Principles of Program Analysis:

Algorithms

Transparencies based on Chapter 6 of the book: Flemming Nielson, Hanne Riis Nielson and Chris Hankin: Principles of Program Analysis. Springer Verlag 2005.

Worklist Algorithms

We abstract away from the details of a particular analysis:

We want to compute the solution to a set of equations

$$\{x_1 = t_1, \quad \cdots, \quad x_N = t_N\}$$

or inequations

$$\{x_1 \sqsupseteq t_1, \quad \cdots, \quad x_N \sqsupseteq t_N\}$$

defined in terms of a set of flow variables $x_1, \cdots, x_N$; here $t_1, \cdots, t_N$ are terms using the flow variables.

Equations or inequations?

It does not really matter:

- A solution of the equation system $\{x_1 = t_1, \cdots, x_N = t_N\}$ is also a solution of the inequation system $\{x_1 \sqsupseteq t_1, \cdots, x_N \sqsupseteq t_N\}$

- The least solution to the inequation systems $\{x_1 \sqsupseteq t_1, \cdots, x_N \sqsupseteq t_N\}$ is also a solution to the equation system $\{x_1 = t_1, \cdots, x_N = t_N\}$
  - The inequation system $\{x \sqsupseteq t_1, \cdots, x \sqsupseteq t_n\}$ (same left hand sides) and the equation $\{x = x \sqcup t_1 \sqcup \cdots \sqcup t_n\}$ have the same solutions.
  - The least solution to the equation $\{x = x \sqcup t_1 \sqcup \cdots \sqcup t_n\}$ is also the least solution of $\{x = t_1 \sqcup \cdots \sqcup t_n\}$ (where the x component has been removed on the right hand side).

Example While program

Reaching Definitions Analysis of

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

gives equations of the form

$$\mathsf{RD}_{entry}(1) = X_?$$
$$\mathsf{RD}_{entry}(2) = \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(3)$$
$$\mathsf{RD}_{entry}(3) = \mathsf{RD}_{exit}(2)$$
$$\mathsf{RD}_{entry}(4) = \mathsf{RD}_{exit}(1) \cup \mathsf{RD}_{exit}(5)$$
$$\mathsf{RD}_{entry}(5) = \mathsf{RD}_{exit}(4)$$
$$\mathsf{RD}_{entry}(6) = \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(4)$$

$$\mathsf{RD}_{exit}(1) = \mathsf{RD}_{entry}(1)$$
$$\mathsf{RD}_{exit}(2) = \mathsf{RD}_{entry}(2)$$
$$\mathsf{RD}_{exit}(3) = (\mathsf{RD}_{entry}(3) \backslash X_{356?}) \cup X_3$$
$$\mathsf{RD}_{exit}(4) = \mathsf{RD}_{entry}(4)$$
$$\mathsf{RD}_{exit}(5) = (\mathsf{RD}_{entry}(5) \backslash X_{356?}) \cup X_5$$
$$\mathsf{RD}_{exit}(6) = (\mathsf{RD}_{entry}(6) \backslash X_{356?}) \cup X_6$$

where e.g. $X_{356?}$ denotes the definitions of x at labels 3, 5, 6 and ?

Example (cont.)

Focussing on RD_{entry} and expressed as equations using the flow variables $\{\mathsf{x}_1, \cdots, \mathsf{x}_6\}$:

$$
\begin{array}{rclrcl}
\mathsf{x}_1 & = & X_? & \mathsf{x}_4 & = & \mathsf{x}_1 \cup (\mathsf{x}_5 \backslash X_{356?}) \cup X_5 \\
\mathsf{x}_2 & = & \mathsf{x}_1 \cup (\mathsf{x}_3 \backslash X_{356?}) \cup X_3 & \mathsf{x}_5 & = & \mathsf{x}_4 \\
\mathsf{x}_3 & = & \mathsf{x}_2 & \mathsf{x}_6 & = & \mathsf{x}_2 \cup \mathsf{x}_4
\end{array}
$$

Alternatively we can use inequations:

$$
\begin{array}{rclrclrclrcl}
\mathsf{x}_1 & \supseteq & X_? & \mathsf{x}_2 & \supseteq & X_3 & \mathsf{x}_4 & \supseteq & \mathsf{x}_1 & \mathsf{x}_5 & \supseteq & \mathsf{x}_4 \\
\mathsf{x}_2 & \supseteq & \mathsf{x}_1 & \mathsf{x}_3 & \supseteq & \mathsf{x}_2 & \mathsf{x}_4 & \supseteq & \mathsf{x}_5 \backslash X_{356?} & \mathsf{x}_6 & \supseteq & \mathsf{x}_2 \\
\mathsf{x}_2 & \supseteq & \mathsf{x}_3 \backslash X_{356?} & & & & \mathsf{x}_4 & \supseteq & X_5 & \mathsf{x}_6 & \supseteq & \mathsf{x}_4
\end{array}
$$

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 x_i are not necessarily distinct; the form of the terms t_i of the right hand sides is left unspecified.

- The set $FV(t_i)$ of flow variables occurring in 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 to 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 value of t relative to ψ.

- The interpretation $[\![t]\!]\psi$ of a term t is monotone in ψ and its value only depends on the values of the flow variables occurring in t.

Abstract Worklist Algorithm

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

OUTPUT: The least solution: Analysis

DATA STRUCTURES:

- W: worklist of constraints
- A: array indexed by flow variables containing elements of the lattice L (the current value of the flow variable)
- Infl: array indexed by flow variables containing the set of constraints influenced by the flow variable

Worklist Algorithm: initialisation

W := empty;
for all $x \sqsupseteq t$ in $\mathcal{S}$ do
  W := insert(($x \sqsupseteq t$),W); initially all constraints in the worklist
  Analysis[x] := $\bot$; initialised to the least element of L
  infl[x] := $\emptyset$;
for all $x \sqsupseteq t$ in $\mathcal{S}$ do
  for all x' in $FV(t)$ do
    infl[x'] := infl[x'] $\cup$ $\{x \sqsupseteq t\}$; changes to x' might influence x via the constraint $x \sqsupseteq t$

OBS: After the initialisation we have infl[x'] $= \{(x \sqsupseteq t)$ in $\mathcal{S} \mid x' \in FV(t)\}$

Worklist Algorithm: iteration

```
while W ≠ empty do
  ((x ⊒ t),W) := extract(W);              consider the next constraint
  new := eval(t,Analysis);
  if Analysis[x] ⋣ new then               any work to do?
    Analysis[x] := Analysis[x] ⊔ new;     update the analysis information
    for all x′ ⊒ t′ in infl[x] do
      W := insert((x′ ⊒ t′),W);           update the worklist
```

Operations on worklists

- 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; it is normally used as in

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

  so as to update the worklist W to contain the new constraint $x \sqsupseteq t$;

- 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 $x \sqsupseteq t$; it is used as in

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

  so as to select and remove a constraint from W.

Organising the worklist

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$\setminus\{x \sqsupseteq t\}$) for some $x \sqsupseteq t$ in W

Extraction based on LIFO

The worklist is represented as a list of constraints with the following operations:

empty = nil

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

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

Extraction based on FIFO

The worklist is represented as a list of constraints:

empty = nil

function insert(($x \sqsupseteq t$),W)
return append(W,[$x \sqsupseteq t$])

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

Example: initialisation

Equations:

$$\begin{array}{lcl} \mathsf{x}_1 & = & X_? \\ \mathsf{x}_2 & = & \mathsf{x}_1 \cup (\mathsf{x}_3 \backslash X_{356?}) \cup X_3 \\ \mathsf{x}_3 & = & \mathsf{x}_2 \end{array} \qquad \begin{array}{lcl} \mathsf{x}_4 & = & \mathsf{x}_1 \cup (\mathsf{x}_5 \backslash X_{356?}) \cup X_5 \\ \mathsf{x}_5 & = & \mathsf{x}_4 \\ \mathsf{x}_6 & = & \mathsf{x}_2 \cup \mathsf{x}_4 \end{array}$$

Initialised data structures:

	x_1	x_2	x_3	x_4	x_5	x_6
infl	$\{\mathsf{x}_2, \mathsf{x}_4\}$	$\{\mathsf{x}_3, \mathsf{x}_6\}$	$\{\mathsf{x}_2\}$	$\{\mathsf{x}_5, \mathsf{x}_6\}$	$\{\mathsf{x}_4\}$	$\emptyset$
A	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
W	$[\mathsf{x}_1, \mathsf{x}_2, \mathsf{x}_3, \mathsf{x}_4, \mathsf{x}_5, \mathsf{x}_6]$					

OBS: in this example the left hand sides of the equations uniquely identify the equations

Example: iteration

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

Correctness of the algorithm

Given a system of constraints, $\mathcal{S} = (x_i \sqsupseteq t_i)_{i=1}^{N}$, we define

$$F_{\mathcal{S}} : (X \to L) \to (X \to L)$$

by:

$$F_{\mathcal{S}}(\psi)(x) = \bigsqcup \{ [\![t]\!]\psi \mid x \sqsupseteq t \text{ in } \mathcal{S} \}$$

This is a monotone function over a complete lattice $X \to L$.

It follows from Tarski's Fixed Point Theorem:

> If $f : L \to L$ is a monotone function on a complete lattice $(L, \sqsubseteq)$ then it has a least fixed point $\mathit{lfp}(f) = \sqcap \mathit{Red}(f) \in \mathit{Fix}(f)$

that $F_{\mathcal{S}}$ has a least fixed point, $\mu_{\mathcal{S}}$, which is the least solution to the constraints $\mathcal{S}$.

Tarski's Fixed Point Theorem (again)

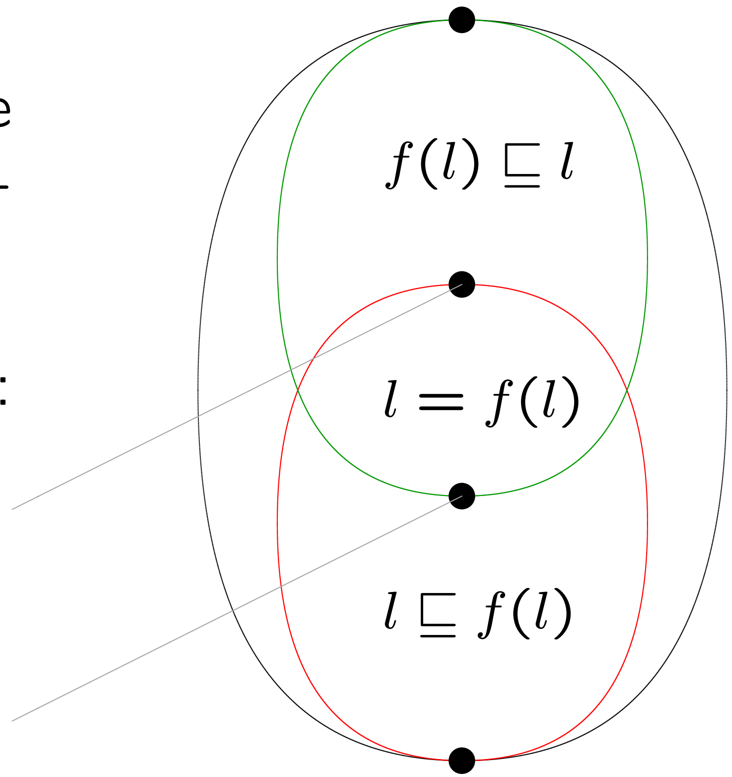
Let $L = (L, \sqsubseteq)$ be a complete lattice and let $f : L \to L$ be a monotone function.

The greatest fixed point *gfp*(f) satisfy:

$$gfp(f) = \bigsqcup\{l \mid l \sqsubseteq f(l)\} \in \{l \mid f(l) = l\}$$

The least fixed point *lfp*(f) satisfy:

$$lfp(f) = \bigsqcap\{l \mid f(l) \sqsubseteq l\} \in \{l \mid f(l) = l\}$$



Correctness of the algorithm (2)

Since L satisfies the Ascending Chain Condition and since X is finite it follows that also $X \to L$ satisfies the Ascending Chain Condition; therefore $\mu_{\mathcal{S}}$ is given by

$$\mu_{\mathcal{S}} = \mathit{lfp}(F_{\mathcal{S}}) = \bigsqcup_{j \geq 0} F_{\mathcal{S}}^{j}(\bot)$$

and the chain $(F_{\mathcal{S}}^{n}(\bot))_n$ eventually stabilises.

Lemma

Given the assumptions, the abstract worklist algorithm computes the least solution of the given constraint system, $\mathcal{S}$.

Proof

- termination – of initialisation and iteration loop
- correctness is established in three steps:
  - $\mathsf{A} \sqsubseteq \mu_{\mathcal{S}}$ – holds initially and is preserved by the loop
  - $F_{\mathcal{S}}(\mathsf{A}) \sqsubseteq \mathsf{A}$ – proved by contradiction
  - $\mu_{\mathcal{S}} \sqsubseteq \mathsf{A}$ – follows from Tarski's fixed point theorem
- complexity: $O(h \cdot M^2 \cdot N)$ for h being the height of L, M being the maximal size of the right hand sides of the constraints and N being the number of constraints

Worklist & Reverse Postorder

- Changes should be propagated throughout the rest of the program before returning to re-evaluate a constraint.

- To ensure that every other constraint is evaluated before re-evaluating the constraint which caused the change is to impose some total order on the constraints.

- We shall impose a graph structure on the constraints and then use an iteration order based on reverse postorder.

Graph structure of constraint system

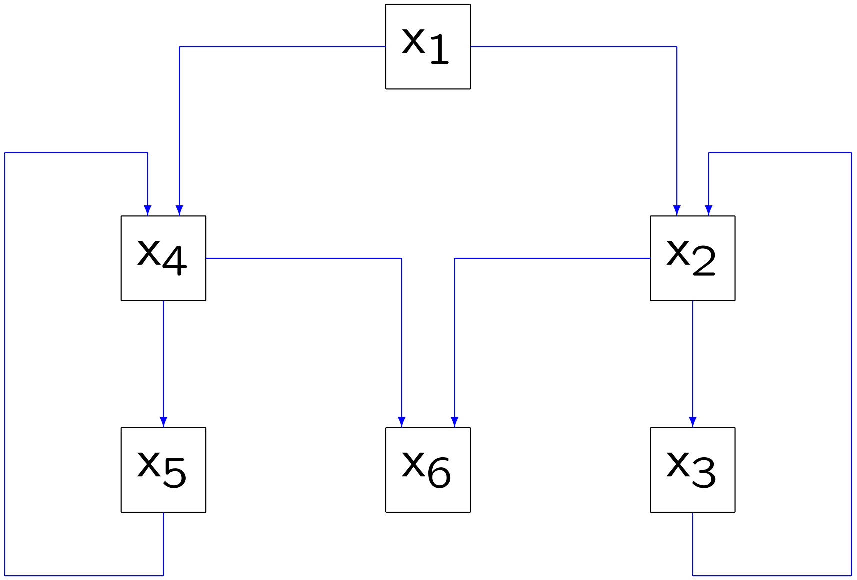
Given a constraint system $\mathcal{S} = (x_i \sqsupseteq t_i)_{i=1}^N$ we can construct a graphical representation $G_\mathcal{S}$ of the dependencies between the constraints in the following way:

- 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.

Example: graph representation

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



Handles and roots

Observations:

- A constraint systems corresponding to forward analyses of While programs will have a root

- A constraint systems corresponding to backward analyses for While programs will not have a single root

A handle is 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.

- A graph G has a root r if and only if G has $\{r\}$ as a handle

- Minimal handles always exist (but they need not be unique)

Depth-First Spanning Forest

We can then construct a depth-first spanning forest (abbreviated DFSF) from the graph $G_\mathcal{S}$ and handle $H_\mathcal{S}$:

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

OUTPUT: (1) A DFSF $\mathsf{T} = (N, A_T)$, 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

Algorithm for DFSF

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

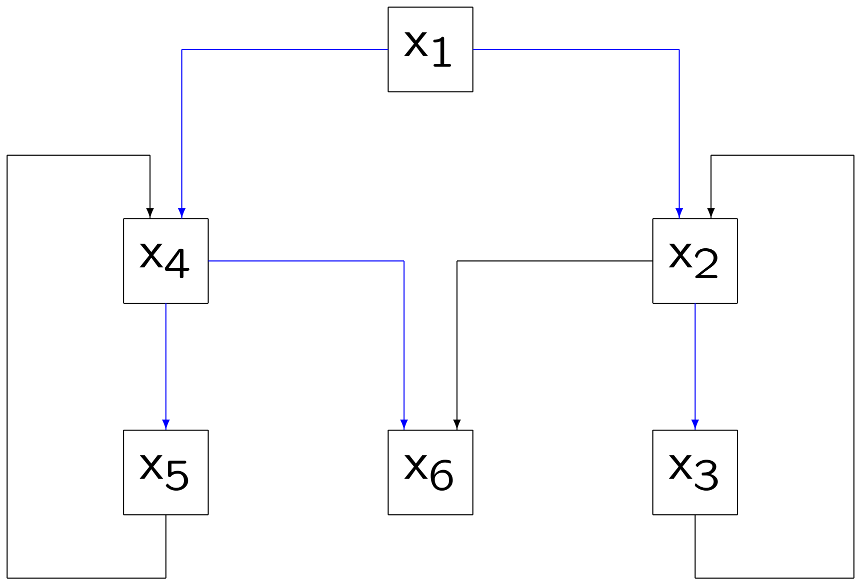
  mark n as visited;

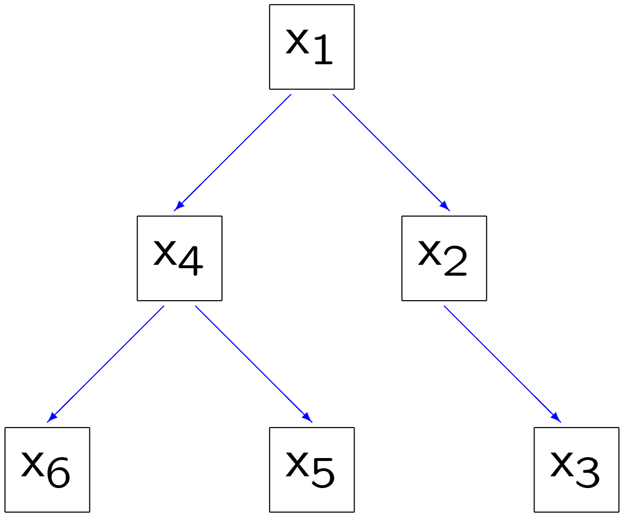
  while $(n, n') \in A$ and n' has not been visited do

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

  rPostorder[n] := i; i := i $-$ 1;

Example: DFST





reverse postorder: $x_1, x_2, x_3, x_4, x_5, x_6$
pre-order: $x_1, x_4, x_6, x_5, x_2, x_3$
breadth-first order: $x_1, x_4, x_2, x_6, x_5, x_3$

Categorisation of edges

Given a spanning forest one can categorise the edges in the original graph as follows:

- Tree edges: edges present in the spanning forest.
- Forward edges: edges that are not tree edges and that go from a node to a proper descendant in the tree.
- Back edges: edges that go from descendants to ancestors (including self-loops).
- Cross edges: edges that go between nodes that are unrelated by the ancestor and descendant relations.

Properties of Reverse Postorder

Let $G = (N, A)$ be a directed graph, T a depth-first spanning forest of G and rPostorder the associated ordering computed by the algorithm.

- $(n, n') \in A$ is a back edge if and only if $\text{rPostorder}[n] \geq \text{rPostorder}[n']$.
- $(n, n') \in A$ is a self-loop if and only if $\text{rPostorder}[n] = \text{rPostorder}[n']$.
- Any cycle of G contains at least one back edge.
- Reverse postorder (rPostorder) topologically sorts tree edges as well as the forward and cross edges.
- Preorder and breadth-first order also sorts tree edges and forward edges but not necessarily cross edges.

Extraction based on Reverse Postorder

Idea: The iteration amounts to an outer iteration that contains an inner iteration that visits the nodes in reverse postorder:

We organise the worklist W as a pair (W.c,W.p) of two structures:

- W.c is a list of current nodes to be visited in the current inner iteration.
- 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 inner iteration we must sort W.p in the reverse postorder given by rPostorder and assign the result to W.c.

Iterating in Reverse Postorder

empty = (nil,$\emptyset$)

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

return (W.c,(W.p $\cup$ $\{x \sqsupseteq t\}$)) — insert into pending set

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

if W.c = nil then — no more constraints in current list

    W.c := sort_rPostorder(W.p); — sort pending set and update

    W.p := $\emptyset$ — current list and pending set

return (head(W.c), (tail(W.c),W.p)) — extract from current round

Example: Reverse Postorder iteration

W.c	W.p	x_1	x_2	x_3	x_4	x_5	x_6
[ ]	$\{x_1, \cdots, x_6\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$[x_2, x_3, x_4, x_5, x_6]$	$\{x_2, x_4\}$	$X_?$	—	—	—	—	—
$[x_3, x_4, x_5, x_6]$	$\{x_2, x_3, x_4, x_6\}$	—	$X_{3?}$	—	—	—	—
$[x_4, x_5, x_6]$	$\{x_2, x_3, x_4, x_6\}$	—	—	$X_{3?}$	—	—	—
$[x_5, x_6]$	$\{x_2, \cdots, x_6\}$	—	—	—	$X_{5?}$	—	—
$[x_6]$	$\{x_2, \cdots, x_6\}$	—	—	—	—	$X_{5?}$	—
$[x_2, x_3, x_4, x_5, x_6]$	$\emptyset$	—	—	—	—	—	$X_{35?}$
$[x_3, x_4, x_5, x_6]$	$\emptyset$	—	—	—	—	—	—
$[x_4, x_5, x_6]$	$\emptyset$	—	—	—	—	—	—
$[x_5, x_6]$	$\emptyset$	—	—	—	—	—	—
$[x_6]$	$\emptyset$	—	—	—	—	—	—
[ ]	$\emptyset$	—	—	—	—	—	—

$$x_1 = X_? \qquad x_3 = x_2 \qquad x_5 = x_4$$
$$x_2 = x_1 \cup (x_3 \backslash X_{356?}) \cup X_3 \qquad x_4 = x_1 \cup (x_5 \backslash X_{356?}) \cup X_5 \qquad x_6 = x_2 \cup x_4$$

Complexity

- A list of N elements can be sorted in $O(N \cdot \log_2(N))$ steps.

- If we use a linked list representation of lists then inserting an element to the front of a list and extracting the head of a list can be done in constant time.

- The overall complexity for processing N insertions and N extractions is $O(N \cdot \log_2(N))$.

The Round Robin Algorithm

Assumption: the constraints are sorted in reverse postorder.

- each time W.c is exhausted we assign it the list $[1, \cdots, N]$
- W.p is replaced by a boolean, change, that is false whenever W.p is empty
- the iterations are split into an outer iteration with an explicit inner iteration; each inner iteration is a simple iteration through all constraints in reverse postorder.

Round Robin Iteration

```
empty = (nil,false)

function insert((x ⊒ t),(W.c,change))
return (W.c,true)                       pending constraints

function extract((W.c,change))
if W.c = nil then                       a new round is needed
  W.c := [1, ··· , N];                   all constraints are re-considered
  change := false                        no pending constraints
return (head(W.c),(tail(W.c),change))
```

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: Initialisation

for all $x \in X$ do Analysis$[x]$:= $\perp$

change := true;

The Round Robin Algorithm (cont.)

METHOD: Iteration (updating Analysis)

```
while change do
  change := false;
  for i := 1 to N do
    new := eval(t_i,Analysis);
    if Analysis[x_i] ⋣ new then
      change := true;
      Analysis[x_i] := Analysis[x_i] ⊔ new;
```

Lemma:

The Round Robin algorithm computes the least solution of the given constraint system, $\mathcal{S}$.

Example: Round Robin iteration

change	x_1	x_2	x_3	x_4	x_5	x_6
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	—	—	—	—	—	—
false	—	—	—	—	—	—
false	—	—	—	—	—	—
false	—	—	—	—	—	—
false	—	—	—	—	—	—
false	—	—	—	—	—	—

$$\begin{aligned}
x_1 &= X_? \\
x_2 &= x_1 \cup (x_3 \backslash X_{356?}) \cup X_3 \\
x_3 &= x_2 \\
x_4 &= x_1 \cup (x_5 \backslash X_{356?}) \cup X_5 \\
x_5 &= x_4 \\
x_6 &= x_2 \cup x_4
\end{aligned}$$

Loop connectness parameter

Consider a depth-first spanning forest T and a reverse postorder rPostorder constructed for the graph G with handle H.

The loop connectedness parameter $d(G,T)$ is defined as the largest number of back edges found on any cycle-free path of G.

For While programs the loop connectedness parameter equals the maximal nesting depth of `while` loops.

Empirical studies of Fortran programs show that the loop connectness parameter seldom exceeds 3.

Complexity

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 $x_{j_i} \in X$.

Lemma:

> For Bit Vector Frameworks, the Round Robin Algorithm terminates after at most $d(G,T)+3$ iterations.
>
> It performs at most $O((d(G,T)+1)\cdot N)$ assignments.

For While programs: the overall complexity is $O((d+1)\cdot b)$ where d is the maximal nesting depth of while-loops and b is the number of elementary blocks.

Worklist & Strong Components

Two nodes n and n' are said to be strongly connected whenever there is a (possibly trivial) directed path from n to n' and a (possibly trivial) directed path from n' to n. Defining

$$\mathcal{SC} = \{(n, n') \mid n \text{ and } n' \text{ are strongly connected}\}$$

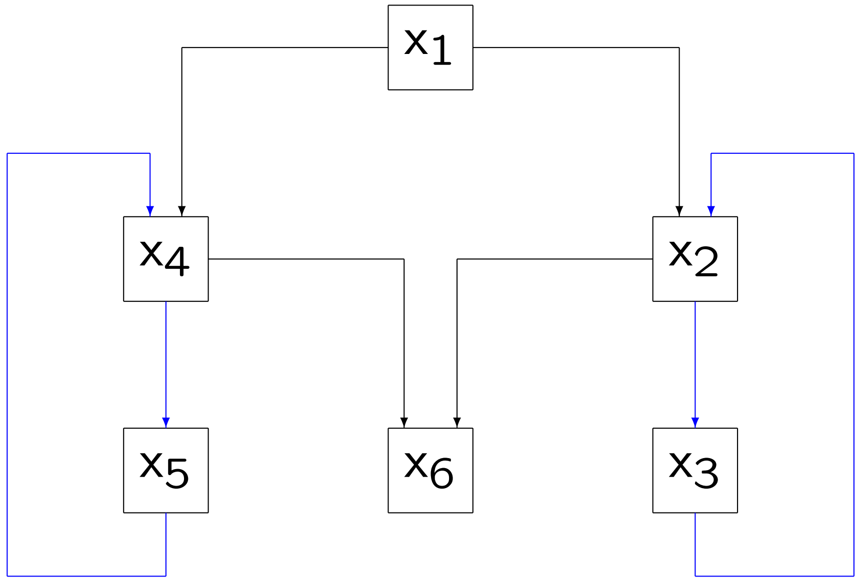
we obtain a binary relation $\mathcal{SC} \subseteq N \times N$.

- $\mathcal{SC}$ is an equivalence relation.

- The equivalence classes of $\mathcal{SC}$ are called the strong components.

A graph is said to be strongly connected whenever it contains exactly one strongly connected component.

Example: Strong Components



Reduced graph

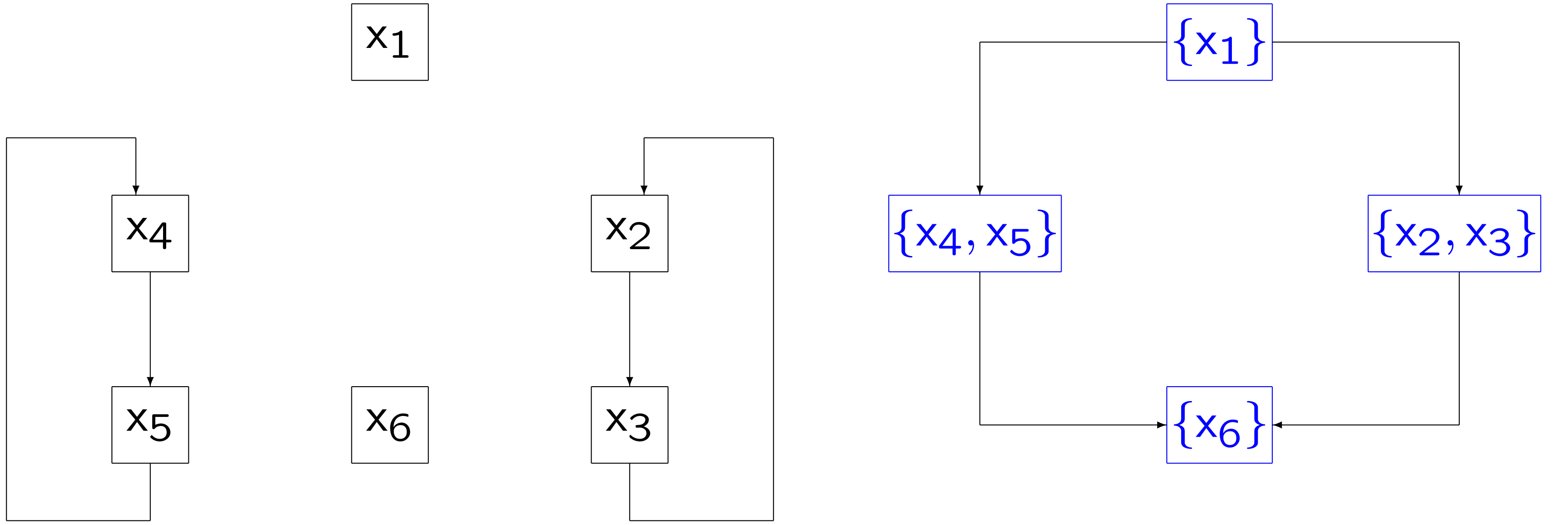
The interconnections between strong components can be represented by the reduced graph.

- nodes: the strongly connected components
- edges: there is an edge from one node to another distinct node if and only if there is an edge from some node in the first strongly connected component to a node in the second in the original graph.

For any graph G the reduced graph is a DAG.

The strong components can be linearly ordered in topological order: $SC_1 \leq SC_2$ whenever there is an edge from SC_1 to SC_2.

Example: Strong Components and reduced graph



The overall idea behind the algorithm

Idea: strong components are visited in topological order with nodes being visited in reverse postorder within each strong component.

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;
- the inner level performs one pass in reverse postorder over the appropriate constraints.

To make this work for each constraint we record

- the strong component it occurs in and
- its number in the local reverse postorder for that strong component.

Pseudocode for constraint numbering

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;
```

Organisation of the worklist

The worklist W as a pair (W.c,W.p) of two structures:

- W.c, is a list of current nodes to be visited in the current inner iteration.
- W.p, is a set of pending nodes to be visited in a later intermediate or outer 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 extract a strong component from W.p, sort it and assign the result to W.c.

An inner iteration ends when W.c is exhausted, an intermediate iteration ends when scc gets a higher value than last time it was computed, and the outer iteration ends when both W.c and W.p are exhausted.

Iterating through Strong Components

```
empty = (nil,∅)

function insert((x ⊒ t),(W.c,W.p))
return (W.c,(W.p ∪ {x ⊒ t}))

function extract((W.c,W.p))
local variables: scc, W_scc
if W.c = nil then
    scc := min{fst(srPostorder[x ⊒ t]) | (x ⊒ t) ∈ W.p};
    W_scc := {(x ⊒ t) ∈ W.p | fst(srPostorder[x ⊒ t]) = scc};
    W.c := sort_srPostorder(W_scc);
    W.p := W.p \ W_scc;
return ( head(W.c), (tail(W.c),W.p) )
```

Example: Strong Component iteration

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

$$x_1 = X_? \qquad x_3 = x_2 \qquad x_5 = x_4$$
$$x_2 = x_1 \cup (x_3 \backslash X_{356?}) \cup X_3 \qquad x_4 = x_1 \cup (x_5 \backslash X_{356?}) \cup X_5 \qquad x_6 = x_2 \cup x_4$$