Ad (2):

We only consider the assertion for monotonic \( f_i \).

Let \( D_0 \) denote the least solution. We show:

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

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:

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

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:

\[
D[x_i] = D[x_i] \cup t_i \implies :\n\]

- In presence of widening, we replace:

\[
D[x_i] = D[x_i] \cup t_i \implies D[x_i] = D[x_i] \uplus t_i
\]

- In presence of Narrowing, we replace:

\[
D[x_i] = D[x_i] \cup t_i \implies D[x_i] = D[x_i] \downarrow t_i
\]

... and update the test to \( t \sqsubseteq D[x_i] \).
The Algorithm

\[ W = [x_1, \ldots, x_n]; \]
while \((W \neq [])\) \{ 
\[ x_i = \text{extract } W; \]
\[ t = f_i \text{ eval}; \]
if \((t \not\subseteq D[x_i])\) \{ 
\[ D[x_i] = D[x_i] \cup t; \]
\[ W = \text{append } l[x_i] \ W; \]
\}
\}
where: \( \text{eval } x_j = D[x_j] \)

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:
\[
D[x_i] = D[x_i] \cup t; \quad \implies \quad ;
\]
- In presence of widening, we replace:
\[
D[x_i] = D[x_i] \cup t; \quad \implies \quad D[x_i] = D[x_i] \uplus t;
\]
- In presence of Narrowing, we replace:
\[
D[x_i] = D[x_i] \cap t; \quad \implies \quad D[x_i] = D[x_i] \cap t;
\]
... and update the test to \( t \sqsubseteq D[x_i] \).

The Algorithm

\[ W = [x_1, \ldots, x_n]; \]
while \((W \neq [])\) \{ 
\[ x_i = \text{extract } W; \]
\[ t = f_i \text{ eval}; \]
if \((t \not\subseteq D[x_i])\) \{ 
\[ D[x_i] = D[x_i] \cup t; \]
\[ W = \text{append } l[x_i] \ W; \]
\}
\}
where: \( \text{eval } x_j = D[x_j] \)

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:
\[
D[x_i] = D[x_i] \cup t; \quad \implies \quad ;
\]
- In presence of widening, we replace:
\[
D[x_i] = D[x_i] \cup t; \quad \implies \quad D[x_i] = D[x_i] \uplus t;
\]
- In presence of Narrowing, we replace:
\[
D[x_i] = D[x_i] \cap t; \quad \implies \quad D[x_i] = D[x_i] \cap t;
\]
... and update the test to \( t \sqsubseteq D[x_i] \).
Caveat

- 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 ...
  \[ \implies \text{recursive evaluation} \ldots \]

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 \( I[x_j] \).
  \[ \text{eval } x_j = \text{solve } x_j; \]
  \[ I[x_j] = I[x_j] \cup \{ x_i \}; \]
  \[ D[x_j]; \]
- In order to prevent recursion to descend infinitely, a set \( \text{Stable} \) of unknown is maintained for which \( \text{solve} \) just looks up their values.
  Initially, \( \text{Stable} = \emptyset \ldots \)

The Algorithm

\[
W = \{ x_1, \ldots, x_n \};
\]
while \( (W \neq \emptyset) \) {
  \[ x_i = \text{extract } W; \]
  \[ t = (f_i \text{ eval}); \]
  if \( (t \not\in D[x_i]) \) {
    \[ D[x_i] = D[x_i] \cup t; \]
    \[ W = \text{append } I[x_i] W; \]
  }
}
where: \( \text{eval } x_j = D[x_j] \)
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 $I[x_j]$.

\[
\begin{align*}
\text{eval } x_i & \quad \text{solve } x_j; \\
I[x_i] & = I[x_j] \cup \{x_i\}; \\
D[x_i] & = D[x_j];
\end{align*}
\]

→ In order to prevent recursion to descend infinitely, a set $\text{Stable}$ of unknown is maintained for which solve just looks up their values.
Initially, $\text{Stable} = \emptyset$ ...

The Function solve

\[
\begin{align*}
solve x_i & = \begin{cases} x_i & \text{if } (x_i \notin \text{Stable}) \\
\text{app solve } \{x_i\}; \\
\text{if } (t \notin D[x_i]) \\
D[x_i] & = D[x_i] \cup t; \\
W & = I[x_i]; \\
\text{Stable} & = \text{Stable} \setminus W; \\
\text{app solve } W;
\end{cases}
\end{align*}
\]

Example

Consider our standard example:

\[
\begin{align*}
x_1 & \supset \{a\} \cup x_3 \\
x_2 & \supset x_3 \cap \{a, b\} \\
x_3 & \supset x_1 \cup \{c\}
\end{align*}
\]

A trace of the fixpoint algorithm then looks as follows:
Example

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

\[
\begin{array}{|c|c|c|c|}
\hline
D[x_1] & D[x_2] & D[x_3] & W \\
\hline
\emptyset & \emptyset & \emptyset & x_1, x_2, x_3 \\
\{a\} & \emptyset & \emptyset & x_2, x_3 \\
\{a\} & \emptyset & \emptyset & x_3 \\
\{a\} & \emptyset & \emptyset & \emptyset \\
\{a, c\} & \emptyset & \emptyset & x_2 \\
\{a, c\} & \emptyset & \emptyset & x_3 \\
\{a, c\} & \emptyset & \emptyset & x_1 \\
\{a, c\} & \{a\} & \{a, c\} & \emptyset \\
\hline
\end{array}
\]

\[
I
\]

\[
\begin{array}{|c|c|c|}
\hline
x_1 & x_2 & x_3 \\
\hline
\{x_3\} & \emptyset & \{x_1, x_2\} \\
\{x_3\} & \emptyset & \emptyset \\
\{x_3\} & \emptyset & \{x_1, x_2\} \\
\{x_3\} & \emptyset & \emptyset \\
\{x_3\} & \emptyset & \emptyset \\
\{x_3\} & \emptyset & \emptyset \\
\{x_3\} & \emptyset & \emptyset \\
\{x_3\} & \emptyset & \emptyset \\
\hline
\end{array}
\]

→ Evaluation starts with an interesting unknown \(x_i\) (e.g., the value at \textbf{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 !!!

\[\implies\] interprocedural analysis

Caveat II

- The recursive algorithm may not evaluate right-hand sides atomically.
- Evaluations of right-hand sides may be continued which have been started with out-dated data. \[\implies\] in some cases, it may fail to determine the \textbf{least} solution !?!

Idea

- Identify outdated computations …
- Abort !!!
1.7 Eliminating Partial Redundancies

Example

\[ x = M[a], \quad y_1 = x + 1; \]

\[ w = x + 1; \]

\[ M[x] = y_1 + y_2; \]

// \( x + 1 \) is evaluated on every path ...
// on one path, however, even twice.

Idea

1. Insert assignments \( T_a = e \) such that \( e \) is available at all points where the value of \( e \) is required.
2. Thereby spare program points where \( e \) either is already available or will definitely be computed in future.

Expressions with the latter property are called very busy.

3. Replace the original evaluations of \( e \) by accesses to the variable \( T_a \).

\[ \rightarrow \quad \text{we require a novel analysis ...} \]
Goal

\[ x = M[a]; \]
\[ y_1 = x + 1; \]
\[ x = M[a]; \]
\[ T = x + 1; \]
\[ y_1 = T; \]
\[ M[x] = y_1 + y_2; \]
\[ M[x] = y_1 + T; \]

Idea

(1) Insert assignments \( T_e = e \); such that \( e \) is available at all points where the value of \( e \) is required.
(2) Thereby spare program points where \( e \) either is already available or will definitely be computed in future. Expressions with the latter property are called very busy.
(3) Replace the original evaluations of \( e \) by accesses to the variable \( T_e \).

\[ \Rightarrow \text{ we require a novel analysis ...} \]

Idea

An expression \( e \) is called busy along a path \( \pi \), if the expression \( e \) is evaluated before any of the variables \( x \in \text{Vars}(e) \) is overwritten.

\[ // \text{ backward analysis!} \]

\( e \) is called very busy at \( u \), if \( e \) is busy along every path \( \pi : u \rightarrow \text{stop} \).
An expression \( e \) is called **busy** along a path \( \pi \), if the expression \( e \) is evaluated before any of the variables \( x \in \text{Vars}(e) \) is overwritten.

// backward analysis!

\[ e \] is called **very busy** at \( u \), if \( e \) is busy along every path \( \pi : u \to^* \text{stop} \).

Accordingly, we require:

\[ B[u] = \bigcap\{\beta \mid \beta : u \to^* \text{stop}\} \]

where for \( \pi = k_1 \ldots k_m \):

\[ \beta^* = [k_1]^* \circ \ldots \circ [k_m]^* \]

---

Our complete lattice is given by:

\[ B = 2^\text{Expr} \text{Vars} \]

where \( \mathbb{C} = \square \).

The effect \([k]^*\) of an edge \( k = (u, \text{lab}, v)\) only depends on \( \text{lab} \), i.e., \([k]^* = [\text{lab}]^*\) where:

\[
\begin{align*}
[k]^* B &= B \\
[\text{Pos}(e)]^* B &= [\text{Neg}(e)]^* B = B \cup \{e\} \\
[x = e]^* B &= (B \setminus \text{Expr}_e) \cup \{e\} \\
[x = M[e]^* B &= (B \setminus \text{Expr}_e) \cup \{e\} \\
[M[e_1 = c_1]^* B &= B \cup \{c_1, e_2\} \\
[M[e_1 = c_2]^* B &= B \cup \{c_1, e_2\} \\
\end{align*}
\]

---

Our complete lattice is given by:

\[ B = 2^\text{Expr} \text{Vars} \]

where \( \mathbb{C} = \square \).

The effect \([k]^*\) of an edge \( k = (u, \text{lab}, v)\) only depends on \( \text{lab} \), i.e., \([k]^* = [\text{lab}]^*\) where:

\[
\begin{align*}
[k]^* B &= B \\
[\text{Pos}(e)]^* B &= [\text{Neg}(e)]^* B = B \cup \{e\} \\
[x = e]^* B &= (B \setminus \text{Expr}_e) \cup \{e\} \\
[x = M[e]^* B &= (B \setminus \text{Expr}_e) \cup \{e\} \\
[M[e_1 = c_1]^* B &= B \cup \{c_1, e_2\} \\
[M[e_1 = c_2]^* B &= B \cup \{c_1, e_2\} \\
\end{align*}
\]

The diagram at the bottom right shows a specific case where \( x = e \) and \( e = c_1 \), with \( e \in \mathbb{C} \).
These effects are all distributive. Thus, the least solution of the constraint system yields precisely the MOP — given that stop is reachable from every program point.

Example

\[ z = M[a]; \]
\[ y_1 = x + 1; \]
\[ y_2 = x + 1; \]
\[ M[x] = y_1 + y_2; \]

\[
\begin{array}{c|c}
\text{7} & \emptyset \\
\text{6} & \{y_1 + y_2\} \\
\text{5} & \{x + 1\} \\
\text{4} & \{x + 1\} \\
\text{3} & \{x + 1\} \\
\text{2} & \{x + 1\} \\
\text{1} & \emptyset \\
\text{0} & \emptyset \\
\end{array}
\]

A point \( u \) is called safe for \( e \), if \( e \in A[u] \cup B[u] \), i.e., \( e \) is either available or very busy.

Idea

- We insert computations of \( e \) such that \( e \) becomes available at all safe program points.
- We insert \( T_e = c_e \) after every edge \( \{u, lab, v\} \) with \( e \in B[v] \cap \{lab\} \cap (A[u] \cup B[u]) \).

In the Example

\[ z = M[a]; \]
\[ y_1 = x + 1; \]
\[ y_2 = x + 1; \]
\[ M[x] = y_1 + y_2; \]

\[
\begin{array}{c|c|c}
\text{A} & \text{B} \\
\hline
\text{0} & \emptyset & \emptyset \\
\text{1} & \emptyset & \emptyset \\
\text{2} & \emptyset & \{x + 1\} \\
\text{3} & \emptyset & \{x + 1\} \\
\text{4} & \{x + 1\} & \{x + 1\} \\
\text{5} & \emptyset & \{x + 1\} \\
\text{6} & \{x + 1\} & \{y_1 + y_2\} \\
\text{7} & \{x + 1, y_1 + y_2\} & \emptyset \\
\end{array}
\]

In the Example

\[ z = M[a]; \]
\[ y_1 = x + 1; \]
\[ y_2 = x + 1; \]
\[ M[x] = y_1 + y_2; \]

\[
\begin{array}{c|c|c}
\text{A} & \text{B} \\
\hline
\text{0} & \emptyset & \emptyset \\
\text{1} & \emptyset & \emptyset \\
\text{2} & \emptyset & \{x + 1\} \\
\text{3} & \emptyset & \{x + 1\} \\
\text{4} & \{x + 1\} & \{x + 1\} \\
\text{5} & \emptyset & \{x + 1\} \\
\text{6} & \{x + 1\} & \{y_1 + y_2\} \\
\text{7} & \{x + 1, y_1 + y_2\} & \emptyset \\
\end{array}
\]
A point \( u \) is called safe for \( e \), if \( e \in A[u] \cup B[u] \), i.e., \( e \) is either available or very busy.

**Idea**
- We insert computations of \( e \) such that \( e \) becomes available at all safe program points.
- We insert \( T_e = e; \) after every edge \( (u, \text{lab}, v) \) with
  \[
  e \in B[v] \setminus \{\text{lab}\} \cup (A[u] \cup B[u])
  \]

### Transformation 5.1
- \( T_e = e; \quad (e \in B[v] \setminus \{\text{lab}\} \cup (A[u] \cup B[u])) \)
- \( T_e = e; \quad (e \in B[v]) \)
Transformation 5.2

\[
\text{\begin{align*}
\mathbb{E} & = e; \\
x & = T_c;
\end{align*}}\]

// analogously for the other uses of \(e\)
// at old edges of the program.

Transformation 5.1

\[
\text{\begin{align*}
\mathbb{E} & = e; \\
x & = T_c; \\
(\mathbb{E} \in \mathbb{E}[\text{lab}]_A(\mathbb{E}[u] \cup \mathbb{E}[u]))
\end{align*}}\]

\[
\text{\begin{align*}
\mathbb{E} & = e; \\
(\mathbb{E} \in \mathbb{E}[\text{lab}])
\end{align*}}\]

Correctness

Let \(\pi\) denote a path reaching \(v\) after which a computation of an edge with \(e\) follows.

Then there is a maximal suffix of \(\pi\) such that for every edge \(k = (u, \text{lab}, u')\) in the suffix:

\[
e \in [\text{lab}]_A(\mathbb{E}[u] \cup \mathbb{E}[u])
\]

In the Example

\[
\begin{array}{c|cc}
\text{ step } & A & B \\
\hline
0 & 0 & 0 \\
1 & 0 & 0 \\
2 & 0 & \{x+1\} \\
3 & 0 & \{x+1\} \\
4 & \{x+1\} & \{x+1\} \\
5 & 0 & \{x+1\} \\
6 & \{x+1\} & \{y_1+y_2\} \\
7 & \{x+1, y_1+y_2\} & 0 \\
\end{array}
\]
Correctness

Let $\pi$ denote a path reaching $v$ after which a computation of an edge with $e$ follows.

Then there is a maximal suffix of $\pi$ such that for every edge $k = (u, \text{lab}, u')$ in the suffix:

$$e \in \{\text{lab}\}_{A}^{1}(A[u] \cup B[v])$$

In particular, no variable in $e$ receives a new value.

Then $T_e = e_i$ is inserted before the suffix.

Correctness

Let $\pi$ denote a path reaching $v$ after which a computation of an edge with $e$ follows.

Then there is a maximal suffix of $\pi$ such that for every edge $k = (u, \text{lab}, u')$ in the suffix:

$$e \in \{\text{lab}\}_{A}^{1}(A[u] \cup B[v])$$

In particular, no variable in $e$ receives a new value.

Then $T_e = e_i$ is inserted before the suffix.
We conclude

- Whenever the value of \( e \) is required, \( e \) is available.
  \[\implies\text{correctness of the transformation}\]

- Every \( T = e \); which is inserted into a path corresponds to an \( e \) which is replaced with \( T \).
  \[\implies\text{non-degradation of the efficiency}\]

1.8 Application: Loop-invariant Code

Example

\[
\text{for } (i = 0; i < n; i++) \\
    a[i] = b + 3; \\
\]

// The expression \( b + 3 \) is recomputed in every iteration.
// This should be avoided!

We conclude

- Whenever the value of \( e \) is required, \( e \) is available.
  \[\implies\text{correctness of the transformation}\]

- Every \( T = e \); which is inserted into a path corresponds to an \( e \) which is replaced with \( T \).
  \[\implies\text{non-degradation of the efficiency}\]

Caveat \( T = b + 3; \) may not be placed before the loop:

\[\implies\text{There is no decent place for } T = b + 3;\]
Caveat \( T = b + 3 \); may not be placed before the loop:

\[
\begin{align*}
0 & \quad i = 0; \\
1 & \quad T = b + 3; \\
7 & \quad \text{Neg}(i < n) \\
3 & \quad y = T; \\
4 & \quad A_1 = A + i; \\
5 & \quad M[A_1] = y; \\
4 & \quad i = i + 1;
\end{align*}
\]

\[\implies \text{There is no decent place for } T = b + 3.\]

The Control-flow Graph

\[
\begin{align*}
0 & \quad i = 0; \\
1 & \quad \text{Neg}(i < n) \\
7 & \quad \text{Pos}(i < n) \\
2 & \quad y = b + 3; \\
3 & \quad A_1 = A + i; \\
4 & \quad M[A_1] = y; \\
5 & \quad i = i + 1;
\end{align*}
\]

\[\implies \text{There is no decent place for } T = b + 3.\]

Caveat \( T = b + 3 \); may not be placed before the loop:

\[
\begin{align*}
0 & \quad i = 0; \\
1 & \quad T = b + 3; \\
7 & \quad \text{Neg}(i < n) \\
2 & \quad y = T; \\
3 & \quad A_1 = A + i; \\
4 & \quad M[A_1] = y; \\
4 & \quad i = i + 1;
\end{align*}
\]

\[\implies \text{There is no decent place for } T = b + 3.\]

... now there is a place for \( T = c \);

\[
\begin{align*}
0 & \quad i = 0; \\
1 & \quad \text{Neg}(i < n) \\
7 & \quad \text{Pos}(i < n) \\
2 & \quad y = T; \\
3 & \quad A_1 = A + i; \\
4 & \quad M[A_1] = y; \\
4 & \quad i = i + 1;
\end{align*}
\]
Idea: Transform into a do-while-loop ...

... now there is a place for $T = e_i$.

Idea: Transform into a do-while-loop ...