Background

- So far, we have emphasized so-called “data-flow frameworks” for describing static analyses.
- In this lecture, we will consider “constraint-based analyses” (CBA) – actually, a different style of expressing static analyses.
- The CBA style has its origins in the programming language community (specifically, the functional programming language community).

Background

- CBA is a lower-level framework
  - Instead of working at the level of transfer functions where are tied to specific types of abstract syntax tree nodes, CBA expresses flow information by building a graph whose structure is not as directly tied to the CFG/AST.
  - This makes CBA engines simpler because they can work on this lower-level representation with only one or two types of arcs expressing value flow.
  - The textbook argues that this lower-level presentation also makes it easier for the flow-graph to grow/evolve dynamically – which is required for dealing with functional and OO languages where a precise call-graph cannot be determined without considering some value flow information.

Background

- CBA is less structured
  - It is not a framework that can just be parameterized with (forwards/backwards), (GFP/LFP), (intra/inter procedural), (flow sensitive/insensitive), etc. As presented in the literature, these things must be coded by hand although there have been some attempts to create frameworks.
  - CBA has typically been used to express closure analysis (call-graph construction for functional langauges), type inference algorithms, etc.

Background: Flow-sensitive vs Flow-insensitive

- DFA is oriented toward flow sensitive analysis
  - ...hence the “flow” in DFA
  - it is possible (although awkward) to hack a flow insensitive
- CBA is oriented toward flow-insensitive analysis
  - it is not too hard to formulate a flow sensitive analysis by replicating /specializing constraint system variables for program components and then exploiting information about def-use of those components
  - SSA gives you this for value flow problems

Simplified Overview of Methodology

- Choose type of constrains (equality constraints, inclusion constraints, etc).
- Describe a syntax-directed generation of constraints
- Solve constraints (e.g., using a work-list algorithm)

Initial Example

- To illustrate the basic steps in a constraint-based analysis, we will consider an (unrealistic) analysis that propagates sets of integer values — each set will represent the set of variables that may flow into a particular program variable or AST node.
- Later on, we will consider an object-flow analysis for an OO language that acts as a relatively precise call-graph construction algorithm

Basic Scenario for a Flow Insensitive CBA

- Use a flowgraph node (AKA constraint set variable) for each
  - program variable
  - AST node
- In the set-based analyses that we will consider, each node will hold a set of flow values
- For each syntax construct C, write a definition of how to generate constraints for C-shaped AST nodes
- Use a work-list algorithm to solve constraints
- This yields final value flow information for each program variable and AST node

Components (following text)

Values flowing through the graph

- $\hat{v} \in \widehat{\textbf{Val}} = \mathcal{P}(\textsf{Int} \cup \{\top\})$
- $\top$ represents "any integer value". All integer expressions will return $\top$ because we will not attempt to do anything smarter in this simple example analysis.

The two types of nodes (variable nodes, AST nodes) in the value flow-graph are represented by the following structures

- $\hat{\rho} \in \widehat{\textsf{Env}} = \textbf{Var} \to \widehat{\textbf{Val}}$
- $\hat{\textsf{C}} \in \widehat{\textsf{Cache}} = \textbf{Lab} \to \widehat{\textbf{Val}}$

Example Program

Program

```
x := [2]^1;
y := [3]^2;
if [[x]^4 > [y]^5]^3
then
  z := [2]^6;
else
  y := [x]^7;
endif;
z := [[3]^9 + [2]^10]^8
```

Constraints

$$\{2\} \subseteq \hat{\mathsf{C}}(1), \hat{\mathsf{C}}(1) \subseteq \hat{\rho}(x)$$

$$\{3\} \subseteq \hat{\mathsf{C}}(2), \hat{\mathsf{C}}(2) \subseteq \hat{\rho}(y)$$

$$\hat{\rho}(x) \subseteq \hat{\mathsf{C}}(4), \hat{\rho}(y) \subseteq \hat{\mathsf{C}}(5),$$

$$\{\top\} \subseteq \hat{\mathsf{C}}(3)$$

$$\{2\} \subseteq \hat{\mathsf{C}}(6), \hat{\mathsf{C}}(6) \subseteq \hat{\rho}(z)$$

$$\hat{\rho}(x) \subseteq \hat{\mathsf{C}}(7), \hat{\mathsf{C}}(7) \subseteq \hat{\rho}(y)$$

$$\{3\} \subseteq \hat{\mathsf{C}}(9), \{2\} \subseteq \hat{\mathsf{C}}(10),$$

$$\{\top\} \subseteq \hat{\mathsf{C}}(8), \hat{\mathsf{C}}(8) \subseteq \hat{\rho}(z)$$

Syntax-directed Constraint Generation (following text)

$$
\begin{array}{lrcl}
\text{[con]} & \mathcal{C}_*[\![n^l]\!] & = & \{\{n\} \subseteq \hat{\mathsf{C}}(l)\} \\
\text{[var]} & \mathcal{C}_*[\![x^l]\!] & = & \{\hat{\rho}(x) \subseteq \hat{\mathsf{C}}(l)\} \\
\text{[op]} & \mathcal{C}_*[\![(e_1 \,\mathsf{op}\, e_2)^l]\!] & = & \{\{\top\} \subseteq \hat{\mathsf{C}}(l)\} \cup \mathcal{C}_*[\![e_1]\!] \cup \mathcal{C}_*[\![e_2]\!] \\
\text{[assign]} & \mathcal{C}_*[\![x := e^l]\!] & = & \{\hat{\mathsf{C}}(l) \subseteq \hat{\mathsf{C}}(x)\} \cup \mathcal{C}_*[\![e]\!] \\
\text{[if]} & \mathcal{C}_*[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!] & = & \mathcal{C}_*[\![e_1]\!] \cup \mathcal{C}_*[\![e_2]\!] \cup \mathcal{C}_*[\![e_3]\!]
\end{array}
$$

Assessment

- Dataflow functions captured transformation of “cache” (dataflow facts) across AST nodes.
- CBA constraints directly capture flow of information from producer AST nodes to consumer AST nodes.
- Both are syntax-directed, but CBA “wires things together” at a finer level of granularity.
- You can think of the function $\mathcal{C}_*$ as compiling an AST to a constraint set which is then interpreted by the constraint solver.

Coming up: OO Control-flow Graph Construction

Challenges:

- Inheritance and method override make difficult to determine interprocedural control-flow and data-flow.

```
o.foo(x,y,z);  /* virtual method call */
```

- All Bandera components need to know (as precisely as possible) the method code that could be executed due to this invocation

Original Goal for Bandera Call-graph Construction:

- A highly flexible, parameterized analysis to detect this information (dramatic generalization of interprocedural points-to analysis).

Coming up: Bandera Object Flow Analysis (BOFA)

What information do we want from the analysis?

- For each AST node, determine the class of each object that flows out of that node
  - sometimes more precise: at what site was the object allocated
  - even more precise: on what loop iteration (or on what sequence of method invocations) was an object allocated.