02247 Compiler Construction

Matthias Larsen, s103437 27/04/2016

Introduction

Implementation

First attempt

My first attempt at a parser was very crude. I 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 is 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 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.

I chose to use a FunctionPass as I based my initial implementation on the SkeletonPass and thought to start solve the problem at a function level before expanding to a whole file / project.

This version of the pass could take the following code as input

```
#include <stdio.h>
int main( )
{
  printf("Hello World!\n");
  printf("Hello World!\n");
  if (1) {
    printf("Hello World 2!\n");
    printf("Hello World!\n");
  }
}
```

and would produce the folling warnings

```
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
```

Second attempt

I realised that my first attempt was very naive and would result in extremely long running times if the code 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 duplicates

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 with each other instead, thus drastically reducing the time complexity of the Pass.

Comparing instructions

Future work

I had a few difficulties getting the variable name from a load instruction such that I could compare function arguments in ‘add(a, a)’ vs ‘add(a, b)’. This section is my thoughts on how I’d extend the pass, had I time.

Improving detection

Handling arithmic expressions (such as a+b and a*a) could be solved by creating a method that could take a Value as input and convert it to a string representation – this would make it easier to handle complex cases than the current code. That is, to convert ‘Doing so would allow checking complex arguments with nested arithmic operations as well.

Pseudo code:

```
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)
```

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

Detecting blocks

Detecting blocks 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 the same and then trigger a message when encountering a line that is just similar or way off.

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.

Conclusion