Lecture Notes: Pointer Analysis

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

Lecture 9

1 Motivation for Pointer Analysis

In programs with pointers, program analysis can become more challenging. Consider constant-propagation analysis of the following program:

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

In order 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:

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

When we know exactly what a variable x points to, we say that 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 that we are uncertain to which of several distinct locations p points. For example:

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

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:

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

2 Andersen's Points-To Analysis

Two common kinds of pointer analysis are alias analysis and points-to analysis. Alias analysis computes a set S holding pairs of variables (p, q), where p and q may (or must) point to the same location. On the other hand, points-to analysis, as described above, computes a relation $\textit{points-to}(p, x)$, where p may (or must) point to the location of the variable x. We will focus our study in this lecture on points-to analysis, and will begin with a simple but useful approach originally proposed by Andersen.

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 into four instruction types: taking the address of a variable, copying a pointer from one variable to another, assigning through a pointer, and dereferencing a pointer:

$$
\begin{array}{lll}
I & ::= & \ldots \\
 & | & p := \&x \\
 & | & p := q \\
 & | & *p := q \\
 & | & p := *q
\end{array}
$$

Andersen's points-to analysis is a context-insensitive interprocedural analysis. It is also a *flow-insensitive analysis*, that is an analysis that (unlike dataflow analysis) does not take into consideration the order of program statements. Context- and flow-insensitivity are used to improve the performance of the analysis, as precise pointer analysis can be notoriously expensive in practice.

We will formulate Andersen's analysis by generating set constraints which can later be processed by a set constraint solver using a number of technologies. Constraint generation for each statement works as given in the following set of rules. 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.

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

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

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

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

The constraints generated are all set constraints. 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.

A number of specialized set constraint solvers exist and constraints in the form above can be translated into the input for these. The dereference operation (the $*$ in $*p \supseteq q$) is not standard in set constraints, but it can be encoded—see Fähndrich's Ph.D. thesis for an example of how to encode Andersen's points-to analysis for the BANE constraint solving engine. 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 program above. Note that in this example 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.

We can also apply Andersen's analysis to programs with dynamic memory allocation, such as:

$$\begin{array}{ll} 1: & q := \textit{malloc}_1() \\ 2: & p := \textit{malloc}_2() \\ 3: & p := q \\ 4: & r := \&p \\ 5: & s := \textit{malloc}_3() \\ 6: & *r := s \\ 7: & t := \&s \\ 8: & u := *t \end{array}$$

In this example, 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. We can use the rules:

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

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.

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.

We can derive the execution time from a theorem by David McAllester published in SAS'99. 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.

2.1 Field-Sensitive Analysis

The algorithm above works in C-like languages for pointers to single memory cells. However, what about 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:

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

A field-insensitive analysis would tell us (imprecisely) that $p.f$ could point to y.

In order to be more precise, we can track the contents each field of each abstract location separately. In the discussion below, we assume a setting in which we cannot take the address of a field; this assumption is true for Java but not for C. We can define a new kind of constraints for fields:

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

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

Now assume that objects (e.g. in Java) are represented by abstract locations l. We can 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.

3 Steensgaard's Points-To Analysis

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

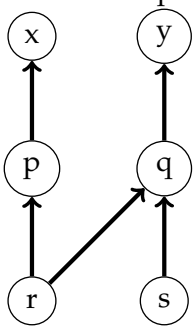
The first challenge in designing a near-linear time points-to analysis is finding a way to represent the results in linear space. This is nontrivial because over the course of program execution, any given pointer p could potentially point to the location of any other variable or pointer q. Representing 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.

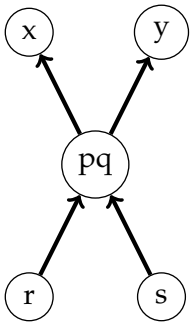
For example, consider the program below:

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

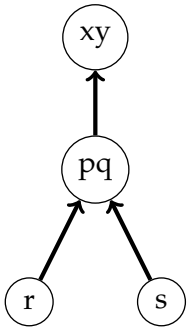
Andersen's points-to analysis would produce the following graph:



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:



Now let us define Steensgaard's analysis more precisely. We will study a simplified version of the analysis that does not consider function pointers. The analysis can be specified as follows:

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

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

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

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

With each abstract location p, we associate the abstract location that p points to, denoted $*p$. Abstract locations are implemented as a union-find[1] 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(e1, e2)
    if (e1 == e2)
        return
    e1next = *e1
    e2next = *e2
```

[1] See any algorithms textbook

```
unify(e1, e2)
join(e1next, e2next)
```

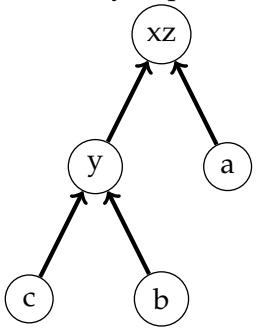
Once again, we implicitly invoke *find* on an abstract location before comparing it for equality, looking up the abstract location it points to, or calling *join* recursively.

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:

$$
\begin{array}{ll}
1: & a := \&x \\
2: & b := \&y \\
3: & \text{if } p \text{ then} \\
4: & \quad y := \&z \\
5: & \text{else} \\
6: & \quad y := \&x \\
7: & c := \&y
\end{array}
$$

His analysis produces the following graph for this program:



Rayside illustrates a situation in which Andersen must do more work than Steensgaard:

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

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. In contrast, 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 node. 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.

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.

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.

4 Adding Context Sensitivity to Andersen's Algorithm

We can define a version of Andersen's points-to algorithm that is context-sensitive. In the following approach, we analyze each function separately for each calling point. The analysis keeps track of the current context, the calling point n of the current procedure. In the constraints, we track separate values for each variable x_n according to the calling context n of the procedure defining it, and we track separate values for each memory location l_n^k according to the calling context n active when that location was allocated at new instruction k. The rules are as follows:

$$\frac{n \vdash p := \mathbf{new}_k\ A}{l_n^k \in p_n}\ \textit{new}$$

$$\frac{n \vdash p := q \quad l_n \in q_n}{l_n \in p_n}\ \textit{copy}$$

$$\frac{n \vdash x.f := y \quad l_x \in x_n \quad l_y \in y_n}{l_y \in l_x.f}\ \textit{field-read}$$

$$\frac{n \vdash x := y.f \quad l_y \in y_n \quad l_z \in l_y.f}{l_z \in x_n}\ \textit{field-assign}$$

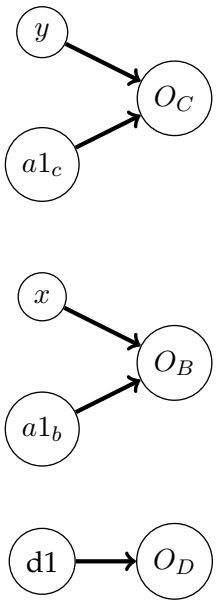
$$\frac{n \vdash f_k(y) \quad l_y \in y_n \quad [\![f(z) = e]\!] \in \textit{Program}}{l_y \in z_k \quad k \vdash e}\ \textit{call}$$

To illustrate this analysis, imagine we have the following code:

```
interface A { void g(); }
class B implements A { void g() { ... } }
class C implements A { void g() { ... } }
class D {
    A f(A a1) { return a1; }
}

// in main()
D d1 = new D();
if (...) {
    A x = d1.f(new B());
        x.g() // which g is called?
else
    A y = d1.f(new C());
        y.g() // which g is called?
```

The analysis produces the following aliasing graph:



In this example, tracking two separate versions of the variable $a1$ is sufficient to distinguish the objects of type B and C as they are passed through method f, meaning that the analysis can accurately track which version of g is called in each program location.

Call-string context sensitivity has its limits, however. Consider the following example, adapted from notes by Ryder:

```
interface X { void g(); }
class Y implements X { void g() { ... } }
class Z implements X { void g() { ... } }
class A {
        X x;
        void setX(X v) { helper(v)^h; }
        void helper(X vh) { x = vh; }
        X getX() { return x; }
}

// in main()
A a1 = new A();    // allocates Oa1
A a2 = new A();    // allocates Oa2
a1.setX(new Y())^Y; // allocates OY
a2.setX(new Z())^Z; // allocates OZ
X x1 = a1.getX();
X x2 = a2.getX();
```

```
x1.g();          // which g() is called?
x2.g();          // which g() is called?
```

If we analyze this example with a 1-CFA style call-string sensitive pointer analysis, we get the following analysis results:

Context	Variable	Location	Notes
•	a1	Oa1	
•	a2	Oa2	
Y	this	Oa1	
Y	v	OY	
h	this	Oa1	
h	vh	OY	
Oa1	x	OY	
Z	this	Oa2	
Z	v	OZ	
h	this	Oa1,Oa2	updated
h	vh	OY, OZ	updated
Oa1	x	OY, OZ	updated
Oa2	x	OY, OZ	
•	x1	OY, OZ	
•	x1	OY, OZ	

Essentially, because of the helper method, one function call's worth of context sensitivity is insufficient to distinguish the calls to setX and helper for the objects Oa1 and Oa2. We could fix this by increasing context sensitivity, e.g. by going to a 2-CFA analysis that tracks call strings of length two. This has a very high cost in practice, however; 2-CFA does not scale well to large object-oriented programs.

A better solution comes from the insight that in the above example, call-strings are really tracking the wrong kind of context. What we need to do is distinguish between Oa1 and Oa2. In other words, the call chain does not matter so much; we want to be sensitive to the receiver object.

An alternative approach based on this idea is called object-sensitive analysis. It uses for the context not the call site, but rather the receiver object. In this case, we index everything not by a calling point n but instead by a receiver object l. The rules are as follows:

$$\frac{l \vdash p := \mathbf{new}_k\ A}{l_l^k \in p_l}\ \textit{new}$$

$$\frac{l \vdash p := q \quad l_l \in q_l}{l_l \in p_l}\ \textit{copy}$$

$$\frac{l \vdash x.f := y \quad l_x \in x_l \quad l_y \in y_l}{l_y \in l_x.f}\ \textit{field-read}$$

$$\frac{l \vdash x := y.f \quad l_y \in y_l \quad l_z \in l_y.f}{l_z \in x_l}\ \textit{field-assign}$$

$$\frac{l \vdash x.f(y) \quad l_x \in x_l \quad l_y \in y_l \quad [\![f(z) = e]\!] \in \textit{Program}}{l_x \in \mathbf{this}_{l_x} \quad l_y \in z_{l_x} \quad l_x \vdash e}\ \textit{call}$$

Now if we reanalyze the example above, we get:

Context	Variable	Location
•	a1	Oa1
•	a2	Oa2
Oa1	v	OY
Oa1	vh	OY
Oa1	x	OY
Oa2	v	OZ
Oa2	vh	OZ
Oa2	x	OZ
•	x1	OY
•	x1	OZ

In practice, object-sensitive analysis appears to be the best approach to context sensitivity in the pointer or call-graph construction analysis of object-oriented programs. Intuitively, it seems that organizing a program around objects makes the objects themselves the most interesting thing to analyze.

The state of the art implementation technique for points-to analysis of object-oriented programs was presented by Bravenboer and Smaragdakis in OOPSLA 2009. Their approach generates declarative Datalog code to represent the input program, and a datalog evaluation engine solves what

are essentially declarative constraints to get the analysis result.

In an more recent POPL 2011 paper analyzing object-sensitivity, Smaragdakis, Bravenboer, and Lhoták demonstrate that it is more effective than call-string sensitivity. They also propose a technique known as type-sensitive analysis which tracks only the type of the receiver (and, for depths $\geqslant 2$, the type of the object that created the reciever, etc.), and show that type-sensitive analysis is nearly as precise as object-sensitive analysis and much more scalable.