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

```plaintext
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

\[
\begin{align*}
x & : \text{Ref}^R \text{Nat} \\
y & : \text{Ref}^R \text{Nat} \\
z & : \text{Ref}^R \text{Nat}
\end{align*}
\]
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 \(\alpha\)
    - Trivially, \([R] = \{R\}\) for any constant \(R\)

- For example, assume
  - \(x\) has type \(\text{Ref}^\alpha \text{Nat}\)
  - \([\alpha] = \{R_1, R_2\}\)
  - 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 occurrence
  - Add a fresh variable on each reference type and generate a subtyping constraint between constant and variable
    - $\alpha_1 \leq \alpha_2$ means $[\alpha_1] \subseteq [\alpha_2]$

\[
\Gamma \vdash e : T \\
R \leq \alpha \\
[T-\text{Ref}] \frac{R - \text{fresh}}{\alpha - \text{fresh}} \quad \frac{\Gamma \vdash \text{ref}^R e : \text{Ref}^\alpha T}
\]
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

\[
\Gamma \vdash e : \text{Bool} \\
\Gamma \vdash e_1 : \text{Ref}^{\rho_1} T \\
\Gamma \vdash e_2 : \text{Ref}^{\rho_1} T \\
\rho_1 \leq \rho \\
\rho_2 \leq \rho \\
\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \text{Ref}^\rho T
\]
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, $Ref^p Nat$ represents the sets of pointers to integers labeled with $[\rho]$  
  - Extend $\leq$ to a relation $T \leq T$ on types

\[
\begin{align*}
Nat & \leq Nat \\
\rho_1 \leq \rho_2 & \quad Nat \leq Nat \\
Ref^{\rho_1} Nat & \leq Ref^{\rho_2} Nat
\end{align*}
\]
Subsumption

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

\[
\frac{\Gamma \vdash e : T \quad T \leq T'}{\Gamma \vdash e : T'}
\]

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

```ocaml
let x = ref 0 in // x : Ref^\alpha Nat
let y = ref 1 in // y : Ref^\beta Nat
let z = if true then x else y in // z : Ref^\gamma Nat
  x := 42
```

- Types of \( x \) and \( y \) must match as conditional

\[
\frac{\alpha \leq \gamma}{\Gamma \vdash x : Ref^\alpha Nat}
\frac{Ref^\alpha Nat \leq Ref^\gamma Nat}{\Gamma \vdash x : Ref^\gamma Nat}
\]

- So, we have \( z : Ref^\gamma Nat \) 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

\[
\begin{align*}
\rho_1 &\leq \rho_2 \\
T_1 &\leq T_2
\end{align*}
\]

\[
Ref^{\rho_1} T_1 \leq Ref^{\rho_2} T_2
\]

- This is broken

```haskell
let x = ref^{R_0} (ref^{R_0} 0) in
let y = x in
  y := ref^{R_1} 1;
  !!x := 3
// x : Ref^α Ref^β Nat, R_0 \leq β
// y : Ref^γ Ref^δ Nat, β \leq δ
// R_1 \leq= δ
// deref of β
```

- We can pick \([β] = \{ R_0 \}, [δ] = \{ R_0, R_1 \}\)
  - 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 $T_2 \leq T_1$

\[
\frac{\rho_1 \leq \rho_2}{Ref^{\rho_1}} \frac{T_1 \leq T_2}{Ref^{\rho_2}} \frac{T_2 \leq T_1}{Ref^{\rho_2}}
\]
Subtyping on function types

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

\[
\frac{T'_1 \leq T_1 \quad T_2 \leq T'_2}{T_1 \rightarrow T_2 \leq T'_1 \rightarrow T'_2}
\]

- 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

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

\[
\begin{align*}
[T-\text{If}] & : \quad \Gamma \vdash e : \text{Bool} \\
& \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T \\
& \quad \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T
\end{align*}
\]

\[
\begin{align*}
[T-\text{Let}] & : \quad \Gamma, x : T_1 \vdash e_2 : T_2 \\
& \quad \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2
\end{align*}
\]

\[
\begin{align*}
[T-\text{Ref}] & : \quad \Gamma \vdash e, R \leq \alpha, R - \text{fresh, } \alpha - \text{fresh} \\
& \quad \Gamma \vdash \text{ref}^R e : \text{Ref}^\alpha T
\end{align*}
\]

\[
\begin{align*}
[T-\text{Deref}] & : \quad \Gamma \vdash e : \text{Ref}^\alpha T \\
& \quad \Gamma \vdash !e : T
\end{align*}
\]

\[
\begin{align*}
[T-\text{Assign}] & : \quad \Gamma \vdash e_1 : \text{Ref}^\alpha T \\
& \quad \Gamma \vdash e_2 : T \\
& \quad \Gamma \vdash e_1 := e_2 : \text{Unit}
\end{align*}
\]

\[
\begin{align*}
[T-\text{Sub}] & : \quad \Gamma \vdash e : T_1 \\
& \quad T_1 \leq T_2 \\
& \quad \Gamma \vdash e : T_2
\end{align*}
\]
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.

\[
[S-\text{Nat}] \quad T_1' \leq T_1 \quad T_2 \leq T_2' \quad \frac{T_1 \rightarrow T_2 \leq T_1' \rightarrow T_2'}{}
\]

\[
[S-\text{Nat}] \quad \text{Nat} \leq \text{Nat} \\
[S-\text{Bool}] \quad \text{Bool} \leq \text{Bool}
\]

\[
[S-\text{Unit}] \quad \text{Unit} \leq \text{Unit} \\
[S-\text{Ref}] \quad \frac{T_1 \leq T_2 \quad T_2 \leq T_1}{} \\
\]

Pratikakis (CSD)
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 subsumption

- 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 $T_1 \leq T_2$

\[
\begin{align*}
\Gamma \vdash e_1 : T_1 \rightarrow T' \\
\Gamma \vdash e_2 : T_2 \\
\frac{T_2 \leq T_1}{\Gamma \vdash (e_1 \ e_2) : T'} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : Bool \\
\Gamma \vdash e_1 : T_1 \\ 
\Gamma \vdash e_2 : T_2 \\
\frac{T_1 \leq T \quad T_2 \leq T}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T} \\
\end{align*}
\]

- 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