Language Based Security (LBT)

Secure compilation

Chiara Bodei, Gian-Luigi Ferrari

Lecture May, 6 2024

Outline

Compilation security issues: overview

Dead store elimination

Undefined behaviour and unstable code

(un)-security of a programming language

Avoid discrepancies between:

- what the programmer has in mind
- what the compiler/interpreter understands
- how the executable code may behave…

At least three gaps

Gaps:

- between a system's specification and its implementation: despite demonstrating a security property for the specification, an attacker might still be able to violate it in the implementation
- between a system's source code and its compiled executable
- between the program and the machine executing it

What you see is not what you execute
Balakrishnan and Rep

Compiler-induced security vulnerabilities

- High-level languages provide abstractions and mechanisms that may be used to enforce security properties
- Security properties should be preserved after compilation
- Something can be done on the compiler tool chain

Which defenses?

As seen in the previous lectures, there are defences that the compiler can take and that contribute to mitigate these issues:

- **“Platform”-level defences** to protect from memory corruption, such as stack canaries, Address Space Randomization, Shadow stack, NX memory, Fat pointers, various forms of integrity checks on control flow. Code generation must take them into account to generate the code that exploits the defences
- **Static Analysis techniques**, such as type systems, Information Flow and Taint Analysis
- **Compiler instrumentation** that takes advantage of the compiler’s low-level representation of the code and the compiler’s analyses to make the instrumentation precise and “surgical,” by instrumenting only the subset of events of interest. Examples: CFI
- **Enforcement mechanisms** via compiler instrumentation
- **Security policiess**

Compiler-induced security vulnerabilities (cont.)

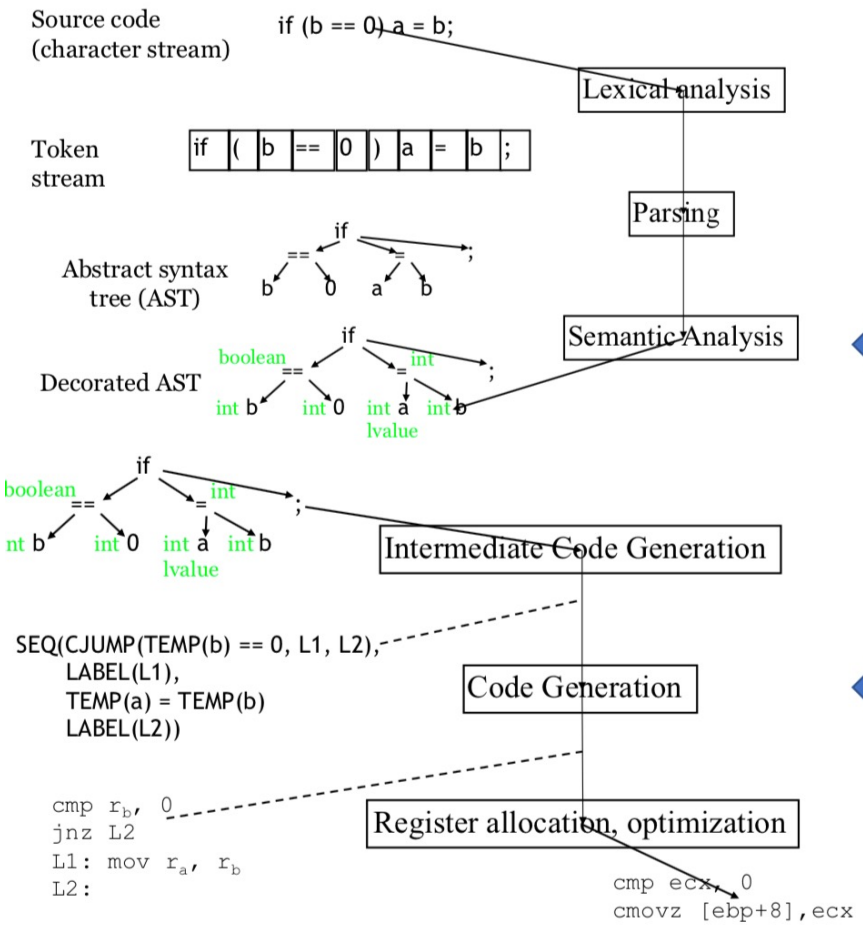
- High-level languages provide abstractions and mechanisms that may be used to enforce security properties
- Security properties should be preserved after compilation
- Something can be done on the compiler tool chain
- Even compiler optimizations may hinder security

Compiler-induced security vulnerabilities (cont.)

- High-level languages provide abstractions and mechanisms that may be used to enforce security properties
- Security properties should be preserved after compilation
- Something can be done on the compiler tool chain
- Even **compiler optimizations** may hinder security

- We first focus on the last one

Compiler tool chain



Types and type cheching
Control Flow Integrity
Information flow analysis
Taint Analysis
Abstract Interpretation

Run-time structure organization
Trusted Execution Environments

Optimization performed non only here

Vulnerabilities introduced by compiler optimizations

The compiler may correctly optimize source code designed for security purposes in a way that might negate the intended security guarantee

A security-oriented analysis of optimizations*

- A sound, correctly implemented, compiler optimization can violate security guarantees (they were not designed to satisfy)
- Security weaknesses introduced by compiler optimizations are mainly:
  - information leaks through persistent state (e.g., operations that scrub memory of sensitive data can lead to dead stores)
  - elimination of security-relevant code due to undefined behaviour [optimization-unstable code]
  - introduction of side channels (e.g., reducing operation in one IF-branch, making timing info inferable)

* Vijay D'Silva, Mathias Payer, Dawn Song. *The Correctness-Security Gap in Compiler Optimization*. IEEE Symposium on Security and Privacy Workshops 2015

Compiler-Introduced Security Bugs*

Fundamental and general causes:

- **Implicit Specification**: the compiler makes assumptions that conflict with the functionality of the code in respect to the security property. This allows the compiler to perform aggressive optimizations to dismantle the security property
- **Orthogonal Specification**: The security property exceeds the semantic functionality scope of language specifications (orthogonal w.r.t. correctness), e.g., execution time or the lifetime/region of sensitive data

* J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M.s Payer, B. Mao: *Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs*. USENIX Security Symposium 2023: 3655-3672

Compiler-Introduced Security Bugs

Table 4: A three-layer taxonomy of compiler introduced security bugs.

Root cause	Insecure optimization behaviors	Security consequences
Implicit Specification §4 (*No-UB*, *Default-behavior*, and *Environment*)	Eliminating security related code §4.1	Elimination of security checks
		Elimination of critical memory operations
	Reordering order-sensitive security code §4.2	Disorder between order-sensitive memory operations
		Disorder between security checks and dangerous operations
	Introducing insecure instructions §4.3	Introduction of invalid instructions of certain environments
		Introduction of insecure logic
Orthogonal Specification §5	Making sensitive data out of bound §5.1	Violation of sensitive data's living-time boundary
		Violation of sensitive data's space(memory) boundary
	Breaking timing guarantees §5.2	Introduction of the time side channel
		Disordered concurrency sequence due to modification of duration
	Introducing micro-architectural side effects §5.3	Introduction of bounds check bypass vulnerability

Elimination of security-relevant code due to undefined behaviour

The C Standard says that if a program has signed integer overflow its behavior is undefined, and the undefined behavior can even precede the overflow.

```
if (password == expected_password)
  allow_superuser_privileges ();
else if (counter++ == INT_MAX)
  abort ();
else
  printf ("%d password mismatches\n", counter);
```

Worse, if an earlier bug in the program lets the compiler deduce that `counter == INT_MAX` or that `counter` previously overflowed, the C standard allows the compiler to optimize away the password test and generate code that allows superuser privileges unconditionally.

- No "semantic obligation": compilers may opportunistically assume that the code following a computation that triggers undefined behaviour can be deleted
- Aggressive optimizations: this deletion may also suppress security checks
- Result: dangerous gaps between programmer intention and code produced

Information leaks through persistent state

- The optimization allowed sensible data to persist more than intended*
- A persistent state security violation is triggered when when data remains in memory beyond the boundaries set by the developers:
  - **Dead store elimination**
  - **Function call inlining**
  - **Code motion**

* Erasure by overwritings: it can be removed by a compiler during optimisation (as there are no later readings), and can also be neutralised by cache mechanisms

Dead store elimination

- A **store** is an assignment to a local variable. A store is **dead** if a value that is assigned is not read in subsequent program statements

```
crypt(){
key = 0xC0DE; // read key
... // work with the secure key
key = 0x0; // scrub memory
}
```

- `key` is not read after that assignment: hence can be eliminated
- **Dead store elimination** improves time and memory use
- Operations that scrub memory of sensitive data such as passwords or cryptographic keys can lead to dead stores
- It is a case of the more general **dead code elimination** issue

Function call inlining

- **Function call inlining** or **inline expansion** replaces a function call site with the body of the function being called to eliminate the call overhead.

```
char *getPWHash() {
// accepts a psw, stores it and computes the hash
}
void compute() {
//is a function using a psw hash
// local variables
long i, j;
char *sha;
// computation
...
//call secure function
sha=getPWHash();
...
}
```

With inlining, the local variables of getPWHash, which contain sensitive information and must live in a secure stack frame, will now be alive for the lifetime of compute()

Function call inlining

Function call inlining

- merges the stack frames of the caller and the callee and avoids executing the prologue and epilogue of the callee
- can affect security-sensitive code: it increases the longevity of variables in a stack frame and can eliminate the boundaries between certain stack frames

- No guarantee that data/variables/memory accessible in the function are secure for the lifetime of the callee

Code motion

Code motion allows the compiler to reorder instructions and basic blocks in the program based on their dependencies

- Code motion affects the layout of stack frames, the liveness of variables, and the timing of individual execution paths through the control flow graph

```
1 // Code before optimization
2 int secret = 0;
3 if (priv)
4     secret = 0xFBADC0DE;
```


Operation to be executed only in in a trusted and secure environment

```
1 // After applying code motion
2 int secret = 0xFBADC0DE;
3 if (!priv)
4     secret = 0;
```

Side channels

As seen before, **side channels** leak information about the state of the system

- Side channel attacks allow an attacker external to the system to observe the internal state of a computation without direct access to the internal state
- Attackers can use timing information to extract information about security operations and sensitive data such as cryptographic keys

- **Processor-specific optimizations** or **microarchitectural features like speculative execution** may help the attacker

Bibliography

- Vijay D'Silva, Mathias Payer, Dawn Song. *The Correctness-Security Gap in Compiler Optimization*. IEEE Symposium on Security and Privacy Workshops 2015
- J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M.s Payer, B. Mao: *Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs*. USENIX Security Symposium 2023: 3655-3672

Outline

Compilation security issues: overview

Dead store elimination

Undefined behaviour and unstable code

Compiler optimization and static analysis

Some optimizations are always valid, e.g.:

- x * 2 --> x + x
- x * 4 --> x << 2

Most others apply, provided some conditions are satisfied:

- x = 4 ! x >> 2 only if x ≥ 0
- x + 1 --> 1 only if x = 0
- if x < y then c1 else c2 → c1 only if x < y
- x := y + 1 → skip only if x unused later

We need a static analysis prior to the actual code transformation

- Static analysis is able to determine some properties valid for all concrete executions of a program

Dead store elimination

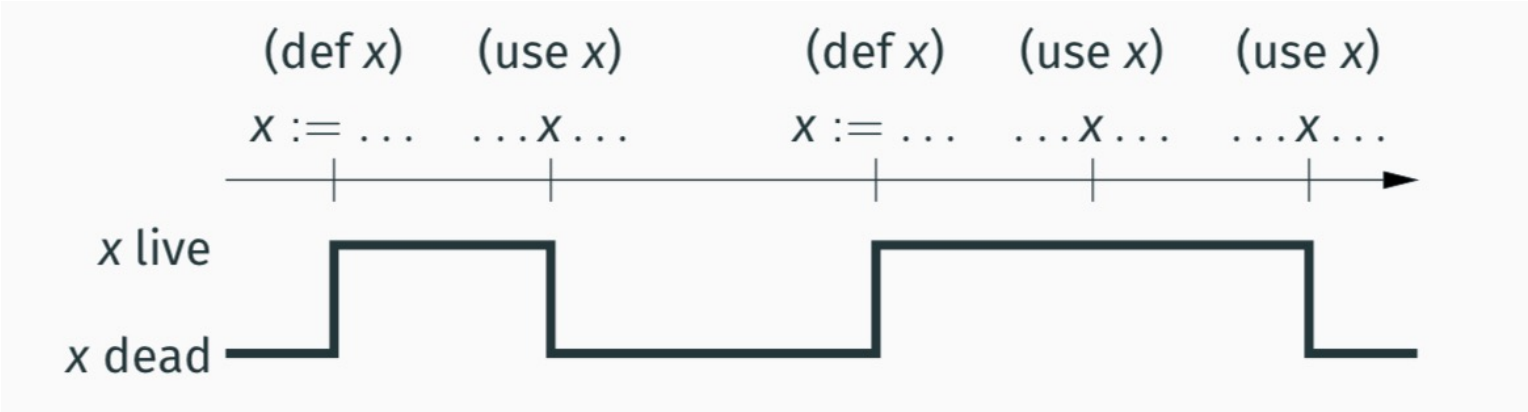
- store `x := 0` becomes skip because x is not live, i.e. used "after" the assignment
- P and Q have the same input-output behavior: the transformation is correct
- The secret password is leaked through the stack in Q: the transformation is insecure
- DSE builds on a static analysis called liveness analysis

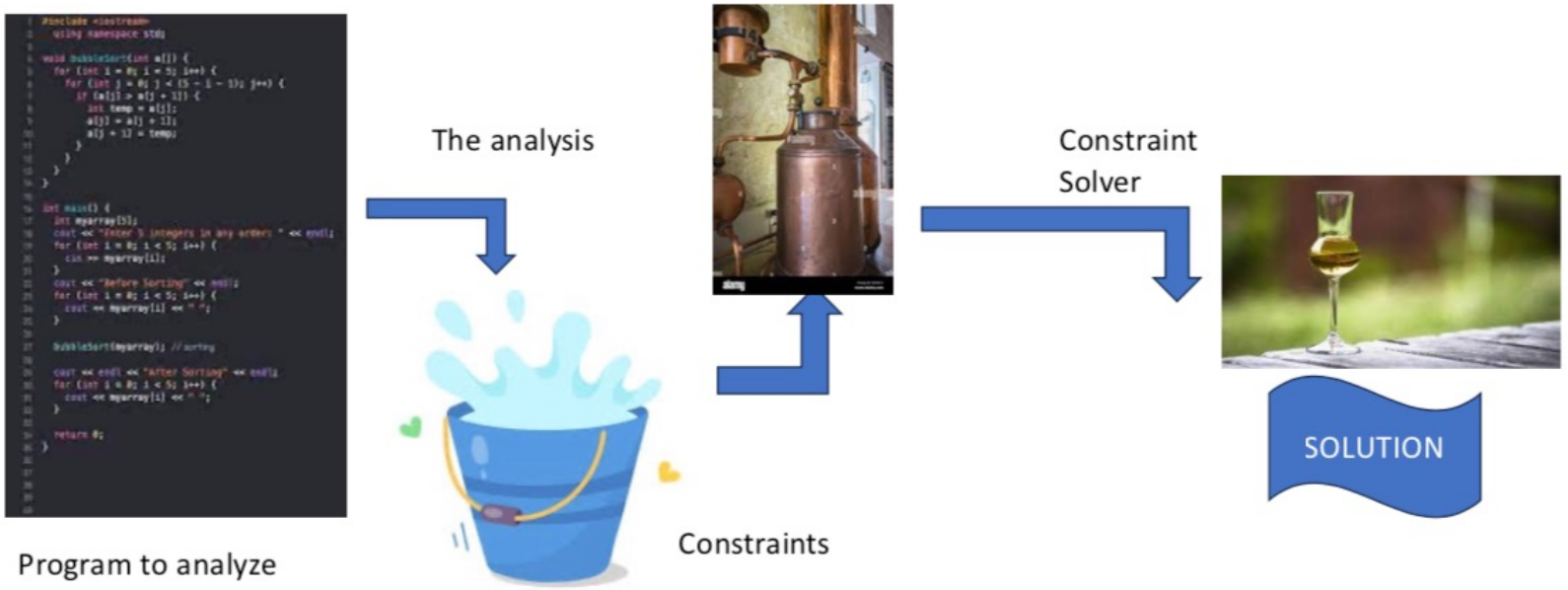
Outline

Live Variables Analysis
Difficulties for security validation
Syntax and Semantics
Post-domination
Information leakage
Axiomatic semantics
A Taint Proof System
Secure dead store elimination algorithm
Conclusions and what next

Live variables analysis

- Dead Store Elimination builds on a static analysis called liveness analysis
- It is a backward (and *may*) dataflow analysis
- A variable x is live at a program point p IFF there exists an execution path where its value is read (used) later with no writes (definitions) to x in between. Undecidable!
- A variable x is dead otherwise: its value has **no impact** on the rest of the program execution!





Liveness analysis

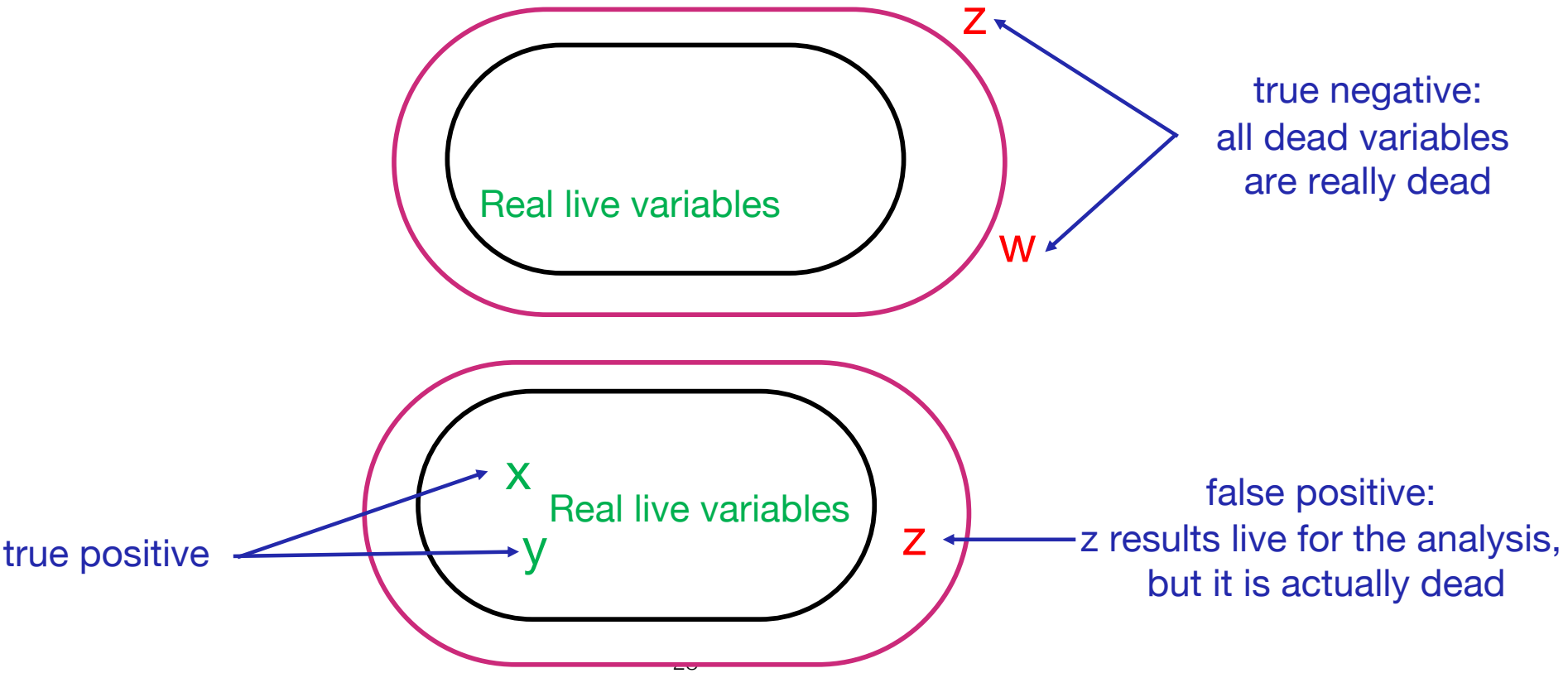
- This **data-flow analysis** determines the set of potentially **live variables in each point** of a given program
- Applications: **remove dead stores** and **dead code**

Liveness analysis*

- We want a conservative analysis: it must only answer “dead” if the variable is really dead
- Optimization: no need to store the values of dead variables
- It is possible to compute a safe over-approximation of liveness
- It is a particular case of the previously mentioned code liveness

* Static Program Analysis - Anders Møller & Michael I. Schwartzbach

Conservative analysis



Example

Are there any dead stores?

```
x = 2;
x = 5;
<< use x >>
```

Example

Are there any dead stores?

```
x = 2;
x = 5;
<< use x >>
```

The value of x is not read (used) before the following assignment and therefore, x is dead

Example

Are there any dead stores?

```
x = 2;
if (y > 0) {
   x = y+5;
} else {
   x = y+10;
}
<< use x >>
```

Example

Are there any dead stores?

```
x = 2;
if (y > 0) {
   x = y+5;
} else {
   x = y+10;
}
<< use x >>
```

The value of x is not read (used) before the following assignment and therefore, x is dead

Example

Are there any dead stores?

```
x = 2;
if (y > 50) {
   x = 1;   // x def
} else {
   x = x+2; // x read
}
<< use x >>
```

The value of x is not read (used) before the following assignment and therefore, x is dead

Example

Are there any dead stores?

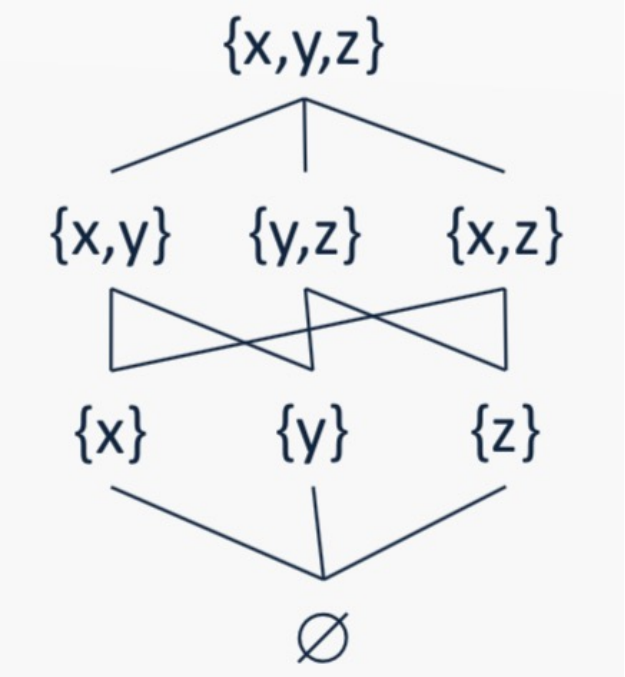
```
x = 2;
if (y > 50) {
   x = 1;   // x def
} else {
   x = x+2; // x read
}
<< use x >>
```

The value of x is read (used) read before one of the following assignments and therefore, x is live

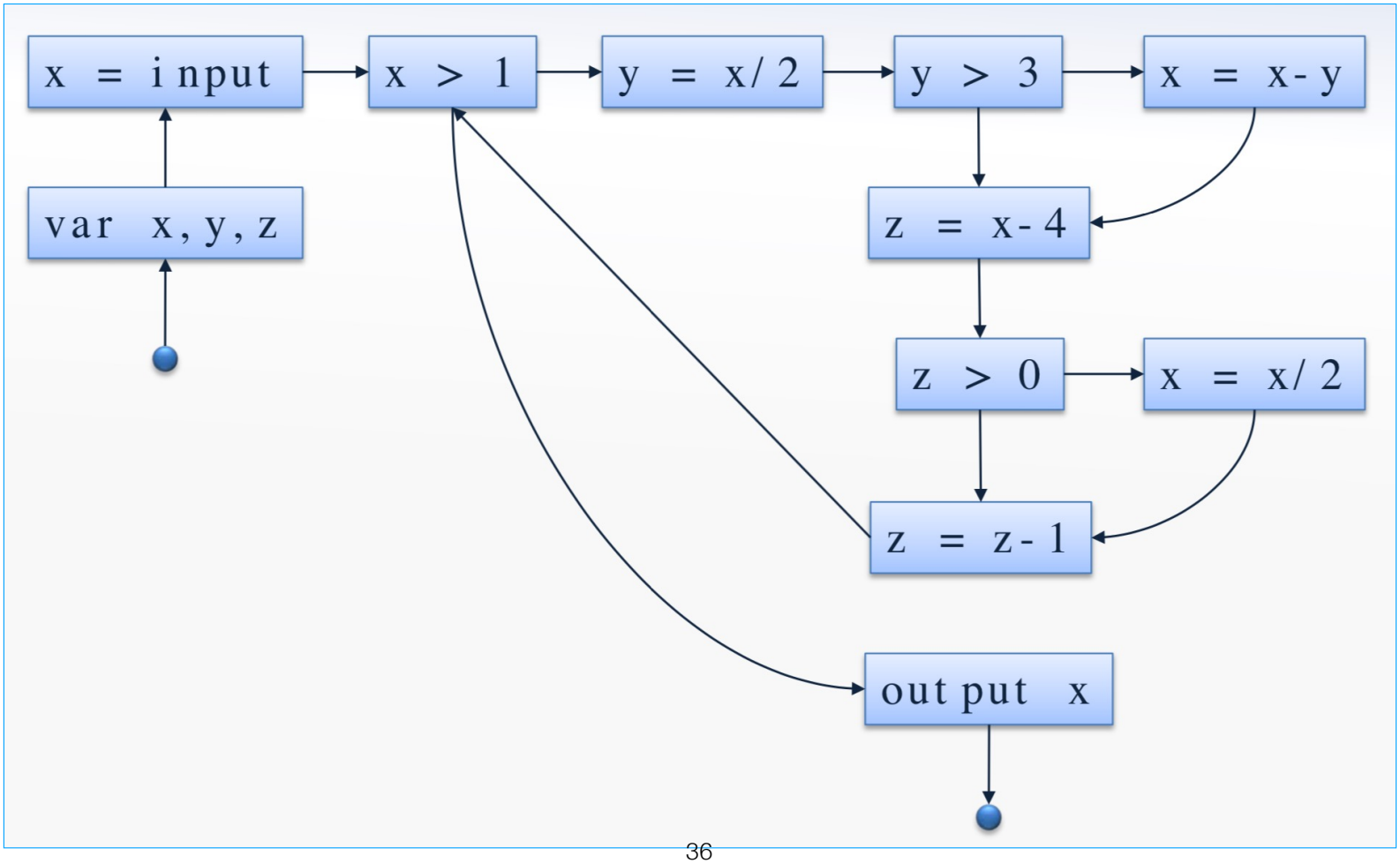
Liveness analysis: the lattice

The lattice modelling our abstract States is **(P({Var});⊆)** with Var, variables in the program

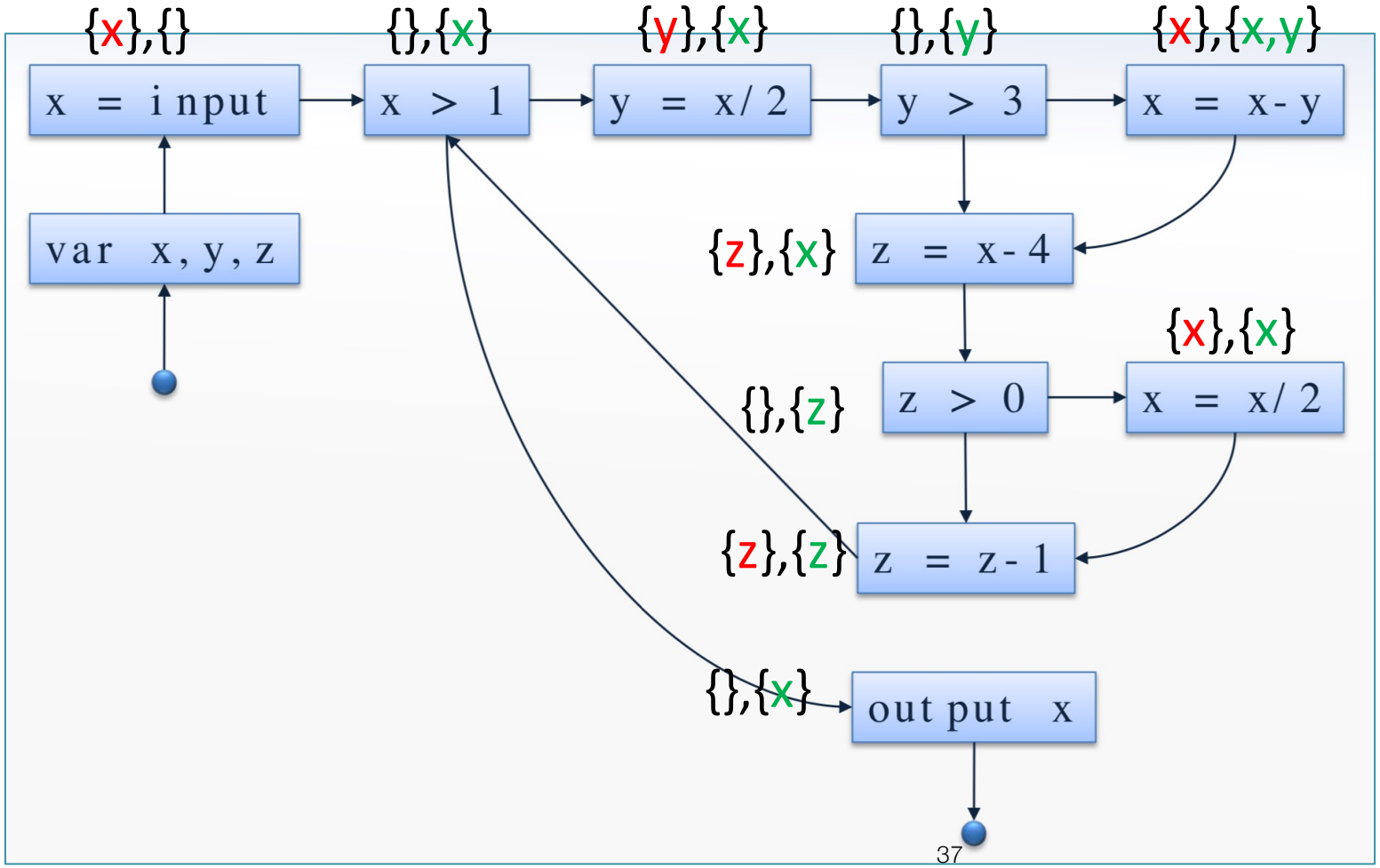
```
var x,y,z;
x = input;
while (x>1) {
y = x/2;
if (y>3) x = x-y;
z = x-4;
if (z>0) x = x/2;
z = z-1;
}
output x;
```



Liveness analysis: CFG

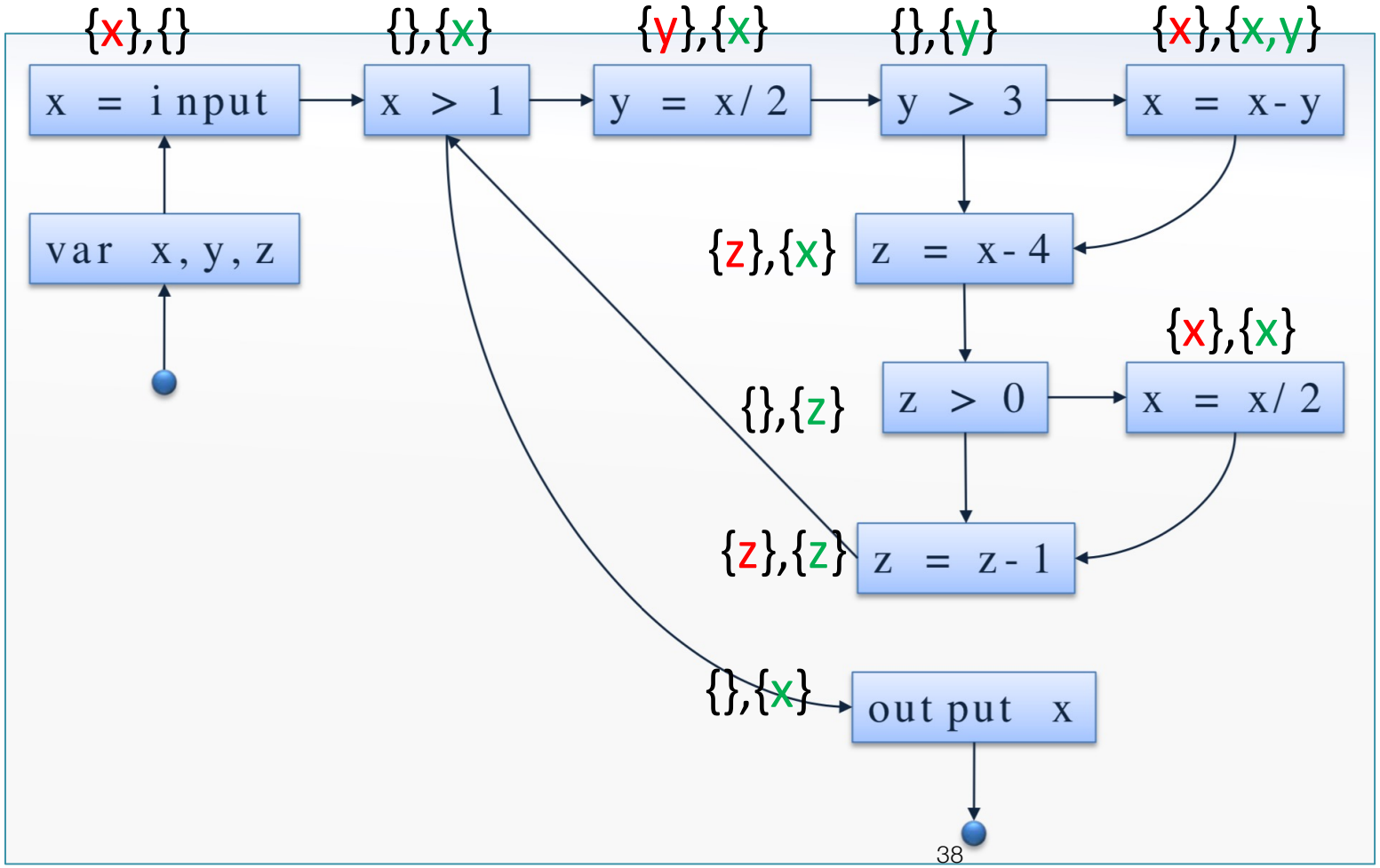


Liveness analysis: CFG



- An instruction makes a variable live when it references
- /uses it
- An instruction makes a variable dead when it defines/assigns to it

Liveness analysis: constraint variables



- For every node **v**
- **[[v]]** constraint variable that denotes the variables that are live at the program point before v

Live Variables Analysis: constraints

Derive a system of equations (equality constraints) with one constraint variable for each program point

$$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$$

- Unlike in sign analysis, JOIN combines abstract states from the successors instead of the predecessors
- It is a backward analysis

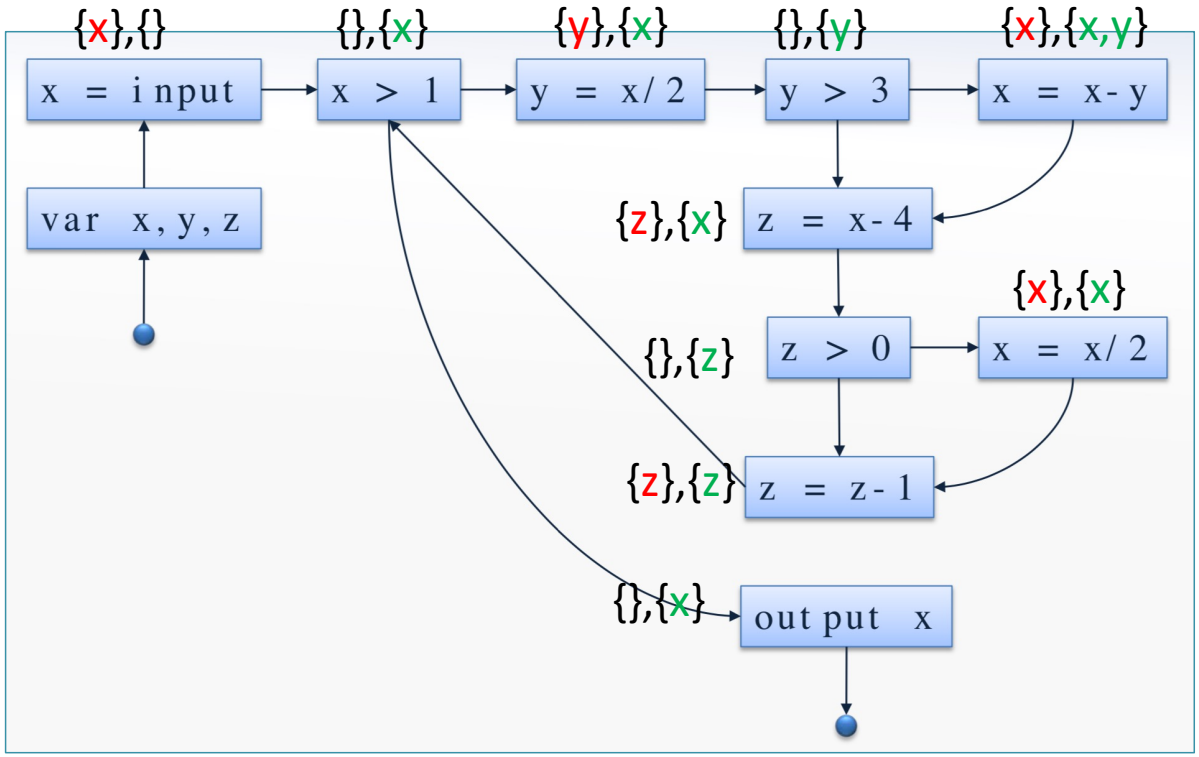
Live Variables Analysis: constraints

- ⟦var x1,...,xn⟧= JOIN(v) \{x1, ..., xn}
- ⟦exit⟧= ∅
- ⟦if (E)⟧= ⟦while (E)⟧ = ⟦output E⟧= JOIN(v) U vars(E) (variables occurring in E)
- ⟦x = E⟧= JOIN(v) \{x} U vars(E)
- ⟦v⟧= JOIN(v) killed variable generated variable

where right-handsides are monotone

Back to our example

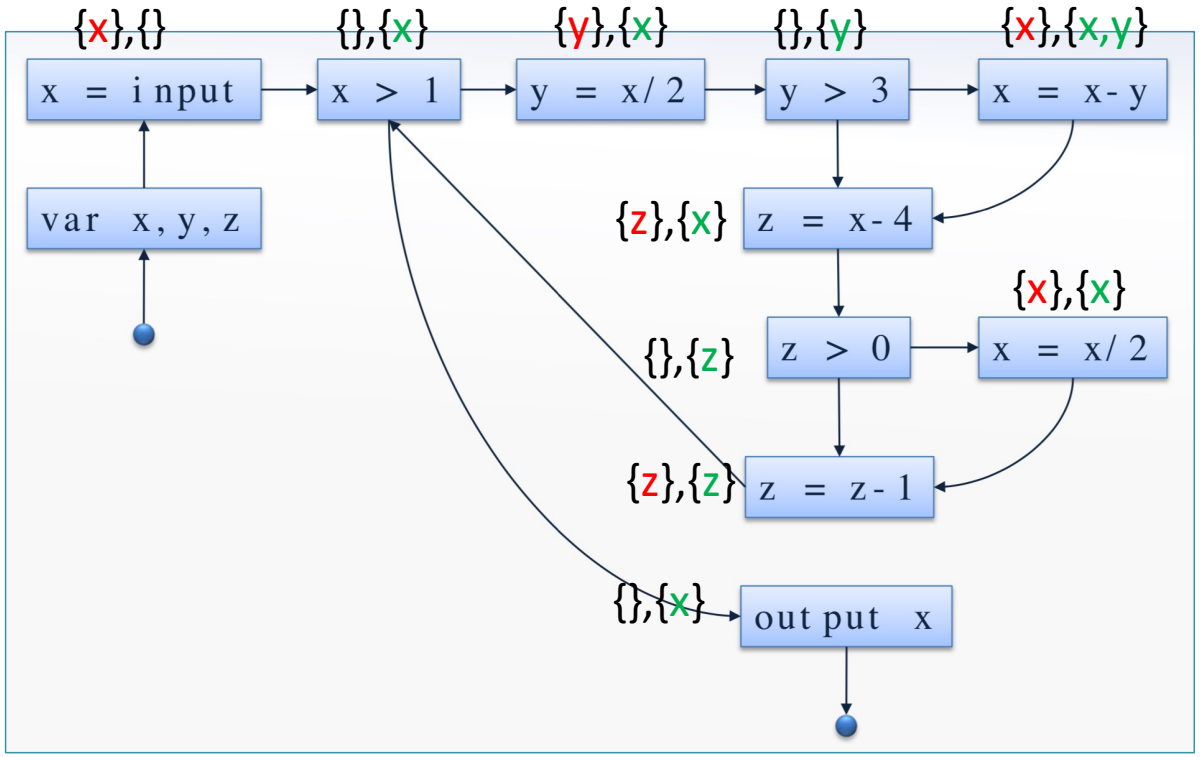
⟦var x1,...,xn⟧= JOIN(v) \{x1, ..., xn}



⟦var x, y, z⟧ = ⟦x=input⟧ \ {x,y,z}

Back to our example

⟦x = E⟧= JOIN(v) \{x} U vars(E)

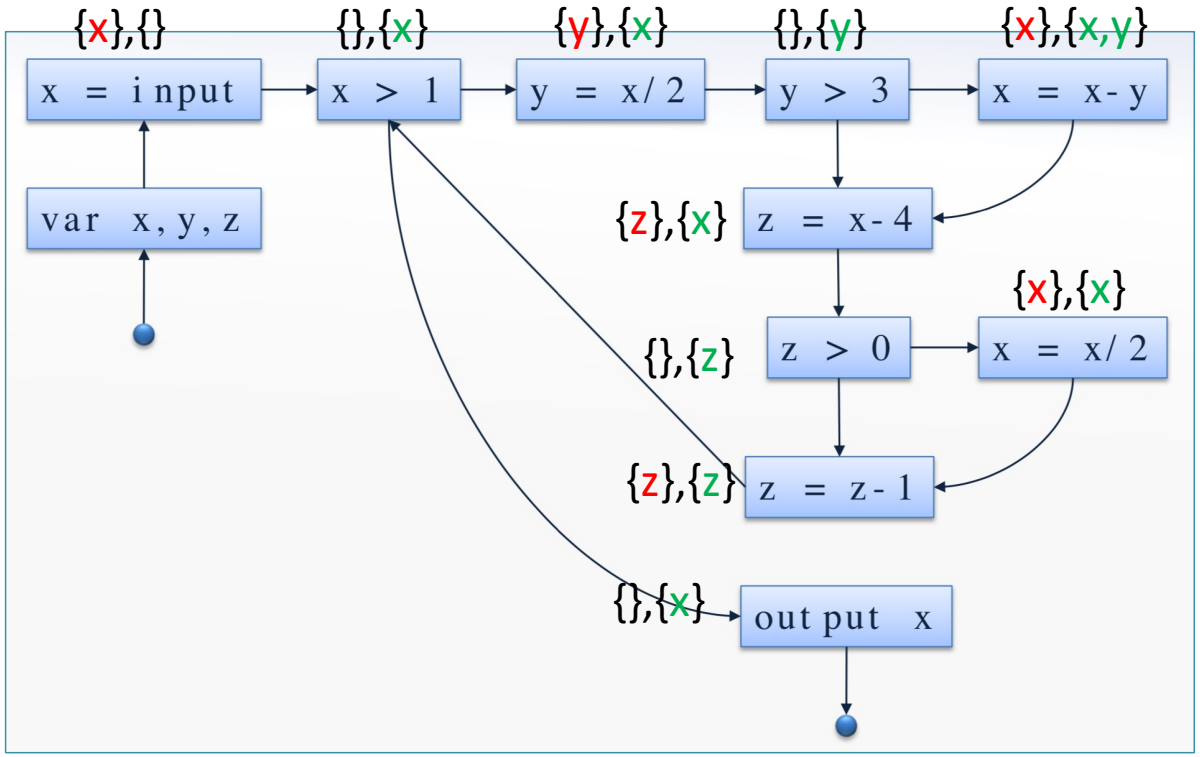


⟦var x, y, z⟧ = ⟦x=input⟧ \ {x,y,z}

⟦x=input⟧ = ⟦x>1⟧ \ {x}

Back to our example

⟦if (E)⟧= JOIN(v) U vars(E)



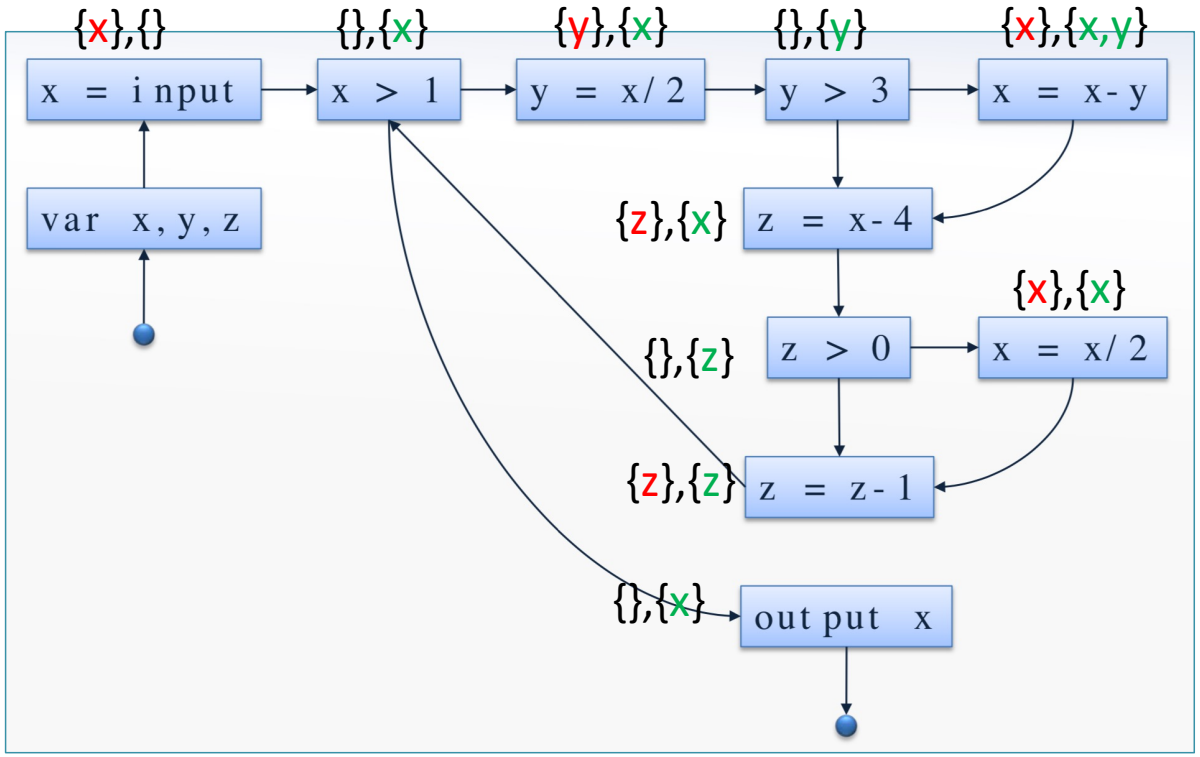
⟦var x, y, z⟧ = ⟦x=i nput ⟧ \ {x,y,z}

⟦x=i nput ⟧ = ⟦x>1⟧ \ {x}

⟦x>1⟧ = (⟦y=x/ 2⟧ ∪ ⟦out put x⟧) ∪ {x}

Back to our example

⟦x = E⟧= JOIN(v) \{x} U vars(E)



⟦var x, y, z⟧ = ⟦x=i nput ⟧ \ {x,y,z}

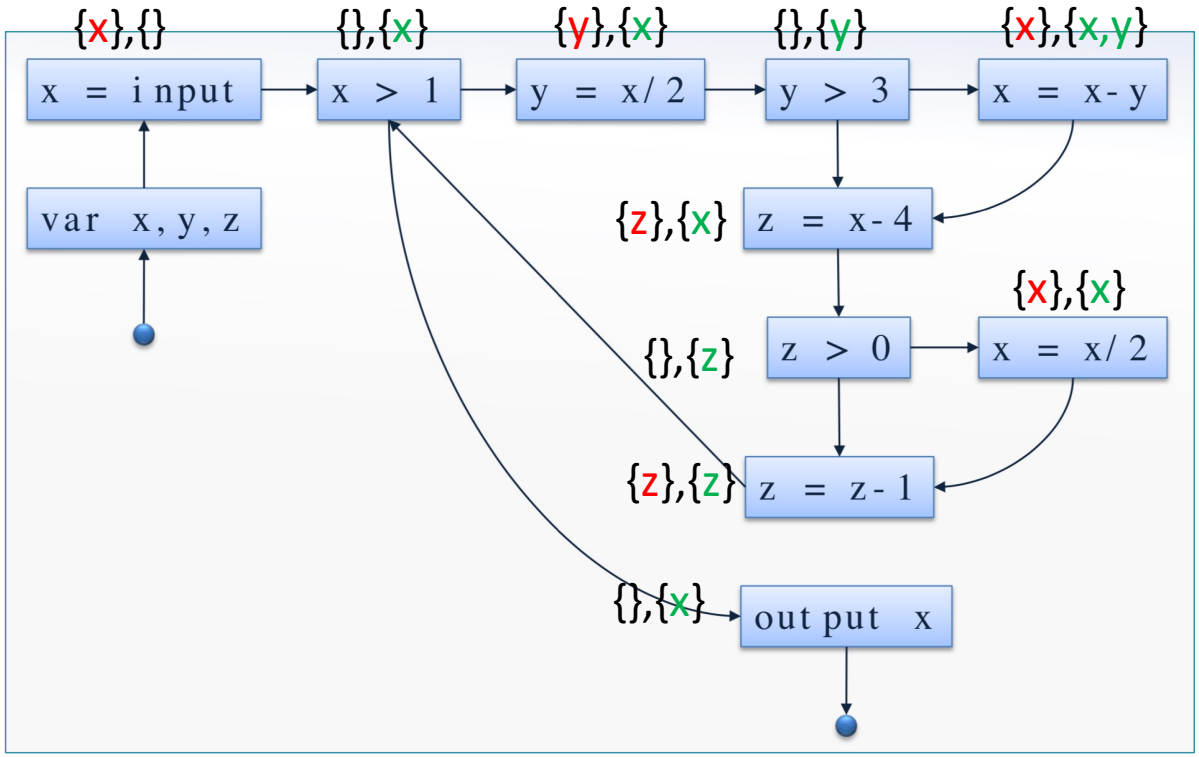
⟦x=i nput ⟧ = ⟦x>1⟧ \ {x}

⟦x>1⟧ = (⟦y=x/ 2⟧ ∪ ⟦out put x⟧) ∪ {x}

⟦y=x/ 2⟧ = (⟦y>3⟧ \ {y}) ∪ {x}

Back to our example

⟦if (E)⟧= JOIN(v) U vars(E)



⟦var x, y, z⟧ = ⟦x=i nput ⟧ \ {x,y,z}

⟦x=i nput ⟧ = ⟦x>1⟧ \ {x}

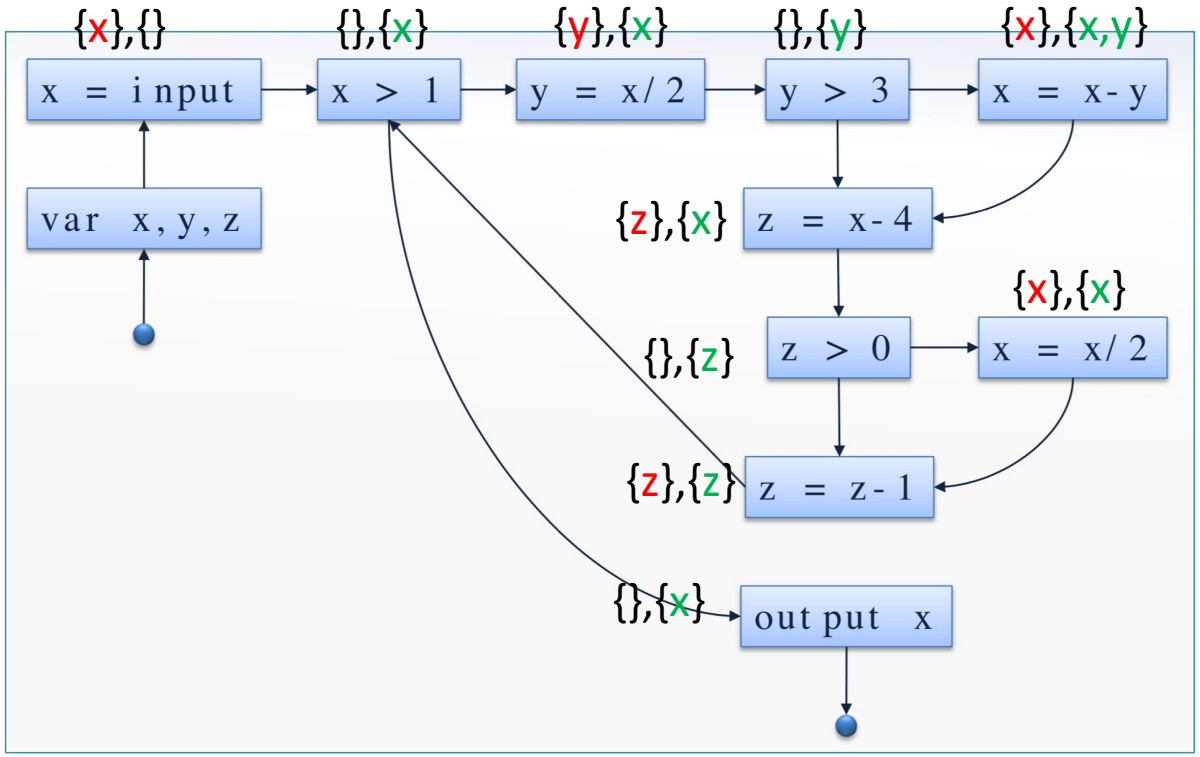
⟦x>1⟧ = (⟦y=x/ 2⟧ ∪ ⟦out put x⟧) ∪ {x}

⟦y=x/ 2⟧ = (⟦y>3⟧ \ {y}) ∪ {x}

⟦y>3⟧ = ⟦x=x- y⟧ ∪ ⟦z=x- 4⟧ ∪ {y}

Back to our example

⟦x = E⟧= JOIN(v) \{x} U vars(E)



⟦var x, y, z⟧ = ⟦x=i nput ⟧ \ {x,y,z}

⟦x=i nput ⟧ = ⟦x>1⟧ \ {x}

⟦x>1⟧ = (⟦y=x/ 2⟧ ∪ ⟦out put x⟧) ∪ {x}

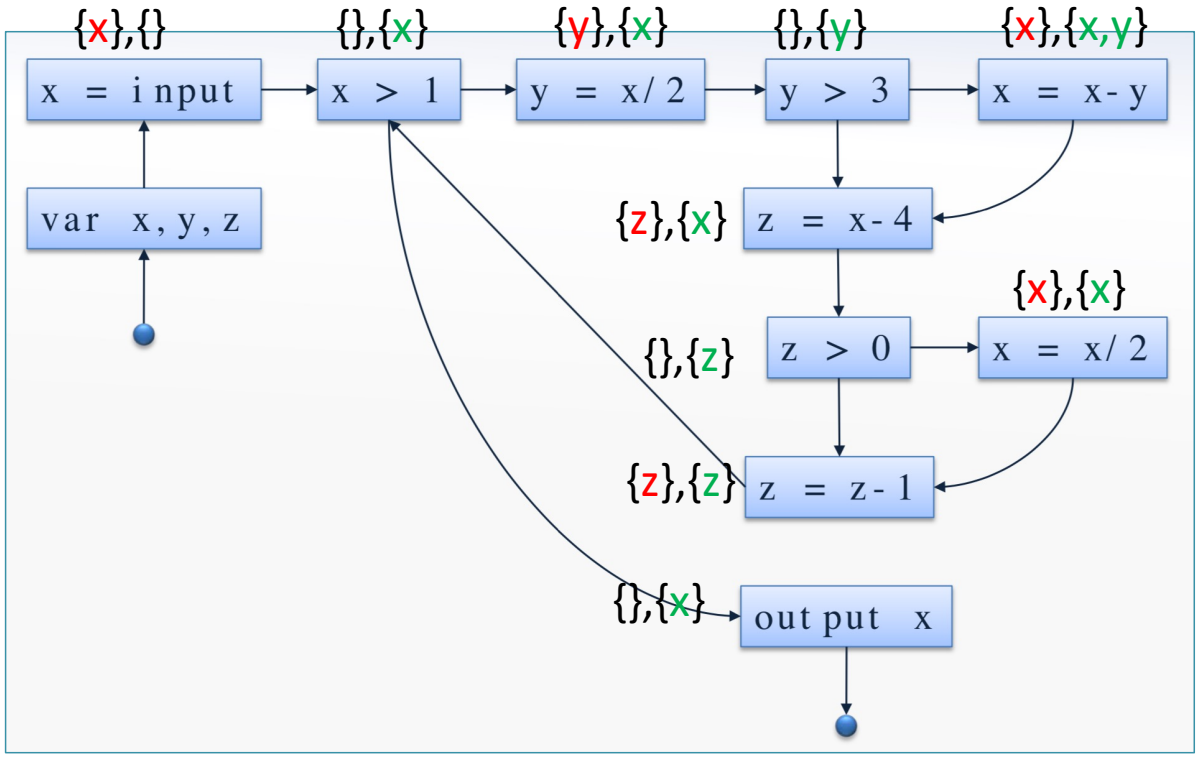
⟦y=x/ 2⟧ = (⟦y>3⟧ \ {y}) ∪ {x}

⟦y>3⟧ = ⟦x=x- y⟧ ∪ ⟦z=x- 4⟧ ∪ {y}

⟦x=x- y⟧ = (⟦z=x- 4⟧ \ {x}) ∪ {x,y}

Back to our example

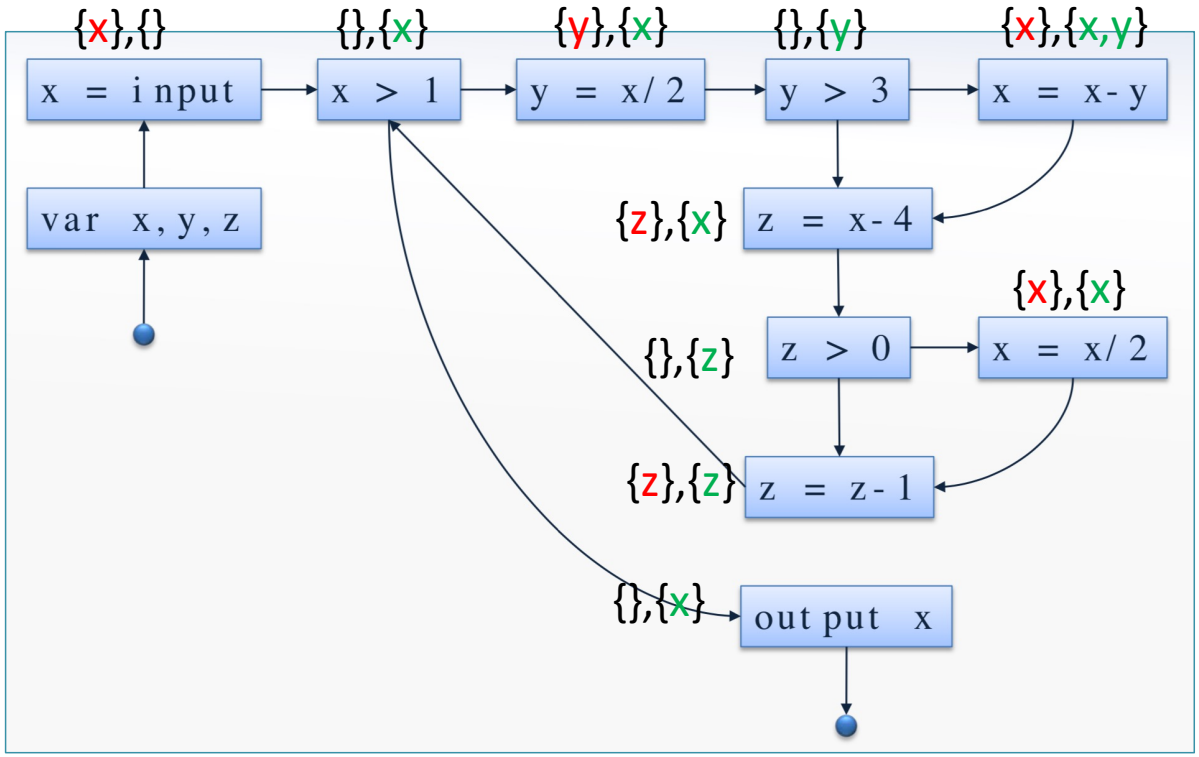
⟦x = E⟧= JOIN(v) \{x} U vars(E)



⟦var x, y, z⟧ = ⟦x=i nput ⟧ \ {x,y,z}
⟦x=i nput ⟧ = ⟦x>1⟧ \ {x}
⟦x>1⟧ = (⟦y=x/ 2⟧ ∪ ⟦out put x⟧) ∪ {x}
⟦y=x/ 2⟧ = (⟦y>3⟧ \ {y}) ∪ {x}
⟦y>3⟧ = ⟦x=x- y⟧ ∪ ⟦z=x- 4⟧ ∪ {y}
⟦x=x- y⟧ = (⟦z=x- 4⟧ \ {x}) ∪ {x,y}
⟦z=x- 4⟧ = (⟦z>0⟧ \ {z}) ∪ {x}

Back to our example

⟦if (E)⟧= JOIN(v) U vars(E)



⟦var x, y, z⟧ = ⟦x=i nput ⟧ \ {x,y,z}

⟦x=i nput ⟧ = ⟦x>1⟧ \ {x}

⟦x>1⟧ = (⟦y=x/ 2⟧ ∪ ⟦out put x⟧) ∪ {x}

⟦y=x/ 2⟧ = (⟦y>3⟧ \ {y}) ∪ {x}

⟦y>3⟧ = ⟦x=x- y⟧ ∪ ⟦z=x- 4⟧ ∪ {y}

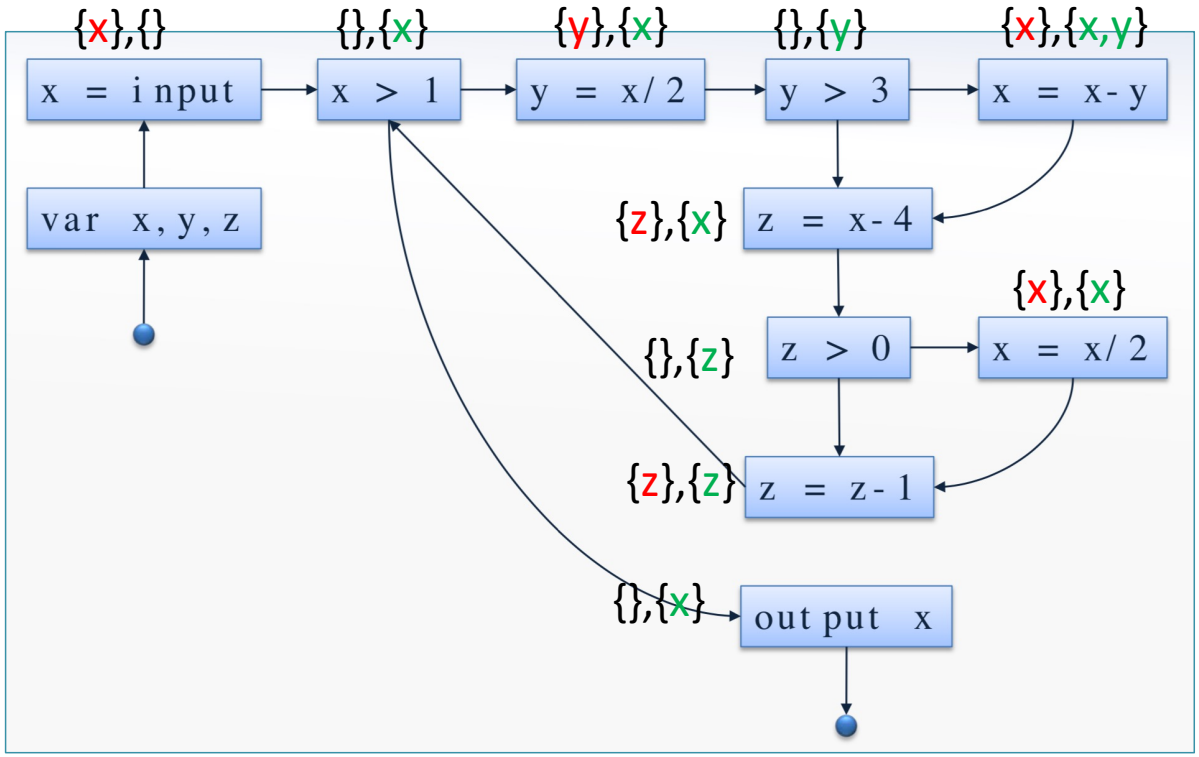
⟦x=x- y⟧ = (⟦z=x- 4⟧ \ {x}) ∪ {x,y}

⟦z=x- 4⟧ = (⟦z>0⟧ \ {z}) ∪ {x}

⟦z>0⟧ = ⟦x=x/ 2⟧ ∪ ⟦z=z- 1⟧ ∪ {z}

Back to our example

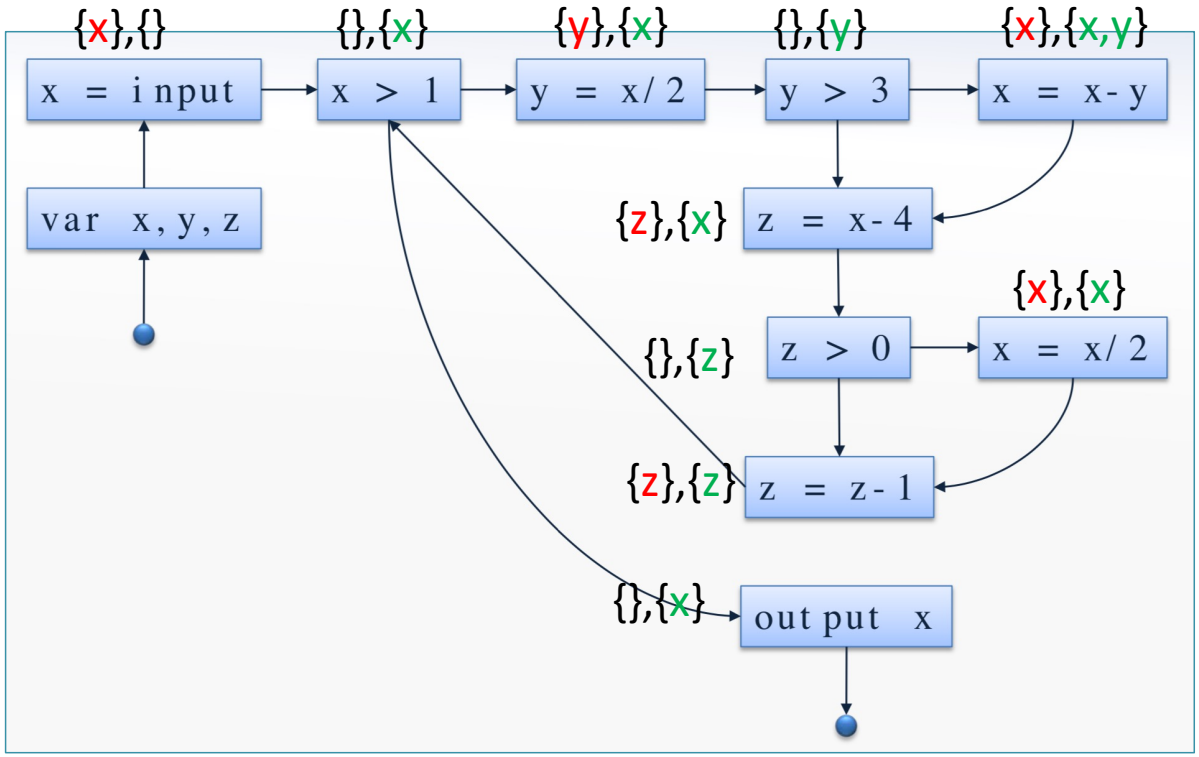
⟦x = E⟧= JOIN(v) \{x} U vars(E)



⟦var x, y, z⟧ = ⟦x=i nput ⟧ \ {x,y,z}
⟦x=i nput ⟧ = ⟦x>1⟧ \ {x}
⟦x>1⟧ = (⟦y=x/ 2⟧ ∪ ⟦out put x⟧) ∪ {x}
⟦y=x/ 2⟧ = (⟦y>3⟧ \ {y}) ∪ {x}
⟦y>3⟧ = ⟦x=x- y⟧ ∪ ⟦z=x- 4⟧ ∪ {y}
⟦x=x- y⟧ = (⟦z=x- 4⟧ \ {x}) ∪ {x,y}
⟦z=x- 4⟧ = (⟦z>0⟧ \ {z}) ∪ {x}
⟦z>0⟧ = ⟦x=x/ 2⟧ ∪ ⟦z=z- 1⟧ ∪ {z}
⟦x=x/ 2⟧ = (⟦z=z- 1⟧ \ {x}) ∪ {x}

Back to our example

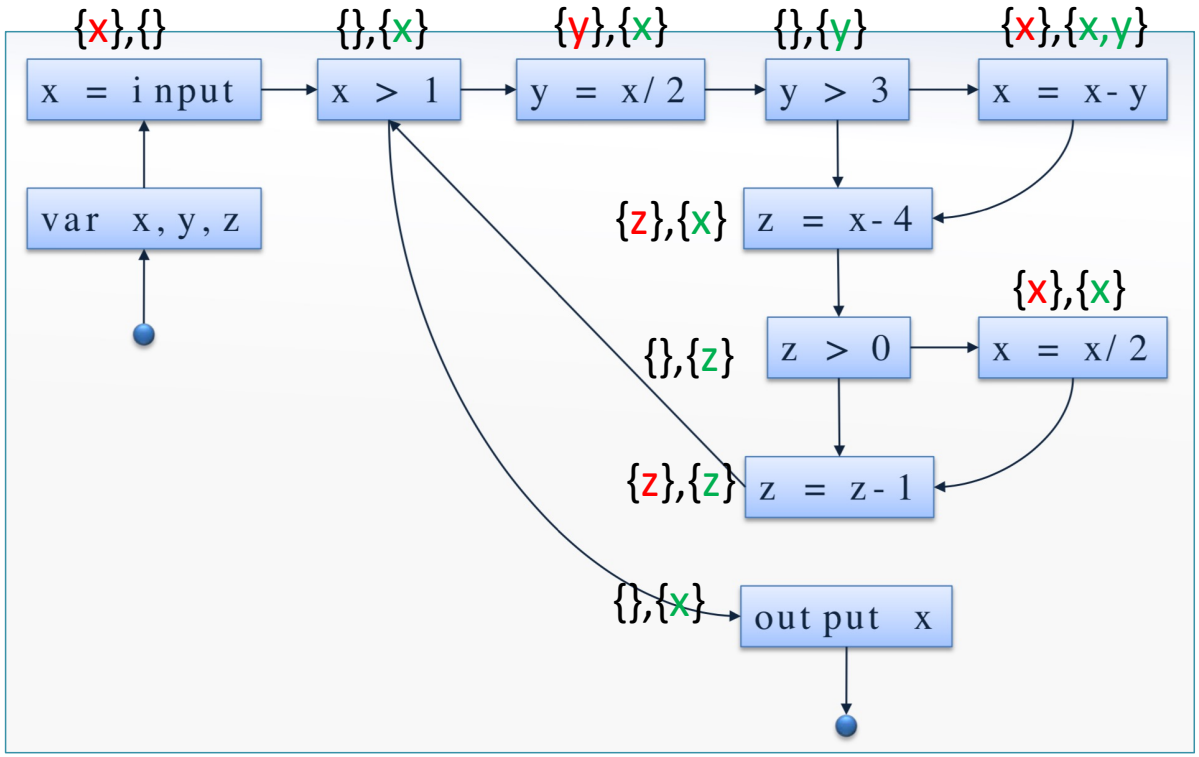
⟦x = E⟧= JOIN(v) \{x} U vars(E)



⟦var x, y, z⟧ = ⟦x=i nput ⟧ \ {x,y,z}
⟦x=i nput ⟧ = ⟦x>1⟧ \ {x}
⟦x>1⟧ = (⟦y=x/ 2⟧ ∪ ⟦out put x⟧) ∪ {x}
⟦y=x/ 2⟧ = (⟦y>3⟧ \ {y}) ∪ {x}
⟦y>3⟧ = ⟦x=x- y⟧ ∪ ⟦z=x- 4⟧ ∪ {y}
⟦x=x- y⟧ = (⟦z=x- 4⟧ \ {x}) ∪ {x,y}
⟦z=x- 4⟧ = (⟦z>0⟧ \ {z}) ∪ {x}
⟦z>0⟧ = ⟦x=x/ 2⟧ ∪ ⟦z=z- 1⟧ ∪ {z}
⟦x=x/ 2⟧ = (⟦z=z- 1⟧ \ {x}) ∪ {x}
⟦z=z- 1⟧ = (⟦x>1⟧ \ {z}) ∪ {z}

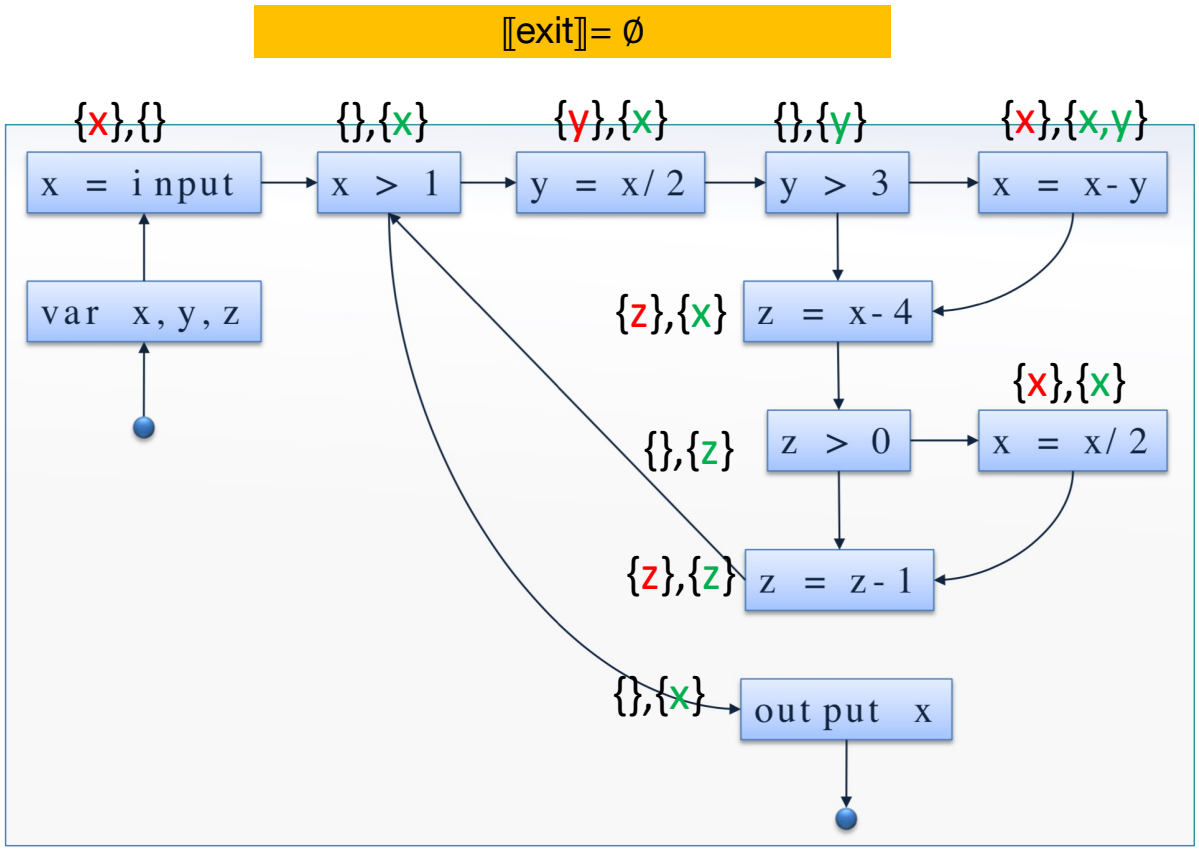
Back to our example

⟦output E⟧= JOIN(v) U vars(E)



⟦var x, y, z⟧ = ⟦x=i nput ⟧ \ {x,y,z}
⟦x=i nput ⟧ = ⟦x>1⟧ \ {x}
⟦x>1⟧ = (⟦y=x/ 2⟧ ∪ ⟦out put x⟧) ∪ {x}
⟦y=x/ 2⟧ = (⟦y>3⟧ \ {y}) ∪ {x}
⟦y>3⟧ = ⟦x=x- y⟧ ∪ ⟦z=x- 4⟧ ∪ {y}
⟦x=x- y⟧ = (⟦z=x- 4⟧ \ {x}) ∪ {x,y}
⟦z=x- 4⟧ = (⟦z>0⟧ \ {z}) ∪ {x}
⟦z>0⟧ = ⟦x=x/ 2⟧ ∪ ⟦z=z- 1⟧ ∪ {z}
⟦x=x/ 2⟧ = (⟦z=z- 1⟧ \ {x}) ∪ {x}
⟦z=z- 1⟧ = (⟦x>1⟧ \ {z}) ∪ {z}
⟦out put x⟧ = ⟦*exit*⟧ ∪ {x}

Back to our example



$[\![\text{var } x, y, z]\!] = [\![x=\text{input}]\!] \setminus \{x,y,z\}$

$[\![x=\text{input}]\!] = [\![x>1]\!] \setminus \{x\}$

$[\![x>1]\!] = ([\![y=x/2]\!] \cup [\![\text{out put } x]\!]) \cup \{x\}$

$[\![y=x/2]\!] = ([\![y>3]\!] \setminus \{y\}) \cup \{x\}$

$[\![y>3]\!] = [\![x=x-y]\!] \cup [\![z=x-4]\!] \cup \{y\}$

$[\![x=x-y]\!] = ([\![z=x-4]\!] \setminus \{x\}) \cup \{x,y\}$

$[\![z=x-4]\!] = ([\![z>0]\!] \setminus \{z\}) \cup \{x\}$

$[\![z>0]\!] = [\![x=x/2]\!] \cup [\![z=z-1]\!] \cup \{z\}$

$[\![x=x/2]\!] = ([\![z=z-1]\!] \setminus \{x\}) \cup \{x\}$

$[\![z=z-1]\!] = ([\![x>1]\!] \setminus \{z\}) \cup \{z\}$

$[\![\text{out put } x]\!] = [\![exit]\!] \cup \{x\}$

$[\![exit]\!] = \varnothing$

Back to our example

[[entry]] = Ø
[[var x,y,z]] = [[x=input]] \ {x, y, z}
[[x=input]] = [[x>1]] \ {x}
[[x>1]] = [[y=x/2]] ∪ [[output x]] ∪ {x}
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x}
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y}
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y}
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x}
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z}
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x}
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z}
[[output x]] = [[exit]] ∪ {x}
[[exit]] = Ø

Back to our example: 1° iteration

[[entry]] = [[var x,y,z]] = ∅
[[var x,y,z]] = [[x=input]] \ {x, y, z} = ∅
[[x=input]] = [[x>1]] \ {x} = ∅
[[x>1]] = [[y=x/2]] ∪ [[output x]] ∪ {x} = ∅
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x} = ∅
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y} = ∅
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y} = ∅
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x} = ∅
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z} = ∅
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x} = ∅
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z} = ∅
[[exit]] = ∅

1° iteration
begin with JOIN(v) = ∅
for each v

- The sequence
  ∅, F(∅), F(F(∅)), …, F^n(∅), …
  converges to the smallest fixpoint of F
- There are smarter algorithms
  but this one works

Back to our example: 1° iteration

[[entry]] = [[var x,y,z]] = ∅
[[var x,y,z]] = [[x=input]] \ {x, y, z} = ∅
[[x=input]] = [[x>1]] \ {x} = ∅
[[x>1]] = [[y=x/2]] ∪ [[output x]] ∪ {x} = ∅
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x} = ∅
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y} = ∅
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y} = ∅
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x} = ∅
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z} = ∅
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x} = ∅
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z} = ∅
[[exit]] = ∅

1° iteration
begin with JOIN(v) = ∅
for each v

Back to our example: 2° iteration

[[entry]] = [[var x,y,z]] = ∅
[[var x,y,z]] = [[x=input]] \ {x, y, z} = ∅ \ {x, y, z}
[[x=input]] = [[x>1]] \ {x} = ∅ \ {x}
[[x>1]] = [[y=x/2]] ∪ [[output x]] ∪ {x} = ∅ ∪ ∅ ∪ {x}
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x} = (∅ \ {y}) ∪ {x}
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y} = ∅ ∪ ∅ ∪ {y}
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y} = ∅ \ {x} ∪ {x,y}
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x} = (∅ \ {z}) ∪ {x}
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z} = ∅ ∪ ∅ ∪ {z}
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x} = (∅ \ {x}) ∪ {x}
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z} = (∅ \ {z}) ∪ {z}
[[output x]] = [[exit]] ∪ {x} = ∅ ∪ {x}
[[exit]] = ∅

2° iteration
begin with the values obtained in the previous iterations for each v

Back to our example: 2° iteration

[[entry]] = [[var x,y,z]] = ∅ = ∅
[[var x,y,z]] = [[x=input]] \ {x, y, z} = ∅ \ {x, y, z} = ∅
[[x=input]] = [[x>1]] \ {x} = ∅ \ {x} = ∅
[[x>1]] = [[y=x/2]] ∪ [[output x]] ∪ {x} = ∅ ∪ ∅ ∪ {x} = {x}
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x} = (∅ \ {y}) ∪ {x} = {x}
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y} = ∅ ∪ ∅ ∪ {y} = {y}
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y} = ∅ \ {x} ∪ {x,y} = {x,y}
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x} = (∅ \ {z}) ∪ {x} = {x}
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z} = ∅ ∪ ∅ ∪ {z} = {z}
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x} = (∅ \ {x}) ∪ {x} = {x}
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z} = (∅ \ {z}) ∪ {z} = {z}
[[output x]] = [[exit]] ∪ {x} = ∅ ∪ {x} = {x}
[[exit]] = ∅ = ∅

2° iteration
begin with the values obtained in the previous iterations

Back to our example: 3° iteration

[[entry]] = [[var x,y,z]] = ∅ = ∅
[[var x,y,z]] = [[x=input]] \ {x, y, z} = ∅ \ {x, y, z} = ∅
[[x=input]] = [[x>1]] \ {x} = {x} \ {x} = ∅ \ {x} = ∅
[[x>1]] = [[y=x/2]] ∪ [[output x]] ∪ {x} = {x} ∪ {x} ∪ {x} = {x}
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x} = ({y} \ {y}) ∪ {x} = {x}
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y} = {x,y} ∪ {x} ∪ {y} = {x,y}
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y} = ({x} \ {x}) ∪ {x,y} = {x,y}
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x} = ({z} \ {z}) ∪ {x} = {x}
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z} = {x} ∪ {z} ∪ {z} = {x,z}
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x} = ({x,z} \ {x}) ∪ {x} = {x,z}
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z} = ({x} \ {z}) ∪ {z} = {x,z}
[[output x]] = [[exit]] ∪ {x} = ∅ ∪ {x} = {x}
[[exit]] = ∅ = ∅

3° iteration
begin with the values obtained in the previous iterations

Back to our example: 3° iteration

[[entry]] = [[var x,y,z]] = ∅ = ∅
[[var x,y,z]] = [[x=input]] \ {x, y, z} = ∅ \ {x, y, z} = ∅
[[x=input]] = [[x>1]] \ {x} = {x} \ {x} = ∅
[[x>1]] = [[y=x/2]] ∪ [[output x]] ∪ {x} = {x} ∪ {x} ∪ {x} = {x}
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x} = ({x,y} \ {y}) ∪ {x} = {x,y}
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y} = {x,y} ∪ {x} ∪ {y} = {x,y}
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y} = ({x} \ {x}) ∪ {x,y} = {x,y}
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x} = ({x,z} \ {z}) ∪ {x} = {x,z}
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z} = {x} ∪ {z} ∪ {z} = {x,z}
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x} = ({x,z} \ {x}) ∪ {x} = {x,z}
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z} = ({x} \ {z}) ∪ {z} = {x,z}
[[output x]] = [[exit]] ∪ {x} = ∅ ∪ {x} = {x}
[[exit]] = ∅ = ∅

3° iteration
begin with the values obtained in the previous iterations

Back to our example: fix point

[[entry]] = [[var x,y,z]] = ∅ = Ø
[[var x,y,z]] = [[x=input]] \ {x, y, z} = ∅ \ {x, y, z} = Ø
[[x=input]] = [[x>1]] \ {x} = {x} \ {x} = ∅
[[x>1]] = [[y=x/2]] ∪ [[output x]] ∪ {x} = {x} ∪ {x} ∪ {x} = {x}
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x} = ({x,y} \ {y}) ∪ {x} = {x}
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y} = {x,y} ∪ {x} ∪ {y} = {x,y}
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y} = ({x,z} \ {x}) ∪ {x,y} = {x,y}
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x} = ({x,z} \ {z}) ∪ {x} = {x}
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z} = {x,z} ∪ {x,z} ∪ {z} = {x,z}
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x} = ({x,z} \ {x}) ∪ {x} = {x,z}
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z} = ({x} \ {z}) ∪ {z} = {x,z}
[[output x]] = [[exit]] ∪ {x} = ∅ ∪ {x} = {x}
[[exit]] = ∅ = Ø

4° iteration
values do not change: fixpoint

Back to our example: table

NODE	1° ITERATION	2° ITERATION	3° ITERATION	4° ITERATION
[[entry]] = ∅	∅	∅	∅	∅
[[var x,y,z]] = **[[x=input]]** \ {x, y, z}	∅	∅	∅	∅
[[x=input]] = [[x>1]] \ {x}	∅	∅	∅	∅
[[x>1]] = [[y=x/2]] ∪ [[output x]] ∪ {x}	∅	**{x}**	{x}	{x}
[[y=x/2]] = ([[y>3]] \ {y}) ∪ {x}	∅	**{x}**	{x}	{x}
[[y>3]] = [[x=x-y]] ∪ [[z=x-4]] ∪ {y}	∅	**{y}**	**{x,y}**	{x,y}
[[x=x-y]] = ([[z=x-4]] \ {x}) ∪ {x,y}	∅	**{x,y}**	{x,y}	{x,y}
[[z=x-4]] = ([[z>0]] \ {z}) ∪ {x}	∅	**{x}**	{x}	{x}
[[z>0]] = [[x=x/2]] ∪ [[z=z-1]] ∪ {z}	∅	**{z}**	**{x,z}**	{x,z}
[[x=x/2]] = ([[z=z-1]] \ {x}) ∪ {x}	∅	**{x}**	**{x,z}**	{x,z}
[[z=z-1]] = ([[x>1]] \ {z}) ∪ {z}	∅	**{z}**	**{x,z}**	{x,z}
[[output x]] = [[exit]] ∪ {x}	∅	**{x}**	{x}	{x}
[[exit]] = ∅	∅	∅	∅	∅

Fix-point

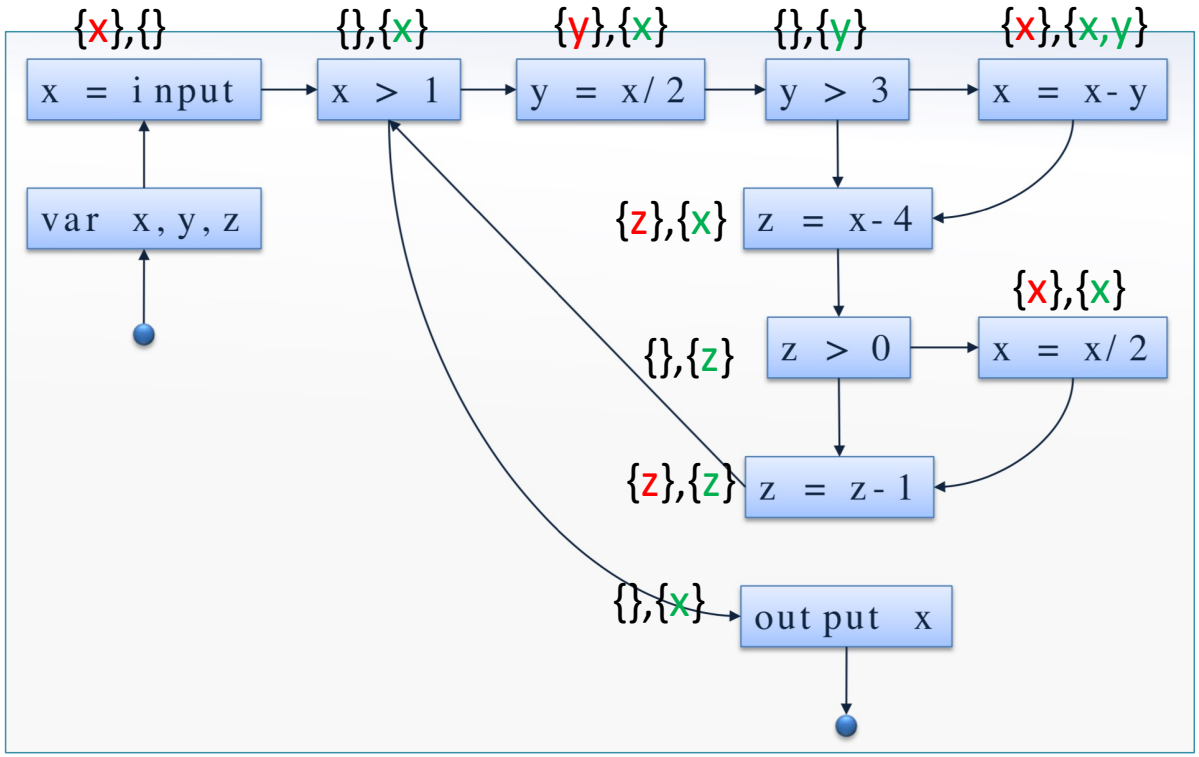
Back to our example

$$[\![\text{var } x, y, z]\!] = [\![x = \text{input}]\!] \setminus \{x,y,z\}$$
$$[\![x = \text{input}]\!] = [\![x > 1]\!] \setminus \{x\}$$
$$[\![x > 1]\!] = ([\![y = x/2]\!] \cup [\![\text{output } x]\!]) \cup \{x\}$$
$$[\![y = x/2]\!] = ([\![y > 3]\!] \setminus \{y\}) \cup \{x\}$$
$$[\![y > 3]\!] = [\![x = x - y]\!] \cup [\![z = x - 4]\!] \cup \{y\}$$
$$[\![x = x - y]\!] = ([\![z = x - 4]\!] \setminus \{x\}) \cup \{x,y\}$$
$$[\![z = x - 4]\!] = ([\![z > 0]\!] \setminus \{z\}) \cup \{x\}$$
$$[\![z > 0]\!] = [\![x = x/2]\!] \cup [\![z = z - 1]\!] \cup \{z\}$$
$$[\![x = x/2]\!] = ([\![z = z - 1]\!] \setminus \{x\}) \cup \{x\}$$
$$[\![z = z - 1]\!] = ([\![x > 1]\!] \setminus \{z\}) \cup \{z\}$$
$$[\![\text{output } x]\!] = [\![\mathit{exit}]\!] \cup \{x\}$$
$$[\![\mathit{exit}]\!] = \varnothing$$

$$[\![\mathit{entry}]\!] = \varnothing$$
$$[\![\text{var } x, y, z]\!] = \varnothing$$
$$[\![x = \text{input}]\!] = \varnothing$$
$$[\![x > 1]\!] = \{x\}$$
$$[\![y = x/2]\!] = \{x\}$$
$$[\![y > 3]\!] = \{x,y\}$$
$$[\![x = x - y]\!] = \{x,y\}$$
$$[\![z = x - 4]\!] = \{x\}$$

$$[\![z > 0]\!] = \{x,z\}$$
$$[\![x = x/2]\!] = \{x,z\}$$
$$[\![z = z - 1]\!] = \{x,z\}$$
$$[\![\text{output } x]\!] = \{x\}$$
$$[\![\mathit{exit}]\!] = \varnothing$$

Solution



$[\![entry]\!] = \varnothing$

$[\![var\ \ x, y, z]\!] = \varnothing$

$[\![x=input]\!] = \varnothing$

$[\![x>1]\!] = \{x\}$

$[\![y=x/2]\!] = \{x\}$

$[\![y>3]\!] = \{x,y\}$

$[\![x=x-y]\!] = \{x,y\}$

$[\![z=x-4]\!] = \{x\}$

$[\![z>0]\!] = \{x,z\}$

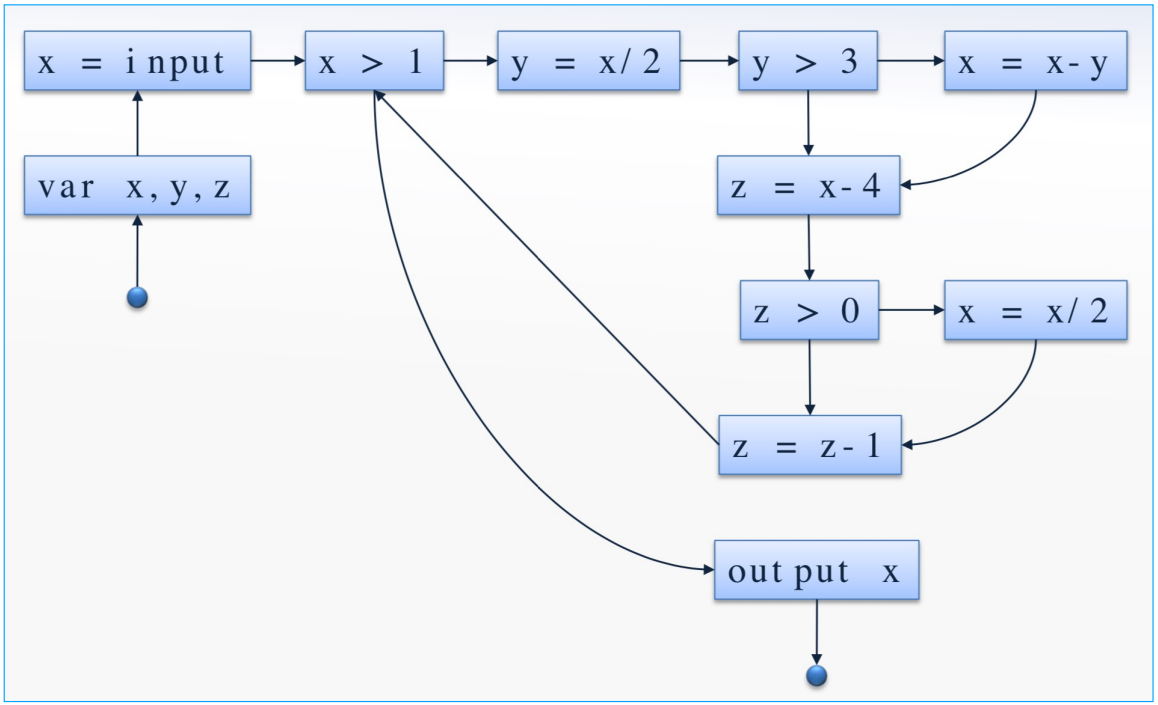
$[\![x=x/2]\!] = \{x,z\}$

$[\![z=z-1]\!] = \{x,z\}$

$[\![output\ \ x]\!] = \{x\}$

$[\![exit]\!] = \varnothing$

Solution



$[\![entry]\!] = \varnothing$
$[\![var\ \ x, y, z]\!] = \varnothing$
$[\![x=input]\!] = \varnothing$
$[\![x>1]\!] = \{x\}$
$[\![y=x/2]\!] = \{x\}$
$[\![y>3]\!] = \{x,y\}$
$[\![x=x-y]\!] = \{x,y\}$
$[\![z=x-4]\!] = \{x\}$

$[\![z>0]\!] = \{x,z\}$
$[\![x=x/2]\!] = \{x,z\}$
$[\![z=z-1]\!] = \{x,z\}$
$[\![output\ \ x]\!] = \{x\}$
$[\![exit]\!] = \varnothing$

- y and z are never live at the same time
- the value assigned in z=z-1 is never read (used) after the assignment

Example optimization

- y and z are never live at the same time ⇒ they can share the same variable location
- the value assigned in z=z-1 is never read (used) after the assignment ⇒ the assignment can be skipped

```
var x,y,z;
x = input;
while (x>1) {
y = x/2;
if (y>3) x = x-y;
z = x-4;
if (z>0) x = x/2;
z = z-1;
}
output x;
```

```
var x,yz;
x = input;
while (x>1) {
yz = x/2;
if (yz>3) x = x-yz;
yz = x-4;
if (yz>0) x = x/2;
/* z = z-1; */
}
output x;
```

Example optimization

- y and z are never live at the same time ⇒ they can share the same variable location
- the value assigned in z=z-1 is never read (used) after the assignment ⇒ the assignment can be skipped

```
var x,y,z;
x = input;
while (x>1) {
y = x/2;
if (y>3) x = x-y;
z = x-4;
if (z>0) x = x/2;
z = z-1;
}
output x;
```

```
var x,yz;
x = input;
while (x>1) {
yz = x/2;
if (yz>3) x = x-yz;
yz = x-4;
if (yz>0) x = x/2;
/* z = z-1; */
}
output x;
```

Dead Store Elimination procedure

- Represent program P as a **control-flow graph** (CFG)
- Use the algorithm to **remove a subset of the dead (not live) stores**, replacing them with `skip`; the result is Q
- DSE is **run multiple times**, as transformations may introduce further dead stores

Outline

Live Variables Analysis
Difficulties for security validation
Syntax and Semantics
Post-domination
Information leakage
Axiomatic semantics
A Taint Proof System
Secure dead store elimination algorithm
Conclusions and what next

Securing a compiler transformation *

- A compiler can be correct and yet be insecure
- There are issues in dead store elimination optimizations
- There are workarounds which can be applied to fix problems, but they rely on programmers' awareness

E.g., in C if `x` is declared `volatile`, the
compiler does not remove any assignment to x: not very portable
It is a compiler specific solution: programmer awareness is required
It is also strong, it inhibits any assignment to x

- A possible solution is **translation validation**:
  - given an instance of a correct transformation, checks whether it is secure

Considering just a single run of the compiler at the time on the single program and not the compilation in general

* Chaoqiang Deng, Kedar S. Namjoshi. *Securing a compiler transformation*. Formal Methods Syst. Des. 53(2), 2018.

Inherent difficulties for security validation

- How difficult is it to check the security of a transformation a posteriori?
  Given programs P, Q, and a list of eliminated stores,
  - validating correctness is in PTIME, but
  - validating security is undecidable
- Any sound validator will be incomplete
- Is there a practical, provably secure dead-store elimination procedure?
  - New procedure, based on taint propagation and program flow

Verifying Correctness with Translation Validation

To verify the **correctness** of a DSE instance (P,Q,D), with

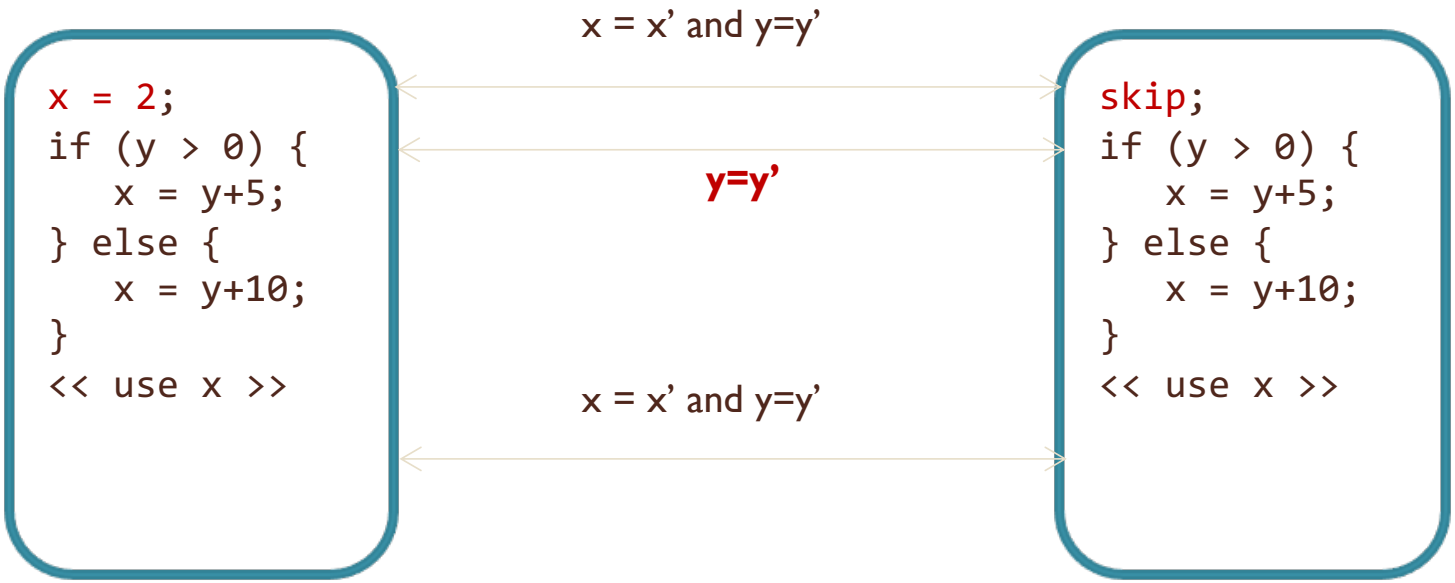
- P input program,
- Q resulting program, and
- D list of removed stores

- Compute dead stores D_P in P; check that $D_P \supseteq D$
- Check that P and Q have bisimilar control-flow graphs up to the labeling of edges in D

- This procedure is in PTIME

Induced refinement relation

At corresponding program points p (in P) and q (in Q), every live variable at p has identical values at p and q



But security properties are not preserved

Verifying Correctness with Translation Validation

Validating **security** is instead undecidable

The difference between correctness and security is fundamentally due to the fact that:

- correctness can be defined by considering individual executions
- information flow requires the consideration of pairs of executions

- Need for a stronger notion of refinement which preserves information flow, and use it to show that several common compiler optimizations do preserve information flow properties

The procedure

- The algorithm takes as input a program P and a list of dead assignments
- It prunes that list to those assignments whose removal does not introduce a new information leak
- removes them from P, obtaining the result program Q
- The analysis of each assignment relies on
  - taint information (via a Hoare-style proof system) and
  - control flow information from P (in the form of dominance relations)

- The algorithm can be proven secure, according to a notion of secure transformation

Premises

The algorithm

1. is sub-optimal: it may retain more stores than is strictly necessary
2. enforces relative rather than absolute security: it does not eliminate information leaks from P, it only ensures that no new leaks are introduced in the transformation from P to Q
3. It gives guarantee only for information leakage. Other security aspects must be checked separately

Example

⟦x = 2⟧= ⟦ y>0⟧ \{x}
⟦y>0⟧= ⟦x = y+5⟧ U ⟦x = y+10⟧ U {y}
⟦x = y+5⟧ = ⟦<< use x >>⟧ \{x} U {y}
⟦x = y+10⟧ = ⟦<< use x >>⟧ \{x} U {y}
⟦<< use x >>⟧ = … U {x}

```
x = 2;
if (y > 0) {
    x = y+5;
} else {
    x = y+10;
}
<< use x >>
```

Example

⟦x = 2⟧= ⟦ y>0⟧ \{x}
⟦y>0⟧= ⟦x = y+5⟧ U ⟦x = y+10⟧ U {y}
⟦x = y+5⟧ = ⟦<< use x >>⟧ \{x} U {y}
⟦x = y+10⟧ = ⟦<< use x >>⟧ \{x} U {y}
⟦<< use x >>⟧ = … U {x}

```
x = 2;
if (y > 0) {
    x = y+5;
} else {
    x = y+10;
}
<< use x >>
```

On every execution path:
x is redefined and the value assigned in
x = 2 is never used

Is it safe to remove?

Preliminaries: program syntax

- All variables have Integer type
- Variables are partitioned into input and state variables:
  - **state variables** are low security (in set **L**),
  - input variables may be low or high security (in sets **L** and **H**)

$$
\begin{array}{ll}
x \in \mathbb{X} & \text{variables} \\
e \in \mathbb{E} ::= c \mid x \mid f(e_1, \ldots, e_n) & \text{expressions: } f \text{ is a function, c a constant} \\
g \in \mathbb{G} & \text{Boolean conditions on } \mathbb{X} \\
S \in \mathbb{S} ::= \mathsf{skip} \mid \mathsf{out}(e) \mid x := e \mid S_1; S_2 \mid \mathsf{if}\ g\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid & \\
\quad \mathsf{while}\ g\ \mathsf{do}\ S\ \mathsf{od} & \text{statements}
\end{array}
$$

Preliminaries: control flow graphs

A program can be represented by its **control flow graph**, where:

- nodes represent program locations
- edge are labeled with a guarded command of the form, with g boolean predicate and e an expression
  - g → x := e or
  - g → skip, or g → out(e)
- a special node, entry (exit), defines the initial/final program location
- values for input variables are specified at the beginning and remain constant

Preliminaries: program semantics

A **program state** s is a pair (m, p), where:

- m is a CFG node (location of s)
- p is a function from each variable to a value from its type (also extended to expressions)

- In the initial state, located at the entry node, state variables have a fixed valuation

Preliminaries: program semantics (cont.)

Transition relation: two states s = (m,p), t=(n,q) are in the relation if:

- there is an edge f = (m, n) of the CFG, and the guarded command on that edge is either
- in the form g → x := e, where
  - g holds of p, and q(y) = p(y) for all variables y ≠ x, or
- in the form g → skip, where
  - g holds of p, and q is identical to p

A computation is an execution trace starting from the initial state

Outline

Live Variables Analysis
Difficulties for security validation
Syntax and Semantics
Post-domination
Information leakage
Axiomatic semantics
A Taint Proof System
Secure dead store elimination algorithm
Conclusions and what next

Preliminaries: [post]-dominance

A set of nodes N in CFG **dominates** a node m

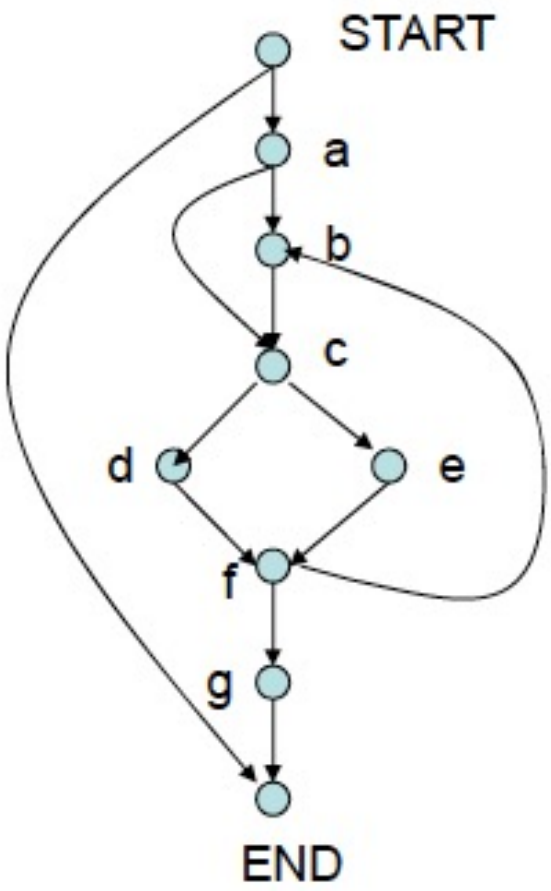
- if each path in the CFG from the entry node to m has to pass through at least one node of N

A set of nodes N in CFG **post-dominates** a node m

- if each path in the CFG from m to the exit node has to pass through at least one node of N, i.e., contains at least one node of N

Domination relations are graph-theory relations and can be built using a dataflow analysis

Preliminaries: [post]-dominance



- c is dominated by a and by b
- c is post-dominated by f and by g

Outline

Live Variables Analysis
Difficulties for security validation
Syntax and Semantics
Post-domination
Information leakage
Axiomatic semantics
A Taint Proof System
Secure dead store elimination algorithm
Conclusions and what next

Preliminaries: information leakage

Consider a deterministic program P

- Partition input variables into Low (L) and High (H) security
- All state variables are Low security

P leaks information if ∃ inputs (H=a,L=c) and (H=b,L=c) s.t. the resulting computations:

- differ in the sequence of output values, or
- terminate in different states (differ in the value of one of the L-variables)

We call (a, b, c) a **leaky triple** for program P

Preliminaries: correct transformation

A transformation from P to Q is **correct** if, for every input value a, the sequence of output values for executions of P and Q from a is identical

Q is at least correct as P; it does not assure the correctness of either program with respect to a specification.

Preliminaries: information leakage

Ex: for input values h=0, h=1, the final values of x differ

Explicit flow

```
High h
Low x initially 0

x = h;
```

```
High h
Low x initially 0

if (h > 0) {
   x = 10;
} else {
   x = 20;
}
```

Implicit flow

A transformation from P to Q is **secure** if every leaky triple for Q is also leaky for P (Q does not add any

End