Lecture Notes: Advanced Interprocedural Analysis: Pointer Analysis and Object-Oriented Call Graph Construction

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

We have successfully extended our interprocedural dataflow analysis framework to a small functional programming language, which required us to reason explicitly about which functions might be called, where. This provides insight into similar problems in other programming paradigms, namely *dynamic dispatch*. Precisely addressing dynamic dispatch relies on techniques for *pointer analysis*, which establishes which pointers can point to which locations. Analyses that address real programming languages (whether they use dynamic dispatch or not) must address pointers, because ignoring them dramatically impacts analysis precision. Thus, in the interest of adapting our framework to real languages, we turn our attention to these issues.

1 Pointer Analysis

Pointers are variables whose value refers to another value elsewhere in memory, by storing the address of that stored value. To illustrate why they matter in analyzing real programs, consider constant-propagation analysis of the following program:

$$\begin{array}{ll} 1: & z := 1 \\ 2: & p := \&z \\ 3: & *p := 2 \\ 4: & \text{print } z \end{array}$$

To analyze this program correctly we must be aware that at instruction 3, p points to z. If this information is available we can use it in a flow function as follows:

$$f_{CP}[\![*p := y]\!](\sigma) \quad = \sigma[z \mapsto \sigma(y) \mid z \in \textit{must-point-to}(p)]$$

When we know exactly what a variable x points to, we have *must-point-to* information, and we can perform a *strong update* of the target variable z, because we know with confidence that assigning to $*p$ assigns to z. A technicality in the rule is quantifying over all z such that p must point to z. How is this possible? It is not possible in C or Java; however, in a language with pass-by-reference, for example C++, it is possible that two names for the same location are in scope.

Of course, it is also possible to be uncertain to which of several distinct locations p points:

*These notes were developed together with Jonathan Aldrich

$$\begin{array}{ll} 1: & z := 1 \\ 2: & \textbf{if } (cond)\ p := \&y \textbf{ else } p := \&z \\ 3: & *p := 2 \\ 4: & \textbf{print } z \end{array}$$

Now constant propagation analysis must conservatively assume that z could hold either 1 or 2. We can represent this with a flow function that uses may-point-to information:

$$f_{CP}[\![*p := y]\!](\sigma) \quad = \sigma[z \mapsto \sigma(z) \sqcup \sigma(y) \mid z \in \textit{may-point-to}(p)]$$

1.1 Andersen's Points-To Analysis

Two common kinds of pointer analysis are *alias analysis* and *points-to* analysis. Alias analysis computes sets S holding pairs of variables (p, q), where p and q may (or must) point to the same location. Points-to analysis computes the set *points-to*(p), for each pointer variable p, where the set contains a variable x if p may (or must) point to the location of the variable x. We will focus primarily on points-to analysis, beginning with a simple but useful approach originally proposed by Andersen.[1]

Our initial setting will be C programs. We are interested in analyzing instructions that are relevant to pointers in the program. Ignoring for the moment memory allocation and arrays, we can decompose all pointer operations in C into four types:

$$\begin{array}{lcll} I & ::= & \ldots & \\ & | & p := \&x & \text{taking the address of a variable} \\ & | & p := q & \text{copying a pointer from one variable to another} \\ & | & *p := q & \text{assigning through a pointer} \\ & | & p := *q & \text{dereferencing a pointer} \end{array}$$

Andersen's points-to analysis is a context-insensitive interprocedural analysis. It is also a *flow-insensitive analysis*, that is an analysis that does not consider program statement order. Context- and flow-insensitivity improve analysis performance, as precise pointer analysis can be notoriously expensive.

We will formulate Andersen's analysis by generating set constraints which can later be processed by a set constraint solver, much like we did for CFA. Because the analysis is flow-insensitive, we do not care what order the instructions in the program come in; we simply generate a set of constraints and solve them. Constraint generation for each statement works by these rules:

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p}\ \textit{address-of}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q}\ \textit{copy}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q}\ \textit{assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q}\ \textit{dereference}$$

[1]PhD thesis: "Program Analysis and Specialization for the C Programming Language."

The first rule states that a constant location l_x, representation the address of x, is in the set of location pointed to by p. The second rule states that the set of locations pointed to by p must be a superset of those pointed to by q. The last two rules state the same, but take into account that one or the other pointer is dereferenced. Note that if Andersen's algorithm says that the set p points to only one location l_z, we have *must-point-to* information, whereas if the set p contains more than one location, we have only *may-point-to* information.

A number of specialized set constraint solvers exist, and constraints in the form above can be translated into input for them.[2] We will treat constraint-solving abstractly using the following constraint propagation rules:

$$\frac{p \supseteq q \quad l_x \in q}{l_x \in p}\ \textit{copy}$$

$$\frac{*p \supseteq q \quad l_r \in p \quad l_x \in q}{l_x \in r}\ \textit{assign}$$

$$\frac{p \supseteq *q \quad l_r \in q \quad l_x \in r}{l_x \in p}\ \textit{dereference}$$

We can now apply Andersen's points-to analysis to the programs above. We can also apply it to programs with dynamic memory allocation, such as:

$$\begin{array}{rl}
1: & q := \textit{malloc}() \\
2: & p := \textit{malloc}() \\
3: & p := q \\
4: & r := \&p \\
5: & s := \textit{malloc}() \\
6: & *r := s \\
7: & t := \&s \\
8: & u := *t
\end{array}$$

The analysis is run the same way, but we treat the memory cell allocated at each *malloc* or *new* statement as an abstract location labeled by the location n of the allocation point:

$$\frac{}{[\![n\colon p := \textit{malloc}()]\!] \hookrightarrow l_n \in p}\ \textit{malloc}$$

We must be careful because a *malloc* statement can be executed more than once, and each time it executes, a new memory cell is allocated. Unless we have some other means of proving that the malloc executes only once, we must assume that if some variable p only points to one abstract malloc'd location l_n, that is still may-alias information (i.e. p points to only one of the many actual cells allocated at the given program location) and not must-alias information.

Efficiency. Analyzing the efficiency of Andersen's algorithm, we can see that all constraints can be generated in a linear $O(n)$ pass over the program. The solution size is $O(n^2)$, because each of the $O(n)$ variables defined in the program could potentially point to $O(n)$ other variables.

[2]Note that the dereference operation (the $*$ in $*p \supseteq q$) is not standard, but can be encoded,

We can derive the execution time as follows:[3] There are $O(n)$ flow constraints generated of the form $p \supseteq q$, $*p \supseteq q$, or $p \supseteq *q$. How many times could a constraint propagation rule fire for each flow constraint? For a $p \supseteq q$ constraint, the rule may fire at most $O(n)$ times, because there are at most $O(n)$ premises of the proper form $l_x \in p$. However, a constraint of the form $p \supseteq *q$ could cause $O(n^2)$ rule firings, because there are $O(n)$ premises each of the form $l_x \in p$ and $l_r \in q$. With $O(n)$ constraints of the form $p \supseteq *q$ and $O(n^2)$ firings for each, we have $O(n^3)$ constraint firings overall. A similar analysis applies for $*p \supseteq q$ constraints. McAllester's theorem states that the analysis with $O(n^3)$ rule firings can be implemented in $O(n^3)$ time. Thus we have derived that Andersen's algorithm is cubic in the size of the program, in the worst case.

Interestingly, Andersen's algorithm can be executed in $O(n^2)$ time for *k-sparse* programs.[4] The *k-sparse* assumption requires that at most k statements dereference each variable, and that the flow graph is sparse. The publication showing this result also showed that typical Java programs are *k-sparse*, and that Andersen's algorithm scales quadratically in practice.

1.2 Field Sensitivity

What happens when we have a pointer to a struct in C, or an object in an object-oriented language? In this case, we would like the pointer analysis to tell us what each field in the struct or object points to. A simple solution is to be *field-insensitive*, treating all fields in a struct as equivalent. Thus if p points to a struct with two fields f and g, and we assign:

$$\begin{array}{rl} 1: & p.f := \&x \\ 2: & p.g := \&y \end{array}$$

A field-insensitive analysis would tell us (imprecisely) that $p.f$ could point to y. We can modify the rules above by treating any field dereference or field assignment to $p.f$ as a pointer dereference $*p$. Essentially, you can think of this as just considering all fields to be named $*$.

To be more precise, we can instead track the contents each field of each abstract location separately. In the discussion below, we assume a Java-like setting, in which all objects are allocated on the heap and where we cannot take the address of a field. A slightly more complicated variant of this scheme works in C-like languages.

We will use the *malloc* and *copy* rules unchanged from above.[5] We drop the *assign* and *dereference* rules, and replace them with:

$$\frac{}{[\![p := q.f]\!] \hookrightarrow p \supseteq q.f} \textit{ field-read}$$

$$\frac{}{[\![p.f := q]\!] \hookrightarrow p.f \supseteq q} \textit{ field-assign}$$

Now assume that objects (e.g. in Java) are represented by abstract locations l. We will have two forms of basic facts. The first is the same as before: $l_n \in p$, where l_n is an object allocated in a **new** statement at line n. The second basic fact is $l_n \in l_m.f$, which states that the field f of the object represented by l_m may point to an object represented by l_n.

[3]David A. McAllester. 1999. On the Complexity Analysis of Static Analyses. In Proceedings of the 6th International Symposium on Static Analysis (SAS 99): 312329.

[4]Manu Sridharan and Stephen J. Fink. 2009. The Complexity of Andersens Analysis in Practice. In Proceedings of the 16th International Symposium on Static Analysis (SAS 09): 205221.

[5]In Java, the **new** expression plays the role of `malloc`

We can now process field constraints with the following rules:

$$\frac{p \supseteq q.f \quad l_q \in q \quad l_f \in l_q.f}{l_f \in p} \textit{ field-read}$$

$$\frac{p.f \supseteq q \quad l_p \in p \quad l_q \in q}{l_q \in l_p.f} \textit{ field-assign}$$

If we run this analysis on the code above, we find that it can distinguish that $p.f$ points to x and $p.g$ points to y.

1.3 Steensgaard's Points-To Analysis

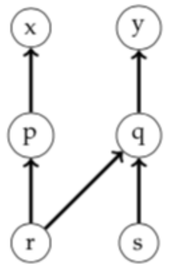
For very large programs, a quadratic-in-practice algorithm is too inefficient. Steensgaard proposed an pointer analysis algorithm that operates in near-linear time, supporting essentially unlimited practical scalability.

The first challenge in designing a near-linear time points-to analysis is to represent the results in linear space. This is nontrivial because over the course of program execution, any given pointer p could potentially point to the location of any other variable or pointer q. Representing all of these pointers explicitly will inherently take $O(n^2)$ space.

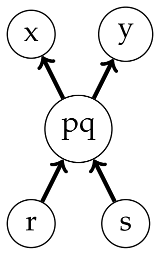
The solution Steensgaard found is based on using constant space for each variable in the program. His analysis associates each variable p with an abstract location named after the variable. Then, it tracks a single points-to relation between that abstract location p and another one q, to which it may point. Now, it is possible that in some real program p may point to both q and some other variable r. In this situation, Steensgaard's algorithm *unifies* the abstract locations for q and r, creating a single abstract location representing both of them. Now we can track the fact that p may point to either variable using a single points-to relationship.

For example, consider the program to the left, and the graph that Andersen's points-to analysis would produce (right):

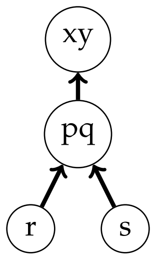
$$\begin{array}{ll} 1: & p := \&x \\ 2: & r := \&p \\ 3: & q := \&y \\ 4: & s := \&q \\ 5: & r := s \end{array}$$



But in Steensgaard's setting, when we discover that r could point both to q and to p, we must merge q and p into a single node:



Notice that we have lost precision: by merging the nodes for p and q our graph now implies that s could point to p, which is not the case in the actual program. But we are not done. Now pq has two outgoing arrows, so we must merge nodes x and y. The final graph produced by Steensgaard's algorithm is therefore:



We study Steensgaard's analysis more precisely by specifying a simplified version that ignores function pointers:

$$\frac{}{[\![p := q]\!] \hookrightarrow \mathit{join}(*p, *q)}\ \mathit{copy}$$

$$\frac{}{[\![p := \&x]\!] \hookrightarrow \mathit{join}(*p, x)}\ \mathit{address\text{-}of}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow \mathit{join}(*p, **q)}\ \mathit{dereference}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow \mathit{join}(**p, *q)}\ \mathit{assign}$$

With each abstract location p, we associate the abstract location that p points to, denoted $*p$. Abstract locations are implemented as a union-find[6] data structure so that we can merge two abstract locations efficiently. In the rules above, we implicitly invoke *find* on an abstract location before calling *join* on it, or before looking up the location it points to.

The *join* operation essentially implements a union operation on the abstract locations. However, since we are tracking what each abstract location points to, we must update this information also. The algorithm to do so is as follows:

```
join($\ell_1$, $\ell_2$)
    if (find($\ell_1$) == find($\ell_2$))
        return
    $n_1 \leftarrow *\ell_1$
    $n_2 \leftarrow *\ell_2$
    union($\ell_1$, $\ell_2$)
    join($n_1$, $n_2$)
```

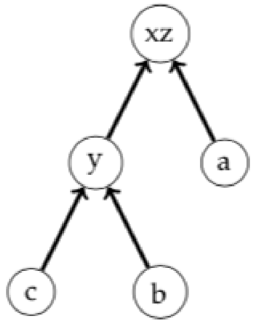
[6]See any algorithms textbook

Once again, we implicitly invoke *find* on an abstract location before comparing it for equality, looking up the abstract location it points to, or calling *join* recursively.

As an optimization, Steensgaard does not perform the join if the right hand side is not a pointer. For example, if we have an assignment $[\![p := q]\!]$ and q has not been assigned any pointer value so far in the analysis, we ignore the assignment. If later we find that q may hold a pointer, we must revisit the assignment to get a sound result.

Steensgaard illustrated his algorithm using the following program, and the graph the algorithm produces:

```
1 :  a := &x
2 :  b := &y
3 :  if p then
4 :     y := &z
5 :  else
6 :     y := &x
7 :  c := &y
```



Efficiency. Rayside illustrates how Andersen must sometimes do more work than Steensgaard:

```
1 :  q := &x
2 :  q := &y
3 :  p := q
4 :  q := &z
```

After processing the first three statements, Steensgaard's algorithm will have unified variables x and y, with p and q both pointing to the unified node. Andersen's algorithm will have both p and q pointing to both x and y. When the fourth statement is processed, Steensgaard's algorithm does only a constant amount of work, merging z in with the already-merged xy. On the other hand, Andersen's algorithm must not just create a points-to relation from q to z, but must also propagate that relationship to p. It is this additional propagation step that results in the significant performance difference between these algorithms.[7]

Analyzing Steensgaard's pointer analysis for efficiency, we observe that each of n statements in the program is processed once. The processing is linear, except for *find* operations on the union-find data structure (which may take amortized time $O(\alpha(n))$ each) and the *join* operations. We note that in the *join* algorithm, the short-circuit test will fail at most $O(n)$ times—at most once for each variable in the program. Each time the short-circuit fails, two abstract locations are unified, at cost $O(\alpha(n))$. The unification assures the short-circuit will not fail again for one of these two variables. Because we have at most $O(n)$ operations and the amortized cost of each operation is at most $O(\alpha(n))$, the overall running time of the algorithm is near linear: $O(n * \alpha(n))$. Space consumption is linear, as no space is used beyond that used to represent abstract locations for all the variables in the program text.

[7]For fun, try adding a new statement $r := p$ after statement 3. Then z has to be propagated to the points-to sets of both p and r. In general, the number of propagations can be linear in the number of copies and the number of address-of operators, which makes it quadratic overall even for programs in the simple form above.

Based on this asymptotic efficiency, Steensgaard's algorithm was run on a 1 million line program (Microsoft Word) in 1996; this was an order of magnitude greater scalability than other pointer analyses known at the time.

Steensgaard's pointer analysis is field-insensitive; making it field-sensitive would mean that it is no longer linear.

2 Dynamic dispatch

Dynamic dispatch is the process of selecting which implementation of a method or function should be called at runtime; it is a defining characteristic object-oriented programming languages and systems, but is not limited to them (e.g., calling through function pointers in C). To construct a precise call graph in such languages, an analysis must determine the type of the receiver object is at each call site. Flow analysis techniques similar to points-to analysis can be used to compute this information, but using an interprocedural flow analysis off the shelf requires a call graph, which is exactly what we are trying to construct. Therefore, object-oriented call graph construction algorithms must simultaneously build a call graph and compute dataflow information describing the types of the objects to which each variable could point.

2.1 Simple approaches

Before examining a full-fledged dataflow analysis-based call graph construction algorithm, we will consider two simpler approaches that do not require flow analysis. These approaches have the side benefit of being very efficient, and so are used in settings such as JIT compilers where analysis time is scarce.

The simplest approach, *class hierarchy analysis*, uses the type of a variable, together with the class hierarchy, to determine what types of object the variable could point to. Unsurprisingly, this is very imprecise, but can be computed very efficiently in $O(n * t)$ time, because it visits n call sites and at each call site traverses a subtree of size t of the class hierarchy.

An improvement to class hierarchy analysis is *rapid type analysis*, which eliminates from the hierarchy classes that are never instantiated. The analysis iteratively builds a set of instantiated types, method names invoked, and concrete methods called. Initially, it assumes that `main` is the only concrete method that is called, and that no objects are instantiated. It then analyzes concrete methods known to be called, one by one. When a method name is invoked, it is added to the list, and all concrete methods with that name defined within (or inherited by) types known to be instantiated are added to the called list. When an object is instantiated, its type is added to the list of instantiated types, and all its concrete methods that have a method name that is invoked are added to the called list. This proceeds iteratively until a fixed point is reached, at which point the analysis knows all of the object types that may actually be created at run time.

Rapid type analysis can be considerably more precise than class hierarchy analysis in programs that use libraries that define many types, only a few of which are used by the program. It remains extremely efficient, because it only needs to traverse the program once (in $O(n)$ time) and then build the call graph by visiting each of n call sites and considering a subtree of size t of the class hierarchy, for a total of $O(n * t)$ time.

2.2 0-CFA Style Object-Oriented Call Graph Construction

Object-oriented call graphs can also be constructed using a pointer analysis such as Andersen's algorithm, either context-insensitive or context-sensitive. The context-sensitive versions are called k-CFA by analogy with control-flow analysis for functional programs. The context-insensitive version is called 0-CFA for the same reason. Essentially, the analysis proceeds as in Andersen's algorithm, but the call graph is built up incrementally as the analysis discovers the types of the objects to which each variable in the program can point.

Even 0-CFA analysis can be considerably more precise than Rapid Type Analysis. For example, in the program below, RTA would assume that any implementation of foo() could be invoked at any program location, but 0-CFA can distinguish the two call sites:

```
class A { A foo(A x) { return x; } }
class B extends A { A foo(A x) { return new D(); } }
class D extends A { A foo(A x) { return new A(); } }
class C extends A { A foo(A x) { return this; } }

// in main()
A x = new A();
while (...)
    x = x.foo(new B()); // may call A.foo, B.foo, or D.foo
A y = new C();
y.foo(x);               // only calls C.foo
```