APA
Dataflow analysis

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Department of Information and Computing Sciences, Universiteit Utrecht

April 15, 2013

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

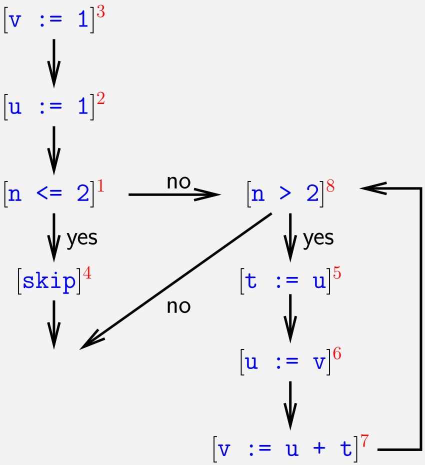
1. Preliminaries

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: $[\texttt{skip}]^2$, $[\texttt{(x <= 2)}]^3$, $[\texttt{x := x + 1}]^{31}$

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


- ```
  [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);
  ```
- **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.
  ```

2. Intraprocedural Analysis

Available Expression Analysis


- ```
  [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.
- For each program point, which (non-trivial) expressions *must* already have been computed, and not later modified, on all paths to the program point.
- Each a subset of **AExp**$_*$ $= \{a + b, (a + b) * x, a * b, a + 1\}$
- Associated optimization: values of available expression may be cached for use at $[\mathtt{B}]^\ell$.
- To exploit this, *all* paths to $[\mathtt{B}]^\ell$ must 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)
  ```
- $AE_{\mathrm{N}}(1) = \emptyset$
  - 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)
  ```
- $AE_{\mathrm{X}}(3) = AE_{\mathrm{N}}(3) \cup \{a + b, a * b\}$
  - 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:
  - what expressions become available $\mathit{gen}_{AE}(B^\ell)$
  - what expressions become unavailable $\mathit{kill}_{AE}(B^\ell)$
- 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

- What to remove for assignments:
  $\mathit{kill}_{AE}([\texttt{x := a}]^\ell) = \{a' \in \mathbf{AExp}_* \mid x \in FV(a')\}$
- What to add for assignments:
  $\mathit{gen}_{AE}([\texttt{x := a}]^\ell) = \{a' \in \mathbf{AExp}(a) \mid x \notin FV(a')\}$
- Why $x \notin FV(a')$?
- Example:

```
[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:
  - $\mathit{kill}_{AE}([\texttt{skip}]^{\ell}) = \emptyset$
  - $\mathit{gen}_{AE}([\texttt{skip}]^{\ell}) = \emptyset$
- For conditions:
  - $\mathit{kill}_{AE}([\mathsf{b}]^{\ell}) = \emptyset$
  - $\mathit{gen}_{AE}([\mathsf{b}]^{\ell}) = \textbf{AExp}(b)$
- We only save arithmetic expressions, not complete boolean ones.
  - Higher precision lead to higher costs.

Analysis functions for Available Expressions §2

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:

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

- Flow functions do not work for programs starting with a loop. Why?
- 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 $\overrightarrow{\mathsf{AE}}$:

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

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

- 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 $\overrightarrow{\mathrm{AE}}$ 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, $\{a+b, a*b\} \sqsubseteq \{a+b\}$
- 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 $F(\overrightarrow{\text{AE}}) \sqsubseteq F(\overrightarrow{\text{AE'}})$
- 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 $F_{\text{entry}}(3)$, are monotone.
- With AE, $\sqsubseteq$ is in fact $\supseteq$
- For $F_{\text{entry}}(3)$:

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

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 `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

- $[\texttt{x := 2}]^1$; $[\texttt{y := 4}]^2$; $[\texttt{x := 1}]^3$;
  (if $[\texttt{B}]^4$ then $[\texttt{z := y}]^5$
  else $[\texttt{z := x*x}]^6$);
  $[\texttt{x := z}]^7$;
- Variable $\texttt{x}$ 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.
- Assignments to dead variables is dead code and might be removed
- In contrast with AE, LV is a backward analysis

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 initial set of variables of interest.

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

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

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 §2

- 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 out sets, by using $\cup$ (may).

3. 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.

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 $[B]^\ell$
  - $\text{Analysis}_\bullet(\ell)$ or effect value: it shows the effect of $[B]^\ell$ 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 $\mathsf{Analysis}_\circ(\ell)$

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

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

where

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

- 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:
  - $\bigsqcup = \bigcap$
  - $F = \{(1,2),(2,3),(3,4),(4,5),(5,3)\}$
  - $E = \{1\}$
  - $\iota = \emptyset$
  - $\bot = \textbf{AExp}_*$ (because $x \cap \textbf{AExp}_* = x$)
- Transfer functions f_ℓ will have to wait a bit.
  ```

Interlude

First, we consider the datatypes for Analysis$_\circ$ and Analysis$_\bullet$: complete lattices satisfying the Ascending Chain Condition.

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 these statements at a later time.

Example


- ```
  while [n < 10]^1 do
      if [n >= 5]^2
      then [n := 2*n]^3
      else [n := n + 1]^4;
  ```
- Sign analysis: For each variable compute the signs it may have at/before each program point $(-, +, 0)$.
- For simplicity, we consider only the variable `n`.
- Example constraints that influence analysis result $A[1]$:
  $\{0\} \subseteq A[1]$,
  $A[3] \subseteq A[1]$,
  $A[4] \subseteq A[1]$.
  ```

A more operational view

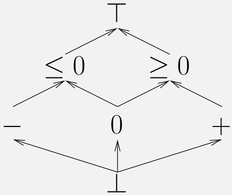
- Constraints: $\{0\} \subseteq A[1]$, $A[3] \subseteq A[1]$, $A[4] \subseteq A[1]$.
- Alternate view: $A[1]$ is a function f_1 of $A[3]$ and $A[4]$. When they change, $A[1]$ may also need an update.
- In this case, $f_1(A) = A[3] \cup A[4] \cup \{0\}$.
- A system of constraints leads to a function F that maps A to a new, updated A, hopefully closer to the solution.
- Iterate until a fixed point $F(A) = A$ is reached.
- F must be monotone: larger inputs do not lead to smaller outputs.
- When can we be sure it stops, and is the answer any good?

Essential ingredients: joins and termination

- During program analysis:
  - we need need to "join" information from various execution paths.
    - The condition of a while can be reached from at least two places.
  - We can typically identify a best possible and a worst possible value.
- Lattices encapsulate what we need.
- Iteration should terminate in finite number of iterations.
- Guaranteed if function is monotone and lattice satisfies Ascending Chain Condition.
- At termination, we have the best possible (least) fixed point.
  - In the example, smallest possible sets of signs

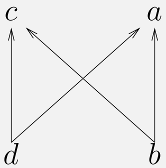
Introductory example

- Take a set of values, say $\{\bot, -, 0, +, \leq 0, \geq 0, \top\}$.
  - 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 everyhing
  - ≤ 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 $x = y$.
- Examples:
  - "(type t') is an instance of (type t)" is a partial order
  - $\leq$ and $\geq$ are partial orders on the natural numbers **N**, and so is $=$.
- Partial order P conventionally drawn as a Hasse diagram:



Lattices

- If for all $x, y \in L$, it holds that there exists a smallest $z \in L$ with $x \sqsubseteq z$ and $y \sqsubseteq z$, then the partial order is called a lattice (tralie in Dutch).
- If z exists, then it is unique and denoted $x \sqcup y$ (the join of x and y).
- Similarly for the greatest lower bound $x \sqcap y$, the meet of x and y.
- Reason: we want $\sqcup$ and $\sqcap$ to be total binary functions, i.e., operators.
- Duality: reversing all edges in the lattice gives another lattice.

Example lattices

- $(\mathbf{N}, =)$ is not a lattice: $x \sqcup y$ is undefined for all $x \neq y$.
- $\mathcal{T}$ is a lattice, because of specially added error type:
  - $\mathsf{Int} \ \sqcup \ \mathsf{Int} \to \mathsf{Int} = \top$.
- $(\mathbf{N}, \leq)$ and $(\mathbf{N}, \geq)$ are (dual) lattices.
- The partial order P is not a lattice.

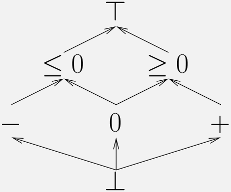
$$\begin{tikzcd} c & a \\ d \arrow[u] \arrow[ur] & b \arrow[u] \arrow[ul] \end{tikzcd}$$

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, $\bigsqcup X$ is defined and unique for all $X \subseteq L$.
- $\bigsqcup \emptyset = \bot$ and $\bigsqcup L = \top$.
- Is every finite lattice complete?
- No, complete lattices must have a bottom and top element.
- But a finite lattice with a bottom is complete.

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.
- 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`
  $\quad X = F(X)$;
  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 induction.
- 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`
  $\quad X = F(X);$
  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.

Interlude

End of interlude.

Property spaces: the data type of the analysis §3

- 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 $\perp$ can be used to initialize the computation
  - Intuitively, $\perp$ represents *most informative* element of L
- Greatest element $\top$ (usually) means *no useful* or *inconsistent information*

Examples LV

- Live Variables (for program S_*):
  - $L = \mathcal{P}(\textbf{Var}_*)$, finite sets of variables,
  - for $x, y \in L$: $x \sqsubseteq y$ if and only if $x \subseteq y$,
  - $\sqcup = \cup$,
  - $\bot = \emptyset$ and $\top = \textbf{Var}_*$.
- Why not $L = \mathcal{P}(\textbf{Var})$ 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 $\textbf{Var}_*$ 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 §3

- 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 §3

- 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 $f_{.}$ from the labels **Lab**$_*$ to functions in $\mathcal{F}$

Example Available Expressions continued

- $[\texttt{x := (a + b) * x}]^1$;
  $[\texttt{y := a * b}]^2$;
  $\texttt{while}\ [\texttt{a * b > a + b}]^3\ \texttt{do}$
  $\quad([\texttt{a := a + 1}]^4$;
  $\quad\ [\texttt{x := a + b}]^5)$
- $(L, \sqsubseteq) = (\mathcal{P}(\mathbf{AExp}_*), \supseteq)$ 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
- $f_\ell(l) = (l - \mathit{kill}_{AE}([\mathsf{B}]^\ell)) \cup \mathit{gen}_{AE}([\mathsf{B}]^\ell)$ where $[\mathsf{B}]^\ell \in \texttt{blocks}(S_*)$

Available Expressions is a Monotone Framework §3

- 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

- Consider analysis info ℓ_1 and ℓ_2 for two executions leading up to a block
- Two ways to proceed:
  - join before transfer: $f(\ell_1 \sqcup \ell_2)$ (MFP)
  - join after transfers: $f(\ell_1) \sqcup f(\ell_2)$ (MOP)
- By monotonicity $f(\ell_1) \sqcup f(\ell_2) \sqsubseteq f(\ell_1 \sqcup \ell_2)$
  - So the second possibility is never worse than the first
- If f is distributive then both ways are equivalent:

  $$f(\ell_1 \sqcup \ell_2) \sqsubseteq f(\ell_1) \sqcup f(\ell_2).$$

  - In distributive frameworks doing a join before the transfer does not lose information
- Verify that AE is distributive: $f(l \cap l') = f(l) \cap f(l')$
- Distributivity is good: faster algorithms, higher precision.
- Not all monotone frameworks are distributive.

4. Constant propagation

Constant Propagation

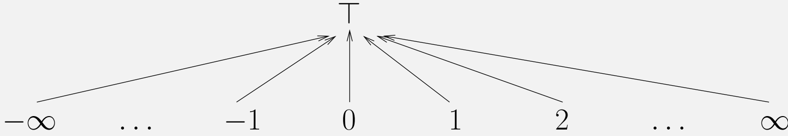
- Constant Propagation: Determine at each program point and for each variable whether the variable always has the same value there.
- We are not interested to see which variables never change
  - Although we shall find that out too
- For every variable we either know
  - the single integer value it can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

Some examples

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

The Constant Propagation lattice

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



- The property space L is the complete lattice of total functions from $\mathbf{Var}_*$ to $\mathbf{Z}^{\top}$.
- Our total functions can be interpreted as finite sets of pairs $\mathbf{Var}_* \times \mathbf{Z}^{\top}$ where every variable occurs exactly once.
- 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

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

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: Analysis$_\circ(4) = [y \mapsto 2, z \mapsto 1, x \mapsto \top]$
- Later: Analysis$_\circ(4) = [y \mapsto 2, z \mapsto \top, x \mapsto \top]$, because $z \mapsto 1$ in Analysis$_\bullet(3)$ and $z \mapsto 2$ in 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 $\mathcal{F}_{CP}$

Constant Propagation is not distributive

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

5. Solving a monotone framework

The Meet/Merge Over all Paths (MOP) solution §5

- 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:
  $\mathsf{path}_\circ(\ell) = \{[\ell_1, \ldots, \ell_{n-1}] \mid n \geq 1, \forall i < n : (\ell_i, \ell_{i+1}) \in F, \ell = \ell_n, \ell_1 \in E\}$
- 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:
  $$\mathsf{MOP}_\circ(\ell) = \bigsqcup \{f_p(\iota) \mid p \in \mathsf{path}_\circ(\ell)\}$$

And similarly...

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

$$\mathsf{MOP}_\bullet(\ell) = \bigsqcup\{f_p(\iota) \mid p \in \mathsf{path}_\bullet(\ell)\}$$

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 $\mathsf{MFP}_\circ(\ell)$ and $\mathsf{MFP}_\bullet(\ell)$ 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 $\mathsf{A}[\ell] \sqsubseteq \mathsf{MFP}_\circ(\ell)$.
- 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.

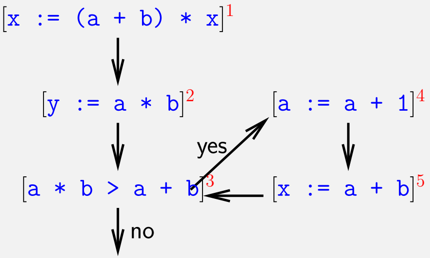
The code of the algorithm

- 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` $\mathrm{MFP}_\circ(\ell)$ `and` $f_\ell(A[\ell])$ `into` $\mathrm{MFP}_\bullet(\ell)$.

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: an invariant

- 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: $A \sqsubseteq \text{Analysis}_\circ$ is an invariant of the while loop.
- The base case: at initialization
  - $\perp \sqsubseteq \text{Analysis}_\circ(\ell)$ for $\ell \notin E$, and
  - $\iota \sqsubseteq \text{Analysis}_\circ(\ell)$ 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 $\neq$ 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
- Two example analyses
- Monotone frameworks
- Algorithms for computing a solution for an instance of a monotone framework.
- Properties of such a solution