Inter-procedural Control Flow Analysis

Using Constraint-based Approach
The Dynamic Dispatch Problem

- Which function is called by \( p(x) \)?
  ```c
  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

```c
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
  
  \[ l \ x \]  // variable reference
  
  \[ l \ \text{fun} \ f \ x \Rightarrow e0 \] // function with name f, parameter x, and body 30
  
  \[ l \ e1 \ e2 \]  // invoking function e1 with argument e2
  
  \[ l \ 	ext{if} \ e0 \ 	ext{then} \ e1 \ 	ext{else} \ e2 \]  // if e0 is true, return e1, else return e2
  
  \[ l \ 	ext{let} \ x = e1 \ 	ext{in} \ e2 \]  // 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
  \[
  ((\text{fun } f \, x \Rightarrow x) \, (\text{fun } g \, y \Rightarrow y))
  \]
  - Labels: 1: \( x \);
    2: (fun f x => x)
    3: \( y \);
    4: (fun g y => y)
    5: ((fun f x => x) \, (\text{fun } g \, y \Rightarrow y))

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

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{fun g y =&gt; y}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{fun f x =&gt; x}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{fun g y =&gt; y}</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{fun g y =&gt; y}</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>{ fun g y =&gt; y}</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>{fun f x =&gt; x}</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>{ fun g y =&gt; y}</td>
<td></td>
</tr>
</tbody>
</table>
Solution Space of CFA

- Formally
  - Abstract values: \( \text{Val} = \text{Power} (\text{Term}) \)
    - Each term is a function definition in the form (\text{fun f x => e0})
  - Abstract environment: \( \text{Env} = \text{Var} \rightarrow \text{Val} \)
    - \text{Var}: the set of all variables (including function parameters)
  - Abstract cache: \( \text{Cache} = \text{Label} \rightarrow \text{Val} \)
    - \text{Label}: the set of labels (expressions)
  - Each solution: a pair of \((P,C) \subseteq (\text{Env}, \text{Cache})\)
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)l iff P(x) ⊆ C(l)
    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)l0)l1 iff (C,P) |= (e0)l0 and
    {fun f x => e0} ⊆ C(l1) and {fun f x => e0} ⊆ P(f)
    The solution for a function definition(abstraction) label must include the function definition(abstraction)
Specification of CFA (2)

- **Function invocation (application)**
  \[(C,P) \vdash ((e_1)_{l_1} (e_2)_{l_2})_{l_3}\] iff \[(C,P) \vdash (e_1)_{l_1}, (C,P) \vdash (e_2)_{l_2},\] and
  \[\forall (\text{fun } f \ x \Rightarrow (e_0)_{l_0}) \in C(l_1): (C,P)\vdash(e_0)_{l_0}, C(l_2) \subseteq P(x) \text{ and } C(l_0) \subseteq C(l_2)\]
  - The solution for function parameter \(x\) must contain that of the invocation argument \((e_2)\);
  - The solution of the function invocation must contain that of the function body

- **Local variables (nested scopes)**
  \[(C,P) \vdash (\text{let } x = (e_1)_{l_1} \text{ in } (e_2)_{l_2})_{l_3}\] iff \[(C,P) \vdash (e_1)_{l_1}, (C,P) \vdash (e_2)_{l_2},\]
  \[C(l_1) \subseteq P(x) \text{ and } C(l_2) \subseteq C(l_3)\]
  - 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**
  \[(C,P) \vdash (\text{if } (e_0)_{l_0} \text{ then } (e_1)_{l_1} \text{ else } (e_2)_{l_2})_{l_3}\] iff \[(C,P) \vdash (e_0)_{l_0}, (C,P) \vdash (e_1)_{l_1},\]
  \[(C,P) \vdash (e_2)_{l_2}, \text{ and } C(l_2) \subseteq C(l_3) \text{ and } C(l_2) \subseteq C(l_3)\]
  - 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’) | (C,P) |= ((fun f x => x) (fun g y => y)) | (C’,P’) |= ((fun f x => x) (fun g y => y)) |
  |--------|--------|-----------------|-----------------|
  | 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} |                |                      |
Well-definedness of CFA Analysis

- **Difficulty**: Cannot build \((C,P) \models e\) by structural induction on the expression \(e\)
  - E.g. function invocation (application)
    \[
    (C,P) \models ((e_1)l_1 (e_2)l_2)l_3 \text{ iff } (C,P) \models (e_1)l_1, (C,P) \models (e_2)l_2, \text{ and }
    \]
    \[
    \forall \ (\text{fun } f \ x => (e_0)l_0) \in C(l_1), (C,P) \models (e_0)l_0, C(l_2) \subseteq P(x) \text{ and } C(l_0) \subseteq C(l_2)
    \]
  - There is no guarantee that \(C(l_0)\) has been computed correctly before computing \(C(l_2)\)

- 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
  \[(\text{sol1} \subseteq \text{sol2}) \text{ or } (\{t\} \subseteq \text{sol}) \text{ or } (\{t\} \subseteq \text{sol1} \Rightarrow \text{sol2} \subseteq \text{sol3})\]
  - where
    - Each sol is either \(C(l)\) or \(P(x)\)
      - \(l\) is label, \(x\) is a variable
    - Each \(t\) is either \((\text{fn } x \Rightarrow e0)\) or \((\text{fun } f x \Rightarrow e0)\)
Constraint-based Analysis

For each expression e, compute Cond[e]

- Cond[c] = ∅  // constants
- Cond[(x)]: = { P(x) ⊆ C(l) }  // variables
- Cond[(fun f x => e0)]: = Cond[e0] ∪
  \{ \{fun f x => e0\} ⊆ C(l) \} ∪ \{ \{fun f x => e0\} ⊆ P(f) \}  // function def.
- Cond[(e1) (e2) (e3)] = Cond[e1] ∪ Cond[e2] ∪
  \{ \{t\} ∈ C(l1) => C(l2) ⊆ P(x) ∀ t = (fun f x => (e0)\} ∪
  \{ \{t\} ∈ C(l1) => C(l0) ⊆ C(l3) ∀ t = (fun f x => (e0)\} \\
- Cond[(let x = (e1) in (e2) (e3) ] =
  Cond[e1] ∪ Cond[e2] ∪ \{C(l1) ⊆ P(x)\} ∪ \{C(l2) ⊆ C(l3)\}
- Cond [(if (e0) then (e1) else (e2) (e3) ] =
  Cond[e0] ∪ Cond[e1] ∪ Cond[e2] ∪ \{C(l2) ⊆ C(l3)\} ∪ \{C(l2) ⊆ C(l3)\}
Example: Constraint Construction

\[
\text{Cond}[(\text{fun } f \ x \Rightarrow (x)1)2 \ (\text{fun } g \ y \Rightarrow (y)3)4 )5]
\]
\[
= \{ \{\text{fun } f \ x \Rightarrow (x)\} \subseteq C(2), \{\text{fun } f \ x \Rightarrow (x)\} \subseteq P(f), \\
\quad P(x) \subseteq C(1), \\
\quad \{\text{fun } g \ y \Rightarrow (y)\} \subseteq C(4), \{\text{fun } g \ y \Rightarrow (y)\} \subseteq P(g), \\
\quad P(y) \subseteq C(3), \\
\quad \{\text{fun } f \ x \Rightarrow (x)\} \subseteq C(2) \Rightarrow C(4) \subseteq P(x), \\
\quad \{\text{fun } f \ x \Rightarrow (x)\} \subseteq C(2) \Rightarrow C(1) \subseteq C(5), \\
\quad \{\text{fun } g \ y \Rightarrow (y)\} \subseteq C(2) \Rightarrow C(4) \subseteq P(y), \\
\quad \{\text{fun } g \ y \Rightarrow (y)\} \subseteq C(2) \Rightarrow C(3) \subseteq C(5) \} 
\]
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 \( C(\hat{l}) \) or \( P(x) \Rightarrow \text{val}(n) \)
  - Add an edge from node \( n_1 \) to \( n_2 \) if any change to \( \text{val}(n_1) \) may require modifications to \( \text{val}(n_2) \)
- Use a worklist to keep track of nodes to change
**Constraint Solving Algorithm (1)**

- Define \( \text{add}(t, p) = \{ \text{if } (t \not\subseteq p) \{ p = p \cup t; \text{append}(p, \text{worklist}); \}} \} \}

- **Step 1 Initialization**
  - worklist := nil;
  - for each label \( l \) (or variable \( x \)) do
    - \( \text{Val}[C(l)] = \text{nil}; \text{Edge}(C(l)) = \text{nil}; \) (or \( \text{Val}[P(x)] = \text{nil}; \text{Edge}(P(x)) = \text{nil} \))

- **Step 2 Building the graph**
  - for each \( cc \) in \( \text{Cond}[\text{program}] \) do
    - case \( cc \) of
      - \( \{t\} \subseteq p: \text{add}(t, \text{Val}(p)) \);
        - \( p1 \subseteq p2: \text{append}(cc, \text{Edge}[p1]) \);
        - \( \{t\} \subseteq p \Rightarrow p1 \subseteq p2: \text{append}(cc, \text{Edge}[p1]); \text{append}(cc, \text{Edge}[p]) \);

![Diagram of constraint graph](image)
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);

<table>
<thead>
<tr>
<th>Val</th>
<th>Iteration 0</th>
<th>Propogate C(2)...</th>
<th>Propogate P(x)</th>
<th>Propogate C(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(1)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>{fun g y =&gt; y}</td>
<td>{fun g y =&gt; y}</td>
</tr>
<tr>
<td>C(2)</td>
<td>{fun f x =&gt; x}</td>
<td>{fun f x =&gt; x}</td>
<td>{fun f x =&gt; x}</td>
<td>{fun f x =&gt; x}</td>
</tr>
<tr>
<td>C(3)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>C(4)</td>
<td>{fun g y =&gt; y}</td>
<td>{fun g y =&gt; y}</td>
<td>{fun g y =&gt; y}</td>
<td>{fun g y =&gt; y}</td>
</tr>
<tr>
<td>C(5)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>{fun g y =&gt; y}</td>
</tr>
<tr>
<td>P(x)</td>
<td>(\emptyset)</td>
<td>{fun g y =&gt; y}</td>
<td>{fun g y =&gt; y}</td>
<td>{fun g y =&gt; y}</td>
</tr>
<tr>
<td>P(y)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>P(f)</td>
<td>{fun f x =&gt; x}</td>
<td>{fun f x =&gt; x}</td>
<td>{fun f x =&gt; x}</td>
<td>{fun f x =&gt; x}</td>
</tr>
<tr>
<td>P(g)</td>
<td>{fun g y =&gt; y}</td>
<td>{fun g y =&gt; y}</td>
<td>{fun g y =&gt; y}</td>
<td>{fun g y =&gt; y}</td>
</tr>
</tbody>
</table>
Summary

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

- 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 \( O(n^3) \) steps if the original expression \( e \) has size \( n \).