Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Monotone Frameworks Sections 2.1-2.4 of NNH

Ivo Gabe de Wolff
Most slides by Jurriaan Hage

Department of Information and Computing Sciences, Universiteit Utrecht

May 11, 2022

Introduction and Motivation

What is program analysis?

Program analysis

=

deriving information about the behaviour of computer programs.

Why do program analysis?

- optimization
  - dead code removal, strict evaluation, avoiding run-time type checks
- validation
  - type checking, security analysis, soft typing
- comprehension
  - maintainability monitoring, reverse engineering, architecture reconstruction

Static or dynamic

- Dynamic: testing, run-time instrumentation, profiling
- Very precise for observed executions
  - Not the subject of this course

Static or dynamic

- Dynamic: testing, run-time instrumentation, profiling
- Very precise for observed executions
  - Not the subject of this course
- Static: analysis of the inputs of the compilation
- Often as part of a compiler
- Even for programs with infinite executions, compilation should terminate.
- Analysis must be valid for all executions.

Static or dynamic

- Dynamic: testing, run-time instrumentation, profiling
- Very precise for observed executions
  - Not the subject of this course
- Static: analysis of the inputs of the compilation
- Often as part of a compiler
- Even for programs with infinite executions, compilation should terminate.
- Analysis must be valid for all executions.
- The two forms can complement each other.

Optimization

- Optimizations are silenty applied by a compiler,
- based on information discovered during program analysis.
- Optimizing analysis should never lead to failure to compile.
- Information should be valid for all executions.
- We must be able to trust the results of analysis.

Optimization

- Optimizations are silenty applied by a compiler,
- based on information discovered during program analysis.
- Optimizing analysis should never lead to failure to compile.
- Information should be valid for all executions.
- We must be able to trust the results of analysis.
- Program analysis must be sound (safe) with respect to the language semantics.
  - The analyzer may only err on the safe side
- So prove it.
- Case study: uniqueness typing.
  - Something marked as unique, but used twice, may have been GC'ed away.

Validation

- Verify that a program is type correct
- Verify that a highly secure value does not end up in a lowly secure variable
- Some programs will fail to compile
- This raises the issue of feedback
- Case study: type inferencing/checking, pattern match analysis, security analysis

Comprehension

- Software analysis is often coined as the term here.
- Analysis need not be sound, need not be complete.
- Validation not by proof, but empirical validation.
- Metrics are a typical example:
  - McCabe's Cyclomatic Complexity.
  - The higher the value, the more complex the code
  - Above 50 implies unmaintainable.
- Typically, you can always find examples where metrics do not predict well, but they work very well in practice.
- Cheap to compute.

The setting

Typically,

- a compiler validates a program and generates code.
- For any program, it has to do this in finite time.
- Running the program for all possible inputs is out of the question.
- Halting Problem is undecidable.
  - Decide for any given program and given input whether the program will terminate for that input.
- Every behavioural property of programs is undecidable.
  - Rice's Theorem
- What can we do?

Possible solutions

Verify properties by

- Program verification: verify properties by using (interactive) proof tools.
- Model checking: exhaustively test the property for all reachable states.
- Program analysis: allow (safe) approximate answers, but keep it automatic and efficient.
- We consider the latter possibility and hope our solutions are not too approximate to be of use.

These three areas do overlap in many ways.

Two dimensions of complexity

- Properties of the language:
  - parametric polymorphism
  - higher-order, higher-ranked, polymorphic recursion
  - subtyping
  - by-value (strict) or by-need (lazy) evaluation
  - strictness and other annotations,
- More complex implies more flexibility for programmer.
- Properties of the analysis:
  - subtyping, subeffecting, or poisoning
  - monovariant, polyvariant, higher-ranked
  - flow-sensitive versus flow-insensitive
  - minimal or most general (Holdermans and Hage)
  - whole program or modular
- More complex implies more precision and more expensive.

Make note

- Program analysis is not always restricted to programming languages.
- Can be applied in other places as well:
  - FIRST and FOLLOW for parsing LL(k) languages.
- Admittedly, general recursion/while loops provide most of the essential complications
- Still, even SQL can profit from optimizations.

Program properties

- In dependently typed programming and contract checking, static properties are encoded in the language itself.
  - Programmer-driven static analysis
- In static analysis we tend to not leave this to the programmer.
- The truth is probably somewhere in the middle.
- Contracts and dependently typed programming establish only properties of values, not of the computations.
- Static analysis often addresses issues relating to how something is computed.

Safe and sound

- Strong typing (Haskell, Java,...)
  - Programs are guaranteed not to go wrong.
  - Intended optimization: avoiding run-time checks and validation
  - Conservative: sometimes disallows programs that would go right.
- Soft typing (on languages like Scheme, Perl, Ruby, PHP, Python and Javascript)
  - Allow all programs that might go right.
  - Intended optimization: avoiding run-time checks, some validation
  - Liberal: some programs may go wrong.
  - Add run-time checks/generate warnings
- It all depends on how you will use the analysis results.

Some example analyses

- Dead-code elimination
- Strictness analysis in lazy functional languages
  - Which arguments to a function will always be evaluated at some point?
- Liveness of variables
  - which variables may still be used?
- Available expressions
  - eliminating double computations
- Dynamic dispatch problem
  - dead code with functions being first class citizens

More examples

- Shape analysis
  - for avoiding garbage collection
- Uncaught exceptions in `Java`
- Weak circularity test in attribute grammars
- Escape analysis
  - What does not escape may be allocated on the stack instead of the heap.
- Binding-time analysis
  - What can be partially evaluated at compile-time.
- And many, many more...

In the context of this course

- Dataflow analysis of While language
- Monotone frameworks
- Literature: Chapter 2 of Nielson, Nielson and Hankin
- Project: analyzing the While language with monotone frameworks
- Analysis of higher-order languages later in the course

More detailed roadmap

- First-order, imperative language
- First without, later with procedures
- In both cases, control-flow is fixed.
- Monotone frameworks
  - Conceptual and implementational framework for building dataflow analyses
- Illustrated by Available Expression Analysis, Live Variable Analysis, and Constant Propagation.
- Distributivity

Intraprocedural Analysis

The `While`-language

- Simple and imperative, no procedures (yet)
- Variables: $x, y, \ldots$, integers only
- Statements: assignments, `if`, `while`, `skip` and `;`
- Boolean expressions: constants `true`, `false`, boolean operators `and`, `or`, `not`, and relational operators $<, =, \ldots$
- Integer expressions: $0, -1, 1, -2, 2, \ldots$ and various operators $+, -, \ldots$
- Labels for identification: $[\texttt{skip}]^2$, $[\texttt{(x <= 2)}]^3$, $[\texttt{x := x + 1}]^{31}$

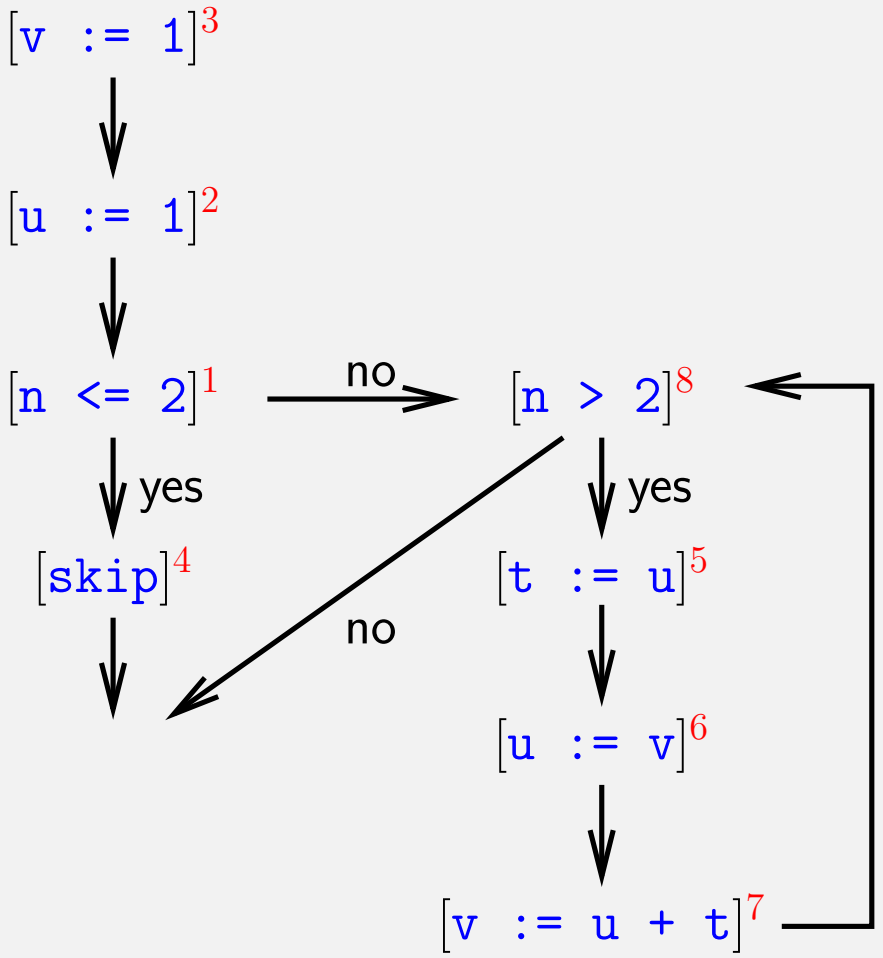
Exercise

Available expressions For each program point, which (non-trivial) expressions *must* already have been computed, and not later modified, on all paths to the program point.

Live variables For each program point, values of which variables may later be needed in the execution of the program.

Example program with its flow graph

```
[v := 1]^3; [u := 1]^2;
if [n <= 2]^1 then
   [skip]^4
else
   while [n > 2]^8 do
      ([t := u]^5;
       [u := v]^6;
       [v := u + t]^7);
```



Information from programs

- $[\texttt{v := 1}]^3$; $[\texttt{u := 1}]^2$;
  if $[\texttt{n <= 2}]^1$ then
  $[\texttt{skip}]^4$
  else
  while $[\texttt{n > 2}]^8$ do
  ($[\texttt{t := u}]^5$; $[\texttt{u := v}]^6$; $[\texttt{v := u + t}]^7$);
- $\texttt{labels}(S) = \{1, \ldots, 8\}$, $\texttt{init}(S) = 3$ and $\texttt{final}(S) = \{8, 4\}$
- $[\texttt{v := 1}]^3$, $[\texttt{skip}]^4$, $\ldots \in \texttt{blocks}(S)$
- $\texttt{flow}(S) = \{(3,2),(2,1),(1,4),(1,8),(8,5),(5,6),(6,7),(7,8)\}$ vs. $\texttt{flow}^R(S) = \{(2,3),(1,2),(4,1),(8,1),(5,8),(6,5),(7,6),(8,7)\}$

Information from programs

- $[\texttt{v := 1}]^3$; $[\texttt{u := 1}]^2$;
  if $[\texttt{n <= 2}]^1$ then
  $[\texttt{skip}]^4$
  else
  while $[\texttt{n > 2}]^8$ do
  ($[\texttt{t := u}]^5$; $[\texttt{u := v}]^6$; $[\texttt{v := u + t}]^7$);
- **AExp**$(u + v * 10) = \{v * 10, u + v * 10\}$ and **AExp**$(S) = \{u + t\}$.
- **AExp**(e) does not include single variables and constants
- Program under analysis is usually denoted S_*.
- We write **AExp**$_*$ instead of **AExp**(S_*) and so on.

Available Expression Analysis

Available Expression Analysis

For each program point, which (non-trivial) expressions must *already have been computed, and not later modified, on all paths to the program point.*

- ```
  [x := (a + b) * x]^1;
  [y := a * b]^2;
  while [a * b > a + b]^3 do
        ([a := a + 1]^4;
         [x := a + b]^5)
  ```
- `a + b` is always available at 3, but `a * b` is not.
- Each a subset of $\mathbf{AExp}_* = \{a+b, (a+b)*x, a*b, a+1\}$
- Associated optimization: values of available expression may be cached for use at $[\texttt{B}]^\ell$.
- To exploit this, *all* paths to $[\texttt{B}]^\ell$ must make it available


  ```

Some equations for Available Expressions

- $[\texttt{x := (a + b) * x}]^1$;
  $[\texttt{y := a * b}]^2$;
  $\texttt{while } [\texttt{a * b > a + b}]^3 \texttt{ do}$
  $(([\texttt{a := a + 1 }]^4;$
  $[\texttt{x := a + b}]^5)$
- $AE_{\mathrm{N}}(1) =$

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Some equations for Available Expressions

- `[x := (a + b) * x]`^{1};
  `[y := a * b]`^{2};
  `while [a * b > a + b]`^{3} `do`
  `([a := a + 1 ]`^{4};
  `[x := a + b]`^{5}`)`
- $AE_{\mathrm{N}}(1) = \emptyset$
  - nothing available at start of program
- $AE_{\mathrm{X}}(2) =$

Some equations for Available Expressions

- $[\texttt{x := (a + b) * x}]^1$;
  $[\texttt{y := a * b}]^2$;
  $\texttt{while } [\texttt{a * b > a + b}]^3 \texttt{ do}$
  $([\texttt{a := a + 1 }]^4;$
  $[\texttt{x := a + b}]^5)$
- $AE_{\mathrm{N}}(1) = \emptyset$
  - nothing available at start of program
- $AE_{\mathrm{X}}(2) = AE_{\mathrm{N}}(2) \cup \{a * b\}$
  - only the non-trivial expressions
- $AE_{\mathrm{N}}(3) =$

Some equations for Available Expressions

- `[x := (a + b) * x]`^{1};
  `[y := a * b]`^{2};
  `while [a * b > a + b]`^{3} `do`
  `([a := a + 1 ]`^{4};
  `[x := a + b]`^{5}`)`
- $AE_{\mathrm{N}}(1) = \emptyset$
  - nothing available at start of program
- $AE_{\mathrm{X}}(2) = AE_{\mathrm{N}}(2) \cup \{a * b\}$
  - only the non-trivial expressions
- $AE_{\mathrm{N}}(3) = AE_{\mathrm{X}}(2) \cap AE_{\mathrm{X}}(5)$
  - only if both paths make it available

Some equations for Available Expressions

- `[x := (a + b) * x]`^{1};
  `[y := a * b]`^{2};
  `while [a * b > a + b]`^{3} `do`
  `(`[`a := a + 1 `]^4;
  [`x := a + b`]^5`)`
- $AE_{\mathrm{X}}(3) =$

Some equations for Available Expressions

- `[x := (a + b) * x]`^{1};
  `[y := a * b]`^{2};
  `while [a * b > a + b]`^{3} `do`
  `(`[`a := a + 1 `]^4;
  [`x := a + b`]^5`)`
- $AE_{\mathrm{X}}(3) = AE_{\mathrm{N}}(3) \cup \{a + b, a * b\}$
  - condition also has effect
- $AE_{\mathrm{X}}(4) =$

Some equations for Available Expressions

- `[x := (a + b) * x]`^{1};
  `[y := a * b]`^{2};
  `while [a * b > a + b]`^{3} `do`
  `([a := a + 1 ]`^{4};
  `[x := a + b]`^{5}`)`
- $AE_{\mathrm{X}}(3) = AE_{\mathrm{N}}(3) \cup \{a + b, a * b\}$
  - condition also has effect
- $AE_{\mathrm{X}}(4) = AE_{\mathrm{N}}(4) - \{a + b, (a + b) * x, a + 1, a * b\}$
  - remove all arithmetic expressions which contain a

Auxiliary functions for Available Expressions

- We construct the analysis by specifying for each block:
  - which expressions become available: $gen_{AE}(B^\ell)$
  - which expressions become unavailable: $kill_{AE}(B^\ell)$
- These we then plug into a generic transfer function, that computes the effect of executing the block on the analysis result.
- Together with "flow" functions that push analysis results through the flow graph, we have a complete analysis.

For assignments

- Remove any expression that contains the assigned variable:
  $kill_{AE}([\texttt{x := a}]^{\ell}) = \{a' \in \mathbf{AExp}_* \mid x \in FV(a')\}$

For assignments

- Remove any expression that contains the assigned variable:
  $kill_{AE}([\texttt{x := a}]^\ell) = \{a' \in \mathbf{AExp}_* \mid x \in FV(a')\}$
- Add some or all subexpressions of the assigned expression:
  $gen_{AE}([\texttt{x := a}]^\ell) = \{a' \in \mathbf{AExp}(a) \mid x \notin FV(a')\}$

For assignments

- Remove any expression that contains the assigned variable:
  $kill_{AE}([\texttt{x := a}]^\ell) = \{a' \in \mathbf{AExp}_* \mid x \in FV(a')\}$
- Add some or all subexpressions of the assigned expression:
  $gen_{AE}([\texttt{x := a}]^\ell) = \{a' \in \mathbf{AExp}(a) \mid x \notin FV(a')\}$
- Why $x \notin FV(a')$?
- `(a + b) * x`, computed in 1, is not available before 2:

```
[x := (a + b) * x]1;
if [(a + b) * x > a + b + 14)]2 then
...
```

- It helps to have side-effect free expressions.

For skip and conditions

- For the remaining blocks, we do the same.
- For skip:
  - $kill_{AE}([\texttt{skip}]^{\ell}) = \emptyset$
  - $gen_{AE}([\texttt{skip}]^{\ell}) = \emptyset$
- For conditions:
  - $kill_{AE}([\texttt{b}]^{\ell}) = \emptyset$
  - $gen_{AE}([\texttt{b}]^{\ell}) = \mathbf{AExp}(b)$
- We only save arithmetic expressions, not complete boolean ones.
  - Higher precision leads to higher costs.

Analysis functions for Available Expressions

Flow functions:

$$AE_{\mathrm{N}}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \mathtt{init}(S_*) \\ \bigcap\{AE_{\mathrm{X}}(\ell') \mid (\ell', \ell) \in \mathtt{flow}(S_*)\} & \text{otherwise} \end{cases}$$

Transfer functions:

$$AE_{\mathrm{X}}(\ell) = (AE_{\mathrm{N}}(\ell) - \mathit{kill}_{AE}(B^{\ell})) \cup \mathit{gen}_{AE}(B^{\ell})$$

- Equations or assignments?

Example continued

```
[x := (a + b) * x]1;
[y := a * b]2;
while [a * b > a + b]3 do
   ([a := a + 1 ]4; [x := a + b]5)
```

ℓ	$kill_{AE}(\ell)$	$gen_{AE}(\ell)$
1	$\{(a+b)*x\}$	$\{a+b\}$
2	$\emptyset$	$\{a*b\}$
3	$\emptyset$	$\{a*b, a+b\}$
4	$\{a*b, a+b, (a+b)*x, a+1\}$	$\emptyset$
5	$\{(a+b)*x\}$	$\{a+b\}$

Example continued

```
[x := (a + b) * x]1;
[y := a * b]2;
while [a * b > a + b]3 do
   ([a := a + 1 ]4; [x := a + b]5)
```

ℓ	$AE_{\mathrm{N}}(\ell)$	$AE_{\mathrm{X}}(\ell)$
1	$\emptyset$	$(AE_{\mathrm{N}}(1) - \{(a+b)*x\}) \cup \{a+b\}$
2	$AE_{\mathrm{X}}(1)$	$AE_{\mathrm{N}}(2) \cup \{a*b\}$
3	$AE_{\mathrm{X}}(2) \cap AE_{\mathrm{X}}(5)$	$AE_{\mathrm{N}}(3) \cup \{a*b, a+b\}$
4	$AE_{\mathrm{X}}(3)$	$AE_{\mathrm{N}}(4) - \{a*b, a+b, (a+b)*x, a+1\}$
5	$AE_{\mathrm{X}}(4)$	$(AE_{\mathrm{N}}(5) - \{(a+b)*x\}) \cup \{a+b\}$

Performing Chaotic Iteration

ℓ	$AE_{\mathrm{N}}(\ell)$	$AE_{\mathrm{X}}(\ell)$
1	$\emptyset$	$(AE_{\mathrm{N}}(1) - \{(a+b)*x\}) \cup \{a+b\}$
2	$AE_{\mathrm{X}}(1)$	$AE_{\mathrm{N}}(2) \cup \{a*b\}$
3	$AE_{\mathrm{X}}(2) \cap AE_{\mathrm{X}}(5)$	$AE_{\mathrm{N}}(3) \cup \{a*b, a+b\}$
4	$AE_{\mathrm{X}}(3)$	$AE_{\mathrm{N}}(4) - \{a*b, a+b, (a+b)*x, a+1\}$
5	$AE_{\mathrm{X}}(4)$	$(AE_{\mathrm{N}}(5) - \{(a+b)*x\}) \cup \{a+b\}$

$AE_{\mathrm{N}}(1)$	$\mathbf{AExp}_*$	$\emptyset$	$\emptyset$	$\emptyset$
$AE_{\mathrm{X}}(1)$	$\mathbf{AExp}_*$	$\{a+b\}$	$\{a+b\}$	$\{a+b\}$
$AE_{\mathrm{N}}(2)$	$\mathbf{AExp}_*$	$\{a+b\}$	$\{a+b\}$	$\{a+b\}$
$AE_{\mathrm{X}}(2)$	$\mathbf{AExp}_*$	$\{a+b, a*b\}$	$\{a+b, a*b\}$	$\{a+b, a*b\}$
$AE_{\mathrm{N}}(3)$	$\mathbf{AExp}_*$	$\{a+b, a*b\}$	$\{a+b\}$	$\{a+b\}$
$AE_{\mathrm{X}}(3)$	$\mathbf{AExp}_*$	$\{a+b, a*b\}$	$\{a+b, a*b\}$	$\{a+b, a*b\}$
$AE_{\mathrm{N}}(4)$	$\mathbf{AExp}_*$	$\{a+b, a*b\}$	$\{a+b, a*b\}$	$\{a+b, a*b\}$
$AE_{\mathrm{X}}(4)$	$\mathbf{AExp}_*$	$\emptyset$	$\emptyset$	$\emptyset$
$AE_{\mathrm{N}}(5)$	$\mathbf{AExp}_*$	$\emptyset$	$\emptyset$	$\emptyset$
$AE_{\mathrm{X}}(5)$	$\mathbf{AExp}_*$	$\{a+b\}$	$\{a+b\}$	$\{a+b\}$

A more mathematical formulation

- For every program point ℓ, we have a finite set $AE_{\mathrm{N}}(\ell)$ and $AE_{\mathrm{X}}(\ell)$.
- Total analysis information for the program is a tuple containing all these sets:

$$\overrightarrow{\mathsf{AE}} = (AE_{\mathrm{N}}(1), AE_{\mathrm{X}}(1), \ldots, AE_{\mathrm{N}}(5), AE_{\mathrm{X}}(5))$$

- Initialization:

$$\overrightarrow{\mathsf{AE}} = (\mathbf{AExp}_*, \mathbf{AExp}_*, \ldots, \mathbf{AExp}_*, \mathbf{AExp}_*)$$

- Why not at $\overrightarrow{\mathsf{AE}} = (\emptyset, \ldots, \emptyset)$?

A single "parallel" transfer function

- Equations implicitly define separate transformations on $\overrightarrow{\text{AE}}$:

$$F_{\text{entry}}(3)(\ldots, AE_{\text{X}}(2), \ldots, AE_{\text{X}}(5)) = AE_{\text{X}}(2) \cap AE_{\text{X}}(5)$$

$$F_{\text{exit}}(3)(\ldots, AE_{\text{N}}(3), \ldots) = AE_{\text{N}}(3) \cup \{a * b, a + b\}$$

- Together give a transformation function F, applying the separate transformations elementwise.
- F maps column to column in every single iteration.
  - Not as greedy as Chaotic Iteration

Iterating

- We iterate F, by computing

```
initialize(AE);
while (AE != F(AE)) do
  AE = F(AE);
output solution AE;
```

- A fixpoint (or fixed point) of F is an X so that $F(X) = X$.
- The fixpoint $\overrightarrow{\mathsf{AE}}$ satisfies the equations: $F(\overrightarrow{\mathsf{AE}}) = \overrightarrow{\mathsf{AE}}$.
- Moreover, going on does not help: $F(F(\overrightarrow{\mathsf{AE}})) = \overrightarrow{\mathsf{AE}}$.

Intuitive reading

- We start from our most favourite, most informative answer.
- Iterating makes the values less informative, but also more consistent with the equations.
- We repeat until it is consistent.

Termination

- Does the iteration ever end?
  - No cyclic behaviour: sets in $\overrightarrow{\text{AE}}$ can only shrink.
  - Solutions can not shrink indefinitely:
    - bounded by $\emptyset$ from below, and
    - $\mathbf{AExp}_*$ is finite to begin with.
  - The transfer functions themselves terminate
- Together: computation of a fixed point terminates.

Best possible solution

- The solution is a least fixed point: no avoidable information is included.
- That is, no avoidable information according to the equations.
  - Imprecision comes from imprecision in the equations, not their solution.
- Although F changes all sets in parallel, the separate sets may also be transformed non-deterministically in any order.
- The latter is in fact done when using Chaotic Iteration.

Avoid cyclic behaviour: monotonocity

- Iterating makes the solution less useful.
- $X \sqsubseteq Y$ means that X is at least as useful as Y
  - With AE, $\{a+b, a*b\} \sqsubseteq \{a+b\}$ (Note: $\sqsubseteq = \supseteq$)
- Being less useful should not be an asset: transfer functions must be monotone
- F is monotone if $\overrightarrow{\text{AE}} \sqsubseteq \overrightarrow{\text{AE'}}$ implies $F(\overrightarrow{\text{AE}}) \sqsubseteq F(\overrightarrow{\text{AE'}})$
- For AE, $F(\{a+b\}) \sqsubseteq F(\{a+b, a*b\})$ would imply non-monotonic behaviour.
  - It paid off for $\{a+b\}$ to be less useful than $\{a+b, a*b\}$.
- Monotonocity does not mean that $\overrightarrow{\text{AE}} \sqsubseteq F(\overrightarrow{\text{AE}})$.

Verify that analysis functions are monotone!

- Usually done by verifying that the separate transformations, like $F_{\mathsf{entry}}(3)$, are monotone.
- Recall: $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ and $\sqsubseteq = \supseteq$
- For $F_{\mathsf{entry}}(3) = AE_{\mathrm{X}}(2) \cap AE_{\mathrm{X}}(5)$:

$$AE_{\mathrm{X}}(2) \supseteq AE'_{\mathrm{X}}(2) \text{ and } AE_{\mathrm{X}}(5) \supseteq AE'_{\mathrm{X}}(5)$$

implies

$$AE_{\mathrm{X}}(2) \cap AE_{\mathrm{X}}(5) \supseteq AE'_{\mathrm{X}}(2) \cap AE'_{\mathrm{X}}(5) \ .$$

- If separate transformations are monotone, then so is F.

AE is a forward analysis

$$AE_{\mathrm{N}}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \mathtt{init}(S_*) \\ \bigcap\{AE_{\mathrm{X}}(\ell') \mid (\ell', \ell) \in \mathtt{flow}(S_*)\} & \text{otherwise} \end{cases}$$

- Analysis information flows in the direction of program execution.
- Starting from the beginning of the program.
- In the formulas: we use $\mathtt{flow}$ rather than $\mathtt{flow}^R$.

AE is a must analysis

```
[z := x + y]^1;
while [true]^2 do
   [skip]^3
```

- Writing down the equations, and substituting, you get

$$AE_{\mathrm{N}}(2) = \{x + y\} \cap AE_{\mathrm{N}}(2)$$

- Fixpoints not unique: $\emptyset$ and $\{x + y\}$ are both okay.
- Most informative solution is $\{x + y\}$, so we choose that one.
- Must analysis: use $\cap$ not $\cup$ in the flow equations.
  - All execution paths must make the expressions available.

Live Variables Analysis

Live Variables Analysis by example

- $[\texttt{x := 2}]^1$; $[\texttt{y := 4}]^2$; $[\texttt{x := 1}]^3$;
  (if $[\texttt{B}]^4$ then $[\texttt{z := y}]^5$
  else $[\texttt{z := x*x}]^6$);
  $[\texttt{x := z}]^7$;
- Variable x is not live at the exit of 1
- It is live at the exit of 3,
  - unless we know that $[\texttt{B}]^4$ is never false, and B does not mention x.
- Assignments to dead variables is dead code and might be removed
- In contrast with AE, LV is a backward may analysis

Dealing with assignments

- As usual, assignments are the tricky case
- Consider

  $[\texttt{z := x - z}]^2$

  $[\texttt{x := z + a}]^3$

  and assume we are interested in the values of $\{x, z\}$ at program end.
- Reasoning backward:
  - `x` is defined by 3, so the `x` live after 3 is not "the same" as before 3, so we kill `x`. To compute `x` (that is live after 3) we need `a` and `z`, they become live (but only `a` is newly live)
  - So between 2 and 3, {`a`,`z`} are live
  - Before 2, {`a`,`x`,`z`} are live: `z` is defined, so killed, but the value of `z` that lives before 2 is used in the definition of `z`.

Transfer functions for Live Variables Analysis

$$LV_{\mathrm{X}}(\ell) = \begin{cases} V & \text{if } \ell \in \mathtt{final}(S_*) \\ \bigcup\{LV_{\mathrm{N}}(\ell') \mid (\ell', \ell) \in \mathtt{flow}^R(S_*)\} & \text{otherwise} \end{cases}$$

$$LV_{\mathrm{N}}(\ell) = (LV_{\mathrm{X}}(\ell) - \mathit{kill}_{LV}(B^\ell)) \cup \mathit{gen}_{LV}(B^\ell)$$

Note: V denotes the variables of interest (at program end).

$$\begin{aligned} \mathit{kill}_{LV}([\mathtt{x := a}]^\ell) &= \{x\} \\ \mathit{kill}_{LV}([\mathtt{skip}]^\ell) &= \emptyset \\ \mathit{kill}_{LV}([\mathtt{b}]^\ell) &= \emptyset \end{aligned}$$

$$\begin{aligned} \mathit{gen}_{LV}([\mathtt{x := a}]^\ell) &= FV(a) \\ \mathit{gen}_{LV}([\mathtt{skip}]^\ell) &= \emptyset \\ \mathit{gen}_{LV}([\mathtt{b}]^\ell) &= FV(b) \end{aligned}$$

An example

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

ℓ	$kill_{LV}(\ell)$	$gen_{LV}(\ell)$
1	$\{y\}$	$\{x\}$
2	$\{z\}$	$\emptyset$
3	$\emptyset$	$\{x\}$
4	$\{z\}$	$\{z, x\}$
5	$\{x\}$	$\{x\}$
6	$\{x\}$	$\emptyset$

ℓ	$LV_{\mathrm{X}}(\ell)$	$LV_{\mathrm{N}}(\ell)$
1	$LV_{\mathrm{N}}(2)$	$(LV_{\mathrm{X}}(1) - \{y\}) \cup \{x\}$
2	$LV_{\mathrm{N}}(3)$	$LV_{\mathrm{X}}(2) - \{z\}$
3	$LV_{\mathrm{N}}(4) \cup LV_{\mathrm{N}}(6)$	$LV_{\mathrm{X}}(3) \cup \{x\}$
4	$LV_{\mathrm{N}}(5)$	$(LV_{\mathrm{X}}(4) - \{z\}) \cup \{z, x\}$
5	$LV_{\mathrm{N}}(3)$	$(LV_{\mathrm{X}}(5) - \{x\}) \cup \{x\}$
6	$\{z\}$	$LV_{\mathrm{X}}(6) - \{x\}$

A few computations for Live Variables

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

- Variable of interest: z
- Conclusion: y is not live anywhere so assignment 1 is dead code.

$LV_{\mathrm{X}}(6)$	$\emptyset$	$\{z\}$	$\{z\}$
$LV_{\mathrm{N}}(6)$	$\emptyset$	$\{z\}$	$\{z\}$
$LV_{\mathrm{X}}(5)$	$\emptyset$	$\emptyset$	$\{x, z\}$
$LV_{\mathrm{N}}(5)$	$\emptyset$	$\{x\}$	$\{x, z\}$
$LV_{\mathrm{X}}(4)$	$\emptyset$	$\{x\}$	$\{x, z\}$
$LV_{\mathrm{N}}(4)$	$\emptyset$	$\{x, z\}$	$\{x, z\}$
$LV_{\mathrm{X}}(3)$	$\emptyset$	$\{x, z\}$	$\{x, z\}$
$LV_{\mathrm{N}}(3)$	$\emptyset$	$\{x, z\}$	$\{x, z\}$
$LV_{\mathrm{X}}(2)$	$\emptyset$	$\{x, z\}$	$\{x, z\}$
$LV_{\mathrm{N}}(2)$	$\emptyset$	$\{x\}$	$\{x\}$
$LV_{\mathrm{X}}(1)$	$\emptyset$	$\{x\}$	$\{x\}$
$LV_{\mathrm{N}}(1)$	$\emptyset$	$\{x\}$	$\{x\}$

Live Variables Analysis is a backward analysis

- Backward analysis:
  - Variables used in an assignment are live before the assignment.
  - Variables assigned to are not live before the assignment (except when also used)
- Analysis information moves contrary to execution direction.
- Speed up iteration by starting at program's end.
- If we are not interested in any variable at the end, which variables are then live?

The smaller the better

- Consider

```
while [x>1]^1 do
    [skip]^2;
[y := x + 1]^3
```

- Substition gives $LV_{\mathrm{X}}(1) = LV_{\mathrm{X}}(1) \cup \{x\}$.
- Two safe solutions are $\{x, y\}$ and $\{x\}$.
- The more variables dead (not live), the more we can optimize: we choose $\{x\}$.
- Hence, we start small and grow our sets, by using $\cup$ (may).

Monotone Frameworks

Monotone Frameworks

- A framework that generalizes the example analyses
  - Making them instances
- Identify the commonalities, parameterize by the differences
- Advantages:
  - generic algorithms,
  - generic proof methods for soundness, and
  - less ad-hoc tends to provide better understanding.
- Disadvantage:
  - mathematically more challenging
  - algorithms cannot take advantage of special properties of any specific analysis.

Steps to take

Generalize over

- direction (capture backward and forward analyses)
- datatype (the payload of the analysis)
- modality (capture may and must)

From entry and exit to context and effect

- Thus far, we had an entry and exit set for each label/program point.
- Now, for each label ℓ we shall have
  - $\text{Analysis}_\circ(\ell)$ or the context value: values come from the context of $[\mathtt{B}]^\ell$
  - $\text{Analysis}_\bullet(\ell)$ or effect value: it shows the effect of $[\mathtt{B}]^\ell$ on $\text{Analysis}_\circ(\ell)$
- $\text{Analysis}_\bullet(\ell)$ is defined in terms of $\text{Analysis}_\circ(\ell)$, and
- $\text{Analysis}_\circ(\ell)$ is defined in terms of the $\text{Analysis}_\bullet$ values of other blocks.
- For LV, the context values are the exit sets (backward).
- For AE, the context values are the entry sets (forward).

The formula for $\text{Analysis}_\bullet(\ell)$

- Recall: these describe the effect of the blocks.
- The generic transfer functions:

$$\text{Analysis}_\bullet(\ell) = f_\ell(\text{Analysis}_\circ(\ell))$$

- f_ℓ is the transfer function for $[\mathtt{B}]^\ell$.
- Note: transfer functions can be given per block.
- Thus far, we have specified them uniformly for each language construct.

The formula for $\text{Analysis}_\circ(\ell)$

$$\text{Analysis}_\circ(\ell) = \begin{cases} \iota & \text{if } \ell \in E \\ \bigsqcup\{\text{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F\} & \text{otherwise} \end{cases}$$

- Combination operator $\bigsqcup$ is $\bigcap$ (for *must*) or $\bigcup$ (for *may*)
- F is either $\texttt{flow}(S_*)$ (forward) or its reverse $\texttt{flow}^R(S_*)$ (backward).
- E is the set of extremal labels, e.g. $\{\texttt{init}(S_*)\}$ or $\texttt{final}(S_*)$
- ι is the extremal value for the extremal labels
- 4 combinations: backward vs. forward and must vs. may.

What is wrong with Analysis$_\circ(\ell)$?

- Formula is not correct when $\exists(\ell', \ell) \in F$ with $\ell \in E$.
  - Forward analysis of a program starting with a while loop
  - Backward analysis of a program ending in a while loop
- Consider LV analysis for

  ```
  while [x > 1]^1 do
     [x := x-1]^2
  ```

- We want

$$\text{Analysis}_\circ(1) = \text{Analysis}_\bullet(2) \cup V$$

  and not simply

$$\text{Analysis}_\circ(1) = V \ .$$

- Workaround: start program with skip and end it with skip.

Fixing the formula for $\text{Analysis}_\circ(\ell)$

- Or, the formula for $\text{Analysis}_\circ(\ell)$ should read

$$\text{Analysis}_\circ(\ell) = \bigsqcup\{\text{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota^\ell_E$$

where

$$\iota^\ell_E = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}$$

- Here, $\bot$ (pronounced "bottom") is the zero of $\sqcup$.
  - For all a: $a \sqcup \bot = a$.

Example Available Expressions

- $[$`x := (a + b) * x`$]^1$;
  $[$`y := a * b`$]^2$;
  `while` $[$`a * b > a + b`$]^3$ `do`
  (($[$`a := a + 1`$]^4$;
  $[$`x := a + b`$]^5$)
- In this case:
  - $\bigsqcup = \bigcap$
  - $F = \{(1,2),(2,3),(3,4),(4,5),(5,3)\}$
  - $E = \{1\}$
  - $\iota = \emptyset$
  - $\bot = \mathbf{AExp}_*$ (because $x \cap \mathbf{AExp}_* = x$)
- Transfer functions f_ℓ will have to wait a bit.

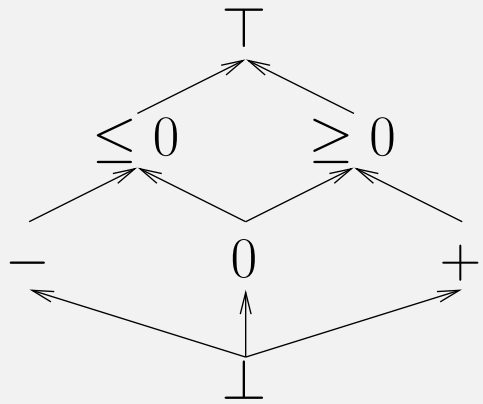
Lattices and the ACC

Fixed points

- Declarative, constraint-based specification of static analysis:
  - specifies all admissible/sound solutions.
- Algorithmically: find the best solution in finite time.
- Best solution is a so-called least fixed point of a function that can be derived from this set of constraints.
- *In the interest of definedness and termination, this is a monotone function computed on a complete lattice that satisfies the Ascending Chain Condition.*
- Come back to read this statement at a later time.

Introductory example to lattices

- Take a set of values, say $\{\bot, -, 0, +, \leq 0, \geq 0, \top\}$.
  - These approximate sets of integers by means of signs
- $\bot$ (pron. bottom) represents $\{\}$ (or $\emptyset$).
- $\top$ (pron. top) represents the set of all integers
- Various relations hold:
  - 0 is more precise than ≤ 0, but also more precise than ≥ 0
  - $\bot$ is more precise than all the others
  - ≤ 0 and ≥ 0 are not comparable
- Represent relations visually in Hasse diagram:

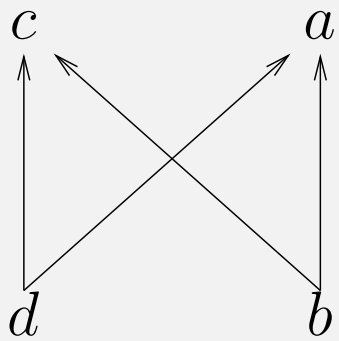


Partial orders

- A binary relation $\sqsubseteq$ on (L, L) (or $L \times L$) is given.
- For simplicity, instead of $(x, y) \in \sqsubseteq$ we write $x \sqsubseteq y$.
- The relation $\sqsubseteq$ is a partial order if it is
  - reflexive: for all $x \in L$, $x \sqsubseteq x$
  - transitive: for all $x, y, z \in L$, if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$
  - anti-symmetric: if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$.

Partial orders

- A binary relation $\sqsubseteq$ on (L, L) (or $L \times L$) is given.
- For simplicity, instead of $(x, y) \in \sqsubseteq$ we write $x \sqsubseteq y$.
- The relation $\sqsubseteq$ is a partial order if it is
  - reflexive: for all $x \in L$, $x \sqsubseteq x$
  - transitive: for all $x, y, z \in L$, if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$
  - anti-symmetric: if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$.
- Examples:
  - $\subseteq$ for subsets of a given set S, and similarly $\supseteq$.
  - $\leq$ and $\geq$ are partial orders on the natural numbers **N**, and so is $=$.
- Partial order P conventionally drawn as a Hasse diagram:

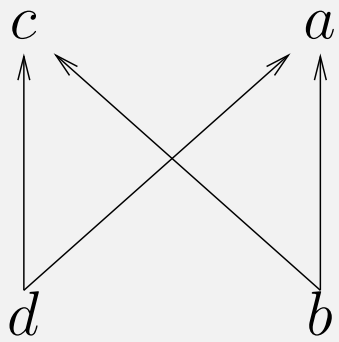


Lattices

- Let $x, y \in L$.
- $z \in L$ is an upper bound of x and y if $x \sqsubseteq z$ and $y \sqsubseteq z$
  - And similarly for lower bound
- If the set of upper (lower) bounds of x and y has a least (greatest) element, then we call it the join (meet) of x and y, written $x \sqcup y$ ($x \sqcap y$).
- $z \in S$ is the least (greatest) element of $S \subseteq L$, if for all $v \in S$, $z \sqsubseteq v$ ($v \sqsubseteq z$)
- A partial order is called a lattice (tralie in Dutch) if for all $x, y \in L$, $x \sqcup y$ and $x \sqcap y$ exist.
  - If they exist, they are unique
- Reason: we want $\sqcup$ and $\sqcap$ to be total binary functions, i.e., binary operators.

Example lattices

- $(\mathbf{N}, =)$ is not a lattice: $x \sqcup y$ is undefined for all $x \neq y$.
- $(\mathbf{N}, \leq)$ and $(\mathbf{N}, \geq)$ are (dual) lattices.
- The partial order P is not a lattice.



- Duality: reversing all edges in the lattice gives another lattice.

Complete lattices

- Consider a subset $X = \{x_1, x_2, \ldots\}$ of the lattice L.
- Then $\bigsqcup X$ is well-defined for finite non-empty X: $x_1 \sqcup (x_2 \sqcup (\ldots x_n) \ldots))$.
- What about the infinite or empty X's?
- In a complete lattice, $\bigsqcup X$ and $\bigsqcap X$ are defined and unique for all $X \subseteq L$.
- $\bigsqcup \emptyset = \bot$ and $\bigsqcup L = \top$.

Complete lattices

- In a complete lattice, $\bigsqcup X$ and $\bigsqcap X$ are defined and unique for all $X \subseteq L$.
- Is every finite lattice complete?

Complete lattices

- In a complete lattice, $\bigsqcup X$ and $\bigsqcap X$ are defined and unique for all $X \subseteq L$.
- Is every finite lattice complete?
- One exception

Complete lattices

- In a complete lattice, $\bigsqcup X$ and $\bigsqcap X$ are defined and unique for all $X \subseteq L$.
- Is every finite lattice complete?
- One exception: empty set
- Every non-empty finite lattice is complete.

Complete lattices

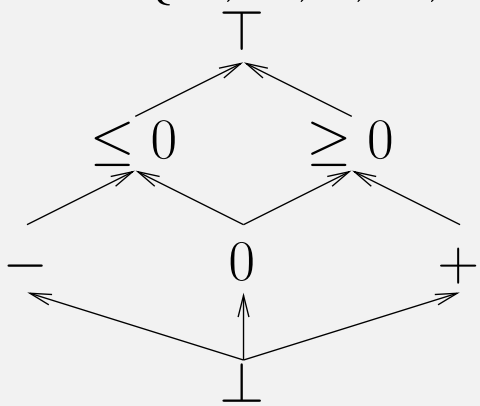
- In a complete lattice, $\bigsqcup X$ and $\bigsqcap X$ are defined and unique for all $X \subseteq L$.
- Is every finite lattice complete?
- One exception: empty set
- Every non-empty finite lattice is complete.
- Is every lattice with top and bottom complete?

Complete lattices

- In a complete lattice, $\bigsqcup X$ and $\bigsqcap X$ are defined and unique for all $X \subseteq L$.
- Is every finite lattice complete?
- One exception: empty set
- Every non-empty finite lattice is complete.
- Is every lattice with top and bottom complete?
- No, there can still be infinite subsets without a lower or upperbound.

Examples

- Subsets of $S = \{0, 1, 2\}$ form a complete lattice ($\sqsubseteq$ is $\subseteq$). Then $\sqcup$ equals $\cup$, and $\emptyset$ is smallest and S largest element.
- Dually, $(S, \supseteq)$ is also one: $\sqcup$ equals $\cap$, $\bot = S$, $\top = \emptyset$.
- $(\mathbf{N}, \leq)$ is a lattice, but has no $\top$. Here, $x \sqcup y = \max(x, y)$.
- $(\mathcal{P}(\mathbf{N}), \subseteq)$ with $\emptyset$ as bottom, $\mathbf{N}$ as top. Here $\sqcup = \cup$.
  - An infinite complete lattice
- $L = \{\bot, -, 0, +, \leq 0, \geq 0, \top\}$ for sign testing



An aside: computational aspects

- How to define lattices or complete lattices in `Haskell`?
- Preferably, like `Eq` and `Ord`, as a type class.
- Preferably most definitions have a default implementation.
- Enforcing algebraic laws is difficult (within the type system).
- $\sqcup$ and $\sqcap$ are associative, commutative binary operators.
- Relation: $x \sqsubseteq y$ if and only if $x \sqcup y = y$.
- Defining $\sqcup$ in terms of $\sqsubseteq$ implies a search of some kind.
- Other way around is direct.
- Provide the lattice with bottom and top element (implicit or explicit).
- Different lattices can be made on the same underlying set!

The ascending chain condition (ACC)

- Necessary to assure needing only a finite number of iterations during fixed point computation.
- Every chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ in the lattice stabilizes: there is an n where $x_n = x_{n+1}$.
  - We can only go *up* a finite number of times
- For finite lattices: ACC trivially satisfied
- ACC holds for $(\mathbf{N}, \geq)$ (top is 0), but not for $(\mathbf{N}, \leq)$
- A lattice with ACC and a bottom element is complete.

The descending chain condition (DCC)

- Descending Chain Condition is the dual.
- Ascending vs. Descending Chain Condition: turn the lattice around.
- $(\mathbf{Z}, \leq)$ has neither ACC or DCC.

Termination of fixpoint algorithm, formally

- $X = \bot$;
  `while` $(X != F(X))$ `do`
  $X = F(X)$;
  where
  - X has datatype T,
  - T forms a lattice with bottom element $\bot$,
  - T has Ascending Chain Condition, and
  - $F : T \to T$ monotone.
- Thm: least fixed point found in finite time.
- Proof by two inductions.
- Base case: by definition $\bot = F^0(\bot) \sqsubseteq F(\bot)$,
- Inductive case: by monotonicity $F^{n-1}(\bot) \sqsubseteq F^n(\bot)$ implies $F^n(\bot) \sqsubseteq F^{n+1}(\bot)$
- ACC now implies, the chain $\bot \sqsubseteq F(\bot) \sqsubseteq F^2(\bot) \ldots$ stabilizes.

Solution is the least fixed point

- $X = \bot$;
  `while` $(X != F(X))$ `do`
  $X = F(X)$;
  where
  - X has datatype T,
  - T forms a lattice with bottom element $\bot$,
  - T has Ascending Chain Condition, and
  - $F : T \to T$ monotone.
- Let S be another fixed point of F: $F(S) = S$
- Prove $F^n(\bot) \sqsubseteq S$ for all n, by induction.
- Base case: by definition $\bot = F^0(\bot) \sqsubseteq S$
- Inductive case: assume $F^n(\bot) \sqsubseteq S$.
  Then $F^{n+1}(\bot) = F(F^n(\bot)) \sqsubseteq F(S) = S$, because F is monotone.

Back to Monotone Frameworks

Property spaces: the data type of the analysis

- Values for Analysis$_\circ$ and Analysis$_\bullet$ taken from the MF's property space L.
- Choosing a complete lattice for L provides us with
  - a join operator $\sqcup$ to combine multiple values into a single one consistent with both.
    - for converging execution paths
  - It provides the most precise value with that property.
- ACC ensures termination of fixed point computation
- Least element $\bot$ can be used to initialize the computation
  - Intuitively, $\bot$ represents *most informative* element of L
- Greatest element $\top$ (usually) means *no useful* or *inconsistent information*

Examples LV

- Live Variables (for program S_*):
  - $L = \mathcal{P}(\mathbf{Var}_*)$, finite sets of variables,
  - for $x, y \in L$: $x \sqsubseteq y$ if and only if $x \subseteq y$,
  - $\sqcup = \cup$,
  - $\bot = \emptyset$ and $\top = \mathbf{Var}_*$.
- Why not $L = \mathcal{P}(\mathbf{Var})$ so that it is the same for all programs?
  - To get a finite lattice and thus automatically ACC.
  - ACC is sufficient, but not necessary: only variables in $\mathbf{Var}_*$ will be added.

Example AE

- Available Expressions (for program S_*):
  - $L = \mathcal{P}(\mathbf{AExp}_*)$, non-trivial subexpressions of S_*,
  - for $x, y \in L$: $x \sqsubseteq y$ if and only if $x \supseteq y$,
  - $\sqcup = \cap$,
  - $\bot = \mathbf{AExp}_*$ and $\top = \emptyset$.

Transfer functions: the dynamics of the analysis

- Start with a collection $\mathcal{F}$ of *monotone* functions on the property space L:

$$\mathcal{F} \subseteq \{ f \mid f : L \to L \text{ and } f \text{ monotone} \} .$$

- Recall: a function f is monotone if

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y) .$$

- $id \in \mathcal{F}$ (for the empty sequence of statements (and `skip`))
- $\mathcal{F}$ closed under function composition $\circ$ (for the sequencing of statements)

Transfer functions: the dynamics of the analysis

- Start with a collection $\mathcal{F}$ of *monotone* functions on the property space L:

$$\mathcal{F} \subseteq \{ f \mid f : L \to L \text{ and } f \text{ monotone} \} .$$

- Recall: a function f is monotone if

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y) .$$

- $id \in \mathcal{F}$ (for the empty sequence of statements (and `skip`))
- $\mathcal{F}$ closed under function composition $\circ$ (for the sequencing of statements)
- For a given program and analysis, we specify for each label a transfer function $f_\ell : L \to L$, all from $\mathcal{F}$.

Finally, monotone frameworks

- A Monotone Framework consists of a property space L and a set $\mathcal{F}$ of monotone functions, as well as
  - the flow F of the program
  - the extremal labels E
  - an extremal value $\iota \in L$
  - a mapping $f_\cdot$ from the labels $\mathbf{Lab}_*$ to functions in $\mathcal{F}$

Example Available Expressions continued

- $[\texttt{x := (a + b) * x}]^1$;
  $[\texttt{y := a * b}]^2$;
  $\texttt{while } [\texttt{a * b > a + b}]^3 \texttt{ do}$
  $([\texttt{a := a + 1}]^4;$
  $[\texttt{x := a + b}]^5)$
- $(L, \sqsubseteq) = (\mathcal{P}(\textbf{AExp}_*), \supseteq)$ as earlier.
- $F = \texttt{flow}_* = \{(1,2),(2,3),(3,4),(4,5),(5,3)\}$,
- $E = \{\texttt{init}(S_*)\} = \{1\}$
- $\iota = \emptyset$
- The function space $\mathcal{F}$ could be all functions of the form $\{f : L \to L \mid \exists l_k, l_g : f(l) = (l - l_k) \cup l_g\}$.
  - All functions that first remove and then add
- $f_\ell(l) = (l - \mathit{kill}_{AE}([\texttt{B}]^\ell)) \cup \mathit{gen}_{AE}([\texttt{B}]^\ell)$ where $[\texttt{B}]^\ell \in \texttt{blocks}(S_*)$

Available Expressions is a Monotone Framework

- Recall $\mathcal{F} = \{f : L \to L \mid \exists l_k, l_g : f(l) = (l - l_k) \cup l_g\}$ and $\sqsubseteq$ equals $\supseteq$.
- Identity function exists in $\mathcal{F}$: take $l_k = l_g = \emptyset$.
- $\mathcal{F}$ is closed under composition: let $f(\ell) = (l - l_k) \cup l_g, f'(\ell) = (l - l'_k) \cup l'_g \in \mathcal{F}$.
  $(f \circ f')(l) = f(f'(l)) = (((l - l'_k) \cup l'_g) - l_k) \cup l_g =$
  $(l - (l'_k \cup l_k)) \cup ((l'_g - l_k) \cup l_g)$
- Thus, kill set for $f \circ f'$ is $l'_k \cup l_k$ and gen set is $(l'_g - l_k) \cup l_g$.
- Monotonicity of $f \in \mathcal{F}$: let $l \supseteq l'$. Then $l - l_k \supseteq l' - l_k$ and finally $(l - l_k) \cup l_g \supseteq (l' - l_k) \cup l_g$

Reflections on burden of proof

- Proof also works when $\sqsubseteq = \subseteq$: other three analyses are also Monotone Frameworks.
- We exploit similarities in the set $\mathcal{F}$ of transfer functions.
  - All analyses choose their transfer functions from $\mathcal{F}$.
  - Easily seen because it is a syntactic property of the functions.
  - One proof works for all.
- Another advantage: each function can be represented by two sets.
- Starting with $\mathcal{F}$ as the set of all monotone functions only moves the burden, and does not allow reuse.

Distributivity vs. Constant Propagation

Constant Propagation

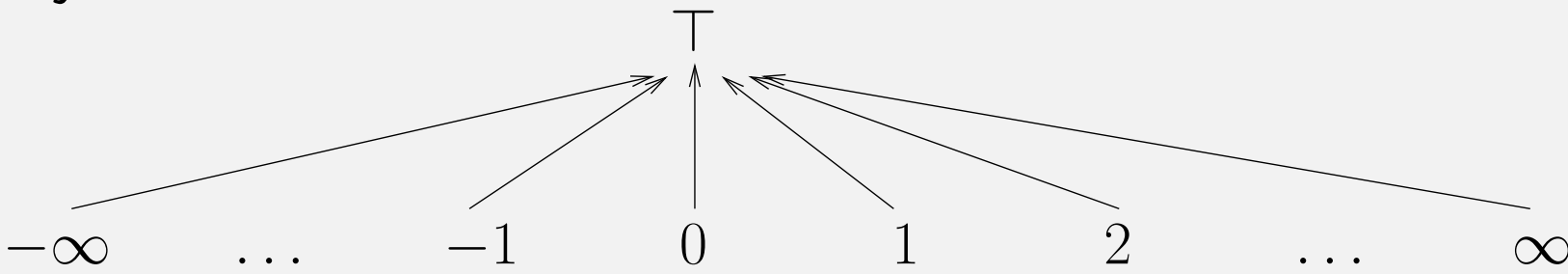
- Constant Propagation: Determine at each program point and for each variable whether the variable always has the same value there.
- We are not interested to see which variables never change
  - Although we shall find that out too
- For every variable we either know
  - the single integer value it can have at that point
  - a special $\top$ value signifying its value is not always the same at that point

Some examples

- $[\texttt{y := 2}]^2$; $[\texttt{z := 1}]^3$;
  $\texttt{while } [\texttt{x>0}]^4 \texttt{ do } ([\texttt{z := z * y}]^5; [\texttt{x := x - 1}]^6)$;
  - $\text{Analysis}_\bullet(3) = [x \mapsto \top, y \mapsto 2, z \mapsto 1]$ and
    $\text{Analysis}_\circ(4) = [x \mapsto \top, y \mapsto 2, z \mapsto \top]$
- $[\texttt{x := 8}]^1$; $[\texttt{y := 2}]^2$; $[\texttt{z := 1}]^3$;
  $\texttt{while } [\texttt{x>0}]^4 \texttt{ do } ([\texttt{z := z * y}]^5; [\texttt{x := x - 1}]^6)$;
  - $\text{Analysis}_\bullet(3) = [x \mapsto 8, y \mapsto 2, z \mapsto 1]$ and
    $\text{Analysis}_\circ(4) = [x \mapsto \top, y \mapsto 2, z \mapsto \top]$
- $[\texttt{x := 8}]^1$; $[\texttt{z := 1}]^3$;
  $\texttt{while } [\texttt{x>0}]^4 \texttt{ do } ([\texttt{z := z * y}]^5; [\texttt{x := x - 1}]^6)$;
  - We cannot know what values y might take so now
    $\text{Analysis}_\bullet(3) = [x \mapsto 8, y \mapsto \top, z \mapsto 1]$ and
    $\text{Analysis}_\circ(4) = \lambda v.\top$

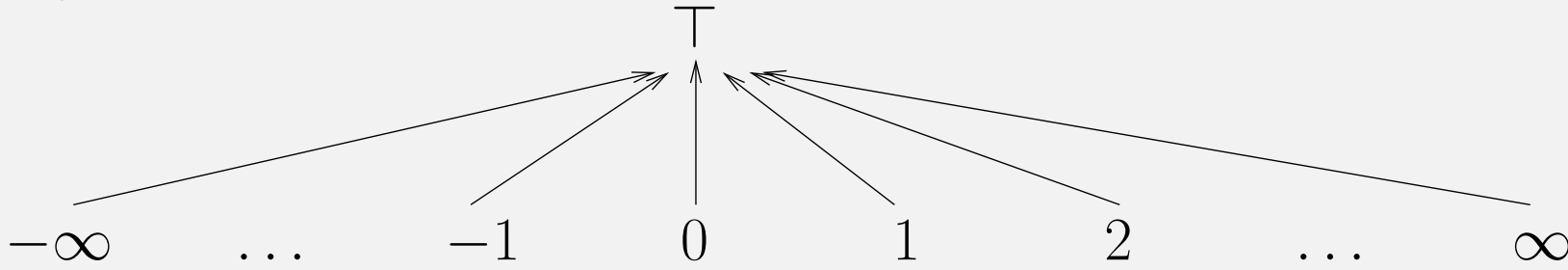
The Constant Propagation lattice

- For values bound to variables we employ the join-semilattice $\mathbf{Z}^{\top}$



The Constant Propagation lattice

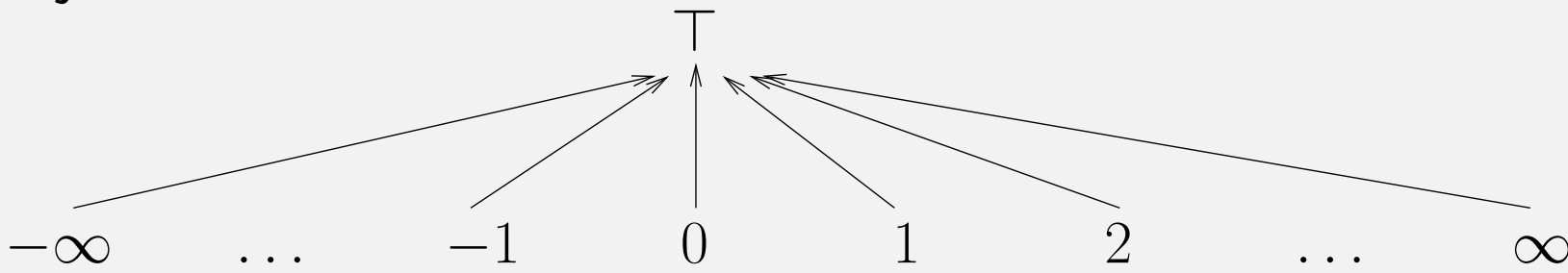
- For values bound to variables we employ the join-semilattice $\mathbf{Z}^\top$



- The property space L is the complete lattice of total functions from $\mathbf{Var}_*$ to $\mathbf{Z}^\top$.

The Constant Propagation lattice

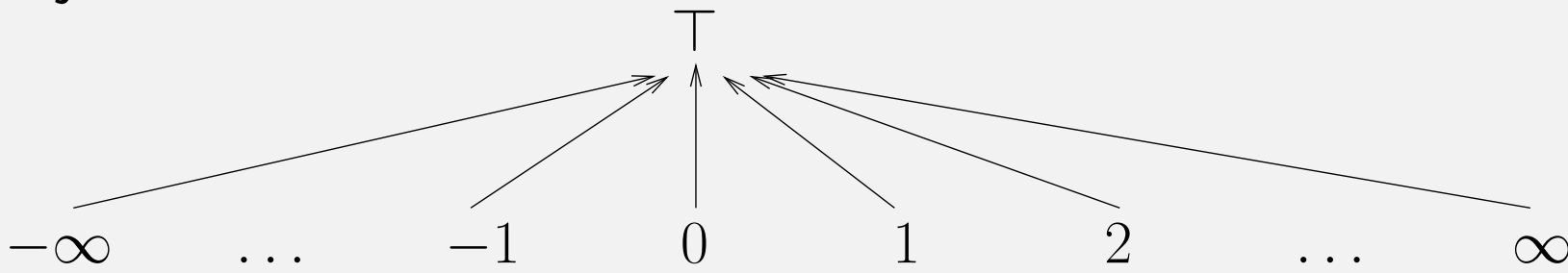
- For values bound to variables we employ the join-semilattice $\mathbf{Z}^\top$



- The property space L is the complete lattice of total functions from $\mathbf{Var}_*$ to $\mathbf{Z}^\top$.
- Add a special element for the always undefined function $\bot$.
- The ordering $\sqsubseteq$ is elementwise for all $\widehat{\sigma}, \widehat{\sigma}' \in L$:
  - $\bot \sqsubseteq \widehat{\sigma}$, and
  - $\widehat{\sigma} \sqsubseteq \widehat{\sigma}'$ if and only if for all $x \in \mathbf{Var}_* : \widehat{\sigma}(x) \sqsubseteq \widehat{\sigma}'(x)$

The Constant Propagation lattice

- For values bound to variables we employ the join-semilattice $\mathbf{Z}^\top$



- The property space L is the complete lattice of total functions from $\mathbf{Var}_*$ to $\mathbf{Z}^\top$.
- Add a special element for the always undefined function $\bot$.
- The ordering $\sqsubseteq$ is elementwise for all $\widehat{\sigma}, \widehat{\sigma}' \in L$:
  - $\bot \sqsubseteq \widehat{\sigma}$, and
  - $\widehat{\sigma} \sqsubseteq \widehat{\sigma}'$ if and only if for all $x \in \mathbf{Var}_* : \widehat{\sigma}(x) \sqsubseteq \widehat{\sigma}'(x)$
- $\mathcal{F}_{CP}$ contains all monotone functions of the correct type.

The transfer functions (different from NNH)

For the three types of statement

$$
\begin{array}{lll}
[\texttt{x := a}]^\ell: & f^{CP}_\ell(\widehat{\sigma}) = \begin{cases} \bot & \text{if } \widehat{\sigma} = \bot \\ \widehat{\sigma}[x \mapsto \mathcal{A}_{CP}[\![a]\!]\widehat{\sigma}] & \text{otherwise} \end{cases} \\
[\texttt{skip}]^\ell: & f^{CP}_\ell(\widehat{\sigma}) = \widehat{\sigma} & \\
[\texttt{b}]^\ell: & f^{CP}_\ell(\widehat{\sigma}) = \widehat{\sigma} &
\end{array}
$$

where we use the function $\mathcal{A}_{CP} : \mathbf{AExp} \to (\mathbf{Var}_* \to \mathbf{Z}^\top) \to \mathbf{Z}^\top$ for evaluation

$$
\begin{array}{ll}
\mathcal{A}_{CP}[\![n]\!]\widehat{\sigma} & = n \\
\mathcal{A}_{CP}[\![x]\!]\widehat{\sigma} & = \widehat{\sigma}(x) \\
\mathcal{A}_{CP}[\![a_1 \ \mathsf{op}_a \ a_2]\!]\widehat{\sigma} & = \mathcal{A}_{CP}[\![a_1]\!]\widehat{\sigma} \ \widehat{\mathsf{op}_a} \ \mathcal{A}_{CP}[\![a_2]\!]\widehat{\sigma}
\end{array}
$$

and it is understood that $x \ \widehat{\mathsf{op}_a} \ y = \begin{cases} x \ \mathsf{op}_a \ y & \text{if } x, y \in \mathbf{Z} \\ \top & \text{otherwise} \end{cases}$

Constant Propagation Analysis example

- ```
  [y := 2]^2;
  [z := 1]^3;
  while [x>0]^4 do
     ([z := z * y]^5;
  [x := x - 1]^6);
  ```
- Initial statement has $\iota = \lambda v.\top$: the only safe answer
- The effect $f_2^{CP}(\iota) = [y \mapsto 2, z \mapsto \top, x \mapsto \top]$
- $f_5^{CP}([y \mapsto 2, z \mapsto 1, x \mapsto \top]) = [y \mapsto 2, z \mapsto 2, x \mapsto \top]$
- The join operator $\sqcup$ proceeds elementwise:
- At first: $\text{Analysis}_\circ(4) = [y \mapsto 2, z \mapsto 1, x \mapsto \top]$
- Later: $\text{Analysis}_\circ(4) = [y \mapsto 2, z \mapsto \top, x \mapsto \top]$, because $z \mapsto 1$ in $\text{Analysis}_\bullet(3)$ and $z \mapsto 2$ in $\text{Analysis}_\bullet(6)$.
  - Joining two different values for a variable leads to $\top$.


  ```

Remarks about Constant Propagation

- Forward analysis
- I use less robust, but simpler notation
- Proof of being a monotone framework is an exercise. Prove that
  - the identity function is an element of $\mathcal{F}_{CP}$
  - $\mathcal{F}_{CP}$ is closed under composition
  - all transfer functions we use are in $\mathcal{F}_{CP}$

Distributivity

Distributivity

- Consider analysis info x_1 and x_2 for two executions leading up to a block
- Two ways to proceed:
  - join before transfer: $f(x_1 \sqcup x_2)$
  - join after transfers: $f(x_1) \sqcup f(x_2)$
- By monotonicity $f(x_1) \sqcup f(x_2) \sqsubseteq f(x_1 \sqcup x_2)$
  - So the second possibility is never worse than the first
- If f is distributive then both ways are equivalent:
  $$f(x_1 \sqcup x_2) \sqsubseteq f(x_1) \sqcup f(x_2).$$
  - In distributive frameworks doing a join before the transfer does not lose information
- Verify that AE is distributive: $f(x \cap x') = f(x) \cap f(x')$
- Distributivity is good: faster algorithms, higher precision.
- Not all monotone frameworks are distributive.

Constant Propagation is not distributive

- Recall distributive: $f(\ell_1 \sqcup \ell_2) \sqsubseteq f(\ell_1) \sqcup f(\ell_2)$.
- Let $[\texttt{y := x * x}]^{\ell}$, $\widehat{\sigma}_1(x) = 1$ and $\widehat{\sigma}_2(x) = -1$.
- Joining before transfer:

$$(\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)(x) = 1 \sqcup -1 = \top$$

- Therefore,

$$f_\ell^{CP}(\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)(y) = \top \ .$$

- Postponing the join of arguments:

$$f_\ell^{CP}(\widehat{\sigma}_1)(y) \sqcup f_\ell^{CP}(\widehat{\sigma}_2)(y) = 1 \sqcup 1 = 1$$

- Indeed, $\top \not\sqsubseteq 1$ so CP is not distributive.

Roadmap

- Monotone frameworks have been defined and illustrated.
- But how to compute an analysis result for a monotone framework?
- Algorithm MFP computes the least fixpoint.
- We want to know how precise the result can be.
- What is the best possible solution we may ever obtain?
  - This is the Meet Over all Paths (MOP) solution.
- MFP is a sound approximation of MOP: MOP $\sqsubseteq$ MFP.
- For distributive frameworks, however, MOP $=$ MFP.

An Algorithm for Monotone Frameworks

The Meet/Merge Over all Paths (MOP) solution

- A complete execution is a path through the control-flow graph F from initial to (some) final label.
- What is an execution?
  - A path from the initial label to any label in the program
- Consider for a particular label ℓ:
  $$\text{path}_\circ(\ell) = \{[\ell_1, \ldots, \ell_{n-1}] \mid n \geq 1, \forall i < n : (\ell_i, \ell_{i+1}) \in F, \ell = \ell_n, \ell_1 \in E\}$$
- The analysis function for one such path, $p = [\ell_1, \ldots, \ell_m]$:
  $$f_p = f_{\ell_m} \circ \ldots \circ f_{\ell_1} \circ id$$
- Applying the function to the extremal value ι gives the analysis result for p.
- Be consistent with all possible executions leading to ℓ:
  $$\text{MOP}_\circ(\ell) = \bigsqcup\{f_p(\iota) \mid p \in \text{path}_\circ(\ell)\}$$

And similarly...

- For paths ending after the transfer function for block ℓ:
  $$\text{path}_\bullet(\ell) = \{[\ell_1, \ldots, \ell_n] \mid n \geq 1, \\ \forall i < n : (\ell_i, \ell_{i+1}) \in F, \ell = \ell_n, \ell_1 \in E\}$$
- The join over these paths is then
  $$\text{MOP}_\bullet(\ell) = \bigsqcup \{f_p(\iota) \mid p \in \text{path}_\bullet(\ell)\}$$

MOP is undecidable

- Without proof.
- Intuition: joining over an infinite number of execution paths: when do you stop?
- For some analyses, MOP is decidable.

Maximal Fixed Point (MFP) - input/output

- Computes the least fixed point of an instance of a monotone framework
- Input: the monotone framework $(L, \mathcal{F}, F, E, \iota, \lambda\ell.f_\ell)$. where
  - L the complete lattice
  - $\mathcal{F}$ the monotone function space containing all the transfer functions
  - F the transitions of the program
  - E the extremal labels
  - ι the extremal value, and finally
  - $\lambda\ell.f_\ell$ the mapping from labels ℓ to transfer functions from $\mathcal{F}$.
- Output: the values $\text{MFP}_\circ(\ell)$ and $\text{MFP}_\bullet(\ell)$ for all $\ell \in \mathbf{Lab}_*$

General idea of MFP

- Work list algorithm: intermediate worklist W.
- An array A that approximates the solution from below $A[\ell] \sqsubseteq MFP_\circ(\ell)$.
- We initialize A to something great, and repeat until consistent with the constraints.
- Array A stores increasingly closer approximations of the answer.
  - Only the context values are stored.
  - If transfer functions expensive to compute, then cache/store also the effect values.

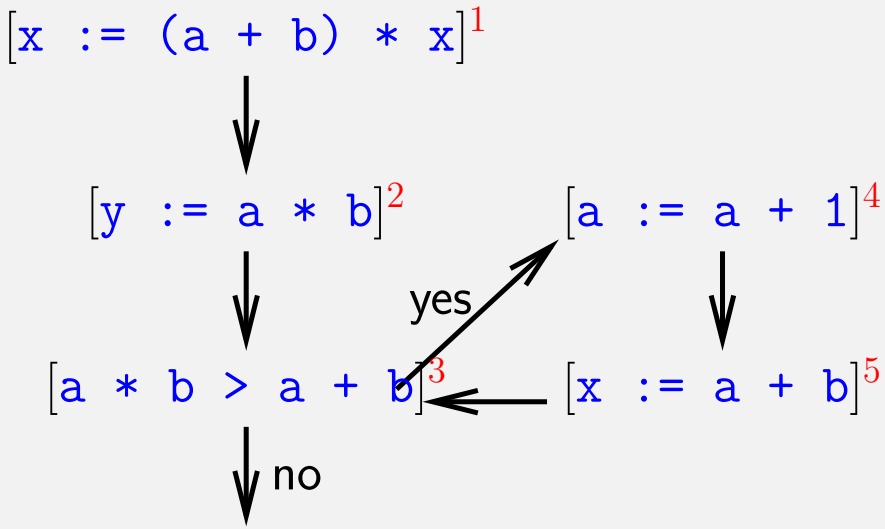
The code of the algorithm

- Step 1 (initialization):
  Set $A[\ell] = \bot$ for $\ell \notin E$,
  set $A[\ell] = \iota$ for $\ell \in E$, and set $W = F$.
- Step 2 (iteration):

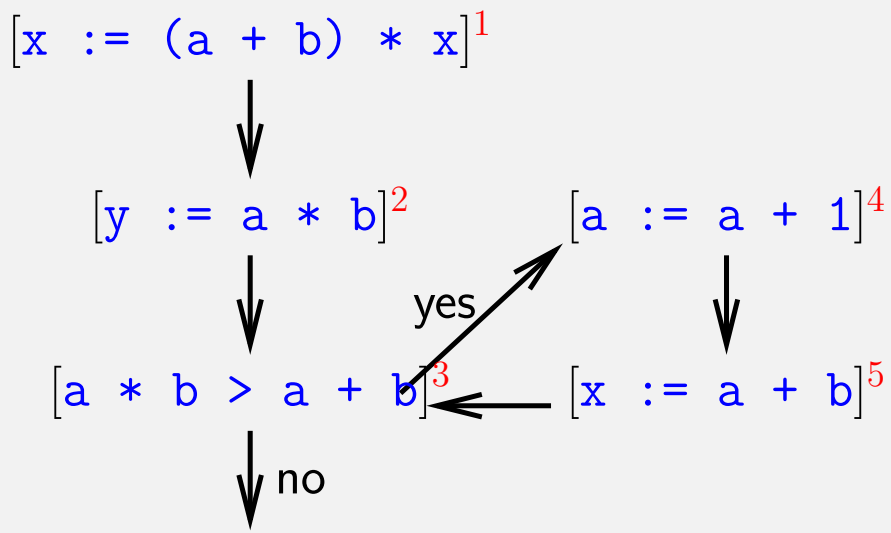
```
while W not empty do
    (ℓ, ℓ') := head(W);                    -- get next edge
    W := tail(W);                  -- drop it from the list
    if f_ℓ(A[ℓ]) ⋢ A[ℓ'] then     -- if not consistent
        A[ℓ'] := A[ℓ'] ⊔ f_ℓ(A[ℓ]);    -- incorporate it
        for all ℓ'' with (ℓ', ℓ'') ∈ F do    -- add all
            W := (ℓ', ℓ''): W;          -- successors to W
```

- Step 3 (finalization):
  Copy $A[\ell]$ into $\mathtt{MFP}_\circ(\ell)$ and $f_\ell(A[\ell])$ into $\mathtt{MFP}_\bullet(\ell)$.

How does it work?



How does it work?



- At some point: $(\ell, \ell') = (5, 3)$ is next up, $A[3] = \{a + b, a * b\}$ and $A[5] = \emptyset$
- Compute $x = f_5(A[5]) = (\emptyset - \{(a + b) * x\}) \cup \{a + b\}$.
- Do the test: is x a superset of $A[3]$?
- No, so set $A[3] = A[3] \sqcup x = A[3] \cap \{a + b\} = \{a + b\}$.
- Add $(3, 4)$ to W: propagate changes.

Part 1 of correctness: invariants

- Similar to correctness of fixpoint iteration.
- Let $\text{Analysis}_\circ(\ell)$ and $\text{Analysis}_\bullet(\ell)$ describe the least solution to the equations.
- To prove: $A \sqsubseteq \text{Analysis}_\circ$ and $A \sqsubseteq F(A)$ are invariants of the while loop.
- The base case: at initialization
  - $\bot \sqsubseteq \text{Analysis}_\circ(\ell), F(\bot)$ for $\ell \notin E$, and
  - $\iota \sqsubseteq \text{Analysis}_\circ(\ell), F(\iota)$ for $\ell \in E$
- The inductive case: consider the flow edge (ℓ, ℓ')
  - If we do not change A, then nothing is changed except W.
  - If we do, then monotonicity saves the day.
- In summary, A stays below (or is on) the least fixpoint.

Part 2 of correctness: at termination

- Previous slide implies: we never "pass by" the intended solution.
- But do we have a solution when the algorithm terminates?
- Two important aspects here:
  - We consider every equation at least once.
    - Because W is initialized to F
  - When a value is updated, we make sure all equations that may be directly influenced are added to the worklist.
- Together implies that at termination we are in a reductive point: $F(A) \sqsubseteq A$.
  - Negate the if-condition in the algorithm.

MFP computes the least fixed point

- Part 1 and 2 together say that $A = F(A)$: it is a fixpoint.
- Since this fixpoint lies below or on the least fixpoint (part 1), it must be that least fixpoint.
- Similar if you consider the effect values.

Termination

- Everytime we add an edge to W it is because a value changed.
- Because of ACC, every $A[\ell]$ can only change a finite number of times.
- This gives termination.

Complexity of the algorithm

- Let L have finite height $h \geq 1$ (length of longest chain).
- Let e be the number of edges in F ($e \geq$ number of labels).
- Step 2 of the algorithm is in $\mathcal{O}(e \cdot h)$
- Reason: every edge can only lead to a change at most h times (after a change). In each case, we do/generate a "constant" amount of work.
- Evaluating f_ℓ, $\sqcup$, updating A are considered basic operations. Running time is measured in terms of how many of these basic operations have to be done.

MFP approximates MOP

- MFP always terminates, MOP is generally undecidable.
- Obviously, MFP is not always MOP, but MOP $\sqsubseteq$ MFP.
  - MOP can be more precise than what MFP computes.
- We saw this earlier for Constant Propagation: joining before transfer loses detail.
- This is where MFP loses precision over MOP.
- Can this be reconciled with the fact that MFP computes the least solution?
- For distributive frameworks: joining before or after makes no difference.
  - Not surprisingly, MFP $=$ MOP

Summary so far

- General idea of program analysis
- Three example analyses: AE, LV, CP
- Monotone frameworks
- Algorithms for computing a solution for an instance of a monotone framework.
- Properties of such a solution

5. Interprocedural Analysis

Procedural programming

- Any sensible programming language supports procedures or functions in some form.
- The main complications that will arise are:
  - How do we propagate analysis information into and out of procedures?
  - A procedure can be jumped to from arbitrarily many locations.
    - Do we join the results over all possible callers?
  - How do we "know" where to return?
    - What if we blindly propagate a single analysis result to all return locations?
- We focus on forward analysis.

Adding procedures to While

- Extend the `While`-language with procedures
- A program takes the form: `begin` D_* S_* `end`
- D_* is a sequence of procedure declarations:

  `proc p(val x, res y) is`$^{\ell_n}$ `S end`$^{\ell_x}$
- `x` and `y` are formal parameters and local to `p`
- A procedure call is a statement: $[\texttt{call p(a,z)}]^{\ell_c}_{\ell_r}$
- `a` is passed by-value and can be any arithmetic expression
- `z` is call-by-result: it can only be used to pass the result back

Information about programs

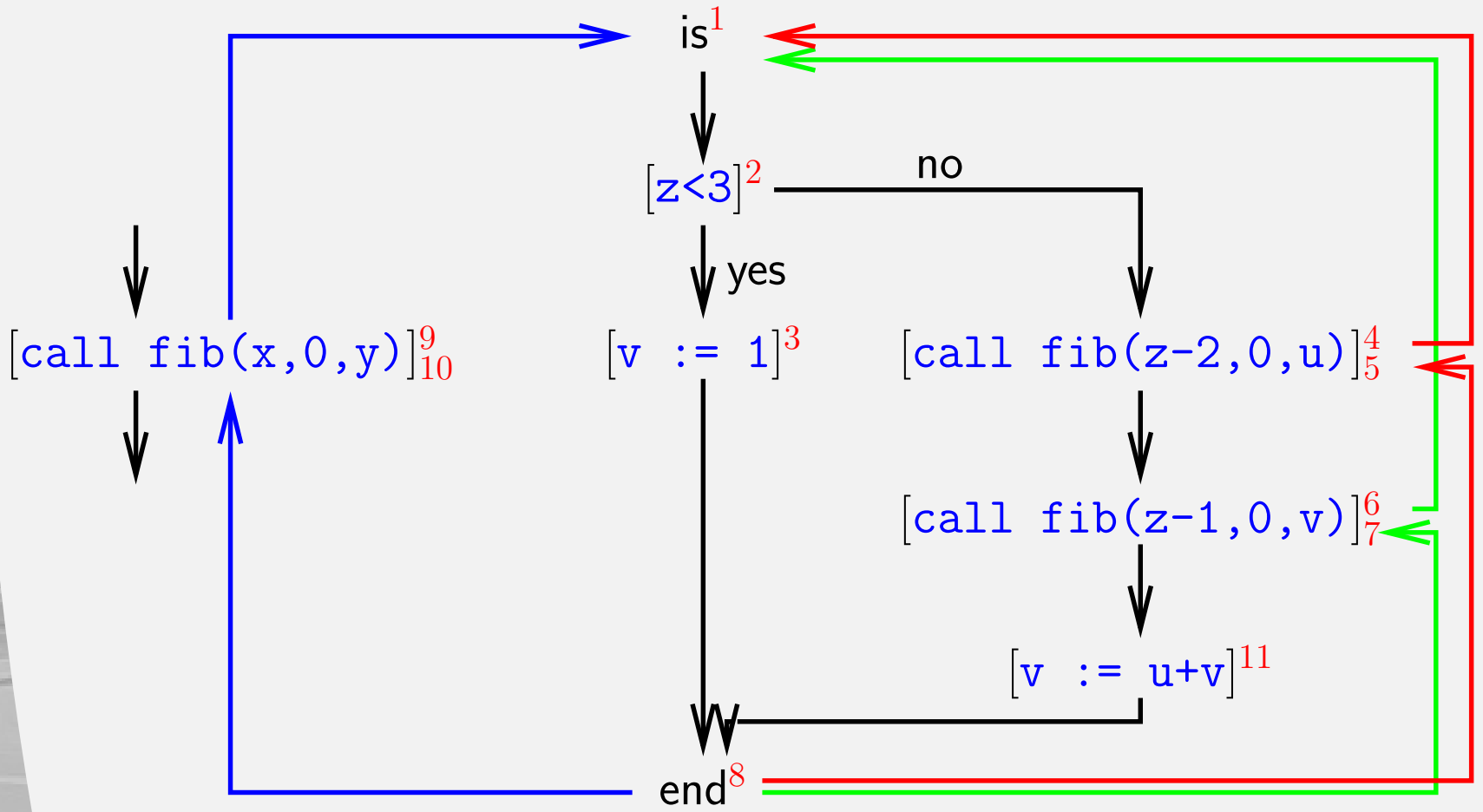
- New block types: `is`, `end` and `call (...)`
- Entry and exit labels attached to `is` and `end`
- Call and return labels attached to `call`
- Add new kind of flow:
  - $(\ell_c; \ell_n)$ for procedure call/entry
  - $(\ell_x; \ell_r)$ for procedure exit/return
- Assume all programs are statically correct:
  - only calls to existing procedures,
  - all labels and procedure names unique.

An example program


```
begin proc fib(val z, u, res v) is[1]
        if [z<3][2] then [v := 1][3]
        else ([call fib(z-2,0,u)][4,5];
              [call fib(z-1,0,v)][6,7];
              [v := v+u][11])
     end[8];
     [call fib(x,0,y)][9,10]
end
```

- Syntax more restrictive than examples imply.
- Mimicking local variables: add by-value parameters (like `u`)
- Variables `x` and `y` have global scope
- The scope of `u`, `v`, `z` is limited to the body of `fib`.

The flow graph



Meet over all valid paths: MVP

- Generalize the utopian $\text{MOP}_\circ$ and $\text{MOP}_\bullet$ solutions to the more precise $\text{MVP}_\circ$ and $\text{MVP}_\bullet$.
  - Later we consider how to adapt monotone frameworks.
- Paths up to ℓ:
  $\text{vpath}_\circ(\ell) =$
  $$\{[\ell_1, \ldots, \ell_{n-1}] \mid n \geq 1, \ell_n = \ell, [\ell_1, \ldots, \ell_n] \text{ a valid path}\}$$
- $\text{MVP}_\circ(\ell) = \bigsqcup\{f_{\overrightarrow{\ell}}(\iota) \mid \overrightarrow{\ell} \in \text{vpath}_\circ(\ell)\}$
- Similarly for the closed case, $\text{MVP}_\bullet(\ell)$.
- But what is a valid path?

Unbalance and poisoning


```
begin proc neg(val z, res u) is[1]
        [u := -z][2])
      end[3];
      [call neg(-1,p)][5][6];
      [call neg(1,n)][7][8]
end
```

- Suppose we treat $(5;1)$ like $(5,1)$?
- Suppose we want to track the signs of all variables.
- Poisoning: information about the first call to `neg` also flows to the second call. Reasonable?
- $\text{path}_\circ$ and $\text{path}_\bullet$ do not always pair call labels correctly with the label of the return.
- Valid paths, on the other hand, are balanced.
- $[5,1,2,3,8]$ is not valid, but $[5,1,2,3,6]$ is.

Use valid paths and context instead §5

- Issues when defining valid paths
  - Consider only balanced executions.
  - During analysis we only consider finite prefixes of these,
  - including finite prefixes of infinite ones.

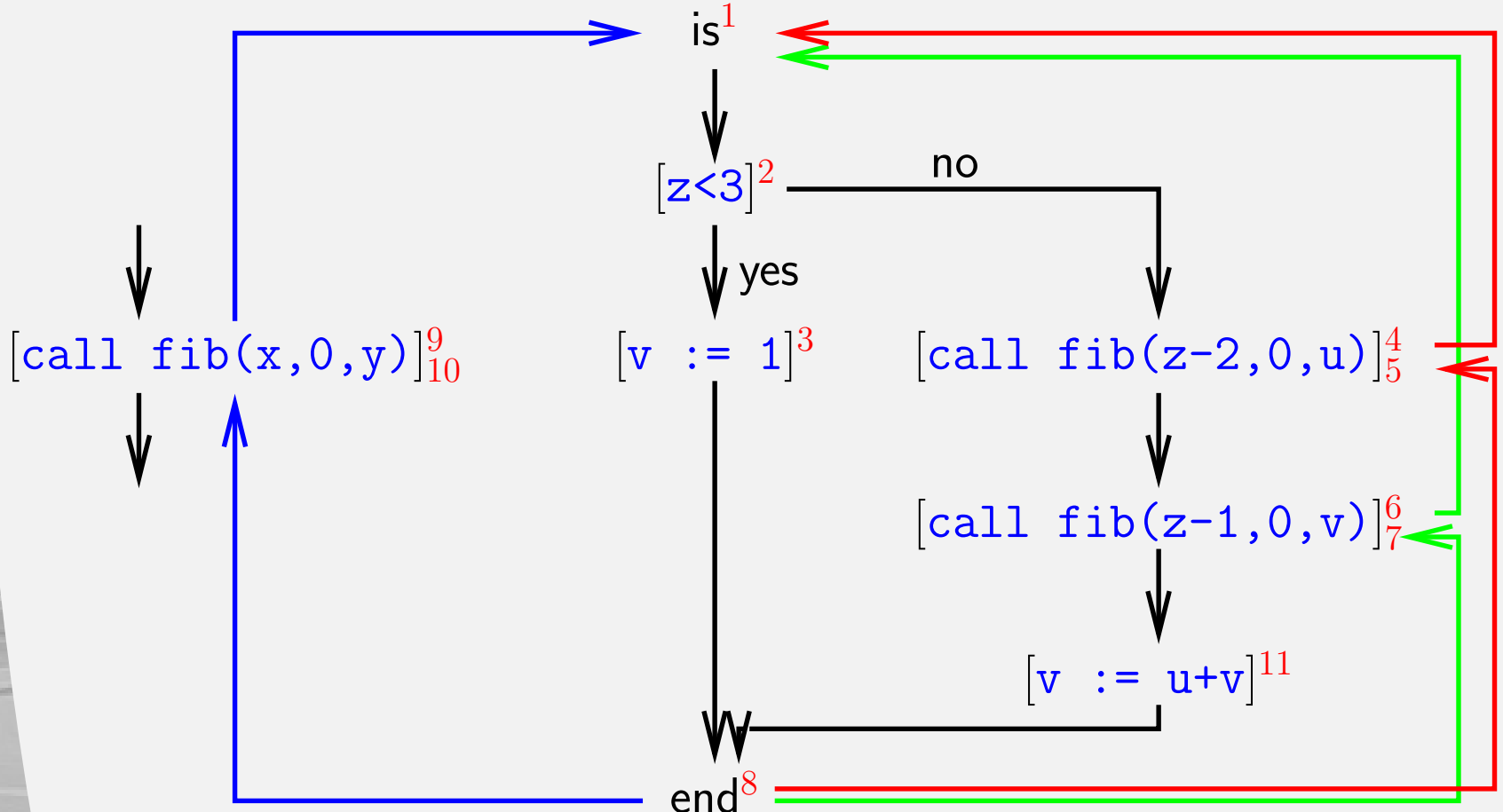
```
begin proc infinite(val n, res x) is[2]
        [call infinite(0,x)][3][4];
    end[5];
    [call infinite(0,x)][1][6]
end
```

- Context can be used to enforce balance:
  - it can simulate/abstract behaviour of a call stack.
- The amount of context determines complexity and precision.

Interprocedural flows

- The previous slides motivate a need to distinguish interprocedural and intraprocedural flow.
- For the fibonacci program:
  $\texttt{flow}(S_*) = \{(1,2),(2,3),(3,8),(2,4),(4;1),(8;5),(5,6),$
  $(6;1),(8;7),(7,11),(11,8),(9;1),(8;10)\}$
- Interprocedural:
  $\texttt{inter-flow}(S_*) = \{(\mathbf{9},\mathbf{1},\mathbf{8},\mathbf{10}),(\mathbf{4},\mathbf{1},\mathbf{8},\mathbf{5}),(\mathbf{6},\mathbf{1},\mathbf{8},\mathbf{7})\}$
  4-tuples of call and corresponding return information.
- $(9,1,8,5) \notin \texttt{inter-flow}(S_*)$
- $\texttt{init}(S_*) = 9$ and $\texttt{final}(S_*) = \{10\}$
- Backward variants exist: $\texttt{flow}^R$ and $\texttt{inter-flow}^R$

The flow graph again



Intermediate summary

- Changes to the programming language have now been made.
  - syntax,
  - scoping rules,
  - MOP is generalized to MVP
- Now come the changes to the monotone framework
  - reuse as much as possible of intraprocedural monotone framework,
  - transfer functions for the new statements,
  - distinguish between certain execution paths via context.

Interprocedural

Towards interprocedural analyses

- Introduce interprocedural flow
  - Extension to the Framework
  - A 4-tuple with labels of caller and callee: $(\ell_c, \ell_n, \ell_x, \ell_r)$
  - Binary transfer functions, taking values of caller and callee.
- Make analysis more precise
  - Embellished Monotone Framework
  - Context aware: track call stacks
  - Lifts a normal instance to an embellished instance
  - No changes to framework needed!

Interprocedural flow

- Keep track of function calls as flow $(\ell_c, \ell_n, \ell_x, \ell_r) \in \texttt{inter-flow}(S_*)$.
  - ℓ_c: Entry label of function call in caller
  - ℓ_n: Entry label of the callee
  - ℓ_x: Exit label of the callee
  - ℓ_r: Exit label of function call in caller
- Could be treated as flow $(\ell_c; \ell_n)$ and $(\ell_x; \ell_r)$.
- Two transfer functions: f_{ℓ_c} and $f^2_{\ell_c,\ell_r}$.

Interprocedural flow

- Keep track of function calls as flow $(\ell_c, \ell_n, \ell_x, \ell_r) \in \texttt{inter-flow}(S_*)$.
  - ℓ_c: Entry label of function call in caller
  - ℓ_n: Entry label of the callee
  - ℓ_x: Exit label of the callee
  - ℓ_r: Exit label of function call in caller
- Could be treated as flow $(\ell_c; \ell_n)$ and $(\ell_x; \ell_r)$.
- Two transfer functions: f_{ℓ_c} and $f^2_{\ell_c,\ell_r}$.
- Should local variables of the caller be passed to the callee?

Interprocedural flow

- Keep track of function calls as flow $(\ell_c, \ell_n, \ell_x, \ell_r) \in \texttt{inter-flow}(S_*)$.
  - ℓ_c: Entry label of function call in caller
  - ℓ_n: Entry label of the callee
  - ℓ_x: Exit label of the callee
  - ℓ_r: Exit label of function call in caller
- Could be treated as flow $(\ell_c; \ell_n)$ and $(\ell_x; \ell_r)$.
- Two transfer functions: f_{ℓ_c} and $f^2_{\ell_c,\ell_r}$.
- Should local variables of the caller be passed to the callee?
  - f_{ℓ_c} can remove those from the analysis value
  - $f^2_{\ell_c,\ell_r}$ should add them back

What happens at procedure return?

- Procedure return encompasses the real difference:

$$A_\bullet(\ell_r) = f^2_{\ell_c,\ell_r}(A_\circ(\ell_c), A_\circ(\ell_r))$$

- Transfers information from inside the procedure and from before the call to just after the call.
- Note: $A_\circ(\ell_r)$ is (normally) just $A_\bullet(\ell_x)$.
- $f^2_{\ell_c,\ell_r}$ may ignore one (or both) arguments.
- For a backward analysis, the transfer functions change arity: the one for call becomes binary, the one for return becomes unary.

What happens at procedure return?

- Information before a call can be passed directly to after the call.
  - Instead of propagating it through the call.
- Worklist: tracks information flow of analysis, not control flow!
  - Don't forget the edge (ℓ_c, ℓ_r).

Embellished Monotone Frameworks

Towards embellished monotone frameworks

- From monotone framework to embellished monotone framework.
- We proceed by example.
  - Define a monotone framework for Detection Of Signs Analysis.
  - Specify the form of transfer functions for calls, entries, exits and returns.
  - Change it to include context so that data flows along balanced paths,
  - by lifting the original transfer functions so that they include context,
  - and making sure that procedure call and return imply a context change.
- Context can be "anything", but we choose contexts that help us to analyze along valid paths.

Detection of Sign Analysis

- Let $(L, \mathcal{F}, F, E, \iota, \lambda\ell.f_\ell)$ be an instance of a monotone framework for Detection of Sign Analysis (Exercise 2.15)
- Detection of Signs gives for each program point what signs each variable may have at that program point.
- Beware: my notation differs from that in NNH.

Detection of Sign Analysis - the lattice

- The complete lattice L consists of sets of functions
- More precisely: elements of $\mathcal{P}(\mathbf{Var}_* \to S)$ with $S = \{-, 0, +\}$
- Each function describes a set of executions leading to a certain program point.
- Example: $\{g, h\} \in L$ with $g(x) = g(y) = +$, and $h(x) = +$ and $h(y) = -$
- In other words, there might be
  - executions where x and y are both positive, and
  - executions where x is positive and y is negative.

Detection of Sign Analysis example

- Assume $\mathbf{Var}_* = \{x, y\}$,
  $g(x) = +$ and $g(y) = +$, and
  $h(x) = +$ and $h(y) = -$.
- Consider the effect of $[\texttt{x := x+y}]^\ell$ on g:
  - the function g' which maps both x and y to $+$ (so $g = g'$)
- The effect of $[\texttt{x := x+y}]^\ell$ on h is
  - map y to $-$, but x to $-, 0$ or $+$
  - the result is described by three functions, h_-, h_0 and h_+, defined as $h_i(y) = -$ and $h_i(x) = i$ (for all i).
- The set $\{g, h\}$ is thus mapped to $\{g', h_-, h_0, h_+\}$.

Relational vs. independent

- Recall: the set $\{g, h\}$ was mapped to $\{g, h_-, h_0, h_+\}$.
  - g tells us y can be mapped to $+$, the h_i that y maps to $-$.
  - The h_i tell us that x can map to any one of the $\{0, -, +\}$.
- Analysis is relational: we store combinations of x and y.
- To save on resources, merge the functions to a set of signs for each variable: x has signs $\{0, -, +\}$ and y has $\{+, -\}$
  - Thereby becoming an independent attributes analysis.
- This value also represents the previously known to be impossible $[x \mapsto -, y \mapsto +]$ and $[x \mapsto 0, y \mapsto +]$.
- The independent attribute analysis is really weaker,
  - but also less resource consuming.

Interpreting expressions

- $\mathcal{A}_s$: **AExp** $\to$ (**Var**$_*$ $\to S$) $\to \mathcal{P}(S)$ gives all possible signs of an expression, when given a sign for each variable.
- $\mathcal{A}_s[\![\texttt{x+y}]\!][x \mapsto +, y \mapsto +] = \{+\}$
- $\mathcal{A}_s[\![\texttt{x+y}]\!][x \mapsto +, y \mapsto -] = \{0, +, -\}$

Transfer functions

- Transfer function for $[\texttt{x := a}]^\ell$ maps sets of functions to sets of functions:

$$f_\ell(Y) = \bigcup \{\phi_\ell(\sigma) \mid \sigma \in Y\}$$

  where $Y \in L$ and $\phi_\ell(\sigma) = \{\sigma[x \mapsto s] \mid s \in \mathcal{A}_s[\![\texttt{a}]\!](\sigma)\}$

- Functions may "split up":

$$\phi_\ell([x \mapsto +, y \mapsto -]) =$$

$$\{[x \mapsto -, y \mapsto -], [x \mapsto 0, y \mapsto -], [x \mapsto +, y \mapsto -]\}$$

- Finally $f_\ell(Y)$ collects everything:

$$\{[x \mapsto +, y \mapsto +], [x \mapsto -, y \mapsto -],$$

$$[x \mapsto 0, y \mapsto -], [x \mapsto +, y \mapsto -]\}$$

Adding context to the lattice

- Add context to get an embellished monotone framework $(\widehat{L}, \widehat{\mathcal{F}}, F, E, \widehat{\iota}, \lambda\ell.\widehat{f_\ell})$
- The complete lattice L becomes $\Delta \to L$:

  $$\mathcal{P}(\mathbf{Var}_* \to S) \text{ becomes } \Delta \to \mathcal{P}(\mathbf{Var}_* \to S)$$

- "Omit" context by taking Δ a one element set.
- For each $\delta \in \Delta$ we may have a different value in L.
  - δ serves as an index.
- $\widehat{L}$ is a complete lattice (page 398 of NNH) .
- In the book they use $\mathcal{P}(\Delta \times (\mathbf{Var}_* \to S)) \cong \Delta \to \mathcal{P}(\mathbf{Var}_* \to S)$. We don't.

Lifting the transfer functions

- We have a transfer function $f_\ell : L \to L$.
- Lift pointwise to $\widehat{f_\ell} : (\Delta \to L) \to (\Delta \to L)$:

$$\widehat{f_\ell}(\widehat{l}) = \lambda\delta \to f_\ell(\widehat{l}(\delta)) \text{ for } \widehat{l} \in \widehat{L}$$

- Or simply, $\widehat{f_\ell}(\widehat{l}) = f_\ell \circ \widehat{l}$
- In words, apply old transfer function independently, i.e., pointwise, for each value in Δ.
- Example:
  $\widehat{f_\ell}([\delta_1 \mapsto \{g\}, \delta_2 \mapsto \{h, g\}]) =$
  $[\delta_1 \mapsto f_\ell(\{g\}), \delta_2 \mapsto f_\ell(\{g, h\})] =$
  $[\delta_1 \mapsto \{g\}, \delta_2 \mapsto \{h_0, h_-, h_+, g\}].$

Data flow in the new set-up

- Information flows along dataflow graph edges:

$$A_\circ(\ell) = \bigsqcup\{A_\bullet(\ell') \mid (\ell', \ell) \in F \vee (\ell'; \ell) \in F\} \sqcup \widehat{\iota^\ell_E}$$

- So for procedure entry labels, we take the join over all callers.
- How do we tell different calls apart? By using context.
- Transfer almost as usual:

$$A_\bullet(\ell) = \widehat{f_\ell}(A_\circ(\ell))$$

- Call and return are somewhat different.

What happens for a (forward) procedure call?

- Assume a call to procedure p:
  $(\ell_c, \ell_n, \ell_x, \ell_r) \in \texttt{inter-flow}(S_*)$.
- Two transfer functions: f_{ℓ_c} and f_{ℓ_n}.
- f_{ℓ_n} is the same for every call to p.
  - In NNH always identity function.
- f_{ℓ_c} can be different for each call to p.
- "Chronologically":
  - transfer value at call $A_\bullet(\ell_c) = f_{\ell_c}(A_\circ(\ell_c))$
  - compute $A_\circ(\ell_n)$ by joining $A_\bullet$ for all calls to p.
  - transfer value at entry: $A_\bullet(\ell_n) = f_{\ell_n}(A_\circ(\ell_n))$
    - Often the identity function
  - value ready to flow through p.
- f_{ℓ_c} is typically a function that knows about context.

What happens at procedure return?

- Procedure return encompasses the real difference:

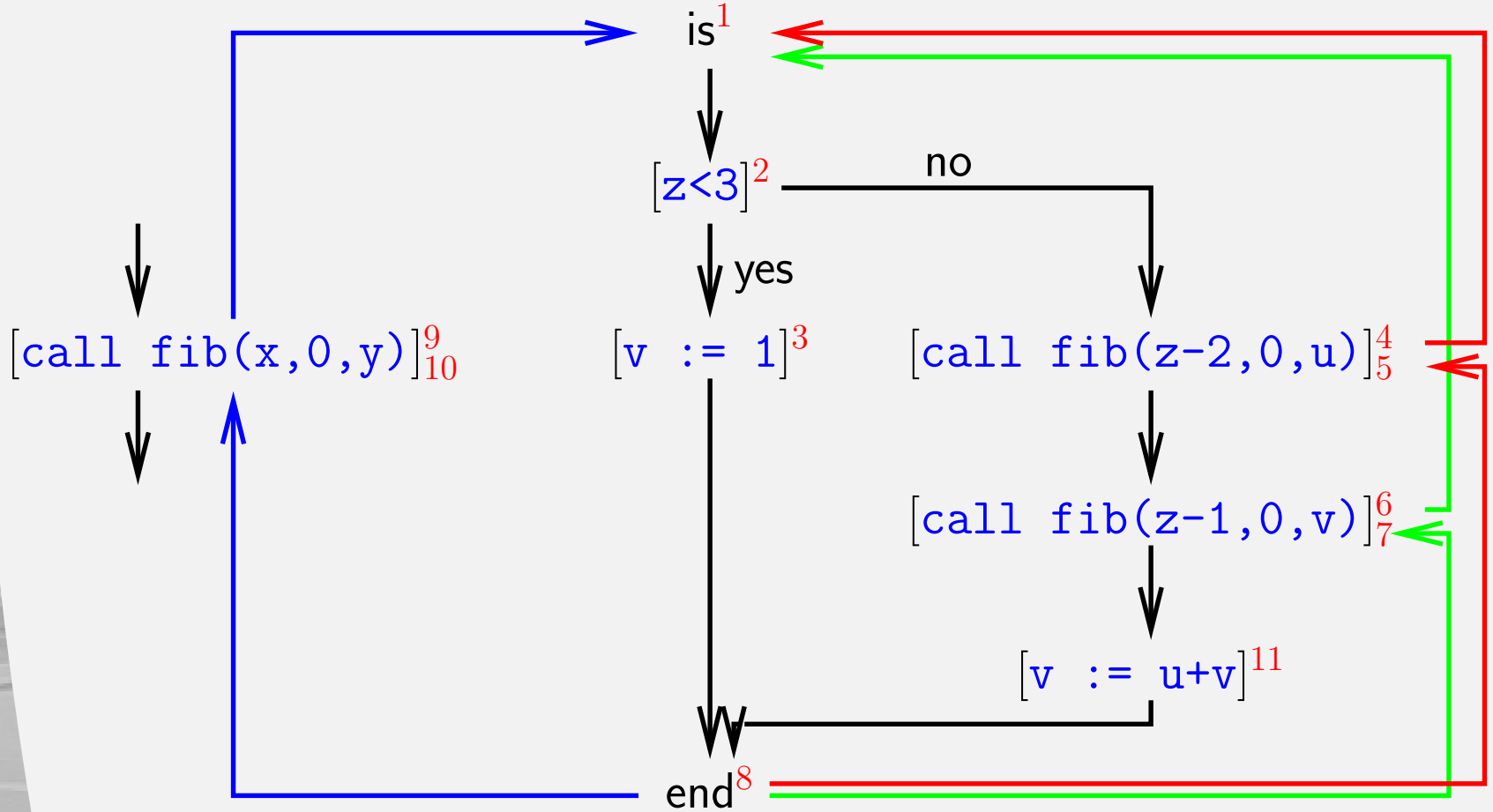
$$A_\bullet(\ell_r) = \widehat{f^2_{\ell_c,\ell_r}}(A_\circ(\ell_c), A_\circ(\ell_r))$$

- Transfers information from inside the procedure and from before the call to just after the call.
- Note: $A_\circ(\ell_r)$ is (normally) just $A_\bullet(\ell_x)$.
- Information before a call can be passed directly to after the call.
  - Instead of propagating it through the call.
- $\widehat{f^2_{\ell_c,\ell_r}}$ may ignore one (or both) arguments.
- For a backward analysis, the transfer functions change arity: the one for call becomes binary, the one for return becomes unary.

Call strings as context

- Context intends to keep analyses of separate calls separated
- Call string: list of addresses from which a call was made.
  - Abstraction of the call stack: $\Delta = [\mathbf{Lab}_*]$
- For `fib`: Λ, $[4], [6], [9], [4,4], \ldots, [9,9], [4,4,4], \ldots$
  - Generate only when needed.
- Call-string abstracts an execution into the labels of calls seen during execution without seeing the corresponding return: $[1,6,5,8,3,2,1,4,2,1,9]$ becomes $[6,9]$
- Procedure call labels are added to the front (stack like).

The flow graph again



Call strings as context

- Call string: list of addresses from which a call was made.
- For $(\ell_c, \ell_n, \ell_x, \ell_r)$ we define

  $$\widehat{f^1_{\ell_c}}(\hat{l})(\ell_c{:}\delta) = f^1_{\ell_c}(\hat{l}(\delta)) \qquad \text{and} \qquad \widehat{f^1_{\ell_c}}(\hat{l})(\Lambda) = \bot$$

- $f^1_\cdot$ computes the effect of a call
- and $\widehat{f^1_\cdot}$ selects where the effect values should go.
- Valid paths simulated by the transferring between "corresponding" call strings.

Call strings as context, return

- Similarly, for procedure return:
  $$\widehat{f^2_{\ell_c,\ell_r}}(\widehat{l},\widehat{l'})(\delta) = f^2_{\ell_c,\ell_r}(\widehat{l}(\delta),\widehat{l'}(\ell_c:\delta))$$
- We use two values:
  - from before the call, which is under the same context as the return,
  - from inside the procedure, which is under the extended call string.

Detection of Signs: procedure calls

- Assume $[\texttt{call p(a,x)}]^{\ell_c}_{\ell_r}$ and
  $\texttt{proc p(val x, res y) is}^{\ell_n}\ \texttt{S end}^{\ell_x}$
- A call consists of two assignments `x := a` and `y := ?`.
  - The context-less transfer function mimicks those.
- For $\sigma = [x \mapsto +, z \mapsto -]$ and $\texttt{a} = \texttt{-x}$ we ought to obtain
  $$\begin{aligned}\phi_{\ell_c}(\sigma) = \{&[x \mapsto -, y \mapsto -, z \mapsto -],\\ &[x \mapsto -, y \mapsto 0, z \mapsto -],\\ &[x \mapsto -, y \mapsto +, z \mapsto -]\}\end{aligned}$$
- Semantics says value of y is undefined (instead of 0).
- New x "shadows" the old.
- In general, unshadow when returning.

Detection of Signs: procedure calls

- Assume $[\texttt{call p(a,x)}]^{\ell_c}_{\ell_r}$ and
  $\texttt{proc p(val x, res y) is}^{\ell_n}\ \texttt{S end}^{\ell_x}$
- For $\sigma = [x \mapsto +, z \mapsto -]$ and $\texttt{a} = \texttt{-x}$ we ought to obtain
  $\phi_{\ell_c}(\sigma) = \{[x \mapsto -, y \mapsto -, z \mapsto -],$
  $[x \mapsto -, y \mapsto 0, z \mapsto -],$
  $[x \mapsto -, y \mapsto +, z \mapsto -]\}$
- $f_{\ell_c}(Z) = \bigcup\{\phi_{\ell_c}(\sigma) \mid \sigma \in Z\}$
- $\phi_{\ell_c}(\sigma) =$
  $\{\sigma[x \mapsto s][y \mapsto s'] \mid s \in \mathcal{A}_s[\![\texttt{-x}]\!](\sigma) \wedge s' \in \{0, +, -\}\}$

Detection of Signs: adding context

- Consider the function $Z \in \widehat{L} = \Delta \to L$

$$Z = [\Lambda \mapsto \sigma_1, \delta_2 \mapsto \sigma_2, \ldots]$$

- We want to obtain

$$[\Lambda \mapsto \bot, [\ell_c] \mapsto f^1_{\ell_c}(\sigma_1), (\ell_c{:}\delta_2) \mapsto f^1_{\ell_c}(\sigma_2), \ldots]$$

- So $\widehat{f^1_{\ell_c}}(Z)$ is such that for all $\delta \in \Delta$

$$\widehat{f^1_{\ell_c}}(Z)(\delta') = \begin{cases} \bot & \text{if } \delta' = \Lambda \\ f^1_{\ell_c}(Z(\delta)) & \text{if } \delta' = \ell_c{:}\delta \end{cases}$$

- Warning: in NNH they give the same general formula, but the example of Detection of Signs (2.38) uses different notation.

Call strings of bounded size

- L might have ACC, but $\Delta \to L$ might not
  - Call strings can be arbitrarily long for recursive programs
- Enforce termination by restricting length call strings to $\leq k$
- For every different list of call labels, potentially a different analysis result: quickly exponential.

What if we run out of bounds?

- Assume $k = 2$.
- Consider call from 4 either with context $[1, 4]$, $[1, 1]$ or $[1]$.
- Then in all three cases, the context inside the call will be $[4, 1]$.
- To stay sound we must join the transferred analysis results.
- Here's where we gain finiteness at the price of precision.
- In a formula

$$\widehat{f^4_{\ell_c}}(Z)([4,1]) = f^4_{\ell_c}(Z([1,4])) \sqcup f^4_{\ell_c}(Z([1,1])) \sqcup f^4_{\ell_c}(Z([1]))$$

- We can choose the level of detail (value of k) with a known price to pay.
  - Take $k = 0$ to omit context: Δ then equals $\{\Lambda\}$

Separate the context from the transfer

- Context is never used to compute the transfer, it only tells you which part of the value to use (and update).
- For different analyses you can use the same kind of context and context change
- In an implementation: decouple the context change from transfer
  - The former selects which values influence a given value.
  - The latter says how.

Flow-sensitive versus flow-insensitive

- Flow-sensitive vs. flow-insensitive: does the result of the analysis depend on the order of statements? Again a matter of cost vs. precision.
- To go from flow-insensitive to flow-sensitive: add program points as a form of context.
- In NNH, flow-sensitivity is hard-coded into the framework.

Final remarks about procedures

- Except for binary transfer functions, the technical changes are slight.
- Conceptually, changes may be bigger.
- For termination, restrict context to finite sets of values.
- Use context to balance cost and precision.
- Simple monotone frameworks can be easily extended to become embellished.
  - A first step in building an analysis.
- Analyzing procedures can be a pain when scoping enters the picture.
- **You can now completely do the first lab assignment**