

Alias analysis

- Memory analysis
 - *Forward dataflow analysis*
 - *Loop optimization*
 - *Alias classes*
- CSE optimization
- Store coalescing



CSE on memory

```
int alias(int[] a, int i, int j) {  
    let x = a[i] in  
    a[j] = 1;  
    let y = a[i] in  
    return x == y;  
}
```

Eliminate with CSE?

```
int alias(int[] a, int[] b, int i) {  
    let x = a[i] in  
    b[i] = 1;  
    let y = a[i] in  
    return x == y;  
}
```



Alias classes

- The problem is uncomputable (naturally)
- The usual approximation is to assign each variable an alias class: an equivalence class containing variables that may alias each other
- Usually, a class is just a number



Levels of alias analysis

- Levels of alias analysis
 - *Basic: use alias classes based on type*
 - *Point-of-definition: define alias classes by allocation point*
 - *These first two “lose” information about stored data*
 - *Alias types for arrays and records*
 - *Set-based/constraint programming*
 - *Analyze possible values of subscripts*



Type-based alias analysis

- In type-safe languages, different types cannot alias
- Covers a small number of interesting cases

```
let alias(a : { l1 : int; ... }, b : { l2 : int; ... }) =  
    a.l1 <- 1;  
    b.l2 <- 2;  
    let x = a.l1 in  
    let y = b.l2 in  
    let z = (x == y) in  
    ...
```



Point-of-definition alias classes

- Arrays/records allocated at different program points can never be aliases

```
int f(int[] x, boolean b) =  
    if b then  
        let y = malloc(...) in  
            alias(x, y)  
    else  
        alias(x, x)  
let alias(int[] a, int[] b) =  
    b[0] <- a[0];
```

...



Alias types

$$\begin{aligned} s & ::= \{v_i, v_j, v_k, \dots\} \\ \tau & ::= \top \mid \text{array}(s, \tau_1 \times \dots \times \tau_n) \end{aligned}$$

- every variable v has an alias type τ
- s is a set of variables that define the possible definition points of a variable (the alias class)
- arrays have
 1. an alias class s
 2. an alias type τ_i for each element
 3. \top is the "unknown" class



Type inference

- We have a program without type information
- The task to assign types to all the variables in the program
- Type inference is a forward-dataflow problem
 - *We can infer types from the way that variables are used*
 - *Variables that are never used are polymorphic*
 - *When a function is called, the types of the parameters must be unified with the types of the arguments*



ML type inference

- Intuitively, this is the way that ML type inference works
 - *First, assume an arbitrary polymorphic type for each variable*
 - *When a variable is used, the context provides type info*
 - *If a variable is added to a number, it must be a number*
 - *The important case is a function call*
 - *If a variable of type t_1 is passed to a function expecting an argument of type t_2 , then t_1 and t_2 must be the same*
 - *This step uses type unification*
 - *If a variable is applied with n arguments, it must be a function expecting at least n arguments*



Alias type system

- We only have array types and “other” types
- Unification:

$$\begin{aligned} \text{unify}(\top, _) &= \top \\ \text{unify}(_, \top) &= \top \\ \text{unify}(\text{array}(s_1, \tau_{1,1} \times \dots \times \tau_{1,n}), \\ &\quad \text{array}(s_2, \tau_{2,1} \times \dots \times \tau_{2,n})) = \\ \text{array}(s_1 \cup s_2, \text{unify}(\tau_{1,1}, \tau_{2,1}) \times \dots) \end{aligned}$$



Starting point

- For each function that can be called from another file: give each parameter the type \top .
- For all other functions, give record parameters the type $array(\{\}, \tau_1 \times \dots \times \tau_n)$, where τ_i is \top if the t_i is not an array type, otherwise $\{\}$.
- For an allocation of variable v , give the variable the type $array(\{v\}, \tau_1 \times \dots \times \tau_n)$ where τ_i is the type of initializer a_i .



Iteration: LetSubscript

- For each LetSubscript (v, ty, a, i, e) , if a has type $\text{array}(s, \tau_1 \times \dots \times \tau_2)$, then give v the type τ_i .
- If i is not known, give v the type τ .



SetSubscript

- For each SetSubscript (v, i, ty, x, e) , if v has type $array(s, \tau_1 \times \dots \times \tau_2)$ and x has type τ , give v the type $array(s, \tau_1 \times \tau_{i-1} \times \tau \times \tau_{i+1} \times \dots \times \tau_2)$
- Also, assign the type to each $v' \in s$
- Otherwise, give v the type $array(s, \top \times \dots \times \top)$



Function Call

- For each function call $f(a_1, \dots, a_n)$, where f has formal parameters v_1, \dots, v_n .
- If a_i has type τ_i
- and v_i has type τ'_i ,
- give v_i the type $unify(\tau_i, \tau'_i)$



Finally

- At each point in the program, each variable v has some type τ
- For LetSubscript (v, ty, a, i, e) , when performing CSE, if there is an available expression for $a[i]$ with type τ , and $a[i]$ still has type τ , the LetSubscript may be replaced with a LetAtom



Set-based analysis

- We can use the same kind of reasoning for integer variables.
- Each integer gets a set type $\{interval_1, \dots, interval_n\}$.
- Type unification is set union.
- given two subscripts $i:s$ and $j:t$, if $s \cap t = \{\}$, then i and j do not alias.

