Towards Tool Support for History Annotations in Similarity Management

Extended Abstract

Thomas Schmorleiz and Ralf Lämmel

Software Languages Team, Department of Computer Science, University of Koblenz-Landau, Germany

Abstract

When a system is needed in different variants to meet different requirements, then some form of product line engineering may need to be used. In practice, it is often preferred to develop the variants in a loosely coupled fashion as opposed to the regime of a proper ('explicit') product line from which to derive variants by some generative mechanism. For instance, the 101haskell chrestomathy (a subchrestomathy of 101) contains many similar, small, Haskell-based systems that are indeed maintained in loosely coupled fashion. In previous work, we and collaborators have proposed an approach to manage such loosely coupled variants by using a *virtual platform* and cloning-related operators. In this extended abstract, we sketch a concrete method with a supporting tool, Ann, for exploring the similarity of variants and annotating them with metadata accordingly. As a direct result, a *propagate* operator is enabled to automatically propagate changes across variants and to synthesize a to-do list for remaining manual actions. We sketch the method and the tool's application in an ongoing case study for capturing and improving the similarity of the Haskell-based variants of the 101haskell chrestomathy.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features; D.2.7 [*SOFTWARE ENGINEERING*]: Distribution, Maintenance, and Enhancement; F.3.2 [*LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages

Keywords Haskell. Software Product Line Engineering. Variability Management. Virtual Platform. Ann.

Acknowledgement

The presented work continues previous joint work [1] with Michal Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Stefan Stanciulescu, Andrzej Wasowski, and Ina Schaefer. The work is also inspired by Julia Rubin's framework for clone management [6].

1. Motivation and background

The corpus of the 101companies project [2] (or just '101') holds a set of variants ('contributions'), all implementing a common feature model. Many of these variants share implementations of some features because their conceptual contribution focuses on the implementation of other features. Thus, cloning of feature implementations is often performed to start the implementation of new variants. While this practice is reasonable in itself, it makes it too hard to understand the similarities of the variants; it also makes it too easy for variants to diverge from each other over time unintentionally. Thus, a form of similarity management is needed. This problem was set up as a challenge for software chrestomathies in [3] and it is, in fact, a challenge in software product line engineering [1].

We have developed a method with a supporting tool, Ann, for exploring the similarity of variants and annotating them with metadata accordingly. This work is directly based on the idea of virtual platform for software product line engineering, as presented in [1]. Our objective is not just to provide the user with information about similarities in a corpus of variants, but also to enable the automatic propagation of changes across variants, and, finally also to reduce overall complexity and unintentional divergence within the corpus. We sketch the method and tool's application in an ongoing case study for capturing and improving the similarity of 101's Haskell segment, i.e., 101haskell [4].

2. A method for variability management

We describe the method by a series of key notions.

2.1 Fragment

We examine similarity of variants and their source-code units (files) at a fragment level. Here is one possible (informal) definition of the fragment notion. That is, a fragment is a range of consecutive lines of source code that correspond to a 'major' node in the associated abstract syntax tree (AST). We assume that syntactic categories for forms of (named) abstractions are favored. Each fragment is identified by a classifier and a name. Classifiers correspond to the syntactic category at hand. For instance, in the case of Haskell, classifiers are 'data', 'type', or 'function'; names are those of data types, type synonyms, or functions.

2.2 Similarity

We compare fragments by adopting an existing approach for detecting near-miss intentional clones [5]. In particular, we pretty-print the source code in a regular manner to lay out compound constructs over several lines so that a simple text-based measure, the diff ratio, can be used for comparison. A similarity measure of '1' means equality. We lift similarity from the fragment level to the levels of

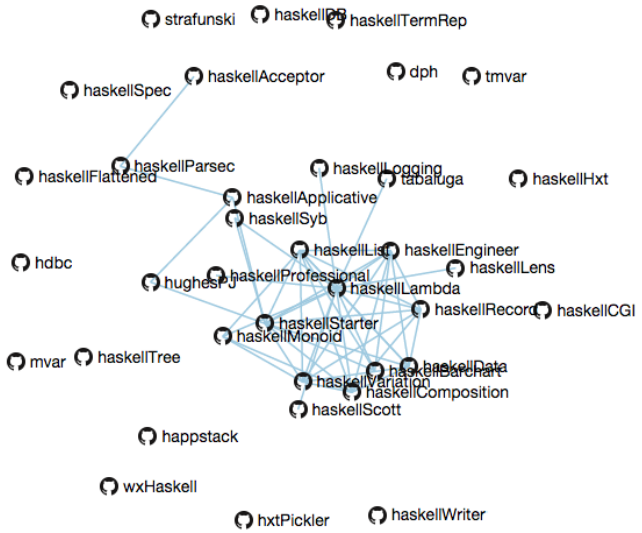

Figure 1. Visualization of similarity across the variants of 101haskell

files, folders, and variants by averaging similarities in a straightforward manner. Similarities are stored as triples of two fragment identifiers with the computed diff ratio.

For illustration, consider Figure 1, which shows the Haskell-based variants of the chrestomathy 101 (i.e., 101haskell [4]). The edges indicate similarity of variants above a certain threshold. The view is computed by the Ann tool.

2.3 Similarity evolution

We track the evolution of similarities between fragments throughout the commit history. To this end, we also need to track the fragments themselves—with regard to variant, file, and fragment renaming. We are specifically interested in two points on the timeline: the point at which a similarity was detected and the current state. Each similarity evolution can be classified according to these *types of evolution*:

Always equal The similarity was always 1.

Eventually equal The similarity was initially below 1, but it is 1 eventually.

Eventually unequal The similarity was initially 1, but it is below 1 eventually.

Always unequal The similarity was always below 1.

2.4 Annotation

Each similarity can be annotated to express the intended treatment of the similarity along evolution. That is, an annotation states how a similarity should be maintained through automated or manual actions. The annotations themselves are to be attached manually, even though defaults may be reasonably assigned in certain cases. The idea is that the user sees the similarities in the order of decreasing similarity (diff ratio), thereby prioritizing annotation of the most similar fragments. There are these *types of annotations*:

Maintain equality An equality at hand should be maintained. This can be done by merging any changes from one fragment to the other, by automated three-way-merge, or possibly by manual conflict resolution.

Maintain similarity A similarity, which is not an equality, should be maintained. A manual action is required, when the similarity (the diff ratio) decreases.

Restore equality A similarity, which is not an equality, should be turned into an equality. In a simple case, either of two fragments may be selected to override the other. It may also be necessary though to change both fragments towards an equal fragment through a manual action.

Increase similarity A similarity, which is not an equality, should be increased, based on the insight that equality is not feasible, while increased similarity is feasible. A manual action is required here.

Ignore similarity The similarity is to be ignored in that it is not reported anymore, when following the diff ratio order.

Whether or not a certain annotation is applicable to a similarity also depends on the type of evolution. For instance, the evolution type 'eventually unequal' cannot be combined with the annotation type 'maintain equality', but it can only be combined with 'restore equality' or all the other types.

2.5 Propagation

Given a repository in a new state with some previously attached similarity annotations, we can determine any sort of similarities that have arrived, increased, decreased, or vanished (assuming some threshold) and whether they have to be restored or increased. Some of these case of similarity evolution may be addressed automatically; others give rise to a to-do list to be addressed by the developer. This semantics is captured as a 'propagate' operator of our method, as adopted from the general approach described in [1].

We are currently working on the propagate operator and its integration into the workflow of the underlying version control system, which is *git* in the case of 101haskell. Overall, the idea is that *git* commands such as *commit*, *pull*, and *push* are enriched by the propagation semantics.

3. The Ann tool

Ann is an interactive tool supporting the exploration of the version history and the set of variants down to the level of folders, files, and fragments. The tool assumes that the variants are maintained by a version control system. In the case of 101, we use *git*.

Figure 2 shows the annotation view of Ann: we are in the context of a specific similarity, namely a fragment shared by two variants. We have decided to annotate the similarity with 'maintain equality' so that automatic propagation of changes would be enabled.

Figure 3 shows all variants of 101haskell on the timeline of the version history. One can study the commits to get quick access to affected variants and files as well as the associated similarities.

Figure 4 gives an impression of the variant-centric dimension of exploration. A variant is picked in the beginning (*haskellEngineer* in the figure). The most similar variants are shown. One can dive into the picked variant to select specific folders or files. One can then further dive into files to eventually annotate similarities at the fragment level.

4. State of research

We have developed all extraction tools that Ann depends on. We have further implemented all essential views, as also illustrated in this extended abstract including two dimensions of annotation: variant-centric versus commit-centric. We have refined the user interface in several iterations since the amount of data to be processed

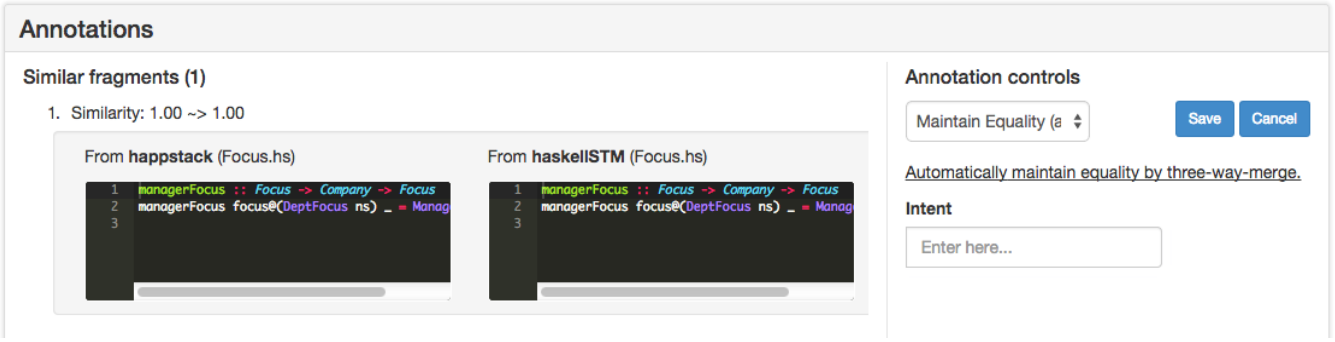

Figure 2. Annotation of a similarity

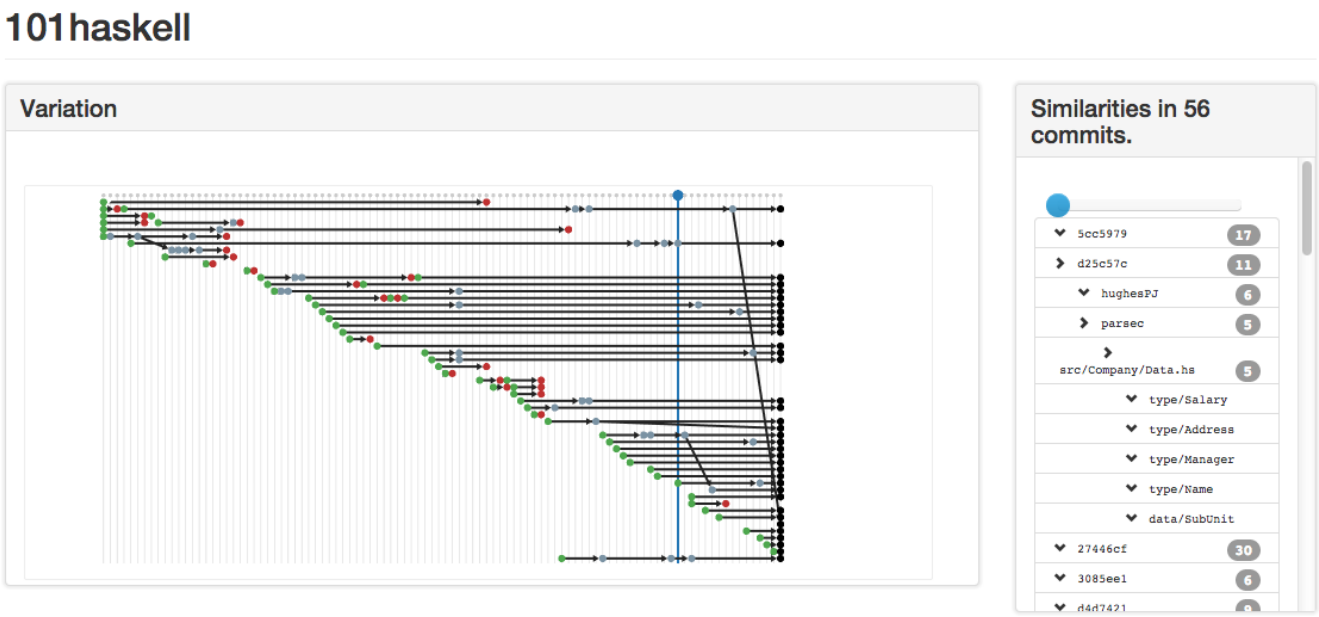

Figure 3. Variations of 101haskell

by the user was initially too large. The iterations applied good user experience (UX) principles to the design of Ann.

The tool already helped us to identify a set of variants that are conceptually or technologically outdated or unnecessarily disconnected (in terms of similarity) to other variants. The next step is to integrate the propagation operator into a git workflow by extending existing git commands and implementing new commands. After that we will aim at improving the similarity across 101haskell by bringing up, for example, the degree of equal fragments in response to the erosion (divergence) that has happened over time, when we had no tool support like Ann.

References

[1] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stanciulescu, A. Wasowski, and I. Schaefer. Flexible product line engineering with a virtual platform. In *Proc. of ICSE 2014*, pages 532–535. ACM, 2014.

[2] J.-M. Favre, R. Lämmel, T. Schmorleiz, and A. Varanovich. 101companies: A Community Project on Software Technologies and Software Languages. In *Proc. of TOOLS 2012*, volume 7304 of *LNCS*, pages 58–74. Springer, 2012.

[3] R. Lämmel. Software chrestomathies. *Sci. Comput. Program.*, 2013. In press.

[4] R. Lämmel, T. Schmorleiz, and A. Varanovich. The 101haskell Chrestomathy—A Whole Bunch of Learnable Lambdas. In *Postproceedings of IFL 2013*, 2014. 12 pages. To appear in ACM DL. Available online `http://softlang.uni-koblenz.de/101haskell/`.

[5] C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. of ICPC 2008*, pages 172–181. IEEE, 2008.

[6] J. Rubin and M. Chechik. A framework for managing cloned product variants. In *Proc. ICSE 2013*, pages 1233–1236. IEEE / ACM, 2013.

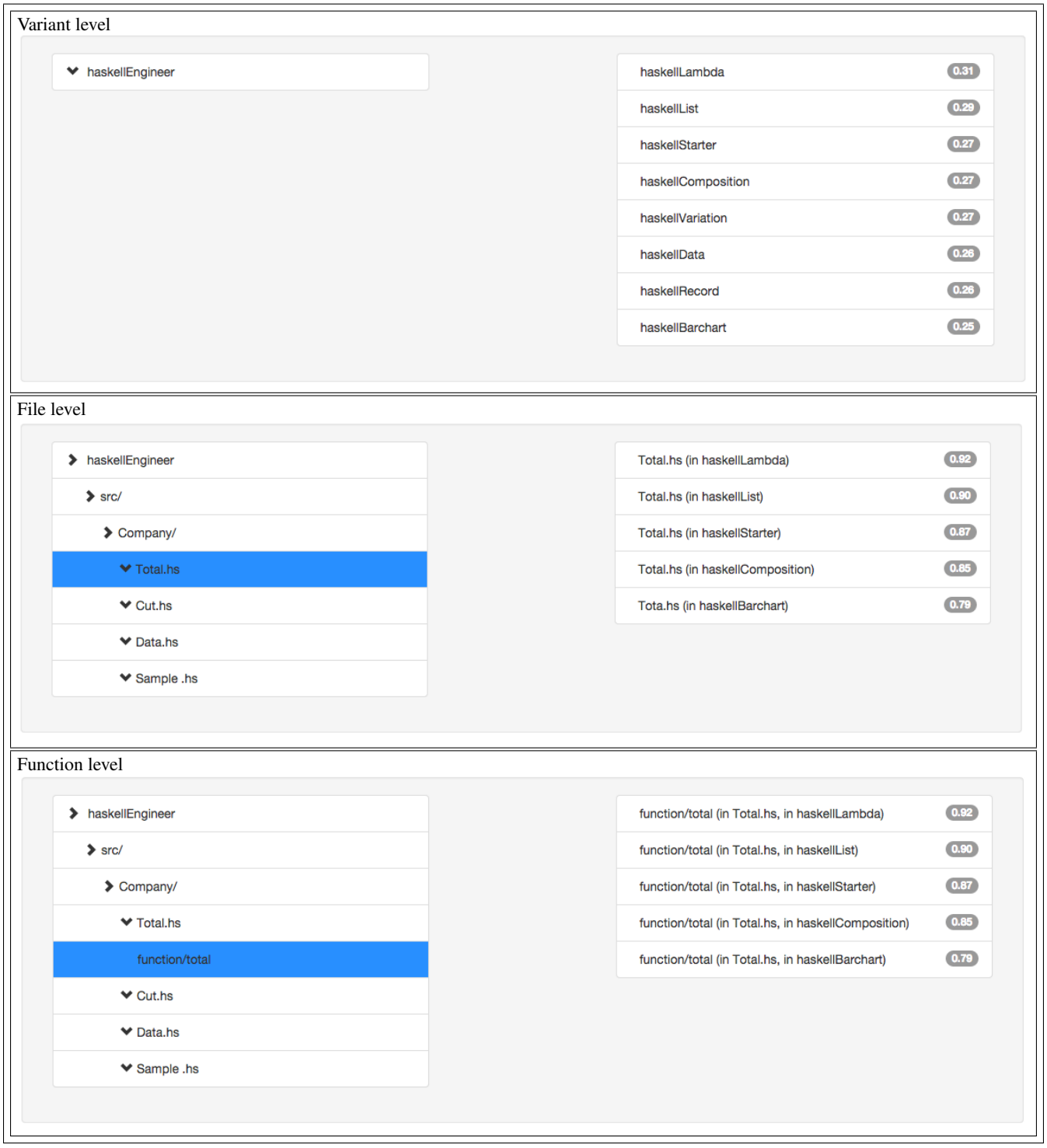

Figure 4. Levels of similarity exploration