Constraint-Based Analysis
(Lecture slides from Alex Aiken, CS 294 lecture 4)

Type Inference Problems
- Type inference problems are described as:
  \[ \tau_i \tau_j = \tau_k \]
  \[ \tau = c(\tau_1, \ldots, \tau_n) | \alpha \]
- \( c \) is a constructor (may be 0-ary)
  - Like function arrow, product, or ref
- System of equations
- Arbitrary expressions on lhs and rhs
- Domain is terms

Dataflow Problems
- Recall Gen/Kill data flow problems look like
  - \( \text{In}(S) = \bigcup_{s \in \text{pred}(S)} \text{Out}(s) \)
  - \( \text{Out}(S) = \text{Gen}(S) \cup (\text{In}(S) - \text{Kill}(S)) \)
- These can be though of as constraints
  - \( \text{In}(S) \) and \( \text{Out}(S) \) are variables
  - We don’t need \( = \), since we’re really computing least solutions

Dataflow Problems as Constraints
- So we can rewrite those equations as
  - \( \forall_{\text{In}(S)} \supset \bigcup_{s \in \text{pred}(S)} \forall_{\text{Out}(s)} \)
  - \( \forall_{\text{Out}(S)} \supset \forall_{\text{I}} \supset (\forall_{\text{In}(S)} \cap \forall_{\text{E}}) \)

Dataflow Problems
- Classical dataflow equations are described as:
  \[ \cap_i v_i \supset E_i \]
  \[ E = E \cup E | E \cap E | v | a \]
- \( v \) is a variable, \( a \) is an atom
- System of inclusion constraints
- Only variables on lhs
- Domain is atoms

Summary
- Dataflow analysis
  - Inclusion constraints over atoms
- Type inference
  - Equations over terms
- Two very different theories
  - With different applications
  - Developed over decades
- But are they really independent?
Set Constraints

- The set expressions are:
  \[ E ::= 0 \mid \alpha \mid E \cup E \mid E \cap E \mid \neg E \mid c(E_1, \ldots, E_n) \mid c_i(E) \]

- A system of set constraints is
  \[ \bigwedge_i E_{i1} \subseteq E_{i2} \]

- Constructors \( c \)
- Set variables \( \alpha \)

Semantics of Set Expressions

E ::= 0 \mid \alpha \mid E \cup E \mid E \cap E \mid \neg E \mid c(E_1, \ldots, E_n) \mid c_i(E)

- One interpretation: Set expressions denote subsets of the Herbrand Universe \( H \)
  - \( H_0 = \emptyset \)
  - \( H_{i+1} = H_i \cup \{ c(t_1, \ldots, t_n) \mid t_i \in H_i, c \in C \} \)
  - \( H \) is the least upper bound of the series \( H_0 \subseteq H_1 \subseteq \ldots \)

- Notice that \( c(\ldots, \emptyset, \ldots) \) can never be in \( H \)

Semantics of Set Expressions (Cont.)

- An assignment maps variables to sets of terms:
  \( \sigma : \text{Vars} \to 2^H \)
- Extend \( \sigma \) to all set expressions:
  \[
  \begin{align*}
  \sigma(0) &= \emptyset \\
  \sigma(E_1 \cup E_2) &= \sigma(E_1) \cup \sigma(E_2) \\
  \sigma(E_1 \cap E_2) &= \sigma(E_1) \cap \sigma(E_2) \\
  \sigma(\neg E) &= H - \sigma(E) \\
  \sigma(c(E_1, \ldots, E_n)) &= \{ c(t_1, \ldots, t_n) \mid t_i \in \sigma(E_i) \} \\
  \sigma(c_i(E)) &= \{ t_i \mid c(t_1, \ldots, t_n) \in \sigma(E) \}
  \end{align*}
  \]

Solutions

- An assignment \( \sigma \) is a solution of the constraints if
  for all \( i \), \( \sigma(E_{i1}) \subseteq \sigma(E_{i2}) \)

Notes on Projection

- Projection can model data selectors
  - \( \text{Car}, \text{cdr}, \text{hd}, \text{tl}, \) etc.
- But projections have another interesting property:
  \[
  c^{-1}(c(A, B)) = \begin{cases} A & \text{if } B \neq 0 \\ 0 & \text{otherwise} \end{cases}
  \]
Conditional

- Projections can be used to encode *conditional* constraints:

\[ B \neq 0 \Rightarrow A \subseteq C \]

is equivalent to

\[ c^{-1}(c(A, B)) \subseteq C \]

---

Complexity

- **Thm.** Deciding whether a system of set constraints has any solutions is NEXPTIME-complete

- Remains NEXPTIME-complete even if we drop projections

- So, focus on tractable sub-theories

---

Sources of Complexity

- For equality constraints with no \( \land, \lor, \neg \)
  - Use union-find; near-linear time

- For (restricted) inclusion constraints
  - Use transitive closure; PTIME

\[ A \subseteq B \subseteq C \Rightarrow A \subseteq C \]

---

Sources of Complexity (Cont.)

- For EXPTIME algorithms, general \( \land, \lor, \neg \)

- For NEXPTIME algorithms, the choice

\[ c(A, B) = 0 \Leftrightarrow A = 0 \lor B = 0 \]

---

Connections

- Set constraints are related to
  - Tree automata
  - Logic (the monadic class)

- Also, implementation techniques are based on graphs and graph algorithms

---

A Tractable Fragment

\[
\begin{align*}
L &::= L \cup L \mid c(L, \ldots, L) \mid \alpha \mid 0 \\
R &::= R \cap R \mid c(R, \ldots, R) \mid \alpha \mid 1
\end{align*}
\]

Let \( C \) be constraints of the form:

\[ L \subseteq R \]

\[ \alpha \neq 0 \Rightarrow L \subseteq R \]
Solving Set Constraints

• The usual strategy:
  - Rewrite constraints, preserving solutions
  - When all possible rewrites have been done, the system is in "solved form"
    - Solutions are manifest

• Note: there are different notions of "solve"
  - Has at least one solution (yes/no)
  - Describe one solution (e.g., the least)
  - Describe all solutions

Resolution Rules 1

- Trivial constraints:
  \[
  S \wedge L \subseteq 1 \Rightarrow S \\
  S \wedge 0 \subseteq R \Rightarrow S \\
  S \wedge x \subseteq x \Rightarrow S
  \]

Resolution Rules 2

More interesting constraints:

\[
L \subseteq R_1 \cap R_2 \Rightarrow L \subseteq R_1 \wedge L \subseteq R_2 \\
L_1 \cup L_2 \subseteq R \Rightarrow L_1 \subseteq R \wedge L_2 \subseteq R \\
c(...) \subseteq \alpha \wedge \alpha \subseteq R \Rightarrow c(...) \subseteq \alpha \wedge \alpha \subseteq R \wedge c(...) \subseteq R
\]

Resolution Rules 3

- And more interesting constraints:
  \[
c(L_1, L_2) \subseteq c(R_1, R_2) \Rightarrow L_1 \subseteq R_1 \wedge L_2 \subseteq R_2 \\
c(\ldots) \subseteq \alpha \wedge (\alpha \neq 0 \Rightarrow L \subseteq R) \Rightarrow c(\ldots) \subseteq \alpha \wedge L \subseteq R
  \]

  - These rules preserve all solutions for non-strict constructors
  - \( c(\ldots,0,\ldots) \times 0 \)
  - Warning: \( c \) can't be the function constructor

Resolution Rules 4

- Note how the rules preserve \( R \) and \( L \):
  \[
c(L_1, L_2) \subseteq c(R_1, R_2) \Rightarrow L_1 \subseteq R_1 \wedge L_2 \subseteq R_2 \\
  \]
  - We can also have constructors with contravariant arguments; e.g., \( \rightarrow \)
    \[
    L ::= \ldots \mid R \rightarrow L \\
    R ::= \ldots \mid L \rightarrow R
    \]
  - \( R_1 \rightarrow L_1 \subseteq L_2 \rightarrow R_2 \Rightarrow L_2 \subseteq R_1 \wedge L_1 \subseteq R_2 \)

An Observation

- Note the resolution rules do not create new expressions
  - Only subexpressions are used, e.g.,
  \[
  L \subseteq R_1 \cap R_2 \Rightarrow L \subseteq R_1 \wedge L \subseteq R_2 \\
  L_1 \cup L_2 \subseteq R \Rightarrow L_1 \subseteq R \wedge L_2 \subseteq R \\
  c(...) \subseteq \alpha \wedge \alpha \subseteq R \Rightarrow c(...) \subseteq \alpha \wedge \alpha \subseteq R \wedge c(...) \subseteq R
  \]
A Graph Interpretation

- Treat each subexpression as a node in a graph
- Constraints \( L \subseteq R \) are directed edges from \( L \) to \( R \)
- Recast resolution rules as graph transformations

Resolution on Graphs 1

\[
\begin{align*}
\alpha & \Rightarrow c(\ldots) \subseteq \alpha \land \alpha \subseteq R \Rightarrow \\
\alpha & \Rightarrow c(\ldots) \subseteq \alpha \land c(\ldots) \subseteq R \\
\end{align*}
\]

Resolution on Graphs 2

\[
\begin{align*}
c(\ldots) \subseteq \alpha \land (\alpha \neq 0 \rightarrow L \subseteq R) \Rightarrow \\
c(\ldots) \subseteq \alpha \land L \subseteq R \\
\end{align*}
\]

Resolution on Graphs 3

\[
\begin{align*}
c(L_1, L_2) \subseteq c(R_1, R_2) \Rightarrow L_1 \subseteq R_1 \land L_2 \subseteq R_2 \\
\end{align*}
\]

The Other Constraints

- Skip presentation of rules for other constraints
  - Trivial constraints
  - Intersection/union constraints
- Easily handled
  - In practice, edges from these constraints are not explicitly represented anyway
  - Tend to keep only constraints on variables

Notes

- The process of adding edges according to a set of rules is called closing the graph
- The closed graph gives the solution of the constraints
Algorithmics

• This algorithm is a dynamic transitive closure
• New edges other than transitive edges are added during the closure procedure
• Can’t use standard transitive closure tricks
  - E.g., Boolean matrix multiplication

Dynamic Transitive Closure

• The best known algorithms for dynamic transitive closure are $O(n^3)$
  - Has not been improved in 30 years
• Sketch: In the worst case, a graph of $n$ nodes
  - May have $n^2$ edges
  - Each edge may be added $O(n)$ times

Applications

Three Applications

• Closure analysis for lambda calculus
• Receiver class analysis for OO languages
• Alias analysis for C

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 in Functional Languages

• Recall the untyped lambda calculus:
  
  $$ e = x | \lambda x.e | e e $$

• Examples:
  - $((\lambda x.x)(\lambda y.y))(\lambda z.z)$
  - $((\lambda x.y.y)(\lambda z.z))(\lambda w.w)$
  - $(\lambda x.x)(\lambda y.y)$
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?
  - \( L(\cdot) \) is a set of static, or syntactic, lambdas
  - \( L(\cdot) \) defines a call graph
  - The set of functions that may be called by an application

A More General Definition

- To compute \( L(\cdot) \) for applications, we will need to 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 \( L(\cdot) \)

\[
\lambda x.e \\
L(\lambda x.e) = \lambda x.e \\
e_1 e_2 \\
\text{for each } \lambda x.e \text{ in } L(e_1) \\
L(e_2) \subseteq L(x) \\
L(e) \subseteq L(e_1 e_2)
\]

The actual argument of the call flows to the formal argument

The value of the application includes the value of the function body

Example \((\lambda x. (\lambda y. y)) (\lambda z. z)\)

\[
\lambda x.x \subseteq L(\lambda x.x) \\
\lambda y.y \subseteq L(\lambda y.y) \\
\lambda z.z \subseteq L(\lambda z.z) \\
L(x) \subseteq L(\lambda x.x) \quad L(y) = L(\lambda y.y) \\
L(z) = L(\lambda z.z) \\
L(\lambda x.x) = L(\lambda y.y) = L(\lambda z.z)
\]

Least solution:

\[
L(\lambda x.x) = \lambda x.x \\
L(\lambda y.y) = \lambda y.y \\
L(\lambda z.z) = \lambda z.z
\]

Rephrasing the Constraints with \( \subseteq \)

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

\[
\lambda x.e \subseteq L(\lambda x.e) \\
e_1 e_2 \subseteq L(e_1) \\
\lambda x.e_0 \subseteq L(e_1) \\
\Rightarrow (L(e_2) \subseteq L(x) \land L(e_0) \subseteq L(e_1 e_2))
\]

Note: Each \( L(e) \) is a constraint variable
Each \( \lambda x.e \) is a constant

The Example \((\lambda x. (\lambda y. y)) (\lambda z. z)\) with Graphs
The solution is given by edges $(\lambda x.e,*)$. The solution for $((\lambda x.x)(\lambda y.y))(\lambda z.z)$ is shown in the diagram.

**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 run time

**An OO Language**

- $P ::= C_1 \ldots C_n \ E$
- $C ::= \text{class ClassId \ [inherits ClassId]}$
- $M ::= \text{method MId(Id) \ E}$
- $E ::= \text{Id := E | E.MId(E,...,E) | E;E | new ClassId | if E E E}$

**Constraints**

- $\text{id := e}$
- $\text{e_0.f(e_1)}$ for each class $A$ with a method $f(x)$
- $C(e) \subseteq C(e_0.f(e_1))$
- $C(e_0) \subseteq C(x) \land C(\text{new A}) \subseteq C(\text{new A})$
- $C(\text{if e_1 e_2 e_3}) \subseteq C(\text{if e_1 e_2 e_3})$

**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

**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 the indirection in 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 Inference

\[ \begin{align*}
\text{id} & := e \\
C(e) & \subseteq C(\text{id}) \\
C(e) & \subseteq C(\text{id} := e) \\
e_1; e_2 & \subseteq C(e_1; e_2) \\
\text{new } A & \subseteq C(\text{new } A) \\
\text{if } e_1; e_2; e_3 & \subseteq C(\text{if } e_1; e_2; e_3) \\
C(e_0) & \subseteq \{ \text{A} \mid \text{A has an } f \text{ method} \}
\end{align*} \]

Type Inference (Cont.)

- These constraints may not have a solution
  - May discover that the constraints require \( (B) \subseteq 0 \)
- If there is a solution, every dispatch will succeed at runtime
  - Note: Requires a whole-program analysis

Alias Analysis

- In languages with side effects, want to know which locations may have *aliases*
  - More than one "name"
  - More than one pointer to them
- This is the same problem as before
  - At \(*x\), what locations may \(x\) point to?
  - Solve with similar techniques

In Practice

- Many natural inclusion-based analysis problems are equivalent to dynamic transitive closure
- Widely believed to be impractical
  - \(O(n^3)\) suggests it may be slow
  - And in fact it is
    - Many implementations have tried

Summary of Constraint-Based Analysis

- Constraints separate
  - Specification (system of constraints)
  - Implementation (constraint resolution)
  - Clear place to apply algorithmic knowledge
- No forwards-backwards distinction
  - Can solve for any unknown
- Infinite domains
- Separate analysis is easy
  - Can always solve constraints

Where is Constraint-Based Analysis Weak?

- Only fairly simple constraints are practical
  - This situation is improving
- Doesn’t capture all of abstract interpretation
  - In particular, situations where there is a favored direction (forwards, backwards) for efficiency reasons
Things We Didn’t Talk About

• Polymorphism
  - Context-free reachability & polymorphic recursion

• Effect Systems
  - A computation has a type & an effect
  - E.g., the set of memory locations written
  - Mixed constraint systems

• Other constraint languages
  - There are some besides = and ≤