Chapter 6: Algorithms

Reporter: Xie Li

December 1, 2021

Overview

Part I: Abstract Worklist Algorithm

Worklist Algorithm: Reaching Definition Analysis Example

Example 6.1 Consider the following WHILE program

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

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

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

where $\mathbf{X}_{356?}$ represents $\{(x,3),(x,5),(x,6),(x,?)\}$

Assumptions

Assumptions:

- A finite contraint system $(x_i \sqsupseteq t_i)_{i=1}^{N}$, where $N \geq 1$
- $\bigcup_i \mathsf{FV}(t_i) \subseteq X = \{x_i \ \mid 1 \leq i \leq N\}$
- A solution is a total function $\psi : X \to L$, where $(L, \sqsubseteq)$ is a complete lattice satisfying the Acending Chain Condition.
- Terms are interpreted by the solutions. $[\![t]\!]\psi \in L$.
- The interpretation $[\![t]\!]\psi$ of a term t is monotone in ψ and its value only depends on the values $\{\psi(x) \mid x \in \mathsf{FV}(t)\}$

Equations vs. inequations:

$$x \sqsupseteq t_1, \ldots x \sqsupseteq t_n$$

and

$$x = x \sqcup t_1 \sqcup \ldots \sqcup t_n$$

have the same solutions, and the least solution of the system is also the least solution of

$$x = t_1 \sqcup \ldots \sqcup t_n$$

Abstract Worklist Algorithms

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 ⊒ t in S do
    W := insert((x ⊒ t),W)
    Analysis[x] := ⊥;
    infl[x] := ∅;
for all x ⊒ t in S do
    for all x' in FV(t) do
        infl[x'] := infl[x'] ∪ {x ⊒ t};
```

Step 2: Iteration (updating W and Analysis)

```
while W ≠ empty do
    ((x ⊒ t),W) := extract(W);
    new := eval(t,Analysis);
    if Analysis[x] ⋣ new then
        Analysis[x] := Analysis[x] ⊔ new;
        for all x' ⊒ t' in infl[x] do
            W := insert((x' ⊒ t'),W);
```

Abstract Worklist Algorithm

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

function eval(t,Analysis)
return $[\![t]\!]$(Analysis)

and

$$\mathtt{infl}[x] = \{(x' \sqsupseteq t') \in \mathcal{S} \mid x \in \mathsf{FV}(t')\}$$

Example of influence:

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

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

Properties of the Algorithm

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

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

by

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

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

- Monotone: can be checked easily. $\phi \sqsubseteq \psi \Rightarrow F_{\mathcal{S}}(\phi) \sqsubseteq F_{\mathcal{S}}(\psi)$
- Ascending chain condition: X is finite and L by assumption satisfies ascending chain condition...

Correctness of the Algorithm

Lemma (6.4)

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

Proof.

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 ⊒ t in S do
    W := insert((x ⊒ t),W)
    Analysis[x] := ⊥;
    infl[x] := ∅;
for all x ⊒ t in S do
    for all x' in FV(t) do
        infl[x'] := infl[x'] ∪ {x ⊒ t};
```

Step 2: Iteration (updating W and Analysis)

```
while W ≠ empty do
   ((x ⊒ t),W) := extract(W);
   new := eval(t,Analysis);
   if Analysis[x] ⋣ new then
       Analysis[x] := Analysis[x] ⊔ new;
       for all x' ⊒ t' in infl[x] do
           W := insert((x' ⊒ t'),W);
```

Termination:
Termination of Step 1 is trivial.
Termination of Step 2 while loop can be proved with the ascending chain condition of L.
Correctness:

- $\forall x.\texttt{Analysis}_i[x] \sqsubseteq \mu_{\mathcal{S}}(x)$ is a invariant of the while loop of Step 2.
- $F_{\mathcal{S}}(\texttt{Analysis}) \sqsubseteq \texttt{Analysis}$

Complexity of the Algorithm

Assumptions:

- The size of RHS of constraints is at most $M \geq 1$ and the evaluation of RHS takes $O(M)$.
- Each assignment takes $O(1)$ step.
- Each constraint is influence by at most M flow variables
- The number of constraints in `infl`$[x]$ is N_x. Then we have $\sum_{x \in X} N_x \leq M \cdot N$
- The maximum height of ascending chain is h.

The total number of constraints added: $O(N + h \cdot N \cdot M)$.
Consider their evaluations: $O(N \cdot M + h \cdot M^2 \cdot N) = O(h \cdot M^2 \cdot N)$

Part II: Preliminaries

Directed Graph

- **Directed graph:** A directed graph $G = (N, A)$. Flow, cycle, SCC...
- **Handles and roots:**
  A handle for G is a $H \subseteq N$ s.t. all $n \in N$ there exsits a node $h \in H$ such that there is a directed path from h to n.
  $\{n\}$ is a handle iff n is a root.
- **Tree and forest:** in-degree, number of nodes with in-degree 0.
- **Dominator:** we call n' the dominator of n if every path from H to n contains n'.

Reverse Postorder

- **Spanning forests:** A spanning forest of a graph is a subgraph containing all the nodes and the subgraph is a forest.

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

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') ∈ 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 of DFSF Algorithm

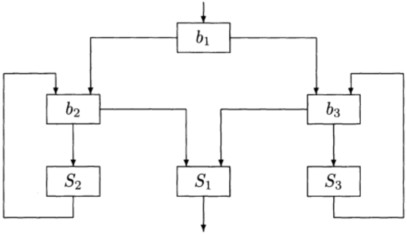

Figure C.1: A flow graph.

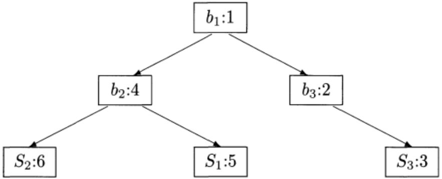

Figure C.2: A DFSF for the graph in Figure C.1.

Properties of Reverse Postorder

Categories of edges:

- 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, incluing self-loops.
- Cross edges: edges that go between nodes that are unrelated by the ancestor and descendant relations.

Properties of Reverse Postorder

Lemma C.9 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 of Table C.1. An edge $(n, n') \in A$ is a back edge if and only if rPostorder$[n] \geq$ rPostorder$[n']$ and is a self-loop if and only if rPostorder$[n] =$ rPostorder$[n']$. ∎

Proof idea:

- Result of self loop is trivial.
- "Only if" direction can be obtained through the algorithm.
  "If" direction relys on a fact that there is no crossedge of the type.

Corollary C.10 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 of Table C.1. Any cycle of G contains at least one back edge. ∎

Corollary C.11 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 of Table C.1. Then rPostorder topologically sorts T as well as the forward and cross edges. ∎

Loop Connectedness

- The loop connectedness of G with respect to a DFSF T is the largest number of back edges found in any cycle-free path of G. Write as $d(G)$.
- Dominator-back edge (n_1, n_2).
- Reducible graph: A directed graph with handle H is reducible iff $(N, A \backslash A_{db})$ is acyclic and H is still a handle.

Lemma C.12 Let $G = (N, A)$ be a reducible graph with handle H, T a depth first spanning forest for G and H, and rPostorder the associated ordering computed by the algorithm of Table C.1. Then an edge is a back edge if and only if it is a dominator-back edge. ■

Corollary C.14 Let $G = (N, A)$ be a reducible graph with handle H. Any cycle-free path in G beginning with a node in the handle, is monotonically increasing by the ordering rPostorder computed by the algorithm of Table C.1. ■

Part III: Different Iterating Methods

Iterating in Reverse Postorder

To concretize the algorithm, we can use FIFO or LIFO to instantiate the worklist.

---

empty = nil

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

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

Table 6.2: Iterating in last-in first-out order (LIFO).

Iterating in Reverse Postorder

The graph structure of a 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$[x_i]$).

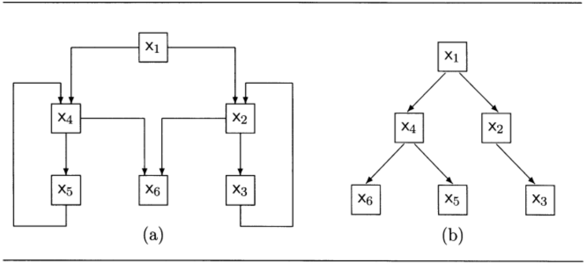

Figure 6.2: (a) Graphical representation. (b) Depth-first spanning tree.

Modify the Algorithm

The working list will be splitted into two list: current list `W.c` and pending list `W.p`.

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

Figure 6.3: Example: Reverse postorder iteration.

Comparison with LIFO

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

Figure 6.1: Example: LIFO iteration.

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)	

Table 6.4: The Round Robin Algorithm.

Theoretical Properties

Lemma 6.11 Given the assumptions, the algorithm of Table 6.4 computes the least solution of the given constraint system, $\mathcal{S}$. ∎

Lemma 6.12 Under the assumptions stated above, the algorithm of Table 6.4 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. ∎

Proof idea of Lemma 6.12:
A path contain d back edges will cause the while loop in step 2 to iterate at most $d + 1$ times.
Overall bound: $O((d + 1) \cdot b)$, where b is the number of the basic blocks.

Iterating through Strong Components

- **The algorithm:** SCCs are visited in topological order and within each SCC the nodes will be visited in reverse postorder.
- Outer loop, intermediate loop and inner loop.
- Priority of the constraint is obtained by pairs like (scc, rp).