Grafting Functional Support on Top of an Imperative Language

How D 2.0 implements immutability and functional purity

Andrei Alexandrescu

Overview of D 2.0

- Systems-level programming language
- Memory model similar to C (pointers!)
- As convenient as a scripting language
- Offers a well-defined machine-checkable subset that is memory-safe
- Powerful generics
- Today: D 2.0 offers a pure functional subset

Why Functional Programming (FP)?

- Increased modularity
  - A part of a program cannot mess another
- Easy debugging
  - The call stack contains all context!
- Safe Composition
- Lazy evaluation offers iterators that never invalidate
- Automatic concurrency
  - Immutable sharing is never contentious

Why Functional Programming (FP)?

- Increased modularity
  - A part of a program cannot mess another
- Easy debugging
  - The call stack contains all context!
- Safe Composition
- Lazy evaluation offers iterators that never invalidate
- Automatic concurrency
  - Immutable sharing is never contentious

Why is FP Difficult?

- The three “no”s of Functional Programming:
- No mutable state
- No side effects
- No flow of control

Why Imperative Programming?

- State makes things easy in many applications
  - Databases, persistence. . .
- Fact: many algorithms are specified in terms of mutable state
- Side effects are useful
  - Input/output, files, networking
- I *know* FP has solutions to all of the above
- Just saying that Some Bad People claim mutable state is easier and simpler for certain things

Mixing the Two

- Ideal language—allow:
  - FP-style programming in parts of a program best suited for FP
  - Imperative programming for the rest
- Programmer controls the ratio
- Language statically rules out nonsensical or dangerous mixes of the two styles
- The D 2.0 language implements such a mix

Challenges in Mixing FP and !FP

- How to ensure that the procedural part does not modify the data of the functional part?
- Complete isolation is not the answer!
  - We want the two realms to communicate complex structures to each other
- It’s not a simple matter of copying!
  - Indirection, aliasing mess things up

Challenges in Mixing FP and !FP (II)

- How to ensure that an FP function never calls a !FP function?
  - If it could, FP functions would have side effects!
- How to ensure that a !FP *thread* doesn't mess with the state of an FP call?
- How to typecheck FP functions?
  - What are the minimum applicable restrictions?

Immutable State

A C++-like `const`?

- Here's an idea:
  - Use the `const` qualifier for all FP data
  - Selectively use non-`const` data otherwise
  - The `const` qualifier is passed along with the type, so no risk of "forgetting" it
  - `const` data cannot be assigned
- Problem solved!

A C++-like `const`?

- `const` won't work because:
  - It is *shallow*
  - Protects only the *direct* fields
  - Indirectly-accessed data remains mutable
- It suffers from *aliasing* with non-const data
  - There may be mutable pointers and references aliasing with `const` pointers and references
  - That happens even if the shallow-ness were solved!

C++ `const` is shallow

```
struct Node { int value; Node* next; ... }
const Node* n1 = new Node;
Node* n2 = n1.next; // fine
```

- We want to enforce that anything reachable from a `const Node` is also `const`
- Otherwise a FP function cannot accept data in confidence that it can’t be changed

C++ const is shallow

Transitivity via const functions:

```
class Node {
    Node* next_;
public:
    Node* next() { return next_; }
    const Node* next() const
    { return next_; }
}
```

Hand-written contracts, not statically checkable

Defining a transitive `const`

- Type constructor, notation: `invariant(T)`
- Rule 0: Can't assign to `invariant(T)`
- Rule 1: if `T.field` has type `U`, then `invariant(T).field` has type `invariant(U)`
- Rule 2: `invariant(invariant(T))` $\equiv$ `invariant(T)`
- Rule 3: `T` implicitly converts to and from `invariant(T)` iff `T` refers no mutable memory

Example

```
struct Node { int value; Node* next; ... }
invariant(Node)* n1 = new invariant(Node);
Node* n2 = n1.next; // error!
invariant(Node)* n3 = n1.next; // fine
invariant(int) x = n1.value; // fine
int y = n1.value; // fine because
         // int has no references
```

Expressiveness Problem

```
void print(invariant(Node)* n);
Node* n = new Node;
print(n); // error!
```

- `invariant` is too strict
- How to define a function that can print `invariant` or mutable nodes?
- Must either duplicate the body of `print` or rely on a cast

Defining `const` as the intermediary

- Type constructor, notation: `const(T)`
- Rule 0: Can't assign to `const(T)`
- Rule 1: if `T.field` has type `U`, then `const(T).field` has type `const(U)`
- Rule 2: `const(const(T))` $\equiv$ `const(T)`
- Rule 3: `T` and `invariant(T)` both implicitly convert to `const(T)`

Defining `const` as the intermediary

- Type constructor, notation: `const(T)`
- Rule 0: Can't assign to `const(T)`
- Rule 1: if `T.field` has type `U`, then `const(T).field` has type `const(U)`
- Rule 2: `const(const(T))` $\equiv$ `const(T)`
- Rule 3: `T` and `invariant(T)` both implicitly convert to `const(T)`

Folding Rules

- Problem: weird types may appear
  - `const(invariant(const(...T...)))`
- Define rules for folding combinations:
- `invariant(const(T))` $\equiv$ `invariant(T)`
- `const(invariant(T))` $\equiv$ `invariant(T)`

Intuition

- `const(T) x`: *I can't* modify `x` or anything reachable from it
- `invariant(T) x`: *Nobody can* modify `x` or anything reachable from it
- `invariant` is great for FP code portions
- Unqualified is great for !FP code portions
- `const` is great for factoring code that accepts data from both worlds!

Initializing `invariant` data

- During construction, an object's fields must be assignable
- Yet they can't be non-invariant: somebody may alias the address of a field to a pointer to mutable data!

```
Node.this() invariant {
   value = 0;
   global = &next;
}
```

Different Constructors

- Unlike in C++, the `invariant` and regular constructor *cannot* be shared
- They typecheck very differently
- The regular constructor is allowed to escape pointers to its members without restriction

“Raw” and “Cooked” States

- Typechecking is done in two stages
- Initially `this` has type `__raw(Node)`
- `__raw` is an internal qualifier not accessible to user code
- `__raw` fields can only be assigned to, that’s it
- Once all members have been assigned to, the compiler switches the object’s type to `invariant(Node)`, at which point it can be normally used

“Raw” and “Cooked” States

```
Node.this() invariant {
   // start as raw
   value = 0;
   next = null;
   // shazam! object got cooked
   // can be passed to functions
   print("Done with creating node ", this);
}
```

Important Observation

- Can you `delete invariant` data?
- If so, all hell breaks loose
- All functional languages rely on garbage collection
- D also offers garbage collection, without which FP in D would not be possible
- Don't do the crime if you can't do the time!

Qualifier Summary

- Transitive qualification is *key*
- Two kinds: `invariant` and `const`
- `invariant`: FP data—never, ever changed
- `const`: just a view to possibly mutable data
- Both are necessary to factor code working with FP and !FP

Pure Functions

Immutable Data Not Enough

```
int fooonctional(invariant(Node) n) {
    static int i = 42;
    writeln(++i);
    return n.value + i;
}
```

- Looks like functional to you?
- Signature suggests so!

Pure Functions

- Need a `pure` storage class for functions:

```
int fun(invariant(Node) n) pure {
   ...
}
```

- Challenge: typecheck the body of `fun` to ensure it does not do any impure action

Pure Functions, Take 1

- Disallow all calls to impure functions
- Disallow all access to non-`invariant` data
- By definition of `invariant` and `pure`, it is easy to infer that the result only depends on the inputs

Pure Functions, Take 1

```
int foonctional(invariant(Node) n) pure {
    static int i = 42; // error!
    writeln(++i); // error!
    return n.value + i; // error!
}
```

Pure Functions, Take 1

```
int fooctional(invariant(Node) n) pure {
    invariant(int) i = 42; // fine
    writeln(i); // error!
    return n.value + i; // fine
}
```

An Unnecessary Restriction

```
int fun(invariant(Node) n) pure {
   int i = 42; // error?
   if (n.value) ++i; // error?
   return n.value + i; // error?
}
```

- Key observation: why disallow mutability of *automatic state*?
- Result is *still* dependent solely on inputs!

Pure Functions, Take 2

- Disallow all calls to impure functions
- Allow access to `invariant` data
- Allow *automatic local* mutable state
- Disallow all other data access
- By definition of `invariant` and `pure`, and by scoping of local state, we can infer that the result only depends on the inputs

Yum

```
int fun(invariant(Node) n) pure {
   int i = 42;
   if (n.value) ++i;
   int accum = 0;
   for (i = 0; i != n.value; ++i) ++accum;
   return n.value + i;
}
```

- Got benefits of both FP and !FP worlds in one place!

Conclusions

- Invariant and mutable data can be harmoniously mixed in a unified type framework
- Transitive qualifiers are key
- Pure functions can be modularly typechecked
- Relaxed immutability inside a pure function
  - Allow !FP techniques to be used
- It all rests on an efficient machine model!