Proceedings of the
8th International Workshop on Graph-Based Tools
(GraBaTs 2014)

A Modular and Statically Typed Effectful Stack for Custom Graph Traversals

Norbert Tausch, Michael Philippsen

14 pages
A Modular and Statically Typed Effectful Stack for Custom Graph Traversals

Norbert Tausch¹, Michael Philippsen²

University of Erlangen-Nuremberg, Programming Systems Group, Germany
¹norbert.tausch@fau.de, ²michael.philippsen@fau.de

Abstract: Programmers often have to implement custom graph traversals by hand as either there are no suitable text-book algorithms for their tasks, or their problems are too complex for a pure querying language that lacks modularity or static typing. Our new Scala-based graph traversal language uses an effectful stack that adapts monads and type classes. Arbitrary graph effect computations and graph processing rules can be defined and composed in a modular and statically typed way. Custom graph traversals become expressible in a concise notation, run both in-memory and on graph databases, and also allow for parallelization. We evaluate the usability of our approach by detecting occurrences of an anti-pattern in a Java source code archive. Our approach outperforms the well-known Gremlin approach due to parallelization.

Keywords: graph, modular, monad, Scala, statically typed, traversal language

1 Introduction

When working with graph-based data, programmers often have to implement graph traversals and transformations by hand, as known graph libraries, query languages, or traversal frameworks seem inapplicable. A flexible and concise way to implement custom graph traversals is to use a general programming language and a workflow-like programming pattern, e.g., Scala (scala-lang.org) collection library, Java 8 streams, or TinkerPop Gremlin (tinkerpop.com) graph traversal language. All those approaches use fluent interfaces [Fow10] that are based on method chaining and data flow programming. We illustrate this with the property graph model in Fig. 1, an object-oriented representation of a directed, labeled and attributed pseudo-graph that is widely used in graph based systems [RN10], e.g., Neo4j (neo4j.org) and TinkerPop.

The property graph model is advantageous for custom graph traversals as it represents graphs as collections of vertices and edges. Fig. 2 holds code for a successors traversal that is based on Scala’s Set collection class. Starting from a set of vertices in (a), flatMap applies the one-to-many value compu-


tation to each vertex to compute a set of outgoing edges (b). With a one-to-one transformation, map then transforms this set into a set of vertices (c) that contains all the successors of all the initial vertices. With Scala’s syntax enhancements, we can shorten the code to `vertices.successors` which indicates the strength of our internal domain-specific language (DSL) for custom graph traversals.

Unfortunately, working with graphs often includes two types of tasks that break this short workflow pattern. Effect computations gather additional information during the traversal or influence its result. For example, `gather-the-current-traversal-path` or `do-not-visit-twice` are effects that are applied after each traversal step (e.g. `successors`). Processing rules influence the traversal order. The above example uses a breadth-first search (BFS) order, but often we need a depth-first search (DFS) order or have to run traversals in parallel.

Without a DSL, complex code is needed to freely compose such tasks for custom graph traversals. Our new graph traversal language that builds upon a so-called effectful stack provides a solution for this problem. Its main attributes are modularity and static typing. Modularity allows to compose arbitrary effect computations and processing rules that influence a graph traversal. It also makes effect computation results accessible in a structured way. Static typing allows to define effects in a way that lets the compiler decide statically whether an effect has to be applied to a traversal and which implementation fits best to the current data set. For example, a `gather-the-current-traversal-path` effect only gathers vertices along a traversal, but it can be skipped on edges for better performance. Our DSL is based on the object-functional programming language Scala. It leverages monads [Mog89] and type classes [WB89] but modifies them for improved modularity and static typing.

Sec. 2 briefly introduces monads and sketches our contributions. Then we address different interest groups: Sec. 3 is for effect programmers who define effects. Sec. 4 explains how the stack infrastructure makes effects available to query programmers, targeted in Sec. 5, who uses the traversal language. Section 6 covers related work. Sec. 7 holds an evaluation.

2 General Concept

2.1 Monads

In Sec. 1 we implemented custom graph traversals by means of (but not limited to) Scala collection classes. This workflow pattern is powerful as it offers monad [Mog89] methods like `map` and `flatMap` that bring three different computations and processing rules to graph traversals. (1) Value computations are distinct steps of a graph traversal, e.g., the `_.outEdges` and `_.target` functions in Fig. 2. Both require that `flatMap` and `map` apply methods of the property graph model to each vertex or edge in the current collection. (2) There can be additional effect computations that are implicitly applied along a traversal. For example in Fig. 2, the `Set` collection automatically removes duplicate elements after each step, such that in (c) vertex 2 only occurs once, although both edges a and c target it. With a `List` collection instead, the results of the `map` operation would be concatenated and hence, vertex 2 would occur twice. (3) Collection monads can have processing rules. In Fig. 2, the traversal automatically skips empty collections and runs in BFS order. Other traversal orders like DFS or even parallel execution order are possible by simply inserting a `.view` or `.par` in front of the `flatMap` operation.
2.2 Basic Idea

The basic idea is to use a collection monad’s workflow pattern, but to add missing task-related additional effect computations. We illustrate this by adding two typical effect computations to the `successors` example. First, a `gather-the-current-traversal-path` effect traces the path that led to the current vertex in a traversal, for instance in a cycle detection. Second, a `do-not-visit-twice` effect assures that vertices are only visited once. Vertex 3 is in the result set of Fig. 2 although it is among the initial vertices and hence visited twice.

As such additional effect computations are cumbersome to implement with extra code, our DSL brings them into the easy to understand workflow (see Fig. 3). First, the user lifts the needed effects into the collection. We provide appropriate `.path` and `.visit` syntax enhancements for that. As a result, each vertex is wrapped into a path object $P$ that holds and represents the path that led to the currently wrapped vertex. The collection of those path objects is wrapped into a visit object $V$ that holds a set of already visited vertices and represents the corresponding effect computation. Instead of a pure value collection (vertices, bold in Fig. 3), the user now gets a nested object comprising a collection, values, and effects ($P$ and $V$). We call such an object an effectful stack. In a second step, the user applies value computations with `flatMap` and `map` – or even better with predefined syntax enhancements like `.run` or `.successors`. The former explicitly performs all effect computations in the stack without a value computation. It adds the initial vertices to the set of already visited vertices in $V$ and adds each current vertex to the current traversal path in each $P$. From this point on, effect computations are implicitly performed after each value computation that is applied to the stack. Fig. 3 shows this for the `successors` step that internally applies value computations with `flatMap` and `map` to the stack. Both operations also automatically trigger the effect computations afterwards. Thus, each path effect $P$ adds its new vertex to its internal path list, previously visited vertices are removed from the collection, and remaining vertices are added to the set of already visited vertices. Hence, in contrast to Fig. 2, vertex 3 is no longer in the resulting vertex set.

In order to get such an effectful stack, we can use monads and monad transformers [LHJ95]. Monads allow for the implementation of effect computations and monad transformers combine several monads into a new one. Unfortunately, due to their powerful but also restrictive interface (`flatMap`), they cannot be combined as freely as we would need them for custom graph traversals. Our DSL brings arbitrary effects in any combination into the stack.

2.3 Contributions

Below we will show how to express arbitrary effects in a modular and statically typed way such that they can be combined together with collection classes to an effectful stack. For free composability of more than a restricted set of simple effects, novel and slightly impure monad transformers are needed that allow for a modular and statically typed effect definition. Moreover,
we present how such effects can be added to a stack without affecting the simple signature of `flatMap` and `map` too much, so that the basic query programmer can still apply operations to collections without noticing the effect processing that is being triggered under the hood – and that of course is implemented efficiently by our effect engine.

3 Classes with Monadic Effects

A design goal of our DSL is that an effect programmer can pre-define effect computations for the query programmer to later use them without having to know how the effects are implemented. This section is mainly of interest for the effect programmer.

To define the above mentioned path and visit we need: first, a data type that represents an effect computation on the stack and that holds intermediate effect computation results. And second, the actual implementation of the effect in a modular and statically typed way. For both, we extend monad transformers (that are also monads) and type classes. The latter offer ad-hoc polymorphism and decouple a data type from its monad implementation so that the compiler can statically choose the right type-dependent code for an effect.

We introduce monadic classes that are based on Scala’s implicit feature [OMO10] and make it straightforward to implement new effects as shown in Fig. 4. The effect programmer simply implements the data type and the type class of an effect and automatically inherits all that is needed for the effectful stack.

3.1 Data Type

MonadicData prescribes how an effect programmer has to define effects. This is advantageous over pure monads that usually do not prescribe a distinct data structure at all. The task is to make effects composable and its intermediate computations results accessible within an effectful stack.

For composability data types must resemble monad transformers. In Fig. 4 the data type `PathT` (T for transformer) represents a path effect computation. The type parameter `I[_]` represents a unary nested data type; `A` is the current value type, e.g., a vertex or an edge. MonadicData enforces that sub-classes define their unary type representation using the abstract type member `U[X]`. Thus, the compiler can later use it to select suitable type classes (Sec. 3.2). The unary type representation of `PathT` is `U[X]=PathT[I,X]` where `I` is the unary...
type of a nested monadic class. This results in a multi-layered stack of monadic classes where each stacked data type has its unary type projection that uses the value type of the bottom data type as its type parameter.

For type-safe accessibility to the whole structure of a multi-layered monadic class, a data type has to implement the `inner` attribute declared in `MonadicData`. In the example, `path=ma.inner.inner.pathList` gives access to the `pathList` attribute even if `PathT` is nested in two other monadic classes. As the `MonadicData`'s type parameter `IA` binds the `inner` attribute to `I[A]` and as it holds the concrete type of the nested data type, access to all monadic classes is type-safe. The `pathList` attribute can also be accessed without explicitly denoting any calls of `inner`, which the compiler can insert implicitly without loss of type-safety.

### 3.2 Type Class

A type class implements an effect's computational part. We improve a pure functional monad transformer’s type class [CB14] with both an easier implementation (with focus to our effectful stack) and statically typing (to improve performance and type safety).

The effect programmer has to fulfill the interface `Monadic`. It has two type parameters: `M[_[_]]` is the unary type projection of the monadic classes’ data type. For example, a type class `AnyPath` for `PathT` in Fig. 4 has to use the binding `M[X]=PathT[I,X]`. The second type parameter `B` is bound to `Any`. What is new is that the compiler loads them implicitly if needed. It also loads the type classes of all inner monadic classes that then can be used in a monadic type class implementation. Hence, a type class sees the unary type projection of the inner monadic class as its type parameter `I[_[_]]`.

To ease implementation a type class only has to implement `map`, `smap`, and a value function. A pure structural map (`smap`) is needed for a value computation without effect computation. A type class also needs a value function to access the base value of a multi-layered monadic class. In a pure monad with more layers it gets more unlikely that `flatMap` can be called with a function of type `A=>M[N[O[P[...[B]]]]]]` instead of just `A=>M[B]`. Instead `Monadic` exploits that for graph traversals it suffices to have one-to-one transformations (`map` and `smap`) of the form `A=>B` or one-to-many transformations (`flatMap`) like `A=>C[B]` with an arbitrary collection type `C`. Sec. 4.3 shows how our stack infrastructure automatically reduces the latter to one-to-one transformations.

Multiple parameter type classes [Jon00] improve static typing. A problem of pure monads is that although a type class is chosen at compile time, there is still pattern matching. For example, `map of PathT only has to append the result of the value computation f:A=>B to its recorded path list if B is of type Vertex. Otherwise it has no effect. Pure monads would have to use slow runtime pattern matching on the type of B. To solve this problem `Monadic` has the target type `B` as an extra type parameter. As `map` then returns type `B` we can provide statically selectable type class instances. Two type classes for `PathT` are in Fig. 4. `VertexPath` for `B` is bound to `Vertex` (for path computation) and `AnyPath` for `B` is bound to `Any` (default/no effect).

The strength of multiple parameter type classes becomes even more apparent for monadic classes that work on collection types instead of single value types. The monadic class `VisitT` in Fig. 5 rejects all pre-visited vertices from the resulting collection. A mutable (and synchronized) `HashSet` attribute `visited` is used for that purpose. Beside the obligatory effect-free default
A Modular and StaticallyTyped Effectful Stack for Custom Graph Traversals

AnyVisit, Fig. 5 also depicts CollVisit and ParCollVisit that are selected statically if a value computation results in a collection of vertices. Their map functions filter the vertex collection, reject pre-visited vertices, and otherwise add them to the HashSet.

Together with monadic data types, monadic type classes provide the necessary modularity that allows the query programmer to freely combine the desired effect computations. The next section shows how the effects are combined to an effectful stack.

4 Effectful Stack

How predefined effect computations are combined to an effectful stack is mainly of interest to readers who want to know the details. Sec. 4.1 describes the multi-level nature of our stack for both effect and query programmers. Only effect programmers have to define syntax enhancements (Sec. 4.2) to offer an effect for query programmers who only need rudimentary knowledge about the stack to trigger all effect computations and uniform stack operations (Sec. 4.3).

4.1 Stack Levels

The multi-level stack in Fig. 6 allows a modular composition of monadic classes and collections. It consists of arbitrary different stack levels depending on the needed graph transformation complexity. Each level can hold multiple layers of monadic classes to support the combination of effects within a level. For demonstration, the shown 5 levels suffice.

As effects work on distinct stack levels, an effect programmer needs to know about those levels. Level 1 represents one-to-one transformations using monadic classes (cf. PathT) that work on single values of a level 0 type A. To support multiple values and one-to-many transformations,
Figure 7: Syntactic sugar for multi-level stacks.

level 2 adds a collection class that works on level 1 types (M[A]). To allow for many-to-many
transformations, monadic effects that work on top of collections live on stack level 3. They wrap
a collection class into another multi-layered monadic class M3 that works on the collection type
C[M[A]]. For example in S[A]=VisitT[Id,Set[PathT[Id,A]]] VisitT has the
Set collection as its value type that holds PathT monadic classes for value type A. Hence, the
whole stack is of type S[A]. Id is a pre-defined data-type-free special monadic class that has
to be used on the bottom of each multi-layered monadic class. In order to support repetitions of
many-to-many transformations, we use stack level 4 to put another multi-layered monadic class
M4 on top of a level 3 stack to work on types M3[C[M[A]]], see Sec. 5.2 for the rationale.

4.2 Syntactic Sugar for Multi-Level Stacks

Recall that the successors shorthand works on a vertex collection (Sec. 1). As the effect pro-
grame needs such a syntax enhancement mechanism for all types of effectful stacks, e.g. a ver-
tex based stack, we provide an easy to use implementation pattern that is based on StackType
and Stack which we pre-define, see Figs. 7 and 8. The method vSyntax (line 1) takes an
object of type T that can be decomposed into an effectful stack based on Vertex. vSyntax
results in an anonymous class comprising all syntax enhancements (e.g. successors) that
are applicable to such a stack. This is ensured if the compiler finds a type class instance for
StackType (line 1). The definition of StackType in Fig. 8 that comes with predefined type
classes for each stack level (0 to 4). They decompose a given type T into an effectful
stack and ensure that T can be seen as S[A] with A being a sub-type of a given type A00. The
type members A0–A4 are filled with the concrete value types of the stack. For example, A0
is the level 0 value type A, A1 is the level 1 value type M[A], etc. The type members U0–U4
provide unary type projections for those value types, e.g. U1[X]=M[X], etc. Thus, it suffices to
implement a successors traversal step only once for all stack combinations.

4.3 Multi-Level Stack Operations

Both effect and query programmers need traversals like map, smap, and flatMap that work
on an effectful stack instead of just value collections. We provide such operations within the
predefined Stack type class (Fig. 8) that solve the problem of how to implement such common
operations for an effectful stack. The successors method in line 2 of Fig. 7 internally uses
map and flatMap (not shown) and needs the type class instances S1 and S2 that both provide
the statically typed stack implementations. S1 is based on the StackType’s type v.A0 (line 1)
and ends in an edge-based stack after calling flatMap. S2 is based on the result type of S1
and ends in a vertex-based stack after calling map.

The map and smap functions of a Stack are straightforward to implement. A schematic
body of a map function for a level 4 stack type class instance that accepts a value computation
f of type A=>B is (we use a recursive naming scheme – m4m3cma is a multi-layered monadic
class data type on stack level 4, that has a value type m3cma, that has a value type cma, that ...): M4.map(m4m3cma)(m3cma => M3.map(m3cma)(cma => cma.map(ma => M.map(ma)(f))))

Due to their collection-based nature flatMap and filter cannot be implemented by monadic classes alone. Their code differs from a map's code in the collection part on stack level 2:
cma => cma.flatMap(ma => f(M.value(ma)).map(b => M.map(ma)(a => b)))

Unwrapping each value on stack level 1 with the value function and later re-wrapping into the level 1 monadic class M makes flatMap more expensive than (s)map (see Sec. 7.1).

5 Stack-based Traversal Steps

Query programmers only have to know about the existence of monadic classes and only deal with their data types and pre-defined syntactic sugar. We provide syntax enhancements and additional effect computations to simplify expressing custom graph traversals.

5.1 Traversal Steps using Stack Operations

The pre-defined syntactic sugar for traversal steps in Table 1 is available on all stacks with the denoted value types. For example, predecessors is similar to successors of Sec. 4.2, but computes all adjacent vertices that are reachable via incoming edges. values is a shortcut for unwrapping all level 0 values and produces a collection of type C[A]. run performs a map(a=>a) on an arbitrary stack and triggers all effects in the stack without an explicit value computation (cf. Sec. 2.2). bfs, dfs, and par select an execution order. toList and toSet change the collection type, the latter purges duplicates after each step.

5.2 Traversal Steps using Monadic Classes

We provide pre-defined monadic classes (see Table 2).

Table 1: Traversal steps using stack operations.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 0 value type: Vertex</td>
<td></td>
</tr>
<tr>
<td>successors</td>
<td>All vertices reachable via out-</td>
</tr>
<tr>
<td>predecessors</td>
<td>or incoming edges</td>
</tr>
<tr>
<td>Level 0 value type: Any</td>
<td></td>
</tr>
<tr>
<td>values</td>
<td>All level 0 values</td>
</tr>
<tr>
<td>run</td>
<td>Trigger all effect computations</td>
</tr>
<tr>
<td>Level 2 value type: a collection</td>
<td></td>
</tr>
<tr>
<td>bfs, dfs, par</td>
<td>Change traversal order</td>
</tr>
<tr>
<td>toList, toSet</td>
<td>Change collection type</td>
</tr>
</tbody>
</table>

Value-based Monadic Classes: The path syntax enhancement lifts PathT (Fig. 4) into stack level 1. CountT can be used for faster cycle detection at the expense of a higher memory footprint. It holds a map of vertex/counter pairs and increases a counter whenever a vertex is visited. A counter above 1 indicates a cycle. Access to the map is in constant time, whereas with
Table 2: Traversal steps using monadic classes.

<table>
<thead>
<tr>
<th>Step</th>
<th>Monadic Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: path</td>
<td>PathT</td>
<td>Path computation</td>
</tr>
<tr>
<td>count</td>
<td>CountT</td>
<td>Count vertex visits</td>
</tr>
<tr>
<td>3: context(T =&gt; Boolean)</td>
<td>FilterT</td>
<td>Implicitly filter T</td>
</tr>
<tr>
<td>mfilter1(M[A] =&gt; B)</td>
<td>FilterT</td>
<td>Gather T values</td>
</tr>
<tr>
<td>gather(T =&gt; Boolean)</td>
<td>GatherT</td>
<td></td>
</tr>
<tr>
<td>visit</td>
<td>VisitT</td>
<td></td>
</tr>
<tr>
<td>3 or 4: repeat(T =&gt; T)</td>
<td>RepeatT</td>
<td>Repeat until end</td>
</tr>
</tbody>
</table>

PathT it takes a linear iteration over the path list. CountT and PathT demonstrate the ability to define and to use effect computation in a modular way.

**Collection-based Monadic Classes:** In addition to stack operations like map and filter, there are corresponding monadic classes MapT and FilterT for implicit graph transformations and filtering (cf. context). They apply effect computations continuously after each traversal step. The statement below continuously applies the pre-defined edge filter dependsOn to the traversal and turns a multi-relational graph into a single-relational one:

```
vertices.context(dependsOn).successors.successors.successors
```

Since FilterT works on the value level we also provide mfilter1 to express filter operations that are based on effect computations. It takes a filter function of type M[A]=>Boolean instead of A=>Boolean. The monadic class GatherT gathers elements of type T. As it only works on collections, it has to reside on levels 3 or 4.

**Stack-based Monadic Classes:** The stack level 4 (Fig. 6) is necessary for effect computations that are based on whole stacks, e.g. a repetition of traversal steps. We offer the RepeatT monadic class that takes a function of type T=>T and performs a repeat-until-end effect computation until the collection on stack level 2 is empty. To repeatedly apply the successors step to a level 3 stack one writes: `vertices.visit.repeat(_.successors).run`

### 6 Related Work

Query programmers can choose from many approaches but often only graph traversal languages are suitable for custom graph traversals. The reasons are first, that graph libraries like JUNG (jung.sourceforge.net), SNAP (snap.stanford.edu/snap), or TinkerPop Furnace are often only specialized to single-relational graphs, work in-memory but not on databases, expect a different data format, or do not provide the desired functionality. Second, graph query languages like Neo4j Cypher or SPARQL (w3.org/TR/rdf-sparql-query) are often external DSLs, come with a concise declarative syntax, but also a limited functionality [Woo12]. Query languages are cumbersome to use for traversals that rely on custom traversal orders as it is difficult to influence their internal processing order and optimization technique. On the other hand, graph traversal languages or frameworks like the Neo4j Traverser API or TinkerPop Gremlin combine the advantages of both prementioned worlds and allow for the definition of custom graph traversals in a concise way.

Closest to our effectful stack is TinkerPop Gremlin. It is also an internal DSL, extends its host language syntax (Groovy) with graph traversal operations, and hence overcomes the limitations of query languages. Compared to our approach Gremlin’s modularity is not always sufficient. For example, Gremlin is bound to a DFS traversal order due to its pipe-based architecture. To switch to BFS or parallel execution order a different backend is needed that comes with further requirements. It is also cumbersome to continuously apply implicit graph transformations to
see a multi-relational graph as single-relational for easier implementation. Due to Gremlin’s
dynamically-typed nature, it is also difficult to access the results of additional effects during a
traversal in a structured and statically typed way and it allows to apply traversal steps even on
inappropriate types. The query programmer often only discovers errors at runtime and has to
to guess if a query is correct or not. A Scala-based Gremlin dialect merely provides a front-end to
the existing framework that we consider also untyped.

Pure functional programming on graphs is slower than imperative codes [Erw97], but our
object-functional approach performs much better. King [Kin96] combines pure functional graph
algorithms with state monads but does not offer syntax enhancements. Erkok [Erk02] describes
value recursion on monads but omits effect recursion that we need for RepeatT. Schrijvers
and Oliveira [SO11] suggest the use of so-called zippers and views to provide modular monadic
components. While being purely functional, zippers mask layers within a monad stack and are
an alternative to our object-oriented multiple-parameter type class. But as they do not consider
collection monads in their stack, zippers are unsuitable for graph traversals.

7 Evaluation

Sec. 7.1 evaluates the runtime overhead caused by the effectful stack infrastructure that uses
multiple wrapper classes and type class instances. Statically typed monadic type class instances
are faster than untyped ones (Sec. 7.2). Sec. 7.3 benchmarks a detection of dependency cycles in
several real-world Java codes (see Table 3). We extracted the corresponding vertices and edges
with the help of the ASM library (asm.ow2.org).

For all our measurements we mask outliers by taking the best out of 10 mea-
surements on a Windows 7 x64 PC with
16 GB RAM and a quad-core CPU with
hyperthreading (8 cores) and a nominal
3.5 GHz. Test programs use Scala 2.10.1
and run on a Java SE 7u25 (x64) VM. We
use Gremlin 2.3.0 and Groovy 2.0.7. 12
GB have been allocated for Java to mini-
mize garbage collection runs.

<table>
<thead>
<tr>
<th>ID</th>
<th>Project</th>
<th>Element vertices</th>
<th>Count edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Cobertura 2.0$^\alpha$</td>
<td>12,310</td>
<td>26,385</td>
</tr>
<tr>
<td>B</td>
<td>Squirrel SQL Client 3.5.0$^\alpha$</td>
<td>24,617</td>
<td>57,105</td>
</tr>
<tr>
<td>C</td>
<td>Apache Ant 1.9.1$^\alpha$</td>
<td>44,801</td>
<td>103,580</td>
</tr>
<tr>
<td>D</td>
<td>Eclipse Debug UI 3.8.2$^\beta$</td>
<td>76,904</td>
<td>195,227</td>
</tr>
<tr>
<td>E</td>
<td>Eclipse JDT UI 3.8.2$^\beta$</td>
<td>340,378</td>
<td>896,326</td>
</tr>
<tr>
<td>F</td>
<td>Scala Compiler 2.10.1$^\gamma$</td>
<td>562,122</td>
<td>1,555,970</td>
</tr>
<tr>
<td></td>
<td>Combined Java codes:</td>
<td>1,061,132</td>
<td>2,834,593</td>
</tr>
</tbody>
</table>

$^\alpha$: sourceforge.net, $^\beta$: eclipse.org, $^\gamma$: scala-lang.org

7.1 Infrastructure Runtime Overhead

Although all parts of an effectful stack influence the runtime, we are interested in the infrastruc-
ture overhead caused just by monadic classes, the multi-level stack, and the type class instances
created and executed at runtime. Hence, we work with effect-free \texttt{IdT} monadic classes and a
list collection with an effect-free \texttt{flatMap} operation.

The stack operation determines which functions of the monadic classes are called, especially
on level 1. For example, the \texttt{map} operation in Sec. 4.3 is only called per level 1 monadic class,
whereas \texttt{flatMap} uses \texttt{value} and \texttt{map} in addition to a collection-based \texttt{map}. The com-
plicity of \texttt{filter/foreach} lies between \texttt{map} and \texttt{flatMap}. Hence, to evaluate we group
stack operations according to their complexity and measure the overhead for \texttt{map/smap} (using \texttt{map(_.outEdges.size)}, \texttt{filter/foreach} (using \texttt{filter(Class)}), and \texttt{flatMap} (using \texttt{flatMap(_.outEdges)}) on a vertex set. We also distinguish level 1 and level 3 monadic classes. For example, a simple \texttt{map} operation on a level 3 stack has to trigger all level 1 monadic classes within the collection, whereas a level 3 monadic class is triggered only once.

Table 4 shows the results of our measurements. As a worst case scenario, we used all vertices of the combined projects A–F (Table 3) and a level 2 effectful stack using a vertex list (in a list, \texttt{flatMap} has no additional effect after a step) with the shown number of level 1 \texttt{IdT} monadic classes in it. We applied each of the mentioned functions to the stack once. On the large graph, we can see the influence of multiple level 1 monadic classes. \texttt{map} causes little infrastructure overhead, about 0.6% per monadic class layer, whereas \texttt{filter} (8.4%) and \texttt{flatMap} (18%) are more costly. The last line of Table 4 shows that parallelization boosts performance for a computation with three monadic classes (3x \texttt{IdT}). We also measured the overhead of level 3 and level 4 monadic classes, but as they only produce overhead per traversal step, the vertex count does not matter and the runtime overhead (about 0%) can be ignored.

### 7.2 Advantages of Static Typing

To illustrate the advantage of statically typed monadic type class instances over untyped monad type class instances, we compare the runtime of a graph traversal that uses \texttt{VisitT} to a monad-based implementation. We again traverse the combined Java code graph of projects A–F with:

\begin{verbatim}
vertex.visit.repeat(_.successors).run
\end{verbatim}

The statement starts at the project’s \texttt{CodeBase} vertex and traverses in BFS order to each vertex once. With \texttt{VisitT} this takes 5.63 seconds which is 16.5% faster than the monad-based code (6.56 seconds). Hence, in addition to a productivity gain due to bug detection at compile-time, statically typed monadic classes also have a performance advantage, mainly due to static code selection instead of runtime pattern matching.

### 7.3 Dependency Cycle Detection

We now benchmark a custom graph traversal that detects dependency cycles in Java code, which is an anti-pattern to good software [MT06]. Fig. 9 holds an example code graph. Cycles among top level types (not nested within other types) are the anti-pattern. The code in Fig. 10 finds the first 10 disjunct elementary cycles [Joh75] that consist of at least 4 different vertices. This code works in two steps:

![Figure 9: Some dependsOn edges between top level types.](image)
Step 1 computes the dependencies between top level types and persist them temporarily by means of new edges of type dependsOn. In Fig. 9 the top level type Class1 depends on Class2 and Class3, as at least one of its members has a dependency to a member of each of the target classes. As Class4 is not a top level type it is not a dependence for Class1. This is coded in Lines 1–2 of Fig. 10. The method step1 expects a top level type vertex and gathers in s all its nested code elements (using pre-defined vertex filters CodeElement and nesting). Note the additional GatherT monadic class. We lift it into stack level 4 (RepeatT) to make all visited vertices available in the gathered attribute. Afterwards (not shown) the function traverses over existing dependencies to all dependent code elements and gathers their top level types. We then create new dependsOn edges between the vertex v and all of those top level type vertices (without creating any self-loops). step1 returns its input vertex v in line 2 as it performs an in-place transformation. Later, line 4 applies the step1 method to all vertices once, using the effectful stack’s map operation.

Step 2 searches for cycles among top level types using only dependsOn edges until an abort criterion is met (lines 3–6). First, we create a HashMap for the found elementary cycles that are represented as a vertex list and stored using this Set representation in order to filter out isomorphic cycles. Second, we lift all necessary effects into the stack in line 4: CountT, PathT, and FilterT (using context). We do not need a VisitT as there is no need to filter out previously visited vertices. Instead we need to detect a cycle first and then filter out already visited vertices. We use another FilterT for that in lines 4–5. It returns false if we already found 10 cycles or if the current vertex was already visited on its traversal path. In the latter case, the count attribute of CountT equals to 2 and the addPath function (not shown, but always returns true) checks if the found cycle has an appropriate length and adds it to cMap if necessary. Line 6 contains the core traversal function that repeats a successors step until the level 2 collection is empty and hence until we collected 10 relevant cycles or we reached the end of the graph. The last statement returns the found cycles.

On a Neo4j 1.9.2 database we get the results in Table 5 when applying both traversal steps to each project A–F. Table 5(a) shows the measurements for step 1 in milliseconds. As expected, the BFS runtime (average is set to 100% in the last column) grows proportional with the project graph size (all projects have a similar edge/vertex ratio of about 2.45). Parallelization (insert .par before map in line 4) speeds up the runtime by about a factor of 4 on average. It only takes 24% of the sequential BFS effort. As step 1 has to transform the whole graph, DFS order (insert .dfs before map in line 4) has the same runtime, even when parallelized. Hence, Gremlin cannot benefit from its pipe-based architecture and shows slower results.

Table 5(b) shows both the number of dependsOn edges that step 1 inserts into the graphs and the ratio of the projects’ total edge counts and the dependsOn edge counts. The ratios of
projects A and F are well below average. The ratio directly influences the runtime of step 2 as a high `dependsOn` density makes cycles more likely.

Table 5(c) shows the measurements for step 2 in seconds (for DFS order in milliseconds) that were run in BFS (set to 100%), BFS+PAR, and DFS order. As cycle detection is a DFS problem, DFS order is much faster (on average 0.3% of the BFS runtime) even with parallelization engaged for BFS. There are two outliers in the DFS performance. As project A has no cycle that fits the requirements, the whole graph has to be traversed which results in a bad DFS performance. In project F the DFS runtime is better than in project E due to the low edge ratio, even if the total graph size of F is twice that of project E.

The total runtimes to find dependency cycles in two steps are in Table 5(d). As Gremlin cannot parallelize step 1 and is also slower in step 2, our effectful stack performs much better. Moreover, the Gremlin code needs about 70% more code (measured using compiler tokens) than our monadic graph stack approach. This use case also clearly shows the advantages of the free combinability of any traversal order with parallelization, simply by adding `dfs/par` traversal steps to a query. Our effectful stack allows to add necessary effect computations in a modular way, e.g. path computations, as they are fully interoperable with all traversal orders.

### 8 Conclusion

This paper presents an effectful stack for custom graph traversals that allows for the modular application of additional effect computations and processing rules. We leverage the concepts of monads, monad transformers, and type classes, but modify them into monadic classes to achieve modular composability of multi-layered monadic classes that also provide structured access to their data. Monadic classes are easy to implement and statically typed. The latter speeds things up as only necessary effects are executed. Monadic classes are combined with collection monads into an effectful stack. Syntax enhancements ease modifying and working with this stack. The infrastructure runtime overhead of our effectful stack is low. We showed the applicability of our ideas by finding an anti-pattern (elementary cycles) in real-world Java codes. With the ability of running traversals in parallel we outperform a Gremlin version of this analysis.
References


