Review on *Verified Functional Programming in Agda*

By Aaron Stump
ACM, ISBN: 978-1-97000-126-6, 246 pages, 2016

MATTEO PRADELLA, Politecnico di Milano, Italy

CCS Concepts: • **Software and its engineering → Functional languages**; **Software verification**.

Additional Key Words and Phrases: Dependent types

ACM Reference Format:
Matteo Pradella. 2022. Review on *Verified Functional Programming in Agda By Aaron Stump* ACM, ISBN: 978-1-97000-126-6, 246 pages, 2016 . *Form. Asp. Comput.* 1, 1 (January 2022), 2 pages. https://doi.org/XXXXXXX.XXXXXXX

"Verified Functional Programming in Agda" is a practical introduction to verified functional programming, using the dependently-typed functional programming language Agda. Agda is a programming language and proof checker developed at Chalmers University by Ulf Norell; the version covered in the book is Agda 2. An important aspect to be noted is that the book is not based on Agda's standard library: the author developed an ad hoc library for the book, called IAL (Iowa Agda Library). The book under review is based on the 1.2 version of the library, freely available here: https://github.com/cedille/ial

The compelling idea of Agda is that, thanks to its very expressive and flexible type system, it can be used both as a programming language and as a proof checker. Dependently typed languages offer types that can depend on values, so actual values can be used at the type level. In a nutshell, the idea of verified functional programming is to exploit the Curry-Howard isomprphism between proofs and programs to write functions that verify the properties encoded in their types. The Curry-Howard isomorphism is basically a correspondence between Intuitionistic (or Constructive) Logic and Typed Lambda Calculus, so proofs correspond to code and logic formulae to types: to give the reader an intuitive idea, conjunction corresponds to product types, disjunction to sum types, implication to function types, existential quantification to dependent pair types (where the type of the second component depends on the value of the first component), and universal quantification to dependent function types (where the return type depends on the value of the input value).

A typical example of a dependent type is that of vectors encoding the value of their length together with the vector content: in this kind of setting, it is possible for instance to state in the signature of the *append* function the property that the length of the resulting vector is the sum of the lengths of the input vectors. This approach can be used in general to define very expressive types, and then to use them in the signature of functions to state some of their properties. This is called in the book *internal verification*, because the property is expressed directly in the type of the function, while its proof is in its code: if the code is correct for the type checker, then the property is proved.

Author's address: Matteo Pradella, matteo.pradella@polimi.it, Politecnico di Milano, DEIB, Via Ponzio 34/5, Milano, Italy, 20133.

0934-5043/2022/1-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

Agda can be also used as a proof checker: we can define a function where the signature is in fact a statement (e.g. a complex property of other functions), and its implementation is its actual proof — this is called in the book *external verification*.

The approach used throughout the book is quite practical: it contains many examples and hands-on sessions with the text editor Emacs, which is the de facto standard IDE for Agda.

The structure of the book is the following: Chapter 1 works as a basic introduction on Agda, functional programming and the Emacs environment as an IDE — the main subject is the Boolean data type. Chapter 2 is about constructive proofs; the Curry-Howard isomorphism is introduced, while the reader is led to use Agda as a proof-checker, by proving some basic results on Booleans. Chapters 3 and 4 introduce natural numbers and lists, respectively, with definitions and some classical theorems. Chapter 5 is about one of the central concepts of dependent types: internal verification. In particular, it covers the following data structures: Vectors, Binary Search Trees, and Braun Trees. Chapter 6 introduces the concept of type-level computation, working with integers, formatted printing, and the notion of proof by reflection. Chapters 7 and 8 are devoted to some comprehensive practical examples: parser generation with the *gratr* tool, and Huffman encoding and decoding, respectively. Probably the most complex part is at the end of the book: Chapter 9 is on reasoning about termination, where an operational semantics for SK combinators, a minimal, untyped Turing-complete language, is presented; Chapter 10 focuses on intuitionistic logic and Kripke semantics. Some important features of Agda that are explicitly not covered in the book are co-inductive types (i.e., types for infinite data structures), and monadic programming; record and modules are only partially covered.

As far as needed background is concerned, the author of the book assumes that the reader is a student in Computer Science, with some basic knowledge in discrete mathematics and programming in general (functional programming is not required). On the other hand, some previous experience with statically typed functional programming languages such as OCaml, Haskell or F#, as well as some basic knowledge on classical mathematical logic (at least propositional and first-order) can be very helpful.

All in all, I found the book an enjoyable and very effective introduction to a dependently-typed functional programming language such as Agda, and in general to verified functional programming. This is a good tool for teaching such concepts to beginning graduate students in Computer Science, since it is quite practical and hands-on, while at the same time providing many useful pointers to deeper presentations of the more advanced concepts.