Dataflow Analysis

Lorenzo Ceragioli

November 29, 2024

IMT Lucca

Static Analysis

Objectives

- Assess Formal Correctness
- Compute Information (e.g. for optimization)

Example of pipeline

- Controlflow analysis (build the CFG) – trivial for us
- Dataflow analysis – compute available data at different locations of the program

Dataflow Analyses

Different kinds of information depending on the analysis

- Live variables: which variables (or registers) hold values that will be (possibly) used later
- Reaching definitions: which definitions (may) have determined the value of variables (or registers)

Roughly the same (parametric) procedure

1. Associate initial value to each block of the CFG
2. Define how the local information is updated w.r.t. the value of the neighbors
3. Iterate the updates until a fixpoint is reached

Dataflow Analysis Recap

Computed Value: some Complete Partial Order

Analysis State: Associate a pair of values to each block (*in* and *out*)

Local Update: update the value associated with a block

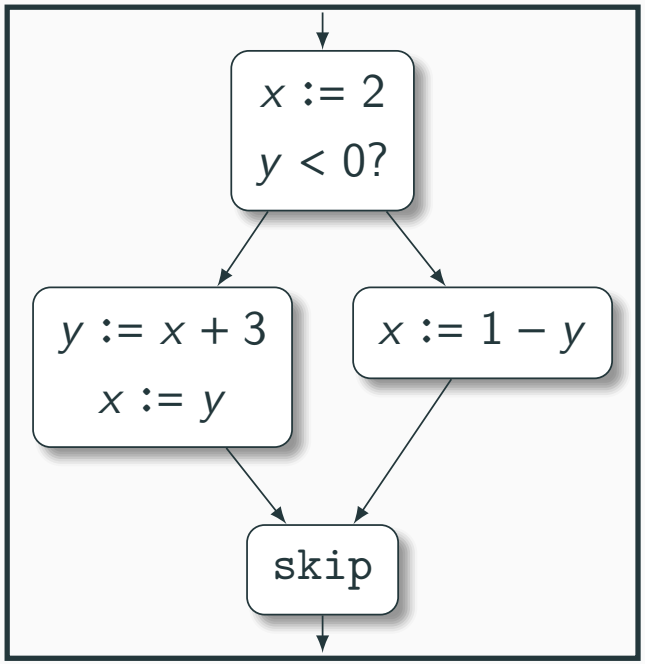
- From the block itself (*in* to *out* or vice-versa)
- From a block to the others (next or preceding blocks)

Global Update: all local updates are repeated until fixpoint is reached (there are more clever ideas, but this is enough for us)

... then possibly refine the analysis for the single instructions inside the block.

Two Alternatives for CFG

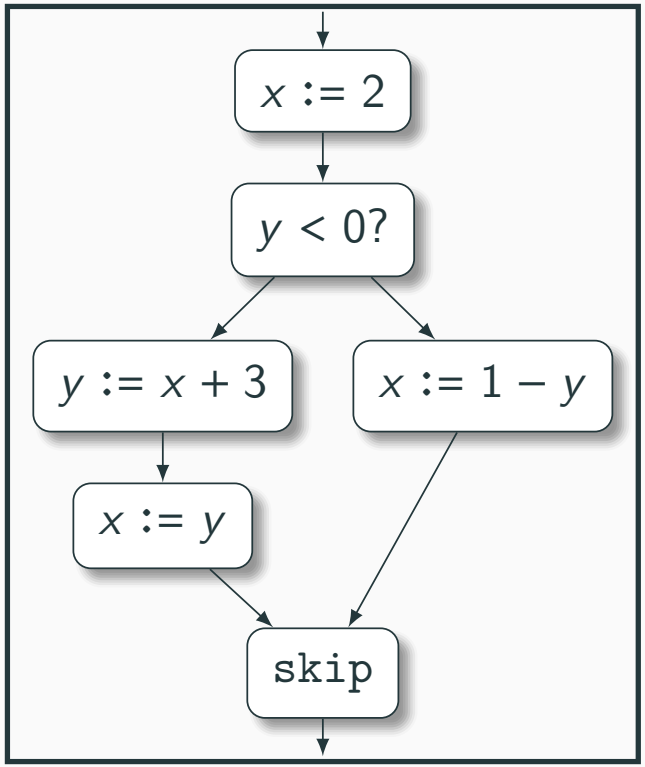
Maximal Blocks



1) Perform the analysis on blocks

2) Recover information for single instructions

Minimal Blocks



Blocks are single instructions :)

Simplest: Defined Variables

Computed Value: Set of defined variables (Registers for the IR)

Analysis State: Associate a pair of values to each block (*in* and *out*)

Local Update: update the value associated with a block

- From the block itself: variables defined at the exit of the block are those defined when entering plus the ones defined by the block's commands
- From a block to the others: variables defined at beginning of a block are those defined **in every** preceding block

Global Update: all local updates until fixpoint

Then check that each instruction uses variables that are defined either at the beginning of the block or in the block before the current instruction.

Simplest: Defined Variables (a forward analysis)

Computed Value: $\mathcal{P}(R)$

Analysis State:

- Formally $dv : L \longrightarrow \mathcal{P}(R) \times \mathcal{P}(R)$
- More handy $dv_{in} : L \longrightarrow \mathcal{P}(R)$ and $dv_{out} : L \longrightarrow \mathcal{P}(R)$

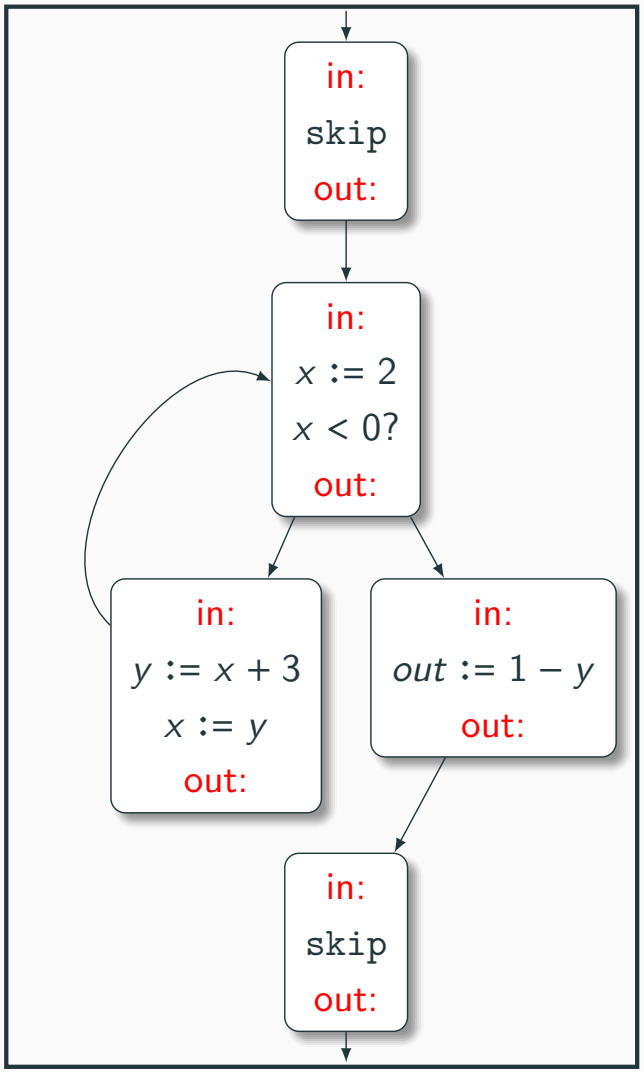
Local Update:

- $lub(dv_{out}(L)) = dv_{in}(L) \cup \{\text{variables defined in } L\}$
- $$lucf(dv_{in}(L)) = \begin{cases} \{in \text{ (register for the input)}\} & \text{if } L \text{ is initial} \\ \bigcap_{(L',L) \in \text{CFG edges}} dv_{out}(L') & \text{otherwise} \end{cases}$$

Global Update: $gu(dv_{in})(L) = lucf(dv_{in}(L))$ and $gu(dv_{out})(L) = lub(dv_{out}(L))$ until fixpoint

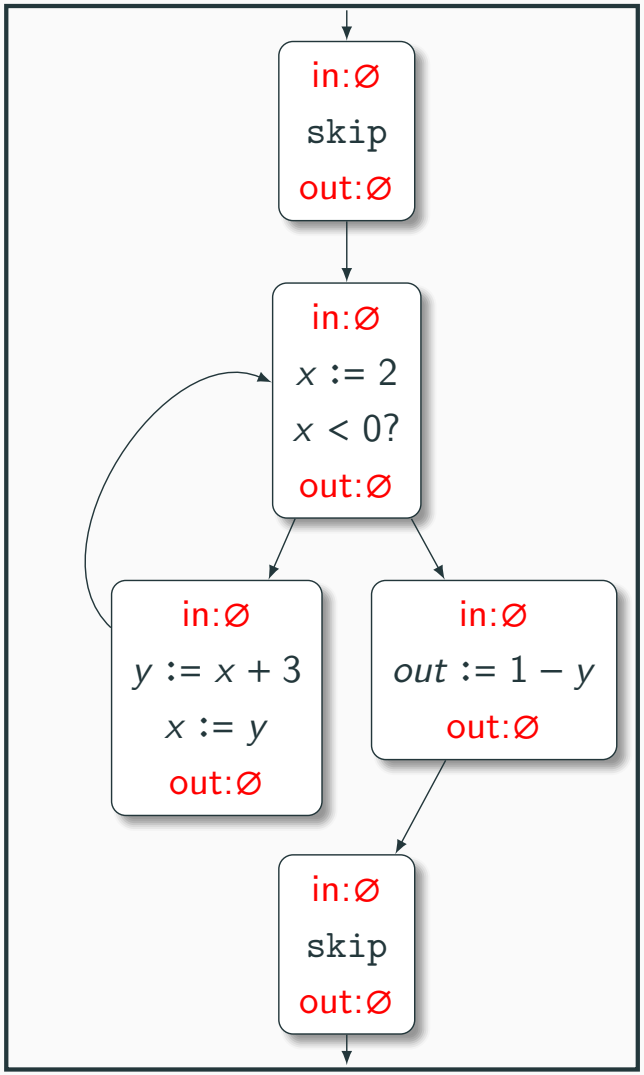
Then check each instruction in blocks.

Defined Variables – Example



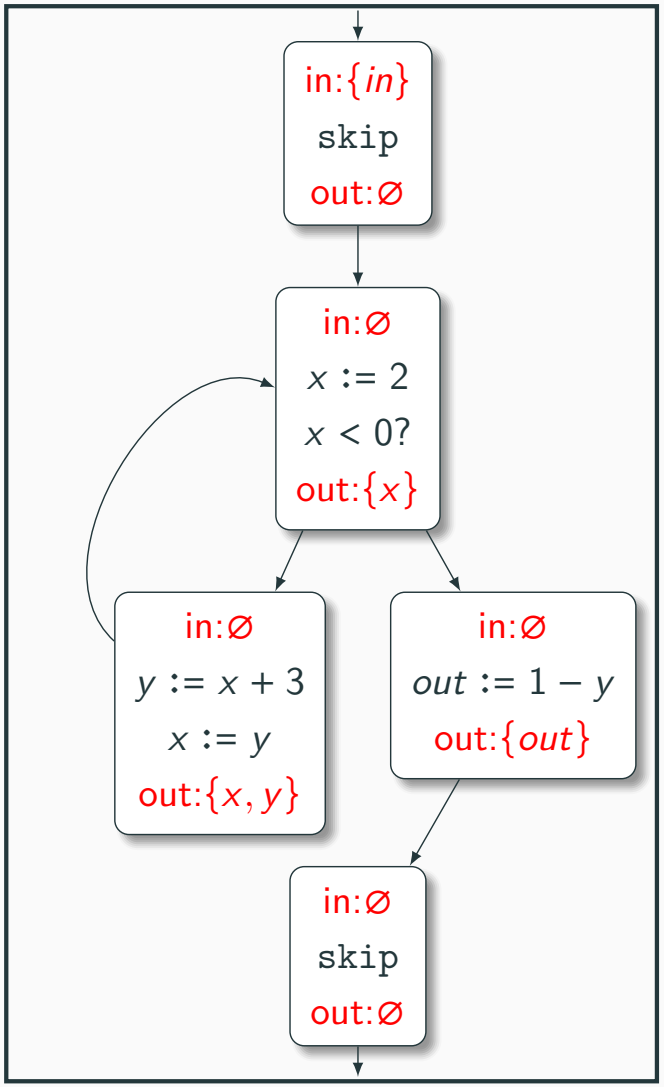
- The variable y in $out := y - 1$ is undefined!
- Notice that the analysis is very coarse grained, we can do better, as we will se in the next lesson

Defined Variables – Example



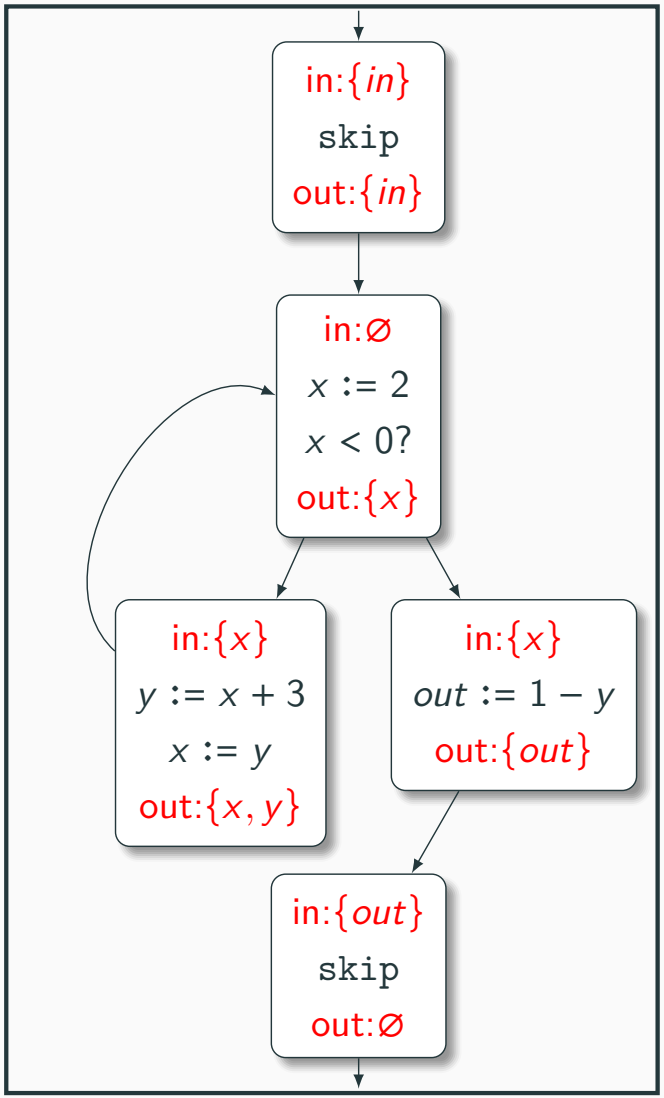
- The variable y in $out := y - 1$ is undefined!
- Notice that the analysis is very coarse grained, we can do better, as we will se in the next lesson

Defined Variables – Example



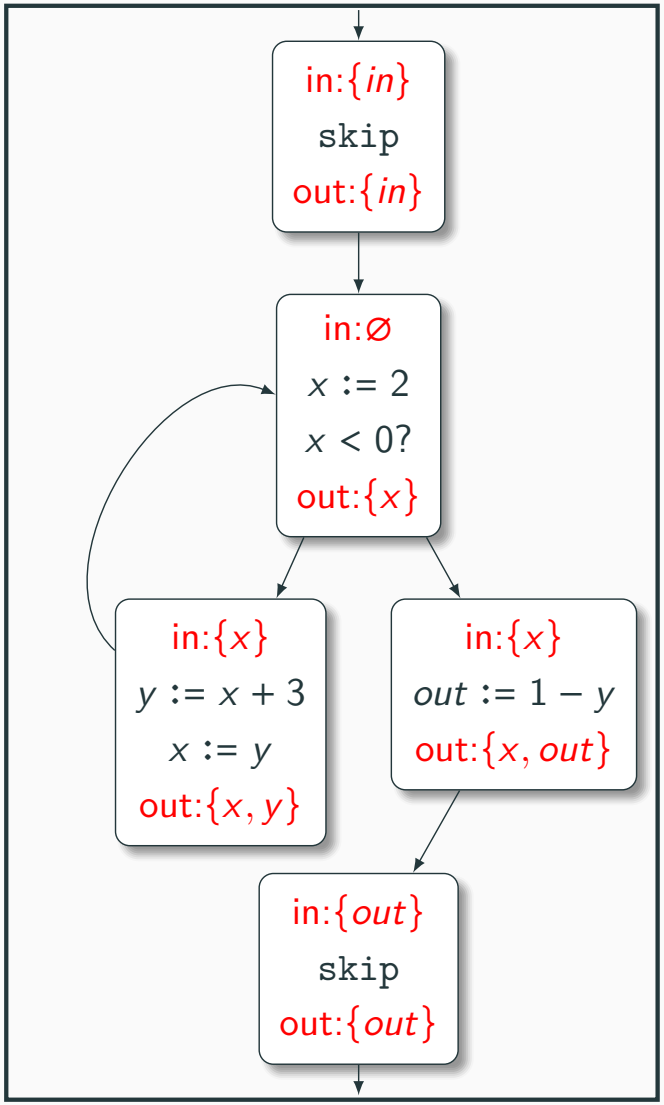
- The variable y in $out := y - 1$ is undefined!
- Notice that the analysis is very coarse grained, we can do better, as we will se in the next lesson

Defined Variables – Example



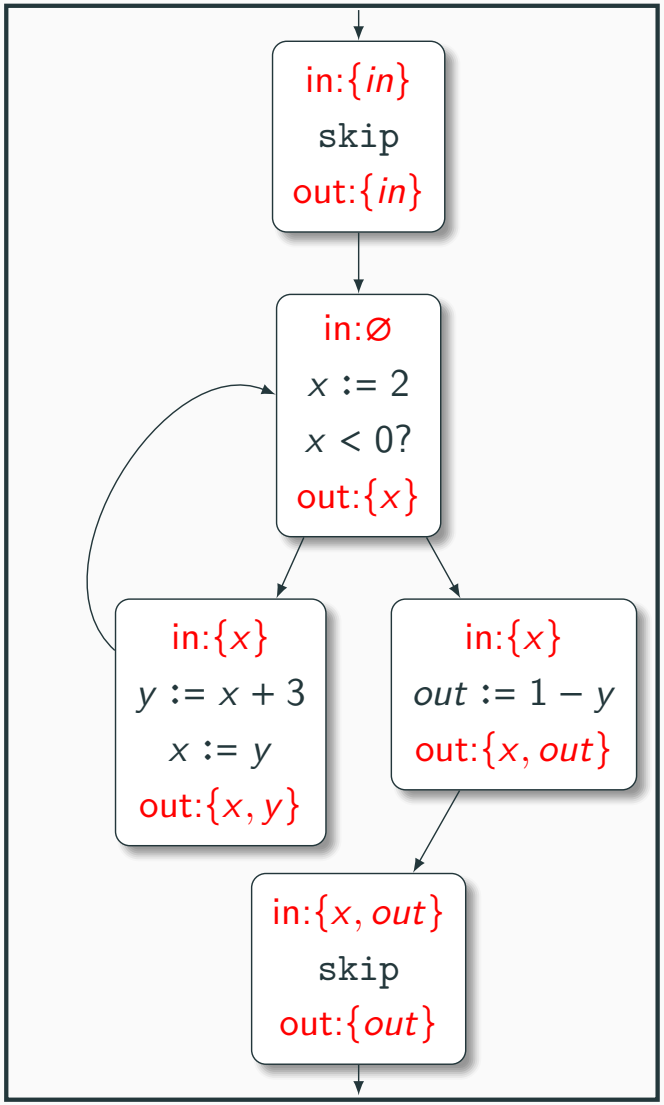
- The variable y in $out := y - 1$ is undefined!
- Notice that the analysis is very coarse grained, we can do better, as we will se in the next lesson

Defined Variables – Example



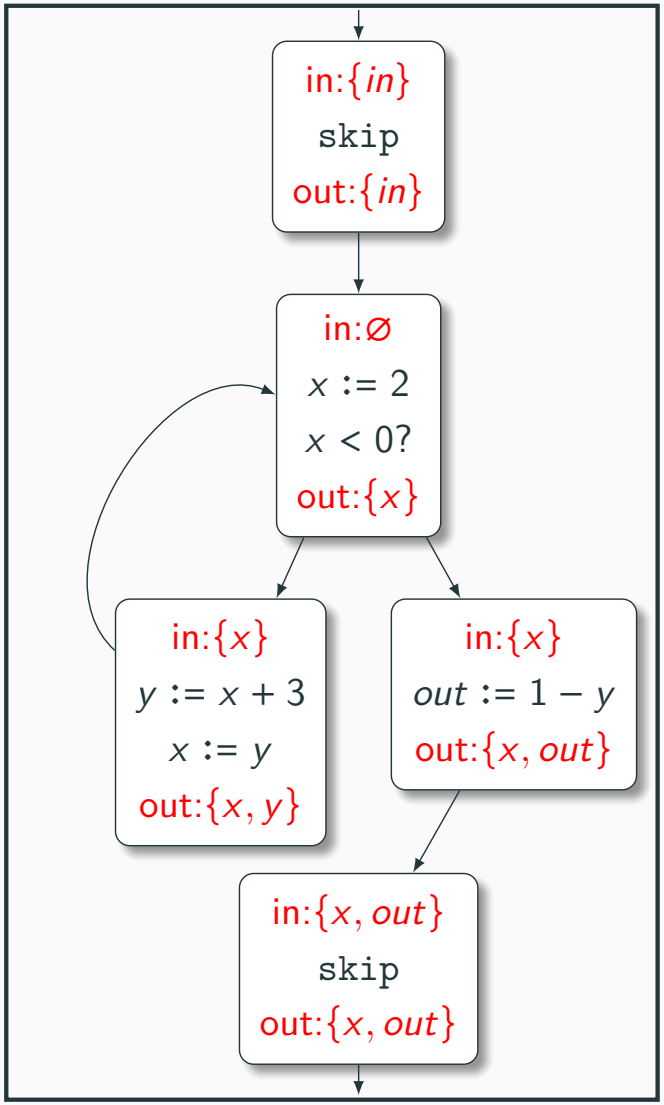
- The variable y in $out := y - 1$ is undefined!
- Notice that the analysis is very coarse grained, we can do better, as we will se in the next lesson

Defined Variables – Example



- The variable y in $out := y - 1$ is undefined!
- Notice that the analysis is very coarse grained, we can do better, as we will se in the next lesson

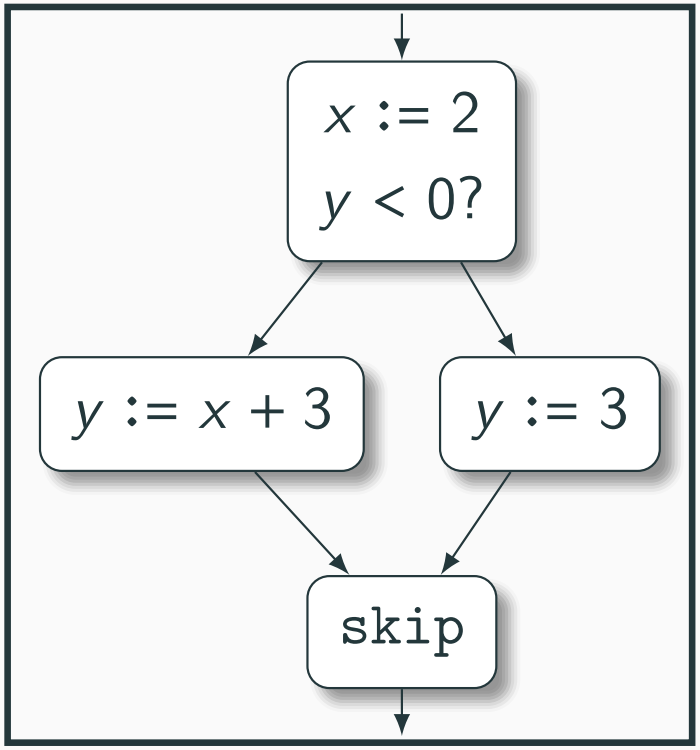
Defined Variables – Example



- The variable y in $out := y - 1$ is undefined!
- Notice that the analysis is very coarse grained, we can do better, as we will se in the next lesson

A More Intricate Case: Live Variables (or Registers)

Roughly: a variable is live if it is defined and not overwritten before its next usage... may or must?



We focus on **may**: we want to be sure that it cannot be live (we can overwrite it)

Liveness – Formally

Liveness is better described in the transition system defined by the small step semantics for a given input.

In MiniRISC, the register r is live before $\langle \xi, b, \sigma_R, \sigma_M \rangle$ if

- $\sigma_R(r)$ is defined, and
- there exists $\langle \xi, c \cdot b', \sigma'_R, \sigma'_M \rangle$ such that:
  - $\langle \xi, b, \sigma_R, \sigma_M \rangle \longrightarrow^* \langle \xi, c \cdot b', \sigma'_R, \sigma'_M \rangle$;
  - c reads the register r;
  - for any $\langle \xi, c' \cdot b'', \sigma''_R, \sigma''_M \rangle$ such that $\langle \xi, b, \sigma_R, \sigma_M \rangle \longrightarrow^* \langle \xi, c' \cdot b'', \sigma''_R, \sigma''_M \rangle \longrightarrow^+ \langle \xi, c \cdot b', \sigma'_R, \sigma'_M \rangle$, c' does not overwrite the register r

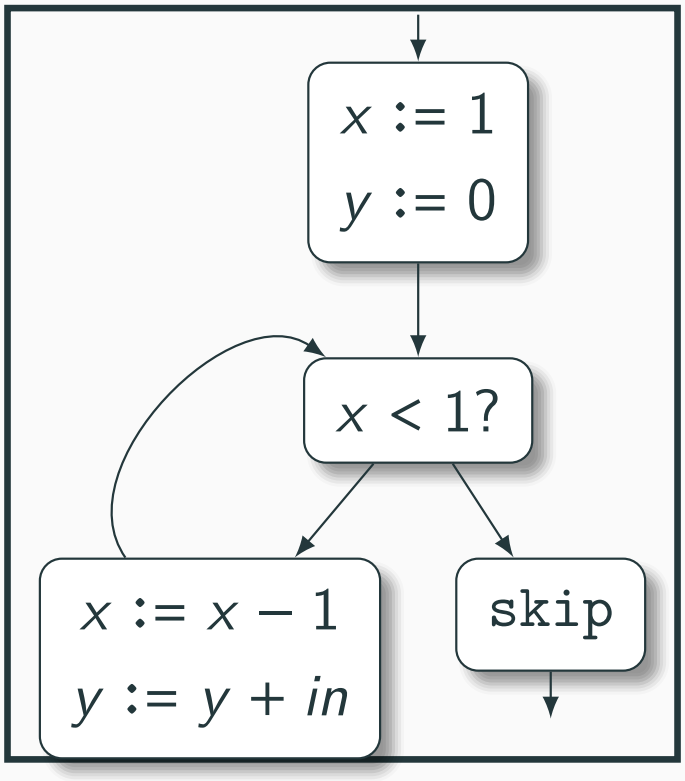
But this information is **not decidable**

May Liveness

May Liveness is instead defined on the CFG of the program

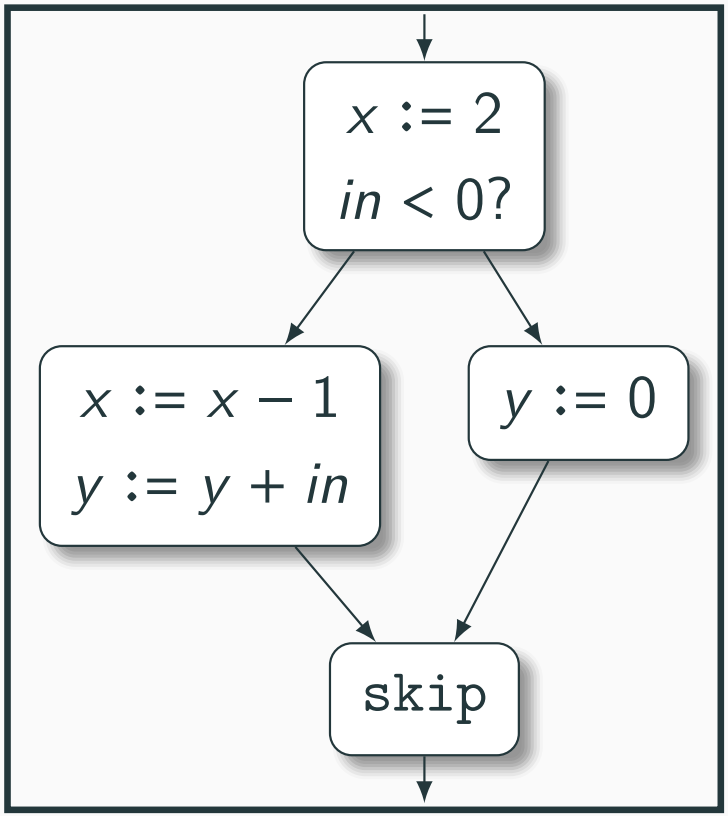
- Instead of considering all the possible (infinite) transition systems for the program (one for any input integer), we consider its CFG, which approximates these concrete transition systems
- Instead of considering the points in time of the concrete executions, we consider positions in the code

Points in Time to Points in the Code – Approximation



Is y live at the exit of the guard block $x < 1?$

Different Inputs and Different Executions – Approximation



Is x live at the exit of the initial block?

Why Liveness Analysis? Register Reuse

Registers are not infinite in the real world, nor they will be infinite in our target system

- Not a problem, we have memory
- But accessing memory is time consuming, better to avoid it if possible
- Reuse registers if the current value is not used later
- If different registers are not live together, they can be merged (mapped to the same concrete register of the target machine)

How to perform Liveness Analysis (a backward analysis)

Computed Value: $\mathcal{P}(R)$

Analysis State:

- $lv_{in} : L \longrightarrow \mathcal{P}(R)$ and $lv_{out} : L \longrightarrow \mathcal{P}(R)$

Local Update:

- $lub(lv_{in}(L)) = \{r \text{ used in } L\} \cup (lv_{out}(L) \setminus \{r \text{ defined in } L\})$
- $lucf(lv_{out}(L)) = \begin{cases} \{out \text{ (register for the output)}\} & \text{if } L \text{ is final} \\ \bigcup_{(L,L') \in \text{CFG edges}} dv_{in}(L') & \text{otherwise} \end{cases}$

Global Update applies all the local updates as before.

- The *used* is informal, there is some missing detail if you use maximal blocks
- The *defined* is missing a detail for the initial node

Project Fragment

- Write a function for checking that no register is ever used before being initialized with some value in a MiniRISC CFG (mind the initial register *in* which is always initialized, and *out* which is always used – if you prefer, you can perform this task on the MiniImp CFG of the program)
- Write a module containing a function for computing liveness analysis on MiniRISC CFGs (we will use it later)
- Detail your implementation choices in the report