Meta-compilation

Michael Donohue

Coverity

What is metacompilation?

- I didn't know either

> •**Metacompilation** is a computation which involves metasystem transitions (MST) from a computing machine *M* to a metamachine *M'* which controls, analyzes and imitates the work of *M*. Semantics-based program transformation, such as partial evaluation and supercompilation (SCP), is metacomputation.
>
> -Wikipedia

What is compilation?

```
void GetBirth(int x) { char
query[100]; snprintf(query,
sizeof(query), "select * from
person where id = %i", x);
GetMysql(query); eprint( 0,
"<font> %s's Birth date is:
%s</font>\n", row[1], row[2]); }
```

→

```
leal    4(%esp), %ecx
andl    $-16, %esp
pushl   -4(%ecx)
pushl   %ebp
movl    %esp, %ebp
pushl   %ecx
subl    $16, %esp
movl    $0, -12(%ebp)
jmp     .L2
```

What is compilation?

- To some, it is the backend optimizer – intermediate representations, optimizations, instruction selection, register allocation
- To others, it is all the techniques used for parsing
- For Coverity, compilation is parsing and abstract syntax trees, with some help from the backend analysis

Compilation at Coverity

- For C/C++, compilation takes source code and builds abstract syntax trees
- The abstract syntax trees are directly used for analysis, in contrast with the traditional compilation step of using an intermediate representation
- Coverity has different goals than a compiler - we want to explain to a human how a bug can occur

Compilation at Coverity

- For Java, compilation starts at the bytecode generated by the java compiler
- Parsing consists of reading bytecode, and verifying all appropriate debugging information is included
- Why do we need debugging information?

What is metacompilation?

- Using compiler algorithms
- Do something beside generate code
- Find many defects

Interpretation

- Parsing allows us to understand the structure of the code
- Compiler techniques allow us to understand the relationship between statements in the code
- Interpretation means we walk down every path of the code
- Our technique is called “Abstract Interpretation” because we leave some values abstract

Finding a bug

```
public static void foo(Object a) {
   if(a == null) {
      System.out.println("a is null");
   }
   System.out.println(a.toString());
}
```

The analysis sees

```
public static void foo(Object a) {
  if(a == null) {
    System.out.println(“a is null”);
  }
  System.out.println(a.toString());
}
```

```
0:   aload_0
1:   ifnonnull       12
4:   getstatic       #5;
7:   ldc     #6;
9:   invokevirtual   #7;
12:  getstatic       #5;
15:  aload_0
16:  invokevirtual   #8;
19:  invokevirtual   #7;
22:  return
```

Execution of the bytecode

```
0:   aload_0
1:   ifnonnull      12
4:   getstatic      #5; //Field System.out
7:   ldc     #6; //“String a is null”
9:   invokevirtual   #7; //Method java/io/PrintStream.println
12:  getstatic       #5; //Field System.out
15:  aload_0
16:  invokevirtual   #8; //Method java/lang/Object.toString
19:  invokevirtual   #7; //Method java/io/PrintStream.println
22:  return
```

Abstractions

- The previous example shows a bug in the null vs not-null abstraction
- The only values we tracked in the execution were “null” “not-null” or “don’t know”
- No explicit pointer values were calculated

Abstractions

- But the example is contrived
- Using only those three values, we get a false-positive here:

```
public static void foo(Object a, int b) {
   if(a == null && b == 7) {
      a = new Object();
   }
   if(b == 7) {
      System.out.println(a.toString());
   }
}
```

Abstractions

- A null-pointer abstraction finds bugs, but it can’t tell whether 7 == 7.
- An integer abstraction can figure out the sevens, but it doesn’t find null pointer bugs
- Solution: Run them together – the integer abstraction can tell you about impossible combinations, while the null-pointer abstraction tells you about bugs

Abstractions

- Without path pruning, this method has 6 paths
- Integer pruning eliminates two of those – and thus eliminates the false positive

```
public static void foo(Object a, int b) {
   if(a == null && b == 7) {
      a = new Object();
   }
   if(b == 7) {
      System.out.println(a.toString());
   }
}
```

Abstractions

- If the abstraction eliminates impossible combinations we call it a “False Path Pruner”
- If the abstraction finds defects, we call it a “Checker”
- Abstractions don’t communicate with one another

False Path Pruners

- Integer constants
- Type checks
- Null and nonnull values

Limits of abstraction

- Examples so far have been sound
- Tracking values in the heap is difficult
- We allow false negatives

```
public class Tree {
    Tree left;
    Tree right;
    public int count() {
        return left.count() + right.count();
    }
    public static test() { new Tree().count(); }
}
```

Going deeper

- Finding local bugs is nice
- Not likely to get people excited about the technology
- Lets go interprocedural
- Where do we start?

Going deeper

- We already have a good local analysis
- We know how to do compiler optimizations
- Two compiler based phases stand out – code generation and method inlining
- First we generate code
- Then we inline it

Transform the code

```
public void foo(Object a, Object b) {
  if(a == null) {
    System.out.println(“a is null”);
  }
  printIt(a);
}

Public void printIt(Object obj) {
  System.out.println(obj.toString());
}
```

```
public void foo(Object a, Object b)
{
  if(a == null) {
    System.out.println(“a is null”);
  }
  // this came from printIt()
  a.toString();
}
```

Getting the code

- It turns out that commercial software shops don't know where their code is
- This is a huge problem for C and C++
- Headers must be found, classnames have no relationship to the filenames
- Java solved all of this – filenames must rigidly match their package name, and there are no includes

Getting the code

- But, commercial software shops don’t even know where their Java code is
- An open-source corollary is the Eclipse project – there are hundreds of plugins and each plugin has a separate code base
- However, everyone knows how to build their software – they have to, otherwise they couldn’t release it
- The solution is to mine the data we need out of the build process

Failing to get the code

- For C, nearly everyone uses make
- Idea: run ‘make’ in verbose mode, save all the commands in our own file, and then rerun them later
- The ‘rerun’ them part turns out to be highly context sensitive. Running ‘deltree /y .’ without an appropriate ‘cd output_dir’ preceding it has very unexpected results

cov-build

- Our solution is to invisibly wrap around the build process for a piece of software
- Intercept all calls to the compiler and understand the command line options
- Save a copy of all input files to the compiler
- Analyze later

Customer site visits

- “Eclipse already does this”
- “Stop denigrating lint!”
- Commercial software really is different than open-source
- C programmers make poor use of Java tools

Demo

Checkers

- Null-pointer issues
- Resource leaks
- Incorrect use of a database connection

An example

```
if(a) {
   a->init();
}
a->start();
```

FunctionBody

if a->start();

a a->init();