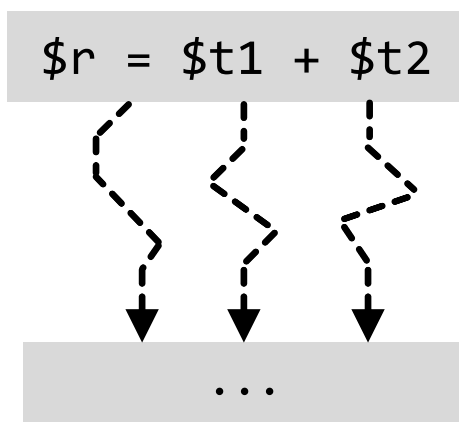
Chapter 8. Optimization (Part 3)

Prof. Jaeseung Choi
Dept. of Computer Science and Engineering
Sogang University

Available Expression (AE) Analysis

- **Consider an expression e and program point p**
  - e can have various forms: "`$t1`", "`$t1 + $t2`", "`4 * $t1`"
  - We can choose the scope of expression to trace
- **Intuitively, e is available at p in register `$r` if the re-computation of e at p produces a value stored in `$r`**
  - Note the difference with reaching definition

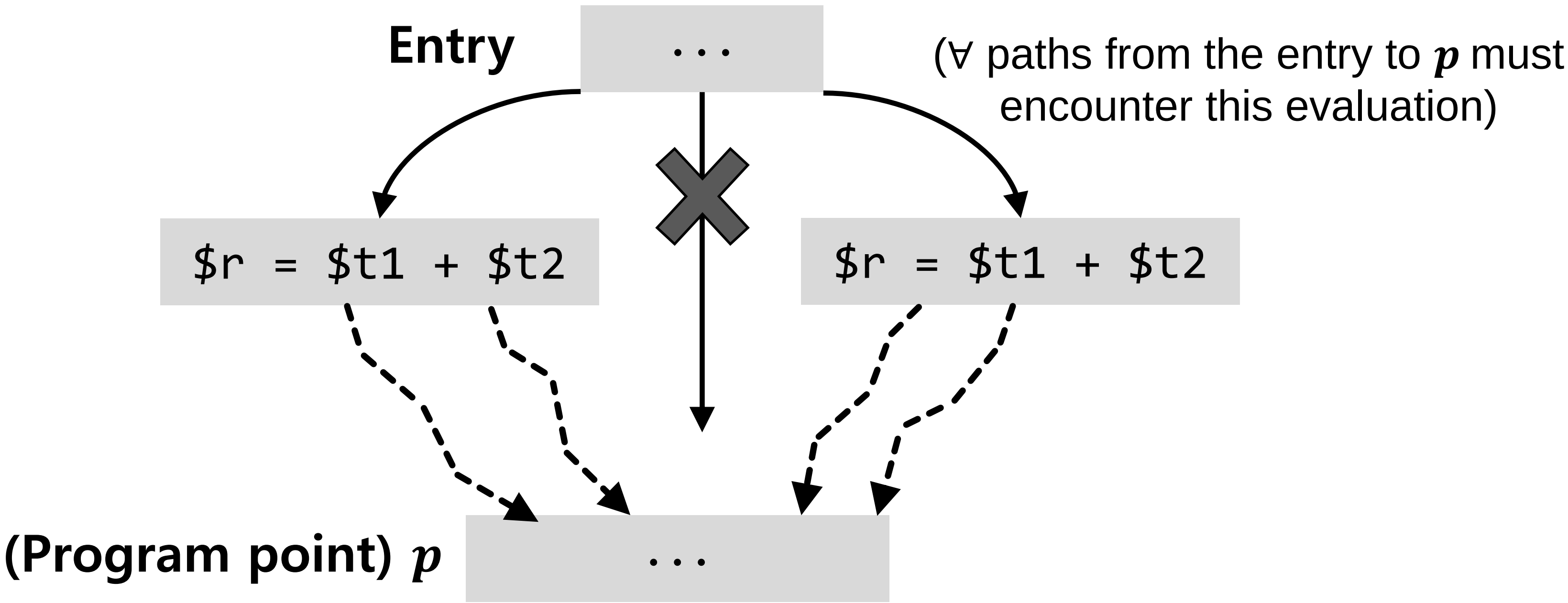


(Program point) p

(∀ execution paths, none of `$r`, `$t1`, `$t2` is redefined)

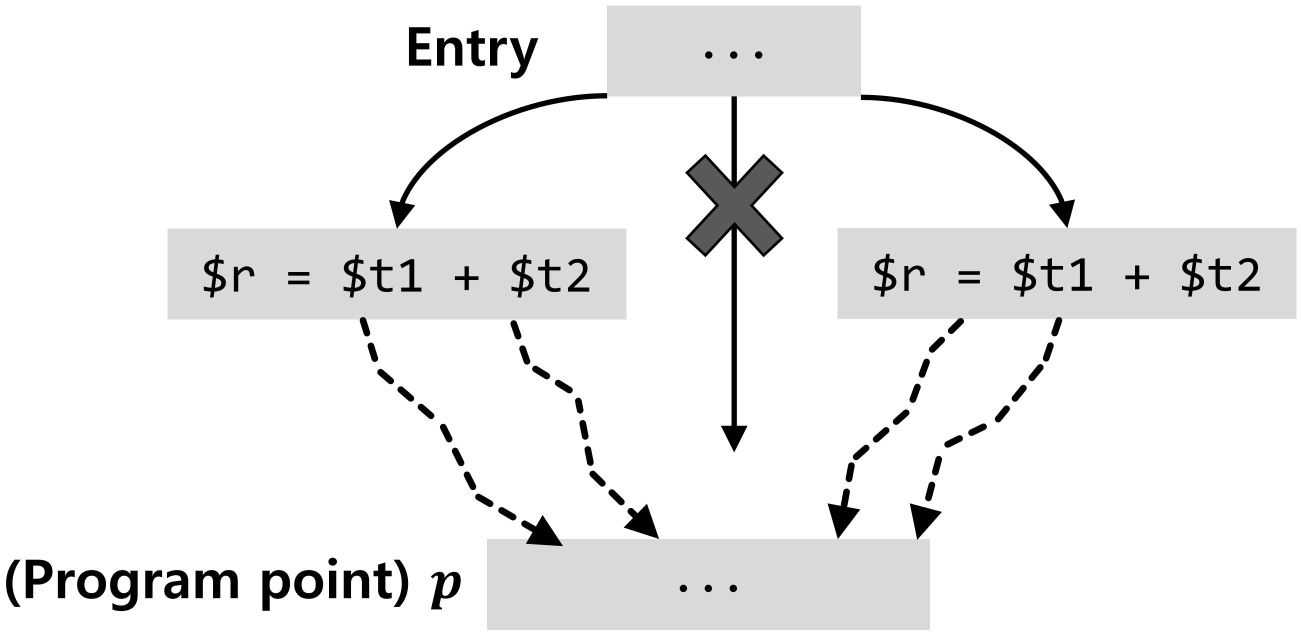
Available Expression (AE) Analysis

- **In addition, all the paths from the (function) entry to p must evaluate e and store the result in \$r**
  - In other words, if there is any path that can arrive at p without such evaluation, e is not available at p



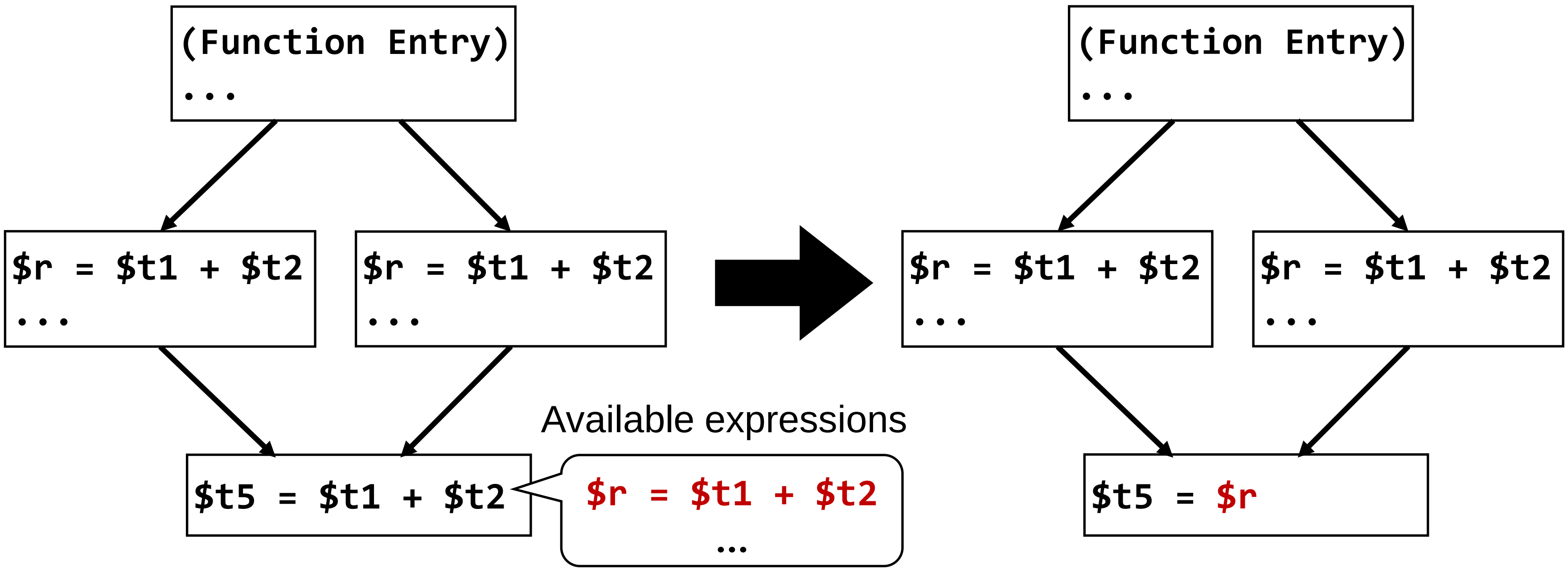
Available Expression (AE) Analysis

- Now, formally speaking: ***e*** **is available at *p* in register $r** if every path from the entry to ***p*** evaluates ***e*** and stores the result in **$r**, and after such last evaluation, there is no subsequent update to **$r** or registers used in ***e***



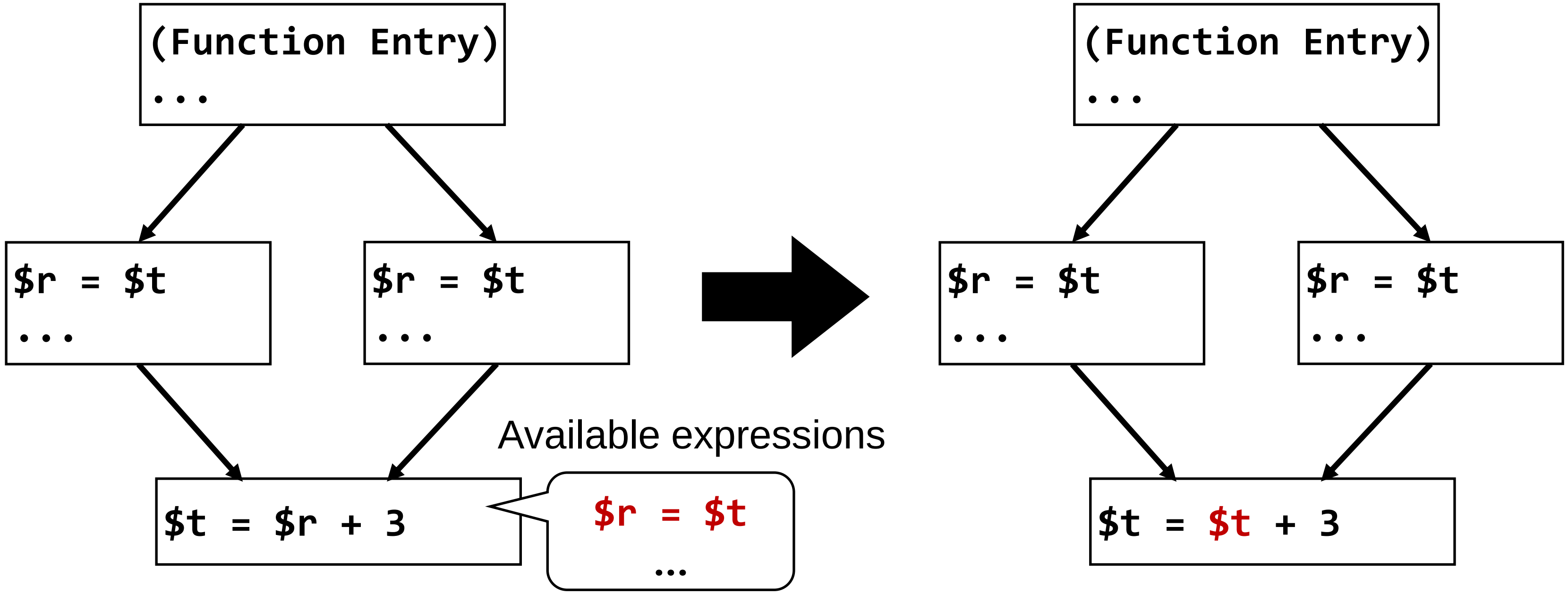
Use of Available Expression (AE)

- **From the definition of AE, it is straightforward to perform common subexpression elimination (CSE)**
  - If the RHS expression of an assign statement is available in register **`$r`**, that RHS can be substituted with **`$r`**



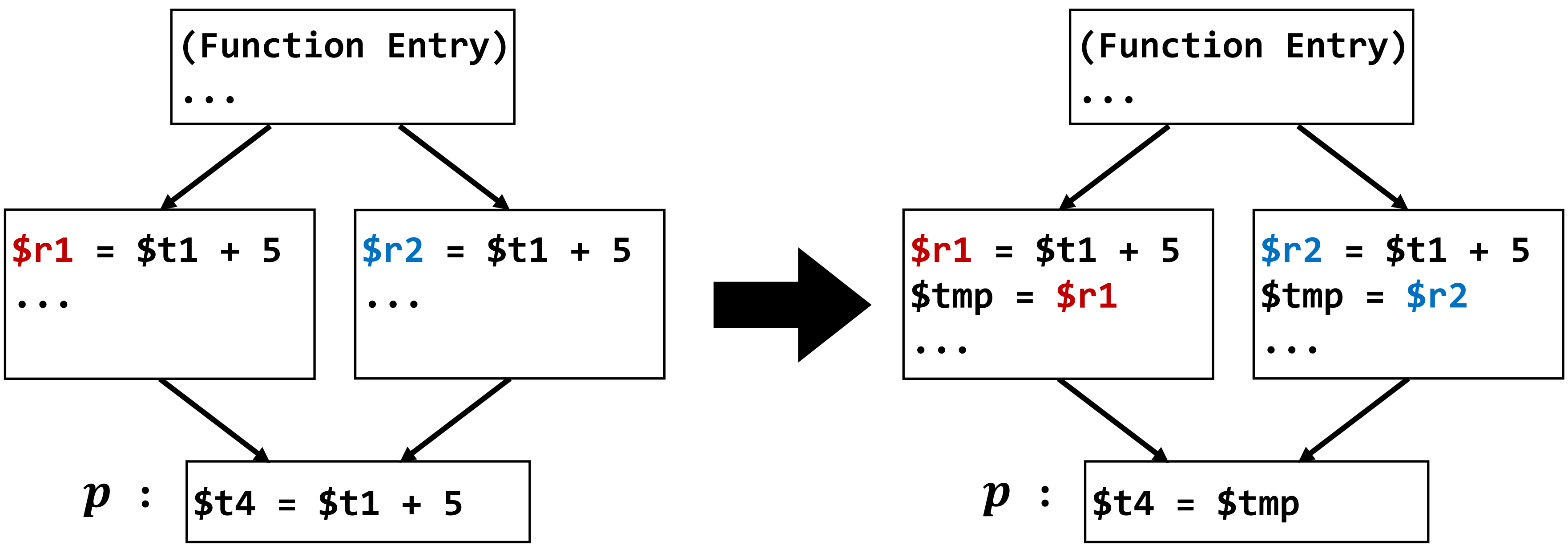
Use of Available Expression (AE)

- **Also, if the available expression *e* has the form of a single variable, we can use it for copy propagation**
  - If expression **`$t`** is available in register **`$r`**, then these two registers can be used interchangeably
  - This time, we will replace **`$r`** with **`$t`**, for copy propagation



Note: Other Form of AE Analysis

- **In most textbooks, AE analysis does not care about the register that stores the evaluation result**
  - Such textbook will say that ***\$t1 + 5*** *is available at* p
  - Why? Because CSE is still possible by introducing a new registers in such cases (but we will not do this in our course)

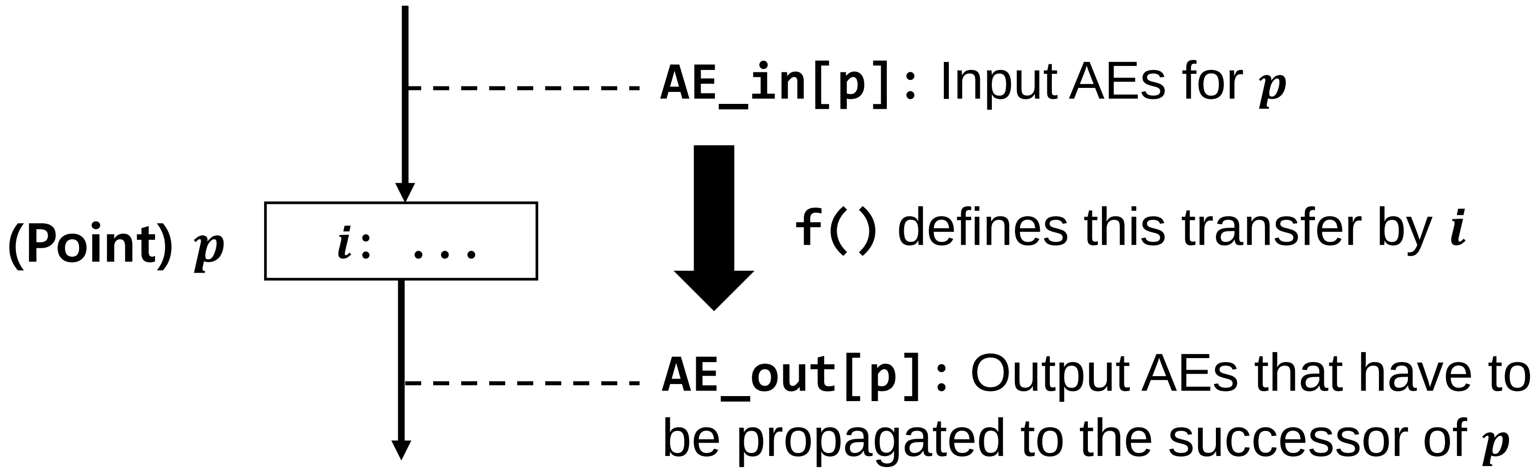


Remind: Conservativeness

- **The analysis and optimization must be conservative**
  - This time, the definition of **"conservative"** is different
  - For CSE, the analysis result must only contain AEs that are always available at runtime (no false positive allowed)
  - Instead, it is okay (and inevitable) to under-approximate the AEs
- **Again, if we have false positives in the result of AE analysis, we may end up with a wrong optimization**

AE Analysis: Transfer Function

- **As before, we must define how the AE set will change by the execution of a single instruction**
- **Assume that program point p contains instruction i**
  - Transfer function `f` defines output AE in terms of input AE and i
  - `AE_out[p] = f(AE_in[p], i)`



AE Analysis: Transfer Function

- If instruction i has the form of "`$r` = e", where e is a register or binary/unary operation:
  - `f(IN, i) = IN` – { AE ∈ IN | AE contains `$r` } ∪ { e in `$r`}
  - Ex) "`2 * $x`" stored in `$z`" is an AE that contain `$x`
  - Ex) "`$y` stored in `$x`" is also an AE that contain `$x`
  - Corner case: what should happen when i is "`$x=$x+1`"?
    - Actually, we must add { e in `$r`} only if e does not contain `$r`

(Point) p

`AE_in[p]` = {2*$x in $z, $y in $x, $y/4 in $w}

`i: $x = $y + 5`

Transferred by `f()`

`AE_out[p]` = {$y+5 in $x, $y/4 in $w}

AE Analysis: Transfer Function

- Note that "`$r = alloc(N)`" must not generate any AE
  - `f(IN, `*i*`) = IN` – { AE ∈ `IN` | AE contains `$r` }
- For other instructions that does not define any register:
  - `f(IN, `*i*`) = IN`
- Q. What about the load instruction, "`$r = load $t`"?
  - To support this form of expression in AE, we must also consider the effect of **`store`** instruction (I will leave this as your exercise)

(Point) p

`AE_in[p]` = {2*$x in $z, $y in $x, $y/4 in $w}

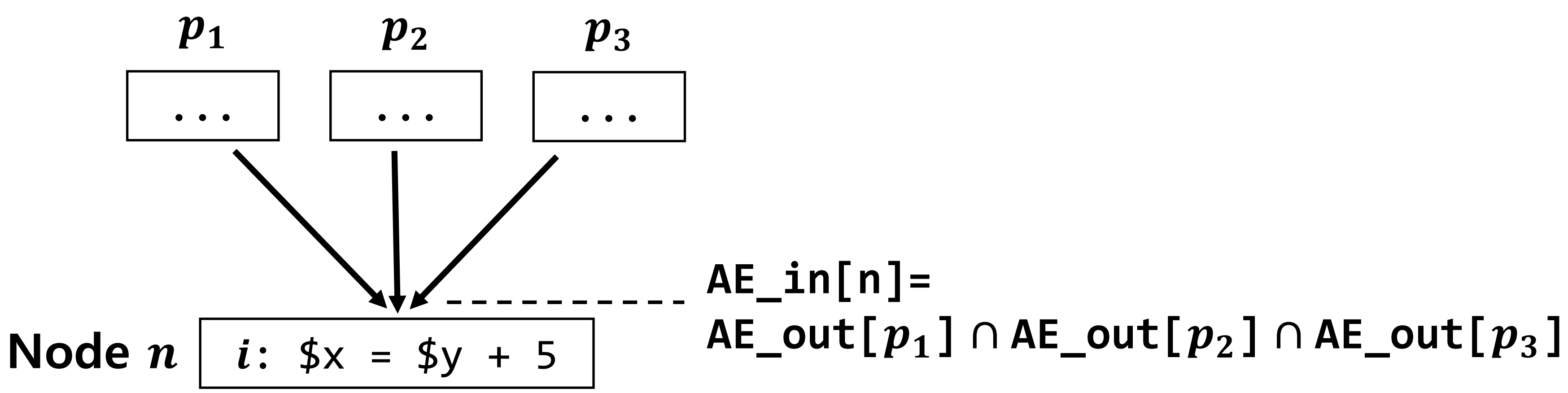
i: `$x = load $y`

Transferred by `f()`

`AE_out[p]` = { ? }

AE Analysis: Propagation

- How should we propagate AEs along the control-flows?
- An expression is available at a program point if that expression is available after all of the predecessors
- For node n, the output AEs of n's predecessors must be joined with intersection (∩) and used as n's input



AE Analysis: Iterative Algorithm

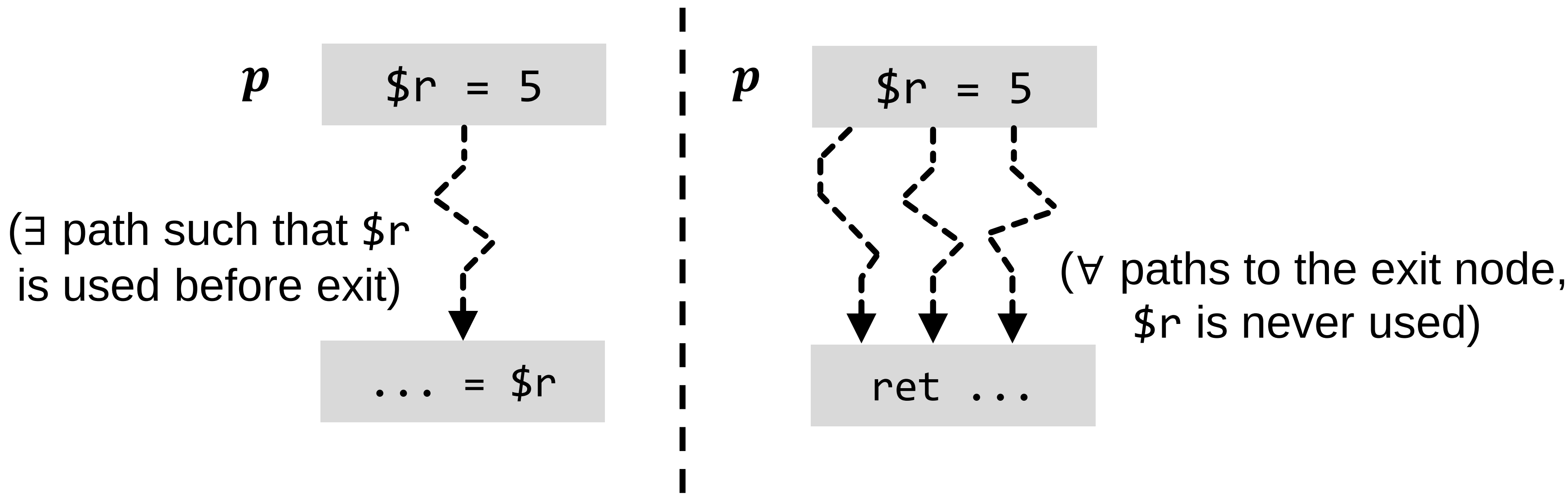
- **Now we can put things together and run the following iterative algorithm (a.k.a. fixpoint algorithm)**
  - There can be several variations, but the basic idea is same
  - In the algorithm below, $\boldsymbol{U}$ denotes the universal set of all the expressions that can appear in the program

```
for each node n { AE_out[n] = U; }

while (there is any change to AE_out[]) {
  for each node n and its instruction i {
    AE_in[n] = ∩_{p∈pred(n)} AE_out[p];
    AE_out[n] = f(AE_in[n],i);
  }
}
```

Liveness Analysis

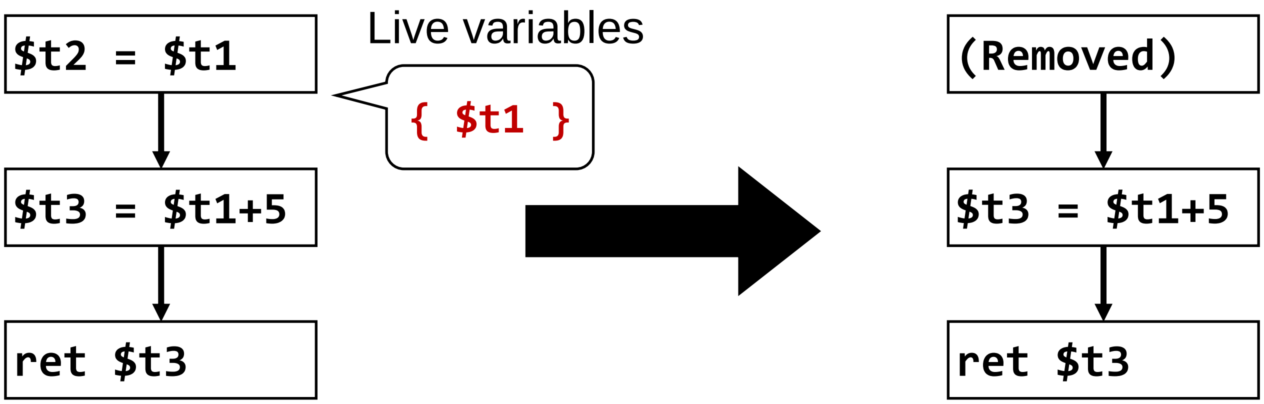
- **A variable/register `$r` is live at (after) program point p if its value can be used after the execution of p**
- ***Live variable analysis*, or *liveness analysis*, computes the set of live variables at each program point**



`$r` is *live* at p | `$r` is *dead* at p

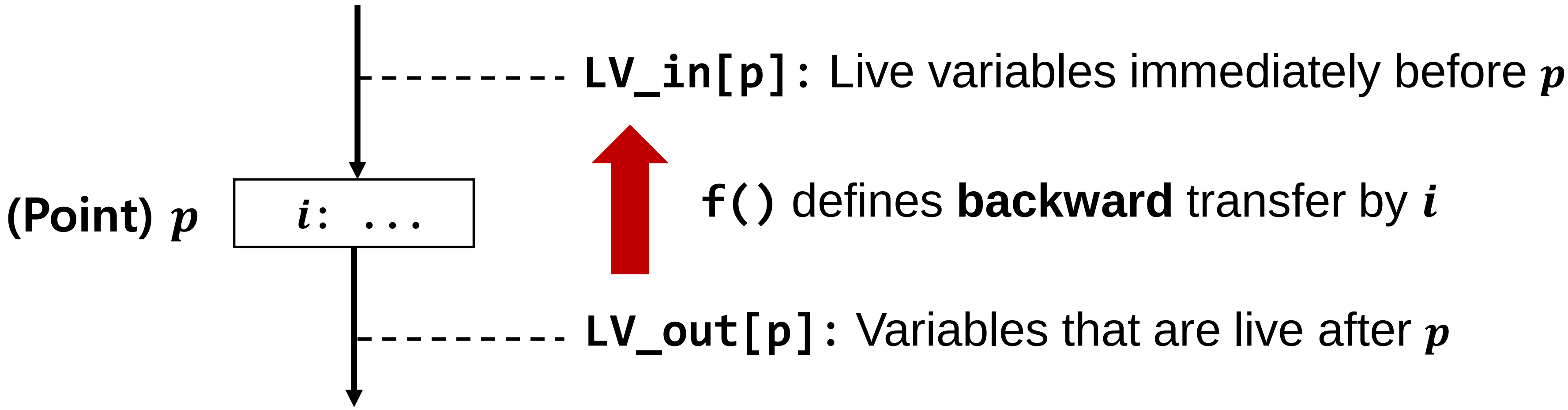
Use of Liveness Analysis

- **We can perform dead code elimination (DCE) with the result of liveness analysis**
  - If an instruction is defining a register that is ***not live*** after that instruction, we can remove such dead instruction
- **Again, remember that we have to be conservative**
  - The analysis result can over-approximate the live variables, but should not under-approximate them



Liveness Analysis: Transfer Function

- **Note that liveness analysis must be performed backward**
  - Start from the exit node and goes backward
- **Therefore, the transfer function must define the input live variables in terms of the output live variables**
  - For point p with instruction i: `LV_in[`p`] = f(LV_out[`p`],` i`)`



(Continued in Part 4)