OO-Lint: Preparing C++ Codes for Operator Overloading

19th Euro AD Workshop

Alexander Hück
alexander.hueck@sc.tu-darmstadt.de

This is joint work with
• Christian Bischof
  christian.bischof@sc.tu-darmstadt.de
• Jean Utke
  jutke@allstate.com
Motivation

• Techniques to perform automatic differentiation (AD):
  • Source transformation (e.g. ADIFOR, ADIC, OpenAD, TAPENADE)
  • Operator Overloading (e.g. ADOL-C, CoDiPack, dco)
  • Combinations (e.g. ADiMat)

• Operator Overloading (OO)
  • OO allows the introduction of AD just by redeclaring the underlying floating point type with a suitable AD type.
  • In reality, OO will most likely „break“ an existing code
Semantic Augmentation

A type change from double to the user-defined class adouble
Semantic Augmentation

Ideally the type is already defined by a typedef statement

- typedef double scalar;

Augmented by changing the typedef statement

```c
1 #include "adouble.h"
2 typedef adouble double scalar;
3 scalar f(scalar x) {
4    scalar y = x * x + x;
5    return y;
6 }
```

```c
1 typedef double scalar;
2 scalar f(scalar x) {
3    scalar y = x * x + x;
4    return y;
5 }
```
Type Change in OpenFOAM (1)

```c
if (is.version() == 2.0) {...}
```

After type change the following error message appears:

```c
In file included from lnInclude/Field.H:367:0,
   from lnInclude/labelField.H:40,
   from lnInclude/primitiveFields.H:38,
   from lnInclude/pointField.H:37,
   from lnInclude/cellModel.H:43,
   from lnInclude/cellModeller.H:41,
   from lnInclude/globalCellModeller.C:30,
   from Make/linux64GccDPOpt/global.C:73:
lnInclude/Field.C: In constructor ‘Foam::Field<Type>::Field(const Foam::word&, const Foam::dictionary&, Foam::label)’:
lnInclude/Field.C:256:33: error: no match for ‘operator==’ in ‘is.Foam::IOstream::version() == 2.0e+0’
lnInclude/Field.C:256:33: note: candidates are:
lnInclude/IOstream.H:157:22: note: bool Foam::IOstream::versionNumber::operator==(const Foam::IOstream::versionNumber&)
lnInclude/IOstream.H:157:22: note: no known conversion for argument 1 from ‘double’ to ‘const Foam::IOstream::versionNumber&’
```
Type Change in OpenFOAM (2)

```cpp
struct adouble {
    adouble(double a);
};

struct IOstream {
    struct version {
        version(scalar vn);
        bool operator==(const version& other);
    };
};

if(is.version() == 2.0){...}
```
Type Change in OpenFOAM (2)

```plaintext
struct adouble {
    adouble(double a);
};

struct IOstream {
    struct version {
        version(scalar vn);
        bool operator==(const version& other);
    };
};

if(is.version() == 2.0) {...}
```
struct adouble {
    adouble(double a);
};

struct IOstream {
    struct version {
        version(scalar vn);
        bool operator==(const version& other);
    };
};

if(is.version() == 2.0){...}
Type Change in OpenFOAM (2)

```cpp
struct adouble {
    adouble(double a);
};

struct IOstream {
    struct version {
        version(scalar vn);
        bool operator==(const version& other);
    };
};

if(is.version() == 2.0){...
```
Problematic Code Constructs

There are several C++ code constructs causing compile time errors

- Need for identification and strategies to solve code constructs where operator overloading „fails“
- The language is evolving: Code adhering to the C++03 language standard is more limited than code targeting C++11
Problematic Code Constructs

Implicit Conversions

- Multiple *implicit conversions* on the same value
  - “At most one user-defined conversion [...] is implicitly applied to a single value.” [C++’03 Standard, §12.3-4]
- Also:
  
  ```
  scalar a = ...;
  int b = a;
  ```

Implicit (boolean) conversion (inside conditions)

  ```
  scalar a = ...;
  if(a) { ... }
  ```

  ```
  scalar a = ...;
  if(!a) { ... }
  ```

Make conversions explicit

Make explicit comparison
Problematic Code Constructs

Explicit conversion of `scalar` values

```c++
scalar a = ...;
int value = int(a);
```

- C++11: explicit conversion function `explicit operator int();`

Unions with `scalar` member

```c++
union {
    float a;
    scalar b;
}
```

- Types "with non-trivial constructor [...] cannot be a a member of a union" [C++’03 Standard, §9.5]
- Somewhat relaxed with the C++11 standard
Solution

1) Develop coding standards for numerical tools that avoid problems when applying OO.

2) Develop coding standards for OO tools that avoid problems.

For Legacy Codes we developed OO-Lint

- Flags problematic code locations
- Ideally auto-corrects them
- Uses the LLVM/Clang infrastructure
- [https://www.github.com/ahueck/opovlint](https://www.github.com/ahueck/opovlint)
OO-Lint Architecture
based on LLVM/Clang

In:
- Compilation Database
- Sources
- Config. File

Matcher Modules
- Implicit Conversion
- Boolean Conversion
- Explicit Conversion
- Union
- ...

Out:
- Lint-like Messages
- Replacements
AST Matcher Example

**AST Matcher**

Represents queries over the C++ AST:

```
    AST
      ↓
   MatchFinder
        ↓
        1..n
AST Matcher
```

Find implicit conversions of floating point-types inside `if` conditional of the form:

```cpp
    scalar a = 1.0; or double a = 1.0;
    if(a) { … }
```
AST Matcher Example

```c
scalar a = 1.0;
if(a) { ... }
if(1) { ... }
if(a + 1) { ... }
```

Matcher Expression:
```
ifStmt();
```

Shortened AST for: `if(a)`
AST Matcher Example

```c
scalar a = 1.0;
if(a) { ... }
if(1) { ... }
if(a + 1) { ... }
```

Matcher Expression:
```
ifStmt(hasCondition(implicitCastExpr()));
```

Shortened AST for: if(a)
AST Matcher Example

```c
scalar a = 1.0;
if(a) { … }
if(1) {...}
if(a + 1) { … }
```

Matcher Expression:
```
ifStmt(hasCondition(
    implicitCastExpr(isFloatingToBoolean()))
));
```

Shortened AST for: `if(a)`
```
- ImplicitCastExpr '_Bool' <FloatingToBoolean>
- ImplicitCastExpr 'scalar':'double' <LValueToRValue>
- DeclRefExpr 'scalar':'double' lvalue Var 'a'
```

Custom matcher extension
**AST Matcher Example**

```c
scalar a = 1.0;
if(a) { ... }
if(1){...}
if(a+1){...}
```

Matcher Expression:
```
ifStmt(hasCondition(implicitCastExpr(isFloatingToBoolean(),
    hasSourceExpression(ignoringImpCasts(declRefExpr("a")))))
));
```

Shortened AST for: `if(a)`
```
| `-ImplicitCastExpr 'bool' <FloatingToBoolean>
| `ImplicitCastExpr 'scalar':'double' <LValueToRValue>
| `DeclRefExpr 'scalar':'double' lvalue Var 'a'
```
Replacement

Container storing position and replacement string

- Collected and serialized to disk (yaml format)
- Applied by external Clang tool “clang-apply-replacement”

```c
scalar a = 1.0;
if(a) { ... }
```

```c
scalar a = 1.0;
if(a != scalar(0.0)) { ... }
```

---

**MainSourceFile**: /project/src/main.cpp

**Replacements**:
- **FilePath**: /project/src/main.cpp
- **Offset**: 317
- **Length**: 0
- **ReplacementText**: ' != scalar(0.0)'
Analyzing OpenFOAM
(www.openfoam.org)

OpenFOAM 2.4.x

- 2360 translation units

<table>
<thead>
<tr>
<th>Code Construct</th>
<th># Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit Conversion</td>
<td>6</td>
</tr>
<tr>
<td>Boolean Conversion</td>
<td>2</td>
</tr>
<tr>
<td>Explicit Conversion</td>
<td>57</td>
</tr>
<tr>
<td>Union</td>
<td>1</td>
</tr>
</tbody>
</table>
Evolution of ISSM w.r.t. Problematic Code Constructs

Ice Sheet System Model (https://issm.jpl.nasa.gov/)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Late 2011</th>
<th>Mid 2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td>912</td>
<td>857</td>
</tr>
<tr>
<td>LOC</td>
<td>66,573</td>
<td>80,044</td>
</tr>
<tr>
<td>Translation Units</td>
<td>298</td>
<td>254</td>
</tr>
<tr>
<td>#Explicit Conversion fix in #files</td>
<td>170 in 42</td>
<td>0</td>
</tr>
<tr>
<td>#Implicit Bool Conversion fix in #files</td>
<td>41 in 8</td>
<td>0</td>
</tr>
<tr>
<td>#Explicit Conversion match in #files</td>
<td>192 in 46</td>
<td>0</td>
</tr>
<tr>
<td>#Implicit Bool Conversion match in #files</td>
<td>44 in 10</td>
<td>0</td>
</tr>
</tbody>
</table>
On Performance

Static code analysis is slower than compiling

• Each translation unit is analyzed separately, can be parallelized

Compiling/Analyzing a part of OpenFOAM-2.4 with 4 Threads/Processes:

• Compilation with Clang: 1m17s
• Static Analysis: 4m23s (Overhead factor: ~3.4)
Summary and Conclusions

OO-Lint:

• Simplifies OO-preprocessing of C++ codes

• Template handling is missing

```cpp
typedef double scalar;
function<scalar>(...) <-> function<double>(...)
```

• `typedef` information is not present after template instantiation

• Can be extended for the automatic redeclaration of floating point types
References
