An experimental framework for Pragma handling in Clang

Simone Pellegrini (spellegrini@dps.uibk.ac.at)

University of Innsbruck – Institut für Informatik

Euro-LLVM Meeting, 2013
Background

This work has been done as part of the *Insieme Compiler* (www.insieme-compiler.org)

- A Source-to-Source compiler infrastructure
- Uses LLVM/Clang as a frontend, but relies on its own IR (*INSPIRE*)
- Targets HPC and research issues of parallel paradigms, i.e. OpenMP/MPI/OpenCL
- Developed by the University of Innsbruck\(^1\)

---
\(^1\)Funded by FWF Austrian Science Fund and by the Austrian Research Promotion Agency.
Motivation & Goal
Pragma Directives

“The #pragma directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself.”

```c
#pragma omp parallel for num_threads(x-2) (i)
for(unsigned i=0; i<1000; ++i) {
    do_embarrassingly_parallel_work();
    #pragma omp barrier (ii)
}
```

Their actions are either associated with the following statement/declaration (i) or the position (ii).
Motivation

- Researchers love defining new #pragmas to augment compiler’s knowledge

**Compiler Extensions:** Intel Compiler, Microsoft Visual Studio, PGI, GCC, etc.

**Programming paradigms:** OpenMP, OpenACC, StarSS, etc.

- Clang makes it very difficult!
Pragma Handling in Clang

Clang provides an interface to *react* to new `#pragmas`

class PragmaHandler {
    virtual void HandlePragma(
        Preprocessor &PP,
        PragmaIntroducerKind Introducer,
        Token &FirstToken) = 0;
};

// Hierarchical pragmas can be defined with
class PragmaNamespace : PragmaHandler {
    void AddPragma (PragmaHandler *Handler);
};
Token Tok;
PP.Lex(Tok);
if (Tok.isNot(tok::l_paren))
    throw ...; // error, expected '('

bool LexID = true; // expected 'identifier' next
while(true) {
    PP.Lex(Tok); // consumes next token

    if(LexID) {
        if (Tok.is(tok::identifier)) {
            // save the id for sema checks
            Lex = false;
            continue;
        }
        throw ...; // error, expected 'identifier'
    }
}
#pragma unused(id(,,id)*)

if (Tok.is(tok::comma)) {
  LexID = true; // expected 'identifier' next
  continue;
}

if (Tok.is(tok::r paren))
  break; // success

  throw ...; // error, illegal token
}

Next... semantic checks.
Once gathered the information

=> Sema.ActOnPragmaUnused(...)

- Check semantics (access to the clang::Parser and context)
- Bind pragmas to stmts/decls
- Store/Apply pragma semantics
clang::Sema

- Once gathered the information
  => Sema.ActOnPragmaUnused(...)

  - Check semantics (access to the clang::Parser and context)
  - Bind pragmas to stmts/decls
  - Store/Apply pragma semantics

- Very little is automated!
Not \#pragma friendly!

Defining new pragmas in Clang is cumbersome:

- User has to directly interface with the lexer and preprocessor
- New pragmas cannot be defined without modifying core data structures (e.g. clang::Sema)
  - Use of patches (updated every new LLVM release)
  - Difficult to implement pragmas as Clang extensions (e.g. LibTooling interface)
- Most of the code can be factorized!
Features of a pragma framework

1. Adding a new pragma possible without touching core classes

2. Pragma syntax defined in a declarative form
   - Automatic syntactic checks and generation of error messages with completion hints
   - Easy access to useful information

3. Mapping of pragmas to associated statements/declarations
Pragma Definition
Pragma definition (1/2)

Declarative form\(^2\), similar to EBNF

```
#pragma unused( identifier (, identifier)* )
```

\(^2\)Inspired by the Boost::Spirit parser
Pragma definition (1/2)

Declarative form\(^2\), similar to EBNF

\[ \# \text{pragma} \ \text{unused}( \ \text{identifier} \ (, \ \text{identifier})* ) \]

\[ \# \text{pragma} \ \text{kwd}(\text{`unused'}) \]

\(^2\)Inspired by the Boost::Spirit parser
Pragma definition (1/2)

Declarative form\(^2\), similar to EBNF

\[
\texttt{#pragma unused}(\texttt{identifier (, identifier)*})
\]

\[
\texttt{#pragma kwd('unused')}
\]

\[
.\texttt{followedBy( tok::l_paren )}
\]

\[
.\texttt{followedBy( tok::identifier )}
\]

\[
.\texttt{followedBy(}
\]

\(^2\)Inspired by the Boost\texttt{::Spirit} parser
Pragma definition (1/2)

Declarative form, similar to EBNF

```
#pragma unused( identifier (, identifier)* )
#pragma kwd('unused')
 .followedBy( tok::l_paren )
 .followedBy( tok::identifier )
 .followedBy(
   .repeat <0, inf> ( ( tok::comma )
     .followedBy( tok::identifier )
   )
)
```

²Inspired by the Boost::Spirit parser
Pragma definition (1/2)

Declarative form\(^2\), similar to EBNF

\#pragma unused( identifier (, identifier)* )

\#pragma kwd(‘unused’)

  .followedBy( tok::l_paren )
  .followedBy( tok::identifier )
  .followedBy(
    .repeat <0,inf> ( ( tok::comma )
      .followedBy( tok::identifier )
    )
  ).followedBy( tok::r_paren )
  .followedBy( tok::eod )

\(^2\)Inspired by the Boost::Spirit parser
Pragma definition (2/2)

Use convenience operators (because C++ is awesome):

\[
\begin{align*}
a\text{.followedBy}(b) & \Rightarrow \ a \ » \ b \quad (binary) \\
repeat<0,\text{inf}>(a) & \Rightarrow \ *a \quad (unary)
\end{align*}
\]
Pragma definition (2/2)

Use convenience operators (because C++ is awesome):

```
a.followedBy(b) => a » b (binary)
repeat<0,inf>(a) => *a (unary)
```

```cpp
#pragma kwd('unused')
>> tok::l_paren
   >> tok::identifier
      >> *( tok::comma >> tok::identifier )
>> tok::r_paren >> tok::eod
```
Other operators

Given a position (●) within a stream: \( t_{-1}, t_0 \bullet t_1, t_2, t_3, \ldots \)

- \( a \rightarrow b \): ‘concatenation’, matches iff \( t_1 = a \) and \( t_2 = b \)
- \( a \mid b \): ‘choice’, matches if either \( t_1 = a \) or \( t_2 = b \)
- \( !a \): ‘option’, matches if \( t_1 = a \) or \( \epsilon \) (empty rule)
- \( *a \): ‘repetition’, matches if \( t_1 = \cdots = t_N = a \) or \( \epsilon \)

- Expressions can be combined
- Brackets ( ) can be used to control associativity and priority
Tokens (1/2)

Leaf elements used within pragma specifications:

```cpp
template<clang::tok::TokenKind T>
struct Tok : public node { ... };
```

Import Tokens defined within the Clang lexer:

```cpp
#define PUNCTUATOR(N, _) \
    static Tok<clang::tok::N> N = Tok<clang::tok::N>();
#define TOK(N) \
    static Tok<clang::tok::N> N = Tok<clang::tok::N>();
#include <clang/Basic/TokenKinds.def>
#undef PUNCTUATOR
#undef TOK
```
Tokens (2/2)

Special “semantic tokens” (syntax + sema)

**kwd**: 1 token defining *new keywords* for the DSL supporting the pragma (e.g. `num_threads`)

**var**: 1 token which is a valid *identifier* (i.e. `tok::identifier`) and declared as a *variable*

**expr**: placeholder for a sequence of tokens forming a *syntactically* and *semantically* valid C/C++ expression
Classes organization

```
node
+match(in PP::clang::Preprocessor): bool
+clone(): node*
+operator>>(in n:node): concat
+operator|(in n:node): choice
+operator*(): star
+operator!(): option

val_single
#node: node*
+clone(): node*

Tok
+match(PP::clang::Preprocessor): bool

val_pair
#nodes: std::pair<node*,node>*
+clone(): node*

option
+match(PP::clang::Preprocessor): bool

star
+match(PP::clang::Preprocessor): bool

concat
+match(PP::clang::Preprocessor): bool

choice
+match(PP::clang::Preprocessor): bool
```
Parsing
From spec. to matching

Every concrete node implements the `bool match(clang::Preprocessor& p)` method.

```cpp
bool concat::match(clang::Preprocessor& PP) {
    PP.EnableBacktrackAtThisPos();
    if (lhs.match(PP) && rhs.match(PP)) {
        PP.CommitBacktrackedTokens();
        return true;
    }
    PP.Backtrack();
    return false;
}
```
bool choice::match(clang::Preprocessor& PP) {
    PP.EnableBacktrackAtThisPos();
    if (lhs.match(PP)) {
        PP.CommitBacktrackedTokens();
        return true;
    }
    PP.Backtrack();
    PP.EnableBacktrackAtThisPos();
    if (rhs.match(PP)) {
        PP.CommitBacktrackedTokens();
        return true;
    }
    PP.Backtrack();
    return false;
}
From spec. to matching

Implements a **top-down recursive descent parser** with backtracking

- Not particularly efficient, but practical for small DSLs
From spec. to matching

Implements a top-down recursive descent parser with backtracking

- Not particularly efficient, but practical for small DSLs

```cpp
auto var_list =
    l_paren >> var >> *(comma >> var) >> r_paren;
auto for_clause = (  
    ( kwd("first_private") >> var_list )
| ( kwd("last_private") >> var_list )
| ( kwd("collapse") >> l_paren >> expr >> r_paren )
| kwd("nowait")
| ...  
);
auto omp_for = Tok<tok::kw_for>() >> *for_clause >> eod;
```
Hack for expr parsing

We don’t want to write the grammar for C expressions, the \texttt{clang::Parser} already does it for free!

Why not expose the \texttt{clang::Parser} instance?
Hack for expr parsing

We don’t want to write the grammar for C expressions, the `clang::Parser` already does it for free!

Why not expose the `clang::Parser` instance?

```cpp
struct ParserProxy {
    clang::Parser* mParser;
    ParserProxy(clang::Parser* parser): mParser(parser) { }

public:
    clang::Expr* ParseExpression(clang::Preprocessor& PP);
    clang::Token& ConsumeToken();
    clang::Token& CurrentToken();
    ...
};
```

ParserProxy is declared as a `friend` class of `clang::Parser` (via patch)
Extracting Information
Extract useful information

Within pragmas, some information is not semantically relevant (e.g. punctuation)

For example in the pragma:

```c
#pragma omp for private(a,b) schedule(static)
...```

We are interested in the fact that:

1. This is an OpenMP “for” pragma
2. Variables a and b must be “private”
3. Scheduling is “static”

No interest in: , ( )
The **MatchMap** object

A generic object which stores any relevant information:

```cpp
class MatchMap: std::map<string,
        std::vector<
            llvm::PointerUnion<clang::Expr*, string*>>
>>
{ ... };
```

**MatchMap** layout for the previous example:

- "for"      → \{ \}
- "private"  → \{ a, b \}
- "schedule" → \{" static" \}

The map is filled while parsing a pragma.
Control over mapping

Two operators used within the pragma specification:

\( a["key"] \): All tokens matched by \( a \) will be referenced by
\( key \) in the MatchMap

\( \sim a \): None of the tokens matched by \( a \) will be stored in the MatchMap
Control over mapping

Two operators used within the pragma specification:

- \texttt{a["key"]}: All tokens matched by \texttt{a} will be referenced by \texttt{key} in the MatchMap

- \texttt{~a}: None of the tokens matched by \texttt{a} will be stored in the MatchMap

```c
auto var_list =
    ~l_paren >> var >> *(~comma >> var) >> ~r_paren;
auto for_clause = (  
    ( kwd("first_private") >> var_list["first_private"] )  
    | ( kwd("last_private") >> var_list["last_private"] )  
    | ...  
);```

Simone Pellegrini / EuroLLVM–2013
Pragma → Stmt
Pragma to stmt association

Hack in `clang::Sema`, works for any new pragma!

- *Correctly* parsed pragmas are stored in a list of *pending* pragmas

- When either a `CompoundStmt`, `IfStmt`, `ForStmt`, `Declarator` or a `FunctionDef` is reduced by `Sema` => an algorithm checks for *association* with pending pragmas *based on source locations*.
  
  ▶ Faster than performing *a-posteriori* traversal of the AST

- For positional pragmas (e.g. `omp barrier`) NOPs are inserted in the AST
Framework interface (1/2)

```cpp
struct OmpPragmaCritical: public Pragma {
    OmpPragmaCritical(
        const SourceLocation& startLoc,
        const SourceLocation& endLoc,
        const MatchMap& mmap) {
    }
    Stmt const* getStatement() const; // derived from Pragma
    Decl const* getDecl() const; // derived from Pragma
    ...
};
```
Framework interface (1/2)

```cpp
struct OmpPragmaCritical: public Pragma {
    OmpPragmaCritical(
        const SourceLocation& startLoc,
        const SourceLocation& endLoc,
        const MatchMap& mmap) {}
    Stmt const* getStatement() const; // derived from Pragma
    Decl const* getDecl() const; // derived from Pragma
    ...
};
PragmaNamespace* omp = new clang::PragmaNamespace("omp");
pp.AddPragmaHandler(omp);
// #pragma omp critical [(name)] new-line
omp->AddPragma(
    PragmaFactory::CreateHandler<OmpPragmaCritical>(
        pp.getIdentifierInfo("critical"),
        !(l_paren >> identifier["critical"] >> r_paren) >> eod)
);
Framework interface (2/2)

MyDriver drv; // instantiates the compiler and registers pragma handlers
TranslationUnit& tu = drv.loadTU("omp_critical.c");

constPragmaList& pl = tu.getPragmaList();
const ClangCompiler& comp = tu.getCompiler(); // contains ASTContext

EXPECT_EQ(pl.size(), 4u);
// first pragma is at location [(4:2) - (4:22)]
PragmaPtr p = pl[0];
{
    CHECK_LOCATION(p->getStartLocation(), comp.getSourceManager(), 4, 2);
    CHECK_LOCATION(p->getEndLocation(), comp.getSourceManager(), 4, 22);

    EXPECT_EQ(p->getType(), "omp::critical");
    EXPECT_TRUE(p->isStatement()) << "Pragma is associated with a Stmt";
    const clang::Stmt* stmt = p->getStatement();

    // check the is an omp::critical
    omp::OmpPragmaCritical* omp = dynamic_cast<omp::OmpPragmaCritical*>(p.get());
    EXPECT_TRUE(omp) << "Pragma should be omp::critical";
}
Some performance numbers

Used framework to encode the OpenMP 3.0 standard

Total **frontend time** for some of the OpenMP NAS Parallel Benchmarks:

<table>
<thead>
<tr>
<th>Bench.</th>
<th># Pragmas</th>
<th>w/o OpenMP</th>
<th>w OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>58</td>
<td>45 msecs</td>
<td>48 msecs</td>
</tr>
<tr>
<td>MG</td>
<td>29</td>
<td>36 msecs</td>
<td>39 msecs</td>
</tr>
<tr>
<td>LU</td>
<td>39</td>
<td>47 msecs</td>
<td>54 msecs</td>
</tr>
</tbody>
</table>
Summary

Showed an idea for *easy custom* pragmas in Clang!

The framework code (+Clang 3.2 patches) available at:
https://github.com/motonacciu/clomp

Not integrated into Clang... yet:
- Little time to invest (to change in the near future)
- Requires some restructuring (use of attributes?)
- Level of *interest* shown by the LLVM/Clang community
La Fin!

Questions?

Want to contribute?

https://github.com/motonacciu/clomp