Inter-procedural Control Flow Analysis

Using Constraint-based Approach

The Dynamic Dispatch Problem

- Which function is called by p(x)?

```
int myFunc ( int (*p)(int), …)
{
……
return p(x);
}
```

  - P is a function pointer. What function could p point to (what is the value of p)?
  - P is a function parameter, so the value of p is unknown unless inter-procedural dataflow analysis is performed
    - But inter-procedural data-flow requires an inter-procedural control flow graph (or a call graph)
- The problem is relevant for
  - Imperative languages that allow functions as parameters
  - Object oriented languages and functional languages

Inter-procedural Control flow Analysis

- Example code

```
int f (int (*x)(int) { return x(1); }
int g (int y) { return y + 2; }
int h (int z) { return z + 3; }
int main() {
  return f(g) + f(h);
}
```

- For each function call, what functions may be invoked?

Defining the Analysis

- What is the domain of analysis
  - What is the solution space?
    - What could be the values for each function pointer expression?
- Specification of the analysis
  - How to compute the solution?
    - how to accommodate the information flow from function definitions to function invocations
- Well-definedness of the analysis
  - What are the properties of the solution space?
  - Does it compute a solution?
  - Does the algorithm terminate?
  - Is the solution precise?

Specification of Domain

- What is the solution?
  - For each expression in the program, could it have a function pointer value? If yes, what functions may it point to? (if no, the solution is $\emptyset$)
  - Must keep track of the values of variables (especially function parameters)
- To represent the solution, label each expression within the program, compute
  - An abstract cache (C) so that for each expression e,
    - C(e) contains the set of function values e may have
  - An abstract environment (P) so that for each variable x,
    - P(x) contains the set of function values x may have

The Input Language

- Assume a small functional language

```
e ::= c                  // constant values
   | x                   // variable reference
   | fun f x => e0 // function with name f, parameter x, and body 30
   | e1 e2               // invoking function e1 with argument e2
   | if e0 then e1 else e2   //if e0 is true, return e1, else return e2
   | let x = e1 in e2   // introduce local variable x=e1 in e2
```

- Why functional language?
  - Functions are first-class objects; allow nested functions/scopes
    - Can be used to model virtual functions in object-oriented programming
  - Dataflow is explicit (a single symbolic value for each variable). No variable is ever modified
    - For imperative programming languages, perform global data-flow analysis / build SSA

Example Code and Control-flow Analysis Solution

- Example code

  ((fun f x => x) (fun g y => y))

  - Labels: 1: x;

    2: (fun f x => x)

    3: y;

    4: (fun g y => y)

    5: ((fun f x => x) (fun g y => y))

- Example CFA solution (guesses of the (C,P) mappings)

1	{fun g y => y}
2	{fun f x => x}
3	∅
4	{fun g y => y}
5	{fun g y => y}
x	{ fun g y => y}
y	∅
f	{fun f x => x}
g	{ fun g y => y}

Solution Space of CFA

- Formally
  - Abstract values: Val = Power(Term)
    - Each term is a function definition in the form (fun f x => e0)
  - Abstract environment: Env = Var -> Val
    - Var: the set of all variables (including function parameters)
  - Abstract cache: Cache = Label -> Val
    - Label: the set of labels (expressions)
- Each solution: a pair of (P,C) $\subseteq$(Env, Cache)

Specification of CFA

- What properties must be satisfied by (P,C) to be a correct/acceptable solution?
  - (C,P) |= e means that (C,P) is an acceptable Control Flow Analysis Solution for the expression e
    - (C,P) |= c

      Arbitrary solutions are acceptable for a constant value c
    - (C,P) |= $(x)^{\ell}$ iff $P(x) \subseteq C(\ell)$

      The solution for an variable must be a subset of the solution for its label (each variable has a single value through each of its lifetime)
    - (C,P) |= (fun f x => $(e0)^{\ell_0})^{\ell_1}$ iff (C,P) |= $(e0)^{\ell_0}$ and

      {fun f x => e0} $\subseteq C(\ell_1)$ and {fun f x => e0} $\subseteq$ P(f)

      The solution for a function definition(abstraction) label must include the function definition(abstraction)

Specification of CFA (2)

- Function invocation (application)
  - (C,P) |= $((e1)^{l_1}\ (e2)^{l_2})^{l_3}$ iff (C,P) |= $(e1)^{l_1}$, (C,P) |= $(e2)^{l_2}$, and

    $\forall$ (fun f x => $(e0)^{l_0}) \in C(l_1)$: (C,P)|=$(e0)^{l_0}$, $C(l_2) \subseteq P(x)$ and $C(l_0) \subseteq C(l_2)$
    - The solution for function parameter (x) must contain that of the invocation argument (e2);
    - The solution of the function invocation must contain that of the function body
- Local variables (nested scopes)
  - (C,P) |= (let x = $(e1)^{l_1}$ in $(e2)^{l_2})^{l_3}$ iff (C,P) |= $(e1)^{l_1}$, (C,P) |= $(e2)^{l_2}$,

    $C(l_1) \subseteq P(x)$ and $C(l_2) \subseteq C(l_3)$
    - The solution for the local variable (x) must contain that of its defined value
    - The solution of the outer scope must contain that of the inner scope
- Conditionals
  - (C,P) |= (if $(e0)^{l_0}$ then $(e1)^{l_1}$ else $(e2)^{l_2})^{l_3}$ iff (C,P) |= $(e0)^{l_0}$, (C,P) |= $(e1)^{l_1}$, (C,P) |= $(e2)^{l_2}$, and $C(l_2) \subseteq C(l_3)$ and $C(l_2) \subseteq C(l_3)$
    - The solution of the outer scope must contain that of the inner scopes (both branches)

Example Code and Control-flow Analysis Solution

- Example code

  ((fun f x => x) (fun g y => y))

  - Labels: 1: x;

    2: (fun f x => x)

    3: y;

    4: (fun g y => y)

    5: ((fun f x => x) (fun g y => y))

- Example CFA solution (guesses of the (C,P) mappings). Are the valid?

	(C,P)	(C',P')
1	{fun g y => y}	{fun g y => y}
2	{fun f x => x}	{fun f x => x}
3	∅	∅
4	{fun g y => y}	{fun g y => y}
5	{fun g y => y}	{fun g y => y}
x	{ fun g y => y}	∅
y	∅	∅
f	{fun f x => x}	{fun f x => x}
g	{fun g y => y}	{fun g y => y}

(C,P) $\models$ ((fun f x => x) (fun g y => y))

(C',P') $\not\models$ ((fun f x => x) (fun g y => y))

Well-definedness of CFA Analysis

- Difficulty: Cannot build (C,P) |= e by structural induction on the expression e
  - E.g. function invocation (application)

    (C,P) |= $((e1)^{\ell_1}\ (e2)^{\ell_2})^{\ell_3}$ iff (C,P) |= $(e1)^{\ell_1}$, (C,P) |= $(e2)^{\ell_2}$, and

    $\forall$ (fun f x => $(e0)^{\ell_0}$) $\in C(\ell_1)$, (C,P) |= $(e0)^{\ell_0}$, $C(\ell_2) \subseteq P(x)$ and $C(\ell_0) \subseteq C(\ell_2)$

    - There is no guarantee that $C(\ell_0)$ has been computed correctly before computing $C(\ell_2)$
- Coinductive definition: the solution space includes all guesses of (C,P) that satisfy the specifications
  - Must apply all constraints to iteratively modify the solutions until they become correct
  - The best solution is the smallest one that satisfies all the constraints

Correctness of Specification

- If there is a possible evaluation of the program such that the function at a call point evaluates to some function definition
  - then this definition has to be in the set of possible definitions computed by the analysis.

- Existence of solutions
  - Every expression accepts a least CFA solution

Constraint based Analysis

- Syntax-directed analysis
  - Reformulate the analysis specification
    - Construct a finite set of constraints based on structural induction
  - Compute the least solution of the set of constraints
- Each constraint has the form

  (sol1 $\subseteq$ sol2) or ({t} $\subseteq$ sol) or ({t} $\subseteq$ sol1 => sol2 $\subseteq$ sol3)
  - where
    - Each sol is either C(ℓ) or P(x)
      - ℓ is label, x is a variable
    - Each t is either (fn x => e0) or (fun f x => e0)

Constraint-based Analysis

- For each expression e, compute Cond[e]
  - Cond[c] = $\emptyset$ //constants
  - Cond[(x)ℓ] = { P(x) $\subseteq$ C(ℓ) } // variables
  - Cond[(fun f x => e0)ℓ] = Cond[e0] $\cup$

    { {fun f x=>e0}$\subseteq$ C(ℓ) } $\cup$ { {fun f x => e0} $\subseteq$ P(f) } // function def.
  - Cond[((e1)ℓ_1 (e2)ℓ_2)ℓ_3] = Cond[e1] $\cup$ Cond[e2] $\cup$

{ {t} $\in$ C(ℓ_1)=>C(ℓ_2) $\subseteq$ P(x) $\forall$ t = (fun f x => (e0)ℓ_0) } $\cup$

{ {t} $\in$ C(ℓ_1)=> C(ℓ_0) $\subseteq$ C(ℓ_3) $\forall$ t = (fun f x => (e0)ℓ_0) }

  - Cond[(let x = (e1)ℓ_1 in (e2)ℓ_2)ℓ_3] =

    Cond[e1] $\cup$ Cond[e2] $\cup$ {C(ℓ_1) $\subseteq$ P(x)} $\cup$ {C(ℓ_2) $\subseteq$ C(ℓ_3)}
  - Cond [(if (e0)ℓ_0 then (e1)ℓ_1 else (e2)ℓ_2)ℓ_3] =

    Cond[e0] $\cup$ Cond[e1] $\cup$ Cond[e2] $\cup$ {C(ℓ_2) $\subseteq$ C(ℓ_3)} $\cup$ {C(ℓ_2) $\subseteq$ C(ℓ_3) }

Example: Constraint Construction

Cond[((fun f x => (x)1)2 (fun g y => (y)3)4)5]

= { {fun f x => (x)} $\subseteq$ C(2), {fun f x => (x)} $\subseteq$ P(f),

P(x) $\subseteq$ C(1),

{fun g y => (y)}$\subseteq$C(4), {fun g y => (y)}$\subseteq$P(g),

P(y) $\subseteq$ C(3),

{fun f x => (x)} $\subseteq$ C(2) => C(4) $\subseteq$ P(x),

{fun f x => (x)} $\subseteq$ C(2) => C(1) $\subseteq$ C(5),

{fun g y => (y)} $\subseteq$ C(2) => C(4) $\subseteq$ P(y),

{fun g y => (y)} $\subseteq$ C(2) => C(3) $\subseteq$ C(5) }

Solving the constraints

- Input: a set of constraints for the entire program
- Output: the least solution (C,P) to the constraints
- Idea: equivalent to finding the least fixed point of a monotone function defined by the constraints
  - Straight-forward iterative algorithm has n^5 cost, where n is the size of the program (expression)
  - A more sophisticated algorithm takes n^3 complexity

- The graph-based algorithm
  - Build a graph where
    - Each node n corresponds to a unique C(ℓ) or P(x) =>val(n)
    - Add an edge from node n1 to n2 if any change to val(n1) may require modifications to val(n2)
  - Use a worklist to keep track of nodes to change

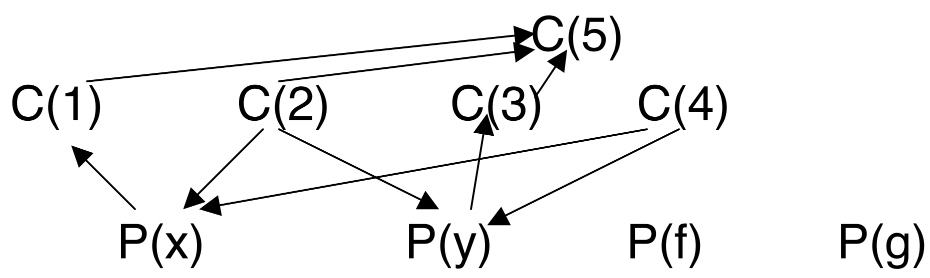
Constraint Solving Algorithm (1)

- Define add(t, p) = { if (t $\not\subseteq$ p) { p = p $\cup$ t; append(p,worklist);} }
- Step 1 Initialization
  - worklist := nil;
  - for each label ℓ (or variable x) do
    - Val[C(ℓ)] = nil; Edge(C(ℓ)) = nil; (or Val[P(x)] = nil; Edge(P(x)) = nil)
- Step 2 Building the graph
  - for each cc in Cond[program] do

    case cc of {t} $\subseteq$ p: add(t,Val(p));

    p1 $\subseteq$ p2: append(cc, Edge[p1]);

    {t} $\subseteq$ p => p1 $\subseteq$ p2: append(cc,Edge[p1]); append(cc,Edge[p]);



Constraint Solving Algorithm(2)

- Step 3 Iteration
  - while worklist is not empty do

    q := Remove-first(W);

    for each cc in Edge[q] do

    case cc of p1 $\subseteq$ p2: add(p2, Val[p1]);

    {t} $\subseteq$ p => p1 $\subseteq$ p2: if t $\subseteq$ Val[p] then add(val(p1), p2);

Val	Iteration 0	Propogate C(2)…	Propoage P(x)	Propoage C(1)
C(1)	$\varnothing$	$\varnothing$	{fun g y => y}	{fun g y => y}
C(2)	{fun f x => x}	{fun f x => x}	{fun f x => x}	{fun f x => x}
C(3)	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
C(4)	{fun g y => y}	{fun g y => y}	{fun g y => y}	{fun g y => y}
C(5)	$\varnothing$	$\varnothing$	$\varnothing$	{fun g y => y}
P(x)	$\varnothing$	{fun g y => y}	{fun g y => y}	{fun g y => y}
P(y)	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
P(f)	{fun f x => x}	{fun f x => x}	{fun f x => x}	{fun f x => x}
P(g)	{fun g y => y}	{fun g y => y}	{fun g y => y}	{fun g y => y}

Summary

- Recording the solution of CFA analysis
  - for each label ℓ (or variable x) do
    - C(ℓ) = Val[C(ℓ)] (P(x) = Val[P(x)])

- Correctness and Termination
  - The worklist algorithm terminates and the result produced by the algorithm is the least solution to C[[e]].
  - Complexity: The algorithm takes at most O(n3) steps if the original expression e has size n.