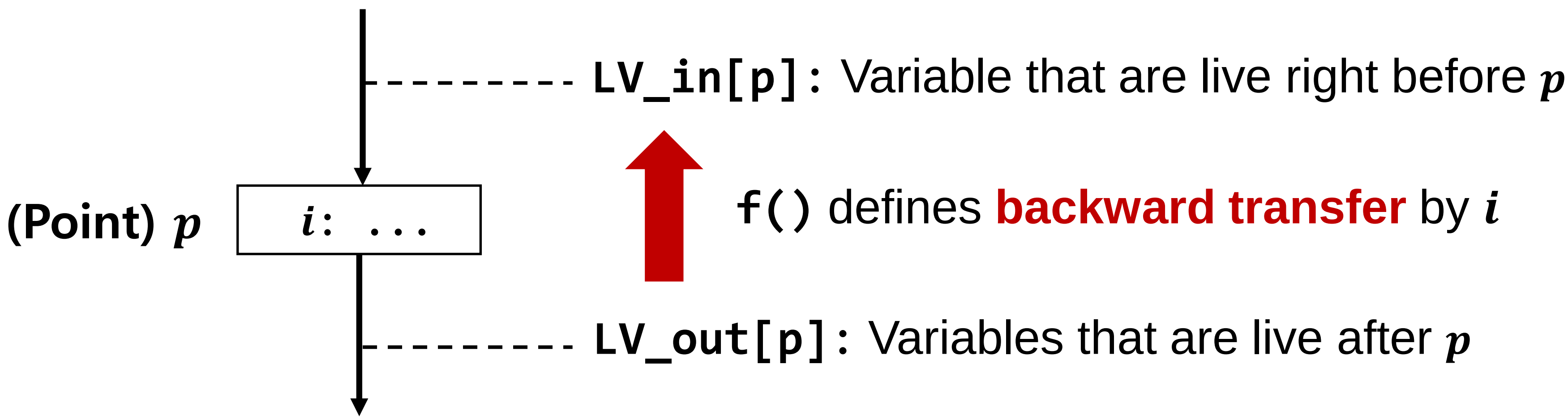
Chapter 8. Optimization (Part 4)

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

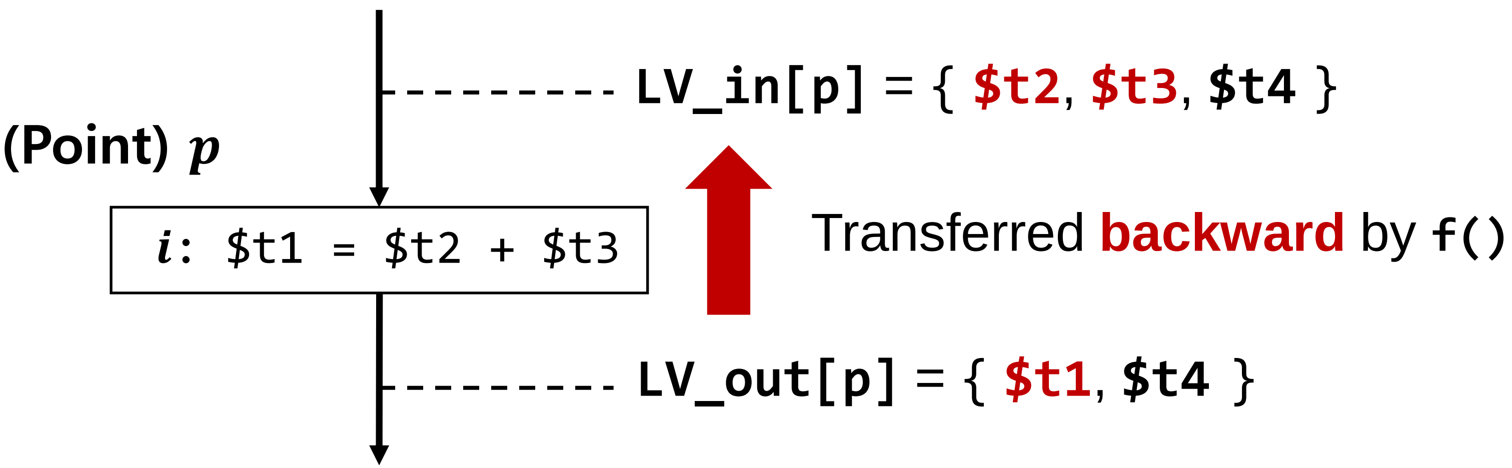
Liveness Analysis: Transfer Function

- **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)`**



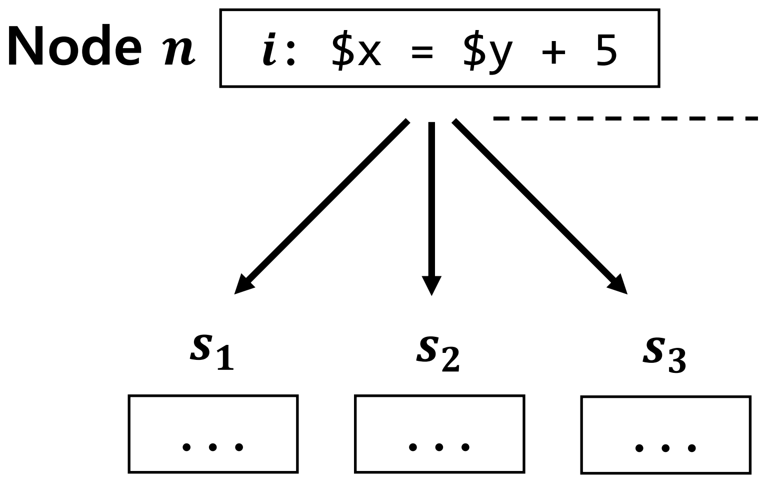
Liveness Analysis: Transfer Function

- **For instruction i, let's introduce the following two sets:**
  - **`def(`i`)`** denotes the set of registers defined by i
  - **`use(`i`)`** denotes the set of registers used by i
  - Ex) **`def("$t1 = $t2 + $t3") = { $t1 }`**
  - Ex) **`use("$t1 = $t2 + $t3") = { $t2, $t3 }`**
- **Now we can define: `f(LV, `i`) = LV – def(`i`) ∪ use(`i`)`**



Liveness Analysis: Propagation

- **If a variable is *live before* a certain node, then it is also *live after* its predecessors**
- **Therefore, live variables after node n can be obtained by joining the live variables before the successors of n**
  - These live variables must be joined with **union (**∪**)**



LV_out[n]=

LV_in[s_1] ∪ LV_in[s_2] ∪ LV_in[s_3]

Liveness Analysis: Iterative Algorithm

- **Now we can put things together and run the following iterative algorithm (a.k.a. fixpoint algorithm)**
  - There can be variations of algorithm, but the basic idea is same

```
for each node n { LV_in[n] = ∅; }

while (there is any change to LV_in[]) {
  for each node n and its instruction i {
    LV_out[n] = ⋃_{s∈succ(n)} LV_in[s];
    LV_in[n] = f(LV_out[n],i);
  }
}
```

Optimization with Memory?

- **Recall that compilers focus on optimizing register computations, and usually ignore variables in memory**
- **For example, let's consider *dead-store elimination:***
  - Why don't we remove a **`store`** instruction if it is dead?
  - Unfortunately, it is not trivial to decide whether a **`store`** updates a variable that is not going to be used anymore
  - It requires the compiler to analyze which memory addresses can be accessed by each **`load`** and **`store`**
- **As a result, real-world compilers usually perform only limited form of dead-store elimination**
  - By targeting trivial and obvious patterns

Mem2Reg Optimization

- **Mem2Reg optimization moves some variables from the memory to register (a.k.a., *promotion* to register)**
  - Once it is promoted, various optimizations can be applied
  - Ex) Consider C code "**`int x = 100; x = x + 1;`**"

&x

```
$t1 = alloc 4
store 100, $t1
$t2 = load $t1
$t3 = $t2 + 1
store $t3, $t1
```

Promote x to register

```
%x = 100
$t1 = %x + 1
%x = $t1
```

Memory

2000	101

Registers

$t1	2000
...	...

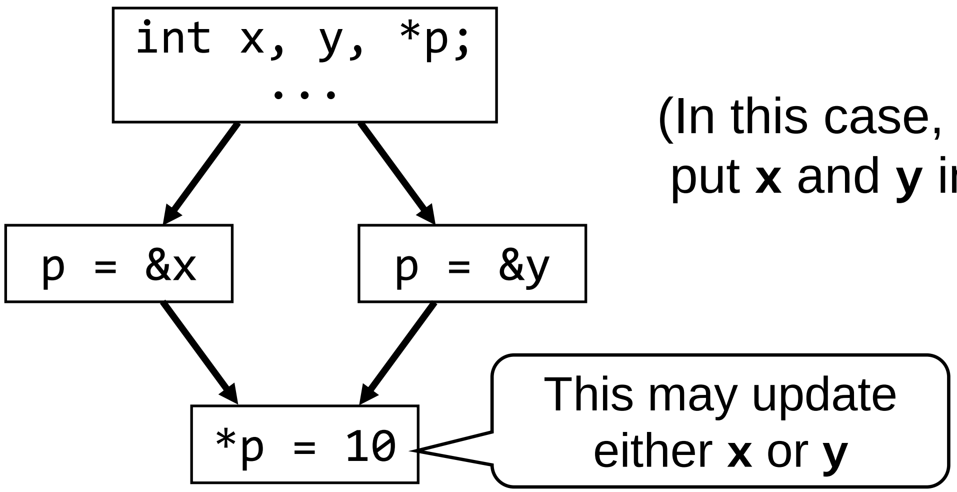
Memory

Registers

%x	101
...	...

Mem2Reg Optimization

- **Of course, not all variables can be promoted to register**
  - Compiler must carefully decide which variables can be promoted
- **Array type variables cannot be promoted to register**
  - Ex) **`int arr[16]; //`** We cannot put this in register
- **Pointer operations also disqualify some variables from the promotion to register**



(In this case, we cannot put **x** and **y** in register)

Appendix

Constant Folding + Propagation

- **Part 1 slide mentioned advanced form of optimization for "constant folding + constant propagation" at once**
  - Unfortunately, we don’t have time to discuss it in this course
  - If you are interested, read Chapter 9.4 of our textbook*

- **Sometimes, we can reduce the number of repetition by making each optimization pass smarter**
  - For example, we may perform constant folding and constant propagation at once (we will see such optimization later)