Brandon Holt

Quick LLVM Tricks

30 Jan 2014

Registering passes automatically.

My colleague Adrian Sampson showed me a really good trick for running LLVM passes automatically, and luckily he wrote a really nice blog post about it.

I won't steal his thunder, but I will just mention that this has come in handy for me. And because my project uses C++11 throughout, I thought I'd point out that you can make it ever-so-slightly prettier with a lambda which is automatically coerced to a function pointer:

```
////////////////////////////////////////////////////////////////
// Register as default pass, to run before any other optimization passes
static RegisterStandardPasses MyPassRegistration(PassManagerBuilder::EP_EarlyAsPossible,
  [](const PassManagerBuilder&, PassManagerBase& PM) {
    errs() << "Registered pass!\n";
    PM.add(new MyPass());
  });
```

And then to run, simply tell clang to load your shared library (arguments to be passed to clang need to be prefaced with `-Xclang`):

```
$ clang++ -Xclang -load -Xclang /path/to/LLVMMyPass.so
```

"Easy" function annotations

Here's a way to get arbitrary annotations on functions down into your LLVM pass. GCC has long had this "annotate" attribute that allows arbitrary strings to be attached to things like variable declarations and functions, and Clang supports the same syntax. Adding the attribute looks like this:

```
__attribute__((annotate("async")) void foo() {}
```

Clang generates LLVM IR for this annotation in kind of a funky way: it creates a new global variable with a list of tuples that encode the function pointer, annotation string, source file, and line number. But it's really opaque:

```
@.str1 = private unnamed_addr constant [6 x i8] c"asyn
c\00", section "llvm.metadata"
@.str2 = private unnamed_addr constant [39 x i8] c"/User
s/bholt/dev/test/async_finish.cpp\00", section "llvm.met
adata"
@.str3 = private unnamed_addr constant [13 x i8] c"hello
again\0A\00", align 1
@.str4 = private unnamed_addr constant [13 x i8] c"hello
world\0A\00", align 1
@llvm.global.annotations = appending global [2 x { i8*,
i8*, i8*, i32 }] [{ i8*, i8*, i8*, i32 } { i8* bitcast
(void (%struct.Task*)* @"_ZNK4TaskIZ4mainE3$_1_0EclEv" t
o i8*), i8* getelementptr inbounds ([6 x i8]* @.str1, i3
2 0, i32 0), i8* getelementptr inbounds ([39 x i8]* @.st
r2, i32 0, i32 0), i32 12 }, { i8*, i8*, i8*, i32 } { i
8* bitcast (void (%struct.Task.3*)* @"_ZNK4TaskIZ4mainE
3$_0EclEv" to i8*), i8* getelementptr inbounds ([6 x i
8]* @.str1, i32 0, i32 0), i8* getelementptr inbounds
([39 x i8]* @.str2, i32 0, i32 0), i32 12 }], section "l
lvm.metadata"
```

I worked through how to get the values out of here using a couple tricks. One is to get the C++ code to generate the IR you want.

```
$ clang++ -emit-llvm -S foo.cpp
$ llc -march=cpp foo.ll -o ir_generator.cpp
```

Finally, I worked out which operands I wanted, which `Value*` 's were `GlobalVariable` and which were `Constant*` , and ended up with:

```
//////////////////////////
// Find 'task' functions
std::set<Function*> async_fns;

auto global_annos = M.getNamedGlobal("llvm.global.annota
tions");
if (global_annos) {
  auto a = cast<ConstantArray>(global_annos->getOperan
d(0));
  for (int i=0; i<a->getNumOperands(); i++) {
    auto e = cast<ConstantStruct>(a->getOperand(i));

    if (auto fn = dyn_cast<Function>(e->getOperand(0)->g
etOperand(0))) {
      auto anno = cast<ConstantDataArray>(cast<GlobalVar
iable>(e->getOperand(1)->getOperand(0))->getOperand(0))-
>getAsCString();

      if (anno == "async") { async_fns.insert(fn); }
    }
  }
}
```

Edit: (03/11/2014)

I've learned a new trick related to simple function annotations since writing the original post. LLVM supports adding named "function attributes" to the `llvm::Function` object. Rather than keeping track of a `std::set` of the functions with the attribute as above, you can instead do the following:

```
auto global_annos = M.getNamedGlobal("llvm.global.annota
tions");
if (global_annos) {
  auto a = cast<ConstantArray>(global_annos->getOperan
d(0));
  for (int i=0; i<a->getNumOperands(); i++) {
    auto e = cast<ConstantStruct>(a->getOperand(i));

    if (auto fn = dyn_cast<Function>(e->getOperand(0)->g
etOperand(0))) {
      auto anno = cast<ConstantDataArray>(cast<GlobalVar
iable>(e->getOperand(1)->getOperand(0))->getOperand(0))-
>getAsCString();
      fn.addFnAttr(anno); // <-- add function annotation
here
    }
  }
}
```

Then, in other code, you can check for the existence of this attribute simply:

```
for (Function* fn : module) {
  if (fn->hasFnAttribute("myattribute")) {
    outs() << fn->getName() << " has my attribute!\n";
  }
}
```