Y790-32707 - HW #2: Reverse Outlining

Caner Derici

Reverse Outline of : **The Structure and Interpretation of the Computer Science Curriculum [1]**

Abstract

1. Despite the fame and glory of the book *Structure and Interpretation of Computer Programs (SICP)* and its influence on the introductory computing courses, Scheme and functional programming have lost their good impression due to shortcomings of the book and the quaintness of Scheme.
2. Proposing an alternative approach for the role of functional programming in the first-year curriculum, and introducing a new book, and discussing how the approach provides a well preparation for a course on object-oriented programming.

1 History and critique

1. The book SICP liberated the introductory courses from to be designed around the syntax of a famous programming language, to be focused on the study of important ideas in computing such as functional abstraction, data abstraction, streams, data-directed programming, etc.
2. Quickly both the course idea and the Scheme language became the first choice of many universities.
3. In early 90s, the influence of SICP and Scheme started to decay due to some complaints (e.g. instructors saying SICP is too hard, or programmers saying Scheme was too different than mainstream languages).
4. This paper addresses the complaints and tries to overcome the problems of SICP approach. Paper structure is outlined.

2 Structure

2.1 Solving constraints

1. A computing curriculum should produce programmers and software engineers capable of adapting to the ever-evolving nature of the field.
2. It's difficult to impose this goal to the first-year course due to a range of unrelated constraints (e.g. Faculty colleagues want a specific industry language).
3. Some faculty demand that the first course should teach a language that'll be used in upstream courses.
4. First-year students (and/or parents) come with strong, preconvinced notions about programming and computing (e.g. latest tech industry trends disucced in a magazine).
5. The state of the first year students' education also poses some constraints.
6. Students come with a wide range of expectations, but almost everyone expects that the college will help them find internships and professional positions.
7. Satisfying the primary goal subject to these constraints is hard. We need to choose a programming language and introduce programming ASAP in the curriculum, but we also need to convey the choices with good reasons.
8. Proposing a solution by having a second look on the goal and constraints. Curriculum cannot be based on the latest industry trends, as it changes too fast. Still the students should be prepared for the industry. We need to concentrate on the first-summer (internship), and the last-year (interview for positions).

9. We need to concentrate on teaching the principles most of the time, and show how these principles apply to the real world during the second semester of the first year and last year.

10. Year should start with a heavy emphasis on principles and second semester should illustrate the use of these principles in a currently fashionable programming langauge.

2.2 Principles of programming

1. We should identify the technical principles for the first-year programming course. Good program design habits should be taught.

2. Students must learn to read problem statements carefully (e.g. identify classes of data that play a role) and rewrite these into useful pieces (e.g. example data and program usage).

3. Students must learn to organize programs to match for example the identified classes of data. Small changes in the problem translate to small changes in the program (helps industry too, rapid change in requirements).

4. Students must learn to use and walk through the examples they've constructed before writing any code.

5. The last point in particular suggests that functional languages with their natural model-view separation are superior choices for this first year.

2.3 Principles of teaching

1. It's a challange for first-year instructor to understand the teaching priorities. Typically the teaching of language constructs is explicit while the teaching of design principles remains implicit.

2. Conventional approach to teaching programming reverses the natural roles of data and control. We should let data drive the syllabus. This would force students to understand how to go from data to design explicitly, and pick up language constructs implicitly.

3. Most students are active learners. Examples should concentrate on the design instead of specific langauge constructs.

4. Summary. First course should focus on the principles of program design, and should avoid any kind of distractions from these principles (e.g. picking problems from a complex domain or use complex language).

3 Interpretation: functional versus object-oriented programming

1. Returning to the choice of programming language. Suggesting that the first semester uses a simple functional langauge and the second semester uses an industrially fashionable language (e.g. Java).

3.1 Functional and object-oriented programming

1. Functional and object-oriented programming share the desired curricular focus on data as the starting point for program design.

2. The computational model of a functional language is a minor extension of that of secondary school algebra. The model of object-oriented computation requires a bit mote sophistication, but it builds upon the model of functional langauges.

3. Using a functional language followed by object-oriented language is thus the natural choice.

3.2 The role of Scheme

1. We picked Scheme langauge as the starting point.
2. Scheme's syntax is simple.
3. Scheme's semantics is easy to understand (the language is a generalization of high school algebra, students doesn't need to think about registers, stacks, memory cells, and other low-level concepts).
4. Scheme is safe. It detects and pinpoints the errors, giving a valuable feedback.
5. Scheme is dynamically typed. We don't need to explain type errors vs syntax errors. We can also informally introduce a type system as sets of values which is more intuitive to the students (and provides a better transition to the second semester object oriented langauge).

3.3 Programming environments

1. It's not only a language choice, the programming environment also matters a lot.
2. Like the langauge, the first programming environment should be a lightweight and easy-to-use tool.
3. We believe that the lack of such a programming environment hurt the SICP approach of teaching and the functional community in general. For this we have produced an environment that supports teaching program design principles with Scheme.

4 Interpretation: teaching design principles

4.1 Structure and Interpretation of Computer Programs

1. SICP covers many important program design ideas.
2. SICP suffers from not looking at the programmers' perspective (e.g. not explaining how and when programmers need a particular idea, why do they need it)
3. While covering important ideas in programming with examples, SICP leaves the program design and the organization of these ideas implicit.
4. SICP also suffers from selecting exercises that uses complex domain knowledge.
5. While these examples may be interesting to the experience students, they're not to the beginners. Students are spending a lot of time on the domain knowledge and often end up confusing domain knowledge and program design knowledge.
6. SICP is great to introduce important concepts in computer science, but it fails to recognize the role of the first course in the curriculum.

4.2 How to Design Programs

1. Introducing the book that addresses SICP's deficiencies along four dimensions. 1) Explicit discussion on program design. 2) Series of teaching languages (subsets of Scheme) with different knowledge levels. 3) Exercises focus on the program design, only a few require any domain knowledge. 4) More accessible forms of domain knowledge than SICP.
2. Chapters and exercises in HtDP focuses on program design, only the "extended exercises" concern domain knowledge.
3. Explicit design knowledge is formulated as the design recipe. Introducing basic design habits that the design recipe enforces.

4. The book contains a series of design recipes for designing different kinds of abstractions over different kinds of data. Each design recipe especially discusses when to use a technique or mode.

5. The recipes also introduce a new distinction into program design: structural vs. generative recursion.

6. Compare the programming with structural and generative recursion over example functions 'insert' and 'kwik'.

7. Distinguishing the two forms of recursion and focusing on the structural case makes our approach scalable to the object-oriented (OO) world.

8. SICP fails to distinguish the two notion of recursion. Failure to recognize structural (recursive) reasoning causes it to omit the discussion of reasoning about classes of data, which is the essence of OO programming.

9. HTDP introduces the idea of iterative refinement for both programs and data separately, which in turn helps students produce complex programs systematically, combining the design recipes with the idea of refinement.

10. HTDP and SICP also vastly differ with regard to the treatment of language syntax. HTDP gradually increases the complexity of the language while also explicitly discussing the need to do so.

11. Finally, HTDP uses domain knowledge differently from SICP. It uses domain knowledge that is within reach of most students.

5 Experience and outlook

1. HTDP approach has been implemented at colleges and high schools. College level (measurable) results are very strong.

2. High School teachers report similar success results. Female students like the HtDP curriculum exceptionally well.

3. HTDP has validated the usefulness of functional programming and functional programming languages in the first programming course. The two keys to success were to tame Scheme into simple teaching languages and to distill well-known functional principles of programming into generally applicable design recipes.

4. Hoping other functional communities can replicate this success in different contexts. Teaching programming in plain Erlang, Haskell, or ML implicitly will not do. We need to understand the role of functional programming in our curricula and the needs of our students.

References

Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S. (2004). The structure and interpretation of the computer science curriculum. Journal of Functional Programming, 14(04), 365-378.