Chapter 8. Optimization (Part 2)

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

Topics

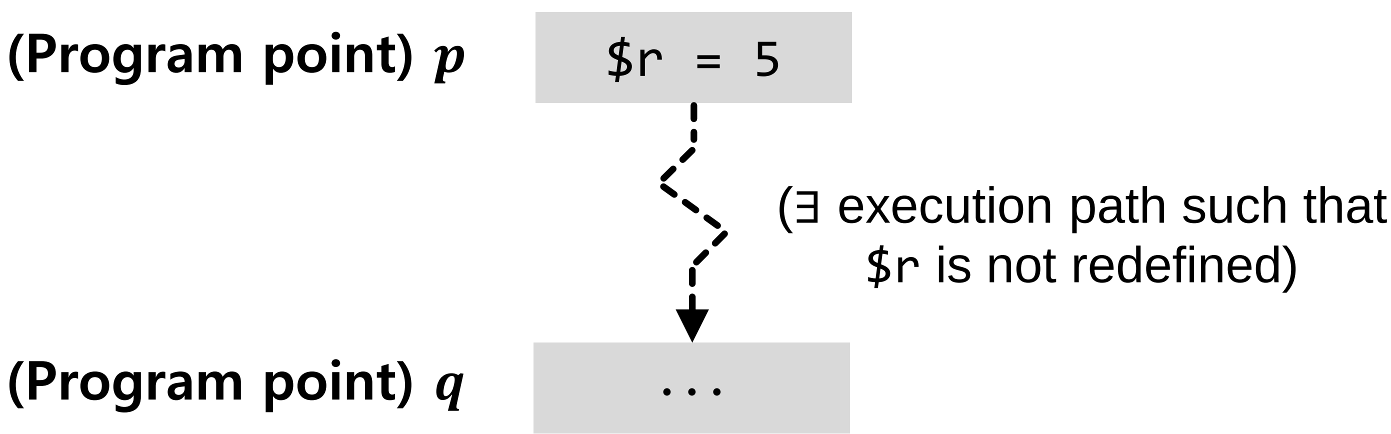
- **General background**
  - Goal and principle of optimization
  - Scope of optimization
  - Basic block and CFG
- **Common types of optimization**
  - Constant folding, constant propagation, copy propagation, common subexpression elimination, dead code elimination, ...
- **Data-flow analyses to realize the optimization**
  - Reaching-definition analysis, constant propagation analysis, available expression analysis, liveness analysis, ...

Data-Flow Analysis

- **As we have observed before, correct optimization requires analysis of program behavior**
  - Especially, about the data-flows along the program execution
- **Various kinds of data-flow analyses are used**
- **However, there is a common principle and framework shared by those various data-flow analyses**
  - Let's start with a concrete example, and generalize it later

Reaching Definition (RD) Analysis

- A **definition of `$r` at p reaches q** if there is a path from p to q such that `$r` is not redefined along the path
- Reaching definition analysis computes the set of reaching definitions for each program point



Use of Reaching Definition (RD)

- **In real-world compiler, reaching definition analysis is one of the most important analyses**
  - Used for lots of optimizations in practice
  - But in our course, only related to **constant propagation**
- **Assume that we analyzed the RDs for a program point with instruction `$t7 = $t1 + 10`**
  - If ***all*** the RD of **`$t1`** is defined with a constant **`C`**, **`$t1`** can be safely replaced with **`C`**

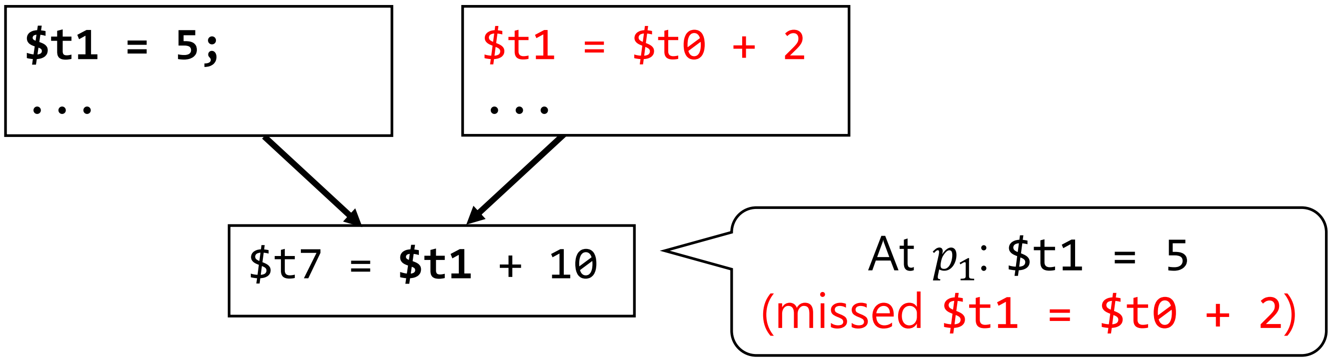
```
...
$t7 = $t1 + 10
```

Reaching definitions

At p_1: `$t1 = 5`
At p_2: `$t1 = 5`
…
(and no more reaching definition for `$t1`)

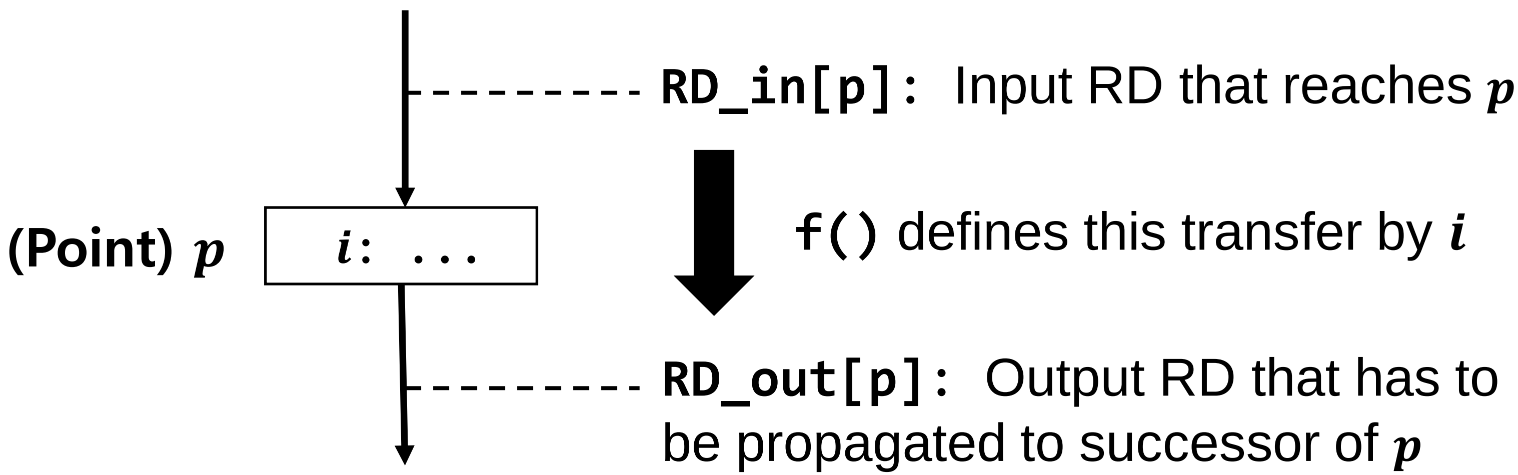
Principle: Conservativeness

- **The analysis and optimization must be conservative**
  - For constant propagation, the analysis must not miss any reaching definition that can occur at runtime
  - Instead, it is okay to over-approximate the reaching definition set (such over-approximation is usually inevitable)
- **Q. What if the analysis misses some reaching definition?**
  - We may end up replacing a register that should not be replaced!



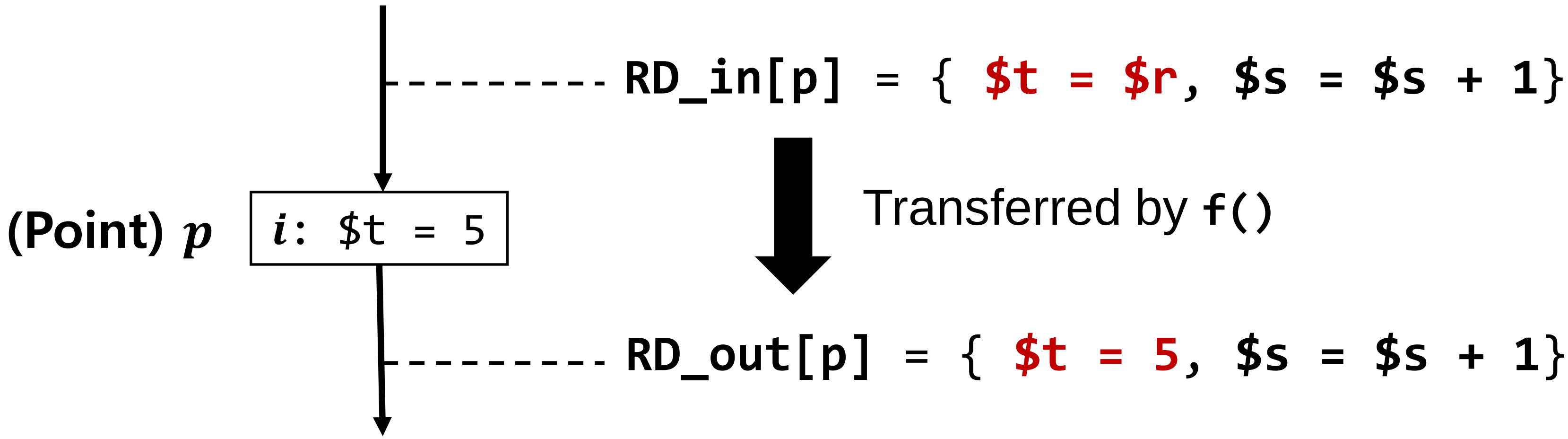
RD Analysis: Transfer Function

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



RD Analysis: Transfer Function

- **If the instruction i at p has the form of "`$t = ...`":**
  - `f(IN, i) = IN – { RD ∈ IN | RD defines $t } ∪ { i }`
  - Or you may include program point p in the RD as well: $\{\langle p, i \rangle\}$
- **For other kinds of instructions (e.g., `store`, `goto`, …):**
  - `f(IN, i) = IN`

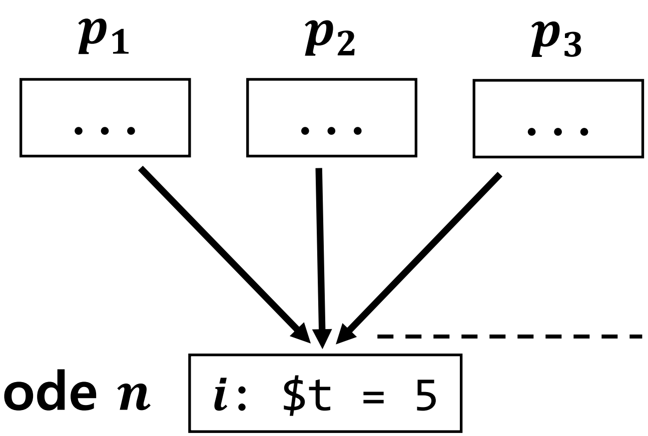


Note: Gen-Kill Form

- **Many compiler textbooks denote the transfer functions of any data-flow analysis in a *gen-kill* form:**
  - $\mathtt{f(IN,}\ i\mathtt{)} = Gen_i \cup (\mathtt{IN} - Kill_i)$
  - For the example in the previous page, `{$t = 5}` **corresponds to** Gen_i **and** `{$t = $r}` **corresponds to** $Kill_i$
- **Although this form has some advantages, IMHO it brings unnecessary complexity (so we will not use it)**
  - But it won’t hurt to know about the existence of this form

RD Analysis: Propagation

- How should we propagate RDs along the control-flows?
- Node of CFG is originally a basic block, but let's simply assume that a node is just a single instruction
- For node n, the output RDs of n's predecessors must be joined with union (∪) and used as n's input



Node n

`RD_in[n]=`
`RD_out[`p_1`]` ∪ `RD_out[`p_2`]` ∪ `RD_out[`p_3`]`

RD Analysis: Iterative Algorithm

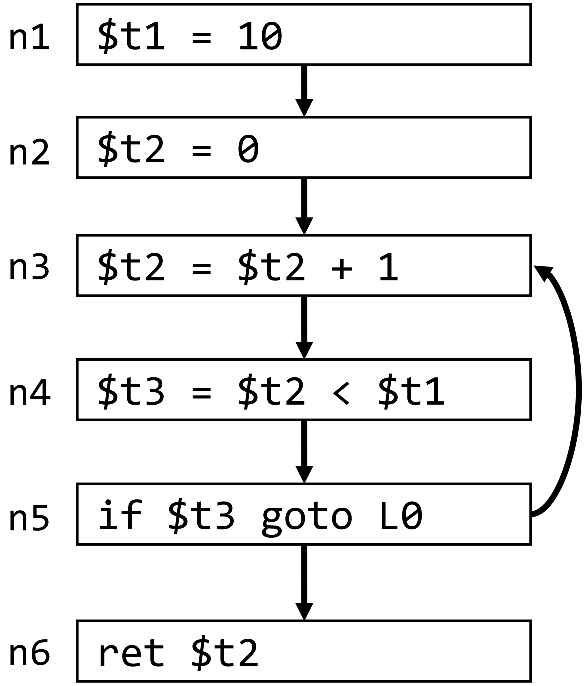
- **Now we can put things together and run the following iterative algorithm (a.k.a. fixpoint algorithm)**
  - There can be several variations, but the basic idea is same

```
for each node n { RD_out[n] = ∅; }

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

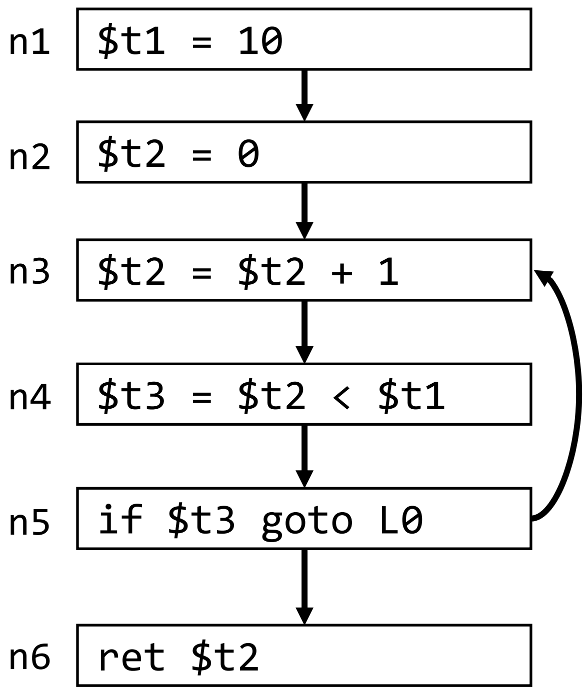
RD Analysis: Example

- Again, for simplicity let's assume that each node of CFG only contains a single instruction
- Compute the reaching definition for the code below



RD Analysis: Example

- Again, for simplicity let's assume that each node of CFG only contains a single instruction
- Compute the reaching definition for the code below



(Fixpoint reached)

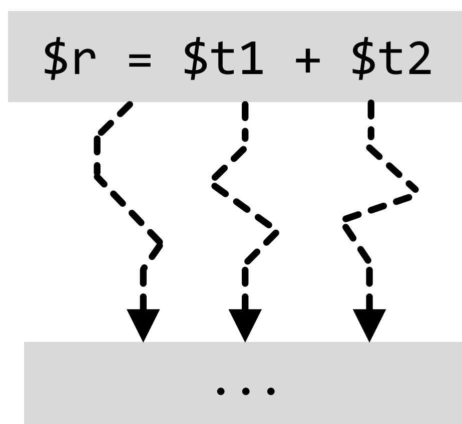
Node	RD_in
n1	∅
n2	**$t1=10**
n3	**$t1=10, $t2=0, $t2=$t2+1, $t3=$t2<$t1**
n4	**$t1=10, $t2=$t2+1, $t3=$t2<$t1**
n5	**$t1=10, $t2=$t2+1, $t3=$t2<$t1**
n6	**$t1=10, $t2=$t2+1, $t3=$t2<$t1**

Generalization

- **Recall that RD analysis had the following flow**
  - Define how the RD is updated by each instruction
  - Define how the RD is propagated along the control
  - Run iterative algorithm until there is no more change in the RD analyzed for all the program points
- **Other data-flow analyses will share the same flow**

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)

(Continued in Part 3)