Lifecycle in CS1: Requirements, Domain Analysis, and Implementation

Aamod Sane
FLAME University
Pune, India
aamod.sane@flame.edu.in

Rustom Mody
Magus Inc.
Pune, India
rustompmody@gmail.com

Anuradha Laxminarayan
FLAME University
Pune, India
anuradha.laxminarayan@flame.edu.in

Jayaraman VK
FLAME University
Pune, India
jayaraman.vk@flame.edu.in

ABSTRACT

In practice, programmers start with informal requirements proposed by a customer, make the requirements precise, and implement them in a program that meets the customers needs. It is difficult for a CS1 course to convey a sense of this journey to students, since they need to understand the notional machine, technological artifacts, and the informal to formal transition at the same time.

In this paper we show how to sequence the teaching of programming ideas in such a way that in a short two-month course, novice students learn to develop simple programming projects from requirements in English to a working program. We start by showing that programs are an extension of school algebra using Pugofer, a pedagogical Haskell-like language. Then we present list comprehensions, which lets us create simple programs with lists and tuples as handy "databases". Next we teach requirements analysis for information processing situations such as library management, and create programs that represent domain concepts using data, and behaviors using functions and comprehensions. Students follow this process in their own projects. We end by teaching recursion, motivating it as a way to implement comprehensions.

We have taught courses in this style three times; each time students have delivered projects as well as standard CS1 exams, and rated the course highly. It has been easy to move to imperative programming in later courses. We believe that our approach can be readily replicated, giving students the benefit of experiencing key elements of the software lifecycle early in their CS education.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Software and its engineering** → **Functional languages**; **Software design techniques**; **Software prototyping**; **Requirements analysis**.

ITiCSE 2022, July 8–13, 2022, Dublin, Ireland

ACM ISBN 978-1-4503-9201-3/22/07…$15.00
https://doi.org/10.1145/3502718.3524798

KEYWORDS

CS1, introductory programming, requirements elicitation, domain analysis, functional programming, software lifecycle

ACM Reference Format:
Aamod Sane, Rustom Mody, Anuradha Laxminarayan, and Jayaraman VK. 2022. Lifecycle in CS1: Requirements, Domain Analysis, and Implementation. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol 1 (ITiCSE 2022), July 8–13, 2022, Dublin, Ireland*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3502718.3524798

1 INTRODUCTION

In industrial projects, the lifecycle of a program begins with gathering requirements and ends with the delivery of tested code. The cycle is repeated as needed for modular parts of programs and changes in requirements. But usually a CS1 course cannot take students through this cycle; they are still learning how to think with algorithms and data structures, how the underlying notional machine renders algorithms executable, and coping with the vagaries of the language and the environment. Thus, almost all CS1 courses omit "the most important function [, the] extraction and refinement of [requirements]", (Brooks, 'No Silver Bullet' [3]), and can say little about the realities of a program lifecycle.

It is important for students to get a sense of what it is like to create real programs early on. Understanding how informal conceptions are made formal is a key aspect of becoming a good programmer as well as progressing to the later stages of their career. If students see how requirements arise and how they are materialized in programs, they obtain a map for their overall journey that supplies later courses with a rationale. Working with a miniature version of a real situation helps teach skills like deciding concept representations, decomposing problems, and prioritizing subproblems to solve by their utility. Students who do not pursue CS also benefit from experiencing what it is to develop a complete program. Should their future career involve roles such as business analysis, they can rely on this map to locate themselves in the new role.

This position paper shows how even in a short, *two month* course we can tackle the entire lifecycle using *Pugofer* [32]. Pugofer is a simpler Haskell that avoids complicated types and mystifying error messages, and adds pedagogy oriented features. Its core elements can be taught in a couple weeks to novices, provided they are reasonably comfortable with, or at least do not dislike, school algebra.

Pugofer variables are algebraic, equations define functions, and substitution is execution. Consequently, we can present Pugofer as an algebraic calculator to students, spending little time on technicalities like environments or syntax.

Our next choice is crucial to the success of our course. We rely on the current student generations' familiarity with sophisticated apps, inviting them to '*imagine your app*'. We ask *students* to *elicit requirements* of tasks similar to those implemented by apps they are familiar with. Then we show them how to analyze and model the information structure of what is going on in the apps they have imagined. In the weeks prior, we have taught them sufficient elements of Pugofer so that they can model the information structure as well as changes to the structure necessary in applications. They write small demos of their work, and write tests as well as invariants of the information model. Students thereby build an effective theory of the domain in the sense of Naur [35] that underpins the logical structure of their apps.

An interesting simplification allows our students to easily "demo" their applications: we rely entirely on the Pugofer REPL[1] with data literals as IO. Students realize that they are interacting with the inner logic of the app, and even students with prior experience adjust to returning results instead of calling `print` all the time. We distinguish between computation and IO, and show how data literals can be presented as HTML or other forms as needed.

Students, both CS and non-CS majors, report that this journey from requirements to a working program stood out as a memorable experience of their first year.

Organization of the paper. Sections 2 to 5 of the paper walk the reader along the instructional path. In our implementation, steps 1-4 take 3 weeks, 5-6 are 2 weeks, and 7-8 use 3 weeks.

(1) Using knowledge of school algebra, introduce the basic concepts of variables, values and operators, and then teach new ideas such as types, definitional equality vs value equality, booleans as values etc.
(2) Show how equations can define functions as in algebra, and recall the use of graphical plots to describe functions. Then use plots to argue that a function can have a value as an object, a set of input-output pairs called the graph, distinct from its "formula".
(3) Use multiple argument functions to introduce higher-order functions, once again using graph models to understand how functions can be passed and returned.
(4) Develop tuples and lists, and show how the ideas of operations in school algebra can be extended for compound types. (Steps 1 to 4 are in Sec. 2).
(5) Introduce the idea of requirements elicitation, by asking students how to operate a book library. Teach prioritization of requirements, and introduce the idea of a *minimum viable product* (MVP) as a starting point for an implementation that they can grow later (Sec. 3).
(6) Show how concepts in a real-life situation like books, patrons and circulating versus on-shelf books are modelled using compound data structures as databases, and manipulated using list comprehensions as queries (Sec. 3).
(7) Student groups then follow a similar process template: collect requirements, refine and prioritize them, pick concept representations, and end with projects that model apps (Sec. 4).
(8) Concurrently, introduce recursive functions as a way to build list comprehensions (Sec. 5), and show how other interesting algorithms are also definable by recursive equations.

Every step in this path applies students' existing knowledge and then extends it. Sec. 6 locates CS1 in the context of our new curriculum. Sec. 7 presents student feedback. Sec. 8 discusses our *algebra-first* / functional-first choice, Sec. 9 considers tradeoffs, and Sec. 10 presents related work. Sec. 11 summarizes our contributions.

2 FROM SCHOOL ALGEBRA TO PROGRAM STRUCTURES

In the first half of the course, we draw out the similarities of Pugofer to algebra, provide a tangible model of function values, and teach pattern matching to handle compound types. The second half uses this knowledge to teach information modeling and processing.

Types and Errors. Natural language sentences can have spelling errors: "helllo", grammar errors: "goes who there?", and meaning errors: "colorless green ideas sleep furiously" [7]. We explain that in Pugofer, the analogs are syntax errors: "`1ab34`", type errors: "`'c' == 12`", and program errors: "`1/0`". Pugofer verifies spelling and grammar before running a program, and halts on detectable meaning errors. Requirement and logic mistakes are up to the programmer.

Type inference is like dimensional analysis. In any equation in physics, say $s = ut + 1/2gt^2$, if we know units for s and t, by 'balancing' the two sides we can determine the units of g, u etc. We use this model to explain how Pugofer determines types in equations, starting from known types such as the types of '1', '3' and '+'. Pugofer is a calculator that calculates values and types.

```
? 1 + 3
4 : Int
```

Thus students are always made aware of types, but because Pugofer has type inference, they don't need to explicitly deal with types early on. Variables in Pugofer are algebraic, so `x = x + 1` is meaningless, but substitution is meaningful and functions work as in algebra. Thus far, this is standard Haskell programming, but the differences in Pugofer start showing up with the treatment of functions.

Function values are graphs. We then show how functions are defined using equations. The syntax for functions is `f.x` in lieu of the $f(x)$ of algebra. In Pugofer, the dot in `f.x` is an operator called the *Dijkstra-dot* that denotes function application [9, 29–31, 33] (its pedagogical impact will be discussed shortly).

```
? sq.x = x * x
sq : Int -> Int
? sq . 3
9 : Int
```

Our next step is to get across the idea that functions can be values. When we ask students to guess the type of `sq` as a function, a handful of students say the type must be **Int**, thinking of the output value, whereas others say that such a thing is not possible, only

[1] Read-Eval-Print Loop, familiar from Python, Lisp, Haskell, etc.

`sq.x` will be meaningful. It is hard to accept that a function is meaningful even in the absence of its argument.

Then we ask them whether the two functions $f(x) = 3 * x$ and $g(x) = x + x + x$ are the "same", and how one might express the sameness. This motivates an extensional notion of a function as its input output mapping, the graph. The graph 'object' is a tangible entity, unlike other explanations[3] of function values.

We say that the graph $\{(1, 1), (2, 4), (3, 9), \ldots\}$ is the actual underlying object that defines the function `sq`. Here we allude to the school depiction of a function on graph paper by plotting points, and extend it to the set theoretic definition. Then we claim that the infinite table of pairs is equivalent to the real function object in Pugofer[2], represented with formulas and executed by substitution, so that application (dot) is *table lookup*, and the type **Int** -> **Int** is a compact representation of the input-output mapping in the graph.

Dijkstra-dot and curried functions. Having justified functions as standalone objects, we move on to multiple-argument functions. At this time, the Dijkstra-dot lets us apply the following mechanical way of thinking about the interaction of the "dot" operator and the "arrow" type level operator: For every dot (and argument) supplied in an application from left to right, one arrow is removed from the function type from left to right.

```
? isAsciiCode.char.code = ord.char == code
isAsciiCode : Char -> Int -> Bool
? isAsciiCode . 'a'
isAsciiCode.'a' : Int -> Bool
? isAsciiCode . 'a' . 96
False : Bool
```

For students, this gives a mechanical procedure to hold on to while they are still getting familiar with the idea of treating functions as values. A visible application operator makes teaching many ideas simpler; for instance, that application "dot" `(.)` is itself a higher-order function [30].

Graphs explain function inputs. To teach function values as inputs, we once again rely on graphs. We argue that `isAsciiCode` returns functions, so its graph has infinite graphs as parts of its defining tuples,

$$isAsciiCode = \{ (\text{'a'}, \{\ldots (1, False), \ldots (97, True), \ldots\}), \ldots\}$$

and in a similar vein, the function `play`

```
? play.f.x = f.x + f.(2*x)
play : (Int -> Int) -> Int -> Int
```

with its graph as depicted below (assuming square as the input function in the pair shown) is a meaningful object.

$$play = \{ (\{(1, 1), (2, 4), \ldots\}, \{(1, 5), (2, 20), \ldots\}), \ldots \}$$

This is an unusual object, but is still tangible to students[3]; it is followed by examples of substitution instances. Later we use function parameters to relate list comprehensions and higher-order functions (HOFs, Sec. 5).

[2]The occasional enterprising student is told about `sq = \x -> x*x`, and intensional distinctions like complexity.

[3]By way of comparison, [12, 48] motivate function values via abstraction, while [2] claim "Notwithstanding the rather elevated terminology, the idea is very simple and not at all mysterious", and refer to the d/dx operator.

Compound types and generic function types. The next target is to get across the idea of compound types, and that operators on compound types have generic function parameters. For example, `fst`, a function that returns the first element of a tuple, has the following type

```
? fst
fst : (a, b) -> a
```

where `a` and `b` are unknown types, and the only purpose of the letters is to induce a dependence between input and output types.

Once again, graphs help. It seems evident that since `fst` must give the correct type for `(1, 2)`, `('p', 'q')`, `("uv", 1)`, etc. the function graph can be defined as the set

$$\{((1, 2), 1), \ldots, (('p', 'q'), 'p'), \ldots, (("uv", 1), "uv"), \ldots\}$$

and the pairs in this set can be described as (some-type-called-a, some-type-called-b) -> same-type-a.

3 REQUIREMENTS ELICITATION, DOMAIN ANALYSIS, AND ALGEBRA

By the fourth week, students have enough background knowledge to walk through an example lifecycle: gather requirements, choose among them, understand the domain and implement.

Requirement gathering and refinement. We ask students to write — on chat in these online times — one sentence per student describing some attributes of a library of books and some actions associated with libraries. In no time, one ends up with the usual list of the functionality of a library related to book circulation. Attributes not relevant to information modeling also come up, for instance, the need for silence or the fact that there are reading rooms and so forth. We then engage in an exercise to winnow the attributes and actions, ending with a 'minimum viable product' library with three operations: find, borrow, and return books.

Programming is like algebraic modeling. At this point, we ask students to solve classic algebra word problems[4] such as boats-in-streams and soldiers-and-food. We ask them how they chose to model elements of the problems as variables and equations, and then announce that we will do exactly the same thing in developing a program for the library, but with one exception: our equations will involve tuples and lists and their operators, not just numbers.

"Newtonian" Concept Models. The next step is to show how to model the information structure of a library in code. The hard part here is not the model, but rather the idea that quite flimsy seeming entities in a programming language can model significant real world phenomena.

A favorite example of ours that shows the power of the modeling approach is the case of astronomy. In the usual Newtonian model of the solar system, familiar from high-school, planets are modeled as geometrical points and their orbits as elliptic loci. We are so used to this simplification that we think nothing of it, but there is something surpassingly strange about the fact that a geometrical point, an apparently trivial three-parameter entity, is considered a useful model of an object as complex as a planet.

[4]Polya [38] : "the most important single task of [school maths] is to teach the setting up of equations to solve word problems", an analog of our 'informal to formal' theme.

Once students grasp the strangeness and the power of modeling with "flimsy" entities, it seems reasonable that Books are modeled by `Strings`, Patrons can be `Ints`, `Lists` of `Strings` suffice to model books on shelf, and a list of pairs of the form `(Int, String)` is an adequate model of books in circulation. The computing entity and modeled entity are clearly different, but in the appropriate context the model is fine, much the same way that Newton can use a point to model a planet[5].

In their projects, students chose (Sec. 4) sensible concept representations. It appears that Newton's example and the library program get the point across.

Action Models. Besides the entities in the domain, we also need to model actions. We teach list comprehensions as a simple form of loops that suffices to build elementary models of actions.

What does it mean to borrow a book? One definition in terms of the above model is that the book moves from one list – the model of on-shelf books – to another list, the model of in-circulation books. This kind of transformation is easily achieved using a list comprehension.

```
removeBk . rb . bklist = [ bk | bk <- bklist , bk /= rb ]
borrowBk . pat . bbk . ( shelfBks , circulatingBks ) =
    ( removeBk.bbk.shelfBks, (pat,bbk) :: circulatingBks )
```

The comprehension [4, 43] queries the "database" – the booklist – selecting all books except the borrowed book, and the tuple (patron, book) is added ('cons'ed) to the circulation list. Comprehensions are easy to understand: they transform and select or deselect data elements one at a time. For simple information processing requirements, they are sufficient. Students quickly adopt comprehensions for their code.

Domain Invariants. We point out that in our simple model, one needs additional conditions to ensure that the model is suitable. For example, we may have a "law" or invariant that the Books (Strings) in a list must be unique, or figure out a way to represent multiple copies. Similarly, an invariant is required to ensure that one Book may be borrowed by only one Patron at a time. To students, an interesting sounding invariant, analogous to conservation laws of physics, is the "law of conservation of books": a book may be either on the shelf or borrowed. Real-life libraries need to deal with many variations on these invariants, but these are enough to get started, as is evident to the students.

We explicitly model the invariants as predicates in Pugofer, and in their projects ask students to investigate similar laws.

4 STUDENTS INVENT THEIR OWN 'APPS'

Once students have seen how the library program is built, we spell out the steps involved as a *process template.*

(1) Gather requirements and distill into a minimum viable product.
(2) Ask what are the objects of interest and their operators. Example: a library has book/patron/shelf/circulation objects, and find/borrow/return operators.
(3) Find a suitable program structure to represent the information. Example: The tuple `(Int, String)` represents a borrowed book.
(4) Model actions that change information structure in the domain using functions, e.g., `borrowBk`, with the type[6]

```
borrowBk: Int -> String -> ( [String],
    [(Int, String)] ) -> ( [String], [(Int, String)] )
```

(5) Describe domain invariants using predicates, e.g.,

```
lawConservationOfBks : ( [String], [(Int, String)] )
-> ( [String], [(Int, String)] ) -> Bool
```

checks the invariance of books given the input and output of the operations borrow and return.

We ask students to imagine their apps, and build them using the above process. Here are some excerpts from apps imagined by two student groups.

Student Group 1: Sweet Shop App. One group collected requirements for a sweet shop management app, and refined them into the following 'minimum viable' version.

```
 --- Information ---
-Paying for the order
-Calculating the stock of the items in the shop
-Delivery confirmation system for orders
---The real world objects of interest --
-Food items
-Customers
-Employees
---Functions/Operators ---
-Order
-Delivery confirmation system
-Payment confirmation
--- Laws --
Law of marginal utility
```

As you can see, in the initial submission, students used as "invariants" a law they had heard of in economics classes, the "law of marginal utility". After discussion, they changed them to include a law to check that "if an order is fulfilled, there must be some profit". Here is a snippet from their code:

```
-- starting inventory
originventory = [("A",100) , ("B",100), ... ]
-- auxiliary functions
geteachitem.item.inventory = if item == fst .( head.inventory) ...
-- actual functions
custorder.x.y.z.inventory = if geteachitem.x.inventory ...
```

To our delight, students had naturally created auxiliary functions to organize their programming, as they had seen the teacher do. Helper functions are needed in any case as we do not have time to teach scope and local variables, so as Ramsey [39] observes, compositional solutions are encouraged.

Student Group 2: Ride Sharing App. The second project models an app for a ride-sharing service. The following is taken from their requirements document.

[5]Fans of Newton will note that Newton did not casually choose to model planets as points, but proved his famous Shell theorem to show that such a model is more than a mere approximation.

[6]We mention declared types similar to `type Book = String`, records etc., but as discussed in Sec. 8 postpone their use.

```
-- Information in the domain --
Every driver has a profile set up.
Every rider selects payment option before the ride.
-- Domain Analysis --
Objects: Drivers, Riders, ...
Operators: Booking, Riding, Confirming ride is over..
Laws: Drivers are either in the occupied list
      or in the non occupied list
```

Their code declares the shape of structures, provides some functions, and mentions domain invariants (conservation law of taxis).

```
-- Drivers, Riders are strings
-- Occupied and Unoccupied drivers will be differentiated
-- via lists, olist and ulist
-- a driver cannot be in both lists (conservation law of taxis)
--- Ancillary Functions
 ratingfilter . dlist . frtg = [(drv, drtg, ou) | ...
-- function2 - implemented after the ride is over,
function2 .dr .rd . dlist . rlist . grrtg .gdrtg = ...
```

Notice that this group of students named the ancillary function[7], but not the primary one! These examples will help us in later years to discuss code quality. For the term course, we are satisfied that their fundamental approach is sound: they are decomposing the problem, writing ancillary functions, and composing overall program behavior.

For novices, many with nominal experience of programming in high-school, this level of thinking about code, and going from requirements to the corresponding code after a short eight-week term course is remarkable.

5 ENDING WITH RECURSION AND HOFS

We introduce recursive functions as well as the HOFs (higher-order functions) map and filter as two ways to build list comprehensions, using as examples problems that students have seen earlier.

```
removeBk . rb . [] = []
removeBk . rb . (bk :: bks) = if rb == bk
   then removeBk.rb.bks else bk :: removeBk.rb.bks
```

We then move on to show that recursive equations alone suffice to define functions for problems such as converting strings of digits to integers, simple searches, rainfall [46], max/min etc.

Our assignments and exams work through familiar problems: write a function to "join a list of words with commas in between", "add mixed fractions represented using tuples" etc. Students end up with some of the usual CS1 skills, as well as the unusual experience of having glimpsed the software lifecycle.

6 COURSE AND CURRICULUM

CS1 is the first course of a new curriculum, *Computer Science for the Smartphone Generation* [41]. We teach three term courses during a "sampler" first year: CS1, Computational Modeling (CM) (Python), and Hardware (HD) (Arduino/C). In CM, students work by themselves through a few sections of Downey's friendly book 'Think Python' [10]. We focus on teaching the design of loops using induction to reason about traces [16, 17, 19]. In HD, student groups write C programs to build Arduino projects. We present simple URM-like [45] assembler programs and teach hand-compilation of C statements.

[7]The functions have a very large number of explicit arguments since we do not have time to discuss records.

7 FEEDBACK FROM STUDENTS

CS1 is taught thrice to classes of 20, of whom 25% choose CS, 15% Maths, 20% Econ, 20% Business, rest are various other majors. We interviewed 23 students out of 60 in groups of 4 or so each with the following quantitative and freeform questions.

Q1	What is programming, prior to CS1?
Q2	What is programming, after CS1?
Q3	What were the memorable aspects of CS1?
Q4	How helpful were the references to school algebra?
Q5	How important was the modeling project to you?
Q6	How comfortable are you in selecting concept representations?

Here is the overall structure and distribution of the answers. Answers for Q1 included

43%	Syntax and knowledge of languages
39%	Some algorithms we had memorized
9%	I had built a Python app, at school or with family
9%	No idea; mostly typing fast, like movie hackers

Q2 indicates that programmatic thinking is getting through.

57%	How to organize ideas and understand details
26%	Its very logical, like maths but different
17%	I think I could build a program that I want

For Q3, the answers were

48%	Domain analysis and concept representation
31%	I liked how mathematical and logical it all is
17%	I liked the course
4%	It was ok, different from school

For the quantitative questions, the answers were

	Very	Neutral	Not very
Q4	65%	26%	9%
Q5	91%	9%	
Q6	17%	70%	13%

Students who had done many years of programming before did not care about Q4, but new programmers, or those who had not enjoyed school programming, did care. Q6 indicates that in their self estimation, students were uncertain about representations. In future work, we aim to address this issue by studying how students choose representations and where they encounter problems.

Q5 had nearly universal support, other than 'neutral' students who had apprenticed with experienced family. Quite a few said that the end-to-end project made techniques of programming seem purposeful, rather than, in the words of a student, "a scattered collection of tools for solving classroom problems".

8 ALGEBRA-FIRST VS IMPERATIVE AND OO

Pugofer enables us to present programming as a continuation of school algebra[8], minimizing time spent on the language itself. Its laconic interaction style feels like an algebraic calculator. We show students (Sec. 3) how program design is algebra-like in the way one selects representations and sets up equations. These similarities link familiar skills in algebra to programming, which may explain why two months yield fair coverage of the software lifecycle.

Unlike familiar algebra, imperative programming requires that students learn the new skill of reasoning over traces (see p.18, 55, 56 of [44]) before they can express programs. So it seems unlikely that an imperative language can be used for the 'lifecycle-in-CS1 in one term' target (although two terms might suffice).

We do use Python in our next term course (Sec. 6), teaching reasoning with traces. Exam performance suggests that the 'imperative after algebraic-equational' approach works well.

OO-first courses require two new skills for programming, ontological OO-modeling and working with traces[9]. A further difficulty is that they cannot directly use formalization methods learnt earlier, since Newtonian (Sec. 3) parametric school models like '$F = ma$' are far removed from the explicit ontological style. In contrast, Pugofer's 'programs as equations over tuples, ints, strings' matches school formalization, reducing the learning curve of the informal to formal journey. After students learn to formalize, we can explain that parametric models reside inside the brain of the beholder while ontological models can be externalized in languages.

While we limit ourselves to parametric models, Sec. 3 & 4 are informed by Naur's 'Programming as Theory Building' [35], EDSL design [21, 25, 26], RDBMS [4], SETL [8, 43], and OOD [11, 13, 53].

9 TRADEOFFS

Our CS1 misses out on imperative programming, but students are expected to take CM (Sec. 6) to address that. We offer a basic CS0 Python programming course for non-technical students.

List comprehensions make loops algebraic, so students can write simple loops that suffice to do meaningful projects from requirements to implementation. But this means they get less time to learn recursion and practice harder algorithms. Even so, the response to Q5 (Sec. 7) and Brooks' requirements as 'essence' [3] strongly suggest that prioritizing requirements analysis is the right tradeoff.

10 RELATED WORK

Programming as modeling approaches start with well specified problems and expect students to modify and implement variations. Hansen and Kristensen [18] (HK) give English problem statements and use SML types and signatures to model them. Bennedsen and Caspersen [1] use UML diagrams, while Kristensen et.al. [27], citing Nygaard, teach "to program is to model" with an eShop app and its UML models. Myers' 'Equations, Models, and Programs' [34] teaches the algebraic/equational view of programming.

We think it is important to begin one step earlier, learning *how to pose problems* by eliciting requirements and refining them, and then formalizing them (Sec. 3, 4) by a process akin to algebraic modeling. We prefer Newtonian parametric models to Objects (Sec. 8).

HK report that students sometimes question the relevance of topics taught. With 'imagine your app' (Sec. 1), students pick *their own* apps and learn to implement the underlying logic; it could be why questions of relevance have not arisen (see last para, Sec. 7).

Felleisen et.al. [12] teach recipes for algorithm and data structure design of specified problems. For us, program design is the journey from the informal to the formal, yielding information structures and algorithms that model actions on the structures. Pugofer is a good fit here, in line with the arguments in Wadler [52] and Turner [49].

FP-first CS1 is considered in [5, 22, 23, 37] and compared in [50, 51]. We use FP to link school algebra to programming (Sec 8).

Chen and Hall [6] present a decomposed problem, and teach how collaborating teams develop components. Razak uses a capstone project [40] with requirements presented as a research paper. Use of incomplete or ambiguous problems is discussed in [14, 47], while Schmidt [42] reports on a 4-day capstone where teachers provide ambiguous requirements and students extract precise versions.

We aim at giving students a taste for *finding* problems by observing their world, and building a theory of the domain [35]. The project is the heart of our course; our students work on it even as they are learning algorithmic tools.

11 CONCLUSION

We have introduced a new target for CS1: to show students the lifecycle of a program starting from eliciting informal requirements and ending up with a formal construct, the program. We show them how to extract essential requirements that define a 'minimum viable product', and end up with a mini-theory of the domain, following Brooks [3] and Naur [35]. Our novel algebra-first approach extends algebraic modeling from school to encompass program design, with a simple process template leading to implementation and testing. Techniques like function values as graphs, compound values as databases and comprehensions as queries link programs to the algebra of sets. We order topics differently, teaching comprehensions first, then using them to motivate recursion. These techniques and topic order plus the simplicity of Pugofer[10] enable a rapid journey through the lifecycle. Furthermore, the transition to imperative programming in the next course seems to work well.

Taking for granted familiarity with apps opens up new possibilities in CS1. Our CS1 discusses requirements and domain analysis so that – like Physics1 – students see the magic behind a key element of everyday life, the app. 'Imagine your app' shows that your wishes can become your programs, once you know how to compose spells; or to be more prosaic, the "soup to nuts" experience of imagining, refining and implementing an app of their choice teaches students what it is really like to create a program.

Validation of our techniques and topic order is forthcoming. Until then, we hope our initial results persuade readers that ideas like lifecycle-in-CS1, graphs as functions, algebra-first, 'imagine your app', etc., are worth adopting.

ACKNOWLEDGMENTS

Thanks to Mathai Joseph, Desmond D'Souza, and Sanjay Chandrasekharan for their critique, suggestions, and discussions.

[8] Upto substitution, not symbolic [24] or equational [15, 20, 36].
[9] Not only traces of variables, but also traces of messages between objects.
[10] Pugofer admits to only half a sin out of the "seven sins" [28], and supports all the features the authors recommend.

REFERENCES

[1] Jens Bennedsen and Michael Caspersen. 2008. Model-driven programming. In *Reflections on the Teaching of Programming*. Springer, 116–129.

[2] Richard Bird and Philip Wadler. 1988. *An introduction to functional programming*. Prentice Hall International (UK) Ltd.

[3] Frederick P Brooks Jr. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* 20 (1987), 10–19.

[4] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. Comprehension syntax. *ACM Sigmod Record* 23, 1 (1994), 87–96.

[5] Manuel MT Chakravarty and Gabriele Keller. 2004. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming* 14, 1 (2004), 113–123.

[6] Wei Kian Chen and Brian R Hall. 2013. Applying software engineering in CS1. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 297–302.

[7] Noam Chomsky. 2009. *Syntactic structures*. De Gruyter Mouton.

[8] Robert Dewar. 2013. SETL and the Evolution of Programming. In *From Linear Operators to Computational Biology*. Springer, 39–46.

[9] E. W. Dijkstra. 2000. The notational conventions I adopted, and why. (July 2000). http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF

[10] Allen Downey. 2012. *Think Python*. O'Reilly Media, Inc.

[11] Desmond F D'Souza and Alan Cameron Wills. 1998. *Objects, Components, and Frameworks with UML: the Catalysis approach*. Addison-Wesley.

[12] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing*. MIT Press.

[13] Robert G Fichman and Chris F Kemerer. 1992. Object-oriented and conventional analysis and design methodologies. *Computer* 25, 10 (1992), 22–39.

[14] Judith L Gersting. 1994. A software engineering "frosting" on a traditional CS-1 course. In *Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*. 233–237.

[15] Albert Gräf. 2011. Pure Language and Library Documentation. (2011). https://github.com/agraef/pure-lang

[16] David Gries. 1982. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming* 2, 3 (1982), 207–214.

[17] David Gries. 2012. *The science of programming*. Springer Science & Business Media.

[18] Michael R Hansen and Jens Thyge Kristensen. 2008. Experiences with functional programming in an introductory curriculum. In *Reflections on the Teaching of Programming*. Springer, 30–46.

[19] Eric CR Hehner. 2012. *A practical theory of programming*. Springer Science & Business Media.

[20] Christoph M Hoffmann and Michael J O'Donnell. 1982. Programming with equations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 1 (1982), 83–112.

[21] Paul Hudak. 1997. Domain-specific languages. *Handbook of programming languages* 3, 39-60 (1997), 21.

[22] John Hughes. 2008. Experiences from teaching functional programming at Chalmers. *ACM Sigplan Notices* 43, 11 (2008), 77–80.

[23] Stef Joosten, Klaas Van Den Berg, and Gerrit Van Der Hoeven. 1993. Teaching functional programming to first-year students. *Journal of Functional Programming* 3, 1 (1993), 49–65.

[24] David Joyner, Ondřej Čertík, Aaron Meurer, and Brian E Granger. 2012. Open source computer algebra systems: SymPy. *ACM Communications in Computer Algebra* 45, 3/4 (2012), 225–234.

[25] Sam Kamin. 1998. An implementation-oriented semantics of Wadler's pretty-printing combinators. (1998). https://www.researchgate.net/publication/2721546

[26] Samuel N Kamin and David Hyatt. 1997. A Special-Purpose Language for Picture-Drawing. In *DSL'97*. Usenix, 23–33.

[27] Bent Bruun Kristensen, Palle Nowack, and Michael Caspersen. 2016. "To Program is To Model": Software Development is Stepwise Improvement of Models. In *International Conference on Computer Science Education Innovation & Technology (CSEIT). Proceedings*. Global Science and Technology Forum, 81.

[28] Linda McIver and Damian Conway. 1996. Seven deadly sins of introductory programming language design. In *Proceedings 1996 International Conference Software Engineering: Education and Practice*. IEEE, 309–316.

[29] Rustom Mody. 1995. Haskell to Pug: Motivations and Syntax. (1995). http://github.com/rusimody/pugofer/tree/master/techreports/pug-a-teachers-haskell.pdf

[30] Rustom Mody. 2004. A Thought Dialogue with Edsger Dijkstra. (2004). http://github.com/rusimody/pugofer/tree/master/techreports/ewd-dot-dialogue.pdf

[31] Rustom Mody. 2013. Applying SI on SICP. (2013). http://blog.languager.org/2013/08/applying-si-on-sicp.html

[32] Rustom Mody. 2014. Pugofer: A Platonic Universe that is Good For Equational Reasoning. (2014). http://blog.languager.org/2014/09/pugofer.html

[33] Rustom Mody and Anuradha Laxminarayan. 2012. Doting On the Dot. (2012). http://github.com/rusimody/pugofer/tree/master/techreports/doting-on-the-dot.pdf

[34] Thomas J Myers. 1988. *Equations, Models, and Programs: A Mathematical Introduction to Computer Science*. Prentice Hall.

[35] Peter Naur. 1985. Programming as theory building. *Microprocessing and microprogramming* 15, 5 (1985), 253–261.

[36] Michael J O'Donnell. 1998. Equational logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming* 5 (1998), 69–161.

[37] Rex Page. 1997. Selling Haskell for CS1 in the USA. (1997). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.5278&rep=rep1&type=pdf

[38] George Polya. 1962. *Mathematical discovery, 1962* (combined ed.). John Wiley & Sons, Chapter 2, 59.

[39] Norman Ramsey. 2014. On teaching *How to Design Programs*. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. 153–166.

[40] Saquib Razak. 2013. A case for course capstone projects in CS1. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 693–698.

[41] Aamod Sane, Jayaraman VK, Anuradha Laxminarayan, Prateek Shah, and Kaushik Gopalan. 2021. Computer Science for the Smartphone Generation (Extended Abstract). In *14th ACM Compute India Conference*. https://event.india.acm.org/Compute/2021/

[42] Jessica Young Schmidt. 2020. Reviewing CS1 materials through a collaborative software engineering exercise: An experience report. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 379–385.

[43] Jacob T Schwartz, Robert BK Dewar, Edward Dubinsky, and Edith Schonberg. 2012. *Programming with sets: An introduction to SETL*. Springer Science & Business Media.

[44] Robert Sedgewick and Kevin Wayne. 2017. *Introduction to programming in Java: an interdisciplinary approach*. Addison-Wesley Professional.

[45] John C Shepherdson and Howard E Sturgis. 1963. Computability of recursive functions. *Journal of the ACM (JACM)* 10, 2 (1963), 217–255.

[46] Elliot Soloway. 1984. What do novices know about programming. *Directions in Human-Computer Interaction* (1984).

[47] Madalene Spezialetti. 2016. Thinking about asking: Encouraging a questioning approach to requirements gathering and problem solving. In *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–4.

[48] Simon Thompson. 2011. *Haskell: the craft of functional programming*. Addison-Wesley Publishing Company.

[49] David A Turner. 1984. Functional programs as executable specifications. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 312, 1522 (1984), 363–388.

[50] Peter Van Roy, Joe Armstrong, Matthew Flatt, and Boris Magnusson. 2003. The role of language paradigms in teaching programming. *ACM SIGCSE Bulletin* 35, 1 (2003), 269–270.

[51] Milena Vujošević-Janicic and Dušan Tošic. 2008. The role of programming paradigms in the first programming courses. *The Teaching of Mathematics* 11, 2 (2008), 63–83.

[52] Philip Wadler. 1987. A critique of Abelson and Sussman or why calculating is better than scheming. *ACM SIGPLAN Notices* 22, 3 (1987), 83–94.

[53] Rebecca Wirfs-Brock and Alan McKean. 2003. *Object design: roles, responsibilities, and collaborations*. Addison-Wesley Professional.