

## Experiences from teaching functional programming at Chalmers

John Hughes<sup>1</sup>, 7 April 2008.

This note recounts my experiences of teaching functional programming at Chalmers University in Gothenburg, the successes and the problems I encountered. A functional language has been used for the eight week introductory programming course at Chalmers since the early 1990s, initially using ML. I took over the course in the late 1990s, in connection with a switch to Haskell, and taught it every subsequent year until recently.

In the late eighties and early nineties, many functional programming enthusiasts pushed to teach the subject in the first programming course, believing it to be a good vehicle for conveying basic programming concepts—and perhaps also that students would be “spoiled” if they learned imperative programming first. However, there are problems associated with doing this: it does not meet student expectations; many students have nothing to compare it with, and so cannot appreciate the advantages of a functional approach over a traditional one; students learn that functional programming is good for “toy problems” (because that is all a first programming course contains), but that “to do anything real you need Java/C++/whatever”. Such problems had led to an “I hate ML” club among the students at the time I took over the course. If many students see functional programming *only* in the introductory course, then such a course can do more harm than good.

My highest priority was thus to convince students that they could write real, interesting programs in Haskell by the end of the first course. To this end, I removed from the course all material not directly relevant to programming—such as a section on program proofs—in favour of developing as exciting programs as possible. I taught the material in quite a different order to most functional programming texts, for example introducing Haskell input/output (and yes, a monad) in the first lecture—because it is needed for most “real programs”. I found that whatever is taught late in the course is regarded as difficult, no matter whether it really is or not—because students have little time to practice it before the exam, and so prioritize it less in their study. Input/output is usually deferred to a late chapter of Haskell textbooks, with the result that students believe it to be much harder than it is. Another good example is user-defined types, which are often deferred in Haskell textbooks in favour of lists and tuples—because the latter are already sufficient for many interesting examples. I found user defined types were viewed as difficult, until I moved them to the beginning of the course and taught them as a way to “model the problem”—after which they were considered to be easy.

Although the material is quite advanced for a first programming course, I also included two lectures on GUI programming (using wxHaskell), building a simple straight-line-diagram editor in a couple of pages of code. Students do not expect to see real-time graphics from Haskell programs. As a result, you could hear a pin drop during the demo; this was really strongly motivating for the students. (This is one reason why I chose to use full Haskell for the course, rather than Helium, a beginners’ subset. I feel it is important, when students ask “Can I do X in Haskell?”, to be able to answer “yes”, almost no matter what X is—smart students will go beyond what the course covers, and they must not

---

<sup>1</sup> Dept of Computer Science and Engineering, Chalmers University of Technology, S-41296 GÖTEBORG, Sweden.

encounter any artificial barrier to progress, such as a change from a “toy” version to a real version of the language).

It is important to realise that first-year undergraduates are among the most conservative audience one can encounter, where novel programming languages are concerned. Most are aware of only one or two languages, and expect to learn one of them—ideally one they have seen in job advertisements. They are not *a priori* motivated to learn an obscure language in an obscure paradigm. It is therefore essential to take up reasons for teaching functional programming explicitly. Pedagogical arguments may make some impression, but much more motivating is to ensure students are aware of the industrial applications, however few, where the technology they are learning is making a difference.

There are now quite a number of success stories for functional programming in industry, and I make sure to include up-to-date examples in the course. The majority of such successes are quite small scale... either in start-ups such as Galois or Bluetail, or in small groups within larger companies, such as Ericsson, Intel, or Credit Suisse. I find it useful to explicitly discuss the technology adoption curve—in fact I recommend reading *Crossing the Chasm* to interested students—so that the class understands that technological change always starts small. An image of functional programming as a new technology that early adopters are using to beat the competition, rather than an oddity that academics like, is far more motivating for students. To be convincing, it was essential that I, an academic, be well-informed about industrial developments—which I became by attending the Commercial Users workshop at ICFP for example, and the Erlang User Conference, and by engaging with the industrial pioneers of functional programming—even when they did not adopt every good idea from Haskell and ML! Over the years I have invested considerable time in this, but it has been well worth it. Now that I am an entrepreneur with my own company, using functional programming as a key part of its technology, my credibility with students has seen a further dramatic improvement.

There is a general point here. Programming language technology, such as functional programming, *is* being used to do amazing things in industry—but students remain largely unaware of it. As teachers, we need to take part in those developments as best we can, and bring them to our students—whether by recounting our own experiences, telling success stories about others, or by bringing guest lecturers into the University to talk about them. This may not contribute to covering the course syllabus, but its contribution to student motivation is worth gold.

It is certainly an advantage to be teaching a topic so close to my own research. I reflect that by including a lecture every year reporting on ICFP. Obviously, much that is presented at ICFP is beyond the reach of first year students—but I always succeed in finding *something* interesting to report to them. Knowing that the topic they are studying is close to the research frontier is motivating for at least the better students.

In recent years, I have included QuickCheck (random property-based) testing as part of the course, teaching students first to write the left-hand-side and type of a function, then its property, and finally its right-hand-side. Students did follow this advice, and learned to take testing much more seriously, but I am ambivalent about the experience. Teaching students to use the tool—and especially to write test data generators—takes time that could be spent developing more exciting programs. And for students who are struggling even to define a simple function, defining a QuickCheck property is not

an easier step on the way. Perhaps testing and reasoning about programs should rather be left entirely to a later course.

Some believe that prior knowledge of imperative programming is an obstacle to learning functional programming—perhaps inspired by Dijkstra’s comment that learning Basic mutilates the mind. This is not my experience. Good imperative programmers already understand many concepts—such as formal syntax, a compile-time error, or an algorithm—that absolute beginners struggle with. On the other hand, those with programming experience may *believe* they are finding Haskell difficult—even while they race ahead of their classmates—because they expect to know everything already, and find, to their surprise, that they have plenty to learn. But this can be turned to advantage: many absolute beginners appreciate the apparent “level playing field” created by teaching a language *no-one* knows, while more experienced programmers enjoy (unexpectedly) learning something new. Some students do genuinely find Haskell difficult—but those students generally find Java difficult too, and should probably not be studying programming at all.

An important challenge is ensuring that students are able to use Haskell in later courses too, for students who learn functional programming in their first year, and never use it again, will not appreciate it much in retrospect. This is made more difficult by a lack of suitable textbooks, multiple interacting curriculae which mean that later courses usually include some students who have *not* studied Haskell, and indeed, conservatism on the part of some colleagues. Ensuring that Haskell is at least an option for students to use on subsequent courses is vital to generating enthusiasts five years later among our graduates.

There is a widespread perception that programming languages are irrelevant, that if Java was good enough for Saint Paul, then it’s good enough for the rest of us too. As programming language researchers, we face an ongoing struggle to convince our students, and our colleagues in other areas, that this is not the case. Luckily, the situation is more fluid now than it has been for a long time. Functional languages are seeing sharply rising interest, and clear influence on the mainstream (such as lambda-expressions in C# and Visual Basic). The advent of multicores is leading to speculation about new programming models from quite unexpected quarters. Now is the time for high risk, high impact ventures, such as the “Jobs in Functional Programming” event I organised—the first ever of its kind. *Jobs in FP* could have been a fiasco, but turned into a resounding success—not least thanks to all the students who have become functional programming enthusiasts, thanks to the course discussed here. This success is proving to be worth gold for marketing the subject within the University.

By the time students who are starting their studies now graduate, then very many of them will need to be accomplished parallel programmers—something that is beyond the reach of most, if they must use today’s predominant parallel programming techniques. We face an unprecedented challenge as educators: we must revise our curricula dramatically in the next few years, to teach new parallel programming methods which are, as yet, unidentified. Programming languages should have an important rôle to play in meeting this challenge, and it is our job—by evangelizing our students and colleagues—to see to it that they do.