**Constraint-Based Analysis**

*CMSC 631, Fall 2003*

---

**Type Inference Problems**

- Type inference problems are described as:
  \[ \&_1 \&_2 \quad \&_2 = c(\&_1, \ldots, \&_n) \]

- \(c\) 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:
  \[
  \begin{align*}
  \text{In}(S) &= \bigcup z \text{ in pred}(S) \ \text{Out}(s) \\
  \text{Out}(S) &= \text{Gen}(S) \cup (\text{In}(S) - \text{Kill}(S))
  \end{align*}
  \]

- These can be thought 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:
  \[
  \begin{align*}
  \nu_{\text{In}(S)} &\cup z \text{ in pred}(S) \nu_{\text{Out}(s)} \\
  \nu_{\text{Out}(S)} &\cup E_1 \cup (\nu_{\text{In}(S)} \cup E_2)
  \end{align*}
  \]

---

**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:
  \[ E ::= 0 | \emptyset | E + E | E \& E | \neg E | c(E,...,E) | c_i^{-1}(E) \]
  - Here + is union, \& is intersection, and \neg is complement
- A system of set constraints is
  \[ \&_i E_i \sqsubseteq E \]
- Constructors \( c \)
- Set variables \( \emptyset \)

**Semantics of Set Expressions**

\[ E ::= 0 | \emptyset | E + E | E \& E | \neg E | c(E,...,E) | c_i^{-1}(E) \]

- One interpretation: Set expressions denote subsets of the Herbrand Universe \( H \)
- An assignment maps variables to sets of terms:
  \[ \emptyset : Vars \rightarrow 2^H \]

**Semantics of Set Expressions (Cont.)**

- Extend \( \emptyset \) to all set expressions:
  \[
  \emptyset(0) = \text{empty set} \\
  \emptyset(E_1 + E_2) = \emptyset(E_1) + \emptyset(E_2) \\
  \emptyset(E_1 \& E_2) = \emptyset(E_1) \& \emptyset(E_2) \\
  \emptyset(\neg E) = H - \emptyset(E) \\
  \emptyset(c(E_1,...,E_n)) = \{ c(t_1,...,t_n) \mid t_i \in \emptyset(E_i) \} \\
  \emptyset(c_i^{-1}(E)) = \{ t_i \mid c(t_1,...,t_n) \in \emptyset(E) \} 
  \]

**Solutions**

- An assignment \( \emptyset \) is a solution of the constraints if
  \[ \&_i \emptyset(E_i) \sqsubseteq \emptyset(E) \]

**Notes on Projection**

- Projection can model data selectors
  - \( \text{Car}, \text{cdr}, \text{hd}, \text{tl}, \text{etc.} \)
- But projections have another interesting property:
  \[ c_i^{-1}(c(A,B)) = \begin{cases} 
  A & \text{if } B \neq 0 \\
  \emptyset & \text{otherwise}
  \end{cases} \]
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 +, &, !
  - Use union-find; near-linear time
- For (restricted) inclusion constraints
  - Use transitive closure; PTIME
    \[ A \sqsubseteq B \sqsubseteq C \sqsubseteq A \sqsubseteq C \]

Sources of Complexity (Cont.)

- For EXPTIME algorithms, general +, &, !
- 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

\[
\begin{align*}
L & ::= L + L \mid c(L,...,L) \mid \square \mid 0 \\
R & ::= R \& R \mid c(R,...,R) \mid \square \mid 1 \\
\end{align*}
\]

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

\[
\begin{align*}
L & \square R \\
\square & = 0 \square L \& R
\end{align*}
\]
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:

\[
\begin{align*}
L & \subseteq R_1 \cap R_2 \quad \Rightarrow \quad L \subseteq R_1 \wedge L \subseteq R_2 \\
L_1 \cup L_2 & \subseteq R \quad \Rightarrow \quad L_1 \subseteq R \wedge L_2 \subseteq R \\
\end{align*}
\]

\[ c(\ldots) \subseteq \alpha \wedge \alpha \subseteq R \Rightarrow c(\ldots) \subseteq \alpha \wedge \alpha \subseteq R \wedge c(\ldots) \subseteq R \]

Resolution Rules 3

• And more interesting constraints:

\[
\begin{align*}
c(L_1, L_2) & \subseteq c(R_1, R_2) \quad \Rightarrow \quad L_1 \subseteq R_1 \wedge L_2 \subseteq R_2 \\
c(\ldots) & \subseteq \alpha \wedge (\alpha \neq 0 \rightarrow L \subseteq R) \quad \Rightarrow \quad c(\ldots) \subseteq \alpha \wedge \alpha \subseteq R \wedge c(\ldots) \subseteq R \\
\end{align*}
\]

• These rules preserve all solutions for non-strict constructors
  - \( c(\ldots,0,\ldots) \neq 0 \)

• 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.,

\[
\begin{align*}
L & ::= \ldots | R \Box L \\
R & ::= \ldots | L \Box R \\
L_1 \rightarrow L_1 & \subseteq L_2 \rightarrow R_2 \Rightarrow L_2 \subseteq R_1 \wedge L_1 \subseteq R_2 \\
\end{align*}
\]

An Observation

• Note the resolution rules do not create new expressions
  - Only subexpressions are used, e.g.,

\[
\begin{align*}
L & \subseteq R_1 \cap R_2 \quad \Rightarrow \quad L \subseteq R_1 \wedge L \subseteq R_2 \\
L_1 \cup L_2 & \subseteq R \quad \Rightarrow \quad L_1 \subseteq L \wedge L_2 \subseteq R \\
c(\ldots) & \subseteq \alpha \wedge \alpha \subseteq R \Rightarrow c(\ldots) \subseteq \alpha \wedge \alpha \subseteq R \wedge c(\ldots) \subseteq R \\
\end{align*}
\]
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

$c(\ldots) \subseteq \alpha \land \alpha \subseteq R \Rightarrow$
$c(\ldots) \subseteq \alpha \land \alpha \subseteq R \land c(\ldots) \subseteq R$

Resolution on Graphs 2

$c(\ldots) \subseteq \alpha \land (\alpha \neq 0 \Rightarrow L \subseteq R) \Rightarrow$
$c(\ldots) \subseteq \alpha \land L \subseteq R$

Resolution on Graphs 3

$c(L_1, L_2) \subseteq c(R_1, R_2) \Rightarrow L_1 \subseteq R_1 \land L_2 \subseteq R_2$

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

• 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:
  - $(\lambda x.x)(\lambda y.y)(\lambda z.z)$
  - $(\lambda x.(\lambda y.y)(\lambda z.z))(\lambda w.w)$
  - $(\lambda x.x)(\lambda y.y)$
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 \( L(\cdot) \)

\[
\begin{align*}
\bar{x}.e & \equiv \bar{x}e \\
L(\bar{x}.e) & = \bar{x}e \\
e_1 \ e_2 & \quad \text{for each } \bar{x}e \text{ in } L(e_1) \\
L(e_2) & \subseteq L(x) \\
L(e) & \subseteq L(e_1 \ e_2)
\end{align*}
\]

Rephrasing the Constraints with \( \sqsubseteq \)

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

\[
\begin{align*}
\bar{x}.e & \equiv \bar{x}e \\
L(\bar{x}.e) & = \bar{x}e \\
e_1 \ e_2 & \quad \text{for each } \bar{x}e \text{ in } L(e_1) \\
L(e_2) & \subseteq L(x) \\
L(e) & \subseteq L(e_1 \ e_2)
\end{align*}
\]

Note: Each \( L(e) \) is a constraint variable
- Each \( \bar{x}e \) is a constant

Example \( ((\bar{x}.x) \ ((\bar{y}.y)) \ ((\bar{z}.z)) \)

Least solution:

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

The Example \( ((\bar{x}.x) \ ((\bar{y}.y)) \ ((\bar{z}.z)) \) with Graphs
The Solution for \( ((lx.x)(ly.y))(lz.z)) \)

The solution is given by edges \((lx.e,*))\n
\(lx.x\)
\(lz.z\)
\(ly.y\)

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

\[
P ::= C_1 \ldots C_n E
\]

\[
C ::= \text{class ClassId [inherits ClassId]}
\]

\[
\text{var} \ Id_1 \ldots Id_k \ M_1 \ldots M_n
\]

\[
M ::= \text{method MId(Id) E}
\]

\[
E ::= \text{Id := E | E.MId(E,…,E) | E;E | new ClassId | if E E E}
\]

Constraints

- 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
- 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

Notes
Type Inference

id := e
C(e) ⊑ C(id)
C(e) ⊑ C(id := e)
e1; e2
C(e1) ⊑ C(e1; e2)
new A
{ A } ⊑ C(new A)
if e1 e2 e3
C(e3) ⊑ C(if e1 e2 e3)
C(e3) ⊑ C(if e1 e2 e3)
C(e1) ⊑ { Bool }

e0.f(e1)
for each class A with a method f(x) e

Type Inference (Cont.)

• These constraints may not have a solution
  – May discover that the constraints require \( B \) ⊑ 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

  • E.g.,
  \( Y = \& Z \)
  \( X = Y \)
  \( \* X = 3 \quad /* \text{changes the value of } \* Y */ \)

The Encoding of a Location

\( x : \text{ref}(l_x, \alpha_x, \alpha_x) \)
\( e : \tau \)
\( &e : \text{ref}(0, \tau, \tau) \)
\( e_1 : \tau_1 \quad e_2 : \tau_2 \)
\( \tau_1 \subseteq \text{ref}(1, 1, \alpha) \quad \tau_2 \subseteq \text{ref}(1, 1, \alpha) \quad \beta \subseteq \alpha \)
\( e_1 := e_2 : \tau_2 \)

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
One Problem

- Consider what happens on a cycle in the graph
- A constructed lower bound on any one node is propagated to every node in the cycle

Observation

- A cycle in the graph corresponds to a cycle in the constraints
  - $x_1 \leq x_2 \leq \ldots \leq x_n \leq x_1$
  - All of these variables are equal in all solutions!

- Thus, there is a lot of wasted work in pushing values around cycles
  - And cycles are very common

The Idea

- We want to detect and eliminate cycles on-line
  - Collapse cycles to a single node
  - During constraint resolution

- On-line cycle detection is very hard
  - No known algorithm is significantly better than stopping the graph closure and doing a depth-first search of the entire graph

Partial On-Line Cycle Elimination

- Instead, we will settle for partial cycle elimination
  - For every cycle that exists in the graph, guarantee we find at least a piece of it
  - And do it cheaply

A Different Representation

- We change the representation of the graph
  - Assign every variable $x$ (node) arbitrary index $R(x)$
  - Each node has a list of edges stored with it
  - An edge $(x,y)$ is stored
    - At $x$ if $R(x) > R(y)$ (a successor edge, colored red)
    - At $y$ if $R(y) > R(x)$ (a predecessor edge, colored blue)

- New transitive closure rule:

Cycle Detection Algorithm

- On each edge addition $(x,y)$
  - If $(x,y)$ is a successor edge $(R(x) > R(y))$ then search along predecessor edges from $x$.
    - When a node $z$ s.t. $R(z) < R(y)$ is found, prune that path
    - If $y$ is found, a cycle is detected
  - If $(x,y)$ is a predecessor edge $(R(x) < R(y))$ then search along successor edges from $y$.
    - When a node $z$ s.t. $R(z) > R(x)$ is found, prune that path
    - If $x$ is found, a cycle is detected
Cycle Detection in Pictures

Part of Every Cycle is Detected

• Every cycle has at least one red and one blue edge
  - Indices cannot uniformly increase or decrease around a cycle
• Thus, the transitivity rule always applies
  - Always adds a chord across the cycle, giving a smaller cycle
• Two-cycles are always detected

Analysis of Cycle Detection

• Part of every cycle is detected
• Expected number of nodes visited per edge addition is very low
  - About 2, in theory
  - Why? Long chains of descending, arbitrarily chosen indices are very unlikely
• Can show asymptotic speedup in graph closure for random graphs

Experiments

• Cycle detection is fast
  - In experiments, 1.8 nodes visited/edge addition
  - Constants are very small
• About 80% of nodes in cycles are detected
  - Detected cycles are removed from the graph and put in a union/find data structure
• Gives asymptotic performance improvement
  - For alias analysis of C
    - Allows programs 10X larger to be analyzed than without

Summary

• Dynamic transitive closure algorithms are coming
  - Still "in the lab," but increasingly practical
  - Need more tricks than cycle elimination

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 Õ