Functional Programming

Folding, Multi-paradigm Programming

Joey W. Coleman, Stefan Hallerstede

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

6 may 2014

Admin

Admin items

- FP2 Assignment due: 23:59 Thursday 15 May
  - Task 4 to follow
- Multi-Paradigm Assignment posted
  - Groups due by 12:00 Thursday 8 May
  - Submission due: 23:59 Monday 26 May
  - Demonstration: in class, Tuesday 27 May
- Today: 4 hours of class; Friday: cancelled
  - First two here in 416E
  - Second two in 424E

Functional Assignment 1

- Generally ok
- 4 of 12 need resubmission
  - Resubmit before Friday!
- Some confusion that I'll address

Style

- Put the “then”/“else” branches of an `if` on new lines
- Always put a space before `(`, except when preceded by `(`
- `(+ (+ 1 2) 3)` → `(+ 1 2 3)`
- `(equal? lst '())` versus `(null? lst)`
- Use `letrec` instead of SICP-style nested defines
- Helper functions should be in `letrec`s
- Combine nested `if`/`cond` where possible
- `let` bindings should start on new lines
- `(append (list x) y)` is equivalent to `(cons x y)`, but badly

Efficiency

- Not strictly part of this course, but think about efficiency
- `append` takes $\mathrm{O}(\mathrm{n})$ time, where n is the length of the first list
- `length` on a list is also $\mathrm{O}(\mathrm{n})$ time
- `cons` takes constant time
- List solutions that use `cons` only are (usually) more efficient
- This requires general knowledge of the data structures

Specific Notes

- Task 1 — easiest solution for tailrec is to use `cons` and index backwards
- Task 3 —
- Positive product; what's the difference between:
  - `(foldl (lambda (x y) (* x y)) ...)`
  - `(foldl * ...)`
- Reverse:
  - `(lambda (acc n) (append (list n) acc))`
  - `(lambda (acc el) (cons el acc))`
  - even better: `(flip cons)`
- Map:

```
(foldl (lambda (acc n) (append acc (list (proc n)))) ...)
(reverse (foldl (lambda (acc el) (cons (proc el) acc)) ...))
```

An Unhealthy Obsession with Folding

- The idea of `fold` is more general than just lists
  - . . . though it is horribly named.
- Higher-order programming $\rightarrow$ abstraction of program structure
  - and fold functions are an example of this.
- Let's consider `vector-foldl`, and `tree-foldl`

The Visitor Pattern

- Folds are similar to the standard OO Visitor pattern
- One fold function (visitor template) per data type (class)
- No `apply()` method needed
- Repeat: “design patterns” come from the lack of proper abstraction mechanisms

Multi-Paradigm Development

- Development of a software system in which multiple programming paradigms are used.
- Possibilities:
  - Monolithic system using one language in specific ways
  - Subcomponents use different paradigms + well-defined interfaces
  - Different levels in the system architecture
- Basic idea: fit the paradigm to the problem
- Aside: SQL backends

Advantages

- Components implemented in language suited to the problem
- Productivity gain from not fighting the language
  - Sapir-Whorf/Linguistic relativity
- Possible strong separation of concerns
- Possible increase in modularity

Disadvantages

- “Plumbing” costs
  - Build system fragility
  - Overhead from value/call translations
- Expertise availability
- Poor architectures
- Runtime/Library integration
  - Note that a language is more than just its core

“Plumbing”

- Data representation
  - Very implementation dependent
  - Kawa uses Java classes for each Scheme type
  - Racket has a C interface library
- Runtime integration (main event loops)
  - Where should control rest?
- All bundled into a “FFI” (Foreign Function Interface)

About the Assignment

- java assumed, .Net ok (but!)

Kawa

- Kawa 1.13 — `http://www.gnu.org/software/kawa/`
- Interprets and compiles Scheme to the JVM
  1. Scheme entities are mapped to Java/JVM types
- Eclipse environment — SchemeWay
  - No support from JWC/SHA, but it does exist

Practicalities

- Write Scheme in a `.scm` file
- `java -jar kawa-1.13.jar -C File.scm`
- Produces `File.class`
- Write Java, call the Kawa libraries

Example: Factorial.scm

```
(define factorial
  (letrec ((iter
            (lambda (n r)
              (if (= n 1)
                  r
                  (iter (- n 1) (* r n))))))
    (lambda (n)
      (and (> n 0)
           (iter n 1)))))
```

Example: FactorialTest.java

```
import kawa.standard.Scheme;

class FactorialTest {
    public static void main(String[] args) throws Throwable {
        Scheme.registerEnvironment();
        Scheme scm = new Scheme();

        scm.loadClass("Factorial");

        Object result = scm.eval("(factorial 5)");

        System.out.println(result);
        System.out.println("Done.");
    }
}
```

Example — Putting It Together

- Compile the Scheme:
  `java -jar kawa-1.13.jar -C Factorial.scm`
- Compile the Java:
  `javac -cp kawa-1.13.jar:. FactorialTest.java`
- Run the program:
  `java -cp kawa-1.13.jar:. FactorialTest`

Useful Documentation

- JavaDoc at `http://www.gnu.org/software/kawa/api/`
- Number classes in `gnu.math`
- Runtime classes
  - `kawa.standard.Scheme`
  - `gnu.expr.Language`
- Internals documentation — `http://www.gnu.org/software/kawa/internals/index.html`
  - Especially the *Objects and Values* and *Numbers* sections