COS320: Compiling Techniques

Zak Kincaid

April 7, 2020

Data flow analysis

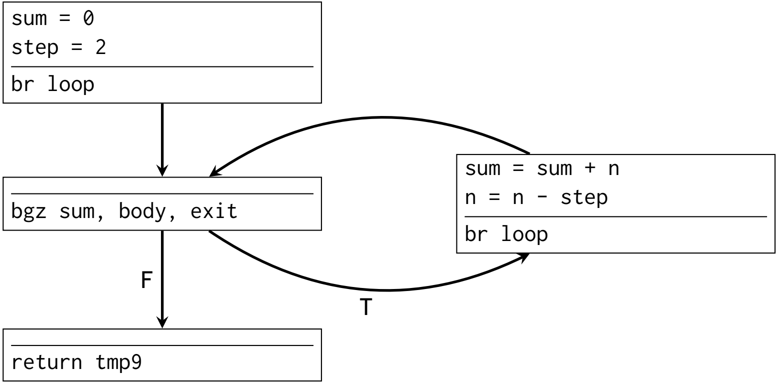
Logistics

- Midterm feedback is on gradscope.com
- Tigar Cyr wrote a syntax highlighting extension for Oat
  `https://marketplace.visualstudio.com/items?itemName=tlcyr4.oat`

Recall: constant propagation

- The goal of constant propagation: determine at each instruction I a *constant environment*
  - 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)
- Say that the assignment $\mathbf{IN}$, **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* $\rightarrow$ *dst* $\in E$,
  $$\mathbf{IN}[dst] \sqsupseteq \mathbf{OUT}[src]$$

```c
int sum2(int n) {
  int sum = 0;
  int step = 2;
  while (n > 0) {
    sum = sum + n;
    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 $\mathbf{IN}[\mathit{bb}]$ and $\mathbf{OUT}[\mathit{bb}]$ 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 additional 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 $n_1, ..., n_k$ such that for each i, there is an edge from $n_i \to n_{i+1} \in E$

Worklist algorithm

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

Worklist algorithm

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

Worklist algorithm

Input : Control flow graph (N, E, s), with variables $x_1, ..., x_n$
Output: Least conservative assignment of constant environments
$\mathbf{IN}[s] = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$;
$\mathbf{OUT}[s] = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$;
$\mathbf{IN}[n] = \mathbf{OUT}[n] = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$ 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* $\setminus \{n\}$;

Worklist algorithm

Input : Control flow graph (N, E, s), with variables $x_1, ..., x_n$
Output: Least conservative assignment of constant environments
$\mathbf{IN}[s] = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$;
$\mathbf{OUT}[s] = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$;
$\mathbf{IN}[n] = \mathbf{OUT}[n] = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$ 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* $\setminus \{n\}$;
  old $\leftarrow \mathbf{OUT}[n]$;
  $\mathbf{IN}[n] \leftarrow \bigsqcup_{p \to n \in E} \mathbf{OUT}[p]$;
  $\mathbf{OUT}[n] \leftarrow$ *post*$(n, \mathbf{IN}[n])$;

Worklist algorithm

Input : Control flow graph (N, E, s), with variables $x_1, ..., x_n$
Output: Least conservative assignment of constant environments
$\mathbf{IN}[s] = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$;
$\mathbf{OUT}[s] = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$;
$\mathbf{IN}[n] = \mathbf{OUT}[n] = \{x_1 \mapsto \bot, ..., x_n \mapsto \bot\}$ 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* $\setminus \{n\}$;
    old $\leftarrow \mathbf{OUT}[n]$;
    $\mathbf{IN}[n] \leftarrow \bigsqcup_{p \to n \in E} \mathbf{OUT}[p]$;
    $\mathbf{OUT}[n] \leftarrow$ *post*$(n, \mathbf{IN}[n])$;
    if $old \neq \mathbf{OUT}(n)$ **then**
        work $\leftarrow$ *work* $\cup$ *succ*(n)
return $\mathbf{IN}, \mathbf{OUT}$

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.

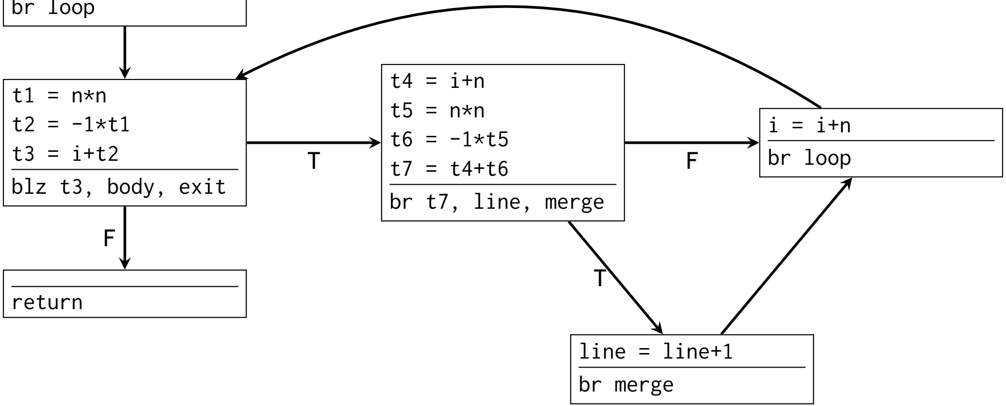
```
void print (long *m, long n) {
  long i,j;
  for (i = 0; i < n*n; i += n) {
    for (j = 0; j < n; j += 1) {
      printf(`` %ld'', *(m + i + j));
    }
    if (i + n < n*n) {
      printf(``\n'');
    }
  }
}
```

$\rightarrow$

```
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 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
F
return
t4 = i+n
t5 = n*n
t6 = -1*t5
t7 = t4+t6
br t7, line, merge
F
T
i = i+n
br loop
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

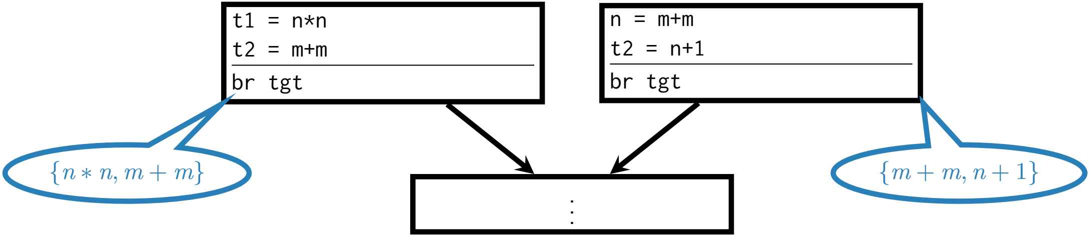
- Given a set of expressions E and an instruction $x = e$
  Assuming the set of expressions E is available *before* the instruction, what expressions are available *after* the instruction?
  - ***post***$_{AE}(x = e, E) = \{e' \in (E \cup \{e\}) : x$ not in $e'\}$
- How do we propagate available expressions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

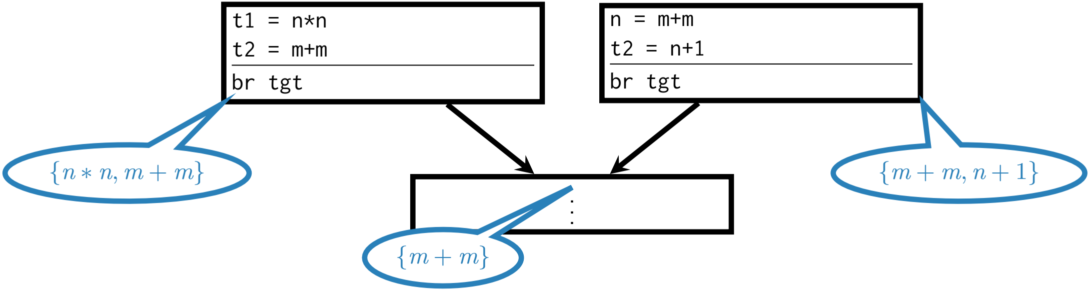
Propagating available expressions

- 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 $instr_1, ..., instr_n, term$.
    take *post*$_{AE}(block, E) =$ *post*$_{AE}(instr_n, ...$*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, $\mathbf{IN}[\mathit{bb}]$ and $\mathbf{OUT}[\mathit{bb}]$
  - $\mathbf{IN}[\mathit{bb}]$ is the set of expressions available at the *entry* of *bb*
  - $\mathbf{OUT}[\mathit{bb}]$ 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* $\in N$,
  $$\mathbf{OUT}[\mathit{bb}] \subseteq \mathit{post}_{\mathit{AE}}(\mathit{bb}, \mathbf{IN}[\mathit{bb}])$$
  3. For each edge *src* $\rightarrow$ *dst* $\in E$,
  $$\mathbf{IN}[\mathit{dst}] \subseteq \mathbf{OUT}[\mathit{src}]$$
- Fact: if $\mathbf{IN}, \mathbf{OUT}$ is a conservative assignment, then:
  - If $e \in \mathbf{IN}[\mathit{bb}]$, then e is available at entry of *bb*
  - Similarly for $\mathbf{OUT}$

Worklist algorithm

Input : Control flow graph (N, E, s), with expressions U
Output: Least 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→n∈E} 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

- $\mathbf{IN}[s] = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$
- 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)

Constant propagation

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

- $\mathbf{IN}[s] = \{x_1 \mapsto \top, ..., x_n \mapsto \top\}$
- 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)
- 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)
- 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]

[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]
- 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]
- 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.

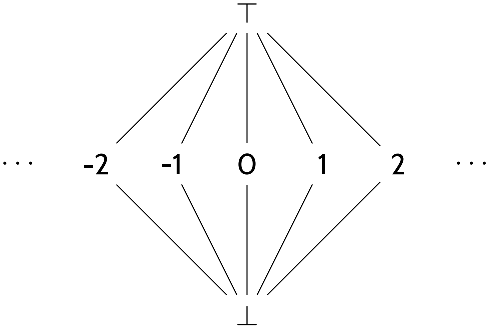
Abstract domains

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]
- 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"), $\perp$
  - $\perp \sqsubseteq x$ for all x
  - $\perp \sqcup x = x \sqcup \perp = x$ 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 $x \sqsubseteq y$ and there is no z with $x \sqsubseteq z \sqsubseteq y$ (y covers x)
  - $x \sqsubseteq y$ iff there is a upwards path from x to y



Function spaces

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

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

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

Function spaces

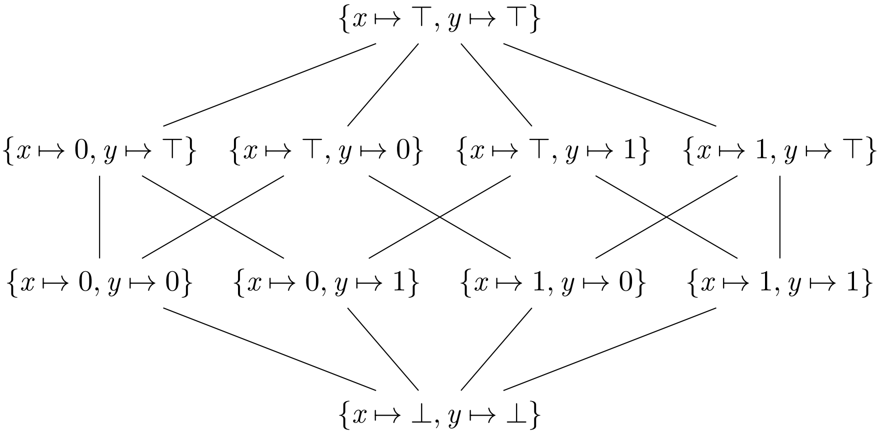
- Constant environments are functions mapping *Variables* $\to \mathbb{Z} \cup \{\bot, \top\}$
  - Environments inherit *pointwise ordering* $\sqsubseteq^*$ from the ordering $\sqsubseteq$ on $\mathbb{Z} \cup \{\bot, \top\}$:
    $f \sqsubseteq^* g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in$ *Variables*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

$$\begin{aligned}\bot^* &= (\textbf{fun } x \to \bot) \\ \top^* &= (\textbf{fun } x \to \top)\end{aligned}$$

  - Environments have least upper bounds

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

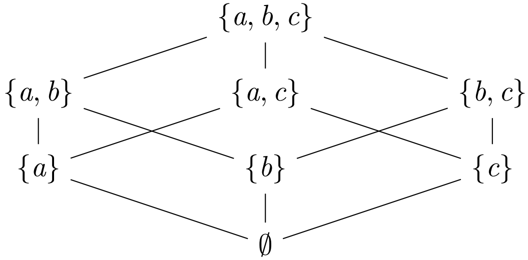
- *This holds more generally*: If $\mathcal{L}$ is an abstract domain and X is any set, the set of functions $X \to \mathcal{L}$ is an abstract domain under the pointwise ordering.


$\{x \mapsto \top, y \mapsto \top\}$
$\{x \mapsto 0, y \mapsto \top\}$
$\{x \mapsto \top, y \mapsto 0\}$
$\{x \mapsto \top, y \mapsto 1\}$
$\{x \mapsto 1, y \mapsto \top\}$
$\{x \mapsto 0, y \mapsto 0\}$
$\{x \mapsto 0, y \mapsto 1\}$
$\{x \mapsto 1, y \mapsto 0\}$
$\{x \mapsto 1, y \mapsto 1\}$
$\{x \mapsto \bot, y \mapsto \bot\}$

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

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

  ("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
    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 $p \to n \in E$, $\mathbf{OUT}[p] \sqsubseteq \mathbf{IN}(n)$

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

Correctness

- When algorithm terminates, all constraints are satisfied. Invariants:
  - $\mathbf{IN}[n] = \top$
  - For any $n \in N$, ***post***$_{\mathcal{L}}(n, \mathbf{IN}[n]) = \mathbf{OUT}[n]$
  - For any $p \to n \in E$ with $\mathbf{OUT}[p] \sqsubseteq \mathbf{IN}(n)$, we have $n \in$ ***work***

Correctness

- When algorithm terminates, all constraints are satisfied. Invariants:
  - $\mathbf{IN}[n] = \top$
  - For any $n \in N$, *post*$_{\mathcal{L}}(n, \mathbf{IN}[n]) = \mathbf{OUT}[n]$
  - For any $p \to n \in E$ with $\mathbf{OUT}[p] \not\sqsubseteq \mathbf{IN}(n)$, we have $n \in$ *work*
- Algorithm computes *least* solution.
  - Invariant: $\mathbf{IN} \sqsubseteq^* \overline{\mathbf{IN}}$ and $\mathbf{OUT} \sqsubseteq^* \overline{\mathbf{OUT}}$, where
    - $\overline{\mathbf{IN}}/\overline{\mathbf{OUT}}$ denotes any solution to the constraint system
    - $\sqsubseteq^*$ is pointwise order on function space $N \to \mathcal{L}$
  - Argument: let $\mathbf{IN}_i/\mathbf{OUT}_i$ be $\mathbf{IN}/\mathbf{OUT}$ at iteration i; n_i be workset item
    - $\mathbf{IN}_{i+1}[n_i] = \bigsqcup_{p \to n_i \in E} \mathbf{OUT}_i[p] \sqsubseteq \bigsqcup_{p \to n_i \in E} \overline{\mathbf{OUT}}[p] \sqsubseteq \overline{\mathbf{IN}}[n_i]$
    - $\mathbf{OUT}_{i+1}[n_i] =$ *post*$_{\mathcal{L}}(n_i, \mathbf{IN}_{i+1}[n_i]) \sqsubseteq$ *post*$_{\mathcal{L}}(n_i, \overline{\mathbf{IN}}[n_i]) \sqsubseteq \overline{\mathbf{OUT}}[n_i]$

Termination

- Why does this algorithm terminate?

Termination

- Why does this algorithm terminate?
  - In general, it doesn’t

Termination

- Why does this algorithm terminate?
  - In general, it doesn't
- Ascending chain condition is sufficient.
  - A partial order $\sqsubseteq$ satisfies the ascending chain condition if any infinite ascending sequence

$$x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots$$

eventually stabilizes: for some i, we have $x_j = x_i$ for all $j \geq i$.

Termination

- Why does this algorithm terminate?
  - In general, it doesn't
- Ascending chain condition is sufficient.
  - A partial order $\sqsubseteq$ satisfies the ascending chain condition if any infinite ascending sequence

$$x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots$$

    eventually stabilizes: for some i, we have $x_j = x_i$ for all $j \geq i$.
  - Fact: X is finite $\Rightarrow$ $(2^X, \subseteq)$ and $(2^X, \supseteq)$ satisfy a.c.c. (*available expressions*)

Termination

- Why does this algorithm terminate?
  - In general, it doesn't
- Ascending chain condition is sufficient.
  - A partial order $\sqsubseteq$ satisfies the ascending chain condition if any infinite ascending sequence

    $$x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots$$

    eventually stabilizes: for some i, we have $x_j = x_i$ for all $j \geq i$.
  - Fact: X is finite $\Rightarrow (2^X, \subseteq)$ and $(2^X, \supseteq)$ satisfy a.c.c. (*available expressions*)
  - Fact: X is finite and $(\mathcal{L}, \sqsubseteq)$ satisfies a.c.c. $\Rightarrow (X \to \mathcal{L}, \sqsubseteq^*)$ satisfies a.c.c. (*constant propagation)*ermina

Termination

- Why does this algorithm terminate?
  - In general, it doesn't
- Ascending chain condition is sufficient.
  - A partial order $\sqsubseteq$ satisfies the ascending chain condition if any infinite ascending sequence

$$x_1 \sqsubseteq x_2 \sqsubseteq x_3 \sqsubseteq \ldots$$

    eventually stabilizes: for some i, we have $x_j = x_i$ for all $j \geq i$.
  - Fact: X is finite $\Rightarrow (2^X, \subseteq)$ and $(2^X, \supseteq)$ satisfy a.c.c. (*available expressions*)
  - Fact: X is finite and $(\mathcal{L}, \sqsubseteq)$ satisfies a.c.c. $\Rightarrow (X \rightarrow \mathcal{L}, \sqsubseteq^*)$ satisfies a.c.c. (*constant propagation)*ermina
- Termination argument:
  - If $(\mathcal{L}, \sqsubseteq)$ satisfies a.c.c., so does the space of annotations $(N \rightarrow \mathcal{L}, \sqsubseteq^*)$
  - $\mathbf{OUT}_0 \sqsubseteq^* \mathbf{OUT}_1 \sqsubseteq^*$ ..., where $\mathbf{OUT}_i$ is the $\mathbf{OUT}$ annotation at iteration i
  - This sequence eventually stabilizes $\Rightarrow$ algorithm terminates