Value Lifetimes and Move Semantics

DConf London '24

Timon Gehr

What is this talk?

- Some more of my ideas about how to evolve the language.
- Largely aspirational. (For reference, this is D 2.109.1.)
- You may however learn something about existing features and their limitations.
- If you'd like to contribute to D, maybe you'll find a project here.

Locations vs Values

Locations

- Stack locations.
- Heap locations.

Values

- Abstract concept.
- A value is stored in one or more locations.

Lifetimes of Locations and Values

Location Lifetimes

- Stack. Nested.
- GC heap. Virtually infinite lifetime.
- Manual. E.g., malloc/free.

Value Lifetimes

- Delimited between constructor and destructor.
- May overlap arbitarily.

Copies vs Moves

Copies

- Default behavior, copied from C.
- ```
  auto a = b; // ‘b‘ copied to ‘a‘
  writeln(a, " ", b); // can use both ‘a‘ and ‘b‘
  ```
- New value is constructed to match old value.

Moves

- Value is moved into new location.
- Currently:
  ```
  auto a = move(b); // ‘b moved to ‘a‘
  writeln(a); // only supposed to use ‘a‘ now
  ```
- Actually moves value of `b` into `a` and reinitializes `b` with `init` value.
  ```

NB. Constructors and destructors

```
struct S{
  @disable this();
  this(int){ writeln("S constructed"); }
  ~this(){ writeln("S destroyed"); }
}

void main(){
  auto s=immutable(S)(0); // S constructed
  // S live here
} // S destroyed
```

Limitations of constructors and destructors

- Seems kind of basic?
- Importantly: As far as I am aware, stack variables are always destroyed now though.
- However: The previous slide was still aspirational.

Constructors and type qualifiers

Stack variables can be accessed before being constructed:

```
@safe:
int[immutable(int)*] cache;
class C{
    immutable int x;
    void foo(){
        if(&x in cache) assert(cache[&x]==x); // fail
        cache[&x]=x;
    }
    this(int x){
        foo();
        this.x=x;
    }
}
void main(){
    auto c=new C(2);
    c.foo();
}
```

Destructors and type qualifiers

```
int* pun(immutable(int)* q)@safe{
  int *r;
  struct S{
    int* p;
    @disable this();
    this(immutable(int)* p)immutable{ this.p=p; }
    ~this(){ r=p; }
  }
  {auto s=immutable(S)(q);}
  return r;
}

void main()@safe{
  immutable x=new immutable(int)(2);
  int* p=pun(x);
  pragma(msg, typeof(x)); // immutable(int*)
  writeln(*x); // 2
  *p=3;
  //assert(p is x);
  writeln(*x); // 2
}
```

Destructors and type qualifiers

```
int* pun(immutable(int)* q)@safe{
  int *r;
  struct S{
    int* p;
    @disable this();
    this(immutable(int)* p)immutable{ this.p=p; }
    ~this(){ r=p; }
  }
  {auto s=immutable(S)(q);}
  return r;
}

void main()@safe{
  immutable x=new immutable(int)(2);
  int* p=pun(x);
  pragma(msg, typeof(x)); // immutable(int*)
  writeln(*x); // 2
  *p=3;
  assert(p is x);
  writeln(*x); // 3
}
```

Total destruction

Stack variables can be destroyed without being constructed:

```
auto foo(){
  int x=2;
  struct T{
    this(int){ writeln("T constructed"); }
    ~this(){ writeln("T destroyed: ",x); }
  }
  return T(3);
}

struct S{
  typeof(foo()) t;
  this(int){
    throw new Exception("oops.");
    t=foo();
  }
}
```

Memory safety vs crash safety

Safety

- Safety means “bad things do not happen”.
- Safety is often qualified.

Memory safety

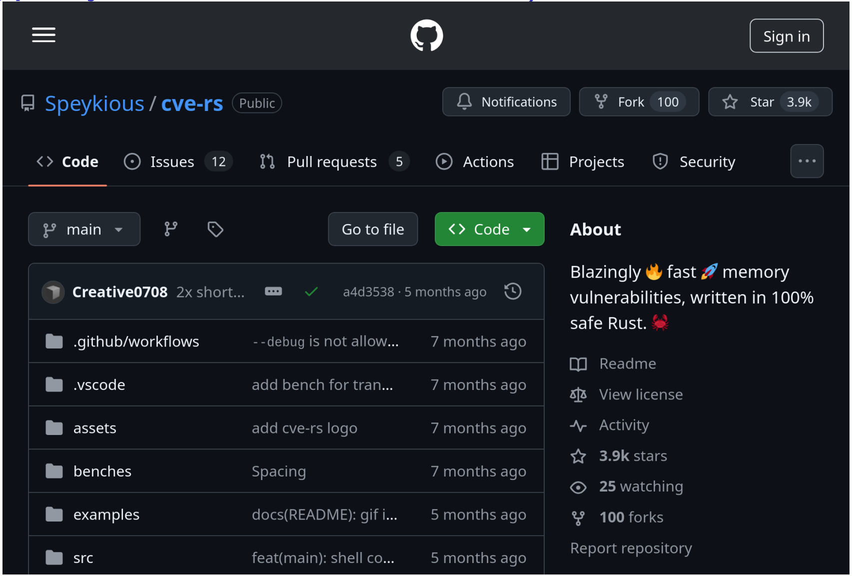
- Memory safety means all behavior of a function is defined.
- Type safety: “Well-typed programs are memory safe.”
- Languages like D or Rust are not type safe.
- Common aim: Conditional type safety.

Memory safety is a lowest-common denominator notion of safety, it is required for any other kind of safety. Memory unsafe programs often suffer from remote code execution exploits.

Dealing with unsoundness

- Good rule of thumb: If it is not formally verified, it is probably unsound.
- If it is formally verified, there is probably a bug in the specification.
- If there is no bug in the specification, there is probably still some other part of your system that is not formally verified.
- If your entire system is formally verified, there is still the possibility of holes in your formal system.
- Hence software licenses usually say "ABSOLUTELY NO WARRANTY OF FITNESS FOR ANY PARTICULAR PURPOSE".
- "Just don't write bugs" is a surprisingly common attitude, but delusional for basically any non-trivial system, without formal methods.
- Type systems are a lightweight form of formal methods that are widely deployed. Formally verifying them is consequential.

(Other type systems are unsound, too)



`@trusted`

- Common misconception: `@trusted` means "memory unsafe, do not check this".
- The opposite is the case, it means "memory safe, but not checked".
- It is the *precondition* for D's aspired conditional type safety guarantee.
- The precondition is vacuously satisfied if you do not write `@trusted` code. Hence `@safe` yields a type safe subset of the language. (Aspirationally.)

`@live`

- `@live` does not improve conditional type safety. It does not give additional safety guarantees for code that is already `@safe`.
- It may or may not help you with improving memory safety of a specific piece of `@system`/`@trusted` code. YMMV.
- Consider it to be a linting tool helpful with a specific, restrictive way of writing code.
- This is not like Rust's borrow checker even though it technically checks borrows.
- May be a good basis for future type system extensions that do give conditional type safety guarantees.

Spreading a bit of GC FUD

(Live Demo.)

```d
import std.stdio, std.random;

struct S{
    ubyte[1024] payload;
    S* next;
}

void main(){
    S* head = new S;
    S* curr = head;
    int i=0;
    for(;; i++){
      curr.next = new S;
      curr = curr.next;
      if(!(i%1000000)) writeln(curr);
    }
    //writeln(*head);
}
```

Garbage collection is undecidable

- Technically, tracing GC is an approximate heuristic.
- It says data can be deallocated when it is no longer reachable.
- Actually, data can be deallocated when it will no longer be accessed.
- Compilers can and do sometimes optimize a program that has a memory leak to one that does not.
- The example program might blow up sporadically in a hard to explain way if a false pointer appears. (Less likely on 64 bit.)

Tracing GC

- In my experience: If I
  - Use the GC.
  - Do not use type qualifiers.
- Then memory safety is very rarely a concern.
- The main potential source of unsafety is escaping stack references.

Reasons to use `@safe`

- Therefore, I think the most important reasons to use `@safe` are:
  - Simple: When working in a team, to ensure people use the language in the "simple" way, that is clearly type safe. (See Robert's "Simple @safe D" talk from DConf'23.)
  - Expressive: Trying to do error-prone things like taking stack references and manually managing memory. While being drunk and/or tired. Without any worry it will cause a week-long debugging frenzy in front of the release deadline.
- One of these is more interesting, but the other one is both easier and more important for getting taken seriously.

Are we type safe yet?

For the simple `@safe` D direction, I think we need:

- Initialization safety.
- Fully reliable stack reference detection. (E.g., slicing static arrays.) Maybe even promote them to the heap.
- A GC that works better both inside and outside of single-threaded batch programs (thanks Steven/Amaury!)
- Find a way to deal with `inout`.
- Fix type checking for qualified delegate contexts.
- Fix closure allocation in loops.
- Think about ways to validate DMD against a formally-verified implementation of the fully lowered D subset. E.g., guided test case generation.
- Fix all the other bugs.

Are we expressive yet?

For the expressive direction, I think DIP1000 has significant limitations while also being quite confusing at first. Probably we can find a better tradeoff.

Things to explore:

- Move semantics. (DIP1040)
- Move constructors. (DIP1040)
- Escape checking for non-nested lifetimes.
- Multiple indirections.
- Effect polymorphism. (Dennis had to break the type system!)
- Attribute inference for recursive functions (hard).
- Or even just conditional attributes?
- Better escape analysis in the frontend.
- @nogc exceptions. (DIP1008)
- Ownership/isolated.

Working with what we have

- "One indirection ought to be enough for anyone". Tuple of arrays.
- @system fields.
- Fake stack references. (E.g., Dennis' arena design)
- Runtime checks instead of or to complement type system features.
  - After all, range checks are how we took care of buffer overruns. We can also do this for use after free.
- `scope`/`pure`/`static` callbacks with DIP1000.

```
    smartPointer.access!((ref x){ smartPointer=other }); //
        runtime crash
```

Benefits of runtime checks

- Typically much more precise.

```
auto v = vectorFrom(1, 2, 3);
assert(i!=j);
scope x = &v[i]; // returns by reference
scope y = &v[j]; // type systems likely to reject this
*x=2;
*y=3;
```

```
auto v = vectorFrom(1, 2, 3);
assert(i!=j);
// if we do not allow aliases, v only has to count borrows
scope x = v.borrow(i);
scope y = v.borrow(j);

// even aliasing can in principle be allowed
// at the cost of higher auxiliary memory usage
// scope z = v.borrow(i);

x.access!((ref int x){ x=2; });
y = 2; // syntax sugar for the above pattern

// v~=2; // would crash at runtime

v.return(x);
v.return(y);

// ok, nothing borrowed out, reallocation would be safe
v~=2;
```

- Not safe against crashes. Requires storing additional data to check time-dependent properties. “Time range check.”

DIP 1040

DIP1040 by Max Haughton and Walter Bright. Post community round 1.

- DIP1040 proposes to move the static last use of a variable:

DIP 1040

- Can disable the copy constructor to force moves:

```
struct S{
  this(int x){ ... }
  @disable S(ref S other); // copy constructor
  S(S other){ ... } // move constructor (also DIP1040)
}

auto a = S(2);
auto b = a; // ok, a is moved
auto c = b; // ok, b is moved

auto x = S(2);
auto y = x; // error: x is copied
auto z = x;
```

Potentially, DIP1040 is a big step up for the usability of move-only types.

Move-only types

Move-only types can support:

- Value-type-like referential transparency.
- Efficient mutable updates.
- No implicit costly duplication.
- They behave like *resources*.
- D essentially supports substructural typing via *@disable* of special member functions.

Move constructors

```
struct S{
  this(S other){ ... }
}
```

- Structs in D are always implicitly moveable.
- Without a move constructor, this means structs cannot have internal references.
- This has implications for expressiveness and C++ interoperability.
- Move constructors

Danger: Implicit destructor elision

In DIP1040, the move constructor implicitly passes by `ref`:

```
struct S{
  private @system int* x;
  T t;
  ...
  ~this()@trusted{
    if(x) free(x);
    x=null;
  }

  // implicitly S(ref other), but recorded as move
     constructor:
  S(S other)@trusted{
    this.x=other.x;
    other.x=null;
    this.t=other.t; // copy
  }
}
```

Tweaking DIP1040

```
struct S(T){
  this(T)(T arg){ ... }
}
```

- Destructor elision is very dangerous.
- Might implicitly break RAII.
- Can cause memory leaks.
- Better approach: Destructor elision should be explicit.
- Also useful for unpacking.

Not addressed by DIP1040

- How to force a move?
- How to move the receiver of a method call?
- How to move a container into a range into an iteration over the range?
- Unpacking/destructuring without copies.
- How to avoid reinitialization with `.init`?
  - Needed to make `private @system` destructors useful.
- Reinitialization.
- Reinitialization with a different type (strong updates).
- Non-lexical variable lifetimes.

Thanks!

Questions?