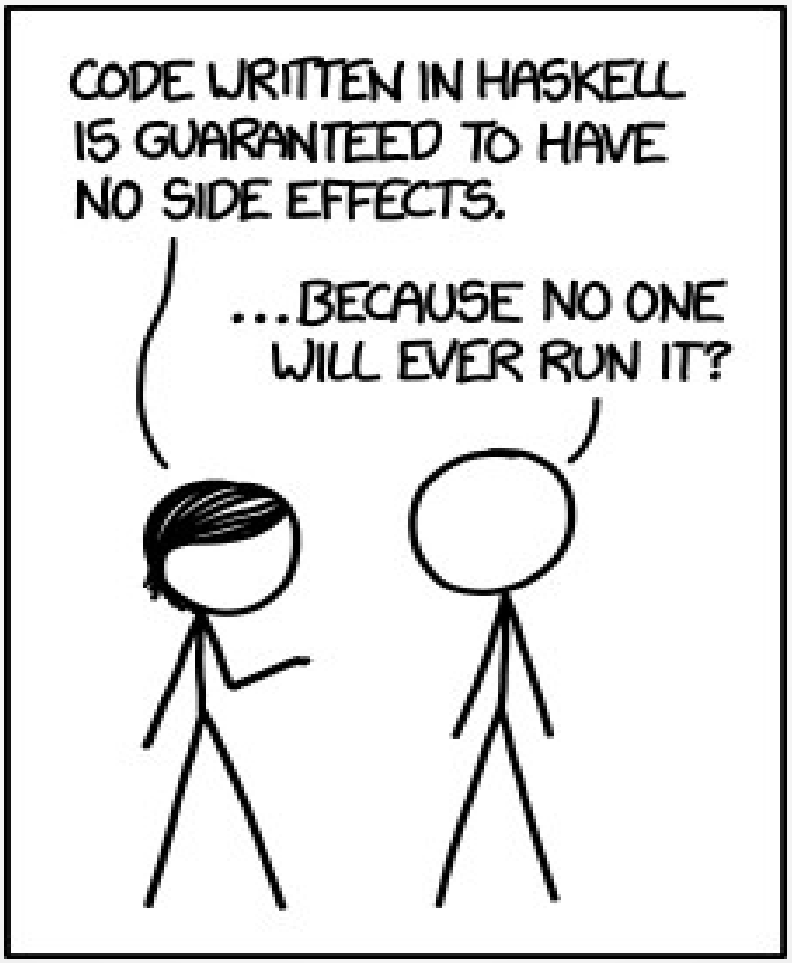
Lazy Evaluation and Reference Counting



Krishna Kumar

Lazy Evaluation - Lazy Fetching

- From the perspective of efficiency, the best computations are those you never perform at all.
- imagine you've got a program that uses large objects containing many constituent fields.
- Such objects must persist across program runs, so they're stored in a database.
- Each object has a unique object identifier that can be used to retrieve the object from the database:
- Because LargeObject instances are big, getting all the data for such an object might be a costly database operation, especially if the data must be retrieved from a remote database and pushed across a network.
- In some cases, the cost of reading all that data would be unnecessary.

Lazy Fetching

- class LargeObject { // large persistent objects
- public:
  - LargeObject(ObjectID id); // restore object from disk
  - const string& field1() const; // value of field 1
  - int field2() const; // value of field 2
  - double field3() const;
  - const string& field4() const;
  - const string& field5() const;
- };
- void restoreAndProcessObject(ObjectID id) {
  - LargeObject object(id);
  - if (object.field2() == 0) { cout << "Object " << id << ": null field2.\n";}
- } //Here only the value of field2 is required, so any effort spent setting up the other fields is wasted

Lazy Fetching (cont...)

```
LargeObject::LargeObject(ObjectID id)
    : oid{id}, field1Value{nullptr}, field2Value{nullptr}, ...{}
const string& LargeObject::field1() const {
    if (!field1Value) { // if null
        read the data for field 1 from the database and make
        field1Value point to it;
    }
    return *field1Value;
}
```

- The best way to say modifying const variable is to declare the pointer fields mutable , which means they can be modified inside any member function, even inside const member functions. That’s why the fields inside LargeObject above are declared mutable .

Lazy Fetching - Implementation

```
class LargeObject {
    public:
        LargeObject(ObjectID id);
        const string& field1() const;
        int field2() const;
        ...
    private:
        ObjectID oid;
        mutable string *field1Value;
        mutable int *field2Value;
        …
};
```

- The lazy approach to this problem is to read no data from disk when a LargeObject object is created.
- Instead, only the “shell” of an object is created, and data is retrieved from the database only when that particular data is needed inside the object.

References

- Bjarne Stroustrup's “C++ Programming Language 4ed”
- Scott Meyer's “Effective Modern C++”
- Herb Sutter's Exceptional C++ and More Exceptional C++
- C++ Templates: the Complete Guide
- http://eli.thegreenplace.net/2014/variadic-templates-in-c/#id1
- http://lbrandy.com/blog/2013/03/variadic_templates/
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2080.pdf