ETH Zürich
Chair of Software Engineering

Software Verification

Sebastian Nanz

Lecture 7: Program Analysis

ETH Zürich

Chair of Software Engineering

Program Analysis

An Informal Overview

Applications of program analysis

Two important application fields of program analysis:

- Program optimizations
  - Program analysis provides techniques for transforming programs during compilation to avoid redundant computations
- Verification
  - Program analysis can provide warnings about possible unintended program behavior (e.g. buffer overflows) or prove programs free from such behavior

Program analysis is a static technique, i.e. analyses are performed without running the program.

How can this work?

We are interested to have questions such as the following answered by an analysis:

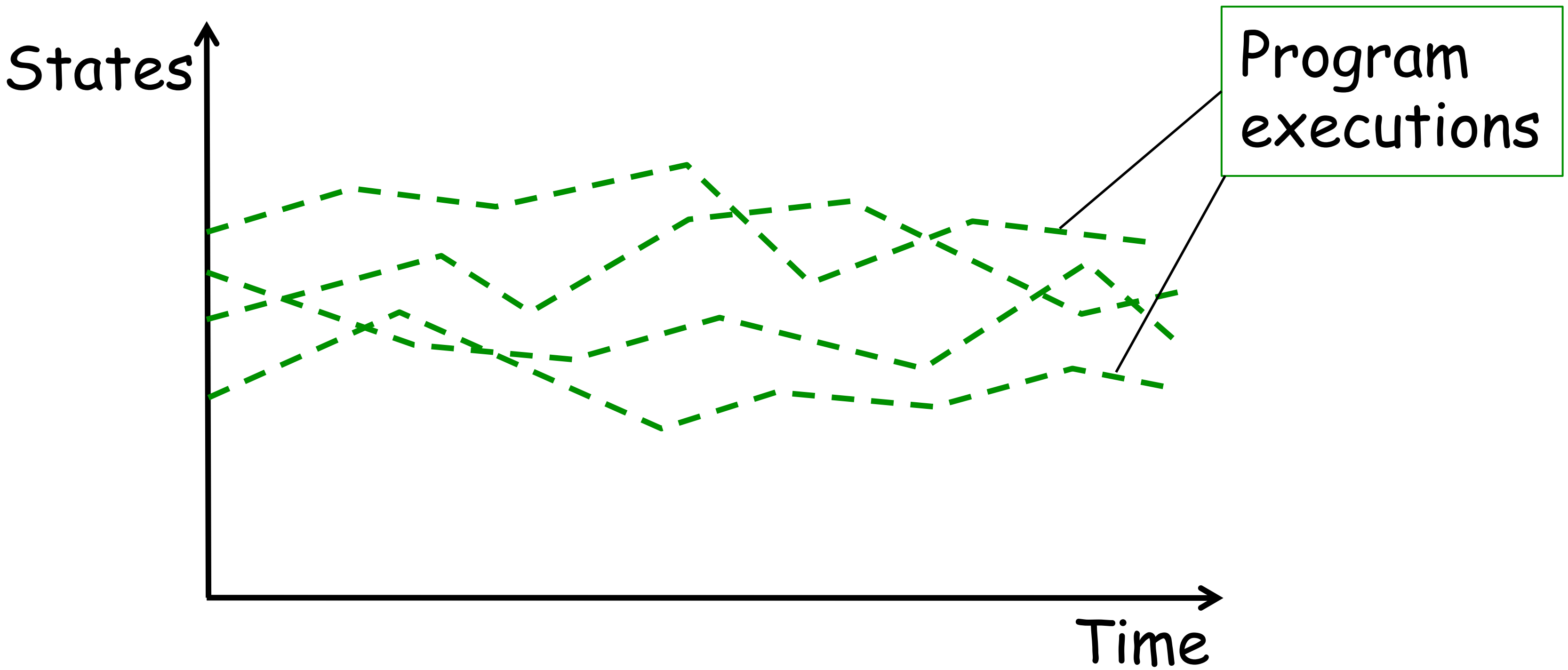
- Will the value of variable x be read in the future?
- Can buffer b overflow in line i of the program?
- Can void dereferencing occur during execution? etc.

From computability theory (Rice's theorem) we know however: "All **non-trivial** questions about the behavior of **Turing-complete** programs are **undecidable**." So, how can this work?

Key idea: We can settle for approximative answers, as explained on the following slides.

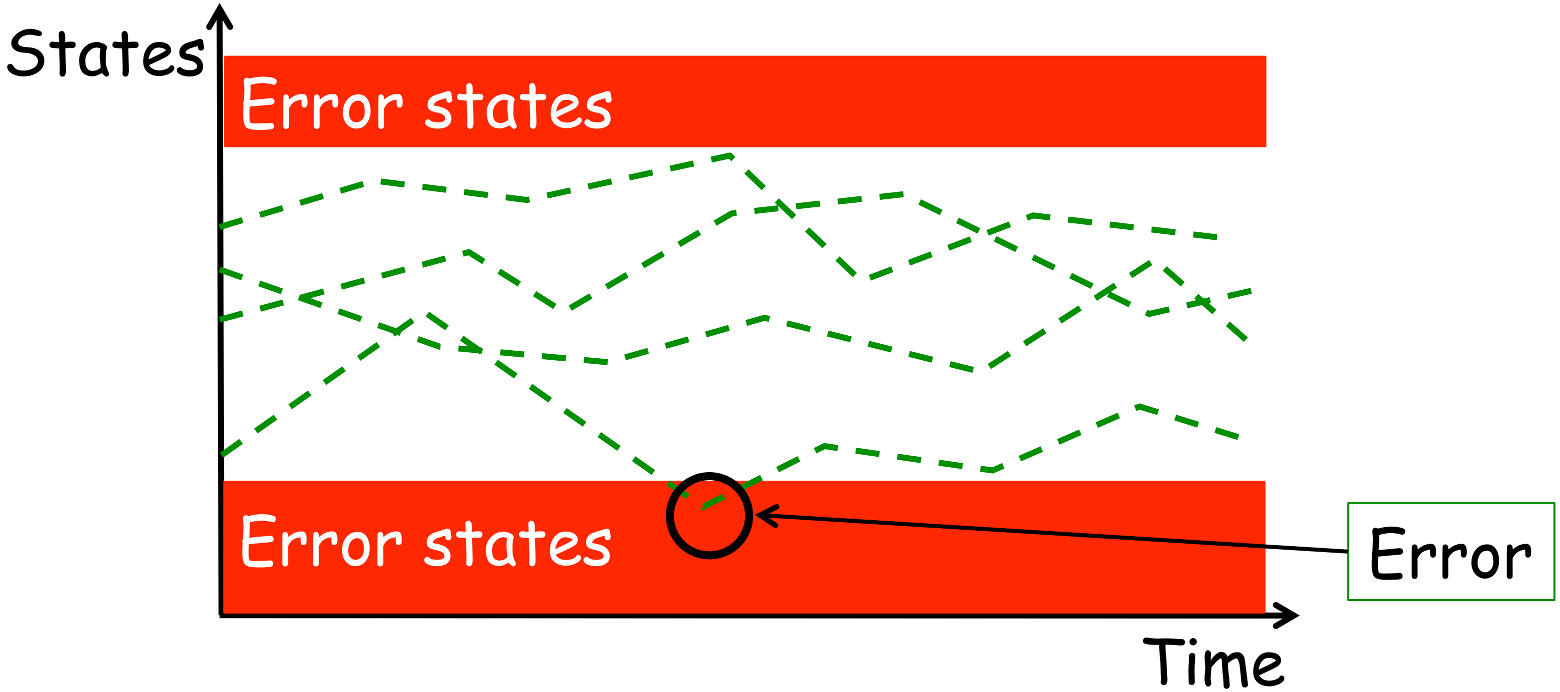
The approach

Assume we depict the set of all possible concrete executions of a program as trajectories through the state space:



Safety properties

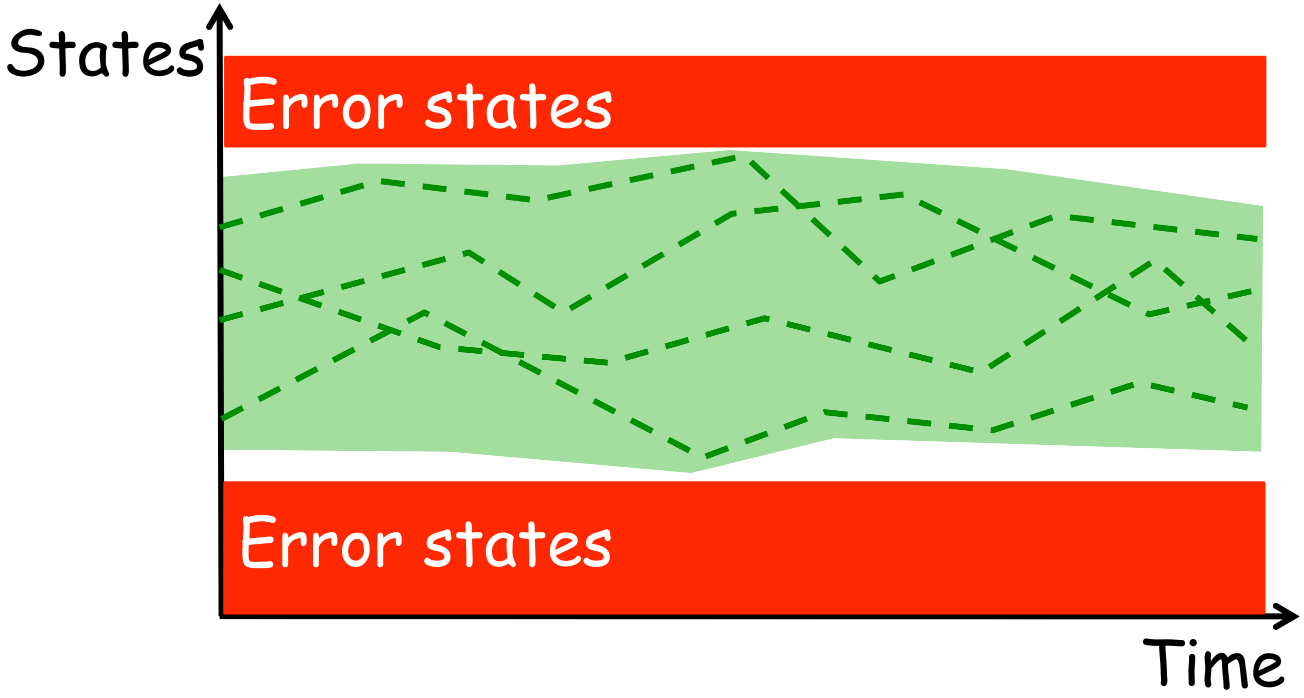
Many program analysis questions can be stated as safety properties, which express that no possible execution can enter an error state (e.g. a state where "buffer b overflows").



Approximations

As mentioned, proving that a non-trivial safety property holds is undecidable for the set of concrete executions.

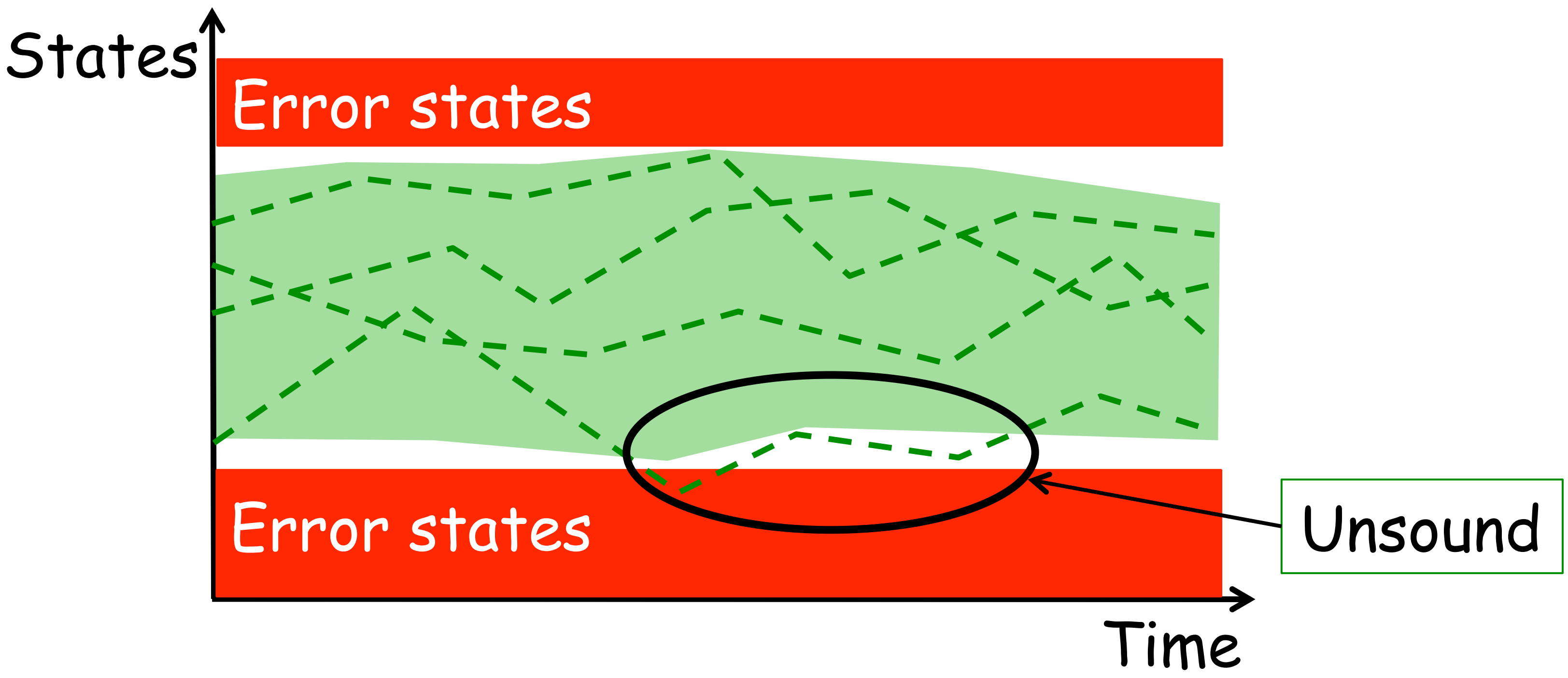
Instead we compute an abstraction of the behavior which over-approximates all concrete executions:



Soundness of the analysis

We want our analysis to be sound so that all possible program executions are captured.

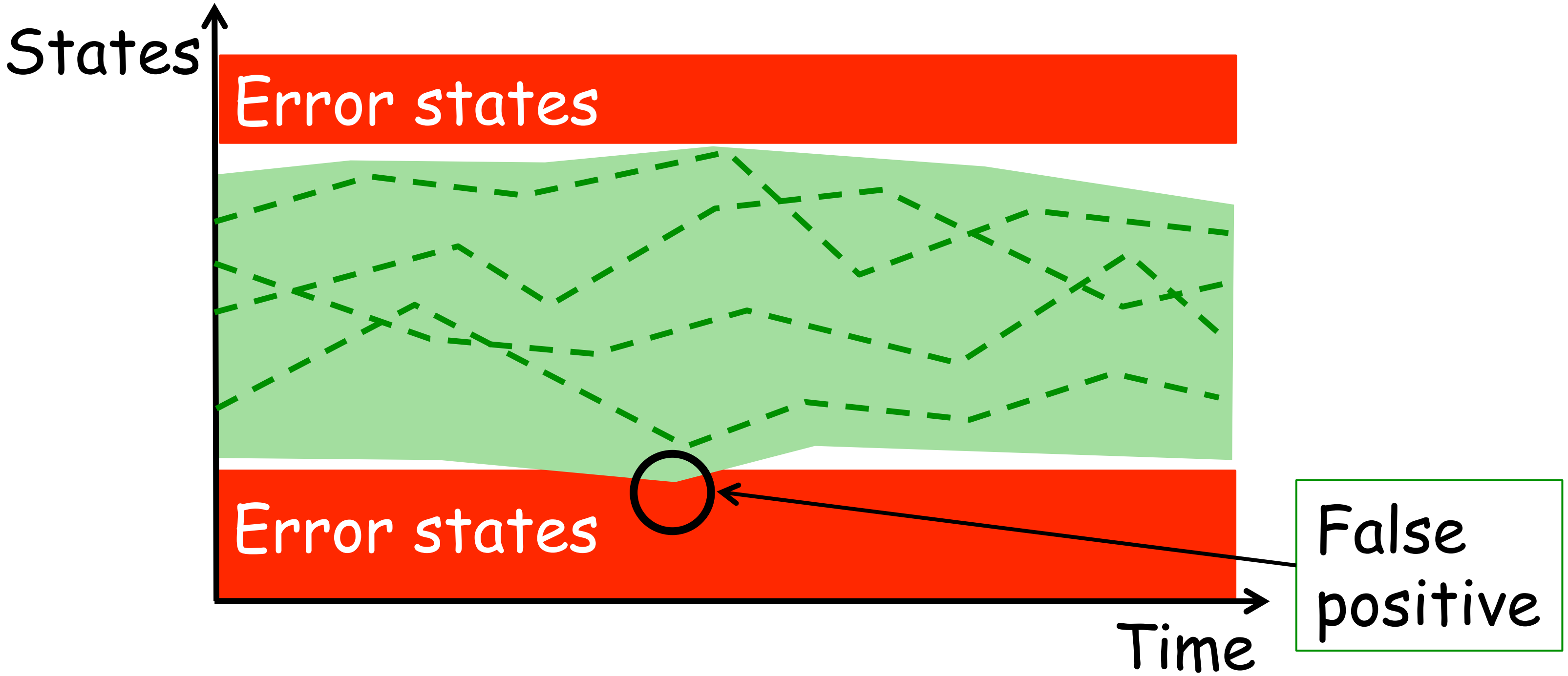
Example of an unsound analysis:



Precision

We also want our analysis to be as precise as possible. Otherwise, if there are too many false alarms, the analysis will be unusable.

Errors reported by the analysis which cannot occur in a concrete execution are called false positives.



Precision vs. efficiency

While we want our analysis to be precise, we often have to trade off precision with efficiency:

- While a very precise analysis might still be computable, it might need to run for too long to be practical.
- Imprecise analyses leave us with a large number of warnings, and manual checking has to show whether a particular warning is an error or a false positive.

Defining new program analyses is thus an art that tries to balance precision and efficiency.

Types of program analyses

Several types of program analyses have been established:

- Data flow analysis
- Control flow analysis
- Abstract interpretation
- Type systems

In this lecture we will focus on data flow analysis, and in the next on abstract interpretation.

Summary

Program analysis provides a set of static techniques for computing sound abstractions of the run-time behavior of a program.

ETH Zürich

Chair of Software Engineering

Data Flow Analysis

Preliminaries

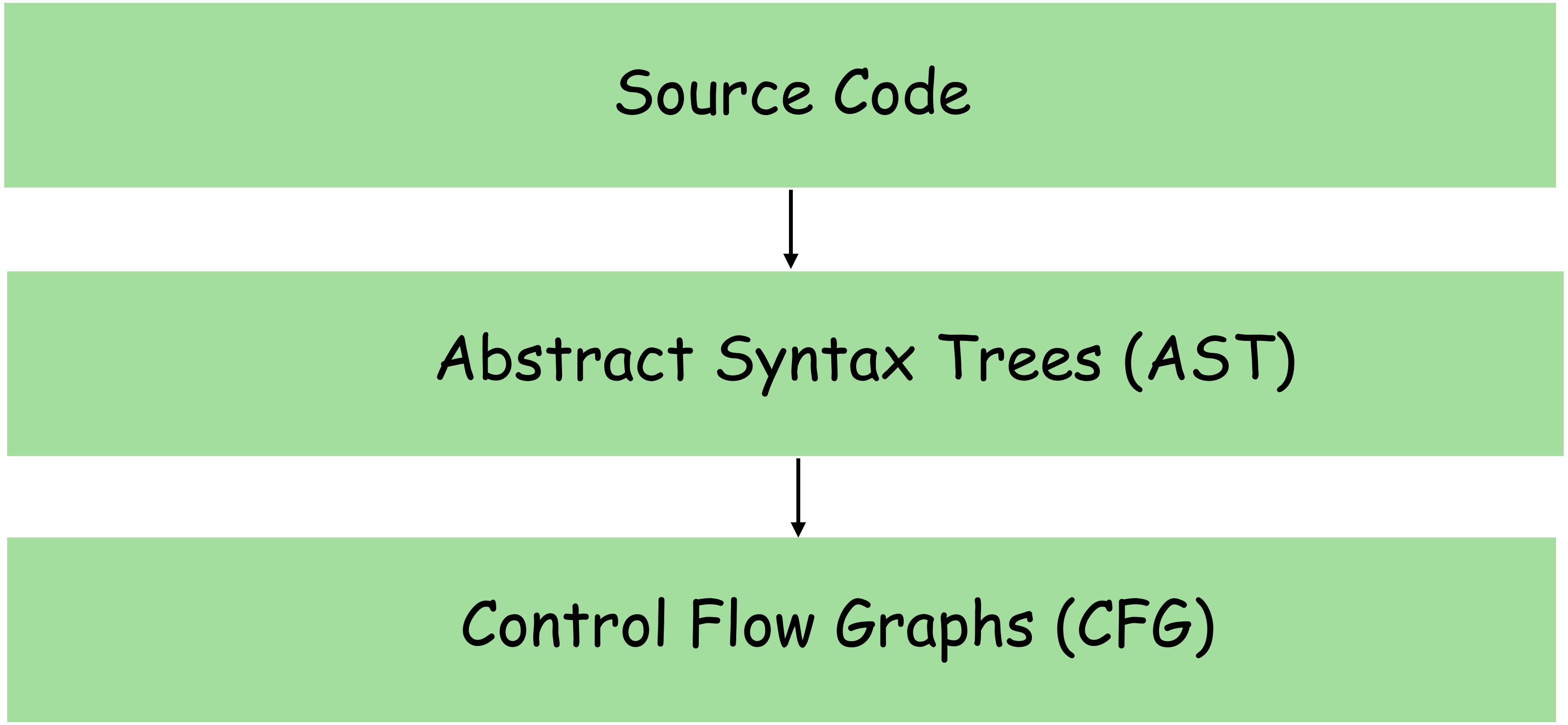
Data flow analysis

Data flow analysis is a technique to derive information about the possible program values produced at a specific program point.

Data flow analyses take as an input the control flow graph of a program, and proceed by examining how data values are changed when being propagated along its edges (hence the name "data flow").

Obtaining control flow graphs

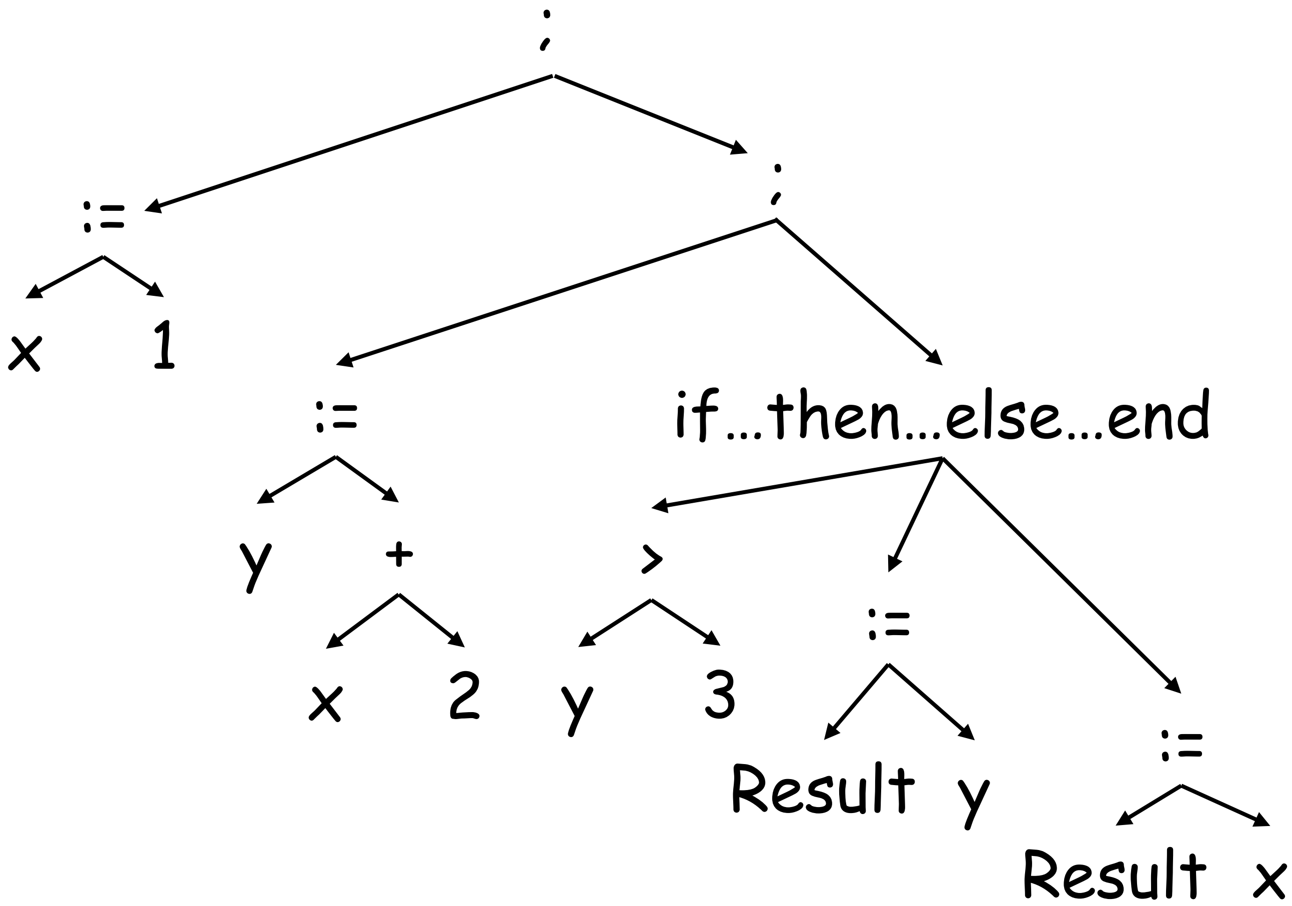
For imperative programs, a control flow graph can be computed straightforwardly from the abstract syntax tree of the program. (In more complicated cases, there are advanced techniques for it: control flow analysis.)



Abstract syntax tree

Abstract syntax tree: tree representation of the syntactic structure of the source code.

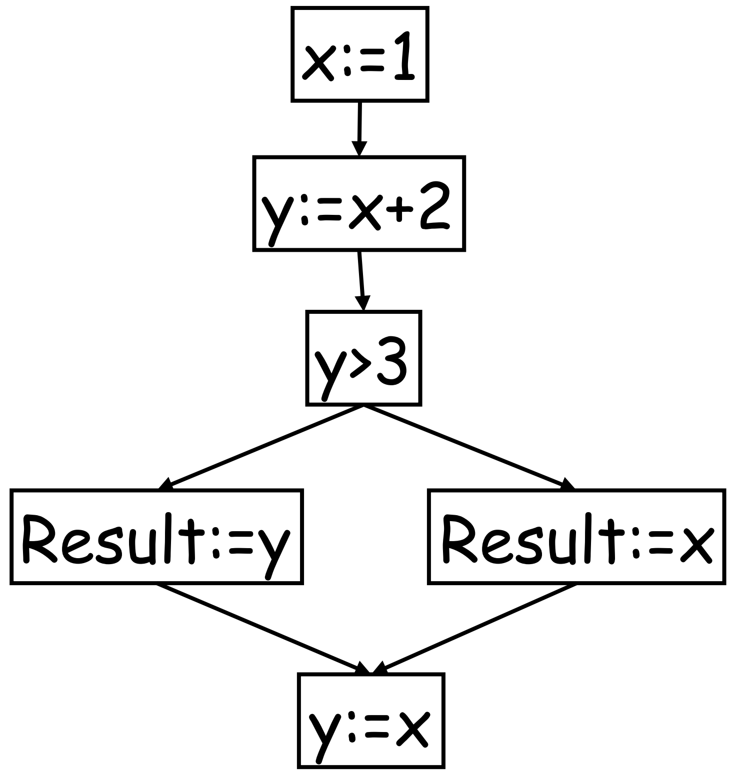
```
x := 1
y := x + 2
if (y > 3) then
        Result := y
else
        Result := x
end
```



Control flow graph

Control flow graph: graph representation of all possible execution paths of a program.

```
x := 1
y := x + 2
if (y > 3) then
        Result := y
else
        Result := x
end
y := x
```

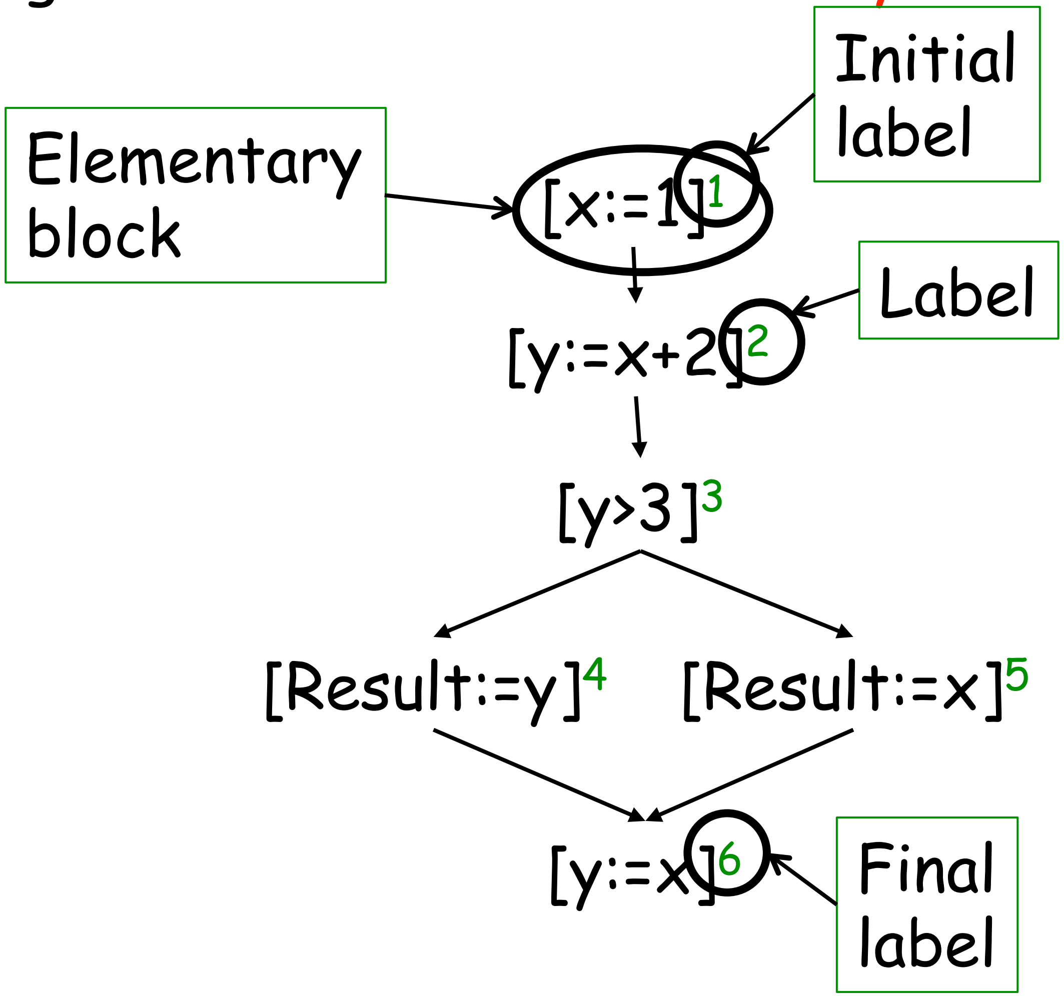


Labels

In order to be able to refer to specific program points, program analyses introduce labels into the program.

The labeled program fragments are called elementary blocks.

```
[x := 1]1
[y := x + 2]2
if [y > 3]3 then
        [Result := y]4
else
        [Result := x]5
end
[y := x]6
```



ETH Zürich
Chair of Software Engineering

Data Flow Analysis

Live Variables Analysis

Live variables analysis

We present a first example of a data flow analysis: live variables (LV) analysis.

- A variable is live at the exit from a block if there is some path from the block to a use of the variable that does not redefine the variable.
- The aim of the live variables analysis is to determine

> "For each program point, which variables *may* be live at the exit from the point."

Example:

[x :=2]¹; [y:=4]²; [x:=1]³;

if [y>x]⁴ **then** [z:=y]⁵ **else** [z:=2*z]⁶ **end**; [x:=z]⁷

Is the variable x live at the exit from block 1?

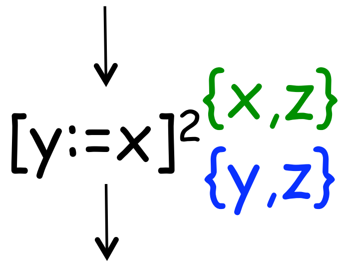
Live variables analysis

Analysis idea:

- Record sets of *possibly live* variables
- Distinguish entry and exit of blocks
- Work backwards

(LV1) Blocks:

$$LV_{entry} = (LV_{exit} \setminus \text{"assigned"}) \cup \text{"used"}$$

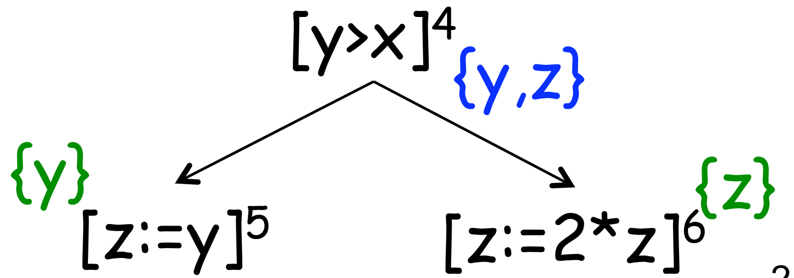


"assigned" – variable that gets assigned in the block

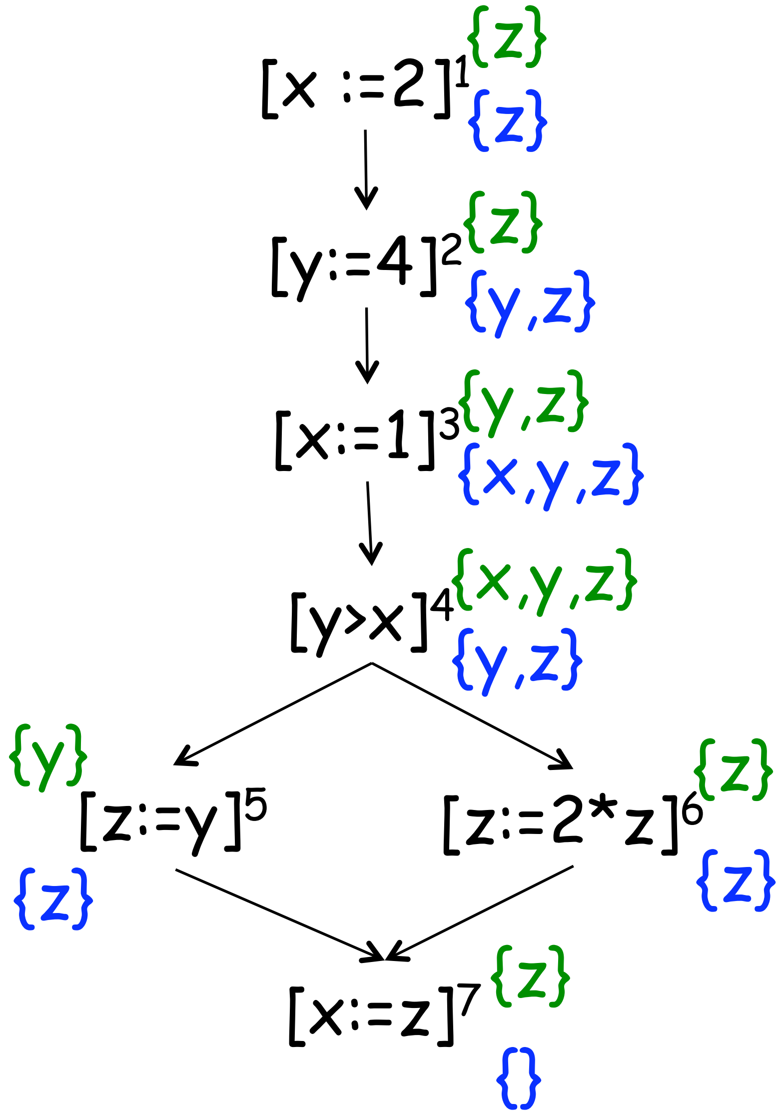
"used" – variables that are used in the block

(LV2) Edges:

$$LV_{exit}(4) = LV_{entry}(5) \cup LV_{entry}(6)$$



Example: Live variables analysis



Application: Dead code elimination

An assignment $[x := a]^l$ is dead if the value of x is not used before it is redefined.

Goal: Eliminate dead assignments from programs.

Example:

We know that $x \notin LV_{exit}(1) = \{z\}$, i.e. variable x not used before it is redefined. Therefore block 1 is dead and can be eliminated:

~~$[x := 2]^1$~~; $[y:=4]^2$; $[x:=1]^3$;
if $[y>x]^4$ **then** $[z:=y]^5$ **else** $[z:=2*z]^6$ **end**; $[x:=z]^7$

How to formalize the analysis idea?

- (LV1) and (LV2) specify equations over a set of variables $LV_{entry}(l)$ and $LV_{exit}(l)$ for any label l.
- This equation system can be solved with standard algorithms (discussed later).
- The equation system itself can be specified more formally, as done on the next slide.

This specification consists of two parts:

1. The definition of the equation system.
2. Auxiliary functions kill and gen, which specify the analysis information removed (killed) and added (generated) when passing through an elementary block.

Formalization: Data flow equations

1. The data flow equations:

$$LV_{exit}(l) = \bigcup_{(l,\, l') \in CFG} LV_{entry}(l')$$

(and $LV_{exit}(l) = \{\}$ if l is the final label)

$$LV_{entry}(l) = (LV_{exit}(l) \setminus kill_{LV}(l)) \cup gen_{LV}(l)$$

2. The auxiliary kill and gen functions:

$kill_{LV}([x:=a]^l) = \{x\}$

$kill_{LV}([b]^l) = \{\}$

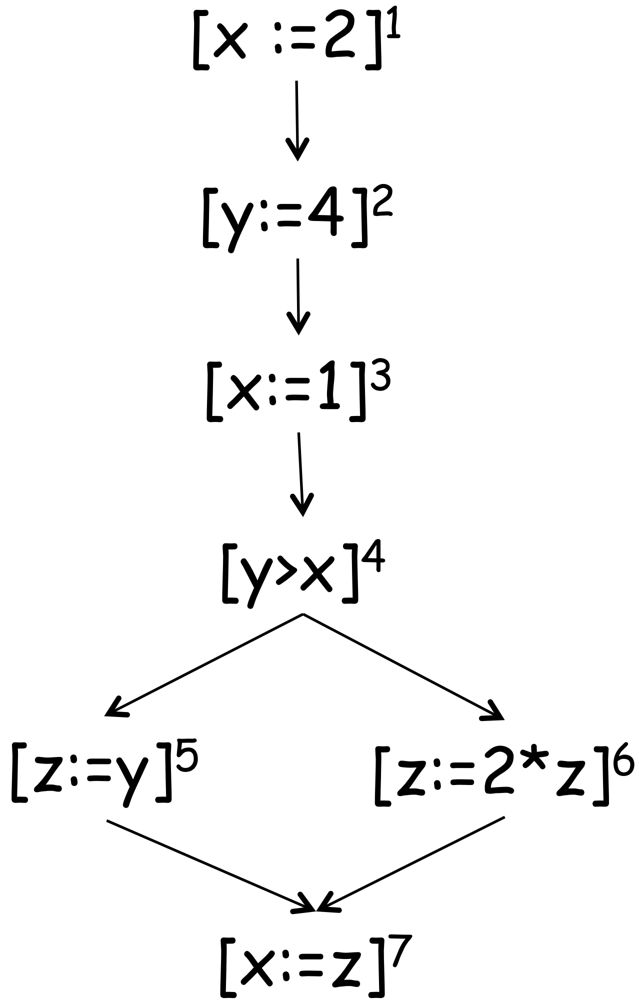
$gen_{LV}([x:=a]^l) = \{y \mid y \text{ is a free variable in } a\}$

$gen_{LV}([b]^l) = \{y \mid y \text{ is a free variable in } b\}$

Example: Equation system for LV analysis

$LV_{entry}(1) = LV_{exit}(1) \setminus \{x\}$
$LV_{entry}(2) = LV_{exit}(2) \setminus \{y\}$
$LV_{entry}(3) = LV_{exit}(3) \setminus \{x\}$
$LV_{entry}(4) = LV_{exit}(4) \cup \{x, y\}$
$LV_{entry}(5) = (LV_{exit}(5) \setminus \{z\}) \cup \{y\}$
$LV_{entry}(6) = (LV_{exit}(6) \setminus \{z\}) \cup \{z\}$
$LV_{entry}(7) = (LV_{exit}(7) \setminus \{x\}) \cup \{z\}$

$LV_{exit}(1) = LV_{entry}(2)$
$LV_{exit}(2) = LV_{entry}(3)$
$LV_{exit}(3) = LV_{entry}(4)$
$LV_{exit}(4) = LV_{entry}(5) \cup LV_{entry}(6)$
$LV_{exit}(5) = LV_{entry}(7)$
$LV_{exit}(6) = LV_{entry}(7)$
$LV_{exit}(7) = \{\}$



Data Flow Analysis

Equation Solving

Fixed point solutions

- The equation system of the example defines the 14 sets

$$LV_{entry}(1), LV_{entry}(2), \ldots, LV_{exit}(7)$$

in terms of each other.

- When writing $\underline{LV}$ for the vector of these 14 sets, the equation system can be written as a function F where

$$\underline{LV} = F(\underline{LV})$$

- Using a vector of variables $\underline{X} = (X_1, \ldots, X_{14})$, the function can be defined as

$$F(\underline{X}) = (f_1(\underline{X}), \ldots, f_{14}(\underline{X}))$$

where for example

$$f_{11}(X_1, \ldots, X_{14}) = X_5 \cup X_6$$

- From the above equation it is clear that the solution $\underline{LV}$ we are looking for is the (least) fixed point of the function F.

Partially ordered sets

For any analysis, we are interested in expressing that one analysis result is "better" (more precise) than another.

In other words, we want the analysis domain to be partially ordered.

A partial ordering is a relation $\sqsubseteq$ that is

- reflexive: $\forall d : d \sqsubseteq d$
- transitive: $\forall d_1, d_2, d_3 : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ imply $d_1 \sqsubseteq d_3$
- anti-symmetric: $\forall d_1, d_2 : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$ imply $d_1 = d_2$

A partially ordered set $(D, \sqsubseteq)$ is a set D with a partial ordering $\sqsubseteq$.

Least element: $a \in D$ s.t. $d \sqsubseteq a$ and $d \in D$ implies $d = a$.

Examples: Real numbers $(\mathbf{R}, \leq)$, power sets $(\wp(S), \subseteq)$, ...

Equation solving

- How can we obtain the least fixed point practically?
- For the least element $\bot \in D$ of a partially ordered set D we have

$$\bot \sqsubseteq F(\bot)$$

- By induction we have for all $n \in \mathbf{N}$

$$F^n(\bot) \sqsubseteq F^{n+1}(\bot)$$

- All elements of the sequence are in the domain D, and therefore, if D is finite, there exists an $n \in \mathbf{N}$ such that

$$F^n(\bot) = F(F^n(\bot))$$

(Requires special properties of D and F, shown later.)

- But this means that $F^n(\bot)$ is a fixed point! (And indeed a least fixed point.)

Chaotic iteration

- Implementing the iteration algorithm naively is computationally too expensive.
- More efficient algorithms exist, and are variants of the simplest scheme which is called chaotic iteration:

-- Initialization

$X_1 := \bot; \ldots; X_n := \bot$

-- Iteration

while $X_j \neq F_j(X_1, \ldots, X_n)$ for some j **do**

$X_j := F_j(X_1, \ldots, X_n)$

end

- A more advanced algorithm is the worklist algorithm, which keeps a list of edges of the control flow graph to indicate which items are in need of recomputation.

A worklist algorithm for solving the equations

Input:

A set of live variables analysis equations

Output:

The least solution to the equations: LV_{exit}

Data structures:

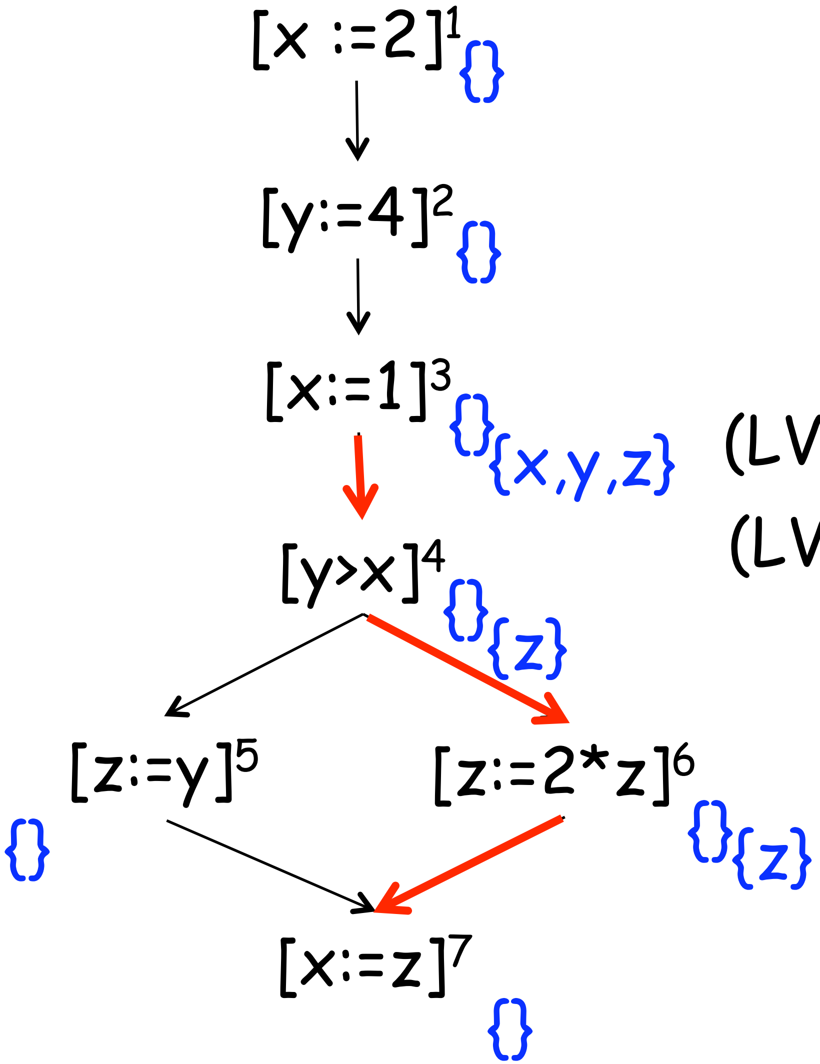
- The current analysis result for block exits: LV_{exit}
- The worklist W: A list of pairs (l, l') indicating that the current analysis result has changed at the entry to the block l' and hence the information must be recomputed for block l.

A worklist algorithm for solving the equations

```
-- Initialization
W := nil
for all (l, l') ∈ CFG do W := cons((l, l'),W) end
for all labels l do LV_exit (l) := {} end
-- Worklist loop
while W ≠ nil do
  (l, l') := head(W)
  W := tail(W)
   if (LV_exit (l') \ kill(l')) ∪ gen(l') ⊈ LV_exit (l) then
      LV_exit (l) := LV_exit (l) ∪ (LV_exit (l') \ kill(l')) ∪ gen(l')
   end
   for all l'' with (l'', l)  ∈ CFG do W := cons((l'', l),W) end
end
```

Note: $(LV_{exit}(l') \setminus kill(l')) \cup gen(l') = LV_{entry}(l')$

Example: Working of the algorithm



W = (6,7),(5,7),...
W = (4,6),(5,7),...
W = (3,4),(5,7),...

$(LV_{exit}(7) \setminus \{x\}) \cup \{z\} \not\subseteq LV_{exit}(6)$

$(LV_{exit}(6) \setminus \{z\}) \cup \{z\} \not\subseteq LV_{exit}(4)$

$LV_{exit}(4) \cup \{x,y\} \not\subseteq LV_{exit}(3)$

ETH Zürich

Chair of Software Engineering

Data Flow Analysis

Reaching Definitions Analysis

Reaching definitions analysis

Another example of a data flow analysis: reaching definitions (RD) analysis.

- The aim of the RD analysis is to determine

> "For each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path."

Note: The word "definition" is used for "assignment"

Example:

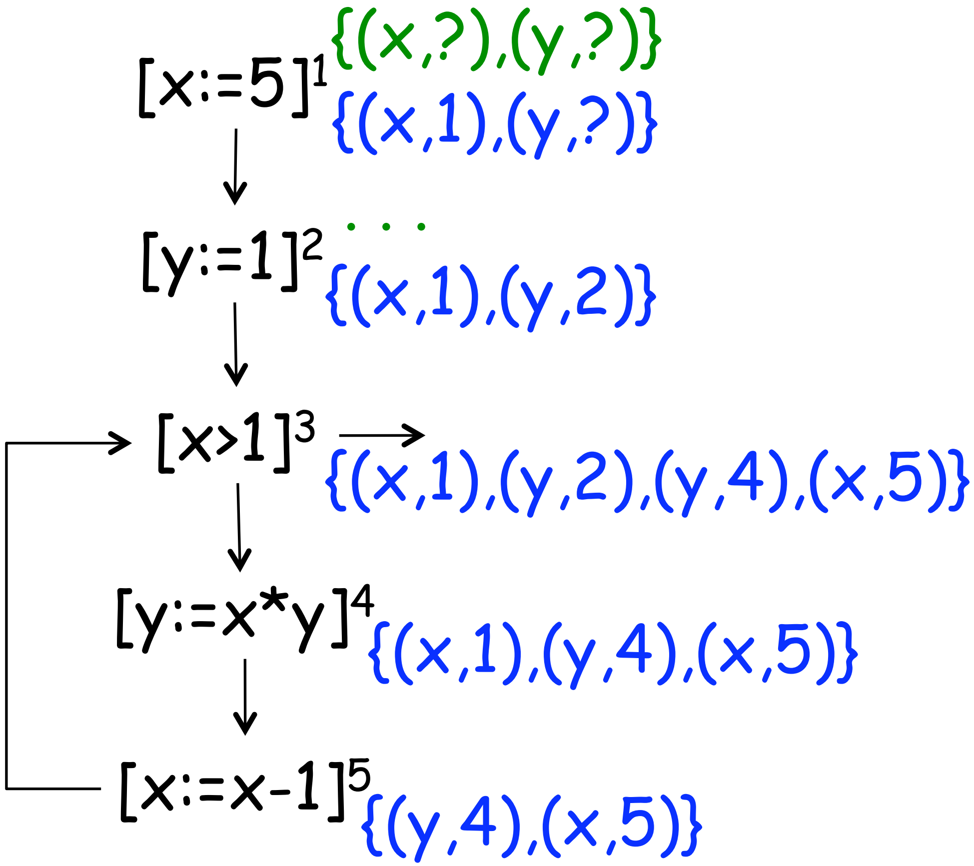
$[x:=5]^1$; $[y:=1]^2$; **while** $[x>1]^3$ **do** $[y:=x*y]^4$; $[x:=x-1]^5$ **end**

Which assignments may reach program point 5?

Reaching definitions analysis

Idea: analysis domain $\wp(Var_* \times Lab_*)$, work forward

- We write (x, l) to describe a definition of x in block l
- We write (x, ?) to describe that x is uninitialized



Formalization: Data flow equations

The reaching definitions analysis can be specified similarly to the scheme for LV analysis.

$$RD_{entry}(l') = \bigcup_{(l,\, l') \in CFG} RD_{exit}(l)$$

(and $RD_{entry}(l) = \{(x,?) \mid x \text{ is a free variable in the program}\}$ if l is the initial label)

$$RD_{exit}(l) = (RD_{entry}(l) \setminus kill_{RD}(l)) \cup gen_{RD}(l)$$

$$kill_{RD}([x:=a]^l) = \{(x,?)\} \cup \{(x,l') \mid \text{block } l' \text{ assigns to } x\}$$

$$kill_{RD}([b]^l) = \{\}$$

$$gen_{RD}([x:=a]^l) = \{(x,l)\}$$

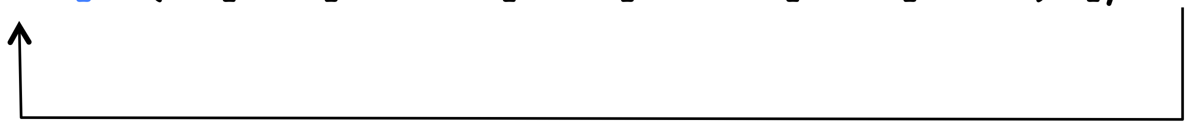
$$gen_{RD}([b]^l) = \{\}$$

Use-Definition and Definition-Use chains

Sometimes it is convenient to directly link statements that produce values to statements that use them and vice versa

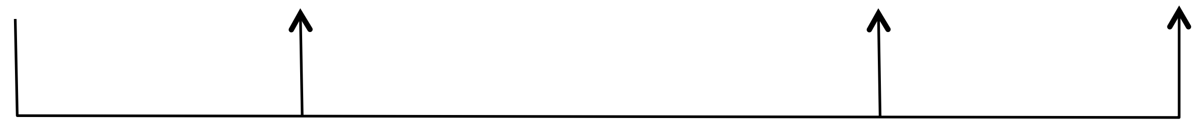
- Use-Definition chains (UD chains): each use of a variable is linked to all assignments that may reach it

[x:=0]^1; **[x:=3]^2**; (**if** [z=x]^3 **then** [z:=0]^4 **else** [z:=**x**]^5 **end**); [y:=**x**]^6; [x:=y+z]^7



- Definition-Use chains (DU chains): each assignment to a variable is linked to all uses of it

[x:=0]^1; **[x:=3]^2**; (**if** [z=x]^3 **then** [z:=0]^4 **else** [z:=x]^5 **end**); [y:=x]^6; [x:=y+z]^7

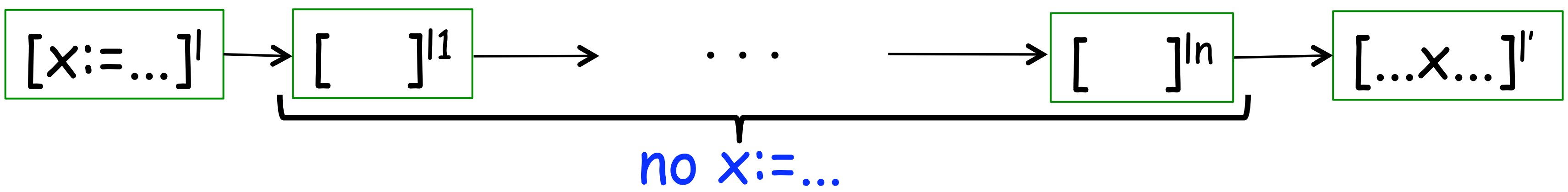


Definition of UD chains

- $UD(x, l')$ returns all the labels where an occurrence of x at l' may have obtained its value.

$$UD(x, l') = \{l \mid [x:=a]^l \text{ and } clear(x,l,l')\} \cup \{? \mid clear(x, l_{init}, l)\}$$

where the predicate $clear(x,l,l')$ describes a definition clear path: none of the intermediate blocks on a path from l to l' contains an assignment to x but block l assigns to x and block l' uses x.



- Can be computed with Reaching Definitions:

$UD(x,l) = \{l' \mid (x, l') \in RD_{entry}(l)\}$ if x is used in block l, else {}

Definition of DU chains

- DU(x, l) returns all the labels where the value assigned to x at l may be used.

- Can be computed from UD chains:

$$DU(x, l) = \{l' \mid l \in UD(x, l')\}$$

Data Flow Analysis

Existence of solutions

Fixed point solutions (recap)

- The equation system of the example defines the 14 sets

$$LV_{entry}(1), LV_{entry}(2), \ldots, LV_{exit}(7)$$

in terms of each other.

- When writing $\underline{LV}$ for the vector of these 14 sets, the equation system can be written as a function F where

$$\underline{LV} = F(\underline{LV})$$

- Using a vector of variables $\underline{X} = (X_1, \ldots, X_{14})$, the function can be defined as

$$F(\underline{X}) = (f_1(\underline{X}), \ldots, f_{14}(\underline{X}))$$

where for example

$$f_{11}(X_1, \ldots, X_{14}) = X_5 \cup X_6$$

- From the above equation it is clear that the solution $\underline{LV}$ we are looking for is the (least) fixed point of the function F.

Equation solving (recap)

- How can we obtain the least fixed point practically?
- For the least element $\perp \in D$ of a partially ordered set D we have

$$\perp \sqsubseteq F(\perp)$$

- By induction we have for all $n \in \mathbf{N}$

$$F^n(\perp) \sqsubseteq F^{n+1}(\perp)$$

- All elements of the sequence are in the domain D, and therefore, if D is finite, there exists an $n \in \mathbf{N}$ such that

$$F^n(\perp) = F(F^n(\perp))$$

(Requires special properties of D and F, shown later.)

- But this means that $F^n(\perp)$ is a fixed point! (And indeed a least fixed point.)

Existence of the fixed point

- To ensure that we can always obtain a result, we would like to know whether a fixed point of F always exists.

- To decide this, we need background on properties of:
  - the analysis domain used to represent the data flow information, i.e. in the case of the LV analysis the domain $\wp(Var_*)$, the power set of all variables occurring in the program
  - the function F, as defined before

Note:

- Var: set of all variables
- Var_*: set of all variables *occurring in the given program*

Partially ordered sets (recap)

For any analysis, we are interested in expressing that one analysis result is "better" (more precise) than another.

In other words, we want the analysis domain to be partially ordered.

A partial ordering is a relation $\sqsubseteq$ that is

- reflexive: $\forall d : d \sqsubseteq d$
- transitive: $\forall d_1, d_2, d_3 : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ imply $d_1 \sqsubseteq d_3$
- anti-symmetric: $\forall d_1, d_2 : d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$ imply $d_1 = d_2$

A partially ordered set $(D, \sqsubseteq)$ is a set D with a partial ordering $\sqsubseteq$.

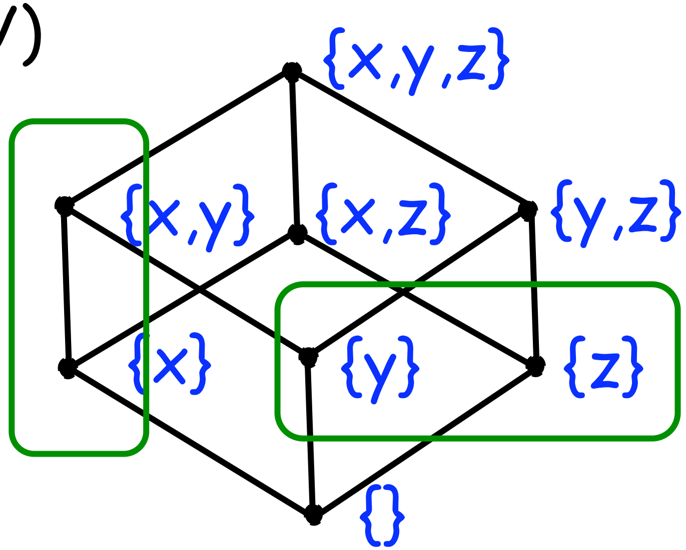
Examples: Real numbers $(\mathbf{R}, \leq)$, power sets $(\wp(S), \subseteq)$, ...

Complete lattices

We are aiming for a specific kind of partially ordered set with even nicer properties: complete lattices.

- $d \in D$ is an upper bound of Y if $\forall d' \in Y : d' \sqsubseteq d$
- A least upper bound d of Y is an upper bound of Y that satisfies $d \sqsubseteq d_0$ whenever d_0 is another upper bound of Y
- A complete lattice is a partially ordered set $(D, \sqsubseteq)$ such that each subset Y has a least upper bound $\sqcup Y$ (and a greatest lower bound $\sqcap Y$)

Example: Power sets



Least upper bound operator, top, bottom

- The least upper bound (lub) operator $\sqcup : \wp(D) \rightarrow D$ (also: join operator) is used to combine analysis information from different paths.
- For example, in the case of the LV analysis, the join is given by ordinary set union $\cup$, and we were using it to combine information from both if-branches:

$$LV_{exit}(4) = LV_{entry}(5) \cup LV_{entry}(6)$$

- Every complete lattice has a least and a greatest element, they are called bottom $\perp$ and top $\top$, respectively.

Tarski's Fixed Point Theorem

Monotone function

A function F : D -> D is called monotone over (D, $\sqsubseteq$) if

$d \sqsubseteq d'$ implies $F(d) \sqsubseteq F(d')$ for all $d, d' \in D$

Fixed point

Assume F : D -> D. A value $d \in D$ such that F(d) = d is called a fixed point of F.

Tarski's Fixed Point Theorem

Let (D, $\sqsubseteq$) be a complete lattice and let F : D -> D be a monotone function. Then the set of all fixed points of F is a complete lattice with respect to $\sqsubseteq$.

In particular, F has a least and a greatest fixed point.

Existence of the least solution

- Using Tarski's fixed point theorem, we know that a least solution exists if
  - the function F describing the equation system is monotone
  - the analysis domain is a complete lattice
- In the case of the LV analysis these properties are easily checked:
  - To prove the monotonicity of F, we prove the monotonicity of each function f_i
  - The domain $\wp(Var_*)$ is trivially a complete lattice (it is a power set)

Why are we interested in the least solution?

Remember the formulation of the goal of the Live Variables Analysis:

> "For each program point, which variables *may* be live at the exit from the point."

Clearly, larger solutions can always be accepted – even the set of all variables would do! – but the least ("smallest") solution is the most precise one.

Data Flow Analysis

Available Expressions Analysis

Available expressions analysis

Another example of a data flow analysis: available expressions (AE) analysis.

➢ The aim of the available expressions analysis is to determine

> "For each program point, which expressions *must* have already been computed, and not later modified, on all paths to the program point."

Example:

$[x:=a+b]^1$; $[y:=a*b]^2$;

while $[y>a+b]^3$ **do** $[a:=a+1]^4$; $[x:=a+b]^5$ **end**

Which expression is always available at the entry to 3?

Formalization: Data flow equations

The available expressions analysis can be specified following the scheme for LV analysis.

Analysis domain: $\wp(AExp_*)$, i.e. sets of arithmetic expressions.

$$AE_{entry}(l') = \bigcap_{(l, l') \in CFG} AE_{exit}(l)$$

(and $AE_{entry}(l) = \{\}$ if l is the initial label)

$$AE_{exit}(l) = (AE_{entry}(l) \setminus kill_{AE}(l)) \cup gen_{AE}(l)$$

$kill_{AE}([x:=a]^l)$ = {all expressions containing x}

$kill_{AE}([b]^l)$ = {}

$gen_{AE}([x:=a]^l)$ = {all subexpressions of a not containing x}

$gen_{AE}([b]^l)$ = {all subexpressions of b}

Application: Common subexpression elimination

Goal: Find computations that are always performed at least twice on a given execution path and then eliminate the second and later occurrences.

Example:

$[x:=a+b]^1$; $[y:=a*b]^2$;
while $[y>a+b]^3$ **do** $[a:=a+1]^4$; $[x:=a+b]^5$ **end**

is transformed into

$[u:=a+b]$; $[x:=u]^1$; $[y:=a*b]^2$;
while $[y>u]^3$ **do** $[a:=a+1]^4$; $[u:=a+b]$; $[x:=u]^5$ **end**

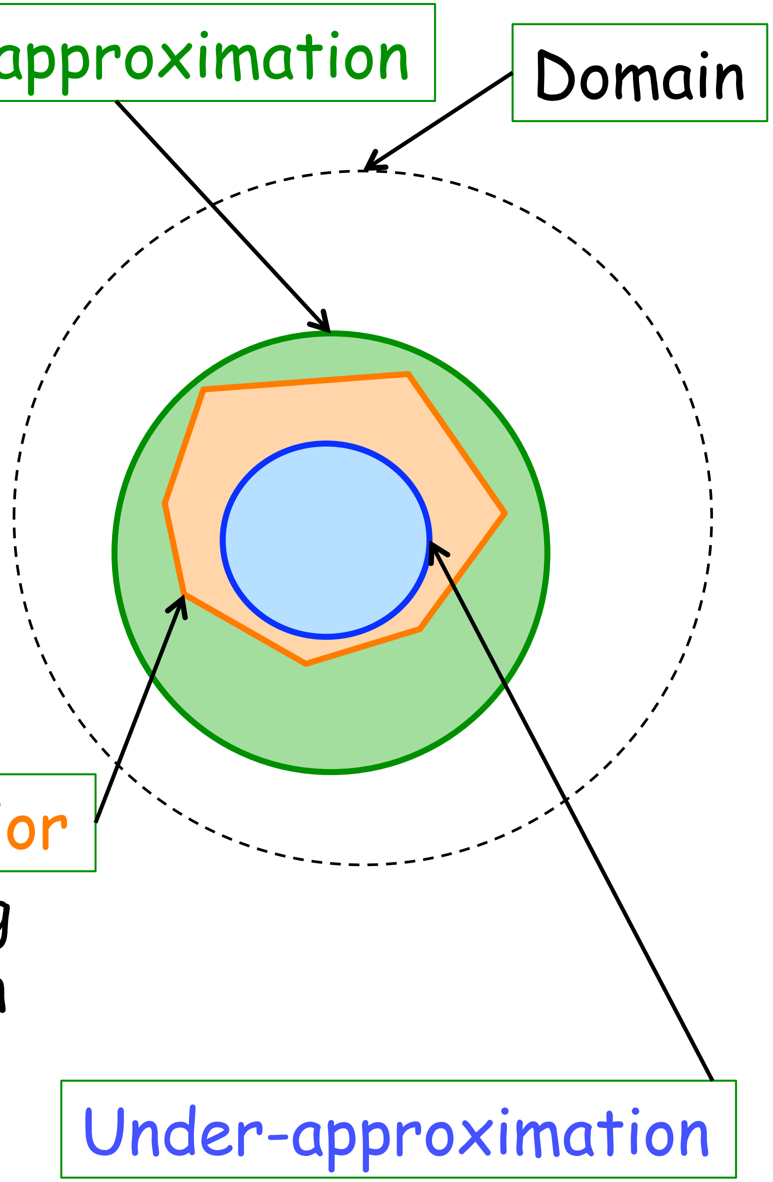
Differences of the LV, RD, and AE analyses

The flow has been reversed:

- LV: backward analysis
- AE, RD: forward analysis

Also, we are now interested in an under-approximation ("must"):

- LV, RD: may analysis
- AE: must analysis



For that reason, in AE we are taking an intersection ∩ instead of a union ∪ on the paths. We are then interested in the greatest solution.

Data Flow Analysis

Bit Vector Frameworks

Four classical analyses

Live Variables

Variables that may be live at a program point.

Reaching Definitions

Assignments that may have been made and not overwritten along some path to a program point.

Available Expressions

Expressions that must have already been computed and not later modified on all paths to a program point.

Very Busy Expressions

Expressions that must be very busy at a program point.

A general schema

The four classical analyses, and many more data flow analyses follow a general schema.

- The analysis domain is always a power set of some finite set, e.g. sets of variables in case of LV.
- The functions that specify how data is propagated through elementary blocks (so-called transfer functions) are all of the form

$$f(d) = (d \setminus kill) \cup gen$$

(It's easy to prove that functions of this form are monotone.)

Bit Vector Frameworks

These properties of classical analyses make for efficient implementation using bit vectors to represent sets.

Example:

LV analysis for a program with variables x, y, z

- Representation:

  {} = 000, {x} = 100, {y} = 010, ..., {x, z} = 101

- Join is very efficient (use boolean **or**):

  $\{x, y\} \cup \{x, z\} = \{x, y, z\}$

  110 **or** 101 = 111

Reading

Textbook:

Flemming Nielson, Hanne Riis Nielson, Chris Hankin:
Principles of Program Analysis, Springer, 2005.

Chapter 1: Sections 1.1-1.3, 1.7

Chapter 2: Sections 2.1, 2.3, 2.4