

D The functional programming language nobody is talking about

Robert Schadek

November 11, 2020

D is a functional programming language

What makes D a functional programming language

C the functional programming language

- D is a \approx superset of C
- C is a functional programming language

Proof by example

High order functions

```
1 int apply(int (*fun)(int), int value) {
2     return (*fun)(value);
3 }
4
5 int addOne(int a) {
6     return a + 1;
7 }
8
9 int main() {
10    return apply(&addOne, 1);
11 }
```

High order functions

```
1 int addOne(int a) {  
2     return a + 1;  
3 }  
4  
5 int addTwo(int a) {  
6     return a + 2;  
7 }  
8  
9 int (*apply(int value))(int) {  
10    return value == 1  
11        ? &addOne  
12        : &addTwo;  
13 }  
14
```

High order functions

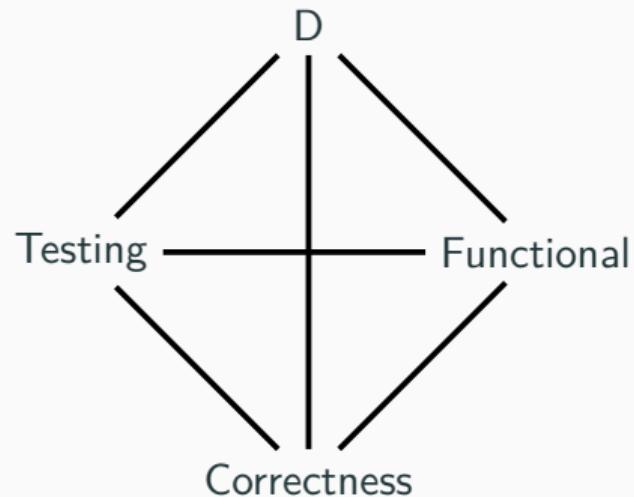
```
1 int apply(int function(int) pure fun
2         , int value) pure
3 {
4     return fun(value);
5 }
6
7 int addOne(int a) pure {
8     return a + 1;
9 }
10
11 int main() {
12     return apply(&addOne, 1);
13 }
```

High order functions

```
1 int addOne(int a) pure {
2     return a + 1;
3 }
4
5 int addTwo(int a) pure {
6     return a + 2;
7 }
8
9 int function(int) pure apply(int value) pure {
10    return value == 1
11    ? &addOne
12    : &addTwo;
13 }
14
```

Why should you care

Why should you care about D



Why should you care

```
1 void main() {  
2     File("rng2.d", "r")  
3         .byLineCopy  
4         .map!(l => l.splitter(" "))  
5         .joiner  
6         .map!(w => w.strip())  
7         .filter!(w => !w.empty)  
8         .array  
9         .sort  
10        .uniq  
11        .count  
12        .writeln;  
13 }
```

Ranges Ranges Ranges

Ranges Ranges Ranges

A thing `foreach` can iterate

Ranges

```
1 struct Range {  
2     int from;  
3     int to;  
4  
5     @property int front() {  
6         return this.from;  
7     }  
8  
9     @property bool empty() const {  
10        return this.from >= this.to;  
11    }  
12  
13    void popFront() {  
14        ++this.from;  
15    }  
16}
```

Ranges

```
1  unittest {
2      foreach(it; Range(0, 10)) {
3          writeln(it); // 0, 1, 2, 3, ... , 9
4      }
5  }
```

Ranges

```
1 unittest {
2     foreach(it; Range(0, 10)) {
3         writeln(it); // 0, 1, 2, 3, ... , 9
4     }
5 }
```



```
1 unittest {
2     for(auto __r = Range(0, 10); !__r.empty; __r.popFront()) {
3         auto it = __r.front;
4
5         writeln(it);
6     }
7 }
```

Range types

- Input Range

Range types

- Input Range
- Forward Range
 - `save()`
- Bidirectional Range
 - `back`
 - `popBack`
- Random Access Range
 - `[]`
- Infinite Range
 - `enum empty = false;`

Ranges types, practically

```
1 import std.array : array;  
2  
3 int[] a = Range(0, 10).array;
```

Ranges

```
1 struct Map(alias fun) {
2     Range range;
3
4     @property int front() {
5         return fun(this.range.front);
6     }
7
8     @property bool empty() const {
9         return this.range.empty;
10    }
11
12    void popFront() {
13        this.range.popFront();
14    }
15 }
```

Ranges

```
1 Map!fun map(alias fun)(Range r) {
2     return Map!fun(r);
3 }
4
5 unittest {
6     auto tt = Range(0, 10).map!(it => it * 2);
7     assert(tt.equal([0, 2, 4, 6, 8, 10, 12, 14, 16, 18]));
8 }
```

Ranges

```
1 struct Map2(alias fun, R) {
2     R range;
3
4     @property int front() {
5         return fun(this.range.front);
6     }
7
8     @property bool empty() const {
9         return this.range.empty;
10    }
11
12    void popFront() {
13        this.range.popFront();
14    }
15 }
```

Ranges

```
1 Map2!(fun,R) map2(alias fun, R)(R r) {
2     return Map2!(fun,R)(r);
3 }
4
5 unittest {
6     import std.range : iota;
7
8     auto tt = iota(0, 10).map2!(it => it * 2);
9     assert(tt.equal([0, 2, 4, 6, 8, 10, 12, 14, 16, 18]));
10 }
```

Uniform functional call syntax (UFCS)

```
1 int theSame(int a) {  
2     return a;  
3 }  
4  
5 int fun2(int a) {  
6     return a;  
7 }  
8  
9 unittest {  
10     fun2(theSame(10));  
11     10.theSame().fun2();  
12 }
```

Ranges

```
1 void main() {  
2     File("rng2.d", "r")  
3         .byLineCopy  
4         .map!(l => l.splitter(" "))  
5         .joiner  
6         .map!(w => w.strip())  
7         .filter!(w => !w.empty)  
8         .array  
9         .sort  
10        .uniq  
11        .count  
12        .writeln;  
13 }
```

Testing

Testing

No tests = wrong

Testing in D

- D has build-in unittesting
- `unittest { }`
- D has in-build test coverage analysis
- 100% is a terrible metric, but still the best we have

Coverage analysis

```
1 int fun(bool b) {
2     return b ? 1 : 0;
3 }
4
5 unittest {
6     assert(fun(true) == 1);
7     assert(fun(false) == 0);
8 }
```

```
dmd -main -cov -unittest -run cov.d
```

Coverage analysis

```
1 | int fun(bool b) {
2 |     return b ? 1 : 0;
3 |
4 |
5 |     |unittest {
6 |         |assert(fun(true) == 1);
7 |         |assert(fun(false) == 0);
8 |     |}
```

Coverage analysis

```
1 int fun2(bool b) {
2     return b
3         ? 1
4         : 0;
5 }
6
7 unittest {
8     assert(fun2(true) == 1);
9     assert(fun2(false) == 0);
10 }
```

Coverage analysis

```
1 | int fun2(bool b) {
2 |     return b
3 |     ? 1
4 |     : 0;
5 |
6 |
7 |     | unittest {
8 |         assert(fun2(true) == 1);
9 |         assert(fun2(false) == 0);
10|    }
```

Coverage analysis

```
1 bool complexCondition(int a, int b) {  
2     return a == 10 && b == 20 || a > 20 || b < 5 ? true : false;  
3 }  
4  
5 unittest {  
6     assert(complexCondition(1,2));  
7 }
```

Coverage analysis

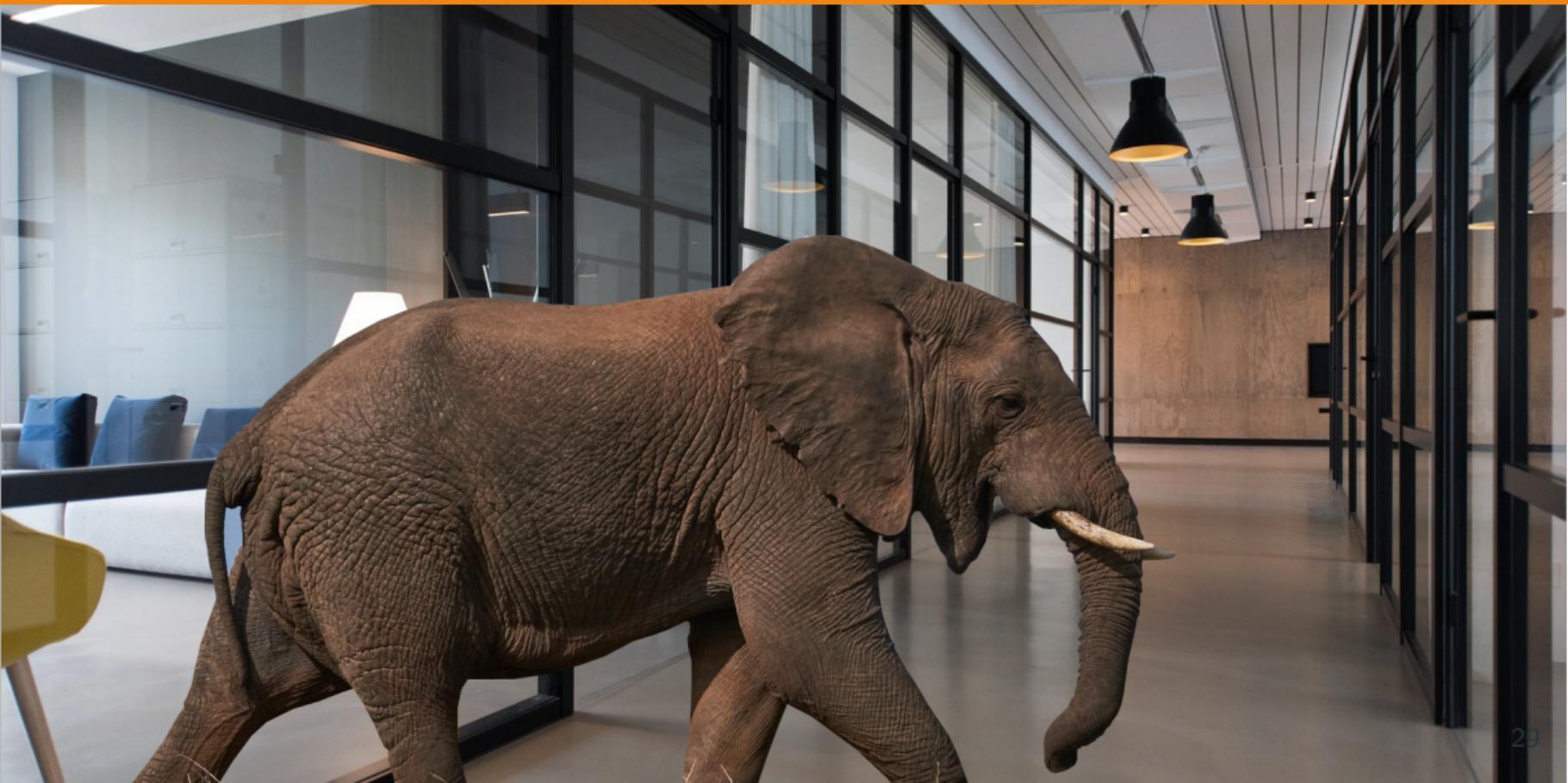
```
1 | bool complexCondition(int a, int b) {
2 |     return a == 10 && b == 20 || a > 20 || b < 5 ? true : false
3 |
4 |
5 |     | unittest {
6 |         | assert(complexCondition(1,2));
7 |     }
```

Coverage analysis

```
1 | bool complexCondition2(int a, int b) {  
2 |     return a == 10  
3 |         && b == 20  
4 |         || a > 20  
5 |         || b < 5  
6 |     ? true  
7 |     : false;  
8 | }  
9 |  
10| unittest {  
11|     assert( complexCondition2(1,2));  
12|     assert( complexCondition2(10,20));  
13|     assert(!complexCondition2(1,50));  
14| }
```

Exceptions

Elephant in the Room



Elephant in the Room

```
1 struct Exp {  
2     @property int front() {  
3         throw new Exception("");  
4     }  
5  
6     @property bool empty() {  
7         return false;  
8     }  
9  
10    void popFront() { }  
11 }  
12  
13 unittest {  
14     Exp e;  
15     assertThrown(e.map!(i => 2).array);  
16 }
```

handle Exceptions

```
1   Exp e;
2   int[] r = e
3       .handle!(Exception, RangePrimitive.front, (e, r) => 0)
4       .take(2)
5       .map!(i => i * 2)
6       .array;
```

try catch

```
1  string s = "12,1337z32,54,2,7,9,1z,6,8";
2
3  int[] i = s.splitter(",")  

4      .map!(n => {
5          try {
6              return to!int(n).nullable();
7          } catch(Exception e) {
8              return Nullable!(int).init;
9          }
10     }()
11     .filter!(n => !n.isNull)
12     .map!(n => n.get())
13     .array;
```

try catch nullable

```
1  Nullable!int parse(string s) nothrow {
2      try {
3          return to!int(s).nullable();
4      } catch(Exception e) {
5          return Nullable!(int).init;
6      }
7  }
8
9  string s = "12,1337z32,54,2,7,9,1z,6,8";
10
11 int[] i = s.splitter(",")  

12     .map!(n => parse(n))  

13     .filter!(n => !n.isNull)  

14     .map!(n => n.get())  

15     .array;
```

State

State

- Most programs are not just input, map, output
- Most programs have some sort of state

State: The model

```
1 struct Group {  
2     long id;  
3     string name;  
4     long[] members;  
5 }  
6  
7 struct State {  
8     Group[] groups;  
9 }
```

State: createGroup

```
1 State createGroup(State old
2     , string name)
3 {
4     Group ng = Group
5         ( old.groups.empty
6             ? 1
7             : old.groups
8                 .map!(g => g.id)
9                     .maxElement
10            , name
11            , []
12        );
13    old.groups ~= ng;
14    return old;
15 }
```

State: findGroup

```
1 Nullable!(const(long)) findGroup(
2     ref const(State) old
3     , string name)
4 {
5     auto f = old.groups
6     .find!(g => g.name == name);
7
8     return f.empty
9     ? typeof(return).init
10    : nullable(f.front.id);
11 }
```

State: addMember

```
1 State addMember(State old
2     , long groupId
3     , long memId)
4 {
5     auto g = old.groups
6         .countUntil!(g => g.id == groupId);
7
8     enforce(g != -1, "Group not found");
9     old.groups[g].members ~= memId;
10    old.groups[g].members = old.groups[g]
11        .members.sort.uniq.array;
12    return old;
13 }
```

State: Usage

```
1  unittest {
2      State s;
3      s = s.createGroup("D_Users");
4
5      Nullable!(const(long)) gId = s
6          .findGroup("D_Users");
7
8      s = s.addMember(gId.get(), 1);
9 }
```

State Two: createGroup

```
1 State createGroup(ref const State old
2     , string name)
3 {
4     Group ng = Group
5         ( old.groups.empty
6             ? 1
7             : old.groups
8                 .map!(g => g.id)
9                     .maxElement
10            , name
11            , []
12        );
13
14     State neu = old.deepcopy();
15     neu.groups ~= ng;
16     return neu;
```

State Two: addMember

```
1 State addMember(ref const State old
2     , long groupId, long memId)
3 {
4     auto g = old.groups
5         .countUntil!(g => g.id == groupId);
6     enforce(g != -1, "Group not found");
7
8     State neu = old.deepcopy();
9     neu.groups[g].members ~= memId;
10    neu.groups[g].members = neu.groups[g]
11        .members.sort.uniq.array;
12    return neu;
13 }
```

State Two: `deepCopy` 1/2

```
1 auto deepCopy(T)(ref const(T) old) {
2     alias UQ = Unqual!T;
3     static if(isBasicType!UQ) {
4         return cast()old;
5     } else static if(isArray!UQ) {
6         alias ET = Unqual!(ElementEncodingType!T);
7         ET[] ret;
8         foreach(ref it; old) {
9             ret ~= deepCopy!(ET)(it);
10        }
11        return ret;
```

State Two: `deepCopy` 2/2

```
1 } else static if(is(T == struct)) {
2     Unqual!T ret;
3     foreach(mem; FieldNameTuple!T) {
4         __traits(getMember, ret, mem) =
5             deepCopy(__traits(getMember, old, mem));
6     }
7     return ret;
8 } else {
9     static assert(false, T.stringof);
10 }
11 }
```

Commercial break

Symmetry Investments

Conclusion

Takeaways

- `if` statements considered harmful
- learn Phobos/std by heart
- when you think exception `goto Nullable`
- no tests = bugs

The End

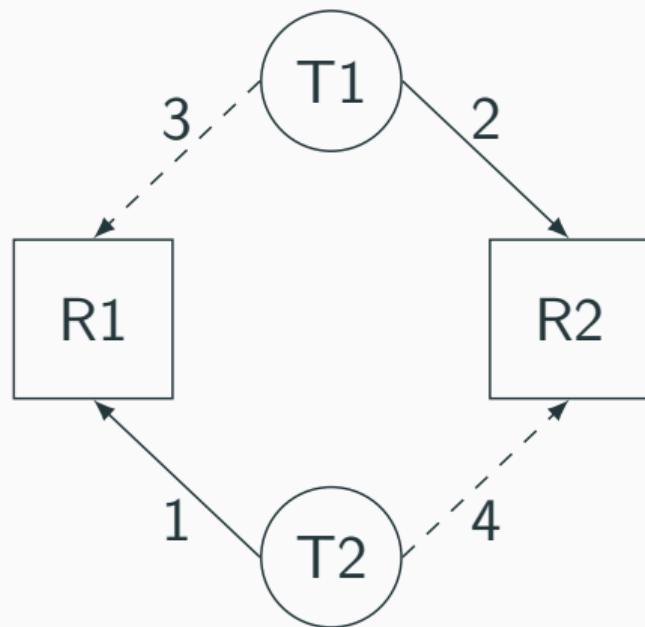
Encore

Dead lock free multi Mutex Systems

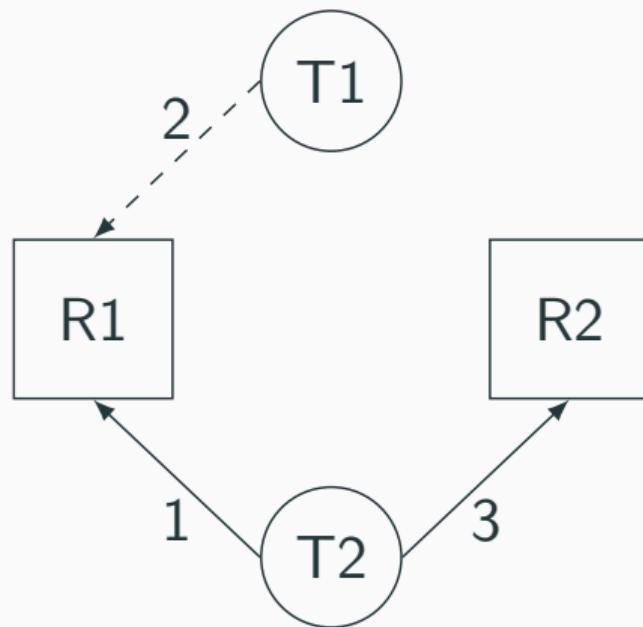
Deadlock Recipe:

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Anti Dead lock: Un-sorted



Anti Dead lock: Sorted



Anti Dead lock

```
1 struct State {  
2     Data[] data;  
3     Mutex[long] mutexes;  
4  
5     void action(ActionData ad  
6                 , long[] mutexIds)  
7     {
```

Anti Dead lock

```
1 struct State {  
2     Data[] data;  
3     Mutex[long] mutexes;  
4  
5     void action(ActionData ad  
6                 , long[] mutexIds)  
7     {  
8         mutexIds  
9             .sort  
10            .each!(it => this.mutexes[id].lock());  
11  
12         //  
13         // perform action  
14         //
```

Out of Slides
