Lecture Notes: Interprocedural Analysis

17-355/17-665/17-819: Program Analysis (Spring 2020)
Claire Le Goues[*]
clegoues@cs.cmu.edu

Consider an extension of WHILE3ADDR that includes functions. We thus add a new syntactic category F (for functions), and two new instruction forms (function call and return), as follows:

$$\begin{array}{lll} F & ::= & \text{fun } f(x) \; \{ \; \overline{n : I} \; \} \\ I & ::= & \ldots \mid \text{return } x \mid y := f(x) \end{array}$$

In the notation above, $\overline{n : I}$, the line is shorthand for a list, so that the body of a function is a list of instructions I with line numbers n. We assume in our formalism that all functions take a single integer argument and return an integer result, but this is easy to generalize if we need to. We can also add global variables to this language by tracking a separate set of variables, **Globals**. We assume simple syntactic scoping.

Note that this is not a truly precise syntactic specification. Specifying even just "possibly empty list of arithmetic expressions" properly takes several intermediate syntactic steps; correctly handling scope requires rather significant refinement to the operational semantics. However, providing such precision is more trouble than it's worth for this discussion. Function names are strings. Functions may return either void or a single integer. We leave the problem of type-checking to another class.

We've made our programming language much easier to use, but dataflow analysis has become rather more difficult. Interprocedural analysis concerns analyzing a program with multiple procedures, ideally taking into account the way that information flows among those procedures. We use zero analysis as our running example throughout, unless otherwise indicated.

1 Two Simple Approaches

Default assumptions. Our first approach assumes a default lattice value for all arguments to a function L_a and a default value for procedure results L_r. In some respects, L_a is equivalent to the initial dataflow information we set at the entry to the program when we were only looking intraprocedurally; now we assume it on entry to every procedure. We check the assumptions hold when analyzing a call or return instruction (trivial if $L_a = L_r = \top$). We then 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:

[*]These notes were developed together with Jonathan Aldrich

$$\begin{aligned} f[\![x := g(y)]\!](\sigma) &= \sigma[x \mapsto L_r] && \text{(error if } \sigma(y) \not\sqsubseteq L_a) \\ f[\![\text{return } x]\!](\sigma) &= \sigma && \text{(error if } \sigma(x) \not\sqsubseteq L_r) \end{aligned}$$

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

```
fun $divByX(x) : int$
    $y := 10/x$
    return $y$
fun $main() : void$
    $z := 5$
    $w := divByX(z)$
```

The results are sound, but imprecise. We can avoid the false positive by using a more optimistic assumption $L_a = L_r = NZ$. But then we get a problem with the following program:

```
fun $double(x : int) : int$
    $y := 2 * x$
    return $y$
fun $main() : void$
    $z := 0$
    $w := double(z)$
```

Now what?

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

$$\begin{aligned} f[\![x := g(y)]\!](\sigma) &= \sigma[x \mapsto annot[\![g]\!].r] && \text{(error if } \sigma(y) \not\sqsubseteq annot[\![g]\!].a) \\ f[\![\text{return } x]\!](\sigma) &= \sigma && \text{(error if } \sigma(x) \not\sqsubseteq annot[\![g]\!].r) \end{aligned}$$

Now we can verify that both of the above programs are safe, given the proper annotations. We will see other example analysis approaches that use annotations later in the semester, though historically, programmer buy-in remains a challenge in practice.

Local vs. global variables. If we add 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:

$$\begin{aligned} f[\![x := g(y)]\!](\sigma) &= \sigma[x \mapsto L_r][z \mapsto L_g \mid z \in \mathbf{Globals}] \\ &\quad \text{(error if } \sigma(y) \not\sqsubseteq L_a \vee \forall z \in \mathbf{Globals} : \sigma(z) \not\sqsubseteq L_g) \\ f[\![\text{return } x]\!](\sigma) &= \sigma \\ &\quad \text{(error if } \sigma(x) \not\sqsubseteq L_r \vee \forall z \in \mathbf{Globals} : \sigma(z) \not\sqsubseteq L_g) \end{aligned}$$

The annotation approach can also be extended in a natural way to handle global variables.

2 Interprocedural Control Flow Graphs

An approach that avoids the burden of annotations, and can capture what a procedure actually does as used in a particular program, is to build a control flow graph for the entire program, rather than just a single 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 examples described above can be successfully analyzed. However, other programs still cause problems:

```
fun $double(x : int) : int$
    $y := 2 * x$
    return $y$
fun $main()$
    $z := 5$
    $w := double(z)$
    $z := 10/w$
    $z := 0$
    $w := double(z)$
```

What's the issue here?

3 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 could get context sensitivity just by duplicating (or inlining) all callees, but this only works 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 will be 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 a 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` within `main` straightforwardly, and in fact we can even cache the results of analyzing `fact(2)` for reuse when analyzing the recursive call inside `fact(3)`.

For the third call to `fact`, the argument is determined at runtime, and so constant propagation uses $\top$ for the calling context. In this case, the recursive call to `fact()` also has $\top$ as the calling context. But we cannot look up the result in the cache yet as analysis of `fact()` with $\top$ has not completed. A naive approach would attempt to analyze `fact()` with $\top$ again, and would 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 make the initial assumption that the result of the recursive call is $\bot$, 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:

```
type $Context$
    val $fn$ : $Function$                                   ▷ the function being called
    val $input$ : $\sigma$                                  ▷ input for this set of calls

type $Summary$                                              ▷ the input/output summary for a context
    val $input$ : $\sigma$
    val $output$ : $\sigma$

val $worklist$ : $Set[Context]$         ▷ contexts we must revisit due to updated analysis information
val $analyzing$ : $Stack[Context]$                          ▷ the contexts we are currently analyzing
val $results$ : $Map[Context, Summary]$                     ▷ the analysis results
val $callers$ : $Map[Context, Set[Context]]$                ▷ the call graph - used for change propagation
function ANALYZEPROGRAM                                     ▷ starting point for interprocedural analysis
    $worklist \leftarrow \{Context(\texttt{main}, \top)\}$
    $results[Context(\texttt{main}, \top)].input \leftarrow \top$
    while NOTEMPTY($worklist$) do
        $ctx \leftarrow$ REMOVE($worklist$)
        ANALYZE($ctx, results[ctx].input$)
    end while
end function

function ANALYZE($ctx, \sigma_i$)
    $\sigma_o \leftarrow results[ctx].output$
    PUSH($analyzing, ctx$)
    $\sigma'_o \leftarrow$ INTRAPROCEDURAL$(ctx, \sigma_i)$
    POP$(analyzing)$
    **if** $\sigma'_o \not\sqsubseteq \sigma_o$ **then**
        $results[ctx] \leftarrow Summary(\sigma_i, \sigma_o \sqcup \sigma'_o)$
        **for** $c \in callers[ctx]$ **do**
            ADD$(worklist, c)$
        **end for**
    **end if**
    **return** $\sigma'_o$
**end function**

**function** FLOW$([\![n\colon x := f(y)]\!], ctx, \sigma_i)$          $\triangleright$ called by intraprocedural analysis
    $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_i(y)]$          $\triangleright$ map $f$'s formal parameter to info on actual from $\sigma_i$
    $calleeCtx \leftarrow$ GETCTX$(f, ctx, n, \sigma_{in})$
    $\sigma_o \leftarrow$ RESULTSFOR$(calleeCtx, \sigma_{in})$
    ADD$(callers[calleeCtx], ctx)$
    **return** $\sigma_i[x \mapsto \sigma_o[result]]$          $\triangleright$ update dataflow with the function's result
**end function**

**function** RESULTSFOR$(ctx, \sigma_i)$
    $\sigma \leftarrow results[ctx].output$
    **if** $\sigma \neq \bot \wedge \sigma_i \sqsubseteq results[ctx].input$ **then**
        **return** $\sigma$          $\triangleright$ existing results are good
    **end if**
    $results[ctx].input \leftarrow results[ctx].input \sqcup \sigma_i$          $\triangleright$ keep track of possibly more general input
    **if** $ctx \in analyzing$ **then**
        **return** $\bot$          $\triangleright$ initially optimistic assumption for recursive calls
    **else**
        **return** ANALYZE$(ctx, results[ctx].input)$
    **end if**
**end function**

**function** GETCTX$(f, callingCtx, n, \sigma_i)$
    **return** $Context(f, \sigma_i)$          $\triangleright$ constructs a new $Context$ with $f$ and $\sigma_i$
**end function**
```

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(); }
```

4 Precision and Termination

Precision. A notable part of the algorithm above is that if we are currently analyzing a context and are asked to analyze it again, we return $\bot$ as the result of the analysis. This has similar benefits to using $\bot$ for initial dataflow values on the back edges of loops: starting with the most optimistic assumptions about code we havent finished analyzing allows us to reach the best possible fixed point. The following example program illustrates a function where the result of analysis will be better if we assume $\perp$ for recursive calls to the same context, vs. for example if we assumed $\top$:

```
int iterativeIdentity(x : int, y : int)
    if x <= 0
        return y
    else
        iterativeIdentity(x-1, y)

void main(z)
    w = iterativeIdentity(z, 5)
```

Termination. When will the algorithm above terminate? *Analyze* is called only when (1) a context has not been analyzed yet, or when (2) it has just been taken off the worklist. So it is called once per reachable context, plus once for every time a reachable context is added to the worklist.

We can bound the total number of worklist additions by (C) the number of reachable contexts, times (H) the height of the lattice (we dont add to the worklist unless results for some context changed, i.e. went up in the lattice relative to an initial assumption of $\perp$ or relative to the last analysis result), times (N) the number of callers of that reachable context. C*N is just the number of edges (E) in the inter-context call graph, so we can see that we will do intraprocedural analysis O(E*H) times.

Thus the algorithm will terminate as long as the lattice is of finite height and there are a finite number of reachable contexts. Note, however, that for some lattices, notably including constant propagation, there are an unbounded number of lattice elements and thus an unbounded number of contexts. If more than a finite number are not reachable, the algorithm will not terminate. So for lattices with an unbounded number of elements, we need to adjust the context-sensitivity approach above to limit the number of contexts that are analyzed.

5 Approaches to Limiting Context-Sensitivity

No context-sensitivity. One approach to limiting the number of contexts is to allow only one for each function. This is equivalent to the interprocedural control flow graph approach described above. We can recast this approach as a variant of the generic interprocedural analysis algorithm by replacing the *Context* type to track only the function being called, and then having the GETCTX method always return the same context:

```
type $Context$
    val $fn : Function$

function GETCTX($f, callingCtx, n, \sigma_i$)
    return $Context(f)$
end function
```

Note that in this approach the same calling context might be used for several different input dataflow information σ_i, one for each call to GETCTX. This is handled correctly by RESULTSFOR, which updates the input information in the *Summary* for that context so that it generalizes all the input to the function seen so far.

Limited contexts. Another approach is to create contexts as in the original algorithm, but once a certain number of contexts have been created for a given function, merge all subsequent calls into a single context. Of course, this means the algorithm will lose precision beyond this bounds. But, if most functions have fewer contexts that are actually used, this can be a good strategy for analyzing most of the program in a context-sensitive way while avoiding performance problems for the minority of functions that are called from many different contexts.

Can you implement a GETCTX *function that represents this strategy?*

Call strings. Another context sensitivity strategy is to differentiate contexts by a *call string*: the call site, its call site, and so forth. In the limit, when considering call strings of arbitrary length, this provides full context sensitivity (but is not guaranteed to terminate for arbitrary recursive functions). Dataflow analysis results for contexts based on arbitrarylength 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 an unacceptable number of times. Multiple contexts must be combined somehow. The call-string approach provides an easy, but naive, way to do this: call strings can simply 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 factorial example. The algorithm is the same as above; however, we use a different implementation of GETCTX that computes the call string suffix:

```
type $Context$
    val $fn : Function$
    val $string : List[Int]$

function GETCTX($f, callingCtx, n, \sigma_i$)
    $newStr$ ←SUFFIX($callingCtx.string$ ++ $n$, CALL_STRING_CUTOFF)
    return $Context(f, newStr)$
end function
```

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 analysis contexts so that their input information is similar. Call string context is a heuristic 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 usually best to use heuristics that are directly based on input information. Unfortunately these heuristics 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.