02247 Compiler Construction

Matthias Larsen, s103437 27/04/2016

Introduction

A compiler is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language), with the latter often having a binary form known as object code. The most common reason for converting source code is to create an executable program.[1]

Copy and pasting happens a lot of the world of computer programming. The reason for copy and pasting varies greatly and can be anything from inexperienced programmers not knowing how to solve a given task to the programmer simply being lazy.

The act of copy and pasting also comes with a very high risk of side effects. Bugs can easily be introduced by assumptions and design decisions made in the original source that no longer applies in the new place. Such code can result in immediate and visible errors from i.e. undefined variables, but may also be subtle bugs due to variables meaning something different in the new context.

LLVM, originally “Low Level Virtual Machine", is a compiler infrastructure with a set of reusable libraries, well defined interfaces and easily extensible. LLVM uses multiple levels of “intermediate representation" (IR) as it goes through it's passes, analyzing and optimizing, to end up in the target language.

This paper describes the experience of extending LLVM with a pass to detect possible copy and paste errors in a program.

Implementation

The goal of the pass is to detect copy-paste statements or blocks and to warn the user of potential problems in a program. The project started with a naive approach to gain knowledge of LLVM and its API. Following that, the entire pass was rewritten from scratch exploiting more of the LLVM API to faster and better find call sites and duplicates. The following sections describes what was implemented in which iteration, what did not work and what was changed.

First attempt

The first attempt at a parser was very crude. It started with a naive approach as a FunctionPass, thus limiting the scope of the copy/paste detection to a single method at a time.

As a starting point, the initial idea was to check every line of the program against the rest of the program, to see if it is identical, similar or not at all.

The LLVM framework provides two methods that can assist with this; `isIdenticalTo` and `isSimilarTo`. The first one checks if the instruction is 100% identical, where as the latter checks if the method call is

[1] https://en.wikipedia.org/wiki/Compiler

the same. The `isSimilarTo` part would then have to be checked further in order to see if its a possble error. It's also the method we have to use in order for us to check for if a method has been pasted wrongly and should be identical to another instruction.

A FunctionPass was chosen for the initial implementation as it was based on the SkeletonPass with the idea to start solve the problem at a function level before expanding to a whole file / project. This version of the pass could take the program shown in listing 1 as input and produce the output shown in listing 2.

```c
#include <stdio.h>
int main( )
{
  printf("Hello World!\n");
  printf("Hello World!\n");
  if (1) {
    printf("Hello World!!\n");
    printf("Hello World!\n");
  }
}
```

Listing 1: test1.c

```
Identical method found. Possbile copy/paste
Original call: printf @ ../../test3.c:4, col 3
  Possble paste error: printf @ ../../test3.c:5, col 3
  Same operation found. Needs deeper check
  Possble paste error: printf @ ../../test3.c:7, col 4
  Possble paste error: printf @ ../../test3.c:8, col 4
```

Listing 2: Pass output for test1.c

Second attempt

I realised that my first attempt was very naive and would result in extremely long running times if the program was big enough. After reading a bit about the LLVM API, I found that by exploiting some of the LLVM methods I could increase the effeciency of the pass by somewhat comprising the flow, making it a bit more unintuitive.

Finding possible errors

Instead of checking one line against the rest of the lines in the function (in case of FunctionPass), I found that I could use the function class to get all the call sites and compare those directly with each other instead, thus drastically reducing the time complexity of the pass.

```cpp
virtual bool runOnFunction(Function &F) {
  std::unordered_set<std::string> visited;

  for (inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    // Get the current method name to be called
    StringRef funcName = getFunctionName(I);

    if (visited.count(funcName.str()) > 0) {
      // We already evaluated this function name
      continue;
    }
    visited.insert(funcName.str());
    int funcLine = getFunctionLine(I);

    // ... snip, method analysis ...
  }
}
```

Listing 3: FunctionPass structural overview

The pass starts by trying to parse the current instructions function name as shown in the structural overview of the pass shown in listing 3. This will resolve to `printf`, `add`, `someMethod` etc. in the case of an actual method call, while `int b` will resolve to the fallback “-1". It also uses the name of the function being called to avoid analysing it multiple times by caching it to a list of visited methods, `visited`.

The `getFunctionName` method, as shown in listing 4, is an auxiliary made for convenience. LLVM is complex and thus, getting the name of the function being called in an instruction is not straight forward.

```
StringRef get_function_name(CallInst *call)
{
    Function *fun = call->getCalledFunction();
    if (fun) // thanks @Anton Korobeynikov
        return fun->getName() ; // inherited from llvm::Value
    else
        return StringRef("indirect call");
}

StringRef getFunctionName(inst_iterator it) {
  CallInst* callInst = dyn_cast<CallInst>(&*it);
  if (!callInst) {
    return "-1";
  }
  return get_function_name(callInst);
}
```

Listing 4: getFunctionName auxiliary method

The method analysis is the core of the pass. If the name of the function being called was parsed from `getFunctionName`, the function name is looked up in the module symbol table. It’s then possible to iterate a list of uses, which represents instructions that’s also using the function. If the function name was not able to be parsed i.e. in the case of a declaration, the pass skips to the next instruction in the program being analyzed.

```
// If its not a direct call
if (funcName != "-1") {
  Module * m = F.getParent();
  Function * f = m->getFunction(funcName);

  // Get all the uses of this method call (including this one)
  for (User *U : f->users()) {
    if (Instruction *Inst = dyn_cast<Instruction>(U)) {
      int curLine = getFunctionLine(Inst);
      if (curLine == funcLine) {
        // This is the same line as we originate from..
        continue;
      }

      // Deeper analysis
    }
  }
}
```

Listing 5: Finding similar call sites

Comparing instructions

Before instructions can be compared and error or warning messages issued, a definition of when and what constitutes a (possible) copy-paste error needs to be estabilished.

A Survey on Software Clone Detection Research by Chanchal Kumar Roy and James R. Cordy[2] describes 4 types of clones. By textual similarity:

Type I Identical code fragments except for variations in whitespace (may be also variations in layout) and comments.

Type II Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.

Type III Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.

As well as functional similarity: If the functionalities of the two code fragments are identical or similar i.e., they have similar pre and post conditions, we call them semantic clones and referred as Type IV clones.

Type IV Two or more code fragments that perform the same computation but implemented through different syntactic variants.

These types of clones increases in both level of subtlety and analytical complexity and difficulty in detecting such clones from Type I through Type IV, with Type IV being the highest and most difficult to detect.

By iterating through all call sites for a function and comparing them would effectively find Type I clones. The LLVM API has a method `isIdenticalTo` on the `Instruction` class that returns true, if two instructions are exactly the same, as previously mentioned. As LLVM ignores whitespace and comments when generating the IR a simple use of `isIdenticalTo` is all that's needed to report identical

[2] http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf

clones on a line by line basis of the analyzed program. I found that the `isSimilarTo` did not work as expected and as such, was replaced by comparing arguments of the `CallSites` as described later.

Type II clones can be found by inspecting all operands of the instructions being compared when the `isIdenticalTo` check returns false.

Qualifacation of an error When a clone is found, it needs to be determined if it should be reported to the user or not. For the purpose of this pass, a simple decision process has been made. If two statements are identical, it's reported to the user under the assumption that it's only in very rare cases that one would want identical statements. Using Listing 1 as input, the first error will be;

```
Identical statements (prinft) found on lines 4 and 5. Did you mean to?
```

If the statements are not identical, they should be analyzed further. First, the number of arguments in each instruction should be compared. The current implementation does not analyze instructions for which the number of arguments differ – This is Type III clones (changed statements by adding or removing arguments). When the arguments are being compared, a counter keeps track of how many of the arguments are the same and how many differs. These numbers are used to check if an error or warning should be output to the user. If the number of arguments is larger than two while either one argument differs or only one argument is the same, a warning is issued[3].

Strings If an argument is of the type `Type::TypeID::ArrayTyID` the string representation of it can be found by the snippet in listing 6 given an `Instruction * I` and `int i` being the index of the argument.

```
CallSite CSI(cast<Value>(&*I));
std::string argI = cast<ConstantDataArray>(
  cast<GlobalVariable>(
    cast<ConstantExpr>(
      CSI.getArgument(i)
    )->getOperand(0)
  )->getInitializer()
)->getAsCString();
```

Listing 6: Getting method argument as a string

The type of an argument can be found by `Inst->getOperand(i)->getType()->getTypeID()`.

In the case of a simple `printf` instruction with string arguments, it's possible to do a simple check to see if any of the two pair-wise arguments is a substring of the other, or if they differ more than that. Listing 7 shows the simplified comparison performed on each argument pair. The string comparison of the pass uses a few more checks at the snipping places, in order to give more verbose messages.

```
int dist = edit_distance(argI, argInst);
if (argI == argInst) {
  // Identical
} else {
  if (argI.find(argInst) != std::string::npos) {
    // argInst is a substring of argI
    // i.e. something was removed
```

[3]Not fully implemented for warnings on string arguments

```
    // ... snip ...
  } else if (argInst.find(argI) != std::string::npos) {
    // And the other way around
    // i.e. something was added
    // ... snip ...
  } else if (dist < MAX_EDIT_DIST) {
    // Levenstein distance
    // ... snip ...
  }
}
```

Listing 7: Compare string arguments

Using this string comparison, the program from listing 1 would give the more verbose output shown in listing 8.

```
Identical statements (printf) found on lines 4 and 8. Did you mean to?
On line 7 argument #1 differs from first sight on line 4:
   'Hello World!' is a substring of 'Hello World!!'.
 Did you intend to add '!' to the end?
Identical statements (printf) found on lines 4 and 5. Did you mean to?
```

Listing 8: Compare string arguments, verbose output

Tokens A token can either be a variable or an integer[4]. At the current state of the implementation, the pass is able to compare tokens if they're either of type `LoadInst` or `ConstantInt`. Listing 10 shows the output when the pass is run on the program from listing 9.

```
 1 #include <stdio.h>
 2
 3 int add(int a, int b, int c)
 4 {
 5   return a+b+c;
 6 }
 7
 8 int main( )
 9 {
10   int a = 2;
11   int b = 3;
12   add(a, a, a);
13   add(b, a, a);
14   return 0;
15 }
```

Listing 9: test2.c

```
On line 13 argument #1 differs from first sight on line 12:
   b
 instead of
   a
```

[4]The pass has only been tested with simple types; integers, string and variables.

```
Is this intended?
```

Listing 10: Pass output for test2.c

If line 13 in listing 9 was changed to “`add(1, a, a);`", the program would output “`1 (32 bit int)`" instead of “`b`" when asking if it was intended. If the analysis can't parse the token it's, currently, simply output as IR so the user can decide himself if it's a possible error. Using the example from above, replacing the `b` in the `add` method with the arithmic operation `a+b`, yields the error message shown in listing 11.

```
On line 14 argument #1 differs from first sight on line 13:
     %9 = add nsw i32 %7, %8, !dbg !26
 instead of
   a
 Is this intended?
```

Listing 11: Pass output for modified test2.c

Future work

There's a lot of room for improvement in the implementation. More advanced types like arithmic operations, other methods as function arguments and pointers are just a few examples. The error or warning messages could also be more intelligent and the detection itself could be expanded to cover code blocks or detect copy and paste in modules or even across multiple files.

This section highlights possible way the current implementation could be extended.

Improving detection

Handling arithmic expressions (such as `a+b` and `a*a`) could be solved by refactoring the code that compares two instructions. The Value class (as most, if not all LLVM classes extend this class) could converted to a string representation before running the string comparison code – this would make it easier to handle complex cases than with the current implementation.

That is, to convert the IR, `%5 = add nsw i32 %3, %4,` dbg !22!, to a string, `a+a`, so that the program could make a string comparison for literals and give more verbose suggestions as shown in listing 8. Doing so would allow checking complex arguments with nested arithmic operations as well. Pseudo code for such an algorithm is shown in listing 12.

```
processExpression(value):
  sequence <- initialize
  visit(sequence, value)
  generateSprintfCallFromSequence(sequence)

visit(sequence, value):
  if value is load:
    sequence.add(load.pointer)
  else if value is binaryop:
    sequence.add(openingParen)
    visit(sequence, binaryop.operand(0))
    sequence.add(binaryop.opcode)
    visit(sequence,  binaryop.operand(1))
    sequence.add(closingParen)
```

Listing 12: Pseudo code for improving comparison of method arguments

Adding the opening and closing paranthesis would be needed to distinguish `(b+c-d)*e` from `b+(c-d)*e`.

Detecting blocks

Detection of blocks instead of a line by line basis can be done in two ways. The first is to introduce some "lookahead" functionality to the current pass. That is, to increment both pointers (`I` and `Inst`) as long as they're similar enough and then trigger a message when encountering a line that is above a given threshold.

The other way is to encode or serialize parts of the code as a graph and compare graphs to find blocks of duplicate code even though graph isomorphism is a hard problem. Detecting blocks can be a very difficult problem, but if done correctly, one could possibly detect Type IV clones.

Conclusion

A few difficulties arose when trying to get the variable name from a load instruction such that function arguments in `add(a, a)` vs `add(a, b)` could be compared in a meaninful way. Therefore, only a basic copy and pasting detector has been implemented, with a lot of potential for extensibility. However, the LLVM API is enourmous and only a small subset of the possible functionality has been used in the implemented pass. Given more knowledge of the LLVM API, the pass could be optmized and made more intelligent by leveraging more of the information about the program and its instructions, that the lower level classes of LLVM holds. Even though there is plenty of room for extensibility in the pass a rudimentary detector could prove useful.

Side effects of copy and pasting and forgetting or missing to rename variables can lay hidden for a long time before being discovered – and in worst case, in production enviroment.

How and what to detect and classify as a possible mistake is no small feat either. This paper suggests when similar statements could be candidates for a mistake on the programmers part and shows how this could be implemented.