Mathematics Is Imprecise (Extended Abstract)

Prabhakar Ragde
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
plragde@uwaterloo.ca

We commonly think of mathematics as bringing precision to application domains, but its relationship with computer science is more complex. This experience report on the use of Racket and Haskell to teach a required first university CS course to students with very good mathematical skills focusses on the ways that programming forces one to get the details right, with consequent benefits in the mathematical domain, and how imprecision in mathematical abstractions and notation can work to the benefit of beginning programmers, if handled carefully.

1 Introduction

The University of Waterloo has the world's largest Faculty of Mathematics, with six departments (including a School of Computer Science), over 200 faculty members, and about 1400 undergraduate students entering each year. These students are required to take two CS courses, and they have a choice of three streams. Two are aimed at majors and non-majors respectively; the third is aimed at students with high mathematical aptitude. A similar high-aptitude stream has existed for the two required math sequences (Calculus and Algebra) for decades, but the CS advanced stream is relatively recent, starting with a single accelerated course in 2008 and moving to a two-course sequence in 2011-2012.

The CS advanced stream currently has a target of 50-75 students per year. Admission is by instructor consent, or by scoring sufficiently high on math or programming contests at the senior high-school level. Consequently, a significant fraction (sometimes more than half) of the students taking the advanced stream are not CS majors (and many who are will take a second major in one of the other Math departments). Functional programming, with its elegant abstractions, is well-suited to provide the right sort of challenges for such students.

In this paper, I'll describe some unusual choices that I made, some techniques that seemed to find favour with students, and some issues that remain to be overcome.

2 The roles of Racket and Haskell

Our major and non-major streams use Racket [5] exclusively in the first course, with the How To Design Programs (HtDP) textbook [2] and the Program By Design (PBD) methodology [4]. (The second courses make a gradual transition to C for majors and Python for non-majors.) Because of migration between streams, the advanced stream should not stray too far from this model, but some deviation is possible. The rest of the curriculum ignores functional programming, so upward compatibility is not an issue.

Among institutions using a functional-first approach, Haskell [3] is a popular choice. Haskell is an elegant and highly-expressive language, and its proximity to mathematics would make it a natural choice

Preliminary Report. Final version to appear in:
TFPIE

for students in the advanced stream. Thus it may surprise you to learn that while the first lecture module uses Haskell, and it is used throughout the advanced course, students program exclusively in Racket. Haskell is used as functional pseudocode.

Conventional pseudocode, at its best, resembles untyped Pascal: imperative, with loops manipulating arrays and pointers. In comparison, code written in a functional language is transparent enough that it often serves the same purpose. However, there are degrees of transparency, and some functional languages are more readable than others. Haskell, with patterns in function definitions and local bindings, and infix notation, is rich in expressivity. But students programming in Haskell have to learn about operator precedence, and have to learn the pattern language. Mistakes in these areas often manifest themselves as type errors, aggravated by type inference making interpretations that the student does not yet know enough to deliberately intend or avoid, and compiler errors designed to inform the expert. Well-written Haskell code is a joy to read; poorly-written, incorrect Haskell code can be a nightmare for the beginner to fix.

Racket's uniform, parenthesized syntax (inherited from Lisp and Scheme) is by contrast relatively straightforward; the teaching language subsets implemented by the DrRacket IDE limit student errors that produce "meaningful nonsense"; and testing is lightweight, facilitating adherence to the PBD methodology. Seeing two languages from the beginning lets students distinguish between concepts and surface syntax, while programming in just one minimizes confusion. When I introduce more advanced features available in full Racket (such as pattern matching and macros), students can appreciate them and put them to use immediately.

Following Hutton, who in his textbook Programming In Haskell does not even mention lazy evaluation until the penultimate chapter, I am vague about the computational model of Haskell. But a precise computational model is important in debugging, and the simplified reduction semantics that HtDP presents is quite useful, especially combined with the DrRacket tool (the Stepper) that illustrates it on student code.

3 Computation and proof

Here is the first program that the students see.

```
data Nat = Z | S Nat
plus x Z     = x
plus x (S y) = S (plus x y)
```

HtDP distinguishes three kinds of recursion: structural recursion, where the structure of the code mirrors a recursive data definition (as above); accumulative recursion, where structural handling of one or more parameters is augmented by allowing other parameters to accumulate information from earlier in the computation (illustrated below); and generative recursion, where the arguments in a recursive application are "generated" from the data (early examples include GCD and Quicksort).

A computational treatment of Peano arithmetic respects this hierarchy while immediately serving notice that mathematical assumptions will be challenged and details are important. This also allows me to address proof early, notably "for all" statements. What I describe as "the anonymous method" is classic $\forall$-introduction, but it is inadequate. Attempts to prove, for example, commutativity or associativity quickly lead, via equational reasoning on small examples, to the concept of induction. They see induction in the Algebra sequence (immediately in the advanced stream, after a few weeks in the regular stream) but it is not applied to "fundamental" properties of arithmetic, which are taken for granted.

Discussing proofs by induction also reinforces the idea that structural recursion, should it work for the problem at hand, is a preferable approach, as it is easier to reason about, even informally. We look at a non-structurally-recursive version of addition:

```
data Nat = Z | S Nat
add x Z     = x
add x (S y) = add (S x) y
```

This function uses accumulative recursion (the first parameter is an accumulator), and it is harder to prove properties such as commutativity and associativity. In fact, the easiest way to do this is to prove that `add` is equivalent to `plus`, and then prove the properties for `plus`.

Surprisingly, this situation carries over into many early uses of accumulative recursion, such as to add up or reverse a list. An accumulator resembles a loop variable, and the correspondence is direct in the case of tail recursion. The conventional approach to proving correctness is to specify a loop invariant that is then proved by induction on the number of iterations (or, in the functional case, the number of times the recursive function is applied). But I found that a direct proof (by structural induction) that the accumulatively-recursive function was equivalent to the structurally-recursive version turned out, in many cases, to be easier. The reason is that many of the standard proofs of loop invariants involve definitions that use notation whose properties themselves require recursive definitions and proofs.

As an example, consider adding up a list.

```
sumh [] acc     = acc
sumh (x:xs) acc = sumh xs (x+acc)
sumlist2 xs = sumh xs 0
```

An informal proof of correctness of `sumlist2`, based on Hoare logic, would use an invariant such as "In every application of the form `sumh ys acc`, the sum of the whole list is equal to `acc` plus the sum of the `ys`." But there really is no better formalization of "the sum of" than the structurally recursive definition of `sumlist`:

```
sumlist []     = 0
sumlist (x:xs) = x + sumlist xs
```

At which point it is easier and more straightforward to prove "For all `xs`, for all `acc`, `sumh xs acc = acc + sumlist xs`" by structural induction on `xs`.

The strong connection between structural recursion and structural induction makes it possible to discuss rigourous proofs of correctness in a way that is not overwhelming (as it typically is for Hoare logic), and this extends to most uses of accumulative recursion. Traditional invariants are easier to work with in the absence of mutation, but still more work than the direct approach of structural induction. Strong induction, or induction on time or number of recursive applications, can thus be deferred until generative recursion is taught.

4 Analyzing efficiency

A traditional CS1-CS2 approach saves algorithm analysis and order notation for the second course, leaving the first one to concentrate on the mechanics of programming. However, efficiency influences not only the design of imperative languages, but the ways in which elementary programming techniques are taught. Efficiency is also the elephant in the room in a functional-first approach, though the source of the problem is different. A structurally-recursive computation where it is natural to repeat a subexpression

involving a recursive application (for example, finding the maximum of a nonempty list) leads to an exponential-time implementation, with noticeable slowdown even on relatively small instances. The solution is awkward unless local variables are prematurely introduced, and even then, the motivation has to be acknowledged. Accumulative recursion is also primarily motivated by efficiency.

Our major stream also postpones order notation to the second course, while reluctantly acknowledging the elephant where necessary. The advanced stream, however, introduces order notation early. An intuitive illustration of time and space complexity is easy with our first example of unary numbers, but a module on representing sets of integers by both unordered and ordered lists more carefully exercises the ideas.

Order notation shares pedagogical pitfalls with another topic commonly introduced in first year, limits in calculus. Both concepts have precise definitions involving nested, alternating quantifiers, but students are encouraged to manipulate them intuitively in a quasi-algebraic fashion. A typical early assignment involves questions like "Prove that $6n^2 - 9n - 7 \mathit{is} O(n^2)$." As with epsilon-delta proofs, not only do weaker students turn the crank on the form without much understanding, but questions like this have little to do with subsequent use of the ideas. The situation is worse with order notation (more quantifiers, discrete domains that are difficult to visualize).

The analysis of imperative programs at the first-year level is little more than adding running times for sequential blocks and multiplying for loop repetitions; in other words, it is compositional based on program structure. The obvious approach for recursive functions involves recurrences. But solving recurrences is not easy, even with standard practices such as omitting inconvenient floors and ceilings, and setting up recurrences is not straightforward, either. I have found that a compositional approach works for many recursive functions, with the aid of a table.

The tabular method works for functions that use structural or accumulative recursion, as long as recursive applications do not "overlap". Racket functions tend to be a `cond` at the top level, and the table has one row for each question-answer pair (equivalently, for each pattern plus guard in a Haskell multipart definition). The row contains entries for the number of times the question is asked (as a function of the "size" of the argument), the cost of asking the question (nearly always constant), the number of times the answer is evaluated, and the cost of evaluating the answer (apart from recursive applications). These are multiplied in pairs and added to give the cost of the row, and then these costs are added up over all rows. Here is how the table might look for `sumlist` (where n is the length of the list argument):

Row	#Q	time Q	#A	time A	total
1	$n+1$	$O(1)$	1	$O(1)$	$O(n)$
2	n	$O(1)$	n	$O(1)$	$O(n)$
					$O(n)$

This does not avoid recurrences, which are necessary to explain, for example, the exponential-time behaviour of naïve list-maximum, but it limits their use.

The imprecision of order notation, both in terms of information lost and in terms of intuitive or fuzzy understanding in the heads of students, lets us present motivation and rationales for common practices.

5 Efficient representations

The approach I take to the efficient representation of integers starts by arguing that the problem with unary arithmetic stems from the use of a single data constructor with interpretation S:$n \mapsto n+1$. Using two data constructors, we must decide on interpretations.

```
data Nat = Z | A Nat | B Nat
```

Effective decoding requires that the range of the two interpretations partition the positive integers. "Dealing out" the positive integers suggests an odd-even split, with interpretations A:$n \mapsto 2n$ and B:$n \mapsto 2n+1$. We then have binary representation (with the rightmost bit outermost), with unique representation enforced by a rule that A should not be applied to Z (corresponding to the omission of leading zeroes).

We cover addition and multiplication in the new representation, and analyze them. This leads to an interesting side effect. Mutual recursion is introduced in HtDP in the context of trees of arbitrary fan-out. But it arises naturally with the linear structures used here. A first attempt at addition might look like this:

```
add x Z = x
add Z y = y

add (A x) (A y) = A (add x y)
add (A x) (B y) = B (add x y)
add (B x) (A y) = B (add x y)
add (B x) (B y) = A (add1 (add x y))

add1 Z = B Z
add1 (A x) = B x
add1 (B x) = A (add1 x)
```

A naïve analysis of add first analyzes add1, which takes $O(s)$ time on a number of size s (number of data constructors used in the representation). Then add takes time $O(m^2)$, where m is the size of the larger argument. However, this analysis is too pessimistic. add actually takes time $O(m)$, since the total work done by all applications of add1 is $O(m)$, not just one application. This is because the recursion in add1 stops when an A is encountered, but the result of applying add1 in add is wrapped in an A.

But this argument is subtle and difficult to comprehend. It is better to replace the last line in the definition of add with an application of an "add plus one" function.

```
add (B x) (B y) = A (addp x y)
```

We then develop addp, which has a similar structure to add, and recursively applies add. It is now easy to see that add has running time linear in the size of the representation, because it (or addp) reduces the size of the arguments at each step.

Another surprising benefit of this approach is that we can easily represent negative numbers simply by introducing the new nullary constructor N, representing -1. The existing rules for add stay the same, and the new ones involving N are easy to work out. The result is isomorphic to two's complement notation, which is normally mystifying to second-year students taking a computer architecture course. Here we have not only a clear explanation but good motivation. The internal representation of numbers in both Racket and Haskell is no longer magic.

This motivates the introduction and exact solution of the recurrence relating an integer n to the size of its representation, which naturally introduces logarithms to the base 2 while making clear the issues of discretization.

Trees are often introduced to mirror structure in data: in HtDP, using family trees, and in our major sequence, using phylogeny trees. An important insight is that introducing tree structure to data not obviously structured in this fashion can yield improvements in efficiency. Unfortunately, the example usually chosen to illustrate this, binary search trees, is not effective at the first-year level. The simplest algorithms are elegant but degenerate to lists in the worst case; there are many versions of balanced search trees, but the invariants are complex and the code lengthy, particularly for deletion.

The above treatment provides a path to an introduction of logarithmic-height binary trees. Consider the problem of representing a sequence of elements so as to allow efficient access to the *i*th element. A list can be viewed as being indexed in unary, with the element of index `Z` stored at the head and the tail containing the sequence of elements of index `S x`, stored in the same fashion but with the common `S` removed from all indices.

Binary representation of numbers suggests storing two subsequences instead of one: the sequence of elements of index `A x`, and the sequence of elements of index `B x`. This leads to a binary tree where an element of index `A x` is accessed by looking for the element of index `x` in the left ("`A`") subtree, and an element of index `B x` is accessed by looking for the element of index `x` in the right ("`B`") subtree. This is just an odd-even test.

But there is a problem, stemming from the lack of unique representation. Not all sequences of `A`'s and `B`'s are possible, since `A` cannot be applied to `Z`. This means that roughly half the nodes (every left child) have no element stored at them. We can avoid this problem by starting the indexing at 1, or, equivalently, retaining indexing starting at 0 but "shifting up" and "shifting down" while deciding direction. In other words, we replace the `A-B` representation with a `C-D` representation, with interpretation $\texttt{C}(n) = \texttt{A}(n+1)-1$ and $\texttt{D}(n) = \texttt{B}(n+1)-1$.

This results in the interpretation $\texttt{C}{:}n \mapsto 2n+1$ and $\texttt{D}{:}n \mapsto 2n+2$. The new `C-D` representation is naturally unique and all sequences are possible, so there are no empty nodes in the tree with "`C`" left subtrees and "`D`" right subtrees. It is easy to show that the tree has depth logarithmic in the total number of elements, that access to the *i*th element takes time $O(\log i)$, and that list operations (cons, first, rest) take logarithmic time by means of very simple purely-functional code. We have rederived the data structure known as a Braun tree [1].

We see that a more mathematical treatment of fundamentals is not in conflict with core computer science content; on the contrary, it supports the content and increases accessibility by providing sensible explanations for choices.

6 Issues

There is more than enough material to fill a first course with topics approached in a purely functional manner, though the winnowing process is painful. The second course, which needs to move towards mainstream computer science, is more problematic. The advanced sequence shares some issues with the major sequence: the more complicated semantics of mutation; the increased difficulty of testing code written in a primarily imperative language; the confusing syntax, weak or absent abstractions, and lack of good support tools associated with popular languages. Added to these for the advanced sequence are the disappointment associated with the comparative lack of elegance and the relatively low-level nature of problem solving typical with such material. It is not the best advertisement for computer science.

The second course remains a work in progress, with hope sustained by the fact that Racket is a good laboratory for language experimentation. With luck I will soon be able to report on a second course which is as rewarding for students as the first one.

7 Bibliography

References

[1] W. Braun & M. Rem (1983): *A logarithmic implementation of flexible arrays*. Technical Report MR83/4, Eindhoven Institute of Technology.

[2] M. Felleisen, M. Flatt, R. Findler & S. Krishnamurthi (2003): *How To Design Programs*. MIT Press.

[3] (2012): *Haskell*. Available at `http://www.haskell.org`.

[4] (2012): *Program By Design*. Available at `http://www.programbydesign.org`.

[5] (2012): *Racket*. Available at `http://www.racket-lang.org`.