presentation addresses the problems of imprecision and incompleteness, raised in Chapter 1 as potential defects of informal semantics in general.

This book will mainly appeal to undergraduate readers who are taking courses on formal semantics. It could serve as the main textbook for a low level course, provided that the lecturer is happy with the selection of material. Many, one suspects, would prefer to include more material in a course (e.g. object-oriented programming, concurrency), possibly at the expense of logic programming. Nonetheless this book could still be a useful source material for parts of such a course. In spite of the inexplicable choice of presenting a formal semantics for dynamic binding in Chapter 5, the book is, on the whole, nicely written and quite readable.

Alex Simpson

The Haskell School of Expression by Paul Hudak, Cambridge Univerity Press, 2000.[1]

doi:10.1017/S0956796807006284

As the title implies, The Haskell School of Expression (SOE) introduces functional programming through the Haskell programming language and through the use of graphics and music. It serves as an effective introduction to both the language and the concepts behind functional programming. This text was published in 2000, but since Haskell 98 is the current standard, it is still a very relevant book.

Haskell's standardization process gives us a window into two different facets of the community: Haskell is designed to be both a stable, standardized language (called Haskell 98), and a platform for experimentation in cutting-edge programming language research. So though we have a standard from 1998, the implementations (both compilers and interpreters) are continually evolving to implement new, experimental features which may or may not make it into the next standard.

For instance, the Glasgow Haskell Compiler has implemented a meta-programming environment called Template Haskell. Haskell is also easy to extend in directions that don't change the language itself, through the use of "Embedded Domain Specific Languages" (EDSLs) such as WASH for web authoring, Parsec for parsing, and Dance (more of Paul Hudak's work) for controlling humanoid robots.

Before we get too far, I should offer a disclaimer. The Haskell community is rather small, and if you scour the net, you may find conversations between myself and Paul Hudak or folks in his research group, since I use some of their software. That said, I don't work directly with Hudak or his research group.

In fact, the small size of the Haskell community is a useful feature. It is very easy to get involved, and folks are always willing to help newbies learn, since we love sharing what we know. You may even find that if you post a question about an exercise in SOE, you'll get a reply from the author himself.

I consider this book to be written in a "tutorial" style. It walks the reader through the building of applications, but doesn't skimp on the concepts (indeed, the chapters are meant to alternate between "concepts" and "applications.") In some ways, the code examples make it a little difficult to jump around, since you are expected to build upon previous code. The web site provides code, however, so you can always grab that and use it to fill in the missing pieces.

For readers who wish to use this book as a tutorial, and to implement all of the examples (which is highly recommended), I suggest that you grab the Hugs interpreter and read the

[1] A version of this review first appeared on Slashdot.

User's Guide while you're reading the first few chapters of SOE. Hugs is very portable, free, and easy to use. It also has an interface with Emacs. Unfortunately, some of the example code has bit-rotted a little, and certain things don't work out-of-the-box for X11-based systems. The bit-rot can be solved by using the "November 2002" version of Hugs. This is all explained on SOE's web page.

SOE should be very effective for programmers who have experience in more traditional languages, and programmers with a Lisp background can probably move quickly through some of the early material. If you've never learned a functional language, I definitely recommend Haskell: since Haskell is purely functional (unlike Lisp), it will more or less prevent you from "cheating" by reverting to a non-functional style. In fact, if you've never really looked at functional programming languages, it may surprise you to learn that Haskell has no looping constructs or destructive assignment (no $x = x + 1$). All of the tasks that you would accomplish through the use of loops is accomplished instead through recursion, or through higher-level abstractions upon recursion.

Since I was already comfortable with recursion when I started this book, it is hard for me to gauge how a reader who has never encountered recursion would find this book's explanation of the concept. SOE introduces recursion early on, in section 1.4. It is used in examples throughout the book, and if you follow along with these examples, you will most certainly be using it a lot. The introduction seems natural enough to me, but I note that Hudak does not give the reader any extra insight or tricks to help them along. Not to worry, though; recursion is very natural in Haskell and the reader may not even notice that they are doing something a little tricky.

The use of multimedia was a lot of fun for me, and should quickly dispel the myth that IO is difficult in Haskell. For instance, Hudak has the reader drawing fractals by page 44, and throughout the book, the reader will be drawing shapes, playing music, and controlling animated robots.

Any book on Haskell must be appraised for its explanation of monads in general and IO specifically. Monads are a purely functional way to elegantly carry state across several computations (rather than passing state explicitly as a parameter to each function). They are a common stumbling block in learning Haskell, though in my opinion, their difficulty is over-hyped.

Since input and output cause side-effects, they are not purely functional, and don't fit nicely into a function-call and recursion structure. Haskell has therefore evolved a way to deal safely and logically with IO through the use of monads, which encapsulate mutable state. In order to perform IO in Haskell, one must use monads, but not necessarily understand them.

Some people find monads confusing; I've even heard a joke that you need a Ph.D. in computer science in order to perform IO in Haskell. This is clearly not true, and this book takes an approach which I whole-heartedly agree with. It gets the reader using monads and IO in chapter 3 without explaining them deeply until chapters 16 (IO) and 18 (monads). By the time you get there, if you have heard that monads are confusing, you might be inclined to say "how is this different from what we've been doing all along?" Over all, I was pleased with the explanation of monads, especially state monads in chapter 18, but I felt that the reader is not given enough exercises where they implement their own monads.

If you're worried that drawing shapes and playing music will not appeal to your mathematical side, you will be pleased by the focus on algebraic reasoning for shapes (section 8.3) and music (section 21.2), and a chapter on proof by induction (chapter 11).

After reading this book you will be prepared to take either of the two paths that Haskell is designed for: you can start writing useful and elegant tools, or you can dig into the fascinating programming language research going on. You will be prepared to approach arrows, a newer addition to Haskell which, like monads, have a deep relationship to category theory. Arrows are used extensively in some of the Yale Haskell group's recent work. You will see a lot of shared concepts between the animation in SOE and Yale's "Functional Reactive

Programming" framework, Yampa. If you like little languages, you'll appreciate how useful Haskell is for embedded domain specific languages. It may be even more useful now that Template Haskell is in the works.

Isaac Jones

Inductive Synthesis of Functional Programs by U. Schmid, Springer Verlag, 2003, 420pp, ISBN 3540401741.
doi:10.1017/S0956796807006296

Program synthesis is one of those deceptively simple propositions: say what you want some program to do, turn the handle and out pops the program. The problem, of course, is to determine what the handle drives, if not a skilled human being.

Deductive synthesis, with which JFP readers will be most familiar, concerns the generation of programs using rigorous techniques that ensure that that they satisfy initial specifications. This approach received an enormous boost when declarative programming emerged from the labs in the late 1970s, egged on by various 5th Generation Programmes in the early 1980's, most significantly in Japan and the UK. The attendant hype, that in declarative programming one identified *what to do* rather than *how to do it*, seemed to offer a seamless link between specification and implementation, through the mediation of theoretically grounded logic and functional languages.

Alas, Hayes & Jones (HJ89) influential 1989 paper effectively ended research into so-called executable specifications. In rightly cautioning that specifications were typically undetermined and infinitary, and hence not directly amenable to unique or indeed correct implementations, they also reasserted the highly contentious Oxford-school idealism that ranks mathematics over computation. Thus, today, it is salutary that there are no mainstream tools for generating implementations from, say, Z specifications, and that "formal methods" of program construction, like program refinement (Mor90), type theory programming (NPS90; MM04) and weakest precondition calculation (Bac03), lurk in quiet corners of academe. The most promising compromise may be the B-Method (Sch01), where specifications may generate results through terminating animation, and implementations may be realised through the production of more and more concrete refinement machines based on increasingly restricted B subsets.

However, in orthogonal research, the mathematical reasoning community has long used techniques based on automatic theorem proving coupled with planning to prove and generate programs satisfying Floyd-Hoare style pre- and post-conditions, and invariants. For example, we have used tactics-based proof planning constrained by rippling (BDHI05) to derive higher-order functions in Standard ML programs that lack them (CIMS05).

Inductive synthesis, rooted in Artificial Intelligence and Cognitive Science, differs radically from deductive synthesis. Rather than using abstract logical pre- and post-conditions to specify requirements, programs are induced from concrete instances of input/output values or program traces. Thus, there is no formal idea of correctness: instead, the process seeks to elaborate program structures that satisfy the specific training instances.

Perhaps the most widely known form of inductive synthesis is genetic programming (Koz92), where an evolutionary algorithm is used to generate syntactically correct program structures which are evaluated for fitness by execution against an input/output set. Initially, untyped forms of LISP were popular: more recently there has been a trend towards typed languages, for example Grant (2000), to try to constrain the potentially vast search space.

The approach to inductive synthesis presented in this challenging volume combines universal planning, plan transformation and folding. To begin with, universal planning is applied to a problem specification in the standard planning language Strips to produce sets of optimal sequences of actions, characterised as function applications, to cover the entire