Performing Source-to-Source Transformations with Clang

European LLVM Conference Paris, 2013

Zellescher Weg 12
Willers-Bau A 105
Tel. +49 351 - 463 - 32442

Olaf Krzikalla (olaf.krzikalla@tu-dresden.de)
Agenda today

1. Some disclaimers (sort of)
   - and some background: source-to-source vectorization

2. Our current solution (working with clang 3.2)
   - traversing the AST
   - editing the AST

3. Best (or worth discussing) practices
   - merging ASTs
   - using TreeTransform
   - cloning

4. Future Directions
Disclaimers

• no strategical elaboration of the source-to-source approach
  • instead a lot of code
• we transform clang's AST!
  • actually not allowed
  • source-to-source transformation → source-to-source compilation
• all blue highlighted code works
  • open source and downloadable at http://scout.zih.tu-dresden.de/
  • project started in 2009 → meanwhile better approaches for some tasks
The big picture: Scout

void g(float* a, float b, float* c)
{
    int i;
    #pragma scout loop vectorize
    for (i = 0; i < 100; ++i)
    {
        c[i] = a[i] + b;
    }
}

void g(float* a, float b, float* c)
{
    __m128 art_vectorized0, art_vectorized2,
           art_vectorized1;
    int i;
    art_vectorized1 = _mm_set1_ps(b);
    for (i = 0; i < 100 - 3; i += 4)
    {
        art_vectorized0 = _mm_loadu_ps(&(a[i]));
        art_vectorized2 =
                           _mm_add_ps(art_vectorized0,
                                       art_vectorized1);
        _mm_storeu_ps(&(c[i]), art_vectorized2);
    }
    for (; i < 100; ++i)
    {
        c[i] = a[i] + b;
    }

warning: start of main
read_once.c:6:3: note: vectorizing efficiency: 3 vectorized ops, 0 unrolled ops
read_once.c:6:3: note: loop vectorized {tgt:14:18}
The medium picture: Components

- LLVM
- clang
  - Lexer
  - Parser
  - AST
  - Rewrite
  - Static Analyzer
- Scout
  - AST Processing
  - AST editing
  - AST cloning
  - AST traversal
  - pragma parsing
  - Vectorizing
2. Our current solution (working with clang 3.2)

AST creation and AST editing

```c++
// stmt_iterator traverses over all Stmts of a given type in a tree
// template< class StmtTy, class IteratorTy = llvm::df_iterator<Stmt*> >
class stmt_iterator
{
    IteratorTy m_Iterator;

    void toNext()
    {
        while (m_Iterator != IteratorTy::end() &&
            !isa<StmtTy>(*m_Iterator))
        {
            ++m_Iterator;
        }
    }
}
```
AST Creation

- central class `StmtEditor`
- interface for the creation of variables, expressions and statements

```cpp
class StmtEditor {
public:
    ASTContext& Ctx();
    BinaryOperator* Assign_(Expr* lhs, Expr* rhs);
    BinaryOperator* Add_(Expr* lhs, Expr* rhs);
    DeclRefExpr* DeclRef_(ValueDecl* VD);
    Expr* Int_(int value);  // simple 32 bit integer
    Expr* Float_(const llvm::APFloat& value, QualType t);
    VarDecl* VarDecl_(QualType tmpType, Expr* init = 0,
                     const tOriginalNameInfo& originalVar = tOriginalNameInfo());
    // aso.
};
```

`clangAddons/include/clang/ASTProcessing/StmtEditor.h`
best way: access the member functions of `StmtEditor` by derivation:

class LoopBlocker : StmtEditor {
    void block(ForStmt* Node) {
        DeclStmt *temp = TmpVar_(Ctx().IntTy), *temp_bound = TmpVar_(Ctx().IntTy),
        *i_bound = TmpVar_(Ctx().IntTy);
        Stmt* innerBody[3] = {
            // temp_bound = i_bound - loopVar;
            Assign_(DeclRef_(temp_bound), Sub_(DeclRef_(i_bound), DeclRef_(loopVar))),
            // temp_bound = temp_bound < tileSize ? temp_bound : tileSize;
            Assign_(DeclRef_(temp_bound), Conditional_(LT_(DeclRef_(temp_bound),
                Int_(tileSize)), DeclRef_(temp_bound), Int_(tileSize))),
            // for (temp=0; temp < temp_bound; ++temp) ...
            For_(Assign_(DeclRef_(temp),Int_(0)), LT_(DeclRef_(temp),DeclRef_(temp_bound)),
                PreInc_(DeclRef_(temp)), Node->getBody()))
        };
        Node->setBody(Compound_(innerBody));
    }
};

clangAddons/include/clang/ASTProcessing/LoopBlocking.cpp
transformation performed:

```cpp
for (...) i < z; ++i)
    for-body
```

```cpp
for (...) i < z; ++i) {
    temp_bound = i_bound - i;
    temp_bound = temp_bound < tileSize ? temp_bound : tileSize;
    for ( temp = 0; temp < temp_bound; ++temp)
        for-Body;
}
```

things missing:

- implementation of StmtEditor
- replace loop index `i` with `temp` mutating an AST enters the true minefield
AST Creation

- creating AST nodes:
  - no problem at statement level

```c++
class StmtEditor {
    static const SourceLocation nopos; // helper
    IfStmt* If_(Expr* cond, Stmt* then, Stmt* else) {
        return new (Ctx()) IfStmt(Ctx(), nopos, 0, cond, then, nopos, else));
    }
};
```
AST Creation

• creating AST nodes:
  • implementation of the most possible naive approach at expression level:

```cpp
BinaryOperator* BinOp_(Expr* lhs, Expr* rhs, BinaryOperator::Opcode opc) {
  if (opc >= BO_MulAssign && opc <= BO_OrAssign)
  {
    return new(Ctx()) CompoundAssignOperator(lhs, rhs, opc, lhs->getType(),
                                              VK_RValue, OK_Ordinary, lhs->getType(), lhs->getType(),
                                              nopos, false));
  }

  QualType resultType = (BinaryOperator::isComparisonOp(opc) ||
                         BinaryOperator::isLogicalOp(opc)) ? Ctx().BoolTy : lhs->getType();
  return new(Ctx()) BinaryOperator(lhs, rhs, opc, resultType,
                                    VK_RValue, OK_Ordinary, nopos, false));
}
```

• fails for various reasons ▣ don't try this at home
  • requires redirection to Sema
AST Editing

- editing AST nodes
  - replacing statements in compound statements is no problem
  - general purpose replacement
    - requires parent map internally maintained by `StmtEditor`
  - and once again: works smoothly at statement level only, but replacing sub-expressions is dangerous

```cpp
class StmtEditor {

    // all staments of S are replaced by Stmts
    void replaceStmts(CompoundStmt* S, Stmt **Stmts, unsigned NumStmts);

    // replaces from in the parent with newStmt, returns newStmt
    Stmt* replaceStatement(Stmt* from, Stmt* newStmt);

};
```

However: all this code shown here is in production → clang can do this kind of transformations!
AST Traversing

- template<typename Derived> class RecursiveASTVisitor;
- processing of different AST classes in one traversal
- uses CRTP → requires sub-classing

Diagram:
- ForStmt
  - Cond:BinaryExpr
  - Inc-Stmt
  - CompoundStmt
- CompoundStmt
  - ...

- ForStmt
  - Cond:UnaryExpr
  - Inc-Stmt
  - CompoundStmt
  - BinaryExpr
  - ...

Olaf Krzikalla
AST Traversing

- `template<class StmtTy> class stmt_iterator`
- forward iterator for a particular AST class given by `StmtTy`
- implementation based on `llvm::df_iterator<Stmt*>`
- usable in floating code:

```cpp
//...
for (stmt_iterator<ForStmt> i = stmt_ibegin(root),
     e = stmt_iend(root); i != e; ++i)
{
    ForStmt* node = *i;
    //...
}
//...
```

`clangAddons/include/clang/ASTProcessing/StmtTraversal.h`
• `template<class StmtTy> class stmt_iterator`
  • processes only one AST class per traversal
  • doesn't handle type decls
3. Best (or worth discussing) practices

questions raised on cfe-dev and our solutions
Cloning

- cloning parts of an AST is important for many transformation tasks
  - e.g. function inlining, loop unrolling aso.
  - just search for "clone" on cfe-dev

```cpp
class StmtClone : public StmtVisitor<StmtClone, Stmt*> {

public:

    template<class StmtTy>
    StmtTy* Clone(StmtTy* S) {
        return static_cast<StmtTy*>(Visit(S));
    }

    Stmt* StmtClone::VisitStmt(Stmt*) {
        assert(0 && "clone incomplete");
        return NULL;
    }

    // visitor functions
};
```

`clangAddons/include/clang/ASTProcessing/StmtClone.h`
Cloning

- cloning parts of an AST is important for many transformation tasks
  - implementation clones recursively
  - as volatile as the AST classes

```cpp
class StmtClone : public StmtVisitor<StmtClone, Stmt*> {
public:
   Stmt* VisitBinaryOperator (BinaryOperator *Node) {
      BinaryOperator* result = new (Ctx) BinaryOperator(
          Clone(Node->getLHS()), Clone(Node->getRHS()),
          Node->getOpcode(), Node->getType(), Node->getValueKind(),
          Node->getObjectKind(), Node->getOperatorLoc(),
          Node->isFPContractable());

      result->setValueDependent(Node->isValueDependent());
      result->setTypeDependent(Node->isTypeDependent());
      return result;
   }
};
```

`clangAddons/lib/ASTProcessing/StmtClone.cpp`
Cloning

- cloning parts of an AST is important for many transformation tasks
- is `TreeTransform` the better cloner?

```cpp
struct StmtClone : TreeTransform<StmtClone>
{
    // ???
    bool AlwaysRebuild() { return true; } // this essentially clones
    // the cast might fail (e.g. for ImplicitCastExpr):
    template<class StmtTy>
    StmtTy* Clone(StmtTy* S) {
        return static_cast<StmtTy*>(Transform(S).get());
    }
};
```
Using TreeTransform

- task: transform $a += b$ to $a = a + b$

```cpp
//...
#include "clang/AST/StmtVisitor.h"
#include "../lib/Sema/TreeTransform.h"

struct CompoundAssignTransform : TreeTransform<CompoundAssignTransform>
{
    CompoundAssignTransform (Sema& s) :
        TreeTransform<CompoundAssignTransform>(s) {}

    //...
};
```

clangAddons/lib/Vectorizing/Analysis.cpp
Using TreeTransform

- task: transform $a \mathbin{+=} b$ to $a = a + b$
- creating `TreeTransform`

```cpp
class RewriteInline : public SemaConsumer
{
   CompilerInstance& CI;
public:
   RewriteInline(CompilerInstance &CInst) : CI(CInst) {}

   virtual void InitializeSema(Sema &S) { CI.setSema(&S); }
   virtual void ForgetSema() { CI.takeSema(); }

   virtual void HandleTranslationUnit(ASTContext &C);
};
```

`clangAddons/lib/Interface/Interface.cpp`
Using TreeTransform

- task: transform \( a += b \) to \( a = a + b \)
- perform the transformation

```cpp
struct CompoundAssignTransform : TreeTransform<CompoundAssignTransform>
{
    //...
    bool AlwaysRebuild() { return true; } // this essentially clones
    ExprResult TransformCompoundAssignOperator(CompoundAssignOperator *E)
    {
        BinaryOperator::Opcode binOpc = transformOpc(E->getOpc());
        ExprResult lhsClone = TransformExpr(E->getLHS());
        ExprResult rhs = RebuildBinaryOperator(E->getOperatorLoc(),
                                              binOpc, lhsClone.get(), E->getRHS());
        return RebuildBinaryOperator(E->getOperatorLoc(),
                                      BO_Assign, E->getLHS(), rhs.get());
    }
};
```

`clangAddons/lib/Vectorizing/Analysis.cpp`
Using TreeTransform

- task: transform \( a += b \) to \( a = a + b \)
- creating and using the transformation

```cpp
int VisitCompoundAssignOperator(CompoundAssignOperator* Node) {
    Sema::ContextRAII raiiHolder(getSema(), &getFnDecl());
    ExprResult res = CompoundAssignTransform(getSema()).
    TransformCompoundAssignOperator(Node);
    if (res.isInvalid())
    {
        return ERROR;
    }
    replaceStatement(Node, res.get());
    return SUCCESS;
}
```

`clangAddons/lib/Vectorizing/Analysis.cpp`
AST Merging

- first way: textual level
  - preprocess complete files
  - requires the same language settings
  - used to get function bodies for inlining

```cpp
std::stringstream completeSource;
const std::list<std::string>& preprocessedFiles = //...
for (std::list<std::string>::const_iterator i =
    preprocessedFiles.begin(), e = preprocessedFiles.end();
    i != e; ++i) {
    if (*i != pFileName)  // [#717]: don't self-preprocess
        completeSource << "#include "" << *i << ""\n"
;
}
completeSource << "#line 1\n";
completeSource << actualSource;
```

clangAddons/lib/Interface/Application.cpp:processFile
AST Merging

- second way: ASTImporter
  - import code snippets
  - the Scout-specific class Configuration holds a source AST

```cpp
ASTImporter* create(CompilerInstance& compiler, // target AST
                    Configuration& config) // holds source AST
{
    return new ASTImporter(
        compiler.getASTContext(), compiler.getFileManager(),
        config.getASTContext(), config.getFileManager(),
        /*minimalImport=*/true);
}
```

clangAddons/lib/Vectorizing/IntrinsicCollector.cpp
AST Merging

- second way: ASTImporter
- getting a persistent ASTContext and FileManager from the source compiler in a separate compilation step:

```cpp
class ParseConfigurationConsumer : public ASTConsumer
{
    Configuration& config;
    llvm::OwningPtr<CompilerInstance>& compiler;

    virtual void HandleTranslationUnit(ASTContext &C)
    {
        // ...
        if (!compiler->getDiagnostics().hasErrorOccurred()) {
            config.m_ASTContext.reset(&compiler->getASTContext());
            config.m_FileManager.reset(&compiler->getFileManager());
            compiler->resetAndLeakASTContext();
            compiler->resetAndLeakFileManager();
        }
    }
};
```

ClangAddons/lib/Vectorizing/Configuration.cpp
4. Future Directions

Can Clang become a suitable tool for source-to-source transformations?
Future Opportunities: Projects

- User-defined pragmas (OpenMP)
- AST Transformation
  - transform clang-parsed AST
  - target source code
  - annotated C++

Olaf Krzikalla
Future Opportunities: Projects

- Generates extended AST
- C++ extended
- clang-Parse extended
- Transforms to pure clang AST
- Target source code
- C++ (extended)
generate user-defined IR

creates clang AST

target source code

user-defined language

user-defined Parser

AST creation

C(++)

DSL
Future: Zoom out to Strategy
Discussion

http://scout.zih.tu-dresden.de/
Future Opportunities: Code

- things that very probably might not work:
  - mirror the AST \( \rightarrow \) duplicates functionality
  - rewrite, parse and rebuild the AST as often as possible \( \rightarrow \) too slow
- keep the \texttt{StmtEditor} interface
  - extended with operator overloading
- backup the implementation with \texttt{Sema}
  - enriched with machine-evaluatable diagnostics
- hard task no.1: maintain the \texttt{Sema} state
- hard task no.2: replacing statements

Can Clang become a suitable tool for source-to-source transformations?
Is the integration of an ASTProcessing lib in clang desired?
Using TreeTransform

- task: transform $a += b$ to $a = a + b$
- old code never really worked
- example from one year ago:

```cpp
// x += y ⟷ x = x + y for arbitrary ops:
typedef CompoundAssignOperator CAO;
for (stmt_iterator<CAO> i = stmt_ibegin<CAO>(Root),
     e = stmt_iend<CAO>(Root); i != e; ++i) {
    CAO* Node = *i;
    BinaryOperator::Opcode binOpc = transformOpc(Node->getOpc());
    Expr* clonedLhs = Clone_(Node->getLHS());
    clonedLhs->setValueKind(VK_RValue);       // the tricky part
    replaceStatement(
        Node,
        Assign_(Node->getLHS(),
            BinaryOp_(clonedLhs, Node->getRHS(), binOpc)));
}
```