Classifying Language Features and Libraries

Raoul-Gabriel Urma
University of Cambridge
raoul.urma@cl.cam.ac.uk

Tomas Petricek
University of Cambridge
tomas.petricek@cl.cam.ac.uk

Alan Mycroft
University of Cambridge
alan.mycroft@cl.cam.ac.uk

Abstract

Programming languages get increasingly more complex as they evolve. In fact, new language features are sometime only understood and used by a privileged set of developers. As a result, different groups of developers only use different aspects of the language.

In this paper, we argue that developers would benefit from language and library designers classifying language and library features explicitly. We propose that programming language features and libraries should be classified based on different dimensions such as typical applications and conceptual complexity.

The classification enables a wide range of uses. We focus on collaborative team development and propose that language features usage should be controlled based on team roles. We briefly sketch other beneficial applications of our ideas including for software comprehension and teaching.

1. Introduction

A new version of languages like C# and Java is released every 2-3 years, but not all developers immediately learn, or even need to use all new features. For example, generics in Java provide additional type safety and a mechanism for code reuse. However, research shows that only one main contributor of a project uses them in practice [10]. In addition, Scala has many features such as implicit parameters, compounds types that few developers understand completely.

This situation creates hiearchies among developers reminding research in social sciences: only a minority of developers understand the full power of features in a programming language. For example, even though collections are widely used, the design of a flexible collection library is a specialized task that should be performed only by advanced developers using complex language features. The same situation arises in concurrency [11].

2. Language and library design

We first consider what programming language and library designers can do to support classification of language and library features, starting with a brief analysis of current discussions in the community.

2.1 Classification in language communities

Many language communities informally distinguish generally accepted language features and libraries that can be safely used by anyone from more advanced features that should be only used by experts. In languages where language features can be explicitly enabled (such as Haskell extensions [2] in GHC), these are also often informally classified.

The following quotes (from forums on StackOverflow site) demonstrate this observation:

- on finalization in Java: *"Only experts in Java should be going anywhere near finalize()."* [4]
- on lock-free structures *"Creating [a] lock free structures is extremely hard, and only experts [...] can do it."* [5]
- on GHC features: *"FlexibleInstances and TypeSynonymInstances are uncontroversial; [..]. UndecidableInstances and OverlappingInstances are dangerous territory"* [6]

Similarly, Odersky proposes to classify Scala features [9]: *"Not every Scala programmer needs to make use of every specialized tool in Scalas arsenal [...]. I therefore propose a scheme where some of the more advanced and contentious language features have to be enabled explicitly."*

These examples confirm that it is desirable for the language and library designers to classify features in some way, but it poses a question – along what dimensions should the features be classified?

2.2 Classification dimensions

We propose that language designers should classify language features and libraries explicitly. For example, they could be classified based on *typical applications* and *conceptual complexity*. Other criteria such as *safety* and *expressiveness* could also influence the classification.

For example, generics in Java are typically used to design type-safe data types. However, they are considered difficult to grasp by most programmers. On the other side of the spectrum, the `foreach` loop is used for control flow and is straightforward to learn and use in comparison.

2.3 Features usage

In addition to classifying language primitives and libraries, it is also desirable to classify certain usage patterns (or code

skeletons [8]). For example, empty blocks and unnecessary constructors are discouraged in Java.

Such discouraged patterns are typically detected by external source code analysis tools [1, 3]. We argue that such code structures should be classified similarly to basic library and language features.

This would allow more advanced and fine-grained control over their usage. For instance, only certain code skeletons (involving multi-threading libraries) could be allowed to junior programmers while senior programmers would be allowed to use the library without restrictions.

3. Language and library usage

Once language designers provide a classification of language features, libraries and patterns, we can use this information in a number of ways. Here, we consider feature access control in software teams.

3.1 Collaborative software development

The development of a software is in general a team effort with people with different skillset collaborating together. For example, one could consider a team developing a trading software consisting of a *scientist* writing complex algorithms, two *designers* working on the user interface, a *senior programmer* responsible for the backend and a *technical lead* supervising the project.

It is desirable to restrict what features certain team members can access. For example, junior programmers should not modify code written by the scientist (built with complex mathematical libraries) or modify code written by the technical lead. Similarly, the scientist should not access network communication code written by the senior programmer.

This scenario raises the following question: How could we control access to language features and libraries within a software development team?

3.2 Features access control

We propose a model based on ideas from access control in security research. We define an access control matrix based on the set of roles in a software team and on the set of language features and libraries classified by language designers. Furthermore, we define a set of actions a role may execute involving a specific feature. Such actions are dependent on the team's workflow. For example, one could consider *writing* (W) the initial code, *modifying* code (M), *check-in* of code with (CR) and without (C) code reviews. The following table shows an example of how to specify access control for the example in Section 3.1.

	Designer	Scientist	Senior Eng.
finalization			W,M,C
collections	W,M,CR	W,M,C	W,M,C
generic calculations		W,M,C	M,CR

Figure 1. Features access control

If language features and libraries were classified by the designers, restrictions such as those just discussed could be enforced by the compiler and standard tools.

Furthermore, access control matrices could be defined and agreed on by the community and provide formal guidance for software development teams. This contrasts with informal views discussed in Section 2.1.

4. Conclusion

We discussed the need for classifying language features and libraries. Programming language communities often classify features based on their *difficulty*, but we propose other, equally important dimensions such as *safety*, *expressiveness* and *typical applications*.

We argue that collaborative software development can benefit from such classification. We consider restricting the access to features based on team roles and typical development workflow operations. Other beneficial applications of such classification include:

Software comprehension. An explicit classification of language features and library by conceptual complexity could be used to quickly understand the complexity of different modules in a software project.

Education. When teaching programming, it is often desirable to use only core language features [7]. Thanks to the classification of features, a full-scale language can be used for teaching as well as for real-world development.

References

[1] Findbugs. `http://findbugs.sourceforge.net`.

[2] GHC language features. `http://www.haskell.org/ghc/docs/7.2.1/html/users_guide/ghc-language-features.html`.

[3] PMD. `http://pmd.sourceforge.net/`.

[4] Why is the finalize() method in java.lang.Object protected? `http://stackoverflow.com/questions/2291470/`

[5] How do I write a lock-free structure? `http://stackoverflow.com/questions/92455/`

[6] Function type specialisation in Haskell. `http://stackoverflow.com/questions/9881726/`

[7] R. Chatley. Kenya. Master's thesis, Imperial College London.

[8] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0-262-53086-4.

[9] M. Odersky. Sip-18: Modularizing language features. Technical report, Scala Improvement Process, 2012.

[10] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, 2011.

[11] V. Sarkar. Inverted Pyramid of Parallel Programming Skills.