Parsing Graphs: Applying Parser Combinators to Graph Traversals[1]

Connected data such as social networks or business process interactions are frequently modeled as graphs, and increasingly often, stored in graph databases. In contrast to relational databases where SQL is the proven query language, there is no established counterpart for graph databases. One way to explore and extract data from a graph database is to specify the structure of paths (partial traversals) through the graph. We show how such traversals can be expressed by combining graph navigation primitives with familiar grammar constructions such as sequencing, choice and repetition – essentially applying the idea of parser combinators to graph traversals. The result is trails, a Scala combinator library that provides an implementation for the neo4j graph database and for the generic graph API blueprints.

Daniel Kröni, Raphael Schweizer | daniel.kroeni@fhnw.ch

Enter the tangled world of Carol where everything centers on friendship and pets. The graph in Figure 1 shows four people, their relationships and their pets.

Carol has a dog and now she is thinking about a cute name. Of course the name should be unique, at least within her circle of friends. She asks herself: "What names did my friends and their friends etc. give their pets?" Here is how she would state that same question after having read this paper:

```
val friends = V("Carol") ~ (out-
("loves") | out("likes")).+
val petNames =
  friends ~> out("pet") ^^
  get[String]("name")
```

This code defines a traversal through the given graph by specifying the "grammar" of the paths to be followed. It consists of graph navigation steps (*V, out*) and combinators (~, ~>, |, +). Applying *petNames* yields four results:

(Carol -likes-> Dave -pet-> Fluffy, "Fluffy")
(Carol -loves-> Bob -pet-> Murphy, "Murphy")
(Carol -loves-> Bob -likes-> Carol -likes->
Dave -pet-> Fluffy, "Fluffy")
(Carol -loves-> Bob -loves-> Alice -loves->
Bob -pet-> Murphy, "Murphy")

Each result contains the full path of the traversal plus a designated value, in our case, the name property of the pet. In this paper we describe how this works:

- We develop the datatype of a graph traverser.
- We describe a set of functions that allows us to combine graph navigation primitives into expressive traversal descriptions.

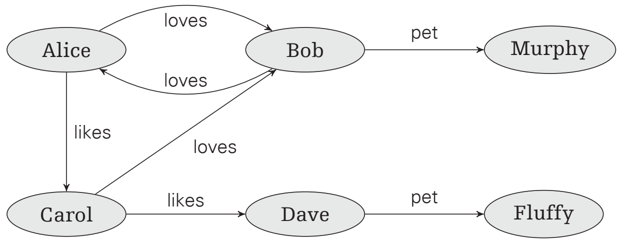


Figure1 : Carol's World, a small example graph

- We extend the library by cycle detection, labeling functionality and subqueries to allow even more elegant traversals.

Graph Traversal Combinators

Step-by-step we develop a graph traversal library. First we shape the type of a traverser. In a first approximation, a traverser Tr_5 is a function that takes a graph as the input and returns a path as the result. *Path* is a list of graph elements that alternate between nodes and edges.

```
type Tr₅ = Graph => Path
```

There may be more than one path that fits the specification of a traverser – or none at all. We account for this by letting the result be a *Stream* of paths, as proposed by Wadler [3]. *Stream* allows us to lazily yield result paths on demand rather than to eagerly compute all results.

```
type Tr₄ = Graph => Stream[Path]
```

A traverser may start with an empty path, but usually it describes an extension of the preceding path. To model this scenario we extend the type of traverser and add the preceding path as a further input parameter:

```
type Tr₃ = Graph => Path => Stream[Path]
```

In the end, a traverser might return an arbitrary value besides the path, for example, the value of a property:

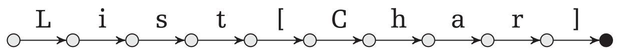
```
type Tr₂[+A] = Graph => Path
  => Stream[(Path,A)]
```

1 Die Originalversion dieses Artikels erschien in den Proceedings of the 4th Workshop on Scala, ECOOP '13, Montpellier, Juli 2013

There is no reason to restrict *Tr*$_2$ to *Graph* and *Path*. We will see that a traverser may read from any environment *E*, it may transform some state *S* and it may yield potentially multiple results:

```
type Tr1[-E,S,+A] = E => S
  => Stream[(S,A)]
```

In this sense, it is a very general structure. In fact, Hutton's and Meijer's monadic parsers [1] are using this type for parsing strings. This is because a *List[Char]* is a graph:



When parsing text, the state is the string being consumed. In contrast, when traversing a graph, we build up the path, which is the sequence of visited nodes and edges. Depending on the head of a path there are different steps we can next take. On an edge, for example, it makes no sense to ask for outgoing edges which are only available on nodes.

To accommodate this fact, we differentiate between the input state type *I* and the output state type *O*. This allows us to statically express whether a traverser expects a node or an edge and thereby rejecting meaningless patterns during compilation.

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

The above is the type of traversers we will use in the following discussion. It is worth noting that it combines three well known monads:

- Nondeterminism: The multiple results of a traverser, represented as a *Stream*.
- Indexed-State: The state which is threaded through – potentially changing its type from *I* to *O*.
- Reader: The read-only environment *E* which is passed to each traverser.

Given the type *Tr*, we can explore primitive graph navigation traversers as well as grammar-like combinators.

Traverser Primitives

The two most basic traversers are *success* and *fail*. *success* creates a traverser that always succeeds with the given value a, leaving the state untouched. *fail* is a traverser that contains no results – a dead end, which allows no further traversal:

```
def success[E,S,A](a: A): Tr[E,S,S,A] =
  _ => s => Stream((s,a))
def fail[E,S,A]: Tr[E,S,S,A] =
  _ => _ => Stream()
```

In a similar fashion, we define the three traversers *getEnv* to read the environment, *getState* to read the state, and *setState* to write the state. Below are their signatures:

```
def getEnv[E, S]: Tr[E, S, S, E]
def getState[E, S]: Tr[E, S, S, S]
def setState[E, I, O](o: O):
  Tr[E, I, O, Unit]
```

Until now, the presented traversers have not been specific to graph traversal but were primitive building blocks for more specific traversers. We will now focus on the graph-specific navigation traversers for directed graphs which consist of nodes and edges, each with associated key-value pairs. The environment and its corresponding graph-element types are fixed to an implementation-dependent graph API. The accompanying state carries the type of the head of its path[2] as a phantom type:

```
import org.neo4j.{graphdb => neo4j}
type GraphAPI =
  neo4j.GraphDatabaseService
type Elem = neo4j.PropertyContainer
type Node <: Elem = neo4j.Node
type Edge <: Elem = neo4j.Relationship
case class State[+Head <: Elem]
  (path: List[Elem])
```

To navigate the graph we propose a few primitives whose names are borrowed from *Gremlin* [9]. Navigation primitives extend their input path by appending the elements they yield. The following traversers need to be implemented for each specific graph database:

Function	Description
`V, V(id)`	all nodes, node identified by id
`E, E(id)`	all edges, edge identified by id
`outE, outE(t)`	all out-edges, out-edges with tag t [3]
`inE, inE(t)`	all in-edges, in-edges with tag t [3]
`outV, inV`	start node, end node of an edge

As an example, we will look at *outE*'s function signature:

```
def outE(tagName: String):
  Tr[GraphAPI,State[Node],State[Edge],
  Edge]
```

which has the following expanded return type:

```
GraphAPI => State[Node]
  => Stream[(State[Edge],Edge)]
```

This function takes a graph and a path that ends in a *Node* and from there it steps onto all outgoing *edges* with the given *tagName*. This leads to paths which end in an Edge. Together, this edge is then returned with the extended path.

In order to access properties on nodes and edges, the function get must be implemented as well:

```
def get[A](key: String)(e: Elem): A
```

These primitives will be combined into powerful traversal definitions.

2 In principle, it is sufficient to track only the current position in the graph, however, we are often interested in the trace.
3 neo4j and blueprints support tagged edges, in contrast to untagged nodes.

Traverser Combinators

We now want to combine these primitive traversers into complex path expressions, which results again in traversers. This property is a key to their compositional nature.

The following table shows the name of those combinators as well as the sugar we provide to concisely express traversals:

Function	Sugar	Description
`seq`	`a ~ b`	First a then b
`choice`	`a | b`	Follow both branches
`opt`	`a.?`	Repeat 0..1
`many`	`a.*`	Repeat 0..n
`many1`	`a.+`	Repeat 1..n

flatMap is used to sequentially combine any two traversers. It passes through the same environment to both traversers, threads the state through the first traverser into the second one and returns the final states together with the results:

```
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] =
  e => i => tr(e)(i).flatMap {
    case (m,a) => f(a)(e)(m)
  }
```

Note that the inner *flatMap* is called on *Stream*, and how the different input and output state types *I*, *M* and *O* line up – from *[_,I,M,_]* and *[_,M,O,_]* to *[_,I,O,_]*. To allow recursive definitions, all combinators take their traverser arguments by-name.

Tr together with *flatMap* and *success* becomes a structure that is slightly more general than monadic, due to the state types [4]. Luckily, Scala's for-comprehension does not worry about this.

Now *map* and *filter*, using *flatMap*, *success* and *fail*, can be implemented as follows:

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

def filter[E,I,O,A](tr: => Tr[E,I,O,A])
  (f: A => Boolean): Tr[E,I,O,A] =
  flatMap(tr)(a => if(f(a))
  success(a) else fail)
```

There is another, less powerful but often sufficient way to sequentially combine two traversers. *seq* does not use the result of the first traverser to obtain the subsequent traverser as in *flatMap* but simply returns both values in a fancy-looking tuple named ~:

```
case class ~[+A,+B](a: A, b: B)

def seq[E,I,M,O,A,B](fst: => Tr[E,I,M,A],
  snd: => Tr[E,M,O,B]): Tr[E,I,O,A~B] =
  for(a <- fst; b <- snd)
  yield new ~(a,b)
```

The related functions ~, ~> and <~ return the whole tuple, the right-hand-side and the left-hand side. These functions as well as the infix sugar for *map* ^^ are courtesy of Scala's parser combinators [2, 727-755]. They allow the writing of good-looking sequential compositions of traversers such as *out* which first navigates from a node to an outgoing edge and from there to the target node:

```
def out(tagName: String)
  : Tr[GraphAPI,State[Node],State[Node],
Node] = outE(tagName) ~> inV()
```

In addition to the above sequencing function, a means is needed to express branching: *choice*. Since we are interested in all matching result paths this combinator follows *both* arguments using the same state and concatenates (*#:::*) their results. This is different to typical combinator parsers which for reason of speed often try the second alternative only if the first one fails:

```
def choice[E,I,O,A](
  either: => Tr[E,I,O,A],
  or: => Tr[E,I,O,A]): Tr[E,I,O,A] =
  e => i => either(e)(i) #::: or(e)(i)
```

Now we have all the ingredients to implement *opt*, *many* and *many1*. Note that they restrict their argument traverser to start and end on the same state type *S*. The implementations are straight forward:

```
def opt[E,S,A](tr: => Tr[E,S,S,A])
  : Tr[E,S,S,Option[A]] =
  choice(success(None),
  map(tr)(Some[A](_)))

def many[E,S,A](tr: => Tr[E,S,S,A])
  : Tr[E,S,S,Stream[A]] =
  choice(success(Stream()), many1(tr))

def many1[E,S,A](tr: => Tr[E,S,S,A])
  : Tr[E,S,S,Stream[A]] =
  for(a <- tr; as <- many(tr))
  yield a #:: as
```

This concludes the basic functionality of our graph traversal combinators. Improvements to this minimalistic design are discussed in the further sections. We make our traversers cycle-aware and add the ability to label values, which can then be referred to in queries. Finally we show how to implement subqueries.

Cycle Detection

Consider the following traverser:

```
V("Alice") ~> (out("loves")
| out("likes")).+
```

Since there is no inherent ordering of the edges, a possible sequence of result paths could look like this:

Alice -loves-> Bob
Alice -loves-> Bob -loves-> Alice

Alice -loves-> Bob -loves-> Alice -loves-> Bob ...

The given implementation would never stop generating longer and longer expansions of the cycle and never yield the following path: *Alice -likes-> Carol*.

Note that the queried graph does not need to contain cycles: e.g. *(out("pet") ~ in("pet")).+* is problematic by itself.

In general the application of *many* and *many1* may cause problems. Clearly this behavior is undesirable. Cycles should be detected and handled appropriately. Our implementation adheres to the following definition: If, within an application[4] of *many* or *many1* , the repeated traverser yields the same snippet a second time, then it is a cycle. Consistent with our definition this path, *Carol -loves-> Bob -loves-> Alice -likes-> Carol -loves-> Bob -pet-> Murphy*, is discarded from the result mentioned in the introduction due to the repeated *-loves-> Bob* snippet.

Detecting cycles requires the snippets to be tracked, therefore we extend the state:

```
case class State[+Head <: Elem]
  (path: List[Elem],
  cycles: Set[List[Elem]])
```

For the sake of simplicity our implementation follows cycles only once, which might be returned as part of the result as well.

Labels

As a further extension, we allow the values that are emitted by a traverser to be labeled. This requires additional state of type *Map[String,List [Any]]* which maps a label to a list of values. Why use a list of values and not just a single value? The answer is that labeling inside a repetition might produce more than one value, or perhaps none at all.

For example an application of labels is looking for unhappy lovers – people who love another person but that person does not return this love:

```
val unhappyLovers = for {
  beloved <- V.as("lvr") ~
  out("loves") ~> out("loves")
  lover <- label("lvr") if
  !lover.contains(beloved)
} yield lover
```

Executing this query on the introductory graph yields the single node *Carol*.

Subqueries

The last extension we implement are subqueries. Essentially subqueries are traversers whose values are preserved while their state changes are discarded. Thus subqueries allow to "match" patterns without having the matched paths polluting the result. Here is the definition of *sub* which runs its argument *tr* as a subquery and in turn yields the stream of *tr*'s results:

```
def sub[E,I,O,A](tr: Tr[E,I,O,A])
  : Tr[E,I,I,Stream[A]] =
  e => i => Stream((i, tr(e)
  (i).map(_._2)))
```

Using *sub* we can search for beloved pet owners:

```
val belovedPetOwners = for {
  petOwner <- V
  pets <- sub(out("pet"))
  if pets.nonEmpty
  lover <- in("loves")
} yield (petOwner, lover)
```

This yields (*Carol -loves-> Bob, (Bob, Carol)*) and *(Alice -loves-> Bob, (Bob, Alice))*. Note that the pets do not show up in the result.

4 Top-level applications only, not mutual recursive calls.

Conclusion and Related Work

We have developed a simple combinator library to concisely express graph traversals. It is currently being evaluated and extended in the context of a business intelligence project [10]. Ongoing work can be observed on the *trails* webpage [6].

To access information stored in graph databases we have found low-level APIs, imperative, embedded graph traversal languages such as the Gremlin family [9] and declarative approaches e.g. Cypher [11] or SPARQL [12]. While others stress expressiveness or good computational complexity [5] *trails* focuses on simplicity – in terms of an educative value, implementation and application.

References

[1] G. Hutton & E. Meijer. Monadic parser combinators, 1996.

[2] M. Odersky, L. Spoon, and B. Venners. Programming in Scala: A Comprehensive Step-by-Step Guide. Artima Inc., 2nd edition, 2010.

[3] P. Wadler. How to replace failure by a list of successes. In Conference on Functional Programming Languages and Computer Architecture, pages 113--128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[4] P. Wadler. Monads and composable continuations. LISP and Symbolic Computation, 7(1):39--55, jan 1994.

[5] P. T. Wood. Query languages for graph databases. SIGMOD Record, 41(1):50--60, 2012.

[6] http://www.github.com/danielkroeni/trails, 2013.

[7] http://www.neo4j.org, 2013.

[8] http://blueprints.tinkerpop.com/, 2013.

[9] http://gremlin.tinkerpop.com/, 2013.

[10] http://www.fhnw.ch/technik/imvs/forschung/projekte/babefisch/babelfish, 2013.

[11] http://docs.neo4j.org/chunked/stable/cypher-query-lang.html, 2013.

[12] http://www.w3.org/TR/rdf-sparql-query, 2013.