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 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

\[ \alpha ::= \tau \times \lambda \quad \text{Types of values} \]
\[ \tau ::= \bot \mid \text{ref}(\alpha) \quad \text{Types of locations} \]
\[ \lambda ::= \bot \mid \text{lam}(\alpha_1..\alpha_n)(\alpha_{n+1}..\alpha_{n+m}) \quad \text{Types of functions} \]

In english...

A type is a pair of a \( \tau \) and a \( \lambda \):

\( \tau \) represent pointers to data
\( \lambda \) represent pointers to functions
\( \bot \) (pronounced bottom) means not yet known to be a pointer
Example

\[ p: \tau_1 = \text{ref}(\tau_2 \times \bot) \text{ i.e., a pointer to } \tau_2 \]
\[ q: \tau_2 = \text{ref}(\tau_3 \times \bot) \text{ i.e., a pointer to } \tau_3 \]
\[ r: \tau_2 \]
\[ x: \tau_3 = \text{ref}(\bot \times \bot) \text{ i.e., not known to be a pointer} \]
\[ y: \tau_3 \]

Typing rules

• Not too different from programming language types

\[
\text{typing requirements} \Rightarrow \text{expression}
\]

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

A ⊢ x: ref(α1)
A ⊢ y: ref(α2)

α2 ≤ α1

A ⊢ welltyped(x = y)

Type inference view of the rule:
when you see x = y, assert that α2 ≤ α1
(i.e., α2 is bottom or α2 = α1)

Example

x = y

x: τ1 = ref(τ2, _)  
y: τ2 = ref(τ4, _)

τ3 and τ4 are made equivalent
Address assignment typing rule

\[
\begin{align*}
A \vdash x : \text{ref}(\tau \times _) \\
A \vdash y : \tau \\
\hline
A \vdash \text{welltyped}(x = &y)
\end{align*}
\]

Type inference view of the rule:
when you see \( x = &y \), assert that \( x \) is a pointer to \( \tau \) which is \( y \)'s type
Note: don't need a \( \leq \) requirement since \( &y \) is a pointer by definition

Example

\( x = &y \)

\( x : \tau_1 = \text{ref}(\tau_3, _) \)
\( z : \tau_3 = \text{ref}(\tau_5, _) \)
\( v : \tau_5 \)
\( y : \tau_2 = \text{ref}(\tau_4, _) \)
\( w : \tau_4 \)

\( \tau_2 \) and \( \tau_3 \) are made equivalent
which causes \( \tau_4 \) and \( \tau_5 \) to become equivalent
Dereferenced assignment type rule

\[
A \vdash x : \text{ref}(\alpha_1) \\
A \vdash y : \text{ref}(\text{ref}(\alpha_2) \times _) \\
\alpha_2 \leq \alpha_1 \\
A \vdash \text{welltyped}(x = ^*y)
\]

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

\(x = ^*y\)

\(x : \tau_1 = \text{ref}(\tau_3, _)\)
\(z : \tau_3\)

\(y : \tau_2 = \text{ref}(\tau_4, _)\)
\(\tau_4 = \text{ref}(\tau_5, _)\)
\(w : \tau_5\)

\(\tau_3\) and \(\tau_5\) are made equivalent (assume both are pointers)
Function pointer assignment rule

A \triangleright x: \text{ref}(\_ \times \text{lam}(\alpha_1)(\alpha_2))
A \triangleright f: \text{ref}(\alpha_1)
A \triangleright r: \text{ref}(\alpha_2)
\forall s \in S^*: A \triangleright \text{welltyped}(s)

\text{welltyped}(x = \text{fun}(f)\rightarrow(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 $\bot$ then it should not be merged immediately with other types-causes imprecision
  – Defer merges of $\bot$. If it is found to be a pointer, then merge then.

An example

\[
\begin{align*}
\text{A} \vdash x & : \text{ref}(\alpha_1) \\
\text{A} \vdash y & : \text{ref}(\alpha_2) \\
\alpha_2 & \leq \alpha_1 \\
\text{A} \vdash \text{welltyped}(x = y) \\

x & = y \\
\text{let } & \text{ref}(\tau_1 \times \lambda_1) = \text{type(ecr(x))} \\
& \text{ref}(\tau_2 \times \lambda_2) = \text{type(ecr(y)) in} \\
\text{if } & \tau_1 \neq \tau_2 \text{ then } \text{cjoin}(\tau_1, \tau_2) \\
\text{if } & \lambda_1 \neq \lambda_2 \text{ then } \text{cjoin}(\lambda_1, \lambda_2)
\end{align*}
\]
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