Types and Tuples in D

DConf London '23

Timon Gehr

$$\frac{\Gamma, \texttt{T1 x} \vdash \texttt{e: T2}}{\Gamma \vdash \texttt{((T1 x)=>e): T2 delegate(T1)}}\,?$$

```
auto (x, (y, z))= t;
```

DCONF '23
August 29 - September 1

What is this talk?

Originally proposed:

- Types in D. 45min talk.
- Tuples in D. 25min talk.

What is this talk?

Originally proposed:

- Types in D. 45min talk.
- Tuples in D. 25min talk.

Further considerations:

- Types talk fits conference schedule.
- Tuples talk got majority vote.

What is this talk?

Originally proposed:

- Types in D. 45min talk.
- Tuples in D. 25min talk.

Further considerations:

- Types talk fits conference schedule.
- Tuples talk got majority vote.

⇒ **Types talk or tuples talk? Yes.**

What is this talk?

Originally proposed:

- Types in D. 45min talk.
- Tuples in D. 25min talk.

Further considerations:

- Types talk fits conference schedule.
- Tuples talk got majority vote.

$\Rightarrow$ **Types talk or tuples talk? Yes.**

Basically, I will freely associate about D's type system for hopefully ~ 45 minutes.

What is this talk?

Originally proposed:

- Types in D. 45min talk.
- Tuples in D. 25min talk.

Further considerations:

- Types talk fits conference schedule.
- Tuples talk got majority vote.

$\Rightarrow$ **Types talk or tuples talk? Yes.**

Basically, I will freely associate about D's type system for hopefully ~ 45 minutes. Disclaimer: We will look at some dark corners of the language. This is not a particularly representative sample of language rules. Here be dragons!

What is a type?

- Metadata describing the (run-time) effect of an expression.
- Often classifies values for arguments/results. (e.g., `typeof(2*3)` is `int`)
- Often tracks side effects. (e.g., `pure @safe nothrow`)
- Sometimes prescribes *memory layout*.
- Information often partially or fully erased at runtime.

What is a type?

- Metadata describing the (run-time) effect of an expression.
- Often classifies values for arguments/results. (e.g., `typeof(2*3)` is `int`)
- Often tracks side effects. (e.g., `pure @safe nothrow`)
- Sometimes prescribes *memory layout*.
- Information often partially or fully erased at runtime.
- Early type systems were mostly about proving consistency/termination.

What is a type?

- Metadata describing the (run-time) effect of an expression.
- Often classifies values for arguments/results. (e.g., `typeof(2*3)` is `int`)
- Often tracks side effects. (e.g., `pure @safe nothrow`)
- Sometimes prescribes *memory layout*.
- Information often partially or fully erased at runtime.
- Early type systems were mostly about proving consistency/termination.

$$\{X \mid X \notin X\}$$

$$(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

What is a type?

- Metadata describing the (run-time) effect of an expression.
- Often classifies values for arguments/results. (e.g., `typeof(2*3)` is `int`)
- Often tracks side effects. (e.g., `pure @safe nothrow`)
- Sometimes prescribes *memory layout*.
- Information often partially or fully erased at runtime.
- Early type systems were mostly about proving consistency/termination.

$$\{X \mid X \notin X\}$$

$$(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

```
for(;;){}
while(true){}
```

Basic Types

```
short i = 1;
int j = 2;
float x = 3.0f;
double y = 4.0;
```

Value Range Propagation

D tracks an overapproximation of the range of any integral expression.

```
int k = 1;
short i = k; // error

short j = 1; // ok
static assert(is(typeof(1) == int));

int a = ...;
ushort b = a; // error

ushort c = a & 0xffff; // ok
```

Value Range Propagation

D tracks an overapproximation of the range of any integral expression.

```
int k = 1;
short i = k; // error

short j = 1; // ok
static assert(is(typeof(1) == int));

int a = ...;
ushort b = a; // error

ushort c = a & 0xffff; // ok
```

Works using interval arithmetic.

Value Range Propagation

D tracks an overapproximation of the range of any integral expression.

```
int k = 1;
short i = k; // error

short j = 1; // ok
static assert(is(typeof(1) == int));

int a = ...;
ushort b = a; // error

ushort c = a & 0xffff; // ok
```

Works using interval arithmetic.
Fun exercise: Deriving best transformers for bitwise operations.

Type Constructors

```
int[] a = [1, 2, 3];
const(int)[] b = a;
immutable(int)[] c = [1, 2, 3];

a[0] = 2; // ok
assert(a == [2, 2, 3]);
assert(b == [2, 2, 3]);

b[0] = 3; // error
c[0] = 4; // error

b = c; // ok
```

```
int[3] c = [1, 2, 3];
int[int] d = [0:1, 1:2, 2:3]
```

Type Constructors

```
int[] a = [1, 2, 3];
const(int)[] b = a;
immutable(int)[] c = [1, 2, 3];

a[0] = 2; // ok
assert(a == [2, 2, 3]);
assert(b == [2, 2, 3]);

b[0] = 3; // error
c[0] = 4; // error

b = c; // ok
```

```
int[3] c = [1, 2, 3];
int[int] d = [0:1, 1:2, 2:3]
```

Magic, users cannot define their own type constructors.

Lvalues/Rvalues

Expressions can be lvalues or rvalues. Some types are meaningful only for lvalues:

```
enum n = cast(const)5;

pragma(msg, n); // 5
pragma(msg, typeof(n)); // const(int)
```

Lvalues/Rvalues

Expressions can be lvalues or rvalues. Some types are meaningful only for lvalues:

```
enum n = cast(const)5;

pragma(msg, n); // 5
pragma(msg, typeof(n)); // const(int)
```

```
5 = 6; // error, 5 is rvalue

int m = n; // ok
m = 6; // ok, m is lvalue

int[] a = [n]; // ok (array literal magic)
assert(a[0] == 5);
a[0] = 6; // ok, a[0] is lvalue
```

Storage Classes

Example:

```
immutable int x = 2;
```

Storage Classes

Example:

```
immutable int x = 2;
```

What people think it means:

```
(immutable int) x = 2;
```

Storage Classes

Example:

```
immutable int x = 2;
```

What people think it means:

```
(immutable int) x = 2;
```

What it actually means:

```
immutable { int x = 2; }
```

Storage Classes

Example:

```
immutable int x = 2;
```

What people think it means:

```
(immutable int) x = 2;
```

What it actually means:

```
immutable { int x = 2; }
```

OTOH, this is allowed:

```
alias Int = immutable(int);
alias Int = immutable int;

enum x = cast(immutable int)5;
```

Storage classes (cont.)

```
immutable(int) x = 2; // storage class copied to variable
```

```
immutable int x = 2;  // qualifier copied to type
```

```
immutable immutable(int) x = 2; // fully spelled out
```

Functions: Storage classes

```
immutable int* foo(int* p) => p;
```

To untrained eye, may look like:

```
immutable(int*) foo(int *p) => p;
```

Actually means:

```
immutable { int* foo(int* p) => p; }
```

```
int* foo(int* p)immutable => p; // (qualifies context pointer)
```

Can throw off people, `ref` qualifies function, *in fact relates* to the return value.

Functions: Storage classes

This is even the case for `ref`:

```
ref int foo(return ref int x) => x;
```

Is the same as:

```
ref { int foo(return ref int x) => x; }
```

```
int foo(return ref int x)ref => x;
```

Can throw off people, `ref` qualifies function, *in fact relates* to the return value.

NB: ref on delegate types

```
ref int foo(return ref int x) => x;
int foo(return ref int x)ref => x;
```

NB: ref on delegate types

```
ref int foo(return ref int x) => x;
int foo(return ref int x)ref => x;
```

```
ref int delegate(return ref int) dg = x=>x; // nope
```

NB: ref on delegate types

```
ref int foo(return ref int x) => x;
int foo(return ref int x)ref => x;
```

```
ref int delegate(return ref int) dg = x=>x; // nope
```

```
int delegate(return ref int)ref dg = x=>x; // nope
```

NB: ref on delegate types

```
ref int foo(return ref int x) => x;
int foo(return ref int x)ref => x;
```

```
ref int delegate(return ref int) dg = x=>x; // nope
```

```
int delegate(return ref int)ref dg = x=>x; // nope
```

Even though this works:

```
int foo(int x)immutable => x; // ok (if context exists)
int delegate(int)immutable dg = x=>x; // ok
```

NB: ref on delegate types

```
ref int foo(return ref int x) => x;
int foo(return ref int x)ref => x;
```

```
ref int delegate(return ref int) dg = x=>x; // nope
```

```
int delegate(return ref int)ref dg = x=>x; // nope
```

Even though this works:

```
int foo(int x)immutable => x; // ok (if context exists)
int delegate(int)immutable dg = x=>x; // ok
```

Need some movement on this.

Functions: Overloading

The static type can actually determine which code runs at runtime.

```
void foo(int x){ ... }
void foo(int[] a){ ... }

foo(2);         // calls first overload
foo([1,2,3]); // calls second overload
```

Functions: Overloading

The static type can actually determine which code runs at runtime.

```
void foo(int x){ ... }
void foo(int[] a){ ... }

foo(2);         // calls first overload
foo([1,2,3]); // calls second overload
```

The `const(int)` rvalue strikes back:

```
void foo(int x){ ... }
void foo(const(int) x){ ... }

foo(2);                 // calls first overload
foo(cast(const)2); // calls second overload
```

Fun fact: Template instantiation during overload resolution

```
void foo(int delegate(int) dg){ ... }
void foo(double delegate(double) dg){ ... }

foo((x){ pragma(msg, typeof(x)); return 2; }); // first overl.
```

```
int
double
```

Type deduction

```
auto x = 2; // type left out, inferred as ‘int x = 2;‘
```

Type deduction

```
auto x = 2; // type left out, inferred as `int x = 2;`
```

- Common misconception: `auto` is a wildcard, replaced by type.

Type deduction

```
auto x = 2; // type left out, inferred as ‘int x = 2;‘
```

- Common misconception: `auto` is a wildcard, replaced by type.
- Actually: `auto` is just needed here to introduce a declaration.

Type deduction

```
auto x = 2; // type left out, inferred as ‘int x = 2;‘
```

- Common misconception: `auto` is a wildcard, replaced by type.
- Actually: `auto` is just needed here to introduce a declaration.
- (`x=2;` would be an assignment to an existing `x`.)

Type deduction

```
auto x = 2; // type left out, inferred as ‘int x = 2;‘
```

- Common misconception: `auto` is a wildcard, replaced by type.
- Actually: `auto` is just needed here to introduce a declaration.
- (`x=2;` would be an assignment to an existing `x`.)

Type deduction

```
auto x = 2; // type left out, inferred as `int x = 2;`
```

- Common misconception: `auto` is a wildcard, replaced by type.
- Actually: `auto` is just needed here to introduce a declaration.
- (`x=2;` would be an assignment to an existing `x`.)

```
const x = 2; // type left out, inferred as `const int x = 2;`
```

Type deduction

```
auto x = 2; // type left out, inferred as 'int x = 2;'
```

- Common misconception: `auto` is a wildcard, replaced by type.
- Actually: `auto` is just needed here to introduce a declaration.
- (`x=2;` would be an assignment to an existing `x`.)

```
const x = 2; // type left out, inferred as 'const int x = 2;'
```

NB: Another way a `const` rvalue may take effect:

```
auto x = cast(const)2; // inferred as 'const(int) x = ...;'
```

Type deduction: Limitations

Only works forward:

```
auto x = [];
x ~= 1; // error: cannot append `int` to `void[]`
```

Type deduction: Limitations

Only works forward:

```
auto x = [];
x ~= 1; // error: cannot append `int` to `void[]`
```

```
int[] x = [];
x ~= 1; // ok
```

Type deduction: Limitations

Only works forward:

```
auto x = [];
x ~= 1; // error: cannot append ‘int‘ to ‘void[]‘
```

```
int[] x = [];
x ~= 1; // ok
```

(NB: `typeof([])` should be `noreturn[]`.)

Type deduction: Limitations

Can break code by returning more refined type instead:

```
class C{}
class D: C{}

C foo(){ ... }
C bar(){ ... }

void main(){
  auto result = foo();
  result = bar();
}
```

Type deduction: Limitations

Can break code by returning more refined type instead:

```
class C{}
class D: C{}

D foo(){ ... }
C bar(){ ... }

void main(){
  auto result = foo();
  result = bar(); // error
}
```

Type deduction: Limitations

Can break code by returning more refined type instead:

```
class C{}
class D: C{}

D foo(){ ... }
C bar(){ ... }

void main(){
  C result = foo();
  result = bar(); // ok
}
```

Desirable properties of type systems

- Simplicity
- Ergonomics (Readability/Writability)
- Expressiveness
- Conciseness
- Extendability
- Inference
- Good Diagnostics
- Orthogonality, no magic
- Modularity
- Separate compilation
- Type safety
- Fast type checking
- Compilation (Simple, fast, results in performant code)
- Eraseability
- Type-directed syntactic sugar
-

Expressiveness tradeoffs

Pro:

- Code is more self-documenting.
- More metadata accessible by tooling.
- Detects errors more efficiently than tests would.

Con:

- Tests needed anyway.
- Can lead to annotation overhead.
- More likely to become non-modular.

Expressiveness tradeoffs

Pro:

- Code is more self-documenting.
- More metadata accessible by tooling.
- Detects errors more efficiently than tests would.

Con:

- Tests needed anyway.
- Can lead to annotation overhead.
- More likely to become non-modular.

Tendency to *evolve* from less expressive type systems to more expressive ones.

Expressiveness tradeoffs

Pro:

- Code is more self-documenting.
- More metadata accessible by tooling.
- Detects errors more efficiently than tests would.

Con:

- Tests needed anyway.
- Can lead to annotation overhead.
- More likely to become non-modular.

Tendency to *evolve* from less expressive type systems to more expressive ones.
Keeping language simple throughout this evolution is challenging.

Expressiveness tradeoffs

Pro:

- Code is more self-documenting.
- More metadata accessible by tooling.
- Detects errors more efficiently than tests would.

Con:

- Tests needed anyway.
- Can lead to annotation overhead.
- More likely to become non-modular.

Tendency to *evolve* from less expressive type systems to more expressive ones.
Keeping language simple throughout this evolution is challenging.
However, keep in mind, there *are* simple systems that are very expressive.

Modularity

- Ability to factor out patterns into functions/aggregates.
- Type information must be able to cross function/aggregate boundaries.

Example: Successful refactoring

```
void main(){
  int x = 2;
  int y = x+3;
  ...
}
```

Example: Successful refactoring

```
void main(){
  int x = 2;
  int y = x+3;
  ...
}
```

```
int f(int a, int b) => a + b;

void main(){
  int x = 2;
  int y = f(x,3);
  ...
}
```

Example: Unsuccessful refactoring

```
void main() {
  auto x = readln().strip.to!int;
  ubyte y = x & 0xff;
  ...
}
```

Example: Unsuccessful refactoring

```
void main() {
  auto x = readln().strip.to!int;
  ubyte y = x & 0xff;
  ...
}
```

```
int mask_low(int x, int nbits) => x & ((1 << nbits) - 1);

void main() {
  auto x = readln().strip.to!int;
  ubyte y = mask_low(x, 8);
  // error: cannot convert ‘int‘ to ‘ubyte‘
  ...
}
```

Example: Unsuccessful refactoring

```
void main() {
  auto x = readln().strip.to!int;
  ubyte y = x & 0xff;
  ...
}
```

```
int mask_low(int x, int nbits) => x & ((1 << nbits) - 1);

void main() {
  auto x = readln().strip.to!int;
  ubyte y = mask_low(x, 8);
  // error: cannot convert `int` to `ubyte`
  ...
}
```

Issue: No way to pass value ranges across function boundaries.

Type safety

- Types define invariants on program state.
  - *safe* values.
  - aliasing constraints.
  - ...

Type safety

- Types define invariants on program state.
  - *safe* values.
  - aliasing constraints.
  - ...
- Operations have defined behavior if invariants hold.

Type safety

- Types define invariants on program state.
  - *safe* values.
  - aliasing constraints.
  - ...
- Operations have defined behavior if invariants hold.
- Operations preserve invariants.

Example: Violation of type safety

```
import std.stdio;

void main(){
    bool b = void;
    if(b){
        writeln("b is true");
    }
    if(!b){
        writeln("b is false");
    }
}
```

Type safety tradeoffs

Pro:

- Programs do not lie. Types act as compiler-checked documentation.
- Higher confidence in refactoring.
- Can support *safety guarantees*.

Con:

- Perfectly fine programs may be rejected.
- Leads to pressure to increase expressiveness and modularity.
- As a consequence, usually leads to higher annotation overhead.

D's approach to type safety

Lack of type safety is essentially treated as a *side effect*.

- `@system`: Types are suggestions, programmers are infallible.
- `@trusted`: Type safe interface, programmer is infallible, caller is restricted but adversarial.
- `@safe`: Type safe. Programs have defined behavior, programmers are adversarial, callers are restricted but adversarial.

D's approach to type safety

Lack of type safety is essentially treated as a *side effect*.

- `@system`: Types are suggestions, programmers are infallible.
- `@trusted`: Type safe interface, programmer is infallible, caller is restricted but adversarial.
- `@safe`: Type safe. Programs have defined behavior, programmers are adversarial, callers are restricted but adversarial.

Overall goal:

- Type safety and low-level control coexist in the same program.
- `@trusted` functions or the language are to blame for any holes in `@safe`.
- Introduces a sort of compile-time modularity for memory safety.
- Run-time effects of memory unsafety still unrestricted.

Effect tracking for functions

D also prevents other side effects. E.g., function types have an *effect* annotation.

- `@safe`: no memory corruption / undefined behavior.
- `pure`: no mutation of global state (**mostly**)
- `nothrow`: no throwing of `Exception`.
- `@nogc`: no GC pauses, no GC allocation.

Effect tracking for functions

D also prevents other side effects. E.g., function types have an *effect* annotation.

- `@safe`: no memory corruption / undefined behavior.
- `pure`: no mutation of global state (**mostly**)
- `nothrow`: no throwing of `Exception`.
- `@nogc`: no GC pauses, no GC allocation.

Other effects one could prevent (but D does not attempt to):

- Dynamic memory allocation
- Irreversibility
- Errors
- Nontermination
- ...

Some practical challenges for D's effect tracking

- `pure` is underspecified. E.g., when can a function be `@trusted pure`?
- Often, people want to check their code non-transitively.
- `@trusted` is tricky to use correctly, hard to specify in a fine-grained way which operations one wants to trust, particularly in generic code.
- As a result, even the standard library uses `@trusted` lambdas with an unsafe interface. `(()@trusted => cast(...)...)();`
- Should probably be made more usable.

NB: Inference and Defaults

Inference:

- `auto` return functions infer effects.
- Functions in template instances infer effects.
- Currently no other way to turn on inference if return type specified.
- Inference currently unreliable when there is mutual recursion.

NB: Inference and Defaults

Inference:

- `auto` return functions infer effects.
- Functions in template instances infer effects.
- Currently no other way to turn on inference if return type specified.
- Inference currently unreliable when there is mutual recursion.

`@safe` by default:

- DIP 1028 proposed making `@safe` the default.
- Ultimately retracted, because that's not a sane default for `extern(C)` function prototypes.

NB: Inference and Defaults

Inference:

- `auto` return functions infer effects.
- Functions in template instances infer effects.
- Currently no other way to turn on inference if return type specified.
- Inference currently unreliable when there is mutual recursion.

`@safe` by default:

- DIP 1028 proposed making `@safe` the default.
- Ultimately retracted, because that's not a sane default for `extern(C)` function prototypes.

Switching defaults:

- Currently there is no way to switch defaults without turning off inference.

Effect tracking on lvalues

Type qualifiers restrict or enhance allowed effects taking place on their values.

- (unqualified) `T`: can be mutated, may not be shared
- `immutable(T)`: cannot be mutated globally, can be shared freely
- `const(T)`: cannot be mutated through current reference
- `inout(T)`: wildcard, more later
- `shared(T)`: may be shared, must properly synchronize access

Qualifiers are transitive. No mutation really means no mutation.

Non-modularity of Effect Tracking

Challenge: Create a delegate that composes two `int delegate(int)`s:

```
alias ComposeType =
  int delegate(int)
    delegate(
      int delegate(int),
      int delegate(int),
    )@safe pure nothrow;

ComposeType compose = (
  int delegate(int) a,
  int delegate(int) b,
)@safe pure nothrow{
  return x=>a(b(x));
};
```

Non-modularity of Effect Tracking

Challenge: Create a delegate that composes two `int delegate(int)`s:

```
alias ComposeType =
  int delegate(int)
    delegate(
      int delegate(int),
      int delegate(int),
    )@safe pure nothrow;

ComposeType compose = (
  int delegate(int) a,
  int delegate(int) b,
)@safe pure nothrow{
  return x=>a(b(x));
};
```

Issue: No type of delegate can correctly preserve `@safe pure nothrow`.

Non-modularity of Effect Tracking

Challenge: Create a delegate that composes two `int delegate(int)`s:

```
alias ComposeType =
  int delegate(int)
    delegate(
      int delegate(int),
      int delegate(int),
    )@safe pure nothrow;

ComposeType compose = (
  int delegate(int) a,
  int delegate(int) b,
)@safe pure nothrow{
  return x=>a(b(x));
};
```

Issue: No type of delegate can correctly preserve `@safe pure nothrow`.

Pure *modularity* problem: if we inline `compose` into its callers, this works.

Less contrived example: (virtual) opApply

```
class ASTNode{
  int componentsImpl(scope int delegate(ASTNode) dg){
    return dg(this);
  }
  auto components()return scope{
    struct Components{
      ASTNode self;
      int opApply(scope int delegate(ASTNode) dg){
        return self.componentsImpl(dg);
      }
    }
    return Components(this);
  }
}
```

- Might reach for opApply, as ranges less well-suited for traversing a tree.
- Supporting all combinations of qualifiers is a heavy burden on library authors.

More contrived example: Identity function

```
int delegate(int) delegate(int delegate(int)) id = dg=>dg;
```

- Implementation above strips all function qualifiers.
- In practice can get relatively far with templated functions.

```
    T id(T)(T dg)=>dg;
```

- Issue: It's either templates or virtual calls. :(
- Templates are usually the better option, even ignoring qualifiers. (IFTI does not match lambdas very reliably.)
- Leads to template bloat, larger compile times.

This has happened before!

The same issue exists for the mutability qualifiers.

`inout` was invented to address this.

```
inout(int)* id(inout(int)* p) => p;
```

Replaces:

```
int* id(int* p) => p;
const(int)* id(const(int)* p) => p;
immutable(int)* id(immutable(int)* p) => p;
```

Plus, `&id` is a single virtual function.

inout qualifier

Issue:

- There is only one inout qualifier.
- There can be multiple functions.
- Which function relates to which inout annotation in any given context is determined by ad-hoc rules, and they are inconsistent.

```
alias F = inout(int)* delegate(inout int);

F id(F dg) => dg;
auto dg = id((immutable int)=>new immutable(int)); // error
```

inout qualifier (cont.)

Furthermore, `inout`-qualified data cannot be stored in aggregates.

```
auto foo(inout(int)* p){
  struct S{
    inout(int)* p;
  }
  auto s = S(p);
  // ...
}
```

Among other things, this means `inout` is incompatible with most of the Phobos range API. It permits zero data abstraction.

Breaking `inout`: First try

Observations:

- Nested functions share `inout` with their enclosing function (if any).
- Higher-order functions within return types have their own `inout`.

This is inconsistent! However, the obvious way to exploit it has been patched:

Breaking inout: First try

Observations:

- Nested functions share `inout` with their enclosing function (if any).
- Higher-order functions within return types have their own `inout`.

This is inconsistent! However, the obvious way to exploit it has been patched:

```
inout(int)* delegate(inout(int)*) foo(inout(int)* x){
    inout(int)* bar(inout(int)* y){ return x; }
    return &bar; // note: ok. 8)
}
void main(){
    pragma(msg, typeof(&foo));
    //inout(int)* delegate(inout(int)*) function(inout(int)* x)

    int x=2;
    pragma(msg, typeof(foo(&x)));
    //const(int)* delegate(const(int)*)     (!)
}
```

NB: The generic templated identity function

```
T id(T)(T arg) => arg;
// clearly, this will be true for all T:
enum preservesType(T) = is(typeof(id(T.init))==T);
```

NB: The generic templated identity function

```
T id(T)(T arg) => arg;
// clearly, this will be true for all T:
enum preservesType(T) = is(typeof(id(T.init))==T);
```

Not so fast! Ironically, an effort targeted towards preserving qualifiers through an identity function broke the basic property that an identity function should preserve the type.

```
alias T = inout(inout(int)* delegate(inout(int)*));
static assert(preservesType!T); // fails!
```

Circumventing the patchwork

```
@safe:
int a;
immutable(int) b=2;

auto foo(inout(int)* y){
  inout(int)* bar(inout(int)* p){
    return y;
  }
  return ()=>&bar;
}
void main(){
  int* y=foo(&b)()(&a);
  *y=3;
  assert(&b is y); // passes. ouch.
  assert(b is *&b); // fails!
}
```

return parameter attribute

```
ref int foo(return ref x) => x;
```

The return attribute has some similarities to inout:

- Interaction with nested and higher-order functions seems a bit tricky.
- return scope pointers and return references cannot be individually stored as struct fields.
- As there is only one name for a return attribute, lifetimes tend to get conflated.

However, as it does not require substitution in the result type and return scope can be pushed up structs, return avoids some inout pitfalls.

Qualifier Polymorphism (Suggested Extension)

A good way to express this kind of thing would be *polymorphism*. E.g., instead of having only one name `inout`, add an explicit polymorphic parameter:

```
qual(int)* id[mutability qual](qual(int)* p)=>p;
// still works the same with higher-order functions:
qual(int)* delegate(qual(int)*)
id[mutability qual](
  qual(int)* delegate(qual(int)*) dg
){
  return dg;
}
// can easily express case where argument is polymorphic:
qual1(int)* delegate[mutability qual1](qual1(int)*)
id(
  qual2(int)* delegate[mutability qual2](qual2(int)*) p
){
  return p;
}
```

Effect Polymorphism (Suggested Extension)

```
alias ComposeType =
  int delegate(int)eff
    delegate[effect eff](
      int delegate(int)eff,
      int delegate(int)eff,
    )@safe pure nothrow @nogc;

ComposeType compose = [effect eff](
  int delegate(int)eff a,
  int delegate(int)eff b,
)@safe pure nothrow @nogc{
  return x=>a(b(x));
};
```

Effect Polymorphism (Suggested Extension)

```
class ASTNode{
  int compImpl[effect e](scope int delegate(ASTNode)e dg){
    return dg(this);
  }
  auto components()return scope{
    struct Components{
      ASTNode self;
      int opApply[effect e](scope int delegate(ASTNode)e dg){
        return self.compImpl(dg);
      }
    }
    return Components(this);
  }
}
```

CTFE

- CTFE semantics technically part of the type system.
- But CTFE functions cannot manipulate types directly.
- Should probably be fixed.

Forward references and introspection

```
static if(!is(typeof(x))) enum x = 2;
```

Forward references and introspection

```
static if(!is(typeof(x))) enum x = 2;
```

```
static if(is(typeof(y))) enum x = 2;
static if(!is(typeof(x))) enum y = 2;
```

Forward references and introspection

```
static if(!is(typeof(x))) enum x = 2;
```

```
static if(is(typeof(y))) enum x = 2;
static if(!is(typeof(x))) enum y = 2;
```

```
static if(!is(typeof(x))) enum y = 2;
static if(!is(typeof(y))) enum x = 2;
```

Dependency on compilation order

```
module a;
import b;
static if(!is(typeof(x))) enum y = 2;
static if(is(typeof(y))) pragma(msg, "y defined");
```

Dependency on compilation order

```
module a;
import b;
static if(!is(typeof(x))) enum y = 2;
static if(is(typeof(y))) pragma(msg, "y defined");
```

```
module b;
import a;
static if(!is(typeof(y))) enum x = 2;
static if(is(typeof(x))) pragma(msg, "x defined");
```

Dependency on compilation order

```d
module a;
import b;
static if(!is(typeof(x))) enum y = 2;
static if(is(typeof(y))) pragma(msg, "y defined");
```

```d
module b;
import a;
static if(!is(typeof(y))) enum x = 2;
static if(is(typeof(x))) pragma(msg, "x defined");
```

```
$ dmd -o- a.d b.d
x defined
$ dmd -o- b.d a.d
y defined
```

Experimental frontend

- I partially built a D frontend with explicit dependency tracking

Experimental frontend

- I partially built a D frontend with explicit dependency tracking
  - `https://github.com/tgehr/d-compiler`

Experimental frontend

- I partially built a D frontend with explicit dependency tracking
  - `https://github.com/tgehr/d-compiler`
- It catches ambiguities and contradictions.

Experimental frontend

- I partially built a D frontend with explicit dependency tracking
  - `https://github.com/tgehr/d-compiler`
- It catches ambiguities and contradictions.
- It stopped compiling with DMD 2.061.

Experimental frontend

- I partially built a D frontend with explicit dependency tracking
  - `https://github.com/tgehr/d-compiler`
- It catches ambiguities and contradictions.
- It stopped compiling with DMD 2.061.
- Ironically, the reason was going too crazy with code generation and introspection.

~~TypeTuple~~ AliasSeq

```
alias AliasSeq(T...)=T;  // or ‘import std.meta: AliasSeq;‘
void main(){
  AliasSeq!(int, int, int) s = AliasSeq!(1, 2, 3);

  auto a = [s];
  assert(a == [1, 2, 3]);

  auto p = &s; // error

  auto typeof(s) foo()=>s; // error, cannot return sequence

  int sum = s[0]+s[1]+s[2];
}
```

~~TypeTuple~~ AliasSeq

```d
alias AliasSeq(T...)=T;  // or ‘import std.meta: AliasSeq;‘
void main(){
  AliasSeq!(int, int, int) s = AliasSeq!(1, 2, 3);

  auto a = [s];
  assert(a == [1, 2, 3]);

  auto p = &s; // error

  auto typeof(s) foo()=>s; // error, cannot return sequence

  int sum = s[0]+s[1]+s[2];
}
```

- AliasSeq can contain arbitrary aliases
- If all are types, can use like a type, to declare multiple variables
- Metaprogramming tool

~~TypeTuple~~ AliasSeq

```d
alias AliasSeq(T...)=T;  // or `import std.meta: AliasSeq;`
void main(){
  AliasSeq!(int, int, int) s = AliasSeq!(1, 2, 3);

  auto a = [s];
  assert(a == [1, 2, 3]);

  auto p = &s; // error

  auto typeof(s) foo()=>s; // error, cannot return sequence

  int sum = s[0]+s[1]+s[2];
}
```

- AliasSeq can contain arbitrary aliases
- If all are types, can use like a type, to declare multiple variables
- Metaprogramming tool

AliasSeq is not a tuple type. This is because it is not a type.

What is a tuple?

- In general: Product type.
  - Aggregate of multiple values.
  - Can get back out whatever we stuck in.
  - Is no more than the sum [sic] of its parts.

What is a tuple?

- In general: Product type.
  - Aggregate of multiple values.
  - Can get back out whatever we stuck in.
  - Is no more than the sum [sic] of its parts.
- Sounds a bit like `struct`.

What is a tuple?

- In general: Product type.
  - Aggregate of multiple values.
  - Can get back out whatever we stuck in.
  - Is no more than the sum [sic] of its parts.
- Sounds a bit like `struct`.
- `std.typecons.tuple`?

What is a tuple?

- In general: Product type.
  - Aggregate of multiple values.
  - Can get back out whatever we stuck in.
  - Is no more than the sum [sic] of its parts.
- Sounds a bit like `struct`.
- `std.typecons.tuple`?

Simplified:

```
struct Tuple(T...){
  T expand;
  alias T this;
}
auto tuple(T...)(T args)=>Tuple!T(args);
```

What to use it for?

- Creating ad-hoc groupings when calling into generic code.
- For very localized data structures within a function or in small scripts.
- Returning multiple values.

Tuples: What is missing?

Sticking things in.

```
auto foo(inout(int)* w){
  auto t = tuple(w,w); // error
  return t;
}
```

Tuples: What is missing?

Getting things back out.

```
int* foo(return scope int* a){
  scope int* b = ...;
  scope t = tuple(a, b);
  return t[0]; // error
}
```

Tuples: What is missing?

Destructuring.

```
import std.typecons: tuple;
auto t = tuple(1, tuple(2, 3));
auto (x, (y, z)) = t;
```

A bit of History

9rnsr commented on Aug 27, 2011 · Contributor

http://d.puremagic.com/issues/show_bug.cgi?id=6365

Based on #321, declare multiple variables at once, and initialize them in the corresponding tuple fields.

Syntax:

```
TupleDeclaration:
    StorageClasses ( TupleTypeList ) = Initializer ;
    ( TupleTypeList ) = Initializer ;

TupleTypeList:
    TupleType
    TupleType , TupleTypeList

TupleType:
    StorageClasses BasicType Declarator
    BasicType Declarator
    StorageClasses Identifier
    Identifier
```

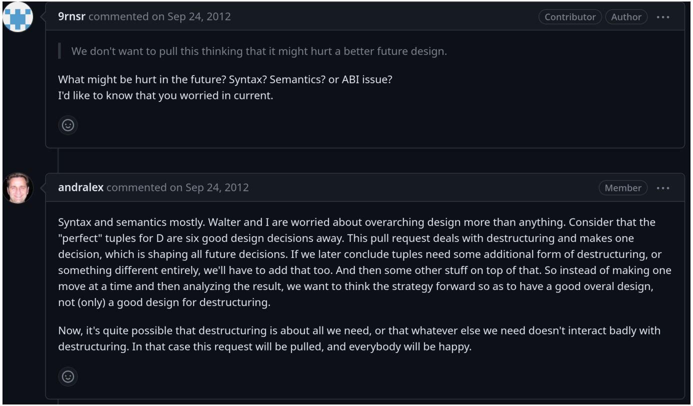
TupleTypeList is similar to ForeachTypeList.

If you want to write left parenthesis at beginning, you cannot omit the type or storage class of first variable.

```
(x, y) = getCoordinate();  // NG, this is not multiple var declaration
(auto x, y) = getCoordinate();  // OK
(double x, y) = getCoordinate();  // OK, types are specified explicitly
```

https://github.com/dlang/dmd/pull/341

A bit of History



9rnsr commented on Sep 24, 2012 — Contributor · Author

> We don't want to pull this thinking that it might hurt a better future design.

What might be hurt in the future? Syntax? Semantics? or ABI issue?
I'd like to know that you worried in current.

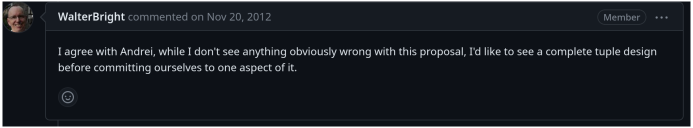
andralex commented on Sep 24, 2012 — Member

Syntax and semantics mostly. Walter and I are worried about overarching design more than anything. Consider that the "perfect" tuples for D are six good design decisions away. This pull request deals with destructuring and makes one decision, which is shaping all future decisions. If we later conclude tuples need some additional form of destructuring, or something different entirely, we'll have to add that too. And then some other stuff on top of that. So instead of making one move at a time and then analyzing the result, we want to think the strategy forward so as to have a good overal design, not (only) a good design for destructuring.

Now, it's quite possible that destructuring is about all we need, or that whatever else we need doesn't interact badly with destructuring. In that case this request will be pulled, and everybody will be happy.

`https://github.com/dlang/dmd/pull/341`

A bit of History



`https://github.com/dlang/dmd/pull/341`

What's wrong with this proposal?

```
auto (x, y) = tuple(1, "123");
(int x, y) = tuple(1, "123");
```

What's wrong with this proposal?

```
auto (x, y) = tuple(1, "123");
(int x, y) = tuple(1, "123");
```

- Looks a bit like it should mean: declare new `x`, assign to existing `y`.

What's wrong with this proposal?

```
auto (x, y) = tuple(1, "123");
(int x, y) = tuple(1, "123");
```

- Looks a bit like it should mean: declare new `x`, assign to existing `y`.

Otherwise: IMNSHO: Slighly generalize it and ship it.

```
auto (x, (y, z)) = tuple(1, "123");
```

Slipped past Walter and Andrei

```
auto a = [tuple(1, "2"), tuple(3, "4"), tuple(4, "5")];
foreach(x, y; a) {
  writeln(x, " ", y);
}
```

```
0 Tuple!(int, string)(1, "2")
1 Tuple!(int, string)(3, "4")
2 Tuple!(int, string)(4, "5")
```

Slipped past Walter and Andrei

```
auto a = [tuple(1, "2"), tuple(3, "4"), tuple(4, "5")];
foreach(x, y; a.map!(x => x)) {
  writeln(x, " ", y);
}
```

```
1 2
3 4
4 5
```

Tuples: What is the hold-up?

- Benefit of `static foreach`: very small design space.
- In contrast, for "full tuple design": many moving parts.
- Has to fit into existing language.
- Unfortunately, different syntax preferences exist.
  - Standard would be `(1, 2, 3)`.
    - Can be used soon, as comma operator is deprecated as an expression.
    - More tricky to integrate with function parameter lists.

Tuples: What is the hold-up?

- Benefit of `static foreach`: very small design space.
- In contrast, for "full tuple design": many moving parts.
- Has to fit into existing language.
- Unfortunately, different syntax preferences exist.
  - Standard would be `(1, 2, 3)`.
    - Can be used soon, as comma operator is deprecated as an expression.
    - More tricky to integrate with function parameter lists.
  - DIP32 proposed `{1, 2, 3}`, to avoid comma operator clash.
    - Clashed with delegates instead.
    - `{a, b} => a+b` syntax (matches single tuple)

Tuples: What is the hold-up?

- Benefit of `static foreach`: very small design space.
- In contrast, for "full tuple design": many moving parts.
- Has to fit into existing language.
- Unfortunately, different syntax preferences exist.
  - Standard would be `(1, 2, 3)`.
    - Can be used soon, as comma operator is deprecated as an expression.
    - More tricky to integrate with function parameter lists.
  - DIP32 proposed `{1, 2, 3}`, to avoid comma operator clash.
    - Clashed with delegates instead.
    - `{a, b} => a+b` syntax (matches single tuple)
  - Some people want to reuse `[1, 2, 3]`.
    - A whole "unify tuples and arrays" discussion.
    - Does not work in D.
    - Static arrays are already built to interoperate with dynamic arrays.
    - Dynamic array are reference types.
    - Heterogeneous slices don't seem to make all that much sense.
    - Special case `void[]`, etc.
    - ...

Quest for "full tuple design"

Decisions needed:

- How to unpack.
  - Syntax.
  - Lifetime issues.
    - Unpack by copy?
    - Unpack by move?

```
auto (a, b) = tuple(1, "2");
(int a, string b) = tuple(1, "2");

foreach((x, y); [tuple(1, "2"), tuple(3, "4"), tuple(5, "6")]){
    writeln(x, " ", y);
}
```

```
auto (x, y) = (1, "2");
```

Quest for "full tuple design"

Decisions needed:

- Where to unpack.
  - Local declarations
  - Assignments
  - `foreach`. Some baggage exists.
  - Function parameter lists.
    - Implicitly match top level as the parameter list?

```
int x = 1, y = 2;
(x, y) = (y, x);
assert((x, y) == (2, 1));

writeln(sum(zip(a,b).map!((x, y) => x*y)));    // ?

writeln(sum(zip(a,b).map!(((x, y)) => x*y)));
```

```
auto (x, y) = (1, "2");
```

Quest for "full tuple design"

Decisions needed:

- Currently, tuple indexing works via `alias this`.
- `static opIndex`?
- `static opSlice`?
  - Otherwise, `t[i..j]` is a sequence, not a tuple.
- `opCallRight` for unpacking?

```
auto (x, y) = (1, "2");
```

Quest for “full tuple design”

Decisions needed:

- Tuple literals, tuple types.
- Named tuple components.
  - Interaction with named arguments.

```
auto (x, y) = (1, "2");
```

Tuples: Suggestions

1. Fix tuple literal syntax. Ideally: `(1, 2, 3)` (or `[1, 2, 3]`, but tricky).
2. Finish tuple unpacking implementation, match tuple literal syntax.
   - `auto (x,y)= (1,2);`
   - In particular includes questions on lifetimes.
   - Seems tricky to avoid copies. Happy to chat about it.
3. Pull tuple unpacking ASAP.
4. Continue the bikeshedding on everything downstream from that.

WIP tuple implementation

`github.com/tgehr/DIPs/blob/tuple-syntax/DIPs/DIP1xxx-tg.md`
`https://github.com/tgehr/dmd/tree/tuple-syntax`

DIP and implementation still need updates.
Help welcome!

Hackathon projects

- Implement some of the suggestions from this talk.
- Improve type system soundness.
- Fix delegate context qualifiers.
- Work on tuple design.

Questions?