Lecture Notes: Pointer Analysis

17-355/17-665: Program Analysis *(Spring 2017)*
Jonathan Aldrich
aldrich@cs.cmu.edu

1 Motivation for Pointer Analysis

In the spirit of extending our understanding of analysis to more realistic languages, consider programs with pointers, or variables whose value refers to another value stored elsewhere in memory by storing the address of that stored value. Pointers are very common in imperative and object-oriented programs, and ignoring them can dramatically impact the precision of other analyses that we have discussed. 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:

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

When we know exactly what a variable x points to, we 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:

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

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 sets S holding pairs of variables (p, q), where p and q may (or must) point to the same location. Points-to analysis, as described above, computes a relation *points-to*(p, x), where 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 (PhD thesis: "Program Analysis and Specialization for the C Programming Language").

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

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

Andersen's points-to analysis is a context-insensitive interprocedural analysis. It is also a *flow-insensitive analysis*, that is an analysis that does not consider program statement order. 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:

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

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

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:

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

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

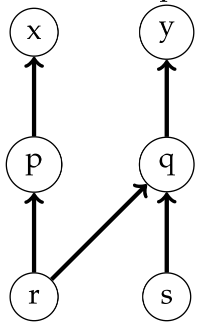
The first challenge in designing a near-linear time points-to analysis is to represent the results in linear space. This is nontrivial because over the course of program execution, any given pointer p could potentially point to the location of any other variable or pointer q. Representing all of these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

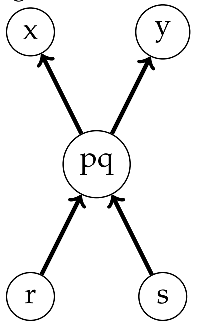
For example, consider the program below:

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

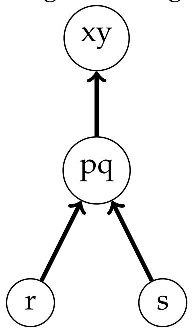
Andersen's points-to analysis would produce the following graph:



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:



To define Steensgaard's analysis more precisely, we will study a simplified version of that ignores function pointers. It can be specified as follows:

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

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

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

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

With each abstract location p, we associate the abstract location that p points to, denoted $*p$. Abstract locations are implemented as a union-find[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
    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.

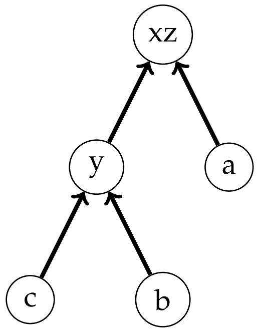
As an optimization, Steensgaard does not perform the join if the right hand side is not a pointer. For example, if we have an assignment $[\![p := q]\!]$ and q has not been assigned any pointer value so far in the analysis, we ignore the assignment. If later we find that q may hold a pointer, we must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

Steensgaard illustrated his algorithm using the following program:

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

His analysis produces the following graph for this program:

[1] See any algorithms textbook



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

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

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.

Acknowledgements

I thank Claire Le Goues for greatly appreciated extensions and refinements to these notes.