D The functional programming language nobody is talking about

Robert Schadek

November 19, 2020

D is a functional programming language

What makes D a functional programming language

C the functional programming language

- D is a ≈ superset of C
- C is a functional programming language

Proof by example

High order functions

```c
int apply(int(*fun)(int), int value) {
  return (*fun)(value);
}

int addOne(int a) {
  return a + 1;
}

int main() {
  return apply(&addOne, 1);
}
```

High order functions

```
int addOne(int a) {
  return a + 1;
}

int addTwo(int a) {
  return a + 2;
}

int (*apply(int value))(int) {
  return value == 1
    ? &addOne
    : &addTwo;
}

```

High order functions

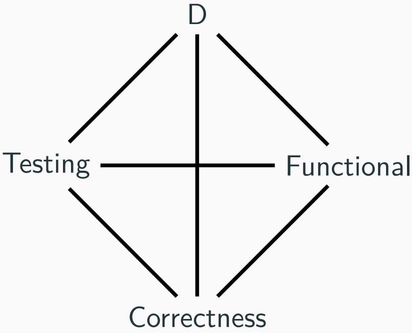
```
int apply(int function(int) pure fun
    , int value) pure
{
  return fun(value);
}

int addOne(int a) pure {
  return a + 1;
}

int main() {
  return apply(&addOne, 1);
}
```

High order functions

```
int addOne(int a) pure {
  return a + 1;
}

int addTwo(int a) pure {
  return a + 2;
}

int function(int) pure apply(int value) pure {
  return value == 1
    ? &addOne
    : &addTwo;
}

```

Why should you care

Why should you care about D



Why should you care

```d
void main() {
  File("rng2.d", "r")
    .byLineCopy
    .map!(l => l.splitter(" "))
    .joiner
    .map!(w => w.strip())
    .filter!(w => !w.empty)
    .array
    .sort
    .uniq
    .count
    .writeln;
}
```

Ranges Ranges Ranges

Ranges Ranges Ranges

A thing `foreach` can iterate

Ranges

```
struct Range {
  int from;
  int to;

  @property int front() {
    return this.from;
  }

  @property bool empty() const {
    return this.from >= this.to;
  }

  void popFront() {
    ++this.from;
  }
}
```

Ranges

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

Ranges

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

```
unittest {
  for(auto __r = Range(0, 10); !__r.empty; __r.popFront()) {
    auto it = __r.front;

    writeln(it);
  }
}
```

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

```
import std.array : array;

int[] a = Range(0, 10).array;
```

Ranges

```
struct Map(alias fun) {
  Range range;

  @property int front() {
    return fun(this.range.front);
  }

  @property bool empty() const {
    return this.range.empty;
  }

  void popFront() {
    this.range.popFront();
  }
}
```

Ranges

```
Map!fun map(alias fun)(Range r) {
  return Map!fun(r);
}

unittest {
  auto tt = Range(0, 10).map!(it => it * 2);
  assert(tt.equal([0, 2, 4, 6, 8, 10, 12, 14, 16, 18]));
}
```

Ranges

```
struct Map2(alias fun, R) {
  R range;

  @property int front() {
    return fun(this.range.front);
  }

  @property bool empty() const {
    return this.range.empty;
  }

  void popFront() {
    this.range.popFront();
  }
}
```

Ranges

```d
Map2!(fun,R) map2(alias fun, R)(R r) {
  return Map2!(fun,R)(r);
}

unittest {
  import std.range : iota;

  auto tt = iota(0, 10).map2!(it => it * 2);
  assert(tt.equal([0, 2, 4, 6, 8, 10, 12, 14, 16, 18]));
}
```

Uniform functional call syntax (UFCS)

```d
int theSame(int a) {
  return a;
}

int fun2(int a) {
  return a;
}

unittest {
  fun2(theSame(10));
  10.theSame().fun2();
}
```

Ranges

```d
void main() {
  File("rng2.d", "r")
    .byLineCopy
    .map!(l => l.splitter(" "))
    .joiner
    .map!(w => w.strip())
    .filter!(w => !w.empty)
    .array
    .sort
    .uniq
    .count
    .writeln;
}
```

Testing

Testing

No tests = wrong

Testing in D

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

Coverage analysis

```d
int fun(bool b) {
  return b ? 1 : 0;
}

unittest {
  assert(fun(true) == 1);
  assert(fun(false) == 0);
}
```

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

Coverage analysis

```
 |int fun(bool b) {
4|  return b ? 1 : 0;
 |}
 |
 |unittest {
1|  assert(fun(true) == 1);
1|  assert(fun(false) == 0);
 |}
```

Coverage analysis

```
int fun2(bool b) {
  return b
    ? 1
    : 0;
}

unittest {
  assert(fun2(true) == 1);
  assert(fun2(false) == 0);
}
```

Coverage analysis

```
 |int fun2(bool b) {
2|  return b
1|    ? 1
1|    : 0;
 |}
 |
 |unittest {
1|  assert(fun2(true) == 1);
1|  assert(fun2(false) == 0);
 |}
```

Coverage analysis

```
bool complexCondition(int a, int b) {
  return a == 10 && b == 20 || a > 20 || b < 5 ? true : false;
}

unittest {
  assert(complexCondition(1,2));
}
```

Coverage analysis

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

Coverage analysis

```
 |bool complexCondition2(int a, int b) {
3|  return a == 10
1|      && b == 20
2|      || a > 20
2|      || b < 5
2|    ? true
1|    : false;
 |}
 |
 |unittest {
1|  assert( complexCondition2(1,2));
1|  assert( complexCondition2(10,20));
1|  assert(!complexCondition2(1,50));
 |}
```

Exceptions

Elephant in the Room

Elephant in the Room

```d
struct Exp {
  @property int front() {
    throw new Exception("");
  }

  @property bool empty() {
    return false;
  }

  void popFront() { }
}

unittest {
  Exp e;
  assertThrown(e.map!(i => 2).array);
}
```

handle Exceptions

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

try catch

```
string s = "12,1337z32,54,2,7,9,1z,6,8";

int[] i = s.splitter(",")
  .map!(n => {
    try {
      return to!int(n).nullable();
    } catch(Exception e) {
      return Nullable!(int).init;
    }
  }())
  .filter!(n => !n.isNull)
  .map!(n => n.get())
  .array;
```

try catch nullable

```
Nullable!int parse(string s) nothrow {
  try {
    return to!int(s).nullable();
  } catch(Exception e) {
    return Nullable!(int).init;
  }
}

string s = "12,1337z32,54,2,7,9,1z,6,8";

int[] i = s.splitter(",")
  .map!(n => parse(n))
  .filter!(n => !n.isNull)
  .map!(n => n.get())
  .array;
```

State

State

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

State: The model

```
struct Group {
  long id;
  string name;
  long[] members;
}

struct State {
  Group[] groups;
}
```

State: `createGroup`

```
State createGroup(State old
    , string name)
{
  Group ng = Group
    ( old.groups.empty
      ? 1
      : old.groups
        .map!(g => g.id)
      .maxElement
    , name
    , []
    );
  old.groups ~= ng;
  return old;
}
```

State: `findGroup`

```d
Nullable!(const(long)) findGroup(
    ref const(State) old
    , string name)
{
  auto f = old.groups
    .find!(g => g.name == name);

  return f.empty
    ? typeof(return).init
    : nullable(f.front.id);
}
```

State: `addMember`

```d
State addMember(State old
    , long groupId
    , long memId)
{
  auto g = old.groups
    .countUntil!(g => g.id == groupId);

  enforce(g != -1, "Group not found");
  old.groups[g].members ~= memId;
  old.groups[g].members = old.groups[g]
    .members.sort.uniq.array;
  return old;
}
```

State: Usage

```
unittest {
  State s;
  s = s.createGroup("D_Users");

  Nullable!(const(long)) gId = s
    .findGroup("D_Users");

  s = s.addMember(gId.get(), 1);
}
```

State Two: `createGroup`

```
State createGroup(ref const State old
    , string name)
{
  Group ng = Group
    ( old.groups.empty
      ? 1
      : old.groups
        .map!(g => g.id)
      .maxElement
    , name
    , []
    );

  State neu = old.deepCopy();
  neu.groups ~= ng;
  return neu;
```

State Two: `addMember`

```d
State addMember(ref const State old
    , long groupId, long memId)
{
  auto g = old.groups
    .countUntil!(g => g.id == groupId);
  enforce(g != -1, "Group not found");

  State neu = old.deepCopy();
  neu.groups[g].members ~= memId;
  neu.groups[g].members = neu.groups[g]
    .members.sort.uniq.array;
  return neu;
}
```

State Two: `deepCopy` 1/2

```
auto deepCopy(T)(ref const(T) old) {
  alias UQ = Unqual!T;
  static if(isBasicType!UQ) {
    return cast()old;
  } else static if(isArray!UQ) {
    alias ET = Unqual!(ElementEncodingType!T);
    ET[] ret;
    foreach(ref it; old) {
      ret ~= deepCopy!(ET)(it);
    }
    return ret;
```

State Two: `deepCopy` 2/2

```
  } else static if(is(T == struct)) {
    Unqual!T ret;
    foreach(mem; FieldNameTuple!T) {
      __traits(getMember, ret, mem) =
        deepCopy(__traits(getMember, old, mem));
    }
    return ret;
  } else {
    static assert(false, T.stringof);
  }
}
```

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

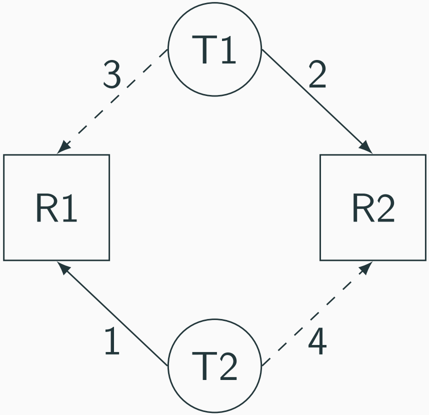
Encore

Dead lock free multi Mutex Systems

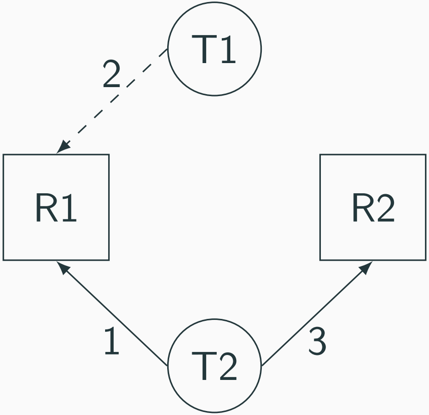
Deadlock Recipe:

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

Anti Dead lock: Un-sorted



Anti Dead lock: Sorted



Anti Dead lock

```
struct State {
  Data[] data;
  Mutex[long] mutexes;

  void action(ActionData ad
      , long[] mutexIds)
  {
```

Anti Dead lock

```
struct State {
  Data[] data;
  Mutex[long] mutexes;

  void action(ActionData ad
      , long[] mutexIds)
  {
    mutexIds
      .sort
      .each!(it => this.mutexes[id].lock());

    //
    // perform action
    //
```

Out of Slides