A Modular, Extensible, and Polymorphic Graph Query DSL

Alexander Taylor

December 28, 2018

Name:	**Alexander Taylor**
CRSID:	**at736**
College:	**St John's College**
Module:	**Metaprogramming, L305**
Assignment:	**4**
Word Count:	**todo**[1]

Abstract

This project is an extension of my Part II project [2], the purpose of which was to build a purely functional graph database system. Although the DSL I produced for the project did provide Scala-compile-time type checking, it used an ADT to construct DSL terms and hence was not tagless-final. Further more, the DSL required implementations to return values of a specific monad type, which made it unsuitable for implementations other than immediate interpreters, such as an implementation that compiles queries for later execution. While my Part II project focused on the full stack and optimisations of particular backends, this L305 project focuses on the front end of a graph database system. The project's primary aim is to open up the interface of the DSL to more varied implementations. It replicates the original DSL in a Modular tagless-final fashion, allowing implementors to choose which operations and syntax they support. Furthermore, it provides a high degree of polymorphism, both in the types used to represent queries, the return types of the database, and in the typeclasses used by an implementation to classify which types can be stored in the database. The project also includes combinators to generate "free" implementations of parts of the specification of DSLs. Finally, I present three new back-end implementations: a simple, set based, in memory interpreter, interfacing code to plug in the back-ends for my original part II project to this part III project, and a simple optimising compiler implementation that compiles queries to a bytecode for later execution.

[2] Dissertation: https://github.com/Al153/PartIIProject/blob/master/diss/diss.pdf,
Code: https://github.com/Al153/PartIIProject/tree/master/src

Contents

Chapter 1

The Original Project

1.1 Introduction

My original part II project built the full stack of a graph database system. It provided a strongly typed DSL, a type-class based framework for allowing an application developer's types to be used in queries, and several different backends targeting an in-memory datastore, PostgreSQL, and the LMDB memory-mapped datastore. This project acts as a DSL generator of sorts for faster, more consistent generation of DSLs for graph-query executors.

1.1.1 The Query Language

The original project uses an algebraic-datatype-based query language to specify queries that search for pairs of related objects and individual objects. As defined in my dissertation, the basic query constructors are as follows.

$$
\begin{aligned}
P \rightarrow\ & Rel(R) \text{ Find pairs related by the named relation R} \\
\mid\ & RevRel(R) \text{ Find pairs related by the named relation R in the reverse direction} \\
\mid\ & Chain(P, P) \text{ Find pairs related by the first subquery followed by the second} \\
\mid\ & And(P, P) \text{ Find pairs related by both of the sub-queries} \\
\mid\ & AndRight(P, S) \text{ Find pairs related by P where the right value is a result of S} \\
\mid\ & AndLeft(P, S) \text{ Find pairs related by P where the left value is a result of S} \\
\mid\ & Or(P, P) \text{ Find pairs related by either of the sub-queries} \\
\mid\ & Distinct(P) \text{ Find pairs related by P that are not symmetrical} \\
\mid\ & Id_A \text{ Identity relation} \\
\mid\ & Exactly(n, P) \text{ Find pairs related by n repetitions of P} \\
\mid\ & Upto(n, P) \text{ Find pairs related by up to n repetitions of P} \\
\mid\ & FixedPoint(P) \text{ Find the transitive closure of P}
\end{aligned}
\tag{1.1}
$$

$$
\begin{aligned}
S \rightarrow\ & Find(F) \text{ Find values that match the findable F} \\
\mid\ & From(S, P) \text{ Find values that are reachable from results of S via P} \\
\mid\ & AndS(S, S) \text{ Find values that are results of both subqueries} \\
\mid\ & OrS(S, S) \text{ Find values that are results of either subquery}
\end{aligned}
\tag{1.2}
$$

Full details, including the type system that applies over this query language can be found in **Section of my dissertation**. Once a query is built, it may be executed in one of several ways by the `DBExecutor` of a given implementation. For example, the executor may simply read the queries from the database or perhaps do pathfinding over the graph with the queries.

1.2 Flaws in the Project

There are several flaws in the front end of this project, as necessitated by the time constraints of a part II project, as well as my desire to look at the big picture of the system rather than focusing for too long on the front end.

1.2.1 Fixed Query ADT

The original project constructed a fixed ADT representing each query. ADT was then traversed or otherwise evaluated to interpret the results of the query. In order to get pattern-match-exhaustivity checking, one has to seal a trait in Scala. This means that the ADT backing the DSL is fixed and cannot be directly extended or reduced. A tagless-final DSL is much easier to extend or reduce.

1.2.2 Lack of Polymorphism

the lack of polymorphism in the original DSL puts unnecessary constraints on back-end implementation.

Types Stored in the Database The original project uses the `SchemaObject` type-class to specify which types can be stored in the database. This type-class is fixed, only allowing types that can be represented as tuples of simple primitive types to be stored in the database. Allowing polymorphism over the type-class used to verify types manipulated by the DSL allows for more freedom of implementation. For example, we may choose to store more complex types to the database, such as wide floating point values, or higher order relations.

Fixed Return Types The original database interface requires implementations to return instances of the specific `Operation` monad (a stack of asynchrony, state, error monads). And only allows read operations to return the eager `Set` type of results. That is, the return type is `Operation[Error, Set[(A, B)]]`. This specific return type constraint prevents implementors from returning more interesting values, such as compiled code for later use, or lazy alternatives to sets. Furthermore, it locked the implementation into using the same monadic, branching time, update semantics specified by the `Operation` monad.

Chapter 2

The New DSL

This section introduces the new structure of the DSL.

2.1 Introduction

In the use cases of this project, I make reference to several different classes of people.

- **spec writer** - The person, in this case myself, building the modules and combinators used to construct back end implementations.
- **Implementor** - The person using this project to build a graph query DSL.
- **Application Programmer** - The person using a graph query DSL, built by the implementor, to build an application.

2.2 Tagless-Final

The new DSL is a modular tagless-final based system. There are several traits to implement, each containing a subset of the operations defined above. This allows for implementations to only partially implement the specification or to add new modules. The DSL is polymorphic in the type constructors `Pair` and `Single` which are used to specify queries returning pairs or single objects. This allows more flexibility than the original ADT based interface.

2.3 Modularity

The new DSL framework is built in a modular way. Allows more freedom to implementors to pick and choose which functionality they want to provide to the database system. This allows more specific DSLs to be designed to use particular parts of the algebra.

2.3.1 Important Classes/Types/Traits

Query Building The tagless-final interface for building queries is separated into several pieces. **Check names**

- **Simple Pairs** - `and`, `or`, `id`, etc.
- **Singles** - All of the single query combinators.
- **Simple Repetitions** - `upto`, `exactly`

- **Fixed Point** - `fixedpoint`

This allows implementors to implement the interfaces in separate parts. It also allows for construction of less expressive DSLs, for example, without the fixed point combinator.

Execution The methods for interpreting a query are separated out into separate interfaces for each function, straight interpretation of queries, path-finding using queries, and updates to the database. These are separated as in the case of queries to allow for partial implementation.

Assertion and Test Suite I have replicated a small subset of the unit testing suite I used in my part II project. This is implemented as a mix-in trait depending on a full DSL implementation, and a monad typeclass for the return type. It consists of boolean operations for testing the structure of unexecuted queries and the result of testing queries, an assertion operation for checking the results of queries, and a syntax module for more natural expression of tests. This suite can be found in section **Where?**.

2.3.2 Dependency Management

This module system allows us to delegate dependency management to the type-system. Each module is designed as a mix-in trait. It can make requirements on the rest of the implementations by requiring mix-ins using self type annotation. Hence, modules without the correct dependency on the implementation will not compile. This gives back-end implementors a set of type-level combinators for building the shape of their back-end. It also makes life easier for application writers, since the available methods are visible at compile-time, or write-time when using an IDE, such as Intellij Idea. If one was build a similar modular system in a Java style, this information would not be visible in the type signature of the database system, and could lead to exceptions being thrown at runtime due to an unimplemented method being called.

The dependency management scheme can also be used by implementors to add their own extension modules.

A simple example of type-level dependency management is the `query.dsl.components.BatchInserts` trait, which supplies some syntactic sugar for inserting items.

```
trait BatchInserts[M[_], ToInsert[_, _], Valid[_]] {
   self: Writes[M, ToInsert, Valid] =>
   final def inserts[A: Valid, B: Valid](xs:  ToInsert[A, B]*): M[Unit] =
        self.insert(xs.seq)
}
```

The trait requires itself to be mixed in with a Writes implementation, and hence provides some syntax to the class. There are further examples of this in the sections about syntax providers and free optimisations.

2.4 Syntax

To facilitate the use of Neo-4j-like arrow syntax, there are a series of syntax provider modules. A syntax provider is a module providing the implicit conversions to allow the required syntax. Each syntax provider is a mix-in, parameter-polymorphic, trait using self-type annotations [1] to manage dependencies using the type-system as discussed in section 2.3.2. Hence the application programmer need only import the DSL implementation's internal methods to get access to the syntax.

```
val dsl: DSL[Type Params] = ...
import dsl._
```

[1] https://docs.scala-lang.org/tour/self-types.html

```
val result = read(Knows -->--> Owns >> (Has4Wheels))
// result: M[Se[Person, Car]]
```

Syntax providers are kept modular in order to facilitate extended or partial implementations of the spec. The providers are required, as opposed to globally scoped objects due to the large number of polymorphic type parameters, which the Scala type-system is unable to infer. **Put this last sentence earlier.**

2.5 Polymorphism

A lot of this flexibility and deferral to the scala type system is made possible by the large amount of polymorphism allowed in the system. Each trait is polymorphic in as many variables as possible in order to generalise the system to the greatest extent.

```
trait DSL[M[_], Se[_], Pair[_, _], Single[_], Find[_], Path[_], ToInsert[_, _], Valid[_]]
extends Backend[M, Se, Pair, Single, Find, Path, ToInsert, Valid]
  with PairSyntaxProvider[Pair, Single, Valid]
  with SingleSyntaxProvider[Pair, Single, Find, Valid]
  with SymmetricSyntaxProvider[Pair, Single, Valid]
  with SimplePairs[Pair, Single, Valid]
  with SimpleRepetition[Pair, Valid]
  with FixedPoint[Pair, Valid]
  with SingleQueries[Pair, Single, Find, Valid]
  with BatchInserts[M, ToInsert, Valid]
```

note: the extremely polymorphic DSL trait is parameterised in as many types as possible.

2.5.1 Query Types

As with most tagless-final interfaces, the `Pair[A, B]` and `Single[A]` type constructors are provided as type parameters.

2.5.2 Type Validation

The `SchemaObject` type-class used in the original project to store application-programmers' types in the database is replaced by the `Valid[_]` type-class, allowing implementors to specify their own validity type-classes. **Have I explained what the validity type class is for?**

2.5.3 Return Types

The project is also polymorphic in the return type container and sequencing type constructors. The container type, `M` wraps the result of all operations, as it generalises the `Operation` monad used in the original project. The container type can be required to be a monad by using the `HasMonad[M]` mix-in. Other implementations may want to use another kind of type, such as `Rep` as the container type. The sequencing type constructor is the type of results returned by read queries. In the original project, it is implemented using eager `Sets`. Other implementations may want to use lazy sets instead, especially when dealing with large databases.

2.5.4 Managing Polymorphism

The large number of polymorphic type parameters bloats the type signatures of components and means we have to be careful about type inference. For example, if we provided a globally available implicit class with syntax for generating pair queries, it would need to look something like the following.

```
implicit class PairSyntax[P[_, _], S[_], Valid[_], A: Valid, B: Valid](p: P[A, B]){
    def &(q: P[A, B]): P[A, B] = and(p, q)
    def -->>(s: S[B]): P[A, B] = andRight(p, s)
    ...
}
```

In order to apply this syntax to `p: P[A, B]`, the Scala type system would need to correctly infer types of `S`, `Valid`. This is not possible for the Scala type system, meaning that `Nothing` types would populate the un-inferred type parameters. Ideally, we want Scala's type system to infer as many type parameters as possible and for application programmers to use as few type annotations as possible. As a result of this, having mixed-in syntax providers which provide the correct type parameters gives us the cleanest application syntax.

Chapter 3

Back-end implementations

I've provided several simple implementations for this project to showcase features.

3.1 Trivial implementation

The first implementation is one that trivially uses sets to store objects in memory. The validity type-class is the "Universe" type class that verifies that there exists a finite universe of objects of the parameter type in memory. This implementation uses simple techniques to generate the DSL in a free manner.

3.2 Original Back-End

A second implementation is one that provides a set of connectors to allow a suitable DB instance to be used as as a L305-Project DSL instance. This is a simple case of deferring functionality to the instance's original methods and correctly filling in types.

```
...
= new DSL[...]
    with RuntimeTestTools[...]
    with HasMonad[Op]
    with AssertionTools[Op]
    with Lifts[...] {
    ...
    override def and[A: SchemaObject, B: SchemaObject](
        p: FindPair[A, B], q: FindPair[A, B]
    ): FindPair[A, B] =
        And(p, q)
    ...
    override def upto[A: SchemaObject](
        p: FindPair[A, A], n: Int
    ): FindPair[A, A] =
        Upto(n, p)
    ...
    override def readPair[A: SchemaObject, B: SchemaObject](
        p: FindPair[A, B]
    ): Operation[Err, Set[(A, B)]] =
        d.executor.findPairs(p)
    ...
```

```
    override def shortestPath[A: SchemaObject](
        start: A, end: A, p: FindPair[A, A]
    ): Operation[Err, Option[Path[A]]] =
        d.executor.shortestPath(start, end, p)
    ...
}
```

Note: example of some some implementation of the part II backend converter

3.3 Byte Code Back-End

In addition to these directly executing back-ends, I have also built a partial back end implementation for constructing back-ends that compile queries to a simple byte-code for later execution. I then created a simple implementation of this back-end which executes the bytecode in an imperative fashion in memory.

3.3.1 Further Parameterisation

This partial implementation allows parameterisation

3.3.2 Compilation Steps

This implementation first maps tagless final terms to a type-erased AST **Link to code file?**. That is, the object types associated with each AST term are discarded. This allows us to make optimising transformations on the AST more easily. `Upto` terms are replaced with the equivalent `Exactly` terms, as seen **where?**. Finally, primitive relations, identity relations, and findables are converted to nodes indicating a call to external procedures provided by the sub-implementation.

The `Reads` module's methods are implemented by next compiling the AST into a bytecode program. Operations such as joins, unions, and intersections have opcode equivalents, whereas `Exactly` and `FixedPoint` operations are implemented using conditional loops. Furthermore, simple loop unrolling is used if the number of repetitions in an `Exactly` is less than 5. A binary exponentiation algorithm is used to find the fixed point.

The `Reads` module's methods finally return a `Rep` type (not the same as in LMS) whose `run` method runs the packaged interpreter over the bytecode then uses subimplementation-provided methods to read values of the correct type from the result.

3.3.3 The Bytecode

The bytecode consists of set of stack machine instructions and is type-parameterized by a type of `Labels` for making jumps and `Procedures` for calculating primitive relations and sets. This allows an implementation to pick types that suit the exact use case. The bytecode consists of the following instructions.

- Relation manipulation: `And`, `Or`, `Join`, etc.
- An external procedure call: `Call`
- Stack manipulation instructions: `Swap`, `Dup`, `Drop`
- Conditional and unconditional branches `DecrementAndJump`, `BranchIfNotEqual` **name?**, and `Jump`.
- **Any more?**

3.3.4 Interpreters

Subimplementations should provide an interpreter for the bytecode. The partial implementation is parameterised in such a way that implementations may provide a safe, monadic, locked down interpreter or a faster, less safe one.

3.3.5 Implementation

As an example, I have constructed a simple full implementation of the bytecode interpreter back-end. It implements the bytecode interpreter in a simple, imperative, stack machine. This does not use monads to control execution or handle errors and instead simply allows any exceptions to bubble up as in a Java program. The `Compilable` typeclass maps individual objects to a simple identifier which is processed by the stack machine.

Chapter 4

Free Implementations and Optimisation

Since the module system exposes polymorphism over many parts of the implementation of a DSL, we can look for ways to exploit this polymorphism to reduce the burden on DSL implementors.

4.1 Implementations

Some modules of DSL implementations can be defined naturally in terms of other modules. For example, as proven in the original project **Which section?**, one can formulate the `exactly(n, P)` operation as a collection of `chain` operations. Furthermore, one can formulate `upto(n, P)` as `exactly(n, or(p, Id))`. Hence we can freely implement `exactly` and `upto` given an implementation of the `SimplePairs` module. Free implementations give us combinators for generating default methods of DSLs in a clean manner.

In general, we can construct a free implementation of a trait using mix-ins as follows:

```
trait Free[TypeParams] extends ToBeImplemented[Types] {
    self: Dependency1[Types1] with Dependency2[Types2] =>
    // implement the methods of ‘ToBeImplemented‘ using
    // methods of ‘Dependency1‘ and ‘Dependency1‘
    def foo[A, B](a: A): B = ...
}
```

A free implementation can be used by mixing it in with implementations of the required dependencies.

```
object MyDSL extends Dependency1[Types1] with Dependency2[Types2] with Free[Types]
```

As an example, I have created a free implementation of **SimpleRepetitions** depending on **SimplePairs** as explained above. This implementation uses the exponentiation-by-squaring technique explained in my part II project **which section** and can be found in **which section?**

One downside to overuse of free implementations is that they may not provide well optimised or performant queries for particular back ends. For example, it may be possible to carry out a backend-specific optimisation that is not applicable in the general case.

4.2 Optimisation

There currently exist (and it may be possible to find more) examples of queries which can be optimised across many implementations. For example, `Distinct` distributes over `and` and `chain` distributes over

`or`, and `and` and `or` distribute over each other. These identities can be exploited to push "narrowing" operations (which decrease the size of the result, e.g. `and`) to the leaves of the AST and "widening" operations (which increase the size of the result, e.g. `or`) to the root of the AST. This means that the majority of query evaluation handles smaller subqueries and hence runs faster.

An additional extension, given more time, might be to construct a combinator, which given a `PairQueries` implementation, returns a new implementation that can freely apply well typed optimisations to an internal GADT before folding over the GADT to produce a query of the original type. Unfortunately, as it is cumbersome to properly fold functions over the GADT in Scala, due to the type-system, I ended up cancelling this extension to the project.

Chapter 5

Running The Code and Examples

Viewing and running this project is perhaps best performed in the Intellij Idea IDE, with SBT installed, as one can inspect the inferred type of a term using **Alt + =** , follow the definition of terms using **ctrl + b**, and go back to previous views using **ctrl + alt + left-arrow/right-arrow**. As the project has some dependencies on my part II project, SBT will download it from github and hence may take some time to run when the project is first compiled.

Examples can be demonstrated by running the main methods in `examples.TrivialExamples`, `examples.BytecodeExa` and `examples.Part2Examples`. Each class gives some example queries and also definitions of the appropriate typeclass.

Chapter 6

Conclusion

This project has presented a set of tools and example uses of these tools for constructing graph database domain specific languages. It has done so making heavy use of the Scala type-system to allow flexibility of implementation and to ensure type and dependency-safety.

Given more time, I would like to have explored a construction of a type algebra to allow polymorphic optimisations to be applied, entirely provided using mix-in traits. E.g.

```
object MyFastImplementation extends FreeOptimisations(MyImplementation)
    with DistributeJoinsUnions[...]
    with DistinctElimination[...]
    with JoinIdentity[...]
```

Further, I would like to have had more time to explore bytecode compilation. I would have liked to provide a tagless-final interface for constructing bytecode programs, in order to allow the code to be compiled to more exotic backends, such as cross-compilation.