Lecture 19: Alias analysis

Subtyping

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Programming Languages

Based on slides by Jeff Foster

Last time

- Label-flow analysis
  - Assign a label at every "interesting" program point (pointers)
  - Aliasing question: does label R_1 "flow" to label R_2 at runtime?
- Type-based label-flow (for pointers)
  - Annotate types with labels
  - Type-checking is flow checking
- An inference system
  - Type system creates "fresh" label variables
  - Typing creates constraints among variables
  - Constraint solution gives aliasing information
    - ★ We used unification to solve constraints

Limitation of unification

- Unification creates "backwards flow" of labels
- When x and y both alias z, they alias each other too
- For example

```
let x = ref 1 in
let y = ref 2 in
let z = if true then x else y in
  x := 42;
  y := 0;
```

- Unification gives

$$\mathsf{x} : Ref^{R}\ Nat$$
$$\mathsf{y} : Ref^{R}\ Nat$$
$$\mathsf{z} : Ref^{R}\ Nat$$

Subtyping

- We can solve this problem using *subtyping*
  - Each label variable represents a *set* of labels
    - In unification, a variable could only stand for one label
  - We write $[\alpha]$ for the set of labels represented by α
    - Trivially, $[R] = \{R\}$ for any constant R
- For example, assume
  - x has type $Ref^{\alpha}\ Nat$
  - $[\alpha] = \{R_1, R_2\}$
  - Then x may point to either location R_1 or location R_2
    - Again, labels R_1 and R_2 are static approximations, they may refer to many runtime locations

Labels on references

- Labeling is slightly different
  - We assume each allocation has a unique constant label
    - Generate a fresh one for each syntactic occurence
  - Add a fresh variable on each reference type and generate a *subtyping* constraint between constant and variable
    - $\alpha_1 \leq \alpha_2$ means $[\alpha_1] \subseteq [\alpha_2]$

$$[\text{T-Ref}]\frac{\begin{array}{c}\Gamma \vdash e : T \\ R \leq \alpha \\ R - \text{fresh} \qquad \alpha - \text{fresh}\end{array}}{\Gamma \vdash \mathsf{ref}^R\ e : Ref^{\alpha}\ T}$$

Subtype inference

- The same approach as before
  - Visit the AST, generate constraints
  - Constraints allow subsets, instead of equalities
- We could change all rules that generate constraints to allow inequalities
  - For example

$$\frac{\begin{array}{c}\Gamma \vdash e : Bool \\ \Gamma \vdash e_1 : Ref^{\rho_1}\ T \qquad \Gamma \vdash e_2 : Ref^{\rho_1}\ T \\ \rho_1 \leq \rho \qquad\qquad\qquad \rho_2 \leq \rho\end{array}}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : Ref^{\rho}\ T}$$

Subtyping constraints

- We need to generalize to arbitrary types
  - Think of types as representing sets of values
    - For example Nat represents the set of natural numbers
    - So, $Ref^{\rho}\ Nat$ represents the sets of pointers to integers labeled with $[\rho]$
  - Extend $\leq$ to a relation $T \leq T$ on types

$$\frac{}{Nat \leq Nat} \qquad \frac{\rho_1 \leq \rho_2 \qquad Nat \leq Nat}{Ref^{\rho_1}\ Nat \leq Ref^{\rho_2}\ Nat}$$

Subsumption

- Instead of modifying all rules with constraints, add one more typing rule (remember subtyping from λ-calculus)

$$\frac{\Gamma \vdash e : T \qquad T \leq T'}{\Gamma \vdash e : T'}$$

- Like normal subtyping: we can use a supertype anywhere a subtype is expected

Example

```
let x = ref 0 in                          // x : Ref^α Nat
let y = ref 1 in                          // y : Ref^β Nat
let z = if true then x else y in          // z : Ref^γ Nat
  x := 42
```

- Types of x and y must match as conditional

$$\dfrac{\Gamma \vdash x : \mathit{Ref}^{\alpha}\ \mathit{Nat} \qquad \dfrac{\alpha \leq \gamma}{\mathit{Ref}^{\alpha}\ \mathit{Nat} \leq \mathit{Ref}^{\gamma}\ \mathit{Nat}}}{\Gamma \vdash x : \mathit{Ref}^{\gamma}\ \mathit{Nat}}$$

- So, we have $z : \mathit{Ref}^{\gamma}\ \mathit{Nat}$ with $\alpha \leq \gamma$ and $\beta \leq \gamma$
  - And we can pick $[\alpha] = \{R_x\}, [\beta] = \{R_y\}, [\gamma] = \{R_x, R_y\}$

Subtyping references

- Let's try to generalize to arbitrary types

$$\frac{\rho_1 \leq \rho_2 \quad T_1 \leq T_2}{Ref^{\rho_1}\ T_1 \leq Ref^{\rho_2}\ T_2}$$

- This is broken

let x = ref^{R_x} (ref^{R_0} 0) **in**	// x : $Ref^{\alpha}\ Ref^{\beta}\ Nat$, $R_0 \leq \beta$
let y = x **in**	// y : $Ref^{\gamma}\ Ref^{\delta}\ Nat$, $\beta \leq \delta$
y := ref^{R_1} 1;	// $R_1 \leq\leq \delta$
!!x := 3	// deref of β

- We can pick $[\beta] = \{R_0\}$, $[\delta] = \{R_0, R_1\}$
  - Then writing through β doesn't write R_1

Aliasing

- Through subtyping, we have multiple names for the same memory location
  - They have different types
  - We can write different types on the same memory location
- Solution: require equality under a ref
  - We saw this before: subtyping and references
  - We can write $T_1 = T_2$ as $T_1 \leq T_2$ and $T_2 \leq T_1$

$$\frac{\rho_1 \leq \rho_2 \qquad T_1 \leq T_2 \qquad T_2 \leq T_1}{Ref^{\rho_1}\ T_1 \leq Ref^{\rho_2}\ T_2}$$

Subtyping on function types

- When is a function type $T_1 \to T_2$ subtype of another function type $T'_1 \to T'_2$?
- Similar to standard subtyping
  - Contravariant on the argument type
  - Covariant on the result type

$$\frac{T'_1 \leq T_1 \qquad T_2 \leq T'_2}{T_1 \to T_2 \leq T'_1 \to T'_2}$$

- Example: we can always use a function that returns a pointer to $\{R_1\}$ as if it could return $\{R_1, R_2\}$
- Example: if a function expects a pointer to $\{R_1, R_2\}$ we can always give it a pointer to $\{R_1\}$

Type system

- Typing is similar, generates $\leq$ instead of $=$ constraints

$$[\text{T-VAR}]\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad [\text{T-NAT}]\ \frac{}{\Gamma \vdash n : \mathit{Nat}}$$

$$[\text{T-TRUE}]\ \frac{}{\Gamma \vdash \mathsf{true} : \mathit{Bool}} \qquad [\text{T-FALSE}]\ \frac{}{\Gamma \vdash \mathsf{false} : \mathit{Bool}}$$

$$[\text{T-UNIT}]\ \frac{}{\Gamma \vdash () : \mathit{Unit}} \qquad [\text{T-SEQ}]\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \mathit{Unit} \\ \Gamma \vdash e_2 : T\end{array}}{\Gamma \vdash (e_1; e_2) : T}$$

$$[\text{T-LAM}]\ \frac{\begin{array}{c}\Gamma, x : S \vdash e : T' \\ T = \text{fresh}(S)\end{array}}{\Gamma \vdash \lambda x : S.e : T \to T'} \qquad [\text{T-APP}]\ \frac{\begin{array}{c}\Gamma \vdash e_1 : T \to T' \\ \Gamma \vdash e_2 : T\end{array}}{\Gamma \vdash (e_1\ e_2) : T'}$$

Type system (cont'd)

$$[\textsc{T-If}]\ \frac{\Gamma \vdash e : \mathit{Bool} \qquad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T}$$

$$[\textsc{T-Let}]\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T_2}$$

$$[\textsc{T-Ref}]\ \frac{\Gamma \vdash e : T \quad R \leq \alpha \qquad R - \mathrm{fresh} \quad \alpha - \mathrm{fresh}}{\Gamma \vdash \mathsf{ref}^R\ e : \mathit{Ref}^\alpha\ T}$$

$$[\textsc{T-Deref}]\ \frac{\Gamma \vdash e : \mathit{Ref}^\alpha\ T}{\Gamma \vdash !e : T}$$

$$[\textsc{T-Assign}]\ \frac{\Gamma \vdash e_1 : \mathit{Ref}^\alpha\ T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \mathit{Unit}}$$

$$[\textsc{T-Sub}]\ \frac{\Gamma \vdash e : T_1 \qquad T_1 \leq T_2}{\Gamma \vdash e : T_2}$$

Subtyping relation

- In unification, we simplify $T_1 = T_2$ constraints to get $\rho_1 = \rho_2$ constraints
- We can use the subtyping relation $T_1 \leq T_2$ to do the same

$$[\textsc{S-Nat}]\ \frac{T_1' \leq T_1 \qquad T_2 \leq T_2'}{T_1 \to T_2 \leq T_1' \to T_2'}$$

$$[\textsc{S-Nat}]\ \frac{}{\mathit{Nat} \leq \mathit{Nat}} \qquad [\textsc{S-Bool}]\ \frac{}{\mathit{Bool} \leq \mathit{Bool}}$$

$$[\textsc{S-Unit}]\ \frac{}{\mathit{Unit} \leq \mathit{Unit}} \qquad [\textsc{S-Ref}]\ \frac{\begin{array}{c}\rho_1 \leq \rho_2 \\ T_1 \leq T_2 \qquad T_2 \leq T_1\end{array}}{\mathit{Ref}^{\rho_1}\ T_1 \leq \mathit{Ref}^{\rho_2}\ T_2}$$

The problem: subsumption

- We can apply subsumption at any time
  - Makes it hard to develop a deterministic algorithm
  - Type checking is not *syntax-driven*
- Fortunately, not many choices
  - For each expression e we need to decide
    - ⋆ Do we apply the "regular" syntax-driven rule for e?
    - ⋆ or do we apply subsumption (and how many times)?

Getting rid of subsumtion

- Lemma: Multiple sequential uses of subsumption can be collapsed into a single use
  - Proof: transitivity of $\leq$
- We need at most one application of subsumption after typing an expression
- We can get rid of that one application
  - Integrate it into the rest of the rules
  - Each rule is the syntax-driven typing, plus a subsumption

Getting rid of subsumption (cont'd)

- All rules that introduced $T_1 = T_2$ constraints in unification, now introduce subtyping $T_1 \leq T_2$

$$[\text{T-App}]\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \to T' \\ \Gamma \vdash e_2 : T_2 \\ T_2 \leq T_1\end{array}}{\Gamma \vdash (e_1\ e_2) : T'}$$

$$[\text{T-If}]\frac{\begin{array}{c}\Gamma \vdash e : Bool \\ \Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2 \\ T_1 \leq T \qquad\qquad T_2 \leq T\end{array}}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T}$$

- Etc, for the other rules
- We are left with an algorithmic, syntax-directed type system

Solving the constraints

- Solving computes transitive closure of $\rho \leq \rho'$
- As in unification, use a rewriting system to simplify constraints
- Except we have already solved the structural part and only have $r \leq \rho_1$ constraints left
  - If $\{\rho_1 \leq \rho_2\}$ and $\{\rho_2 \leq \rho_3\}$ then add $\{\rho_1 \leq \rho_3\}$
- Repeat until no new edges can be added
- At most $O(N^2)$
- Points-to set $[\rho]$ is then $[\rho] = \{R \mid R \leq \rho\}$

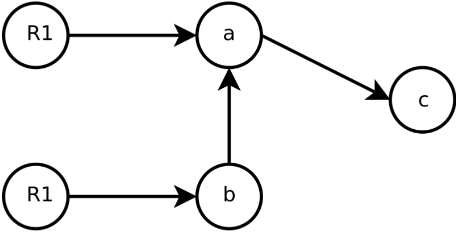
Graph reachability

$$R_1 \leq a$$

$$R_2 \leq b$$

$$a \leq c$$

$$b \leq a$$



Andersen's analysis

- Flow-insensitive
- Context-insensitive
- Subtyping-based
- Properties
  - Still very scalable in practice
  - Much less coarse than Steensgaard's analysis
  - Precision can still be improved