Semantic Tension:

The Edge Cases of D's Introspective Power

DConf London '25

Timon Gehr

Compile-time Introspection in D

Enabled by (roughly):

- Code Generation
  - `template` (hygienic)
  - `mixin` (non-hygienic)
- Queries and Unification
  - `__traits`
  - `is` expressions
  - IFTI
- Conditional Compilation
  - `static if`
  - `static foreach`

__traits

```
TraitsExpression:
    __traits ( TraitsKeyword , TraitsArguments )

TraitsKeyword:
    isAbstractClass
    isArithmetic
    isAssociativeArray
    isFinalClass
    isPOD
    isNested
    isFuture
    isDeprecated
    isFloating
    isIntegral
    isScalar
    isStaticArray
    isUnsigned
    isDisabled
    isVirtualFunction
    isVirtualMethod
    isAbstractFunction
    isFinalFunction
    isStaticFunction
    isOverrideFunction
    isTemplate
    isRef
    isOut
    isLazy
```

`__traits(compiles, ...)`

26.5.29 *compiles*

1. Returns a bool **`true`** if all of the arguments compile (are semantically correct). The arguments can be symbols, types, or expressions that are syntactically correct. The arguments cannot be statements or declarations - instead these can be wrapped in a function literal expression.

2. If there are no arguments, the result is **`false`**.

```
static assert(!__traits(compiles));
static assert(__traits(compiles, 1 + 1)); // expression
static assert(__traits(compiles, typeof(1))); // type
static assert(__traits(compiles, object)); // symbol
static assert(__traits(compiles, 1, 2, 3, int, long));
static assert(!__traits(compiles, 3[1])); // semantic error
static assert(!__traits(compiles, 1, 2, 3, int, long, 3[1]));

enum n = 3;
// wrap a declaration/statement in a function literal
static assert(__traits(compiles, { int[n] arr; }));
static assert(!__traits(compiles, { foreach (e; n) {} }));

struct S
{
    static int s1;
    int s2;
}

static assert(__traits(compiles, S.s1 = 0));
static assert(!__traits(compiles, S.s2 = 0));
static assert(!__traits(compiles, S.s3));
```

__traits(allMembers, ...)

26.5.26 *allMembers*

1. Takes a single argument, which must evaluate to either a module, a struct, a union, a class, an interface, an enum, or a template instantiation.

2. A sequence of string literals is returned, each of which is the name of a member of that argument combined with all of the members of its base classes (if the argument is a class). No name is repeated. Builtin properties are not included.

```
import std.stdio;

class D
{
    this() { }
    ~this() { }
    void foo() { }
    int foo(int) { return 0; }
}

void main()
{
    auto b = [ __traits(allMembers, D) ];
    writeln(b);
    // ["__ctor", "__dtor", "foo", "toString", "toHash", "opCmp", "opEquals",
    // "Monitor", "factory"]
}
```

static if

Report a bug · Improve this page

25. Conditional Compilation

master ▾

1. *Conditional compilation* is the process of selecting which code to compile and which code to not compile.

```
ConditionalDeclaration:
    Condition DeclarationBlock
    Condition DeclarationBlock else DeclarationBlock
    Condition : DeclDefsopt
    Condition DeclarationBlock else : DeclDefsopt

ConditionalStatement:
    Condition NoScopeNonEmptyStatement
    Condition NoScopeNonEmptyStatement else
NoScopeNonEmptyStatement
```

Contents [hide]

2. If the *Condition* is satisfied, then the following *DeclarationBlock* or *Statement* is compiled in. If it is not satisfied, the *DeclarationBlock* or *Statement* after the optional **`else`** is compiled in.

3. Any *DeclarationBlock* or *Statement* that is not compiled in still must be syntactically correct.

4. No new scope is introduced, even if the *DeclarationBlock* or *Statement* is enclosed by **`{ }`**.

5. *ConditionalDeclaration*s and *ConditionalStatement*s can be nested.

6. The *StaticAssert* can be used to issue errors at compilation time for branches of the conditional compilation that are errors.

25.5 Static If Condition

```
StaticIfCondition:
    static if ( AssignExpression )
```

1. *AssignExpression* is implicitly converted to a boolean type, and is evaluated at compile time. The condition is satisfied if it evaluates to **`true`**. It is not satisfied if it evaluates to **`false`**.

Forward References

```d
static if(__traits(compiles, z)) {
    pragma(msg, "z exists");
}

pragma(msg, x); // 1

static if(__traits(compiles, x)) {
    enum y = 2;
}
enum x = 1;

pragma(msg, x, " ", y); // 1 2
```

```
$ dmd -o- test.d
1
1 2
```

Multiple Solutions

The following code is appended to all examples:

```
static if(__traits(compiles, x)) {
    pragma(msg, "x exists");
}
static if(__traits(compiles, y)) {
    pragma(msg, "y exists");
}
static if(__traits(compiles, z)) {
    pragma(msg, "z exists");
}
```

Multiple Solutions

```
enum x = 1;

static if(__traits(compiles, x)) {
    enum y = 2;
}
```

```
$ dmd -o- test.d
x exists
y exists
```

Multiple Solutions

```
static if(__traits(compiles, y)) {
    enum x = 1;
}

enum y = 2;
```

```
$ dmd -o- test.d
x exists
y exists
```

Multiple Solutions

```
static if(__traits(compiles, y)) {
    enum x = 1;
}
static if(__traits(compiles, x)) {
    enum y = 2;
}
```

```
$ dmd -o- test.d
$
```

Multiple Solutions (Cont.)

```d
enum x = 1;

static if(!__traits(compiles, x)) {
    enum y = 2;
}
```

```
$ dmd -o- test.d
x exists
$
```

Multiple Solutions (Cont.)

```d
static if(!__traits(compiles, y)) {
    enum x = 1;
}

enum y = 2;
```

```
$ dmd -o- test.d
y exists
$
```

Multiple Solutions (Cont.)

```
static if(!__traits(compiles, y)) {
    enum x = 1;
}
static if(!__traits(compiles, x)) {
    enum y = 2;
}
```

```
$ dmd -o- test.d
x exists
$
```

Multiple Solutions (Cont.)

```d
static if(__traits(compiles, z)) {
    static if(!__traits(compiles, y)) {
        enum x = 1;
    }
}
static if(!__traits(compiles, x)) {
    enum y = 2;
}

enum z = 3;
```

```
$ dmd -o- test.d
x exists
z exists
$
```

Multiple Solutions (Cont.)

```d
static if(__traits(compiles, z)) {
    static if(!__traits(compiles, y)) {
        enum x = 1;
    }
}
static if(!__traits(compiles, x)) {
    enum y = 2;
}
static if(__traits(compiles, y)) {
    enum z = 3;
}
```

```
$ dmd -o- test.d
y exists
z exists
$
```

Module order

```d
module a;   import b;
static if(!__traits(compiles, y)) {
    enum x = 1;
}
```

```d
module b;   import a;
static if(!__traits(compiles, x)) {
    enum y = 2;
}
```

```
$ dmd -o- a.d b.d
y exists
$
```

```
$ dmd -o- b.d a.d
x exists
$
```

Self-contradictory code

```
static if(__traits(compiles, x)) {
    pragma(msg, "x defined above");
}
static if(!__traits(compiles, x)) {
    enum x = 2;
}
```

```
$ dmd -o- test.d
x exists
$
```

The same condition in the same scope evaluates to different results.

Self-contradictory code (Cont.)

```
string foo() => q{static string foo() => q{int bar() => 2;};};

struct S{
    mixin(foo());
}

pragma(msg, __traits(allMembers, S));
```

```
$ dmd -o- test.d
AliasSeq!("foo")
$
```

Self-contradictory code (Cont.)

```
string foo() => q{static string foo() => q{int bar() => 2;};};

struct S{
    mixin(foo());
    mixin(foo());
}

pragma(msg, __traits(allMembers, S));
```

```
$ dmd -o- test.d
AliasSeq!("foo","bar")
$
```

Self-contradictory code (Cont.)

```
string foo() => q{static string foo() => q{int bar() => 2;};};

struct S{
    alias a = foo;
    mixin(foo());
    alias b = foo;
    static assert(__traits(isSame, a, b)); // fails
}
```

Self-contradictory code (Cont.)

```
string foo() => q{static string foo() => q{int bar() => 2;};};

struct S{
    static if(__traits(compiles, b)){
        alias a = foo;
    }
    mixin(foo());
    alias b = foo;
    static assert(__traits(isSame, a, b)); // passes
}
```

```
int foo(long z) => 1;
static if(foo(1) == 1) {
    static assert(foo(1) == 1);

    // ...
}
```

```
int foo(long z) => 1;
static if(foo(1) == 1) {
    static assert(foo(1) == 1); // fails

    int foo(int x) => 2;
}
```

Caching

```
int foo(long z) => 1;
enum Foo(alias x)=foo(x);

static assert(Foo!1 == foo(1)); // fails

alias a = Foo!1;
static if(foo(1) == 1) {
    int foo(int x) => 2;
}
```

Invalid Solutions

```
static if(__traits(compiles, x)) {
    enum y = true;
}
static if(!__traits(compiles, z)) {
    enum x = true;
}
```

```
$ dmd -o- test.d
x exists
$
```

Invalid Solutions

```
static if(!__traits(compiles, z)) {
    enum x = true;
}
static if(__traits(compiles, x)) {
    enum y = true;
}
```

```
$ dmd -o- test.d
x exists
y exists
$
```

What else could D do?

- Use fixed evaluation order.
  - Breaks forward referencing, inconsistent.
  - Would need to define global order on modules. Brittle.

What else could D do?

- Actually solve as a constraint satisfaction problem.
  - Truly order-independent.
  - Infeasible in the general case.

N.B. Unique-1-in-3SAT

We can encode any unique-1-in-3SAT problem in conditional variable declarations, e.g.:

$$\varphi(\mathbf{x}) = (\neg x_1 \vee \neg x_5 \vee x_3) \wedge (\neg x_2 \vee \neg x_4 \vee x_1) \wedge \ldots$$

```
static if(__traits(compiles, x1) && __traits(compiles, x5)) {
    void x3(){} // using functions so we can generate overloads
}
static if(__traits(compiles, x2) && !__traits(compiles, x4)) {
    void x1(){}
}
```

(Variables that only occur in negative clauses have to be manually pinned at true one-by-one to verify unsatisfiability.) This is NP-hard (under randomized reductions).

What else could D do?

- Explicit staging.
  - Can only reflect on properties determined at a lower stage.
  - Annotation overhead.
  - Suboptimal modularity.
  - May disallow useful code.

What else could D do?

- Track dependencies, analyze until saturation, then force decisions.
  - Source-order-independent.
  - Still dependent on nesting order.
  - Only produces consistent solutions.
  - Sometimes disallows useful code.
  - May ppick one of multiple possible solutions.
- Hybrid versions (not much explored)
  - E.g., iterate to saturation, then use tie-breakers based on source order to schedule decisions.

Analyzing until Saturation

```
static if(__traits(compiles, x)) {
    enum y = 2;
}
static if(!__traits(compiles, z)) {
    enum x = 1;
    mixin(q{enum k = 0;});
}
static if(__traits(compiles, w)) {
    enum z = 3;
}
```

```
$ dmd -o- test.d
x exists
$
```

Analyzing until Saturation

```
static if(__traits(compiles, x)) { // A: depends on B
    enum y = 2;
}
static if(!__traits(compiles, z)) { // B: depends on C
    enum x = 1;
    mixin(q{enum k = 0;}); // treated as opaque
}
static if(__traits(compiles, w)) { // depends on B
    enum z = 3;
}
```

```
A -> B <-> C
```

Analyzing until Saturation

```
static if(__traits(compiles, x)) {
    enum y = 2;
}
// z does not exist
enum x = 1;
enum k = 0;

// w does not exist


//
```

Analyzing until Saturation

```
enum y = 2; // y exists!

// z does not exist
enum x = 1;
enum k = 0;

// w does not exist


//
```

Analyzing Contradictions

```
static if(!__traits(compiles, x)) {
    enum x = 1;
}
```

Analyzing Contradictions

```
// x does not exist
enum x = 1;
```

Analyzing Contradictions

```
// x does not exist
enum x = 1; // error: x introduced anyway
```

Experimental D frontend

`https://github.com/tgehr/d-compiler`

- Explores analyzing to saturation iteratively.
- Started out as a lexer and parser for D.
- My first side project using D.
- 2011-2017. (Manual version control 2011-2013, git later.)
- Very incomplete, but does have templates with IFTI, and CTFE.
- Does some things better than DMD, does not do some other things at all.

How is it implemented?

- Compile-time “visitor pattern”.
- Single *reentrant* method for semantic analysis.
- Conceptually, every AST node is a coroutine.
  - In practice, most analysis is handled via direct recursion.
  - Explicit *scheduler* detects and resolves cyclic dependencies.
  - String mixins for handling suspension.

Compile-time “visitor pattern”

```
// from visitors.d:
mixin template Visitors(){
    mixin Semantic!(typeof(this)); // semantic analysis
    mixin Analyze; // ad-hoc visitors, often templated
    mixin CTFEInterpret!(typeof(this)); // ctfe bytecode
        interpreter
    mixin DeepDup!(typeof(this)); // create a deep duplicate of
         an AST
}
```

Compile-time “visitor pattern” (Cont.)

Then in every node class:

```
class Node{
    // ...
    mixin Visitors
}
class Expression{
    // ...
    mixin Visitors;
}
class Statement{
    // ...
    mixin Visitors;
}
// ...
```

Compile-time “visitor pattern” (Cont.)

In `semantic.d`:

```
template Semantic(T) if(is(T==Node)){
    // visitors can add fields in addition to methods
    Node rewrite; // for lazy substitution
    SemState sstate = SemState.begin;
    ubyte needRetry = false;

    void semantic(Scope sc)in{assert(sstate>=SemState.begin);}body{
        mixin(SemPrlg); // register with scheduler, return if completed, etc.
        sc.error("feature not implemented",loc);
        mixin(ErrEplg); // ``terminate'' node with error
    }

    // analysis is trapped because of circular await-relationship involving
        this node
    void noHope(Scope sc){}
}

template Semantic(T) if(is(T==Expression)){ ... }
template Semantic(T) if(is(T==Statement)){ ... }
```

Tradeoffs

Pro:

- Good conceptual encapsulation.
  - Like visitor pattern, but nicer to use.
  - Can add multiple related fields and methods.
  - Everything related to one method is in one place.
  - Can add mixin templates in parent class as customization points.
- Only one virtual call instead of two.
- All information available at compile time for introspection.
- Can implement virtual methods for different templated classes at once.

Con:

- Terrible actual encapsulation.
  - Template mixins are unhygienic.
  - Therefore everything imports everything else.
  - Visibility modifier meanings don't correspond to code organization.

Coming clean

```
// actually from visitors.d:
mixin template Visitors(){
    // workaround for DMD bug: Interpret goes first
    /*static if(is(typeof({mixin Semantic!(typeof(this));})))*/
    static if(is(typeof(this):Expression)&&!is(typeof(this):Type)) mixin Interpret!(typeof(this));// TODO: minimize and report bug
    static assert(is(TypeTuple==class));
    static if(!IsNonASTType!(typeof(this))) mixin Semantic!(typeof(this));
    // another workaround for DMD bug, other part is in expression.Node
    static if(!is(typeof(this)==Node)){
        static if(!is(typeof(this)==AggregateTy)){
            mixin Analyze; // wtf?
            mixin CTFEInterpret!(typeof(this));
        }
        static if(!is(typeof(this)==AggregateTy)) mixin DeepDup!(typeof(this));
```

Live code browsing

- Lexer
- Parser
- Semantic analysis
- Scheduler
- Bytecode Interpreter

Content warning

- I will proceed to show code with `is(typeof(...))`.
- Viewer discretion is advised.
- Don't use `is(typeof(...))`.
- `__traits(compiles, ...))` is strictly superior.
- Except my frontend does not implement `__traits`...
- `typeof` eagerly determines types for expressions that would not compile!

Demo

- Detection of contradictory/ambiguous code
- Template demo
- CTFE demo
- Improved IFTI
  - Matching template aliases
  - Improved unification features

Not explored: Attribute inference

I had also wanted the frontend to do better at attribute inference, but never got around to it.

- The frontend does not yet infer attributes.
- Attribute inference in DMD does not work for recursive functions.
- Interesting questions related to attribute inference and introspection.
- Can do something based on fixed points as there is a natural lattice structure.

`static foreach`

- `static foreach` first prototyped in my frontend
- Similar implementation worked in DMD.

Experimental Frontend Death

- With DMD 2.061, I was finally no longer able to work around the regressions: semantic.d(7834): Error: function type.BasicType.Visitors!().Semantic!(BasicType).implicitlyConvertsTo does not override any function, did you mean 'expression.Expression.Visitors!().Semantic!(Expression).implicitlyConvertTo'
- Rainer Schütze made a patch that got the frontend running under 2.074, but it was never pulled as it broke other tests.
- Some reduced issues were fixed but they invariably failed in other ways.

Thanks!

Questions?