COS320: Compiling Techniques

Zak Kincaid

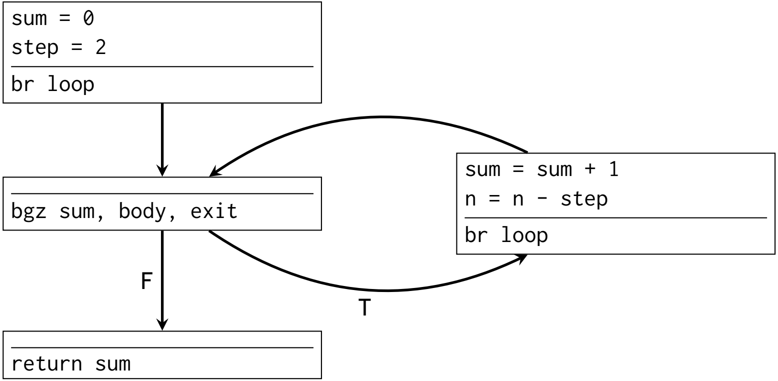
January 29, 2026

Data flow analysis

Recall: constant propagation

- A constant environment is a symbol table mapping each variable x to one of:
  - an integer n (indicating that x's value is n whenever the program is at I)
  - $\top$ (indicating that x might take more than one value at I)
  - $\bot$ (indicating that x may take no values at run-time – I is unreachable)
- An *assignment* $\mathbf{IN}$, **OUT** : $N \to$ *ConstEnv* for a CFG (N, E, s) maps each vertex to
  - $\mathbf{IN}[bb]$: a constant environment that holds immediately *before* bb
  - $\mathbf{OUT}[bb]$: a constant environment that holds immediately *after* bb
- Say that an assignment $\mathbf{IN}, \mathbf{OUT}$ is conservative if
  1. $\mathbf{IN}[s]$ assigns each variable $\top$
  2. For each node $bb \in N$,
  $$\mathbf{OUT}[bb] \sqsupseteq post(bb, \mathbf{IN}[bb])$$
  3. For each edge $src \to dst \in E$,
  $$\mathbf{IN}[dst] \sqsupseteq \mathbf{OUT}[src]$$

```
int sum2(int n) {
  int sum = 0;
  int step = 2;
  while (n > 0) {
    sum = sum + 1;
    n = n - step;
  }
  return sum;
}
```



High-level constant propagation algorithm

- Initialize $\mathbf{IN}[s]$ to the constant environment that sends every variable to $\top$ and $\mathbf{OUT}[s]$ to the constant environment that sends every variable to $\bot$.
- Initialize $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$ to the constant environment that sends every variable to $\bot$ for every other basic block

High-level constant propagation algorithm

- Initialize $\mathbf{IN}[s]$ to the constant environment that sends every variable to $\top$ and $\mathbf{OUT}[s]$ to the constant environment that sends every variable to $\bot$.
- Initialize $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$ to the constant environment that sends every variable to $\bot$ for every other basic block
- Choose a constraint that is *not* satisfied by $\mathbf{IN}, \mathbf{OUT}$
  - If there is basic block bb with $\mathbf{OUT}[bb] \not\sqsupseteq \mathit{post}(bb, \mathbf{IN}[bb])$, then set
  $$\mathbf{OUT}[bb] := \mathit{post}(bb, \mathbf{IN}[bb])$$
  - If there is an edge $\mathit{src} \to \mathit{dst} \in E$ with $\mathbf{IN}[\mathit{dst}] \not\sqsupseteq \mathbf{OUT}[\mathit{src}]$, then set
  $$\mathbf{IN}[\mathit{dst}] := \mathbf{IN}[\mathit{dst}] \sqcup \mathbf{OUT}[\mathit{src}]$$
- Terminate when all constraints are satisfied.

Some vocabulary:

- Define *pred*$(n) = \{m \in N : m \to n \in E\}$ (control flow predecessors)
- Define *succ*$(n) = \{m \in N : n \to m \in E\}$ (control flow successors)
- Path = sequence of nodes $n_1, \ldots, n_k$ such that for each i, there is an edge from $n_i \to n_{i+1} \in E$

Workset algorithm

Input : Control flow graph (N, E, s), with variables $x_1, \ldots, x_n$
Output: Least conservative assignment of constant environments

Workset algorithm

Input : Control flow graph (N, E, s), with variables $x_1, \ldots, x_n$
Output: Least conservative assignment of constant environments
$\mathbf{IN}[s] = \{x_1 \mapsto \top, \ldots, x_n \mapsto \top\}$;
$\mathbf{OUT}[s] = \{x_1 \mapsto \bot, \ldots, x_n \mapsto \bot\}$;
$\mathbf{IN}[n] = \mathbf{OUT}[n] = \{x_1 \mapsto \bot, \ldots, x_n \mapsto \bot\}$ for all other nodes n;
work $\leftarrow N$; `/* Set of nodes that may violate spec */`

Workset algorithm

Input : Control flow graph (N, E, s), with variables $x_1, \ldots, x_n$
Output: Least conservative assignment of constant environments
$\mathbf{IN}[s] = \{x_1 \mapsto \top, \ldots, x_n \mapsto \top\}$;
$\mathbf{OUT}[s] = \{x_1 \mapsto \bot, \ldots, x_n \mapsto \bot\}$;
$\mathbf{IN}[n] = \mathbf{OUT}[n] = \{x_1 \mapsto \bot, \ldots, x_n \mapsto \bot\}$ for all other nodes n;
work $\leftarrow N$; `/* Set of nodes that may violate spec */`
while *work* $\neq \emptyset$ **do**

return $\mathbf{IN}, \mathbf{OUT}$

Workset algorithm

Input : Control flow graph (N, E, s), with variables $x_1, \ldots, x_n$
Output: Least conservative assignment of constant environments

$\mathbf{IN}[s] = \{x_1 \mapsto \top, \ldots, x_n \mapsto \top\}$;
$\mathbf{OUT}[s] = \{x_1 \mapsto \bot, \ldots, x_n \mapsto \bot\}$;
$\mathbf{IN}[n] = \mathbf{OUT}[n] = \{x_1 \mapsto \bot, \ldots, x_n \mapsto \bot\}$ for all other nodes n;
work $\leftarrow N$; `/* Set of nodes that may violate spec */`
while *work* $\neq \emptyset$ **do**
    Pick some n from work;
    work $\leftarrow$ *work* $\setminus \{n\}$;

return $\mathbf{IN}, \mathbf{OUT}$

Workset algorithm

Input : Control flow graph (N, E, s), with variables $x_1, \ldots, x_n$
Output: Least conservative assignment of constant environments

```
IN[s] = {x_1 ↦ ⊤, ..., x_n ↦ ⊤};
OUT[s] = {x_1 ↦ ⊥, ..., x_n ↦ ⊥};
IN[n] = OUT[n] = {x_1 ↦ ⊥, ..., x_n ↦ ⊥} for all other nodes n;
work ← N;                                   /* Set of nodes that may violate spec */
while work ≠ ∅ do
    Pick some n from work;
    work ← work \ {n} ;

    IN[n] ← ⊔_{p ∈ pred(n)} OUT[p];
    OUT[n] ← post(n, IN[n]);

return IN, OUT
```

Workset algorithm

Input : Control flow graph (N, E, s), with variables $x_1, \ldots, x_n$
Output: Least conservative assignment of constant environments

```
IN[s] = {x_1 ↦ ⊤, ..., x_n ↦ ⊤};
OUT[s] = {x_1 ↦ ⊥, ..., x_n ↦ ⊥};
IN[n] = OUT[n] = {x_1 ↦ ⊥, ..., x_n ↦ ⊥} for all other nodes n;
work ← N;                                   /* Set of nodes that may violate spec */
while work ≠ ∅ do
    Pick some n from work;
    work ← work \ {n} ;
    old ← OUT[n];
    IN[n] ← ⊔_{p ∈ pred(n)} OUT[p];
    OUT[n] ← post(n, IN[n]);
    if old ≠ OUT[n] then
        work ← work ∪ succ(n)
return IN, OUT
```

Common subexpression elimination

- Common subexpression elimination searches for expressions that
  - appear at multiple points in a program
  - evaluate to the same value at those points

  and (possibly) save the cost of re-evaluation by storing that value.

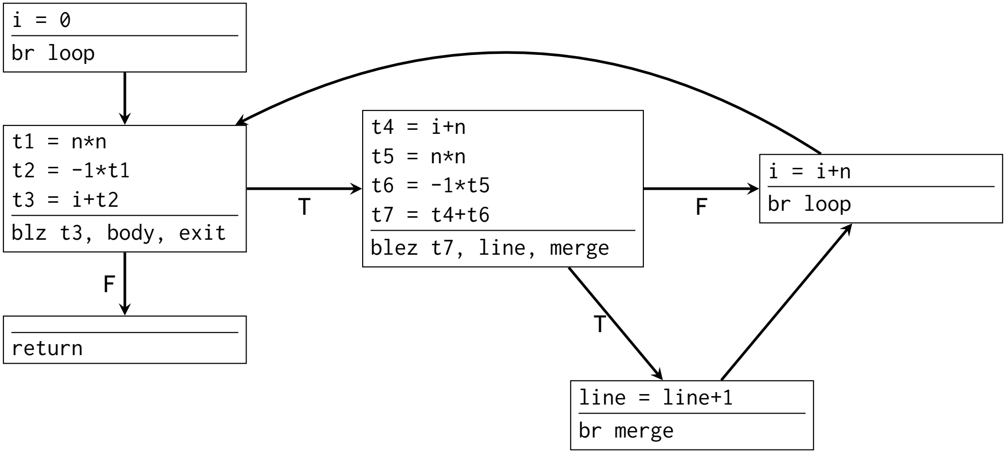
```
void print (long *m, long n) {
  long i,j;
  for (i = 0; i < n*n; i += n) {
    for (j = 0; j < n; j += 1) {
      printf('' %ld'', *(m + i + j));
    }
    if (i + n < n*n) {
      printf(''\n'');
    }
  }
}
```

→

```
void print (long *m, long n) {
  long i,j;
  long n_times_n = n*n;
  for (i = 0; i < n_times_n; ) {
    for (j = 0; j < n; j += 1) {
      printf('' %ld'', *(m + i + j));
    }
    long i_plus_n = i+n;
    if (i_plus_n < n_times_n) {
      printf(''\n'');
    }
    i = i_plus_n;
  }
}
```

Available expressions

- An *expression* in our simple imperative language has one of the following forms:
  - `add <opn> <opn>`
  - `mul <opn> <opn>`
- Fix control flow graph $G = (N, E, s)$
- An expression e is *available* at basic block $n \in N$ if for every path from s to n in G:
  1. the expression e is evaluated along the path
  2. after the *last* evaluation of e along the path, no variables in e are overwritten
- Idea: if expression e is available at node n, then we can eliminate redundant computations of e within n


i = 0
br loop
t1 = n*n
t2 = -1*t1
t3 = i+t2
blz t3, body, exit
T
t4 = i+n
t5 = n*n
t6 = -1*t5
t7 = t4+t6
blez t7, line, merge
F
i = i+n
br loop
F
return
T
line = line+1
br merge

Propagating available expressions

- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?

Propagating available expressions

- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?
  - *post*$_{AE}(x = e, E) = \{e' \in (E \cup \{e\}) : x$ not in $e'\}$

Propagating available expressions

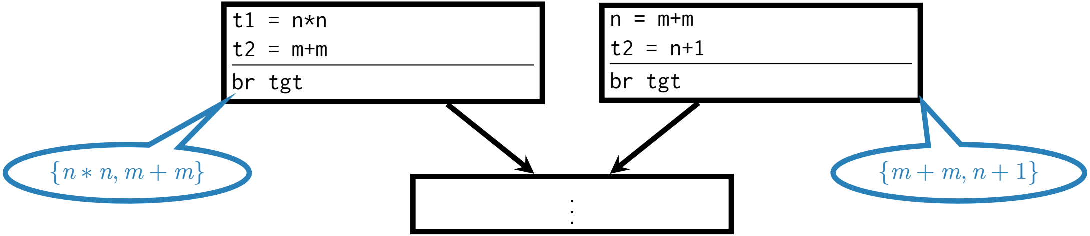
- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?
  - $\textit{post}_{AE}(x = e, E) = \{e' \in (E \cup \{e\}) : x \text{ not in } e'\}$
- How do we propagate available expressions through a basic block?

Propagating available expressions

- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?
  - ***post***$_{AE}(x = e, E) = \{e' \in (E \cup \{e\}) : x$ not in $e'\}$
- How do we propagate available expressions through a basic block?
  - Block takes the form $instr_1, \ldots, instr_n, term$.
    take ***post***$_{AE}(block, E) =$ ***post***$_{AE}(instr_n, \ldots$ ***post***$_{AE}(instr_1, E))$

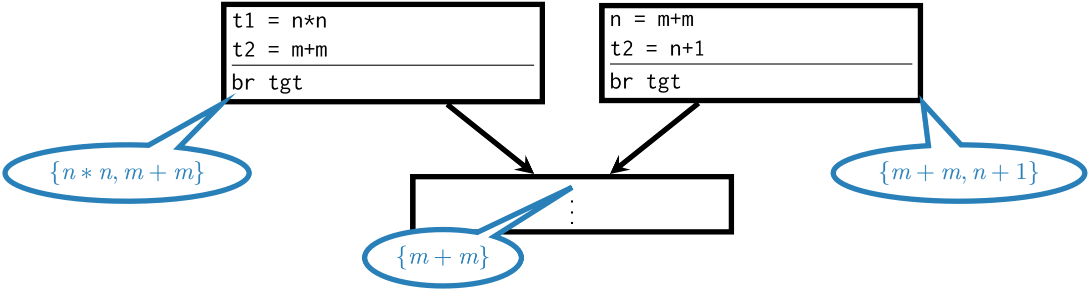
Propagating available expressions

- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?
  - *post*$_{AE}(x = e, E) = \{e' \in (E \cup \{e\}) : x$ not in $e'\}$
- How do we propagate available expressions through a basic block?
  - Block takes the form $instr_1, \ldots, instr_n, term$.
    take *post*$_{AE}(block, E) =$ *post*$_{AE}(instr_n, \ldots$ *post*$_{AE}(instr_1, E))$
- How do we combine information from multiple predecessors?



Propagating available expressions

- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?
  - *post*$_{AE}(x = e, E) = \{e' \in (E \cup \{e\}) : x$ not in $e'\}$
- How do we propagate available expressions through a basic block?
  - Block takes the form $instr_1, \ldots, instr_n, term$.
    take *post*$_{AE}(block, E) =$ *post*$_{AE}(instr_n, \ldots$ *post*$_{AE}(instr_1, E))$
- How do we combine information from multiple predecessors? *Intersection*



Available expressions as a constraint system

- Let $G = (N, E, s)$ be a control flow graph.
- For each basic block *bb* $\in N$, associate two sets of expressions, $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$
  - $\mathbf{IN}[bb]$ is the set of expressions available at the *entry* of *bb*
  - $\mathbf{OUT}[bb]$ is the set of expressions available at the *exit* of *bb*

Available expressions as a constraint system

- Let $G = (N, E, s)$ be a control flow graph.
- For each basic block *bb* $\in N$, associate two sets of expressions, $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$
  - $\mathbf{IN}[bb]$ is the set of expressions available at the *entry* of *bb*
  - $\mathbf{OUT}[bb]$ is the set of expressions available at the *exit* of *bb*
- Say that the assignment $\mathbf{IN}$, **OUT** is conservative if
  1. $\mathbf{IN}[s] = \emptyset$
  2. For each node *bb* $\in N$,
  $$\mathbf{OUT}[bb] \subseteq post_{AE}(bb, \mathbf{IN}[bb])$$
  3. For each edge *src* $\rightarrow$ *dst* $\in E$,
  $$\mathbf{IN}[dst] \subseteq \mathbf{OUT}[src]$$

Available expressions as a constraint system

- Let $G = (N, E, s)$ be a control flow graph.
- For each basic block *bb* $\in N$, associate two sets of expressions, $\mathbf{IN}[bb]$ and $\mathbf{OUT}[bb]$
  - $\mathbf{IN}[bb]$ is the set of expressions available at the *entry* of *bb*
  - $\mathbf{OUT}[bb]$ is the set of expressions available at the *exit* of *bb*
- Say that the assignment $\mathbf{IN}$, **OUT** is **conservative** if
  1. $\mathbf{IN}[s] = \emptyset$
  2. For each node *bb* $\in N$,
  $$\mathbf{OUT}[bb] \subseteq post_{AE}(bb, \mathbf{IN}[bb])$$
  3. For each edge *src* $\rightarrow$ *dst* $\in E$,
  $$\mathbf{IN}[dst] \subseteq \mathbf{OUT}[src]$$
- Fact: if $\mathbf{IN}, \mathbf{OUT}$ is a conservative assignment, then:
  - If $e \in \mathbf{IN}[bb]$, then e is available at entry of *bb*
  - Similarly for $\mathbf{OUT}$

Workset algorithm

Input : Control flow graph (N, E, s), with expressions U
Output: Greatest conservative assignment of available expressions

```
IN[s] = ∅;
OUT[s] = U;
IN[n] = OUT[n] = U for all other nodes n;
work ← N;                                   /* Set of nodes that may violate spec */
while work ≠ ∅ do
    Pick some n from work;
    work ← work \ {n} ;
    old ← OUT[n];
    IN[n] ← ⋂_{p∈pred(n)} OUT[p];
    OUT[n] ← post_AE(n, IN[n]);
    if old ≠ OUT[n] then
        work ← work ∪ succ(n)
return IN, OUT
```

Constant propagation

Want *smallest* assignment $\mathbf{IN}, \mathbf{OUT}$ such that

- $\mathbf{IN}[s] = \{x_1 \mapsto \top, \ldots, x_n \mapsto \top\}$
- For each $n \in N$, $\mathbf{OUT}[n] \sqsupseteq post_{CP}(n, \mathbf{IN}[n])$
- For each $p \to n \in E$, $\mathbf{OUT}[p] \sqsubseteq \mathbf{IN}[n]$

Available expressions

Want *greatest* assignment $\mathbf{IN}, \mathbf{OUT}$ such that

- $\mathbf{IN}[s] = \emptyset$
- For each $n \in N$, $\mathbf{OUT}[n] \subseteq post_{AE}(n, \mathbf{IN}[n])$
- For each $p \to n \in E$, $\mathbf{OUT}[p] \supseteq \mathbf{IN}[n]$

- **Commonality**: consant propagation and available expressions are characterized by optimal solutions to a system of local constraints
  - "Local": defined in terms of *edges*; contrast with "global", which depends on the structure of the whole graph (e.g., paths)

Constant propagation

Want *smallest* assignment $\mathbf{IN}, \mathbf{OUT}$ such that

- $\mathbf{IN}[s] = \{x_1 \mapsto \top, \dots, x_n \mapsto \top\}$
- For each $n \in N$, $\mathbf{OUT}[n] \sqsupseteq post_{\mathsf{CP}}(n, \mathbf{IN}[n])$
- For each $p \to n \in E$, $\mathbf{OUT}[p] \sqsubseteq \mathbf{IN}[n]$

Available expressions

Want *greatest* assignment $\mathbf{IN}, \mathbf{OUT}$ such that

- $\mathbf{IN}[s] = \emptyset$
- For each $n \in N$, $\mathbf{OUT}[n] \subseteq post_{\mathsf{AE}}(n, \mathbf{IN}[n])$
- For each $p \to n \in E$, $\mathbf{OUT}[p] \supseteq \mathbf{IN}[n]$

- **Commonality**: consant propagation and available expressions are characterized by optimal solutions to a system of local constraints
  - "Local": defined in terms of *edges*; contrast with "global", which depends on the structure of the whole graph (e.g., paths)
- The algorithms for constant propagation & available expressions are *essentially the same*

Dataflow analysis

- *Dataflow analysis* is an approach to program analysis that unifies the presentation and implementation of many different analyses
  - **Formulate** problem as a system of constraints
  - **Solve** the constraints iteratively (using some variation of the workset algorithm)

Dataflow analysis

- *Dataflow analysis* is an approach to program analysis that unifies the presentation and implementation of many different analyses
  - **Formulate** problem as a system of constraints
  - **Solve** the constraints iteratively (using some variation of the workset algorithm)
- What now:
  - General theory & algorithms
  - Conditions under which the approach works
  - Guarantees about the solution

Dataflow analysis

- *Dataflow analysis* is an approach to program analysis that unifies the presentation and implementation of many different analyses
  - **Formulate** problem as a system of constraints
  - **Solve** the constraints iteratively (using some variation of the workset algorithm)
- What now:
  - General theory & algorithms
  - Conditions under which the approach works
  - Guarantees about the solution
- Not covered: *abstract interpretation* – a general theory for relating program analysis to program semantics
  - What does it mean for a constraint system to be correct?
  - How do we prove it?

A (forward) dataflow analysis consists of:

- An **abstract domain** $\mathcal{L}$
  - Defines the space of program "properties" that we are interested in
- An **abstract transformer** *post*$_{\mathcal{L}}$
  - Determines how each basic block transforms properties
  - i.e., if property p holds *before* n, then *post*$_{\mathcal{L}}(n, p)$ is a property that holds *after* n

Abstract domains

An abstract domain is a set $\mathcal{L}$ equipped with:

- A partial order $\sqsubseteq$
  - $x \sqsubseteq y$ means that x represents more precise information about the program than y[1]
  - $\sqsubset$ denotes corresponding *irreflexive* relation

[1] The other direction also works, and is the one taken in classical compilers literature. In this class, we will stick to this direction, which is the convention established in abstract interpretation.

Abstract domains

An abstract domain is a set $\mathcal{L}$ equipped with:

- A partial order $\sqsubseteq$
  - $x \sqsubseteq y$ means that x represents more precise information about the program than y[1]
  - $\sqsubset$ denotes corresponding *irreflexive* relation
- A *least upper bound* ("join") operator, $\sqcup$
  1. $x \sqsubseteq x \sqcup y$
  2. $y \sqsubseteq x \sqcup y$
  3. $x \sqcup y \sqsubseteq z$ for any z satisfying 1 and 2

[1] The other direction also works, and is the one taken in classical compilers literature. In this class, we will stick to this direction, which is the convention established in abstract interpretation.

Abstract domains

An abstract domain is a set $\mathcal{L}$ equipped with:

- A partial order $\sqsubseteq$
  - $x \sqsubseteq y$ means that x represents more precise information about the program than y[1]
  - $\sqsubset$ denotes corresponding *irreflexive* relation
- A *least upper bound* ("join") operator, $\sqcup$
  1. $x \sqsubseteq x \sqcup y$
  2. $y \sqsubseteq x \sqcup y$
  3. $x \sqcup y \sqsubseteq z$ for any z satisfying 1 and 2
- A *least element* ("bottom"), $\bot$
  - $\bot \sqsubseteq x$ for all x
  - $\bot \sqcup x = x \sqcup \bot = x$ for all x

[1] The other direction also works, and is the one taken in classical compilers literature. In this class, we will stick to this direction, which is the convention established in abstract interpretation.

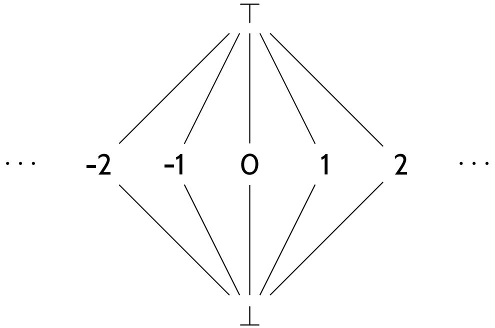
Abstract domains

An abstract domain is a set $\mathcal{L}$ equipped with:

- A partial order $\sqsubseteq$
  - $x \sqsubseteq y$ means that x represents more precise information about the program than y[1]
  - $\sqsubset$ denotes corresponding *irreflexive* relation
- A *least upper bound* ("join") operator, $\sqcup$
  1. $x \sqsubseteq x \sqcup y$
  2. $y \sqsubseteq x \sqcup y$
  3. $x \sqcup y \sqsubseteq z$ for any z satisfying 1 and 2
- A *least element* ("bottom"), $\bot$
  - $\bot \sqsubseteq x$ for all x
  - $\bot \sqcup x = x \sqcup \bot = x$ for all x
- A *greatest element* ("top"), $\top$
  - $x \sqsubseteq \top$ for all x
  - $\top \sqcup x = x \sqcup \top = \top$ for all x

[1] The other direction also works, and is the one taken in classical compilers literature. In this class, we will stick to this direction, which is the convention established in abstract interpretation.

- Often convenient to depict partial order as *Haase diagram*
  - Draw a line from x to y if $x \sqsubset y$ and there is no z with $x \sqsubset z \sqsubset y$ (y covers x)
  - $x \sqsubseteq y$ iff there is a upwards path from x to y



Function spaces

- Constant environments are functions mapping *Variables* $\to \mathbb{Z} \cup \{\bot, \top\}$

Function spaces

- Constant environments are functions mapping *Variables* $\rightarrow \mathbb{Z} \cup \{\bot, \top\}$
  - Environments inherit *pointwise ordering* $\sqsubseteq^*$ from the ordering $\sqsubseteq$ on $\mathbb{Z} \cup \{\bot, \top\}$:
    $f \sqsubseteq^* g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in$ *Variables*
  - There is a least and greatest environment

$$\bot^* = (\textbf{fun } x \rightarrow \bot)$$
$$\top^* = (\textbf{fun } x \rightarrow \top)$$

  - Environments have least upper bounds

$$f \sqcup^* g = (\textbf{fun } (x)\text{->}f(x) \sqcup g(x))$$

Function spaces

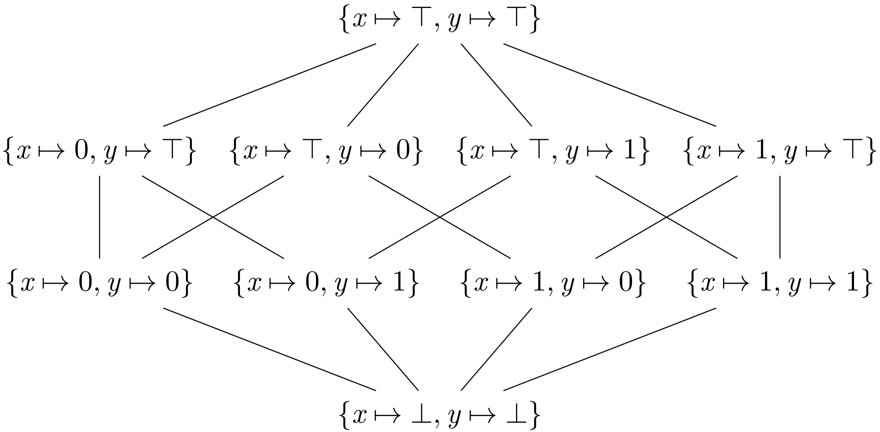
- Constant environments are functions mapping *Variables* $\to \mathbb{Z} \cup \{\bot, \top\}$
  - Environments inherit *pointwise ordering* $\sqsubseteq^*$ from the ordering $\sqsubseteq$ on $\mathbb{Z} \cup \{\bot, \top\}$:
    $f \sqsubseteq^* g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in$ *Variables*
  - There is a least and greatest environment

$$\bot^* = (\textbf{fun } x \to \bot)$$
$$\top^* = (\textbf{fun } x \to \top)$$

  - Environments have least upper bounds

$$f \sqcup^* g = (\textbf{fun } (x)\text{->}f(x) \sqcup g(x))$$

- *This holds more generally*: If $\mathcal{L}$ is an abstract domain and X is any set, the set of functions $X \to \mathcal{L}$ is an abstract domain under the pointwise ordering.

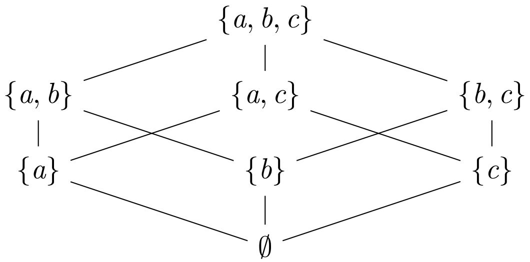


(Identifying $\{x \mapsto \bot, y \mapsto \bot\}$ with all functions that map either x or y to $\bot$)

Powersets

For any set X, the set 2^X of subsets of X is an abstract domain:

- Order $\subseteq$, least element $\emptyset$, greatest element X, join $\cup$
- Order $\supseteq$, least element X, greatest element $\emptyset$, join $\cap$ (*Available Expressions*)



Transfer functions

A transfer function $\textit{post}_{\mathcal{L}} : \textit{Basic Block} \times \mathcal{L} \to \mathcal{L}$ maps each basic block & "pre-state" value to a "post-state" value

- Technical requirement: $\textit{post}_{\mathcal{L}}$ is monotone

$$x \sqsubseteq y \Rightarrow \textbf{\textit{post}}_{\mathcal{L}}(n, x) \sqsubseteq \textbf{\textit{post}}_{\mathcal{L}}(n, y)$$

  ("more information in $\Rightarrow$ more information out")

  - Note: monotonicity is *not* the same as $x \sqsubseteq f(x)$ for all x

Generic (forward) dataflow analysis algorithm

- Given:
  - Abstract domain $(\mathcal{L}, \sqsubseteq, \sqcup, \bot, \top)$
  - Transfer function
    post$_{\mathcal{L}}$: *Basic Block* $\times \mathcal{L} \to \mathcal{L}$
  - Control flow graph $G = (N, E, s)$
- Compute: *least* annotation $\mathbf{IN}$, $\mathbf{OUT}$ such that
  1. $\mathbf{IN}(s) = \top$
  2. For all $n \in N$, *post*$_{\mathcal{L}}(n, \mathbf{IN}[n]) \sqsubseteq \mathbf{OUT}[n]$
  3. For all $p \to n \in E$, $\mathbf{OUT}[p] \sqsubseteq \mathbf{IN}(n)$

Generic (forward) dataflow analysis algorithm

- Given:
  - Abstract domain $(\mathcal{L}, \sqsubseteq, \sqcup, \bot, \top)$
  - Transfer function
    $\mathit{post}_{\mathcal{L}} : \mathit{Basic\ Block} \times \mathcal{L} \to \mathcal{L}$
  - Control flow graph $G = (N, E, s)$
- Compute: *least* annotation $\mathbf{IN}$, $\mathbf{OUT}$ such that
  1. $\mathbf{IN}(s) = \top$
  2. For all $n \in N$, $\mathit{post}_{\mathcal{L}}(n, \mathbf{IN}[n]) \sqsubseteq \mathbf{OUT}[n]$
  3. For all $p \to n \in E$, $\mathbf{OUT}[p] \sqsubseteq \mathbf{IN}(n)$

$\mathbf{IN}[s] = \top, \mathbf{OUT}[s] = \bot$;
$\mathbf{IN}[n] = \mathbf{OUT}[n] = \bot$
  for all other nodes n;
work $\leftarrow N$;
while *work* $\neq \emptyset$ **do**
  Pick some n from work;
  work $\leftarrow$ *work* $\setminus \{n\}$;
  old $\leftarrow \mathbf{OUT}[n]$;
  $\mathbf{IN}[n] \leftarrow \bigsqcup_{p \in \mathit{pred}(n)} \mathbf{OUT}[p]$;
  $\mathbf{OUT}[n] \leftarrow \mathit{post}_{\mathcal{L}}(n, \mathbf{IN}[n])$;
  if $\mathit{old} \neq \mathbf{OUT}[n]$ **then**
    work $\leftarrow$ *work* $\cup$ *succ*(n)
return $\mathbf{IN}, \mathbf{OUT}$

Summary

- Program analyses share common structure
  - Can implement a single workset algorithm and get multiple analyses by "plugging in" different abstract domains and transfer functions
  - Can prove correctness of workset algorithm once-and-for-all in an abstract setting
- Next time: correctness of the general worklist algorithm