell-specific sub-chrestomathy of 101. (We refer to [26] for a broader discussion of the chrestomathy notion and to [14] for a basic introduction of 101.)

Chrestomathies in philology or linguistics

The term chrestomathy refers to a collection of sample texts (‘literary passages’), usually in one language, possibly from different authors, designed by one or more authors, designed to be useful in learning a language, specifically its grammar. For instance and quoting from [28], the ‘Coptic Gnostic Chrestomathy’ [29] collects texts in Coptic language; these texts are systematically edited to include annotations for grammatical analysis such as relationships between prepositions, verbs, and nouns. The term chrestomathy is formed from Greek ‘chresto’ (Engl.: ‘useful’) and ‘mathein’ (Engl.: ‘to learn’).

Program chrestomathies

The notion of program (or programming) chrestomathy has been in use in the broad programming community for several years now in the wild [26] [41] A program chrestomathy is a collection of sample programs in one or more programming languages designed to be useful in learning about programming and programming languages. Such a chrestomathy may focus on specific language aspects such as comparison of programming style, expressiveness, and applicable programming techniques in one language or across different languages; see, e.g., [35] [43] [44] [50]. Importantly, the collected programs are expected to implement certain features (say, tasks or requirements), as prescribed by the chrestomathy.

Software chrestomathies

When a program chrestomathy starts to collect (buildable, runnable, modularized) systems rather than just programs in the sense of single source-code units, then we also speak of a software chrestomathy. When moving from programs to systems, a number of software engineering-related aspects may be covered, e.g., build management, testing, documentation, and modeling of the features to be implemented by the systems.

The 101companies project

The 101companies project [14]. ‘101’ for short, is a community project aimed at collecting systems that implement certain features of a small, hypothetical information system, the 101system for human resource management, in different ways. These systems are referred to as ‘contributions’; they are maintained in a federated GitHub-based repository— the 101repo. All contributions are documented on a wiki for the project—the 101wiki.

Here are some metrics for the 101 software chrestomathy:

- 25 features that can be implemented, subject to a feature model [5], e.g.: Feature Hierarchical company for the data model of a company to break down into departments recursively with employees and a manager per department; Feature Total for computing the salary total for all employees in a company; Feature Logging for logging all salary changes. About 10 features are popular. The remaining features are either more specific or perhaps premature or obsolete.

- There are 193 contributions, i.e., implementations of the 101system. (Several of these contributions are premature, unmaintained, insufficiently documented, or otherwise suboptimal in terms of software engineering best practices.)

- 40 languages (mostly programming languages) are exercised.

- 96 technologies (e.g., libraries) are exercised.

- 477 concepts are referenced by the wiki-based documentation. The term concept is used here in the broad sense to include programming techniques, programming domains, classifiers (for programming languages, technologies, and features), etc.

The knowledge engineering challenge

With all the many features, contributions, languages, technologies, and concepts in scope, one can easily understand that some degree of knowledge engineering is needed for maintaining the quality and usefulness of a chrestomathy. This is the central research challenge addressed by the present paper. We identify the following research questions:

- What is the ontology administering a chrestomathy so that all involved entities (such as code artifacts, languages, technologies, and concepts) are organized (classified, associated)?

- What is the vocabulary of such an ontology, e.g., the concrete programming concepts to be covered by the feature model or to be mentioned in the documentation of contributions?

- How can existing knowledge resources, such as textbooks or Wikipedia, be usefully integrated in the scope of a chrestomathy for the immediate benefit of users?

The 101haskell chrestomathy

101haskell may be understood simply as the Haskell-specific cut through 101 and it thus demonstrates different functional programming techniques and technologies. There are 35 Haskell-based contributions.

It turns out that 101haskell served as the driving force in advancing the chrestomathy notion assumed in 101 over the last 18 months. As a result, 101haskell is best suited to demonstrate some aspects of an advanced chrestomathy:

- Software engineering best practices [4]:
  - The contributions are hosted on a sub-repo of 101repo.
  - The Haskell code is modularized.
  - Haskell’s Cabal is used for build management.
  - Haskell’s HUnit is used for unit testing.
  - The documentation is semantically enriched [4].

- Haskell textbooks are integrated with the 101wiki [5].

3. An excerpt of 101haskell

We present the excerpt in a feature-driven manner: one feature per subsection, with the following format for the subsections:

Requirement specification

The feature is explained at the level of a requirement specification for an information system. For instance, we may encounter a functional requirement for a specific functionality expected by a user of the 101system.

Chrestomathic purpose

The feature is motivated in terms of its value for a software chrestomathy. That is, we mention relevant programming domains, concepts, techniques, or classes of technologies that are naturally associated with the feature.

Haskell illustration

One or more Haskell-based contributions (‘implementations’) are briefly sketched and explained. The
emphasis is on pointing out the involved programming techniques, technologies (libraries), and concepts.

### 3.1 Feature Flat company

#### Requirement specification

A data model of companies is to be supported as follows. A company has a name and aggregates employees. Each employee has a name, an address, and a salary. Names of companies and employees are strings; addresses are strings, too; salaries are floating-point numbers. For now, companies are ‘flat’ in that they aggregate employees without any hierarchical company structure, but see ‘non-flat’ companies in §3.2.

**Chrestomathic purpose**

The data model exercises very basic data modeling facets: primitive types, tuples, lists, and non-recursive type declarations.

**Haskell illustration** *(Contribution haskellStarter)*

The data model is defined in terms of a system of type synonyms. Company and employee term types are composed as tuples. Employees are aggregated as lists. Salaries are represented as single-precision floating-point numbers. Thus:

```haskell
type Company = (Name, [Employee])
type Employee = (Name, Address, Salary)
type Name = String
type Address = String
type Salary = Float
```

Consider the following sample instance (with an elision):

```haskell
sampleCompany :: Company
sampleCompany =
  ("Acme Corporation",
   [Department "Research"
    (["Craig", "Redmond", 123456],
     ["Erik", "Utrecht", 12345],
     ("Ralf", "Koblenz", 1234)),
   ...)
```

### 3.2 Feature Hierarchical company

#### Requirement specification

Departments are added to the data model of **Feature Flat company**. Companies aggregate (top-level) departments. Each department has a name, a manager, and it aggregates both sub-departments as well as employees—other than the manager.

**Chrestomathic purpose**

The nesting of departments necessitates the data modeling facet of recursive types. Also, the fact that a department aggregates both sub-departments and employees implies that a choice must be made as to whether homogeneous or heterogeneous collections are used. In the first case, a department has two separate (homogeneous) containers—one for employees, another for sub-departments. In the second case, employees and departments end up in one (heterogeneous) container.

**Haskell illustration** *(Contribution haskellComposition)*

The homogeneous option is exercised here. The data model is defined mainly in terms of type synonyms, as before, but an algebraic data type is used for recursive departments since type synonyms cannot be recursive.

```haskell
data Department
  = Department Name Manager [Department] [Employee]
```

Consider the following sample instance (with an elision):

```haskell
classCompany :: Company
classCompany =
  ("Acme Corporation",
   [Department "Research"
    (["Craig", "Redmond", 123456],
     ["Erik", "Utrecht", 12345],
     ("Ralf", "Koblenz", 1234)),
   ...)
```

### 3.3 Feature Total

#### Requirement specification

Given a company, the salaries of all its employees are to be totaled. An implementation of this functional requirement is to be demonstrated for a sample company, e.g., by means of a unit test.

**Chrestomathic purpose**

Conceptually, the required functionality corresponds to a query over the company structure such that all employees are projected to their salaries, which are then summed up. Depending on the data modeling and programming paradigm at hand, it may be straightforward to quantify all employees (e.g., in a relational database with a table for employees) or it may require some sort of walking over the tree-like structure of a company (e.g., in an object-oriented or functional program). In the latter case, various programming techniques for traversal could be exercised, e.g., visitors in OO programming or generic functions in functional programming [27][43].

**Haskell illustration** *(Contribution haskellList)*

The following implementation assumes the most basic data model (as of §3.1). Some non-basic bits of functional programming come to play in this contribution. That is, we use the map combinator for list processing and local scope for arranging helper functions in the scope of total. Also, function composition is used explicitly, thereby providing an example of point-free style. Thus:

```haskell
total :: Company → Float
total = sum . salaries
where
  salaries :: Company → [Salary]
salaries (n, es) = map getSalary es
  where
    getSalary :: Employee → Float
    getSalary (n, a, s) = s
```

Haskell’s **HUnit** framework for unit testing is leveraged for exercising the query. That is, a comparison of the expected result of total with the actual result is declared as follows:

```haskell
hUnitTest :: Test totalTest
totalTest = total sampleCompany ≈ totalTest = 399747.0 ≈ sampleCompany
```

### 3.4 Feature Cut

#### Requirement specification

Given a company, the salaries of all employees of the company are to be cut in half. An implementation of this functional requirement is to be demonstrated for a sample company, e.g., by means of a unit test.

**Chrestomathic purpose**

Conceptually, the required functionality corresponds to a transformation over the company structure such that each employee is updated in terms of the salary, while preserving the structure and the
The following implementation assumes the data model with nested departments (as of 12). The transformation is modeled as a pure function. Local scope and map is used for clarity and conciseness.

```haskell
cut :: Company -> Company
cut (n, ds) = (n, (map cutD ds))
  where
    -- Cut all salaries in a department
cutD :: Department -> Department
cutD (Department n m ds es) = Department n (cutE m) (map cutD ds) (map cutE es)
    where
      -- Cut the salary of an employee in half
      cutE :: Employee -> Employee
cutE (n, a, s) = (n, a, s/2)
```

The transformation is exercised by the following HUnit test, which simply verifies that the individual cuts of the employees equate to cutting in half the total:

```haskell
cutTest :: Test
cutTest = total sampleCompany / 2 ~=? total (cut sampleCompany)
```

The following implementation leverages generic functional programming according to the 'Scrap your boilerplate' style (SYB 27):

```haskell

cut :: Company -> Company
cut = everywhere (extT id (/2::Float))
```

The generic function is clearly more concise than the previous attempt. The function is applicable to any data model for companies for as long as floating-point numbers are used for salaries only, but the function could also be revised slightly to apply instead more carefully to salaries.

Various other generic programming approaches could be illustrated as well, also including variations on SYB; see, for example, a comparison of approaches in 43. In fact, 101haskell covers a few approaches already.

All 101haskell contributions rely on Cabal for build management. In this manner, they are easy to build including dependency chasing on Hackage; they are easy to test, too. Here is the Cabal spec for Contribution haskellSyb (with an elision):

```haskell
name: haskellSyb
version: 0.1.0.0
synopsis: Generic Programming in Haskell with SyB
homepage: http://101companies.org/wiki/Contribution:haskellSyb
build-type: Simple
cabal-version: > = 1.9.2
library
  exposed-modules:
    Company.Data
    Company.Sample
    Company.Generic
    Company.Total
    Company.Cut

  build-depends: base >= 4.4 && < 5.0, syb >= 0.3 ...
```

### 3.5 Feature Parsing

**Requirement specification**

Company data is to be represented in a format that is suitable for human consumption and editing, e.g., in some well-defined textual concrete syntax, a schema-defined XML-based format, or CVS. This representation is to be parsed for the purpose of carrying out computations. For brevity, we omit here any discussion of demonstration (testing).

**Chrestomathic purpose**

Parsing brings the programming domain of language processing into the scope of the chrestomathy. In this manner, different kinds of parsers (e.g., context-free grammar-based ones or XML parsers), different parser technologies, and different styles of parsing can be exercised and possibly compared.

The corresponding parsing function follows:

```haskell
parseDepartment :: Parser
parseDepartment = do
  parseString "department"
  parseLiteral
  parseString "{"
    parseManager
    many parseSubUnit
  parseString "}"
```

### 3.6 Feature Logging

**Requirement specification**

Updates of salaries are to be logged so that salary changes can be reviewed afterwards. Each log entry identifies the relevant employee by name and lists salary before and after update. In particular, such logging is to be supported for Feature Cut. Here is an example of a log in CSV format:

```
"Craig", 123456.0, 61728.0
"Erik", 12345.0, 6172.5
"Ralf", 1234.0, 617.0 ...
```

The log is to be analyzed in a statistical manner to determine the median and the mean of all salary deltas. Such demonstration (testing) is omitted for brevity.

**Chrestomathic purpose**

Logging updates requires some level of program instrumentation so that a plain computation involving updates does indeed log those updates. Different programming techniques may be used for
this purpose. Logging calls for separation of concerns such that logging should not affect parts of the program that are conceptually unrelated to logging. Indeed, logging is a favorite aspect-oriented programming scenario \[24\] in that aspects (advanced modules) may facilitate separation of concerns.

1st Haskell illustration  
(Contribution \texttt{haskellLogging})

Logs may be modeled in Haskell as lists of entries as follows:

\begin{verbatim}
data LogEntry = LogEntry {  
    name :: String,
    oldSalary :: Float,
    newSalary :: Float  
}
\end{verbatim}

A knowledgeable Haskell programmer may immediately suggest the use of the \texttt{writer monad} for logging. However, let us start with a beginner’s implementation. When learning about monads, a non-monadic implementation is definitely helpful. Consider, the following function for cutting salaries at the department level, with logging integrated:

\begin{verbatim}
cutD :: Department \rightarrow (Department, Log) 
cutD (Department n m ds es) = (Department n m' ds' es', log)
where
    -- Cut the manager’s salary
    (m', log) = cutE m
    -- Cut all salaries in the sub-departments
    (ds', log1) = unzip (map cutD ds)
    -- Cut all salaries of all immediate employees
    (es', log2) = unzip (map cutE es)
    -- Compose intermediate logs
    log = concat (log1) ++ log2 ++ log3
\end{verbatim}

Unfortunately, the need for logging affects the overall processing of departments; see the occurrences of the idioms for grouping and unzipping in the case of lists.

2nd Haskell illustration  
(Contribution \texttt{haskellWriter})

A particular \texttt{monad} \[49\], the \texttt{writer monad}, can be used to encapsulate logging. Functionality involving salary updates is to be converted to monadic style. We revise Contribution \texttt{haskellLogging} as follows:

\begin{verbatim}
cutD (Department n m ds es) = 
do
    m' <- cutE m
    ds' <- mapM cutD ds
    es' <- mapM cutE es
return (Department n m' ds' es')
\end{verbatim}

The function is oblivious to logging; it is prepared for any effect.

4. A chrestomathic ontology

A chrestomathy breaks down into physical entities (contributions, individual source files, fragments thereof, and source code illustrations other than contributions) and conceptual entities (languages, technologies, concepts, features, and others). These entities engage in certain ontological relationships, as we discuss in this section. In the case of 101, all documentation is manifest on the 101wiki. Thus, the ontology is modeled and maintained through a so-called Semantic Wiki \[9\]. The ontology can also be accessed programmatically through an RDF-based triple store.

4.1 Entity types

101 aggregates knowledge about entities of the following types:

- **Contributor**: People who submit and maintain contributions.
- **Feature**: Tasks to be implemented by contributions.
- **Language**: Software languages used by the contributions.
- **Technology**: Software technologies used by the contributions.
- **Concept**: Software concepts exercised by the contributions.
- **Theme**: Sets of contributions covering specific topics.

The types serve as namespaces on the wiki, while they correspond to proper types in the underlying RDF representation.

4.2 Metadata

Entities (say, resources in the sense of RDF) can be associated through triples with appropriate predicates. We also refer to such triples as (semantic) metadata.

Metadata for contributions

- **Contribution \texttt{developedBy}**: Contributor: a contribution, e.g., \texttt{Contribution \texttt{haskellHxt}} was developed by a contributor, e.g., \texttt{Contributor \texttt{Thomas Schmorleiz}}.
- **Contribution \texttt{implements}** Feature: a contribution, e.g., \texttt{Contribution \texttt{haskellHxt}} implements a feature, e.g., \texttt{Feature \texttt{Cut}}.
- **Contribution \texttt{uses}** Language: a contribution, e.g., \texttt{Contribution \texttt{haskellHxt}} uses a language, e.g., Language \texttt{XML}. All \texttt{101haskell contributions} use Language \texttt{Haskell}.
- **Contribution \texttt{uses}** Technology: a contribution, e.g., \texttt{Contribution \texttt{haskellHxt}} uses a technology, e.g., Technology \texttt{Parsec}. All \texttt{101haskell contributions} use Technology \texttt{GHC}, Technology \texttt{Cabal}, and Technology \texttt{HUnit}.
- **Contribution \texttt{emphasizes}** Concept: a contribution, emphasizes the relevance of some concept. For instance, \texttt{Contribution \texttt{haskellWriter}} emphasizes Concept \texttt{Writer monad}. Such triples are synthesized from mentions of concepts in the section motivating the chrestomathic purpose.
- **Contribution \texttt{relatesTo}** Contribution: a contribution, relates to another contribution in terms of reuse. For instance, \texttt{Contribution \texttt{haskellWriter}} was derived from Contribution \texttt{haskellLogging} by setting up monadic style. Such triples are synthesized from mentions of contributions in the section explaining relationships between contributions.

General metadata

- **Concept isA Concept**: a concept, such as Concept \texttt{Functional programming language} is a specialization of a concept, such as Concept \texttt{Programming language}.
- **Entity \texttt{instanceOf}** Entity: an entity, such as Language \texttt{Haskell} is an instance of an entity, such as Concept \texttt{Functional programming language}. Also, Feature \texttt{Cut} is an instance of Concept \texttt{Functional requirement}.
- **Entity \texttt{partOf}** Entity: an entity, such as Technology \texttt{GHC} is part of another entity, such as Technology \texttt{Haskell Platform}.
- **Entity mentions** Entity: an entity, such as Contribution \texttt{haskellWriter} mentions another entity, such as Concept \texttt{Writer monad}, which in turn mentions Concept \texttt{Output}. Such triples are synthesized from ordinary links on the wiki pages.

Consider Figure 1 for an illustration: declared (but not synthesized) metadata is shown for Contribution \texttt{haskellWriter}. 

29
### Metadata triples

- this `developedBy` Contributor: Ralf Lmme1
- this `developedBy` Contributor: Thomas Schmorleiz
- this `implements` Feature: Cut
- this `implements` Feature: Hierarchical company
- this `implements` Feature: Logging
- this `memberOf` Theme: Haskell introduction
- this `memberOf` Theme: Haskell potpourri
- this `uses` Language: Haskell
- this `uses` Technology: Cabal
- this `uses` Technology: GHC
- this `uses` Technology: HUnit

![Figure 1. Metadata triples with `haskellWriter` as the subject (abbreviated as ‘this’) and other entities as `objects`.](image)

**Themes of contributions** In Figure 1, the ‘member-of’ predicate is used for assembling themes, i.e., sets of contributions. Haskell-centric themes collect 101haskell contributions that address a specific topic (say, stakeholder), e.g., introduction to Haskell or generic functional programming; see Figure 2 for a theme. 101haskell contributions also participate in themes that are not Haskell-specific, thereby demonstrating the ‘Haskell way’ of addressing some problem, e.g., parsing or web programming.

#### 4.3 References to external resources

The 101wiki also links to external resources such as Wikipedia, DBpedia, or HaskellWiki. The idea is here to make good use of external knowledge resources rather than reproducing much content on the 101wiki. Such references are authored again as triples in the metadata section of an entity’s page. The references are to be qualified by an appropriate predicate to express the degree of semantical similarity between the internal and external resources.

- **Entity sameAs URL**: The 101wiki and the other resource are assumed to be semantically identical. For instance, the 101wiki uses ‘Zipper’ in the same sense as Wikipedia.[7]
- **Entity similarTo URL**: The resources are semantically similar.[7] For instance, the 101wiki uses ‘equality’ in close reference to programming, whereas Wikipedia associates equality primarily with mathematics.[4]
- **Entity linksTo URL**: Any relationship weaker than ‘similarTo’.

---

### A 101wiki paragraph from `haskellSyb`

Several of the operations on companies can be implemented in a very concise manner based on the SYB style of generic programming. For instance, the operation for totaling salaries simply extracts all floats from the given term and reduces them by addition:

```haskell
{-# Total all salaries in a company

total :: Company -> Float

total = everything (+) (extend (const 0) id)
```

![Figure 3. The wiki paragraph contains a code fragment from which one can navigate right away to the relevant code in the repository; see the ‘Explore’ button. The underlying markup specifies the location of the file ‘src/Company/Total.hs’, the syntactic category ‘pattern’ of the fragment, and the name ‘total’.](image)

#### 4.4 References to code fragments

The illustration section of a contribution tends to include code fragments from the actual contributions—very much like we did in[3] Copying and pasting fragments into wiki pages would imply two drawbacks. First, the code on the page may go out sync with the code in the 101repo. Second, the wiki user would be insufficiently supported in navigating from the wiki to the repo. Thus, we extend a resource-centric approach such that the wiki may refer to fragments in the repo. Consider Figure 3 for an illustration.

These references rely on a URI scheme (as illustrated in the figure), a language-specific fragment locator (to look up the relevant code fragment), and a language-specific renderer. In this manner, source code (repo) and documentation (wiki) may evolve in a loosely coupled manner with possibly different authors for both artifacts, while being still effectively connected, thereby supporting navigation by wiki users. We note that the illustration on the wiki must not be confused with regular code documentation. That is, the illustration serves for highlighting a contribution’s characteristics with regard to the chrestomathy, as opposed to the comprehensive documentation of a given contribution as an individual software system. Thus, classic techniques, such as literate programming[25], are not applicable. We need to enable loose coupling for wiki and repo.

### 5. Chrestomathic knowledge integration

We speak of knowledge integration here in the sense of the process that leads to the consistent interlinkage of the chrestomathy’s documentation (on the 101wiki) and appropriate external resources such as web pages on external wiki pages or textbook paragraphs.

Consider Figure 4 for an illustration based on the programming concept `Monad`. The concept is reified on the 101wiki. The chrestomathy contains contributions that exercise monads, see[3,5] there are also additional small examples in the 101repo. The 101wiki links to Wikipedia, HaskellWiki, and two online textbooks[24,59]. The 101wiki page for monads is not meant to be comprehensive on the subject, but it links to textbook chapters for additional information about monads. For instance, several chapters of[59] are linked to: Monads. Monad transformers. Programming with monads. Error handling. The Parsec parsing library.

In the present section, we explain the semi-automatic, primarily textbook-driven process for knowledge integration, as it was ap-
Monads in Haskell can be thought of as composable computation descriptions. ... monad
Concept: Maybe monad
Concept: Writer monad
Contribution: haskellLogging
Contribution: haskellWriter

Figure 4. Illustration of knowledge integration for the ‘Monad’ concept according to different knowledge resources.

plied to 101haskell. This process yields terms to be covered on the 101wiki and links to wikis and textbooks, subject to text mining and summarization techniques. The process is supported by the designated 101integrate framework which helps with processing (textbook) resources for the raw vocabulary extraction, mapping ‘raw’ terms to ‘consolidated’ terms used on the wiki, and linking terms back to a resource’s paragraphs in the interactive experience. A deliberate limitation of our approach is its assumption of a vocabulary with a manageable number of terms—amenable to human-based validation and content enrichment.

The rest of the section is structured as follows. First, we briefly explain the initial task of textbook selection. Second, we explain the largely automatic process of term extraction providing us with ‘interesting’ textbook terms. Third, we explain the less automated aspect of vocabulary consolidation aiming at reification of textbook terms on the 101wiki, also subject to linking to Wikipedia or other authoritative resources. Finally, we explain vocabulary monitoring meant to help with assessing and improving coverage/interlinkage of the vocabulary across the 101wiki.

5.1 Selection of textbooks

We selected four popular textbooks on functional programming in Haskell [21, 34, 39, 46]. These books, when combined, cover introductory topics, but also more advanced topics such as functors, monads, or testing. Two of the books, [21, 39], are available online with open access so that these books can be linked to the wiki. That is, 101wiki may link to specific chapters, sections, or paragraphs in those books.

The other two books, [21, 39], make valuable vocabulary contributions. The resource [21] is an established introductory text. The resource [46] is also an established text on Haskell; its mathematical or logical approach complements other texts. The textbook authors provided us with access and allowed us to publish the extracted vocabularies.

5.2 Term extraction

Index and content normalization The textbooks are normalized (cleaned up) upfront. The rest of the extraction process is completely uniform for all books. The index and the raw content are extracted from each book, while performing resource-specific data cleaning steps so that formatting markup is eliminated. Source code was also excluded to eliminate related mining challenges.

The raw index terms and the raw content are subsequently stemmed. As a result, one obtains a normalized set of raw terms that is free of trivial redundancy. Subsequently, the list of raw terms is reduced automatically by applying a threshold rank for ‘common English’ The threshold was defined manually by inspection of the raw terms sorted by rank. This process led to approximately 2000 index entries for the four books combined. Common English was also removed from the raw content.

Mining of candidate terms All index entries are matched with all the content of all books. Match counts for the index entries are broken down for the book chapters. Based on these counts, candidate terms are determined by text mining and summarization, as explained in a second. Candidate terms are either chapter terms, popular terms, or title terms, as explained below. Candidate terms are those terms per book and per chapter that are matched ‘most frequently’ in a chapter (say, the most frequent five terms per chapter), that also appear ‘often enough’ (say, at least three times) in one or more chapters of the book, but only in ‘few’ chapters (say, in 25% of the chapters or less). This is a variation on

10 We use the Natural Language Toolkit (NLTK [7]).
11 We use http://www.wordcount.org for “common English”.


dt=11

some terms are not selected; instead, locally frequent terms without many scattered occurrences are selected. Figure 5 shows a good part of the chapter terms for one of the Haskell textbooks. There is one row per term and one column per chapter. A bullet in a cell associates a term with a chapter. The size of the bullet represents matching frequency relative to other chapter terms.

Popular terms are those frequently matched technical terms that are actually scattered over many chapters; think of ‘function’, for example. Such terms cannot be identified by inverse document frequency. Instead, we pick popular terms per book by ordering matched terms by frequency, excluding terms that are already covered by the chapter terms, and selecting all the more popular terms up to a threshold (such as the terms with >10 % of topmost term’s frequency). It turns out 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. Only very few terms were added in this manner; see Table 1.

Activation of sub-vocabularies All terms are to be organized in a simple taxonomy of sub-vocabularies, thereby helping to understand the consolidated vocabulary. These are some of the sub-vocabularies used for the Haskell textbooks:

- 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 `list`.
- Programming theory: Concepts associated with mathematical or formal treatment of programs, e.g., induction.

At this stage, with a consolidated vocabulary, we can compare the textbooks in terms of the contributions and the description of features should make use of the vocabulary. In this manner, we would be able to claim that the chrestomathy covers the textbooks and is interlinked with these knowledge resources. Therefore, we monitor coverage and interlinkage, as described below.

### Table 1. Numbers of candidate terms

<table>
<thead>
<tr>
<th>Book</th>
<th>Chapter terms</th>
<th>Title terms</th>
<th>Popular terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRAFT</td>
<td>52/552</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>PIH</td>
<td>25/31</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>RWHR</td>
<td>65/72</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>LYaH</td>
<td>34/38</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

### Term validation

Chapter terms and popular terms are validated to exclude ‘uninteresting’ terms. The idea is here to focus on terms that concern functional programming, Haskell programming, or generally programming. A term could end up being uninteresting because it relates to a specific example in the book, which is not worth interlinking. For instance, the term ‘picture’ appears frequently in 46 for the sake of a specific example. Also, the application of a stop list for ‘common English’ may have missed terms that are ‘common’ in the specific technical text at hand. For instance, the term ‘model’ is used frequently in 46 in the style of “We can model a tournament by this type definition”.

The discussed thresholds can clearly be used to control the number of candidate terms. As we mentioned before, it is important to aim at a manageable number because of the subsequent steps involving human intervention. Table 1 summarizes the numbers for the Haskell textbooks; the first column also shows the difference between chapter terms before and after validation.

### 5.3 Vocabulary consolidation

At this stage, the overall objective is to map candidate terms from the books to a consolidated vocabulary on the 101wiki.

### Term mapping and reification

We systematically process each candidate term that is not already present on the 101wiki as follows. If the term goes by a different name on the 101wiki, then we record this correspondence so that an accordingly aggregated mapping can be used for interlinking wiki terms and resources. For instance, the term ‘class’ appears as a chapter term of 46’s chapter ‘Overloading type classes and type checking’. The term ‘class’ would be overly ambiguous in a broader context of programming, which is the context assumed by 101. Thus, we map ‘class’ to ‘type class’ on the wiki.

The remaining case is when we face a candidate term that is not yet present on the wiki. (Most of the textbook terms were initially missing, indeed.) In this case, we need to reify (‘make existent’) the term on the wiki. To this end, we locate the term on Wikipedia or HaskellWiki. In this manner, each term on the 101wiki is linked to yet another relatively standardized and stable URL (URL). Following [21] we aim at locating a Wikipedia and/or HaskellWiki page for a concept that is the ‘same as’ the emerging concept on the 101wiki.

### Figure 6. The derived Haskell vocabulary

As for the textbooks at hand, the most popular vocabularies are (in decreasing order) Programming, Data, and Functional programming. The remaining vocabularies are considerably less frequent. In particular, the Haskell vocabulary was only populated by a few concepts; see Figure 6. This means that the books do not operate at a Haskell-specific abstraction level.

### Comparison of the resources

At this stage, with a consolidated vocabulary, we can compare the textbooks in terms of the contributing vocabularies. In particular, we review the terms uniquely contributed by each textbook; see Figure 7. At the bottom of the figure, all remaining (‘non-unique’) terms are listed. We observe that each book makes a contribution to the consolidated vocabulary. 46 contributes terms related profoundly to formal or mathematical areas of functional programming such as ‘Proof’ and ‘Calculation’. 21 contributes the fewest terms and much of them are concerned with basic functional programming concepts such as ‘Function application’ and ‘Function definition’. 39 contributes the most terms, overall, and it mentions several technologies; the other books do not. 13 contributes terms related to advanced functional programming concepts, such as zippers and applicative functors.

### 5.4 Monitoring vocabulary usage

Subject to appropriate metrics, all terms should be ‘reasonably’ referenced by the 101wiki. In particular, the documentation of contributions and the description of features should make use of the vocabulary. In this manner, we would be able to claim that the chrestomathy covers the textbooks and is interlinked with these knowledge resources. Therefore, we monitor coverage and interlinkage, as described below.
Figure 7. Comparison of the different Haskell textbooks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Anonymous function (2/10)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Map function (2/10)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Parsing (2/8)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Recursion (2/2)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Fold function (2/1)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Parser combinator (2/1)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Function application (1/1)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Lambda abstraction (1/1)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>String (1/1)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Type class (1/1)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Type class instance (1/0)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>Function (0/5)</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>

Figure 8. Vocabulary usage for Haskell script, too generic term, Equation, Function application, Parser combinator, Identity element, Declaration, Function definition, Product function, Lambda abstraction

| Terms in [46] 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 [21] only: Haskell script, too generic term, Equation, Function application, Parser combinator, Identity element, Declaration, Function definition, Product function, Lambda abstraction |
| Terms in [39] only: Foreign function interface, Predicate, Operator precedence, Polymorphism, Thread, Performance, Var, 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 more than one book: Monad, 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: GHC, Text, Tail, Regular expression, Map function, Language: Haskell, Induction, Function, Pattern matching, Prelude, Stack, Eager evaluation, List, Maybe type, Monad, Module, Guard, Boolean, File |

6. A chrestomathy-based course

101haskell, as described in the present paper, was used in an introductory functional programming course during summer semester 2013 at the authors’ university, (2/3 of the students were in the second semester and had already basic Java programming skills. The remaining students were in the first semester.) The present section describes the underlying teaching concept, motivates designated course content on top of 101haskell, and discusses a limited course evaluation.

6.1 Teaching concept

We highlight aspects that set apart the present teaching concept from common practice. These aspects relate to the use of the 101haskell and infrastructure of 101.

“Favor live programming.” Most of the lecture time is dedicated to live programming, where all relevant concepts are systematically illustrated. The list of concepts for each lecture is published on the 101wiki. The illustrations given during live programming are essentially variations on the illustrations readily available on the wiki. More complex examples, such as non-trivial 101haskell contributions, are not developed from scratch. They are readily demonstrated, as available from the 101repo. Slides are not used. Some amount of wiki content may be projected, though. Also, 101wiki pages may contain embedded media.

“Embrace multiple external resources.” Past teaching experience has suggested that our students are hardly willing to follow given textbook recommendations; instead, unstructured search is popular. In this course, we respond to this attitude by helping the students leverage available online resources more systematically. In particular, Wikipedia, HaskellWiki, and Haskell textbooks are readily linked from the course material, as discussed in 5.

“Complement the running example.” The lectures spend considerable time on explaining all concepts with the help of diverse, basic examples which are unrelated to the 101system. Even these examples are available through the 101repo. Implementations of the 101system serve typically as less basic illustrations. The homework assignments are not necessarily tied to the 101system. Occasionally, an assignment could be concerned with the modification of a given contribution.

“Open source and open linked data.” Absolutely all course material is open. Reuse in courses and collaborative advancement is appreciated and straightforward. In particular, reuse does not cause any copyright issues whatsoever because lecturers may reuse wiki content and repo content simply by linking to it, without copy-and-paste as needed for slide-based reuse.

6.2 Course content

Most of the content is readily available via the wiki pages for contributions, concepts, and others. The only course-specific content is the lineup of all lectures and per-lecture scripts for the itemized and linked content of the lectures.

Figure 9 shows the lineup of the lectures for the course. Two lecture slots are repurposed for the final exam and its dry-run. In the next edition of the course, we expect to make space for an extra lecture slot, in which case we plan to cover functional data structures as an additional topic.

Figure 10 shows a particular lecture script, as it is rendered on the 101wiki. Thus, each lecture comes with a headline (a title), a summary, a longer list of concepts, and a shorter list of 101haskell contributions covered by the lecture. This also clarifies the modus operandi of the lecturer: the listed concepts are illustrated in some order; the listed contributions are eventually explored. The exact order is unspecified; it may be influenced by the dynamics of the lecture. If time turns out to be insufficient, some concepts or contributions may also be delegated to the lab.

6.3 Evaluation

The authors’ university runs evaluations for all courses. However, student participation in the polls is voluntary. The questionnaires are relatively complex, which may add to the low turnout. 10 out of 73 enrolled students submitted their scores for the functional programming course. All results are available from this paper’s website. The authors’ experience with other introductory courses (first or second semester) suggests that these courses tend to be less well received. (A significant percentage of students cancels their studies during this period; there is no ‘numerus clausus’ for computer science.) The present course received mostly favorable scores. The course received an overall score of 2.3 (‘good’) on a 1-5 (very good to insufficient) scale.

In Figure 11 we show poll results for a question related to the use of practical examples. We take the results to mean that the balanced use of the 101system as the running example was appreciated.

The written final contained basic tasks for the first seven lecture topics of Figure 9 and almost all the students succeeded in the exam. (This is rather surprising for a first/second semester course.) As we have not held the course previously, we cannot compare learning results.

7. Related work

We set up a scope for the related work discussion. The present paper is concerned with aggregating, organizing, accessing, and maintaining knowledge in the programming domain, to be useful, specifically, for learning. Our approach is worked out well in the functional programming context.

**MOOCs** Massive open online courses (MOOCs [35][38]) provide a ‘content delivery model’, which optimizes the aspects of accessibility of course content and group experience. Our research focuses instead on ontology development and knowledge integration.

**E-Learning** Semantic enrichment of teaching content, is also an established idea in E-learning [31][32]. Our approach makes an original contribution in so far that external resources and chrestomathy-based source-code examples are integrated as well.
The domain model of courses The functional programming community has a track record on reflecting on the ‘functional approach’ towards CS education [10, 16, 23, 49]. In addition to motivational aspects, the underlying ‘domain model’ (e.g., ‘soccer’) is often an important consideration [1]. 101haskell assumes the domain model of an ‘information system’, which is not, in any way, specific to any programming paradigm. Instead, the feature model is designed to superficially, but broadly, cover programming concepts, techniques, and technologies.

Collaborative Wikis Wikis are commonly used to create collaborative knowledge. Science Online [17], as a collaborative knowledge construction platform, connects collective models of knowledge production and learning. The approach advocates learning by working on personally meaningful projects. A particular idiom of collaboration is worth mentioning. That is, Science Online assumes a bibliographic extension to manage references to external resources through extra pages, thereby also motivating a centralized discussion of such linked resources.

Collaborative E-learning As a ‘curriculum execution model’, collaborative E-learning aims to complement traditional offline approaches for courses with online techniques. The approach of [13] combines collaborative forums and wiki technologies to motivate students’ engagement, better coordination and progress monitoring. Students are encouraged to produce new material based on the discussion. Empirical results indicate better performance among active participants. 101 is a community project which stimulates ‘contributors’ to submit initial content and to participate in knowledge integration while being guided by ‘gatekeepers’.

Knowledge integration We assume the following definition of knowledge integration: “Knowledge integration is the dynamic process of linking, connecting, distinguishing, organizing and, structuring models of scientific phenomena” [13]. Knowledge integration environments (KIEs) have been proposed [6] to join tools, networks, people into a single learning and instructing experience with social support. ‘Tried-and-true’ methods, such as books, are complemented with online resources. Different ways to define knowledge and knowledge structures in software development have been proposed [42] with the notion of ‘schema’ – pre-organized knowledge concepts which describe ‘specific links among various knowledge elements’. Such schemas are implemented in CASE tools for software comprehension. In [11], Linked Data [29] principles are applied to a knowledge integration challenge, i.e., to address the challenge of integrating the fragmented landscape of technology-enhanced learning (TEL) repositories to provide “rich and well-interlinked data for the educational domain”. The project operates on the existing TEL resources, addressing the heterogeneity of APIs, schemas, and response message formats. Knowledge integration in 101 also surfaces all underlying entities (resources) and links as Linked Data.

Vocabulary engineering We were inspired by the established notion of reviewing domain literature and the reuse of existing large-scale categories (taxonomies) such as Wikipedia in the context of taxonomy development [37, 40]. When compared to rich related work on vocabulary mining [45], we aim at the informal (hence, semi-automated) consultation of multiple technical, readily indexed textbooks for the sake of deriving a consolidated and manageable vocabulary with confirmed links to key sources such as Wikipedia. Vocabulary mapping is established in the context of ontology matching [12, 38] with vocabularies of substantial size that may need to be matched largely automatically. In our context, the size needs to be limited to allow for human intervention.

Ontologies Ontologies are applied in many ways in software engineering [3]. Ontologies serve, for example, for distributed software development [51], software testing [4], educational content modeling [8]. Ontologies help with understanding the domain and sharing as well as reuse of knowledge. In the same sense, the knowledge domain of 101 can be said to be programming.

8. Conclusion

Summary This paper described the foundations of a software chrestomathy with an attached ontology for knowledge organization and a customized, semi-automatic process for knowledge integration, in fact, vocabulary engineering tailored towards producing a manageable number of interesting terms to be used for metadata in documentation. The foundations are realized for 101haskell—the Haskell-specific sub-chrestomathy of the more general 101 project. A functional programming course has been developed on top of the chrestomathy. Our experiences substantiate that an appropriately organized and enriched software chrestomathy can be very useful for learning (and teaching).

Call to arms 101 and 101haskell are always work in progress. 101 is a community project which supports open source and open Linked Data. New contributors and users are welcome, of course.

Future work The presented ontology may be extended along several dimensions. For instance, we should model more precise relationships between contributions, e.g., in the sense of ‘refinement’, ‘translation to another language’, ‘design variation’, or ‘technology variation’. When combined with feature-detection support, we may be able to help users understand the commonalities of the contributions as well as the differences.

101integrate, the framework underlying [5] needs to be further matured and generalized so that vocabulary extraction and the overall process of knowledge integration is more straightforward.

While we have designed means to monitor vocabulary usage including metrics that define targets for improvements, we lack process support for such improvement. We think of a form of issue tracking to guide the work of a designated vocabulary engineer. A similar challenge applies to ontology engineering in the sense of maintaining the classification hierarchies of 101 according to some metrics.

We also plan to research opportunities for integrating community resources such as GitHub and StackOverflow, which represent important knowledge resources on programming and software engineering. For instance, in the case of StackOverflow, this would be ideally a bidirectional relationship between 101 and StackOverflow resources; in one direction to enrich 101 entities with StackOverflow answers; in the other direction to associate StackOverflow questions with relevant 101 illustrations.

Acknowledgment

We are grateful to several people (fellow ‘101ers’) who helped on the technical part of this work—specifically Kevin Klein, Oleixy Lashyn, Martin Leinberger, Sebastian Jackel, and Arkadi Schmidt. Further, we are grateful to Graham Hutton and Simon Thompson for sharing the sources of the Haskell books [21, 46] with us for the purpose of this research.

References
