Generic Programming Galore Using D

Andrei Alexandrescu, PhD

Research Scientist

Facebook

This Talk

- Generic Programming
- The Generative Connection
- Summary

Generic Programming

What is Generic Programming?

What is Generic Programming?

- Find most general algorithm representation
  - Narrowest requirements
  - Widest guarantees
  - Big-$O()$ encapsulation should be a crime
  - Low-level efficiency important too: reduces impetus for regressing to hand-written code

What is Generic Programming?

- Find most general algorithm representation
  - Narrowest requirements
  - Widest guarantees
  - Big-$O()$ encapsulation should be a crime
  - Low-level efficiency important too: reduces impetus for regressing to hand-written code
- Define types to implement said requirements

What is Generic Programming?

- Find most general algorithm representation
  - Narrowest requirements
  - Widest guarantees
  - Big-$O()$ encapsulation should be a crime
  - Low-level efficiency important too: reduces impetus for regressing to hand-written code
- Define types to implement said requirements
- Leverage the algorithm for ultimate reuse

What is Generic Programming?

- Find most general algorithm representation
  - Narrowest requirements
  - Widest guarantees
  - Big-$O()$ encapsulation should be a crime
  - Low-level efficiency important too: reduces impetus for regressing to hand-written code
- Define types to implement said requirements
- Leverage the algorithm for ultimate reuse
- …
- Profit!

What is Generic Programming?

- Find most general algorithm representation
  - Narrowest requirements
  - Widest guarantees
  - Big-$O()$ encapsulation should be a crime
  - Low-level efficiency important too: reduces impetus for regressing to hand-written code
- Define types to implement said requirements
- Leverage the algorithm for ultimate reuse
- …
- Profit!

- Arguably one of the noblest endeavors of our métier

Generic Programming

- “Write once, instantiate anywhere”
  - Implementations tailored for enhanced interfaces are welcome
- Prefers static type information and static expansion (macro style)
- Fosters strong mathematical underpinnings
  - Minimizing assumptions common theme in math
- Has tenuous relationship with binary interfaces
- Starts with algorithms, not interfaces or objects
- “Premature encapsulation is the root of some derangement”

Warmup: the “dream `min`”

- Should work at efficiency comparable to *hand-written code*
- Stable
- Take *variadic arguments*: `min(a, b, ...)`
  - Avoid nonsensical calls `min()` and `min(a)`
- Work for *all ordered types* and *conversions*
- *Decline* compilation for incompatible types, *without prejudice*

First prototype

```
auto min(L, R)(L lhs, R rhs) {
   return rhs < lhs ? rhs : lhs;
}
auto a = min(x, y);
```

- This function is as efficient as any specialized, handwritten version (true genericity)
- Deduction for argument types and result type

(Compare At

```
template <class T>
T& min(T& lhs, T& rhs) {
   return rhs < lhs ? rhs : lhs;
}
template <class T>
const T& min(const T& lhs, const T& rhs) {
   return rhs < lhs ? rhs : lhs;
}
```

- Function is incomplete
- Revised implementation, rejected proposal N2199: 175 lines, 10 types, 12 specializations

)

Variadic arguments

```
auto min(T...)(T x) {
  static if (x.length > 2)
    return min(min(x[0], x[1]), x[2 .. $]);
  else
    return x[0] > x[1] ? x[1] : x[0];
}
...
auto m = min(a + b, 100, c);
```

- `x` is not an array
- This is not classic recursion

Reject nonsensical calls

```
auto min(T...)(T x) if (x.length > 1) {
   ...
}
```

- Rejects calls with 0 or 1 arguments
- Allows other overloads to take over, e.g. min element over a collection
- More work needed
  - Only accept types with a valid intersection
  - Only accept types comparable with "<"

Common type

- Task: Given a list of types, find the common type of all

```
template CommonType(T...)
{
  static if (T.length == 1)
    alias T[0] CommonType;
  else
    static if (is(typeof(1 ? T[0].init : T[1].init) U))
      alias CommonType!(U, T[2 .. $]) CommonType;
  else
    alias void CommonType;
}
// Usage
static assert(is(CommonType!(int, short, long) == long));
```

Using CommonType

```
CommonType!T min(T...)(T x)
if (x.length > 1
  && is(typeof(CommonType!T.init < CommonType!T.init)
        == bool))
{
  static if (x.length > 2)
    return min(min(x[0], x[1]), x[2 .. $]);
  else
    return x[0] > x[1] ? x[1] : x[0];
}
```

How about `min` over many elements?

```
auto min(R)(R range)
if (isInputRange!R &&
    is(typeof(range.front < range.front) == bool))
{
  auto result = range.front;
  range.popFront();
  foreach (e; range) {
    if (e < result) result = e;
  }
  return result;
}
auto m = min([ 1, 5, 2, 0, 7, 9 ]);
```

- Works over *anything* that can be iterated

How about argmin now?

```
auto argmin(alias fun, R)(R r)
if (isInputRange!R &&
    is(typeof(fun(r.front) < fun(r.front)) == bool))
{
  auto result = r.front;
  auto cache = fun(result);
  r.popFront();
  foreach (e; r) {
    auto cand = fun(e);
    if (cand > cache) continue;
    result = e;
    cache = cand;
  }
  return result;
}
```

argmin

```
auto s = ["abc", "a", "xz"];
auto m = argmin!((x) {return x.length;})(s);
```

- Works on anything iterable and any predicate
- Predicate is passed by *alias*
- No loss of efficiency

The Generative Connection

Generative programming

- In brief: code that generates code
- Generic programming often requires algorithm specialization
- Specification often present in a DSL
  - String templates, format strings
  - *Regular expressions*
  - Grammars

Compile-Time Evaluation

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

```
ulong factorial(uint n) {
    ulong result = 1;
    foreach (i; 2 .. n) result *= i;
    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`...

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;
```

Compile-time regular expressions

- GSoC project by Dmitry Olshansky: FReD
- 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

Benchmark results

- `dna-regex` program from Computer Shootout

Library	Time
FReD (compile-time)	**3.4 sec**
V8 JS	3.7 sec
RE2	4.8 sec
FReD (run-time)	6.6 sec
Xpressive	15.72 sec
Java 7 Server	23.72 sec
...	

- All top entries use advanced technology
- FReD uses technology *unavailable* to the others

Summary

Summary

- Generic and generative programming have long held unattained promise
- D offers tools to tap into both
  - Powerful base language (variadics, type deduction)
  - Advanced type manipulation
  - Low-cost higher order functions
  - Compile-time evaluation
  - Code generation

Summary

- Generic and generative programming have long held unattained promise
- D offers tools to tap into both
  - Powerful base language (variadics, type deduction)
  - Advanced type manipulation
  - Low-cost higher order functions
  - Compile-time evaluation
  - Code generation

Grill the Speaker!

Questions & Comments

More about ranges? • Thought the talk was boring • Intriguing! • He lost me at `mixin` • Awesome • Went over my head • No, went under my feet • I want to implement that • Too self-congratulatory • I wonder how I can implement binary search • What's for dinner? • Accent is bothersome • How about Phobos vs. Tango? • Real programmers use C • Must... control... fist... of... death...