Experience Report
Graph Rewriting with Lexical Effect Handlers

Marvin Borner, University of Tübingen

HOPE 2026, Functional Programming Workshop, Paris

Marvin Borner

- Master student in Jiří Beneš' **Effekt** course
- Student compiler engineer at **Tübingen PL Center**

Graph Rewriting with Lexical Effect Handlers

- Graph rewriting? **Interaction combinators**!
  - graphical model of computation
  - rewrite relation
  - as web application

Graph Rewriting with Lexical Effect Handlers

- Graph rewriting? **Interaction combinators**!
  - graphical model of computation
  - rewrite relation
  - as web application
- Lexical effect handlers? **Effekt**!
  - tangible effect semantics and syntax
  - lightweight effect polymorphism

Graph Rewriting with Lexical Effect Handlers

- Graph rewriting? **Interaction combinators**!
  - graphical model of computation
  - rewrite relation
  - as web application
- Lexical effect handlers? **Effekt**!
  - tangible effect semantics and syntax
  - lightweight effect polymorphism
- Experience report
  - effect-directed programming

Effekt Recap

Ambiguous Puzzle

```
interface Exc {
  def throw(): Unit
}

try { // action
  someFunction { () => do throw() }
} with Exc { // handler
  def throw() { /* ... */ }
}
```

Ambiguous Puzzle

```
interface Exc {
  def throw(): Unit
}

try { // action
  someFunction { () => do throw() }
} with Exc { // handler
  def throw() { /* ... */ }
}
```

```
def someFunction { f } {
  try {
    f()
  } with Exc {
    def throw() { /* ... */ }
  }
}
```

Slightly better: Named Handlers

```
interface Exc {
  def throw(): Unit
}

try { // action
  someFunction { () => exc.throw() }
} with exc: Exc { // handler
  def throw() { /* ... */ }
}
```

Slightly better: Named Handlers

```
interface Exc {
  def throw(): Unit
}

try { // action
  someFunction { () => exc.throw() }
} with exc: Exc { // handler
  def throw() { /* ... */ }
}
```

exc is a **capability** that enables **lexical scoping**

Operational Semantics: Capability-Passing Style


```
def catch[A] { action: () => A / { Exc } }
    { handler: String => A / {} }: A / {}
```

$\Rightarrow$

```
def catch[A] { action: {Exc} => A }
    { handler: String => A }: A
```

Operational Semantics: Capability-Passing Style


```
def catch[A] { action: () => A / { Exc } }
    { handler: String => A / {} }: A / {}
```

⇒

```
def catch[A] { action: {Exc} => A }
    { handler: String => A }: A
```

⇒ lexical scoping!

How?

Observation: If a function cannot leave the dynamic scope it is defined in, we **do not need to track** its effects.

How?

Observation: If a function cannot leave the dynamic scope it is defined in, we **do not need to track** its effects.

Data is first-class

- values: `"foo"`, `42`, `false`
- can be returned/stored

Computation is second-class*

- blocks/functions: `map`, `{ x => x * 2 }`
- cannot be returned/stored

How?

Observation: If a function cannot leave the dynamic scope it is defined in, we **do not need to track** its effects.

Data is first-class

- values: `"foo"`, `42`, `false`
- can be returned/stored

Computation is second-class*

- blocks/functions: `map`, `{ x => x * 2 }`
- cannot be returned/stored

*by default: `box` gives **first-class functions**

Syntax Sugar

```
def freshener[R] { prog: => R / fresh }: R = /* ... */

 def transform(term: Term) = {
   freshener {
     do fresh()
     /* ... */
   }
 }
```

Syntax Sugar

```
def freshener[R] { prog: => R / fresh }: R = /* ... */
```

```
def transform(term: Term) = {
  freshener {
    do fresh()
    /* ... */
  }
}
```

```
def transform(term: Term) = {
  with freshener
  do fresh()
  /* ... */
}
```

Implication: Novel Reading

```
f: A => B / { E }
```

- **Traditional**: “Given the value of type `A`, the function produces a value of type `B` and potentially uses effect `E`.”
- **Effekt**: “Given the value of type `A`, the function produces a value of type `B` and **requires the calling context** to handle effect `E`.”

Implication: Novel Reading

```
f: A => B / { E }
```

- **Traditional**: “Given the value of type `A`, the function produces a value of type `B` and potentially uses effect `E`.”
- **Effekt**: “Given the value of type `A`, the function produces a value of type `B` and **requires the calling context** to handle effect `E`.”

⇒ Effects are **context requirements**

Implication: Different “Purity”

```
f: A => B / {}
```

- not necessarily *pure*, except in top-level
- effects at definition-site may still require handling

⇒ “The call-site of `f` does not have to handle any of its effects”

Effekt Summary

- Effects are **context requirements**
- Enable local, scope-based reasoning
- Computation is second-class by default

Effekt Summary

- Effects are **context requirements**
- Enable local, scope-based reasoning
- Computation is second-class by default

Effects are context requirements!

Transfer: Object-Oriented Programming

- OOP: “Program against an **interface** whose implementation is supplied by **context**.”
- Effekt: “Program against **effects** whose implementation is supplied by **handlers**.”

Effect-Directed Programming: My Approach

Known fallacy from OOP: **Everything becomes an interface**.

Effect-Directed Programming: My Approach

Known fallacy from OOP: **Everything becomes an interface**.

$\Rightarrow$ systematic approach to solving a given problem.

- **functions** to decompose solution *declaratively*
- **effects** to describe what it needs from context
  - especially if there might be multiple handlers (Parnas, 1972)
- **handlers** at lexical boundaries to provide their **interpretation**

Effect-Directed Programming: My Approach

- program against effects and types
  - algorithm in function, put into context by handler

$\Rightarrow$ **plan** $\rightarrow$ **interface** $\rightarrow$ **declaration** $\rightarrow$ **implementation**

Experience Report

Graph Rewriting with Effect-Directed Programming

Observation (spoiler)

Effect-directed programming **encourages** maintainable architecture.

- **high cohesion, low coupling**
  - do not have to understand module A to understand module B
  - communication & interpretation via effects & handlers
  - testable by mock handlers

Observation (spoiler)

Effect-directed programming **encourages** maintainable architecture.

- **high cohesion, low coupling**
  - do not have to understand module A to understand module B
  - communication & interpretation via effects & handlers
  - testable by mock handlers
- **local reasoning**
  - comprehend code just from local scope (signatures!)
  - all in same *level of abstraction*
    - “pure” core, “impure” shell

Plan: Graph Rewriting Application

- **interaction combinators**: graph-based model of computation

Plan: Graph Rewriting Application

- **interaction combinators**: graph-based model of computation
- **web** interface (react-ish):
  - **parse** calculus (two modes)
  - **compile** to interaction net (graph)
  - **rule** stepper (by DSL)
  - interactive **force-directed** rendering

Plan: Graph Rewriting Application

- **interaction combinators**: graph-based model of computation
- **web** interface (react-ish):
  - **parse** calculus (two modes)
  - **compile** to interaction net (graph)
  - **rule** stepper (by DSL)
  - interactive **force-directed** rendering
- **cli** interface (*also* react-ish):
  - … (same model)

Demo

Language Pipeline, Streamingly (1)

- parse
  - `/ { next[Char], emit[Term], ... } // out`
- compile
  - `{ / emit[Term] } // in`
  - `/ { emit[Constructor], ... } // out`
- step
  - `{ / emit[Constructor], emit[Redex] } // in`
  - `/ { emit[Constructor], emit[Redex], ... } // out`

Language Pipeline, Streamingly (2)

- decompose
  - `{ / emit[Constructor] } // in`
  - `/ { emit[Node], emit[Edge] } // out`
- layout
  - `{ / emit[Node], emit[Edge] } // in`
  - `/ { emit[Node], emit[Edge] } // out`
- render
  - `{ / emit[Node], emit[Edge] }`
  - `/ { emit[Arrow], emit[Triangle], emit[Circle] }`

Language Pipeline, Streamingly (3)

- draw
  - `{ / emit[Arrow], emit[Triangle], emit[Circle] }`
  - `: Canvas`
- `applyTo(node: Canvas, node: HTMLNode)`

Why Streams? (1)

Decoupling of producers and consumers:

- producers emit one value at a time
- consumers filter/reify/transform stream

Why Streams? (1)

Decoupling of producers and consumers:

- producers emit one value at a time
- consumers filter/reify/transform stream

```
def nodePositions()
  { nodes: => Unit / emit[Node] }: Unit / emit[Vector]

def springForce(node: Node, length: Double)
  { net: => Unit / { emit[Node], emit[Edge] } }: Vector / Force
```

Why Streams? Composition! (2)

```
// signatures
parse() / { Exception[WrongFormat], Scan[Char], emit[/* */] }
check() / { Exception[WrongFormat], next[/* */], emit[/* */] }
uniquify() / { next[/* */], emit[/* */] } // can't fail
```

Why Streams? Composition! (2)

```
// signatures
parse() / { Exception[WrongFormat], Scan[Char], emit[/* */] }
check() / { Exception[WrongFormat], next[/* */], emit[/* */] }
uniquify() / { next[/* */], emit[/* */] } // can't fail


def compile() = // declarative: plug together
  with source[Constructor] { parse() }
  with source[Constructor] { check() }
  uniquify()

def main() = // imperative shell
  with on[WrongFormat].panic() // or whatever
  compile()
```

Stream Abstractions

- `effect NetStream = { emit[Node], emit[Wire] }`
- `type Program = () => Unit / emit[Constructor] at {}`
- `type TextStream = () => Unit / emit[Char] at {}`

First Step: From UI to `emit[Char]`

Plan: Model

What capabilities should the (shared) model have?

- **render** net
- **step** reduction
- **tick** animation
- **interact** with context (e.g. canvas, terminal)
- **modify** state (e.g. source, mode)

Interface: UI Model

```
interface Model[C, O] {
  def render(): Unit / Redraw
  def step(): Unit / Redraw
  def tick(): Unit / Redraw
  def nextMode(): Unit / Redraw
  def setSource(s: String): Unit / Redraw
  def onContext(op: O): Unit / Redraw

  def getMode(): Mode
  def getSource(mode: Mode): TextStream
  def getContext(): C
  def getErrors(): String
}
```

Declarative: UI View (web)

```
type Event {
  Render();
  Step();
  Tick();
  NextMode();
  SetSource(s: String);
  PanCanvas(v: Vector);
  ZoomCanvas(d: Double);
}
```

Declarative: UI View (web)

```
div([ "split" ], [
  textarea([
    OnKeyUp(box { s => SetSource(s) }),
  ], ["left", "code"], do getSource(do getMode())),
  div([
    OnDrag(box { v => PanCanvas(v) }),
    OnScroll(box { d => ZoomCanvas(d) }) ], ["right"], [
    canvas([
      OnResize(box { v => SetCanvasSize(v) }),
      OnTick(Step()),
    ], ["inet"], do getContext())
  ]),
]), /* ... */
```

Declarative: UI View (web)

```
def dispatch(msg: Event): Unit /
    { Model[Canvas, CanvasOp], Redraw } =
  msg match {
    // ...
    case SetCanvasSize(v) =>
      do onContext(Size(v))
    case PanCanvas(v) =>
      do onContext(Pan(v))
    case ZoomCanvas(d) =>
      do onContext(Zoom(d))
  }
```

Implementation: UI View (web)

```
run(appDiv, model::empty(), HtmlView(
  update = box { event =>
    with web
    event.dispatch()
  },
  view = box {
    with web
    view()
  }))
```

Implementation: UI View (CLI)

```
run(empty(), model::empty(), CliView(
  update = box { ev =>
    with cli
    ev.dispatch()
  },
  view = box {
    with cli
    with Formatted::formatting
    view()
  }))
```

Implementation: Partial Redrawing (web)

```
interface Redraw {
  def redrawCallee(): Unit
  def redrawFull(): Unit
}

// in web framework
...
  case Event(tag, ev) =>
    with redraw { tag.mark(Dirty()) } { dirty.set(Dirty()) }
    with statefully(state)
    update(event)
...
```

Next Step: Parsing

Interface: Effectful Lexing

```
// what can a lexer do?
interface Lexer {
  def peek(): Option[Token]
  def next(): Token
}
```

Interface: Effectful Lexing

```
// what can a lexer do?
interface Lexer {
  def peek(): Option[Token]
  def next(): Token
}

// reinterpretation!
def skipWhitespace[R] { prog: => R / Lexer }: R / Lexer =
  try { prog() } with Lexer {
    def peek() = { skipSpaces(); resume(do peek()) }
    def next() = { skipSpaces(); resume(do next()) }
  }
```

Interface: Effectful Parsing

```
effect Parser = { Nondet, Lexer }

interface Nondet {
  def alt(): Bool
  def fail(msg: String): Nothing
}
```

Interface: Effectful Parsing

```
effect Parser = { Nondet, Lexer }

interface Nondet {
  def alt(): Bool
  def fail(msg: String): Nothing
}

def or[R] { p: => R } { q: => R } =
  if (do alt()) { p() } else { q() }

def many[R] { p: => R }: List[R] / Parser =
  or { some { p() } } { Nil[R]() }
```

Declaration: Effectful Parsing

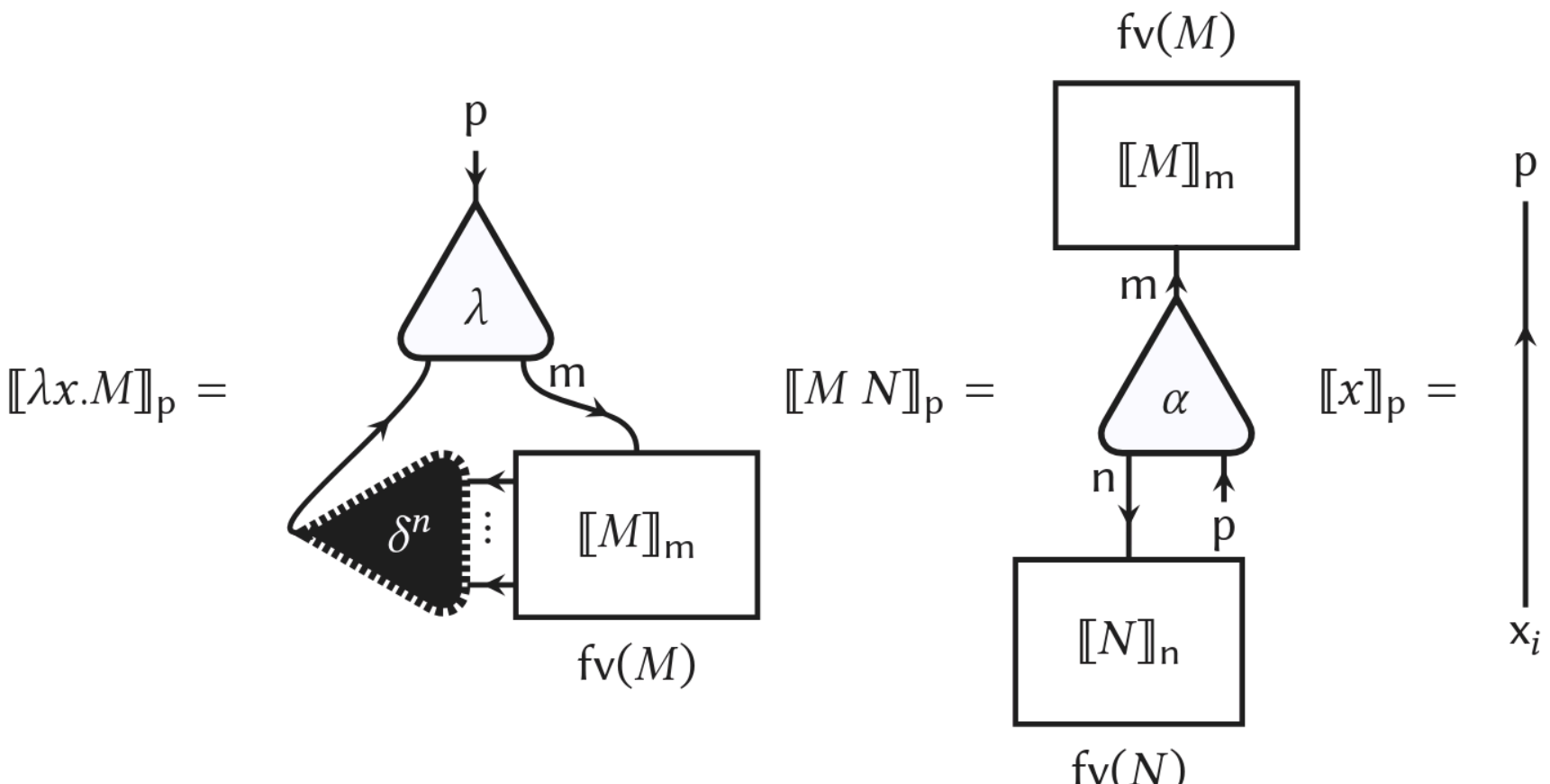
```
def parseAbstraction(): Term / Parser = {
  val name = ident()
  punct("=>")
  Abstraction(name, parseTerm())
}

def parseTerm(): Term / Parser =
  or
    { parseAbs() }
    { parseApp() }
    { parseSym() }
```

Implementation: Effectful Parsing

```
def parse(): { Scan[Char], emit[Term], Exception[WrongFormat] } =
  try { /* ... */ }
  with Nondet {
    def alt() = resume(true) match {
      case Error(_, _) => resume(false)
      case Success(res) => Success(res)
    }
    def fail(msg) = Error(WrongFormat(), msg)
  }
```

...some intermediate steps...


$[\![\lambda x.M]\!]_\mathsf{p} =$
p
λ
m
δ^n
$[\![M]\!]_\mathsf{m}$
$\mathrm{fv}(M)$
$[\![M\ N]\!]_\mathsf{p} =$
$\mathrm{fv}(M)$
$[\![M]\!]_\mathsf{m}$
m
α
n
p
$[\![N]\!]_\mathsf{n}$
$\mathrm{fv}(N)$
$[\![x]\!]_\mathsf{p} =$
p
x_i

Next Step: Interaction

Plan: Interaction Combinators

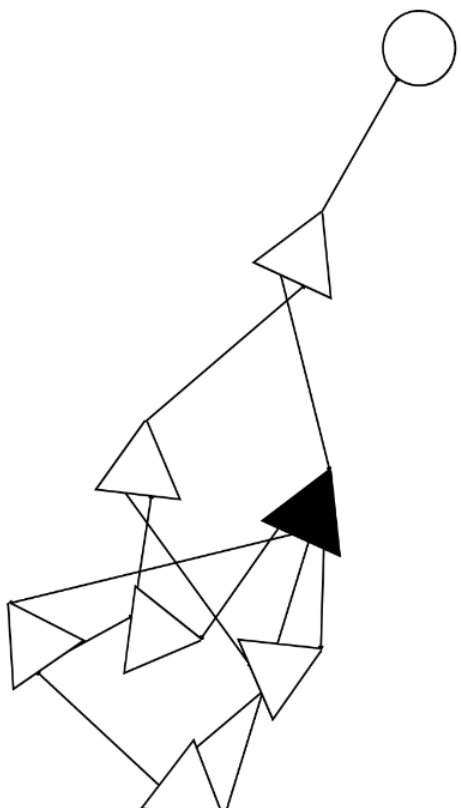
Idea: Encode computation into a graph representing its fundamental operations.

Plan: Interaction Combinators

Idea: Encode computation into a graph representing its fundamental operations.

What capabilities should the IC reduction have?

- **Exchange**: Edge crossing
- **Contraction**: Node duplication
- **Weakening**: Node erasure



Interface: Interaction Combinators

```
interface Interact {
  def rewire(rewiring: Map[String, Substitution]): Unit
  def deconstruct(ctor: Constructor): Unit
  def construct(ctor: Constructor): Unit
}
```

Declaration: Interaction Combinators

```
def interact(redex: Redex): Unit / { Interact, fresh } = redex match {
  case Redex(Applicator(Neg(f1), Neg(a), Pos(k)),
             Abstractor(Pos(x), Neg(b), Pos(f2))) and f1 == f2 => {
    do rewire(/* substitutions */)
  }

  case Redex(Duplicator(/* ... */), Abstractor(/* ... */)) => {
    val (xsNeg, xsPos) = n.freshEdges()
    do construct(/* ... */)
    // ...
  }

  // ...
}
```

Implementation: Interaction Combinators

```
val deconstructed = ref[Set[Constructor]](empty())
val constructed = ref[List[Constructor]](Nil())
val wiring = ref(program.wiring())

try {
  redex.interact()
  do deconstruct(redex.left)
  do deconstruct(redex.right)
} with Interact {
  def deconstruct(ctor) = resume(deconstructed.map { s =>
    s.insert(ctor) })
  def construct(ctor) = resume(do emit(ctor))
  def rewire(rewiring) = resume(wiring.rewire(rewiring))
  ...
```

Handled at Boundary: `fresh`

```
def normalize(program: Program, redexes: Redexes): Program = {
  var state = State(program, redexes)
  with freshener // <---------
  with exhaustively
  state = collect { state.program.step(state.redexes) }
  state.program
}
```

Handled at Boundary: `fresh`

```
def freshener[R] { prog: => R / fresh }: R = {
  var counter = 0
  try prog() with fresh {
    counter = counter + 1
    resume(counter - 1)
  }
}
```

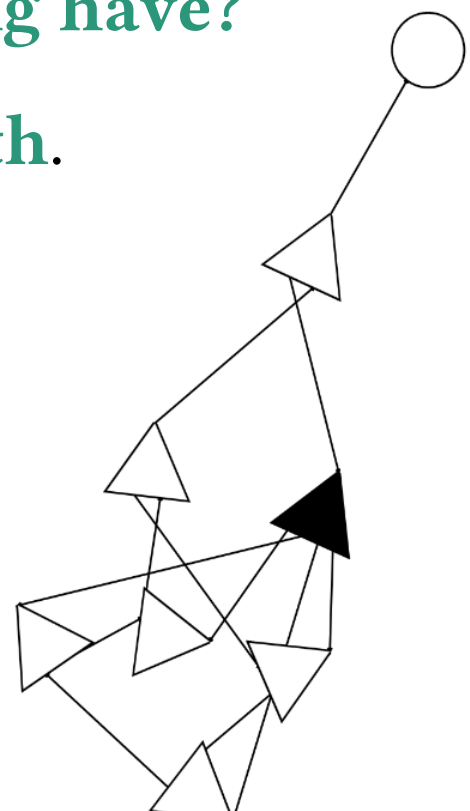
Final Step: Force-Directed Rendering

Plan: Force-Directed Rendering

What capabilities should force-directed rendering have?

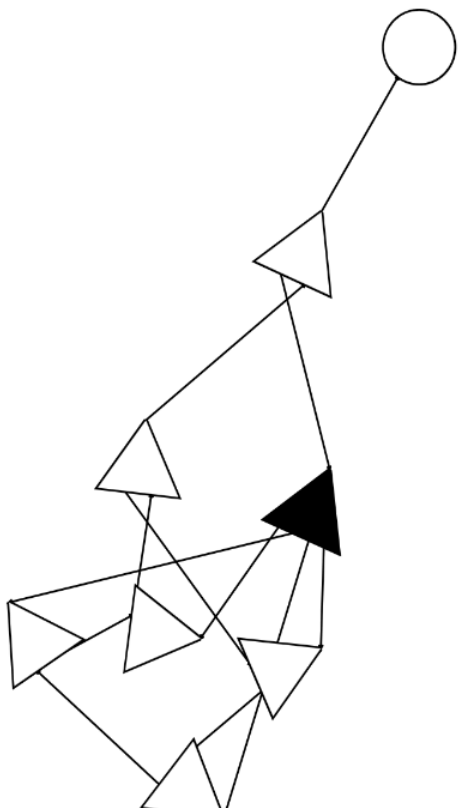
Physicist's answer: Force has an **origin** and a **strength**.

$\Rightarrow$ interpretation is provided by context.



Interface: Force-Directed Rendering

```
interface Force {
  def origin(): Vector
  def strength(x: Double): Double
}
```



Declaration: Force-Directed Rendering (1)

Simple **vector operations**, no knowledge about force/strength:

```
def repulsion(from: Vector, position: Vector): Vector / Force =
  force(position - from) // vector `from` to `position`

def attraction(towards: Vector, position: Vector): Vector / Force =
  force(towards - position) // vector `position` to `towards`

def centralGravitation(node: Node): Vector / Force =
  Vector(0.0, 0.0).attraction(node.pos)
```

Declaration: Force-Directed Rendering (2)

```
def springForce(node: Node, length: Double)
    { net: => Unit / NetStream }: Vector / Force =
  with forceSum
  with node.edgeForce(length) { nodes{net} }
  node.attachedEdges { edges{net} }

def coulombForce(node: Node)
    { nodes: => Unit / emit[Node] }: Vector / Force =
  with forceSum
  with source[Vector] { nodePositions{nodes} }
  with loop
  do emit(repulsion(do next[Vector], node.pos))
```

Contrast (1): Dependency Injection in OOP

```
val force = new Force(origin, strength)
repulsion(force, from, to)
attraction(force, from, to)
coulombForce(force, nodeList, node)
...
```

Contrast (1): Dependency Injection in OOP

```
val force = new Force(origin, strength)
repulsion(force, from, to)
attraction(force, from, to)
coulombForce(force, nodeList, node)
...
```

Contextual interpretations should not be value arguments!

Contrast (2): Coulomb Force with Monads

```haskell
coulombForce :: View Position n ⇒ Node -> WithGraph n Force
coulombForce node = do
  n ← examine position <$> readNode node
  ns ← fmap (map $ examine position)
    (mapM readNode =<< readNodeList)
  return $ fsum [repulsion n' n | n' ← ns]
```

Rochel, Jan “`graph-rewriting-layout`” (2024)

(https://hackage.haskell.org/package/graph-rewriting-layout)

Implementation: Force-Directed Rendering (2)

```
// finally implement what "force" represents
def force(f: Vector): Vector / Force =
  do origin() + f.normalize * do strength(f.magnitude)

def withForce(origin: Vector) { f: => Vector / Force }
    { strength: Double => Double }: Vector =
  try f() with Force {
    def origin() = resume(origin)
    def strength(x) = resume(strength(x))
  }
```

Implementation: Force-Directed Rendering (3)

```
def layoutStep(node: Node) { net: => Unit / NetStream }: Node = {
  val cf = withForce(node.pos) {
    coulombForce(node) { nodes{net} } } {
      x => min(/* ... */, x * /* ... */) }
  val cgf = withForce(cf) {
    node.centralGravitation } {
      x => min(/* ... */, x * /* ... */) }
  val sf = withForce(cgf) {
    node.springForce(/* ... */) {net} } {
      x => min(/* ... */, x * /* ... */) }

  node.map { _ => sf }
}
```

Maintainability as Side Effect

- **Dependency injection without parameter threading**
  - `Force` parameter
- **Program against an interface**
  - functions depend on `Force`
- **High cohesion**
  - constants, vector arithmetic are in single place
- **Low coupling**
  - changing force constants does not change other functions
- **One level of abstraction**
  - every function does exactly what is expected from it

Testability as Side Effect

```
test("layout: spring force") {
  val springPos = withForce(start) { node.springForce(150.0) {
    do emit(node)
    do emit(Node("b", 3, [], LightTriangle(),
      Vector(1000.0, 1000.0)))
    do emit(Edge(1, 3))
    do emit(Node("c", 4, [], LightTriangle(),
      Vector(-1000.0, -1000.0)))
    do emit(Edge(2, 4))
  } } { r => r * 0.1 }
  assertEqual(springPos, Vector(850.0, 850.0))
}
```

Conclusion

Conclusion

- **Effekt** implements lexical effect handlers
  - effects are **context requirements**
- **Encourages** maintainable architecture
  - vs. OOP: “Program against **effects**”
  - high cohesion, low coupling
  - local reasoning

Questions?