Dataflow Examples, and Correctness and Termination (2/N)

Claire Le Goues

15-819O: Program Analysis

EXAMPLE ANALYSES

Reaching Definitions Analysis

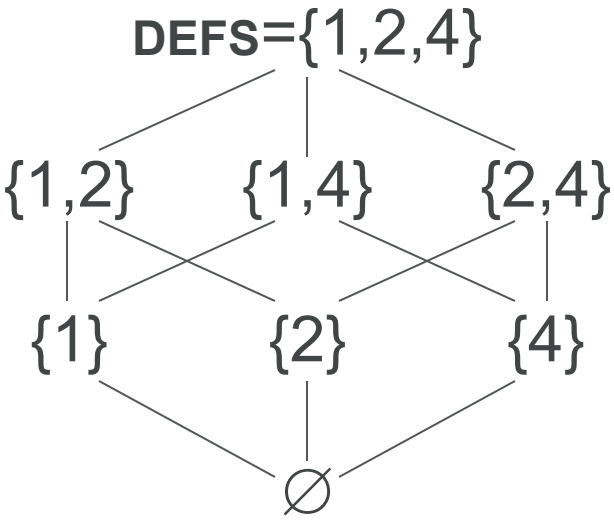
- Goal: determine which is the most recent assignment to a variable that precedes its use:

```
[y := x]1;
[z := 1]2;
while [y>1]3 do
  [z := z * y]4;
  [y := y – 1]5;
[y := 0]6;
```

- Example: definitions 1 and 5 reach the use of y at 4
- Simpler version of constant propagation, zero analysis, etc.
  - Just look at the reaching definitions for constants!
  - If definitions reaching use include "undefined" sentinal, then we may be using an undefined variable

Reaching Definitions

- Set Lattice ($\mathbb{P}^{\mathbf{DEFS}}$, $\sqsubseteq_{RD}$, $\sqcup_{RD}$, $\varnothing$, **DEFS**)
  - **DEFS**: the set of definitions in the program
  - Each element of the lattice is a subset of defs
  - $\mathbb{P}^{\mathbf{DEFS}}$ is the powerset of **DEFS**
- Approximation: A definition *d* may reach program point *P* if *d* is in the lattice at *P*
  - We call this a *may analysis*
  - $x \sqsubseteq_{RD} y$ iff $x \subseteq y$
  - $x \sqcup_{RD} y = x \cup y$
    - This is a direct consequence of the definition of $\sqsubseteq_{RD}$
  - $\bot = \varnothing$ (no reaching definitions)
  - $\top =$ **DEFS** (all definitions reach)



Reaching Definitions

- Initial assumptions?
  - Either dummy assignments, or empty set.
  - Represents passed values for parameters
  - Represents uninitialized for non-parameters
- Common notation in set-based analyses:
  - *Kill* set: elements removed from a set by an instruction.
  - *Gen* set: elements added to a set by an instruction.

Flow functions

$$f_{RD}[\![I]\!](\sigma) = \sigma - KILL_{RD}[\![I]\!] \cup GEN_{RD}[\![I]\!]$$

$$\text{KILL}_{RD}[\![n : x := ...]\!] = \{x_m | x_m \in \text{DEFS}(x)\}$$

$$\text{KILL}_{RD}[\![I]\!] = \varnothing \quad \text{if } I \text{ is not an assignment}$$

$$\text{GEN}_{RD}[\![n : x := ...]\!] = \{x_n\}$$

$$\text{GEN}_{RD}[\![I]\!] = \varnothing \quad \text{if } I \text{ is not an assignment}$$

Reaching Definitions Example

$[y := x]_1$;
$[z := 1]_2$;
while $[y>1]_3$ do
    $[z := z * y]_4$;
    $[y := y - 1]_5$;
$[y := 0]_6$;

Position	Worklist	Lattice Element

Reaching Definitions Example

$[y := x]_1$;
$[z := 1]_2$;
while $[y>1]_3$ do
    $[z := z * y]_4$;
    $[y := y - 1]_5$;
$[y := 0]_6$;

Position	Worklist	Lattice Element
0	1	$\{x_0, y_0, z_0\}$
1	2	$\{x_0, y_1, z_0\}$
2	3	$\{x_0, y_1, z_2\}$
3	4,6	$\{x_0, y_1, z_2\}$
4	5,6	$\{x_0, y_1, z_4\}$
5	3,6	$\{x_0, y_5, z_4\}$
3	4,6	$\{x_0, y_1, y_5, z_2, z_4\}$
4	5,6	$\{x_0, y_1, y_5, z_4\}$
5	6	$\{x_0, y_5, z_4\}$
6		$\{x_0, y_6, z_2, z_4\}$

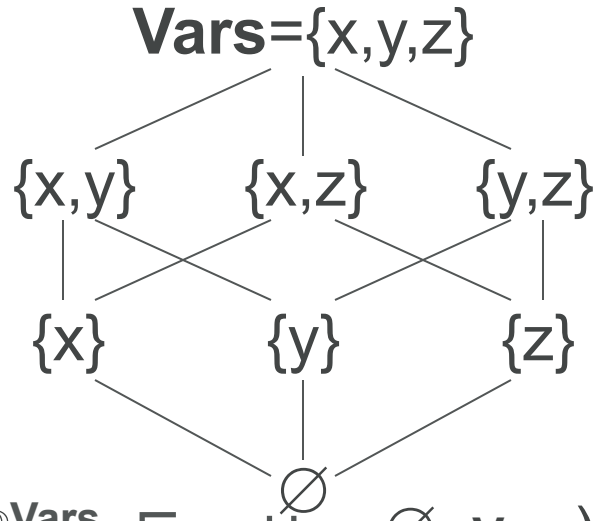
Live Variables Analysis

- Goal: determine which variables may be used again before they are redefined (i.e. are live) at the current program point.

```
[y := x]1;
[z := 1]2;
while [y>1]3 do
  [z := z * y]4;
  [y := y – 1]5;
[y := 0]6
```

- Example: after statement 1, y is live, but x and z are not
- Optimization applications: If a variable is not live after it is defined, can remove the definition statement (e.g. 6 in the example)

Live Variables Definition



- Set Lattice ($\mathbb{P}^{\mathbf{Vars}}$, $\sqsubseteq_{LV}$, $\sqcup_{LV}$, $\emptyset$, **Vars**)
  - **Vars** is the set of variables in the program
  - Each element of the lattice is a subset of **Vars**
    - $\mathbb{P}^{\mathbf{Vars}}$ is the powerset of **Vars**, i.e. the set of all subsets of **Vars**
  - $x \sqsubseteq_{LV} y$ iff $x \subseteq y$
  - $x \sqcup_{LV} y = x \cup y$
  - Most precise element $\bot = \emptyset$ (no live variables)
  - Least precise element $\top$ =**DEFS** (all variables live)

Live Variables Definition

- Live Variables is a *backwards* analysis
  - To figure out if a variable is live, you have to look at the future execution of the program
- Will x be used before it is redefined?
  - When x is defined, assume it is not live
  - When x is used, assume it is live
  - Propagate lattice elements as usual, except backwards
- Initially assume return value is live
  - $\boldsymbol{i}_{LV}$ = { *x* } where *x* is the variable returned from the function

Flow Function Practice

- Write flow functions for Live Variable analysis:
  - $\boldsymbol{f}_{LV}(\sigma, [x := e]_k)$ =
  - $\boldsymbol{f}_{LV}(\sigma, [e]_k)$ =
  - $\boldsymbol{f}_{LV}(\sigma,$ */* any other */* $)$ =

Flow Function Practice

- Write flow functions for Live Variable analysis:
  - $\boldsymbol{f}_{LV}(\sigma, [x := e]_k) = (\sigma - \{ x \}) \cup vars(e)$
    - Kills (removes from set) the variable x
    - Generates (adds to set) the variables in e
    - Note: must kill first then generate (what if e = x?)
  - $\boldsymbol{f}_{LV}(\sigma, [e]_k) = \sigma \cup vars(e)$
  - $\boldsymbol{f}_{LV}(\sigma, /*\ any\ other\ */) = \sigma$

Live variables practice

$[y := x]_1$;
$[z := 1]_2$;
while $[y>1]_3$ do
    $[z := z * y]_4$;
    $[y := y - 1]_5$;
$[y := 0]_6$;
return z;

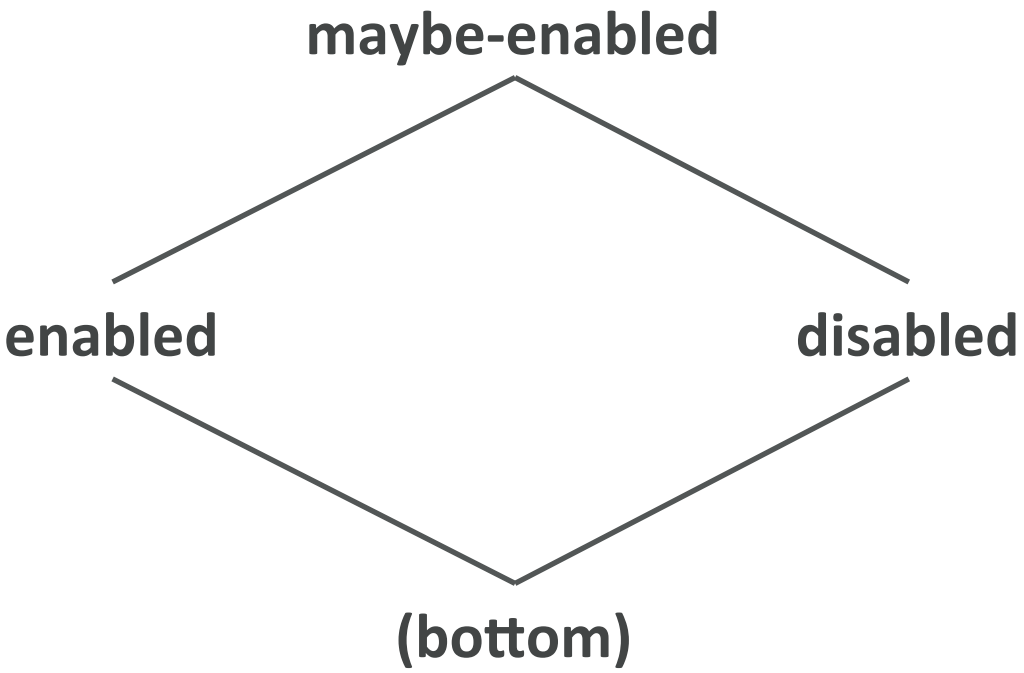
Position	Worklist	Lattice Value

Live Variables Example

$[y := x]_1$;
$[z := 1]_2$;
while $[y>1]_3$ do
    $[z := z * y]_4$;
    $[y := y - 1]_5$;
$[y := 0]_6$;
return z;

Position	Worklist	Lattice Element
exit	6	{z}
6	3	{z}
3	5,2	{y,z}
5	4,2	{y,z}
4	3,2	{y,z}
3	2	{y,z}
2	1	{y}
1		{x}

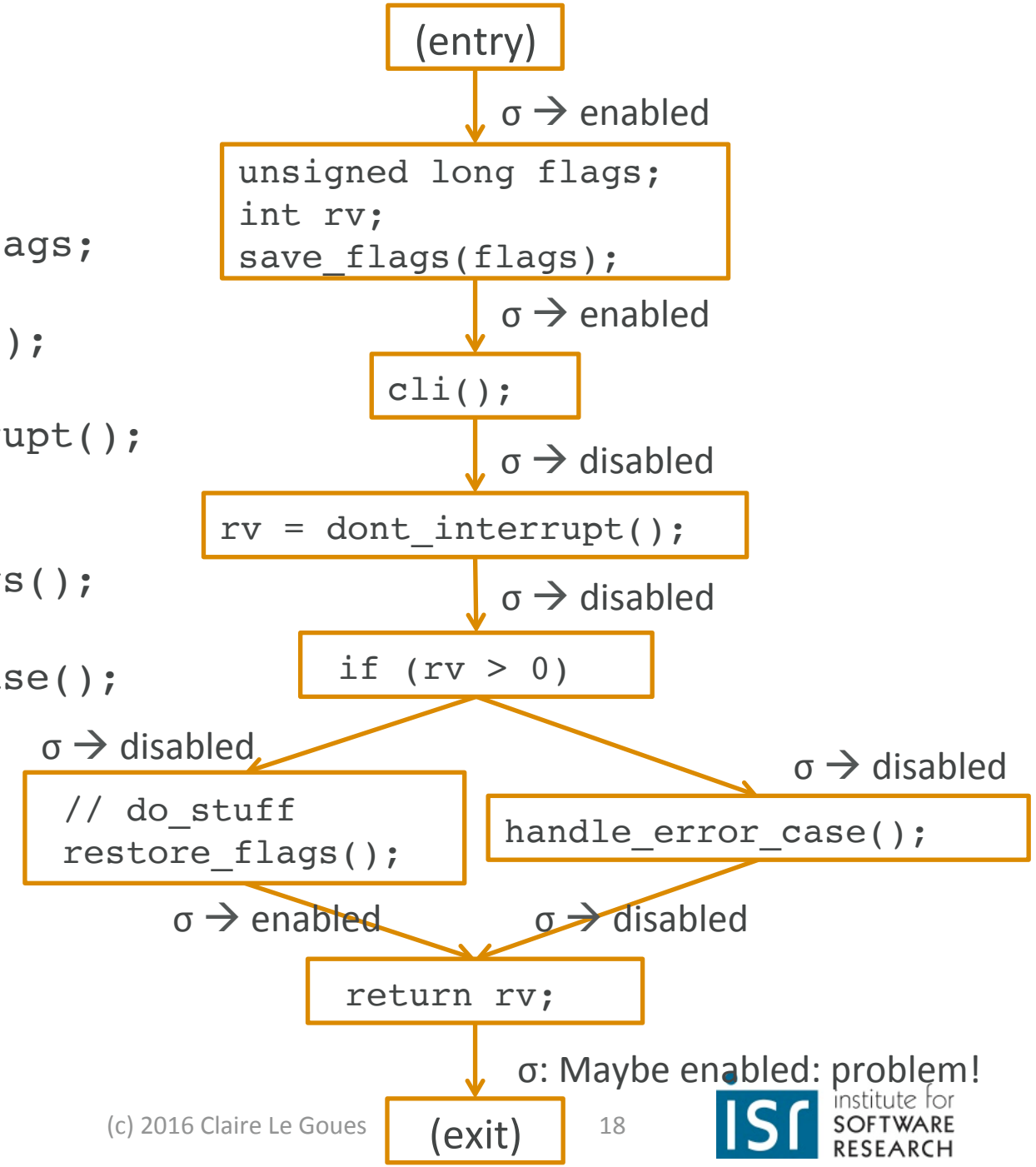
Example: interrupt checker



An interrupt checker

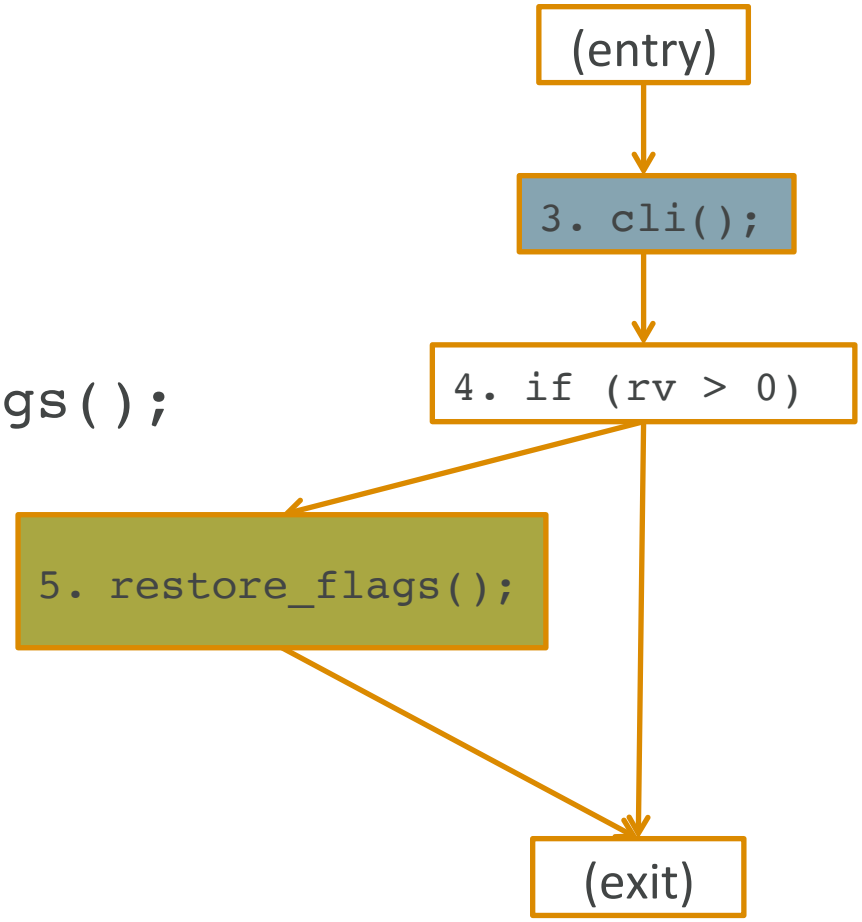
- **Abstraction**
  - Three abstract states: enabled, disabled, maybe-enabled
  - Warning if we can reach the end of the function with interrupts disabled.
- Transfer function:
  - If a basic block includes a call to `cli()`, then it moves the state of the analysis from **disabled** to **enabled**.
  - If a basic block includes a call to `restore_flags()`, then it moves the state of the analysis from **enabled** to **disabled**.

```
1.  int foo() {
2.     unsigned long flags;
3.     int rv;
4.     save_flags(flags);
5.     cli();
6.     rv = dont_interrupt();
7.     if (rv > 0) {
8.        // do_stuff
9.         restore_flags();
10.    } else {
11.     handle_error_case();
12.    }
13.    return rv;
14. }
```



Abstraction

```
1.  void foo() {
2.      …
3.      cli();
4.    if (a) {
5.        restore_flags();
6.      }
7.  }
```



TERMINATION

Termination definitions

- **Ascending chain**: A sequence σ_k is an *ascending chain* iff $n \leq m$ implies $\sigma_n \sqsubseteq \sigma_m$.
- **Height of an ascending chain**: An ascending chain σ_k has finite height *h* if it contains *h*+1 distinct elements.
- **Height of a lattice**: A lattice (L, $\sqsubseteq$) has finite height *h* if there is an ascending chain in the lattice of height *h*, and no ascending chain in the lattice has height greater than *h.*
- **Monotonicity:** Function *f* is *monotonic* iff $\sigma_1 \sqsubseteq \sigma_2$ implies $f(\sigma_1) \sqsubseteq f(\sigma_2)$

Theorem: Dataflow Analysis Termination

IF THE DATAFLOW LATTICE $(L, \sqsubseteq)$ HAS FINITE HEIGHT, AND THE FLOW FUNCTIONS ARE MONOTONIC, THE WORKLIST ALGORITHM WILL TERMINATE.

Why? Proof by induction

- Assume: The input state at every program point (other than entry) starts at $\perp$
- *Base case:* The first time the flow function is run on each instruction, the result will be at least as high in the lattice as before (because nothing is lower than $\perp$).
- Assume that the previous time we ran the flow function, we had input information σ_i and output information σ_o.
- If we are running it again, it's because the input information has changed to some new σ_i'. By the induction hypothesis, we can assume $\sigma_i \sqsubseteq \sigma_i$'.
- We thus just need to prove is that $\sigma_o \sqsubseteq \sigma_o$', which will be true if our flow functions are monotonic (by definition).

Why? Proof by induction

- (Start of) Induction step:
  - Assume that the previous time we ran the flow function, we had input information σ_i and output information σ_o.
  - If we are running it again, it's because the input information has changed to some new σ_i'. By the induction hypothesis, we can assume $\sigma_i \sqsubseteq \sigma_i'$.
- So, for termination, we just need to prove $\sigma_o \sqsubseteq \sigma_o'$, which will be true if our flow functions are monotonic (by definition).

…Wait, why?

- Monotonicity means that the dataflow value at each program point *i* can only *increase* each time σ[i] is assigned.
  - So, the assignment can happen a maximum of *h (lattice height)* times for each program point.
- This bounds the number of elements added to the worklist to *h * e* (e=control flow graph edges).
- Since we remove one element of the worklist each time the loop executes, the loop will execute no more than *h * e* times.
- Thus, the algorithm will always terminate.

Termination definitions

- **Ascending chain**: A sequence σ_k is an *ascending chain* iff $n \leq m$ implies $\sigma_n \sqsubseteq \sigma_m$.
- **Height of an ascending chain**: An ascending chain σ_k has finite height *h* if it contains *h*+1 distinct elements.
- **Height of a lattice**: A lattice (L, $\sqsubseteq$) has finite height *h* if there is an ascending chain in the lattice of height *h*, and no ascending chain in the lattice has height greater than *h*.
- **Monotonicity:** Function *f* is *monotonic* iff $\sigma_1 \sqsubseteq \sigma_2$ implies $f(\sigma_1) \sqsubseteq f(\sigma_2)$

Zero analysis monotoncitiy

- *Case* $f_Z[\![x := 0]\!](\sigma) = [x \mapsto Z]\sigma$
  - Assume $\sigma_1 \sqsubseteq \sigma_2$
  - According to $\sqsubseteq$'s pointwise definition [x $\mapsto$ Z]$\sigma_1 \sqsubseteq$ [x $\mapsto$ Z]σ_2
- *Case* $f_Z[\![x := y]\!](\sigma) = [x \mapsto \sigma(y)]\sigma$
  - Assume $\sigma_1 \sqsubseteq \sigma_2$
  - $\sqsubseteq$ pointwise definition means that σ_1(y) $\sqsubseteq$ *simple* σ_2(y)
  - Therefore, using the pointwise definition of $\sqsubseteq$ again, [x $\mapsto \sigma_1$(y)]$\sigma_1 \sqsubseteq$[x $\mapsto \sigma_2$(y)]σ_2

Moar monotonicity!

LET'S DO ANOTHER RULE TOGETHER

Tricksiness

- This only works if the lattice is of finite height…
- …hmmmm….
  - (spoiler alert!)

Correctness: Intuition

PROGRAM ANALYSIS RESULTS SHOULD CORRECTLY DESCRIBE EVERY ACTUAL CORRESPONDING PROGRAM EXECUTION.

Correctness definitions

- **Program Trace *T*** of a program *P* is a potentially infinite sequence { c_0, c_1, ... } of configurations, where $c_0 = E_0$, 1 is the initial configuration, and for every i ≥ 0, $P \vdash c_i \rightsquigarrow c_{i+1}$
- The result { σ_i | i ∈ P } of a **dataflow analysis** on program ***P* is sound** iff, for all traces *T* of *P*, ∀ *i* s.t. 0 ≤ *i* < *length(T)* , $\alpha(c_i) \sqsubseteq \sigma_{n_i}$
- A dataflow analysis result { σ_i | i ∈ P } is **a fixed point** iff $\sigma_o \sqsubseteq \sigma_1$ where σ_o is the initial analysis information and σ_1 is the dataflow result before the first instruction, and for each instruction *i* we have $\sigma_i = \bigsqcup_{j \in \texttt{preds(i)}} f[\![P[j]]\!](\sigma_j)$

Exercise

- Consider the following (incorrect) flow function for zero analysis:

$$fz[\![x := y + z]\!](\sigma) = [x \mapsto Z]\sigma$$

- Why? *Prove it.*
- Let's do another example together, for practice.

Local soundness

- A flow function *f* is *locally sound* iff:
  - $P \vdash c_i \rightsquigarrow c_{i+1}$
  - and $\alpha(c_i) \sqsubseteq \sigma_i$ and $f[\![P[n_i]]\!](\sigma_i) = \sigma_{i+1}$ imply $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$

$$fz[\![x := y + z]\!](\sigma) = [x \mapsto Z]\sigma$$

Why? Prove it!

SO THIS DOESN'T WORK FOR OUR FALSE FLOW FUNCTION…

Zero analysis: assign to zero

Case $f_Z[\![x := 0]\!](\sigma) = [x \mapsto Z]\sigma$

- Assume c_i = E,n and α(E) = σ_i
- Thus $\sigma_{i+1} = f_Z[\![x := 0]\!](\sigma_i) = [x \mapsto Z]\alpha(E)$
- *step-const* says $c_{i+1} = [x \mapsto 0]E, n+1$
- the definition of α says $\alpha([x \mapsto 0]E) = [x \mapsto Z]\alpha(E)$
- Therefore $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$

Zero analysis: assign to not zero

Case $f_Z[\![x := m]\!](\sigma_i) = [x \mapsto N]\sigma_i \quad \text{where } m \neq 0$

- Assume c_i = E,n and α(E) = σ_i
- Thus $\sigma_{i+1} = f_Z[\![x := m]\!](\sigma_i) = [x \mapsto N]\alpha(E)$
- *step-const* says $c_{i+1} = [x \mapsto m]E, n+1$
- Now $\alpha([x \mapsto m]E) = [x \mapsto N]\alpha(E)$ by the definition of α and the assumption that m ≠ 0.
- Therefore, $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$

Zero analysis: operators

Case $f_Z[\![x := y \ op \ z]\!](\sigma_i) = [x \mapsto ?]\sigma_i$

- Assume c_i = E,n and α(E) = σ_i
- Thus $\sigma_{i+1} = f_Z[\![x := y \ op \ z]\!](\sigma_i) = [x \mapsto ?]\alpha(E)$
- *step-const* says that, for some *k*,

$$c_{i+1} = [x \mapsto k]E, n+1$$

- Now $\alpha([x \mapsto k]E) \sqsubseteq [x \mapsto ?]\alpha(E)$ because the map is equal for all keys except *x*, and for *x* we have $\alpha_{simple}(k) \sqsubseteq_{simple} ?$ for all *k*
- Therefore, $\alpha(c_{i+1}) \sqsubseteq \sigma_{i+1}$

Zero analysis: assign to variable

Case $f_Z[\![x := y]\!](\sigma) = [x \mapsto \sigma(y)]\sigma$:

ALMOST THERE!

Correctness definitions

- **Program Trace *T*** of a program *P* is a potentially infinite sequence { c_0, c_1, ... } of configurations, where $c_0 = E_0$, 1 is the initial configuration, and for every i ≥ 0, $P \vdash c_i \rightsquigarrow c_{i+1}$
- The result { σ_i | i ∈ P } of a **dataflow analysis** on program ***P* is sound** iff, for all traces *T* of *P*, ∀ *i* s.t. 0 ≤ *i* < *length(T)* , $\alpha(c_i) \sqsubseteq \sigma_{n_i}$
- A dataflow analysis result { σ_i | i ∈ P } is **a fixed point** iff $\sigma_o \sqsubseteq \sigma_1$ where σ_o is the initial analysis information and σ_1 is the dataflow result before the first instruction, and for each instruction *i* we have $\sigma_i = \bigsqcup_{j \in \texttt{preds(i)}} f[\![P[j]]\!](\sigma_j)$

Local Soundness implies Global Soundness: If a dataflow analysis's flow function *f* is monotonic and locally sound, and for all traces *T* we have $\alpha(c_0) \sqsubseteq \sigma_o$ where σ_o is the initial analysis information, then any fixed point $\{ \sigma_i \mid i \in P \}$ of the analysis is also sound.

Proof: induction on trace *T*

Case c_0:

- $\alpha(c_0) \sqsubseteq \sigma_o$ by assumption.
- $\sigma_o \sqsubseteq \sigma_{n_0}$ by the definition of a fixed point.
- $\alpha(c_0) \sqsubseteq \sigma_{n_0}$ by the transitivity of $\sqsubseteq$

Proof: induction on trace *T*

Case c_{i+1}:

- The induction hypothesis says: $\alpha(c_i) \sqsubseteq \sigma_{n_i}$
- The definition of a trace says: $P \vdash c_i \rightsquigarrow c_{i+1}$
- Local soundness says: $\alpha(c_{i+1}) \sqsubseteq f[\![P[n_i]]\!](\alpha(c_i))$
- Because *f* is monotone: $f[\![P[n_i]]\!](\alpha(c_i)) \sqsubseteq f[\![P[n_i]]\!](\sigma_{n_i})$
- The definition of a fixed point says:

$$\sigma_{n_{i+1}} = f[\![P[n_i]]\!](\sigma_{n_i}) \sqcup \ldots$$

- The properties of join mean that: $f[\![P[n_i]]\!](\sigma_{n_i}) \sqsubseteq \sigma_{n_{i+1}}$
- And because $\sqsubseteq$ is transitive, $\alpha(c_{i+1}) \sqsubseteq \sigma_{n_{i+1}}$

Conclusion

- So since :
  - The abstraction lattice maps to reality.
  - The lattice has finite height.
  - The flow functions are monotonic and locally sound.
- …Zero analysis is also sound/correct, meaning its results on any program P overapproximate (but never misrepresent) reality.

Q. E. D.