Parsing Graphs

Applying Parser Combinators to Graph Traversals

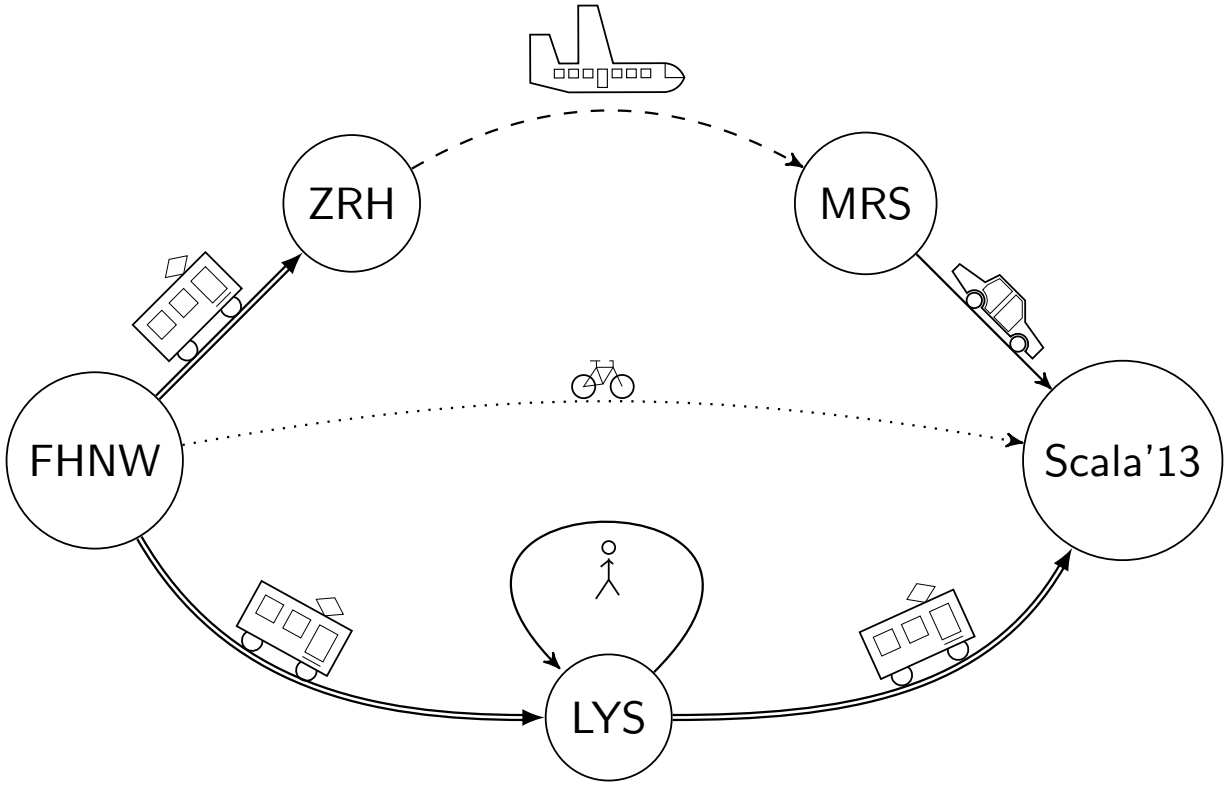
D. Kröni R. Schweizer

Institute of Mobile and Distributed Systems
University of Applied Sciences and Arts Northwestern Switzerland

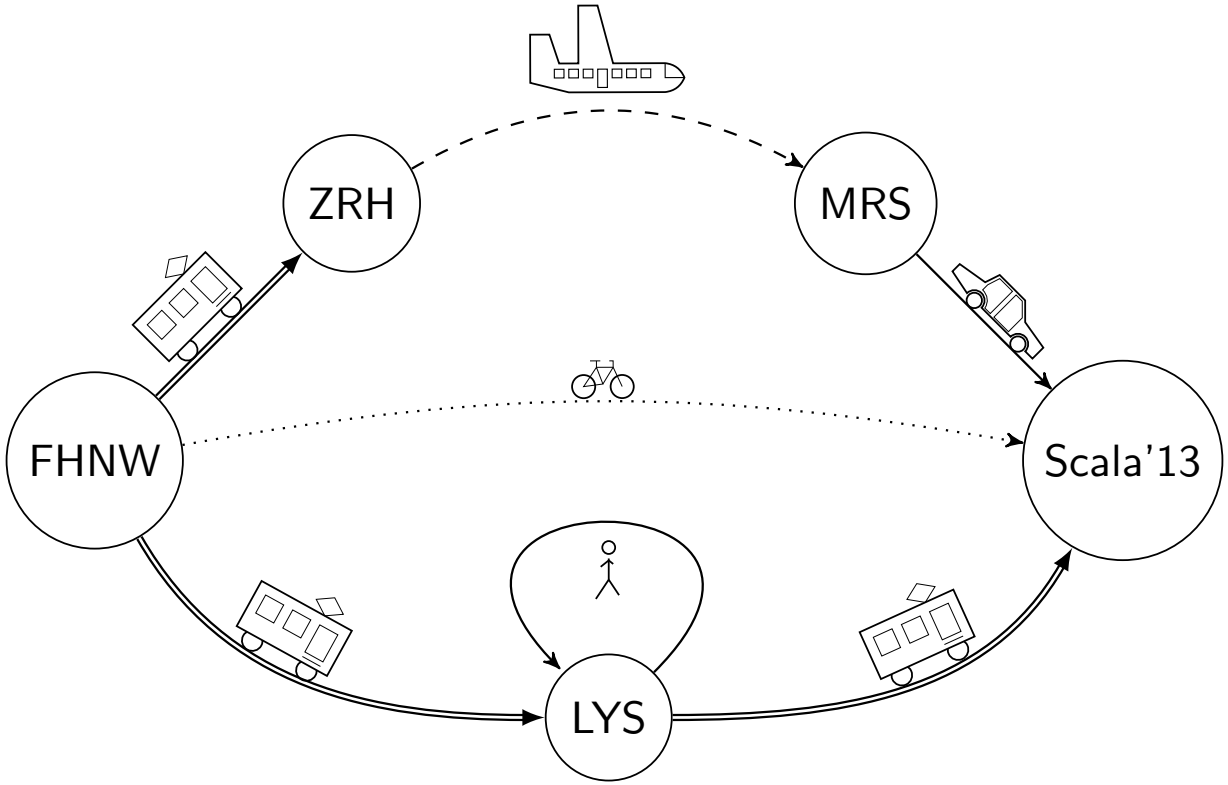
Scala 2013, the Fourth Annual Scala Workshop
Montpellier, France

University of Applied Sciences and Arts
Northwestern Switzerland

Travel to Montpellier

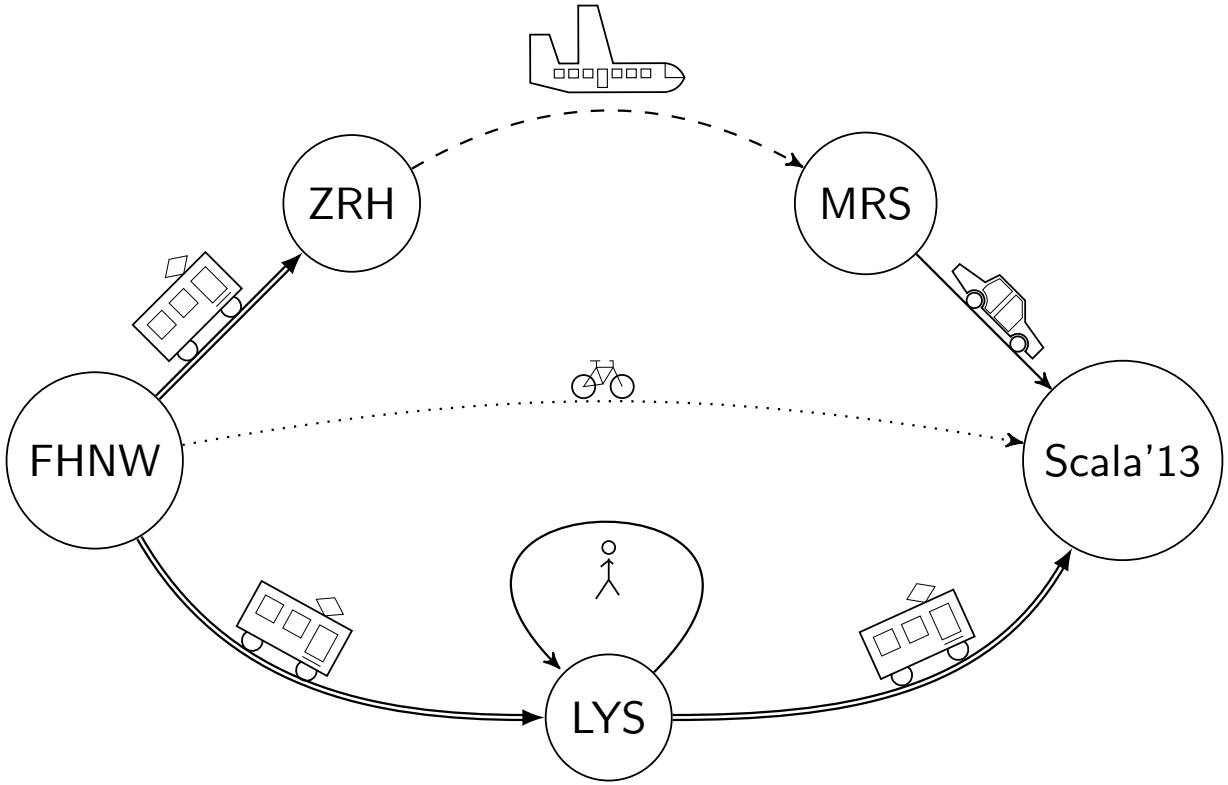


Travel to Montpellier




```
val journey  =  (FHNW ~ 🚋 ~ ✈ ~ 🚗)
             |  (FHNW ~ 🚋 ~ 🚶.* ~ 🚋)
```

Travel to Montpellier




```
val journey  =  FHNW ~ 🚋 ~ ((✈ ~ 🚗) | (🚶 .* ~ 🚋))
```

The Type of a Traverser

```
type Tr[E,I,O,A] = E => I => Stream[(O,A)]
```

- Environment (graph)
- State `I => O` (current position, ...)
- An arbitrary result (e.g. property value)
- Stream (many results)

The Type of a Traverser

```
type Tr[E,I,O,A] = E => I => Stream[(O,A)]
```

- Environment (graph)
- State `I => O` (current position, ...)
- An arbitrary result (e.g. property value)
- Stream (many results)

The Type of a Traverser

```
type Tr[E,I,O,A] = E => I => Stream[(O,A)]
```

- Environment (graph)
- State `I => O` (current position, ...)
- An arbitrary result (e.g. property value)
- Stream (many results)

The Type of a Traverser

```
type Tr[E,I,O,A] = E => I => Stream[(O,A)]
```

- Environment (graph)
- State `I => O` (current position, ...)
- An arbitrary result (e.g. property value)
- Stream (many results)

The Type of a Traverser

```
type Tr[E,I,O,A] = E => I => Stream[(O,A)]
```

- Environment (graph)
- State `I => O` (current position, ...)
- An arbitrary result (e.g. property value)
- Stream (many results)

The Type of a Traverser

```
type Tr[E,I,O,A] = E => I => Stream[(O,A)]
```

- Environment (graph)
- State `I => O` (current position, ...)
- An arbitrary result (e.g. property value)
- Stream (many results)

The Type of a Traverser

```
type Tr[E,I,O,A] = E => I => Stream[(O,A)]
```

- Environment (graph)
- State `I => O` (current position, ...)
- An arbitrary result (e.g. property value)
- Stream (many results)

Primitives

- **Start**

```
V(): Tr[G,Any,N,N]
E(): Tr[G,Any,E,E]
```

- **Navigation**

```
outE(t: String): Tr[G,N,E,E]
inV(): Tr[G,E,N,N]
inE(t: String): Tr[G,N,E,E]
outV(): Tr[G,E,N,N]
```


- **Properties**

```
get[A](key: String): Tr[G,El,El,A]
```

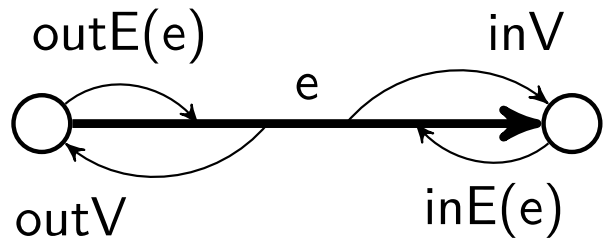
Primitives

- **Start**

```
V(): Tr[G,Any,N,N]
E(): Tr[G,Any,E,E]
```

- **Navigation**

```
outE(t: String): Tr[G,N,E,E]
inV(): Tr[G,E,N,N]
inE(t: String): Tr[G,N,E,E]
outV(): Tr[G,E,N,N]
```



- **Properties**

```
get[A](key: String): Tr[G,El,El,A]
```

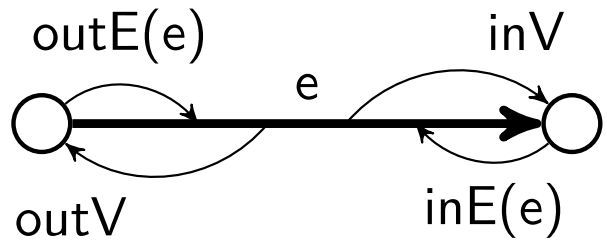
Primitives

- **Start**

  ```
  V(): Tr[G,Any,N,N]
  E(): Tr[G,Any,E,E]
  ```

- **Navigation**

  ```
  outE(t: String): Tr[G,N,E,E]
  inV(): Tr[G,E,N,N]
  inE(t: String): Tr[G,N,E,E]
  outV(): Tr[G,E,N,N]
  ```

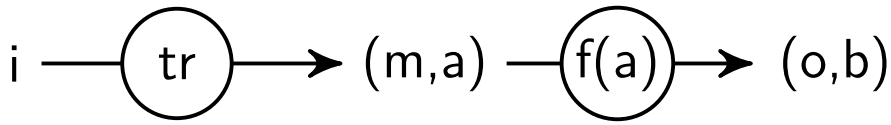


- **Properties**

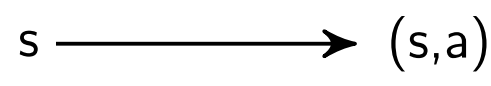
  ```
  get[A](key: String): Tr[G,El,El,A]
  ```

Combinators – Building Blocks

– **Sequential composition**




```
def flatMap[E,I,M,O,A,B]
  (tr: Tr[E,I,M,A])(f: A => Tr[E,M,O,B]): Tr[E,I,O,B]

def success[E,S,A](a: A): Tr[E,S,S,A]
```

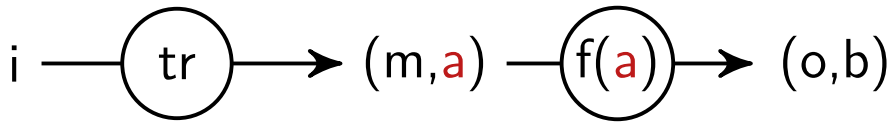


– **Parallel composition** `tr1|tr2`


```
def choice[E,I,O,A]
  (tr1: Tr[E,I,O,A], tr2: => Tr[E,I,O,A]): Tr[E,I,O,A]

def fail[E,S,A]: Tr[E,S,S,A]
```

Combinators – Building Blocks

– **Sequential composition**



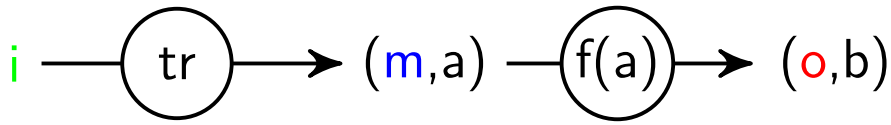

```
def flatMap[E,I,M,O,A,B]
  (tr: Tr[E,I,M,A])(f: A => Tr[E,M,O,B]): Tr[E,I,O,B]

def success[E,S,A](a: A): Tr[E,S,S,A]
```


– **Parallel composition** `tr1 | tr2`


```
def choice[E,I,O,A]
  (tr1: Tr[E,I,O,A], tr2: => Tr[E,I,O,A]): Tr[E,I,O,A]

def fail[E,S,A]: Tr[E,S,S,A]
```

Combinators – Building Blocks

– **Sequential composition**



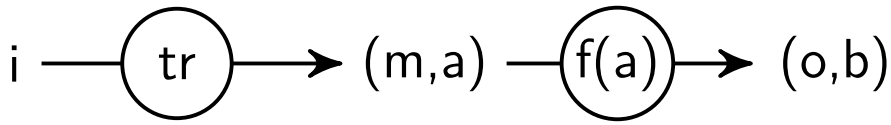

```
def flatMap[E,I,M,O,A,B]
  (tr: Tr[E,I,M,A])(f: A => Tr[E,M,O,B]): Tr[E,I,O,B]

def success[E,S,A](a: A): Tr[E,S,S,A]
```


– **Parallel composition** `tr1 | tr2`


```
def choice[E,I,O,A]
  (tr1: Tr[E,I,O,A], tr2: => Tr[E,I,O,A]): Tr[E,I,O,A]

def fail[E,S,A]: Tr[E,S,S,A]
```

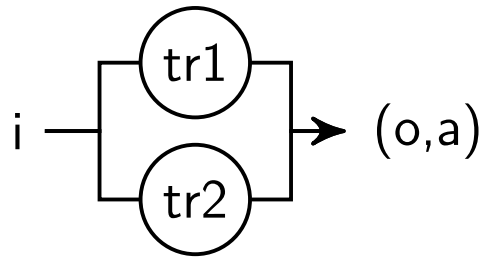
Combinators – Building Blocks

– **Sequential composition**

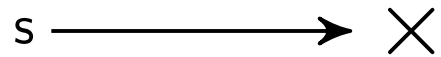



```
def flatMap[E,I,M,O,A,B]
  (tr: Tr[E,I,M,A])(f: A => Tr[E,M,O,B]): Tr[E,I,O,B]

def success[E,S,A](a: A): Tr[E,S,S,A]
```

– **Parallel composition** `tr1|tr2`

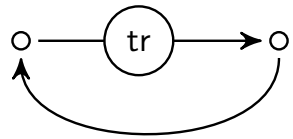


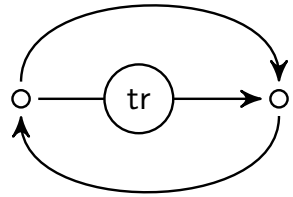

```
def choice[E,I,O,A]
  (tr1: Tr[E,I,O,A], tr2: => Tr[E,I,O,A]): Tr[E,I,O,A]

def fail[E,S,A]: Tr[E,S,S,A]
```

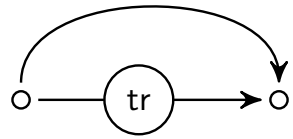


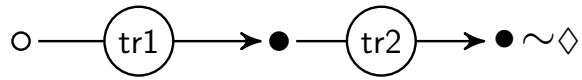
Combinators

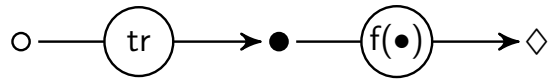
`many1(tr)`	`tr.+`
`many(tr)`	`tr.*`
`opt(tr)`	`tr.?`
`seq(tr1, tr2)`	`tr1 ~ tr2`
`map(tr, f)`	`tr ^^ f`
`filter(tr, p)`	`tr.filter(p)`

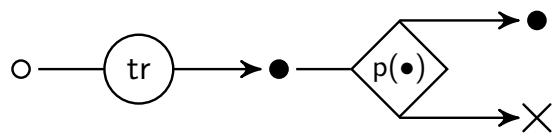




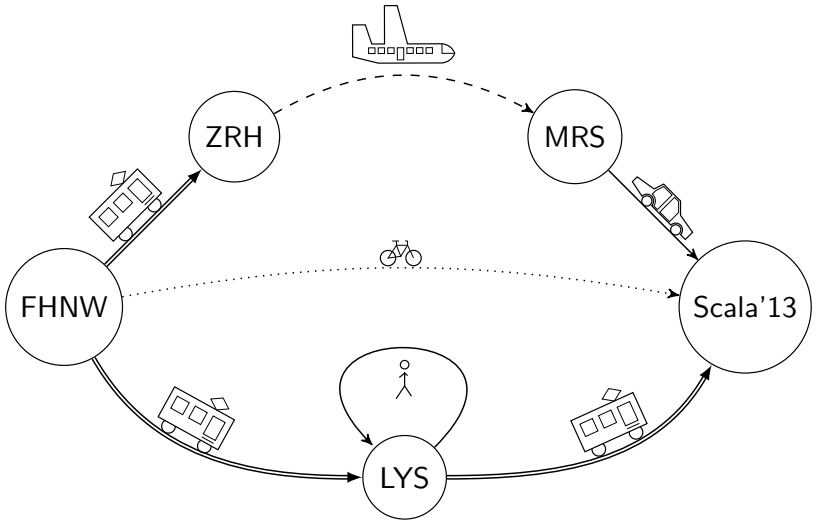




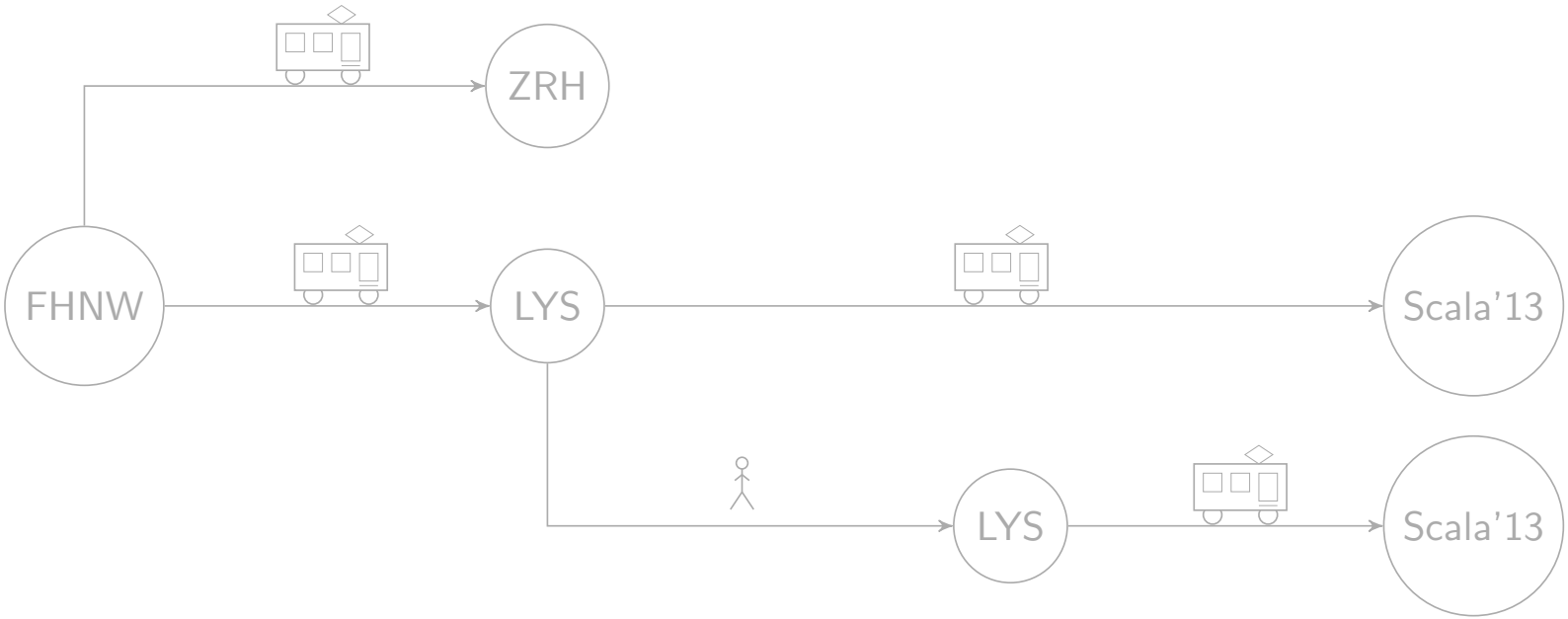




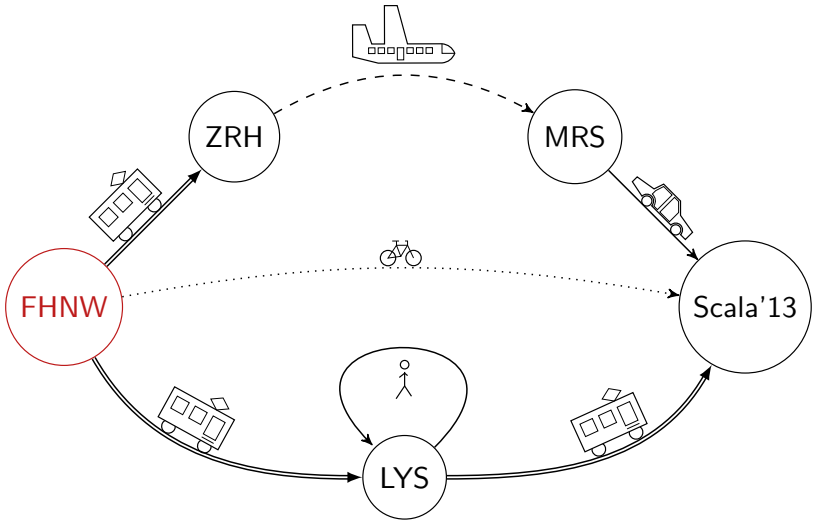
Travel to Montpellier



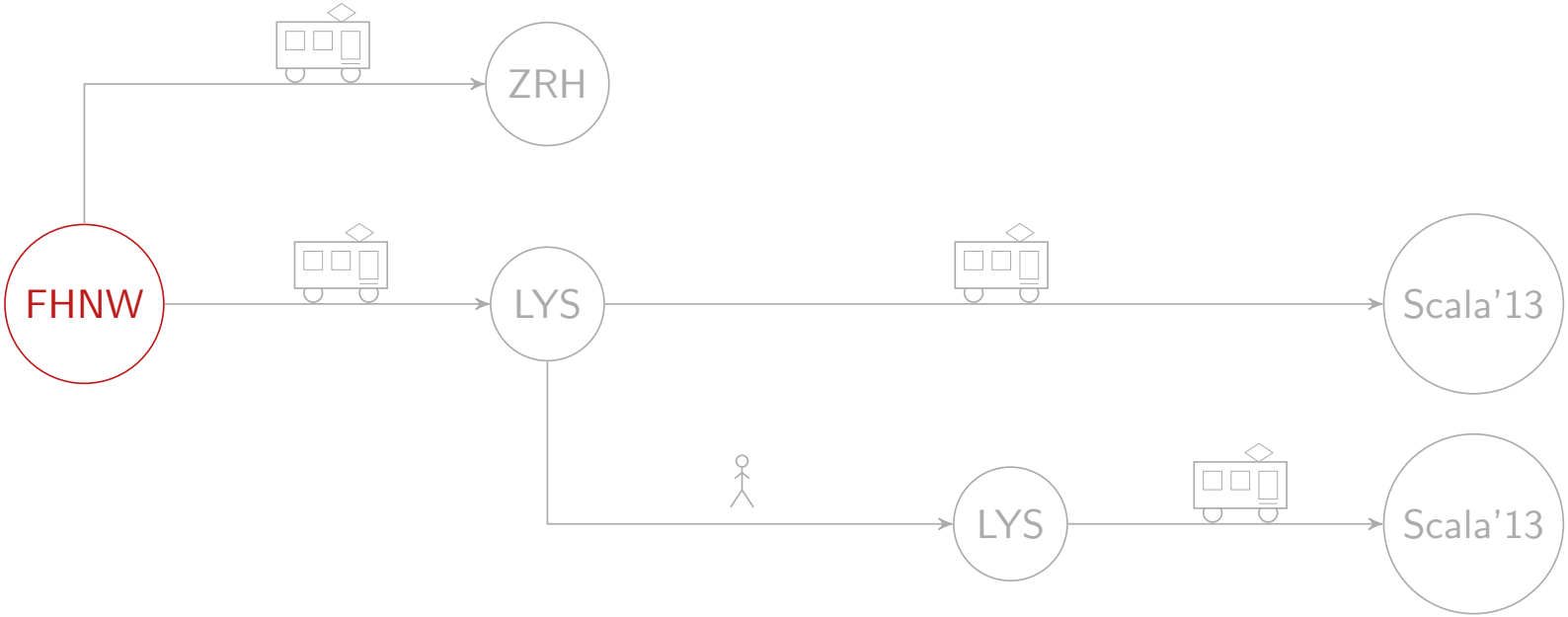

```
val journey =
```



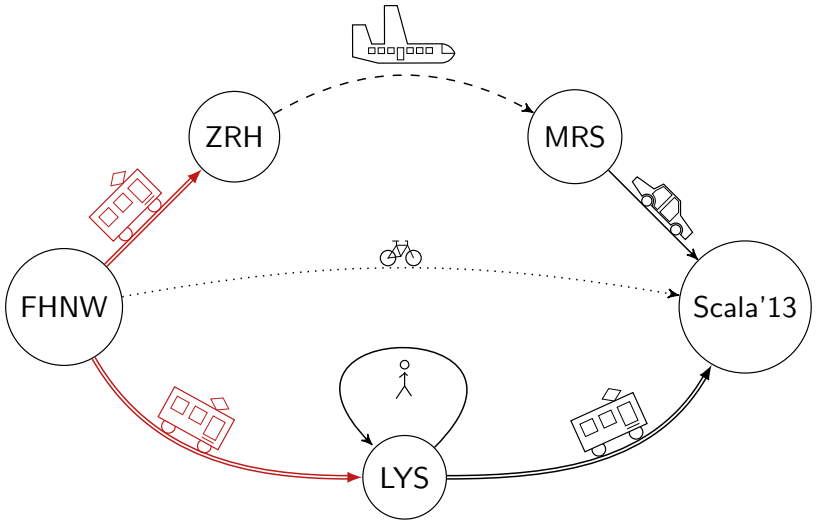
Travel to Montpellier



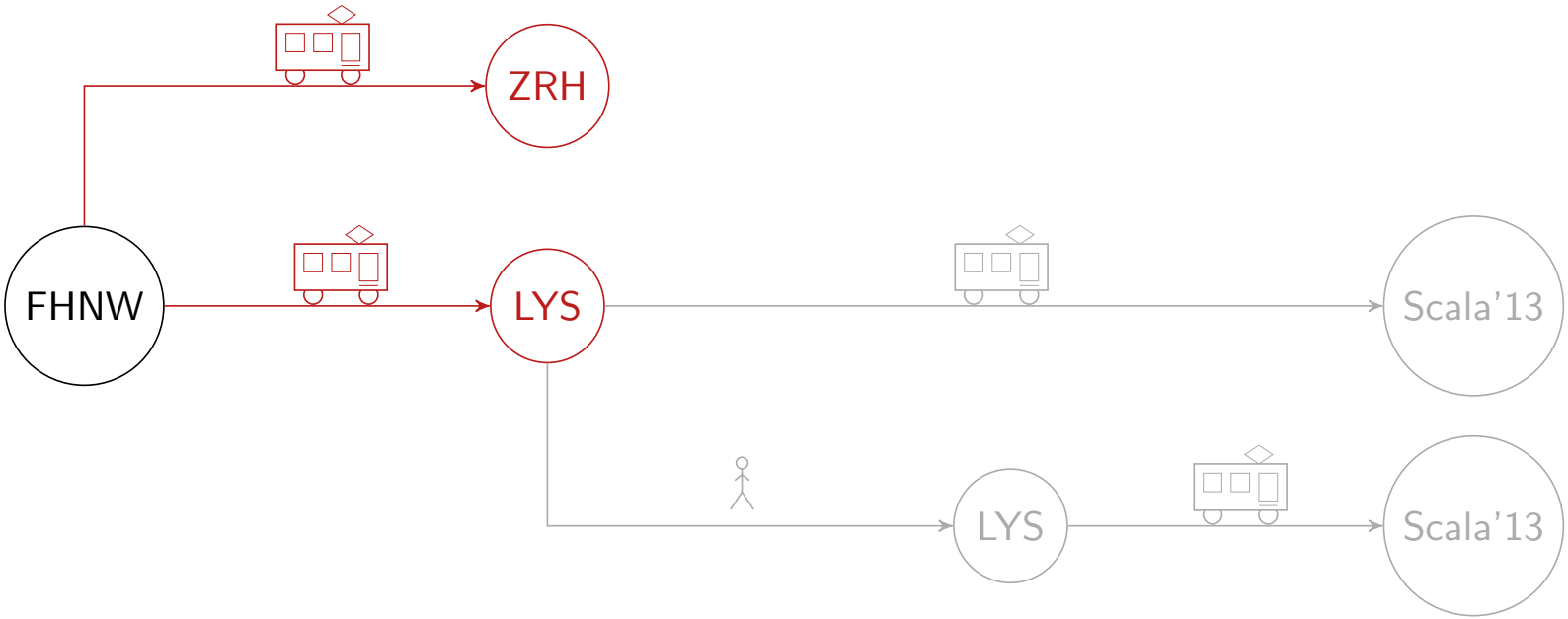

```
val journey =
```



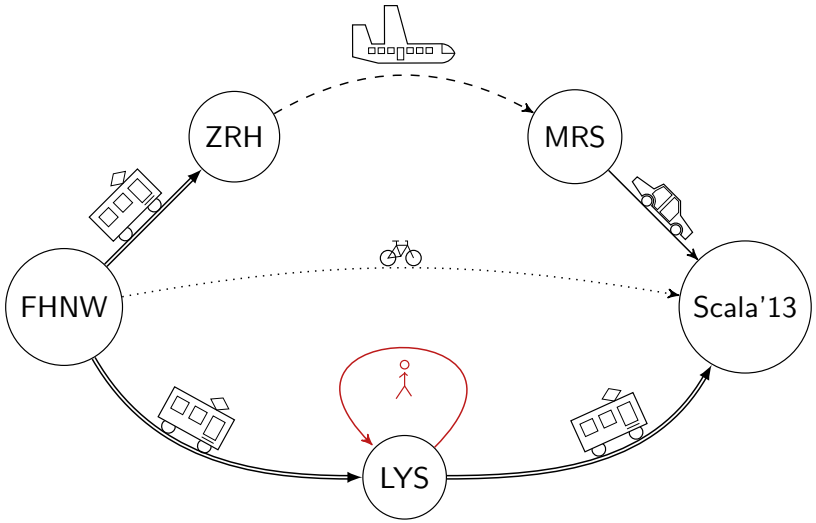
Travel to Montpellier



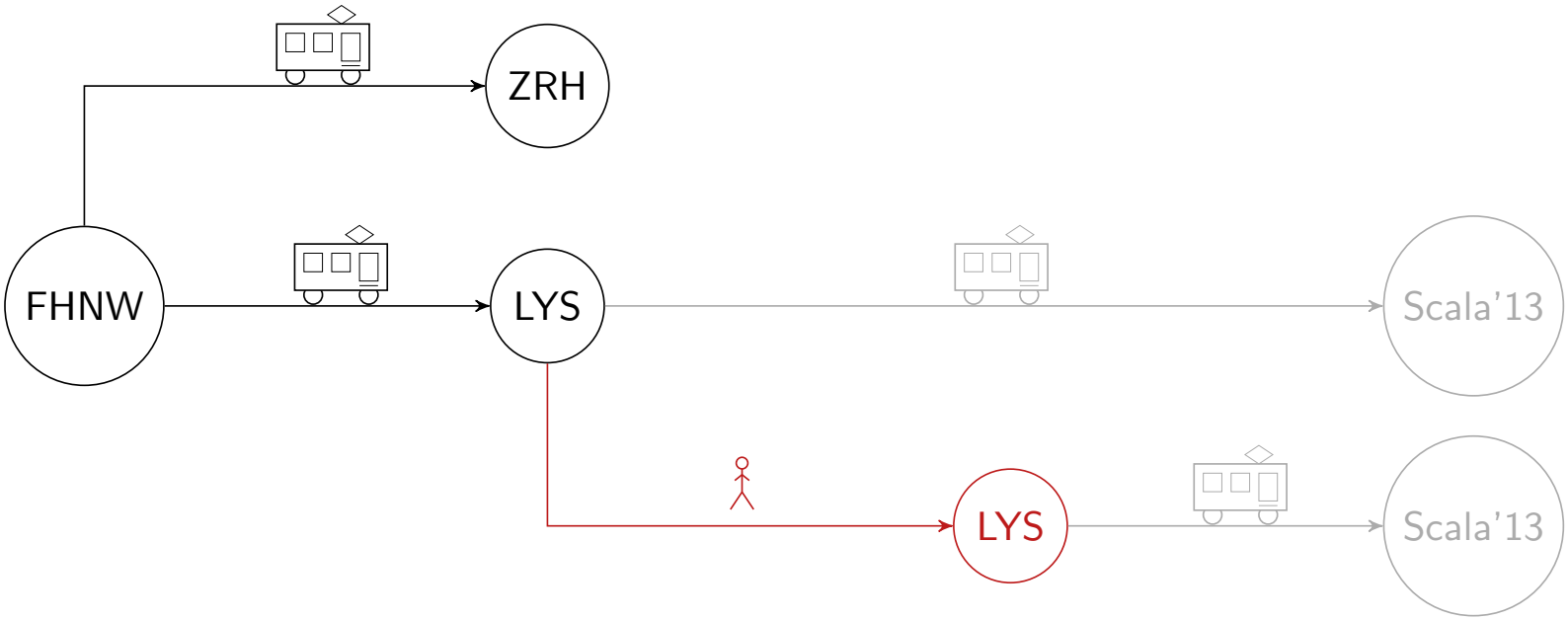

```
val journey =
```



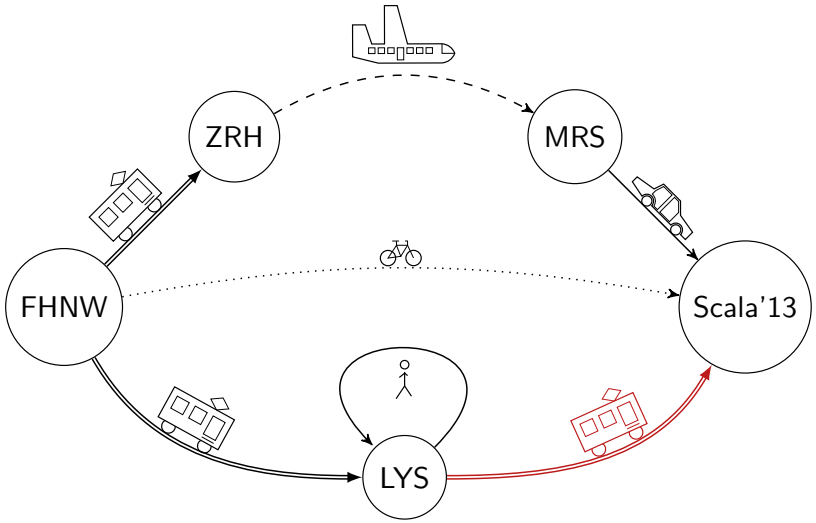
Travel to Montpellier



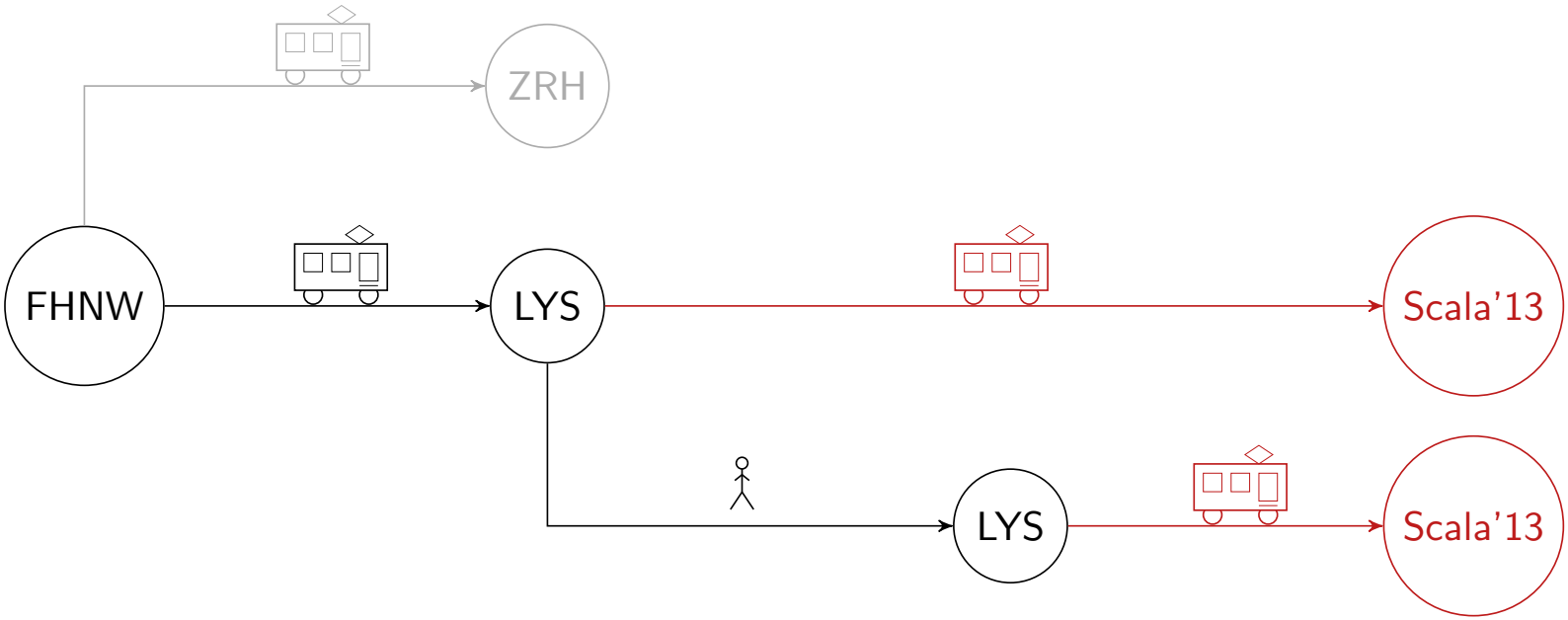

```
val journey =
```



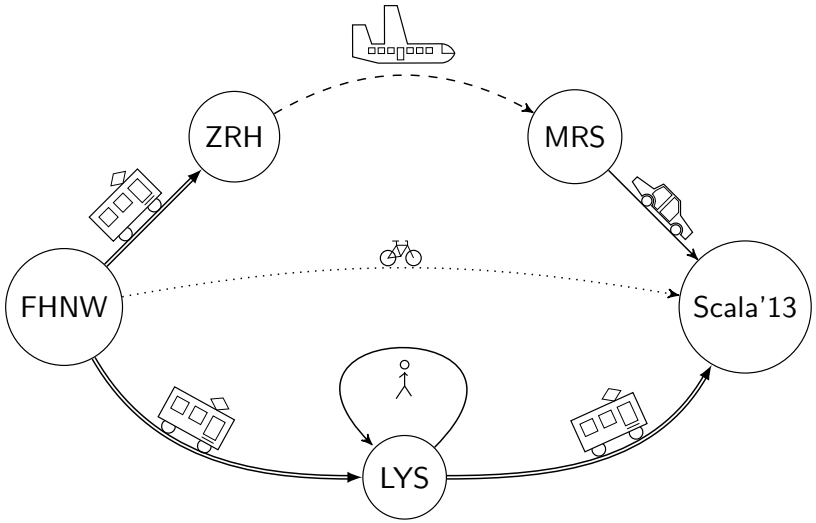
Travel to Montpellier



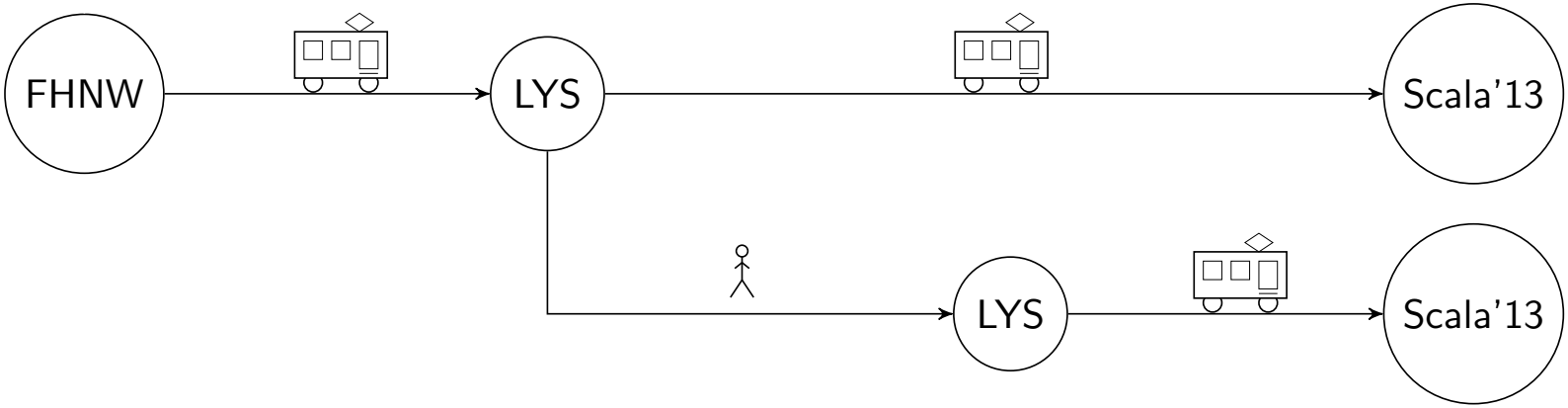

```
val journey =
```



Travel to Montpellier




```
val journey =
```



Extensions

- **Cycle detection**
  Repetition combinators yield every path at most once.
- **Subqueries**
  Allow nested traversals without polluting the path.
- **Type safety**
  Validate traversals against a static schema.
- **Labels**
  Collect and name values while traversing.

Extensions

- **Cycle detection**
  Repetition combinators yield every path at most once.

- **Subqueries**
  Allow nested traversals without polluting the path.

- **Type safety**
  Validate traversals against a static schema.

- **Labels**
  Collect and name values while traversing.

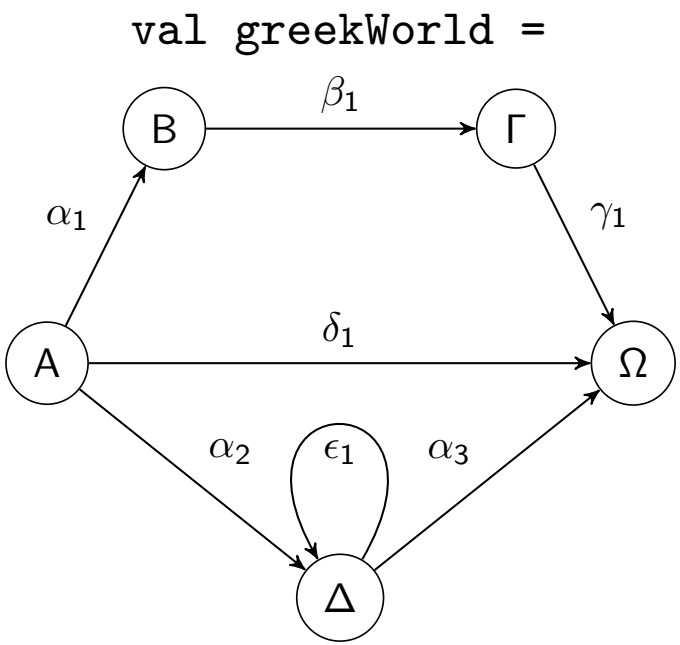
Extensions

- **Cycle detection**
  Repetition combinators yield every path at most once.
- **Subqueries**
  Allow nested traversals without polluting the path.
- **Type safety**
  Validate traversals against a static schema.
- **Labels**
  Collect and name values while traversing.

Extensions

- **Cycle detection**
  Repetition combinators yield every path at most once.
- **Subqueries**
  Allow nested traversals without polluting the path.
- **Type safety**
  Validate traversals against a static schema.
- **Labels**
  Collect and name values while traversing.

Real Code




```
val question =
  V("A") ~ out(α) ~ out(ϵ).* ~> out(α)
val answer = Tr.run(question, greekWorld)

assert(answer.size === 2)

assert(answer.toSet === Set(
  (List(A, α2, Δ, α3, Ω), Ω),
  (List(A, α2, Δ, ϵ1, Δ, α3, Ω), Ω)
))
```

Parsing Graphs

Summary

- Graph traversals can be described using navigation primitives and grammar constructions.
- We have shown how to implement this idea as a purely functional combinator library.

Outlook

- Breadth first traversals
- Performance optimizations

`https://github.com/danielkroeni/trails`

- Implementations for neo4j and blueprints

Parsing Graphs

Summary

- Graph traversals can be described using navigation primitives and grammar constructions.
- We have shown how to implement this idea as a purely functional combinator library.

Outlook

- Breadth first traversals
- Performance optimizations

`https://github.com/danielkroeni/trails`

- Implementations for neo4j and blueprints

Parsing Graphs

Summary

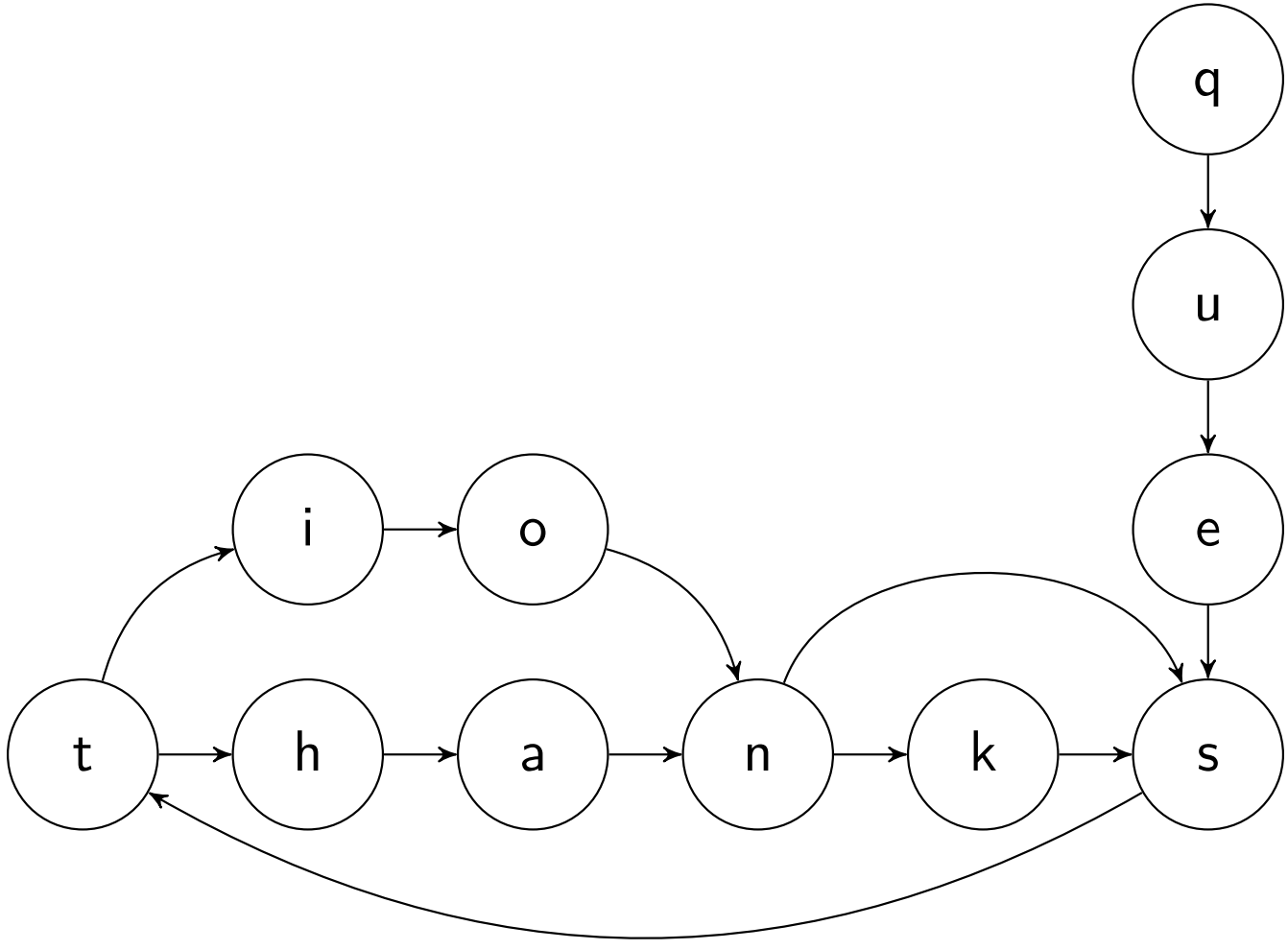
- Graph traversals can be described using navigation primitives and grammar constructions.
- We have shown how to implement this idea as a purely functional combinator library.

Outlook

- Breadth first traversals
- Performance optimizations

`https://github.com/danielkroeni/trails`

- Implementations for neo4j and blueprints


q
u
e
i
o
t
h
a
n
k
s