Lecture Notes on Alias Analysis

15-411: Compiler Design
André Platzer

Lecture 16
October 19, 2010

1 Introduction

So far we have seen how to implement and compile programs with pointers, but we have not seen how to optimize any expressions involving pointers. For primitive scalar type variables, we can easily see if x and y represent the same memory (or register) location. They represent the same location (and will thus always be guaranteed to hold the same value) if and only if x and y are the same variable. For instance we can optimize by constant propagation

$$\text{x=1; y=2; z=x} \quad \rightsquigarrow \quad \text{x=1; y=2; z=1}$$

We would like to be able to constant propagate also in the presence of pointers:

$$\text{*x=1; *y=2; z=*x} \quad \rightsquigarrow^{???} \quad \text{*x=1; *y=2; z=1} \quad ???$$

$$\text{x}{-}\text{>f=1; y}{-}\text{>f=2; z=x}{-}\text{>f} \quad \rightsquigarrow^{???} \quad \text{x}{-}\text{>f=1; y}{-}\text{>f=2; z=1} \quad ???$$

In the presence of pointers, this reasoning breaks down, however, because two syntactically different pointers p and q may still hold the same memory address, so if we dereference p (e.g., when changing $*p = 5$) then we might have changed the contents $*q$ of q without touching q itself syntactically, just based on the fact that p and q happen to hold the same address at the moment. Names do not denote memory addresses any more, but now they only denote access paths to addresses. Two different access paths, however, may still end up in the same address. This is the most fundamental question for any optimizations called *aliasing problem*.

For more information on alias analysis also see [ALSU06, Ch 12.2-12.5] and [App98, Ch 17.5.].

2 Interprocedurality

Successful alias analysis often has to be interprocedural, because, otherwise, we have to assume that every function call could have changed all memory contents arbitrarily. C0 is nice in that it does not have function pointer calls and no virtual method dispatch. So at least we know directly which function could possibly be called. In object-oriented programming languages like Java, we can have

```
Object o;
...
o = new Component();
...
String s = o.toString();
...
```

If we do not keep track of what kind of objects o could refer to, we would have to expect that any effect of any toString() implementation could happen at the assignment to s. If we know all possible points to information about o then we can do better. In other words, interprocedural analysis and points to analysis are interrelated in languages with indirect function calls.

3 May Point To Analysis

The general question is what the objects on the heap are that a variable p of pointer type $\tau*$ could possibly point to. There are several choices for pointer analysis. We choose to track if p can point to a heap object h. Yet, we do not explicitly track all the heap objects during compiler optimizations. Instead, we throw all objects created at one statement h together and track only if p can point to any of the heap objects created at statement h. That is, we use allocation site analysis, where the allocation site is the abstraction primitive.

Whenever we find a pointer assignment $q = p$ we need to track how the possible points-tos of p relate to those of q. In what is called *inclusion-based analysis*, $q = p$ will have the effect in our analysis that q may point to all objects that p may point to (but not the other way around). In *equivalence-based* analysis, $q = p$ would instead merge the points-tos of p and q into one

common equivalence class. Equivalence-based analysis is not particularly useful for alias analysis, because it merges too much information. But it can be useful to determine which variables point to the same kind of objects, because it directly tracks equivalence classes.

We write $points(x, X)$ if the variable x may point to a heap object X. Here we decide to classify heap objects by their allocation site, i.e., the line of code that was used when creating them. That is, we write $points(x, l)$ if the variable x may point to a heap location created at statement l. The first rule says that a memory allocation at statement l obviously causes the target to point to a memory allocation allocated at l:

$$\frac{l : x = \textbf{alloc}(\tau)}{points(x, l)}\ P_N$$

From now on, we do not really care what a heap object X is. Rule P_N seeds them from allocation sites, but other more precise analyses would be possible too, like for instance allocation site plus call site. This is useful if all objects of a type are allocated in an object factory function like

```
node* newListNode(int value) {
  node *p = alloc(node);
  p->v=value;
  return p;
}
```

In inclusion-based analysis, if we assign $x = y$ then x may point to any heap object Y that y may point to:

$$\frac{l : x = y \quad points(y, Y)}{points(x, Y)}\ P_C$$

This is an example where we loose information by tracking globally and in ignorance of the context what variables x may point to. Because any information we still have about $points(x, Z)$ from prior lines of code is still around, but no longer of relevance for code dominated by this assignment $x = y$. Flow-sensitive analysis could do better here, but is computationally more expensive.

Suppose we update a pointer field of a pointer by $x{->}f = y$. Then if x may point to X and y may point to Y then the f field of X may point to Y: We introduce the notation $mpoints(l, f, l')$ if the field f of a struct that

has been created in heap memory at program location l can point to a heap memory location that has been created at program location l'.

$$\frac{l : x{->}f = y \quad points(x, X) \quad points(y, Y)}{mpoints(X, f, Y)}\ P_S$$

Likewise when we read a pointer field $x = y{->}f$, then if y may point to Y and the f field of Y may point to X then x may now point to X:

$$\frac{l : x = y{->}f \quad points(y, Y) \quad mpoints(Y, f, X)}{points(x, X)}\ P_L$$

4 Type-Based Analysis

The general question is what the objects on the heap are that a variable p of pointer type $\tau*$ could possibly point to. In C, we do not know a whole lot about that a priori without doing extensive analysis. And even then it is quite difficult to figure out what p may point to.

In C0, we have a secret weapon. The language is type-safe. At least know that, at any time, $p : \tau*$ can only point to objects of type τ. Type-preservation implies that, whenever e is an expression of type τ, then all evaluation of e by $e \Rightarrow v$ can only yield values v of type τ. Especially, since $*p$ is an expression of type τ, p will point to an object of type τ. When comparing $p : \tau*$ to a pointer $q : \sigma*$ to a different (and disjoint) type $\sigma \neq \tau$, then we immediately know that they cannot alias. We write $\neg aliasable(e, e')$, if e and e' cannot possibly be aliases, because the types mismatch. We write $aliasable(e, e')$ if, indeed, for all we know from a type perspective, e and e' could still alias, but the points-to analysis might be able to tell us more. To make it more explicit when two pointer expressions can alias and when they cannot, we state the positive and negative rule:

$$\frac{e : \tau* \quad e' : \tau*}{aliasable(e, e')}\ P_*^+ \qquad \frac{e : \tau* \quad e' : \sigma* \quad \sigma \neq \tau \text{ disjoint}}{\neg aliasable(e, e')}\ P_*^-$$

That is, if expressions e and e' have the same pointer type $\tau*$, then they can alias. If expression e has type $\tau*$ and expression e' has different type $\sigma*$ (and $\sigma \neq \tau$ are disjoint types), then they cannot alias. Note that, when including other analyses, there could be other reasons denying aliasability (see last sections), but they are not just based on the types. For polymorphic

languages where types can overlap (e.g., a String is an Object), aliasability is handled similarly, but there are more cases, because types can be different but still not disjoint.

Note that, for this to work out, we also need to know that C0, unlike C, does not allow arbitrary memory references converted from non-pointer data (e.g., ints) or pointer arithmetic, because those would spoil type-safety. C0 also does not have an address-of operator $\&e$ that would allow to obtain a pointer to an arbitrary location e.

The C0 type system guarantees that we know something about what a struct field access can alias with. It can only alias with the same field of the same struct, nothing else.

$$\frac{e : \textbf{struct}\ s \quad e' : \textbf{struct}\ s \quad aliasable(e, e')}{aliasable(e.a, e'.a)}\ P_{.}^{+} \qquad \frac{e : \textbf{struct}\ s \quad e' : \textbf{struct}\ t \quad s \neq t}{\neg aliasable(e.a, e'.a)}\ P_{.}^{-}$$

A field access $e.a$ can only alias with a field access $e'.a$ if e and e' are aliasable and both have the same struct type.

C0 array access $e[i]$ can only alias with array access $e'[i]$ if e and e' alias and have the same type and $i == j$.

$$\frac{e : \tau[] \quad e' : \tau[] \quad aliasable(e, e')}{aliasable(e[i], e'[j])}\ P_{[]}^{+} \qquad \frac{e : \tau[] \quad e' : \sigma[] \quad \sigma \neq \tau}{\neg aliasable(e[i], e'[j])}\ P_{[]}^{-}$$

The most useful conclusion of these rules occurs when e and e' are variables of array type (and not complicated path navigation). Then two arrays declared in local variables `int a[];` and `int b[];` with different names can never alias. That is `a[i]` and `b[j]` are always different. Note that this does not hold if `int a[]` and `int b[]` are function parameters. When one of the two is a function parameter and the other one is local, then at least we know that they will be different, because newly allocated arrays cannot alias with previously existing arrays.

None of this holds for C, where arrays can come from anywhere.

5 Combining Type-Based Analysis and May Point To Analysis

When we have information about two expressions e and f that cannot possibly alias, then we know that whatever e may point to cannot be what f

may point to:

$$\frac{\neg aliasable(e, f) \quad points(e, E)}{\neg points(f, E)}\ A???$$

But now we have multiple rules that can conclude whether $points(f, E)$ holds or not! Previous rules conclude positive statements but rule $A???$ concludes a negative statement. In previous lectures, whenever we had a negative fact, there was a good reason why we could use it. One reason was, because it never occurs in positive form, then we can just rename the negative predicate $\neg p(x)$ to the positive predicate $p_{not}(x)$. If the predicate occurs both positively and negatively, then the other reason why the rules still worked in previous lectures was that there was a way to stratify the rules into two sets of rules such that the first can saturate the predicate with only positive occurrences and after that the second set can use negative occurrences but never produce new instances of that particular predicate.

Here the situation is different. The rules all produce $points$ conclusions and could thus possibly contradict each other. The question is whether we could possibly conclude both $points(f, E)$ and $\neg points(f, E)$ for then we would have inconsistent rules. This would be terrible, so we cannot just use arbitrary negations in logic programming rule without thinking about it carefully.

What saves us now is precedence. We have two separate analysis reasons, and if both typing and points-to analysis tell us something then the typing information should take precedence. The first attempt of getting this right is that let rule $A???$ overwrite conflicting $points(f, E)$ information. Hence, the negative conclusion takes precedence over positive facts in the database in order to maintain consistency. But that doesn't quite work. In particular, if we have reason to believe that e may point to E, i.e., $points(e, E)$, and also that f may point to E, i.e., $points(f, E)$, then we could use rule $A???$ both ways to derive either $\neg points(f, E)$ or $\neg points(e, E)$. Which one do we remove? The answer might be arbitrary and at least depends on how we derived the facts. This is getting too tricky and subtle to get right easily. The branch of logic that deals with facts that come and go and are made consistent again by various notions of precedence is called nonmonotonic logic. But we are looking for an easier answer.

Instead, what we will do is not even derive $points(f, E)$ facts in the first place, if they would contradict what we can conclude from the typing information about possible aliasability. We achieve this by adding an extra assumption to all points to rules that checks if the conclusion would make

sense from a type perspective:

$$\frac{l : x = y \quad points(y, Y) \quad aliasable(x, y)}{points(x, Y)} \; P_C^+$$

This rule makes sense. If y may point to Y and we assign $x = y$ then x may also point to Y if both are aliasable, that is, unless we know that x cannot alias y.

6 Points-to Analysis and Types

The points-to analysis is entirely ignorant for typing information. It may conclude possible point-tos that cannot happen because of a type mismatch. We can directly integrate points-to analysis and types on the usual cases. In order to take types into account, we use a predicate $mtype(X, \tau)$ that specifies for a heap object X what type it has. If the type is not known precisely, some conservative overapproximation would be used, e.g., the most specific type in common when the type system forms a lattice. We write $mtype(X, \tau_{\leq})$ if the type of heap object X is type τ or one of its subtypes $\sigma \leq \tau$. For instance, in Java, $mtype(X, Number_{\leq})$ would be the disjunction

$$mtype(X, Integer) \vee mtype(X, Float) \vee mtype(X, BigInteger) \vee mtype(X, BigDecimal) \vee ...$$

$$\frac{l : x = \textbf{alloc}(\tau)}{\begin{array}{c} points(x, l) \\ mtype(l, \tau) \end{array}} \; P_N$$

$$\frac{l : x = y \quad x : \tau \quad points(y, Y) \quad mtype(Y, \tau_{\leq})}{points(x, Y)} \; P_C$$

$$\frac{l : x{-}{>}f = y \quad points(x, X) \quad points(y, Y) \quad mtype(X, s_{\leq}) \quad \textbf{struct}\ s\ \{\ldots \tau f\} \quad mtype(Y, \tau_{\leq})}{mpoints(X, f, Y)} \; P_S$$

$$\frac{l : x = y{-}{>}f \quad x : \tau \quad points(y, Y) \quad mpoints(Y, f, X) \quad mtype(X, \tau_{\leq})}{points(x, X)} \; P_L$$

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman, Boston, MA, USA, 2nd edition, 2006.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.