Using type inference to discover interesting properties about programs: Pointer Analysis

Amer Diwan

What is pointer analysis?

- Find out what each pointer could point to
  - For each point in the program (flow sensitive)
  - For all points in the program (flow insensitive)
- Example:

```
p = &x;      <- p points to x
*p = 10;                              p points to x or y
p = &y;      <- p points to y
*p = 20;
```

Why do we need pointer analysis? (I)

- f(int *p, int *q) {
  *p = E1;
  *q = E2;
  print(*p); }
- Accessing memory (e.g., *p) is expensive:
- f(int *p, int *q) {
  *p = E1;
  *q = E2;
  print(E1); }
- When is this legal?

Why do we need pointer analysis (II)

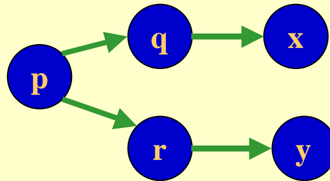
- f(int **p, int **q) {
  *q = NIL;
  **p = E; }
- Segmentation violation!!!

Why is *p NIL???

Storage shape graph

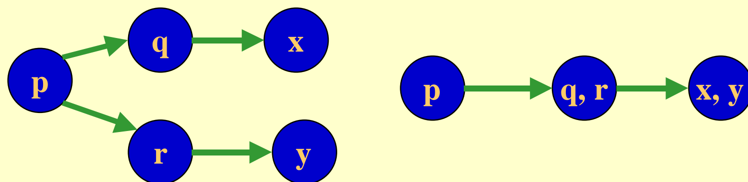
- Describes the shape of storage
  - Nodes represent one or more memory locations
  - Edges represent points-to relationships
- E.g.,

```
q = &x;
r = &y;
if (cond) p = &q
else p = & r
```



Storage shape graph and approximations

- Steensgaard computes computes SSGs in which nodes have at most one outgoing edge



Algorithm approach

- Traditional types specify allowable values
- Paper computes "non-standard" types that specify points-to relationships
- Even traditional types have some points-to information built in
  e.g., variable v: T and u: U cannot point to the same object of T and U are "incompatible types"

Paper approach (cont.)

- Give a set of rules that indicate when an expression is well typed
- Use rules to perform type inference
- Inferred types make up the storage shape graph

How does this topic fit into this class?

- The problem is simpler than SML
  - We can see a full type inference system in action
- Idea that different kinds of type systems exist
  - Type system describes properties of programs

The types

$\alpha ::= \tau \times \lambda$ Types of values

$\tau ::= \perp \mid ref(\alpha)$ Types of locations

$\lambda ::= \perp \mid lam(\alpha_1..\alpha_n)(\alpha_{n+1}..\alpha_{n+m})$ Types of functions

In english...

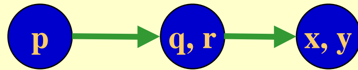
A type is a pair of a τ and a λ:

τ represent pointers to data

λ represent pointers to functions

$\perp$ (pronounced bottom) means not yet known to be a pointer

Example



p: $\tau 1 = ref(\tau 2 \times \bot)$ i.e., a pointer to $\tau 2$
q: $\tau 2 = ref(\tau 3 \times \bot)$ i.e., a pointer to $\tau 3$
r: $\tau 2$
x: $\tau 3 = ref(\bot \times \bot)$ i.e., not known to be a pointer
y: $\tau 3$

Typing rules

- Not too different from programming language types

$$\frac{\text{typing requirements}}{\text{expression}}$$

Two ways of thinking about it:
If typing requirements are satisfied then expression is legal
If you see expression then assert that typing requirements must hold

Simple assignment typing rule

$$\frac{A \vdash x: \mathrm{ref}(\alpha 1) \quad A \vdash y: \mathrm{ref}(\alpha 2) \quad \alpha 2 \leq \alpha 1}{A \vdash \mathrm{welltyped}(x = y)}$$

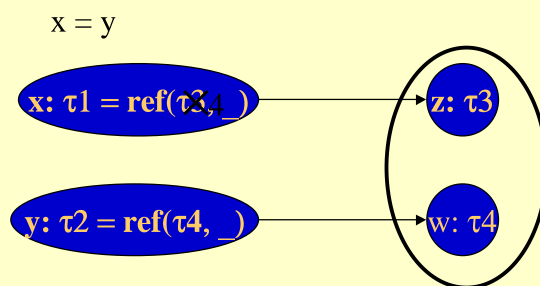
Type inference view of the rule:

when you see x = y, assert that $\alpha 2 \leq \alpha 1$

(i.e., $\alpha 2$ is bottom or $\alpha 2 = \alpha 1$)

Example

x = y



τ3 and τ4 are made equivalent

Address assignment typing rule

$$\frac{A \vdash x: ref(\tau \times _) \qquad A \vdash y: \tau}{A \vdash welltyped(x = \&y)}$$

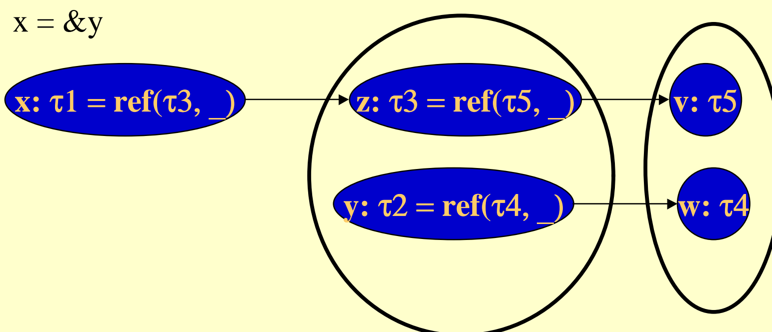
Type inference view of the rule:

when you see x = &y, assert that x is a pointer to τ which is y's type

Note: don't need a ≤ requirement since &y is a pointer by definition

Example

x = &y



τ2 and τ3 are made equivalent
which causes τ4 and τ5 to become equivalent

Dereferenced assignment type rule

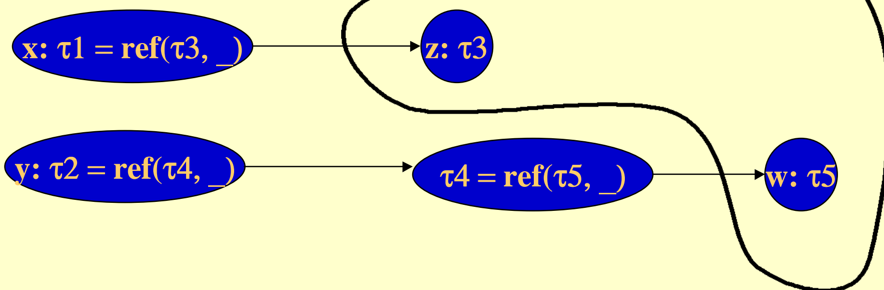
$$\frac{A \vdash x: ref(\alpha 1) \quad A \vdash y: ref(ref(\alpha 2) \times _) \quad \alpha 2 \leq \alpha 1}{A \vdash welltyped(x = *y)}$$

Type inference view of the rule:

when you see x = *y, assert that x is a pointer to whatever *y was a pointer to if *y might be a pointer

Example

x = *y



τ3 and τ5 are made equivalent (assume both are pointers)

Function pointer assignment rule

$A \vdash x: ref(_ \times lam(\alpha 1)(\alpha 2))$
$A \vdash f: ref(\alpha 1)$
$A \vdash r: ref(\alpha 2)$
forall s in S^* : $A \vdash welltyped(s)$

$A \vdash welltyped(x = fun(f)\text{->}(r): S^*)$

Things to note

- Each type “points to” at most one other location
  - => SSG nodes have at most one outgoing edge
  - => Merging two types causes their referent types to also be merged

Type inference rules

- Type inference rules are derived directly from the typing rules
- Complexity:
  - If a type is $\perp$ then it should not be merged immediately with other types-causes imprecision
  - Defer merges of $\perp$. If it is found to be a pointer, then merge then.

An example

$$\frac{A \vdash x: ref(\alpha1) \quad A \vdash y: ref(\alpha2) \quad \alpha2 \leq \alpha1}{A \vdash welltyped(x = y)}$$

```
x = y
   let ref(τ1 × λ1) = type(ecr(x))
       ref (τ2 × λ2) = type(ecr(y)) in
     if τ1 != τ2 then cjoin(τ1, τ2)
     if λ1 != λ2 then cjoin(λ1, λ2)
```

Advantages and disadvantages of this approach

- Advantages
  - type inference based: can derive an algorithm systematically from typing rules
  - fast
- Disadvantages
  - imprecise

Summary

- Non-standard types can be used to compute interesting properties about programs
- Next lecture: Lackwit (reading on web page)
  - Using type inference to find bugs