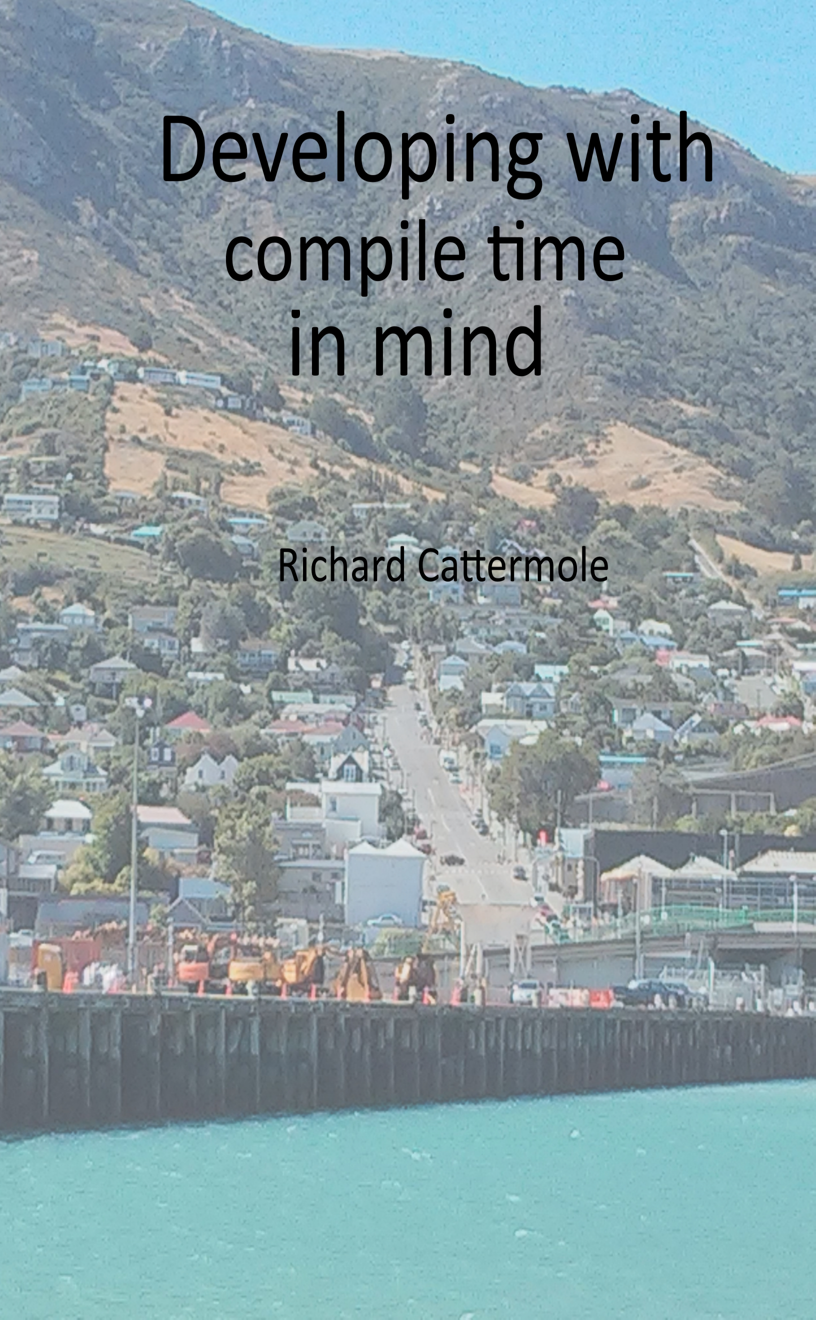


Developing with compile time in mind

Richard Cattermole

Developing with compile time in mind

richard cattermole

This book is for sale at
http://leanpub.com/ctfe

This version was published on 2015-10-12

Leanpub

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help richard cattermole by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ThinkCTFE.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#ThinkCTFE

Also By richard cattermole

The way to program

Contents

Preface

To me this book represents hope. Hope for the future learning this book will provide. Exploring of new language designs and how it can affect the real world usage.

As the author, I've felt the research and resource into CTFE is just lacking. I discovered this during 2014. I was writing a research paper for my degree all about how D could benefit business and using CTFE as an example. All the while only finding a couple research papers in total that used CTFE in any form. In almost all the cases they were not using it to the lengths possible in the D programming language.

Because of the striking lack of information and some free time, I started work on this book to remedy based upon what I had learned by creating a range of libraries which utilized CTFE heavily.

Introduction

Developing software is an amazing process that can take many different forms. In the case of this book we look towards a little known and used concept. Compile Time Function Execution or Evaluation for those inclined, is the ability to execute code during the compilation phase of a program. This opens up many different avenues for automation of code generation. But it is a double edged sward. The time it takes during compilation is heavily dependent upon what the compiler must do.

When reading this book, have in mind that very few languages actually support CTFE currently. Those that do generally do not support it to the extent that is expected for this book. Because of this and the experiance with the D programming language. It has been chosen by the author to use D as the base language refered to within the book.

Exploring language support

Language support for CTFE is rather varied. From the most utter basic of constant expansion such as parsing of a number literal. To the more complex of having a GUI toolkit running.

Because of how varied CTFE support can be, it must be classified. The LISP family languages are a great starting point in everything from simple to complex designs. The D programing language on the other hand, can be more familiar because of its C family origination. Unlike the C/C++ where the macro preprocessor was used, D does not have this. However in some ways the macro pre-processor in C/C++ could be considered a form of CTFE.

The D programming language

To fully grasp the usage of CTFE in a codebase. Let's start off with a language that has a semi decent support. For this the D programming language will be used. D has good CTFE support but it does have limitations which are intentional.

D has meta-programming support which has been described as compile time arguments. A particularly unique language feature D has is known as a mixin template. A mixin template is a template that instead of creates a new type, can be effectively be thought of as output AST in a given context[1].

A mixin template should not be confused with a normal template. Where a normal template effectively creates a new symbol uniquely to the arguments given[2].

[1] http://dlang.org/template-mixin.html

[2] http://dlang.org/template.html

To further make use of CTFE code, it can be useful to take input in some form or another. That does not rely on hard coding into the file the values. There are two known ways for this, string imports and using a tool such as Bin2D[ˆBin2DGithub].

String imports have the syntax of `enum string thefile = import("file");`. The search path for the file must be provided to the compiler via the `-J` flag or for dub via `stringImportPaths` property.

Bin2D is a very useful tool in that it can generate a file that contains many directories or files as byte arrays. Alternatively it can also export at runtime on request. However because of it being an external tool it does require a preprocess action which can be slower.

Developing CTFE'able code

To develop D code compatible with compile time, there is one main restriction. All required information to execute must be passed in. There is no global data in any form including static variables within a function.

As a result of this restriction the `pure` attribute is in heavy use by the author in his code bases. The `pure` attribute reflects the same restrictions that CTFE comprises of. No access to global data. Most of the time it would also be wise to add `@safe`. Where by removing the direct use of pointers.

An exception to the rule of no global data is constants. A constant such as enum is well known by the compiler, as such it can be utilized. However the opposite of this rule is no external code such as extern(C) functions. An example of a CTFE'able function would be factorial.

Factorial function callable at compile time.

```
1  size_t factorial(size_t n) pure {
2          assert(n < 11);
3
4          if (n == 0)
5                  return 1;
6          else
7                  return n * factorial(n - 1);
8  }
9
10 static assert(factorial(5) == 120);
```

As shown by the code sample for a factorial callable at compile time it must execute and output the value of 120 during compilation. We know that it must by the static assert statement on line 10.

The pure attribute is listed on the right of the statement declaration before the opening bracket. For those unfamiliar with D this has the same effect as it being of the left. However contested. Some consider it good practice for attributes on the right to refer to the function and the left return type. Further, size_t is used as the data type. This is an unsigned integer dependent upon the word size of the resulting binary. Such as for 32bit, uint.

During optimised targets (`-release`) assert at compile time (not static version) should also be assumed to work and work like a static assert.

The declaration of `size_t` is an alias within two version statements. One for x86 the other for x86_64. Lastly, there is two different types of asserts in D. First the normal runtime based assert and secondly the compile time static assert. The static assert is a declaration at compile time that must be true to compile. Whereas during a debug build an assert will throw an exception if it is not true. This is useful for contract based programming.

However this is not the only type of static statement, there is another: Static if! Static if is just like a regularly if statement except it can conditionally include code based upon the statement at runtime. This is comparable to the #if or #ifdef macro in C/C++. However do note, this is part of the language and not part of a preprocessor like in C/C++ [3].

Going forward it is necessary to discern the difference between runnable at runtime and not. Not all code that is written with CTFE in mind should be ever ran during runtime. This produces three groups of code. 1. Runnable during runtime 2. Runnable during compile time 3. Runnable during runtime and compile time

The first is the most obvious, in most languages you write for this predominately. If manipulation of types is required, runtime based reflection is used. In D however this is not possible comparatively without explicit compile time knowledge and expectation of this. Second, to execute at compile time is mostly what this book is about. However it would be useless if it couldn't produce code to be executed during runtime. This is the third group.

Third, to execute at both compile time and runtime. To execute during runtime, information is passed in during compilation. While this may include the most basic generic like

[3] http://www.cplusplus.com/doc/tutorial/preprocessor/

functionality found in Java. For all all intents and purposes it is not for this book. That is categories under the first, runtime based. This is because while it does change code generation, it does not produce code specifically based upon the input. Code that uses this functionality include ranges and generics for e.g. all three string types (string, dstring and wstring).

Reflection

Previously it has been said that in the D programming language there is very limited means of reflecting over types during runtime. While this produces more solid, performant code it does introduce some serious restrictions. Because of this the usage of CTFE is heavily used when reflection is required.

D's support for reflection usage during compilation is separated out into two different parts. First the __traits expression. In a way this can be thought of as a pragma[4], a compiler specific expression to gain information about

[4]http://en.wikipedia.org/wiki/Directive_(programming)

types. The second being a module within the standard library, std.traits. Which provides many useful functions to gain information about types by using the information provided by e.g. __traits.

There is three useful parts of the language. First `typeof` statement. Which gives you the type of any variable or expression. Second `stringof` property. This is a property of a type where it gives you a representation of it. This can be changed at any time, it is not standard defined. Lastly, .mangleof for types (including methods) will give you the name the ABI has generated.

__traits

Traits are extensions to a language, to enable at compile time, to get information internal to the compiler. This is also known as compile time reflection. It is done as a special, easily extended syntax (similar to Pragmas) so that new capabilities can be added as required. There are many trait expressions that the D language support. Only a few will be mentioned here. They are the most useful when working at compile time and using it to identify and produce code. At the time of this writing there are many traits. The following will be covered.

- compiles
- getAttributes
- allMembers
- hasMember

This is far from complete. A full list would be the length of this page. The list is available on the D language reference traits page [5].

"compiles" The *compiles* expression is useful for when differentiating between code that can be used and cannot. This has two uses. 1. There is no method via e.g. traits to determine something that is available in the actual language. 2. Workaround compiler issues, with unknown workable code.

While it would be preferable that the second didn't exist, realistically even if the compiler was stable bug wise issues would still

[5] http://dlang.org/traits.html

cause it to be needed. To demonstrate a simple hello world example will be used.

Example code where compiles trait checks that writeln is defined

```
import std.stdio;

static if (__traits(compiles, { write\
ln("hi"); })) {
        void main() {
                writeln("hi");
        }
}
```

In this example importing of the std.stdio module is included. Then a static if to check that writeln is provided. If so include in a main function that write outs.

“getAttributes” This expression, getAttributes is used in conjunction with UDA’s or User Defined Types. These are heavily used in languages such as Java using the term annotation. These languages use them for everything from data models property definitions for ORM’s to serializers. An example of using UDA’s with the getAttributes traits expression is to get all *attributes* on a given function.

__traits getAttributes example usage with a function declaration

```
@("a")
@("b")
void func() {}

pragma(msg, __traits(getAttributes, f\
unc));
```

Output

```
tuple("a", "b")
```

In the given code, a function (func) has two separate UDA's applied to it. These then are printed during compilation by using the msg pragma. This also works with types and properties.

“allMembers” The allMembers trait is a highly important one when in usage in libraries that will automatically register types or declarations. It can be used to get all declarations and types within a module and class/struct/union/enum. It has such great importance that the author wants to express that this is considered a pet language feature for him. This is because, if this fails in any way it can be the difference between configuring all data models at runtime for a web service and not. An important quirk to be aware of is that in any module in D, it will automatically import object.d. This provides everything from type info for types to exceptions. Because of this, some extra symbols can be outputted from it. They may need to be checked for depending on the use case. For an example the difference between an enum, module and class will be used:

Traits allMembers example for a module, enum and class

```
module mymodule;

enum MyEnum {
        AValue,
        BValue
}

class MyClass {
        int x;
        bool y;

        void myfunc() {}
        int myfunc2() { return 0; }
}

pragma(msg, __traits(allMembers, mymo\
dule));
pragma(msg, __traits(allMembers, MyEn\
um));
pragma(msg, __traits(allMembers, MyCl\
ass));
```

In the given code, there has been declared an enum and a class. Do note that modules can be referred to, just liked imported modules by name. If the given code was to be ran it would output:

Output

```
tuple("object", "MyEnum", "MyClass")
tuple("AValue", "BValue")
tuple("x", "y", "myfunc", "myfunc2", \
"toString", "toHash", "opCmp", "opEqu\
als", "Monitor", "factory")
```

"hasMember" The last of the traits that will be discussed, but this one has some great uses in both workaround and in general feature detection. A great example of feature detection is to determine if the given method on a class was provided by Object or a super class/interface. To showcase this, a simple output if a class has a method:

Traits hasMember example, comparing a class and the Object class

```
class MyClass {

        void myfunc() {}
}

pragma(msg, __traits(hasMember, MyCla\
ss, "myfunc"));
pragma(msg, __traits(hasMember, MyCla\
ss, "toString"));
pragma(msg, __traits(hasMember, Objec\
t, "toString"));
```

In the given code, it should output true three times. This is a bool value returned by the trait. So it is safe to check against.

Standard library (std.traits)

Unlike the language supplied traits, the standard library functions does not add new support. They utilized the language support, parses the mangling to gain information about types and declarations. Not all functions are needed to be understood, but here are some key ones:

- moduleName
- isBasicType
- isSomeString
- ReturnType
- ParameterTypeTuple
- ParameterIdentifierTuple

These can be divided into information about a type and manipulating a type to get information. Information about a type is things like its module and manipulating a type (for a function) is getting it's return type.

Lisp family of languages

The LISP family languages cover a large variety. Common-LISP, Dylan and Converge are great examples. They each have some form of support for CTFE. They base their implementation upon macros. Common-LISP is the most basic of all the support. The macro support evaluating new code based upon the call. But expands out in a forced inline. While this is fairly powerful, Dylan provides much more access to the compiler. Dylan's macros are meant for direct extension to the language for both statements, definitions and operators. What is unique about Dylan is many features that would be considered on a type is moved into macros. Lastly in Converge, CTFE is thought of as compile-time metaprogramming. This is different in other LISP languages primarily in that it gives full access to the AST at that macros entry point. This is by using a "CEI" object. This is considered the main method at CTFE to interact with the compiler.

Different Types

Implementation of any functionality can vary widely. In the case of CTFE there is a number of different versions that can be implemented. To understand what can be used in hypothetical languages or implementations we will be categorizing them as:

- Constant expansion
- Macros
- Execution
- AST execution
- External execution

Constant expansion

An implementation defined as constant expansion is used primarily for numbers. While it does not form function execution during runtime, this is the basis for majority of the implementations.

All languages used this to some degree. Some have fancy support where by number constants can be in any number of formats. They could even form other types such as strings.

Macros

The macro implementation, has many great examples but primarily this can be defined as a preprocessor of sorts. A preprocessor such as C/C++'s would be the end definition.

Having support for conditional compilation, definition of values and conditional manipulation of input data these are functionality to be expected. However string manipulation might be out of scope. The preprocessor functionality will be heavily defined and limited. It should not be expected to extend it.

While this is called a macro implementation it does not include support for modifying a language statements or expressions the compiler supports. As seen in LISP family of languages. Nor does it allow for directly manipulating the compiler. A LISP macro is not the same as a macro defined by this.

Execution

It is expected that an execution implementation is capable of:

- Executing self contained functions that might have limitiations upon. These include:
  - No global data
  - Only functions with sources may be executed
  - Raw memory manipulation, not allowed
- Syntax might be special for function execution
- Meta-programming (compile time arguments) in some form is heavily recommended
- AST being able to be queried in some form. This may be limited to the compile unit or symbols passed via meta-programming

AST execution

Unlike Execution style CTFE, AST execution provides full access and manipulation directly to the compiler and AST. This can be very powerful in producing executable code that is highly efficient based upon reflected information.

Due to the direct nature, a compiler front end that supports this will have multiple iterations of code when it runs CTFE.

1. Argument deduction
2. Parse files
3. CTFE execution
4. Repeat 2 and 3 till there is none left to do
5. Produce binary

Of course there will be custom requirements based upon language needs.

External execution

External execution utilize the same support that AST execution supports except it has the capacity to execute unknown but linkable functions during compile time. This includes the C standard library for e.g. printf.

Most likely this will include access to three functions comparable to these three *nix functions:

- dlopen
- dlsym
- dlclose

At the very least, if the references gotten from it are wrapped in a type that can and will have its destructor called during compilation when the compile time value is no longer used. Then it should not leak and force the Operating System to clean up after the compiler.

Design patterns

To fully take advantage of compile time functionality, it may not be possible to use design patterns that are well known. Normally they take advantage of well definated virtual interfaces to classes to enable swapping between implementations. This is a hinderence when working with compile time and meta-programming. For these situations you must know the exact implementation that you are deal with at all points. A great example of a common design pattern that can be adapted to compile time easily is the visitor pattern. The core difference is in the element instead of specifying the visitor abstraction interface, you use an abstract class as part of the meta programming arguments. An Example in the D programming language:

```
class ConcreteElement : Element {
        override void accept(T : Visitor)(T \
visitor) {
                visitor.visit(this);
        }
}
```

At this point it would be simple on Visitor to implement a specific implementation for the Element implementation. Alternatively a similar method to the one above could be used to handle it abstractly. Such as:

```
class ConcreteVisitor : Visitor {
        void visit(T : Element)(T obj) {
                pragma(msg, T.stringof);
        }
}
```

Of note is the runtime if statement support for determining if a class instance inherits from a specific interfaces/class.

```
if (Type to = cast(Type)from) {
        ...
}
```

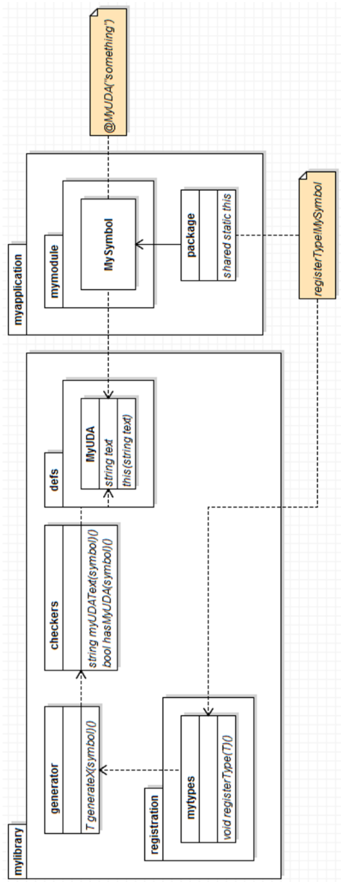
This might be useful at compile time if the possible types were limited. However if the purpose is to generate code for runtime usage then checking type using a `static if` is the right tool.

Helper generators

This design pattern is a general use case one. The term generator is a rather awful designation, but it pushes the thought of what a generator does. It creates something to be used. Under this design pattern there is two types of functions. A helper function is a single purpose function that is designed to execute only at compile time. It mostly requires some form of compile time arguments via e.g. metaprogramming. It uses type information to derive new information and return it. Attributes and returning a specific piece of information from it are a great example. A generator function produces a function to be executed at runtime, but uses compile time information. This calls helper functions to do so.

Get checker generators

As suggested in helper generator section, the usage of helper functions is to gain information from types. In this case by utilizing attributes on types and function/method declarations. This is followed from a registration function call. From this call the required hooking into the generators occur. The registration of types, functions ext. occur in a centralized manner. It is common to expect during registration some form of determinacy of what the input type is. The checking of types and determining what they are capable of is done by functions called checkers these are in the same category as helper functions however they unlike helper functions may do many things to archive there purpose.



Get checker design pattern

As shown in the a UML diagram, is how this would be used in the D programming language. The main thing to be aware of is D has module (a file) constructors. Enabling initiation and running of code before the main function. This handles registration of types.

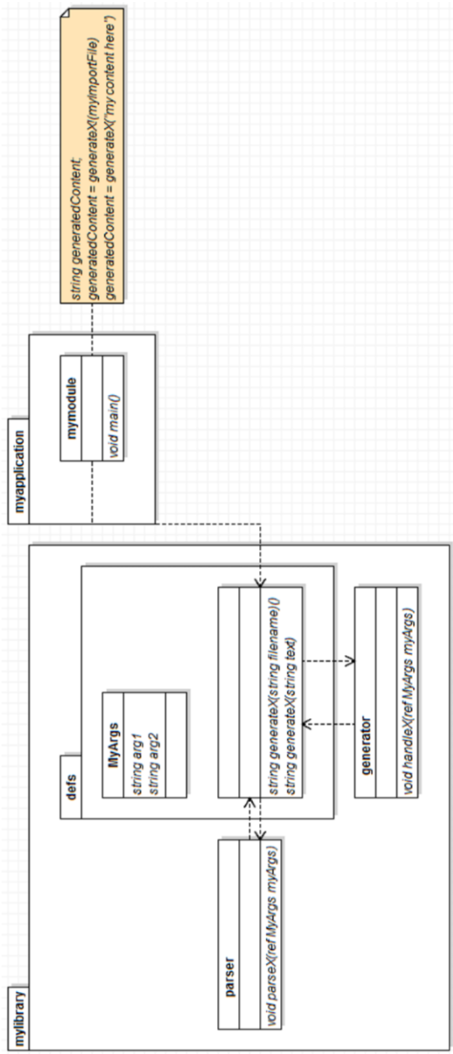
Import checker generators

This is an unusual design pattern in that it can be used both at compile time and runtime. The reason why this work at compile time is that it assumes all information required to do its work is available in some form of context data type. As an example the D programming language is used to facilitate demonstrating how it could be designed.

The meta-programming version of generateX is not required. This is only useful for passing in symbols or doing some other import related transformation that is only possible at compile time aware features.

The generation function acts in in a linear algorithm. Possible psuedo code is:

- Optional: transform input
- Call parseX
- Call handleX with result from parseX
- Return apropriete value(s) from the intermediate context variable from parseX/generateX.



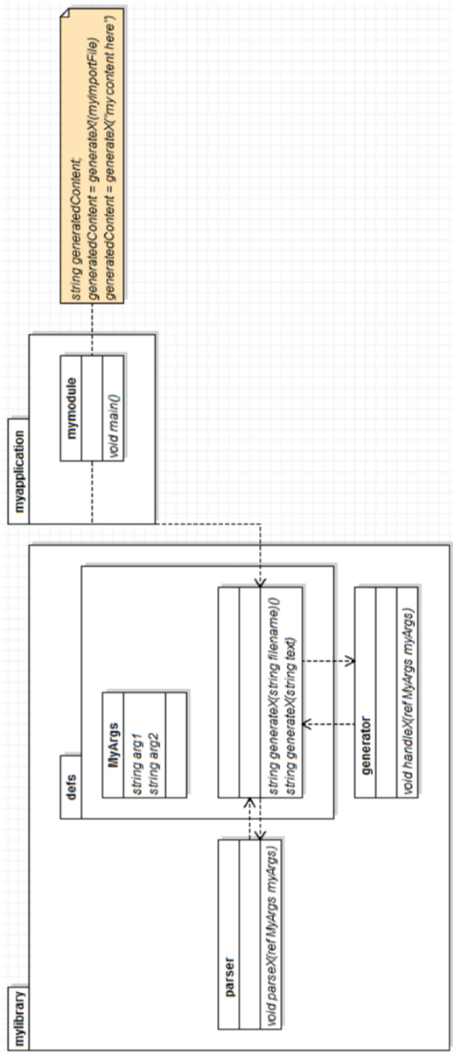
Import checker generator design pattern

Mix gen checked

This design pattern has similarities to the get checker generator pattern. The two main difference in them are:

- Get checker generator uses UDA's and a type for input during generation. Mix gen checked instead uses other types such as templated structs.
- Mix gen checked does not require types to be manually registered. Instead the usage of a language feature such as D's mixin template, is its own registration mechanism.

This pattern can be thought of as the external api version of the get checker generator pattern. Where it being the internal version.



Import checker generator design pattern

As shown in the UML diagram example, the structural implementation for the D programming language. It can be made slightly differently in that the checkers may be combined with the generator. The arguments to the mixin template within definitions may also be changed to accommodate per the purpose. Because of this, it is safe to assume that they might be modified before being passed to the generator.

Use cases

Web service development is quite a major use case for compile time reflection and execution. The reason being, you really don't want any code running at runtime then what needs to be.

So in a way for web development CTFE is an optimization technique to get faster page request completion. It's preferable in larger web services to take penalties at the start of execution then during it. Combining this with CTFE can be a very powerful ally to a web developer.

Web servics - Routing

Routing is one of the most fundamental parts to any web service framework. The Get checker generator pattern is ideal for generate efficient route checking code. In a breakdown the components required for this system are:

- A mechanism to register a unit (either a module/file or a class/struct or lastly the actual function/method.
- Appropriate UDA's along with the required helper generators functions for manipulating them.
- What will it be generated into? This is the runtime based library that will handle the usage of the compile time generation.
- Parser that handles the given input from the registration mechanism. Should only accept e.g. a function or for a class/struct class along with the method name.
- Generator, which generates based upon the parsers instructions for the runtime library to work with.

Web services - ORM

Unlike routing there is considerable amount of unknown parts to the system. Database providers can be swapped at runtime. This can also be a hindrance to optimizing calls to the database.

Usage of CTFE can be broken down into two types. Serializing/deserializing data to and from database safely, and enabling a custom query syntax per data structure.

Unless you are going full Hibernate like for UDA's the issue of serialization is not a worry. However if you want good support you may need quite a few functions one for each primitive type in the database provider. Remember UDA's are just instructions on how to handle the serialization and deserialization.

Query generation is the hard part here. These are data structures that contains queries to the data base. They should be customized fully to the data model and data backend. The data model part should be generated at compile time and database provider should be handled by the database provider.

Web services - Templating

Vibe.d has an implementation for templating running at compile time called Diet. Diet is based off of Jade a templating solution for Node.js.

Compile time templating has the benefit of optimizing a template to straight native code. Providing faster response time and memory. It does have the down side of not enabling runtime reloading if it supports the languages code like Diet does.

This use case can take advantage of the import checker generator pattern. Of course at the end of the generation it would need to produce some sort of function/delegate that can be executed at runtime given some arguments requested by the template.

When paired with shared libraries for reloading and some means to detect and recompile (with required imports) this can be an effective development and production mechanism.

JNI - Wrapping Java Types

Java Native Interface is a C API to interact with the JVM. To effectively use Java classes in a language the ability to wrap the objects in native types and enabling calling/creating them is preferable. This includes both strings, primitive types and actual objects.

Compilation time will increase significantly if definitions for the generated code is not used instead of the actual source files. Expansion of the definitions provided is a major requirement for large interfaces. If for example implementation of full bindings to the Java API was to be made. Further the generated interfaces should be paired with e.g. a static library with the actual implementation. This should be handled by the build manager.

In such a library there is a clear structure that the code must be made to meet.

- Definition

  Raw bindings to JNI.
- Linkage

Will contain niceties so raw JNI usage should not be needed in user code. It should be wrapped up into types including for calling of methods. There will also be the ability to transform a definition and mock it in some form with the actual JNI calling code. This could be generated in place of the definition or as a seperate object.

- Interfaces

  This is the bindings for e.g. the Java API. Will utilise the Linkage code.

- User code

  Uses Interfaces and almost never should have to interact with the raw JNI or linkage sections. There will most likely some sort of object that governs the interactions to JNI that can be configured. Within the linkage package. Which should be used.

The implementation will most likely utilise the mix gen checked design pattern for simplicity of the definitions.

Tutorials

Currently there are two tutorials on how to work with CTFE. The first is a simple example which just mutates some text and runs it. The second is a far more complicated and partially incomplete ORM implementation.

The code for both is provided at the back of the book and accompanying zip file.

ORM

In understanding anything, writing code is the best way. Of course the next best thing to actually doing it yourself, is following along with a tutorial. The tutorial that you are reading is going to implement reflection capabilities for an ORM. It is the key component to transfering the data models around by.

Steps to do this:

1. Define the subset of the language that will be supported. Classes and structs. Using UDA's to alter mapping and primitive types for data. As well as array support for e.g. strings.
2. Create example data model (to model what is intended)
3. Implement UDA's to match the given code
4. Create wrappers around a registered data model For this you will need to implement helper functions as well as traits. For each of the UDA's. This provides a reflection interface to use.

The tutorial code is based upon DNetDev's[6] future web service framework. For future reference, when this code was written `hasUDA` and `getUDA` where not in Phobos. The ORM that this is for supports D (static)methods on a data model for quering the database. These queries utilise the underlying architecture that the ORM supports for queries. But these are specific instances for the datamodel. But they must still obey the restrictions of the ORM e.g. types for arguments.

[6]http://github.com/DNetDev

Lets now step through the list of things that we need to do. Define the types that will be supported for fields and methods.

- ubyte
- byte
- ushort
- short
- uint
- int
- ulong
- long
- float
- double
- string
- wstring
- dstring
- data model
- ubyte[]
- byte[]
- ushort[]
- short[]
- uint[]
- int[]

- ulong[]
- long[]
- float[]
- double[]
- string[]
- wstring[]
- dstring[]
- data model[]

A very long list of types that are allowed. With recursion and `std.traits` this isn't really all that much work. The example implementation that does the real work in determining if a type is viable is `doIsADataModel` function in `webdev.base.traits.are`. You can see how it is applied in the function `isADataModel`. In essence it checks every property and validates the types associated with it. For application in ORM query methods (both static and non), check out `isADataModelQueryMethod` / `isADataModelQueryStaticMethod` which performs the validation of the types.

Where data model is a valid data model type.

The next and last stage to making this reflection algorithm is also the most complex. As part of this API there are:

- Type reflection instance
  - Constructing of an instance
  - Querying fields/query methods
  - Get table name
- Type instance reflection instance
  - Modification of fields
  - Execute ORM query
- Storing type reflection instances for lookup

Of note is that the implementation given is highly overload unfriendly. It should test and error out if overloads exist or work with them. It will also allocate constantly. It is highly recommend that it goes through an allocator that reserves lazily and calls destructors and finalizes the memory after every request. So it can be reallocated for the next one.

References

- Cmsed, web service framework in D by the author:

  https://github.com/rikkimax/Cmsed
- Converge, CTFE support:

  http://convergepl.org/documentation/2.0/ctmp/
- Dvorm, ORM in D by the author:

  https://github.com/rikkimax/Dvorm
- Dylan, CTFE support:

  http://opendylan.org/books/drm/Macros
- Jade, templating language:

  http://jade-lang.com/
- The D programming language reference on traits expressions:

  http://dlang.org/traits.html
- The D programming language standard library reference for traits:

  http://dlang.org/phobos/std_traits.html

- Vibe.d, asynchronous IO framework: https://github.com/rejectedsoftware/vibe.d

Glossary

- API, Application Program Interface
- AST, Abstract Syntax Tree
- CEI, Compiler External Interface
- CTFE, Compile Time Function Execution (or Evaluation)
- JNI, Java Native Interface
- JVM, Java Virtual Machine
- ORM, Object Relational Model
- PHP, Hypertext Preprocessor
- UDA, User Defined Attribute
- UML, Unified Modeling Language

Example code

The below code is here for reference purposes. However it is highly recommend to look at the extra zip file provided with this ebook instead of relying on this.

Tutorial code

Simple example

simpleExample.d

```
string doIt(string value) pure @safe\
 {
    import std.array : replace;
    return value.replace("Hello", "He\
y");
}

mixin(doIt(`
void main() {
    import std.stdio : writeln;
    writeln("Hello World!");
}
`));
```

Example model

model.d

```
module webdev.base.models.pagetemplat\
e;
import webdev.base.orm;
import webdev.base.udas;

@ormTableName("PageTemplate")
struct PageTemplateModel {
    @ormId {
        /**
         * The name of the page templ\
ate.
         */
        @ormPropertyHint(OrmPropertyT\
ypes.String, 0)
        @ormDescription("The name of \
the template. Should include location\
 information such as file path.")
        string name;

        /**
         * UTC+0 time of when this wa\
s last edited.
```

```
         */
        @ormPropertyHint(OrmPropertyT\
ypes.DateTime, 8) // we could hint at\
 this being able to store at the more\
 common definition or 4 bytes
        @ormDescription("When was thi\
s last changed. Do not change.")
        long lastEdited;
    }

    @ormOptional
    @ormPropertyHint(OrmPropertyTypes\
.Blob, 0)
    @ormDescription("Optionally the t\
emplate itself")
    string value;

    //TODO: mixin OrmModel!PageTempla\
teModel;

    bool isValid() {
        return lastEdited > 0 && name\
 !is null; // custom validation
    }

```

```
    @ormQuery {
        void setValue(string value) {
            this.value = value;
            updateLastEdited;
        }

        void updateLastEdited() {
            import webdev.base.util.t\
ime : utc0Time;
            lastEdited = utc0Time();
        }

        static {
            PageTemplateModel[] allLa\
testVersions() {
                PageTemplateModel[] r\
et;
                // TODO: some query h\
ere!
                return ret;
            }
        }
    }
}
```

User defined attributes

udas.d

```
module webdev.base.udas;

struct ormTableName {
    string name;
}

struct ormPropertyName {
    string name;
}

struct ormIgnore {}
struct ormOptional {}
struct ormId {}
struct ormOverrideUseArrays {}

struct ormDescription {
    string text;
}

enum OrmPropertyTypes {
    Integer,
    Float,
```

```
    Number,
    String,
    Blob,
    DateTime
}

struct ormPropertyHint {
    /**
     * What type is the property
     */
    OrmPropertyTypes type;

    /**
     * The number of bytes that shoul\
d be stored
     * Use 0 for unknown
     */
    size_t size;
}

struct ormQuery {}
```

Reflection wrapper

reflection.d

```
module webdev.base.reflection.model;
import webdev.base.traits.are : isADa\
taModel, isADataModelProperty, isData\
ModelMemberId, isADataModelQueryMetho\
d, isADataModelQueryStaticMethod;
import webdev.base.traits.have : getD\
ataModelName, getDataModelDescription\
, getDataModelPropertyDescription, ge\
tDataModelPropertyHints, getDataModel\
PropertyName;
import webdev.base.udas : OrmProperty\
Types;

private __gshared {
    import std.variant : Algebraic;
    import std.traits : fullyQualifie\
dName, isArray, ReturnType, Parameter\
TypeTuple;

    AReflectedModel*[string] models;
    AReflectedModel*[string] modelsBy\
TableName;
```

```
}

/*
 * Basic interactions of the differen\
t kinds of models
 */

/**
 * Gets all names of data models regi\
stered
 * Uses the fully qualified name (pac\
kage + module + class/struct name)
 *
 * Returns:
 *          The names to all data mode\
ls
 */
string[] reflectedModelNames() {
    return models.keys;
}

/**
 * Lazily registers and gets a reflec\
ted model given the data model type
 *
```

```
 * Returns:
 *          The reflected model
 */
AReflectedModel* getReflectModel(T)()\
 if(isADataModel!T) {
    return getReflectedModel(fullyQua\
lifiedName!T);
}


/**
 * Gets a reflected model based upon \
its fully qualified name
 *
 * Params:
 *          name    =    Name of the m\
odel
 *
 * Returns:
 *          The reflected model
 */
AReflectedModel* getReflectedModel(st\
ring name) {
    if (name !in models) return null;
    return models[name].dup();
```

```
}

/**
 * Gets a reflected model based upon \
its table name
 *
 * Params:
 *         name    =    Name of the m\
odel
 *
 * Returns:
 *         The reflected model
 */
AReflectedModel* getReflectedModelByT\
ableName(string name) {
    if (name !in modelsByTableName) r\
eturn null;
    return modelsByTableName[name].du\
p();
}

/*
 * General reflection based types
 */

```

```
///
enum OrmActualPropertyTypes {
    Unknown,

    UByte,
    Byte,
    UShort,
    Short,
    UInt,
    Int,
    ULong,
    Long,
    Float,
    Double,
    String,
    WString,
    DString,

    Array,
    DataModel
}

///
alias ModelValidTypes = Algebraic!(AR\
eflectedModelInstance*, ubyte, byte, \
```

```
ushort, short, uint, int, ulong, long\
, float, double, string, wstring, dst\
ring);

///
struct PropertyHint {
    ///
    string name;

    ///
    OrmActualPropertyTypes actualType;

    ///
    OrmActualPropertyTypes arrayActua\
lType;

    ///
    AReflectedModel* objectActualType;

    ///
    OrmPropertyTypes hintType;

    ///
    size_t size;

```

```
    ///
    string description;

    ///
    bool isId;
}

///
struct QueryDescriptor {
    ///
    QueryTypeDescriptor[] arguments;

    ///
    QueryTypeDescriptor returnType;

    struct QueryTypeDescriptor {
        ///
        OrmActualPropertyTypes actual\
Type;

        ///
        OrmActualPropertyTypes arrayA\
ctualType;

        ///
```

```
        AReflectedModel* objectActual\
Type;
    }
}

/*
 * The actual reflection mechanism
 */

struct AReflectedModel {
    static AReflectedModel* reflect(T\
)() if(isADataModel!T) {
        AReflectedModel* ret = new AR\
eflectedModel;

        ret.dup = () { return AReflec\
tedModel.reflect!T(); };

        /// constructs function deleg\
ates for a model instance aware of th\
e type
        void funcCalls(AReflectedMode\
lInstance* retm) {
            static if (__traits(hasMe\
mber, T, "isValid")) {
```

```
                retm.isValid = () { r\
eturn (cast(T*)retm.instance_).isVali\
d(); };
            } else {
                retm.isValid = () { r\
eturn true; };
            }

            retm.get = (string name) {
                foreach(member; __tra\
its(allMembers, T)) {
                    static if (isADat\
aModelProperty!(T, member)) {
                        if (member ==\
 name) {
                            return ne\
w ModelValidTypes(mixin("(cast(T*)ret\
m.instance_)." ~ member));
                        }
                    }
                }

                return null;
            };

```

```
                    retm.set = (string name, \
ModelValidTypes* value) {
                         foreach(member; __tra\
its(allMembers, T)) {
                             mixin("alias MTYP\
E = typeof(T." ~ member ~ ");");

                             static if (isADat\
aModelProperty!(T, member)) {
                                 if (member ==\
 name) {
                                     static if\
 (__traits(compiles, {MTYPE t = null;\
})) {
                                         if (v\
alue is null) {
                                             m\
ixin("(cast(T*)retm.instance_)." ~ me\
mber ~ " = null;");
                                         } els\
e if (value.convertsTo!MTYPE) {
                                             m\
ixin("(cast(T*)retm.instance_)." ~ me\
mber ~ " = value.get!MTYPE;");
                                         } els\
```

```
e {
                                    r\
eflectedAssert(0, value.type.toString\
 ~ " is not convertable to " ~ MTYPE.\
stringof);
                                }
                            } else {
                                if (v\
alue !is null && value.convertsTo!MTY\
PE) {
                                    m\
ixin("(cast(T*)retm.instance_)." ~ me\
mber ~ " = value.get!MTYPE;");
                                } els\
e {
                                    r\
eflectedAssert(0, value.type.toString\
 ~ " is not convertable to " ~ MTYPE.\
stringof);
                                }
                            }
                        }
                    }
                }
            };
```

```

            retm.query = (string name\
, ModelValidTypes[] values...) {
                foreach(member; __tra\
its(allMembers, T)) {
                    mixin("alias MTYP\
E = typeof(T." ~ member ~ ");");

                    static if (isADat\
aModelQueryMethod!(T, member)) {
                        alias MRET = \
ReturnType!MTYPE;

                        if (member ==\
 name) {
                            alias ARG\
U = ParameterTypeTuple!MTYPE;

                            if (value\
s.length != ARGU.length)
                                refle\
ctedAssert(0, "Not enough arguments")\
;

                            foreach(i\
```

```
, ARG; ARGU) {
                                        if (!\
values[i].convertsTo!ARG)
                                            r\
eflectedAssert(0, "Wrong types for ar\
guments");
                                    }

                                    static if\
 (is(MRET == void)) {
                                        // re\
turn call
                                        mixin\
("(cast(T*)retm.instance)." ~ member \
~ "(" ~ getCallToMethodSyntaxVarient!\
("values", "ARGU", ARGU) ~ ");");
                                        retur\
n cast(ModelValidTypes[])null;
                                    } else {
                                        // ca\
ll
                                        mixin\
("auto ret = (cast(T*)retm.instance_)\
." ~ member ~ "(" ~ getCallToMethodSy\
ntaxVarient!("values", "ARGU", ARGU) \
```

```
~ ");");

                                                  stati\
c if (isArray!MRET) {
                                                      M\
odelValidTypes[] ret2;

                                                      f\
oreach(v; ret) {
                                                       \
    ret2 ~= ModelValidTypes(v);
                                                      }

                                                      r\
eturn ret2;
                                                  } els\
e {
                                                      r\
eturn cast(ModelValidTypes[])[ModelVa\
lidTypes(v)];
                                                  }
                                             }
                                        }
                                   }
                              }
```

```

                reflectedAssert(0);
            };
        }

        ret.create = () {
            AReflectedModelInstance* \
retm = new AReflectedModelInstance;
            retm.model_ = ret;

            static if (is(T == class))
                retm.instance_ = &(ne\
w T);
            else static if (is(T == s\
truct))
                retm.instance_ = new \
T;
            else static assert(0);

            funcCalls(retm);
            return retm;
        };

        ret.fromInstance = (void* val\
ue) {
```

```
            /// in

            reflectedAssert(value !is\
 null);

            if (T* ttv = cast(T*)valu\
e){}
            else reflectedAssert(0, "\
Argument is not of type " ~ T.stringo\
f);

            /// body

            AReflectedModelInstance* \
retm = new AReflectedModelInstance;
            retm.model_ = ret;

            retm.instance_ = value;

            funcCalls(retm);
            return retm;
        };

        ret.tableName = () { return g\
etDataModelName!T; };
```

```
        ret.fullName = () { return fu\
llyQualifiedName!T; };
        ret.description = () { return\
 getDataModelDescription!T; };

        ret.propertyNames = () {
            string[] retm;

            foreach(member; __traits(\
allMembers, T)) {
                static if (isADataMod\
elProperty!(T, member)) {
                    retm ~= member;
                }
            }

            return retm;
        };

        ret.propertyHints = (string n\
ame) {
            PropertyHint retm;

            foreach(member; __traits(\
allMembers, T)) {
```

```
                  static if (isADataMod\
elProperty!(T, member)) {
                      if (member == nam\
e) {
                          mixin("alias \
MTYPE = typeof(T." ~ member ~ ");");

                          retm.name = g\
etDataModelPropertyName!(T, member);

                          // actualType
                          enum MTYPEA =\
 actualTypeFromType!MTYPE;
                          retm.actualTy\
pe = MTYPEA;

                          // arrayActua\
lType
                          static if (MT\
YPEA == OrmActualPropertyTypes.Array)
                              retm.arra\
yActualType = actualTypeFromType!(typ\
eof(MTYPE.init)[0]);

                          // objectActu\
```

```
alType
                        static if (MT\
YPEA == OrmActualPropertyTypes.DataMo\
del)
                            retm.obje\
ctActualType = getReflectModel!MTYPE;

                        auto hint = g\
etDataModelPropertyHints!(T, member);

                        // hintType
                        retm.hintType\
 = hint.type;

                        // size
                        if (hint.size\
 == 0) {
                            static if\
 (isArray!MTYPE) {
                            } else
                                hint.\
size = MTYPE.sizeof;
                        } else
                            retm.size\
 = hint.size;
```

```

                        // description
                        retm.descript\
ion = getDataModelPropertyDescription\
!(T, member);

                        // isId
                        retm.isId = i\
sDataModelMemberId!(T, member);
                    }
                }
            }

            return retm;
        };

        ret.query = (string name, Mod\
elValidTypes[] values...) {
            foreach(member; __traits(\
allMembers, T)) {
                mixin("alias MTYPE = \
typeof(T." ~ member ~ ");");

                static if (isADataMod\
elQueryStaticMethod!(T, member)) {
```

```
                                        alias MRET = Retu\
rnType!MTYPE;

                                        if (member == nam\
e) {
                                                alias ARGU = \
ParameterTypeTuple!MTYPE;

                                                if (values.le\
ngth != ARGU.length)
                                                        reflected\
Assert(0, "Not enough arguments");

                                                foreach(i, AR\
G; ARGU) {
                                                        if (!valu\
es[i].convertsTo!ARG)
                                                                refle\
ctedAssert(0, "Wrong types for argume\
nts");
                                                }

                                                static if (is\
(MRET == void)) {
                                                        // return\
```

```
 call
                                            mixin("T.\
" ~ member ~ "(" ~ getCallToMethodSyn\
taxVarient!("values", "ARGU", ARGU) ~\
 ");");
                                            return ca\
st(ModelValidTypes[])null;
                                        } else {
                                            // call
                                            mixin("au\
to ret = T." ~ member ~ "(" ~ getCall\
ToMethodSyntaxVarient!("values", "ARG\
U", ARGU) ~ ");");

                                            ModelVali\
dTypes[] ret2;

                                            static if\
 (isArray!MRET) {
                                                forea\
ch(v; ret) {
                                                    s\
tatic if (is(typeof(v) == class) || i\
s(typeof(v) == struct)) {
                                                     \
```

```
    ret2 ~= ModelValidTypes(getReflect\
Model!(typeof(v))().fromInstance(&v))\
;
                                    }\
 else {
                                     \
    ret2 ~= ModelValidTypes(v);
                                    }
                                }
                            } else {
                                stati\
c if (is(MRET == class) || is(MRET ==\
 struct)) {
                                    r\
et2 ~= ModelValidTypes(getReflectMode\
l!(MRET)().fromInstance(&ret));
                                } els\
e {
                                    r\
et2 ~= ModelValidTypes(ret);
                                }
                            }

                            return re\
t2;
```

```
                        }
                    }
                }
            }

            reflectedAssert(0);
        };

        // queryNames
        ret.queryNames = () {
            string[] ret;

            foreach(member; __traits(\
allMembers, T)) {
                mixin("alias MTYPE = \
typeof(T." ~ member ~ ");");

                static if (isADataMod\
elQueryStaticMethod!(T, member)) {
                    ret ~= member;
                }
            }

            return ret;
        };
```

```

        // queryParameters
        ret.queryParameters = (string\
 name) {
            QueryDescriptor ret;

            foreach(member; __traits(\
allMembers, T)) {
                static if (isADataMod\
elQueryStaticMethod!(T, member)) {
                    if (member == nam\
e) {

                        // arguments
                        foreach(ARG; \
ParameterTypeTuple!(mixin("T." ~ name\
))) {
                            QueryDesc\
riptor.QueryTypeDescriptor rett;

                            // actual\
Type
                            enum MTYP\
EA = actualTypeFromType!ARG;
                            rett.actu\
```

```
alType = MTYPEA;

                                                    // arrayA\
ctualType
                                                    static if\
 (MTYPEA == OrmActualPropertyTypes.Ar\
ray)
                                                        rett.\
arrayActualType = actualTypeFromType!\
(typeof(ARG.init)[0]);

                                                    // object\
ActualType
                                                    static if\
 (MTYPEA == OrmActualPropertyTypes.Da\
taModel)
                                                        rett.\
objectActualType = getReflectModel!AR\
G;

                                                    ret.argum\
ents ~= rett;
                                        }

                                                // return type
```

```

                                alias RETM = \
ReturnType!(mixin("T." ~ m));

                                // actualType
                                enum MTYPEA =\
 actualTypeFromType!RETM;
                                ret.returnTyp\
e.actualType = MTYPEA;

                                // arrayActua\
lType
                                static if (MT\
YPEA == OrmActualPropertyTypes.Array)
                                    ret.retur\
nType.arrayActualType = actualTypeFro\
mType!(typeof(ARG.init)[0]);

                                // objectActu\
alType
                                static if (MT\
YPEA == OrmActualPropertyTypes.DataMo\
del)
                                    ret.retur\
nType.objectActualType = getReflectMo\
```

```
del!ARG;
                    }
                }
            }

            return ret;
        };

        // queryInstanceNames
        ret.queryInstanceNames = () {
            string[] ret;

            foreach(member; __traits(\
allMembers, T)) {
                mixin("alias MTYPE = \
typeof(T." ~ member ~ ");");

                static if (isADataMod\
elQueryMethod!(T, member)) {
                    ret ~= member;
                }
            }

            return ret;
        };
```

```

        // queryInstanceParameters
        ret.queryInstanceParameters =\
 (string name) {
            QueryDescriptor ret;

            foreach(member; __traits(\
allMembers, T)) {
                static if (isADataMod\
elQueryMethod!(T, member)) {
                    if (member == nam\
e) {

                        // arguments
                        foreach(ARG; \
ParameterTypeTuple!(mixin("T." ~ name\
))) {
                            QueryDesc\
riptor.QueryTypeDescriptor rett;

                            // actual\
Type
                            enum MTYP\
EA = actualTypeFromType!ARG;
                            rett.actu\
```

```
alType = MTYPEA;

                                        // arrayA\
ctualType
                                        static if\
 (MTYPEA == OrmActualPropertyTypes.Ar\
ray)
                                            rett.\
arrayActualType = actualTypeFromType!\
(typeof(ARG.init)[0]);

                                        // object\
ActualType
                                        static if\
 (MTYPEA == OrmActualPropertyTypes.Da\
taModel)
                                            rett.\
objectActualType = getReflectModel!AR\
G;

                                        ret.argum\
ents ~= rett;
                        }

                                    // return type
```

```

                        alias RETM = \
ReturnType!(mixin("T." ~ m));

                        // actualType
                        enum MTYPEA =\
 actualTypeFromType!RETM;
                        ret.returnTyp\
e.actualType = MTYPEA;

                        // arrayActua\
lType
                        static if (MT\
YPEA == OrmActualPropertyTypes.Array)
                            ret.retur\
nType.arrayActualType = actualTypeFro\
mType!(typeof(ARG.init)[0]);

                        // objectActu\
alType
                        static if (MT\
YPEA == OrmActualPropertyTypes.DataMo\
del)
                            ret.retur\
nType.objectActualType = getReflectMo\
```

```
del!ARG;
                    }
                }
            }

            return ret;
        };

        models[fullyQualifiedName!T] \
= ret;
        models[ret.tableName()] = ret;
        return ret;
    }

    AReflectedModel* delegate() dup;
    AReflectedModelInstance* delegate\
() create;
    AReflectedModelInstance* delegate\
(void*) fromInstance;

    string delegate() tableName;
    string delegate() fullName;
    string delegate() description;

    const(string[]) delegate() proper\
```

```
tyNames;
    PropertyHint delegate(string name\
) propertyHints;

    ModelValidTypes[] delegate(string\
 func, ModelValidTypes[] values...) q\
uery;
    const(string[]) delegate() queryN\
ames;
    QueryDescriptor delegate(string n\
ame) queryParameters;

    const(string[]) delegate() queryI\
nstanceNames;
    QueryDescriptor delegate(string n\
ame) queryInstanceParameters;
}

struct AReflectedModelInstance {
    private {
        AReflectedModel* model_;

        void* instance_;
    }

```

```
    AReflectedModel* model() { return\
 model; }
    void* instance() { return instanc\
e; }

    /*
     *
     * Shouldn't everything below thi\
s, stored in the model instead of the\
 state?
     *
     */

    bool delegate() isValid;

    /*
     * Get/Set for all approved prope\
rties given a name and the value for \
a model instance
     */
    ModelValidTypes* delegate(string \
name) get;
    void delegate(string name, ModelV\
alidTypes* value) set;
```

```
    ModelValidTypes[] delegate(string\
 func, ModelValidTypes[] values...) q\
uery;
}

alias NotReflectedModel = Exception;

private {
    void reflectedAssert(lazy bool va\
lue, string msg="", string file = __F\
ILE__, size_t line = __LINE__) {
        if (!value)
            throw new NotReflectedMod\
el(msg, file, line);
    }

    OrmActualPropertyTypes actualType\
FromType(T)() {
        static if (is(T == ubyte))
            return OrmActualPropertyT\
ypes.UByte;
        else static if (is(T == byte))
            return OrmActualPropertyT\
ypes.Byte;
        else static if (is(T == ushor\
```

```
t))
            return OrmActualPropertyT\
ypes.UShort;
        else static if (is(T == short\
))
            return OrmActualPropertyT\
ypes.Short;
        else static if (is(T == uint))
            return OrmActualPropertyT\
ypes.UInt;
        else static if (is(T == int))
            return OrmActualPropertyT\
ypes.Int;
        else static if (is(T == ulong\
))
            return OrmActualPropertyT\
ypes.ULong;
        else static if (is(T == long))
            return OrmActualPropertyT\
ypes.Long;
        else static if (is(T == float\
))
            return OrmActualPropertyT\
ypes.Float;
        else static if (is(T == doubl\
```

```
e))
            return OrmActualPropertyT\
ypes.Double;
        else static if (is(T == strin\
g))
            return OrmActualPropertyT\
ypes.String;
        else static if (is(T == wstri\
ng))
            return OrmActualPropertyT\
ypes.WString;
        else static if (is(T == dstri\
ng))
            return OrmActualPropertyT\
ypes.DString;
        else static if (is(T == class\
) || is(T == struct))
            return OrmActualPropertyT\
ypes.DataModel;
        else static if (isArray!T)
            return OrmActualPropertyT\
ypes.Array;
        else
            return OrmActualPropertyT\
ypes.Unknown;
```

```
    }

    string getCallToMethodSyntaxVarie\
nt(string valuesName, string argName,\
 ARGS...)() pure {
        import std.conv : text;
        string ret;

        foreach(i, ARG; ARGS) {
            string TI = text(i);
            ret ~= valuesName ~ "[" ~\
 TI ~ "].get!(" ~ argName ~ "[" ~ TI \
~ "]), ";
        }

        if (ARGS.length > 0)
            ret.length -= 2;

        return ret;
    }
}
```

Compatibility UDA code

defs.d

```
module webdev.base.traits.defs;

template hasUDA(UDATYPE) {
    bool hasUDA(T)() pure {
        static if (__traits(compiles,\
 __traits(getAttributes, T))) {

            foreach(uda; __traits(get\
Attributes, T)) {
                static if (is(typeof(\
uda) == UDATYPE)) {

                    return true;
                }
            }

        }

        return false;
    }

    bool hasUDA(T, string member)() p\
```

```
ure {
        static if (__traits(compiles,\
 __traits(getAttributes, mixin("T." ~\
 member)))) {

            foreach(uda; __traits(get\
Attributes, mixin("T." ~ member))) {

                static if (__traits(c\
ompiles, {alias U = typeof(uda);})) {
                    static if (is(typ\
eof(uda) == UDATYPE)) {
                        return true;
                    }
                } else {
                    static if (is(uda\
 == UDATYPE)) {
                        return true;
                    }
                }

            }

        }

```

```
        return false;
    }
}

template getUDA(UDATYPE) {
    UDATYPE getUDA(T)() pure {

        foreach(uda; __traits(getAttr\
ibutes, T)) {
            static if (is(typeof(uda)\
 == UDATYPE)) {
                return uda;
            }
        }

        return UDATYPE.init;
    }

    UDATYPE getUDA(T, string member)(\
) pure {

        foreach(uda; __traits(getAttr\
ibutes, mixin("T." ~ member))) {
            static if (__traits(compi\
les, {alias U = typeof(uda);})) {
```

```
                static if (is(typeof(\
uda) == UDATYPE)) {
                    return uda;
                }
            } else {
                static if (is(uda == \
UDATYPE)) {
                    return uda;
                }
            }
        }

        return UDATYPE.init;
    }
}
```

Does symbol have UDA?

are.d

```
module webdev.base.traits.are;
import webdev.base.traits.defs;
import webdev.base.udas;

///
bool isADataModel(T, bool inADataMode\
l = false)() pure {
    static if (isADataModelBase!T) {
        bool ret = true;
        bool gotProperties;
        bool hasIds;

        foreach(member; __traits(allM\
embers, T)) {
            doIsADataModel!(T, mixin(\
"typeof(T." ~ member ~ ")"), member, \
inADataModel)(ret, gotProperties, has\
Ids);
        }

        return ret && gotProperties &\
& hasIds;
```

```
    } else {
        return false;
    }
}

///
bool isADataModelProperty(T, string n\
ame)() pure {
    static if (__traits(hasMember, T,\
 name)) {
        bool ret = true;
        bool gotProperties;
        bool hasIds;

        doIsADataModel!(T, mixin("typ\
eof(T." ~ name ~ ")"), name, false)(r\
et, gotProperties, hasIds);

        return ret && gotProperties;
    } else {
        return false;
    }
}

///
```

```
bool isADataModelQueryMethod(T, strin\
g name)() pure {
    static if (__traits(hasMember, T,\
 name)) {
        static if(!__traits(isStaticF\
unction, mixin("T." ~ name)) && isDat\
aModelMethodQueryUDA!(T, name)) {
            bool ret = true;

            // check argument types
            foreach(ARG; ParameterTyp\
eTuple!(mixin("T." ~ name))) {
                // ensure return type\
 is not a pointer!
                static if (isBasicTyp\
e!ARG || isSomeString!ARG || isADataM\
odel!ARG) {
                } else static if (isA\
rray!ARG) {
                    alias AT = typeof\
(ARG.init[0]);

                    static if (isBasi\
cType!AT || isSomeString!AT || isADat\
aModel!AT) {
```

```
                    } else {
                        ret = false;
                    }
                } else {
                    ret = false;
                }
            }

            // check return type
            alias MRET = ReturnType!(\
mixin("T." ~ m));

            // ensure return type is \
not a pointer!
            static if (isBasicType!MR\
ET || isSomeString!MRET || isADataMod\
el!MRET) {
            } else static if (isArray\
!MRET) {
                alias AT2 = typeof(MR\
ET.init[0]);

                static if (isBasicTyp\
e!AT2 || isSomeString!AT2 || isADataM\
odel!AT2) {
```

```
                } else {
                    ret = false;
                }
            } else {
                ret = false;
            }

            return ret;
        } else {
            return false;
        }
    } else {
        return false;
    }
}

///
bool isADataModelQueryStaticMethod(T,\
 string name)() pure {
    static if (__traits(hasMember, T,\
 name)) {
        static if(__traits(isStaticFu\
nction, mixin("T." ~ name)) && isData\
ModelMethodQueryUDA!(T, name)) {
            bool ret = true;
```

```

            // check argument types
            foreach(ARG; ParameterTyp\
eTuple!(mixin("T." ~ name))) {
                // ensure return type\
 is not a pointer!
                static if (isBasicTyp\
e!ARG || isSomeString!ARG || isADataM\
odel!ARG) {
                } else static if (isA\
rray!ARG) {
                    alias AT = typeof\
(ARG.init[0]);

                    static if (isBasi\
cType!AT || isSomeString!AT || isADat\
aModel!AT) {
                    } else {
                        ret = false;
                    }
                } else {
                    ret = false;
                }
            }

```

```
            // check return type
            alias MRET = ReturnType!(\
mixin("T." ~ m));

            // ensure return type is \
not a pointer!
            static if (isBasicType!MR\
ET || isSomeString!MRET || isADataMod\
el!MRET) {
            } else static if (isArray\
!MRET) {
                alias AT2 = typeof(MR\
ET.init[0]);

                static if (isBasicTyp\
e!AT2 || isSomeString!AT2 || isADataM\
odel!AT2) {
                } else {
                    ret = false;
                }
            } else {
                ret = false;
            }

            return ret;
```

```
        } else {
            return false;
        }
    } else {
        return false;
    }
}

alias isADataModelBase = hasUDA!ormTa\
bleName;
alias isDataModelMemberIgnored = hasU\
DA!ormIgnore;
alias isDataModelMemberId = hasUDA!or\
mId;
alias isDataModelMemberUseArrays = ha\
sUDA!ormOverrideUseArrays;

private {
    import std.traits : isBasicType, \
isSomeString, isArray, ReturnType, Pa\
rameterTypeTuple;

    alias isDataModelMethodQueryUDA =\
 hasUDA!ormQuery;

```

```
    /**
     *
     * Params:
     *          T               =
     *          MTYPE           =
     *          member          =
     *          inADataModel    =
     *          noArray         =    \
Bad practice to use arrays for data m\
odel properties! As databases don't s\
upport it. Default: true, currently n\
ot overridable.
     *
     *             ret               \
 =
     *             gotProperties    =\

     *          hasIds          =
     */
    void doIsADataModel(T, MTYPE, str\
ing member, bool inADataModel, bool n\
oArray = true)(ref bool ret, ref bool\
 gotProperties, ref bool hasIds) pure\
 {
        static if(isDataModelMemberIg\
```

```
nored!(T, member)) {
            // if it is ignored, it w\
on't be included in the definition pe\
riod
        } else {
            enum IsID = isDataModelMe\
mberId!(T, member);

            static if (IsID) {
                hasIds = true;
            }

            static if (is(MTYPE == cl\
ass) || is(MTYPE == struct)) {
                static if (!IsID) {
                    gotProperties = t\
rue;

                    static if (!isADa\
taModel!(MTYPE, true))
                        ret = false;
                } else {
                    ret = false;
                    pragma(msg, "Recu\
rsive data models as id's are not all\
```

```
owed. Perhaps alias this is in order?\
 At: " ~ T.stringof ~ " " ~ member);
                }
            } else static if (isSomeS\
tring!MTYPE || isBasicType!MTYPE) {
                gotProperties = true;
            } else static if (isArray\
!MTYPE) {
                // ugg databases don'\
t support arrays, atleast normally
                static if (isDataMode\
lMemberUseArrays!T || isDataModelMemb\
erUseArrays!(T, member)) {
                    // allow the back\
end to fake it or use the database's \
support
                    static if (!noArr\
ay) {
                        doIsADataMode\
l!(T, typeof((MTYPE.init)[0]), member\
, inADataModel, true)(ret, gotPropert\
ies, hasIds);
                    } else {
                        pragma(msg, "\
Array of array is not a valid propert\
```

```
y type");
                    }
                } else {
                    pragma(msg, "Tip!\
 Databases generally don't support ar\
rays! So why does " ~ T.stringof ~ " \
use them?");
                }
            } else static if (is(MTYP\
E == delegate) || is(MTYPE == functio\
n)) {
                // don't care about m\
ethods or function/delegate pointers
                // they can't and wil\
l never be able to be serialized to t\
he database
            } else {
                pragma(msg, MTYPE.str\
ingof ~ " cannot be used as a data mo\
del property type!");
                ret = false;
            }
        }
    }
}
```

UDA value

have.d

```d
module webdev.base.traits.have;
import webdev.base.traits.defs;
import webdev.base.traits.are;
import webdev.base.udas;

private {
    alias getDataModelUDA = getUDA!ormTableName;
    alias getDataModelPropertyNameUDA = getUDA!ormPropertyName;
    alias getDataModelDescriptionUDA = getUDA!ormDescription;
    alias getDataModelPropertyHintsUDA = getUDA!ormPropertyHint;
}

///
string getDataModelName(T)() pure if(isADataModel!T) {
    return (getDataModelUDA!T).name;
}

```

```
///
string getDataModelPropertyName(T, st\
ring member)() pure if(isADataModel!T\
) {
    string ret = (getDataModelPropert\
yNameUDA!(T, member)).name;

    if (ret is null)
        return member;
    else
        return ret;
}

///
string getDataModelDescription(T)() p\
ure if(isADataModel!T) {
    return (getDataModelDescriptionUD\
A!T).text;
}

///
string getDataModelPropertyDescriptio\
n(T, string member)() pure if(isAData\
Model!T) {
    return (getDataModelDescriptionUD\
```

```
A!(T, member)).text;
}

///
ormPropertyHint getDataModelPropertyH\
ints(T, string member)() pure if(isAD\
ataModel!T) {
    return getDataModelPropertyHintsU\
DA!(T, member);
}
```