**Script** generated by TTT

Title: Seidl: Programmoptimierung (26.11.2012)

Date: Mon Nov 26 15:11:58 CET 2012

Duration: 81:40 min

Pages: 39

The analysis iterates over all edges once:

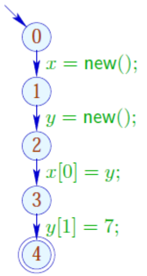
$$\pi = \{\{x\}, \{x[\,]\} \mid x \in Vars\};$$
$$\text{forall} \quad k = (\_, lab, \_) \quad \text{do} \quad \pi = [\![lab]\!]^\sharp\, \pi;$$

where:

$$\begin{array}{lll} [\![x = y;]\!]^\sharp\, \pi & = & \text{union}^*\,(\pi, x, y) \\ [\![x = y[e];]\!]^\sharp\, \pi & = & \text{union}^*\,(\pi, x, y[\,]) \\ [\![y[e] = x;]\!]^\sharp\, \pi & = & \text{union}^*\,(\pi, x, y[\,]) \\ [\![lab]\!]^\sharp\, \pi & = & \pi \qquad \text{otherwise} \end{array}$$



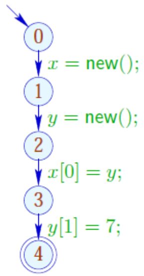
## ... in the Simple Example:




| | |
|---|---|
| | $\{\{x\}, \{y\}, \{x[\,]\}, \{y[\,]\}\}$ |
| $(0, 1)$ | $\{\{x\}, \{y\}, \{x[\,]\}, \{y[\,]\}\}$ |
| $(1, 2)$ | $\{\{x\}, \{y\}, \{x[\,]\}, \{y[\,]\}\}$ |
| $(2, 3)$ | $\{\{x\}, \boxed{\{y, x[\,]\}}, \{y[\,]\}\}$ |
| $(3, 4)$ | $\{\{x\}, \{y, x[\,]\}, \{y[\,]\}\}$ |



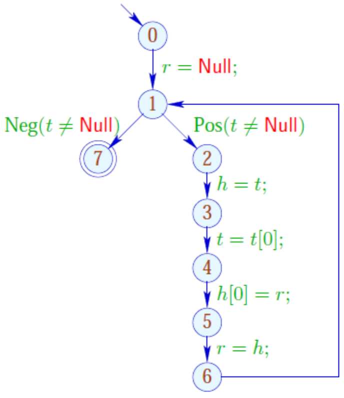
... in the Simple Example:



| | |
|---|---|
| | $\{\{x\}, \{y\}, \{x[\,]\}, \{y[\,]\}\}$ |
| $(0, 1)$ | $\{\{x\}, \{y\}, \{x[\,]\}, \{y[\,]\}\}$ |
| $(1, 2)$ | $\{\{x\}, \{y\}, \{x[\,]\}, \{y[\,]\}\}$ |
| $(2, 3)$ | $\{\{x\}, \boxed{\{y, x[\,]\}}, \{y[\,]\}\}$ |
| $(3, 4)$ | $\{\{x\}, \{y, x[\,]\}, \{y[\,]\}\}$ |

388

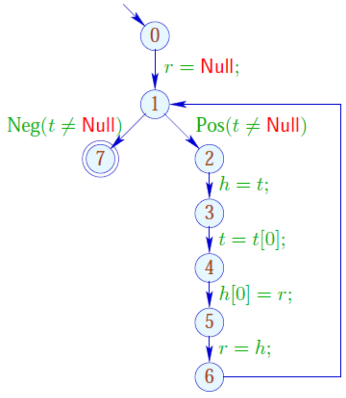
## ... in the More Complex Example:




| | $\{\{h\},\{r\},\{t\},\{h[\,]\},\{t[\,]\}\}$ |
|---|---|
| $(2,3)$ | $\{\boxed{\{h,t\}},\{r\},\boxed{\{h[\,],t[\,]\}}\}$ |
| $(3,4)$ | $\{\boxed{\{h,t,h[\,],t[\,]\}},\{r\}\}$ |
| $(4,5)$ | $\{\boxed{\{h,t,r,h[\,],t[\,]\}}\}$ |
| $(5,6)$ | $\{\{h,t,r,h[\,],t[\,]\}\}$ |



... in the More Complex Example:




| | $\{\{h\},\{r\},\{t\},\{h[\,]\},\{t[\,]\}\}$ |
|---|---|
| $(2,3)$ | $\{\boxed{\{h,t\}},\{r\},\boxed{\{h[\,],t[\,]\}}\}$ |
| $(3,4)$ | $\{\boxed{\{h,t,h[\,],t[\,]\}},\{r\}\}$ |
| $(4,5)$ | $\{\boxed{\{h,t,r,h[\,],t[\,]\}}\}$ |
| $(5,6)$ | $\{\{h,t,r,h[\,],t[\,]\}\}$ |



## Caveat:

In order to find something, we must assume that variables / addresses always receive a value before they are accessed.

## Complexity:

we have:

$\mathcal{O}(\#\,edges + \#\,Vars)$ calls of union*

$\mathcal{O}(\#\,edges + \#\,Vars)$ calls of find

$\mathcal{O}(\#\,Vars)$ calls of union

$\Longrightarrow$ We require efficient Union-Find data-structure :-)



The analysis iterates over all edges once:

$$\pi = \{\{x\},\{x[\,]\} \mid x \in Vars\};$$
$$\text{forall}\quad k = (\_, lab, \_)\quad \text{do}\quad \pi = [\![lab]\!]^\sharp\,\pi;$$

where:

$$\begin{array}{lll}
[\![x = y;]\!]^\sharp\,\pi & = & \text{union}^*\,(\pi, x, y) \\
[\![x = y[e];]\!]^\sharp\,\pi & = & \text{union}^*\,(\pi, x, y[\,]) \\
[\![y[e] = x;]\!]^\sharp\,\pi & = & \text{union}^*\,(\pi, x, y[\,]) \\
[\![lab]\!]^\sharp\,\pi & = & \pi \qquad \text{otherwise}
\end{array}$$

## Caveat:

In order to find something, we must assume that variables / addresses always receive a value before they are accessed.

## Complexity:

we have:

| | | |
|---|---|---|
| $\mathcal{O}(\#\, edges + \#\, Vars)$ | calls of | union* |
| $\mathcal{O}(\#\, edges + \#\, Vars)$ | calls of | find |
| $\mathcal{O}(\#\, Vars)$ | calls of | union |

$\Longrightarrow$ We require efficient Union-Find data-structure :-)



## Idea:

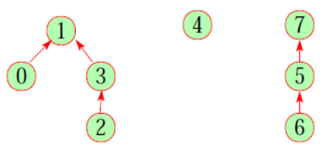
Represent partition of $U$ as directed forest:

- For $u \in U$ a reference $F[u]$ to the father is maintained;
- Roots are elements $u$ with $F[u] = u$ .

Single trees represent equivalence classes.

Their roots are their representatives ...





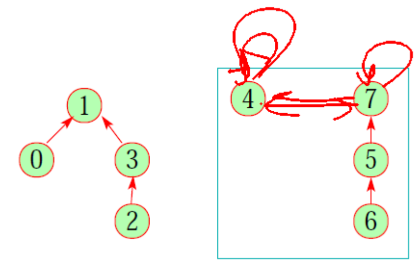

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 1 | 4 | 7 | 5 | 7 |

0 1 2 3 4 5 6 7

$\rightarrow$ find $(\pi, u)$ follows the father references :-)

$\rightarrow$ union $(\pi, u_1, u_2)$ re-directs the father reference of one $u_i$ ...






| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 1 | 4 | 7 | 5 | 7 |

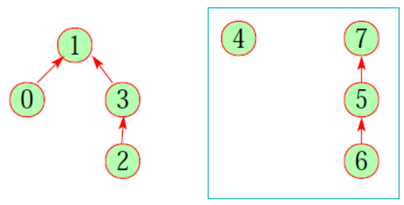
## The Costs:

| | | | |
|---|---|---|---|
| union | : | $\mathcal{O}(1)$ | :-) |
| find | : | $\mathcal{O}(depth(\pi))$ | :-( |

## Strategy to Avoid Deep Trees:

- Put the smaller tree below the bigger !
- Use find to compress paths ...

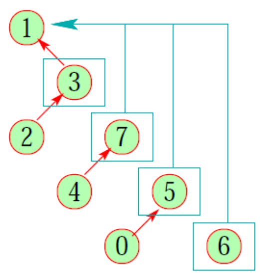





| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 1 | 4 | 7 | 5 | 7 |






| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 5 | 1 | 3 | 1 | 1 | 7 | 1 | 1 |



## Note:

- By this data-structure, $n$ union- und $m$ find operations require time $\mathcal{O}(n + m \cdot \alpha(n, n))$

  // $\alpha$ the inverse Ackermann-function :-)
- For our application, we only must modify union such that roots are from *Vars* whenever possible.
- This modification does not increase the asymptotic run-time. :-)

## Summary:

The analysis is extremely fast — but may not find very much.

## Note:

- By this data-structure, $n$ union- und $m$ find operations require time $\mathcal{O}(n + m \cdot \alpha(n,n))$

  // $\alpha$ the inverse Ackermann-function :-)
- For our application, we only must modify union such that roots are from *Vars* whenever possible.
- This modification does not increase the asymptotic run-time. :-)

## Summary:

The analysis is extremely fast — but may not find very much.



# Background 3: Fixpoint Algorithms

Consider: $x_i \sqsupseteq f_i(x_1, \ldots, x_n), \quad i = 1, \ldots, n$

## Observation:

RR-Iteration is inefficient:

→ We require a complete round in order to detect termination :-(

→ If in some round, the value of just one unknown is changed, then we still re-compute all :-(

→ The practical run-time depends on the ordering on the variables :-(



## Idea: Worklist Iteration

If an unknown $x_i$ changes its value, we re-compute all unknowns which depend on $x_i$. Technically, we require:

→ the lists $Dep\ f_i$ of unknowns which are accessed during evaluation of $f_i$. From that, we compute the lists:

$$I[x_i] = \{x_j \mid x_i \in Dep\ f_j\}$$

i.e., a list of all $x_j$ which depend on the value of $x_i$;

→ the values $D[x_i]$ of the $x_i$ where initially $D[x_i] = \bot$;

→ a list $W$ of all unknowns whose value must be recomputed ...



## The Algorithm:

$f_i : (Vars \to \mathbb{D}) \to \mathbb{D}$

```
W = [x_1, ..., x_n];
while (W ≠ []) {
    x_i = extract W;
    t = f_i eval;
    t = D[x_i] ⊔ t;
    if (t ≠ D[x_i]) {
        D[x_i] = t;
        W = append I[x_i] W;
    }
}
```

where : $eval\ x_j = D[x_j]$

## Example:

$$x_1 \supseteq \{a\} \cup x_3$$
$$x_2 \supseteq x_3 \cap \{a, b\}$$
$$x_3 \supseteq x_1 \cup \{c\}$$

| | $I$ |
|---|---|
| $x_1$ | $\{x_3\}$ |
| $x_2$ | $\emptyset$ |
| $x_3$ | $\{x_1, x_2\}$ |



## Example:

$$x_1 \supseteq \{a\} \cup x_3$$
$$x_2 \supseteq x_3 \cap \{a, b\}$$
$$x_3 \supseteq x_1 \cup \{c\}$$

| | $I$ |
|---|---|
| $x_1$ | $\{x_3\}$ |
| $x_2$ | $\emptyset$ |
| $x_3$ | $\{x_1, x_2\}$ |

| $D[x_1]$ | $D[x_2]$ | $D[x_3]$ | $W$ |
|---|---|---|---|
| $\emptyset$ | $\emptyset$ | $\emptyset$ | $\boxed{x_1}, x_2, x_3$ |
| $\boxed{\{a\}}$ | $\emptyset$ | $\emptyset$ | $\boxed{x_2}, x_3$ |
| $\{a\}$ | $\emptyset$ | $\emptyset$ | $\boxed{x_3}$ |
| $\{a\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_1}, x_2$ |
| $\{a, c\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_3}, x_2$ |
| $\{a, c\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_2}$ |
| $\{a, c\}$ | $\{a\}$ | $\{a, c\}$ | $[\,]$ |



## Theorem

Let $x_i \sqsupseteq f_i(x_1, \ldots, x_n)$, $i = 1, \ldots, n$ denote a constraint system over the complete lattice $\mathbb{D}$ of hight $h > 0$ .

(1) The algorithm terminates after at most $h \cdot N$ evaluations of right-hand sides where

$$N = \sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \qquad // \quad \text{size of the system} \quad :\text{-)}$$

(2) The algorithm returns a solution. If all $f_i$ are monotonic, it returns the least one.



## Example:

$$x_1 \supseteq \{a\} \cup x_3$$
$$x_2 \supseteq x_3 \cap \{a, b\}$$
$$x_3 \supseteq x_1 \cup \{c\}$$

| | $I$ |
|---|---|
| $x_1$ | $\{x_3\}$ |
| $x_2$ | $\emptyset$ |
| $x_3$ | $\{x_1, x_2\}$ |

| $D[x_1]$ | $D[x_2]$ | $D[x_3]$ | $W$ |
|---|---|---|---|
| $\emptyset$ | $\emptyset$ | $\emptyset$ | $\boxed{x_1}, x_2, x_3$ |
| $\{a\}$ | $\emptyset$ | $\emptyset$ | $\boxed{x_2}, x_3$ |
| $\{a\}$ | $\emptyset$ | $\emptyset$ | $\boxed{x_3}$ |
| $\{a\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_1}, x_2$ |
| $\{a, c\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_3}, x_2$ |
| $\{a, c\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_2}$ |
| $\{a, c\}$ | $\{a\}$ | $\{a, c\}$ | $[\,]$ |

## Theorem

Let $x_i \sqsupseteq f_i\,(x_1, \ldots, x_n)$ , $i = 1, \ldots, n$ denote a constraint system over the complete lattice $\mathbb{D}$ of hight $h > 0$ .

(1) The algorithm terminates after at most $h \cdot N$ evaluations of right-hand sides where

$$N = \sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \qquad // \text{ size of the system } \text{:-)}$$

(2) The algorithm returns a solution.
If all $f_i$ are monotonic, it returns the least one.



## Example:

$$x_1 \supseteq \{a\} \cup x_3$$
$$x_2 \supseteq x_3 \cap \{a, b\}$$
$$x_3 \supseteq x_1 \cup \{c\}$$

| | $I$ |
|---|---|
| $x_1$ | $\{x_3\}$ |
| $x_2$ | $\emptyset$ |
| $x_3$ | $\{x_1, x_2\}$ |

| $D[x_1]$ | $D[x_2]$ | $D[x_3]$ | $W$ |
|---|---|---|---|
| $\emptyset$ | $\emptyset$ | $\emptyset$ | $\boxed{x_1}, x_2, x_3$ |
| $\{a\}$ | $\emptyset$ | $\emptyset$ | $\boxed{x_2}, x_3$ |
| $\{a\}$ | $\emptyset$ | $\emptyset$ | $\boxed{x_3}$ |
| $\{a\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_1}, x_2$ |
| $\{a, c\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_3}, x_2$ |
| $\{a, c\}$ | $\emptyset$ | $\{a, c\}$ | $\boxed{x_2}$ |
| $\{a, c\}$ | $\{a\}$ | $\{a, c\}$ | [ ] |



## Theorem

Let $x_i \sqsupseteq f_i\,(x_1, \ldots, x_n)$ , $i = 1, \ldots, n$ denote a constraint system over the complete lattice $\mathbb{D}$ of hight $h > 0$ .

(1) The algorithm terminates after at most $h \cdot N$ evaluations of right-hand sides where

$$N = \sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \qquad // \text{ size of the system } \text{:-)}$$

(2) The algorithm returns a solution.
If all $f_i$ are monotonic, it returns the least one.

410

## Proof:

### Ad (1):

Every unknown $x_i$ may change its value at most $h$ times :-)

Each time, the list $I[x_i]$ is added to $W$ .

Thus, the total number of evaluations is:

$$\begin{aligned}
&\leq n + \textstyle\sum_{i=1}^{n} (h \cdot \#\,(I[x_i])) \\
&= n + h \cdot \textstyle\sum_{i=1}^{n} \#\,(I[x_i]) \\
&= n + h \cdot \textstyle\sum_{i=1}^{n} \#\,(Dep\ f_i) \\
&\leq h \cdot \textstyle\sum_{i=1}^{n} (1 + \#\,(Dep\ f_i)) \\
&= h \cdot N
\end{aligned}$$

Ad (2):

We only consider the assertion for monotonic $f_i$ .

Let $D_0$ denote the least solution. We show:

- $D_0[x_i] \sqsupseteq D[x_i]$ (all the time)
- $D[x_i] \not\sqsupseteq f_i\,\mathsf{eval} \implies x_i \in W$ (at exit of the loop body)
- On termination, the algo returns a solution :-))



## The Algorithm:

$$
\begin{array}{l}
W = [x_1, \ldots, x_n];\\
\mathsf{while}\ (W \neq [\,])\ \{\\
\qquad x_i \;=\; \mathsf{extract}\ W;\\
\qquad t \;=\; f_i\ \mathsf{eval};\\
\qquad t \;=\; D[x_i] \sqcup t;\\
\qquad \mathsf{if}\ (t \neq D[x_i])\ \{\\
\qquad\qquad D[x_i] \;=\; t;\\
\qquad\qquad W \;=\; \mathsf{append}\ I[x_i]\ W;\\
\qquad \}\\
\}\\
\text{where : } \mathit{eval}\ x_j \;=\; D[x_j]
\end{array}
$$



## Example:

$$
\begin{array}{lll}
x_1 & \supseteq & \{a\} \cup x_3\\
x_2 & \supseteq & x_3 \cap \{a, b\}\\
x_3 & \supseteq & x_1 \cup \{c\}
\end{array}
$$

| | $I$ |
|---|---|
| $x_1$ | $\{x_3\}$ |
| $x_2$ | $\emptyset$ |
| $x_3$ | $\{x_1, x_2\}$ |



## Warning:

- The algorithm relies on explicit dependencies among the unknowns.
  So far in our applications, these were obvious. This need not always be the case :-(
- We need some strategy for extract which determines the next unknown to be evaluated.
- It would be ingenious if we always evaluated first and then accessed the result ... :-)

$\implies$ recursive evaluation ...

## Idea:

→ If during evaluation of $f_i$, an unknown $x_j$ is accessed, $x_j$ is first solved recursively. Then $x_i$ is added to $I[x_j]$ :-)

$$\begin{aligned} \textsf{eval}\ x_i\ x_j \quad = \quad & \textsf{solve}\, x_j; \\ & I[x_j] = I[x_j] \cup \{x_i\}; \\ & D[x_j]; \end{aligned}$$

→ In order to prevent recursion to descend infinitely, a set *Stable* of unknown is maintained for which solve just looks up their values :-)

Initially, $Stable = \emptyset$ ...



## Discussion:

- In the example, fewer evaluations of right-hand sides are required than for RR-iteration :-)
- The algo also works for non-monotonic $f_i$ :-)
- For monotonic $f_i$, the algo can be simplified:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{;}$$

- In presence of widening, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t = D[x_i] \sqcup\!\!\!\sqcup\, t;}$$

- In presence of Narrowing, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \quad \Longrightarrow \quad \boxed{t = D[x_i] \sqcap\!\!\!\sqcap\, t;}$$



## Warning:

- The algorithm relies on explicit dependencies among the unknowns.

  So far in our applications, these were obvious. This need not always be the case :-(
- We need some strategy for extract which determines the next unknown to be evaluated.
- It would be ingenious if we always evaluated first and then accessed the result ... :-)

$\Longrightarrow$ recursive evaluation ...



## Idea:

→ If during evaluation of $f_i$, an unknown $x_j$ is accessed, $x_j$ is first solved recursively. Then $x_i$ is added to $I[x_j]$ :-)

$$\begin{aligned} \textsf{eval}\ x_i\ x_j \quad = \quad & \textsf{solve}\, x_j; \\ & I[x_j] = I[x_j] \cup \{x_i\}; \\ & D[x_j]; \end{aligned}$$

→ In order to prevent recursion to descend infinitely, a set *Stable* of unknown is maintained for which solve just looks up their values :-)

Initially, $Stable = \emptyset$ ...

415

$x_i \supseteq x_1$ else

## The Function solve :

solve $x_i$ = if $(x_i \notin \mathit{Stable})$ {
 $\mathit{Stable} = \mathit{Stable} \cup \{x_i\}$;
 $t = f_i$ (eval $x_i$);
 $t = D[x_i] \sqcup t$;
 if $(t \neq D[x_i])$ {
  $W = I[x_i]$;  $I[x_i] = \emptyset$;
  $D[x_i] = t$;
  $\mathit{Stable} = \mathit{Stable} \backslash W$;
  app solve $W$;
 }
}



## Example:

Consider our standard example:

$$\begin{aligned} x_1 &\supseteq \{a\} \cup x_3 \\ x_2 &\supseteq x_3 \cap \{a, b\} \\ x_3 &\supseteq x_1 \cup \{c\} \end{aligned}$$

A trace of the fixpoint algorithm then looks as follows:



| solve $x_2$ | eval $x_2$ $x_3$ | solve $x_3$ | eval $x_3$ $x_1$ | solve $x_1$ | eval $x_1$ $x_3$ | solve $x_3$ |
|---|---|---|---|---|---|---|
| | | | | | | stable! |
| | | | | | $I[x_3] = \{x_1\}$ $\Rightarrow$ $\emptyset$ | |
| | | | | $\boxed{D[x_1] = \{a\}}$ | | |
| | | | $I[x_1] = \{x_3\}$ $\Rightarrow$ $\{a\}$ | | | |
| | | $\boxed{D[x_3] = \{a, c\}}$ | | | | |
| | | $I[x_3] = \emptyset$ | | | | |
| | | solve $x_1$ | eval $x_1$ $x_3$ | solve $x_3$ stable! | | |
| | | | $I[x_3] = \{x_1\}$ $\Rightarrow$ $\{a, c\}$ | | | |
| | | $\boxed{D[x_1] = \{a, c\}}$ | | | | |
| | | $I[x_1] = \emptyset$ | | | | |
| | | solve $x_3$ | eval $x_3$ $x_1$ | solve $x_1$ stable! | | |
| | | | $I[x_1] = \{x_3\}$ $\Rightarrow$ $\{a, c\}$ | | | |
| | | $\boxed{\text{ok}}$ | | | | |
| | $I[x_3] = \{x_1, x_2\}$ $\Rightarrow$ $\{a, c\}$ | | | | | |
| $\boxed{D[x_2] = \{a\}}$ | | | | | | |




| solve $x_2$ | eval $x_2$ $x_3$ | solve $x_3$ | eval $x_3$ $x_1$ | solve $x_1$ | eval $x_1$ $x_3$ | solve $x_3$ |
|---|---|---|---|---|---|---|
| | | | | | | stable! |
| | | | | | $I[x_3] = \{x_1\}$ $\Rightarrow$ $\emptyset$ | |
| | | | | $\boxed{D[x_1] = \{a\}}$ | | |
| | | | $I[x_1] = \{x_3\}$ $\Rightarrow$ $\{a\}$ | | | |
| | | $\boxed{D[x_3] = \{a, c\}}$ | | | | |
| | | $I[x_3] = \emptyset$ | | | | |
| | | solve $x_1$ | eval $x_1$ $x_3$ | solve $x_3$ stable! | | |
| | | | $I[x_3] = \{x_1\}$ $\Rightarrow$ $\{a, c\}$ | | | |
| | | $\boxed{D[x_1] = \{a, c\}}$ | | | | |
| | | $I[x_1] = \emptyset$ | | | | |
| | | solve $x_3$ | eval $x_3$ $x_1$ | solve $x_1$ stable! | | |
| | | | $I[x_1] = \{x_3\}$ $\Rightarrow$ $\{a, c\}$ | | | |
| | | $\boxed{\text{ok}}$ | | | | |
| | $I[x_3] = \{x_1, x_2\}$ $\Rightarrow$ $\{a, c\}$ | | | | | |
| $\boxed{D[x_2] = \{a\}}$ | | | | | | |

419