Cubic-Time Set Constraints

Terms

- 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
  - $T = \{ f(t_1,\ldots,t_{a(f)}) \mid f \in C, t_i \in T \}$

A Restricted Class of Set Constraints

$\wedge_i E_{i1} \subseteq E_{i2}$

$E ::= 0 \mid E_1 \cap E_2 \mid E_1 \cup E_2 \mid \neg E \mid c(E_1,\ldots,E_{a(c)}) \mid v$

$\wedge_i E_{i1} \subseteq E_{i2}$

$E ::= c(E_1,\ldots,E_{a(c)}) \mid v$

Not So Restricted

$\wedge_i L_i \subseteq R_i$

$L ::= 0 \mid L_1 \cup L_2 \mid c(L_1,\ldots,L_{a(c)}) \mid v$

$R ::= 1 \mid R_1 \cap R_2 \mid c(R_1,\ldots,R_{a(c)}) \mid v$

$\wedge_i E_{i1} \subseteq E_{i2}$

$E ::= c(E_1,\ldots,E_{a(c)}) \mid v$

These Are Equivalent

$\wedge_i L_i \subseteq R_i$

$L ::= 0 \mid L_1 \cup L_2 \mid c(L_1,\ldots,L_{a(c)}) \mid v$

$R ::= 1 \mid R_1 \cap R_2 \mid c(R_1,\ldots,R_{a(c)}) \mid v$

$0 \subseteq R$

$L \subseteq 1$

$L_1 \cup L_2 \subseteq R$

$L \subseteq R_1 \cap R_2$

$c(L_1,\ldots,L_{a(c)}) \subseteq c(R_1,\ldots,R_{a(c)})$

Three Kinds of Constraints

$\wedge_i L_i \subseteq R_i$

$v \subseteq c(R_1,\ldots,R_{a(c)})$

$c(L_1,\ldots,L_{a(c)}) \subseteq v$

$v_1 \subseteq v_2$

Conditional Constraints

$v \neq 0 \Rightarrow L \subseteq R$

which is equivalent to

$c(v,L) \subseteq c(1,R)$

for some (strict) binary constructor c

- Note
  - The explicit constructors are *lazy*
  - But conditional constraints are defined in terms of an (implicit) *strict* constructor
  - Only difference is what happens if arg. of a constructor is 0

What is Left Out?

- Any form of negation
  - Negation itself, obviously
  - But also unions in negative (R) positives
  - Intersections in positive (L) positions

One More Extension

- Constructor arguments can have *polarity*
  - Either positive or negative
- For example
  - Cons(+,+)
  - Function space $- \rightarrow +$
    - Or $\rightarrow (-,+)$

With The Extension

$\wedge_i L_i \subseteq R_i$

$L ::= 0 \mid R_1 \cup R_2 \mid c(L_1,\ldots,L_{a(c)}) \mid \rightarrow(R,L) \mid v$

$R ::= 1 \mid R_1 \cap R_2 \mid c(R_1,\ldots,R_{a(c)}) \mid \rightarrow(L,R) \mid v$

Constraint Resolution with Negative Polarities

$C \wedge 0 \subseteq R \Leftrightarrow C$

$C \wedge L \subseteq 1 \Leftrightarrow C$

$C \wedge L_1 \cup L_2 \subseteq R \Leftrightarrow C \wedge L_1 \subseteq R \wedge L_2 \subseteq R$

$C \wedge L \subseteq R_1 \cap R_2 \Leftrightarrow C \wedge L_1 \subseteq R \wedge L_2 \subseteq R$

$C \wedge c(L_1,\ldots,L_{a(c)}) \subseteq c(R_1,\ldots,R_{a(c)}) \Leftrightarrow C \wedge_i L_i \subseteq R_i$

$C \wedge R_1 \rightarrow L_1 \subseteq L_2 \rightarrow R_2 \Rightarrow C \wedge L_2 \subseteq R_1 \wedge L_1 \subseteq R_2$

$C \wedge L \subseteq v \wedge v \subseteq R \Rightarrow C \wedge L \subseteq v \wedge v \subseteq R \wedge L \subseteq R$

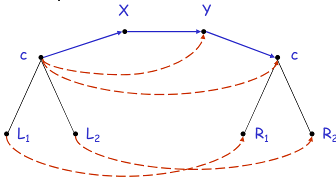
Observation: No new expressions are created by any rule

Algorithm

- Run rules to saturation
  - Until no new constraints can be generated
- Begin with system of size $O(N)$
  - Final system is of size $O(N^2)$
  - Takes time $O(N^3)$

Constraints as Graphs

Graph:



Constraints:

$c(L_1, L_2) \subseteq X$

$X \subseteq Y$

$Y \subseteq c(R_1, R_2)$

Rewrite rules:

$L \subseteq v \subseteq R \Rightarrow L \subseteq R$

$c(L_1,\ldots,L_n) \subseteq c(R_1,\ldots R_n) \Rightarrow \bigwedge_i L_i \subseteq R_i$

Solutions

- Solution size is potentially $O(n^2)$
  - May be the complete graph
- Solution time is $O(n^3)$
  - Each of $O(n^2)$ edges may be added in $O(n)$ ways

Comments

- A single representation for *all* solutions
- A great deal of sharing among the solutions
- Characteristic of PTIME techniques
  - Intuitively, this must be the case
  - Exponential number of incomparable solutions requires more than PTIME
    - Consider SAT, BDDs, general set constraints

And One More Extension

- The closed constraints make explicit the *least* solution
  - Read off the lower bound for each variable
- Allows us to generalize the permitted constraints

$L := L \cap c(0,0) \mid \ldots$

Applications

- Closure analysis
- OO type inference
- Points-to analysis

Applications

- Closure analysis
- OO type inference
- Points-to analysis

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:

$$e = x \mid \lambda x.e \mid e\ e$$

- Examples:
  - $((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z)$
  - $((\lambda x.\lambda y.y)\ (\lambda z.z))\ (\lambda w.w)$
  - $(\lambda x.x\ x)\ (\lambda y.y\ y)$

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(…) 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(…) 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(…)

$\lambda x.e$

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

$e_1\ e_2$

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 ⊆

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

$\lambda x.e$

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

$e_1\ e_2$

$\lambda x.e_0 \cap L(e_1) \Rightarrow (L(e_2) \subseteq L(x) \wedge L(e_0) \subseteq L(e_1\ e_2))$

Note: Each L(e) is a constraint variable

Each $\lambda x.e$ is a constant

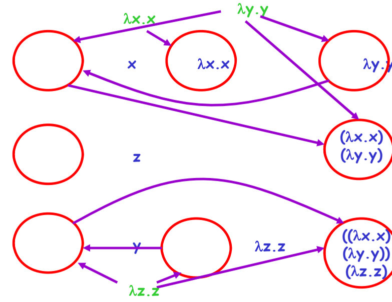
Example $((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z)$

$\lambda x.x \subseteq L(\lambda x.x)$
$\lambda y.y \subseteq L(\lambda y.y)$
$\lambda z.z \subseteq L(\lambda z.z)$
$L(\lambda y.y) \subseteq L(x)$
$L(x) \subseteq L((\lambda x.x)\ (\lambda y.y))$
$L(\lambda z.z) \subseteq L(y)$
$L(y) \subseteq L(((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z))$

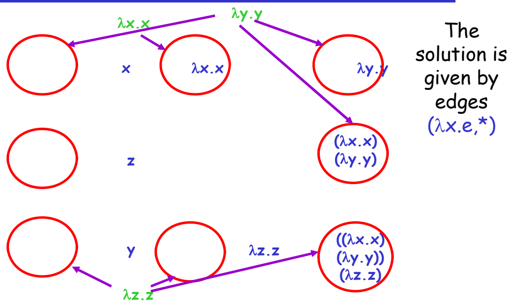
Least solution:

$L(\lambda x.x) = \lambda x.x$
$L(\lambda y.y) = \lambda y.y$
$L(\lambda z.z) = \lambda z.z$
$L(\lambda y.y) = L(x) = L((\lambda x.x)\ (\lambda y.y))$
$L(\lambda z.z) = L(y) = L(((\lambda x.x)\ (\lambda y.y))\ (\lambda z.z))$

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 edges $(\lambda x.e,*)$

Control Flow Graphs in OO Languages

- Consider a method call $e_0.f(e_1,\dots,e_n)$
- 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

$P ::= C_1 \dots C_n\ E$
$C ::=$ class ClassId [inherits ClassId]
    var $Id_1 \dots Id_k\ M_1 \dots M_n$

$M ::=$ method MId(Id) E

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

Constraints

id := e
  $C(e) \subseteq C(id)$
  $C(e) \subseteq C(id := e)$
$e_1; e_2$
  $C(e_2) \subseteq C(e_1; e_2)$
new A
  $A \subseteq C(\text{new } A)$
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 $f(x)\ e$
    $A \cap C(e_0) \Rightarrow$
      $C(e_1) \subseteq C(x) \wedge$
      $C(e) \subseteq C(e_0.f(e_1))$

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

$e_0.f(e_1)$
  for each class A with a method f(x) e
    $A \cap C(e_0) \Rightarrow$
      $C(e_1) \subseteq C(x) \wedge$
      $C(e) \subseteq C(e_0.f(e_1))$
$C(e_0) \subseteq \{ A \mid A \text{ has an f method} \}$

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 (Review)

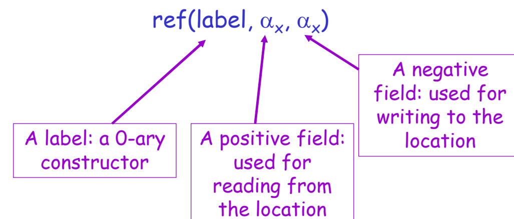
- In languages with side effects, want to know which locations may have *aliases*
  - More than one "name"
  - More than one pointer to them
- E.g.,

```
Y = &Z
X = Y
*X = 3     /* changes the value of *Y */
```


The Encoding of a Location

- For a program variable x:

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



Inference Rules

$$\frac{}{x : ref(l_x, \alpha_x, \alpha_x)} \qquad \frac{e : \tau}{\&e : ref(0, \tau, \tau)}$$

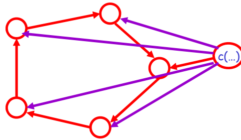
$$\frac{e : \tau \quad \tau \subseteq ref(1, \alpha, 0) \quad \alpha \text{ fresh}}{{}^*e : \alpha} \qquad \frac{e_1 : \tau_1 \quad e_2 : \tau_2 \quad \tau_1 \subseteq ref(1,1,\alpha) \;\; \alpha \text{ fresh} \quad \tau_2 \subseteq ref(1,\beta,0) \;\; \beta \text{ fresh} \quad \beta \subseteq \alpha}{e_1 = e_2 : \tau_2}$$

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 naïve implementations have failed

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
  - $x_1 \subseteq x_2 \subseteq \ldots \subseteq x_n \subseteq x_1$
  - 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
  - Collapse cycles to a single node
  - During constraint resolution
- Not obvious how to do this
  - Stopping the graph closure and doing a depth-first search of the entire graph adds $O(n)$ work in the inner loop of the algorithm

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

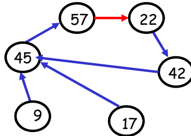
A Different Representation

- We change the representation of the graph
  - Assign every variable x (node) arbitray 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

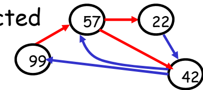
- On each edge addition (x,y)
  - If (x,y) is a successor edge (R(x) > R(y)) then search along predecessor edges from x.
    - When a node z s.t. R(z) < R(y) is found, prune that path
    - If y is found, a cycle is detected
  - If (x,y) is a predecessor edge (R(x) < R(y)) then search along successor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

Experience

- 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
- Asymptotic performance improvement
  - Now standard for (monomorphic) alias analysis