Automated Testing & Verification

Intraprocedural Dataflow Analysis

Galeotti/Gorla/Rau
Saarland University

Dataflow Static Analyzers

- All tools that evaluate code directly (no code execution)
- **All** possible executions are modeled
  - Over an abstraction of the program state
    - Less precise than dynamic analysis (no concrete program executions)
    - Safer than underapproximating
- Targets "mechanical" errors (difficult to find through Testing or code inspection)
    - Memory usage errors (null dereferences, uninitialized data, double free errors)
    - Resource Leaking (locks, files, memory)
    - Vulnerabilities (buffer overruns, SQL injection)
    - API violations, private data being exposed
    - Non handled exceptions, concurrency issues (race conditions), etc.
- Property Inference
  - Specifications, invariants, resource usage

Dataflow Analysis

Motivation:

- Find interesting properties/errors
  - x is null, x is copied to y, y is a des-referenced
- Optimization
  - Execution time, memory consumption, etc
- **Guarantee of Correctness**
  - Critical Systems
    - To make sure abscence of a certain kind of errors
  - Guarantees are needed for Optimizing
    - Example: eliminate dead code, move a statement outside the loop
- **Automaticity**
  - Cost Reducction
  - Unfeasible manually

Dataflow analysis: common uses

- **For code optimization**
  - Detect unused variables;
  - Remove dead code
  - Detect frequently used expressions.
  - Purity (method with no side-effects)
  - Valid object dereference (avoid null checking).
  - …
- **For program understanding**
  - Infer the type of a function
  - Obtain pre/post, invariants.
  - Resource Usage
  - Reverse engineering
    - Call-graph of a OO program
    - Behavioral models
    - …

Examples

- Classical:
  - Live variable analysis, Reaching definitions, Available expressions, etc..
  - Mainly for optimization
- Safety / Program Understanding
  - Zero analysis
  - Null pointer
  - Intervals: array ranges
  - Invariants
- For further analysis:
  - Points-to analysis
  - Call graph
  - Aliasing

Available expressions

- Detect which expressions are available at each program point
- Remove redundant computations
- An expression (*x op y*) is *available* in a given program point if for **all program executions** leading to that point
  - *x op y* is computed at least once
  - *x* e *y* is not redefined since the moment when *x op y was computed*

```
…
{   }
int b = a + 2;
{ a + 2}
int c = b*b;
{ a + 2, b*b }
int d = c + 1;
{ a + 2, b*b, c+1}
c = 4;
{ a + 2, b*b }
if(b < c) b = 2;
else c = a+1;
{ a + 2 }
return d;
```

Live variable analysis

- Identify which variables are "alive" (namely, will be used later in the program)
  - *x* is *alive* since the moment it was defined until the **last** use or until it is redefined

- Uses:
  - Assign variables to registers
  - Remove dead code
    - Remove code linked to assigning variables no longer alive

```
…
{ a }
int b = a + 2;
{ b }
int c = b*b;
{ c }
int d = c + 1;
{ d }
c = 4;
{ d, c }
return d+c;
{ }
```

Zero analysis

- Infer the value for each variable and answer
  - Is X equal to Zero?
- Uses:
  - Early bug detection
    - Division by zero
    - Null dereference
  - Constant Propagation

```
x := 8;
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-2;
    z := 5;
```

Zero analysis

- Infer the value for each variable and answer
  - Is X equal to Zero?

- Uses:
  - Early bug detection
    - Division by zero
    - Null dereference
  - Constant Propagation

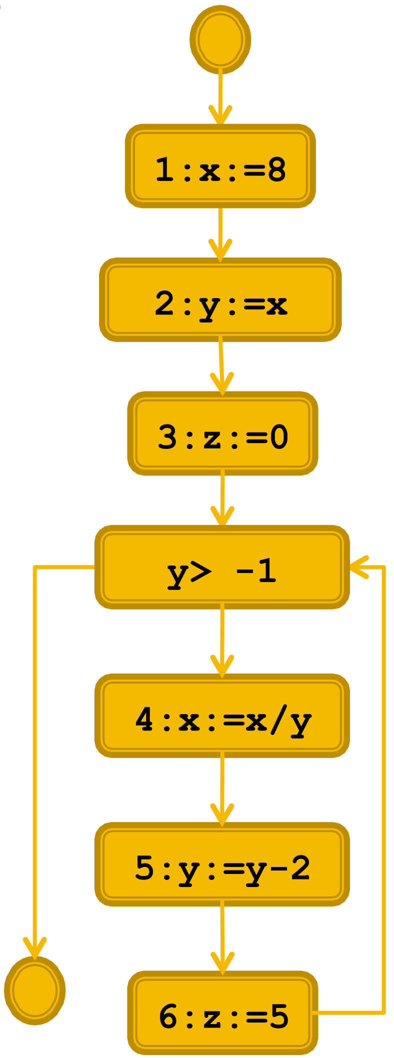
```
[]
x := 8;
[x→ NZ]
y := x;
[x→ NZ, y→ NZ]
z := 0;
[x→ NZ, y→ NZ, z→ Z]
while y > -1 do
    [x→ NZ, y→ MZ, z→ MZ]
    x := x / y;
    [x→ NZ, y→ MZ, z→ MZ]
    y := y-2;
    [x→ NZ, y→ MZ, z→ MZ]
    z := 5;
    [x→ NZ, y→ MZ, z→ NZ]
```

Dataflow Analysis

- One of the most popular static analysis techniques.

- **Purpose**: Infer **automatically** interesting properties of a program
  - Specifically, to a given program point

- **Principle**: Model execution of a program as the solution of a set of equations describing the flow of values throught the program instructions.

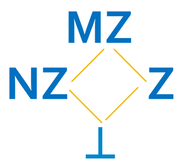


Dataflow Analysis: Intuition

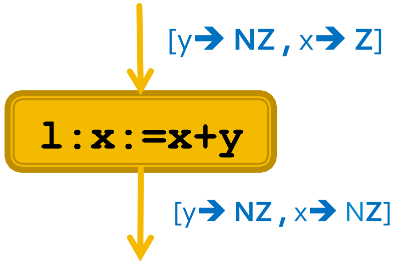
- "execute" the program using abstract values.
- Collect in each program point all the information flowing to that point
  - It can give us information for each program point.
    - Which are the possible values of variable Y after executing instruction #5?
    - Can the "null" value flow towards x in any instruction?
  - It can distinguish instruction order
    - Was a file read after it was closed?
- Flow sensitive
  - Needs a control-flow graph

Dataflow Analysis: elements

- **Control-flow graph**: A representation of the flow of control in the program
- **Abstract values**: represent information flowing through the program



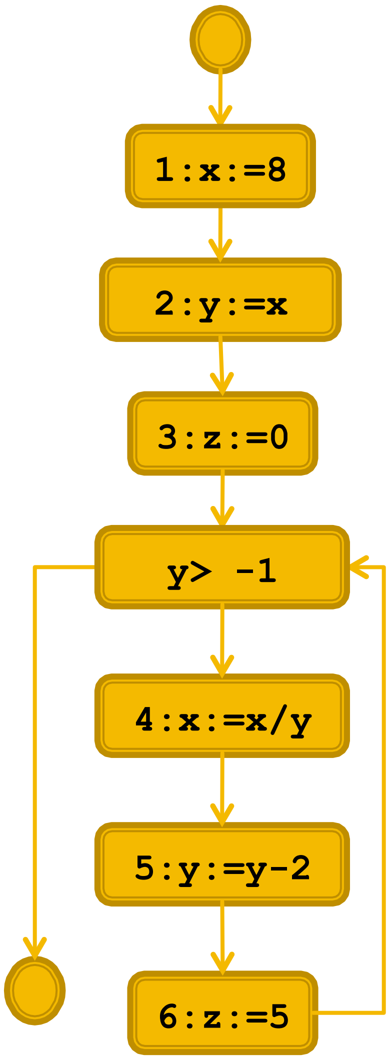
- **Transfer function**: what is the effect over the program state for every program instruction



- **Dataflow Equations**: how abstract values flow according to the control flow of the program

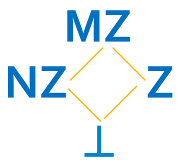
Control-Flow Graph

- Shows execution order
- Operations are usually dicomposed into simpler instructions
  - Example (3-address code)
    - a = b + c + d => t1 = b+c ; a = t1 + d
- Iterative constructs are removed (while, for, repeat)
  - They are modeled as backward-edges in the control-flow graph
- Objective: to analyze a simpler operation one at a time.



Abstract values

- Choose an abstraction according to the interesting property
  - X can be equal to Zero?
  - Was expression a+b computed previously?
  - Do we need variable x at this point of the program?
  - Where does the value being assigned to x came from?
  - Is this file open?
  - Is variable x equal to null when it is de-referenced?
  - Do x and y represent the same object?

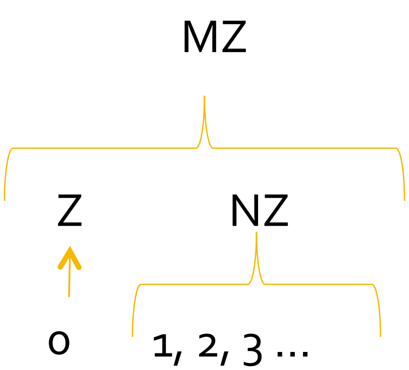


- **Key:** The abstract state must be <u>tractable</u>
  - Abstract values must belong to a **<u>lattice</u>.**
  - Typically <u>finite lattice</u> (at least lattice height)

Abstraction requires approximation

- Abstraction => do not handle the concrete state
  - We do not handle actual information

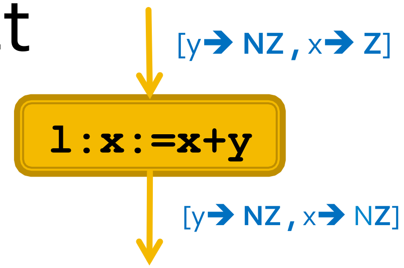
- Example: Natural numbers
  - 3 – 3 = 0
  - Abs(3) = NZ
  - How much is NZ – NZ?
    - Abs(3) = NZ
    - Abs(3-3)=NZ-NZ => MZ



Transfer function

- Indicates the effect of each instruction on the abstract state

- Given a node (instruction) and a abstract state it creates a new abstract state
  - $F_{stmt}(\sigma) = \sigma'$

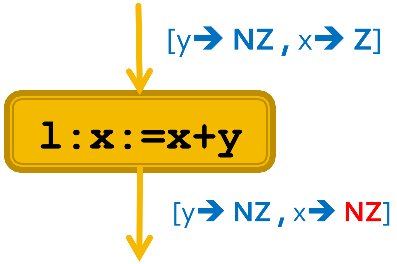


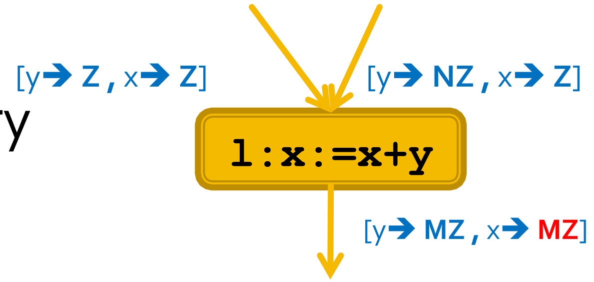
- Example:
  - $F_{x:=x+y}$([y→ NZ , x→ Z]) = [y→ NZ , x→ **NZ**]

- Some properties:
  - It has to be monotonous : x <= y → f(x) <= f(y)
  - Closed under composition (f(f(x)) is always defined)

Dataflow Analysis: elements

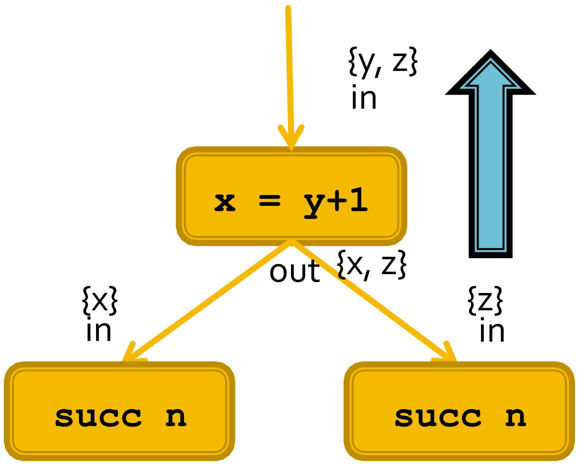
- Dataflow equations:
  - They provide how a node's output is computed given its inputs
  - In which order data flow and how it is combined
    - **Forward**: From the program entry towards its exit
      - Zero analysis, available expressions, etc
    - **Backward**: From the program exit to its entry
      - Live variables analysis
  - How to interpret the collected data
    - What to do if there are data flowing from to different nodes:
      - Apply the "upper bound"/ **MAY Analysis**
      - Apply the "lower bound"/ **MUST Analysis**



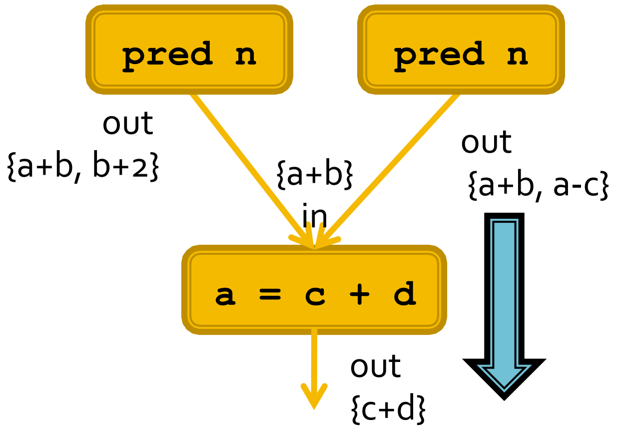


Equation examples

- Live variables analysis
  - in[n], out[n] = set of variables
  - *transfer[n](X) = gen(n) U (X – kill(n))*
    - gen(n) = read accesses to variables in node n
    - kill(n) = write accesses to variables in n
  - $\oplus = \cup$ (of sets)
  - ***out***[*n*] := ∪ { ***in***[*m*] | *m* ∈ ***succ***(*n*) }
  - ***in***[n] := ***transfer***[*n*](***out***[*n*])



- Available expressions
  - in[n], out[n] = set of expressions
  - *transfer[n](X) = gen(n) U (X ∩ kill(n))*
    - gen(n) = new expressions created
    - kill(n) = exprs containing variables written by n
  - $\oplus = \cap$ (of sets)
  - ***in***[*n*] := ∩ { ***out***[*m*] | *m* ∈ ***pred***(*n*) }
  - ***out***[n] := ***transfer***[*n*](***in***[*n*])



Framework Dataflow

For each node *n:*

- in[*n]: abstract values* before program point *n*
- out[*n]: abstract values* after program point *n*
- transfer[n]*: operation to apply on the values flowing through node n*

For each analysis:

- ⊕: join operator (for joining several input/output values)

Direction\⊕	**∪ (MAY)**	**∩ (MUST)**
Forward	Given in[n], compute out[n] Apply transfer[n] to predecessors[n] Property holds in some path (**reaching defs**, **zero analysis**)	Given in[n], compute out[n] Apply transfer[n] to predecessors[n] Property holds in all paths (**available expressions**)
Backward	Given out[n], compute in[n] Apply transfer[n] to successors[n] Property holds in some path (**live variable analysis**)	Given out[n], compute in[n] Apply transfer[n] to successors[n] Property holds in all paths (**very busy expressions**)

Iterative algorithm

Compute out[*n*] ***for each $n \in N$:***
out[*n*] := ⊥ (or TOP if MUST analysis)
Repeat
    For each *n*
        in[*n*] := $\bigoplus$ { ***out***[*m*] | *m* $\in$ ***pred***(*n*) }
        out[*n*] := ***transfer***[*n*](***in***[*n*])
Until no further changes to **in/out**