Embedded Typesafe Domain Specific Languages for Java

Jevgeni Kabanov
Dept. of Computer Science
University of Tartu
Liivi 2, Tartu, Estonia
ekabanov@gmail.com

Rein Raudjärv
Dept. of Computer Science
University of Tartu
Liivi 2, Tartu, Estonia
reinra@gmail.com

ABSTRACT

With projects like jMock and Hibernate Criteria Query embedded DSLs have been introduced into Java. We describe two case studies that develop embedded typesafe DSLs for building SQL queries and engineering Java bytecode. We proceed to extract several patterns useful for developing a typesafe DSL for arbitrary domains. We also show the limits of type safety that can be achieved in Java. Both projects developed for the case study are available as open source.

1. INTRODUCTION

Domain specific language usually refers to a small sublanguage that has very little overhead when expressing domain specific data and behaviour. DSL is a broad term [10, 2] and can refer both to a fully implemented language and a specialized API that looks like a sublanguage [8], but still written using some general-purpose language. Such DSLs in the latter meaning have been introduced by both the functional [3] and dynamic languages community [5]. Both these communities (and especially functional) took advantage of function composition and operator overloading to build combinator-based languages that look nothing like the original. The functional communities also strongly support the notion of type safety; therefore DSLs they create are usually statically typed.

In Java the DSLs are also becoming more popular. The first examples like jMock [6] and Hibernate Criteria [1] were coined as Fluent Interface. However most of these DSLs are not typesafe and will allow wrong statements to be compiled. In this paper we introduce some patterns that help making Java DSLs safer.

TODO

2. TYPESAFE SQL

Let's start with a very simple example of an SQL query in Java.

```
ResultSet rs = SqlUtil.executeQuery(
  "SELECT name, height, birthday" +
  "FROM person" +
  "WHERE height >= " + 170);
while (rs.next()) {
   String name = rs.getString("name");
   Integer height = rs.getInt("height");
   Date birthday = rs.getDate("birthday");
   System.out.println(
     name + " " + height + " " + birthday);
}
```

Already this example, simple as it is, illustrates the mistakes we can make:

- We could have misspelled any SQL command, like "SELETC".
- We could be misspelling the table name or the field names.
- We could be mistaken about the field type (e.g. height being a *boolean*, where *true* means "over 170 cm").
- We could be reading wrong types from the result set.

The problem is that we would only find out about those errors when the query is executed. To make it worse, some errors would not even be reported and since most queries are assembled dynamically we can't ever be sure that it is error free.

The solution we propose is to build on recent innovation in the area and embeds the whole of the SQL as a typesafe embedded DSL. The following example shows what we propose it to look like:

```
Person p = new Person();

List<Tuple3<String, Integer, Date>> persons =
  new QueryBuilder(datasource)
  .from(p)
  .where(gt(p.height, 170)
  .select(p.name, p.height, p.birthday)
  .find();
for (Tuple3<String, Integer, Date> person : persons) {
  String name = person.v1();
  Integer height = person.v2();
  Date birthday = person.v3();
  System.out.println(
    name + " " + height + " " + birthday);
}
```

Although this example does the same as the previous there is not a single string and any kind of misspelling or type inconsistency will show immediately as a compile-time error[1]. In the following sections we explain how to achieve full type safety in this DSL.

2.1 Tuples

You may have already noticed that we make sure the result set types are not inconsistent by combining them into a class called `Tuple3`.

Tuples are sequence of values where each component of a tuple is a value of specified type. Often used in functional languages they are not natively supported in Java. All the same the corresponding classes can be easily generated. For example a tuple with the length of two is following:

```
public class Tuple2<T1, T2> implements Tuple {
  public final T1 v1;
  public final T2 v2;

  public Tuple2(T1 v1, T2 v2) {
    this.v1 = v1;
    this.v2 = v2;
  }

  public T1 v1() { return v1; }
  public T2 v2() { return v2; }
}
```

We use tuples to return the query results with the right types. Instead of `Tuple1` we can just use the unboxed type.

2.2 Metadata Dictionary

The first step towards type safety of the query itself is ensuring that table and column names we use do in fact exist and are spelled correctly. To ensure that we use the database metadata about the tables and columns to generate a typesafe *metadata dictionary*.

Metadata dictionary is a set of information about database describing tables and columns with their types. In Java the dictionary of the table Person could be the following[2]:

```
public class Person implements Table {
  public String getName() { return "person"; };

  public Column<Person, String> name =
    new Column<Person, String>(
      this, "name", String.class);
  public Column<Person, Integer> height =
    new Column<Person, Integer>(
      this, "height", Integer.class);
  public Column<Person, Date> birthday =
    new Column<Person, Date>(
      this, "birthday", Date.class);
}
```

This metadata dictionary for the `Person` table associates the table with its name and columns. Each field is in turn associated with its name, type and owner table. The generic type variables in the column definition provide us with compile-time type information.

[1]Even more importantly with a sufficiently advanced IDE it will be marked as an error directly in the text of the program providing immediate feedback.

[2]We do not care how the dictionary is generated as it is irrelevant to how it can be used

2.3 Builders

To make use of the metadata we need to build the query itself. We proceed by separating the query building into stages (*from*, *where*, *select*, ...) and delegating a *builder* for each of those stages. Thus we can make sure that the basic syntax of the query is always correct by making it impossible to write it wrongly without a compile-time error.

One of the main idioms in creating Java DSLs is hiding the return type by chaining the calls on the previous call result. Although typically most methods will return "this" we can use it to stage the query building and allow only relevant methods to be called.

To examine this in detail let's recall our previous example, but omit the "where" part for the moment:

```
Person p = new Person();

List<Tuple3<String, Integer, Date>> persons =
  new QueryBuilder(datasource)
  .from(p)
  .select(p.name, p.height, p.birthday)
  .find();
```

The `QueryBuilder` does not do much more than store the datasource. The `from()` method returns the `FromBuilder` that stores the table from the dictionary:

```
public class QueryBuilder {
  ...
  public <T extends Table> FromBuilder<T>
    from(T table);
}
```

The `FromBuilder.select()` method returns `SelectBuilder3` that stores three specific columns from the table.

```
public class FromBuilder<T extends Table>
  extends QueryBuilder {
  ...
  public <C1> SelectBuilder1<T, C1>
    select(Column<T, C1> c1);
  public <C1, C2> SelectBuilder2<T, C1, C2>
    select(Column<T, C1> c1, Column<T, C2> c2);
  public <C1, C2, C3> SelectBuilder3<T, C1, C2, C3>
    select(
      Column<T, C1> c1,
      Column<T, C2> c2,
      Column<T, C3> c3);
  ...
}
```

Note that instead of having only one `SelectBuilder` we choose to have many of them, numbered according to the amount of columns selected. Each of them carries all of the selected column types as generic type variables. This type variables can be used to generate tupled result or use the query as a subquery in a *from* clause.

Finally `SelectBuilder3.find()` constructs the SQL query, executes it and returns the result:

```
public class SelectBuilder3<T extends Table,C1,C2,C3>
  extends FromBuilder<T> {
  ...
  public List<Tuple3<C1,C2,C3>> find() { ... }
}
```

Note that since our builders carry the type of the table passed in `from()` and that the `FromBuilder` only accepts the columns belonging to the same type. This provides additional safety as the programmer cannot select fields from a table that was not written in from.

However this solution is hard to extend when there is more than one table in the *from* clause. Although we could apply the same idiom and tuple all the builders over the *from* table types, but it's hard to later use this types in the actual methods. The problem is that to check the type we need the programmer to explicitly set the index of the table type in the from clause (e.g. by writing `select1()`, `select2()`, ...). Since this is uncomfortable and is influenced by changes in *from* clause we decided to leave this check out altogether and the builders do not carry the table types at all.

2.4 Expressions

Now that we have the basic structure of the SQL queries set we need to encode arbitrary functions, aggregates and expressions. In a usual fluent interface they would be accessible using the same chained notation we used for building the query. However we chose instead to use static methods, imported to the local namespace using the `import static` feature introduced in Java 5.

As far as we care a general SQL expression can be expressed with the following interface:

```
public interface Expression<E> {
  String getSqlString();
  List<Object> getSqlArguments();
  Class<E> getType();
}
```

The `E` type variable is the type of the value that the expression produces on evaluation. The corresponding class is returned by the `getType()` method. Finally `getSqlString()` returns the corresponding SQL fragment and `getSqlArguments()` returns the arguments to be inserted instead of "?" in the query.

In *where* clause we only permit to use expressions of type `Expression<Boolean>`[3] such as "like", "<", "=", etc. The operands of these expressions can already be arbitrary. To create these expressions we could use the following API:

```
public class ExpressionUtil {
  public static <E> Expression<E>
    constant(E value);

  public static <E> Expression<Boolean>
    eq(Expression<E> e1, Expression<E> e2);

  public static <E> Expression<Boolean>
    gt(Expression<E> e1, Expression<E> e2);

  public static <E> Expression<Boolean>
    lt(Expression<E> e1, Expression<E> e2);

  public static <E> Expression<Boolean>
    like(Expression<E> e, Expression<String> pattern);

  public static Expression<Boolean>
    not(Expression<Boolean> e);

  public static Expression<Boolean>
    and(Expression<Boolean>... e);

  public static Expression<Boolean>
    or(Expression<Boolean>... e);
}
```

The `constant()` method returns an expression corresponding to the value. Since we can overload all of the methods to also accept basic values and just call `constant()` for them we will not call it explicitly in the upcoming examples.

Now that we introduced the `Expression` type we can finally define the `Column` type we used to encode column metadata:

```
public class Column<T extends Table, C>
  implements Expression<C> {
  ...
  public Class<C>
    getType() { return type; }
  public String
    getSqlString() { return name; }
  public List<Object>
    getSqlArguments() { return null; }
}
```

It is as straightforward as the constant expression and just returns the name of the column as the SQL fragment.

We can now easily encode the expression `WHERE name = 'Peter' or height > 170`:

```
List<Tuple3<String, Integer, Date>> persons =
  new QueryBuilder(datasource)
    .from(Person.TABLE)
    .where(or(
      eq(Person.NAME, "Peter"),
      gt(Person.HEIGHT, 170))
    )
    .select(Person.NAME, Person.HEIGHT, Person.BIRTHDAY)
    .find();
```

So far we have only allowed to select columns from the table. In general we want to select an arbitrary expression (such as `name || ', ' || birthday`). Therefore the `FromBuilder` class should just accept `Expressions` instead of `Columns`:

```
public class FromBuilder extends QueryBuilder {
  ...
  public <C1> SelectBuilder1<C1>
    select(Expression<C1> c1);
  public <C1, C2> SelectBuilder2<C1, C2>
```

[3] Since we would like to permit arbitrary many such expressions to be passed to methods the "varargs" feature introduced in Java 5 is used. However since array component types cannot be generic we have to introduce the `BooleanExpression extends Expression<Boolean>` which complicates things a bit. We ignore this complication in the examples.

```
    select(Expression<C1> c1, Expression<C2> c2);
  public <C1, C2, C3> SelectBuilder3<C1, C2, C3>
    select(
      Expression<C1> c1,
      Expression<C2> c2,
      Expression<C3> c3);
  ...
}
```

2.5 Aliases

TODO: Table aliases are simple and boring. I wanted here described the expression aliases that can be used to introduce expressions into both select and where clauses. E.g. "SELECT concat(first_name, " ", last_name) as full_name WHERE full_name != "Jevgeni Kabanov"". This would look like

```
Person p = new Person();
Alias<String> fullName =
  concat(p.firstName, " ", p.last_name);
new QueryBuilder(datasource)
  .from(p)
  .where(not(eq(fullName, "Jevgeni Kabanov")))
  .select(fullName);
```

TODO: Everything else concerning tables in this section was obvious, this example is enough:

```
Person person = new Person();
Person father = new Person();

List<Tuple2<String, String>> names =
  new QueryBuilder(datasource)
    .from(person, father)
    .where(eq(person.fatherId, father.id))
    .select(person.name, father.name)
    .find();
```

2.6 Control Flow and Reuse

TODO: Control flow discussion is missing, closures introduced out of the blue.

TODO: Appenders are actually very interesting, but may be too long to discuss here. We'll see after the rest of the content is in.

The general closure can be used to alter tables and where conditions:

```
public interface Closure {
  void apply(Builder builder);
}
```

Each select builder (with certain number of columns) has also a corresponding closure that is able to add and remove columns:

```
public class SelectBuilderC2<C1,C2>
  extends SelectBuilder {
  ...
  public SelectBuilderC2<C1,C2>
    closure(Closure closure) {

    closure.apply(this);
    return this;
  }
}
```

The following example illustrates how to abstract adding a person's father name to the select.

```
final Person person = new Person("p");

new QueryBuilder(datasource)
  .from(person)
  .select(person.name)
  .closure(new ClosureC1<String>() {
    public void apply(Builder builder) {
      Person father = new Person("f");
      builder
        .addTables(father)
        .addColumn(father.name)
        .addConditions(
          eq(person.fatherId, father.id));
    }
  });
```

Here we use a closure to add another table, column and where condition to the select. Since we change the number of columns we have to either get the results from the builder in our closure or pass it to another one. Here we refer another closure that gets the results and prints them out.

2.7 The Rest Of SQL

TODO: Discuss briefly ORDER BY, GROUP BY, extensions (RDBMS-specific), untyped expression, functions and procedures and others. Reader must understand that implementing full SQL is only a matter of time. Don't go into too much detail.

3. TYPED BYTECODE ENGINEERING

Java bytecode is a relatively simple stack-based language. All the code is contained in methods, class structure is mostly preserved from source (fields and methods, both can be static, constructors and static initialisers are turned into special methods named `<init>` and `<clinit>` correspondingly).

Inside the methods we have the variables, referred by an index with 0 being *this*, 1 being the first parameter and so on. Local variable indexes start after parameters. We can *load* and *store* variables. Every method has its own stack, where we can *push*, *pop* and *duplicate* values. We have a number of basic operations on the stack (like *add* and *multiply*) as well as *method invocation*. When invoking the methods parameters are gathered from the stack with last parameter being on top of the stack. Finally we have some flow control, namely conditional (and unconditional) *jumps*. For more information see Java Virtual Machine Specification [9].

One of the best libraries for working with Java bytecode is ASM [4]. It provides both a lightweight visitor-based interface and a more comfortable tree-based object-oriented interface. Unfortunately both of them (and especially visitor-based one) are completely untyped and the produced bytecode is only verified during runtime[4].

We are going to use this simple Java example in the rest of this section:

[4]Even that verification is quite unsatisfactory, since the JVM verifier will produce an error, but will not specify the exact place where it occurs.

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

When we compile it and then dump the bytecode this example becomes a bit more complicated. The following code is dumped by running `javap -c HelloWorld`:

```
public class HelloWorld {
  public <init>()V
    ALOAD 0
    INVOKESPECIAL Object.<init>()V
    RETURN

  public static main([LString;)V
    GETSTATIC System.out : LPrintStream;
    LDC "Hello, World!"
    INVOKEVIRTUAL PrintStream.println(LString;)V
    RETURN
}
```

As we can see the Java compiler has generated a default constructor for the class. Additionally all classes are referred by their full names[5]. The instructions we see do the following:

- `ALOAD` loads the local variables to the top of the stack, index 0 is *this*.
- `INVOKESPECIAL` in this case invokes super method consuming an `Object` from the stack.
- `GETSTATIC` retrieves the value of the static fields and puts it on the stack (in this case a `PrintStream`).
- `LDC` pushes a constant value to the stack.
- `INVOKEVIRTUAL` invokes a usual (virtual) method, consuming the parameters from the stack and pushing the result to the stack.
- `RETURN` exits from the method

We are not going to examine ASM library in detail, suffice to say its API corresponds almost one-to-one to Java bytecode instructions.

3.1 Typesafe DSL

Let's ignore the default constructor for a second and concentrate on the `main()` method. The typesafe DSL that we propose would engineer this bytecode like this:

```
new ClassBuilder(
  cw, V1_4, ACC_PUBLIC, "HelloWorld", "Object", null)
  .beginStaticMethod(
    ACC_PUBLIC | ACC_STATIC,
    "main", void.class, String[].class)
  .getStatic(System.class, "out", PrintStream.class)
  .push("Hello, World!")
  .invokeVirtualVoid(
    INVOKEVIRTUAL,
    PrintStream.class, "println", String.class)
  .returnVoid()
  .endMethod();
```

[5]In this example and further we omit the package from the class name for brevity

What do we mean under typesafe in this case? Well in addition to most parameters being passed as class literals and the instructions syntax being part of the API we also track stack and local variable types. For example if we exchange the `push("Hello, World")!` to `push(10)` the compiler will give the following error:

```
invokeVirtualVoid(..., Class<? super Integer>)
in MethodBuilderS2V1<PrintStream, Integer, String[]>
is not applicable to (..., Class<String>)
```

The error means that our DSL tracks the types of the stack slots and since the method expects a `String` parameter or compatible and the stack contains an `Integer` compiler produces an error.

3.2 Tracking Stack and Local Variables

To achieve type safety we need to track stack size and types. We wouldn't want to allow to `pop()` off an empty stack. And we would want to ensure that when you invoke a method all the parameters are there.

Since there is a different set of operations allowed for different stack sizes it is natural to have a class for each stack size. We choose to name (and number them) `MethodBuilderS0`, `MethodBuilderS1`, `MethodBuilderS2` and so on.Each of them has only the methods possible with the current stack size. Now the method for `pop()` will not even show up in autocompletion if the stack is not large enough.

We can apply the same idea to tracking variables, allowing to add only one variable at a time and providing methods `loadVar*` and `stroreVar*`. This way our class name becomes `MethodBuilderS*V*` where S stands for stack size and V stands for variable count.

Of course in addition to the sizes we also have to track the actual types. To do that each `MethodBuilder` is parametrized by N type variables where $N = S + V$. The `MethodBuilderS2V1<PrintStream, Integer, String[]>` we saw previously in the compiler output means that two stack slots have types `PrintStream` and `Integer`, whereas the only local variable has type `String[]`.

To understand how the types are inferred let's see the implementation of `push()` and `pop()`:

```
public class MethodBuilderS2V1 <S0, S1, V0> {
  public MethodBuilderS1V1<S0, V0> pop() {
    ...
    return
      new MethodBuilderS1V1<S0, V0>(cb, mv);
  }

  public <S> MethodBuilderS3V1<S0, S1, S, V0>
    push(S value) {
    ...
    return
      new MethodBuilderS3V1<S0, S1, S, V0>(cb, mv);
  }
```

The class in question, `MethodBuilderS2V1` is parametrised by two stack types and one variable types. The method `pop()` throws away the top stack type and returns an instance of `MethodBuilderS1V1`. The method push on the other hand infers an addition type from the parameter passed to it and returns an instance of `MethodBuilderS3V1` adding the inferred type to the top of the stack types.

What does the `invokeVirtualVoid()` look like? First of all, it needs to consume two stack variables, therefore we need at least `MethodBuilderS2V*` class to call it. Therefore all classes with less stack variables will not have this method. Secondly we need to check that the types in the stack are fitting, but must allow some leniency due to subtyping:

```
public class MethodBuilderS2V1<S0, S1, V0> {
  public MethodBuilderS0V1<V0>
    invokeVirtualVoid(
      int kind,
      Class< ? super S0> owner,
      String name,
      Class< ? super S1> parameter1) {
    ...
    return new MethodBuilderS0V1<V0>(cb, mv);
  }
}
```

As you can see it indeed consumes two stack values returning `MethodBuilderS0V1`. If our method would also return a result we would need to pass the result type as well, and return a class with stack depth only one less. The expression `? super S0` means that we require the actual parameter type to be a superclass of the stack type, which provides the required leniency.

3.3 Unsafe Assumptions

One of the problems with the DSL we proposed here is that it assumes all the types exist. However it is very often the case that some of the types (most prominently the class currently being created) do not. You can somewhat alleviate the problem by introducing a special placeholder `Self` type and use it instead of the current class name, but it won't solve the problem of other classes still awaiting construction.

A different (but connected) problem is that you are not always constructing the full method, instead you could be just creating a prelude for a particular method or replacing one instruction with a series of your own. To solve this we should allow escaping from the rigid typesafe world by having unsafe operations, which issue compiler warnings.

This means that instead of missing `pop()` from a type with no stack slots we should just deprecate it or otherwise issue a warning. This also means that we should have `invoke*()` methods that take strings as parameter types, similarly deprecated.

However, since we know it's not a perfect world we'd like to at least protect ourselves a bit better. Therefore we should allow users to document their assumptions.

This means that when we need to write just a fragment of bytecode we want to document what stack values and variables will it need. For that we introduce methods `assumePush()`/`assumePop()` and `assumeVar*()`, which do not push any values, but just add the corresponding type variables:

```
public class MethodBuilderS2V1<S0, S1, V0> {
  public <S> MethodBuilderS3V1<S0, S1, S, V0>
      assumePush(Class<S> type) {
    return
      new MethodBuilderS3V1<S0, S1, S, V0>(cb, mv);
  }

  public MethodBuilderS1V1<S0, V0> assumePop() {
    return new MethodBuilderS1V1<S0, V0>(cb, mv);
  }

  public <V> MethodBuilderS2V2<S0, S1, V0, V>
      assumeVar1(Class<V> type) {
    return
      new MethodBuilderS2V2<S0, S1, V0, V>(cb, mv);
  }
}
```

Using them we can document our expectations and let the compiler validate them. The following is an example of how we can use the assumptions to document that we expect `PrintStream` to be on stack and `String[]` to be the variable 0.

```
private static void
    genSayHello(MethodBuilderS0V0 mb) {
  mb.assumeVar0(String[].class)
  .assumePush(PrintStream.class)
  .loadVar0(String[].class)
  .push(0)
  .arrayLoad(
    String[].class,
    Integer.class,
    String.class)
  .invokeVirtualVoid(
    INVOKEVIRTUAL,
    PrintStream.class,
    "println",
    String.class);
}
```

3.4 Control Flow and Reuse

The next problem is how to mix the DSL with the general-purpose control flow and method calls. The problem here is that (although it is hidden from the user) we return a different type every time. Therefore if we add a conditional operation, we would need to save the type in a variable, which would be complicated, as it contains a lot of inferred types. It also wouldn't cope with changes, since the inferred types would change every time.

To solve this we introduce two types of closures – the strongly typed and weakly typed. The weakly typed ones lose all accumulated type information and are meant first of all for reusable methods that expect a base type as the argument:

```
public interface Closure {
  public void apply(MethodBuilderS0V0 mb);
}
```

The strongly typed closures are generated with the class and are meant for using in control flow. Notice that although the method parameter includes all of the type variables they are undefined and will not change with the stack:

```
public class MethodBuilderS2V1 <S0, S1, V0> {
  public MethodBuilderS2V1<S0, S1, V0>
      closure(ClosureS2V1 closure) {
    closure.apply(this);
    return this;
  }
```

```
  public MethodBuilderS2V1<S0, S1, V0>
      closure(Closure closure) {
    closure.apply(new MethodBuilderS0V0(cb, mv));
    return this;
  }

  interface ClosureS2V1 {
    <S0, S1, V0> void
      apply(MethodBuilderS2V1<S0, S1, V0> mb);
  }
}
```

We illustrate the weakly typed closures by calling the method `genSayHello()` introduced previously:

```
.beginStaticMethod(
  ACC_PUBLIC | ACC_STATIC,
  "main", void.class, String[].class)
.getStatic(System.class, "out", PrintStream.class)
.closure(new Closure() {
  public void apply(MethodBuilderS0V0 mb) {
    genSayHello(mb);
  }
})
.returnVoid()
.endMethod();
```

Of course with the introduction of actual closures in Java 7 this would look much shorter.

4. PATTERNS AND DISCUSSION

Let us now take a step back and look at the patterns showing up in the design of the two DSLs we have introduced. We will try to formulate a one-sentence summary for each of the patterns that we have identified and then follow it with examples and informal discussion. Although we could have put them down in a more formal way as in Design Patterns [7], we find that the issues are too broad and the DSL design too much of an art to assume such formality.

Note that as previously in text we refer to the classes that implement the DSL API as *builders*.

4.1 Type Shaping

> *All of the relevant information about the current building state should be encoded in the builder type and the builder should have precisely the methods allowed in that state.*

We saw several examples of this pattern at work

- SQL query builders allowed *from*, *where* and *select* to be called once and only once.
- Select builders types encode information about selected columns.
- Select builders return only tuples that have precisely the selected data with types known ahead.
- Bytecode builders encode information about stack slot and local variable types.
- Bytecode builders hide methods that consume more stack slots than is available.
- Bytecode builder methods that consume stack slots must consume types fitting to the ones currently on stack.

This pattern is the basis of a properly designed typesafe DSL as instead of disallowing wrong behaviour we only allow *precisely* right behaviour to begin with.

In both our examples the builder types encoded a list of types, which made it necessary to declare types with same (or similar) methods but different number of encoded types. We conjecture that it is a common enough pattern if you are encoding a complex enough DSL. However there are some ways around it, which we will review shortly.

4.2 Typesafe Metadata

> *All of the metadata used by your DSL should include compile-time type information.*

Both our DSLs made use of typesafe metadata, some of those uses were less obvious than others:

- SQL made use of pregenerated metadata dictionary that contained type information about database objects including tables and columns.
- The SQL expressions and named aliases provided information about the type of expression. This metadata was accessible via the local variable that the expression was saved to.
- The SQL `SelectBuilder` encoded metadata about its column types, which would be important if we wanted to use it as a subquery again saved in a variable.
- The bytecode DSL used class literals, which are typesafe metadata embedded in Java language.
- We could have required each bytecode DSL local variable to be declared separately as a `Variable<T>` class that would encode the type of the actual variable and could be saved locally.

As you can see the metadata needs to be typesafe, but it can still be come from a runtime expression saved to a local variable. This allows a lot of flexibility beyond the predefined dictionary-style metadata.

4.3 Unsafe Assumptions

> *Allow the user to do type unsafe actions, but make sure he has to document his assumptions.*

TODO

4.4 Call Chaining and Static Imports

TODO

4.5 Closures

TODO

5. TYPE SAFETY LIMITATIONS

TODO

6. ACKNOWLEDGEMENTS

TODO

7. CONCLUSIONS

TODO

8. REFERENCES

[1] C. Bauer and G. King. *Hibernate in action*. Manning, 2005.

[2] J. Bentley and J. Bentley. *Programming Pearls*. Addison-Wesley Professional, 1999.

[3] B. Bringert, A. Höckersten, C. Andersson, M. Andersson, M. Bergman, V. Blomqvist, and T. Martin. Student paper: HaskellDB improved. *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 108–115, 2004.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems*, 2002.

[5] J. Cuadrado and J. Molina. Building Domain-Specific Languages for Model-Driven Development. *Software, IEEE*, 24(5):48–55, 2007.

[6] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. jMock: supporting responsibility-based design with mock objects. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 4–5, 2004.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[8] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, page 196, 1996.

[9] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[10] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.