Is Functional Programming Better for Modularity?

Ismael Figueroa[*]
Escuela de Ingeniería Informática
Pontificia Universidad Católica de Valparaíso
ismael.figueroa@pucv.cl

Romain Robbes
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
rrobbes@dcc.uchile.cl

ABSTRACT

In 1989 John Hughes published an influential position paper entitled *Why Functional Programming Matters*. The article extolls the virtues of lazy functional programming by developing several examples: the Newton-Rhapson squares root method, numerical differentiation and integration, and an alpha-beta minimax search. A main conclusion of that work is that higher-order functions and lazy evaluation significantly contribute to modularity. We have found that recent articles from 2010 to 2014 cite Hughes' work as seminal work supporting that functional programming is, in general, good for modularity. We believe this reflects an unstated hypothesis in part of the research community: functional programming is inherently better at modularity than other paradigms such as typical procedural and object-oriented programming. To the best of our knowledge there are no (large-scale) empirical evaluations of this characteristic. We discuss the influence of *Why Functional Programming Matters* on current beliefs regarding the advantages of functional programming, the recent citations that intrigues us, and provide a small experiment on the GHC Haskell compiler, suggesting the existence of modularity issues in it.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures*

Keywords

modularity, functional programming, topic analysis, ghc

1. INTRODUCTION

Modularity is a key property of software that helps minimize maintenance efforts and which also improves the quality of the software [7]. The essential idea behind modularity is to *decompose* a system into several *modules* that isolate important design decisions, as well as implementation details that are likely to change [18]. The quest for modularity goes back to the beginning of computer programming. From structured programming [6] to object-oriented programming [14, 15], as well as recent developments such as aspect-oriented [16], context-oriented [10], and feature-oriented [1] programming. Functional programming developed concurrently to object orientation, but remained as an academic niche.

However modularity can be a fuzzy and unprecise concept. In a famous essay, John Hughes, one of the original developers of Haskell, states [13]:

> "[...] we have argued that modularity is the key to successful programming. Languages that aim to improve productivity must support modular programming well. [...] modularity means more than modules. Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To support modular programming, a language must provide good glue."

Entitled as *Why Functional Programming Matters*—from now on simply *Why FP*—the origins of this classic paper go back to 1984 as an internal memo at Chalmers University, which was later revised and published in the Computer Journal in 1989 [13]. The article argues, by developing several example programs in the Miranda language, an early influence on Haskell, how higher-order functions and lazy evaluation *significantly contribute to modularity* [13]. The article concludes:

> "since modularity is the key to successful programming, functional programming offers important advantages for software development."

To this date *Why FP* has 107 citations in the ACM Digital Library[1] and around 900 informal citations (*e.g.* including web pages and other non-indexed publications) according to Google Scholar. Regarding official cita-

[*]Partially funded by FONDECYT 3150672.

[1]`http://dl.acm.org/citation.cfm?id=63411`

tions, the most recent ones are 6 citations in 2014. In the last 5 years, *Why FP* has 33 citations between 2010 and 2014, with an average of 6 citations per year.

Regarding informal citations, Google Scholar shows a constant following since its publication, with a notable increase since 2006.[2] Albeit 5 of the most recent citations in 2014 refer to *Why FP* in a neutral or historical context, we became intrigued by this quote [4]:

> "Experts have extolled the benefits of laziness for decades now. As Hughes explained [Hughes, 1989], lazy programming languages enable programmers to design code in a modular way;[...]"

because it appears that the alleged benefits of lazy functional programming for modularity are taken as a fact without much evidence. To the best of our knowledge there are no large-scale empirical evaluations regarding the modularity benefits of functional programming.

2. DOES IT REALLY MATTER?

Although for space reasons it is difficult to contextualize the programming landscape of 1989, we believe that the paper claims—in its historical context—make a lot of sense. Specifically, it is crucial to be aware that lazy functional languages as we know them today were developed mostly during the '70s and the '80s. A major milestone of this process is the first release of Haskell in 1990, born as the unification of several existing ideas on lazy functional languages [12].

By examining a small sample of quotes from research papers in top conferences and journals, we found that earlier citations to *Why FP* are, in our opinion, quite appropriate because they carefully state the contributions of the article. Another finding is that a large part of the citations are properly used to discuss some historical context. However, there is a small set of recent citations in top conferences that appear to overemphasize the real contributions of *Why FP*.

To begin we quote an 1989 ACM Computing Survey [11] that refers to higher-order functions as a distinguishing feature of modern functional languages:

> "Limiting the values over which abstraction occurs to non-functions seems unreasonable; lifting that restriction results in higher-order functions. Hughes makes a slightly different but equally compelling argument in [Hughes, 1984], where he emphasizes the importance of *modularity* in programming and argues convincingly that higher-order functions increase modularity [...]"

We believe this citation is quite fair. Indeed it clearly states that Hughes just put forward a *compelling and convincing argument* regarding modularity and functional programming. Seven years later, in the first ICFP conference in 1996, Okasaki [17] refers to *Why FP* in the first sentence of the introduction:

> "Functional programmers have long debated the relative merits of strict versus lazy evaluation. Although lazy evaluation has many benefits [Hughes, 1989], strict evaluation is clearly superior in at least one area [...]"

This might be one of the first formal citations where *Why FP* is used as a seminal reference to the virtues of (lazy) functional programming. The idea is reinforced by the fact that Okasaki's paper is not related at all to modularity, but to the amortized analysis of algorithms taking laziness into account; there is no further reference or discussion about Hughes' work on it.

We have found similar situations in at least one paper per year, from 2010 to 2014, where *Why FP* is used as an authoritative reference to the modularity benefits of functional programming. We start with the most recent article, in 2014 at POPL [4]:

> "Experts have extolled the benefits of laziness for decades now. As Hughes explained [Hughes, 1989], lazy programming languages enable programmers to design code in a modular way;[...]"

In 2013 at ESOP [3]:

> "A lazy functional language naturally supports the construction of reusable components and their composition into reasonably efficient programs [Hughes, 1989]"

Here we take reusability and composition as the key terms that refer to modularity. In 2012 at ICFP [20]:

> "Non-strict functional programming languages, such as Haskell, offer important benefits in terms of modularity and abstraction [Hughes, 1989]."

appears as the starting sentence in the introduction. Then in 2011 at IFL [5]:

> "While laziness enables modularization [Hughes, 1989], it unfortunately also reduces a programmer's ability to predict the ordering of evaluations."

also appears as the first sentence in the introduction. Finally, in 2010 at SAC [19]:

> "For instance, using a language such as C++ or C# allows the file system to be written in a high-performance, object-oriented language. On the other hand, functional languages like Haskell and OCaml provide tools such as higher-order functions and lazy evaluation that facilitate modularity and, in turn, productivity [Hughes, 1989]."

This last paragraph discusses the different language bindings for FUSE file system implementations. The paper also confers to OCaml the same alleged benefits as those of Haskell, even though OCaml is strict by default (with a module for explicit lazy computations), and it has primitive side-effecting operations. Furthermore, mod-

[2] `https://scholar.google.com/citations?view_op=view_citation&citation_for_view=adD4xmAAAAAJ:u-x6o8ySG0sC`

ularity seems to be regarded as a feature of functional programming, but not of object-oriented programming.

It is both intriguing and surprising for us that *Why FP* appears to be the primordial citation to praise the modularity benefits of functional programming. In contrast to recent research on the modularity properties of object-oriented software [8, 7, 21] *we are not aware of any empirical evaluation of modularity in functional programming.* Moreover, we believe that this gap is not due to an inherent difficulty to measure modularity in functional programming, but, we conjecture, is due either to lack of interest from the community, or from unstated assumptions like the one described in this paper. We continue by describing two empirical approaches to assess modularity and the application of one regarding the modularity of the GHC Haskell compiler[3], one of the largest software projects written in Haskell.

3. MEASURING MODULARITY

Modularity is traditionally measured by two structural metrics: coupling and cohesion. Coupling measures how much a module relies on other modules, while cohesion measures how close all the methods, functions or attributes of a module work to implement a common concern. It is commonly accepted that a system (or at least a concern) is modular when it is loosely coupled and highly cohesive. A non-modular concern is *scattered* (high coupling) and/or *tangled* (low cohesion).

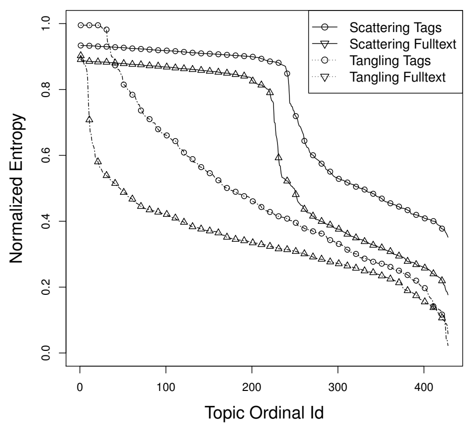
Although the traditional approach can be used to compare modularity across programs written in *the same* programming language, it is difficult to make comparable observations across several programs written in *different* languages. Two language-agnostic methodologies to assess modularity can be used to overcome this issue.

Analysis of software evolution. It focuses mostly on scattering by considering the whole evolution history of the software to detect changes that consistently affect multiple entities or artifacts of the system. The seminal work of Eick *et al.* [8] studies 10 years of the evolution of a major telephone system. One of its main results is the *breakdown of modularity over time* reflected in that the *span of changes*, *i.e.*, the quantity of files affected by an individual code change, increases over time.

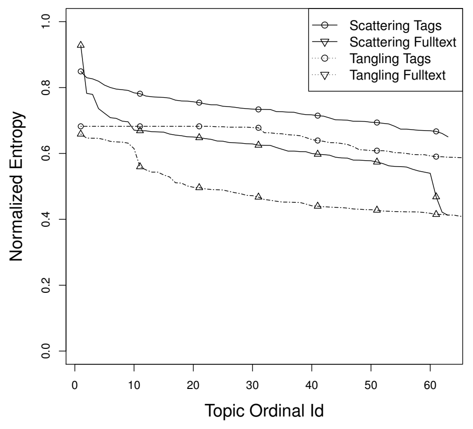
Analysis of concerns as latent topics. Based on information theory, Baldi *et al.* [2] developed a theory to assess the prevalence of crosscutting concerns in software. The theory is summarized as follows [2]: "concerns are latent topics that can be automatically extracted using statistical topic modeling techniques adapted to software. Software scattering and tangling can be measured precisely by the entropies of the underlying topic-over-files and files-over-topics distributions."

These methodologies provide quantitative measurements about the modularity of a system. Moreover,

[3]`https://www.haskell.org/ghc/`



(a) $k_1 = 428$ topics



(b) $k_2 = 63$ topics

Figure 1: Experimental results for topic analysis for $k_1 = 428$ and $k_2 = 63$ number of topics on the `ghc/compiler` subsystem of a snapshot of GHC.

they are not mutually exclusive and can be used together to obtain a better picture of the state of the software. A potential weakness of analyzing software evolution to measure modularity is its focus on scattering. It seems difficult to assess tangling considering only the metadata of affected software artifacts. On the other hand, topic analysis handles both scattering and tangling, but its results are probabilistic in nature and depend on the fine tuning of several parameters. We believe that a mixed approach should yield the best results. Nevertheless, as a first approach we perfomed a simple empirical analisys using the second methodology.

4. PRELIMINARY ASSESSMENT

As a first approach we did a topic analysis following the theory of Baldi *et al.* [2] to assess the modularity of the GHC compiler. The complete material used in this experiment is available at [9]. Using the `topicmodels` R package[4] we used the latent Dirichlet allocation (LDA) algorithm on the `ghc/compiler` subsystem of GHC to construct scattering and tangling curves as in [2].

We considered two parameters, the topic count k and the document corpus. As a simple heuristic we used the total number of modules $k_1 = 428$—in Haskell each module is defined in exactly one file—and the number of top-level subfolders from `ghc/compiler`, $k_2 = 63$, each representing a different subsystem. As the document corpus we first considered the full text of the files, put into proper form using standard unix tools `tr` and `awk`. Additionally we used a custom parser that only keeps the identifiers—or *tags*—of each type and each function used or declared inside a particular file.[5] We performed two simple normalization steps for the vocabulary. First, following [2] we filtered all identifiers from the `Prelude`, which is the set of identifiers available by default in Haskell, including functions to manipulate list, tuples, logical operators, etc. Second, we split identifiers in *camelCase* or in *under_score*; this process still needs improvement.

The resulting scattering and tangling curves are shown in Figure 1. The plots represent normalized entropy [2], a comparable measure of entropy between 0 and 1, of both metrics. Maximum scattering entropy for a topic means that it appears in every document. Likewise, maximum tangling entropy means that a file contains every topic in similar proportion. The curves are similar to those in [2], where entropies above 0.5 were regarded as high scattering or tangling. Based on this we can conclude that for both k_1 and k_2 the scattering entropy is high, which suggest that there exists some modularity issues in this project. Tangling looks less of an issue than scattering, but might be worth exploring for k_2.

5. CONCLUSIONS

Modularity is a key property of software which arises from the proper decomposition of a system into modules. The decomposition mechanisms of a programming language are crucial to determine the opportunities to write modular and maintenable code. Our hypothesis is that parts of the functional-programming community in particular, and of the software-development community in general, overestimate the modularity benefits of functional programming. Our bibliographic analysis supports this hypothesis by showing that several articles in top conferences cite *Why FP* as the main supporting evidence. Following the recent trend on empirical studies on programming languages, we believe that a single paper that develops a handful of sample programs is unsatisfactory evidence for such strong claims. Initial empirical evidence suggest modularity issues in GHC, one of the most emblematic functional programming software projects. Perhaps functional programming is actually better for modularity—we do not know it yet—but currently there is not enough supporting evidence.

[4]`http://cran.r-project.org/package=topicmodels`

[5]`https://github.com/ifigueroap/hothasktags`

References

[1] S. Apel and C. Kästner. "An Overview of Feature-Oriented Software Development". In: *JOT* (2009).

[2] P. F. Baldi et al. In: *OOPSLA 2008*.

[3] S. Chang. "Laziness by Need". In: *ESOP 2013*.

[4] S. Chang and M. Felleisen. "Profiling for Laziness". In: *POPL 2014*.

[5] S. Chang et al. "From Stack Traces to Lazy Rewriting Sequences". In: IFL'11. 2012, pp. 100–115.

[6] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds. *Structured Programming*. 1972.

[7] M. Eaddy et al. "Do Crosscutting Concerns Cause Defects?" In: *TSE* 34.4 (July 2008).

[8] S. G. Eick et al. "Does Code Decay? Assessing the Evidence from Change Management Data". In: *TSE* 27.1 (Jan. 2001).

[9] I. Figueroa and R. Robbes. *Initial Topic Analysis of GHC*. `http://www.inf.ucv.cl/~ifigueroa/doku.php/research/plateau2015`.

[10] R. Hirschfeld, P. Costanza, and O. Nierstrasz. "Context-oriented Programming". In: *JOT* (2008).

[11] P. Hudak. "Conception, Evolution, and Application of Functional Programming Languages". In: *ACM CSUR* (Sept. 1989).

[12] P. Hudak et al. "A History of Haskell: Being Lazy with Class". In: *HOPL III*. 2007.

[13] J. Hughes. "Why Functional Programming Matters". In: *Comput. J.* 32.2 (Apr. 1989), pp. 98–107.

[14] D. H. H. Ingalls. "The Smalltalk-76 Programming System Design and Implementation". In: *POPL '78*.

[15] A. C. Kay. "The Early History of Smalltalk". In: *ACM SIGPLAN Notices* 28.3 (Mar. 1993), pp. 69–95.

[16] G. Kiczales et al. "Aspect Oriented Programming". In: *Special Issues in Object-Oriented Programming*. 1996.

[17] C. Okasaki. "The Role of Lazy Evaluation in Amortized Data Structures". In: *ICFP '96*. May 1996.

[18] D. Parnas. "On the criteria for decomposing systems into modules". In: *Communications of the ACM* (1972).

[19] A. Rajgarhia and A. Gehani. "Performance and Extension of User Space File Systems". In: SAC 2010.

[20] H. Simões et al. "Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs". In: *ICFP 2012*. Sept. 2012, pp. 165–176.

[21] R. J. Walker, S. Rawal, and J. Sillito. "Do Crosscutting Concerns Cause Modularity Problems?" In: *FSE 2012*.