CS 340: Programming Paradigms and Patterns

January 9, 2018

1 Overview

The current introductory programming sequence (CS 115 & CS 116 or CS 201), while successful in teaching students the basic syntax and semantics of an object-oriented, imperative programming language, and how to use the language to solve simple problems, falls short of presenting the full spectrum of useful, essential programming techniques. In particular, there is little attention paid to the theory and practice of *functional programming*, which has grown increasingly important in today's software development landscape.

The benefits of functional programming are significant in the areas of *reasoning and verification*, due to the lack of side effects and the natural pairing of induction and recursion, *abstraction*, owing to the emphasis on functions that operate on compound data as a whole instead of iteration (e.g., `map` and `fold`), and *concurrency*, due to referential transparency and the absence of state mutations.

The benefits of strong, sophisticated type systems are a natural extension and boon to functional languages, and are also absent from the introductory programming sequence.

We propose a new course which will focus on teaching the functional programming paradigm and related techniques, and will help give students the confidence and tools to program in the large [1].

In the next section we list the learning outcomes for the class, and in the following section we present and highlight relevant topics and learning objectives from the 2013 ACM Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.

2 Learning Objectives

- employ data-driven program design techniques

- write well-typed functional programs in the Haskell language
- use higher order functions to build powerful abstractions
- develop specifications for and prove and verify program correctness using rigorous techniques
- apply equational, evaluational, and compositional reasoning techniques
- explore mechanisms for partitioning and managing complexity stemming from side effects and mutation
- identify opportunities for parallelism in code and exploit it by choosing appropriate data structures and function designs

3 ACM Curriculum Guidelines for CS UG Degree Programs

3.1 Relevant Knowledge Areas (KAs)

3.1.1 Programming Languages / Functional Programming

Topics:

[Core-Tier1]

- **Effect-free programming**
  - Function calls have no side effects, facilitating compositional reasoning
  - Variables are immutable, preventing unexpected changes to program data by other code
  - Data can be freely aliased or copied without introducing unintended effects from mutation
- **Processing structured data (e.g., trees) via functions with cases for each data variant**
  - Associated language constructs such as discriminated unions and pattern-matching over them
  - Functions defined over compound data in terms of functions applied to the constituent pieces
- First-class functions (taking, returning, and storing functions)

[Core-Tier2]

- **Function closures** (functions using variables in the enclosing lexical environment)
  - Basic meaning and definition – creating closures at run-time by capturing the environment
  - **Canonical idioms**: call-backs, arguments to iterators, reusable code via function arguments
  - Using a closure to encapsulate data in its environment
  - Currying and partial application
- Defining **higher-order operations on aggregates, especially map, reduce/fold, and filter**

Learning outcomes:

[Core-Tier1]

1. **Write basic algorithms that avoid assigning to mutable state or considering reference equality. [Usage]**
2. **Write useful functions that take and return other functions. [Usage]**
3. **Compare and contrast (1) the procedural/functional approach** (defining a function for each operation with the function body providing a case for each data variant) **and (2) the object-oriented approach** (defining a class for each data variant with the class definition providing a method for each operation). Understand both as defining a matrix of operations and variants. [Assessment] This outcome also appears in PL/Object-Oriented Programming.

[Core-Tier2]

1. **Correctly reason about variables and lexical scope in a program using function closures. [Usage]**
2. **Use functional encapsulation mechanisms such as closures and modular interfaces. [Usage]**
3. **Define and use iterators and other operations on aggregates**, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language. [Usage] This outcome also appears in PL/Object-Oriented Programming.

3.1.2 Programming Languages / Advanced Programming Constructs

Topics:

- **Lazy evaluation and infinite streams**
- **Control Abstractions**: Exception Handling, Continuations, Monads
- Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
- **Metaprogramming: Macros, Generative programming, Model-based development**
- **Module systems**
- **String manipulation via pattern-matching (regular expressions)**
- **Dynamic code evaluation ("eval")**
- **Language support for checking assertions, invariants, and pre/post-conditions**

Learning outcomes:

1. **Use various advanced programming constructs and idioms correctly. [Usage]**
2. **Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity. [Familiarity]**
3. **Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Familiarity]**

3.1.3 Programming Languages / Language Pragmatics

Topics:

- Principles of language design such as orthogonality
- **Evaluation order, precedence, and associativity**
- **Eager vs. delayed evaluation**
- **Defining control and iteration constructs**
- **External calls and system libraries**

Learning outcomes:

1. **Discuss the role of concepts such as orthogonality and well-chosen defaults in language design. [Familiarity]**
2. **Use crisp and objective criteria for evaluating language-design decisions. [Usage]**
3. **Give an example program whose result can differ under different rules for evaluation order, precedence, or associativity. [Usage]**
4. **Show uses of delayed evaluation, such as user-defined control abstractions. [Familiarity]**
5. **Discuss the need for allowing calls to external calls and system libraries and the consequences for language implementation. [Familiarity]**

3.1.4 Software Development Fundamentals / Development Methods

Topics:

- Program comprehension
- Program correctness
  - **Types of errors (syntax, logic, run-time)**
  - **The concept of a specification**
  - **Defensive programming (e.g. secure coding, exception handling)**
  - **Code reviews**
  - **Testing fundamentals and test-case generation**
  - **The role and the use of contracts, including pre- and post-conditions**
  - **Unit testing**
- **Simple refactoring**
- **Modern programming environments**
  - Code search
  - Programming using library components and their APIs
- **Debugging strategies**

- Documentation and program style

Learning outcomes:

1. Trace the execution of a variety of code segments and write summaries of their computations. [Assessment]
2. **Explain why the creation of correct program components is important in the production of high-quality software. [Familiarity]**
3. Identify **common coding errors** that lead to insecure programs (e.g., buffer overflows, memory leaks, malicious code) and apply strategies for avoiding such errors. [Usage]
4. Conduct a personal code review (focused on common coding errors) on a program component using a provided checklist. [Usage]
5. Contribute to a **small-team code review** focused on component correctness. [Usage]
6. Describe how a contract can be used to specify the behavior of a program component. [Familiarity]
7. Refactor a program by identifying opportunities to apply procedural abstraction. [Usage]
8. Apply a variety of strategies to the testing and debugging of simple programs. [Usage]
9. **Construct, execute and debug programs using a modern IDE and associated tools such as unit testing tools and visual debuggers**. [Usage]
10. Construct and debug programs using the standard libraries available with a chosen programming language. [Usage]
11. Analyze the extent to which another programmer's code meets documentation and programming style standards. [Assessment]
12. Apply consistent documentation and program style standards that contribute to the readability and maintainability of software. [Usage]

3.1.5 Software Engineering / Tools and Environments

Topics:

- **Software configuration management and version control**
- Release management
- Requirements analysis and design modeling tools
- Testing tools including **static and dynamic analysis tools**
- Programming environments that automate parts of program construction processes (e.g., automated builds)
  - **Continuous integration**
- Tool integration concepts and mechanisms

Learning Outcomes:

1. **Describe the difference between centralized and distributed software configuration management**. [Familiarity]
2. **Describe how version control can be used to help manage software release management. [Familiarity]**
3. **Identify configuration items and use a source code control tool in a small team-based project. [Usage]**
4. **Describe how available static and dynamic test tools can be integrated into the software development environment. [Familiarity]**
5. **Describe the issues that are important in selecting a set of tools for the development of a particular software system**, including tools for requirements tracking, design modeling, implementation, build automation, and testing. [Familiarity]
6. **Demonstrate the capability to use software tools in support of the development of a software product of medium size. [Usage]**

References

[1] Frank DeRemer and Hans Kron. *Programming-in-the-Large versus Programming-in-the-Small.* University of California, Santa Cruz (1975).