

Background 3: Fixpoint Algorithms

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

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 $D[x_i] = \perp$;

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

The Algorithm:

```
 $W = [x_1, \dots, x_n];$   
while ( $W \neq []$ ) {  
     $x_i = \text{extract } W;$   
     $t = f_i \text{ eval};$   
     $t = D[x_i] \sqcup t;$   
    if ( $t \neq D[x_i]$ ) {  
         $D[x_i] = t;$   
         $W = \text{append } I[x_i] W;$   
    }  
}
```

where : $\text{eval } x_j = D[x_j]$

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	x_1, x_2, x_3
$\{a\}$	\emptyset	\emptyset	x_2, x_3
$\{a\}$	\emptyset	\emptyset	x_3
$\{a\}$	\emptyset	$\{a, c\}$	x_1, x_2
$\{a, c\}$	\emptyset	$\{a, c\}$	x_3, x_2
$\{a, c\}$	\emptyset	$\{a, c\}$	x_2
$\{a, c\}$	$\{a\}$	$\{a, c\}$	$[\]$

Theorem

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

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

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

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

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:

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

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

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:

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

- In presence of **widening**, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \implies \boxed{t = D[x_i] \sqcup\!\!\!\sqcup t;}$$

- In presence of **Narrowing**, we replace:

$$\boxed{t = D[x_i] \sqcup t;} \implies \boxed{t = D[x_i] \sqcap\!\!\!\sqcap t;}$$

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 ... :-)

⇒ 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 $I[x_j]$:-)

$\text{eval } x_i \ x_j = \text{solve } x_j;$

$I[x_j] = I[x_j] \cup \{x_i\};$

$D[x_j];$

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

Initially, $\textit{Stable} = \emptyset \dots$

The Function `solve` :

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



Helmut Seidl, TU München ;-)

Example:

Consider our standard example:

$$x_1 \supseteq \{a\} \cup x_3$$

$$x_2 \supseteq x_3 \cap \{a, b\}$$

$$x_3 \supseteq x_1 \cup \{c\}$$

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!

$$I[x_3] = \{x_1\}$$
$$\Rightarrow \emptyset$$

$$D[x_1] = \{a\}$$

$$I[x_1] = \{x_3\}$$
$$\Rightarrow \{a\}$$

$$D[x_3] = \{a, c\}$$

$$I[x_3] = \emptyset$$

solve x_1

eval $x_1 x_3$

solve x_3
stable!

$$I[x_3] = \{x_1\}$$
$$\Rightarrow \{a, c\}$$

$$D[x_1] = \{a, c\}$$

$$I[x_1] = \emptyset$$

solve x_3

eval $x_3 x_1$

solve x_1
stable!

$$I[x_1] = \{x_3\}$$
$$\Rightarrow \{a, c\}$$

ok

$$I[x_3] = \{x_1, x_2\}$$
$$\Rightarrow \{a, c\}$$

$$D[x_2] = \{a\}$$

- Evaluation starts with an **interesting** unknown x_i (e.g., the value at *stop*)
- Then **automatically** all unknowns are evaluated which influence x_i :-)
- The number of evaluations is often smaller than during worklist iteration ;-)
- The algorithm is more complex but does not rely on **pre-computation** of variable dependencies :-))
- It also works if variable dependencies during iteration **change !!!**

⇒ **interprocedural analysis**