Script generated by TTT

Title: Seidl: Programmoptimierung (02.12.2015)

Date: Wed Dec 02 10:24:13 CET 2015

Duration: 87:36 min

Pages: 58

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:

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

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)
- $D[x_i] \not\sqsupseteq f_i$ eval $\Longrightarrow x_i \in W$ (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:

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

- In presence of widening, we replace:

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

- In presence of Narrowing, we replace:

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

... and update the test to $t \sqsubset D[x_i]$.

The Algorithm

```
$W = [x_1, \ldots, x_n];$
while $(W \neq [\,])$ {
        $x_i$  =  extract $W$;
        $t$    =  $f_i$ eval;
        if $(t \not\sqsubseteq D[x_i])$ {
                $D[x_i]$  =  $D[x_i] \cancel{\sqcup} \sqcap t;$
                $W$       =  append $I[x_i]$ $W$;
        }
}
where : $eval\ x_j = D[x_j]$
```

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:

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

- In presence of widening, we replace:

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

- In presence of Narrowing, we replace:

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

... and update the test to $t \sqsubset D[x_i]$.

The Algorithm

```
$W = [x_1, \ldots, x_n];$
while $(W \neq [\,])$ {
        $x_i$  =  extract $W$;
        $t$    =  $f_i$ eval;
        if $(t \not\sqsubseteq D[x_i])$ {
                $D[x_i]$  =  $D[x_i] \cancel{\sqcup} \sqcap t;$
                $W$       =  append $I[x_i]$ $W$;
        }
}
where : $eval\ x_j = D[x_j]$
```

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:

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

- In presence of widening, we replace:

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

- In presence of Narrowing, we replace:

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

... and update the test to $t \sqsubset D[x_i]$.

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 ...

$\Longrightarrow$ recursive evaluation ...

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 ...

$\Longrightarrow$ 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 $I[x_j]$.

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

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

Initially, $\mathit{Stable} = \emptyset$ ...

The Algorithm

```
$W = [x_1, \ldots, x_n]$;
while ($W \neq [\,]$) {
        $x_i$ = extract $W$;
        $t$ = $f_i$ eval;
        if ($t \not\sqsubseteq D[x_i]$) {
                $D[x_i]$ = $D[x_i] \sqcup t$;
                $W$ = append $I[x_i]$ $W$;
        }
}
where : $\mathit{eval}\ x_j = D[x_j]$
```

f_i (eval x_i)

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 $I[x_j]$.

```
eval $x_i$ $x_j$  =  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, $Stable = \emptyset$ ...

The Function solve

```
solve $x_i$  =  if ($x_i \notin Stable$) {
                    $Stable = Stable \cup \{x_i\}$;
                    $t = f_i$ (eval $x_i$);
                    if ($t \not\sqsubseteq D[x_i]$) {
                        $D[x_i] = D[x_i] \sqcup t$;
                        $W = I[x_i]$;    $I[x_i] = \emptyset$;
                        $Stable = Stable \backslash W$;
                        app solve $W$;
                    }
                }
```

The Function solve

app f [x1 ; x2 ; x3]

```
solve $x_i$  =  if ($x_i \notin Stable$) {
                    $Stable = Stable \cup \{x_i\}$;
                    $t = f_i$ (eval $x_i$);
                    if ($t \not\sqsubseteq D[x_i]$) {
                        $D[x_i] = D[x_i] \sqcup t$;
                        $W = I[x_i]$;    $I[x_i] = \emptyset$;
                        $Stable = Stable \backslash W$;
                        app solve $W$;
                    }
                }
```

let rec app f = function [] → ()
| [x] → f x
| x::xs → f x ; app f xs

Example

Consider our standard example:

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

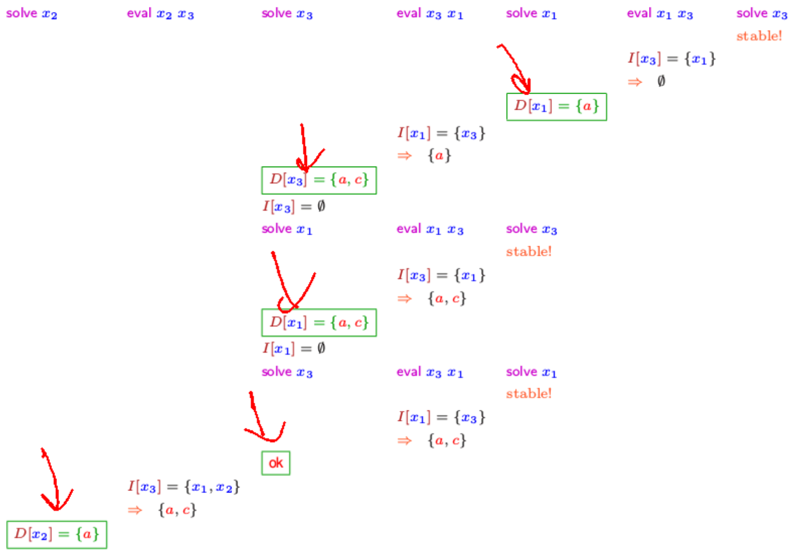
A trace of the fixpoint algorithm then looks as follows:

Example

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

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

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



→ 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 !!!

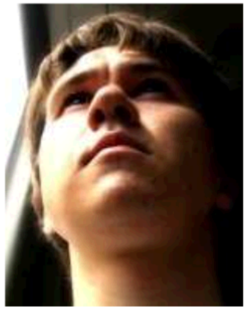
$\Longrightarrow$ interprocedural analysis

Caveat II

- The recursive algorithm may not evaluate right-hand sides atomicly.
- Evaluations of right-hand sides may be continued which have been started with out-dated data. $\Longrightarrow$ in some cases, it may fail to determine the least solution !?!

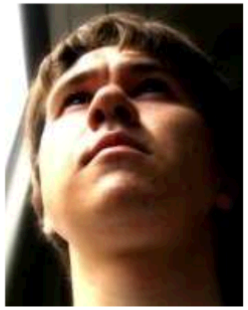
Idea

- Identify outdated computations ...
- Abort !!



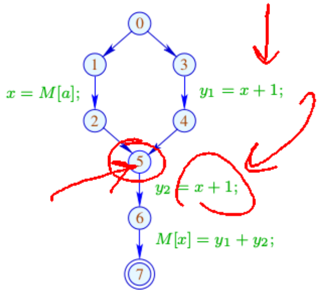
Aleks Karbyshev, TU München

Aleks Karbyshev, TU München



1.7 Eliminating Partial Redundancies

Example



// $x + 1$ is evaluated on every path ...

// on one path, however, even twice.

Idea

(1) Insert assignments $T_e = e;$ such that e is available at all points where the value of e is required.

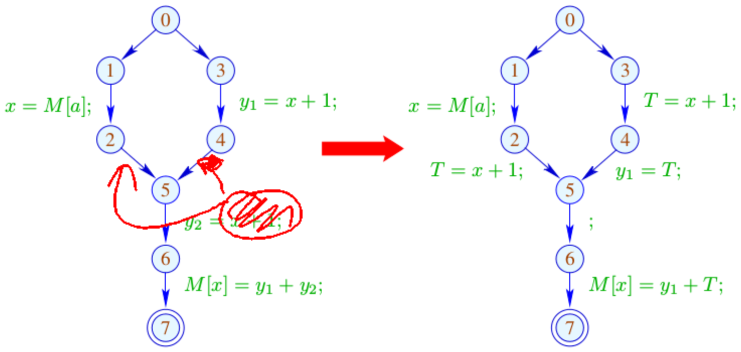
(2) Thereby spare program points where e either is already available or will definitely be computed in future.

Expressions with the latter property are called very busy.

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

$\Longrightarrow$ we require a novel analysis ...

Goal



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.

$\Longrightarrow$ we require a novel analysis ...

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.

$\Longrightarrow$ we require a novel analysis ...

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

// backward analysis!

e is called very busy at u, if e is busy along every path $\pi : u \to^* stop$.

An expression e is called busy along a path π, if the expression e is evaluated before any of the variables $x \in Vars(e)$ is overwriten.

$//$ backward analysis!

e is called very busy at u, if e is busy along every path $\pi : u \rightarrow^* stop$.

Accordingly, we require:

$$\mathcal{B}[u] \;=\; \bigcap \{ [\![\pi]\!]^\sharp\, \emptyset \mid \pi : u \rightarrow^* stop \}$$

where for $\pi = k_1 \ldots k_m$:

$$[\![\pi]\!]^\sharp \;=\; [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_m]\!]^\sharp$$

Our complete lattice is given by:

$$\mathbb{B} = 2^{Expr \setminus Vars} \qquad \text{with} \qquad \sqsubseteq \;=\; \supseteq$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, lab, v)$ only depends on lab, i.e., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ where:

$$\begin{array}{lcl} [\![;]\!]^\sharp\, B & = & B \\ [\![Pos(e)]\!]^\sharp\, B & = & [\![Neg(e)]\!]^\sharp\, B \quad = \quad B \cup \{e\} \\ [\![x = e;]\!]^\sharp\, B & = & (B \backslash Expr_x) \cup \{e\} \\ [\![x = M[e];]\!]^\sharp\, B & = & (B \backslash Expr_x) \cup \{e\} \\ [\![M[e_1] = e_2;]\!]^\sharp\, B & = & B \cup \{e_1, e_2\} \end{array}$$

An expression e is called busy along a path π, if the expression e is evaluated before any of the variables $x \in Vars(e)$ is overwriten.

$//$ backward analysis!

e is called very busy at u, if e is busy along every path $\pi : u \rightarrow^* stop$.

Accordingly, we require:

$$\mathcal{B}[u] \;=\; \bigcap \{ [\![\pi]\!]^\sharp\, \emptyset \mid \pi : u \rightarrow^* stop \}$$

where for $\pi = k_1 \ldots k_m$:

$$[\![\pi]\!]^\sharp \;=\; [\![k_1]\!]^\sharp \circ \ldots \circ [\![k_m]\!]^\sharp$$

Our complete lattice is given by:

$$\mathbb{B} = 2^{Expr \setminus Vars} \qquad \text{with} \qquad \sqsubseteq \;=\; \supseteq$$

The effect $[\![k]\!]^\sharp$ of an edge $k = (u, lab, v)$ only depends on lab, i.e., $[\![k]\!]^\sharp = [\![lab]\!]^\sharp$ where:

$$\begin{array}{lcl} [\![;]\!]^\sharp\, B & = & B \\ [\![Pos(e)]\!]^\sharp\, B & = & [\![Neg(e)]\!]^\sharp\, B \quad = \quad B \cup \{e\} \\ [\![x = e;]\!]^\sharp\, B & = & (B \backslash Expr_x) \cup \{e\} \\ [\![x = M[e];]\!]^\sharp\, B & = & (B \backslash Expr_x) \cup \{e\} \\ [\![M[e_1] = e_2;]\!]^\sharp\, B & = & B \cup \{e_1, e_2\} \end{array}$$

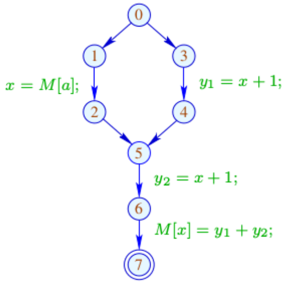
$T_1 = e_1$

$T_2 = e_2$

$M[T_1] = T_2$

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



7	$\emptyset$
6	$\{y_1 + y_2\}$
5	$\{x + 1\}$
4	$\{x + 1\}$
3	$\{x + 1\}$
2	$\{x + 1\}$
1	$\emptyset$
0	$\emptyset$

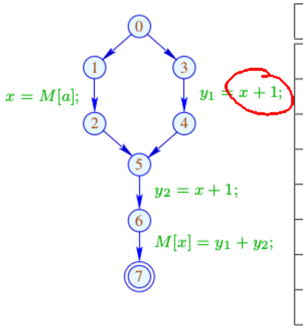
A point u is called safe for e, if $e \in \mathcal{A}[u] \cup \mathcal{B}[u]$, i.e., e is either available or very busy.

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, lab, v) with

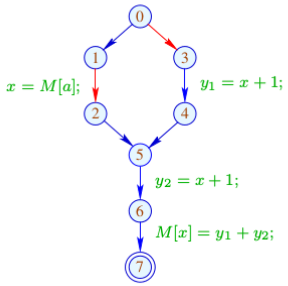
$$e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

In the Example



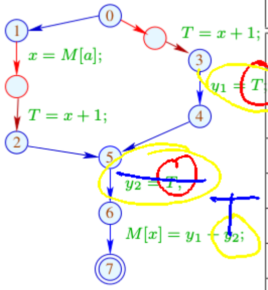
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x + 1\}$
3	$\emptyset$	$\{x + 1\}$
4	$\{x + 1\}$	$\{x + 1\}$
5	$\emptyset$	$\{x + 1\}$
6	$\{x + 1\}$	$\{y_1 + y_2\}$
7	$\{x + 1, y_1 + y_2\}$	$\emptyset$

In the Example



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x + 1\}$
3	$\emptyset$	$\{x + 1\}$
4	$\{x + 1\}$	$\{x + 1\}$
5	$\emptyset$	$\{x + 1\}$
6	$\{x + 1\}$	$\{y_1 + y_2\}$
7	$\{x + 1, y_1 + y_2\}$	$\emptyset$

Im Example



	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\{y_1+y_2\}$
7	$\{x+1, y_1+y_2\}$	$\emptyset$

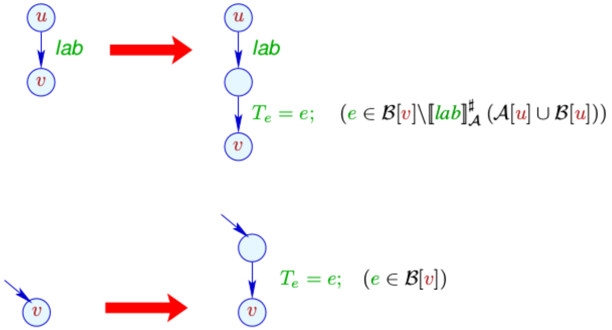
A point u is called safe for e, if $e \in \mathcal{A}[u] \cup \mathcal{B}[u]$, i.e., e is either available or very busy.

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, lab, v) with

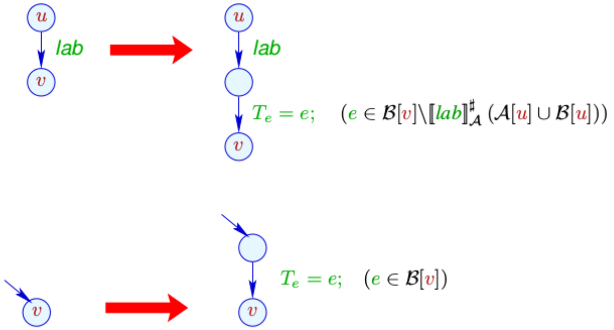
$$e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

Transformation 5.1



$(e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u]))$

$(e \in \mathcal{B}[v])$


Transformation 5.1

$(e \in \mathcal{B}[v] \backslash [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u]))$

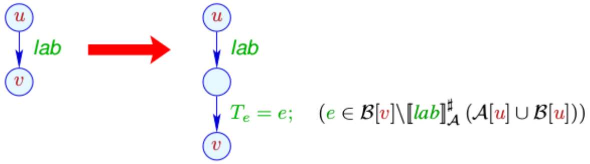
$(e \in \mathcal{B}[v])$

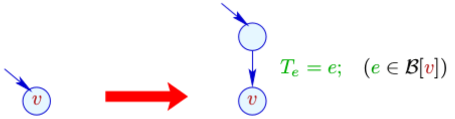
Transformation 5.2

// analogously for the other uses of e

// at old edges of the program.

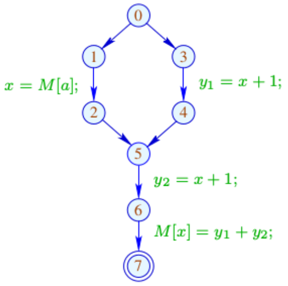
Transformation 5.1





In the Example

$T_e = e$



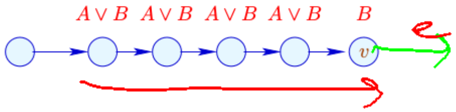
	$\mathcal{A}$	$\mathcal{B}$
0	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{x+1\}$
3	$\emptyset$	$\{x+1\}$
4	$\{x+1\}$	$\{x+1\}$
5	$\emptyset$	$\{x+1\}$
6	$\{x+1\}$	$\{y_1+y_2\}$
7	$\{x+1, y_1+y_2\}$	$\emptyset$

Correctness

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

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

$$e \in [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

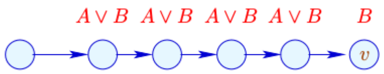


Correctness

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

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

$$e \in [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$



Correctness

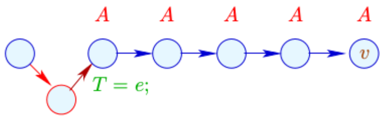
Let π denote a path reaching v after which a computation of an edge with e follows.

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

$$e \in [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

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

Then $T_e = e;$ is inserted before the suffix.



Correctness

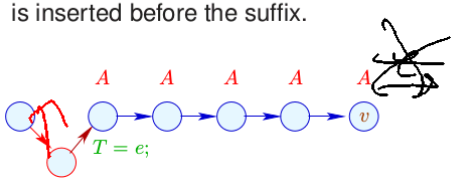
Let π denote a path reaching v after which a computation of an edge with e follows.

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

$$e \in [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

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

Then $T_e = e;$ is inserted before the suffix.



Correctness

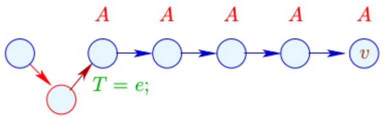
Let π denote a path reaching v after which a computation of an edge with e follows.

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

$$e \in [\![lab]\!]^\sharp_{\mathcal{A}}(\mathcal{A}[u] \cup \mathcal{B}[u])$$

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

Then $T_e = e;$ is inserted before the suffix.



We conclude

- Whenever the value of e is required, e is available.

  $\Longrightarrow$ correctness of the transformation

- Every $T = e;$ which is inserted into a path corresponds to an e which is replaced with T.

  $\Longrightarrow$ non-degradation of the efficiency

We conclude

- Whenever the value of e is required, e is available.

  $\Longrightarrow$ correctness of the transformation

- Every $T = e;$ which is inserted into a path corresponds to an e which is replaced with T.

  $\Longrightarrow$ non-degradation of the efficiency

1.8 Application: Loop-invariant Code

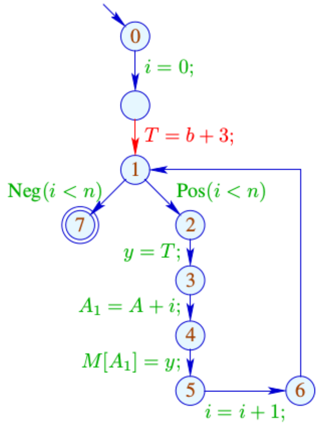
Example

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

// The expression $b + 3$ is recomputed in every iteration.

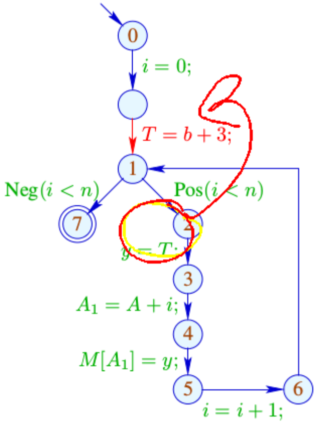
// This should be avoided !

Caveat $T = b + 3;$ may not be placed before the loop :



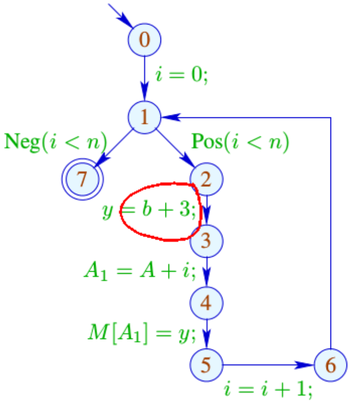
$\Longrightarrow$ There is no decent place for $T = b + 3;$.

Caveat $T = b + 3;$ may not be placed before the loop :

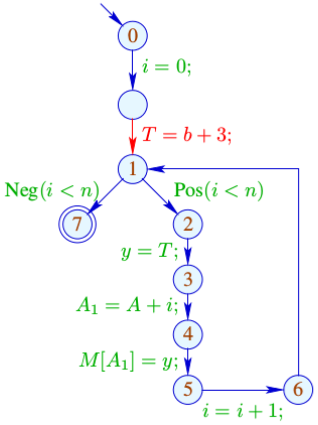


⟹ There is no decent place for $T = b + 3;$.

The Control-flow Graph

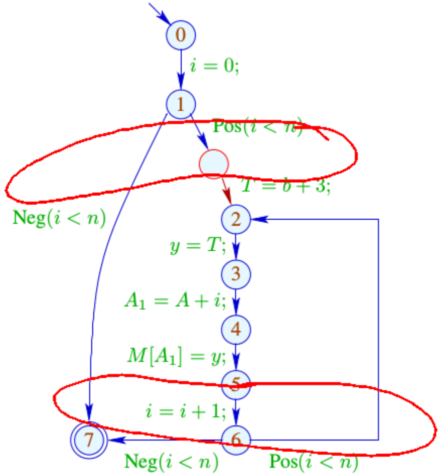


Caveat $T = b + 3;$ may not be placed before the loop :

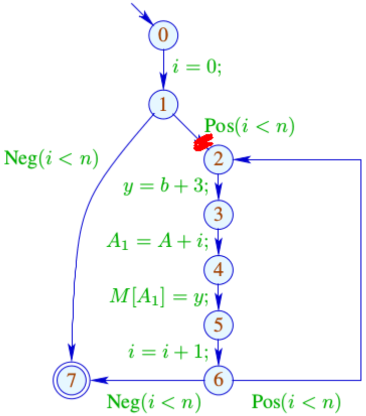


⟹ There is no decent place for $T = b + 3;$.

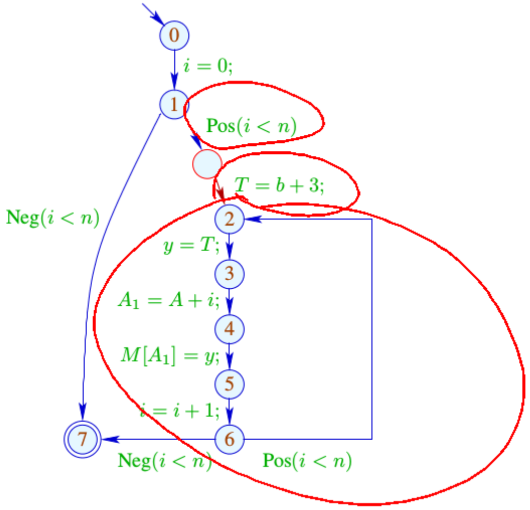
... now there is a place for $T = e;$.



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



... now there is a place for $T = e;$.



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