Traversal Query Language For Scala.Meta

Technical Report

Eric Béguet Eugene Burmako

EPFL, Switzerland
{first.last}@epfl.ch

ABSTRACT

With the rise of metaprogramming in Scala, manipulating ASTs has become a daily job. Yet the standard API provides only low-level mechanisms to transform or to collect information on those data structures. Moreover, those mechanisms often force the programmer to manipulate state in order to retrieve information on these ASTs.

In this report we try to solve those problems by introducing TQL, a high-level combinator Scala library to transform and query data structures in a purely functional way. Parser combinators allow to combine several small parsers to build a bigger one in an expressive way. In this report, we argue that we can apply the same concept to data structure manipulation and construct complicated traversers on top of smaller ones. Yet combinators may feel unnatural or too complicated for certain usage. We therefore built a library on top of TQL to manipulate data structures as a collection.

We then put TQL in practice to scala.meta ASTs, and describe the challenges we face when traversing a real-word data structure, especially performance-wise.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*AST Traversal and Transformation*

General Terms

Languages, Performance

Keywords

combinators, macros, scala, traversal, transformation, optimization

1. INTRODUCTION

1.1 The state of AST traversal and transformation in Scala

```
val allValNames = new Traverser {
  var valNames = Set.empty[String]
  override def traverse(tree: Tree) = tree match {
    case ValDef(_, name,_ ,_) => valNames += name.toString
    case _ => super.traverse
  }
  def apply(tree: Tree) = {
    traverse(tree)
    valNames
  }
}.apply(tree)
```

Figure 1: A simple AST traverser in Scala

The current reflection API in Scala provides only a few basic mechanisms to traverse and transform Scala's ASTs.

Figure 1 presents a simple traverser which collect all different variable names in a given `tree` (a Scala AST). We can see that even for such a simple task, the programmer has to write a lot of boilerplate, while the only *interesting* part is the following line

```
ValDef(_, name,_ ,_) => valNames += name.toString
```

Moreover, the user is forced to manipulate state and to add boilerplate in order to collect the variable names. Furthermore the traversal strategy is implied and is always top-bottom DFS, we cannot change it. AST transformation in Scala currently looks quite the same except that instead of defining a new `Traverser` the programmer needs to instantiate a `Transformer` and override its `transform` method.

The rise of meta-programming in Scala allows access to Scala ASTs to a lot more people than only compiler hackers. This implies that AST manipulation is much more used and that there should exist a way of making it easier to work with.

1.2 Following on, toward a functional traversal API

Our motivations to have a better AST traversal API in Scala lead us to the following requirements for such an API:

- It should be easy to handle basic and repetitive tasks like collecting information on a particular sub-tree (Figure 1) or transforming a sub-tree into another sub-tree.
- It should be possible to handle complex traversal mechanisms while still being fairly easy to write and to read.
- State manipulation should be the exception and not the common case. Arguably purely functional programming makes the code easier to understand and

```
int cyclomaticComplexity(Program p) {
  n = 1;
  visit (p) {
    case ifStat (_ ,_ ,_ ): n += 1;
    case whileStat (_ ,_ ,_ ): n += 1;
    ..
  }
  return n;
}
```

Figure 2: Cyclomatic Complexity rule in Rascal, taken from [21]

to reason about, even though state manipulation can make some part of the code more natural or simpler to write.

- We should be able to run advanced traversal techniques, such as fusing [1] or parallelization in order to improve performances. This requirement would make the API scalable and not force the programmer to change their code into a more low-level API in a performance critical environment.
- Additionally it would be nice if the library was not bound to any particular data structure, not only Scala ASTs. That means that we should be able to use the API on any data structure which presents a specific set of characteristics.

1.3 Contributions

In this section we stated our motivation and requirements for a new traversal API. We should therefore present TQL or Traversal Query Language, a functional library for data structure traversal and transformation in Scala.

The rest of this work is organized in the following way. In section 2 we review the related work, how we came out with the requirements and design of TQL. Section 3 focuses on the design and implementation of TQL. Section 4 shows a practical utilization of TQL in scala.meta. Section 5 discuss the user interface of TQL and how it can be improved. Section 6 presents examples of code written in the current Scala traversal API and how it is translated with TQL. Section 7 discuss the limitations of our API and what can be done in future work. Finally we conclude by summarizing what we learnt, what works and what does not and by briefly introducing Obey: code health for scala.meta, a project which uses TQL.

2. RELATED WORK

2.1 Inspiration

We present here the different existing traversal and transformation frameworks which influence the design of TQL.

2.1.1 Rascal

Rascal [21] is a metaprogramming language used to analyze, generate and manipulate source code. Rascal rewrite rules relies on immutable data, pattern matching and higher-order functions. Those constructs being already at hand in Scala, it makes sense to take inspiration from Rascal to design our library.

Figure 2 presents a small example of a traverser written in Rascal. We can consider `visit` as the `Traverser` and its second parameter as a partial function applied to each node it traverses. One interesting element to notice is that the function generates the result in an imperative way, much like we currently do with Scala's `Traversers`. Figure 3 presents a simple Rascal rewrite rule. Again we can see some similarities with Scala's `Transformers`. The interesting element of the code in that example is that it is possible to define a traverser strategy: the keyword `innermost` before `visit`. Taken from the Rascal documentation [10] we present here 4 traversal strategies which seems relevant to integrate in TQL:

- top-down: visit the subject from root to leaves.
- top-down-break: visit the subject from root to leaves, but stop at the current path when a case matches.
- bottom-up: visit the subject from leaves to root (this is the default).
- bottom-up-break: visit the subject from leaves to root, but stop at the current path when a case matches.

```
CompilationUnit idiomatic(CompilationUnit unit) = innermost visit(unit) {
  case (Stm) `if (!<Expr cond>) <Stm a> else <Stm b>` =>
    (Stm) `if (<Expr cond>)  <Stm b> else <Stm a>`

  case (Stm) `if (<Expr cond>) <Stm a>` =>
    (Stm) `if (<Expr cond>) { <Stm a> }`
    when (Stm) `<Block _>` !:= a

  case (Stm) `if (<Expr cond>) <Stm a> else <Stm b>` =>
    (Stm) `if (<Expr cond>) { <Stm a> } else { <Stm b> }`
    when (Stm) `<Block _>` !:= a

  case (Stm) `if (<Expr cond>) { return true; } else { return false; }` =>
    (Stm) `return <Expr cond>;`
};
```

Figure 3: Canonicalization rule in Rascal, taken from [11]

```
forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(
  hasInitializer(integerLiteral(equals(0))))))))
```

Figure 4: LLVM matcher expression which captures all `for` statement that define a new variable initialized to zero. Taken from [7]

2.1.2 LLVM ASTMatchers

`LLVM AST Matchers` [6] provide a DSL to query and transform Clang AST in C++. `Matchers` are small predicate objects which *match* a certain node of the AST. For instance the `forStmt Matcher` matches **`for`** statements. There are different categories of `Matchers` but the important distinction is between `Matchers` that operate on a node and `Matchers` that define a traversal strategy. Then writing queries or rewrite rules is simply a matter of combining those `Matchers` together in a *matcher expression*. Figure 4 presents such a *matcher expression*. We also take that approach of combining`Matchers` in TQL.

2.1.3 Combinators - HXT

HXT [3] is a combinator library used to traverse and transform XML documents in Haskell. Compared to the other libraries we have introduced HXT does not rely on mutable state in order to produce a result, but rather each combi-

nator returns the result it has computed. Let us consider the following combinator which selects all tables in an XML document:

```
multi (isElem >>> hasName "table")
```

We break down the above code in 4 parts.

- multi: is a *meta-combinator* which takes another combinator as argument. In this case it defines a *top-down* traversal strategy. In fact it is equivalent to the *top-down* traversal strategy from Rascal defined in the above subsection.
- isElem: is a combinator matching an XML element (of the form $<element\ ...>$.
- hasName name: is a combinator which matches when the current node contains an attribute with the name *name*.
- `>>>`: is used to glue `isElem` and `hasName` together so that the latter is only tried when the first one succeeds.

With HXT it is also possible to transform an XML document into another, but the combinators used are all different from the ones used for traversal. We aim to prevent that in TQL.

2.1.4 Combinators - Kiama

We discovered Kiama [5] when the project was well under way. Kiama is Scala library for language processing. It also provides a combinator library inspired by Stratego [18] for the analysis and transformation of structured data. However it presents some differences with TQL.

1 In order to transform a data structure, a traverser must, at each node of the data structure: 1) recursively transform every child of the node 2) reconstruct the node with the newly transformed children. Kiama's traverser can operate on generic data structures, and its implementation uses Scala reflection to reconstruct the transformed nodes. In TQL we aim to not use reflection, but rather build an user-defined traverser for each data structure.

2 To the best of our knowledge Kiama does not allow to combine rewriting rule and information collection at the same time.

3 TQL provides a mechanism to prevent unsafe transformations, such as transforming a `Term` into a `Type`, by providing a type class to whitelist rewritings. In earlier Kiama versions, rewrite rules weren't strongly typed, but the recent versions have rewrite rules that are built from functions that must be `T => T` for some type `T`.

Moreover Kiama supports *attribute grammars*, which are not part of the TQL design. We presently do not intend to make TQL a full language processing tool, but only a traversal and transforming API for arbitrary data structures.

2.2 Other approaches

We are aware of other approaches to data structure traversal and we describe here why we have chosen to not use them in our work.

2.3 Zipper

Several JSON manipulation API in Scala [4, 2] are implemented with a zipper. We did not choose to take this direction because Zippers seem to be more appropriate for manipulating a JSON object or an XML document, when the user wants to change a data at a specific path in the data structure. Moreover it seems complicated to combine several Zippers and even more to fuse them.

2.4 Transducers

The Yield paper [20] presents Transducers as a way to traverse and transform data structures. They represent only a transformation and are not bound to a specific input or output data structure. Those transformations can be composed and chained together such as they do not generate intermediate data structures. While it seems very powerful for linear data structures such as Lists it is not clear how transformation would be carried out on trees with unfixed and unbound number of children such as an AST.

2.5 Scalaz Traverser

Scalaz [13] implements a general Traverser for any *Traversable* data structure as described in *The Essence of the Iterator Pattern* [19]. The structure is very elegant and it is possible to reconstruct/transform the data structure or to collect information and execute reduce operations on it. Indeed any kind of loop can be represented on top of the `traverse` operation:

```
trait Traversable[T[_]] {
    def traverse[F[_] : Applicative, A, B](f: A => F[B]): T[A] => F[T[B]]
}
```

Here we present some examples taken from [23]:

```
val tree: BinaryTree[Int] = Bin(Leaf(1), Leaf(2))
tree.contents must_== List(1, 2)
def count[A]: T[A] => Int = reduce((a: A) => 1)
tree.count must_== 2
```

where `content` is implemented in term of reduce, which is implemented on top of `traverse`.

```
def contents[A]: T[A] => List[A] = reduce((a: A) => List(a))

def reduce[A, M : Monoid](reducer: A => M): T[A] => M = {
  val f = (a: A) => Const[M, Any](reducer(a))
  (ta: T[A]) => traverse[({type l[A]=Const[M, A]})#l, A, Any](f)
  .apply(ta).value
}
```

While the Traverser pattern seems convenient for simple data structures such as a binary tree, there is a lot of boilerplate involved and it was deemed too intricate to make it work on a complex ASTs such as scala.meta trees.

2.6 XPath

XPath is a small language to navigate through XML documents. It has been considered to apply XPath-like queries to our API through a Scala's DSL, but we found out that it was not easily composable. In section 3 we will present a minimalist API on top of combinators which aim to mimic some basic functionality of XPath over arbitrary data structure.

3. TQL - TRAVERSABLE QUERY LANGUAGE

3.1 Architecture

3.1.1 Building blocks: Matcher, MatchResult, Monoid

We call TQL combinators `Matcher` (we borrow the term from LLVM ASTMatchers). A `Matcher` is basically a function from `T` to `MatchResult[A]`, where A is the type of the information collected on the data structure during the traversal. Thus

the application of an element of type `T` on a `Matcher` yield a `MatchResult[A]`, which is an alias for `Option[(T, A)]`. Thus, each combinator returns a transformed `T` and a *result* `A`. This allows to easily construct a traverser which combine results and reconstruct the data structure at the same place. Likewise this allows to transform the data structure and to collect information on it at the same time.

In order to easily compose *results* we require them to be an instance of a *Monoid*. A *Monoid* `A` is an algebraic structure which has one associative binary operation: `append(a: A, b: A): A` and an identity element `zero: A`. In Scala this can be modeled by the following trait:

```
trait Monoid[A]{
  def zero : A
  def append(a: A, b: A): A
}
```

Hence the user does not have to worry about result composition during the traversal and we can freely choose when to compose the results in the implementation of TQL.

Some basic *Monoid* instances are available in the *Monoid* companion object among which:

- `Monoid[List[A]]`
- `Monoid[B[A]] where B[A] <: Traversable[A]` e.g. Sequence, Set...
- `Monoid[B[(A, C)]] where B[(A, C)] <: Traversable[(A, C)]` e.g. Map...
- `Monoid[(A, B)]` where both A and B are also Monoids
- `Monoid[Unit]` when we do not return a result.

3.1.2 Combining traversal and transformation

In TQL we consider traversal as a special case of transformation. One important aspect of TQL is the ability to collect information about the traversed data structure and to transform the data structure at the same time. In order to provide this feature it is necessary for the library to have a traverser which is a transformer at the same time. Every data structure that is to be manipulated with TQL has to have a traverser that conforms to the following interface:

```
trait Traverser[T] {
  def traverse[A : Monoid](t: T, f: Matcher[A]): MatchResult[A]
}
```

where `T` is the type of the data structure we wish to traverse. We provide such a traverser for scala.meta trees (see section 4)

3.1.3 Combinators

We define basic combinators (`Matchers`) in the trait `Combinator[T]`. We distinguish simple combinators such as `focus` or `transform` from *meta combinators*. Meta combinators are combinators which take other combinators as arguments. For example, combinators which define a traversal strategy are meta-combinators, because they traverse the data structure and try to apply another combinator on each of the traversed elements. Several traversal strategies are defined:

- `topDown`: Top down traversal
- `topDownBreak`: Top down traversal but stop as soon as it matches an element
- `bottumUp`: Bottom up traversal
- `bottomUpBreak`: Bottom up traversal but stop as soon as it matches an element

3.1.4 AllowedTransformation

The `AllowedTransformation[I, O]` implicit is a opt-in mechanism to help ensure that only some transformations are executed. If we do not want to allow the user to transform a `Literal` into a `Type` then we simply do not provide an implicit `AllowedTransformation[Literal, Type]`. We will talk about it more in details when we describe the `transform` combinator.

3.2 Deep dive into the API

3.2.1 Combinators

Collect.

We find that one of the most prevalent use cases of AST traversal is to put the result of the traversal into a sequence. For example: returning all function names in a `List`, or returning the name of every variable appearing in the code into a `Set`. The collect combinator does just that, here is its interface:

```
def collect[C[_]] = new CollectInType[C] {
  def apply[A, R](f: PartialFunction[T, A])
  (implicit y: Collector[C[A], A, R]): Matcher[R]
}
```

Its definition is two-fold. First we create a `CollectInType[C]` object which defines the type (`C`) of the collection we want to collect in. Secondly we use its `apply` method to define which type `A`) of element we want to store in the collection. Most of the time, the type of the collection will be `List`. Thus, in order not to force the user to write `collect[List]` in those cases, we mimic a *default type parameter argument* to allow the user to omit the type parameter when he wants to collect in a `List`. We implement this by using the implicit resolution mechanism and by using the following rules:

```
trait Collector[C, A, R]
implicit def nothingToList[A]... =  new Collector[Nothing, A, List[A]]
implicit def otherToCol[A, C[A]]... =  new Collector[C[A], A, C[A]]
```

Where we say that:

- `C` is the type of the inferred Collection type (it is inferred to Nothing if not specified)
- `A` is the type of the element we want to make a collection of.
- `R` is the *real* type of the collection which will be used to return the results.

The `f` parameter is a partial function from `T` to A which simply put the result of `f` in the collection at the nodes `T` where it is defined. This allows us to write the followings:

```
//put every Integer literals in a List.
topDown(collect{case Lit.Int(a) => a})
//put all different String literas in a Set.
topDown(collect[Set]{case Lit.String(a) => a})
```

Visit.

The result of the `visit` combinator must be an instance of a `Monoid`. The difference between the `visit` and the `collect` combinator is that in the latter we put the result directly in a collection. The visit combinator is a more general form of collect:

```
visit{case Lit.Int(a) => List(a)}
```

is equivalent to

```
collect{case Lit.Int(a) => a}
```

For example we can use it to to sum all Integer literals in a scala.meta AST, provided an instance of a Monoid summing Integers:

```
implicit val addMonoid = new Monoid[Int] {
  def zero = 0
  def append(a: Int, b: Int) = a + b
}
val sumAll = topDown(visit{case Lit.Int(a) => a})
```

Focus.

The focus combinator allows us to move in the data structure without performing any action, it does nothing else than *focus* on a node. Thanks to that combinator we are able to implement a traverser which collects only the name of the variables declared directly in the children of an `Term.If` node:

```
val inIfs = topDownBreak(
  focus{case x:Term.If => true}
  andThen
  children(collect{case Decl.Val(..name..) => name})
)
```

Where `andThen` is a composition operator that we define later and `children` allows us to apply the combinator in argument to the children of the current node.

Transform.

The transform combinator allows the modification of a node into another node. But we need to explicitly allow this transformation by defining an `AllowedTransformation` implicit. The definition of the `transform` combinator is separated in two parts. First we define an alias `transformWithResult`:

```
def transformWithResult[I <: T, O <: T, A]
       (f: PartialFunction[I, (O, A)])
       (implicit x: AllowedTransformation[I, O]): Matcher[A]
```

where

- `I` is the type of the *input* node.
- `O` is the type of the *output* node.
- `A` is the type of the result carried out with the transformation. As we will soon discuss it will be rendered optional.
- an **implicit** `AllowedTransformation[I, O]` which allows the transformation of nodes of type `I` to nodes of type `O` is needed.

The second part of the definition consists of a whitebox macro having the following definition:

```
def transform(f: PartialFunction[T, Any]): Matcher[Any]
```

The `transform` macro will transform its parameter and then call `transformWithResult` with the correct type parameters. It has three responsibilities:

1. Helping the type-checker. Indeed, in Scala, the type-checker is not able to infer the type of `I` in `transformWithResult` because it is the *parameter type* of a function. Since `f` is a partial function and we have an upper bound `T`, we can infer the input type of the partial function by looking at left-hand side (lhs) of the **case** `lhs => rhs` statements.
2. Make the result optional. We inspect the AST of `f` and, if the result type `O` of `f` is not a tuple then we change it to `(O, Unit)` to make it compliant with `transformWithResult`.
3. Separate the cases. Let us consider a small type hierarchy:

```
trait A
trait B extends A
trait C extends A
trait B1 extends B
trait B2 extends B
implicit object x extends AllowedTransformation[B1, C]
implicit object y extends AllowedTransformation[B2, C]
```

And let us consider the following transformation:

```
transform{
  case b1: B1 => new C{}
  case b2: B2 => new C{}
}
```

In this example, such a transformation would not be possible without separating the cases, because the type inferred by our macro (the lower upper bound of the lhs of the cases statements) would be `I = B` and `O = C`, and no such AllowedTransformation[B, C] exists in that example. To solve this problem we need to consider each case separately, as a different transformation. Hence the above transformation example is transformed by our macro into:

```
transformWithResult[B1, C, Unit]{
  case b1: B1 => (new C{}, Void)
} orElse transformWithResult[B2, C, Unit]{
  case b2: B2 => (new C{}, Void)
}
```

Where `orElse` is a composition operator that we will define later.

3.2.2 *Composition combinators*

Here we describe some of the *composition operators* which we use to *glue* combinators together. Those are defined as methods on the `Matcher[+A]` trait. A more exhaustive list with more information and examples can be found on the wiki of the project [15].

andThen.

The `andThen` combinator allows to execute a second combinator after and only if the first one has succeeded.

```
def andThen[B](m: => Matcher[B]): Matcher[B]
```

It works on the data structure transformed by the first combinator but discards the result generated.

tupled.

In the expression `a tupled b`, `a` is applied first and then `b` is applied, even if `a` failed. Their results are kept separated in a tuple.

```
def tupled[C >: A : Monoid, B : Monoid](m: => Matcher[B]): Matcher[(C, B)]
```

aggregate (alias: +).

In the expression `a aggregate b`, the combinator `a` is aggregated with `b`. First `a` is applied and then `b` is applied on the data structure transformed by `a` or on the original data structure if `a` failed. At the same time, the result of `a` is *appended* to the result of `b`.

```
def aggregate[B >: A : Monoid](m: => Matcher[B]): Matcher[B]
```

orElse (alias: |).

In the expression `a orElse b`, this combinator first apply `a` and then applies `b` if the application of `a` has failed.

```
def orElse[B >: A : Monoid](m: => Matcher[B]): Matcher[B]
```

map.

`map` simply *map* the result of the combinator. It does not perform any action on the data structure.

```
def map[B](f: A => B): Matcher[B]
```

feed.

In the expression `a feed x => b`, the result of `a` is *given* to `b` so that `b` can use the `x` the result generated by `a`.

```
def feed[B : Monoid](m: => A => Matcher[B]): Matcher[B]
```

3.2.3 Traversal combinators

Even though we already talked about the different traversal strategies in TQL, we describe here the implementation of such strategies in TQL because they demonstrate the use of some combinators. We implement them as *meta-combinators* i.e. combinators that take other combinators as arguments.

children.

```
def children[A : Monoid](f: => Matcher[A]) = Matcher[A]{
  t => this.traverse(t, f)
}
```

This combinator applies the `Matcher f` to all the immediate children of the current node `t` by calling the `traverse` method defined in section 3.1.2.

topDownBreak.

```
def topDownBreak[A : Monoid](m: Matcher[A]) =
  m | children(topDownBreak[A](m))
```

We implement this top down strategy as a either succeeding in applying `m` on the current node or on its children. This implies that we do not continue to match `m` in the children of the current node if it has succeeded.

topDown.

```
def topDown[A : Monoid](m: Matcher[A]): Matcher[A] =
  m + children(topDown[A](m))
```

This is almost the same as the `topDownBreak` combinator except that we continue to traverse the children of the node whether applying `m` to the current node is successful or not.

bottomUpBreak.

```
def bottomUpBreak[A : Monoid](m: Matcher[A]): Matcher[A] =
  oneOfchildren(bottomUpBreak[A](m)) | m
```

Like `topDownBreak` but in the reverse order so as to check the `children` of the node first. We do not use the traversal strategy `children` here because it would fail due to left recursion. Instead, we use `oneOfchildren` which is very much like `children` but succeeds only if one of the children of the node succeeds.

bottomUp.

```
def bottomUp[A : Monoid](m: => Matcher[A]): Matcher[A] =
  children(bottomUp[A](m)) + m
```

Like `topDown` but in the reverse order so as to traverse the `children` of the node first.

3.2.4 Recursion

Data structure traversal and transformation is inherently recursive. Thus a traversal API must provide the different mechanisms required to recurse through the data structure. Here we present those mechanisms and how they are implemented in TQL.

```
val onlyThn: Matcher[Int] = topDownBreak(
  flatMap(_ match {
   case If(cond, thn, els) => for {
    (thn2: Term, res) <- onlyThn(thn)
   } yield (If(cond, thn2, els), 1 + res)
   case _ => None
  })
)
```

Figure 5: A simple traverse which counts the number of If in an imaginary language

Via traversal strategy.

We have already presented the different traversal strategies available in TQL. They are a form of recursion (actually, the simplest form of recursion) as they allow to apply a `Matcher` recursively on a whole data structure. The cases where we would like to stop the recursion upon coming across a particular node are handled by the `topDownBreak` or `bottomUpBreak` combinators described above.

Vanilla recursion.

Sometimes the recursion pattern we want to express is not easy or possible to implement with a traversal strategy only. This is of course the case for the implementation of those traversal strategies. As presented in the previous subsection `topDown`, `topDownBreak` etc. all use vanilla recursion.

FlatMap.

One pattern which is difficult to express with the two previous recursion mechanisms in TQL is the *selection traversal* or the traversal of a selection of children. For example, in an imaginary language, upon the encounter of a `If(cond, thn, els)` node, we may wish to traverse only the `thn` sub-tree. To express this, we can use `flatMap` which, together with the `topDownBreak` (or `bottomUpBreak`) strategy allows us to express this pattern. An example is shown in Figure 5.

3.3 Performance tuning

3.3.1 Fusing, a deeper composition

Basic Fusing.

Let us consider the two following simple traversers:

```
//collect every Integer literals greater than 5.
val m1 = topDown{collect{case Lit.Int(x) if x > 5 => x}}
//collect every Integer literals smaller than 5.
val m2 = topDown{collect{case Lit.Int(x) if x < 5=> x}}
```

We may wish to compute the union of those two traversals and have all the Integer literals greater than 5 and smaller than 5 in a single `List`:

```
val m3 =  m1 aggregate m2
```

What happens here is that in a first phase all elements greater than 5 are collected, then all elements smaller than 5 are collected and we finally put them together (of course it would be trivial to write a combinator which collect all

```
val collectVals = {
  import Traverser._
  topDown(collect{case Lit.String(v) => v})
}
val collectValsFusing = {
  import FusionTraverser._
  topDown(collect{case Lit.String(v) => v})
}
comparePerformance(
  collectVals + collectVals,
  collectValsFusing + collectValsFusing
)
```

Figure 6: Comparing composition and fusing

elements except 5). Thus `m3` traverses the data structure two times. We can do better, we can fuse `m1` and `m2` together in order to traverse the data structure only one time. Indeed we can express the composition of `m1` and `m2` in the following way:

```
val m4 = topDown{
  collect{case Lit.Int(x) if x > 5 => x}
  aggregate
  collect{case Lit.Int(x) if x < 5 => x}
}
```

We can, fortunately, automate this process, by modifying the `topDown` combinator and further modify its `aggregate` method so as to perform fusing with other modified `topDown` combinators. As a result we can automatically transform expressions of type `topDown(a) aggregate down(b)` into `topDown(a aggregate b)`. Obviously the ordering of the results will not be the same anymore. Indeed the result of `m3` will be a `List` of Integers greater than 5 followed by a Integers smaller than 5 while the result of `m4` will be an interleaved `List` of Integers greater than 5 and Integer smaller than 5. In the same way, we implement fusing for the `tupled` *composition operator* which gives us almost the same transformation rule: `topDown(a) tupled topDown(b)` into `topDown(a tupled b)`. In this case however the ordering does not change.

We benchmark fusing on a simple traverser consisting in collecting all String literals in a tree (see Figure 6. We aggregate this same traverser multiple times and compare the performance between simple composition and fusing. Our preliminary results show the following performance improvements:

- The fusing of two aggregated traversals improves performance by about 250%.
- The fusing of four aggregated traversals improves performance by about 330%.
- The fusing of eight aggregated traversals improves performance by about 660%.

More advanced Fusing.

In this section we present some of the rules used to fuse some other kind of combinators:

```
topDown(a).map(f1).aggregate(topDown(b).map(f2))
```

becomes

```
topDown(a.aggregate(b)).map.{case (x, y) => f1(x) + f2(x)}
```

```
topDown(a).map(f1).tupled(topDown(b).map(f2))
```

becomes

```
topDown(a.tupled(b)).map{case (x, y) => (f1(x), f2(x))}
```

```
(topDown(a).feed{x => topDown(b)}).aggregate(topDown(c))
```

becomes

```
topDown(a).feed{x => topDown(b).aggregate(topDown(c))}
```

```
topDown(a).feed{x => topDown(b)}.aggregate(topDown(c).feed {y => topDown(d)})
```

becomes

```
{topDown(a).aggregate(topDown(c).feed{case (x, y) => topDown(b).aggregate(topDown(d))})
```

```
val collectStrings = topDown(optimize(collect{case Lit.String(v) => v}))
val collectInts = topDown(optimize(collect{case Lit.Int(v) => v.toString}))
val collectVals = topDown(optimize(collect{case v: Decl.Val => "val"}))
val collects = collectStrings + collectInts + collectVals
val reptraversers = collects + collects + collects
```

Figure 7: `reptraversers` is the fused traverser we benchmark

3.3.2 The optimize combinator

Scala-abide [1] proposes an interesting optimization which comes naturally with fusing. Let us consider the following traversers

```
topDown(collect{case .. : A => ... : U})
topDown(collect{case .. : B => ... : U})
topDown(collect{case .. : A | B => ... : U})
```

When we fuse them together we obtain the following traverser

```
topDown(
  collect{case .. : A => ... : U} +
  collect{case .. : B => ... : U} +
  collect{case .. : A | B => ... : U}
)
```

The problem here is that all three traversers will be tried on each node of the traversed data structure. We can do better by checking the node beforehand and only applying the traversers which have a chance to match this node. In TQL we do that by checking the type of the patterns in the `case` part of the partial functions. Hence our traverser above becomes (in pseudo-code):

```
topDown({ node match
  case _: A =>
    collect{case .. : A => ... : U} +
    collect{case .. : A | B => ... : U} apply(node)
  case _: B =>
    collect{case .. : B => ... : U} +
    collect{case .. : A | B => ... : U} apply(node)
})
```

We benchmark the use of this `optimize` combinator with three very simple traversers shown in Figure 7. The results we obtain show that we do not gain much, the use of the `optimize` combinator makes the traversal only 18% faster.

Would parallelization make sense?.

When talking about speeding up performances it makes sense to ask ourselves the following question: can we parallelize it? In the context of data structure traversal there are two places where we can think of parallelization:

1 Parallelization inside a traversal, for instance when traversing a `If(cond, thn, els)` node with three children,

would it make sense to traverse the children in parallel? The answer here is: probably not. Indeed the time required to set up parallelization would probably outweigh the small gain that we get from this parallelization.

2 Parallelization instead of fusing. There is probably more to gain in this area. Even though we cannot parallelize rewriting rules in the general case (because rewriting one rule may influence the next one) there is probably a lot to gain when only traversing and collecting information on the data structure. We did not implement this during the semester project due to time constraints.

4. TQL IN PRACTICE: TRAVERSING SCALA META TREES

In order to put TQL into practice and to use it with scala.meta we need to write a traverser for scala.meta trees, this is the only requirement to use TQL. This proved to be complicated and we observe that the way we write this traverser has a huge influence on the performance of TQL. In this section we present the different approaches we tried to build a fast traverser for scala.meta.

4.1 Traversing scala.meta trees

We use macros to generate the scala.meta Traverser. As there are a lot of different nodes to traverse it would be inconvenient to write it by hand. Since the traverse pattern is always the same, as we will see below, we can automate the task with ease. Another reason why macros make a lot of sense in this case is that this technique is resilient to changes in the structure of the AST, thus requiring no effort in modifying the traverser when existing nodes are changed or new nodes are introduced. As we show in Figure 8, the traverser we generate is a big pattern match on all leaves of scala.meta trees. The traversal of a `Leaf` is done in three steps:

1 Traverse each child of the leaf, transform it and collect a result by applying the `Matcher f` on it.

2 If no children has been modified by `f` we can save the cost of reconstructing the leaf. Otherwise we rebuild it.

3 Return the modified `tree` and the appended results collected on the children.

In order to analyze the performances of the TQL traverser we also build a *regular Traverser* and a *regular Transformer*. The former only traverses the nodes and the later only transform the nodes. Both do not collect any result. They are similar to the `Traverser` and `Transformer` in `scala.reflect` that we present in the introduction. We implement the *regular Traverser* and the *regular Transformer* wit a *naive approach* i.e we pattern match on each leaf of the scala.meta structure. We show an excerpt of the code in Figure 9

4.2 Unsatisfying performance

We benchmark our traversers with ScalaMeter [12]. We write a traverser which collect all String literals in a Scala code. We implement this traverser with TQL and the *regular Traverser* on scala.meta trees and with the `Traverser` from scala.reflect on *regular* Scala ASTs. Our benchmark shows that traversing on scala.meta trees is between 8 and 16 times slower than traversing Scala ASTs with the *regular Traverser*. Furthermore we observe that traversing with

```
def traverse[A : Monoid](tree: Tree, f: Matcher[A]): MatchResult[A] =
  tree match {
    case Term.If(cond, thenp, elsep) =>
          for {
        (a1: Term, a2) <- f(cond)
        (b1: Term, b2) <- f(thenp)
        (c1: Term, c2) <- f(elsep)
          } yield (
                if ((a1 eq cond) &&
                   (b1 eq thenp) &&
                   (c1 eq elsep))
                        tree
                else
          Term.If(a1, b1, c1),
          a2 + b2 + c2)
    case Term.While(expr, body) => ..
                ..
  }
```

Figure 8: The TQL scala.meta tree traverser: it traverses each child of a leaf while collecting the results. If the tree is not modified we return the original, otherwise we reconstruct it.

```
class Traverser {
  def traverse(tree: Tree) = tree match {
    case Term.If(cond, thenp, elsep) =>
      traverse(cond)
      traverse(thenp)
      traverse(elsep)
    case Term.While(expr, body) => ..
      ..
  }
}
```

Figure 9: The regular scala.meta Traverser

```
class Traverser {
  def traverse(tree: Tree) = tree match {
    case t: Term => t match {
      case Term.If(cond, thenp, elsep) => ..
      case Term.While(expr, body) => ..
        ..
    }
    case t: Type => t match {
      case Type.Apply(tpe, args) => ..
         ..
    }
       ..
  }
}
```

Figure 10: The hierarchical scala.meta Traverser

```
class Traverser {
  val table = {
    val array = new Array[(Tree, Tree => Unit) => Unit]
        ..
        array(Term.If.$tag) = (tree: Tree, f: Tree => Unit) =>
          tree match {
            case Term.If(cond, thenp, elsep) => ..
          }
          ..
        array
  }
  def traverse(tree: Tree): Unit =
    table(tree.tag)(tree, traverse _)
}
```

Figure 11: Implementation of a regular scala.meta Traverser with `$tag`

TQL is about 10 times slower than traversing with the *regular Traverser*. We notice that transforming scala.meta trees with TQL is only about 1.5 times slower than transforming them with the *regular Transformer*. This is presumably due to the sole overhead of returning a result together with the transformed tree. Note that the number of nodes in a scala.meta tree and a Scala AST varies a lot for the same file. There are sometimes two times less nodes in the scala.meta tree than in the Scala AST and sometimes two times more. This makes benchmarking especially tricky even though the traversal time is always slower with scala.meta trees.

4.3 Reorganizing the pattern matching order

One idea that comes from the Scala `Traverser` is that we should place the cases that are the most visited at the top of the pattern match. Indeed this strategy works well as we observe a 2-3X speedup when traversing of scala.meta trees.

4.4 Hierarchical pattern matching does not work

One idea to speed up the traverser is to hierarchically pattern match the trees. For instance a `Term.If` extends `Term`, thus it should make sense to first check if the `tree` is a `Term` and then pattern match the tree among the different leaf of `Term`. Figure 10 illustrate that approach. The problem is that it implies a lot more pattern matching (at least two levels). Furthermore some leaves are rather deep in the AST structure, which makes this approach unpractical. Indeed it is slower than the naive approach.

4.5 Optimizing with $tag

Each scala.meta tree is associated with a `$tag`. A unique Integer identifier which represent the type of the tree.

Array of functions.

Instead of executing a long pattern match we can register a function traversing a particular leaf in an array at the position of its `tag`. The traverser now only needs to retrieve the `tag` of the tree we want to traverse and execute the function at the `tag` position in the array. This is illustrated in Figure 11. The performances improve only very slightly compared to the naive approach (15% faster). This can be explained by the fact that there is one extra indirection (accessing the `table`), still one pattern matching and one virtual call (to `tree.tag`). This would make the traverse method slower for the most visited nodes.

```
class Traverser {
  def traverse(tree: Tree): Unit =
  (tree.$tag : @scala.annotation.switch) match {
    ..
   case 78 => tree match {
     case Term.If(cond, thn, els) => ..
   }
   ..
  }
}
```

Figure 12: Implementation of a regular scala.meta Traverser with the `@switch annotation`

@Switch annotation.

A similar technique to the *naive approach* is to make the traverser pattern match on the `$tag` of the leaves instead of on the leaves themselves. Since `$tag`s are Integer we can supposedly improve the performances with the `@switch annotation` as we present in figure 12. Unfortunately we obtain about the same performance as with the technique we describe in the last paragraph. This is also probably due to the virtual call to `$tag` and the fact that the pattern match still contains a lot of cases.

4.6 Summary of performances

We apply these optimizations to the TQL traverser but unfortunately only get very slight performance improvement (about 11% faster when changing the ordering of the cases in the pattern match). The probable reason is that there is already too much overhead in the TQL traverser to make the optimizations significant. We further discuss this problem in section 7.5.

We also have to take into account that with the TQL traverser, every traversal is implemented as a transformation. For instance when collecting information on the data structure with the *regular traverser* there is no overhead associated with the reconstruction of the AST. This is not the case with the TQL traverser, we further discuss this case in section 7.6. For example writing a traverser with a *regular Transformer* is about 6 times slower than writing it with a *regular Traverser*.

In short currently traversing with TQL is about 10 times slower than traversing with the traverser for scala.meta (with reordering) and about 25 times slower than traversing with

the *regular Traverser* from the Scala reflection API. Moreover transforming with TQL is 1.5 times slower than transforming with the *regular Transformer* for scala.meta.

5. FOR A MORE INTUITIVE API

5.1 Collection-like API

Combinators may not be the most intuitive way in which programmers would like to manipulate a data structure. In this section we present another, simpler API, built on top of TQL combinators which allows to manipulate a data structure like it were a Scala collection.

For instance, the code below shows how to collect all Integer literals bigger than 10 on a scala.meta tree.

```
val tree: scala.meta.Tree = ...
val x: List[Int] = tree.collect{case Lit.Int(a) if a > 10 => a}
```

Here we notice that calling `collect` directly return a result. For the record here is the same traverser implemented with combinators:

```
val collectInts: Matcher[List[Int]] = topDown(collect{case Lit.Int(a) if a > 10 => a})
val x: List[Int] = collectInts(tree) match {
  case Some((_, result)) => result
  case _ => Nil
}
/*
Which is equivalent to
val x: List[Int] = collectInts(tree).result
*/
```

5.1.1 Supported operations

The operations we can express with this API are more restricted than with the combinator approach. We can still choose with which strategy to traverse the data structure (`topDown`, `topDownBreak`,..) but we implicitly use the top bottom strategy `topDown` when the user does not explicitly select any. Therefore

```
tree.collect{case Lit.Int(a) if a > 10 => a}
```

is equivalent to

```
tree.topDown.collect{case Lit.Int(a) if a > 10 => a}
```

. On top of the traversal strategies the three following operations are supported:

collect.

As already presented, it allows to directly collect information on the data structure. It inherits the capabilities of the combinators, as it is still possible to collect in any type of collection (the default still begin `List`).

transform.

It allows to transform the data structure directly. Here is an example:

```
val tree: scala.meta.Tree = ...
val x: scala.meta.Tree = tree.transform{case x: Lit.Int => Lit.Int(1)}
```

Like `collect` `transform` also inherits its capabilities from its combinator counterpart. It is indeed not possible to execute ill-formed transformation i.e from `Term` to `Type`. It is also still possible to return a result at the same time as transforming the data structure. Moreover the result type of the `transform` operation changes depending on whether the a result is returned or not. If no result is returned, `transform` only return the transformed data structure. But in the other case it returns a tuple consisting of the transformed data structure and the result, as presented in the example below.

```
val x: (scala.meta.Tree, Set[Int]) = tree.transform{
  case Lit.Int(a) => Lit.Int(a * 2) andCollect[Set] a
}
```

focus.

This operation allows to stop the traversal at a specific position in the data structure. For example the code below collects every Integer literal in the children of every `Term.If` node.

```
val x: List[Int] = tree.topDownBreak
  .focus{case Term.If(_,_,_) => true}
  .topDown
  .collect{case Lit.Int(a) => a}
```

5.1.2 Architecture

The architecture of the collection-like UI is quite simple, as it is only a wrapper around TQL combinators. Here we describe how to implement a very simple traversal of type `t.strategy.operation` (as previously stated *strategy* is optional here). All we have to do is to transform it into `strategy(operation)(t)`.

We implement this transformation using two componants:

- An implicit class `Evaluator(t: T)` which allows us to perform the operations described above on the data structure `t`.
- A class `EvaluatorMeta(t: T, strategy: DelayedMeta)` which describe which strategy will be used to traverse `t`.

Hence the transformation is done in the following way. The `t.strategy` call create a new `EvaluatorMeta(t, strategy)`. Then the `operation` call simply apply the operation on `t` using the selected `strategy`.

The implementation of the `focus` operation is a bit more involved and we do not present it here for brevity's sake.

5.1.3 Loosing my combination

An obvious disadvantage of this collection-like UI is that we lose the power of combinators. The trade-off that we are making here is that we gain a more intuitive and more straightforward API at the expense of expressiveness and power. The rationale behind the design of this user interface is that it should be easy to write simple traversals which are only used once. Indeed it is now possible to directly query the data structure for simple information. There is no need to write a traverser and then apply it.

5.1.4 Functional dependencies: implicits or whitebox macros?

In Scala there are two ways of implementing *functional dependencies* (put simply it is the fact that some type parameter depends on others) with whitebox macros [8] or with implicits [22]. For instance, in TQL we choose to implement the collect combinator with implicits instead of macros. Whitebox macros present the following advantages over implicits: 1) they have a shorter definition as we do not need to carry type information in the arguments, replacing them with `Any` (or any super type). 2) They are supposedly easier to implement (without talking about the need to create a separate project for the macro implementation). On the other hand, implicits present the following advantages over macros: 1)

It is more likely to deduce the correct signature by looking at the definition of the function, as we do not need to delve into the implementation. 2) It is more composable as we only need to pass an implicit scope between functions we want to compose. Composing with a macro is more difficult as it, in general, requires that the functions it composes with are macros themselves.

Macros do not integrate well with some development tools, but they involve less compilation overhead than implicit search, as it is quite costly.

5.2 Syntax enhancer, trying to make it like XPath

As another experimental user interface, we try to make the writing of combinators more pleasant. For that purpose we propose a *syntax enhancer*, a collection of expansion methods on the data structure we want to traverse. They present two simple changes to the UI.

First they allow us to write the traversal strategy after the `Matcher` and thus to get rid of some annoying parenthesizes. We can therefore write

```
collect{..}.topDown
```

instead of

```
topDown(collect{..})
```

.

Secondly we try to implement a small XPath-like query DSL on top of TQL combinators by using operator overloading. We then define an expansion operator, shown in Figure 13 which should let us express

```
topDownBreak(focus{..} andThen topDown(collect{..}))(tree)
```

by

```
tree \  focus{..} \\ collect{..}
```

We can immediatly observe that the above code will not work in Scala because of operator associativity. As is, the above code is interpreted as

```
(tree \  focus{..}) \\ collect{..}
```

while we would like something like that

```
tree \  (focus{..} \\ collect{..})
```

Indeed the Scala reference reads:

> The associativity of an operator is determined by the operator's last character. Operators ending in a colon ':' are right-associative. All other operators are left-associative.

We can therefore replace the `\` operator by `\:` to make it right-associative. But we have to take into account that `a \: b` is desugared to `b.\:(a)` and modify the execution path accordingly (see Figure 14).

Even though the XPath-like UI is, for now, very simple we can write some queries in a XPath fashion:

```
tree \:  focus{..} \\: collect{..}
```

6. EXAMPLES

In this section we will translate some examples of Traverser and Transformer written with the current scala.reflect API and highlight the differences.

6.1 Simple Traverser

```
implicit class XPathEnhancer[A](a: Matcher[A]){
  def \\[B : Monoid] (b: Matcher[B]) = a andThen topDown(b)
  def \[B : Monoid] (b: Matcher[B]) = a andThen topDownBreak(b)
  ..
}
```

Figure 13: Adding an expansion operator to TQL Matchers in order to provide an XPath-like user interface

```
implicit class TEnhancer(t: T){
  def \\:[B] (b: Matcher[B]) = b andThen topDownAlias(a)
  def \:[B] (b: Matcher[B]) = b andThen topDownBreakAlias(a)
}
```

Figure 14: Modifying the \operator

This first example shows how to get the name of all methods in a tree. With scala.reflect.

```
def MethodNameCollector(tree: Tree): List[String] = {
  private[this] val funcnames = ListBuffer[String]()
  new Traverser {
    override def traverse(tree: Tree) = tree match {
      case DefDef(_, name, _, _, _, rhs)  =>
        funcnames += name.toString
        traverse(rhs)
      case _ =>
        super.traverse(tree)
    }
  }.traverse(tree)
  funcNames.toList
}
```

With TQL combinators.

```
val MethodNameCollector = topDown(collect{
  case DefDef(_, name, _, _, _, _) =>
    name.toString
  })
```

Here the use of the `topDown` combinator automatically traverse the children of the `DefDef` node. One subtle difference is that in the TQL version all sub-tree will be visited while in the scala.reflect version only the `rhs` sub-tree will be further visited.

With the collection like UI.

```
val tree: scala.meta.Tree: = ..
val methodNames = tree.collect{
  case DefDef(_, name, _, _, _, _) =>
    name.toString
}
```

6.2 Simple Transformer

This transformer, taken from the Slick [14] source code, is used to remove Type annotations. With scala.reflect.

```
object removeTypeAnnotations extends Transformer {
  def apply(tree:Tree ) = transform(tree)
  override def transform(tree: Tree): Tree = {
    super.transform {
      tree match {
        case TypeApply( tree, _ ) => tree
        case Typed( tree, _ ) => tree
        case tree => tree
      }
    }
```

```
  }
}
```

With TQL.

```
val removeTypeAnnotations = transform{
  case TypeApply(tree, _ ) => tree
  case Typed(tree, _ ) => tree
}.topDown
```

Here the `topDown` combinator will automatically recurse on the newly transformed tree.

6.3 Get all definitions

The following traverser, taken from Yin-Yang [16], retrieve all definitions in a given tree. With scala.reflect.

```
class LocalDefCollector extends Traverser {
  private[this] val definedValues, definedMethods = ListBuffer[Symbol]()

  override def traverse(tree: Tree) = tree match {
    case vd @ ValDef(mods, name, tpt, rhs) =>
      definedValues += vd.symbol
      traverse(rhs)
    case dd @ DefDef(mods, name, tparams, vparamss, tpt, rhs) =>
      definedMethods += dd.symbol
      vparamss.flatten.foreach(traverse)
      traverse(rhs)
    case bind @ Bind(name, body) =>
      definedValues += bind.symbol
      traverse(body)
    case _ =>
      super.traverse(tree)
  }
  def definedSymbols(tree: Tree): List[Symbol] = {
    definedValues.clear()
    definedMethods.clear()
    traverse(tree)
    (definedValues ++ definedMethods).toList
  }
}
```

With TQL.

```
val localDefCollector = topDown(
  collect{
    case vd: ValDef => vd.symbol
    case bind: Bind => bind.symbol
  } tupled
  collect{
    case dd: DefDef => dd.symbol
  }
) map {case (vals, meths) => vals ++ meths)}
/*here we assume that the order of appearance in the list
is important otherwise we could simply have written:
topDown(collect{
  case vd: ValDef => vd.symbol
  case bind: Bind => bind.symbol
  case dd: DefDef => dd.symbol
})
*/
```

6.4 Replace Var by Val

The following example comes from Obey [9]. It replaces each **var** only used once by a **val** and return a warning at the same time.

With TQL.

```
val varInsteadOfVal = collect[Set] {
    case Term.Assign(b: Term.Name, _) => b
  }.topDown feed { assign =>
    transform {
      case t @ Defn.Var(a, (b: Term.Name) :: Nil, c, Some(d))
        if (!assign.contains(b)) =>
        Defn.Val(a, b :: Nil, c, d) andCollect
          (b + "should be a Val")
    }.topDown
  }
```

With scala.reflect we need two build two traversers.

```
def varInsteadOfVal(tree: Tree): (Tree, List[String]) = {
  val assign = new ListBuffer[Term.Name]()
  val messages = new ListBuffer[String]()
  new Traverser{
    override def traverse(tree: Tree) = tree match {
      case Term.Assign(b: Term.Name, rhs) =>
        assign += b
        traverse(rhs)
      case _ => super.traverse(tree)
    }
  }.transform(tree)
  val newTree = new Transformer {
    override def transform(tree: Tree) = tree match {
      case t @ Defn.Var(a, (b: Term.Name) :: Nil, c, Some(d))
        if (!assign.contains(b)) =>
        messages += b + "should be a Val"
        Defn.Val(a, b :: Nil, c, d)
          case _ => super.transform(tree)
        }
  }.transform(tree)
  (newTree, messages.toList)
}
```

6.5 Getting the name of the local variables of each method

Our goal is to write a traverser which returns a `Map` of method names to a list of their local variables.

With a *regular Traverser* it is actually quite easy to write such a traverser. However we notice that:

1. We manipulate a state (of course it is possible to do without it, with two mutually recursive traversers, but the implementation would be longer).
2. We need to declare when to continue the traversal in the children.
3. Taking into account **var**s would add at least 4 more lines of code.

```
def getValsInMethods(tree: Tree) = {
  val funcsWithVals = new HashMap[Term.Name, List[Term.Name]]()
  var currentFunc: Term.Name = null
  new Traverser {
    override def traverse(tree: Tree): Unit = tree match {
      case f: Defn.Def =>
        val oldFunc = currentFunc
        currentFunc = f.name
        super.traverse(tree)
        currentFunc = oldFunc
      case Defn.Val(_, (b: Term.Name):: Nil,_, rhs)
        if currentFunc != null =>
        val content = funcsWithVals.getOrElse(currentFunc,Nil)
        funcsWithVals += (currentFunc -> (b.name::content))
        super.traverse(tree)
      case _ => super.traverse(tree)
    }
  }.traverse(tree)
  funcsWithVals.toMap
}
```

In order to implement this traverser with TQL let us first introduce another traversal strategy:

```
def until[A : Monoid, B](m1: Matcher[A], m2: Matcher[B]): Matcher[A] =
  m2 orThen (m1 + children(until(m1, m2)))
```

Where `orThen` is like the `orElse` combinator except that the result `m2` returns is completely discarded and replaced by the *zero value* of `Monoid[A]`. `until` allows us to traverse a data struc-

ture in a top-down dfs way and applies `m1` until `m2` matches.

We can now write our traverser:

```
val getValsInMethods: Matcher[Map[Term.Name, List[Term.Name]]] =
  (focus{case _: Defn.Def => true} feed { defn =>
    (until(
      collect{case Defn.Val(_, (b: Term.Name):: Nil,_, _) => b.name},
      focus{case _: Defn.Def => true}
      ).map(x => Map(defn.name -> x)) + getValsInMethods).children
   }).topDownBreak
```

This requires some explanations:

- We first use the topDownBreak strategy to stop the recursion when we encounter a method.
- Once we are *inside* a method (`defn`) we need to collect all the **`val`**s in its children.
- Since we want only the local **`val`**s of the methods we need to be careful to not continue the recursion inside an inner method of `defn`. For that purpose we use the `until` traversal strategy which stops when a method is encountered.
- Finally we insert the variable names into the map and continue the recursion on the inner methods

Note that this traverser might be doing too much work as it traverses methods multiple times (once to collect the variables and another time to get to the next method to traverse).

It is possible to make it run in one only pass if we maintains two results at the same time: the list of variables in the current method and a `Map` of method names to a list of variables names. We can use the `tupledUntil` traversal strategy which behaves like the `until` strategy except that it keeps the results of both combinators and aggregates them in a tuple.

Using this knowledge, let us now define a new combinator `group` which groups in a `Map` all the results of one combinator `value` which match *between* nodes matching a `key` combinator:

```
def group[K, V: Monoid](key: Matcher[K], values: Matcher[V]) = {
    def inner: Matcher[(V, Map[K, V])] = tupledUntil(
      values,
      key feed { k =>
        inner.children.map{case (v, m) => Map(k -> v) ++ m}
      }
    )
  (key andThen inner).topDownBreak.map(_._2)
}
```

The `getValsInMethods` is then easy to write:

```
val getValsInMethods = group(
  visit{case f: Defn.Def => f.name.toString},
  collect{case Defn.Val(_, (b: Term.Name):: Nil,_, _) => b.name}
)
```

One advantage of using TQL combinators is that if we can easily change the code to also retrieve the locals **`var`**s of the methods. The modification is straightforward:

```
val getValsAndVarsInMethods = group(
  visit{case f: Defn.Def => f.name.toString},
  collect{
    case Defn.Val(_, (b: Term.Name):: Nil,_, _) => b.name
    case Defn.Var(_, (b: Term.Name):: Nil,_, _) => b.name
  }
)
```

Furthermore we can re-use the `group` combinator to build other similar traversers, for instance here is a combinator to retrieve all methods in every class:

```
val getMethodsPerClass = group(
  visit{case c: Defn.Class => c.name},
  collect{case f: Defn.Def => f.name}
)
```

7. LIMITATIONS AND FUTURE WORK

7.1 Traversing scala.meta is slow

The first limitation is that, as we already discuss in section 4, currently, traversing scala.meta trees is slow compared to traversing Scala ASTs. Moreover TQL itself adds noticeable overhead and, in general performance is currently a problem. Therefore performance optimizations are needed will be practically useful.

7.2 Stateful traversal

Stateful traversal allows to modify a state during the manipulation of a data structure. An example of such a state is a counter. It could be used to, for instance, allow a specific transformation only a certain number of time. Here is the pseudo code of a traverser transforming a `Var` into `Val` at most 10 times:

```
topDown({(counter: Int) =>
  transform{
    case Decl.Var(x) if counter < 10 =>
      Decl.Val(x) withState (counter + 1)
  }
})
```

This is currently not supported in TQL even though we present a sketch of such experiments. A first attempt is to add a modified version of each combinator which has the ability to carry a state.

For example the above code would be written in that way:

```
topDown(transformState(0)(counter => {
  case Decl.Var(x) if counter < 10 =>
    Decl.Val(x) withState (counter + 1)
})
```

Although it works, it is not realistic to separate the set of combinators which carry state and those which do not, especially since one of the main argument of TQL is to reunite traversal and transformation under a same API. A similar solution would be to move the state manipulation up to the traversal combinator, as there are less of them.

```
topDownState(0)(counter => {
  transform(
    case Decl.Var(x) if counter < 10 =>
      Decl.Val(x)
  )  withState (counter + 1)
})
```

This solution, however, presents the same disadvantage as the last one about the combinators duplication. Moreover it moves the state manipulation out of the `transform` combinator. This imply that any state manipulation that depends on the `Decl.Var(x)` would not be easily feasible. A final attempt is to write a combinator which has for only responsibility the manipulation of the state. Here is the definition:

```
def stateful[A, B](init: => A)(f: ( => A) => Matcher[(B, A)]): Matcher[B]
```

The example from would ressemble to:

```
topDown(stateful(0){counter =>
  transform{
    case Decl.Var(x) if counter < 10 =>
      Decl.Val(x) withResult (counter + 1)
  }
})
```

While it seems to work in most cases there is a problem with the `collect` combinator. Indeed what we return in the `collect` combinator is instantly put in a collection, the state with it. To further explain this problem let us consider the following example, where instead of transforming `Var` in `Val`, we simply collect up to 10 `Var`s.

```
topDown(stateful(0){counter =>
  collect{
    case x: Decl.Var if counter < 10 =>
      (x, counter + 1)
  }
  /*it will not typcheck here because
  the type of collect{..} is now
  Matcher[List[(Decl.Var, Int)]] and not
  Matcher[(List[Decl.Var], Int)]*/
})
```

Of course we can easily implement the above code with `state` being an external `var` but it is non-functional and so we would like to do better in the future.

7.3 Symbol lookup

Combinators are suitable to traverse and transform data structures like AST. But what if we want to query information about the symbol table? For example we may wish to retrieve the symbol for the method `add` on the Int class. In this case a string query like `lookup("scala/Int/add")` looks much more reasonable than say:

```
filter{_ == "scala"}
  .children.filter{ == "Int"}
  .children.collect{_ == "add"}
```

7.4 UI limitations due to Scala

7.4.1 No default generic type parameter

We presented the implementation of the `collect` combinator in section 3 where we had to use implicit to mimic the existence of a default type parameter i.e transforming `collect` to `collect[List]`. The implementation would have been much nicer if we simply could have done something like:

```
def collect[C[_] = List] = new CollectInType {
  def apply[A](f: PartialFunction[T, A]): Matcher[C[A]]
}
```

7.4.2 No currying for type parameters

In the same example (i.e `collect` combinator) it would have been easier if one were allowed to write:

```
def collect[C[_] = List][A](f: PartialFunction[T, A]): Matcher[C[A]]
```

This is very different from:

```
def collect[C[_], A](f: PartialFunction[T, A]): Matcher[C[A]]
```

Because in this case when we want to specify the type of the collection to `collect` in we would also need to specify the type parameter `A`, which adds boilerplate.

7.5 Removing the composition overhead

Combinators present an important overhead. They imply composition and therefore a lot of object creation and method dispatch, which may be hard on the JVM. In our previous work we show how to remove composition overhead and transform functional code into imperative code in parser combinators [17]. Even though the depth of composition is somewhat smaller in TQL we argue that TQL could benefit from the same optimizations. Furthermore, results of TQL's combinators are Monoid instances. The addition of two results, through the *append* function of the Monoid typeclass, is probably also a source of overhead since we do not directly append those results but call an external Monoid instance to do it. Moreover we could probably gain in performance by implementing `List` concatenation via a `ListBuffer`.

7.6 Re-separating traversal and transformation

The combination of traversal and transformation is practical from an user interface point of view but not so much performance-wise. Naturally transformation always returns a result (the transformed data structure). This make TQL transformation performances comparable to vanilla transformation. As we point out, traversal is in itself faster than transformation, because no result is returned. Currently, in TQL, we represent a traversal by a transformation which returns the original data structure. This means that, in TQL, traversing a data structure is no faster than transforming it. In the future we could imagine TQL being able to detect that a composition of combinators will not modify the data structure, and therefore be able to select a *non-transforming* traverser. Using this strategy we would not lose performances due to TQL being too general.

7.7 MatchResult as a Monad

Currently `MatchResult[A]` is a type alias for `Option[(T, A)]`. Indeed each combinator returns an `Option` and a success is measure by a `Some(..)` while a failure is translated into a `None`. It would be interersting to let the programmer choose what should `MatchResult` be. We could restrict `MatchResult` to be a Monad as it is done in [4] as we would still need to have some way of controling the flow of the result. For example `MatchResult` could become a `Future` and therefore TQL could benefit from parallelization.

8. CONCLUSION

In this report we describe TQL, a combinator library for traversing and transforming data structures. We show that we can describe a complete traverser by combining several combinators in an expressive and natural way. Furthermore we present how we can combine traversal and transformation in a same API and how we can implement common optimzations such as fusing with combinators.

The practical application of TQL to scala.meta shows that our API shorten the code needed to write most traversals while avoiding state mutation. We also underline some of the limitations of TQL and how the library could be improved in the future. Unfortunately, we notice that performance of both scala.meta trees and TQL-based traversals is suboptimal, but we have identified the problematic areas and are hopeful that future work will resolve this problem.

Finally we should conclude by pointing out that TQL is actually in use in the Obey project: code health for scala.meta [9]. Obey allows an user to define rules to detect and automatically rewrite ill-formed Scala code. Its development, in parallel to the development of TQL, has helped tremendously in the design of our API.

Acknowledgements

We would like to thank Lars Hupel from TU München for helping with the early design of TQL and the people at the LAMP at EPFL for many discussions and insightful comments regarding traversers.

9. REFERENCES

[1] Abide: Lint tooling for scala. https://github.com/scala/scala-abide.

[2] Argonaut: Purely functional json in scala. http://argonaut.io/.

[3] HXT: A gentle introduction to the haskell xml toolbox. https://www.haskell.org/haskellwiki/HXT.

[4] JsZipper: Play2 json advanced (and monadic) manipulations. https://github.com/mandubian/play-json-zipper.

[5] Kiama: A scala library for language processing. https://code.google.com/p/kiama/wiki/Research.

[6] LLVM ASTMatcher: Matching the clang ast. http://clang.llvm.org/docs/LibASTMatchers.html.

[7] LLVM ASTMatcher: Matching the clang ast. http://clang.llvm.org/docs/LibASTMatchersTutorial.html.

[8] Macros: Blackbox vs whitebox. http://docs.scala-lang.org/overviews/macros/blackbox-whitebox.html.

[9] Obey: Code health for scala.meta. https://github.com/aghosn/Obey/.

[10] Rascal: Documentation. http://tutor.rascal-mpl.org/Rascal/Expressions/Visit/Visit.html.

[11] Rascal: Metaprogramming language. http://www.rascal-mpl.org/#_Transformations.

[12] ScalaMeter: Automate your performance testing today.. https://scalameter.github.io/.

[13] Scalaz: Functional programming for scala. http://typelevel.org/projects/scalaz/.

[14] Slick: Scala language integrated connection kit. https://github.com/slick/slick.

[15] TQL wiki: Documentation of tql. https://github.com/begeric/TQL-scalameta/wiki.

[16] Yin-Yang: Building deep dsls in a breeze! https://github.com/scala-yinyang/scala-yinyang.

[17] E. Béguet and M. Jonnalagedda. Accelerating parser combinators with macros. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA '14, pages 7–17, New York, NY, USA, 2014. ACM.

[18] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, June 2008.

[19] J. Gibbons and B. c. d. s. Oliveira. The essence of the iterator pattern. *J. Funct. Program.*, 19(3-4):377–402, July 2009.

[20] O. Kiselyov, S. L. P. Jones, and A. Sabry. Lazy v. yield: Incremental, linear pretty-printing. In R. Jhala and A. Igarashi, editors, *APLAS*, volume 7705 of *Lecture Notes in Computer Science*, pages 190–206. Springer, 2012.

[21] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain-specific language for source code analysis and manipulation. In *SCAM*, pages 168–177, 2009.

[22] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. *SIGPLAN Not.*, 45(10):341–360, Oct. 2010.

[23] E. Torreborre. Eric Torreborre's blog: The essence of the iterator pattern. http://etorreborre.blogspot.ch/2011/06/essence-of-iterator-pattern.html.