Lecture Notes: Interprocedural Analysis

15-819O: Program Analysis
Jonathan Aldrich
jonathan.aldrich@cs.cmu.edu

Lecture 8

1 Interprocedural Analysis

Interprocedural analysis concerns analyzing a program with multiple procedures, ideally taking into account the way that information flows among those procedures.

1.1 Default Assumptions

Our first approach assumes a default lattice value for all arguments L_a and a default value for procedure results L_r

We check the assumption holds when analyzing a call instruction or a return instruction (trivial if $L_a = L_r = \top$)

We use the assumption when analyzing the result of a call instruction or starting the analysis of a method. For example, we have $\sigma_0 = \{x \mapsto L_a \mid x \in \mathbf{Var}\}$.

Here is a sample flow function for call and return instructions:

$$\begin{aligned} f[\![x := g(y)]\!](\sigma) &= [x \mapsto L_r]\sigma && \text{where } \sigma(y) \sqsubseteq L_a \\ f[\![\text{return } x]\!](\sigma) &= \sigma && \text{where } \sigma(x) \sqsubseteq L_r \end{aligned}$$

We can apply zero analysis to the following function, using $L_a = L_r = \top$:

$$
\begin{array}{ll}
1: & \text{procedure } \textit{divByX}(x : int) : int \\
2: & \quad y := 10/x \\
3: & \quad \text{return } y \\
4: & \text{procedure } \textit{main}() \\
5: & \quad z := 5 \\
6: & \quad w := \textit{divByX}(z)
\end{array}
$$

We can avoid the error by using a more optimistic assumption $L_a = L_r = NZ$. But then we get a problem with the following program:

$$
\begin{array}{ll}
1: & \text{procedure } \textit{double}(x : int) : int \\
2: & \quad y := 2 * x \\
3: & \quad \text{return } y \\
4: & \text{procedure } \textit{main}() \\
5: & \quad z := 0 \\
6: & \quad w := \textit{double}(z)
\end{array}
$$

1.2 Local vs. global variables

The above analysis assumes we have only local variables. If we have global variables, we must make conservative assumptions about them too. Assume globals should always be described by some lattice value L_g at procedure boundaries. We can extend the flow functions as follows:

$$
\begin{array}{ll}
f[\![x := g(y)]\!](\sigma) & = [x \mapsto L_r][z \mapsto L_g \mid z \in \textbf{Globals}]\sigma \\
& \quad \text{where } \sigma(y) \sqsubseteq L_a \wedge \forall z \in \textbf{Globals} : \sigma(z) \sqsubseteq L_g \\
f[\![\text{return } x]\!](\sigma) & = \sigma \\
& \quad \text{where } \sigma(x) \sqsubseteq L_r \wedge \forall z \in \textbf{Globals} : \sigma(z) \sqsubseteq L_g
\end{array}
$$

1.3 Annotations

An alternative approach is using annotations. This allows us to choose different argument and result assumptions for different procedures. Flow functions might look like:

$$\begin{array}{lll} f[\![x := g(y)]\!](\sigma) & = [x \mapsto \mathit{annot}[\![g]\!].r]\sigma & \text{where } \sigma(y) \sqsubseteq \mathit{annot}[\![g]\!].a \\ f[\![\text{return } x]\!](\sigma) & = \sigma & \text{where } \sigma(x) \sqsubseteq \mathit{annot}[\![g]\!].r \end{array}$$

Now we can verify that both of the above programs are safe. But some programs remain difficult:

```
1 : procedure double(x : int @⊤) : int @⊤
2 :     y := 2 * x
3 :     return y
4 : procedure main()
5 :     z := 5
6 :     w := double(z)
7 :     z := 10/w
```

Annotations can be extended in a natural way to handle global variables.

1.4 Interprocedural Control Flow Graph

An approach that avoids the burden of annotations, and can capture what a procedure actually does as used in a particular program, is building a control flow graph for the entire program, rather than just one procedure. To make this work, we handle call and return instructions specially as follows:

- We add additional edges to the control flow graph. For every call to function g, we add an edge from the call site to the first instruction of g, and from every return statement of g to the instruction following that call.
- When analyzing the first statement of a procedure, we generally gather analysis information from each predecessor as usual. However, we take out all dataflow information related to local variables in the callers. Furthermore, we add dataflow information for parameters in the callee, initializing their dataflow values according to the actual arguments passed in at each call site.

- When analyzing an instruction immediately after a call, we get dataflow information about local variables from the previous statement. Information about global variables is taken from the return sites of the function that was called. Information about the variable that the result of the function call was assigned to comes from the dataflow information about the returned value.

Now the example described above can be successfully analyzed. However, other programs still cause problems:

$1:$ procedure *double*$(x : int\ @\top) : int\ @\top$
$2:$ $\quad y := 2 * x$
$3:$ $\quad$ return y
$4:$ procedure *main*$()$
$5:$ $\quad z := 5$
$6:$ $\quad w :=$ *double*(z)
$7:$ $\quad z := 10/w$
$8:$ $\quad z := 0$
$9:$ $\quad w :=$ *double*(z)

1.5 Context Sensitive Analysis

Context-sensitive analysis analyzes a function either multiple times, or parametrically, so that the analysis results returned to different call sites reflect the different analysis results passed in at those call sites.

We can get context sensitivity just by duplicating all callees. But this works only for non-recursive programs.

A simple solution is to build a summary of each function, mapping dataflow input information to dataflow output information. We will analyze each function once for each *context*, where a context is an abstraction for a set of calls to that function. At a minimum, each context must track the input dataflow information to the function.

Let's look at how this approach allows the program given above to be proven safe by zero analysis.

[Example given in class]

Things become more challenging in the presence of recursive functions, or more generally mutual recursion. Let us consider context-sensitive interprocedural constant propagation analysis of the factorial function called by main. We are not focused on the intraprocedural part of the analysis so we will just show the function in the form of Java or C source code:

```
int fact(int x) {
    if (x == 1)
        return 1;
    else
        return x * fact(x-1);
}
void main() {
    int y = fact(2);
    int z = fact(3);
    int w = fact(getInputFromUser());
}
```

We can analyze the first two calls to fact using the following algorithm:

```
begin()
    // initial context is main() with argument assumptions
    context = get the initial program context
    analyze(context)

analyze(context)
    newResults = intraprocedural(context)
    resultsMap.put(context, newResults)
    return newResults

// called by intraprocedural analysis of "context"
analyzeCall(context, callInfo) : AnalysisResult
    calleeContext = computeCalleeContext(context, callinfo)
    results = getResultsFor(calleeContext)
    return results

getResultsFor(context)
    results = resultsMap.get(context)
    if (results != bottom)
        return results;
    else
        return analyze(context)

computeCalleeContext(callingContext, callinfo)
    // calling context is just the input information
    return callinfo.inputInfo
```

The resultsMap and the function getResultsFor() acts as a cache for anal-

ysis results, so that when fib(3) invokes fib(2), the results from the prior call fib(2) can be reused.

For the third call to fib, the argument is determined at runtime and so constant propagation uses $\top$ for the calling context. In this case the recursive call to fib() also has $\top$ as the calling context. But we cannot look up the result in the cache yet as analysis of fib() with $\top$ has not completed. Thus the algorithm above will attempt to analyze fib() with $\top$ again, and it will therefore not terminate.

We can solve the problem by applying the same idea as in intraprocedural analysis. The recursive call is a kind of a loop. We can make the initial assumption that the result of the recursive call is $\bot$, which is conceptually equivalent to information coming from the back edge of a loop. When we discover the result is a higher point in the lattice then $\bot$, we reanalyze the calling context (and recursively, all calling contexts that depend on it). The algorithm to do so can be expressed as follows:

```
begin()
    // initial context is main() with argument assumptions
    context = get the initial program context
    analyze(context)
    while context = worklist.remove()
        analyze(context)

analyze(context)
    oldResults = resultsMap.get(context)
    newResults = intraprocedural(context)
    if (newResults != oldResults)
        resultsMap.put(context, newResults)
        for ctx in callingContextsOf(context)
            worklist.add(ctx)
    return newResults

// called by intraprocedural analysis of "context"
analyzeCall(context, callInfo) : AnalysisResult
    calleeContext = computeCalleeContext(context, callinfo)
    results = getResultsFor(calleeContext)
    add context to callingContextsOf(calleeContext)
    return results

getResultsFor(context)
```

```
if (context is currently being analyzed)
    return bottom
results = resultsMap.get(context)
if (results != bottom)
    return results;
else
    return analyze(context)
```

The following example shows that the algorithm generalizes naturally to the case of mutually recursive functions:

```
bar() { if (*) return 2 else return foo() }
foo() { if (*) return 1 else return bar() }

main() { foo(); }
```

The description above considers differentiates calling contexts by the input dataflow information. A historical alternative is to differentiate contexts by their call string: the call site, it's call site, and so forth. In the limit, when considering call strings of arbitrary length, this provides full context sensitivity.

Dataflow analysis results for contexts based on arbitrary-length call strings are as precise as the results for contexts based on separate analysis for each different input dataflow information. The latter strategy can be more efficient, however, because it reuses analysis results when a function is called twice with different call strings but the same input dataflow information.

In practice, both strategies (arbitrary-length call strings vs. input dataflow information) can result in reanalyzing each function so many times that performance becomes unacceptable. Thus multiple contexts must be combined somehow to reduce the number of times each function is analyzed. The call-string approach provides an easy, but naive, way to do this: call strings can be cut off at a certain length. For example, if we have call strings "a b c" and "d e b c" (where c is the most recent call site) with a cutoff of 2, the input dataflow information for these two call strings will be merged and the analysis will be run only once, for the context identified by the common length-two suffix of the strings, "b c". We can illustrate this by redoing the analysis of the fibonacci example. The algorithm is the same as above; however, we use a different implementation of compteCalleeContext that computes the call string suffix and, if it has already been analyzed, merges the incoming dataflow analysis information with what is already there:

```
computeCalleeContext(callingContext, callinfo)
    let oldCallString = callingContext.callString
    let newCallString = suffix(oldCallString ++ callinfo.site,
                               CALL_STRING_CUTOFF)
    let newContext = new Context(newCallString, callinfo.inputInfo)

    // look for a previous analysis with the same call string
    // context identity (and map lookup) is determined
    // by the call string
    if (resultsMap.containsKey(newContext))
        let oldContext = resultsMap.findKey(newContext);
        if (newContext.inputInfo ⋢ oldContext.inputInfo)
                // force reanalysis with joined input information
            resultsMap.removeKey(newContext)
            newContext.inputInfo = newContext.inputInfo
                                 ⊔ oldContext.inputInfo

    return newContext
```

Although this strategy reduces the overall number of analyses, it does so in a relatively blind way. If a function is called many times but we only want to analyze it a few times, we want to group the calls into analyzis contexts so that their input information is similar. Call string context is a hueristic way of doing this that sometimes works well. But it can be wasteful: if two different call strings of a given length happen to have exactly the same input analysis information, we will do an unnecessary extra analysis, whereas it would have been better to spend that extra analysis to differentiate calls with longer call strings that have different analysis information.

Given a limited analysis budget, it is smarter to use hueristics that are directly based on input information. Unfortunately these hueristics are harder to design, but they have the potential to do much better than a call-string based approach. We will look at some examples from the literature to illustrate this later in the course.