Abstract

This project is an extension of my Part II project [1], 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. It replicates the original DSL in a Modular tagless-final fashion, allowing implementations 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 contains code to plug in my part II backends to use the new DSL. Finally, the project also contains "free" implementations of modules and a suite of "free" optimisations.

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

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 typeclass 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.

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 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**

1.2 Flaws in the Project

There are several flaws in the front end of this project, as necessatiated 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.

- Fixed query ADT - prevents extension or partial implementation - can fix using tagless Finally
- Lack of polymorphism puts unnecessary constraints on backend implementation
- The fixed `SchemaObject` typeclass restricts what can be stored in a database by a backend. (Can only store simple tuple types). This can be fixed by allowing polymorphism over the typeclass used to verify types manipulated by the DSL.
- Operation monad / set return type prevents us from doing delayed computation. Implementations must directly interpret the queries. It would be better to allow partial evaluation/compilation of queries to give objects that contain optimised code to be run later.
-

Chapter 2

The New DSL

2.1 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

2.2 Syntax

2.3 Polymorphism

2.4 Modularity

2.5 Testing

Chapter 3

Backend implementations

3.1 Trivial implementation

Chapter 4

Free 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]
```

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.

Chapter 5

Free Optimisations