Second-Class Modules for Effekt

Master Thesis Presentation

Roman Schulte

Agenda

I) Modularity

II) Modules in Effekt

III) Effects & Modules

IV) Outlook

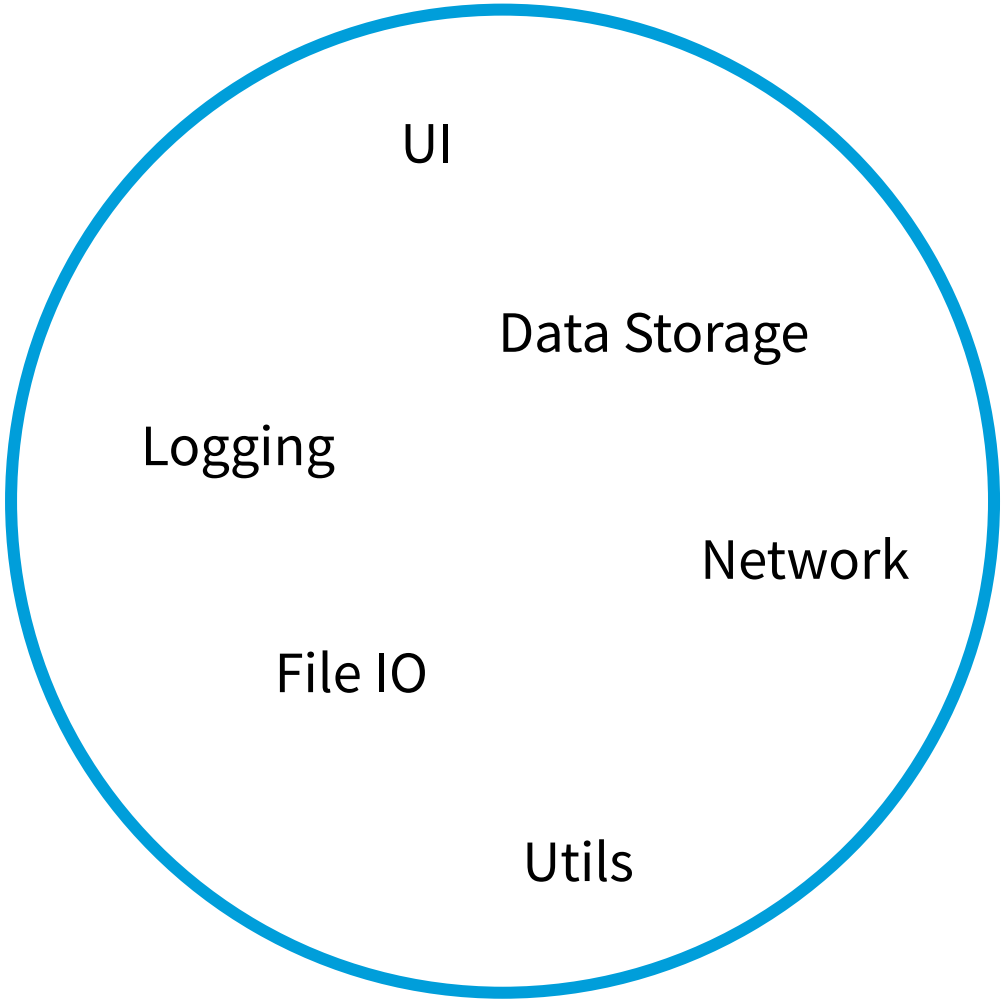
Agenda

I) Modularity

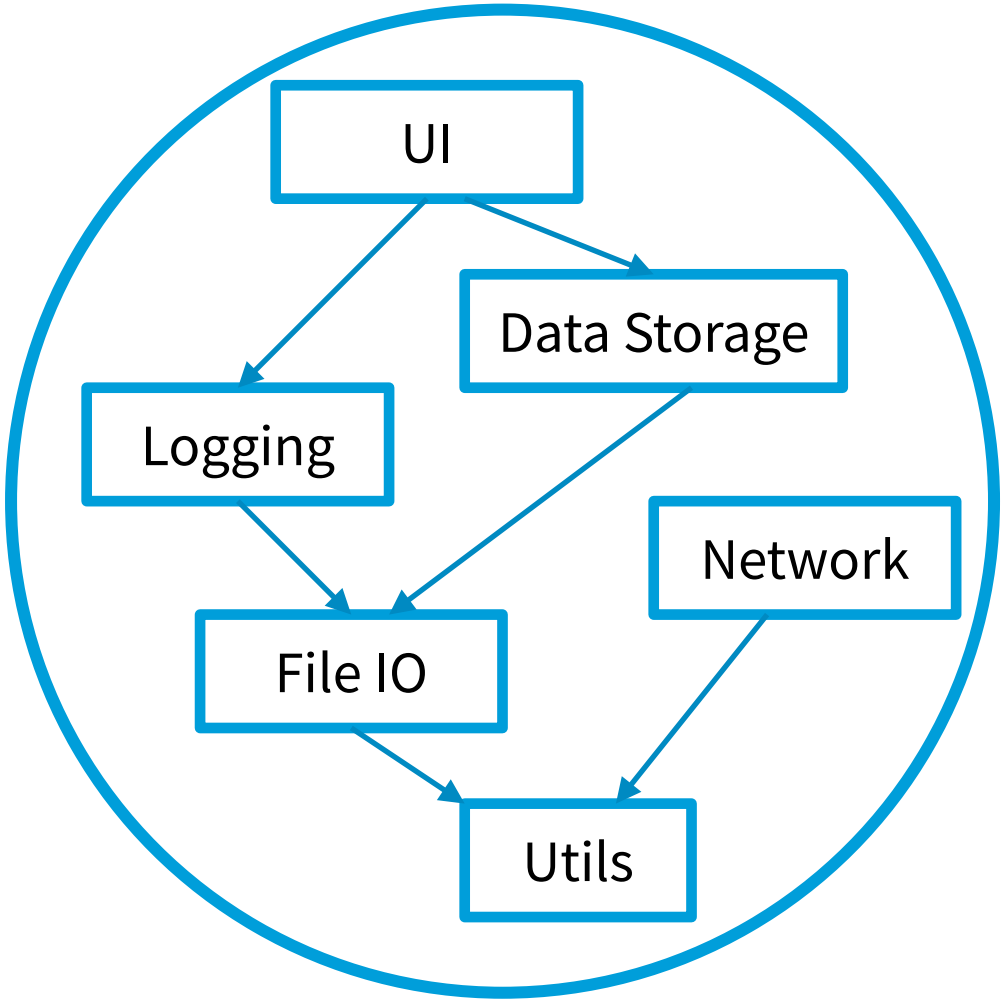
II) Modules in Effekt

III) Effects & Modules

IV) Outlook



Monolithic Software



Modulare Software

Modularity

- Modules: composable software units
  - Interface: specifies operations
  - Implementation: provides actual operations
  - Consumer: interacts with module (via interface)
- Key Benefits
  - Separation of Concerns
  - Separate Compilation
  - Code Reuse

Agenda

I) Modularity

II) Modules in Effekt

III) Effects & Modules

IV) Outlook

Motivation

- Effekt: research programming language
  - Effectful programming
  - Lightweight effects using capability passing
- Main goal: design a module system for Effekt
  - Implement new language features
  - Focus on compability
  - Find similiarities between modules and effects

```
// Source Module
module examples/hello
import text/string

def greet(name: String): String = {
  return “Hello “ ++ name ++ “!“
}
```

```
// Source Module
module examples/hello
import text/string

def greet(name: String): String = {
  return “Hello “ ++ name ++ “!“
}
```

```
// Source Module
module examples/world
import examples/hello

def world() = {
  println(greet(“World“))
}

def main() = {
  world()
}
```

```
// Source Module
module examples/hello
import text/string

def greet(name: String): String = {
  return “Hello “ ++ name ++ “!“
}

def world(): Unit = ...
```

```
// Source Module
module examples/world
import examples/hello

def world() = {
  println(greet(“World“))
}

def main() = {
  world()
}
```

```
// Source Module
module examples/hello
import text/string

def greet(name: String): String = {
  return “Hello “ ++ name ++ “!“
}

def world(): Unit = ...
```

```
// Source Module
module examples/world
import examples/hello

// User Module
module Hello {
  def world() = {
    println(greet(“World“))
  }
}

def main() = {
  world()
  // Module Call
  Hello:world()
}
```

```
interface Worker {
  def todo(): Int
  def done(): Int
}

def work() with { mod: Worker }: Int = {
  mod:todo() + mod:done()
}
```

```
interface Worker {
  def todo(): Int
  def done(): Int
}

def work() with { mod: Worker }: Int = {
  mod:todo() + mod:done()
}
```

```
module Tasks implements Worker {
  def todo(): Int = 40
  def done(): Int = 2
}

def main() = {
  println(work() with Tasks)
}
```

```
interface Worker {
  def todo(): Int
  def done(): Int
}

def work() with { mod: Worker }: Int = {
  mod:todo() + mod:done()
}
```

```
module examples/tasks implements Worker

// imports, etc

def todo(): Int = 40
def done(): Int = 2

def main() = {
  println(work() with /examples/taks)
}
```

Modules in Effekt

- Modules in Effekt are
  - Second-class citiziens
  - Nominal typed via interfaces
  - Stateless
- Two kinds of modules
  - Source module
  - User module
  - Both can implement interfaces

Agenda

I) Modularity

II) Modules in Effekt

III) Effects & Modules

IV) Outlook

Side-Effects

- Side-Effects: operation that modifies execution context
  - Mutating global state
  - Throwing an exception
  - Pause thread (sleep, wait, etc.)
- Algebraic Effects: control the use of side-effects
  - Side-effects annotated to the return type
  - Handling similiar to exceptions (try/catch)
  - Effect-safety: all effects are eventually handled

```
interface Worker {
  def todo(): Int
  def done(): Int
}

def work() with { mod: Worker }: Int = {
  mod:todo() + mod:done()
}

module Tasks implements Worker {
  def todo(): Int = 40
  def done(): Int = 2
}
def main() = {
  println(work() with Tasks)
}
```

```
effect Worker {
  def todo(): Int
  def done(): Int
}
```

```
interface Worker {
  def todo(): Int
  def done(): Int
}

def work() with { mod: Worker }: Int = {
  mod:todo() + mod:done()
}

module Tasks implements Worker {
  def todo(): Int = 40
  def done(): Int = 2
}
def main() = {
  println(work() with Tasks)
}
```

```
effect Worker {
  def todo(): Int
  def done(): Int
}

def work(): Int / {Worker} = {
  do todo() + do done()
}
```

```
interface Worker {
  def todo(): Int
  def done(): Int
}

def work() with { mod: Worker }: Int = {
  mod:todo() + mod:done()
}

module Tasks implements Worker {
  def todo(): Int = 40
  def done(): Int = 2
}
def main() = {
  println(work() with Tasks)
}
```

```
effect Worker {
  def todo(): Int
  def done(): Int
}

def work(): Int / {Worker} = {
  do todo() + do done()
}

def main() = {
  try { println(work()) }
  with Worker {
    def todo() = resume(40)
    def done() = resume(2)
  }
}
```

```
interface Worker {
  def todo(): Int
  def done(): Int
}

def work() with { mod: Worker }: Int = {
  mod:todo() + mod:done()
}

module Tasks implements Worker {
  def todo(): Int = 40
  def done(): Int = 2
}
def main() = {
  println(work() with Tasks)
}
```

```
effect Worker {
  def todo(): Int
  def done(): Int
}

def work(): Int / {Worker} = {
  do todo() + do done()
}

def main() = {
  try { println(work()) }
  with Worker {
    def todo() = resume(40)
    def done() = ()
  }
}
```

```
effect Add(l: Int, r: Int): Int
effect Lit(x: Int): Int

interface Calc {
  def handleAdd { f: Unit / Add }: Unit
  def handleLit { f: Unit / Lit }: Unit
}
```

```
effect Add(l: Int, r: Int): Int
effect Lit(x: Int): Int

interface Calc {
  def handleAdd { f: Unit / Add }: Unit
  def handleLit { f: Unit / Lit }: Unit
}

def term() with { calc: Calc } = {
  calc:handleLit { calc:handleAdd {
    val x = do Add(do Lit(40), do Lit(2))
    println(x)
  }}
}
```

```
effect Add(l: Int, r: Int): Int
effect Lit(x: Int): Int

interface Calc {
  def handleAdd { f: Unit / Add }: Unit
  def handleLit { f: Unit / Lit }: Unit
}

def term() with { calc: Calc } = {
  calc:handleLit { calc:handleAdd {
    val x = do Add(do Lit(40), do Lit(2))
    println(x)
  }}
}
```

```
module Eval implements Calc {
  def handleAdd { f: Unit/Add }: Unit = {
    try { f() }
    with Add { (l, r) => resume(l + r) }
  }

  def handleLit { f: Unit/Lit }: Unit = {
    try { f() }
    with Lit { (x) => resume(x) }
  }
}

def main() = {
    term() with Eval
}
```

Effects & Modules

Modules		Effects
Interface, Implementation, Parameter	==	Signature, Handler, Annotation
Second-Class Citizen	==	Second-Class Citizen
Global Names	!!	Local Handler Definitions
Abstracts Behavior	!!	Abstracts Control Flow

Agenda

I) Modularity

II) Modules in Effekt

III) Effects & Modules

IV) Outlook

Local Modules

- Modules are second-class
  - Variables cannot store modules
  - Functions cannot return modules
  - Identified with qualified name
- Local modules: module definition inside of a function
  - Temporary instance
  - Can capture capabilities
  - Borrow syntax from handlers

```
effect State {
    def get(): Int
    def set(n: Int): Unit
}

interface Counter {
    def next(): Int
    def reset(): Unit
}

def count() with { c: Counter } = ...
```

```
effect State {
    def get(): Int
    def set(n: Int): Unit
}

interface Counter {
    def next(): Int
    def reset(): Unit
}

def count() with { c: Counter } = ...
```

```
try {
  count() with ???




} with State { // Handler
  def get() = ...
  def set(n: Int) = ...
}
```

```
effect State {
    def get(): Int
    def set(n: Int): Unit
}

interface Counter {
    def next(): Int
    def reset(): Unit
}

def count() with { c: Counter } = ...
```

```
try {
  count() with Counter { // Local module
    def next(): Int = {
      val n = do get()
      do set(n + 1)
      return n
    }
    def reset(): Unit = {
      do set(0)
    }
  }
} with State { // Handler
  def get() = ...
  def set(val: Int) = ...
}
```

Outlook

- Further Improvements
  - Local Modules
  - Modules as Handlers
  - Unification of Modules & Effects

Discussion

Feel free to ask questions