Lecture 18: Alias analysis

Unification

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Programming Languages

Based on slides by Jeff Foster
Introduction

- *Aliasing* occurs when different names refer to the same thing
  - Typically, we only care for imperative programs
  - The usual culprit: pointers
- A core building block for other analyses
  - For example in *p = 3;* what does *p* point to?
- Useful for many languages
  - C – lots of pointers all over the place
  - Java – “objects” point to updatable memory
  - ML – ML has updatable references
Aliases analysis

- **Alias analysis** answers the question
  - Do pointers \( p \) and \( q \) alias the same address?
- Unfortunately, undecidable
  - Remember Rice’s theorem: *No program can precisely decide anything interesting about arbitrary source code*
- Usual solution: allow imprecision
  - Decision problem: yes/no – undecidable
  - Approximation: yes/no/maybe – decidable
May alias analysis

- **p** and **q** *may alias* if it is possible that **p** and **q** might point to the same address.
- **Negative answer is precise**
  - “yes” – imprecise, means **p** and **q** might alias
  - “no” – precise, means **p** and **q** never alias
- **If** **p** may *not* alias **q**, then a write through **p** does not affect memory pointed to by **q**
  - *p = 3; x = *q;* means write through **p** does not affect **x**
- **What is the most conservative may-alias analysis?**
Must alias analysis

- **p** and **q** *must alias* if they do point to the same address.
- Positive answer is precise:
  - "yes" – precise, means **p** and **q** definitely alias.
  - "no" – imprecise, means **p** and **q** might not alias.
- If **p** must alias **q**, then a write through **p** always affects memory pointed to by **q**:
  - \(*p = 3;\) \(x = *q;\) means \(x\) is 3.
- What is the most conservative must-alias analysis?
Early alias analysis

- By Landi and Ryder
- Expressed as computing alias pairs
  - E.g., \((\ast p, \ast q)\) means \(p\) and \(q\) may point to the same memory
- Issues?
  - There could be many alias pairs
    - \((\ast p, \ast q), (p->a, q->a), (p->b, q->b), \ldots\)
  - What about cyclic data structures?
    - \((\ast p, p->next), (\ast p, p->next->next), \ldots\)
Points-to analysis

- Determine the set of locations that $p$ may point to
  - E.g., $(p, \{&x\})$ means $p$ may point to the location of $x$
  - To decide if $p$ and $q$ alias, see if their points-to sets overlap

- More compact representation
  - The same aliasing information takes less memory
  - Analysis scales better

- We must name all locations in the program
  - Pick a finite set of location names
    - No problem with cyclic data structures
  - $x = \text{malloc}(\ldots)$; – where does $x$ point to?
    - $(x, \{\text{malloc@42}\})$ – “the \text{malloc()} at line 42”
Flow-sensitivity

- An analysis is *flow-sensitive* if it computes the answer at every program point
  - We saw that dataflow analysis is flow-sensitive
- An analysis is *flow-insensitive* if it does not depend on the order of statements
  - We saw that type systems are flow-insensitive
- Flow-sensitive alias/points-to analysis is much more precise
- ...but also much more expensive
- Flow-insensitive alias analysis is much faster
Example

- Assume the program
  
  ```
  p = &x;
p = &y;
*p = &z;
  ```

- Flow-sensitive analysis – solution per program point
  
  ```
  p = &x;      // (p, {&x})
p = &y;      // (p, {&y})
*p = &z;   // (p, {&y}), (y, {&z})
  ```

- Flow-insensitive analysis – one solution
  
  ```
  (p, {&x, &y})
(x, {&z})
(y, {&z})
  ```
A simple calculus

\[
T ::= T \to T \mid \text{Nat} \mid \text{Bool} \mid \text{Unit} \mid \text{Ref}\ T
\]

\[
e ::= x \mid n \mid \text{true} \mid \text{false} \mid () \mid e; e \mid \lambda x : T. e \mid e\ e \mid \text{let } x = e \text{ in } e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{ref } e \mid !e \mid e := e
\]
Type system

\[
\begin{align*}
\text{[T-VAR]} & \quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \\
\text{[T-NAT]} & \quad \frac{}{\Gamma \vdash n : Nat} \\
\text{[T-TRUE]} & \quad \frac{}{\Gamma \vdash \text{true} : Bool} \\
\text{[T-FALSE]} & \quad \frac{}{\Gamma \vdash \text{false} : Bool} \\
\text{[T-UNIT]} & \quad \frac{}{\Gamma \vdash () : Unit} \\
\text{[T-SEQ]} & \quad \frac{\Gamma \vdash e_1 : Unit \quad \Gamma \vdash e_2 : T}{\Gamma \vdash (e_1; e_2) : T} \\
\text{[T-LAM]} & \quad \frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \lambda x : T.e : T \rightarrow T'} \\
\text{[T-APP]} & \quad \frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash (e_1 e_2) : T'}
\end{align*}
\]
Type system (cont’d)

\[
\begin{align*}
\text{[T-LET]} & \quad \Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2 \\
& \quad \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2 \\
\text{[T-IF]} & \quad \Gamma \vdash e : Bool \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T \\
& \quad \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T \\
\text{[T-REF]} & \quad \Gamma \vdash e : T \\
& \quad \Gamma \vdash \text{ref } e : \text{Ref } T \\
\text{[T-DEREF]} & \quad \Gamma \vdash e : \text{Ref } T \\
& \quad \Gamma \vdash !e : T \\
\text{[T-ASSIGN]} & \quad \Gamma \vdash e_1 : \text{Ref } T \quad \Gamma \vdash e_2 : T \\
& \quad \Gamma \vdash e_1 := e_2 : \text{Unit}
\end{align*}
\]
Label flow analysis

- A way to compute points-to information
- We extend references with labels
  - $e ::= \ldots | \text{ref}^r e | \ldots$
  - A label $r$ identifies this particular allocation instruction
    - Like `malloc@42` identifies a point in the program
    - Drawn from a finite set of labels
  - For now, the programmers add these labels
- Goal of points-to analysis: find the set of labels a pointer may refer to
  - For example:
    
    ```
    let x = \text{ref}^{R_x} 0 \text{ in}
    let y = x \text{ in}
    y := 3 (* y may point to \{R_x\} *)
    ```
Type-based alias analysis

- We will build an alias analysis using the type system
  - Similar to OCaml’s *type inference*
- We use *labeled types* in the analysis
  - Extend reference types with labels: \( T ::= \ldots | \text{Ref}^\ell T | \ldots \)
  - To find the location at a pointer dereference \(!e\) or assignment \(e := \ldots\)
    - Find the type \(T\) of \(e\) (which must be a reference)
    - We look at the reference type to decide which location might be accessed
Type system (with labels)

\[
[T-\text{REF}] \quad \frac{\Gamma \vdash e : T}{\Gamma \vdash \text{ref}^r e : \text{Ref}^r T}
\]

\[
[T-\text{DEREF}] \quad \frac{\Gamma \vdash e : \text{Ref}^r T}{\Gamma \vdash !e : T}
\]

\[
[T-\text{ASSIGN}] \quad \frac{\Gamma \vdash e_1 : \text{Ref}^r T \\
\Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \text{Unit}}
\]
Example

- In the previous program
  
  ```latex
  \textbf{let} \ x = \text{ref}^{\text{R}_x} \ 0 \ \textbf{in}
  \textbf{let} \ y = x \ \textbf{in}
  y := 3
  ```

- $x$ has type $\text{Ref}^{\text{R}_x} \ \text{Nat}$
- $y$ has the same type as $x$
- Therefore, at the assignment expression, we know which location $y$ points to
Another example

- Consider the program

  ```plaintext
  let x = ref^R 1 in
  let y = ref^R 2 in
  let w = ref^Rw 0 in
  let z = if true then x else y in
  z := 3
  ```

- Here, x and y both have type Ref^R Nat
  - They must have the same type because of the if

- At assignment, we write to location R
  - We do not know which location this is exactly, x or y
  - But we know it cannot affect w
And another example

- Another program

```plaintext
let x = ref R 0 in
let y = ref R y x in
let z = ref R 2 in
y := z
```

- Both `x` and `z` have the same label
  - They must have the same type because of the pointed type of `y`
- We do not know whether `y` points to `x` or `y`
Things to notice

- We have a finite set of labels
  - At most one label for each occurrence of a ref in the program
  - A label may represent more than one run-time locations

- Whenever two labels “meet” in the type system, they must be the same
  - Can you see where this happens in the type-rules?

- The system is flow-insensitive
  - Types don’t change after assignment
Type inference

- In practice, the programmer does not write the labels
  - We need to infer them
- Given an unlabeled program that satisfies the standard type system, is there a labeling that satisfies the labeled type system?
  - That labeling is the analysis result
Checking vs. inference

- **Type checking**
  - The programmer annotates the program with types
  - Typing checks that the annotations are correct
  - It is “obvious” how to check

- **Type inference**
  - The programmer does not annotate the program
  - Typing tries to discover correct types
  - It is not “obvious”, requires more work to check

- **Consider the type-system of C**
  - C requires type annotations only at function types and local variable declarations
    - $3 + 4$ does not need a type annotation
  - Trade-off: programmer annotations vs. computed types
A type inference algorithm

- A standard approach in type inference
  - Type the program by introducing *variables* at any point when an annotation is missing
    - We will use *label variables* \( \rho \) here
    - Now \( r \) may be either a constant \( R \) or a variable \( \rho \)

- Typing the unlabeled program does two things
  - Introduces label variables in all \( \text{Ref} \) types
  - Creates *constraints* among labels

- Solve the constraints to find a labeling
  - No solution means no valid labeling: type error
  - Alias analysis solution always exists: everything aliases
Step 1: Introduce labels

- Problem 1: What label to assign to the reference at \([T{\text{-Ref}}]\)?
- Solution: Introduce a fresh, unknown variable

\[
\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{ref } e : \textit{Ref}^\rho T}
\]

- Why a variable and not a constant?
Step 1: Introduce labels (cont’d)

- Problem 2: What type to give to function arguments?
  - Type language \( T \) uses labeled reference types \( \text{Ref}^p \ T \)
  - But the programmer uses unlabeled types \( \text{Ref} \ T \)

- Solution:
  - Use two type languages
    - Standard \( S ::= S \rightarrow S \mid \text{Nat} \mid \text{Bool} \mid \text{Unit} \mid \text{Ref} \ S \)
    - Labeled \( T ::= T \rightarrow T \mid \text{Nat} \mid \text{Bool} \mid \text{Unit} \mid \text{Ref}^p \ T \)
  - Annotate type \( S \) with fresh labels to get a \( T \)
    - We write this as \( T = \text{fresh}(S) \)

\[
\Gamma, x : T \vdash e : T' \\
\frac{T = \text{fresh}(S)}{
\Gamma \vdash \lambda x : S.e : T \rightarrow T'}
\]
Step 2: Generate constraints

- Problem 3: Some rules implicitly require types to be equal
- Solution: Make this explicit using equality constraints
  - We write equality constraints as premises $T_1 = T_2$
  - Each such premise is not checked, instead produces a constraint
  - We solve all generated constraints together after typing

- Rule $[T\text{-If}]$ requires both branches to have the same type

\[
\begin{align*}
\Gamma & \vdash e : Bool \\
\Gamma & \vdash e_1 : T_1 \\
\Gamma & \vdash e_2 : T_2 \\
\end{align*}
\]

\[
\frac{T_1 = T_2}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T_1}
\]
Step 2: Generate constraints (cont’d)

- Rule $[\text{T-Assign}]$ requires that the assigned value has the same type as the pointer

$$
\Gamma \vdash e_1 : \text{Ref } T_1 \\
\Gamma \vdash e_2 : T_2 \\
T_1 = T_2 \\
\Gamma \vdash e_1 := e_2 : \text{Unit}
$$

- We assume that $e_1$ always has a pointer type
  - That is always true
  - We assume the program typechecks with standard types
Step 2: Generate constraints (cont’d)

- Rule $[\text{T-App}]$ requires the formal and actual arguments to have the same type

$$
\Gamma \vdash e_1 : T_1 \rightarrow T' \\
\Gamma \vdash e_2 : T_2 \\
T_1 = T_2 \\
\frac{[\text{T-App}]}{\Gamma \vdash (e_1 \ e_2) : T'}
$$

- Again, we assume $e_1$ has a function type
  - As before, this is always true
  - Because the program typechecks with standard types
Step 3: Solve the constraints

- After applying the type rules, we are left with a set of equality constraints
  - $T_1 = T_2$

- We solve these constraints using rewriting

- Each rewriting step simplifies a constraint into simpler constraints

- $C \Rightarrow C'$ rewrites the set $C$ of all constraints to constraints $C'$
Step 3: Solve the constraints (cont’d)

- $C \cup \{Nat = Nat\} \Rightarrow C$
- $C \cup \{Bool = Bool\} \Rightarrow C$
- $C \cup \{Unit = Unit\} \Rightarrow C$
- $C \cup \{T_1 \rightarrow T_2 = T'_1 \rightarrow T'_2\} \Rightarrow C \cup \{T_1 = T'_1\} \cup \{T_2 = T'_2\}$
- $C \cup \{Ref^{p_1} T_1 = Ref^{p_2} T_2\} \Rightarrow C \cup \{T_1 = T_2\} \cup \{\rho_1 = \rho_2\}$
- $C \cup \{\text{mismatched constructors}\} \Rightarrow \text{error}$
  - Cannot happen if we start with a program that typechecks with standard types

- This algorithm always terminates
- When no further reduction applies, we have only label equalities
Last step: Use solution to add constants

- Compute the sets of labels that are equal
  - Using union-find
- Create a constant label $R$ for each equivalence class of label variables
- Two pointers alias if their types refer to the same constant label
Example

Program

```ocaml
let x = ref 1 in
let y = ref 2 in
let z = ref 3 in
let w = if true then x else y in
w := 42
```

Variable types:

- `x : Ref^a Nat`
- `y : Ref^b Nat`
- `z : Ref^c Nat`
- `w : Ref^a Nat`

- Typing annotates each `ref` expression with a variable `a`, `b`, `c`
- Typing the if creates equality constraint `Ref^a Nat = Ref^b Nat`
- Solving the constraint gives `a = b`
- Two equivalence classes: `{a, b}` and `{c}`
  - Create two constants `R_1` and `R_2` for the equivalence classes
Example (cont’d)

Annotated program

```plaintext
let x = ref^{R_1} 1 in
let y = ref^{R_1} 2 in
let z = ref^{R_2} 3 in
let w = if true then x else y in
  w := 42
```

Variable types:

- $x : \text{Ref}^{R_1} \text{Nat}$
- $y : \text{Ref}^{R_1} \text{Nat}$
- $z : \text{Ref}^{R_2} \text{Nat}$
- $w : \text{Ref}^{R_1} \text{Nat}$

- The assignment writes to one of the locations labeled by $R_1$
- Result: $x, y$ and $w$ may alias either of the first two allocated locations, but $z$ cannot
  - May alias: their types have the same location label
Steensgaard’s Analysis

- Flow-insensitive
- Inter-procedural
  - Can analyze multiple functions together
- Context-insensitive
  - Does not discriminate between different calls to the same function
- Unification-based
  - Analysis named after Bjarne Steensgaard (1996)
  - In practice: implementation for C handles type casts, etc.
- Properties
  - Very scalable
    - What is its complexity?
  - Imprecise