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

Projects like jMock and Hibernate Criteria Query introduced embedded DSLs into Java. We describe two case studies in which we develop embedded typesafe DSLs for building SQL queries and engineering Java bytecode. We proceed to extract several patterns useful for developing typesafe DSLs for arbitrary domains. Unlike most previous Java DSLs we find that mixing the Fluent Interface idiom with static functions, metadata and closures provides for a better user experience than pure method chaining. We also make very liberal use of the Java 5 Generics to improve the type safety properties of the DSLs.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Reliability, Languages, Verification

Keywords

Java, domain-specific, DSL, typesafe

1. INTRODUCTION

Domain specific language usually refers to a small sublanguage that has very low overhead when expressing domain specific data and behaviour. DSL is a broad term [12, 3] and can refer both to a fully implemented language and a specialised API that looks like a sublanguage [10], but still written using some general-purpose language. Such DSLs in the latter meaning have been introduced by both the functional [4] and dynamic language communities [6]. Both these communities (especially functional) took advantage of function composition and operator overloading to build combinator-based languages that look nothing like the host one. The functional community also strongly supports the notion of type safety; therefore DSLs they create are usually statically typed.

The main motivation for using DSLs (whether embedded or external) is threefold. First of all, the key feature of DSLs is encoding domain-specific data and behaviour with low overhead. This means that the code is both easier to comprehend and easier to maintain. Secondly, thanks to the low overhead the DSL text should also be understandable by the domain expert. This makes it easier to collaborate with the expert on encoding the domain-specific logic. Finally, with embedded DSLs you can make use of the compiler advanced features to ensure type safety on the level of DSL constructs, thus eliminating certain types of errors already during compilation.

There is some amount of discussion of using embedded v/s external DSLs. The obvious pros of the former is reusing the platform tooling, which in Java case includes compilers, advanced IDEs, debuggers, profilers and so on. Also embedded DSLs are considerably easier to design and develop, as it boils down to writing an API and using some of the more advanced language features. On the other hand the external DSLs boast better availability to the domain experts, often making it possible for them to interact directly with the DSL text. Additionally, once the compiler or interpreter is implemented it can manipulate the language constructs directly and may provide extra guarantees not possible in a general-purpose setting. The particular choice depends strongly on the domain in question, but we feel that the advanced tools available in the Java ecosystem makes a very strong argument for preferring embedded DSLs when possible.

In Java community the DSLs are becoming increasingly popular. Unfortunately published work in the area is very rare and most of the innovation is done in an ad hoc way by various members of the Java community. Almost the only paper in the area was published by Freeman et al [8] and describes the lessons learnt from designing jMock embedded DSL. Another example is the Hibernate Criteria [2]. Those and some folklore examples introduced a technique for writing embedded DSLs using method call chaining that was coined Fluent Interface by Martin Fowler [7].

In this paper we show that although Fluent Interface is a powerful concept it is not fitting in all contexts. We propose to mix it with static functions, metadata and closures to make full use of Java language capabilities. We also propose to make use of Java Generics to make the embedded DSLs constructs typesafe. We test our proposals on two case studies—embedded DSLs for manipulating SQL queries and

PPPJ 2008, September 9–11, 2008, Modena, Italy.

engineering Java bytecode.

The rest of the paper is organised as follows. Section 2 introduces the *Typesafe SQL* DSL and studies how it implements different aspects of SQL. Section 3 introduces the *Typesafe Bytecode Engineering* DSL and studies how stack and variables can be encoded in a typesafe manner. Section 4 studies and discusses the generic patterns introduced in the case studies. Sections 5 and 6 conclude the paper and discuss some possible directions for further work.

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 " +
  "FORM person" +
  "WHERE heigth >= " + 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 in this simple example, we made a few mistakes:

- We misspelled an SQL command.
- We misspelled a column name.
- We forgot to add a space before "WHERE".
- We could be mistaken about the column type, it could be string in the database.
- 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 embed 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>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .where(gt(p.height, 170))
    .select(p.name, p.height, p.birthday)
    .list();
for (Tuple3<String, Integer, Date> row : rows) {
  String name = row.v1;
  Integer height = row.v2;
  Date birthday = row.v3;
  System.out.println(
    name + " " + height + " " + birthday);
}
```

Unlike before in this example any kind of misspelling or type inconsistency will show up immediately as a compile-time error[1].

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;
  }
}
```

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

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 column 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]How the dictionary is generated is not relevant to how it can be used and thus is not covered. We assume that some translator exists that converts the table and column information in the database descriptors to Java classes.

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 make sure that the basic syntax of the query is always correct since mistakes result in 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)
    .list();
```

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 extends Builder {
  ...
  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 Builder {
  ...
  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. These type variables can be used to generate tupled result or use the query as a subquery in a *from* clause.

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

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

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 columns 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. We could apply the same idiom and tuple all the builders over the *from* table types, but to actually check the type we need the methods to be indexed by the table type indexes (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 in the actual implementation.

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 with the same chained notation we used for building the query. However we chose instead to use static methods, imported to the local namespace with the `import static` feature introduced in Java 5.

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 Expression<Boolean>
```

[3]We would like to use the "varargs" feature introduced in Java 5 with this expressions to allow arbitrary many of them. 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.

```
    like(Expression<?> e,
      Expression<String> pattern);

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

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

The `constant()` method returns an expression that returns "?" as the SQL string and value as the SQL argument. 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`:

```
  Person p = new Person();

  List<Tuple3<String, Integer, Date>> persons =
    new QueryBuilder(datasource)
      .from(p)
      .where(or(
          eq(p.name, "Peter"),
          gt(p.height, 170))
        )
      .select(p.name, p.height, p.birthday)
      .list();
```

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

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

Sometimes we need to have aliased SQL subexpressions that can be used in both *select* and *where* clauses. For this we introduce an `Alias` class that can be used to hold such expressions in variables or fields. E.g. `SELECT concat(first_name, " ", last_name) as full_name WHERE full_name != "Peter Griffin"` would look like:

```
  Person p = new Person();
  Alias<String> fullName =
    alias(concat(p.firstName, " ", p.lastName));

  List<String> names = new QueryBuilder(datasource)
    .from(p)
    .where(not(eq(fullName, "Peter Griffin")))
    .select(fullName)
    .list();
```

The `Alias` class differs from `Expression` by having a `getAliasExpression()` method that returns the original expression appended by `AS (new name)`.

```
public interface Alias<E> extends Expression<E> {
  String getName();
  Expression<E> getAliasExpression();
}
```

Another use of SQL aliases is to use same table more than once in a query. To do that we just create separate instances of the table class:

```
  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)
      .list();
```

2.6 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. To solve this we encourage to use closures for the control flow:

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

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

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

So to find persons by name (and all persons if the name is `null`) we can use the following syntax:

```
Person p = new Person();

List<Tuple2<Integer, String>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .closure(new Closure() {
      public void apply(Builder builder) {
        if (searchName != null) {
          builder.addConditions(
            eq(p.name, searchName));
        }
      }
    })
    .select(p.id, p.name)
    .list();
```

Such an alternative API also allows us to modify the query in reusable methods that can be called using closures.

2.7 The Rest Of SQL

Other SQL features such as *order by*, *group by*, *having* and so on can be implemented similarly:

```
Person p = new Person();
Alias<Integer> count = alias(count(p.id));

List<Tuple2<Integer, Integer>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .where(not(eq(p.name, constant("Peter"))))
    .groupBy(p.fatherId)
    .having(gt(count, constant(3)))
    .orderBy(desc(count))
    .select(p.fatherId, count)
    .list();
```

For a native SQL fragment one can include an untyped expression. It takes the corresponding SQL string, arguments and the expression type:

```
Person p = new Person();

List<Tuple2<String, Integer>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .select(p.name,
      unchecked(Integer.class,
        "util.count_children(id)"))
    .list();
```

Although not covered in this article Aasaru has implemented full SQL select, update, insert and delete query support as a part of his master's thesis [1].

3. TYPESAFE 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 [11].

One of the best libraries for working with Java bytecode is ASM [5]. 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:

```
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.

[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.

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

- `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(
    PrintStream.class, "println", String.class)
  .returnVoid()
  .endMethod();
```

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 parametrised 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()` discards the top stack slot type and returns an instance of `MethodBuilderS1V1`. The method `push()` on the other hand infers an addition type from the argument and returns an instance of `MethodBuilderS3V1` adding the inferred type to the top of the stack slot 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(
      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

Similar as in SQL case we introduce closures to deal with control flow and reuse. The problem is aggravated in this case as we return a different type every time a method is called. We introduce a closure type as before:

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

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

We illustrate using 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 [9], 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 Restricting Syntax

> *At any moment of time the DSL builder should have precisely the methods allowed in the current 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.
- Bytecode builders hide methods that consume more stack slots than is available.

This pattern was discussed in a different setting in [8]. They used the interfaces to encode the syntax of the DSL while still implementing all of the interfaces by a few classes. In our case such an approach would work in the SQL case study language, but not in the bytecode engineering case study language, as we would still need method with different return types depending on the amount of tracked stack slots and local variables.

This is the first principle for building a typesafe DSL as it allows to encode the base syntax in a typesafe manner.

4.2 Type History

You can accumulate type history as a type list and use it to reject actions that do not fit with that history

We saw several examples of this pattern at work

- Select builder types encode information about selected columns.
- Bytecode builders encode information about stack slot and local variable types.
- Bytecode builder methods that consume stack slots must consume types fitting to the ones currently on stack.
- Bytecode builder local variable methods infer types from the values to variables and from variables to stack.

Type history is an important concept as it allows to reject actions based not on the types in the current method call, but also in the previous method calls. We conjecture that in any language that either uses stack or a stack-like environment this should be the best approach to use.

4.3 Typesafe Metadata

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 is basically typesafe metadata embedded in Java language.

There are two type of typesafe metadata we can separate: metadata dictionary and runtime metadata.

Metadata dictionary is either generated or otherwise available in a static manner. This is the main building block of the DSLs that need to act on many domain entities that are not specified ahead of time.

However we also need to extend the dictionary and that is where the runtime metadata comes in. It is encoded as a runtime expression with a static type. The type must encode all the information that we need. This way we can introduce local variables referring to such expressions on demand extending the metadata dictionary while still retaining the type safety property

4.4 Unsafe Assumptions

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

We saw two examples of this pattern:

- In SQL the unchecked expressions still had to declare the expected type of the expression.
- The bytecode DSL allowed ignoring the type history by providing dedicated methods for type unsafe actions.
- In bytecode DSL we saw `assume*` methods dedicated to documenting type assumptions.

Both parts of this pattern are very important. If you don't allow type unsafe actions it is likely that the users will find it limiting at some point and revert to bypassing your DSL. However using two APIs instead of one makes the code harder to read and maintain.

If you don't allow users to document their assumptions you risk errors every time you do something unsafely. Assumptions allow one always to hold the information about types in a local manner; even though it does not guarantee full type safety, it still helps to develop safer code.

4.5 Hierarchical Expressions

Use method chaining when you need context and static functions when you need hierarchy and extensibility.

One of the main problems with method chaining is very poor support for hierarchy. We could have also structured the SQL like this:

```
//...
.where
  .and()
    .eq(p.id, 3)
    .gt(p.age, 17)
  .endAnd()
//...
```

Unfortunately this lacks any inherent structure and the first time we would auto-format the source code we would lose all hints of hierarchy in the code structure.

The other problem with method chaining is extensibility. In the case we just shown `and()` has to return a builder that includes all of the SQL expressions (including the `and()` itself). The problem is that different databases employ very different sets of allowed expressions, which makes it impossible to just encode them all into the builder syntax.

The obvious solution in this case is using static functions (and `import static` to bring them into the local namespace). However static functions suffer from a different problem—unlike chained methods they cannot access the context. The context includes all the state that was accumulated by the previous chained method calls as well as type history.

On the other hand static functions can be added on demand, whereas builders are much harder to extend. In the end it is likely that you will need both for different parts of your language. In the case of SQL we chose to make the statement-like part of the syntax chained and the expression-like part of the system functional.

4.6 Closures

Use closures to escape the method chaining for control flow and reuse.

We introduced closures in both DSLs that we reviewed. They allow for a flexible way to mix your DSL with the control flow of the host language. Of course if Java would have support for true closures it could be made more concise, but even with the current syntax it is quite usable.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have introduced two different domain specific languages, trying to use the most of Java type system to inhibit wrong behaviour at compile-time. Although the languages were from two completely different domains we managed to identify a number of generic patterns, which we conjecture to be domain-independent and which will likely ease the work of future DSL designers.

Unlike most previous Java DSLs we found that mixing the Fluent Interface idiom with static functions, metadata and closures provides a better experience to the users than pure method chaining. This also provides for easier extension of the DSL and easier reuse of the DSL fragments.

We made very liberal use of the Java 5 Generics to improve the type safety properties of the DSLs. We managed to secure both case study DSLs against almost all type mistakes a programmer can make. To do that we introduced type lists that allowed us to verify chained method calls against the history of calls. We also found that it is essential to allow user to both escape the type safety limits and document the assumptions so that the rest of the DSL fragment could still be typesafe.

We plan to continue investigating the limits of typesafe embedded DSLs in Java foremost by implementing new case studies and extracting patterns and idioms of less generic manner. One particular development we plan to pursue at the moment is a context-aware DSL for building hierarchical data (like XML) in Java.

6. ACKNOWLEDGEMENTS

We would like to thank Juhan Aasaru, who developed the SQL DSL as a part of his master's thesis [1]. He did a great job and contributed a lot of ideas that influenced this article.

We would also like to thank the readers of the dow.ngra.de blog that pointed out some of the problems with our initial prototypes.

This work was partially supported by Estonian Science Foundation grant No. 6713.

7. REFERENCES

[1] J. Aasaru. Typesafe DSL for Relational Data Manipulation in Java. Master's thesis, University of Tartu, 2008.

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

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

[4] 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.

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

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

[7] M. Fowler and E. Evans. FluentInterface at `http://www.martinfowler.com/bliki/FluentInterface.html`, 2005.

[8] 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.

[9] 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.

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

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

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