Haddock, A Haskell Documentation Tool

Simon Marlow
Microsoft Research Ltd., Cambridge, U.K.

Abstract

This paper describes Haddock, a tool for automatically generating documentation from Haskell source code. Haddock's unique approach to source code annotations provides a useful separation between the implementation of a library and the interface (and hence also the documentation) of that library, so that as far as possible the documentation annotations in the source code do not affect the programmer's freedom over the structure of the implementation. The internal structure and implementation of Haddock is also discussed.

Categories and Subject Descriptors

I.7.2 [**Document and text processing**]: Document Preparation—*Languages and systems, Markup languages*

General Terms

Design, Languages, Algorithms

Keywords

Haskell, Documentation tool, Documentation generation, Source-code documentation, API documentation, Module system

1 Introduction

Generating documentation directly from source code has recently become fashionable, due in no small part to the popularity of Sun's JavaDoc tool[9]. Nowadays most languages have at least one tool for generating documentation from source code [11, 5, 3, 2], and if you program in C or C++ you are in the fortunate position of having a multitude of tools to choose from.

This paper describes Haddock, a documentation tool for Haskell. Figures 1 and 2 give examples of an annotated Haskell module and the corresponding HTML output produced by Haddock, respectively. Haddock improves on other documentation tools in some important ways, as we shall describe later in this section.

Firstly let us be clear about the problem domain: we are primarily interested in generating documentation for a library, or API (application programming interface), rather than generating nicely-formatted source code. In particular *literate programming* systems[6] do not fall into this category; they are concerned with writing well-documented source code, to be later formatted in its entirety. A consumer of an API or library is not interested in the implementation details of the library; indeed, we would rather implementation details were omitted from the documentation wherever possible, for obvious modularity reasons.

There are several compelling reasons to combine library documentation and source code:

- There is less chance that the documentation will stray out of sync with the reality of the implementation, since the documentation is right next to the implementation.
- In most cases there is already documentation in the source code in the form of comments, and there may well be duplication between the comments in the source code and the documentation. Clearly, if the comments can also be interpreted as the documentation itself, then we can eliminate the duplication and furthermore make it easier for the programmer to keep the documentation up to date (and give the programmer an incentive to keep the comments up to date!).
- There is a great deal of documentation that can be extracted automatically from the source code: APIs, types, data structures, class hierarchies, dependency graphs, and so on. Having a tool to extract this information from the actual implementation is more desirable than trying to duplicate it in separate documentation, and even better is a tool that can include programmer-written documentation along with the extracted information.
- Having interpreted the API from the source code, a documentation tool can automatically cross-reference the documentation it produces. If a function type mentions a particular type constructor, for example, it can be hyperlinked or cross-referenced to the definition of that type constructor. The tool can also generate an index from names to definitions, and even an index from names to uses, without intervention from the programmer.

Our documentation tool, Haddock, provides all of these benefits for Haskell source code. In addition, we believe that the following

Haskell'02, October 3, 2002, Pittsburgh, Pennsylvania, USA.