Set Constraints

CS242
Lecture 15
Approaches to Proving Properties of Programs

Automatic, Low complexity
Simply Typed Lambda Calculus

Automatic, High complexity
Static Analysis

Automatic or Semi-automatic Often undecidable
Invariant Inference

Manual, Undecidable
Dependent Types
Closure Analysis: The Problem

• A call graph is a graph where
  • The nodes are function (method) names
  • There is a directed edge \((f,g)\) if \(f\) may call \(g\)

• Call graphs can be overestimates
  • If \(f\) may call \(g\) at run time, there must be an edge \((f,g)\) in the call graph
  • If \(f\) cannot call \(g\) at run time, there is no requirement on the graph

• Call graphs are used heavily in implementations of programming languages
Recall: Untyped Lambda Calculus

\[ e \rightarrow x \mid \lambda x.e \mid e\ e \]
A Definition

• Assume all bound variables are unique
  • So a bound variable uniquely identifies a function
  • Can be done by renaming variables

• For each application $e_1 e_2$, what is the set of lambda terms $L(e_1)$ to which $e_1$ may evaluate?
  • $L(\ldots)$ is a set of static, or syntactic, lambdas
  • $L(\ldots)$ defines a call graph
    • the set of functions that may be called by an application
A More General Definition

• To compute $L(...)$ for applications, we must compute it for every expression.

• Define:
  $L(e)$ is the set of syntactic lambda abstractions to which $e$ may evaluate

• The problem is to compute $L(e)$ for every expression $e$
Defining $\text{L}(\ldots)$

$\lambda x. e$

$\lambda x. e \subseteq \text{L}(\lambda x. e)$

$e_1 e_2$

for each $\lambda x. e \subseteq \text{L}(e_1)$

$\text{L}(e_2) \subseteq \text{L}(x)$

$\text{L}(e) \subseteq \text{L}(e_1 e_2)$

The actual argument of the call flows to the formal argument

The value of the application includes the value of the function body
Rephrasing the Constraints with $\subseteq$

The following constraints have the same least solution as the original constraints:

\[
\begin{align*}
\lambda x. e & \subseteq L(\lambda x. e) \\
e_1 \quad e_2 & \\
\lambda x. e_0 & \subseteq L(e_1) \Rightarrow (L(e_2) \subseteq L(x) \land L(e_0) \subseteq L(e_1 \ e_2))
\end{align*}
\]

Note: Each $L(e)$ is a set variable
Each $\lambda x. e$ is a constant
Example \(((\lambda x.x) \ (\lambda y.y)) \ (\lambda z.z)\)

\[
\begin{align*}
\lambda x.x & \subseteq L(\lambda x.x) \\
\lambda y.y & \subseteq L(\lambda y.y) \\
\lambda z.z & \subseteq L(\lambda z.z) \\
L(\lambda y.y) & \subseteq L(x) \\
L(x) & \subseteq L((\lambda x.x) \ (\lambda y.y)) \\
L(\lambda z.z) & \subseteq L(y) \\
L(y) & \subseteq L(((\lambda x.x) \ (\lambda y.y)) \ (\lambda z.z))
\end{align*}
\]

Solution:

\[
\begin{align*}
L(\lambda x.x) &= \lambda x.x \\
L(\lambda y.y) &= \lambda y.y \\
L(\lambda z.z) &= \lambda z.z \\
L(\lambda y.y) &= L(x) = L((\lambda x.x) \ (\lambda y.y)) \\
L(\lambda z.z) &= L(y) = L(((\lambda x.x) \ (\lambda y.y)) \ (\lambda z.z))
\end{align*}
\]
The Example $((\lambda x.x) (\lambda y.y)) (\lambda z.z)$ with Graphs
The Solution for \(((\lambda x.x) (\lambda y.y)) (\lambda z.z)\)

The solution is given by the edges whose source is a lambda.
Set Constraints

• A finite set of *constructors*
  
  \[ a, b, c, f, g, h \in C \]

• Each constructor \( c \) has an *arity* \( a(c) \)
  • Can be 0

• Terms
  • \( T = \{ f(t_1, \ldots, t_{a(f)}) \mid f \in C, t_i \in T \} \)
Set Constraints

Constraints: $L \subseteq R$ or $c \subseteq R \Rightarrow L' \subseteq R'$
Solving

\[ S, L_1 \cup L_2 \subseteq R \quad \rightarrow \quad S, L_1 \cup L_2 \subseteq R, L_1 \subseteq R, L_2 \subseteq R \]
\[ S, L \subseteq R_1 \cap R_2 \quad \rightarrow \quad S, L \subseteq R_1 \cap R_2, L \subseteq R_1, L \subseteq R_2 \]
\[ S, c(L_1,\ldots,L_n) \subseteq c(R_1,\ldots,R_n) \quad \rightarrow \quad S, c(L_1,\ldots,L_n) \subseteq c(R_1,\ldots,R_n), L_1 \subseteq R_1, \ldots, L_n \subseteq R_n \]
\[ S, c \subseteq R \Rightarrow L' \subseteq R', c \subseteq R \quad \rightarrow \quad S, c \subseteq R \Rightarrow L' \subseteq R', c \subseteq R, L' \subseteq R' \]
\[ S, L \subseteq x, x \subseteq R \quad \rightarrow \quad S, L \subseteq x, x \subseteq R, L \subseteq R \]

No solutions if \( c(...) \subseteq 0, \ 1 \subseteq c(...), \text{ or } c(...) \subseteq d(...) \)
Add Integers ...

\[ e \rightarrow x \mid \lambda x. e \mid e \ e \mid i \]
Extend $L(\ldots)$ With Integers

$\lambda x. e$

$\lambda x. e \subseteq L(\lambda x. e)$

$i \quad i \subseteq L(i)$

Idea: Treat integers like lambdas and track their flow.

$e_1 \quad e_2$

for each $\lambda x. e \subseteq L(e_1)$

$L(e_2) \subseteq L(x)$

$L(e) \subseteq L(e_1 \quad e_2)$
Application: Type Inference!

\[ \lambda x. e \]
\[ \lambda x. e \subseteq L(\lambda x. e) \]

\[ i \quad i \subseteq L(i) \]

\[ e_1 \quad e_2 \]
for each \[ \lambda x. e \subseteq L(e_1) \]
\[ L(e_2) \subseteq L(x) \]
\[ L(e) \subseteq L(e_1 \ e_2) \]
\[ L(e_1) \subseteq \{ \lambda x. e \mid \lambda x. e \text{ is a lambda abstraction in the program} \} \]
An Example \(((\lambda x.x) \ (\lambda y.y)) \ (\lambda z.z)\)
An Example \(((\lambda x. x) \ 1) \ (\lambda z. z)\)

There is a path from an integer constant to the set of lambdas – a type error.
Discussion

• Idea: For each application $e_1 \, e_2$, check if $i \subseteq L(e_1)$
  • If yes, it’s a type error

• More general than simply typed lambda calculus
  • Intuition: Relax = constraints to $\subseteq$

• Note every pure lambda (with no integers) has a type
Summary So Far

- Set constraints use subset relationships instead of equality

- Natural interpretation as a graph
  - Nodes are sets
  - Directed edges are inclusion constraints
  - Solving the constraints = adding edges to the graph
  - Dynamic transitive closure $O(n^3)$

- Many applications of *closure analysis* in functional programming
  - Often the first analysis to be done
The Next Step

• So far we’ve only considered sets of atoms
  • Sets where the elements have no structure

• Now consider sets where the elements can be sets of data types
  • E.g., cons(A,B) is the cross product of cons(a,b) for every a in A and b in B
Recall: Simple Type Inference Rules

**[Var]**

\[ A, x: \alpha_x \vdash x : \alpha_x \]

**[Abs]**

\[ A, x: \alpha_x \vdash \lambda x : \alpha_x \cdot e : \alpha_x \rightarrow t \]

\[ A \vdash \lambda x: \alpha_x \cdot e: \alpha_x \rightarrow t \]

\[ A \vdash e_1 : t \]

\[ A \vdash e_2 : t' \]

\[ A \vdash e_1 e_2 : \beta \]

\[ t = t' \rightarrow \beta \]
A Small Change

\[ A, x: \alpha_x \vdash x: \alpha_x \]

[Var]

\[ A, x: \alpha_x \vdash e: t \]

[App]

\[ t \subseteq t' \rightarrow \beta \]

\[ A \vdash e_1: t \]

\[ A \vdash e_2: t' \]

\[ A \vdash e_1 e_2: \beta \]

[Abs]

\[ A \vdash \lambda x. e: \alpha_x \rightarrow t \]
Recall: Solving the ( Equality ) Constraints

Apply the following rewrite rules until no new constraints can be added

\[
S, t = \alpha \quad \Rightarrow \quad S, t = \alpha, \alpha = t \quad [\text{Reflexivity}]
\]

\[
S, \alpha = t_1, \alpha = t_2 \quad \Rightarrow \quad S, \alpha = t_1, \alpha = t_2, t_1 = t_2 \quad [\text{Transitivity}]
\]

\[
S, t_1 \rightarrow t_2 = t_3 \rightarrow t_4 \Rightarrow \quad S, t_1 \rightarrow t_2 = t_3 \rightarrow t_4, t_1 = t_3, t_2 = t_4 \quad [\text{Structure}]
\]
Solving the Subset Constraints

Apply the following rewrite rules until no new constraints can be added

\[ S, t_1 \subseteq t_2, t_2 \subseteq t_3 \implies S, t_1 \subseteq t_2, t_2 \subseteq t_3, t_1 \subseteq t_3 \quad \text{[Transitivity]} \]

\[ S, t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4 \implies S, t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4, t_1 \rightarrow t_3, t_2 \rightarrow t_4 \quad \text{[Subtyping]} \]
What is the subtyping rule for $t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4$?
Start With Something Else ...

What is the subtyping rule for $\text{cons}(A,B) \subseteq \text{cons}(A',B')$?

\[ \text{cons}(A,B) \subseteq \text{cons}(A',B') \Rightarrow A \subseteq A' \land B \subseteq B' \]
What is the subtyping rule for $t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4$?

$t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4 \Rightarrow t_3 \subseteq t_1 \wedge t_2 \subseteq t_4$
Terminology

• We say that the function type constructor is
  • Contravariant in the first argument (the domain)
  • Covariant in the second argument (the range)

• Constructors in set constraints have fixed variance
  • Covariant, contravariant, or invariant in every argument position
  • Examples: \texttt{cons(+,+)} \rightarrow +

• Note that contravariance adds nothing to the graph representation
  • The directed edge is just added in the opposite direction for contravariant relationships
  • And in both directions for invariant relationships
Solving the Subset Constraints

Apply the following rewrite rules until no new constraints can be added

\[ S, t_1 \subseteq t_2, t_2 \subseteq t_3 \quad \Rightarrow \quad S, t_1 \subseteq t_2, t_2 \subseteq t_3, t_1 \subseteq t_3 \quad \text{[Transitivity]} \]

\[ S, t_1 \to t_2 \subseteq t_3 \to t_4 \quad \Rightarrow \quad S, t_1 \to t_2 \subseteq t_3 \to t_4, t_3 \subseteq t_1, t_2 \subseteq t_4 \quad \text{[Subtyping]} \]
Comparisons

• The lambda calculus with type equality

• The lambda calculus with subtyping

• Which is equivalent to closure analysis
Recall

\[ \lambda x. e \quad \lambda x. e \subseteq L(\lambda x. e) \]

\[ i \quad i \subseteq L(i) \]

\[ e_1 \ e_2 \]

**for each** \[ \lambda x. e \subseteq L(e_1) \]

\[ L(e_2) \subseteq L(x) \]

\[ L(e) \subseteq L(e_1 \ e_2) \]

\[ L(e_1) \subseteq \{ \lambda x. e \mid \lambda x. e \text{ is a lambda abstraction in the program} \} \]
The Analogy ...

\( e_1 \ e_2 \)

for each \( \lambda x. e \subseteq L(e_1) \)

\( L(e_2) \subseteq L(x) \)

\( L(e) \subseteq L(e_1 \ e_2) \)

\( L(e_1) \subseteq \{ \lambda x. e \mid \lambda x. e \text{ is a lambda abstraction in the program} \} \)

For each \( t_1 \rightarrow t_2 \subseteq t \)
We have \( t_1 \rightarrow t_2 \subseteq t' \rightarrow \beta \)
So \( t' \subseteq t_1 \) and \( t_2 \subseteq \beta \)

Observe \( L(e_2) \equiv t', L(x) \equiv t_1, L(e) \equiv t_2, L(e_1 \ e_2) \equiv \beta \).
Control Flow Graphs in OO Languages

• Consider a method call \( e_0.f(e_1,\ldots,e_n) \)

• To build a control-flow graph, we need to know which \( f \) methods may be called
  • Depends on the class of \( e_0 \) at runtime

• The problem:
  • For each expression, estimate the set of classes it could evaluate to at runtime
An OO Language

\[ \begin{align*}
P &::= C_1 \ldots C_n E \\
C &::= \text{class ClassId } M_1 \ldots M_n \\
M &::= \text{MId(Id) } E \\
E &::= \text{Id := E } | \ E.\text{MId(E) } | \ E;E | \ \text{new ClassId } | \ \text{if } E E E
\end{align*} \]
Example Program

class A

    foo(x) x.bar(x)

    bar(y) z := new B;  z.bar(y)

class B

    bar(w) w.bar(new B)
Constraints

\[ \text{id := e} \]
\[ C(e) \subseteq C(\text{id}) \]
\[ C(e) \subseteq C(\text{id := e}) \]
\[ e_1; e_2 \]
\[ C(e_2) \subseteq C(e_1; e_2) \]
\[ \text{new A} \]
\[ A \subseteq C(\text{new A}) \]
\[ \text{if } e_1 e_2 e_3 \]
\[ C(e_2) \subseteq C(\text{if } e_1 e_2 e_3) \]
\[ C(e_3) \subseteq C(\text{if } e_1 e_2 e_3) \]

\[ e_0.f(e_1) \]

for each class \( A \) with a method \( f(x) \) \( e \)
\[ A \subseteq C(e_0) \Rightarrow \]
\[ C(e_1) \subseteq C(x) \land \]
\[ C(e) \subseteq C(e_0.f(e_1)) \]
Example Program w/Constraints

class A
    foo(x) (new A).bar(x)
    bar(y) z := new B; z.bar(z)

class B
    bar(w) w.bar(new B)

A ⊆ C(new A)
C(x) ⊆ C(y)
C(z := new B; z.bar(z)) ⊆ C((new A).bar(x))
B ⊆ C(new B)
C(new B) ⊆ C(z)
C(new B) ⊆ C(z := new B)
B ⊆ C(z)
C(z) ⊆ C(w)
C(w.bar(new B)) ⊆ C(z.bar(z))
B ⊆ C(w)
C(new B) ⊆ C(w)
C(w.bar(new B)) ⊆ C(w.bar(new B))
Notes

• Receiver class analysis of OO languages and control flow analysis of functional languages are the same problem

• Receiver class analysis is important in practice
  • Heavily object-oriented code pays a high price for indirect method calls
  • If we can show that only one method can be called, the function can be statically bound
    • Or even inlined and optimized
Type Safety

• Notice that our OO language is untyped
  • We can run (new A).f(0) even if A has no f method
  • Gives a runtime error

• By adding upper bounds to the constraints, we can make receiver class analysis into a type inference procedure for our language
Type Inference

id := e
  C(e) ⊆ C(id)
  C(e) ⊆ C(id := e)
e_1; e_2
  C(e_2) ⊆ C(e_1; e_2)
new A
  A ⊆ C(new A)
if e_1 e_2 e_3
  C(e_2) ⊆ C(if e_1 e_2 e_3)
  C(e_3) ⊆ C(if e_1 e_2 e_3)
  C(e_1) ⊆ Bool
e_0.f(e_1)
  for each class A with a method f(x) e
  A ⊆ C(e_0) ⇒
  C(e_1) ⊆ C(x) ∧
  C(e) ⊆ C(e_0.f(e_1))
C(e_0) ⊆ { A | A has an f method }
Example Revisited: Type Safety

class A
    foo(x) (new A).bar(x)
    bar(y) z := new B; z.bar(z)

class B
    bar(w) w.bar(new B)

A ⊆ C(new A)
C(x) ⊆ C(y)
C(z := new B; z.bar(z)) ⊆ C((new A).bar(x))
B ⊆ C(new B)
C(new B) ⊆ C(z)
C(new B) ⊆ C(z := new B)
B ⊆ C(z)
C(z) ⊆ C(w)
C(w.bar(new B)) ⊆ C(z.bar(z))
B ⊆ C(w)
C(new B) ⊆ C(w)
C(w.bar(new B)) ⊆ C(w.bar(new B))
Type Inference (Cont.)

• These constraints may not have a solution

• If there is a solution, every dispatch will succeed at runtime

• Note: Requires a whole-program analysis
Alias Analysis

• In languages with side effects, we want to know which locations may have aliases
  • More than one “name”
  • More than one pointer to them

• E.g.,
  \[ Y = &Z \]
  \[ X = Y \]
  \[ *X = W \quad // \text{changes the value of } *Y \]
  \[ *Y \]
Z = 1
W = 0
Y = &Z
X = Y
*X = W // changes the value of *Y
*Y
The Encoding of a Location

• For a pointer program variable $x$:

$\text{ref}(l_x, \alpha_x, \alpha_x)$

- A label: A set of 0-ary constructors (constants)
- A field used for reading from the location
- A field used for writing to the location
The Encoding of a Location

• For a program variable $x$:

$$\text{ref}(l_x, \alpha_x, \alpha_x)$$

- A label: A set of $0$-ary constructors (constants)
  - covariant

- A field used for reading from the location
  - covariant

- A field used for writing to the location
  - contravariant
\( \text{ref}(l_1, t_1, t_2) \subseteq \text{ref}(l_2, t_3, t_4) \Rightarrow \)
\[ \begin{align*}
\text{ref}(l_1, t_1, t_2) \subseteq \text{ref}(l_2, t_3, t_4),
& l_1 \subseteq l_2, t_1 \subseteq t_3, t_4 \subseteq t_2
\end{align*} \]
A Pointer Language

P ::= S ... S
S ::= E = E | E
E ::= *E | &E | x
Inference Rules

\[
\begin{align*}
\text{[Var]} & \quad x : \text{ref}(l_x, \alpha_x, \alpha_x) \\
\text{e : t} & \\
t & \subseteq \text{ref}(1, \alpha, 0) & \alpha \text{ fresh} \\
\text{e : t} & \\
\text{t} & \subseteq \text{ref}(1, \alpha, 0) & \alpha \text{ fresh} \\
\text{[Deref]} & \quad \ast e : \alpha \\
\text{e : t} & \\
\text{t} & \subseteq \text{ref}(1, \alpha, 0) & \alpha \text{ fresh} \\
\text{e : t} & \\
\text{t} & \subseteq \text{ref}(1, \beta, 0) & \beta \text{ fresh} \\
\text{[Assign]} & \quad \beta \subseteq \alpha \\
\text{e : t} & \\
e_1 & = e_2 : t_2
\end{align*}
\]
Example

Y = &Z
X = Y
*X = W
*Y

Y : ref(l_y, Y, Y)
X : ref(l_x, X, X)
W: ref(l_w, W, W)
Z: ref(l_z, Z, Z)

Y = &Z:
ref(l_y, Y, Y) ⊆ ref(1,1,A)
ref(0, ref(l_z, Z, Z), ref(l_z, Z, Z)) ⊆ ref(1,B,0)
B ⊆ A
A ⊆ Y
ref(l_z, Z, Z) ⊆ B
ref(l_z, Z, Z) ⊆ Y

X = Y:
ref(l_x, X, X) ⊆ ref(1,1,C)
ref(l_y, Y, Y) ⊆ ref(1,D,0)
C ⊆ X
Y ⊆ D
D ⊆ C
Y ⊆ X

*X:
ref(l_x, X, X) ⊆ ref(1, E, 0)
X ⊆ E

*X = W:
E ⊆ ref(1,1,F)
ref(l_w, W, W) ⊆ ref(1,G,0)
ref(l_z, Z, Z) ⊆ Y ⊆ X ⊆ ref(1,1,F)
G ⊆ F
F ⊆ Z
W ⊆ G
W ⊆ Z

*Y:
ref(l_y, Y, Y) ⊆ ref(1, H, 0)
Y ⊆ H
In Practice

• Many natural inclusion-based analysis problems are equivalent to dynamic transitive closure

• Not trivial to implement well
  • $O(n^3)$ suggests it may be slow
  • Many naïve implementations have failed
  • But today these algorithms can scale to millions of lines code
Applications

• Used heavily in compilers, bug-finding tools
  • Reason about data flow in a program
  • Useful for finding opportunities for optimizations
    • E.g., a dynamic dispatch that has exactly one possible target

• Not currently used in type checking or inference
  • The types are harder to read and interpret
Summary

• Set constraints have many applications in static analysis
  • Closure analysis
  • Receiver class analysis
  • Alias analysis
  • And more ...

• Used often in programming tools