Set Constraints
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 → x | λx.e | 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?
  • $L(...)$ is a set of static, or syntactic, lambdas
  • $L(...)$ defines a call graph
    • the set of functions that may be called by an application
A More General Definition

• To compute $L(\ldots)$ 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 $L(...)$

\[ \lambda x . e \]

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

\[ e_1 e_2 \]

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

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

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

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:

\[
\begin{align*}
\lambda x. e & \quad \lambda x. e \subseteq L(\lambda x. e) \\
n_1 n_2 & \\
\lambda x. e_0 \subseteq L(n_1) \Rightarrow (L(e_2) \subseteq L(x) \land L(e_0) \subseteq L(n_1 n_2))
\end{align*}
\]

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

$L \rightarrow 0 ~|~ x ~|~ c(L_1,\ldots,L_n) ~|~ L_1 \cup L_2$

$R \rightarrow 1 ~|~ x ~|~ c(R_1,\ldots,R_n) ~|~ R_1 \cap R_2$

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

\[ S, L_1 \cup L_2 \subseteq R \rightarrow S, L_1 \cup L_2 \subseteq R, L_1 \subseteq R, L_2 \subseteq R \]

\[ S, L \subseteq R_1 \cap R_2 \rightarrow S, L \subseteq R_1 \cap R_2, L \subseteq R_1, L \subseteq R_2 \]

\[ S, c(L_1,...,L_n) \subseteq c(R_1,...,R_n) \rightarrow S, c(L_1,...,L_n) \subseteq c(R_1,...,R_n), L_1 \subseteq R_1, ..., L_n \subseteq R_n \]

\[ S, c \subseteq R \Rightarrow L' \subseteq R', c \subseteq R \rightarrow S, c \subseteq R \Rightarrow L' \subseteq R', c \subseteq R, L' \subseteq R' \]

\[ S, L \subseteq x, x \subseteq R \rightarrow S, L \subseteq x, x \subseteq R, L \subseteq R \]

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

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

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

\[ i \quad i \subseteq L(i) \quad \text{Idea: Treat integers like lambdas and track their flow.} \]

\[ e_1 \quad e_2 \]
\[ \text{for each } \lambda x. e \subseteq L(e_1) \]
\[ L(e_2) \subseteq L(x) \]
\[ L(e) \subseteq L(e_1 \ e_2) \]
Application: Type Inference!

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

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

\[ e_1 \ e_2 \]

for each \[ \lambda x. e \subseteq L(e_1) \]
\[ L(e_2) \subseteq L(x) \]
\[ L(e) \subseteq L(e_1 \ e_2) \]

\[ L(e_1) \subseteq \{ \lambda x. e \mid \lambda x. e \text{ is a lambda abstraction in the program} \} \]
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., \texttt{cons(A,B)} is the cross product of \texttt{cons(a,b)} for every \texttt{a} in \texttt{A} and \texttt{b} in \texttt{B}
Recall: Simple Type Inference Rules

\[
\begin{align*}
\text{[Var]} & : A, x: \alpha_x \vdash x: \alpha_x \\
\text{[Abs]} & : A, x: \alpha_x \vdash e: t \\
& \quad \Rightarrow A \vdash \lambda x: \alpha_x.e: \alpha_x \rightarrow t
\end{align*}
\]

\[
\begin{align*}
\text{[App]} & : t = t' \rightarrow \beta \\
& \quad \Rightarrow A \vdash e_1: t \\
& \quad \Rightarrow A \vdash e_2: t' \\
& \quad \Rightarrow A \vdash e_1 e_2: \beta
\end{align*}
\]
A Small Change

\[ A, x: \alpha_x \vdash x: \alpha_x \]\hspace{1cm}[Var]\hspace{1cm}\[ A, x: \alpha_x \vdash e: t \]\hspace{1cm}[Abs]

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

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

\[ A \vdash e_1: t \]

\[ A \vdash e_2: t' \]\hspace{1cm}[App]\hspace{1cm}\[ A \vdash e_1 e_2: \beta \]
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

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

\[ S, t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4 \Rightarrow S, t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4, t_1 \ ? t_3, t_2 \ ? t_4 \quad \text{[Subtyping]} \]
What is the subtyping rule for $t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4$?
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 $t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4$?

\[
t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4 \Rightarrow t_3 \subseteq t_1 \land t_2 \subseteq t_4
\]
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(+,+)} \quad \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

\[ S, t_1 \subseteq t_2, t_2 \subseteq t_3 \Rightarrow S, t_1 \subseteq t_2, t_2 \subseteq t_3, t_1 \subseteq t_3 \] [Transitivity]

\[ S, t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4 \Rightarrow S, t_1 \rightarrow t_2 \subseteq t_3 \rightarrow t_4, t_3 \subseteq t_1, t_2 \subseteq t_4 \] [Subtyping]
Comparisons

• The lambda calculus with type equality

• The lambda calculus with subtyping

• Which is equivalent to closure analysis
Recall

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

\( i \subseteq L(i) \)

\( e_1 \ e_2 \)
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 $\lambda x. e \subseteq L(e_1)$

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

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

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

$t \subseteq t' \rightarrow \beta$

$A \vdash e_1 : t$

$A \vdash e_2 : t'$

$A \vdash e_1 e_2 : \beta$

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 $e_0.f(e_1, ..., e_n)$

• 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

P ::= C₁ . . . Cₙ E
C ::= class ClassId M₁ . . . Mₙ

M ::= MId(Id) E

E ::= Id := E | E.MId(E) | E;E | new ClassId | if E E E
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

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)

\( e_0.f(e_1) \)

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))
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)
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 \((\text{new } A).f(0)\) 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

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

\[e_0.\text{f}(e_1)\]

\text{for each class } A \text{ with a method } f(x) \text{ e}

\[A \subseteq C(e_0) \Rightarrow\]
\[C(e_1) \subseteq C(x) \land\]
\[C(e) \subseteq C(e_0.\text{f}(e_1))\]
\[C(e_0) \subseteq \{ A \mid A \text{ has an } f \text{ method } \}\]
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 $x$:

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

$\text{ref}(l_x, \alpha_x, \alpha_x)$
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
\[
\text{ref}(l_1, t_1, t_2) \subseteq \text{ref}(l_2, t_3, t_4) \Rightarrow \\
\text{ref}(l_1, t_1, t_2) \subseteq \text{ref}(l_2, t_3, t_4), l_1 \subseteq l_2, t_1 \subseteq t_3, t_4 \subseteq t_2
\]
A Pointer Language

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

\[
\begin{align*}
\text{Var} & : \quad x : \text{ref}(l, \alpha, \alpha) \\
& \quad e : t \\
& \quad t \subseteq \text{ref}(1, \alpha, 0) \\
& \quad \alpha \text{ fresh} \\
\hline
& \quad *e : \alpha
\end{align*}
\]

\[
\begin{align*}
\text{Deref} & : \quad e : t \\
& \quad \&e : \text{ref}(0, t, t) \\
& \quad e_1 : t_1 \quad e_2 : t_2 \\
& \quad t_1 \subseteq \text{ref}(1, 1, \alpha) \quad \alpha \text{ fresh} \\
& \quad t_2 \subseteq \text{ref}(1, \beta, 0) \quad \beta \text{ fresh} \\
& \quad \beta \subseteq \alpha \\
\hline
& \quad e_1 = e_2 : t_2
\end{align*}
\]

\[
\begin{align*}
\text{Assign} & : \quad e : t \\
& \quad e_1 = e_2 : t_2
\end{align*}
\]
Example

\[ Y = \&Z \]
\[ X = Y \]
\[ \ast X = W \]
\[ \ast Y \]

\[ \begin{align*}
Y &= \&Z : \\
X &= Y : \\
\ast X &= W : \\
\ast Y &= \\
\end{align*} \]

\[ \begin{align*}
Y : \text{ref}(l_y, Y, Y) \\
X : \text{ref}(l_x, X, X) \\
W : \text{ref}(l_w, W, W) \\
Z : \text{ref}(l_z, Z, Z) \\
\end{align*} \]

\[ \begin{align*}
\&Z : \text{ref}(0, \text{ref}(l_z, Z, Z), \text{ref}(l_z, Z, Z)) \\
Y &= \&Z : \\
\text{ref}(l_y, Y, Y) &\subseteq \text{ref}(1,1,A) \\
\text{ref}(0, \text{ref}(l_z, Z, Z), \text{ref}(l_z, Z, Z)) &\subseteq \text{ref}(1,B,0) \\
B &\subseteq A \\
A &\subseteq Y \\
\text{ref}(l_z, Z, Z) &\subseteq B \\
\text{ref}(l_z, Z, Z) &\subseteq Y \\
\end{align*} \]

\[ \begin{align*}
X &= Y : \\
\text{ref}(l_x, X, X) &\subseteq \text{ref}(1,1,C) \\
\text{ref}(l_y, Y, Y) &\subseteq \text{ref}(1,D,0) \\
C &\subseteq X \\
Y &\subseteq D \\
D &\subseteq C \\
Y &\subseteq X \\
\end{align*} \]

\[ \begin{align*}
\ast X: \\
\text{ref}(l_x, X, X) &\subseteq \text{ref}(1, E, 0) \\
X &\subseteq E \\
\ast X &= W: \\
E &\subseteq \text{ref}(1,1,F) \\
\text{ref}(l_w, W, W) &\subseteq \text{ref}(1,G,0) \\
\text{ref}(l_z, Z, Z) &\subseteq Y \subseteq X \subseteq \text{ref}(1,1,F) \\
G &\subseteq F \\
F &\subseteq Z \\
W &\subseteq G \\
W &\subseteq Z \\
\ast Y: \\
\text{ref}(l_y, Y, Y) &\subseteq \text{ref}(1, H, 0) \\
Y &\subseteq H \\
\end{align*} \]
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