

Three Cool Things About D

code::dive Conference, Wrocław, Poland

Andrei Alexandrescu, Ph.D.

andrei@erdani.com

Nov 5, 2015

Introduction



Motivation

- Systems-level programming a necessity
- Several application categories
- Faster is better—no “good enough” limit
- Ever-growing modeling needs
- No room below, no escape: all in same language
 - Runtime support
 - Machine, device interface
 - Base library
- This is (literally) where the buck stops

D Design Principles

- Leave no room below
 - share memory model with C
 - statically typed
- Multi-paradigm; balanced
- Practical
- Principled
- Avoid arcana

Why?

- Party line
 - convenience
 - modeling power
 - efficiency
- Actual reasons
 - Produces fast binaries, fast
 - Easier to get into than alternatives
 - Fun

Why not?

- Party line
 -
- Actual reasons
 - Poor on formal specification
 - Little corporate pickup, support
 - Dearth of libraries
 - Large

The “Meh”

```
#!/usr/bin/rdmd
import std.stdio;
void main() {
    writeln("Hello, world!");
}
```

- Why the `std.std` stutter?
- Why import stuff for everything?
- Why no code at top level?
- However:
 - Simple
 - Correct
 - Scriptable

The Innocently Plausible

```
void main() {
    import std.stdio;
    writeln("Hello, world!");
}
```

- Doesn't work in Java, C#
- Career-limiting move in Python, C, C++
- In D, most everything can be scoped everywhere
 - Functions
 - Types (Voldemort types)
 - Even generics
 - Better modularity, reasoning

The Unexpected Emergent

```
void log(T)(T stuff) {
    import std.datetime, std.stdio;
    writeln(Clock.currTime(), ' ', stuff);
}
void main() {
    log("hello");
}
```

- If not instantiated, no import
- imports cached once realized
- Generics faster to build, import
- Less pressure on linker

Heck, add variadics too

```
void log(T...)(T stuff) {
    import std.datetime, std.stdio;
    writeln(Clock.currTime(), ' ', stuff);
}
void main() {
    log("Reached Nirvana level: ", 9);
}
```

Suddenly

Natural lexical scoping
leads to faster builds

Approach to Purity

Thesis

- Writing entire programs in pure style
challenging
- Writing fragments of programs in pure style
easy, useful

- + Easier to verify useful properties, debug
- + Better code generation
- – Challenge: interfacing pure and impure
code

Functional Factorial (yawn)

```
ulong factorial(uint n) {  
    return n <= 1 ? 1 : n * factorial(n - 1);  
}
```

- It's PSPACE!
- Somebody should do hard time for this

However, it's pure

```
pure ulong factorial(uint n) {  
    return n <= 1 ? 1 : n * factorial(n - 1);  
}
```

- Pure is good

Functional Factorial, Fixed

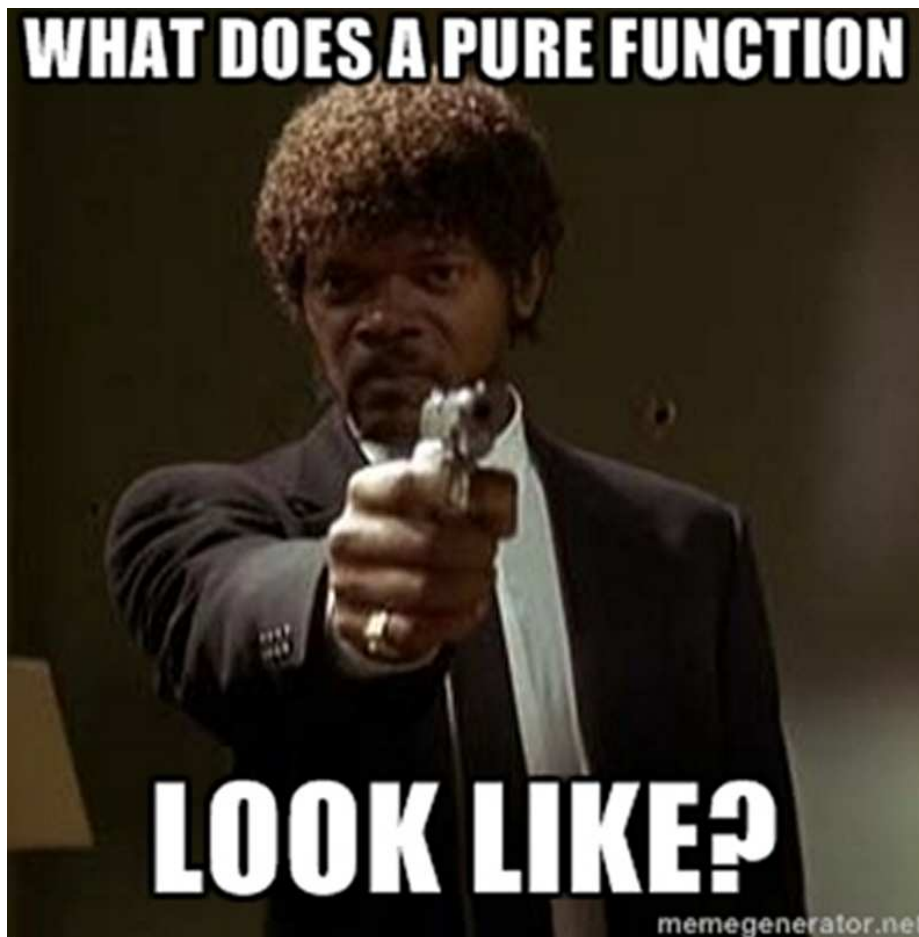
```
pure ulong factorial(uint n) {
    ulong crutch(uint n, ulong result) {
        return n <= 1
            ? result
            : crutch(n - 1, n * result);
    }
    return crutch(n, 1);
}
```

- Threads state through as parameters
- You know what? I don't care for it

Honest Factorial

```
ulong factorial(uint n) {
    ulong result = 1;
    foreach (uint i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

- But no longer pure!
- Well allow me to retort



Pure is as pure does

- “Pure functions always return the same result for the same arguments”
- No reading and writing of global variables
 - (Global *constants* okay)
- No calling of impure functions
- Who said anything about local, transient state *inside the function?*

Transitive State

```
pure void reverse(T)(T[] a) {
    foreach (i; 0 .. a.length / 2) {
        swap(a[i], a[$ - i - 1]);
    }
}
```

- Possibility: disallow
- More useful: relaxed rule
- Operate with transitive closure of state reachable through parameter
- Not functional pure, but an *interesting superset*
- No need for another annotation, it's all in the **signature!**

User-defined types

```
pure BigInt factorial(uint n) {
    BigInt result = 1;
    for (; n > 1; --n) {
        result *= n;
    }
    return result;
}
```

- Better yet: purity deduced for generics and lambdas

Aftermath

- If parameters reach mutable state:
 - Relaxed pure—no globals, no I/O, no `impure` calls
- If parameters can't reach mutable state:
 - “Haskell-grade” *observed* purity
 - Yet imperative implementation possible
 - As long as it's local only

Suddenly

Combining purity with
mutability improves
both

The Generative Connection

Generative programming

- In brief: code that generates code
- Generic programming often requires algorithm specialization
- Specification often present in a DSL

Embedded DSLs

Force into host language's syntax?

Embedded DSLs

- Formatted printing?
- Regular expressions?
- EBNF?
- PEG?
- SQL?

- ... Pasta for everyone!

Embedded DSLs

Here: use with native grammar

Process during compilation

Generate D code accordingly

Compile-Time Evaluation

- A large subset of D available for compile-time evaluation

```
ulong factorial(uint n) {
    ulong result = 1;
    for (; n > 1; --n) result *= n;
    return result;
}
...
auto f1 = factorial(10); // run-time
static f2 = factorial(10); // compile-time
```

Code injection with `mixin`

```
mixin("writeln(\"hello, world\");");  
mixin(generateSomeCode());
```

- Not as glamorous as AST manipulation but darn effective
- Easy to understand and debug

- Now we have compile-time evaluation AND `mixin...`

Wait a minute!

Example: bitfields in library

```
struct A {
  int a;
  mixin(bitfields!(
    uint, "x",      2,
    int,  "y",      3,
    uint, "z",      2,
    bool, "flag",  1));
}
A obj;
obj.x = 2;
obj.z = obj.x;
```

Parser

```
import pegged.grammar; // by Philippe Sigaud
mixin(grammar("
  Expr      <  Factor AddExpr*
  AddExpr   <  ('+'/'-' ) Factor
  Factor    <  Primary MulExpr*
  MulExpr   <  ('*'/ '/' ) Primary
  Primary   <  Parens / Number / Variable
              / '-' Primary
  Parens    <  '(' Expr ')'
  Number    <~ [0-9]+
  Variable  <- Identifier
"));
```

Usage

```
// Parsing at compile-time:  
static parseTree1 = Expr.parse(  
    "1 + 2 - (3*x-5)*6");  
pragma(msg, parseTree1.capture);  
// Parsing at run-time:  
auto parseTree2 = Expr.parse(readln());  
writeln(parseTree2.capture);
```

Scaling up

1000 lines of D
grammar → 3000 lines
D parser

Highly integrated lex+yacc

What about regexen?

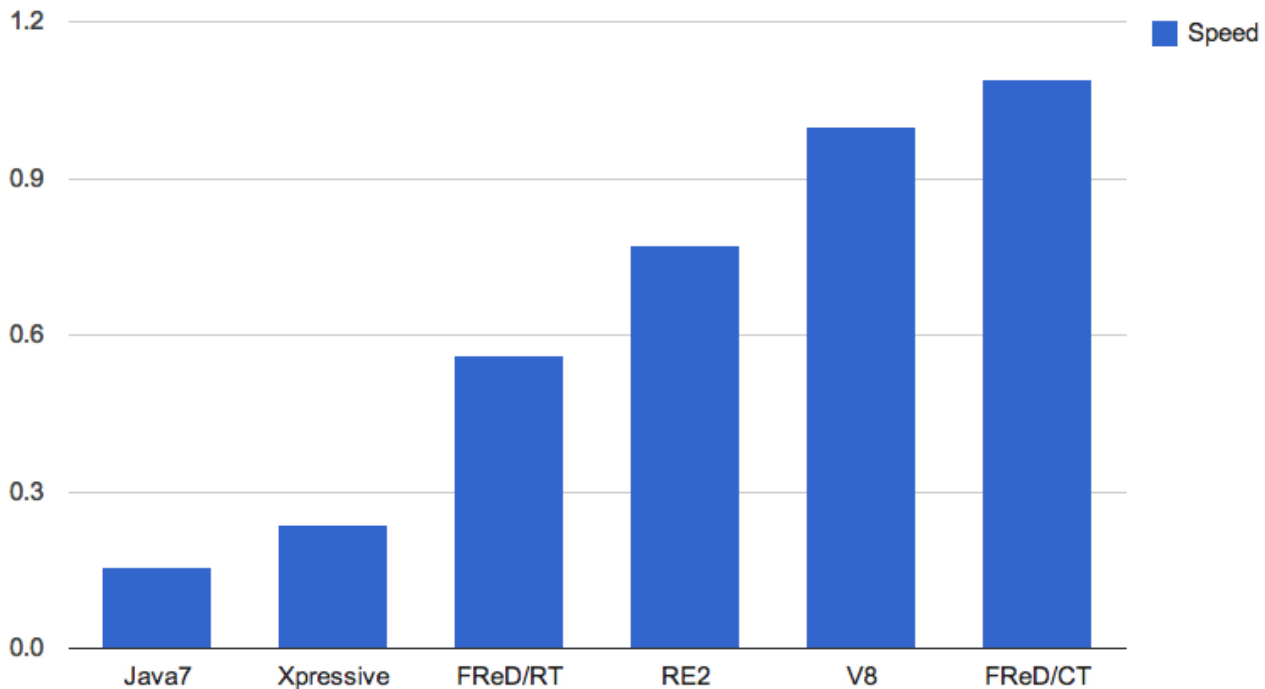
Compile-time regular expressions

- GSoC project by Dmitry Olshansky, merged into `std`
- Fully UTF capable, no special casing for ASCII
- Two modes sharing the same backend:

```
auto r1 = regex("^.*?/([^/]+)/?$");  
static r2 = ctRegex!("^.*?/([^/]+)/?$");
```

- Run-time version uses intrinsics in a few places
- Static version generates specialized automaton, then compiles it

dna-regex from Computer Shootout



Summary

- Modularity + scoping
- Purity + sinning
- Compile-time evaluation + mixing code in