DATA FLOW ANALYSIS

[ HOW TO ANALYZE LANGUAGES AUTOMATICALLY ]

Claus Brabrand
((( brabrand@itu.dk )))

Associate Professor, Ph.D.
((( Programming, Logic, and Semantics )))

IT University of Copenhagen
Abstract

"Data-Flow Analysis":

In this 3*3 hour mini course we will look at data-flow analysis. Rather than just look at the classical "monotone framework" analyses (which are usually synonymous with teaching data-flow analysis: reaching definitions, live variables, available expressions, and very busy expressions), we will instead take one step backwards and look at the general theory and practice behind these analyses. The idea is that you will then learn how to design your own customized data-flow analyses for automatically analyzing whatever aspects of programming languages you want to. (From this perspective, the monotone framework analyses are just special cases.)

Keywords:
- undecidability, approximation, control-flow graphs, partial-orders, lattices, transfer functions, monotonicity, [how to solve] fixed-point equations – and how all of these things combine to enable you to design data-flow analyses.
DATA-FLOW ANALYSIS

- Undecidable problems: halting, reduction principle
- Undecidability (Rice's Theorem)
- Static analysis (approximations)
- Control-flow graphs (influence)
- Programming languages (syntax & semantics)

Mathematics:
- Cross product
- Relations
- Partial orders
- Lattices
- Monotone functions
- Fixed points (R. Fixed-point Theorem)

Computer Science:

Data Flow Analysis

Make analyses
- Faster programs!
- Find errors automatically!
- Appreciate & understand compilers & languages :-)

Practice!
Agenda

- Introduction:
  - Undecidability, Reduction, and Approximation

- Data-flow Analysis:
  - Quick tour of everything & running example

- Control-Flow Graphs:
  - Control-flow, data-flow, and confluence

- "Science-Fiction Math": (next monday)
  - Lattice theory, monotonicity, and fixed-points

- Putting it all together…: (next wednesday)
  - Example revisited
Notes on Static Analysis

"Lecture Notes on Static Analysis"
by Michael I. Schwartzbach
(Aarhus University)

Chapter 1, 2, 4, 5, 6 (until p. 19)
(Excl. "pointers")

Claims to be "not overly formal", but the math involved can be quite challenging (at times)…
Quiz: Optimization?

If you want a fast C-program, should you use:

- LOOP 1:
  ```c
  for (i = 0; i < N; i++) {
    a[i] = a[i] * 2000;
    a[i] = a[i] / 10000;
  }
  ```

- LOOP 2 (optimized by programmer):
  ```c
  b = a;
  for (i = 0; i < N; i++) {
    *b = *b * 2000;
    *b = *b / 10000;
    b++;
  }
  ```

i.e., “array-version” or “optimized pointer-version”?
Results (of running the programs):

<table>
<thead>
<tr>
<th>LOOP</th>
<th>opt. level</th>
<th>SPARC</th>
<th>MIPS</th>
<th>Alpha</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1 (array)</td>
<td>no opt</td>
<td>20.5</td>
<td>21.6</td>
<td>7.85</td>
</tr>
<tr>
<td>#2 (ptr)</td>
<td>no opt</td>
<td>19.5</td>
<td>17.6</td>
<td>7.55</td>
</tr>
</tbody>
</table>

- Compilers use highly sophisticated static analyses for optimization! (you'll learn how to do this!!!)
- Recommendation: focus on writing clear code for people (and compilers) to understand!
Data-Flow Analysis

**Purpose** (of Data-Flow Analysis):
- **Gather information** (on running behavior of program)
  - "∀ program points"

**Usage** (of static analysis):
- Basis for subsequent…:
  - Error Detection
  - Optimization

Static Analysis ➔ **information** ➔ Error Detection
∀ program points ➔ Optimization
Analyses for Error Detection

Example Analyses:

- "Symbol Checking": Catch (dynamic) symbol errors
- "Type Checking": Catch (dynamic) type errors
- "Initialized Variable Analysis": Catch uninitialized variables

...
Analyses for Optimization

Example Analyses:

"Constant Propagation Analysis":
- Precompute constants (e.g., replace ‘5*x+z’ by ‘42’)

"Live Variables Analysis":
- Dead-code elimination (e.g., get rid of unused variable ‘z’)

"Available Expressions Analysis":
- Avoid recomputing already computed exprs (cache results)

...
Conceptual Motivation

- Undecidability
- Reduction principle
- Approximation
Rice’s Theorem (1953)

“Any interesting problem about the runtime behavior of a program* is undecidable”

-- Rice’s Theorem [paraphrased] (1953)

*) written in a turing-complete language

Examples:

- does program ’P’ always halt when run?
- is the value of integer variable ’x’ always positive?
- does variable ’x’ always have the same value?
- which variables can pointer ’p’ point to?
- does expression ’E’ always evaluate to true?
- what are the possible outputs of program ’P’?
- …
Undecidability (self-referentiality)

Consider "The Book-of-all-Books":
- This book contains the titles of all books that do not have a self-reference (i.e. don't contain their title inside)
- Finitely many books; i.e.:
  - We can sit down & figure out whether to include or not...
- Q: What about "The Book-of-all-Books";
  - Should it be included or not?

"Self-referential paradox" (many guises):
- e.g. "This sentence is false"
Termination Undecidable!

- **Assume** termination is *decidable* (in Java);
  - i.e. $\exists$ some program, \texttt{halts}: \texttt{Program} $\rightarrow$ \texttt{bool}

```java
bool halts(Program p) { ... }
```

- Q: Does $P_0$ loop or terminate...? :)  
- **Hence**: \texttt{halts} cannot exist!
  - i.e., "Termination is undecidable" *) for turing-complete languages
Rice’s Theorem (1953)

“Any interesting problem about the runtime behavior program* is undecidable”

-- Rice’s Theorem [paraphrased] (1953)

*) written in a turing-complete language

Examples:

- does program ’P’ always halt?
- is the value of integer variable ’x’ always positive?
- does variable ’x’ always have the same value?
- which variables can pointer ’p’ point to?
- does expression ’E’ always evaluate to true?
- what are the possible outputs of program ’P’?
- …
Reduction: $\vdash \text{solve } \textit{always-pos} \implies \text{solve } \textit{halts}$

1) Assume '$x\text{-is-always-pos}(P)$' is decidable
2) Given $P$ (here’s how we could solve '$\text{halts}(P)$'):
3) Construct (veeeeery clever) reduction program $R$:
   
   ```java
   -- R.java --
   int x = 1;
   $P$ /* insert program $P$ here :-) */
   x = -1;
   ```
4) Run ”supposedly decidable” analysis:
   
   $$\text{res} = \textit{x-is-always-positive}(R)$$
5) Deduce from result:
   
   if (res) then $P$ loops!; else $P$ halts  
   6) \textbf{THUS: } '$x\text{-is-always-pos}(P)$’ must be \textit{undecidable}!
**Reduction Principle**

- **Reduction principle (in short):**

\[
\phi(P) \text{ undecidable } \land \left[ \text{solve } \psi(P) \Rightarrow \text{solve } \phi(P) \right] \\
\psi(P) \text{ undecidable}
\]

- **Example:**

\[
'\text{halts}(P)' \text{ undecidable } \land \left[ \text{solve } 'x-is-always-pos(P)' \Rightarrow \text{solve } '\text{halts}(P)' \right] \\
'x-is-always-pos(P)' \text{ undecidable}
\]

- **Exercise:**
  - Carry out reduction + whole explanation for:
    - "which variables can pointer 'q' point to?"
Answer

1) Assume ’which-var-q-points-to(P)’ is decidable:
2) Given P (here’s how to (cleverly) decide halts(P)):
3) Construct (veeeeery clever) reduction program R:

```
ptr q = 0xffffff;
P /* insert program P (assume w/o 'q') */
q = null;
```

4) Run ’which-var-q-points-to(R)’ = res
5) If (null \not\in res) P halts! else; P loops! :-)
6) THUS:

’which-var-q-points-to(P)’ must be undecidable!
Undecidability

Undecidability means that…:

…no-one can decide this line (for all programs)!

However(!)…
“Side-Stepping Undecidability”

However, just because it’s undecidable, doesn’t mean there aren’t (good) approximations! Indeed, the whole area of static analysis works on “side-stepping undecidability”:

- Compilers use safe approximations (computed via ”static analyses”) such that:

  - Okay!
  - Dunno?
  - Error!
“Side-Stepping Undecidability”

However, just because it’s undecidable, doesn’t mean there aren’t (good) approximations! Indeed, the whole area of static analysis works on “side-stepping undecidability”:

- **Unsafe approximation:**
  
  ![Unsafe approximation diagram]

- For **testing** it may be okay to ”abandon” safety and use **unsafe approximations**:

  ![Unsafe approximation diagram]

Here are some programs for you to (manually) consider!
“Slack”

Undecidability means: “there’ll always be a slack”:

However, still useful:

(possible interpretations of “Dunno?”):

- Treat as error (i.e., reject program):
  - “Sorry, program not accepted!”

- Treat as warning (i.e., warn programmer):
  - “Here are some potential problems: …”
Soundness & Completeness

- **Soundness:**
  - Analysis reports no errors ⇒ Really are no errors

- **Completeness:**
  - Analysis reports an error ⇒ Really is an error

---

...or alternative (equivalent) formulation, via "contra-position":

\[
P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P
\]

- Really are error(s) ⇒ Analysis reports error(s)
- Really no error(s) ⇒ Analysis reports no error(s)
Example: Type Checking

Will this program have type error (when run)?

```java
void f() {
    var b;
    if (<EXP>) {
        b = 42;
    } else {
        b = true;
    }
    ...
    if (b) ...; // error is b is '42'
}
```

Undecidable (because of reduction):
- Type error ⇔ <EXP> evaluates to true
Example: Type Checking

Hence, languages use **static requirements**:

```c
void f() {
    bool b; // instead of "var b;"
    if (<EXP>) {
        b = 42;
    } else {
        b = true;
    }
}
```

- All variables must be **declared**
- And have **only one type** (throughout the program)
- This is (very) easy to check (i.e., "type-checking")

Static compiler error: Regardless of what `<EXP>` evaluates to when run

Claus Brabrand, UFPE, Brazil

DATA-FLOW ANALYSIS
Agenda

- Introduction:
  - Undecidability, Reduction, and Approximation

- Data-flow Analysis:
  - Quick tour & running example

- Control-Flow Graphs:
  - Control-flow, data-flow, and confluence

- "Science-Fiction Math":
  - Lattice theory, monotonicity, and fixed-points

- Putting it all together…:
  - Example revisited
5’ Crash Course on Data-Flow Analysis
IDEA:

"Simulate runtime execution at compile-time using abstract values"

We (only) need 3 things:

- A control-flow graph
- A lattice
- Transfer functions

Example: "(integer) constant propagation"
Control-flow graph

Given program:

```
int x = 1;
int y = 3;
if (...) {
    x = x+2;
} else {
    x <-> y;
}
print(x,y);
```

We (only) need 3 things:
- A control-flow graph
- A lattice
- Transfer functions
Lattice $L$ of *abstract values* of interest and their relationships (i.e. ordering “$\leq$”):

```
... -3 -2 -1  0  1  2  3  ...
```

- “top” $\top$ ~ “I don’t know (could be anything)"
- “bottom” $\bot$ ~ “we haven’t analyzed yet”

Induces *least-upper-bound* operator: $\sqcup$

- for *combining information*
We (only) need 3 things:
- A control-flow graph
- A lattice
- Transfer functions

\[ \lambda E . E[x \mapsto 1] \]
\[ \text{int } x = 1; \]
\[ \lambda E . E[y \mapsto 3] \]
\[ \text{int } y = 3; \]
\[ \lambda E . E[x \mapsto E(x) \oplus 2] \]
\[ x = x + 2; \]
\[ \lambda E . E[x \mapsto E(y), y \mapsto E(x)] \]
\[ x \leftrightarrow y; \]
\[ \text{print}(x, y); \]

\[ x \ y \ \{ \bot, \bot \} \in \text{ENV}_L \]
\[ \{ 1, \bot \} \]
\[ \{ 1, 3 \} \]
\[ \{ 1, 3 \} \]
\[ \{ 1, 3 \} \]
\[ \{ 3, 1 \} \]

We need 3 things:
- A control-flow graph
- A lattice
- Transfer functions

\[ \lambda E . E[x \mapsto 1] \]
\[ \text{int } x = 1; \]
\[ \lambda E . E[y \mapsto 3] \]
\[ \text{int } y = 3; \]
\[ \lambda E . E[x \mapsto E(x) \oplus 2] \]
\[ x = x + 2; \]
\[ \lambda E . E[x \mapsto E(y), y \mapsto E(x)] \]
\[ x \leftrightarrow y; \]
\[ \text{print}(x, y); \]

\[ x \ y \ \{ \bot, \bot \} \in \text{ENV}_L \]
\[ \{ 1, \bot \} \]
\[ \{ 1, 3 \} \]
\[ \{ 1, 3 \} \]
\[ \{ 1, 3 \} \]
\[ \{ 3, 1 \} \]

\[ \lambda E . E[x \mapsto E(y), y \mapsto E(x)] \]
\[ x \leftrightarrow y; \]
\[ \text{print}(x, y); \]

\[ x \ y \ \{ \bot, \bot \} \in \text{ENV}_L \]
\[ \{ 1, \bot \} \]
\[ \{ 1, 3 \} \]
\[ \{ 1, 3 \} \]
\[ \{ 1, 3 \} \]
\[ \{ 3, 1 \} \]
Introduction:
- Undecidability, Reduction, and Approximation

Data-flow Analysis:
- Quick tour & running example

Control-Flow Graphs:
- Control-flow, data-flow, and confluence

“Science-Fiction Math”:
- Lattice theory, monotonicity, and fixed-points

Putting it all together…:
- Example revisited
Control Structures

- **Control Structures**: Statements (or Expr’s) that affect "flow of control".

- **if-else**:  
  - [syntax] \[
  \text{if ( Exp ) Stm}_1 \text{ else Stm}_2
  \]
  - [semantics] The expression must be of type boolean; if it evaluates to true, Statement-1 is executed, otherwise Statement-2 is executed.

- **if**:  
  - [syntax] \[
  \text{if ( Exp ) Stm}
  \]
  - [semantics] The expression must be of type boolean; if it evaluates to true, the given statement is executed, otherwise not.
Control Structures (cont’d)

- **while:**
  
  **[syntax]**
  
  `while ( Exp ) Stm`
  
  **[semantics]**
  
  The expression must be of type `boolean`; if it evaluates to `false`, the given statement is skipped, otherwise it is executed and afterwards the expression is evaluated again. If it is still true, the statement is executed again. This is continued until the expression evaluates to `false`.

- **for:**
  
  **[syntax]**
  
  `for (Exp₁ ; Exp₂ ; Exp₃) Stm`
  
  **[semantics]**
  
  Equivalent to:
  ```
  { Exp₁;
    while ( Exp₂ ) { Stm Exp₃; } 
  }
  ```
Control-flow graph

Given program:

```c
int x = 1;
int y = 3;
if (a>b) {
    x = x+2;
} else {
    x <-> y;
}
print(x,y);
```
Exercise: Draw a Control-Flow Graph for:

```java
public static void main ( String[] args ) {
    int mi, ma;
    if (args.length == 0)
        System.out.println("No numbers");
    else {
        mi = ma = Integer.parseInt(args[0]);
        for (int i=1; i < args.length; i++) {
            int obs = Integer.parseInt(args[i]);
            if (obs > ma)
                ma = obs;
            else
                if (mi < obs) mi = obs;
        }
        System.out.println("min=\" + mi + ",\" + 
                       "max=\" + ma);}
}
```
public static void main(String[] args) {
    int mi, ma;
    if (args.length == 0) /* 1st else */
        System.out.println("No numbers");
    else {
        mi = ma = Integer.parseInt(args[0]);
        for (int i=1; i < args.length; i++) { /* 2nd else */
            obs = Integer.parseInt(args[i]);
            if (obs > ma) /* 3rd else */
                ma = obs;
            else if (mi < obs) /* 4th else */
                mi = obs;
            i++;
        }
        System.out.println("minimum = " + mi + " ; " + "maximum = " + ma);
    }
}

int mi, ma;

System.out.println("No numbers");

int i=1;

i < args.length

int obs = Integer.parseInt(args[i]);

obs > ma

ma = obs;

mi < obs

mi = obs;

System.out.println("min=" + mi + "," + "max=" + ma);
Control Structures (cont’d²)

- **do-while:**
  ```
  do Stm while ( Exp );
  ```

- **"?:"; ”conditional expression”:**
  ```
  Exp₁ ? Exp₂ : Exp₃
  ```

- **"||"; ”lazy disjunction” (aka., ”short-cut ∨”):**
  ```
  Exp₁ || Exp₂
  ```

- **"&&"; ”lazy conjunction” (aka., ”short-cut ∧”):**
  ```
  Exp₁ && Exp₂
  ```

- **switch:**
  ```
  switch ( Exp ) { Swb* }
  ```

  ```
  case Exp : Stm* break;
  default : Stm* break;
  ```

exercise
Control Structures (cont’d³)

- **try-catch-finally** *(exceptions)*:
  
  ```
  try \( Stm_1 \) catch ( \( Exp \) ) \( Stm_2 \) finally \( Stm_3 \)
  ```

- **return / break / continue**:
  
  ```
  return ;  return \( Exp \) ;  break ;  continue ;
  ```

- **”method invocation”**:
  
  - e.g.; \( f(x) \)

- **”recursive method invocation”**:
  
  - e.g.; \( f(x) \)

- **”virtual dispatching”**:
  
  - e.g.; \( f(x) \)
Control Structures (cont’d⁴)

- "function pointers":
  - e.g.; \((\ast f \, x)\)

- "higher-order functions":
  - e.g.; \(\lambda f. \lambda x. (f \ x)\)

- "dynamic evaluation":
  - e.g.; \(\text{eval} \ (\text{some-string-which-has-been-dynamically-computed})\)

Some constructions (and thus languages) require a separate control-flow analysis for determining control-flow in order to do data-flow analysis.
Agenda

- Introduction:
  - Undecidability, Reduction, and Approximation

- Data-flow Analysis:
  - Quick tour & running example

- Control-Flow Graphs:
  - Control-flow, data-flow, and confluence

- "Science-Fiction Math":
  - Lattice theory, monotonicity, and fixed-points

- Putting it all together…:
  - Example revisited
Agenda

- Relations:
  - Crossproducts, powersets, and relations

- Lattices:
  - Partial-Orders, least-upper-bound, and lattices

- Monotone Functions:
  - Monotone Functions and Transfer Functions

- Fixed Points:
  - Fixed Points and Solving Recursive Equations

- Putting it all together…:
  - Example revisited
Crossproduct: ‘×’

- **Crossproduct** (binary operator on sets):
  - Given sets:
    - \( A = \{ 0, 1 \} \)
    - \( B = \{ \text{true}, \text{false} \} \)
  - \( A \times B = \{ (0,\text{true}), (0,\text{false}), (1,\text{true}), (1,\text{false}) \} \)
  - i.e., creates **sets of pairs**

**Exercise:**
- \( A \times A = \{ (0,0), (0,1), (1,0), (1,1) \} \)
- \( Z \times Z = \{ (0,0), (0,1), (0,1), \ldots, (1,0), (1,1), \ldots, (42,87), \ldots \} \)
- \( (A \times A) \times B = \{ ((0,0),\text{true}), ((0,1),\text{true}), \ldots, ((1,1),\text{false}) \} \)
Powersets: \( \mathcal{P}(S) \)

**Powerset** (unary operator on sets):

- Given set "\( S = \{ A, B \} \)";
- \( \mathcal{P}(S) = \{ \emptyset, \{A\}, \{B\}, \{A,B\} = S \} \)
- i.e., creates the **set of all subsets** (of the set)
- Note: \( X \subseteq S \iff X \in \mathcal{P}(S) \)

**Exercise:**

- \( \mathcal{P}(Z) = \{ \emptyset, \{0\}, \{1\}, \{2\}, \ldots, \{0,1\}, \ldots, \{13,42,87\}, \ldots Z \} \)
- \( \mathcal{P}(Z \times Z) = \{ \emptyset, \{(0,0)\}, \{(1,1)\}, \ldots, \{(0,0),(3,2),(4,9)\}, \ldots Z \times Z \} \)
- If a set \( S \) has \( |S| \) elements;
  - How many elements does \( \mathcal{P}(S) \) have? **Answer:** \( 2^{|S|} \)
  - \( \mathcal{P}(S) \) is (therefore) often written \( 2^S \)
Relations

Example: “equals” relation:

- Signature: ‘=’ $\subseteq \mathbb{Z} \times \mathbb{Z}$ … same as saying: ‘=’ $\in \mathcal{P}(\mathbb{Z} \times \mathbb{Z})$
- Relation is: equals = \{(0,0), (1,1), (2,2), (3,3), (4,4), … \}
- Written as: $2 = 2$ as a short-hand for: $(2,2) \in ‘=’$
- … and as: $2 \neq 3$ as a short-hand for: $(2,3) \notin ‘=’$

Example: “less-than” relation:

- Signature: ‘<’ $\subseteq \mathbb{Z} \times \mathbb{Z}$ … same as saying: ‘<’ $\in \mathcal{P}(\mathbb{Z} \times \mathbb{Z})$
- Relation is: less-than = \{(0,1), (0,2), (0,3), …, (1,2), (1,3), … \}
- Written as: $7 < 8$ as a short-hand for: $(7,8) \in ‘<’$
- … and as: $8 \not< 7$ as a short-hand for: $(8,7) \notin ‘<’$
Exercises

Example: “less-than-or-equal-to” relation:

- Signature: $\leq \subseteq \mathbb{Z} \times \mathbb{Z}$  
  ...same as saying: $\leq \in \mathcal{P}(\mathbb{Z} \times \mathbb{Z})$

- Relation is: $\leq = \{(0,0), (0,1), (0,2), \ldots, (1,1), (1,1), \ldots, (2,3), \ldots \}$

- Written as: $2 \leq 3$ as a short-hand for: $(2,3) \in \leq$

- … and as: $3 \not\leq 2$ as a short-hand for: $(3,2) \notin \leq$

Example: “is-congruent-modulo-3” relation:

- Signature: $\equiv_3 \subseteq \mathbb{Z} \times \mathbb{Z}$  
  ...same as saying: $\equiv_3 \in \mathcal{P}(\mathbb{Z} \times \mathbb{Z})$

- Relation is: $\equiv_3 = \{(0,0), (0,3), (0,6), \ldots, (1,1), (1,4), \ldots, (6,9), \ldots \}$

- Written as: $6 \equiv_3 9$ as a short-hand for: $(6,9) \in \equiv_3$

- … and as: $7 \not\equiv_3 8$ as a short-hand for: $(7,8) \notin \equiv_3$
Equivalence Relation

Let ‘∼’ be a binary relation over set A:

‘∼’ ⊆ A × A

∼ is an equivalence relation iff:

- Reflexive:
  \[ \forall x \in A: \ x \sim x \]

- Symmetric:
  \[ \forall x, y \in A: \ x \sim y \iff y \sim x \]

- Transitive:
  \[ \forall x, y, z \in A: \ x \sim y \land y \sim z \implies x \sim z \]
Agenda

- Relations:
  - Crossproducts, powersets, and relations

- Lattices:
  - Partial-Orders, least-upper-bound, and lattices

- Monotone Functions:
  - Monotone Functions and Transfer Functions

- Fixed Points:
  - Fixed Points and Solving Recursive Equations

- Putting it all together…:
  - Example revisited
A **Partial-Order** is a structure \((S, \sqsubseteq)\):
- \(S\) is a set
- \(\sqsubseteq\) is a *binary relation* on \(S\) (i.e., \(\sqsubseteq \subseteq S \times S\)) satisfying:
  - **Reflexivity**: \(\forall x \in S: x \sqsubseteq x\)
  - **Transitivity**: \(\forall x, y, z \in S: x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z\)
  - **Anti-Symmetry**: \(\forall x, y \in S: x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y\)
Visualization: \textit{Hasse Diagram}

Partial-Order \((S, \sqsubseteq)\): \iff Hasse Diagram:

- **Reflexive:**
  \[
  \forall x \in S: \ x \sqsubseteq x
  \]

- **Transitive:**
  \[
  \forall x, y, z \in S: \ x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z
  \]

- **Anti-Symmetric:**
  \[
  \forall x, y \in S: \ x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y
  \]

\(S = \{\bot, \div, 0, +, T\}\)

\('\sqsubseteq' = \{(\bot, \bot), (\bot, \div), (\bot, 0), (\bot, +), (\bot, T), (\div, \div), (\div, T), (0, 0), (0, T), (+, +), (+, T), (T, T)\}\)

Claus Brabrand, UFPE, Brazil
Exercise (Hasse Diagram)

Given Hasse Diagram:

Write down partial order \((B, \sqsubseteq)\):

- Set \(B = \{ \ldots \}\)
- Relation \(\sqsubseteq\):
  - Signature
  - All elements of the relation (i.e., \(\sqsubseteq = \{ \ldots \}\))
  - Give example of element in \(\sqsubseteq\) (w/ + w/o shorthand)
  - Again, but for an element not in the relation
Example Partial-Orders

- Lattice Examples (as Hasse Diagrams):

...depending on what is analysed for!
Least Upper Bound '⊔'
"Least Upper Bound"

- **Upper bound:**
  - We say that 'z' is an upper bound for set 'X'
  - ...written \( X \subseteq z \) if \( \forall x \in X: x \subseteq z \)

- **Least upper bound:**
  - We say that 'z' is the least upper bound of set 'X'
  - ...written \( z = \bigcup X \) if \( X \subseteq z \land \forall z': X \subseteq z' \Rightarrow z \subseteq z' \)
Example: Least upper bound

Analyses use \( \downarrow \) to combine information (at confluence points):

\[ x = 2; \]
\[ x = 0; \]