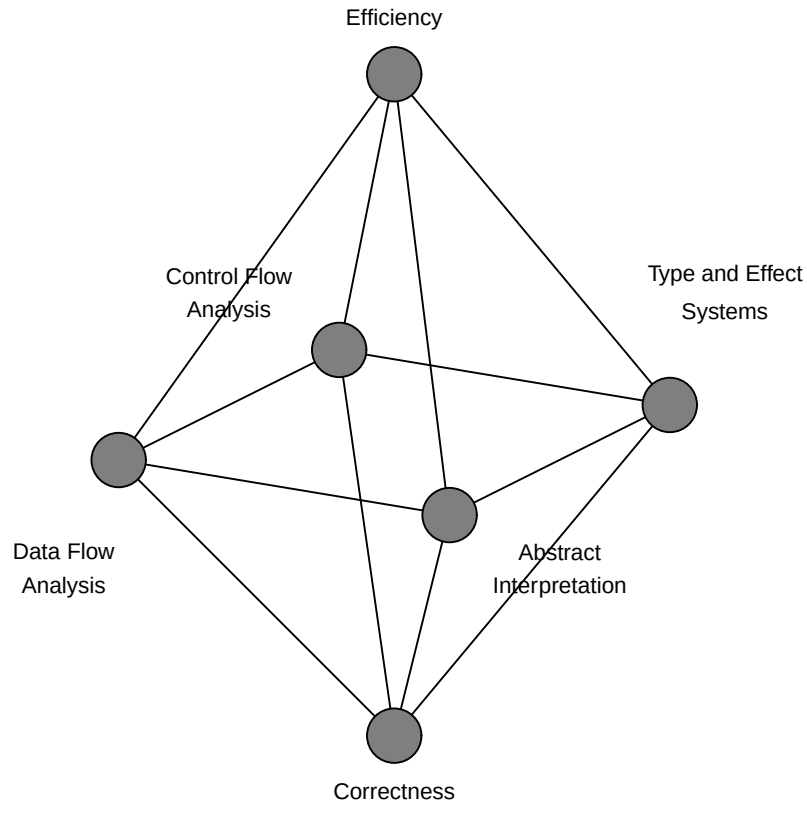
Algorithms


Efficiency
Control Flow
Analysis
Type and Effect
Systems
Data Flow
Analysis
Abstract
Interpretation
Correctness

We will abstract away from the details of a particular analysis by considering equations or inequations in a set of flow variables for which we want to solve the system. For example:

- in Data Flow Analysis there might be separate flow variables for the entry and exit values at each program point,

- whilst in Constraint Based Analysis there would be a separate flow variable for the cache at each program point and for the environment at each program variable.

Consider the following program and Reaching Definitions analysis:

$$\textbf{if } [b_1]^1 \textbf{ then } (\textbf{while } [b_2]^2 \textbf{ do } [\mathsf{x} := a_1]^3)$$
$$\textbf{else } (\textbf{while } [b_3]^4 \textbf{ do } [\mathsf{x} := a_2]^5);$$
$$[\mathsf{x} := a_3]^6$$

$$\mathsf{RD}_{entry}(1) = X_? \qquad \mathsf{RD}_{exit}(1) = \mathsf{RD}_{entry}(1)$$

$$\mathsf{RD}_{entry}(2) = \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(3) \qquad \mathsf{RD}_{exit}(2) = \mathsf{RD}_{entry}(2)$$

$$\mathsf{RD}_{entry}(3) = \mathsf{RD}_{exit}(2)$$

$$\mathsf{RD}_{exit}(3) = (\mathsf{RD}_{entry}(3) \backslash X_{356?}) \cup X_3$$

$$\mathsf{RD}_{entry}(4) = \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(5) \qquad \mathsf{RD}_{exit}(4) = \mathsf{RD}_{entry}(4)$$

$$\mathsf{RD}_{entry}(5) = \mathsf{RD}_{exit}(4)$$

$$\mathsf{RD}_{exit}(5) = (\mathsf{RD}_{entry}(5) \backslash X_{356?}) \cup X_5$$

$$\mathsf{RD}_{entry}(6) = \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(4)$$

$$\mathsf{RD}_{exit}(6) = (\mathsf{RD}_{entry}(6) \backslash X_{356?}) \cup X_6$$

Here $X_\ell = \{(\mathsf{x}, \ell)\}$ and we also allow a string of subscripts on x; for example, $X_{356?} = \{(\mathsf{x}, 3), (\mathsf{x}, 5), (\mathsf{x}, 6), (\mathsf{x}, ?)\}$.

When expressed as a constraint system in the flow variables $\{\mathsf{x}_1, \cdots, \mathsf{x}_{12}\}$ it takes the form

$$
\begin{array}{lcllcl}
\mathsf{x}_1 & = & X_? & \mathsf{x}_7 & = & \mathsf{x}_1 \\
\mathsf{x}_2 & = & \mathsf{x}_7 \cup \mathsf{x}_9 & \mathsf{x}_8 & = & \mathsf{x}_2 \\
\mathsf{x}_3 & = & \mathsf{x}_8 & \mathsf{x}_9 & = & (\mathsf{x}_3 \backslash X_{356?}) \cup X_3 \\
\mathsf{x}_4 & = & \mathsf{x}_7 \cup \mathsf{x}_{11} & \mathsf{x}_{10} & = & \mathsf{x}_4 \\
\mathsf{x}_5 & = & \mathsf{x}_{10} & \mathsf{x}_{11} & = & (\mathsf{x}_5 \backslash X_{356?}) \cup X_5 \\
\mathsf{x}_6 & = & \mathsf{x}_8 \cup \mathsf{x}_{10} & \mathsf{x}_{12} & = & (\mathsf{x}_6 \backslash X_{356?}) \cup X_6
\end{array}
$$

where $\mathsf{x}_1, \cdots, \mathsf{x}_6$ correspond to $\mathsf{RD}_{entry}(1), \cdots, \mathsf{RD}_{entry}(6)$ and $\mathsf{x}_7, \cdots, \mathsf{x}_{12}$ correspond to $\mathsf{RD}_{exit}(1), \cdots, \mathsf{RD}_{exit}(6)$.

Since we are generally interested in the solution for RD_{entry}, we shall in this and subsequent examples consider the following simplified equation system:

$$
\begin{aligned}
\mathrm{x}_1 &= X_? \\
\mathrm{x}_2 &= \mathrm{x}_1 \cup (\mathrm{x}_3 \setminus X_{356?}) \cup X_3 \\
\mathrm{x}_3 &= \mathrm{x}_2 \\
\mathrm{x}_4 &= \mathrm{x}_1 \cup (\mathrm{x}_5 \setminus X_{356?}) \cup X_5 \\
\mathrm{x}_5 &= \mathrm{x}_4 \\
\mathrm{x}_6 &= \mathrm{x}_2 \cup \mathrm{x}_4
\end{aligned}
$$

Clearly nothing is lost by these changes in representation.

The following inequation system (where all left hand sides are the same)

$$x \sqsupseteq t_1 \quad \cdots \quad x \sqsupseteq t_n$$

and the equation

$$x = x \sqcup t_1 \sqcup \cdots \sqcup t_n$$

have the same solutions: any solution of the former is also a solution of the latter and vice versa. Furthermore, the least solution of the above systems is also the least solution of

$$x = t_1 \sqcup \cdots \sqcup t_n$$

We make the following assumptions:

- There is a finite constraint system $\mathcal{S}$ of the form

$$(x_i \sqsupseteq t_i)_{i=1}^{N}$$

  for $N \geq 1$ where the left hand sides are not necessarily distinct.

- The set $\mathit{FV}(t_i)$ of flow variables contained in a right hand side t_i is a subset of the finite set $X = \{x_i \mid 1 \leq i \leq N\}$.

- A solution is a total function, $\psi : X \to L$, assigning each flow variable a value in the complete lattice $(L, \sqsubseteq)$ satisfying the Ascending Chain Condition.

- The terms are interpreted with respect to solutions, $\psi : X \to L$, and we write $[\![t]\!]\psi \in L$ to represent the interpretation of t with respect to ψ.
- The interpretation $[\![t]\!]\psi$ of a term t is monotone in ψ and its value only depends on the values $\{\psi(x) \mid x \in \mathit{FV}(t)\}$ of the solution on the flow variables occurring in the term.

In the interest of generality, we leave the nature of the right hand sides, t_i, unspecified.

Consider the following expression:

$$((\textbf{fn}\ x \Rightarrow x^1)^2\ (\textbf{fn}\ y \Rightarrow y^3)^4)^5$$

Using the notation above, the constraints are:

$\mathsf{x}_1 \supseteq \mathsf{x}_6$

$\mathsf{x}_3 \supseteq \mathsf{x}_7$

$\mathsf{x}_5 \supseteq$ if $\{\textbf{fn}\ x \Rightarrow x^1\} \subseteq \mathsf{x}_2$ then x_1

$\mathsf{x}_6 \supseteq$ if $\{\textbf{fn}\ x \Rightarrow x^1\} \subseteq \mathsf{x}_2$ then x_4

$\mathsf{x}_2 \supseteq \{\textbf{fn}\ x \Rightarrow x^1\}$

$\mathsf{x}_4 \supseteq \{\textbf{fn}\ y \Rightarrow y^3\}$

$\mathsf{x}_5 \supseteq$ if $\{\textbf{fn}\ y \Rightarrow y^3\} \subseteq \mathsf{x}_2$ then x_3

$\mathsf{x}_7 \supseteq$ if $\{\textbf{fn}\ y \Rightarrow y^3\} \subseteq \mathsf{x}_2$ then x_4

Here x_1 to x_5 correspond to $\mathsf{C}(1)$ to $\mathsf{C}(5)$, x_6 corresponds to $\mathsf{r}(\mathsf{x})$ and x_7 corresponds to $\mathsf{r}(\mathsf{y})$.

Our starting point is an abstract variant of the two worklist algorithms. It is abstract because it is parameterised on the details of the worklist and on the associated operations and values:

- empty is the empty worklist;
- insert($(x \sqsupseteq t)$,W) returns a new worklist that is as W except that a new constraint $x \sqsupseteq t$ has been added;
- extract(W) returns a pair whose first component is a constraint $x \sqsupseteq t$ in the worklist and whose second component is the smaller worklist obtained by removing an occurrence of $x \sqsupseteq t$.

In its most abstract form the worklist could be viewed as a set of constraints with the following operations:

empty = $\emptyset$

function insert($(x \sqsupseteq t)$,W)
return W $\cup \{x \sqsupseteq t\}$

function extract(W)
return ($(x \sqsupseteq t)$,W$\backslash\{x \sqsupseteq t\}$) for some $x \sqsupseteq t$ in W

However, it may be more appropriate to regard the worklist as a multiset, as a list with additional structure, or as a combination of other structures.

INPUT: A system $\mathcal{S}$ of constraints: $x_1 \sqsupseteq t_1, \cdots, x_N \sqsupseteq t_N$

OUTPUT: The least solution: Analysis

METHOD: Step 1: Initialisation (of W, Analysis and infl)

W := empty;

for all $x \sqsupseteq t$ in $\mathcal{S}$ do

W := insert($(x \sqsupseteq t)$,W);

Analysis[x] := $\bot$; infl[x] := $\emptyset$;

for all $x \sqsupseteq t$ in $\mathcal{S}$ do

for all x' in *FV*(t) do

infl[x'] := infl[x'] $\cup \{x \sqsupseteq t\}$;

Step 2: Iteration (updating W and Analysis)

while W $\neq$ empty do

  $((x \sqsupseteq t)$,W) := extract(W);

  new := eval(t,Analysis);

  if Analysis[x] $\not\sqsupseteq$ new then

    Analysis[x] := Analysis[x] $\sqcup$ new;

    for all $x' \sqsupseteq t'$ in infl[x] do

      W := insert($(x' \sqsupseteq t')$,W);

USING: function eval(t,Analysis)

return $[\![t]\!]$(Analysis)

value empty

function insert($(x \sqsupseteq t)$,W)

return $\cdots$

function extract(W)

return $\cdots$

Consider the simplified equation system. After step 1 of the abstract worklist algorithm, the worklist w contains all equations and the influence sets are as follows (where we identify the equations with the flow variables on the left hand side):

	x_1	x_2	x_3	x_4	x_5	x_6
infl	$\{x_2, x_4\}$	$\{x_3, x_6\}$	$\{x_2\}$	$\{x_5, x_6\}$	$\{x_4\}$	$\emptyset$

Additionally Analysis is set to $\emptyset$ for all flow variables. We shall continue this example later.

Assume that the size of the right hand sides of constraints is at most $M \geq 1$ and that the evaluation of a right hand side takes $O(M)$ steps; further assume that each assignment takes $O(1)$ step. Each constraint is influenced by at most M flow variables and therefore the initialisation of the influence sets takes $O(N + N \cdot M)$ steps. Writing N_x for the number of constraints in infl[x] we note that $\sum_{x \in X} N_x \leq M \cdot N$.

Assuming that L is of finite height at most $h \geq 1$, the algorithm assigns to Analysis[x] at most h times, adding N_x constraints to the worklist, for each flow variable $x \in X$. Thus, the total number of constraints added to the worklist is bounded from above by:

$$N + (h \cdot \sum_{x \in X} N_x) \leq N + (h \cdot M \cdot N)$$

Since each element on the worklist causes a call to eval, the cost of the calls is $O(N \cdot M + h \cdot M^2 \cdot N)$. This gives an overall complexity of $O(h \cdot M^2 \cdot N)$.

The worklist as a LIFO:

empty = nil

function insert($(x \sqsupseteq t)$,W)
return cons($(x \sqsupseteq t)$,W)

function extract(W)
return (head(W), tail(W))

A constraint of the form $\mathsf{Analysis}[\ell'] \sqsupseteq f_\ell(\mathsf{Analysis}[\ell])$ is represented on the worklist W of the data flow algorithm by the pair (ℓ, ℓ'). The influence sets were indirectly represented through the flow, F; to be more precise, $\mathsf{infl}[\ell'] = \{(\ell', \ell'') \in F \mid \ell'' \in \mathbf{Lab}\}$.

- N is proportional to the number b of elementary blocks.
- It is usual to take $M = 1$.
- Thus the upper bound on the complexity of the abstract algorithm specialises to $O(h \cdot b)$.
- For an analysis such as Reaching Definitions Analysis where h is proportional to b this gives $O(b^2)$.

Constraints of the form $\{t\} \subseteq p$, $p_1 \subseteq p$ or $\{t\} \subseteq p_2 \Rightarrow p_1 \subseteq p$ are represented on the worklist W by any one of the flow variables p_1 or p_2 occurring on the left hand side. The influence sets are represented using the edge array, E; to be more precise, infl[p] $=$ E[p]

- The initialisation of the influence sets in step 1 of the abstract algorithm corresponds to step 2 of the CFA algorithm;
- The inclusion on the worklist of all the constraints in infl[p] is replaced by the for-loop in step 3 of the CFA algorithm.
- Note that Analysis[p] is written as D[p].

The complexity for this instance of the abstract algorithm is related to the earlier worklist algorithm in the following way:

- N for a system generated from Control Flow Analysis is $O(n^2)$ where n is the size of the expression.
- h is bounded by n.
- Once again, it is usual that $M = 1$.
- Thus the upper bound on the complexity of the abstract algorithm specialises to $O(n^3)$.

One disadvantage of the LIFO strategy as presented above, is that we do not check for the presence of a constraint when adding it to the worklist. Hence the worklist may evolve so as to contain multiple copies of the same constraint and this may lead to unnecessarily recalculating terms before their free variables have had much chance of getting new values. This is illustrated in the following example; obviously a remedy is to modify the LIFO strategy such that it never inserts a constraint when it is already present.

$[x_1, x_2, x_3, x_4, x_5, x_6]$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[x_2, x_4, x_2, x_3, x_4, x_5, x_6]$	$X_?$	$-$	$-$	$-$	$-$	$-$
$[x_3, x_6, x_4, x_2, x_3, x_4, x_5, x_6]$	$-$	$X_{3?}$	$-$	$-$	$-$	$-$
$[x_2, x_6, x_4, x_2, x_3, x_4, x_5, x_6]$	$-$	$-$	$X_{3?}$	$-$	$-$	$-$
$[x_6, x_4, x_2, x_3, x_4, x_5, x_6]$	$-$	$-$	$-$	$-$	$-$	$-$
$[x_4, x_2, x_3, x_4, x_5, x_6]$	$-$	$-$	$-$	$-$	$-$	$X_{3?}$
$[x_5, x_6, x_2, x_3, x_4, x_5, x_6]$	$-$	$-$	$-$	$X_{5?}$	$-$	$-$
$[x_4, x_6, x_2, x_3, x_4, x_5, x_6]$	$-$	$-$	$-$	$-$	$X_{5?}$	$-$
$[x_6, x_2, x_3, x_4, x_5, x_6]$	$-$	$-$	$-$	$-$	$-$	$-$
$[x_2, x_3, x_4, x_5, x_6]$	$-$	$-$	$-$	$-$	$-$	$X_{35?}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$[\,]$	$-$	$-$	$-$	$-$	$-$	$-$

An obvious alternative to the use of a LIFO strategy is to use a FIFO strategy where the list is used as a queue. Again it may be worthwhile not to insert a constraint into a worklist when it is already present. However, rather than going deeper into the LIFO and FIFO strategies, we shall embark on a treatment of more advanced insertion and extraction strategies.

A careful organisation of the worklist may lead to algorithms that perform better in practice; however, in general we will not be able to improve our estimation of the worst case complexity to reflect this.

We explore the idea that changes should be propagated throughout the rest of the program before returning to re-evaluate a constraint.

One way of ensuring that every other constraint is evaluated before re-evaluating the constraint which caused the change is to impose some total order on the constraints.

To obtain a suitable ordering we shall impose a graph structure on the constraints and then use an iteration order based on reverse postorder.

Given a constraint system $\mathcal{S} = (x_i \sqsupseteq t_i)_{i=1}^{N}$ we can construct a graphical representation $G_{\mathcal{S}}$ of the dependencies between the constraints in the following way:

- there is a node for each constraint $x_i \sqsupseteq t_i$, and
- there is a directed edge from the node for $x_i \sqsupseteq t_i$ to the node for $x_j \sqsupseteq t_j$ if x_i appears in t_j (i.e. if $x_j \sqsupseteq t_j$ appears in infl[x_i]).

This constructs a directed graph. Sometimes it has a root, i.e. a node from which every other node is reachable through a directed path.

- This will generally be the case for constraint systems corresponding to forward analyses of While programs.
- It will not in general be the case for constraint systems corresponding to backward analyses for While programs nor for constraint systems constructed for Constraint Based Analysis.

We therefore need a generalisation of the concept of root.

One obvious remedy is to add a dummy root and enough dummy edges.

A more elegant formulation, that avoids cluttering the graph with dummy nodes and edges, is to consider a handle, i.e. a set of nodes such that each node in the graph is reachable through a directed path starting from one of the nodes in the handle. Indeed, a graph G has a root r if and only if G has $\{r\}$ as a handle.

One can take the entire set of nodes of a graph as a handle but it is more useful to choose a minimal handle: a handle such that no proper subset is also a handle; minimal handles always exist (although they need not be unique).

We can then construct a depth-first spanning forest from the graph $G_{\mathcal{S}}$ and handle $H_{\mathcal{S}}$ using the algorithm on the next slide. This also produces an array, rPostorder, that associates each node (i.e. each constraint $x \sqsupseteq t$) with its number in a reverse postorder traversal of the spanning forest.

We sometimes demand that a constraint system $(x_i \sqsupseteq t_i)_{i=1}^{N}$ is listed in reverse postorder.

INPUT: A directed graph (N, A) with k nodes and handle H

OUTPUT: (1) A DFSF T $= (N, A_T)$, and
(2) a numbering rPostorder of the nodes indicating the reverse order in which each node was last visited and represented as an element of array $[N]$ of int

METHOD:

i := k;

mark all nodes of N as unvisited;

let A_T be empty;

while unvisited nodes in H exists do

choose a node h in H;

DFS(h);

USING: procedure DFS(n) is

mark n as visited;

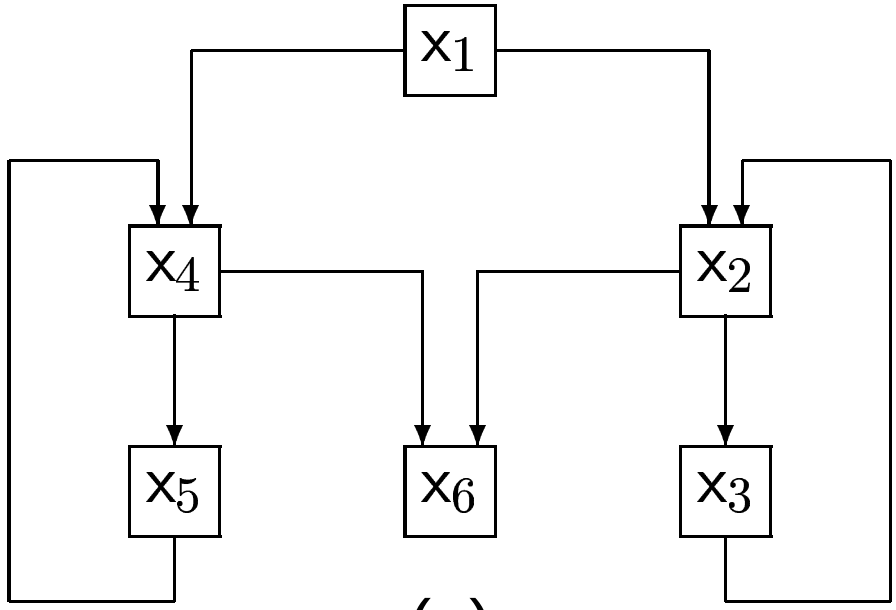
while $(n, n') \in A$ and n' has not been visited do

add the edge (n, n') to A_T;

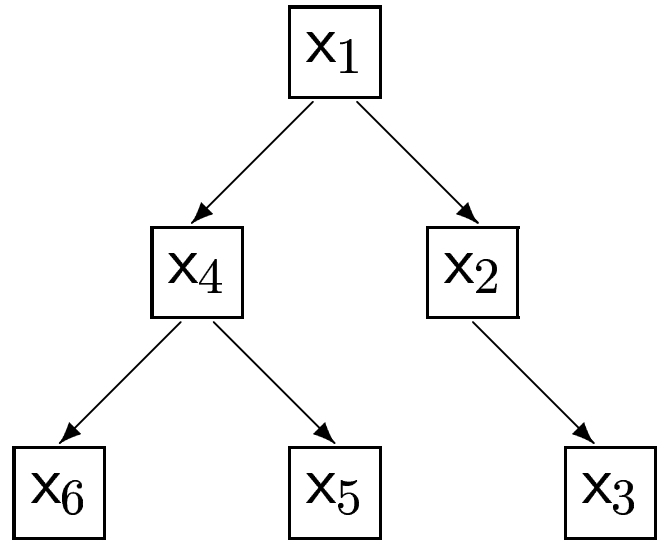
DFS(n');

rPostorder[n] := i;

i := i $-$ 1;



(a)



(b)

Conceptually, we now modify step 2 of the worklist algorithm so that the iteration amounts to an outer iteration that contains an inner iteration that visits the nodes in reverse postorder.

To achieve this, without actually changing the abstract algorithm, we shall organise the worklist W as a pair (W.c,W.p) of two structures:

- The first component, W.c, is a list of *current* nodes to be visited in the current inner iteration.
- The second component, W.p, is a set of *pending* nodes to be visited in a later inner iteration.
- Nodes are always inserted into W.p and always extracted from W.c;
- when W.c is exhausted the current inner iteration has finished and in preparation for the next we must sort W.p in the reverse postorder given by rPostorder and assign the result to W.c.

empty = (nil,$\emptyset$)

function insert($(x \sqsupseteq t)$,(W.c,W.p))

return (W.c,$(\mathsf{W.p} \cup \{x \sqsupseteq t\})$)

function extract((W.c,W.p))

if W.c = nil then

    W.c := sort_rPostorder(W.p);

    W.p := $\emptyset$

return (head(W.c), (tail(W.c),W.p))

$[\,]$	$\{x_1, \cdots, x_6\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[x_2, x_3, x_4, x_5, x_6]$	$\{x_2, x_4\}$	$X_?$	$-$	$-$	$-$	$-$	$-$
$[x_3, x_4, x_5, x_6]$	$\{x_2, x_3, x_4, x_6\}$	$-$	$X_{3?}$	$-$	$-$	$-$	$-$
$[x_4, x_5, x_6]$	$\{x_2, x_3, x_4, x_6\}$	$-$	$-$	$X_{3?}$	$-$	$-$	$-$
$[x_5, x_6]$	$\{x_2, \cdots, x_6\}$	$-$	$-$	$-$	$X_{5?}$	$-$	$-$
$[x_6]$	$\{x_2, \cdots, x_6\}$	$-$	$-$	$-$	$-$	$X_{5?}$	$-$
$[x_2, x_3, x_4, x_5, x_6]$	$\emptyset$	$-$	$-$	$-$	$-$	$-$	$X_{35?}$
$[x_3, x_4, x_5, x_6]$	$\emptyset$	$-$	$-$	$-$	$-$	$-$	$-$
$[x_4, x_5, x_6]$	$\emptyset$	$-$	$-$	$-$	$-$	$-$	$-$
$[x_5, x_6]$	$\emptyset$	$-$	$-$	$-$	$-$	$-$	$-$
$[x_6]$	$\emptyset$	$-$	$-$	$-$	$-$	$-$	$-$
$[\,]$	$\emptyset$	$-$	$-$	$-$	$-$	$-$	$-$

Now suppose that we change the above algorithm such that each time W.c is exhausted we assign it the list $[1, \cdots, N]$ rather than the potentially shorter list obtained by sorting W.p. This may lead to more evaluations of right hand sides of constraints but it simplifies some of the book-keeping details. Now our only interest in W.p is whether or not it is empty; let us introduce a boolean, change, that is false whenever W.p is empty. Also let us split the iterations into an overall outer iteration having an explicit inner iteration; each inner iteration will then be a simple iteration through all constraints in reverse postorder.

Thus we arrive at the Round Robin Algorithm:

INPUT: A system $\mathcal{S}$ of constraints: $x_1 \sqsupseteq t_1, \cdots, x_N \sqsupseteq t_N$ ordered 1 to N in reverse postorder

OUTPUT: The least solution: Analysis

METHOD: Step 1: Initialisation

for all $x \in X$ do

Analysis[x] := $\bot$

change := true;

Step 2: Iteration (updating Analysis)

while change do

change := false;

for i := 1 to *N* do

new := eval(t_i,Analysis);

if Analysis[x_i] $\not\sqsupseteq$ new then

change := true;

Analysis[x_i] := Analysis[x_i] $\sqcup$ new;

USING: function eval(t,Analysis)

return $[\![t]\!]$(Analysis)

true	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$*$ false						
true	$X_?$	–	–	–	–	–
true	–	$X_{3?}$	–	–	–	–
true	–	–	$X_{3?}$	–	–	–
true	–	–	–	$X_{5?}$	–	–
true	–	–	–	–	$X_{5?}$	–
true	–	–	–	–	–	$X_{35?}$
$*$ false						
false	–	–	–	–	–	–
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
false	–	–	–	–	–	–

We shall say that the constraint system $(x_i \sqsupseteq t_i)_{i=1}^{N}$ is an instance of a Bit Vector Framework when:

- $L = \mathcal{P}(D)$ for some finite set D and
- when each right hand side t_i is of the form $(x_{j_i} \cap Y_i^1) \cup Y_i^2$ for sets $Y_i^k \subseteq D$ and variable $x_{j_i} \in X$.

Clearly the classical Data Flow Analyses of Section produce constraint systems of this form (possibly after expansion of a composite constraint $x_i \sqsupseteq t_i^1 \sqcup \cdots \sqcup t_i^{k_i}$ into the individual constraints $x_i \sqsupseteq t_i^1, \cdots, x_i \sqsupseteq t_i^{k_i}$).

Consider a depth-first spanning forest T and a reverse postorder rPostorder constructed for the graph $G_{\mathcal{S}}$ with handle $H_{\mathcal{S}}$.

The loop connectedness parameter $d(G_{\mathcal{S}}, T) \geq 0$ is defined as the largest number of back edges found on any cycle-free path of $G_{\mathcal{S}}$.

The back edges are exactly those edges for which the target of the edge does not have an rPostorder number that is strictly larger than that of the source.

Let us say that the RR algorithm has iterated $n \geq 1$ times if the loop of step 1 has been executed once and the while-loop of step 2 has been executed $n - 1$ times. We then have the following result:

Under the assumptions stated above, the RR algorithm halts after at most $d(G_{\mathcal{S}}, T) + 3$ iterations. It therefore performs at most $O((d(G_{\mathcal{S}}, T) + 1) \cdot N)$ assignments.

For While programs the loop connectedness parameter is independent of the choice of depth first spanning forest, and hence of the choice of the reverse postorder recorded in rPostorder, and that it equals the maximal nesting depth d of while-loops.

It follows that this result gives an overall complexity of $O((d+1) \cdot b)$ where b is the number of elementary blocks.

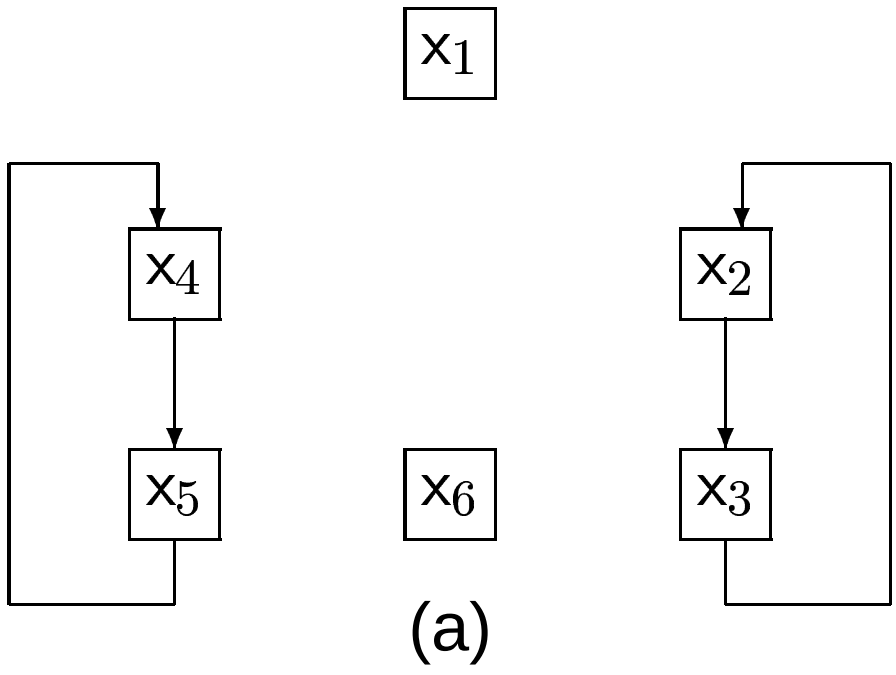
We would normally expect this bound to be significantly smaller than the $O(b^2)$ obtained earlier.

- A graph is strongly connected if every node is reachable from every other node.
- The strong components of a graph are its maximal strongly connected subgraphs.
- The strong components partition the nodes in the graph.
- The interconnections between components can be represented by a reduced graph: each strong component is represented by a node in the reduced graph and there is an edge from one strong component to another if there is an edge in the original graph from some node in the first strong component to a node in the second strong component and provided that the two strong components are not the same.

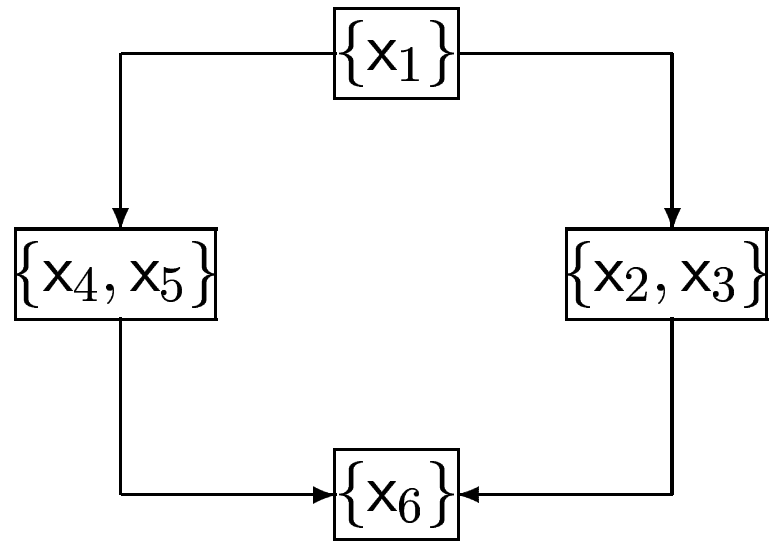
The reduced graph is always a DAG, i.e. a directed, acyclic graph.

As a consequence, the strong components can be linearly ordered in topological order: $SC_1 \leq SC_2$ (where SC_1 and SC_2 are nodes in the reduced graph) whenever there is an edge from SC_1 to SC_2.

Such a topological order can be obtained by constructing a reverse postorder for the reduced graph.



(a)



(b)

There are two possible topological orderings of the strong components. One is $\{x_1\}, \{x_2, x_3\}, \{x_4, x_5\}, \{x_6\}$ and the other is $\{x_1\}, \{x_4, x_5\}, \{x_2, x_3\}, \{x_6\}$.

In this example each strong component was an outermost loop. This holds in general for both forward and backward flow graphs for programs in the While language.

For each constraint we need to record both the strong component it occurs in and its number in the local reverse postorder for that strong component.

We shall do so by means of a numbering srPostorder that to each constraint $x \sqsupseteq t$ assigns a pair (*scc*,*rp*) consisting of the number *scc* of the strong component and the number *rp* of its reverse postorder numbering inside that strong component.

INPUT: A graph partitioned into strong components

OUTPUT: srPostorder

METHOD:

```
scc := 1;
for each scc in topological order do
    rp := 1;
    for each x ⊒ t in the strong component scc
        in local reverse postorder do
            srPostorder[x ⊒ t] := (scc,rp);
            rp := rp + 1
    scc := scc + 1;
```

Conceptually, we now modify step 2 of the worklist algorithm so that the iteration amounts to three levels of iteration; the outermost level deals with the strong components one by one; the intermediate level performs a number of passes over the constraints in the current strong component; and the inner level performs one pass in reverse postorder over the appropriate constraints.

empty = (nil,$\emptyset$)

function insert($(x \sqsupseteq t)$,(W.c,W.p))

return (W.c,$(\mathsf{W.p} \cup \{x \sqsupseteq t\})$)

function extract((W.c,W.p))

local variables: scc, W_scc

if W.c = nil then

scc := min$\{$fst(srPostorder[$x \sqsupseteq t$]) $\mid (x \sqsupseteq t) \in$ W.p$\}$;

W_scc := $\{(x \sqsupseteq t) \in \mathsf{W.p} \mid$ fst(srPostorder[$x \sqsupseteq t$]) $=$ scc$\}$;

W.c := sort_srPostorder(W_scc);

W.p := W.p $\setminus$ W_scc;

return (head(W.c), (tail(W.c),W.p))

Consider the ordering $\{x_1\}, \{x_2, x_3\}, \{x_4, x_5\}, \{x_6\}$ of the strong components of the example.

The algorithm iterating through strong components produces the walkthrough of the following slide when solving the system.

Note that even on this small example the algorithm performs slightly better than the others (ignoring the cost of maintaining the worklist).

$[\,]$	$\{x_1, \cdots, x_6\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[\,]$	$\{x_2, \cdots, x_6\}$	$X_?$	$-$	$-$	$-$	$-$	$-$
$[x_3]$	$\{x_3, \cdots, x_6\}$	$-$	$X_{3?}$	$-$	$-$	$-$	$-$
$[\,]$	$\{x_2, \cdots, x_6\}$	$-$	$-$	$X_{3?}$	$-$	$-$	$-$
$[x_3]$	$\{x_4, x_5, x_6\}$	$-$	$-$	$-$	$-$	$-$	$-$
$[\,]$	$\{x_4, x_5, x_6\}$	$-$	$-$	$-$	$-$	$-$	$-$
$[x_5]$	$\{x_5, x_6\}$	$-$	$-$	$-$	$X_{5?}$	$-$	$-$
$[\,]$	$\{x_4, x_5, x_6\}$	$-$	$-$	$-$	$-$	$X_{5?}$	$-$
$[x_5]$	$\{x_6\}$	$-$	$-$	$-$	$-$	$-$	$-$
$[\,]$	$\{x_6\}$	$-$	$-$	$-$	$-$	$-$	$-$
$[\,]$	$\emptyset$	$-$	$-$	$-$	$-$	$-$	$X_{35?}$