Changing Everything With Clang Plugins:

A Story About Syntax Extensions, Clang's AST, and Quantum Computing

Hal Finkel, Alex McCaskey, Tobi Popoola, Dmitry Lyakh, and Johannes Doerfert

2020 LLVM Developers' Meeting

AIDE|QC
Argonne NATIONAL LABORATORY
OAK RIDGE National Laboratory

You can be forgiven for not knowing that…

Clang supports plugins!

```
clang++ -c source.cpp -fplugin=/path/to/somePlu
```

Provided using the -fplugin command-line option

Each plugin contains one or more *Handler* classes:

PragmaHandler
    Provides new kinds of pragmas

ParsedAttrInfo
    Provides new kinds of attributes

PluginASTAction
    Provides an *AST consumer* to observe node-creation events

Documentation on making Clang plugins is here:

https://clang.llvm.org/docs/ClangPlugins.html

To build the example plugins, configure using `-DCLANG_BUILD_EXAMPLES=ON`

```
      PP.Diag(Tok, diag::ext_pp_extra_tokens_at_eol) << "pra

    if (HandledDecl) {
      DiagnosticsEngine &D = PP.getDiagnostics();
      unsigned ID = D.getCustomDiagID(
        DiagnosticsEngine::Error,
        "#pragma enable_annotate not allowed after declarati
      D.Report(PragmaTok.getLocation(), ID);
    }

    EnableAnnotate = true;
  }
};


}

static FrontendPluginRegistry::Add<AnnotateFunctionsAction>
X("annotate-fns", "annotate functions");

static PragmaHandlerRegistry::Add<PragmaAnnotateHandler>
Y("enable_annotate","enable annotation");
```

Each kind of handler has a registration object

Let's talk about domain-specific languages (DSLs)...

We have lots of DSLs:

For compilers: Lex, Yacc, ANTLR, re2c, and many others. Don't forget TableGen (our LLVM favorite)!

For high-performance computing: SPIRAL, TCE, TACO, Kranc, GraphIt, and many others.

No.

Embedded DSLs are great (e.g., C++ expression templates, template metaprogramming, constexpr programming), but...

Fitting inside the host language imposes often-unfortunate constraints.

Compilers often are not efficient interpreters, so embedded DSLs have high compile times.

Sometimes, a properly-engineered compiler is just the right tool for the task at hand.

But DSLs are often difficult to integrate well into larger projects...

Build-system integration can be difficult, and even if it's not that bad, what about all of your other tooling?

The DSL input is generally in separate source files, impeding your source readability.

How do we want it to work?

```
[[clang::syntax(MyDSL)]] ReturnT myFunction(Arg1T &A1, A
  This is code in MyDSL, not C++, using A1 and A2. It do
}
```

We created a new kind of Clang plugin: The syntax handler!

Available from: https://github.com/hfinkel/llvm-project-csp

How does it work?

- When parsing a function definition, and a `[[clang::syntax(syntax_name)]]` attribute is present
- Capture the token stream - find the closing `}` using balanced delimiter matching
- Replace the function body with `__builtin_unreachable();` and rename the function name to something internal
- Call the plugin to get the replacement text
- Parse that text (as though it were just included via the preprocessor)
- Continue processing as usual

```
      OS.write_escaped(PP.getSpelling(Tok));
    }
    OS << "\";\n";

    // Rewrite syntax original function.
    OS << getDeclText(PP,D) << "{\n";
    OS << "printf(\"%s\",tokens);\n";
    OS <<"}\n";
  }

  void AddToPredefines(llvm::raw_string_ostream &OS) overrid
    OS << "#include <stdio.h>\n";
  }
};

}

static SyntaxHandlerRegistry::Add<PrintTokensHandler>
X("tokens", "collect all tokens");
```

The handler registers itself using the same scheme as other for other handlers

Let's look at some real examples...

TACOPlug

```
[[clang::syntax(taco)]]
    void matrix_vector_mul
    (vector *y,csr *A,vector *x,
         std::string format=
         "  -f=A:ds:0,1 -f=x:d -f=y:d")
   y(i) = A(i,j) * x(j)
}
```

TACO: http://tensor-compiler.org/

```
[[clang::syntax(taco)]]
    void matrix_vector_mul
    (vector *y,csr *A,vector *x,
         std::string format=
         "  -f=A:ds:0,1 -f=x:d -f
   y(i) = A(i,j) * x(j)
}
```

```
// Generated by TACO:
int __taco_comput_1(taco_tensor_t *,
    taco_tensor_t *,taco_tensor_t *);
int __taco_assm_1(taco_tensor_t *,
    taco_tensor_t *,taco_tensor_t *);

// Assembly Code.
int __taco_assm_1
    (taco_tensor_t *y, taco_tensor_t *A
    taco_tensor_t *x) {
  int y1_dimension = (int)(y->dimensior
  ....
  y->vals = (uint8_t*)y_vals;
  return 0;
}

// Compute Code.
int __taco_comput_1(taco_tensor_t *y,
    taco_tensor_t *A, taco_tensor_t *x)
  ....
  #pragma omp parallel for schedule(rur
  for (int32_t i = 0; i < A1_dimension;
    double tjy_val = 0.0;
    for (int32_t jA = A2_pos[i];
        jA < A2_pos[(i + 1)]; jA++) {
      int32_t j = A2_crd[jA];
      tjy_val += A_vals[jA] * x_vals[j]
    }
    y_vals[i] = tjy_val;
  }
  return 0;
}

void
mat_vec_mul(vector *y, csr *A, vector *
            std::string format=
            "-f=A:ds:0,1 -f=y:d -f=x:d"
```

```
[[clang::syntax(taprol)]]
void test(std::vector<std::complex<double>>& t2
          std::shared_ptr<talsh::Tensor> talsh_
          std::shared_ptr<talsh::Tensor> talsh_
          double& norm_x2) {

 //Declaring the TAProL entry point:
 entry: main;

 //Opening a TAProL scope (optional):
 scope main group(tensor_workload);
  //Declaring linear spaces of some dimension:
  space(complex): space0 = [0:255], space1 = [0
  //Declaring subspaces of declared linear spac
  subspace(space0): s0 = [0:127], s1 = [128:255
  subspace(space1): r0 = [0:283], r1 = [284:511
  //Associating index labels with declared subs
  index(s0): i, j, k, l;
  index(r0): a, b, c, d;
```

Note that parameters are used directly in the DSL

```
  //Initializing a tensor by a registered funct
```

QCOR - Programming Quantum Computers

```
[[clang::syntax(quantum)]]
    void ansatz(qreg q, double x) {
  X(q[0]);
  Ry(q[1], x);
  CX(q[1], q[0]);
}
```

QCOR (XACC): https://github.com/ORNL-QCI/qcor

```
[[clang::syntax(quantum)]]
    void ansatz(qreg q, double
  X(q[0]);
  Ry(q[1], x);
  CX(q[1], q[0]);
}
```

```
// SyntaxHandler-generated code for ans
void ansatz(qreg q, double x) {
  void internal_ansatz_call(qreg, doubl
  internal_ansatz_call(q, x);
}
class ansatz :
    public QuantumKernel<ansatz, qreg,
public:
  ansatz(qreg q, double x) :
    QuantumKernel<ansatz, qreg, double>
  virtual ~ansatz() {
   auto [q,x] = args_tuple;
   // --------------------------------
   // Generated from Token Analysis
   auto provider = xacc::getIRProvider(
   auto i0 = provider->createInstructio
   auto i1 =
       provider->createInstruction("Ry"
   auto i2 = provider->createInstructio
   _parent_kernel->addInstructions({i0,
   // --------------------------------
   auto qpu = xacc::getAccelerator("ibm
   qpu->execute(q, _parent_kernel);
  }
}
```

```
[[clang::syntax(quantum)]]
void ccnot(qreg q,
            std::vector<int> bit_config) {
  // Setup the initial bit configuration
  // This is using XASM language
  for (auto [i, bit] : enumerate(bit_config))
    if (bit) {
      X(q[i]);

    }
  }
```

The DSL support naturally intermixing of (properly tokenized) C++ statements (translated for the output)

Concusions

- Clang supports a powerful plugin interface.
- This interface allows inspecting (and, to some extent, modifying) the AST, adding new pragmas, and adding new attributes.
- We have extended the plugin interface to support DSL integration via syntax plugins.
- Syntax plugins allow function bodies to use a DSL-specified syntax.
- We now have several syntax plugins for real scientific use cases, many more are possible.
- We will continue working to create productive programming environments harnessing the best-available tools.

- We would like to thank the LLVM community, without which this work would not have been possible!
- This work has been supported by the US Department of Energy (DOE) Office of Science Advanced Scientific Computing Research (ASCR) Quantum Computing Application Teams (QCAT), Quantum Algorithms Team (QAT), and Accelerated Research in Quantum Computing (ARQC).
- This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. ORNL is managed by UT-Battelle, LLC, for the US Department of Energy under contract no. DE-AC05-00OR22725.
- This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.
- We would also like to acknowledge the Laboratory Directed Research and Development (LDRD) funding from the Oak Ridge National Laboratory (award