CS 164 Programming Languages and Compilers **Spring 2019**

Discussion 4/29: Code Optimization

Instructor: Paul N. Hilfinger GSIs: Nikhil Athreya, Vivant Sakore

1 Optimizations

Consider the following basic block

```
p = 3
r = 10
s = p + r
t = 2*r + s
t = p
u = p + r
v = p + t
w = 3 + x
```

For each of the following blocks, state what optimization(s) were performed on the above block.

1.
```
p = 3
r = 10
s = p + r
t = p
u = p + r
v = p + t
w = 3 + x
```

Answer: Dead Code Elimination

2.
```
p = 3
r = 10
s = p + r
t = 2*r + s
t = p
u = s
v = p + t
w = 3 + x
```

Answer: Common Subexpression Elimination

3.
```
p = 3
```

```
r = 10
s = p + r
t = 2*r + s
t = p
u = p + r
v = p + p
w = 3 + x
```

Answer: Copy Propagation

4.

```
p = 3
r = 10
s = 13
t = 33
t = 3
u = 13
v = 36
w = 3 + x
```

Answer: Constant Folding and Constant Propagation

2 Flow Analysis

Java supports subtype polymorphism by dynamically dispatching methods at execution time, depending on the dynamic type of the receivers. However, dynamic dispatches take longer than calls to known functions. We'd like to do a static flow analysis to improve programs by determining calls with unique, statically known dynamic types, so that these can be changed into direct method calls. Specifically, we'd like a conservative global flow analysis that allows us to determine if a call of the form `V.M(...)`, where `V` is a variable and `M` is a method name, must definitely refer to a particular, known method body. Aside from `if` conditionals, loops, and call statements, we'll restrict our attention to statements of the form:

1. `V1 = new T(...);` // Create a new `T`.
2. `V1 = (T1) V2;` // A cast to type `T1`.

Assume that all variable names are forward declared with types at the beginning of the scope, as in old-style parameter declarations in C.

For simplicity, assume that all types of the forward declarations are compatible with the `new` and `cast` statements. Also assume a cast to an undeclared variable is never performed, variables of the same name are never created with `new` more than once, and restrict the type of the cast to be in the type hierarchy of the original object. You do not need to handle method overriding in this question (if a child and parent class both implement a method `m`, then you do not need to select the child method over the parent method).

1. Describe a flow analysis that implements the above specification. What is a suitable set of abstract values for the purpose of this analysis? What are the initial values? What are the transfer rules of your flow analysis?

Answer:

Use a forward analysis involving rules of the form `Din(V,s)` and `Dout(V,s)`, where `V` is a variable and `s` is a statement. Let the set of all types relevant to the program of interest be called H. The abstract values for this analysis is the power set lattice of H. In this lattice, the bottom element is the empty set, any proper subset of a set is lower than the set itself, and the top element is H. The lowest upper bound (lub) operator is the union operator. Initialize every `Din` and `Dout` to bottom (the empty set). The transfer rules are:

$$\texttt{Din}(V1, s) = \text{lub}\{\texttt{Dout}(V1, p) \mid pred(p, s)\}$$

$$\texttt{Dout}(V1, s) = \begin{cases} T \cup \{X \in H \mid subtype(T, X)\} & s \equiv \texttt{V1 = new T(...);} \\ T1 \cup \{X \in H \mid subtype(T1, X)\} \cup \texttt{Din}(V2, s) & s \equiv \texttt{V1 = (T1) V2;} \end{cases}$$

where $pred(x, y)$ means that x is a predecessor of y in the CFG, and $subtype(p, q)$ means that p is a proper subtype of q.

After running the analysis, we analyze the results. If the analysis yielded a non-empty set for `V` right before `V.M()` is called, search amongst the types in the set for the method `M`, and if there is exactly one such method, return the corresponding type's method. Otherwise, annotate `V.M()` with "not sure".