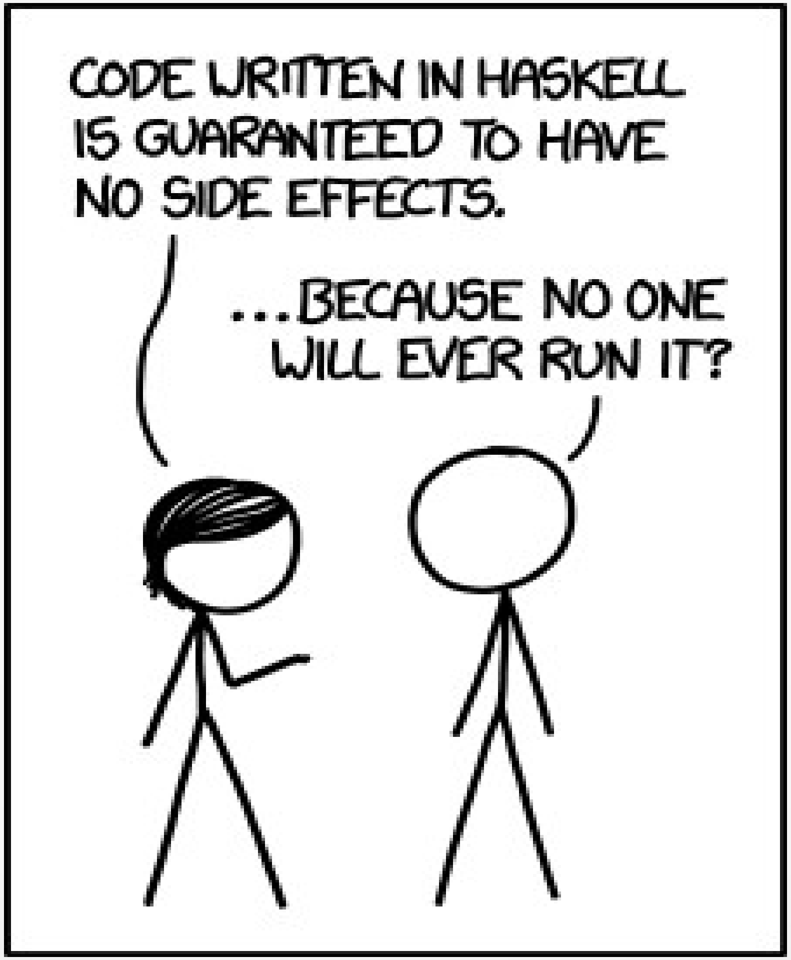
Lazy Evaluation and Reference Counting



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.

Lazy Expression

- template<class T>
- class Matrix { ... }; // for homogeneous matrices
- Matrix<int> m1(1000, 1000);
- Matrix<int> m2(1000, 1000); // a 1000 by 1000 matrix
- ...

  Matrix<int> m3 = m1 + m2; // add m1 and m2
- The usual implementation of operator+ would use eager evaluation; in this case it would compute and return the sum of m1 and m2 . That's a fair amount of computation (1,000,000 additions), and of course there's the cost of allocating the memory to hold all those values, too.

Lazy Expression - Implementation

- The function would usually look something like this:

```
matrix operator +(matrix const& a, matrix const& b);
```

- Now, to make this function lazy, it's enough to return a proxy instead of the actual result:

```
struct matrix_add {
    matrix_add(matrix const& a, matrix const& b) : a(a), b(b) { }
    operator matrix() const {   matrix result;        // Do the addition.
        return result;
    }
private:
    matrix const& a, b;
};
```

matrix_add operator +(matrix const& a, matrix const& b) {   return matrix_add(a, b); }

Lazy Evaluation - Reference Counting

- Consider this code:

  class String { ... }; // a string class

  - String s1 = "Hello";
  - String s2 = s1; // call String copy ctor
- A common implementation for the String copy constructor would result in s1 and s2 each having its own copy of “Hello” after s2 is initialized with s1 . Such a copy constructor would incur a relatively large expense, is called “eager evaluation”.
- The lazy approach is a lot less work. Instead of giving s2 a copy of s1’s value, we have s2 share s1 ’s value. All we have to do is a little book-keeping so we know who’s sharing what.
- cout << s1; // read s1’s value
- cout << s1 + s2; // read s1’s and s2’s values
- s2.convertToUpperCase(); // it’s crucial that only s2 ’s value be changed, not s1

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