02247 Compiler Construction

Matthias Larsen, s103437 27/04/2016

Assignment 4

At this point, the detector can analyse differences in call arguments for strings, integers and single literals and hint to the user if anything looks like its out of place. Complex literals i.e. arithmic expressions are just checked for equality and IR is outputted directly to the user if they're different and deemed a mistake. The pass currently takes one function at a time (FunctionPass) and for each instruction looks up other occurences of that instruction via Module::getFunction(functionName) and then compares the two.

However, one of the very first things the method does, is to grab the Functions parent (the module) in order to fetch similar function calls. I imagine, that it'd be quite simple to reverse the process – that is, to start from a ModulePass and then for each function in the module, ask the initial module for similar calls.

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.