Control-flow Analysis

Letterio Galletta

Agenda

- Control-flow graph
- Control-flow analysis

Reference: Chapter 3 of Principles of Program Analysis by Nielson, Nielson, Hankin

Control-flow Graphs

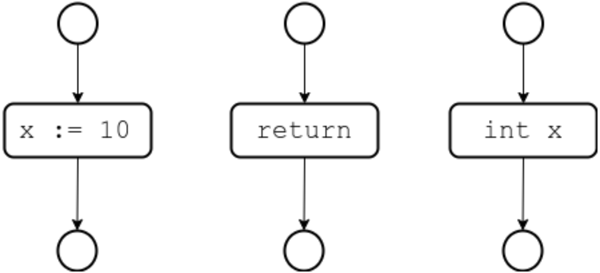
- A control-flow graph (CFG) is a direct graph where nodes corresponds to basic blocks (sequence of simple statements) and edges represent possible control flow
- The CFG provides a graphical representation of the possible runtime control-flow paths
- Typically, the statements inside basic blocks are represented with another IR
- CGF is used to perform Dataflow analysis and program transformation

CFG Construction: Intuition

- We assume that each CFG has *single* entry node
- For the time being, we consider a single procedure at the time and that basic blocks are made of single statements
- The construction is defined inductively on the structure of the statements

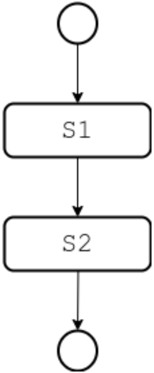
CFG Construction: Intuition

For simple statements, e.g., assignment, return statement, declaration, the CFG looks like



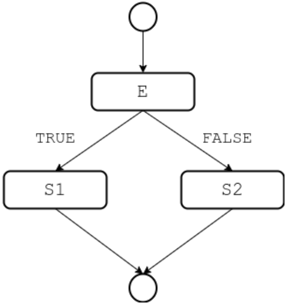
CFG Construction: Intuition

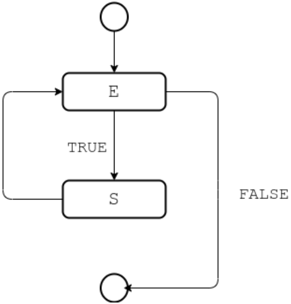
For sequential composition $S_1; S_2$ we build the CFG for S_1 and S_2 inductively, remove the exit node of S_1 and the entry node of S_2 and glue the statements together



CFG Construction: Intuition

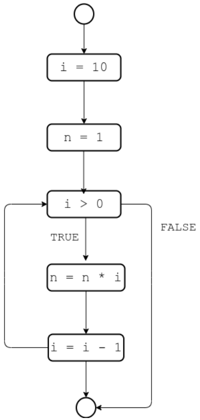
Similarly, for other control flow constructs (we label branches with true and false)





CFG Construction: Example

```
i = 10;
n = 1;
while(i > 0){
  n = n * i;
  i = i - 1;
}
```



CFG Construction (intraprocedural)

$$x \in Id \quad e \in Exp \quad \ell \in Lab$$

$$S ::= [x := e]^\ell \mid [skip]^\ell \mid S_1; S_2 \mid \textbf{while } [e]^\ell \textbf{ do } S \mid \textbf{if } [e]^\ell \textbf{ then } S_1 \textbf{ else } S_2$$

$$\begin{aligned}
init([x := e]^\ell) &= \ell \\
init([skip]^\ell) &= \ell \\
init(S_1; S_2) &= init(S_1) \\
init(\textbf{while } [e]^\ell \textbf{ do } S) &= \ell \\
init(\textbf{if } [e]^\ell \textbf{ then } S_1 \textbf{ else } S_2) &= \ell
\end{aligned}$$

$$\begin{aligned}
final([x := e]^\ell) &= \ell \\
final([skip]^\ell) &= \ell \\
final(S_1; S_2) &= final(S_2) \\
final(\textbf{while } [e]^\ell \textbf{ do } S) &= \ell \\
final(\textbf{if } [e]^\ell \textbf{ then } S_1 \textbf{ else } S_2) &= final(S_1) \cup final(S_2)
\end{aligned}$$

CFG Construction

$$x \in \mathit{Id} \quad e \in \mathit{Exp} \quad \ell \in \mathit{Lab}$$

$$S ::= [x := e]^\ell \mid [\mathit{skip}]^\ell \mid S_1; S_2 \mid \textbf{while } [e]^\ell \textbf{ do } S \mid \textbf{if } [e]^\ell \textbf{ then } S_1 \textbf{ else } S_2$$

$$\begin{aligned}
\mathit{flow}([x := e]^\ell) &= \emptyset \\
\mathit{flow}([\mathit{skip}]^\ell) &= \emptyset \\
\mathit{flow}(S_1; S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_2)), \mid \ell \in \mathit{final}(S_1)\} \\
\mathit{flow}(\textbf{while } [e]^\ell \textbf{ do } S) &= \mathit{flow}(S) \cup \{(\ell, \mathit{init}(S))\} \\
\mathit{flow}(\textbf{if } [e]^\ell \textbf{ then } S_1 \textbf{ else } S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \{(\ell, \mathit{init}(S_1)), (\ell, \mathit{init}(S_2))\}
\end{aligned}$$

CFG Construction

$$x \in \mathit{Id} \quad e \in \mathit{Exp} \quad \ell \in \mathit{Lab}$$

$$S ::= [x := e]^\ell \mid [\mathit{skip}]^\ell \mid S_1; S_2 \mid \textbf{while } [e]^\ell \textbf{ do } S \mid \textbf{if } [e]^\ell \textbf{ then } S_1 \textbf{ else } S_2$$

$$\begin{aligned}
\mathit{blocks}([x := e]^\ell) &= \{[x := e]^\ell\} \\
\mathit{blocks}([\mathit{skip}]^\ell) &= \{[\mathit{skip}]^\ell\} \\
\mathit{blocks}(S_1; S_2) &= \mathit{blocks}(S_1) \cup \mathit{blocks}(S_2) \\
\mathit{blocks}(\textbf{while } [e]^\ell \textbf{ do } S) &= \mathit{blocks}(S) \cup \{[e]^\ell\} \\
\mathit{blocks}(\textbf{if } [e]^\ell \textbf{ then } S_1 \textbf{ else } S_2) &= \mathit{blocks}(S_1) \cup \mathit{blocks}(S_2) \cup \{[e]^\ell\}
\end{aligned}$$

CFG Construction

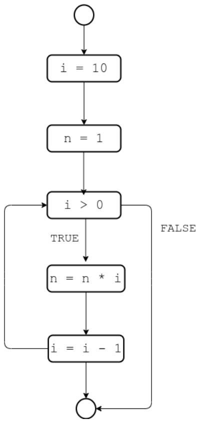
$$x \in \mathit{Id} \quad e \in \mathit{Exp} \quad \ell \in \mathit{Lab}$$

$$S ::= [x := e]^\ell \mid [\mathit{skip}]^\ell \mid S_1; S_2 \mid \textbf{while } [e]^\ell \textbf{ do } S \mid \textbf{if } [e]^\ell \textbf{ then } S_1 \textbf{ else } S_2$$

The control-flow graph of a statement S has the $\mathit{blocks}(S)$ as nodes and $\mathit{flow}(S)$ as arcs

CFG Construction: Example

```
S = [i = 10]^1;
    [n = 1]^2;
    while([i > 0]^3){
        [n = n * i]^4;
        [i = i - 1]^5;
    }
```



$init(S) = 1$

$final(S) = \{3\}$

$flow(S) = \{(1,2),(2,3),(3,4)(4,5)\}$

CFG Construction (interprocedural)

When we consider procedures

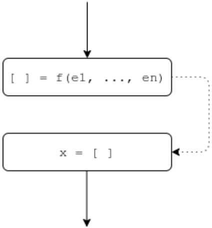
- Build the CFG for all functions bodies
- Glue them together to reflect the different function calls

CFG Construction: calling functions

In a CFG a function call is represented using two nodes:

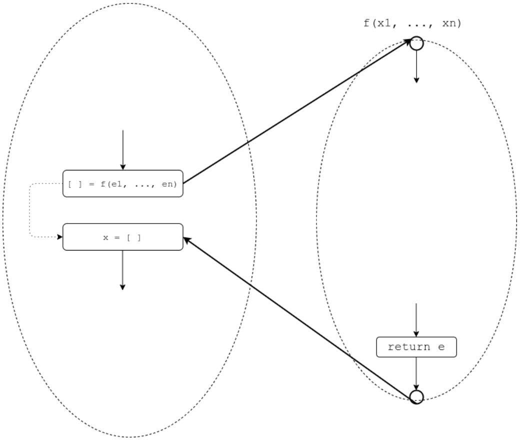
1. a *call-node* representing the connection from the caller to the entry point of the callee
2. a *after-call* node where the execution may resume after the termination of the callee

$$x = f(e_1, \ldots, e_n);$$



CFG Construction: interprocedural

Glue together the caller and the callee



Control-flow analysis (CFA)

Building the CFG of a program is not easy in real languages:

- First-order functions (functional languages)
- Function pointers/procedures as parameters (imperative languages)
- Dynamic dispatch (object-oriented languages)

The actual target of a call is resolved at run-time

Solution: a static analysis to over-approximate the interprocedural control flow of the program

Dynamic dispatch problem

Consider the following snippet of code

```
let f x = x 1
and g y = y + 2
and h z = z + 3 ;;
(f g) + (f h) ;;
```

the function `f` will transfer the code to its formal parameter `x`: the inter-procedural control flow depends on the values `x` denotes

The fun language

$e \in \mathit{Exp} ::= t^{\ell}$	annotated expressions
$t \in \mathit{Term} ::= c$	constant literal
$\mid x$	identifier
$\mid \textbf{if } e_1 \textbf{ then } e_1 \textbf{ else } e_2$	conditionals
$\mid e_1 \diamond e_2$	$\diamond \in \{+, -, *, \ldots\}$ primitive operators
$\mid \textbf{let } x = e_1 \textbf{ in } e_2$	declarations
$\mid \textbf{fun } x \to e$	lambda abstraction
$\mid e_1\ e_2$	function application

The analysis of fun

The goal

- Determine for each sub-expression e a set of function F, e may evaluates to at run-time
- Determine where the flow of control may be transferred when e is a function application

Example (Consider the OCaml expression)

```
(if e then (( * ) 2) else ((+) 1)) 10
```

depending on the value of `e` either `(( * ) 2)` or `((+) 1)` is applied

Example

Consider the expression

$$((\,\textbf{fun}\, x \to x^1)^2(\,\textbf{fun}\, y \to y^3)^4)^5$$

We want to statically compute that

- x is bound to $\textbf{fun}\, y \to y^3$
- the sub-expression at label 1 evaluates to $\textbf{fun}\, y \to y^3$
- the overall expression evaluates to to $\textbf{fun}\, y \to y^3$
- y is never bound to a value

CFA via Flow Logic

Flow Logic is a declarative approach to program analysis that separates the specification from the actual computation.

specification we describe when the analysis results (estimates) are acceptable: the specification consists of a set of clauses defining an acceptability relation

computation from the specification we derive a constraint satisfaction problem: solving the constraints corresponds to compute the analysis.

Our plan for CFA

We define our CFA in three steps:

1. We define an acceptability relation over the syntax of FUN (syntax-directed specification)
2. We turn the syntax-directed specification into an algorithm for computing a finite set of constraints
3. We compute the least solution of the constraints

Abstract domains

The result of a CFA analysis is a pair $(\hat{C}, \hat{\rho})$:

- $\hat{C}$ is the abstract cache associating abstract values with each labelled program point
- $\hat{\rho}$ is the abstract environment associating abstract values with each variable

An abstract value is a set of functions, i.e., terms of the form $\mathbf{fun}\, x \to e$

We assume that all variables and labels are distinct (we can ensure that while building the AST of the program)

Example of a valid estimate

Consider the expression

$$((\textbf{fun}\, x \to x^1)^2(\textbf{fun}\, y \to y^3)^4)^5$$

	$\hat{C}$
1	$\textbf{fun}\, y \to y$
2	$\textbf{fun}\, x \to x$
3	$\emptyset$
4	$\textbf{fun}\, y \to y$
5	$\textbf{fun}\, y \to y$

	$\hat{\rho}$
x	$\textbf{fun}\, y \to y$
y	$\emptyset$

Acceptability relation

The idea: assume to guess a pair $(\hat{C}, \hat{\rho})$ and determine whether or not this guess is an acceptable CFA for the program at hand

$$(\hat{C}, \hat{\rho}) \models e$$

Note

Intuitively, the clauses of our syntax directed specification are a sort of an explicit formulation of Dataflow equations: our constraints have the form $lhs \subseteq rhs$ and we want to compute the least solution

CFA Clauses (1)

$$(\hat{C}, \hat{\rho}) \models c^{\ell} \iff \mathit{true}$$

$$(\hat{C}, \hat{\rho}) \models x^{\ell} \iff \hat{\rho}(x) \subseteq \hat{C}(\ell)$$

$$(\hat{C}, \hat{\rho}) \models (\,\mathbf{fun}\, x \to e)^{\ell} \iff \{\,\mathbf{fun}\, x \to e\} \subseteq \hat{C}(\ell) \wedge (\hat{C}, \hat{\rho}) \models e$$

$$(\hat{C}, \hat{\rho}) \models (e_1 \diamond e_2)^{\ell} \iff (\hat{C}, \hat{\rho}) \models e_1 \wedge (\hat{C}, \hat{\rho}) \models e_2$$

$$\begin{aligned}(\hat{C}, \hat{\rho}) \models (\,\mathbf{let}\, x = t_1^{\ell_1}\, \mathbf{in}\, t_2^{\ell_2})^{\ell} \iff & (\hat{C}, \hat{\rho}) \models t_1^{\ell_1} \wedge (\hat{C}, \hat{\rho}) \models t_2^{\ell_2} \wedge \\ & \hat{C}(\ell_1) \subseteq \hat{\rho}(x) \wedge \hat{C}(\ell_2) \subseteq \hat{C}(\ell)\end{aligned}$$

CFA Clauses (2)

$$
\begin{aligned}
(\hat{C},\hat{\rho}) \models (\,\textbf{if } t_0^{\ell_0} \textbf{ then } t_1^{\ell_1} \textbf{ else } t_2^{\ell_2})^{\ell} \iff\ & (\hat{C},\hat{\rho}) \models t_0^{\ell_0} \wedge (\hat{C},\hat{\rho}) \models t_1^{\ell_1} \wedge (\hat{C},\hat{\rho}) \models t_2^{\ell_2} \wedge \\
& \hat{C}(\ell_1) \subseteq \hat{C}(\ell) \wedge \hat{C}(\ell_2) \subseteq \hat{C}(\ell) \\
(\hat{C},\hat{\rho}) \models (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell} \iff (\hat{C},\hat{\rho}) \models t_1^{\ell_1} \wedge\ & (\hat{C},\hat{\rho}) \models t_2^{\ell_2} \wedge \\
& (\forall (\,\textbf{fun}\, x \to t_0^{\ell_0}) \in \hat{C}(\ell_1) : \hat{C}(\ell_2) \subseteq \hat{\rho}(x) \wedge \\
& \hat{C}(\ell_0) \subseteq \hat{C}(\ell))
\end{aligned}
$$

Example: the requirements of valid estimates (1)

Consider the expression $((\,\textbf{fun}\, x \to x^1)^2(\,\textbf{fun}\, y \to y^3)^4)^5$, the CFA clauses are

$$(\hat{C}, \hat{\rho}) \vDash ((\,\textbf{fun}\, x \to x^1)^2(\,\textbf{fun}\, y \to y^3)^4)^5 \quad \text{iff}$$

$$\begin{aligned}&(\hat{C}, \hat{\rho}) \vDash (\,\textbf{fun}\, x \to x^1)^2 \;\wedge\; (\hat{C}, \hat{\rho}) \vDash (\,\textbf{fun}\, y \to y^3)^4 \;\wedge \\ &\qquad \Big(\forall(\,\textbf{fun}\, z \to t_0^{\ell_0}) \in \hat{C}(2) : \hat{C}(4) \subseteq \rho(z) \wedge \hat{C}(\ell_0) \subseteq \hat{C}(5)\Big) \quad \text{iff}\end{aligned}$$

Example: the requirements of valid estimates (2)

$$\{\,\textbf{fun}\,x \to x\} \subseteq \hat{C}(2) \wedge (\hat{C}, \hat{\rho}) \models x^1 \wedge \{\,\textbf{fun}\,y \to y\} \subseteq \hat{C}(4) \wedge (\hat{C}, \hat{\rho}) \models y^3 \wedge$$
$$\Big(\forall (\,\textbf{fun}\,z \to t_0^{\ell_0}) \in \hat{C}(\ell_2) : \hat{C}(\ell_4) \subseteq \rho(z) \wedge \hat{C}(\ell_0) \subseteq \hat{C}(5)\Big) \quad \text{iff}$$

$$\{\,\textbf{fun}\,x \to x\} \subseteq \hat{C}(2) \wedge \hat{\rho}(x) \subseteq \hat{C}(1) \wedge \{\,\textbf{fun}\,y \to y\} \subseteq \hat{C}(4) \wedge \rho(y) \subseteq \hat{C}(3) \wedge$$
$$\Big(\forall (\,\textbf{fun}\,z \to t_0^{\ell_0}) \in \hat{C}(2) : \hat{C}(4) \subseteq \rho(z) \wedge \hat{C}(\ell_0) \subseteq \hat{C}(5)\Big)$$

Example: the requirements of valid estimates (3)

The requirements a valid estimate must satisfy

$$\{\,\mathbf{fun}\, x \to x\} \subseteq \hat{C}(2) \wedge \hat{\rho}(x) \subseteq \hat{C}(1) \wedge \{\,\mathbf{fun}\, y \to y\} \subseteq \hat{C}(4) \wedge \rho(y) \subseteq \hat{C}(3) \wedge$$
$$\left(\forall(\,\mathbf{fun}\, z \to t_0^{\ell_0}) \in \hat{C}(2) : \hat{C}(4) \subseteq \rho(z) \wedge \hat{C}(\ell_0) \subseteq \hat{C}(5)\right)$$

Our previous guess

$\hat{C}$	1	2	3	4	5
	$\mathbf{fun}\, y \to y$	$\mathbf{fun}\, x \to x$	$\emptyset$	$\mathbf{fun}\, y \to y$	$\mathbf{fun}\, y \to y$

$\hat{\rho}$	x	y
	$\mathbf{fun}\, y \to y$	$\emptyset$

Constraints

$$lhs ::= r(x) \mid C(\ell) \mid \{t\}$$

$$rhs ::= r(x) \mid C(\ell)$$

$$c \in Constr ::= lhs \subseteq rhs \mid \{t\} \subseteq rhs' \implies lhs \subseteq rhs$$

- $r(x)$ is a constraint variable denoting the entry of the abstract environment for x
- $C(\ell)$ is a constraint variable denoting the entry of the abstract label for ℓ
- t is a token denoting expression of the form $\textbf{fun}\, x \to e$
- $\{t\} \subseteq rhs' \implies lhs \subseteq rhs$ should be read as $(\{t\} \subseteq rhs' \implies lhs) \subseteq rhs$

Constraint generation: $\mathcal{C}[\![e]\!]$ (1)

The function $\mathcal{C}[\![\_]\!] : \mathit{Exp} \to \mathit{Constr}$ given an expression returns a set of constraints

$$\mathcal{C}[\![c^\ell]\!] = \emptyset \qquad\qquad \mathcal{C}[\![x^\ell]\!] = \{r(x) \subseteq C(\ell)\}$$

$$\mathcal{C}[\![(\,\mathbf{fun}\, x \to e)^\ell]\!] = \{\{\,\mathbf{fun}\, x \to e\} \subseteq C(\ell)\} \cup \mathcal{C}[\![e]\!]$$

$$\mathcal{C}[\![(\,\mathbf{if}\ t_0^{\ell_0}\ \mathbf{then}\ t_1^{\ell_1}\ \mathbf{else}\ t_2^{\ell_2})^\ell]\!] = \mathcal{C}[\![t_0^{\ell_0}]\!] \cup \mathcal{C}[\![t_1^{\ell_1}]\!] \cup \mathcal{C}[\![t_2^{\ell_2}]\!] \cup \{C(\ell_1) \subseteq C(\ell),\ C(\ell_2) \subseteq C(\ell)\}$$

$$\mathcal{C}[\![(\,\mathbf{let}\ x = t_1^{\ell_1}\ \mathbf{in}\ t_2^{\ell_2})^\ell]\!] = \mathcal{C}[\![t_1^{\ell_1}]\!] \cup \mathcal{C}[\![t_2^{\ell_2}]\!] \cup \{C(\ell_1) \subseteq r(x), C(\ell_2) \subseteq C(\ell)\}$$

Constraint generation: $\mathcal{C}[\![e]\!]$ (2)

The function $\mathcal{C}[\![\_]\!] : \mathit{Exp} \to \mathit{Constr}$ given an expression returns a set of constraints

$$
\begin{aligned}
\mathcal{C}[\![(e_1 \diamond e_2)^{\ell}]\!] &= \mathcal{C}[\![e_1]\!] \cup \mathcal{C}[\![e_2]\!] \\
\mathcal{C}[\![(t_1^{\ell_1}\ t_2^{\ell_2})^{\ell}]\!] &= \mathcal{C}[\![t_1^{\ell_1}]\!] \cup \mathcal{C}[\![t_2^{\ell_2}]\!] \cup \\
&\qquad \{\{t\} \subseteq C(\ell_1) \implies C(\ell_2) \subseteq r(x), \\
&\qquad\ \ \{t\} \subseteq C(\ell_1) \implies C(\ell_0) \subseteq C(\ell) \mid \\
&\qquad t = (\,\mathbf{fun}\, x \to t_0^{\ell_0}) \in \Lambda^*\}
\end{aligned}
$$

where Λ^* is the set of function abstractions occurring in the program to analyze

Example of constraints

Consider the expression

$$e_\star = ((\,\textbf{fun}\, x \to x^1)^2(\,\textbf{fun}\, y \to y^3)^4)^5$$

$$\mathcal{C}[\![e_\star]\!] = \{$$

$$\{\,\textbf{fun}\, x \to x^1\} \subseteq C(2),\ r(x) \subseteq C(1),\ \{\,\textbf{fun}\, y \to y^3\} \subseteq C(4), r(y) \subseteq C(3),$$

$$\{\,\textbf{fun}\, x \to x^1\} \subseteq C(2) \Rightarrow C(4) \subseteq r(x), \{\,\textbf{fun}\, x \to x^1\} \subseteq C(2) \Rightarrow C(1) \subseteq C(5),$$

$$\{\,\textbf{fun}\, y \to y^3\} \subseteq C(2) \Rightarrow C(4) \subseteq r(y), \{\,\textbf{fun}\, y \to y^3\} \subseteq C(2) \Rightarrow C(3) \subseteq C(5)$$

$$\}$$

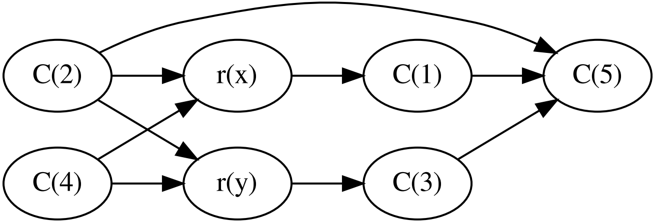
Solving the Constraints

Our constraint solver is based on a graph formulation of the constraints:

- There is a node for each variable $C(\ell)$ and $r(x)$ occurring in the constraints
- Edges are built from the constraints and decorated with them:
  - a constraint $p_1 \subseteq p_2$ results in an edge from p_1 to p_2
  - a constraint $\{t\} \subseteq p \implies p_1 \subseteq p_2$ results in an edge from p_1 to p_2 and from p to p_2
- For each node p we have a data field $D[p]$ containing the tokens associated to that node, initially is given by $D[p] = \{t \mid \{t\} \subseteq p\}$

The idea of the algorithm is to propagate the information from data field to another according to the constraints

Example of Graph representation



Data field:

$D[C(2)] = \{\, \textbf{fun}\, x \to x^1 \}$, $D[C(4)] = \{\, \textbf{fun}\, y \to y^3 \}$, $D[q] = \emptyset$ for other nodes

The Constraint Solver

Input: a set of constraint $\mathcal{C}[\![e_\star]\!]$

Output: the least solution to the constrains (an assignment to variables $C(\ell)$ and $r(x)$)

Data structures:

- a graph with one node for each variable $C(\ell)$ and $r(x)$ and zero, one or two edges for each constraints of $\mathcal{C}[\![e_\star]\!]$
- an worklist W containing the nodes whose data field should be propagated
- a data field D storing for each node the current abstract value
- an array E storing for each node a list of the constraint that the node may affect

The Solver Pseudocode (1)

```
(* Step 1: Initialization *)
W := [];
for q in Vars do
   D[q] := [];
   E[q] := [];
done
```

The Solver Pseudocode (2)

```
(* Step 2:  Build the graph *)
for cc in 𝒞⟦e⋆⟧ do
  match cc with
   | {t} ⊆ p -> add p {t}
   | p1 ⊆ p2 -> E[p1] := cc :: E[p1]
   | {t} ⊆ p  ⟹ p1 ⊆ p2 -> E[p1] := cc :: E[p1];
                             E[p] := cc :: E[p];
done
```

Example: Initialization of data structures

Node q	D[q]	E[q]
C(1)	$\emptyset$	$id_x \subseteq C(2) \Rightarrow C(1) \subseteq C(5)$
C(2)	id_x	$id_x \subseteq C(2) \Rightarrow C(4) \subseteq r(x), id_x \subseteq C(2) \Rightarrow C(1) \subseteq C(5)$
		$id_y \subseteq C(2) \Rightarrow C(4) \subseteq r(y), id_y \subseteq C(2) \Rightarrow C(3) \subseteq C(5)$
C(3)	$\emptyset$	$id_y \subseteq C(2) \Rightarrow C(3) \subseteq C(5)$
C(4)	id_y	$id_x \subseteq C(2) \Rightarrow C(4) \subseteq r(x), id_y \subseteq C(2) \Rightarrow C(4) \subseteq r(y)$
C(5)	$\emptyset$	
r(x)	$\emptyset$	$r(x) \subseteq C(1)$
r(y)	$\emptyset$	$r(y) \subseteq C(3)$

The Solver Pseudocode (3)

```
(* Step 3: Iteration *)

while W != [] do
  q = extract_front W
  for cc in E[q] do
    match cc with
    | p₁ ⊆ p₂ -> add p₂ D[p₁]
    | {t} ⊆ p ⟹ p₁ ⊆ p₂ -> if t in E[p] then add p₂ D[p₁]
  done
done
```

The Solver Pseudocode (4)

```
(* Step 4: Recording the solution *)
for ℓ do C(ℓ) = D[ℓ]; done

for x do r(x) = D[x]; done

(* auxiliary procedure *)
let add q d =
  if ¬ (d ⊆ D[q]) then
      D[q] = d ∪ D[q];
      push W q
```

Example of execution

W	[C(4), C(2)]	[r(x), C(2)]	[C(1), C(2)]	[C(5), C(2)]	[C(2)]	[]
q	D[q]	D[q]	D[q]	D[q]	D[q]	D[q]
C(1)	$\emptyset$	$\emptyset$	id_y	id_y	id_y	id_y
C(2)	id_x	id_x	id_x	id_x	id_x	id_x
C(3)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
C(4)	id_y	id_y	id_y	id_y	id_y	id_y
C(5)	$\emptyset$	$\emptyset$	$\emptyset$	id_y	id_y	id_y
r(x)	$\emptyset$	id_y	id_y	id_y	id_y	id_y
r(y)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Analysis improvements

- Our analysis is a pure CFA it only computes how the control flow may be transferred
  - Actually, the data affect the control: it is possible to extend the analysis we described to take into account also Dataflow information
  - We perform data and control flow analyses together
- Our CFA is *context insensitive* (0-CFA) because it does not distinguish between different calls to the same functions (poor precision, infeasible paths, ...)
  - It is possible to distinguish between different calls by introducing a notion of context (K-CFA), e.g., a call string of finite length
  - The complexity of the analysis increases

Conclusion

- Control-flow graph
- Control-flow analysis

Reference: Chapter 3 of Principles of Program Analysis by Nielson, Nielson, Hankin