Pointer (and Alias) Analysis:

Goal is to determine when two expressions can point to the same memory location.

Memory aliasing

The problem:

```c
foo() {
    int a, k;
    extern int *q;
    ... maybe &a or &k
    ...
    k = a + 6;
    f(a, &k);
    *q = 13;
    k = a + 6  <---- is this necc or not?
    ...
}
```

What could cause the statement to be required?
- call could cause k to be modified
- ptr assignment could cause k or a to be modified.

Aliases can arise from:
- pass by reference parameters
  ```c
  foo(ref i, ref j) { k=i; l=j; if (k==l) ... }
  foo(int* i, int* j) { ... }
  ```
- address of operator
  ```c
  p = &q; q = 5; ...; *p = 5;
  ```
- derefencing pointers
  ```c
  *p
  ```
- array subscripting
  ```c
  i=foo(); j=bar(); k = a[i]; l=a[j]; if (k==l) ... 
  ```
- non-local variables
  ```c
  let var i =
  function foo(j) = { i=j }
  in
  foo(i)
  end
  ```
- assignment
  ```c
  a = new X; b = new X; c = b; ...; c = a;
  ```

Two kinds of alias information:
must-alias information and may-alias information.

```
 p = &q
 a / \ b
 p= &x ... c \ / d
    ... | e
```

at a p must alias with adr of q
at b p must alias with adr of q
at c p must alias with adr of x
at d p must alias with adr of q
at e p MAY alias with adr of x or q

Two kinds of analysis:
flow-insensitive: independent of control-flow. i.e., property holds for entire block/function/program
flow-sensitive: dependent on control-flow, i.e., property holds at program point p
Flow-insensitive:
useful in languages with strong typing.
use type information -> gathering of alias information must happen when types are still present.
----------------
Kinds of alias information you might collect:

Points-to information (may or must):
- at program point p, determine the set of pairs (v, x) which indicate v can point to x.

Alias Pairs (must or may)
- at program point p, determine the set of pairs (exp1, exp2) where exp1 and exp2 must/may point to the same memory location.

Shape Analysis
- at program point p, determine an abstract description of a pointer structure (e.g., list, tree, etc.)

Two types of pointer analysis problems:
- pointers to heap addresses (e.g., from malloc)
  heap analysis
- pointers to stack objects (e.g., &foo where foo is local)
  stack analysis

stack analysis has the benefit that locals have a compile time name

To solve this for heap analysis it is often the case that memory malloced at a line number is given the name "created at line x"

How might it be used:
- improve all previous analysis (CSE, PRE, CP, etc.)
- eliminate redundant loads/stores
  x = *p;
  ...
  y = *p; do we need this?
- parallelization
  can we parallelize across function calls?
- safety analysis

Structure of a PA:

Gathering alias information
- find all the abstract memory locations

Propogating alias information
- manipulate them

Let's do a flow-based may-alias intra-procedural stack analysis that understands pointer arithmetic.

Sets of tuples for each program point:

(t, d, k)
t = variable
d = is an alias class (i.e., an abstract location set)
k = offset into an alias class

At each program point P there is a set of tuples such that if tuple (t,d,k) is in the set, then, variable t may point to alias class d at offset k. Or, t-k points to d.

At the start of the function A = ((FP, frame, 0)) where FP is the frame pointer register and frame is the base of the stack frame for the local variables.

We will generate these tuples using a data flow analysis based on transfer functions:
A is the current state of alias information at program point p.
\(\sigma_t\) = set of all tuples that are type compatible with t
Assume compiler knows about type of new/malloc, i.e.,

\[
a = (\text{struct bar*})\text{malloc}(\text{sizeof(struct bar)})\text{ creates the tuple} \ (a, \text{fresh-bar-space}, 0) \text{ where fresh-bar-space is type} \ \text{compatible with bar and is a new location set.}
\]

DF equations:

\[
in[\text{entry}] = \{(\text{FP, frame, 0})\}
\]
\[
in[n] = \text{Union over all preds, } p: \text{out}[p]
\]
\[
\text{out}[n] = \text{trans}_n(\text{in}[n])
\]

Where \text{trans}_n is defined for each stmt type:

\[
t < - b \quad (A - \sigma_t) \cup \{(t,d,k)| (b,d,k) \in A \}
\]
all possible location sets reached by b are now reached by t

\[
t < - b + k \quad (A - \sigma_t) \cup \{(t,d,i)| (b,d,i-k) \in A \}
\]
this is for k constant.
all possible location sets reached by b are now reached by t with an
offset of k

\[
t < - b \text{ op c} \quad (A - \sigma_t) \cup \{(t,d,i)| (b,d,k) \in A \text{ OR} \ \ (c,d,j) \in A \}
\]

\[
t = \text{M}[b] \quad A \cup \sigma_t
\]

\[
d: t < - \text{new a} \quad (A - \sigma_t) \cup \{(t,d,0)\}
\]
\[
t < - f(...) \quad A \cup \sigma_t
\]

\[
\text{struct intlist} \\
\quad \text{int val;} \\
\quad \text{struct intlist* next;}
\]

\[
\text{struct int foo()} \\
\quad \text{struct intlist* p = 0;} \\
\quad \text{struct intlist* q = 0;} \\
\quad \text{int a = 1;} \\
\quad \text{q = new intlist(0, NULL);} \\
\quad \text{p = new intlist(0, q);} \\
\quad \text{q->val = 5;} \\
\quad \text{a = p->val;} \\
\quad \text{return a;}
\]

1: MOVE t2 <- 0 p
2: MOVE t3 <- 0 q
3: MOVE t4 <- 1 a
4: CALL t5 <- malloc(8) (t5,4,0) q <- new
5: PLUS u1 <- t5, 0 (u1,4,0) q->val = 0
6: STORE MEM[u1] <- 0 q->next = 0
7: PLUS u2 <- t5, 4 (u2,4,4) q->next = q
8: STORE MEM[u2] <- 0
9: MOVE t3 <- t5 (t3,4,0) q
10: CALL t6 <- malloc(8) (t6,10,0) p <- new
11: PLUS u3 <- t6, 0 (u3,10,0) p->new = 0
12: STORE MEM[u3] <- 0
13: PLUS u4 <- t6, 4 (u4,10,4) p->next = q
14: STORE MEM[u4] <- t3 p->next = q
15: MOVE t2 <- t6 (t2,10,0) p
16: PLUS u5 <- t3, 0 (u5,4,0) p
17: STORE MEM[u5] <- 5 q->val = 5
18: PLUS u6 <- t2, 0 (u6,4,0) p
19: LOAD t4 <- MEM[u6] a
20: RET t4
struct intlist* foo() {
    struct intlist* p = 0;
    struct intlist* q = 0;
    int a = 1;

    p = new intlist(0, NULL);
    q = new intlist(6, p);
    if (a == 0) p = q;
    else p->val = 4;
    return p;
}

1: CALL t2 <- malloc(8) (t2,1,0) p
2: PLUS u1 <- t2, 0 (u1,1,0)
3: STR MEM[u1] <- 0 p->val = 0
4: PLUS u2 <- t2, 4 (u2,1,4)
5: STR MEM[u2] <- 0 p->next = 0
6: MOVE t3 <- t2 (t3,1,0) x=p
7: CALL t4 <- malloc(8) (t4,7,0) q
8: PLUS u3 <- t4, 0 (u3,7,0)
9: STR MEM[u3] <- 6 q->val = 6
10: PLUS u4 <- t4, 0 (u4,7,0)
11: STR MEM[u4] <- t3 q->next = p
12: MOVE t5 <- t4 (t5,7,0) q
13: MOVE t6 <- 1 a
14: CJUMP (EQ, t6, 0, ifT0, ifF1 a == 0 ?
15: LABEL ifT0
16: MOVE t3 <- t5 (t3,7,0) x=q
17: LABEL ifF1
18: PLUS u5 <- t3, 0 (u5,7,0), (u5,1,0)
19: STR MEM[u5] <- 4

How do we use this information:

Example: available expressions:

\[
\begin{align*}
\text{gen} & \quad \text{kill} \\
\text{t} \leftarrow \text{b op c} & \quad \text{b op c - kill[s]} \\
\text{t} \leftarrow \text{M[b]} & \quad \text{M[b] - kill[s]} \\
\text{M[a]} \leftarrow \text{b} & \quad \text{} \\
\text{f(...)} & \quad \text{} \\
\end{align*}
\]

However, if we know \( \text{M[a]} \) and \( \text{M[b]} \) are different, then line 3 is too general:

\[
\text{M[a]} \leftarrow \text{b} \quad \text{()}
\]

Example from above: Constant propagation (treat each \((d, k)\) as a variable)

Another way is to treat each location set as a var, so for instance above we could figure out that a has value 0.

This brings up order of optimizations. First we do constant prop, then alias, then constant prop?

---------

interprocedure analysis
shape analysis
new time interprocedural analysis.

What do we do with:

1:p = malloc
2:q = malloc
3:p->next = q;
4:p->next->val = 4;

\[t_1 \leftarrow p \quad (t_1, 1, 0)\]
\[t_2 \leftarrow t_1 + 4 \quad (t_2, 1, 4)\]
\[M[t_2] \leftarrow q \quad p->next = q\]
\[t_3 \leftarrow p \quad (t_3, 1, 0)\]
\[t_4 \leftarrow t_3 + 4 \quad (t_4, 1, 4)\]
\[t_5 \leftarrow M[t_4] \quad (t_5, 1, 0) (t_5, 2, 0)\]
\[t_6 \leftarrow t_4 + 0 \quad (t_5, 1, 0) (t_5, 2, 0)\]
\[M[t_6] \leftarrow 0 \quad ?\]

Shape analysis:
determine if a recursive data structure has the form of a tree, dag, cycle.
tree: unique path from p to any two nodes accessible from p
dag: multiple paths from p, but no nodes can reach themselves
cycle: otherwise, cycle

what can we get from this?
assume p is a tree, then p->a and p->b are definitely not aliased.

How can we determine shape of a data structure?

dataflow analysis which manipulate direction, interference matrices.
Using D and I we can determine shape of each structure at each program point.

To be continued ...