JFP **30**, e16, 3 pages, 2020.
doi:10.1017/S0956796820000143

Book Review

Type-Driven Development with Idris

Programming languages that support dependent types provide us with an expressive environment in which we can specify and reason about properties of our software programs. *Type-Driven Development with Idris* is an introductory text to dependently typed programming using the Idris programming language, which the book introduces, and to how we can use Idris' support for first-class dependent types to engineer software programs that are not only maintainable but programs that can also be formally verified.

Type-Driven Development with Idris is not the first book to describe how dependent types are used to reason about software programs (Chlipala, 2013; Stump, 2016). While these books provide extensive coverage of the subject matter, they are aimed toward the theorists (squigglers) who wish to develop theories using dedicated theorem provers. What is nice about the book, and the Idris language, is that it is aimed more at engineers (bodgers) not used to squiggling or functional programming. Importantly, the book does not shy away from more squiggly topics such as totality, decidability, and theorem proving (which are required components for verifying programs) and introduces them at a gentle pace comfortable for the novice.

The book is divided into three parts.

Part 1 offers a general introduction to the basics of functional programming in Idris that is reminiscent of Hutton's *Programming in Haskell* (Hutton, 2016). The author considers the expected readers' background and ensures that those experienced enough can jump ahead where appropriate, and that those not so experienced are given the time required to learn functional programming in Idris. What is nice here is that the author, using the classic "Lists with Length" example, shows the need for types and how type-driven development helps with the software engineering process before the Idris language is introduced—thus, allowing one to gain an intuition for why type-driven development matters.

Part 2 delves into the core of programming with dependent types in Idris, introducing language features and idioms that allow one to specify and reason more precisely about one's software programs than we can when using existing general-purpose programming languages. Specifically, the author introduces the reader to dependent types and how they can be used to write interactive programs and demonstrates how to verify these same interactive programs by using dependent types to reason, at the type level, about the structure of data and how dependent types can change how we view data to extend pattern matching in interesting and practical ways. Importantly, the book demonstrates how these idiomatic constructs support the construction of efficient verified code, an important aspect when one is developing real programs. The author makes backwards references, refinements, and extensions toward the examples mentioned in Part 1, giving the reader a chance to see previously seen concepts in action. More so, the running examples of a "word guessing

game" and an interactive data store are used to provide motivation and exposition of the concepts being introduced.

While Parts 1 and 2 introduce us to type-driven development, Part 3 demonstrates its application and details how one can work with infinite data, and stateful and concurrent programs, in type-safe ways. Part 3 is important as the examples we have seen so far have been about learning Idris, and we now see what Idris can do. Each chapter within Part 3 builds up how we can work with type-level state machines, describing how we can have total interactive programs, then total interactive stateful programs, then total interactive stateful programs that follow a type-level specification, and finally, total interactive stateful programs that follow a type-level specification and handle feedback and errors. Here, each example builds upon the lessons learned in the previous examples to show how types and total programming help build more reliable verifiable programs. This layering is beneficial as it gradually helps the reader to see how the extensions work together. Notably, we are shown how to simulate an ATM machine to deal with errors, and we end up with a verified word guessing game that compliments the version seen at the end of Part 2.

The book ends by explaining how one can achieve type-safe concurrent programming, bringing together what we have learnt so far in the book. This chapter details how we can use Idris' concurrent primitives in unsafe ways and how we can use types to constrain use of these primitives in ways we know to be safe. We see this with examples that detail list processing and word counting using a simple framework as specified in the chapter.

A pleasant detail found within the book is that there is a plethora of example code, walk-throughs, and exercises. All are appropriately aimed, complete, and clearly demonstrate the mantra *Type, Define, Refine* as one types, defines, and refines their way through the book. With pervasive use of interactive examples requiring user input, the author puts interaction (and practical programming) at the heart of the book's story. The use of Idris' typed-holes here is exemplary, allowing one to engage with the compiler in conversation, *in situ*, and explore the type-driven style of development. This is the best way to experience type-driven development and is reminiscent of academic textbooks that invite the reader on a journey to understand the material being taught. In the act of filling holes, combined with the guiding instructions, one is embedded within a three-way conversation between the reader, the book, and the Idris compiler! This is how programming should be.

Unfortunately, the book ends rather abruptly with the last chapter. It would have been nice to see where one can go from there. The author treats type-driven development within the context of the Idris programming language, and it would have been more fulfilling if the reader was given some hints on how we can transfer this approach to more mainstream software engineering practices and languages. Further, it is not clear what the future holds for type-driven development or for Idris. While Appendix D does indeed cover further reading, I am nonetheless left wanting more, and I suspect other readers will be too. This cannot be the end of the story.

Overall, *Type-Driven Development with Idris* is a great book, well-written, and highly engaging. If, like mine, your background was as a bodger and not originally in formal methods, then I would highly recommend this book to you. The book is accessible and introduces the reader to a brave new world of programming that blends functional programming, formal mechanized reasoning about programs, and software engineering. If

https://doi.org/10.1017/S0956796820000143 Published online by Cambridge University Press

you are like me now, a squiggler-bodger, then this book will show you a programming language with dependent types that supports theorem proving, rather than a theorem prover in which you can program. While the discerning squiggler might shy away from this book, they should not: This book is for you too! The book's presentation style demonstrates brilliantly how we can present squiggly topics and their application to a wider audience. A presentation style I think squigglers everywhere will appreciate. This is a book for squigglers and bodgers alike. It is a book that I wish existed when I started typing and defining, and I am glad it is here now that I am constantly refining.

Conflicts of Interest

Reviewer's note: During my doctoral studies, Edwin Brady was my second supervisor. I was employed by Edwin as a research associate on an EPSRC-funded project to investigate Type-Driven Development of Communicating Systems, and I still regularly collaborate, and publish, with Edwin. During the initial consulting phase for *Type-Driven Development with Idris*, I was asked by Manning to review an early outline.

References

Chlipala, A. (2013) *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press.

Hutton, G. (2016) *Programming in Haskell*, 2nd ed. Cambridge University Press.

Stump, A. (2016) *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool.

JAN DE MUIJNCK-HUGHES
School of Computing Science
University of Glasgow
E-mail: jan.demuijnck-hughes@glasgow.ac.uk

https://doi.org/10.1017/S0956796820000143 Published online by Cambridge University Press