March 12, 2014

Compiler Optimizations

Jon Eyolfson & Patrick Lam

Before: common-subexpression elimination

Recall this example from lecture:

File: cse.cpp

```
int f(int a, int b, int c, int d)
{
    int x = a * b + c;
    int y = a * b * d;
    return x + y;
}
```

After: common-subexpression elimination

File: cse-opt.cpp

```
int f(int a, int b, int c, int d)
{
    int t1 = a * b;
    int x = t1 + c;
    int y = t1 * d;
    return x + y;
}
```

We've pulled common subexpression `a * b` and put it into `t`.

(Compiler computes which expressions have already been computed to avoid re-computation.)

Devirtualization (from 2013 exam): class definitions

File: S.hpp

```
#include <cstdlib>

class R {
public:
    virtual double rand();
};
class S : public R {
    unsigned int seed;
public:
    virtual double rand() { return 5.0; }
};
```

Here we have classes R and S.

- R declares virtual function `rand();`
- S provides a (not very random) definition of `rand()`.

Devirtualization (from 2013 exam): client

File: exam-2013-1.cpp

```
#include "S.hpp"

unsigned long int montecarlo(size_t iterations) {
    size_t i, c = 0; double x, y, z; R * r = new S();
    for (i = 0 ; i < iterations; ++i ) {
        x = (double) r->rand()/RAND_MAX;
        y = (double) r->rand() * i/RAND_MAX;
        z = x*x + y*y;
        if (z <= 1.0) ++c;
    }
    return c;
}
```

- instantiate an S object;
- put it in a variable with declared type R; and
- call `r->rand()`.

Two Optimizations

1. If we know that `r->rand()` always calls S's `rand()` function, we can replace the virtual call with a direct call to `S::rand()`.
2. Once we've done that, then it will surely be worthwhile to inline, replacing the call with the constant value 5.

Devirtualization Again

File: devirt.cpp

```
struct A {
    virtual int foo() { return 0; }
};

struct B : A {
    virtual int foo() { return 1; }
};

int main() {
    A * a = new B();
    return a->foo();
}
```

What happens when we call `a->foo()`?

Virtual Call Implementation: Behind the Scenes

File: devirt-unopt.cpp

```
#include "stdlib.h"

struct A;
struct A_vtable { int (*foo)(struct A * a); };
struct A { const struct A_vtable * A_vtable; };
int A_foo(struct A* a) { return 0; }
struct B : public A {};
int B_foo(struct A* b) { return 1; }
static const struct A_vtable A_vtable = {&A_foo};
static const struct A_vtable B_vtable = {&B_foo};

int main()
{
    struct A * a = (struct B *) malloc(sizeof(struct B));
    a->A_vtable = &(B_vtable);
    return a->A_vtable->foo(a);
}
```

Optimizing the Virtual Call

File: devirt-opt.cpp

```
int main()
{
    return 1;
}
```

Turns out the C++ compiler can devirtualize and inline, as before.