Strict Sequential Constructiveness

Alexander Schulz-Rosengarten
Reinhard von Hanxleden, Michael Mendler

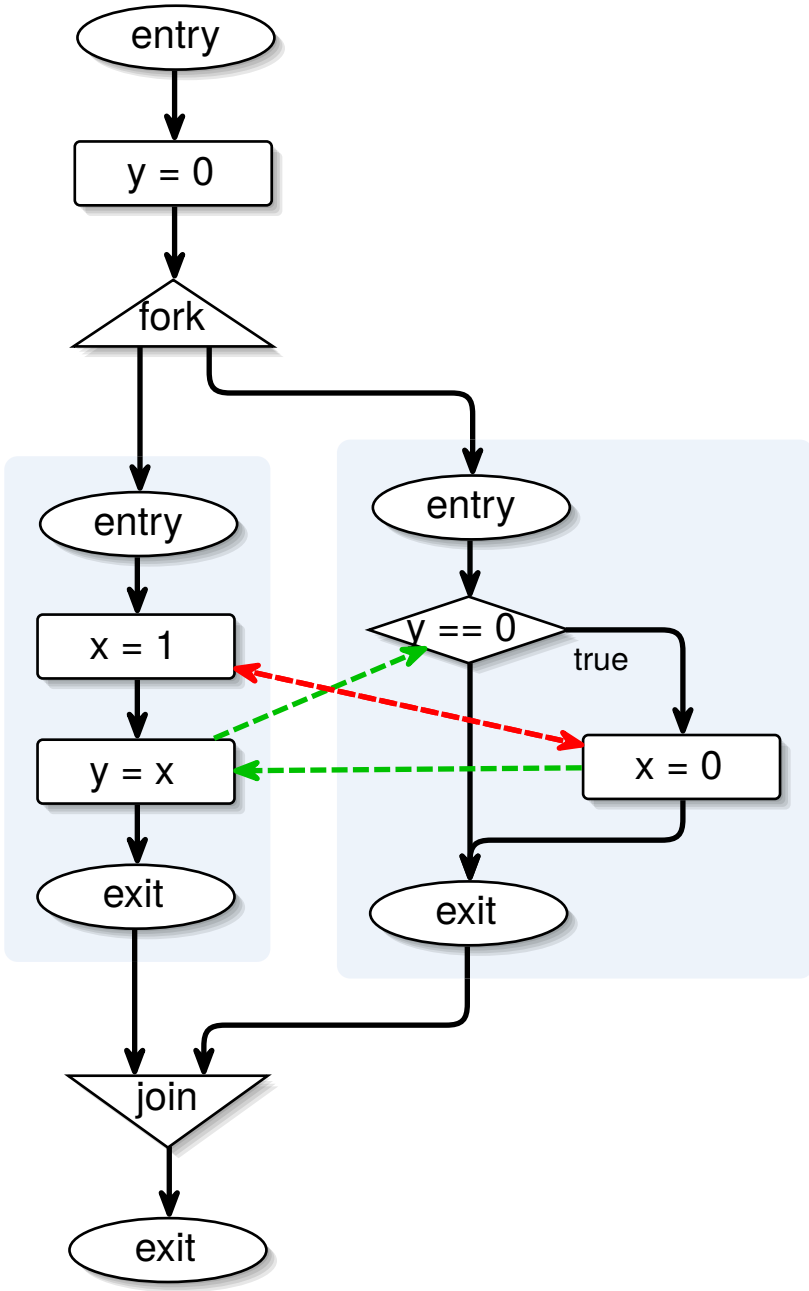
Why restricting sequential constructiveness?

Motivation: Program P10

```
1  module P10
2  int x, y;
3  {
4    y = 0;                  //S1
5    fork
6      x = 1;                //S2
7      y = x                 //S3
8    par
9      if y == 0 then        //S4
10       x = 0               //S5
11     end
12   join
13 }
```

Motivation: Program P10

```
1  module P10
2  int x, y;
3  {
4    y = 0;                    //S1
5    fork
6      x = 1;                  //S2
7      y = x                   //S3
8    par
9      if y == 0 then          //S4
10       x = 0                 //S5
11     end
12   join
13 }
```

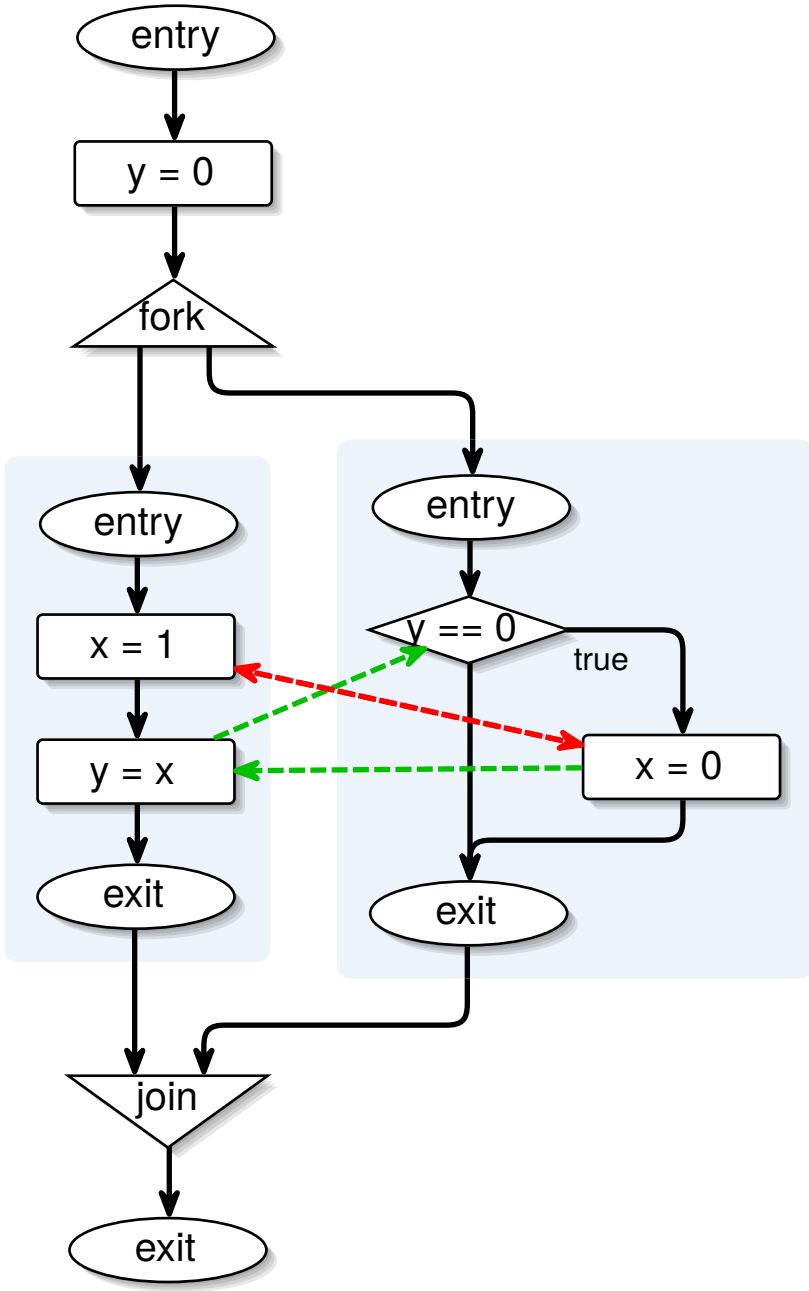


Motivation: Program P10

```
1  module P10
2  int x, y;
3  {
4    y = 0;                    //S1
5    fork
6      x = 1;                  //S2
7      y = x                   //S3
8    par
9      if y == 0 then          //S4
10       x = 0                 //S5
11     end
12   join
13 }
```

SC-admissible Schedule

S1 — S2 — S3 — S4



Problem

P10

- is reactive ($\exists$ SC-admissible Run)

Problem

P10

- is reactive ($\exists$ SC-admissible Run)
- is determinate ($\forall$ SC-admissible Runs : same determinate macro responses)

Problem

P10

- is reactive ($\exists$ SC-admissible Run)
- is determinate ($\forall$ SC-admissible Runs : same determinate macro responses)
- is Sequentially Constructive

Problem

P10

- is reactive ($\exists$ SC-admissible Run)
- is determinate ($\forall$ SC-admissible Runs : same determinate macro responses)
- is Sequentially Constructive
- but is executed in a speculative manner

YOU SHALL NOT

SPECULATE!

Problem

P10:

- is reactive ($\exists$ SC-admissible Run)
- is determinate ($\forall$ SC-admissible Runs : same determinate macro responses)
- is Sequentially Constructive
- but is executed in a speculative manner

Problem

P10:

- is reactive ($\exists$ SC-admissible Run)
- is determinate ($\forall$ SC-admissible Runs : same determinate macro responses)
- is Sequentially Constructive
- but is executed in a speculative manner

Problem

The SC MoC allows speculation

Problem

P10:

- is reactive ($\exists$ SC-admissible Run)
- is determinate ($\forall$ SC-admissible Runs : same determinate macro responses)
- is Sequentially Constructive
- but is executed in a speculative manner

Problem

The SC MoC allows speculation
$\Rightarrow$ SC programs may form non-constructive (delay sensitive) circuits

Restricting Sequential Constructiveness

Strict Sequential Constructiveness
is
Sequential Constructiveness without speculation

Restricting Sequential Constructiveness

Strict Sequential Constructiveness
is
Sequential Constructiveness without speculation

How can we eliminate speculation?

Restricting Sequential Constructiveness

Idea

Ground SC in constructiveness in the spirit of Esterel

Restricting Sequential Constructiveness

Idea

Ground SC in constructiveness in the spirit of Esterel

Constructive Esterel:

- has no speculation

Restricting Sequential Constructiveness

Idea

Ground SC in constructiveness in the spirit of Esterel

Constructive Esterel:

- has no speculation
- always transforms into delay-insensitive (constructive) circuits

Restricting Sequential Constructiveness

Idea

Ground SC in constructiveness in the spirit of Esterel

Constructive Esterel:

- has no speculation
- always transforms into delay-insensitive (constructive) circuits

Restricting Sequential Constructiveness

Idea

Ground SC in constructiveness in the spirit of Esterel

Constructive Esterel:

- has no speculation
- always transforms into delay-insensitive (constructive) circuits

but

- requires globally consistent signal states

Restricting Sequential Constructiveness

Idea

Ground SC in constructiveness in the spirit of Esterel

Constructive Esterel:

- has no speculation
- always transforms into delay-insensitive (constructive) circuits

but

- requires globally consistent signal states
- has no shared variables (write & read)

Concept

Concept

1. Transformation into SSA form

Concept

1. Transformation into SSA form
   - sequential variable behavior

Concept

1. Transformation into SSA form
   - sequential variable behavior
   - iur protocol

Concept

1. Transformation into SSA form
   - sequential variable behavior
   - iur protocol
2. Translation into Esterel

Concept

1. Transformation into SSA form
   - sequential variable behavior
   - iur protocol
2. Translation into Esterel
   - signal encoding

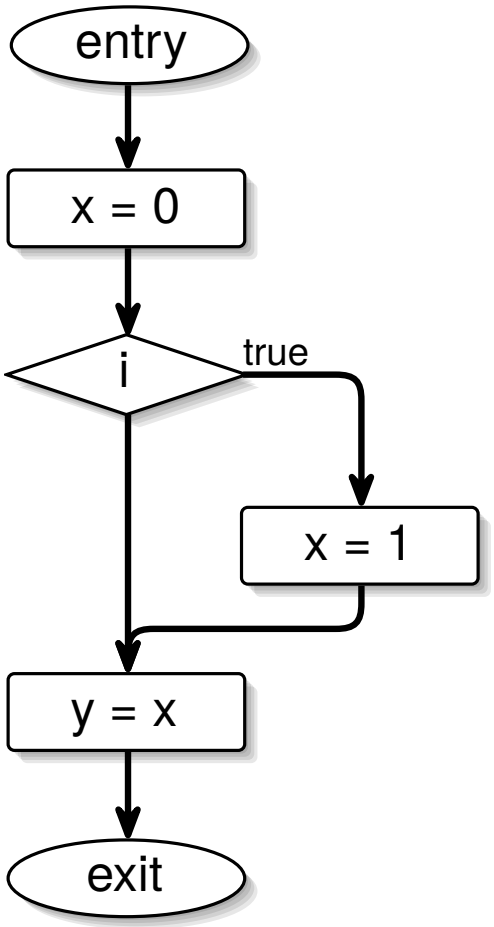
Concept

1. Transformation into SSA form
   - sequential variable behavior
   - iur protocol
2. Translation into Esterel
   - signal encoding
   - SSA functions encoding

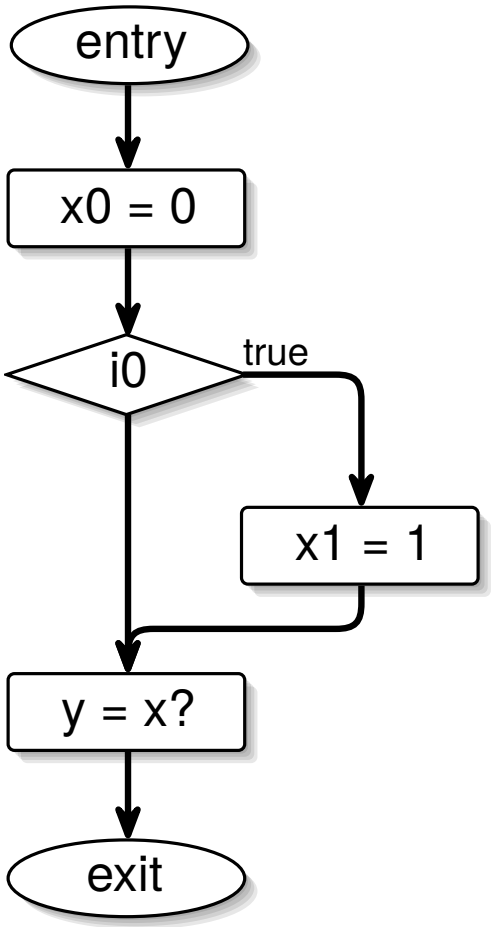
Concept

1. Transformation into SSA form
   - sequential variable behavior
   - iur protocol
2. Translation into Esterel
   - signal encoding
   - SSA functions encoding
3. Esterel constructiveness check

Static Single Assignment Form



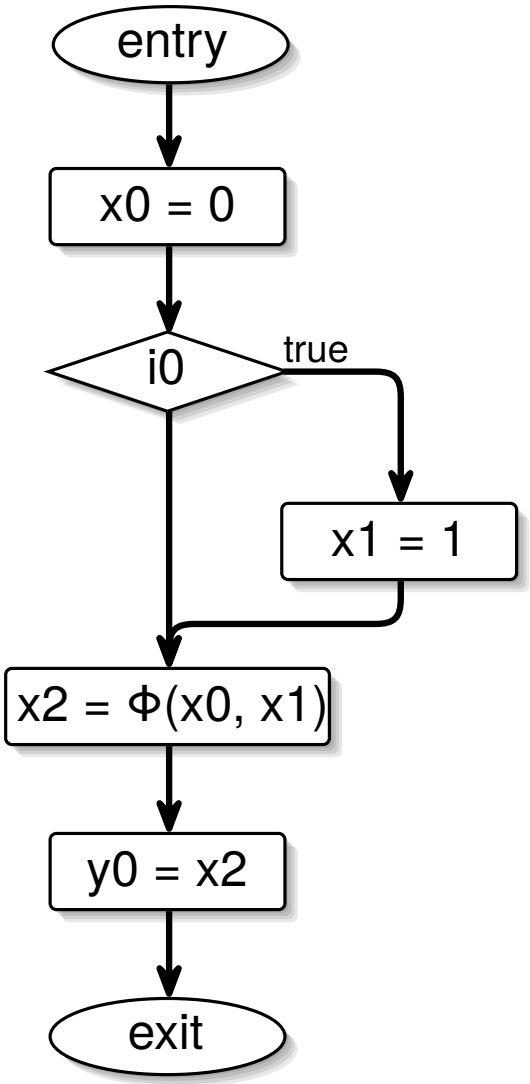
Static Single Assignment Form



Procedure

1. Split up variables into versions

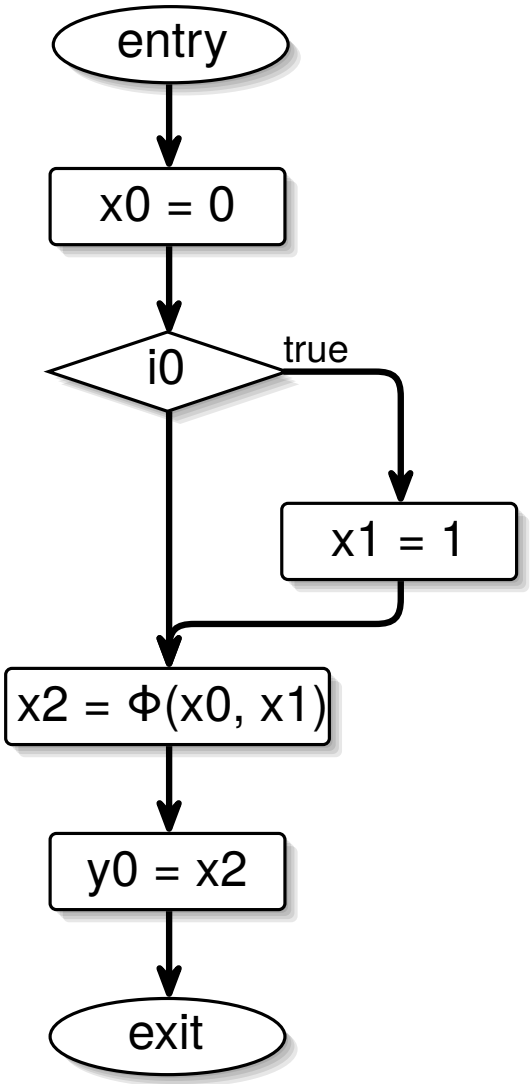
Static Single Assignment Form



Procedure

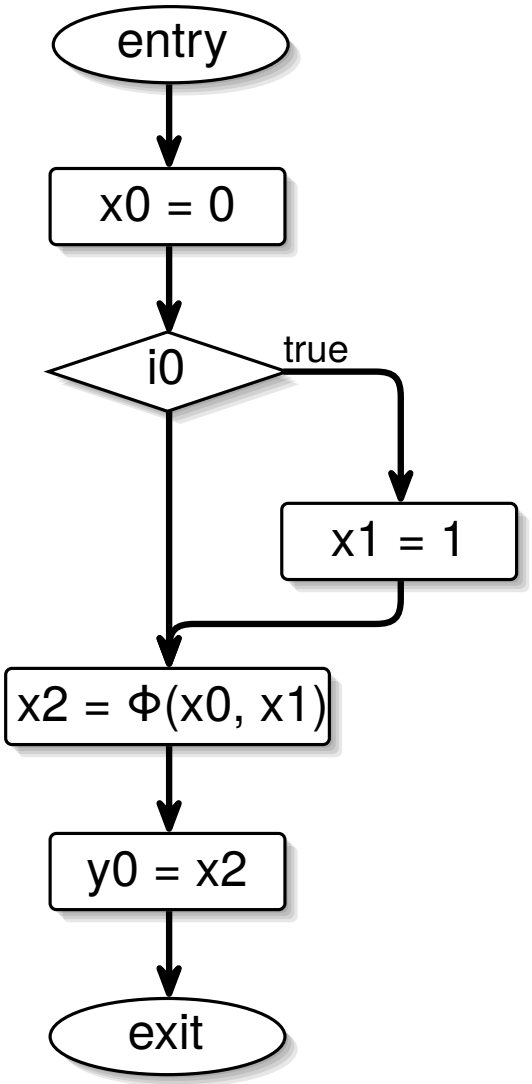
1. Split up variables into versions
2. Introduce ϕ-functions to merge variable versions

Static Single Assignment Form



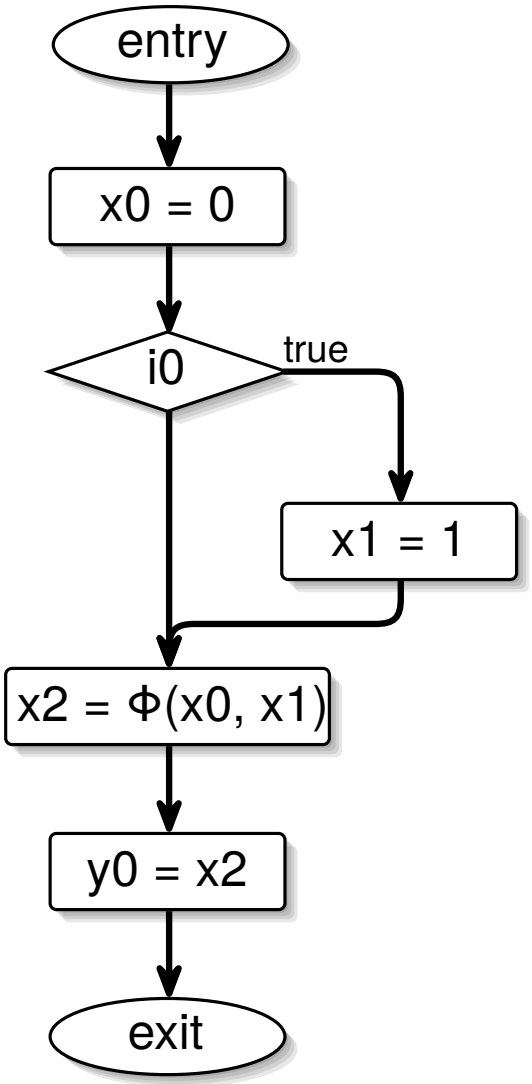
- Each variable is assigned only once (statically)

Static Single Assignment Form



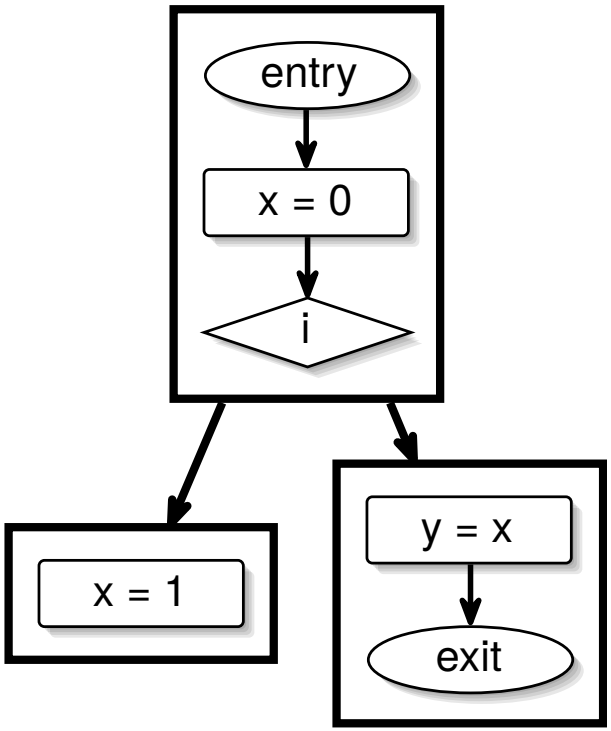
- Each variable is assigned only once (statically)
- Only one reaching definition for each read (def-use-chains)

Static Single Assignment Form



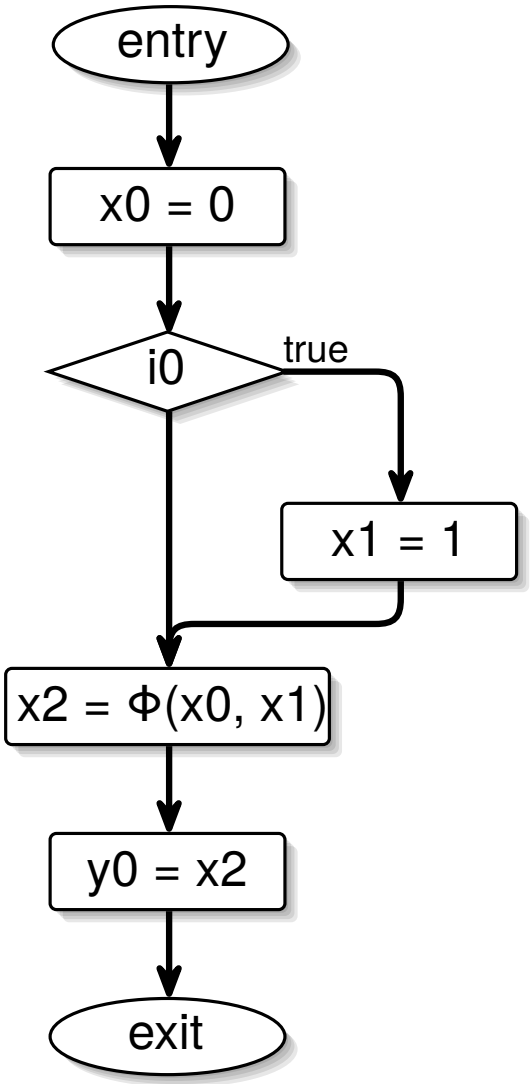
- Each variable is assigned only once (statically)
- Only one reaching definition for each read (def-use-chains)
- Minimal placement of ϕ-nodes using a dominator analysis

Static Single Assignment Form



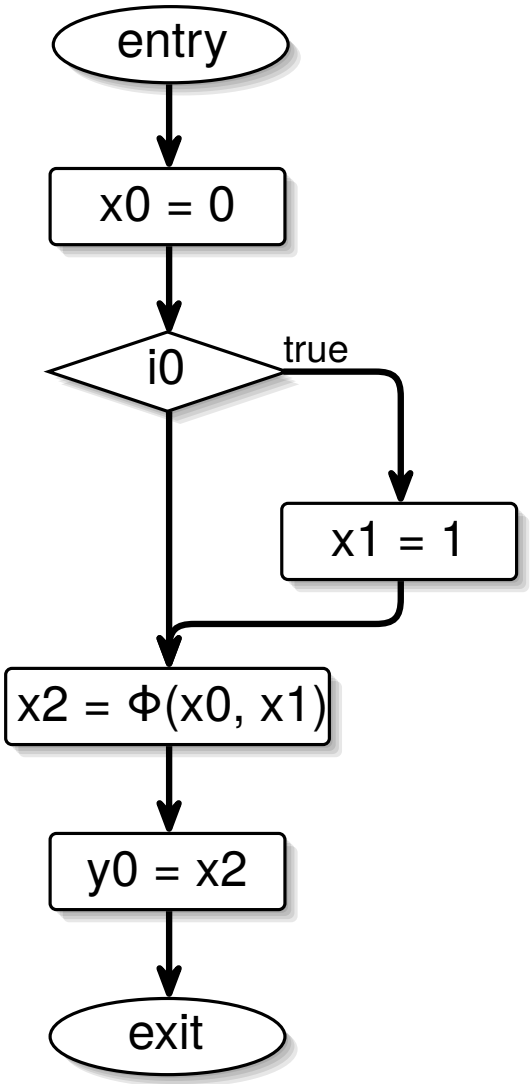
- Each variable is assigned only once (statically)
- Only one reaching definition for each read (def-use-chains)
- Minimal placement of ϕ-nodes using a dominator analysis

Static Single Assignment Form



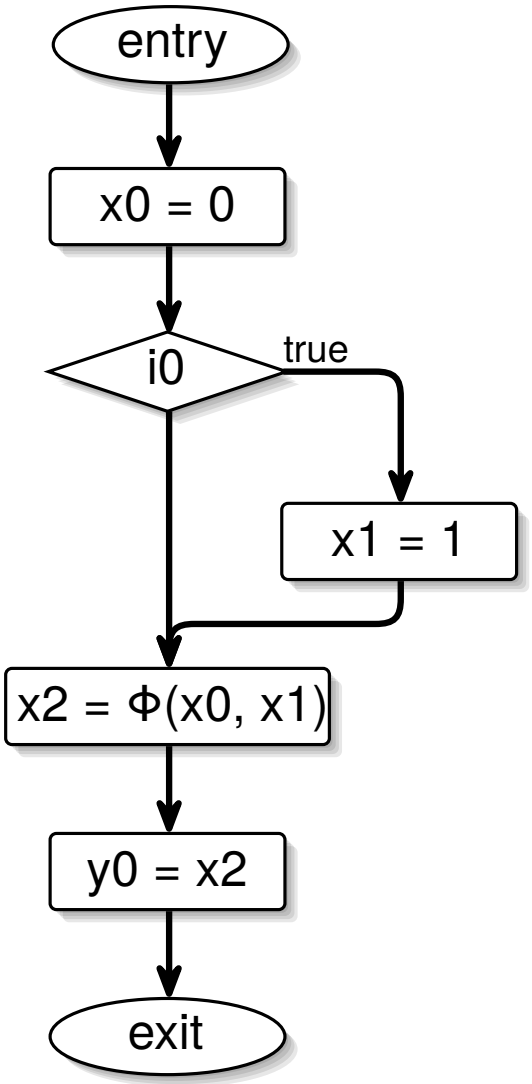
- Each variable is assigned only once (statically)
- Only one reaching definition for each read (def-use-chains)
- Minimal placement of ϕ-nodes using a dominator analysis
- Intermediate representation based on a CFG

Static Single Assignment Form



- Each variable is assigned only once (statically)
- Only one reaching definition for each read (def-use-chains)
- Minimal placement of ϕ-nodes using a dominator analysis
- Intermediate representation based on a CFG

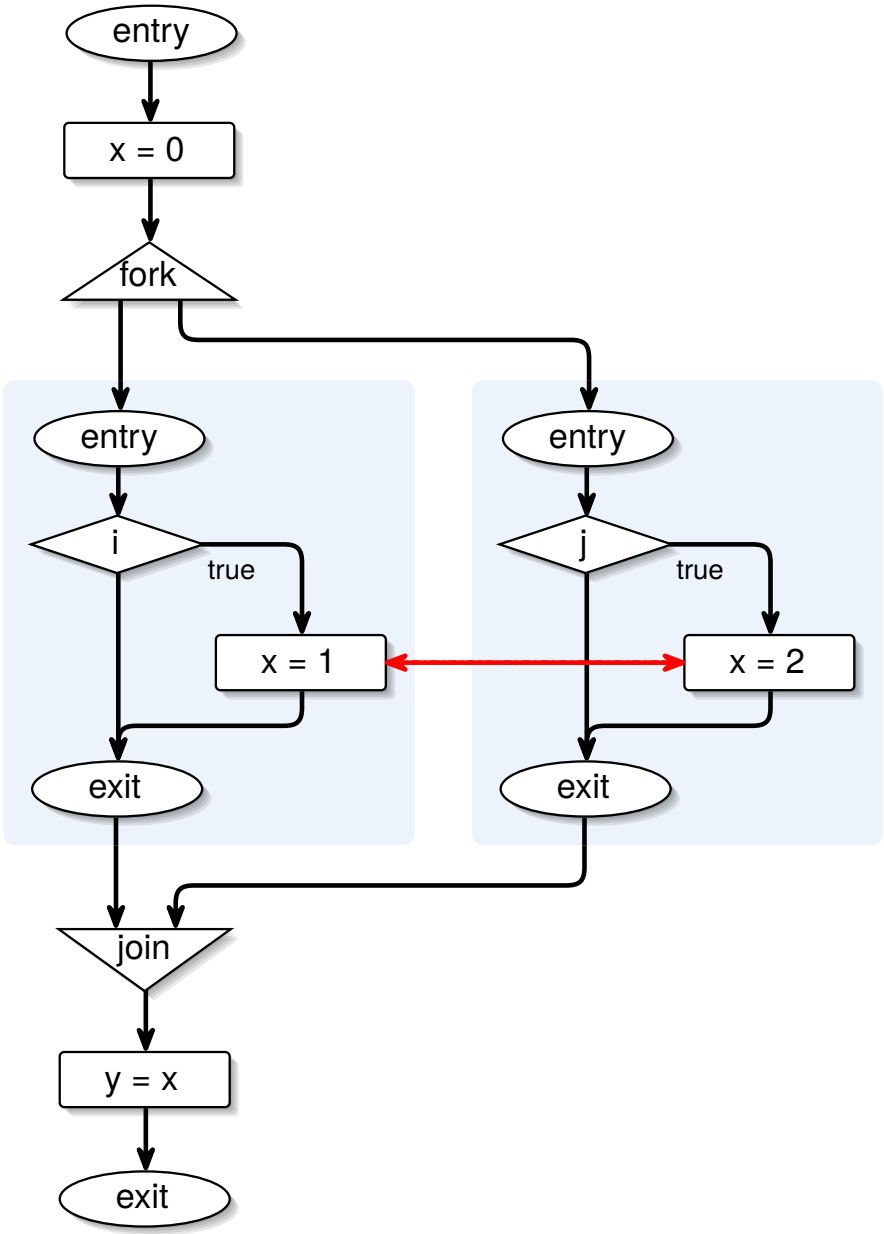
Static Single Assignment Form



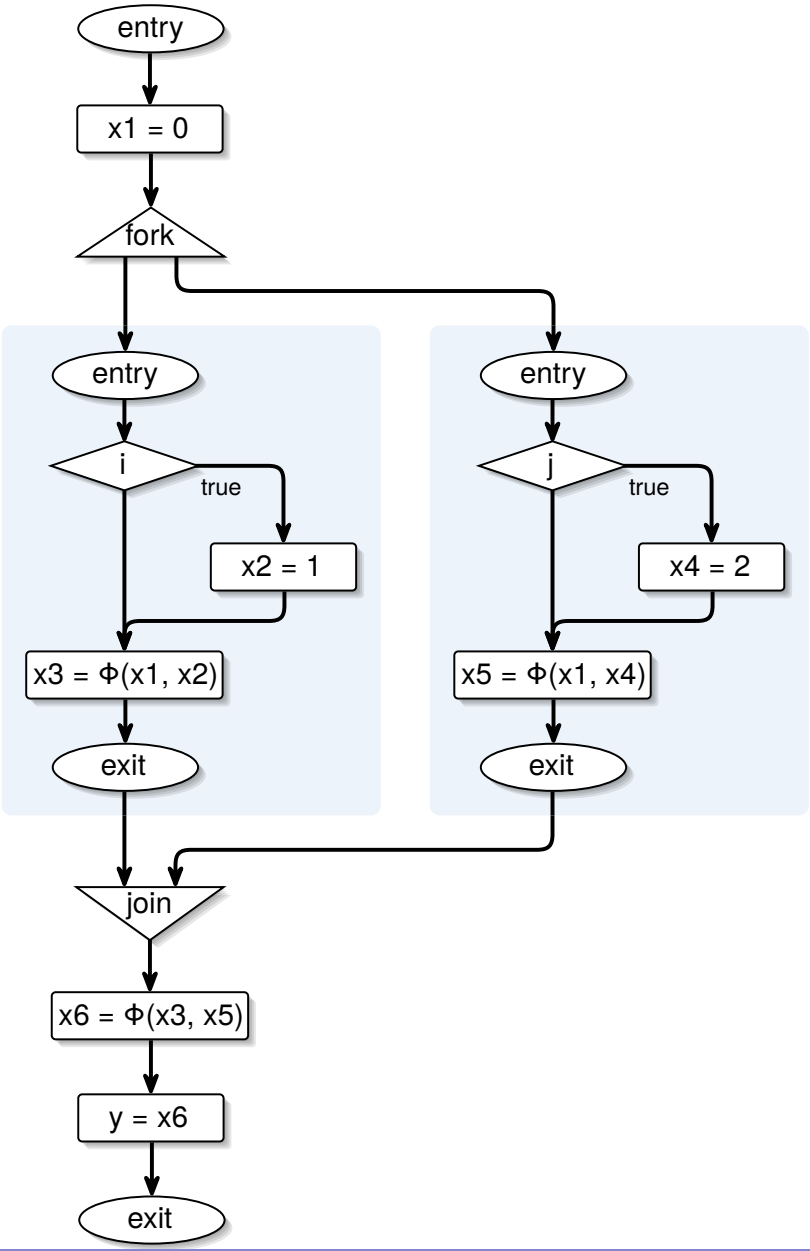
- Each variable is assigned only once (statically)
- Only one reaching definition for each read (def-use-chains)
- Minimal placement of ϕ-nodes using a dominator analysis
- Intermediate representation based on a CFG

What about SCGs with concurrency?

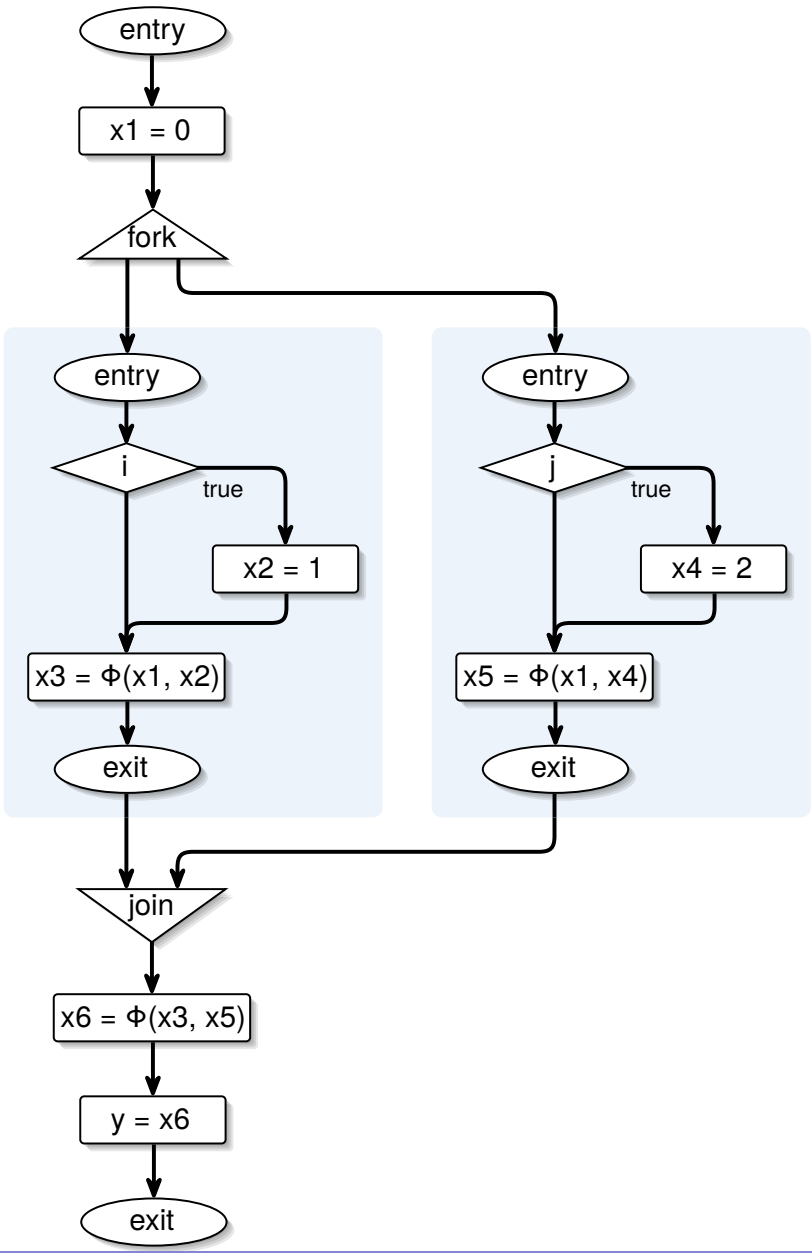
Static Single Assignment Form with Concurrency



Static Single Assignment Form with Concurrency



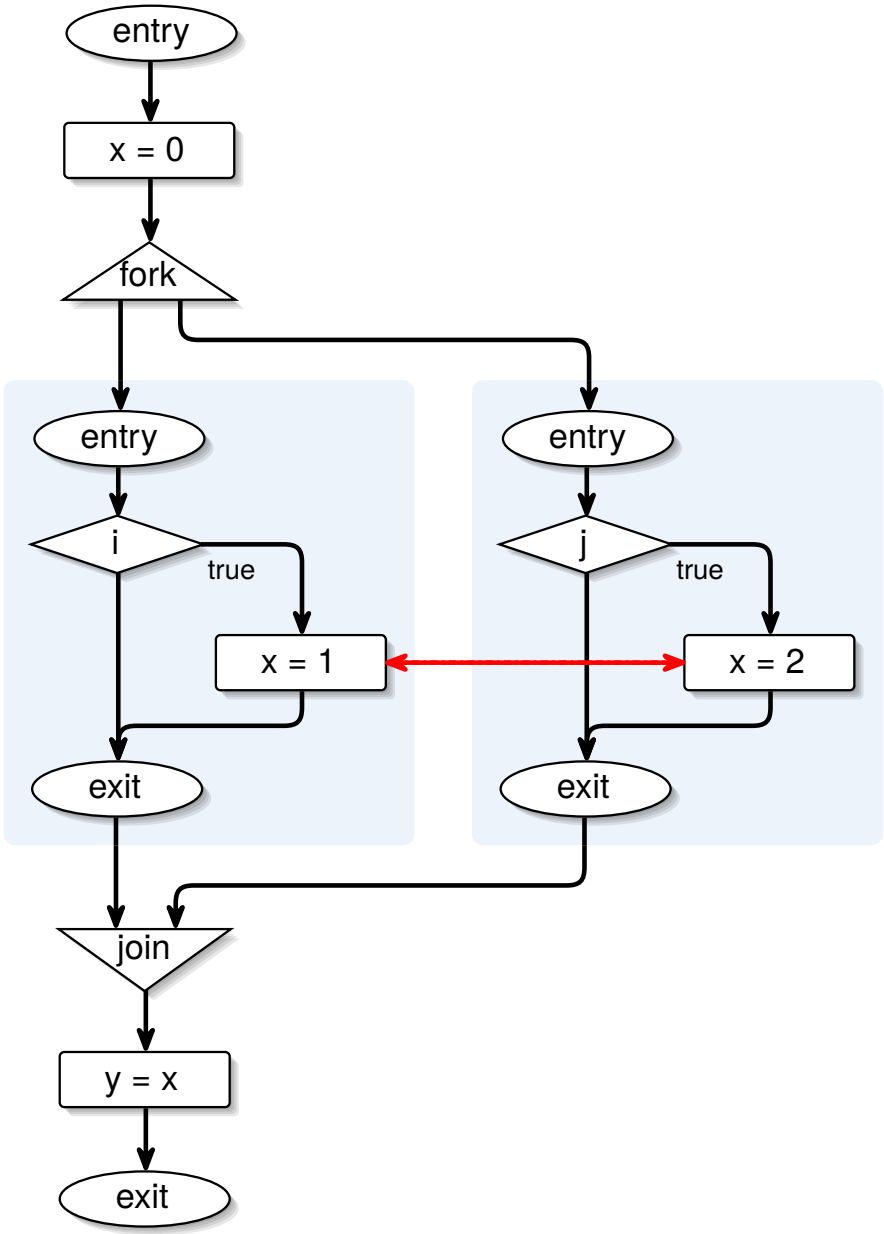
Static Single Assignment Form with Concurrency



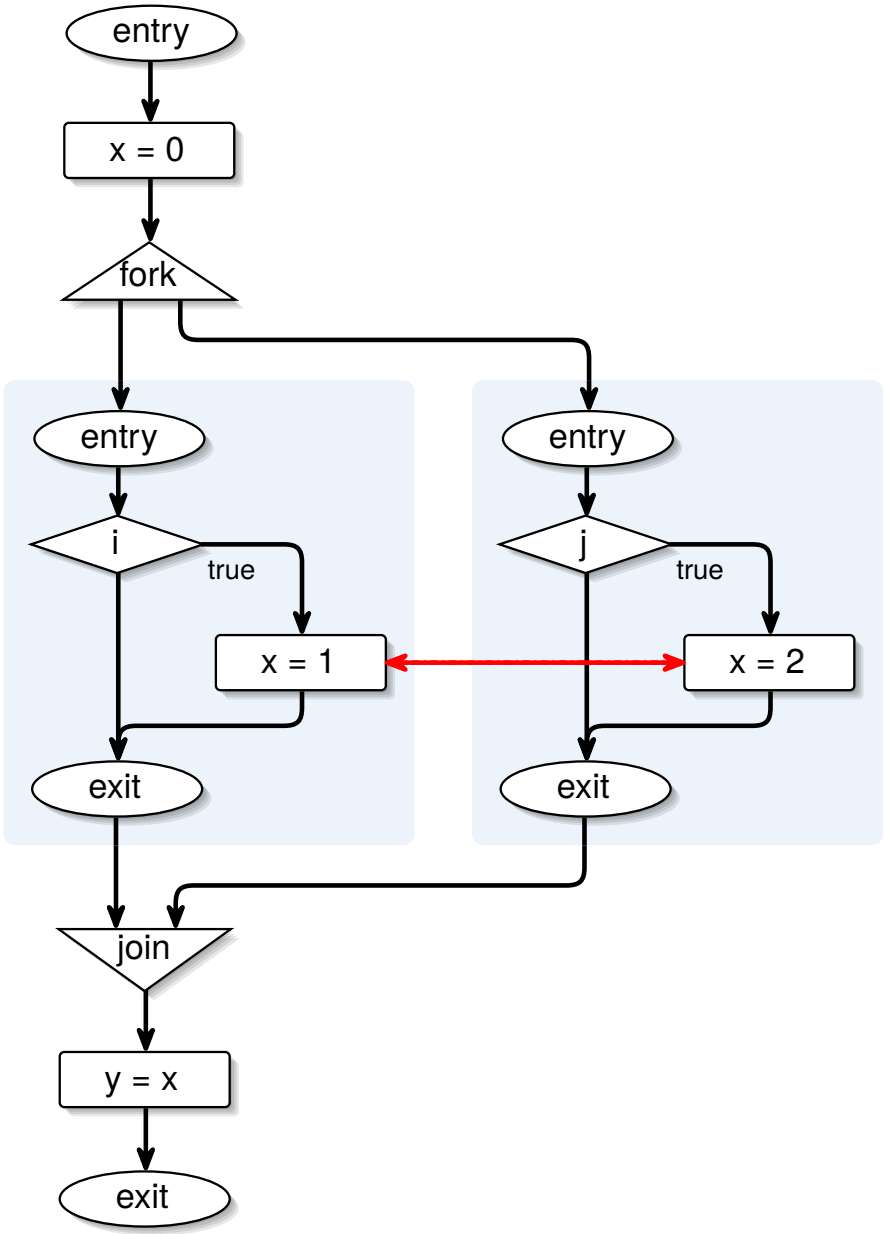
Problem

ϕ-functions cannot handle concurrency

Static Single Assignment Form with Concurrency

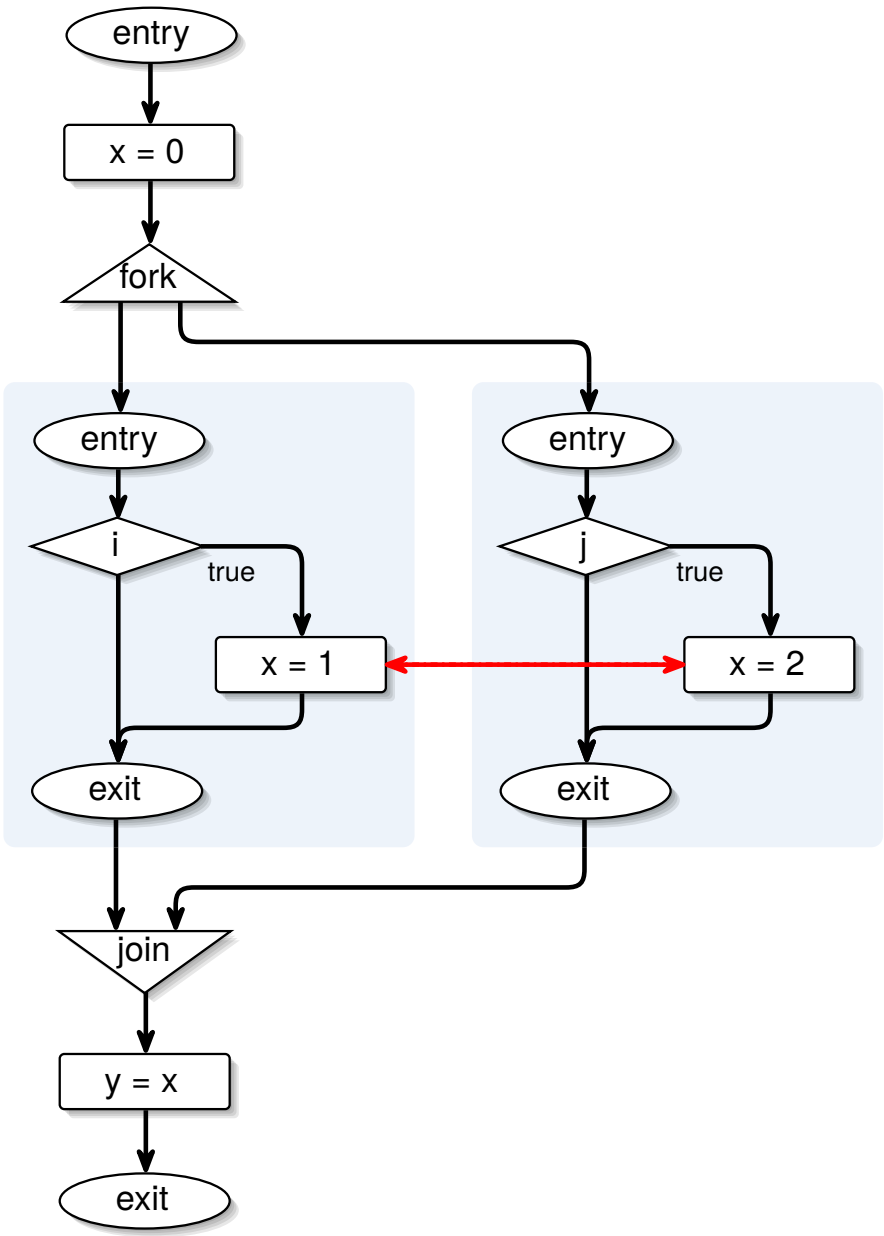


Static Single Assignment Form with Concurrency



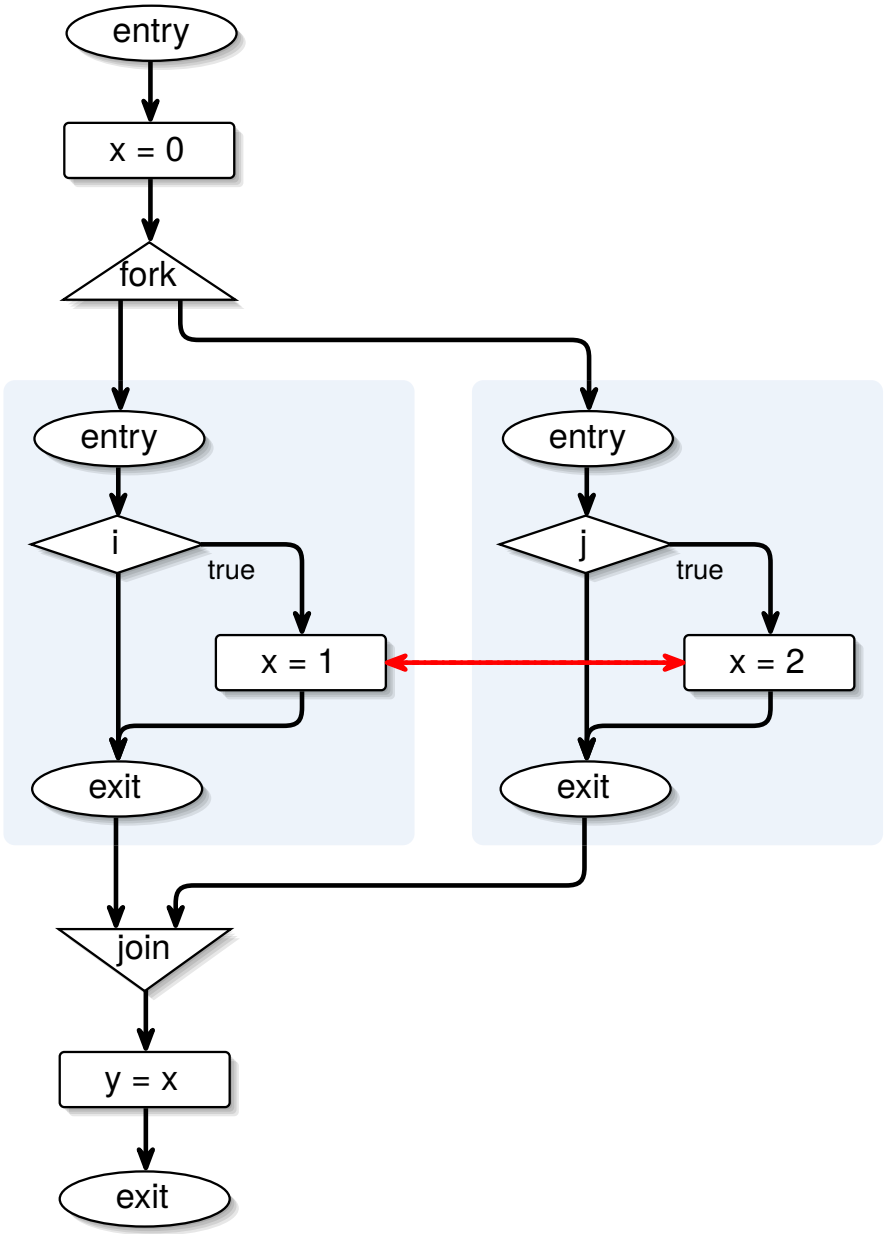
i	j	y
false	false	
false	true	
true	false	
true	true	

Static Single Assignment Form with Concurrency



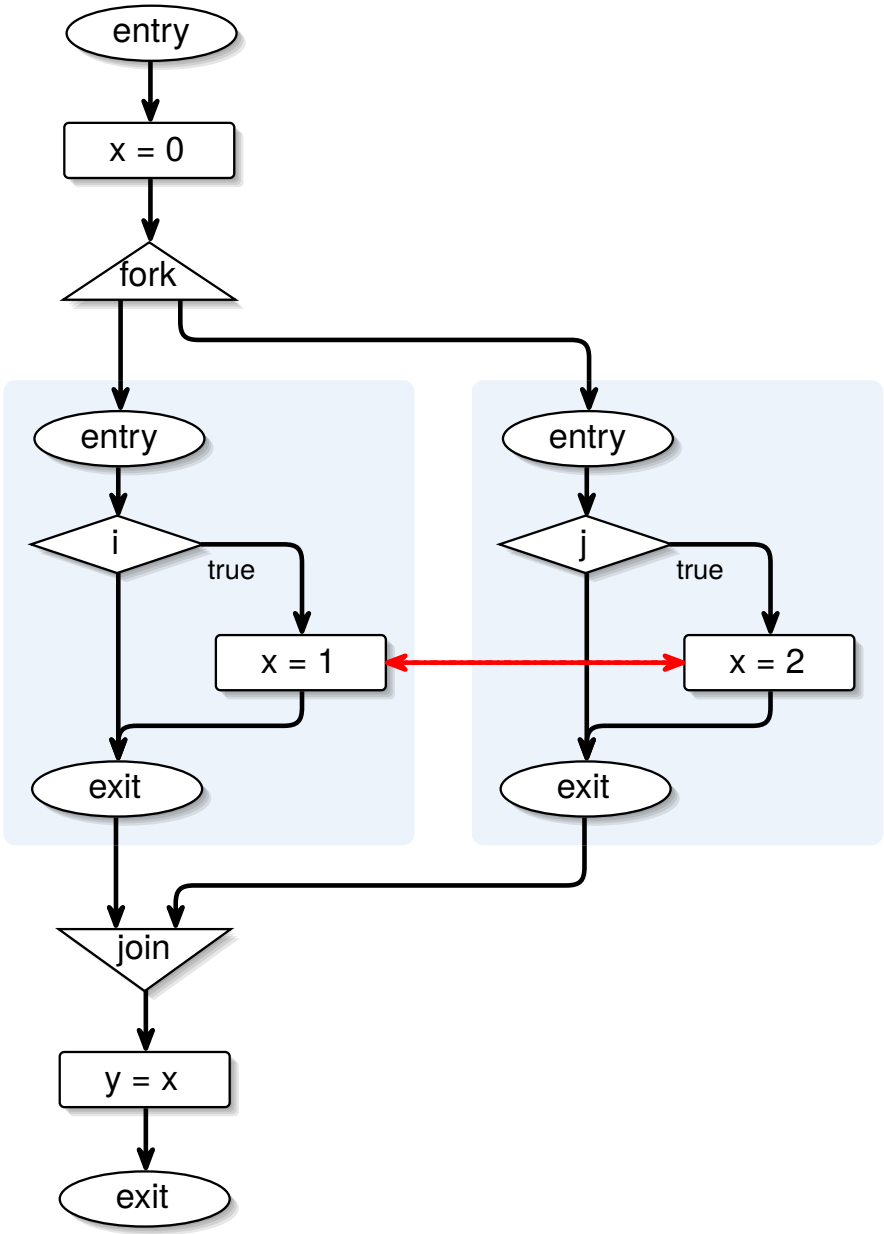
i	j	y
false	false	0
false	true	
true	false	
true	true	

Static Single Assignment Form with Concurrency



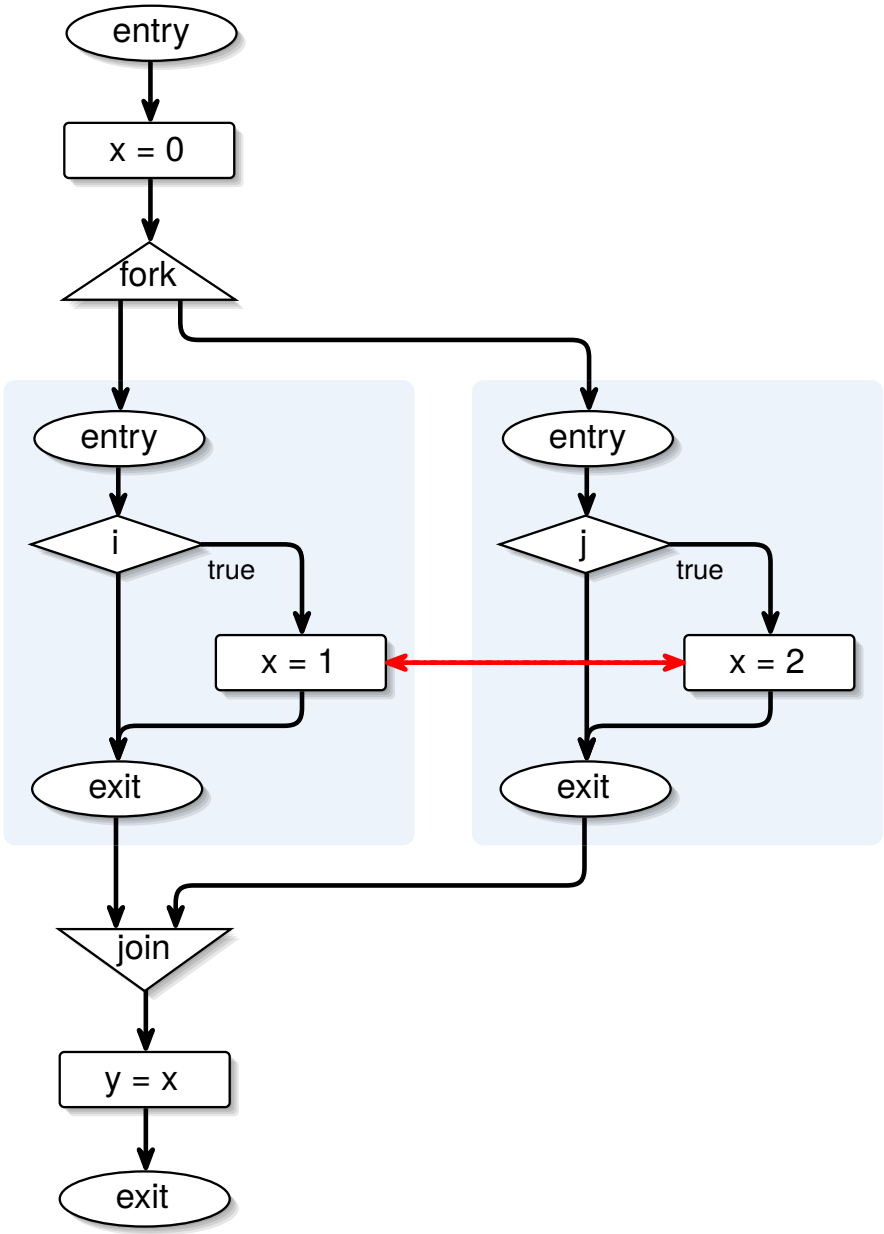
i	j	y
false	false	0
false	true	2
true	false	
true	true	

Static Single Assignment Form with Concurrency



i	j	y
false	false	0
false	true	2
true	false	1
true	true	

Static Single Assignment Form with Concurrency



i	j	y
false	false	0
false	true	2
true	false	1
true	true	**reject**

Static Single Assignment Form for SC Programs

SC-specific merge functions:

- Sequential override

Static Single Assignment Form for SC Programs

SC-specific merge functions:

- Sequential override
- Encode concurrency

Static Single Assignment Form for SC Programs

SC-specific merge functions:

- Sequential override
- Encode concurrency
- Detect confluent writes

Static Single Assignment Form for SC Programs

SC-specific merge functions:

- Sequential override
- Encode concurrency
- Detect confluent writes
- Reject conflicting writes

Static Single Assignment Form for SC Programs

SC-specific merge functions:

- Sequential override
- Encode concurrency
- Detect confluent writes
- Reject conflicting writes

Variable Representation: $\langle \mathrm{x}^p, \mathrm{x} \rangle$

Inspired by valued signals.

x^p: Presence signal

x: Actual variable value

Static Single Assignment Form for SC Programs

SC-specific merge functions:

- Sequential override
- Encode concurrency
- Detect confluent writes
- Reject conflicting writes

Variable Representation: $\langle \mathtt{x}^p, \mathtt{x} \rangle$

Inspired by valued signals.

$\mathtt{x}^p$: Presence signal

$\mathtt{x}$: Actual variable value

```
1 ‖ seq(⟨x_i^p, x_i⟩, ⟨x_j^p, x_j⟩) :=
2 ‖   present x_j^p then
3 ‖     return ⟨x_j^p, x_j⟩
4 ‖   else
5 ‖     present x_i^p then
6 ‖       return ⟨x_i^p, x_i⟩
7 ‖     else
8 ‖       return ⟨absent, nil⟩
```

Static Single Assignment Form for SC Programs

SC-specific merge functions:

- Sequential override
- Encode concurrency
- Detect confluent writes
- Reject conflicting writes

Variable Representation: $\langle \mathrm{x}^p, \mathrm{x} \rangle$

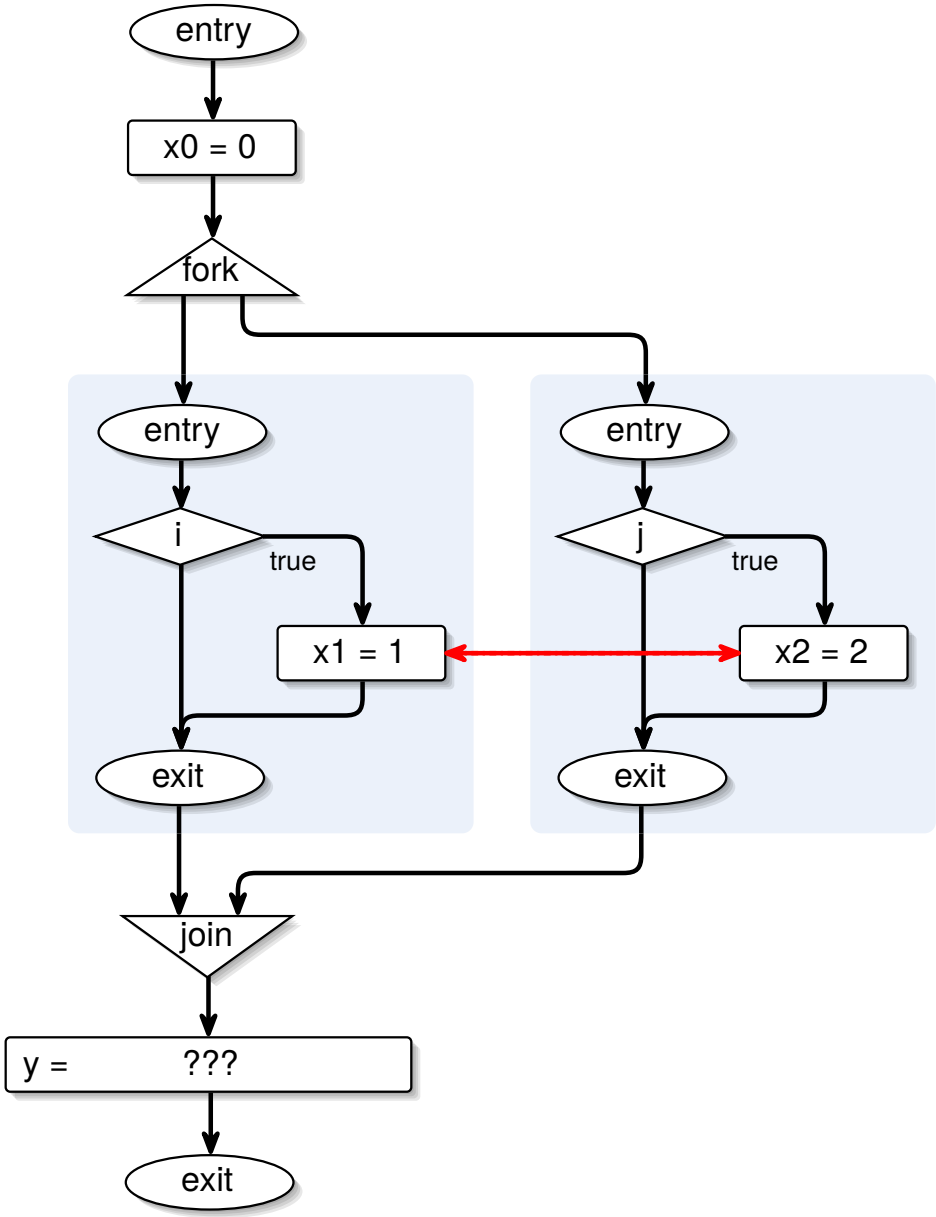
Inspired by valued signals.

x^p: Presence signal

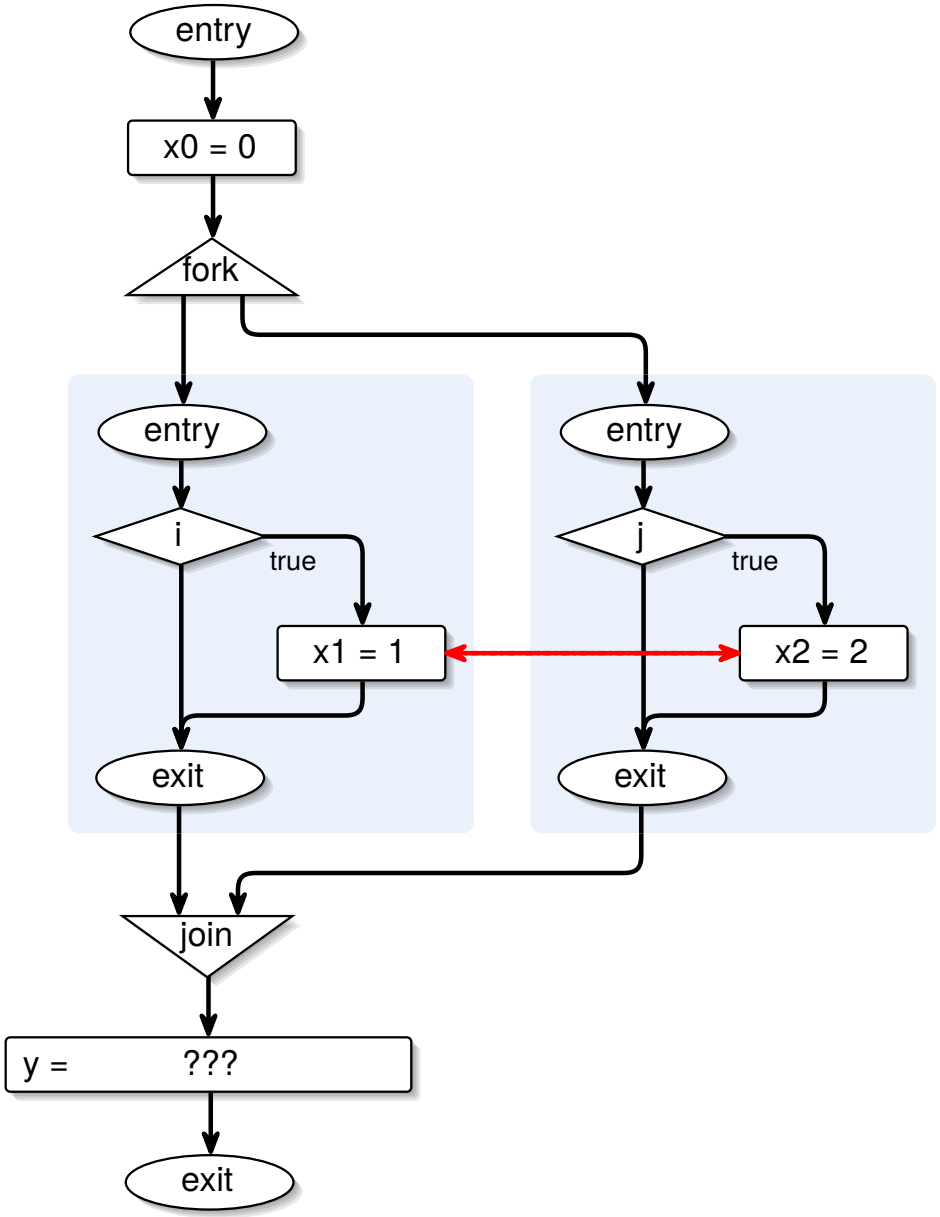
x: Actual variable value

```
1  conc(⟨x_i^p, x_i⟩, ⟨x_j^p, x_j⟩) :=
2    present x_i^p then
3      present x_j^p then
4        if x_i == x_j then
5          return ⟨x_i^p, x_i⟩
6        else
7          reject
8      else
9        return ⟨x_i^p, x_i⟩
10   else
11     present x_j^p then
12       return ⟨x_j^p, x_j⟩
13     else
14       return ⟨absent, nil⟩
```

Static Single Assignment Form for SCGs

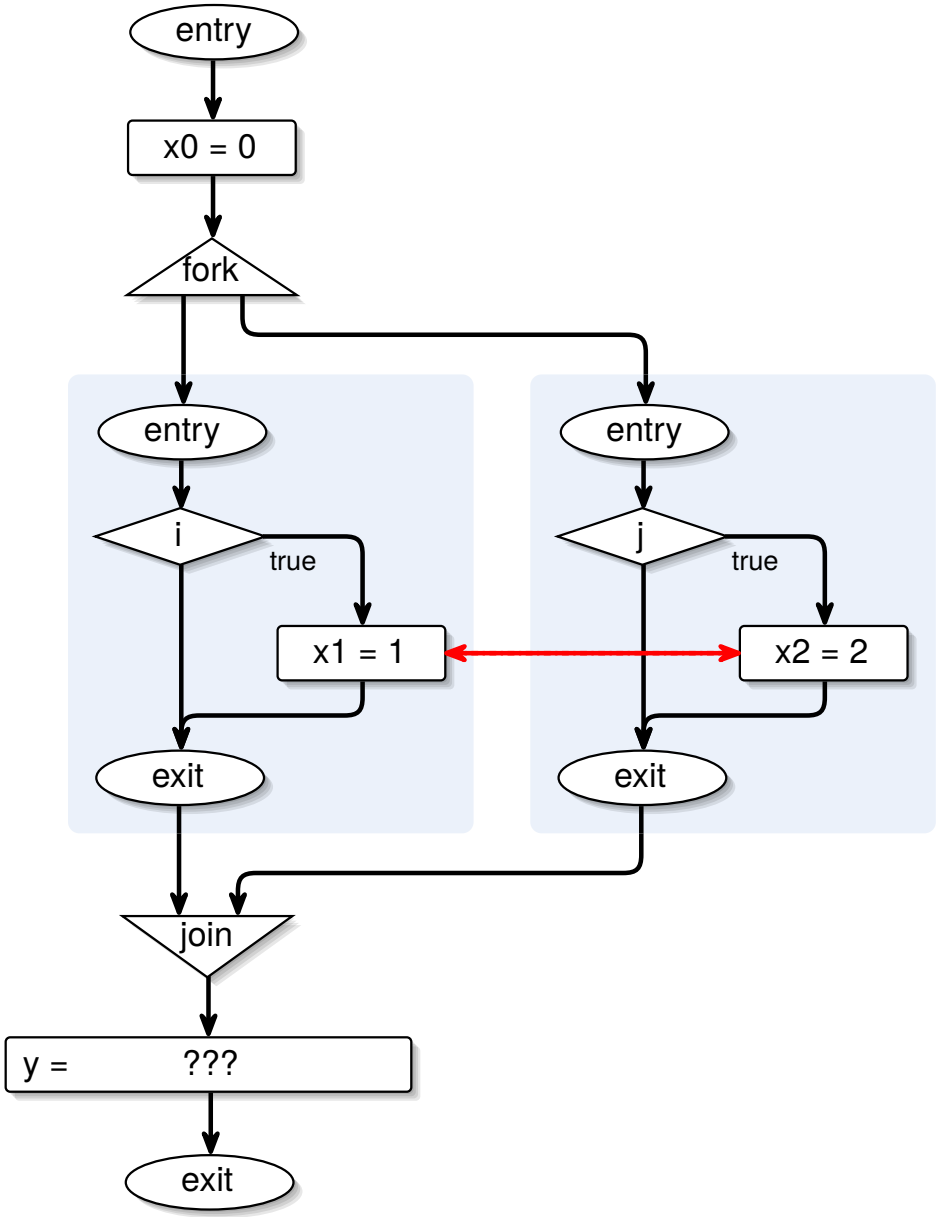


Static Single Assignment Form for SCGs



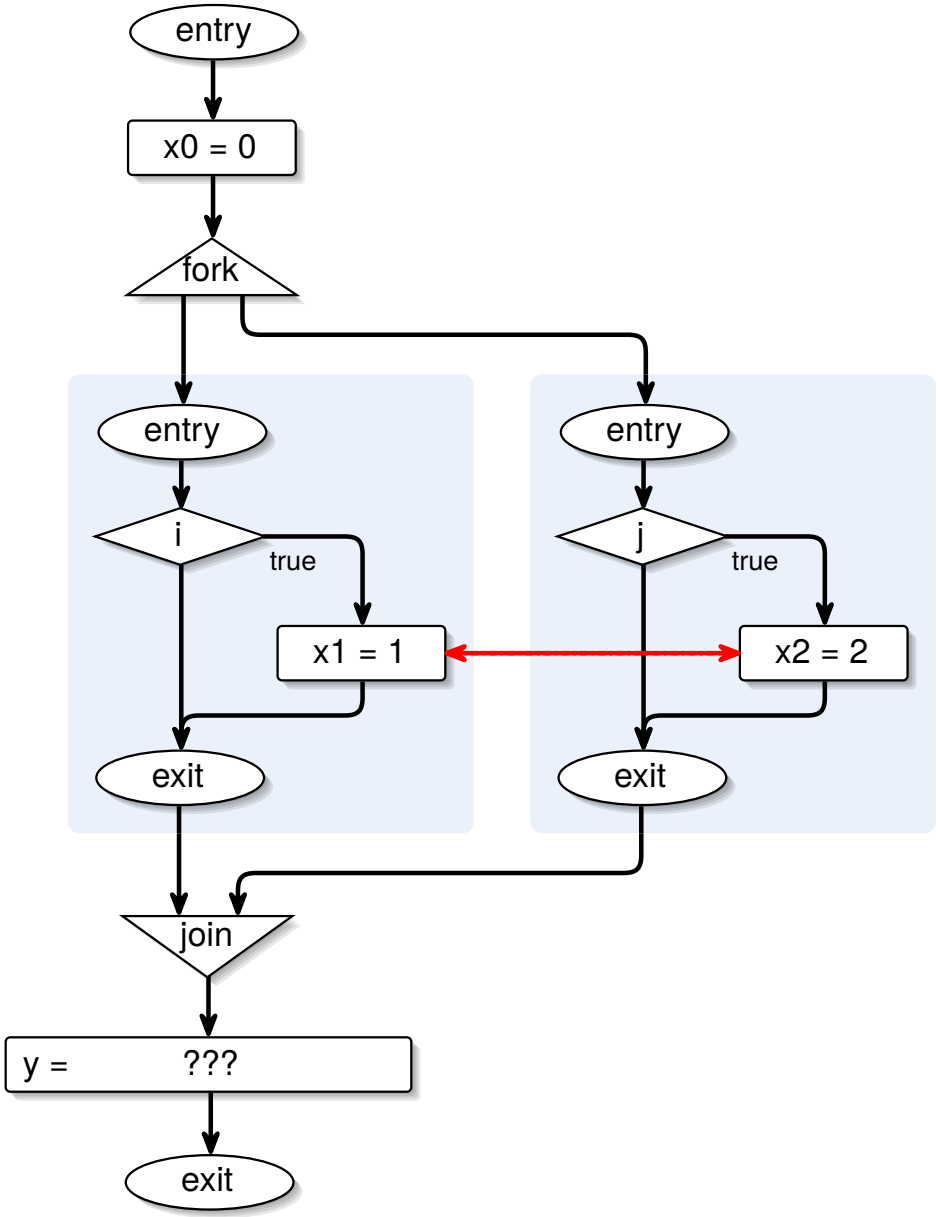
$\texttt{seq}(x_0,$

Static Single Assignment Form for SCGs



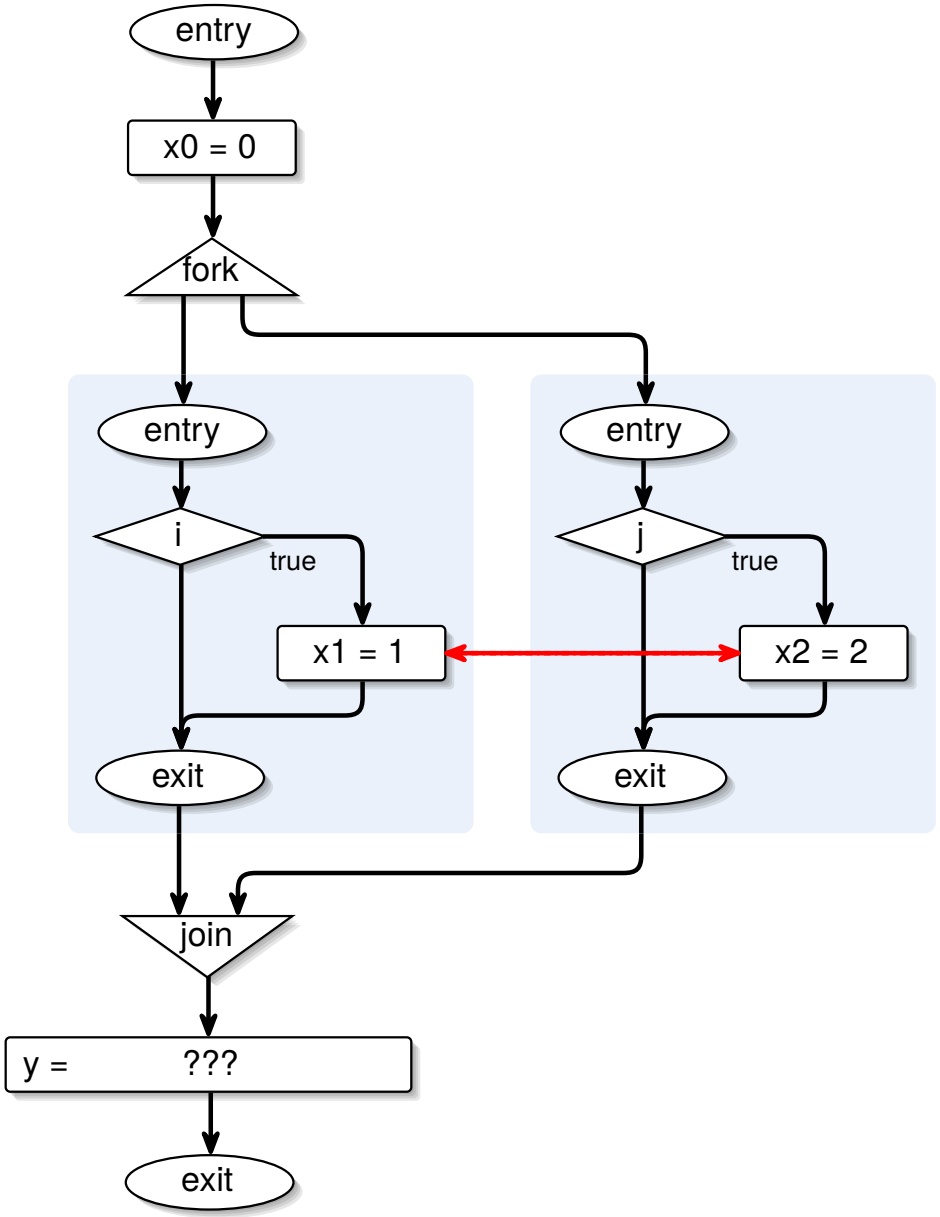
$\texttt{seq}(x_0, \texttt{conc}($

Static Single Assignment Form for SCGs



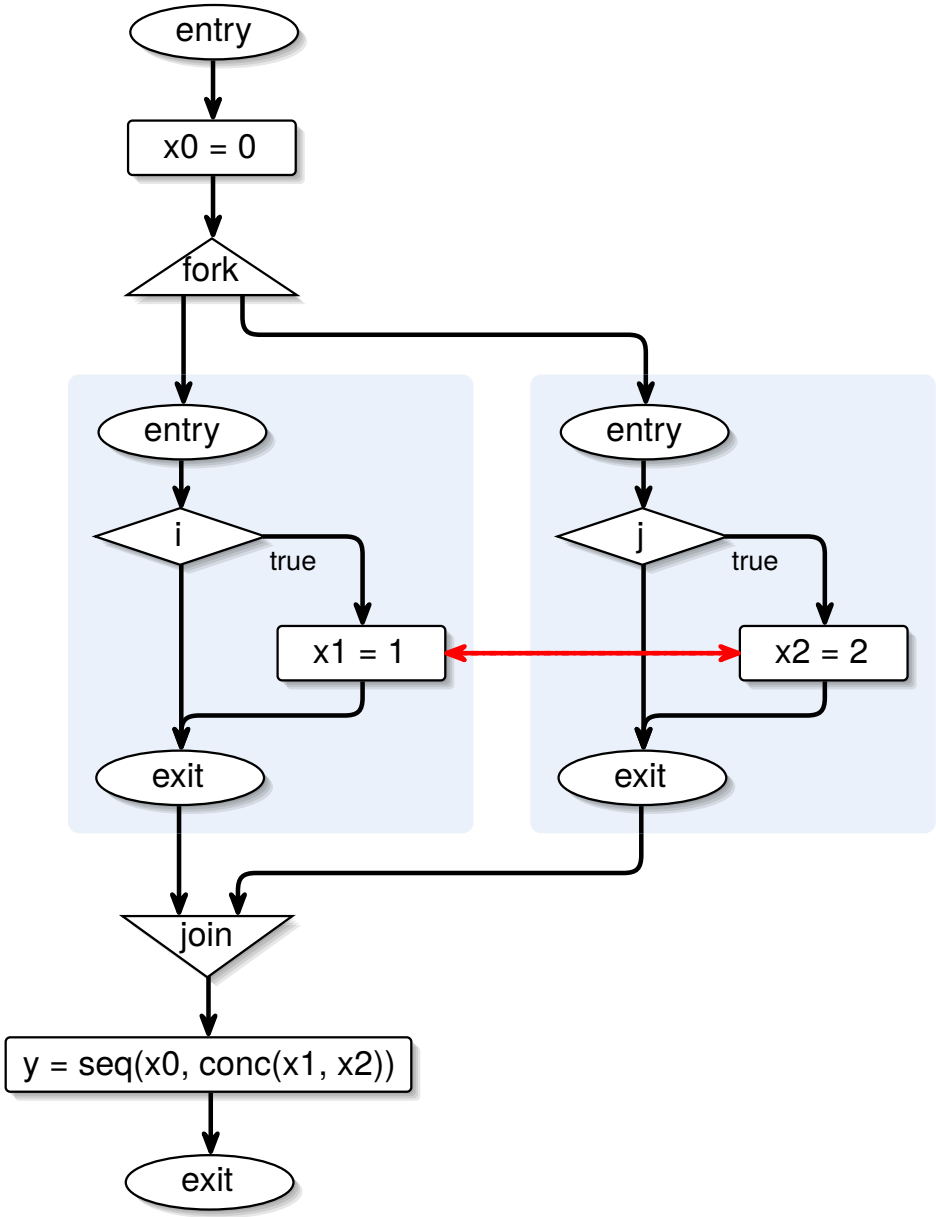
$\texttt{seq}(x_0, \texttt{conc}(x_1,$

Static Single Assignment Form for SCGs



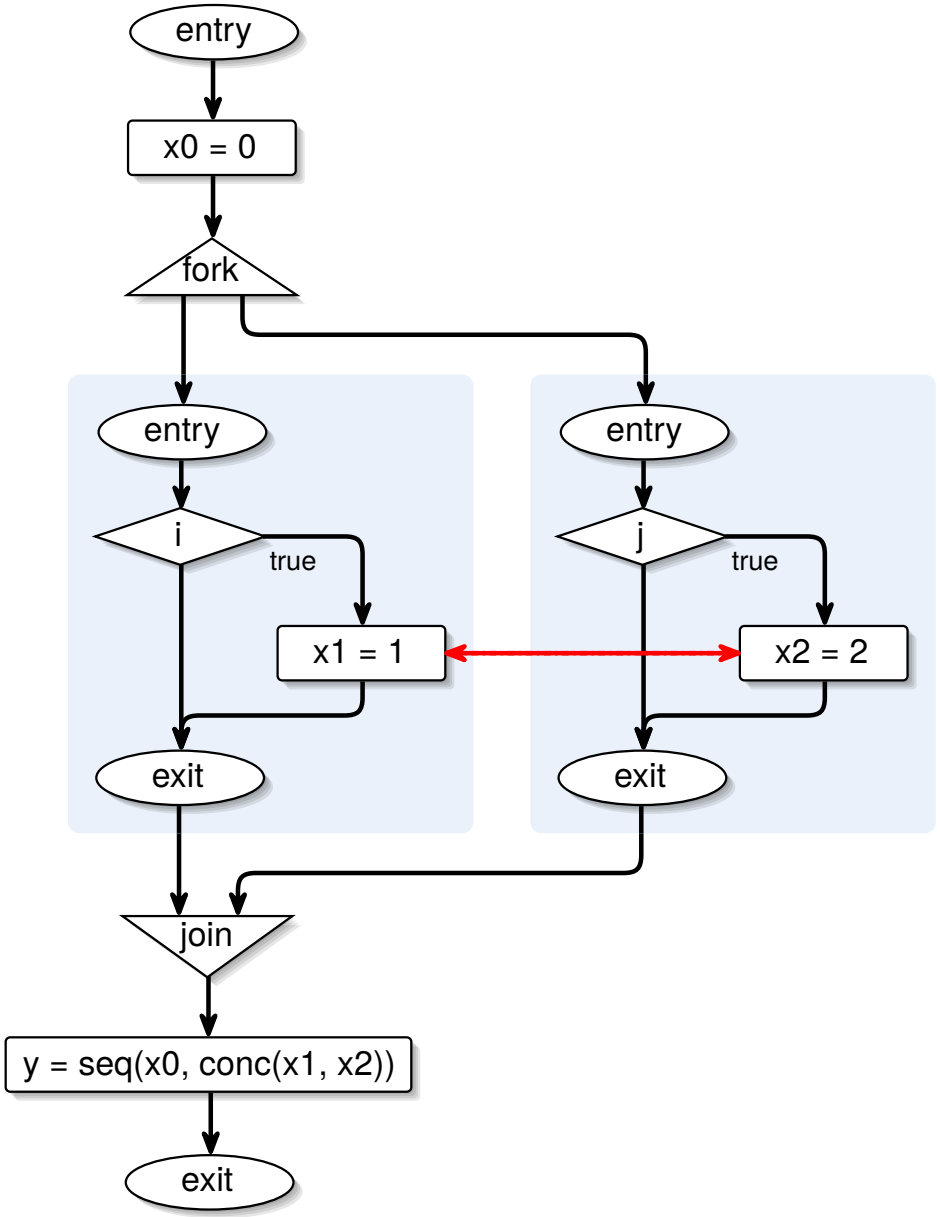
$\texttt{seq}(x_0, \texttt{conc}(x_1, x_2$

Static Single Assignment Form for SCGs



$\texttt{seq}(x_0, \texttt{conc}(x_1, x_2))$

Static Single Assignment Form for SCGs



i	j	y
false	false	0
false	true	2
true	false	1
true	true	**reject**

SSA Form: Further Aspects

- Delays

SSA Form: Further Aspects

- Delays
  - Merge functions use a variable with signals and implicit reset

SSA Form: Further Aspects

- Delays
  - Merge functions use a variable with signals and implicit reset
- Loops

SSA Form: Further Aspects

- Delays
  - Merge functions use a variable with signals and implicit reset
- Loops
  - Merge expressions require explicit sequential ordering

SSA Form: Further Aspects

- Delays
  - Merge functions use a variable with signals and implicit reset
- Loops
  - Merge expressions require explicit sequential ordering
- Updates

SSA Form: Further Aspects

- Delays
  - Merge functions use a variable with signals and implicit reset
- Loops
  - Merge expressions require explicit sequential ordering
- Updates
  - iur protocol ordering

SSA Form: Further Aspects

- Delays
  - Merge functions use a variable with signals and implicit reset
- Loops
  - Merge expressions require explicit sequential ordering
- Updates
  - iur protocol ordering
  - Confluent by definition

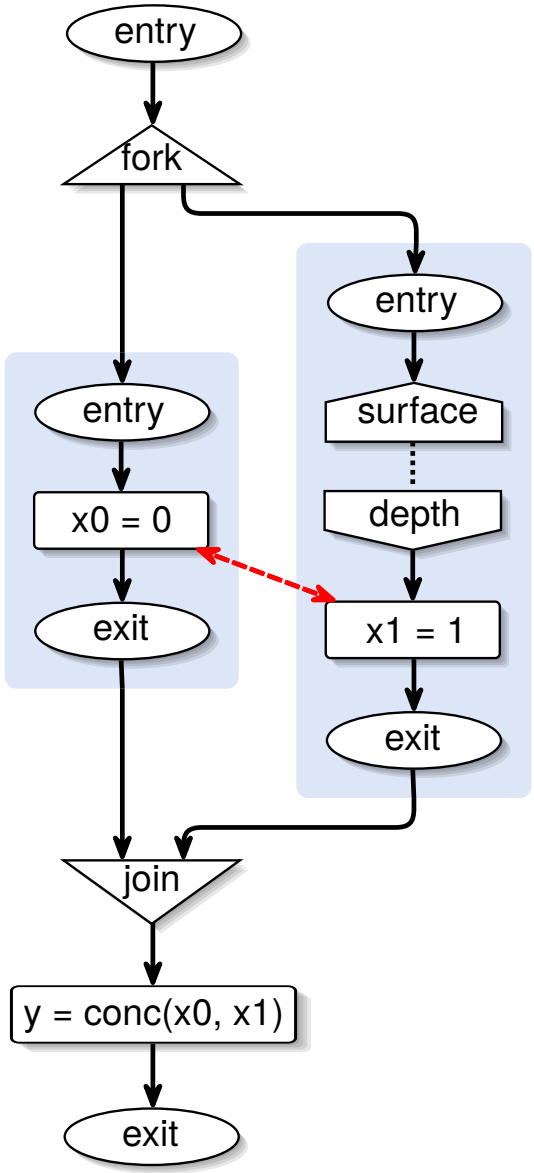
SSA Form: Further Aspects

- Delays
  - Merge functions use a variable with signals and implicit reset
- Loops
  - Merge expressions require explicit sequential ordering
- Updates
  - iur protocol ordering
  - Confluent by definition
- Interface

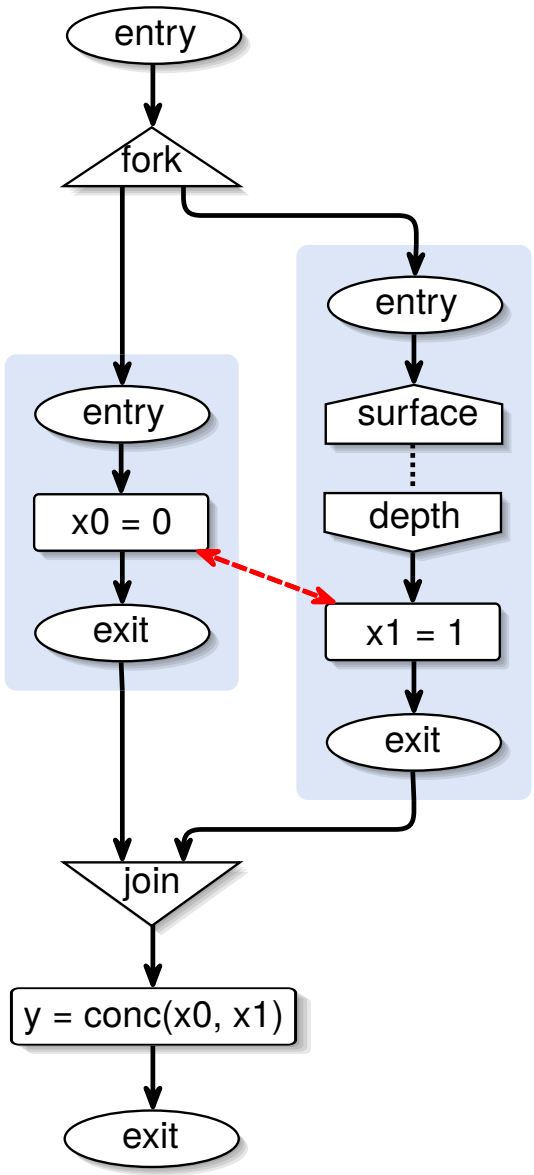
SSA Form: Further Aspects

- Delays
  - Merge functions use a variable with signals and implicit reset
- Loops
  - Merge expressions require explicit sequential ordering
- Updates
  - iur protocol ordering
  - Confluent by definition
- Interface
  - SSA renaming

SSA Form: Delays

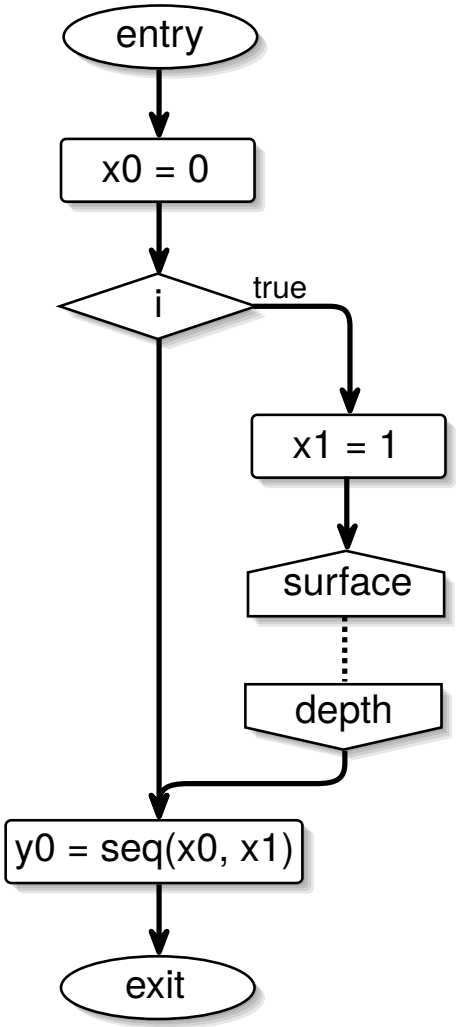


SSA Form: Delays

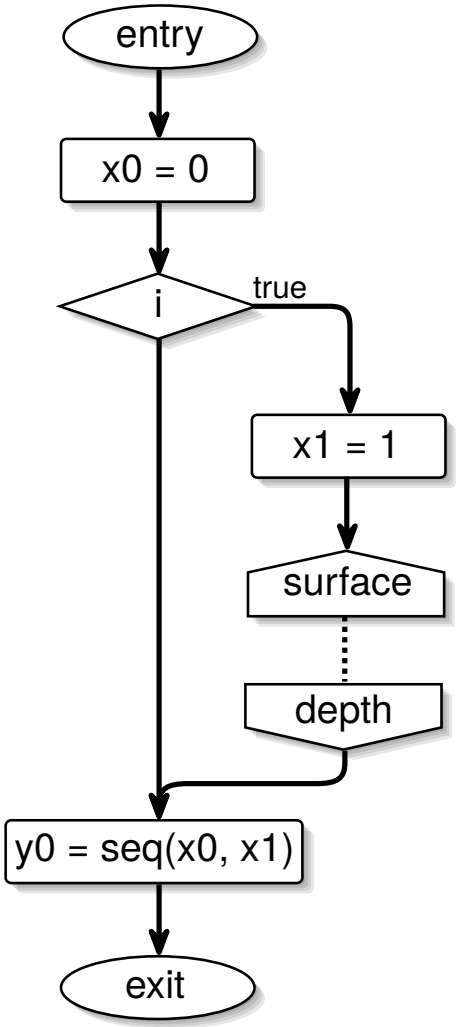



```
1  | conc(⟨x_i^p, x_i⟩, ⟨x_j^p, x_j⟩) :=
2  |   present x_i^p then
3  |     present x_j^p then
4  |       if x_i == x_j then
5  |         return ⟨x_i^p, x_i⟩
6  |       else
7  |         reject
8  |     else
9  |       return ⟨x_i^p, x_i⟩
10 |   else
11 |     present x_j^p then
12 |       return ⟨x_j^p, x_j⟩
13 |     else
14 |       return ⟨absent, nil⟩
```

SSA Form: Delays

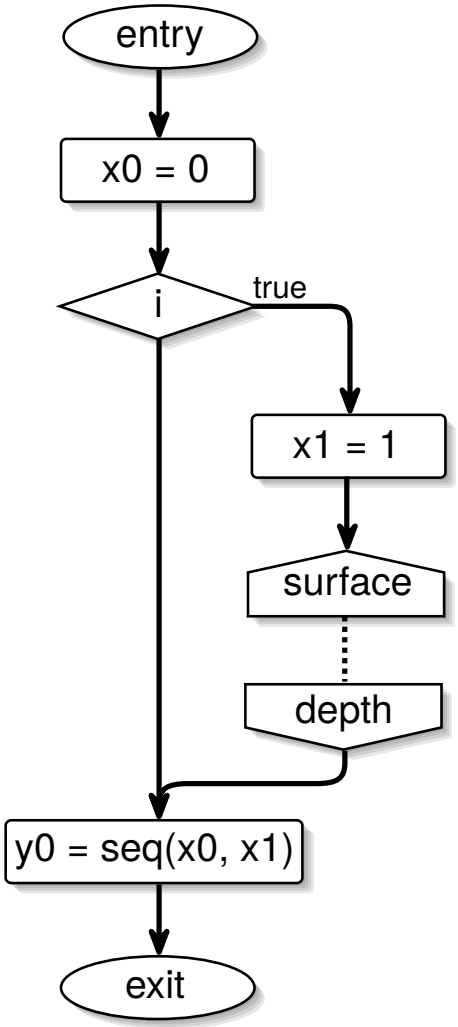


SSA Form: Delays



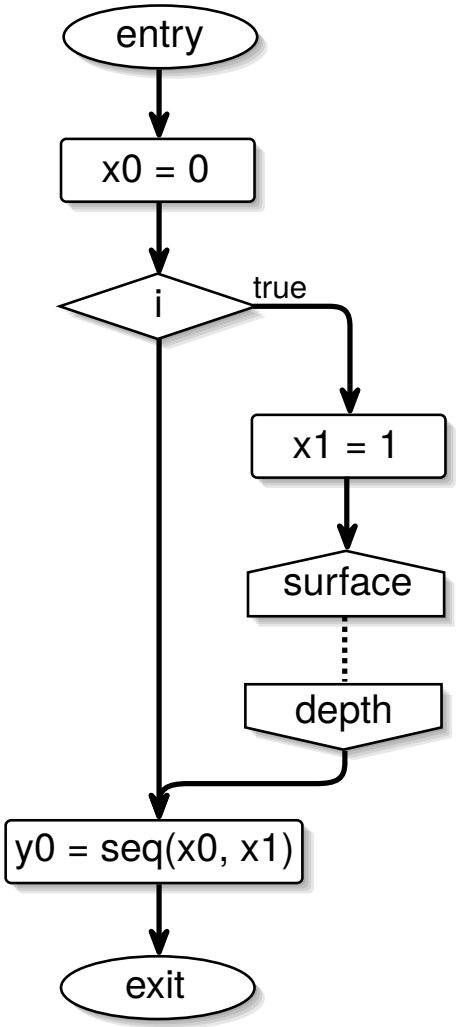

```
1 | seq(⟨x_i^p, x_i⟩, ⟨x_j^p, x_j⟩) :=
2 |   present x_j^p then
3 |     return ⟨x_j^p, x_j⟩
4 |   else
5 |     present x_i^p then
6 |       return ⟨x_i^p, x_i⟩
7 |     else
8 |       return ⟨absent, nil⟩
```

SSA Form: Delays



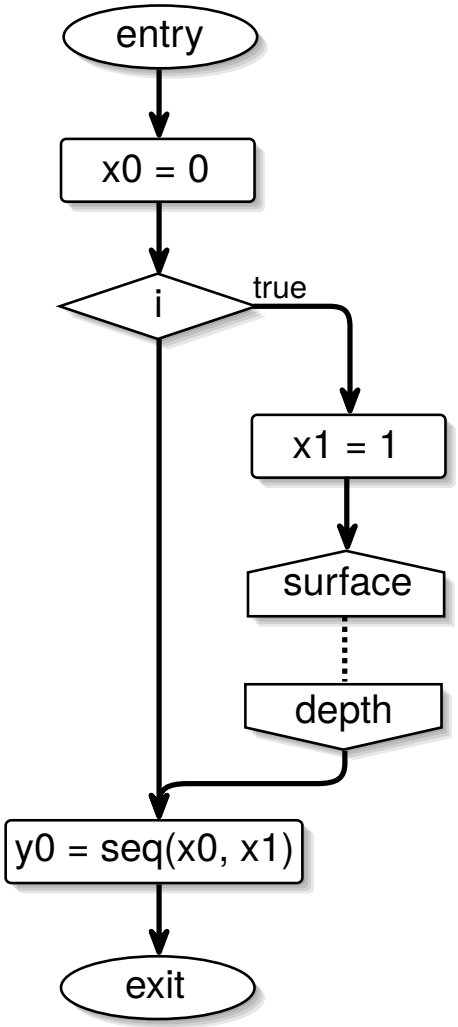
- Presence signals are reset to absent

SSA Form: Delays



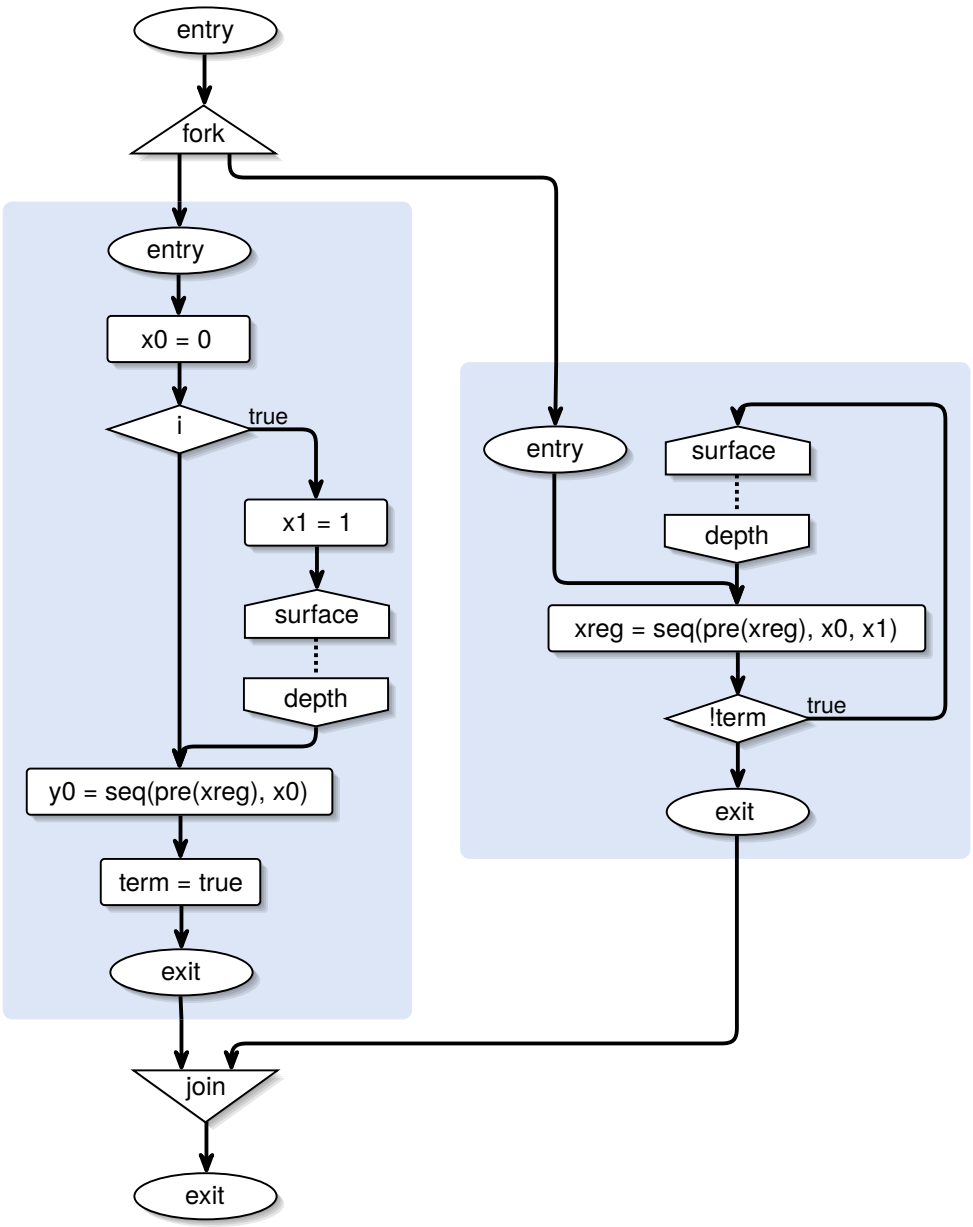
- Presence signals are reset to absent
- Runtime concurrent conflicts can be detected

SSA Form: Delays



- Presence signals are reset to absent
- Runtime concurrent conflicts can be detected
- Merge function cannot resolve value without write in the same tick

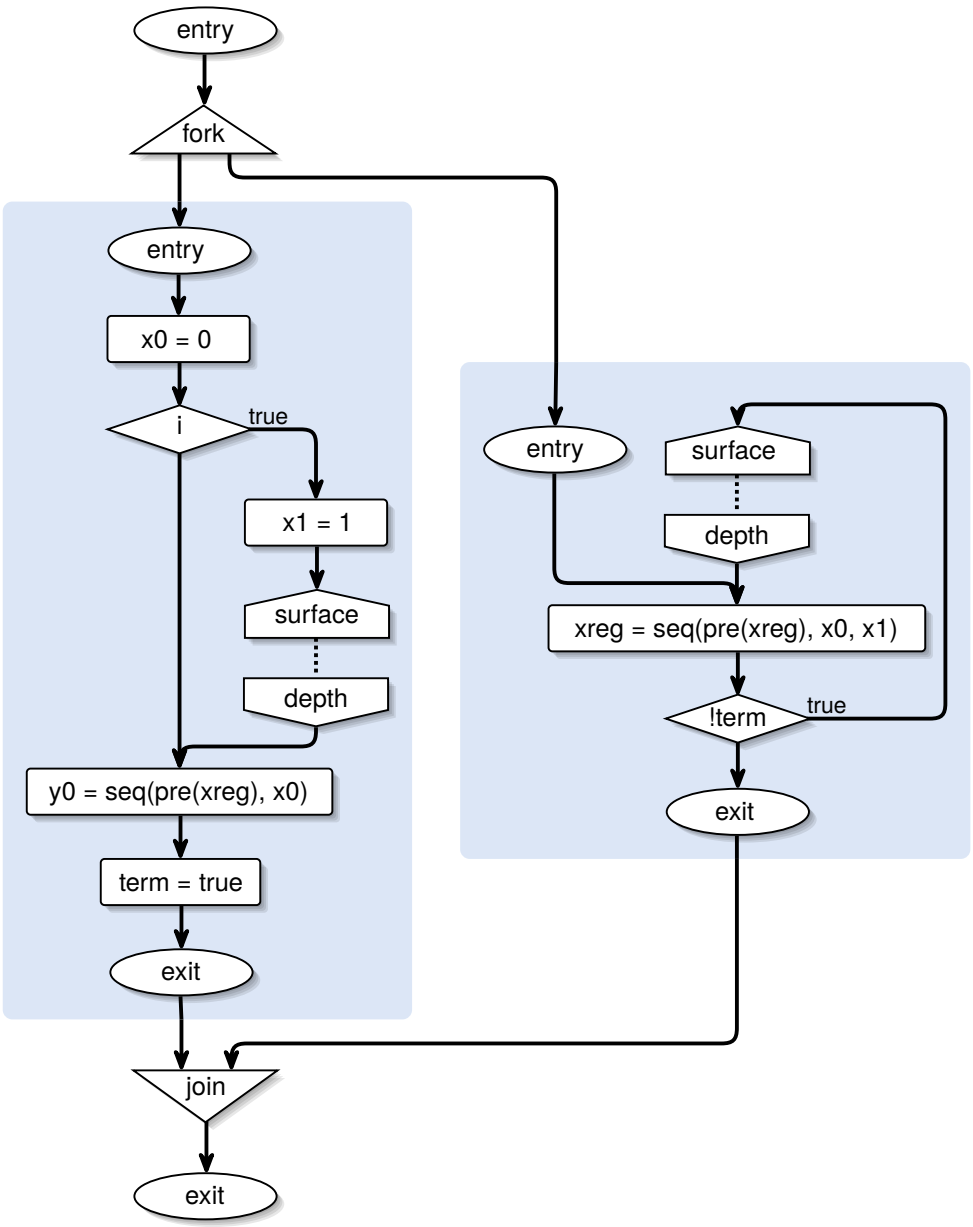
SSA Form: Delays Solved



Solution:

- Resolve and save variable values in each tick

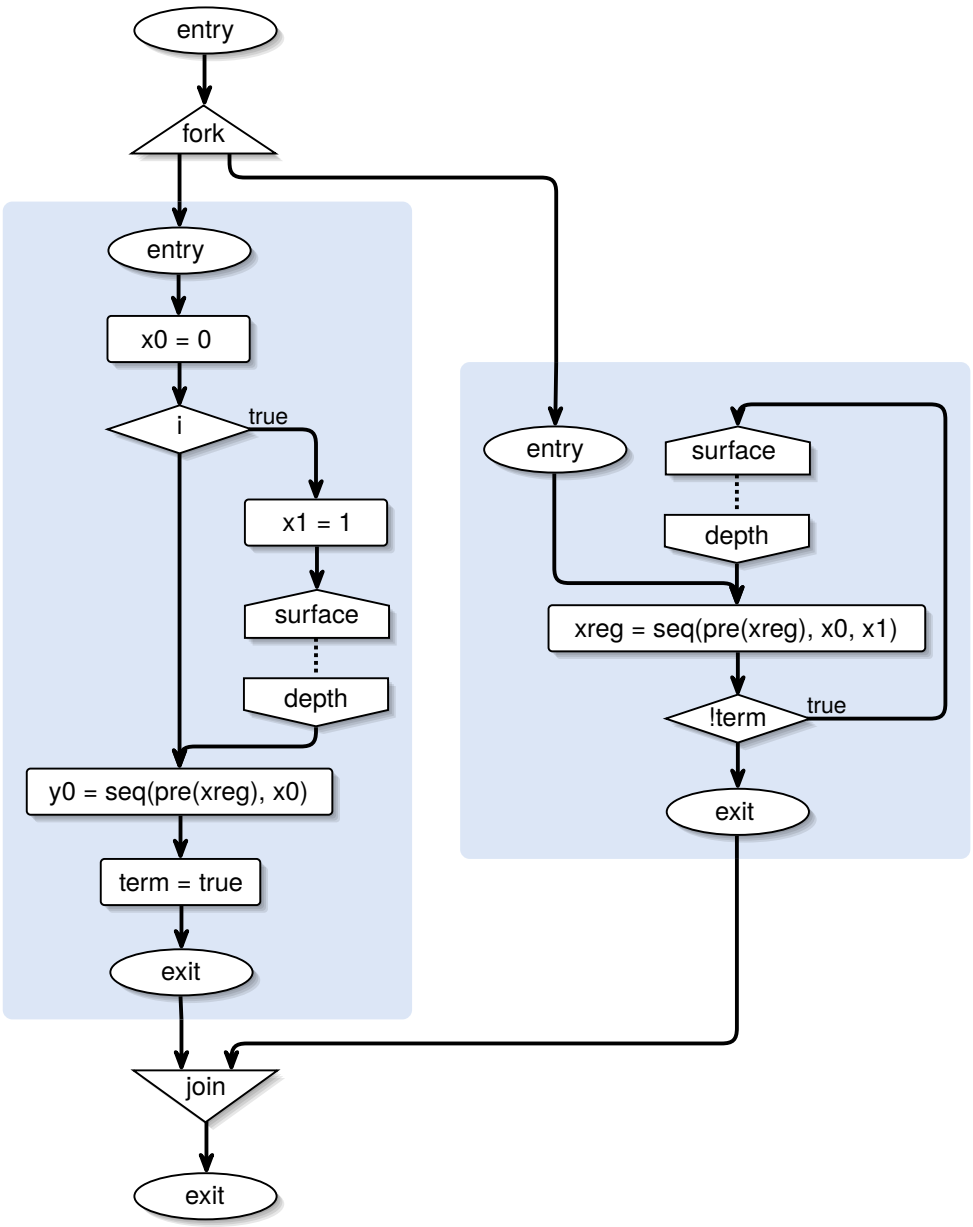
SSA Form: Delays Solved



Solution:

- Resolve and save variable values in each tick
- Store values in register variables

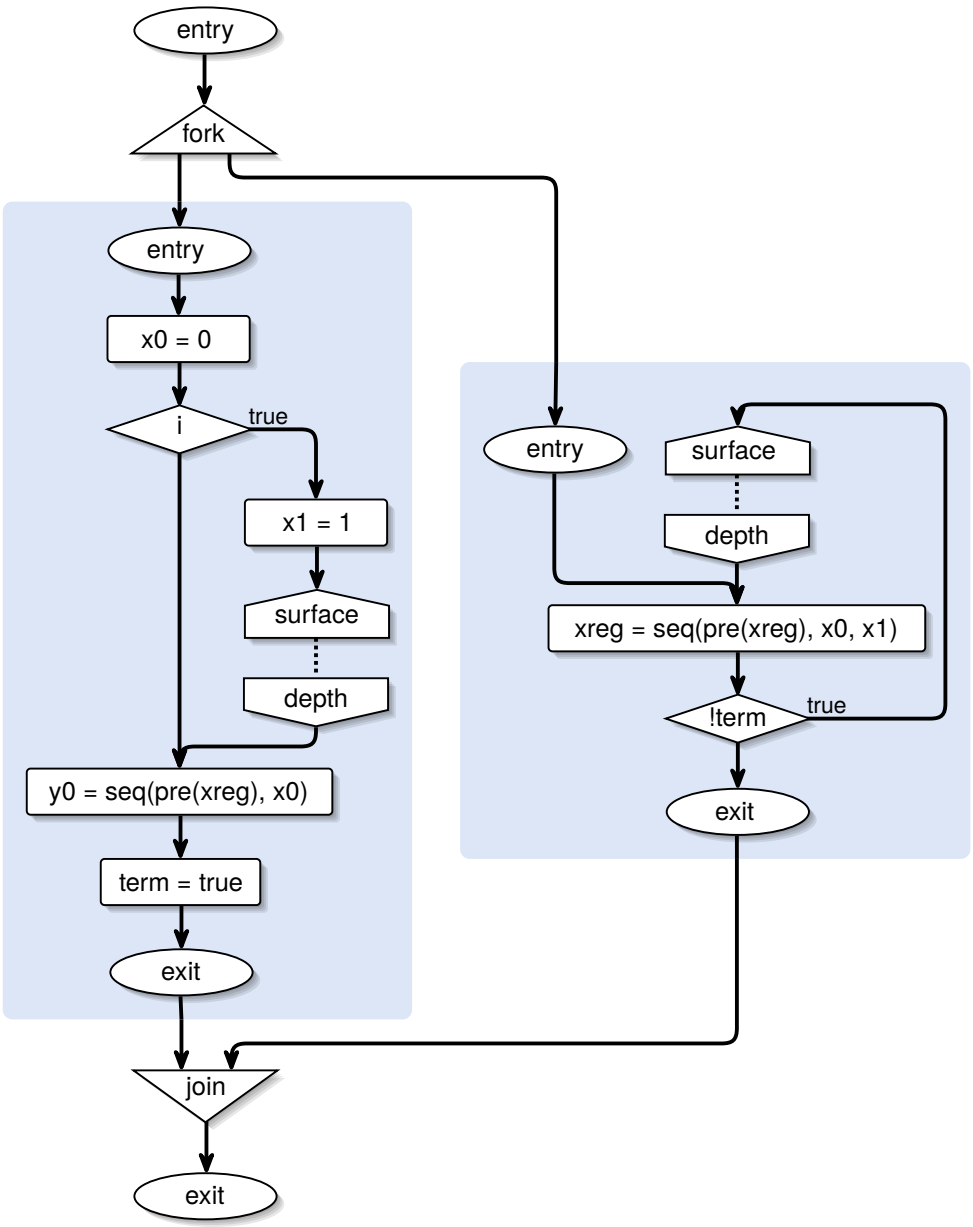
SSA Form: Delays Solved



Solution:

- Resolve and save variable values in each tick
- Store values in register variables
- Use *pre* to consider values of the previous tick in merge expressions

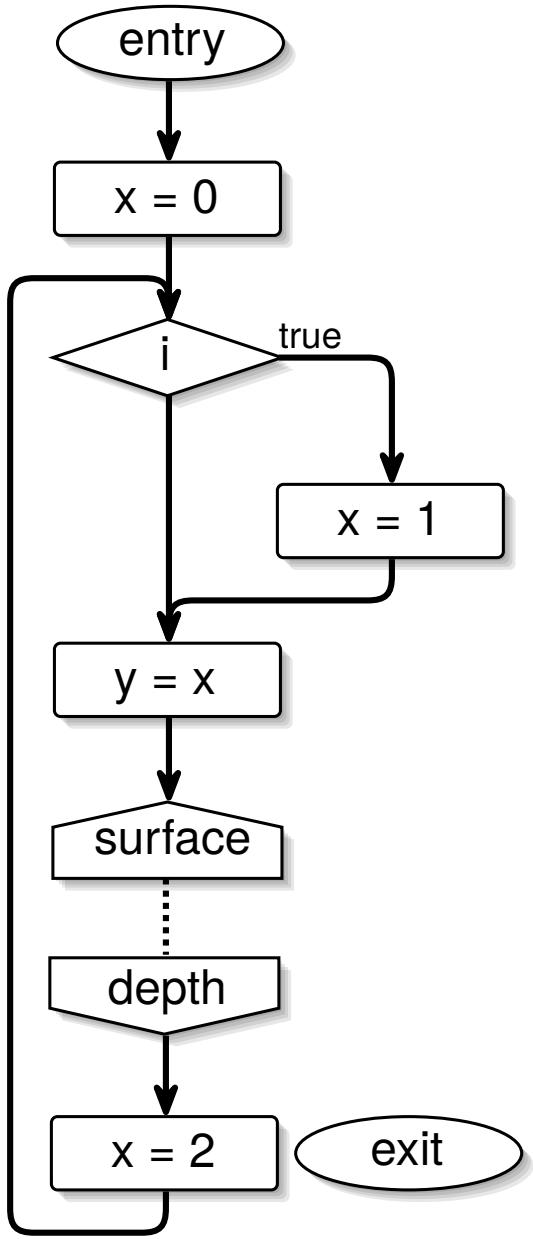
SSA Form: Delays Solved



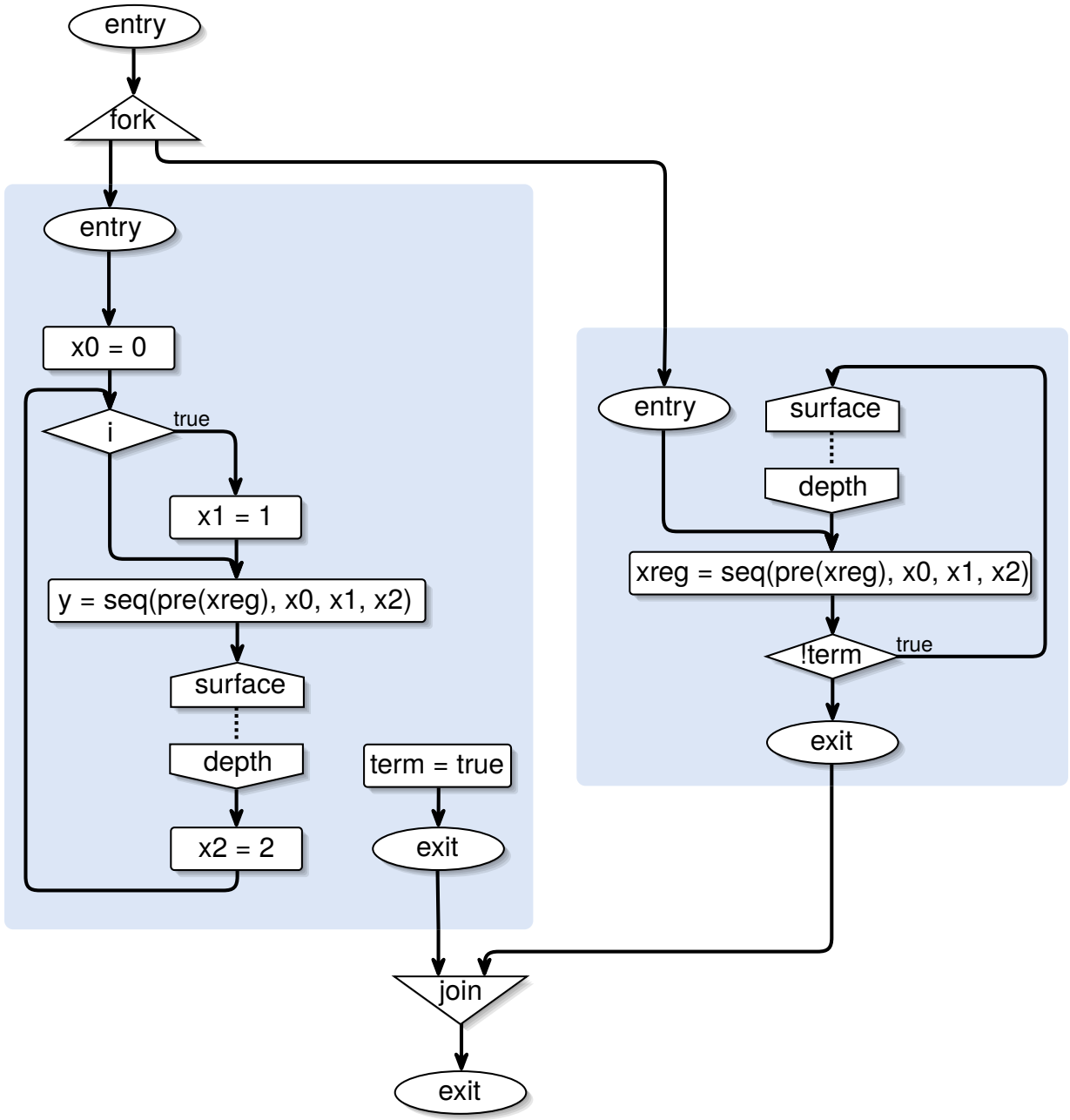
Solution:

- Resolve and save variable values in each tick
- Store values in register variables
- Use *pre* to consider values of the previous tick in merge expressions
- Reduce merge expression based on tick borders

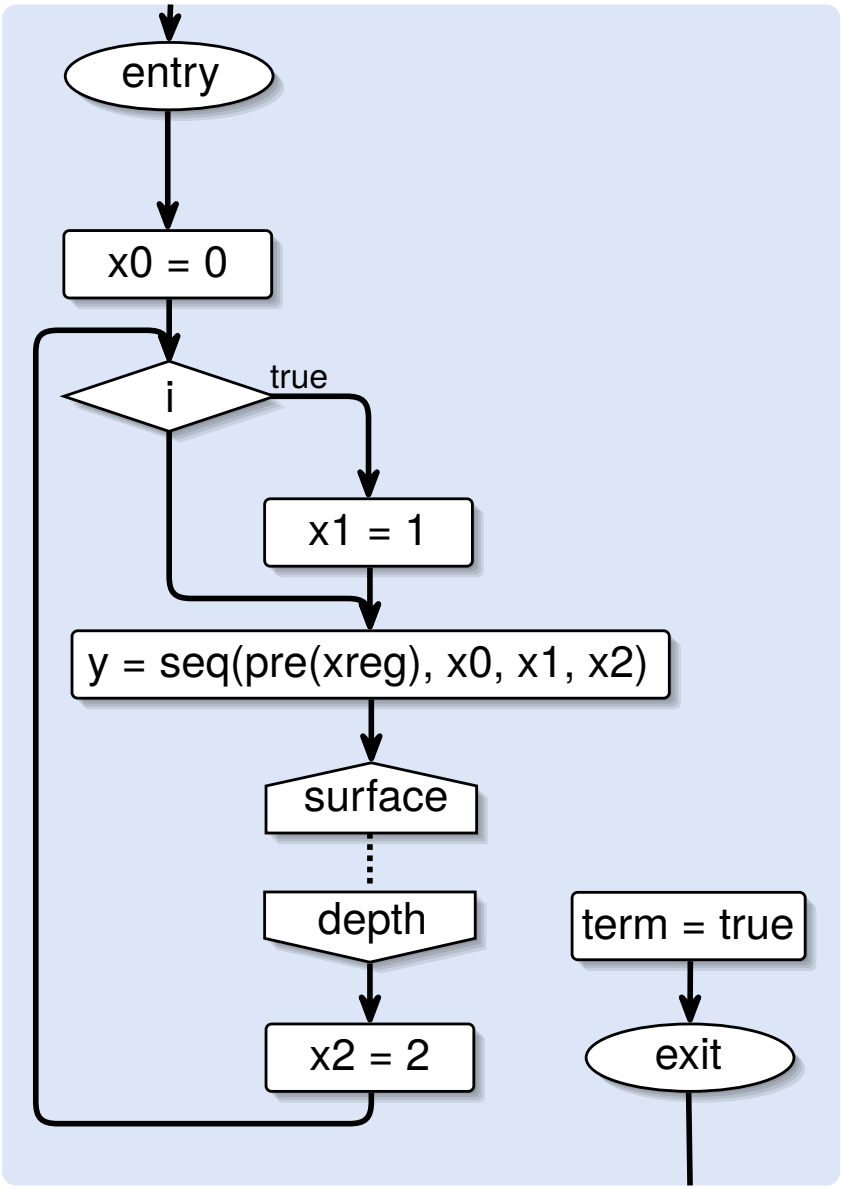
SSA Form: Loops



SSA Form: Loops

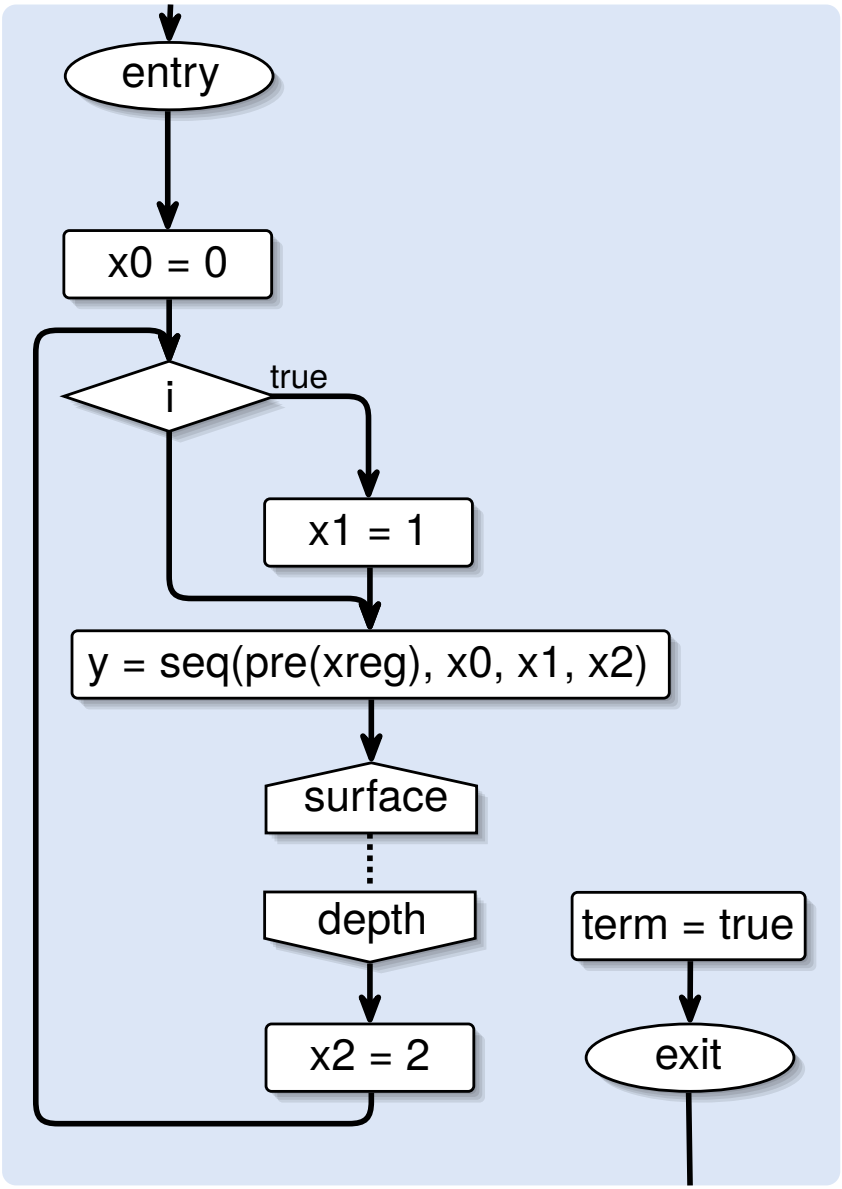


SSA Form: Loop Handling



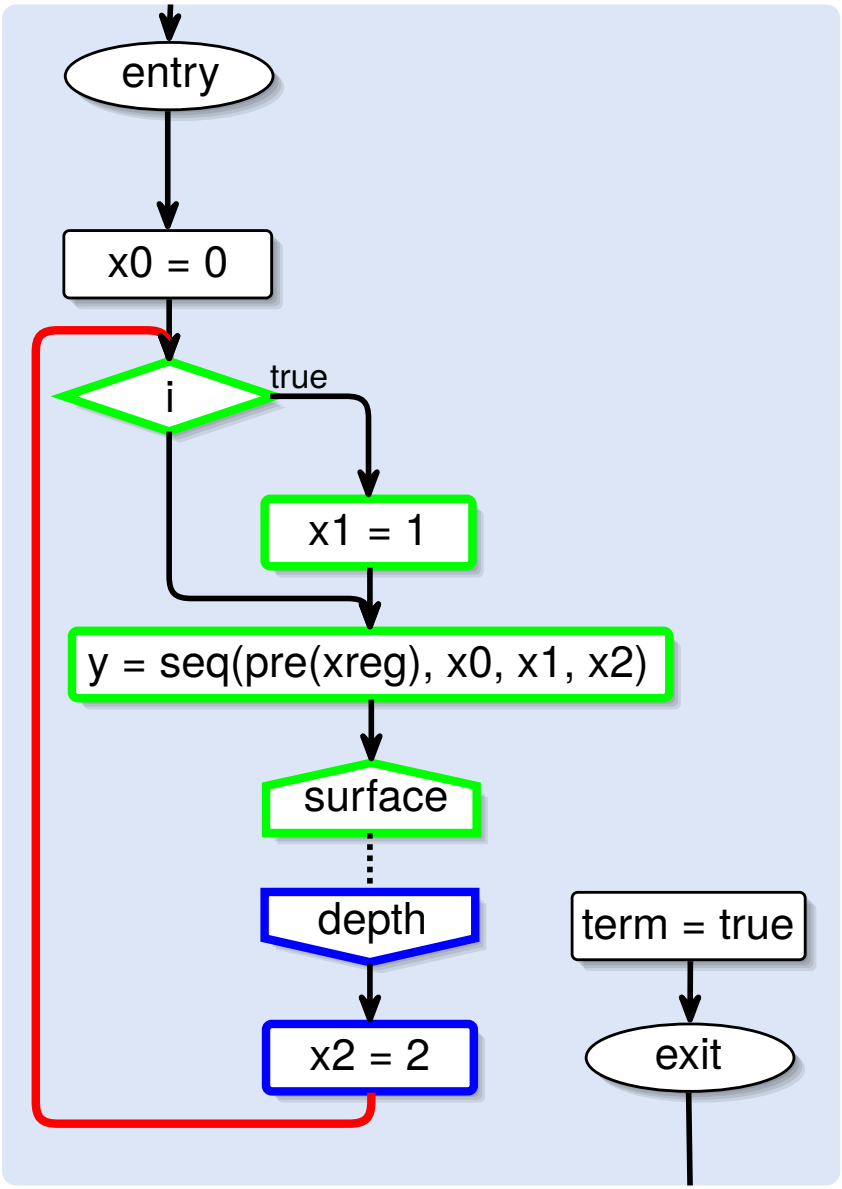
- Merge expressions require static ordering

SSA Form: Loop Handling



- Merge expressions require static ordering
- Wrong ordering due to simple structure analysis

SSA Form: Loop Handling

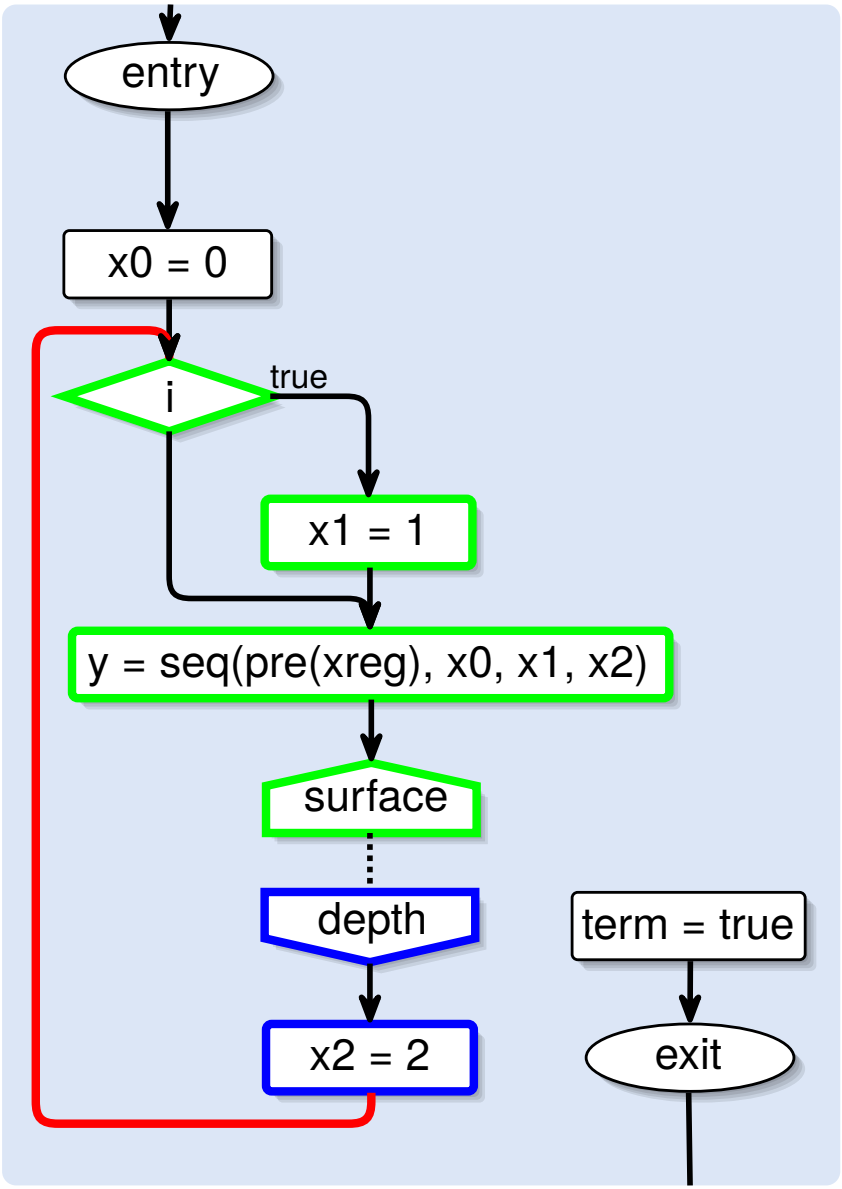


- Merge expressions require static ordering
- Wrong ordering due to simple structure analysis

Solution:

- Surface-Depth analysis

SSA Form: Loop Handling

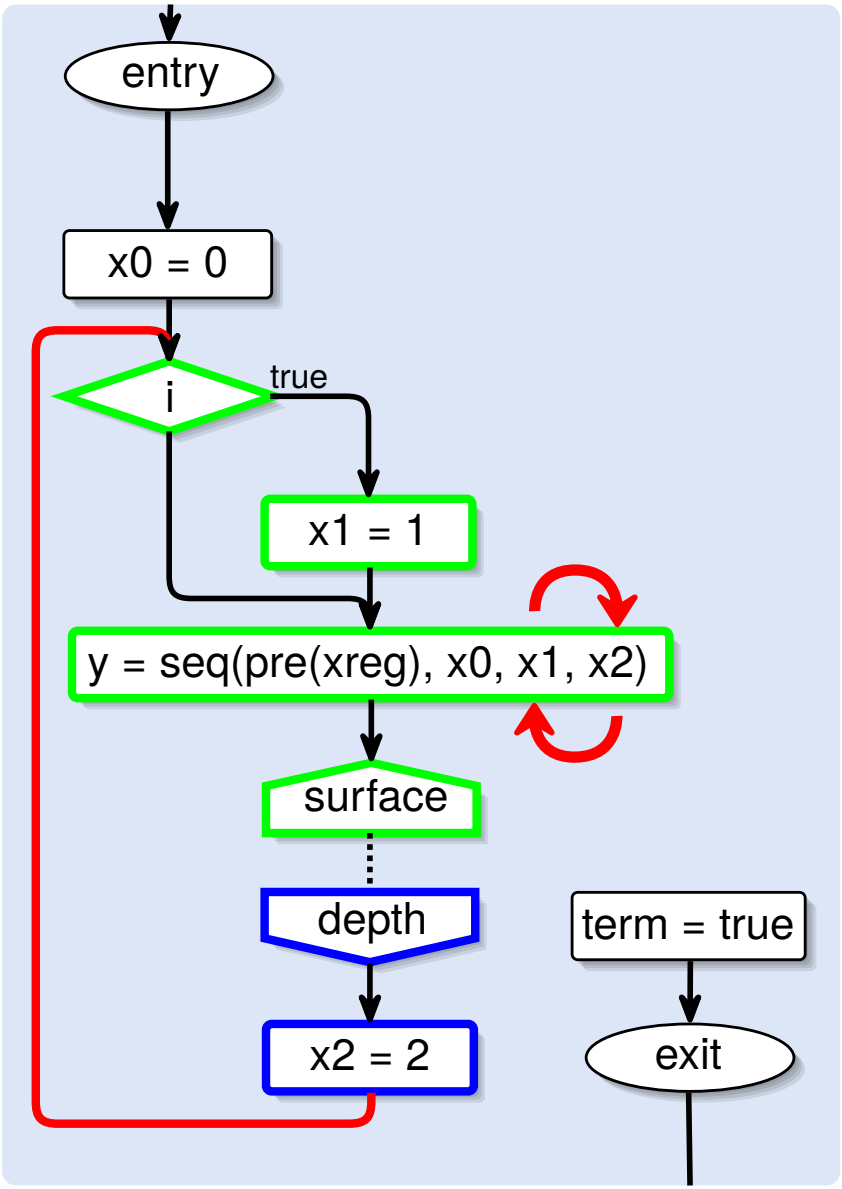


- Merge expressions require static ordering
- Wrong ordering due to simple structure analysis

Solution:

- Surface-Depth analysis
- Requires a pause that is always executed

SSA Form: Loop Handling

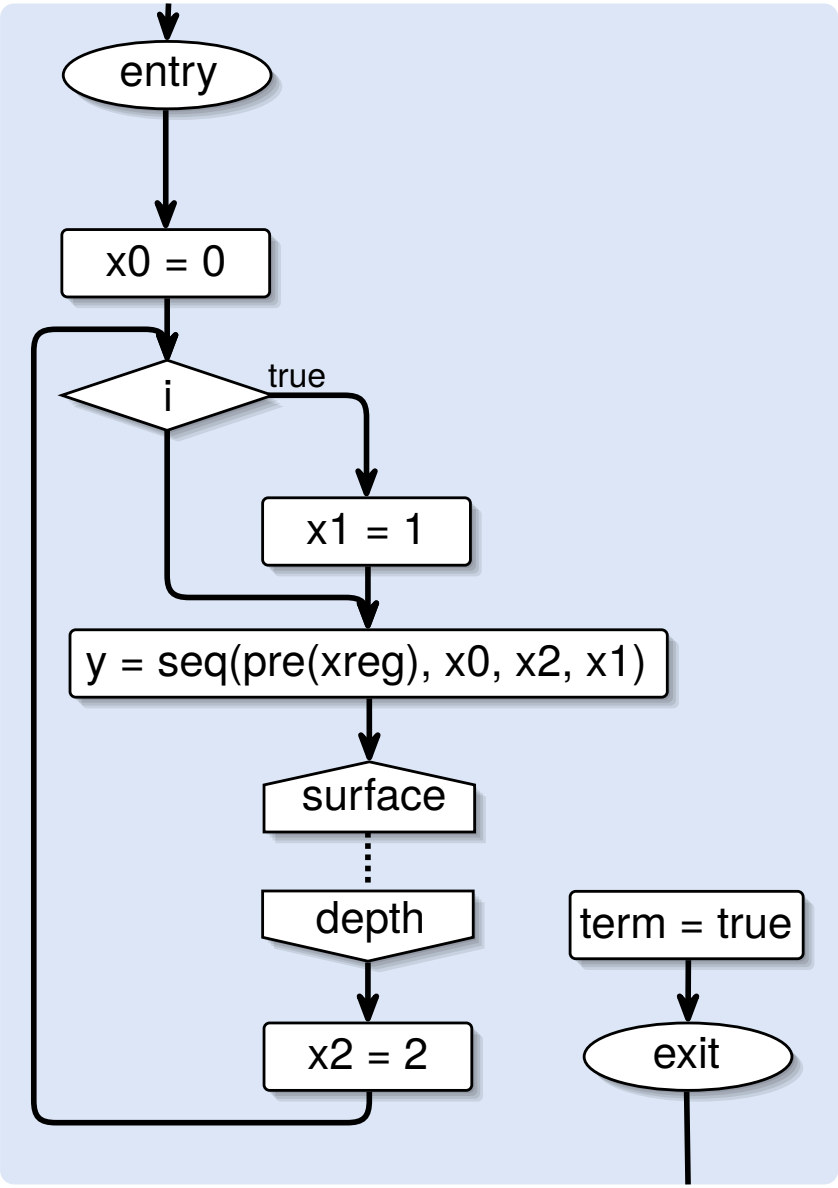


- Merge expressions require static ordering
- Wrong ordering due to simple structure analysis

Solution:

- Surface-Depth analysis
- Requires a pause that is always executed
- Switch order of writes in the surface with depth

SSA Form: Loop Handling

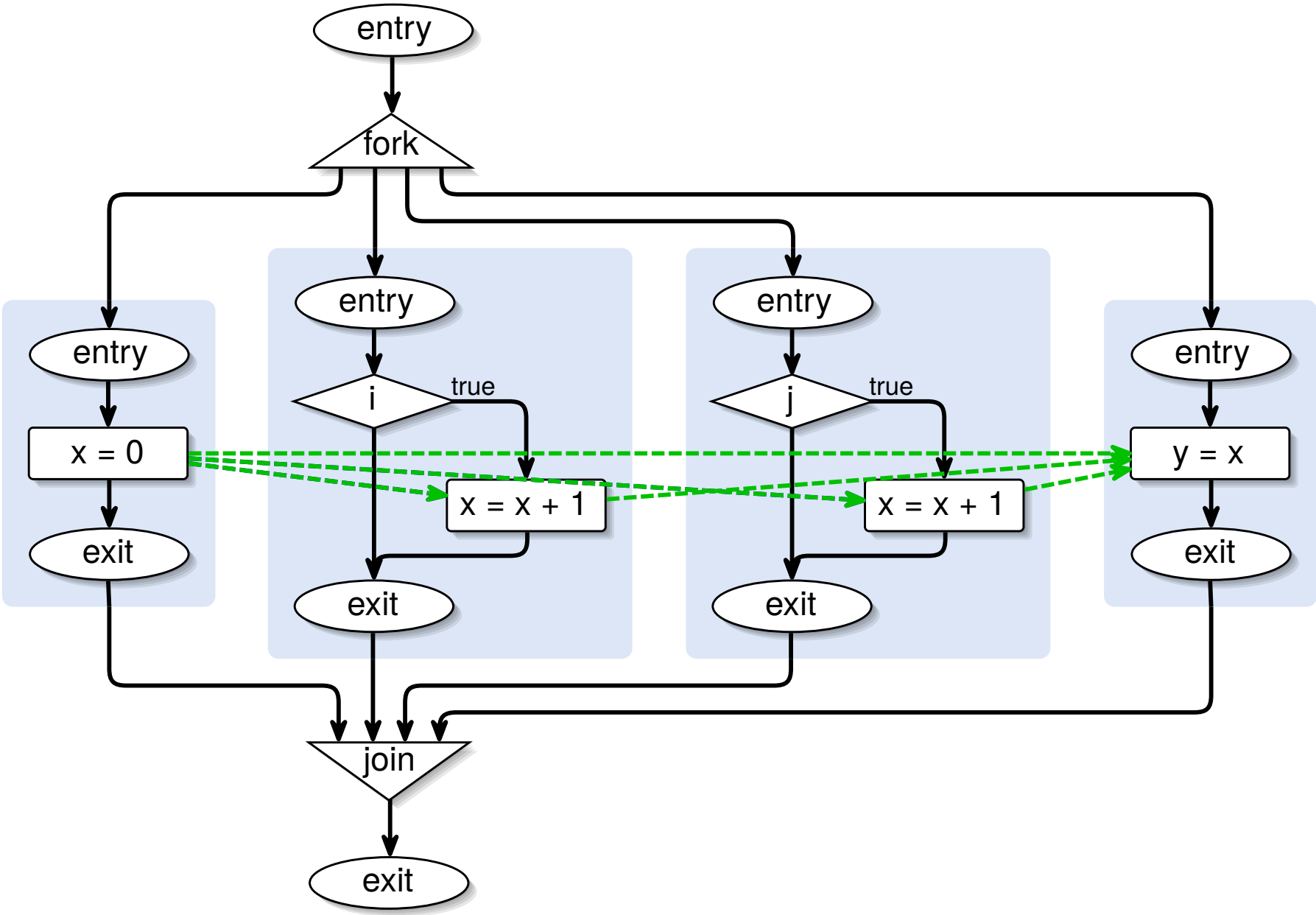


- Merge expressions require static ordering
- Wrong ordering due to simple structure analysis

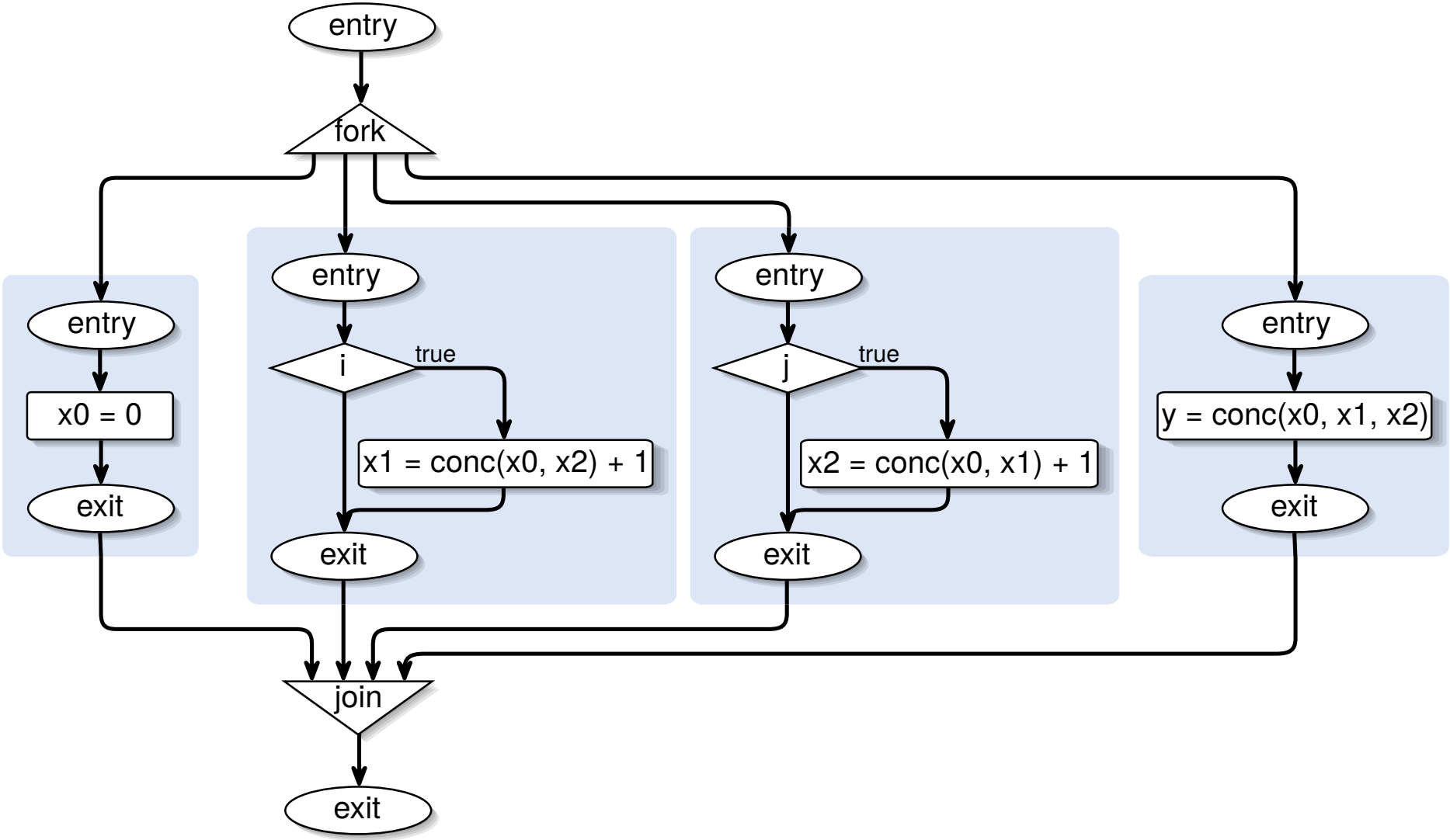
Solution:

- Surface-Depth analysis
- Requires a pause that is always executed
- Switch order of writes in the surface with depth

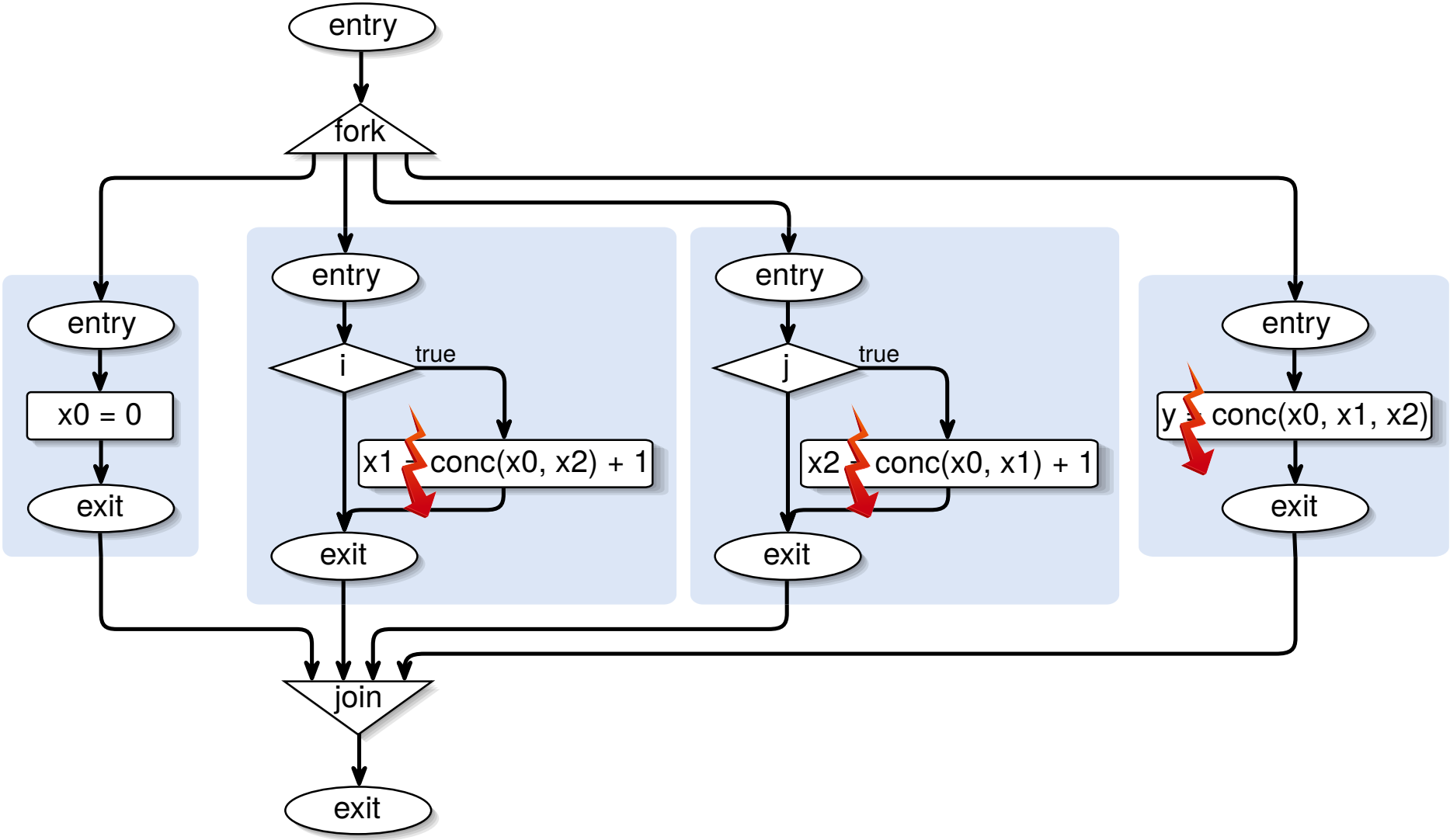
SSA Form: Updates



SSA Form: Updates



SSA Form: Updates



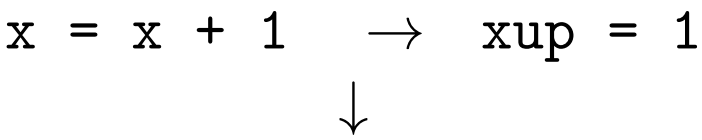
SSA Form: Update Handling

```
x = x + 1
```

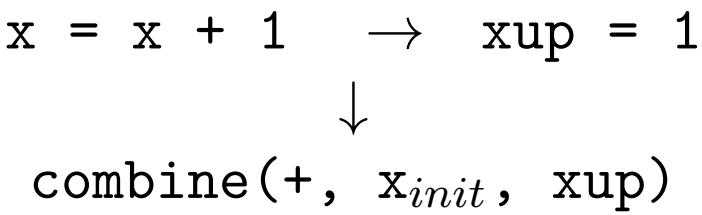
SSA Form: Update Handling

`x = x + 1` $\rightarrow$ `xup = 1`

SSA Form: Update Handling



SSA Form: Update Handling



SSA Form: Update Handling

```
x = x + 1   →   xup = 1
                  ↓
 combine(+, x_init, xup)
```

```
1  combine(f, ⟨x^p, x⟩, ⟨x^p_up, x_up⟩) :=
2    present x^p then
3      present x^p_up then
4        return ⟨x^p, f(x, x_up)⟩
5      else
6        return ⟨x^p, x⟩
7    else
8      present x^p_up then
9        reject
10     else
11       return ⟨absent, nil⟩
```

SSA Form: Update Handling

```
x = x + 1   →   xup = 1
                  ↓
 combine(+, x_init, xup)
```

```
1  combine(f, ⟨x^p, x⟩, ⟨x^p_up, x_up⟩) :=
2    present x^p then
3      present x^p_up then
4        return ⟨x^p, f(x, x_up)⟩
5      else
6        return ⟨x^p, x⟩
7    else
8      present x^p_up then
9        reject
10     else
11       return ⟨absent, nil⟩
```

- Special *seq* function for updates

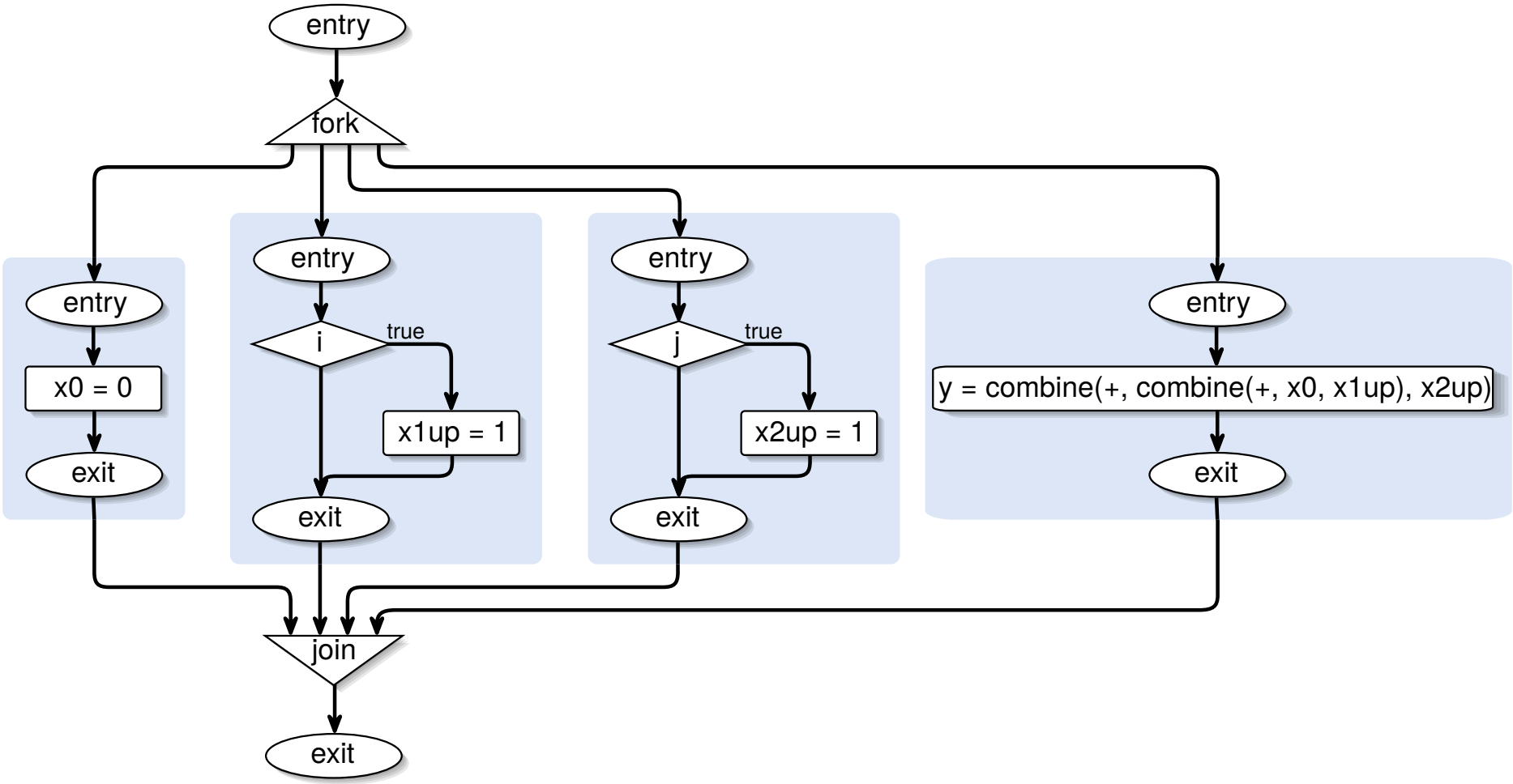
SSA Form: Update Handling

```
x = x + 1   →   xup = 1
                  ↓
 combine(+, x_init, xup)
```

```
1  combine(f, ⟨x^p, x⟩, ⟨x^p_up, x_up⟩) :=
2    present x^p then
3      present x^p_up then
4        return ⟨x^p, f(x, x_up)⟩
5      else
6        return ⟨x^p, x⟩
7    else
8      present x^p_up then
9        reject
10     else
11       return ⟨absent, nil⟩
```

- Special *seq* function for updates
- Requires partial static schedule to generate merge expressions

SSA Form: Updates Solved



SSA Form: Interface

```
1  module IO
2  input int I;
3  output int O;
4  {
5    if I < 0 then
6      I = 0
7    end;
8    O = I;
9    pause;
10   O = O * I
11 }
```

SSA Form: Interface

```
1  module IO
2  input int I;
3  output int O;
4  {
5    if I < 0 then
6      I = 0
7    end;
8    O = I;
9    pause;
10   O = O * I
11 }
```

- SSA renaming should not violate the original interface

SSA Form: Interface

```
1  module IO
2  input int I;
3  output int O;
4  {
5    if I < 0 then
6      I = 0
7    end;
8    O = I;
9    pause;
10   O = O * I
11 }
```

- SSA renaming should not violate the original interface
- Inputs must be read from the environment in each tick

SSA Form: Interface

```
1  module IO
2  input int I;
3  output int O;
4  {
5    if I < 0 then
6      I = 0
7    end;
8    O = I;
9    pause;
10   O = O * I
11 }
```

- SSA renaming should not violate the original interface
- Inputs must be read from the environment in each tick
- Inputs can be locally overridden

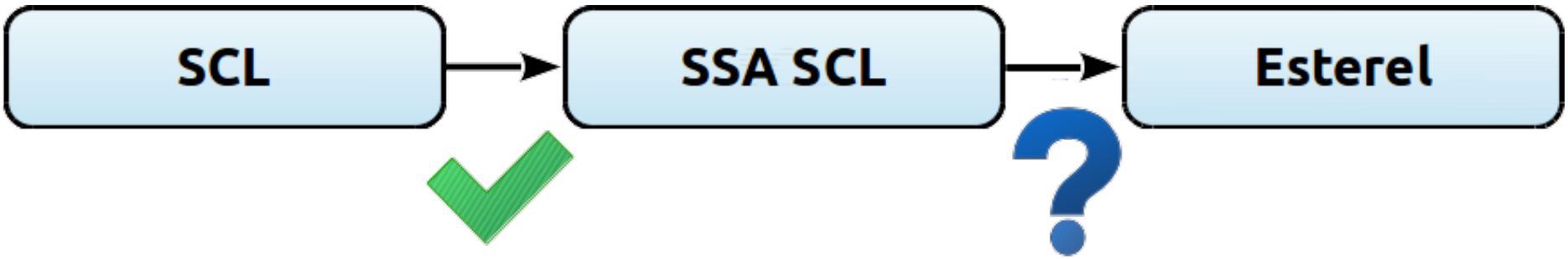
SSA Form: Interface

```
1  module IO
2  input int I;
3  output int O;
4  {
5    if I < 0 then
6      I = 0
7    end;
8    O = I;
9    pause;
10   O = O * I
11 }
```

- SSA renaming should not violate the original interface
- Inputs must be read from the environment in each tick
- Inputs can be locally overridden
- Outputs must be conveyed to the environment in each tick

SSA Form: Interface Solved

```
1  module IO-SSA
2  input int I;
3  int IO;
4  output int O;
5  int O0, O1, Oreg;
6  bool term = false;
7  {
8    fork
9      if I < 0 then
10       IO = 0
11     end;
12     O0 = seq(I, IO)
13     pause;
14     O1 = pre(Oreg) * I;
15     term = true
16   par
17     PauseLoop:
18     Oreg = seq(pre(Oreg), O0, O1);
19     O = Oreg;
20     if !term then
21       pause;
22       goto PauseLoop
23     end
24   join
25 }
```


SCL
SSA SCL
Esterel

Translation into Esterel

Translation of

- program structure

Translation into Esterel

Translation of

- program structure
- variables

Translation into Esterel

Translation of

- program structure
- variables

Esterel data-types:

- Variables

Translation into Esterel

Translation of

- program structure
- variables

Esterel data-types:

- Variables
- Valued signals

Translation into Esterel

Translation of

- program structure
- variables

Esterel data-types:

- Variables
- Valued signals
- Pure signals

Translation into Esterel

Translation of

- program structure
- variables

Esterel data-types:

- Variables
- Valued signals
- Pure signals

Presence Encoding

$\mathbf{x}_i^p \setminus \mathbf{x}_i$	**present**	**absent**
present	true	false
absent	undef	undef

Translation into Esterel

Translation of

- program structure
- variables

Esterel data-types:

- Variables
- Valued signals
- Pure signals

Presence Encoding

$\mathbf{x}_i^p \setminus \mathbf{x}_i$	**present**	**absent**
present	true	false
absent	undef	undef

Dual-Rail Encoding

$\mathbf{x}_i \setminus$ **not_x**$_i$	**present**	**absent**
present	illegal	true
absent	false	undef

Dual-Rail Encoding

x_i = true $\longrightarrow$ **emit** x_i

Dual-Rail Encoding

x_i = true $\longrightarrow$ **emit** x_i

x_i = false $\longrightarrow$ **emit** not_x_i

Dual-Rail Encoding

x_i = true $\rightarrow$ emit x_i

x_i = false $\rightarrow$ emit not_x_i

$x_i = e \rightarrow$

```
present errorExpr(e) then
  emit error
else
  [
    present trueExpr(e) then
      emit x_i
    end
  ||
    present falseExpr(e) then
      emit not_x_i
    end
  ]
end
```

Dual-Rail Encoding

```
if (e) then
  //then-block
else
  //else-block
end
```

→

```
present errorExpr(e) then
  emit error
else
  [
    present trueExpr(e) then
      % then-block
    end
  ||
    present falseExpr(e) then
      % else-block
    end
  ]
end
```

Dual-Rail Encoding

$\mathtt{x}_i$:

trueExpr: $\mathtt{x}_i$

falseExpr: $\mathtt{not\_x}_i$

Dual-Rail Encoding

$\mathtt{x}_i$:

trueExpr: $\mathtt{x}_i$

falseExpr: $\mathtt{not\_x}_i$

conc(e_i, e_j):

errorExpr: $(\mathsf{trueExpr}(e_i) \wedge \mathsf{falseExpr}(e_j)) \vee (\mathsf{falseExpr}(e_i) \wedge \mathsf{trueExpr}(e_j))$

trueExpr: $\mathsf{trueExpr}(e_i) \vee \mathsf{trueExpr}(e_j)$

falseExpr: $\mathsf{falseExpr}(e_i) \vee \mathsf{falseExpr}(e_j)$

Dual-Rail Encoding

$\mathtt{x}_i$:

trueExpr: $\mathtt{x}_i$

falseExpr: $\mathtt{not\_x}_i$

conc(e_i, e_j):

errorExpr: $(\mathsf{trueExpr}(e_i) \wedge \mathsf{falseExpr}(e_j)) \vee (\mathsf{falseExpr}(e_i) \wedge \mathsf{trueExpr}(e_j))$

trueExpr: $\mathsf{trueExpr}(e_i) \vee \mathsf{trueExpr}(e_j)$

falseExpr: $\mathsf{falseExpr}(e_i) \vee \mathsf{falseExpr}(e_j)$

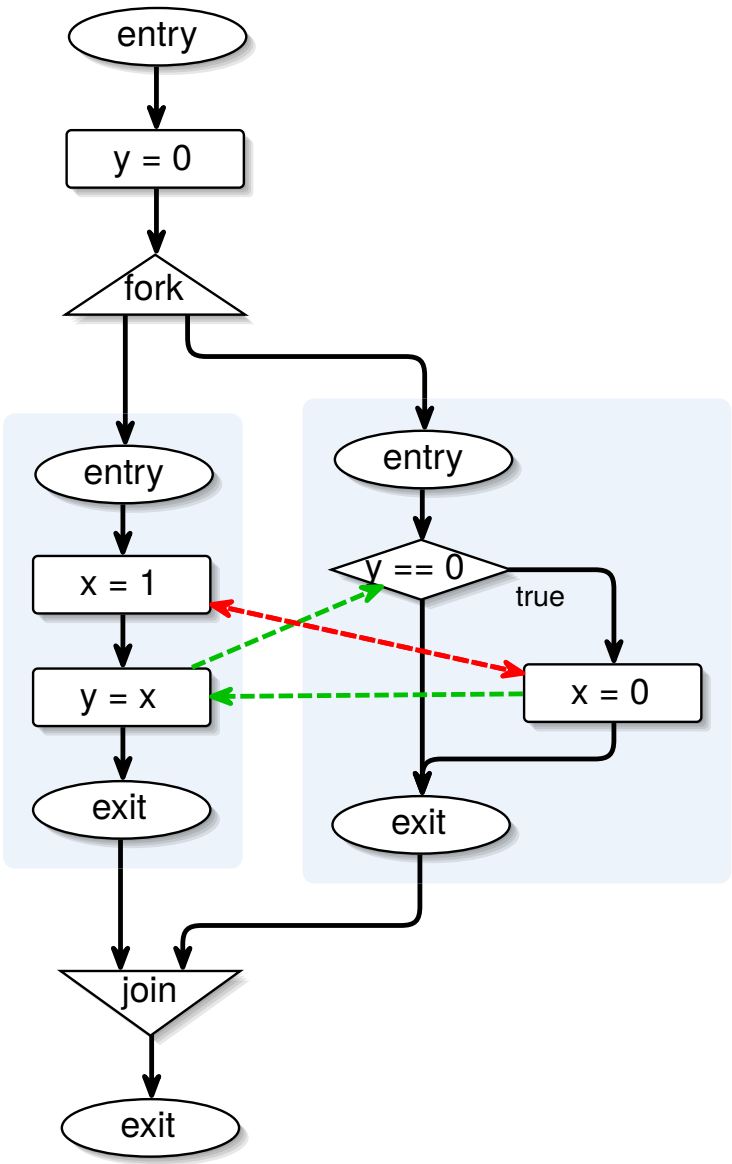
seq(e_i, e_j):

trueExpr: $\mathsf{trueExpr}(e_j) \vee (\neg\mathsf{falseExpr}(e_j) \wedge \mathsf{trueExpr}(e_i))$

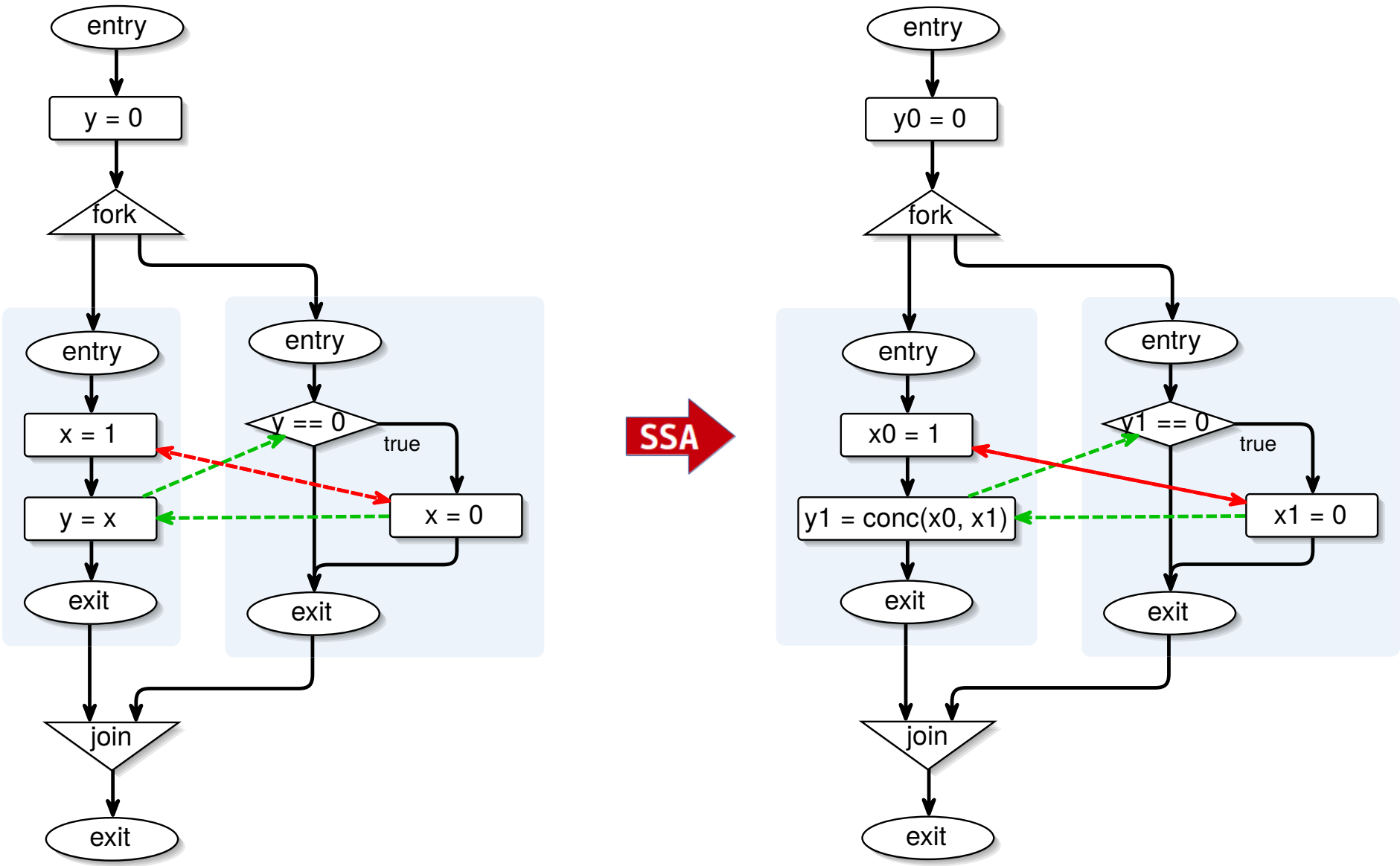
falseExpr: $\mathsf{falseExpr}(e_j) \vee (\neg\mathsf{trueExpr}(e_j) \wedge \mathsf{falseExpr}(e_i))$


SCL
SSA SCL
Esterel

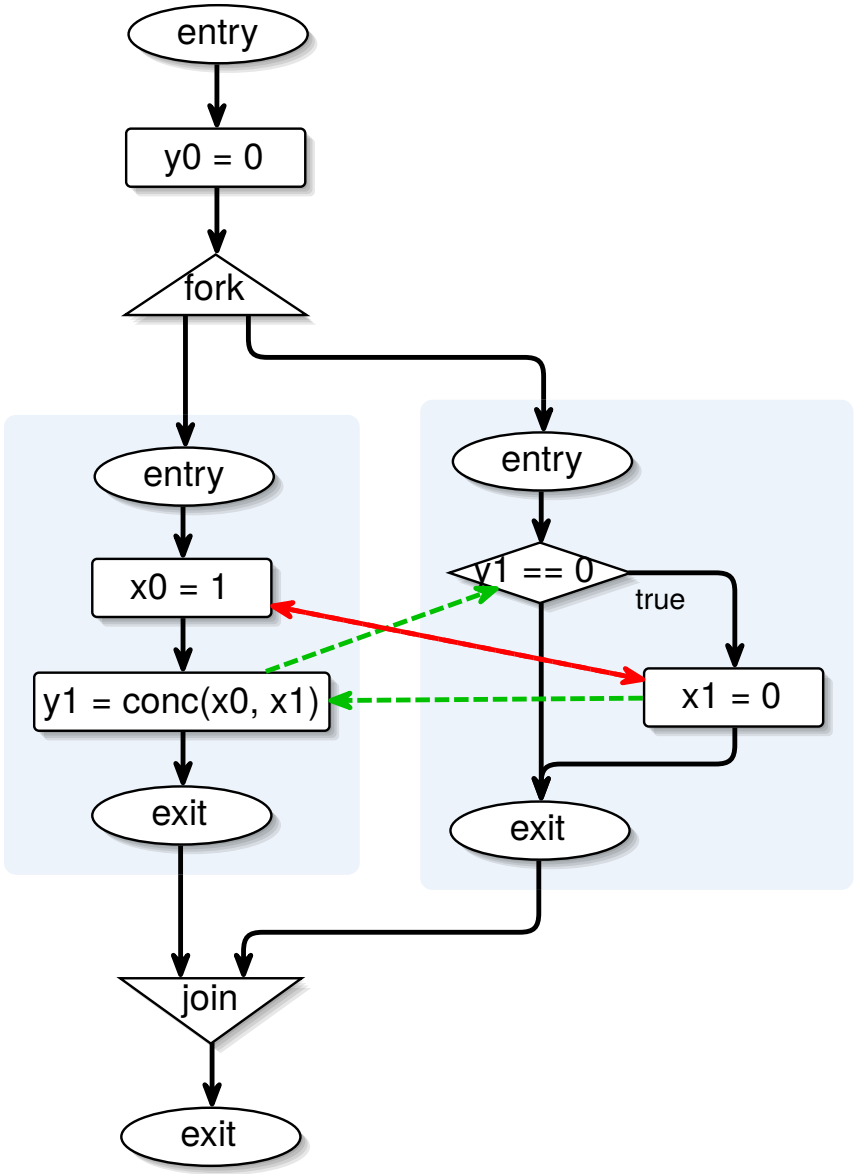
Back to P10



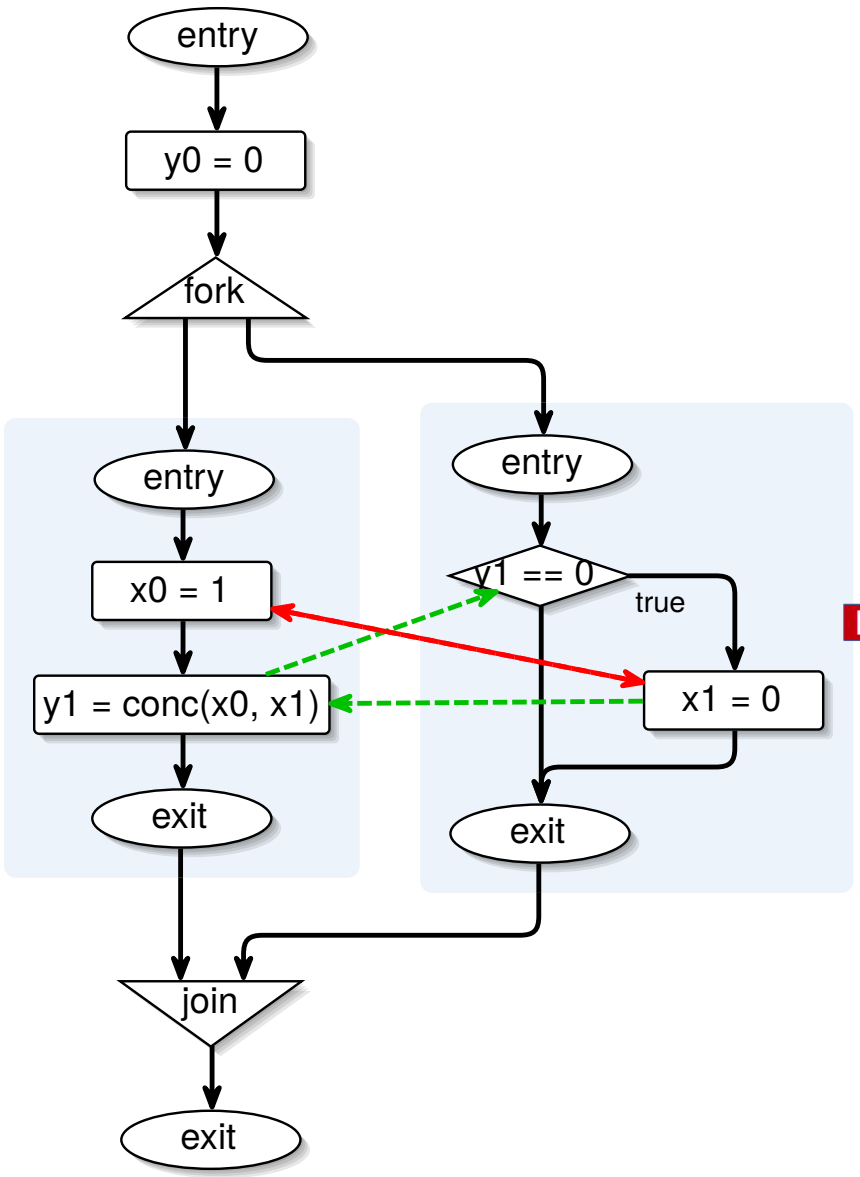
Back to P10


entry
y = 0
fork
entry
x = 1
y = x
exit
entry
y == 0
true
x = 0
exit
join
exit
SSA
entry
y0 = 0
fork
entry
x0 = 1
y1 = conc(x0, x1)
exit
entry
y1 == 0
true
x1 = 0
exit
join
exit

Back to P10


entry
y0 = 0
fork
entry
x0 = 1
y1 = conc(x0, x1)
exit
entry
y1 == 0
true
x1 = 0
exit
join
exit

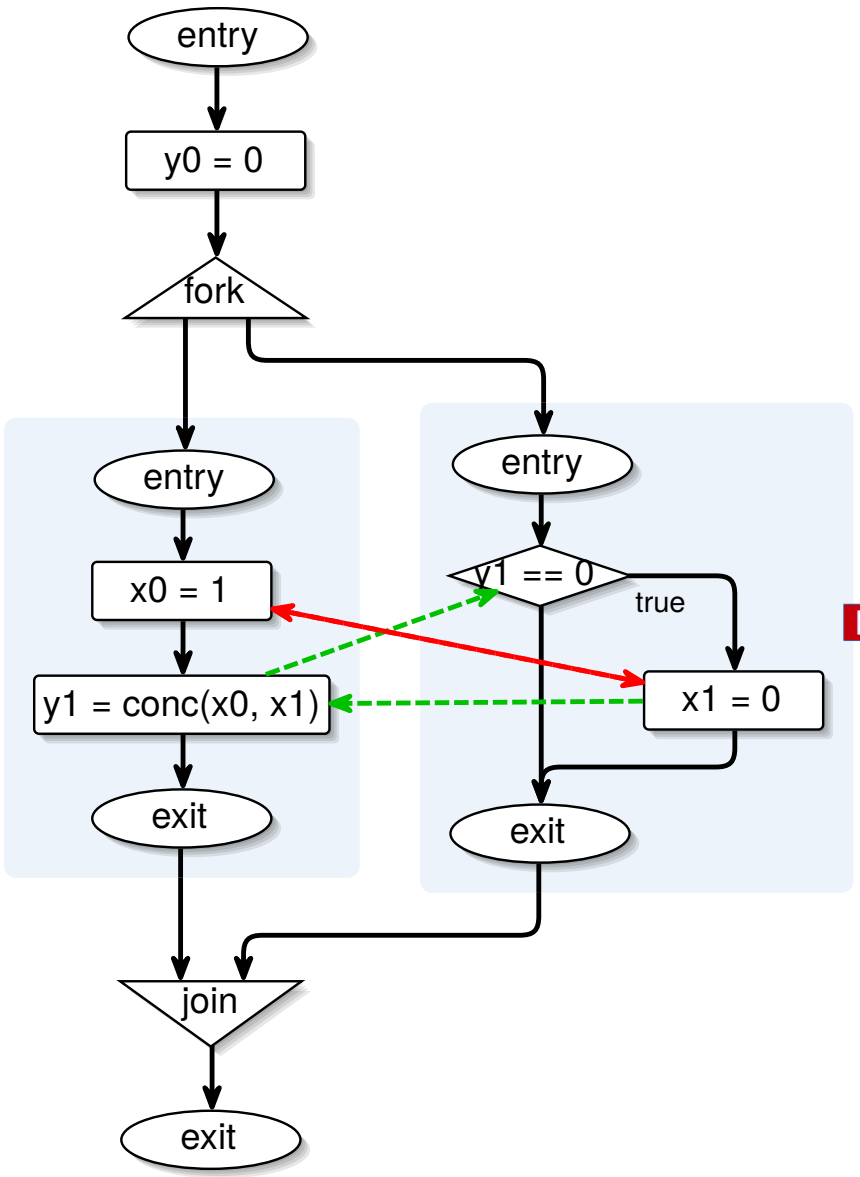
Back to P10



Esterel

```
1  module P10:
2  signal x0, not_x0, x1, not_x1 in
3  signal y0, not_y0, y1, not_y1 in
4  signal error in
5  [
6    emit not_y0;
7    [
8      emit x0;
9      present (x0 and not_x1) or
                  (not_x0 and x1) then
10       emit error
11     else
12       [
13         present x0 or x1 then
14           emit y1
15         end
16       ||
17         present not_x0 or not_x1 then
18           emit not_y1
19         end
20       ]
21     end
22   ||
23     present not_y1 then
24       emit not_x1
25     end
26   ]
27 ||
28   signal err in
29     present error then
30       present err else emit err end
31     end
32   end signal
```

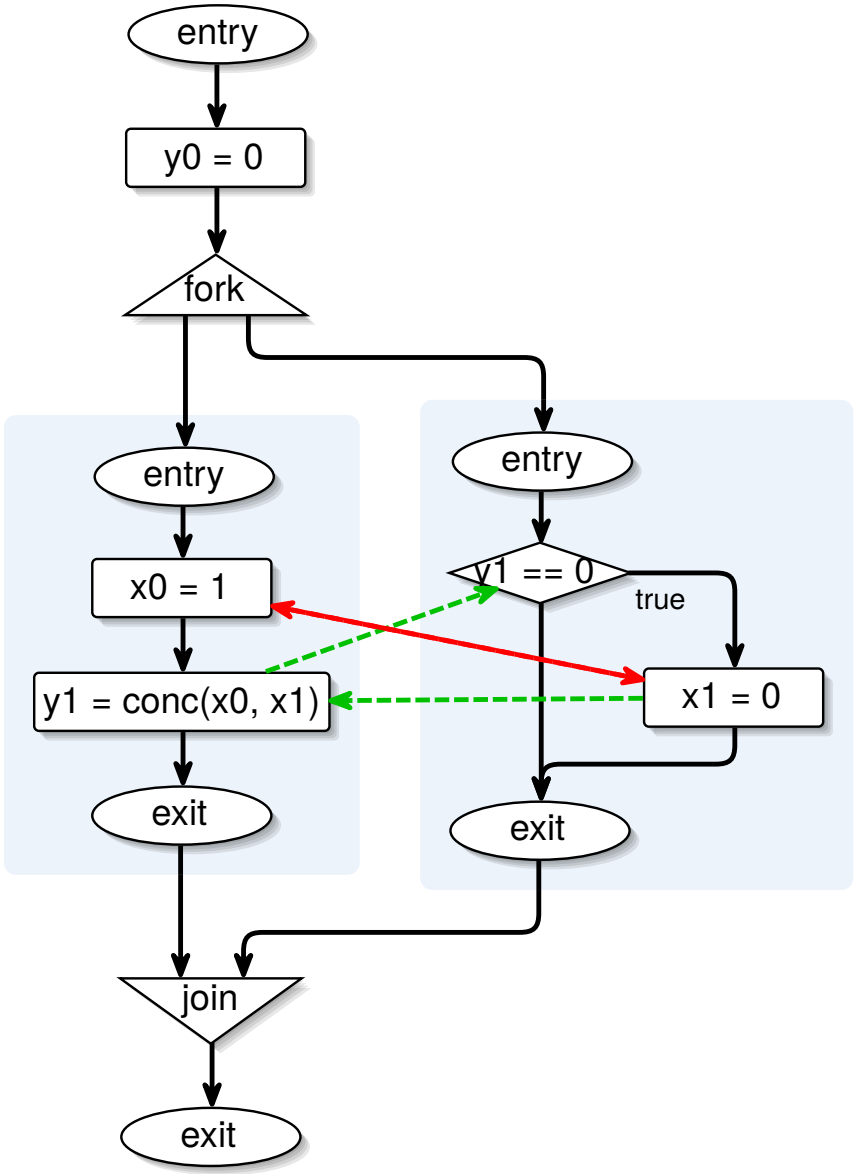
Back to P10



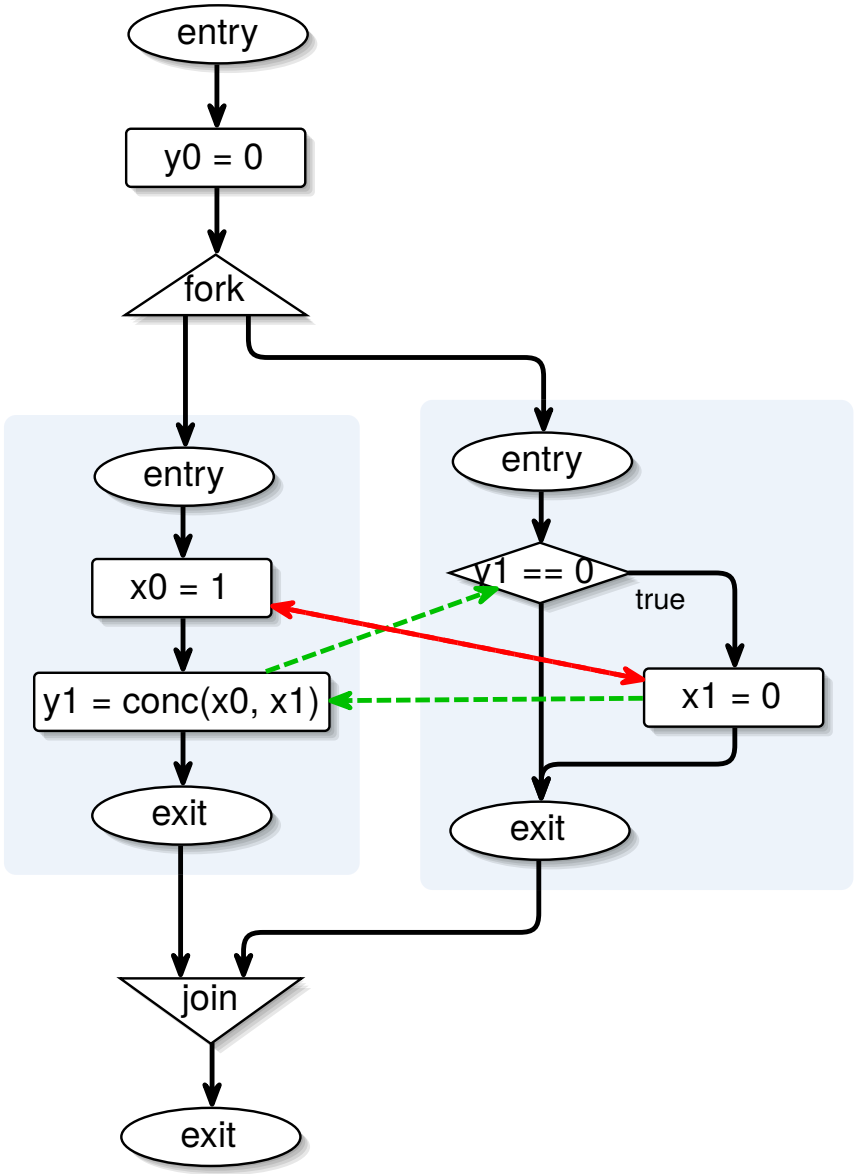
Esterel

```
1  module P10:
2  signal x0, not_x0, x1, not_x1 in
3  signal y0, not_y0, y1, not_y1 in
4  signal error in
5  [
6    emit not_y0;
7    [
8      emit x0;
9      present (x0 and not_x1) or
                 (not_x0 and x1) then
10       emit error
11     else
12       [
13         present x0 or x1 then
14           emit y1
15         end
16       ||
17         present not_x0 or not_x1 then
18           emit not_y1
19         end
20       ]
21     end
22   ||
23     present not_y1 then
24       emit not_x1
25     end
26   ]
27 ||
28   signal err in
29     present error then
30       present err else emit err end
31     end
32   end signal
```

Back to P10

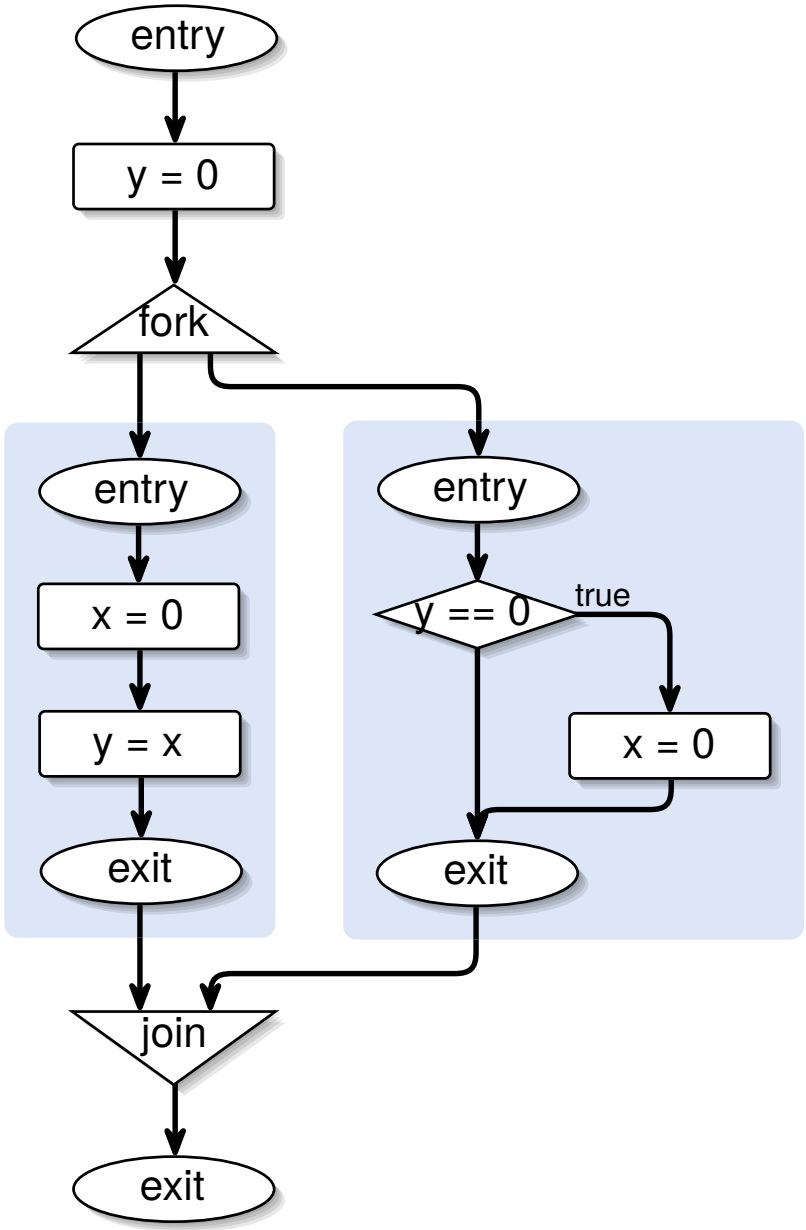

entry
y0 = 0
fork
entry
x0 = 1
y1 = conc(x0, x1)
exit
entry
y1 == 0
true
x1 = 0
exit
join
exit

Back to P10

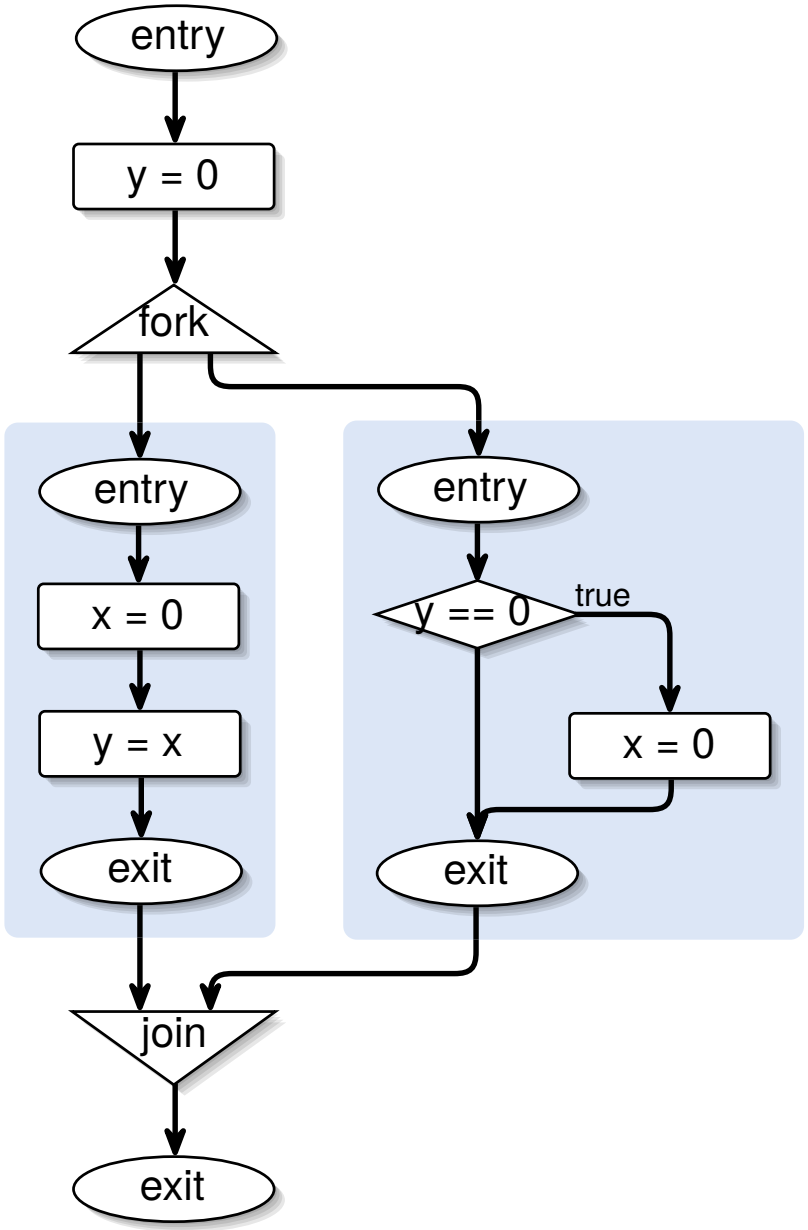


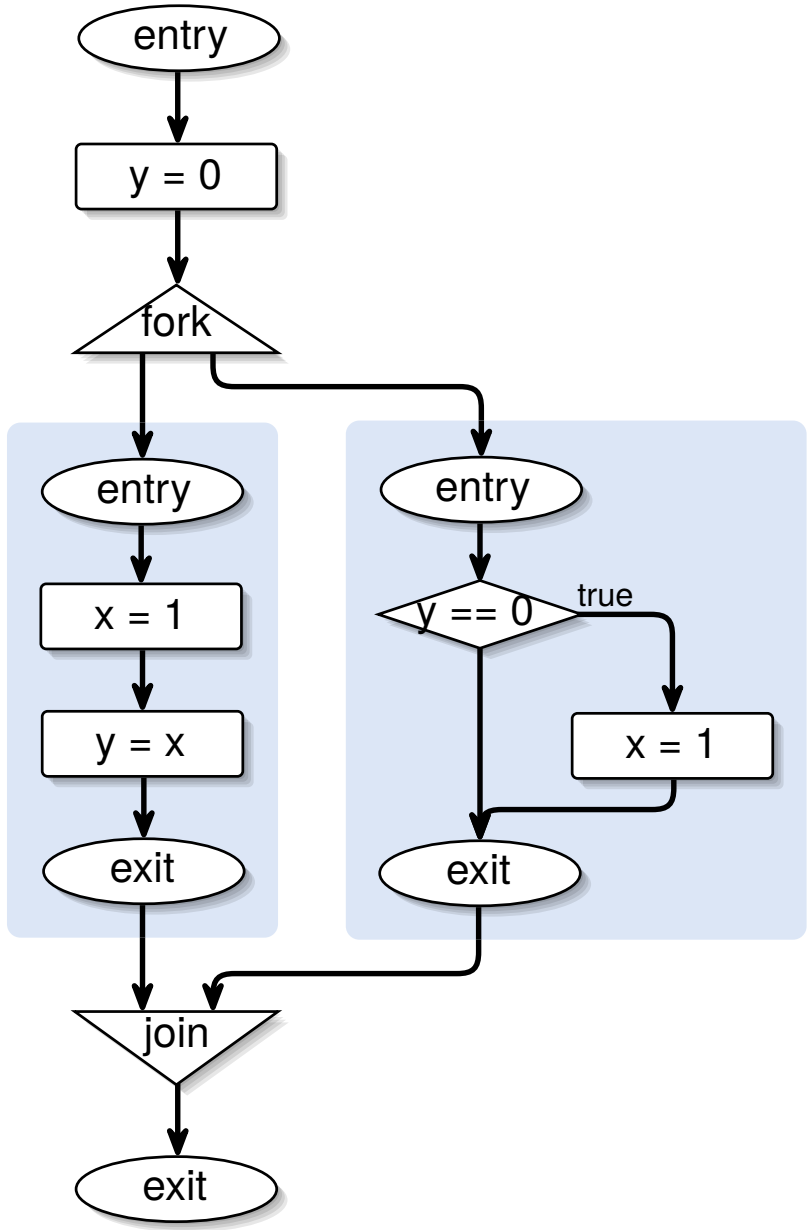
Not constructive in Esterel
$\Rightarrow$ not Strict SC

Confluent P10 Variants



Confluent P10 Variants



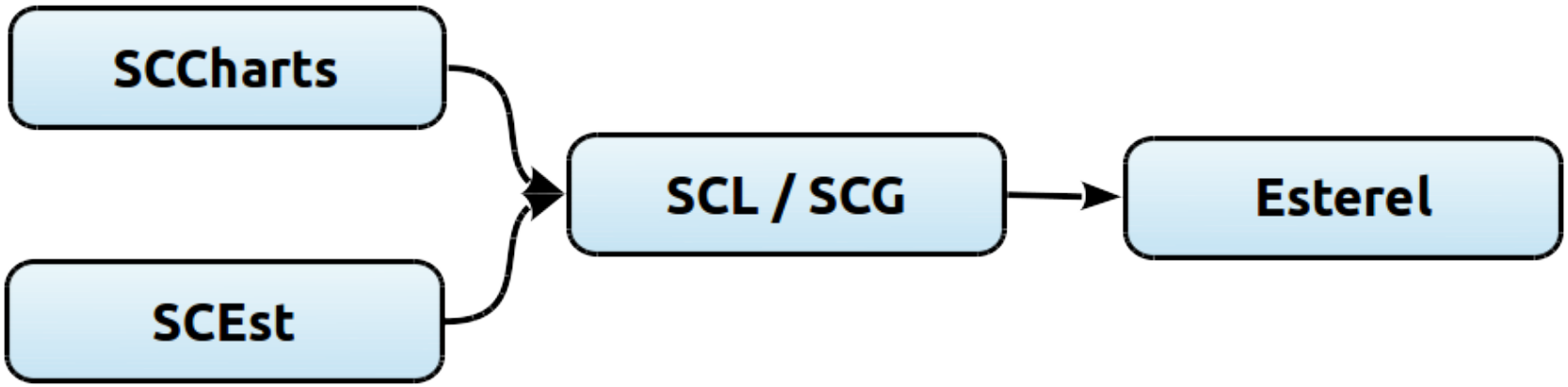


Confluent P10 Variants

```
[
  emit not_y0;
  [
    emit not_x0;
    present (x0 and not_x1) or (
         not_x0 and x1) then
      emit error
    else
      [
        present x0 or x1 then
          emit y1
        end
      ||
        present not_x0 or not_x1 then
          emit not_y1
        end
      ]
    end
  ||
    present not_y1 then
      emit not_x1
    end
  ]
||
  signal err in
  present error then
    present err else emit err end
  end
  end signal
]
```

```
[
  emit not_y0;
  [
    emit x0;
    present (x0 and not_x1) or (
         not_x0 and x1) then
      emit error
    else
      [
        present x0 or x1 then
          emit y1
        end
      ||
        present not_x0 or not_x1 then
          emit not_y1
        end
      ]
    end
  ||
    present not_y1 then
      emit x1
    end
  ]
||
  signal err in
  present error then
    present err else emit err end
  end
  end signal
]
```

New Compile Chain



Future Work

- Optimized translation for SCEst

Future Work

- Optimized translation for SCEst
- Code optimization based on SSA

Future Work

- Optimized translation for SCEst
- Code optimization based on SSA
- Loop unrolling for (bounded) instantaneous loops

Future Work

- Optimized translation for SCEst
- Code optimization based on SSA
- Loop unrolling for (bounded) instantaneous loops
- Dynamic scheduling of updates

Future Work

- Optimized translation for SCEst
- Code optimization based on SSA
- Loop unrolling for (bounded) instantaneous loops
- Dynamic scheduling of updates
- Reduction merge expression insertion