Introduction to Program Analysis and Abstract Interpretation (Part I)

Axel Simon **Olaf Chitil** **Lawrence Beadle**

Materials: http://www.cs.kent.ac.uk/research/groups/tcs/pgradTrain/abstract.html

Acknowledgments: much of this material has been adapted from surveys by the Cousots

Housekeeping + register + pic

11.00	12.00	S110B	Systematic Stuff and Galois Connections	Andy
12.00	1.00	Ruther	Lunch	All
1.00	2.00	S110B	Widening and Narrowing	Andy
2.00	3.45	SW101	Practical Session on Interval Analysis	Axel and Lawrence
3.45	5.00	ELET	Application focus: Polyhedral Analysis	Axel
7.00	9.00	Town	Thomas Becket (subsided by ASTReNet to the value of £300); Magnolia is pre-paid	Any
9.30	10.30	S110B	Lattices	Andy
10.30	11.00	S110B	Symbolic Execution	Andy
11.00	12.00	S110B	Practical Session on Sign Analysis	Axel and Lawrence
12.00	1.00	Ruther	Lunch	All
1.00	2.00	S110B	Type and Effect Systems	Olaf

Why is there so much interest in abstract interpretation?

Abstract interpretation can be applied to systematically construct algorithms for approximating undecidable or complex problems in:

- Verification: can this bad behaviour occur, ie, will this program always terminate/never deadlock?
- Specialisation: how is this code called, ie, can this part of the program even be eliminated?
- Security: can this security problem arise, ie, can a buffer be indexed out of range?
- Semantics: can I build a semantics that only expresses this particular observable?

Pointers to the literature

- SAS, VMCAI, POPL, ESOP, PLDI, ICLP, ICFP,…
- Useful review articles and books:
  - Patrick and Radhia Cousot, Comparing the Galois connection and Widening/Narrowing approaches to Abstract Interpretation, PLILP, LNCS 631, 269-295, 1992. Available from LIX library.
  - Patrick and Radhia Cousot, Abstract interpretation and Application to Logic Programs, The Journal of Logic Programming (JLP), 13(2-3):103-179, 1992.
  - Patrick and Radhia Cousot, Basic Concepts of Abstract Interpretation, Building the Information Society, 539-366, 2004
  - Flemming Neilson, Hanne Riis Neilson and Chris Hankin, Principles of Program Analysis, Springer, 1999.

Transition systems

- Program semantics are often formalised as a transition system $\langle \Sigma, \Sigma_i, \rho \rangle$ where:
  - Σ is a finite set of states
  - $\Sigma_i \subseteq \Sigma$ is a (finite) set of initial states
  - $\rho \subseteq \Sigma \times \Sigma$ is a (finite) binary relation
- Example formalisation and program:
  - $\Sigma = [0, 2^{16} - 1]$
  - $\Sigma_i = \{0\}$
  - $\rho = \{\langle i, i' \rangle \mid i < 16 \wedge i' = i+1\}$

```
i = 0;
while (i < 16)
    i = i + 1;
```

Collecting semantics (sets of possible exact behaviour)

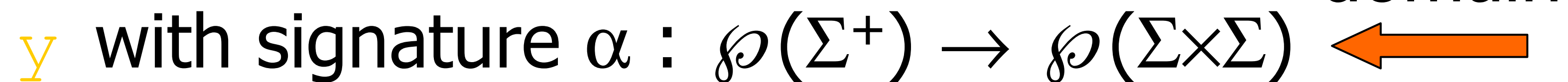
- Set of all traces $T_\infty = \cup_{0<k} T_k$ where:
  - $T_1 = \Sigma_i$
  - $T_{k+1} = \{ t.s.s' \mid t.s \in T_k \wedge \langle s, s' \rangle \in \rho \}$
- Fixpoint semantics for traces:
  - If $F(T) = \Sigma_i \cup \{ t.s.s' \mid t.s \in T \wedge \langle s, s' \rangle \in \rho \}$
  - then $F(T_\infty) = T_\infty$ so T_∞ is a fixpoint
  - If $F(T) = T$
  - then $T_\infty \subseteq T$ so T_∞ is the least fixpoint
- Thus $\mathrm{lfp}(F) = T_\infty$

If $F(T) = T$ then $T_\infty \subseteq T$

- Use induction to show $T_k \subseteq T$ for all k
- Base case: $T_1 = \Sigma_i \subseteq F(T) = T$
- Inductive case:
  - Suppose $T_k \subseteq T$ for some $k > 0$
  - To show $T_{k+1} \subseteq T$ so let $t \in T_{k+1}$
  - Now $t = t'.s.s'$ where $t'.s \in T_k$ and $\langle s, s' \rangle \in \rho$
  - Since $T_k \subseteq T$ it follows $t'.s \in T$ thus $t \in F(T) = T$
- Thus $\cup_{0<k} T_k \subseteq T$ hence $T_\infty \subseteq T$ as required

Binning intermediate state for reachability from Σ_i

- Approximating a single trace:
  - $\alpha(s_1.t.s_n) = \langle s_1, s_n \rangle$ for any $n \geq 1$
  - with signature $\alpha : \Sigma^+ \rightarrow \Sigma \times \Sigma$
- Approximating a set of traces:
  - $\alpha(T) = \{\alpha(t) \mid t \in T\}$
  - with signature $\alpha : \wp(\Sigma^+) \rightarrow \wp(\Sigma \times \Sigma)$



domain

- What is described by a set of pairs $P \subseteq \Sigma \times \Sigma$:
  - $\gamma(P) = \{t \in \Sigma^+ \mid \alpha(t) \in P\}$
  - with signature $\gamma : \wp(\Sigma \times \Sigma) \rightarrow \wp(\Sigma^+)$

Galois connections

- Galois connection when $\alpha(T) \subseteq P$ iff $T \subseteq \gamma(P)$
- Galois connection holds since:
  - $\alpha(T) \subseteq P$ iff
  - $\{\alpha(t) \mid t \in T\} \subseteq P$ iff
  - $\alpha(t) \in P$ for all $t \in T$ iff
  - $T \subseteq \{t \mid \alpha(t) \in P\}$ iff
  - $T \subseteq \gamma(P)$

Seeing the wood *and* the trees

- Want to compute $\alpha(T_\infty)$
- But Σ^+ is infinite even if Σ is finite
- Thus T_∞ may be infinite
- Thus $\alpha(T_\infty)$ is not generally computable
- Engineer $F' : \wp(\Sigma\times\Sigma) \rightarrow \wp(\Sigma\times\Sigma)$
  to simulate $F : \wp(\Sigma^+) \rightarrow \wp(\Sigma^+)$
- Such that $\alpha(\mathrm{lfp}(F)) \subseteq \mathrm{lfp}(F')$
- But $\Sigma\times\Sigma$ and $\wp(\Sigma\times\Sigma)$ are finite
- Then $\alpha(T_\infty) = \alpha(\mathrm{lfp}(F)) \subseteq \mathrm{lfp}(F')$

Nice properties

- Given $\gamma : \wp(\Sigma\times\Sigma) \rightarrow \wp(\Sigma^{+})$
  - Synthesise $\alpha(T) = \cap\{ P\subseteq\Sigma\times\Sigma \mid T \subseteq \gamma(P)\}$
- Both α and γ are monotonic:
  - If $T \subseteq T'$ then $\alpha(T) \subseteq \alpha(T')$
- α is additive: $\alpha(\cup_{i>0}T^{i}) = \cup_{i>0}\, \alpha(T^{i})$
- Special case: $\alpha(T\cup T') = \alpha(T) \cup \alpha(T')$

Constructing F': $\wp(\Sigma\times\Sigma)\to\wp(\Sigma\times\Sigma)$ from F : $\wp(\Sigma^+)\to\wp(\Sigma^+)$

- Put $F'(P) = \alpha(F(\gamma(P)))$
- Recall $F(T) = \Sigma_i \cup \{t.s.s' \mid t.s \in T \wedge \langle s, s'\rangle \in \rho\}$
- Hence $F(\gamma(P))$

  $= \Sigma_i \cup \{(r.t'.s).s' \mid \langle r,s\rangle \in P \wedge \langle s,s'\rangle \in \rho\}$
- Hence $\alpha(F(\gamma(P)))$

  $= \alpha(\Sigma_i \cup \{(r.t'.s).s' \mid \langle r,s\rangle \in P \wedge \langle s,s'\rangle \in \rho\})$

  $= \alpha(\Sigma_i) \cup \alpha(\{(r.t'.s).s' \mid \langle r,s\rangle \in P \wedge \langle s,s'\rangle \in \rho\})$

  $= (\Sigma_i\times\Sigma_i) \cup \{\langle r, s'\rangle \mid \langle r,s\rangle \in P \wedge \langle s,s'\rangle \in \rho\}$

Does it follow that $\alpha(\text{lfp}(F)) \subseteq \text{lfp}(F')$?

- Recall $\text{lfp}(F) = T_\infty = \cup_{0<k} T_k$
- Similarly show $\text{lfp}(F') = \cup_{0<k} T'_k$ where:
  - $T'_1 = \Sigma_i \times \Sigma_i$
  - $T'_{k+1} = \{\langle r, s' \rangle \mid \langle r,s \rangle \in T'_k \wedge \langle s,s' \rangle \in \rho\}$
- Use induction to show $\alpha(T_k) \subseteq T'_k$ for all k
- By additivity it follows that:
  - $\alpha(\cup_{0<k} T_k) = \cup_{0<k} \alpha(T_k) \subseteq \cup_{0<k} T'_k$
- Hence $\alpha(\text{lfp}(F)) \subseteq \text{lfp}(F')$ as required

Answering concrete questions in an abstract setting

- One can sometimes answer a question over $\wp(\Sigma^+)$ by using $\wp(\Sigma\times\Sigma)$
- Given s,s′,s″ does a trace $s.t.s'.t'.s'' \in T_\infty = lfp(F)$?
  - No if $\langle s, s''\rangle \notin lfp(F')$ (sound)
  - Maybe if $\langle s, s''\rangle \in lfp(F')$
- Given s,s″ does a trace $s.t.s'' \in lfp(F)$?
  - No if $\langle s, s''\rangle \notin lfp(F')$ (sound)
  - Yes if $\langle s, s''\rangle \in lfp(F')$ (complete)
- Soundness follows from $\alpha(lfp(F)) \subseteq lfp(F')$

Summary

- Formalise the semantics over Things
- Formulate a collecting semantics as lfp(F) where:
  - F : $\wp$(Things) $\rightarrow$ $\wp$(Things)
- Choose the abstract domain: $\wp$(Descriptions)
- Tie down the abstraction α:
  - α : $\wp$(Things) $\rightarrow$ $\wp$(Descriptions)
- Synthesise γ from α
- Synthesise F′ from F

- But is lfp(F′) also suitably precise (too coarse)?

Widening and narrowing (Part II)

Abstract interpretation does not require a domain to be finite

Interval approximation

- Consider again the following program
- SYNTOX [PLDI'90] inferred the invariants placed in comments
- Invariants occur *between* consecutive lines in the program
- i∈[0,15] asserts 0≤i≤15 whereas i∈[0,0] means i=0

```
i = 0;
/* 1: i∈[0,0] */
while (i < 16) {
        /* 2: i∈[0,15] */
        i = i + 1;
        /* 3: i∈[1,16] */
 }
/* 4: i∈[16,16] */
```

An infinite domain

- Domain of intervals Int includes:
  - $\bot$ to represent the empty set of integers
  - $[l,u]$ where $l \leq u$ and $l,u \in \mathcal{Z}$ for bounded sets
  - $[-\infty,u]$ for sets which are not bounded below but bounded above, etc
- $Int = \{\bot\} \cup \{[l,u] \mid l \leq u \wedge l \in \mathcal{Z} \cup \{-\infty\} \wedge u \in \mathcal{Z} \cup \{\infty\}\}$ where $\leq$ is extended so that:
  - $-\infty \leq \infty$ and $l \leq \infty$ and $-\infty \leq u$ for all $l,u \in \mathcal{Z}$
- The ordering $\subseteq$ extends to Int by:
  - $\bot \subseteq [l,u]$
  - $[l,u] \subseteq [l',u']$ iff $l' \leq l$ and $u \leq u'$

danger

α and γ for Int

- Approximating a set:
  - Let inf(s) = if s is bounded below then min(s) else $-\infty$
  - Then α(s) = if s = $\emptyset$ then $\bot$ else [inf(s), sup(s)]
  - with signature $\alpha : \wp(\mathcal{Z}) \rightarrow$ Int
- Concretising an interval:
  - Synthesise γ(d) = $\cup$\{s $\subseteq \mathcal{Z}$ | α(s) $\subseteq$ d\}
  - with signature γ : Int $\rightarrow \wp(\mathcal{Z})$
- Examples: α(\{3,5,7,…\}) = [3, ∞] and γ([3,∞]) = \{3,4,5,…\}

Weakening intervals

```
if … {
    …
    /* 1: i∈[0,2] */
} else {
    …
    /* 2: i∈[3,5] */
}
/* 3: i∈[0,5] */
```

Join (path merge) is defined:

- Put $d_1 \vee d_2 = d_1$ if $d_2 = \bot$
- d_2 else if $d_1 = \bot$
- $[\min(l_1,l_2), \max(u_1,u_2)]$ otherwise
- whenever $d_1 = [l_1,u_1]$ and $d_2 = [l_2,u_2]$

Strengthening intervals

```
/* 3: i∈[0,5] */
if (2 < i) {
   /* 4: i∈[3,5] */
   …
} else {
   /* 5: i∈[0,2] */
   …
}
```

Meet is defined:

- Put $d_1 \wedge d_2 = \bot$ if $(d_1 = \bot) \vee (d_2 = \bot)$
- $= \bot$ else if $\max(l_1,l_2) > \min(u_1,u_2)$
- $= [\max(l_1,l_2), \min(u_1,u_2)]$ otherwise
- whenever $d_1 = [l_1,u_1]$ and $d_2 = [l_2,u_2]$

Meet and join arise in the transfer functions

- Let I_k denote the interval at program point k
- $I_1 = [0,0]$ since program point (1) immediately follows the i = 0
- $I_2 = (I_1 \vee I_3) \wedge [-\infty, 15]$ since:
  - control from program points (1) and (3) flow into (2)
  - point (2) is reached only if $i < 16$ holds
- $I_3 = \{i + 1 \mid i \in I_2\}$ since (3) is only reachable from (2) via the increment
- $I_4 = (I_1 \vee I_3) \wedge [16, \infty]$ since:
  - control from (1) and (3) flow into (4)
  - point (4) is reached only if $\neg(i < 16)$ holds

Interval iteration

I_1	⊥	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
I_2	⊥	⊥	[0,0]	[0,0]	[0,1]	[0,1]	[0,2]	[0,2]
I_3	⊥	⊥	⊥	[1,1]	[1,1]	[1,2]	[1,2]	[1,3]
I_4	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

I_1	...	[0,0]	[0,0]	[0,0]	[0,0]
I_2	...	[0,15]	[0,15]	[0,15]	[0,15]
I_3	...	[1,15]	[1,16]	[1,16]	[1,16]
I_4	...	⊥	⊥	[16,16]	[16,16]

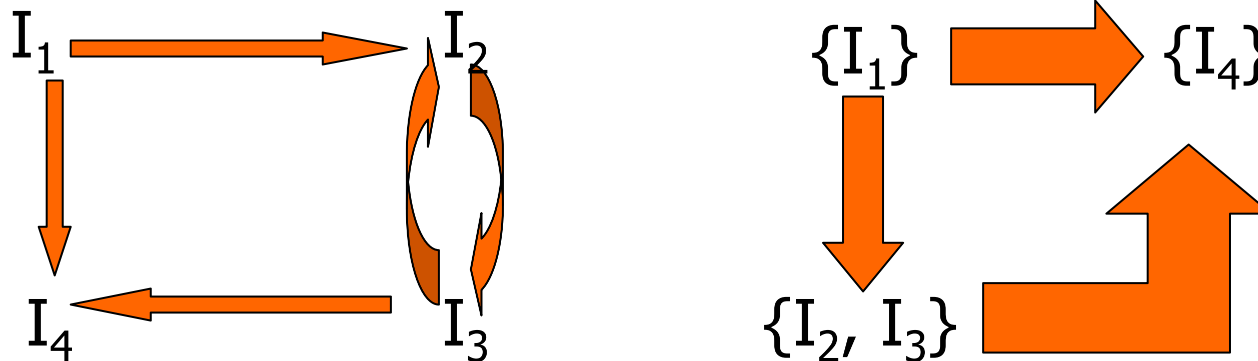
Jacobi versus Gauss-Seidel iteration

- With Jacobi, the new vector $\langle I_1', I_2', I_3', I_4' \rangle$ of intervals is calculated from the old $\langle I_1, I_2, I_3, I_4 \rangle$
- With Gauss-Seidel iteration:
  - I_1' is calculated from $\langle I_1, I_2, I_3, I_4 \rangle$
  - I_2' is calculated from $\langle I_1', I_2, I_3, I_4 \rangle$
  - I_3' is calculated from $\langle I_1', I_2', I_3, I_4 \rangle$
  - I_4' is calculated from $\langle I_1', I_2', I_3', I_4 \rangle$

I_1	⊥	[0,0]	[0,0]	[0,0]	…	[0,0]	[0,0]	[0,0]
I_2	⊥	[0,0]	[0,1]	[0,2]	…	[0,14]	[0,15]	[0,15]
I_3	⊥	[1,1]	[1,2]	[1,3]	…	[1,15]	[1,16]	[1,16]
I_4	⊥	⊥	⊥	⊥	…	⊥	[16,16]	[16,16]

Gauss-Seidel versus chaotic iteration

- The transfer functions induce dependencies between the I_k:



- Observe that I_4 might change if either I_1 or I_3 change, hence evaluate I_4 after I_1 and I_3 stabilise
- Suggests that wait until stability is achieved at one level before starting on the next

Gauss-Seidel versus chaotic iteration (cont')

- Chaotic iteration can postpone evaluating I_i for bounded number of iterations:
  - I_1' is calculated from $\langle I_1,-,-,- \rangle$
  - I_2' and I_3' are calculated Gauss-Seidel style from $\langle I_1,I_2,I_3,- \rangle$
  - I_4' is calculated from $\langle I_1',I_2',I_3',I_4 \rangle$

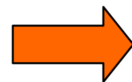
I_1	⊥	[0,0]	[0,0]	[0,0]	…	[0,0]	[0,0]	[0,0]
I_2	⊥	-	[0,0]	[0,1]	…	[0,15]	[0,15]	[0,15]
I_3	⊥	-	[1,1]	[1,2]	…	[1,16]	[1,16]	[1,16]
I_4	⊥	-	-	-	…	-	-	[16,16]

- Fast and (incremental) fixpoint solvers [TOPLAS 22(2):187-223,2000] apply chaotic iteration

Consider changing one symbol

```
i = 0;
/* 1: i∈[0,0] */
while (i < 16) {
        /* 2: i∈[-∞,0] */
        i = i – 1;
        /* 3: i∈[-∞,-1] */
}
/* 4: i∈⊥ */
```

- Int contains the ∞ chain $\bot \subseteq [0,0] \subseteq [-1,0] \subseteq [-2,0] \ldots$
- So Int does not satisfy the ascending chain condition (ACC).



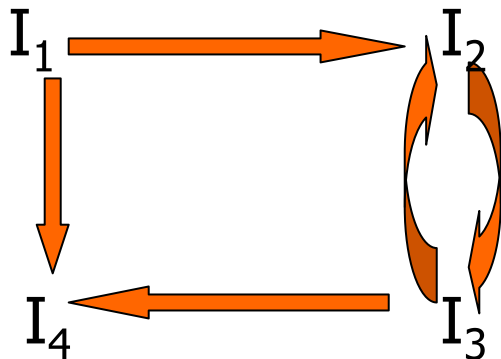
I_1	⊥	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
I_2	⊥	-	[0,0]	[-1,0]	[-2,0]	…
I_3	⊥	-	[-1,0]	[-2,0]	[-3,0]	…
I_4	⊥	-	-	-	-	-

Inducing termination for Int

- Enforce convergence for Int with a widening operator $\nabla : Int \times Int \to Int$
  - $\bot \nabla d = d$
  - $d \nabla \bot = d$
  - $[l_1,u_1] \nabla [l_2,u_2] = [\text{if } l_2<l_1 \text{ then } -\infty \text{ else } l_1, \text{if } u_1<u_2 \text{ then } \infty \text{ else } u_1]$
- Examples:
  - $[1,2]\nabla[1,2] = [1,2]$
  - $[1,2]\nabla[1,3] = [1,\infty]$
  - $[1,3]\nabla[0,2] = [-\infty,3]$

Chaotic iteration with widening

- To terminate it is necessary to traverse each loop a finite number of times



- It is sufficient to pass through I_2 or I_3 a finite number of times [Bourdoncle, 1990]
- Widen at I_3 since its equation is simpler

Chaotic iteration with widening (cont')

- $I_1 = [0,0]$
- $I_2 = (I_1 \vee I_3) \wedge [-\infty, 15]$
- $I_3 = I_3 \nabla \{i-1 \mid i \in I_2\}$
- $I_4 = (I_1 \vee I_3) \wedge [16, \infty]$

I_1	$\perp$	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
I_2	$\perp$	-	[0,0]	[-1,0]	[-∞,0]	[-∞,0]	[-∞,0]
I_3	$\perp$	-	[-1,0]	[-∞,0]	[-∞,0]	[-∞,0]	[-∞,0]
I_4	$\perp$	-	-	-	-	-	$\perp$

- When $I_3 = [-1,0]$ and $I_2 = [-1,0]$ then
  $I_3 \nabla \{i-1 \mid i \in I_2\} = [-1,0] \nabla [-2,-1] = [-\infty,0]$
- And when $I_3 = [-\infty,0]$ and $I_2 = [-\infty,0]$ then
  $I_3 \nabla \{i-1 \mid i \in I_2\} = [-\infty,0] \nabla [-\infty,-1] = [-\infty,0]$

Chaotic iteration with widening (cont' cont')

- $I_1 = [0,0]$
- $I_2 = (I_1 \vee I_3) \wedge [-\infty, 15]$
- $I_3 = I_3 \nabla \{i+1 \mid i \in I_2\}$
- $I_4 = (I_1 \vee I_3) \wedge [16, \infty]$

I_1	$\perp$	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
I_2	$\perp$	-	-	[0,0]	[0,1]	[0,15]	[0,15]	[0,15]
I_3	$\perp$	-	-	[1,1]	$[1,\infty]$	$[1,\infty]$	$[1,\infty]$	$[1,\infty]$
I_4	$\perp$	-	-	-	-	-	-	$[16,\infty]$

- When $I_3 = [1,1]$ and $I_2 = [0,1]$ then
  $I_3 \nabla \{i+1 \mid i \in I_2\} = [1,1] \nabla [1,2] = [1,\infty]$
- And when $I_3 = [1,\infty]$ and $I_2 = [0,15]$ then
  $I_3 \nabla \{i+1 \mid i \in I_2\} = [1,\infty] \nabla [1,16] = [1,\infty]$

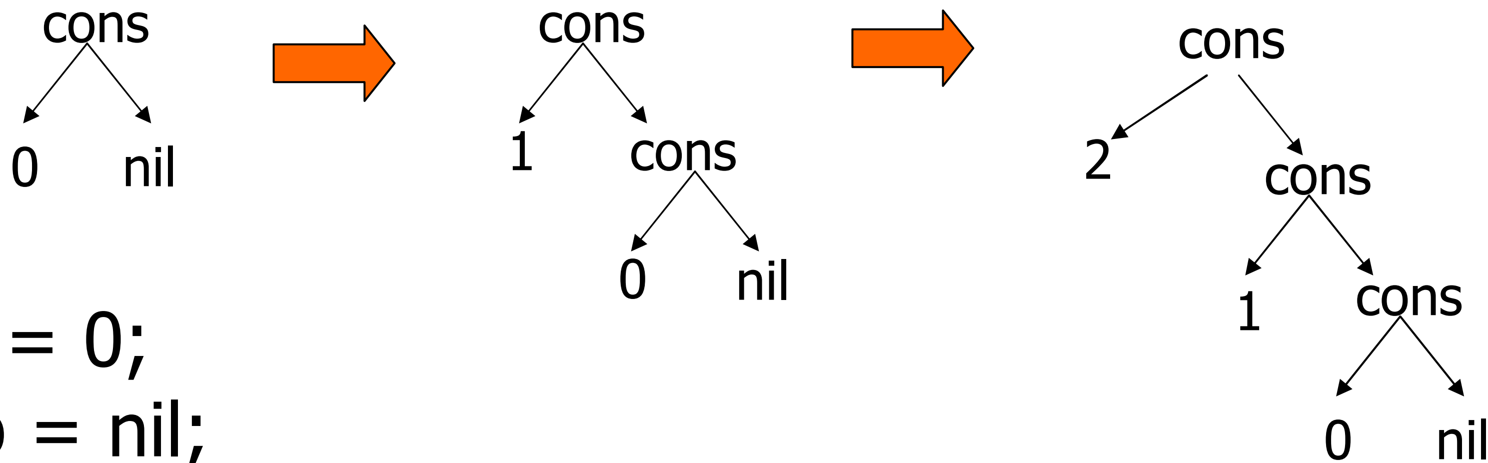
Following widening with narrowing

- $I_1 = [0,0]$
- $I_2 = (I_1 \vee I_3) \wedge [-\infty, 15]$
- $I_3 = I_3 \wedge \{i+1 \mid i \in I_2\}$
- $I_4 = (I_1 \vee I_3) \wedge [16, \infty]$

I_1	…	[0,0]	[0,0]	[0,0]
I_2	…	[0,15]	[0,15]	[0,15]
I_3	…	$[1,\infty]$	[1,16]	[1,16]
I_4	…	-	-	[16,16]

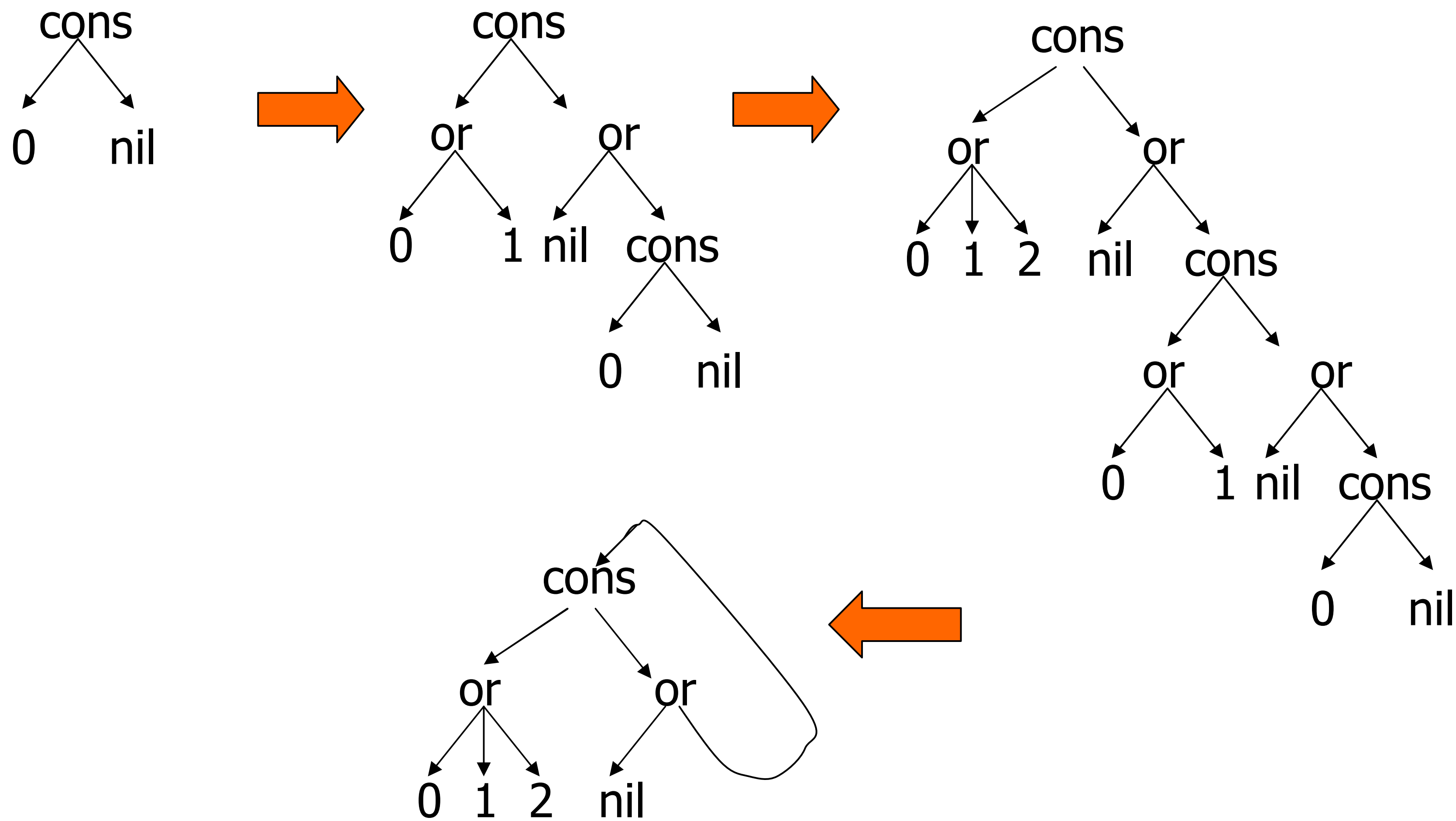
- When $I_3 = [1,\infty]$ and $I_2 = [0,15]$ then $I_3 \wedge \{i+1 \mid i \in I_2\} = [1,\infty] \wedge [1,16] = [1,16]$

An infinite domain of trees




```
i = 0;
p = nil;
while (i < 16)  {
    p = new cons(i, p);
    i = i +1;
    /* 1: p→cons(i,cons(i-1,…)) */
}
```

Widening a chain of trees



- Do you always **need** a Galois connection?
- Do you always **need** widening when the domain is infinite?

Lattices

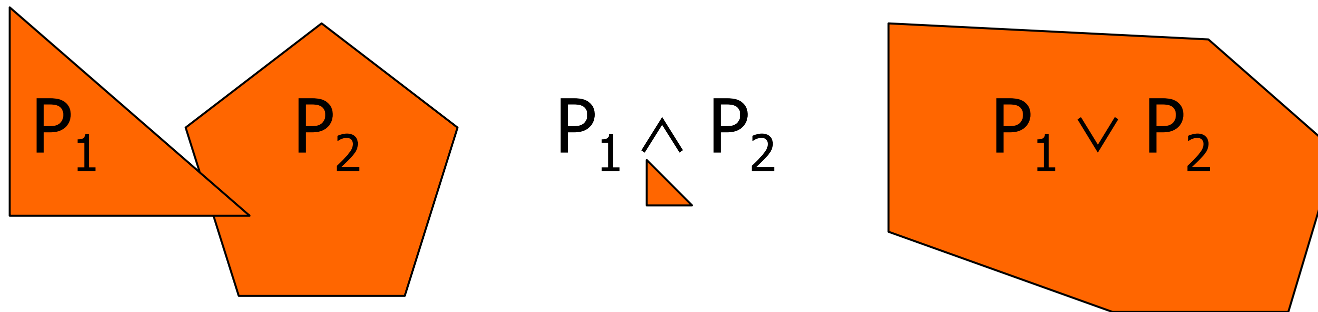
- Suppose $\langle D, \leq \rangle$ is a poset, ie,
  - if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity)
  - $a \leq a$ for all $a \in D$ (reflective)
  - if $a \leq b$ and $b \leq a$ then $a = b$ (anti-symmetric)
- A mapping $\vee : D \times D \rightarrow D$ is a join iff
  - $a \vee b$ is an upper bound of a and b, ie, $a \leq a \vee b$ and $b \leq a \vee b$
  - $a \vee b$ is the least upper bound, ie, if $c \in D$ is an upper bound of a and b, then $a \vee b \leq c$
- The definition of the meet $\wedge : D \times D \rightarrow D$ is analogous
- A lattice $\langle D, \leq, \vee, \wedge \rangle$ is a poset $\langle D, \leq \rangle$ equipped with a join $\vee$ and a meet $\wedge$
- A meet semi-lattice is merely $\langle D, \leq, \wedge \rangle$

Complete lattices

- A lattice is complete if join and meet are defined for arbitrary $E \subseteq D$ rather than merely pairs $\{a,b\} \subseteq D$
- A mapping $\vee : \wp(D) \rightarrow D$ is a join iff whenever $E \subseteq D$:
  - $e \leq (\vee E)$ for all $e \in E$
  - if $e \leq d$ for all $e \in E$ then $(\vee E) \leq d$
- If D is finite then $E = \{e_1, \ldots, e_n\}$ and thus can put $\vee E = e_1 \vee e_2 \vee \ldots \vee e_n$ since $\vee$ is associative
- So any finite lattice is also complete

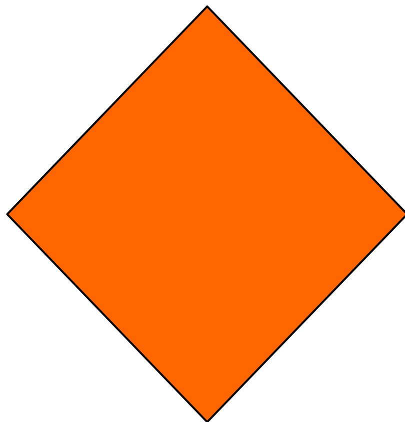
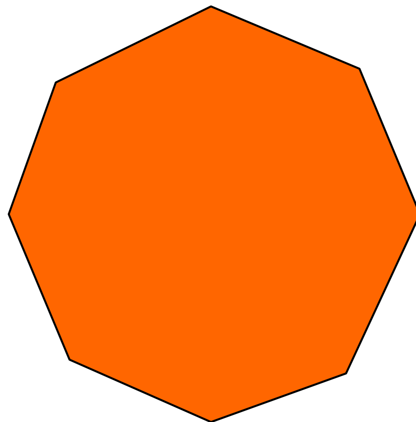
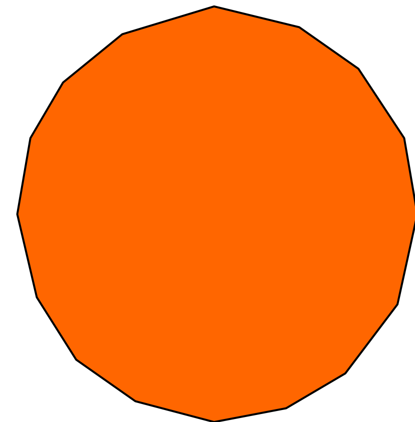
A lattice that is not complete

- For polyhedra P_1 and P_2 in $\mathcal{R}^n$ meet $P_1 \wedge P_2 = P_1 \cap P_2$
- Join $P_1 \vee P_2$ is the (topological closure) of the convex hull of $P_1 \cup P_2$



A lattice that is not complete (cont')

- Consider the following infinite chain of regular polyhedra:

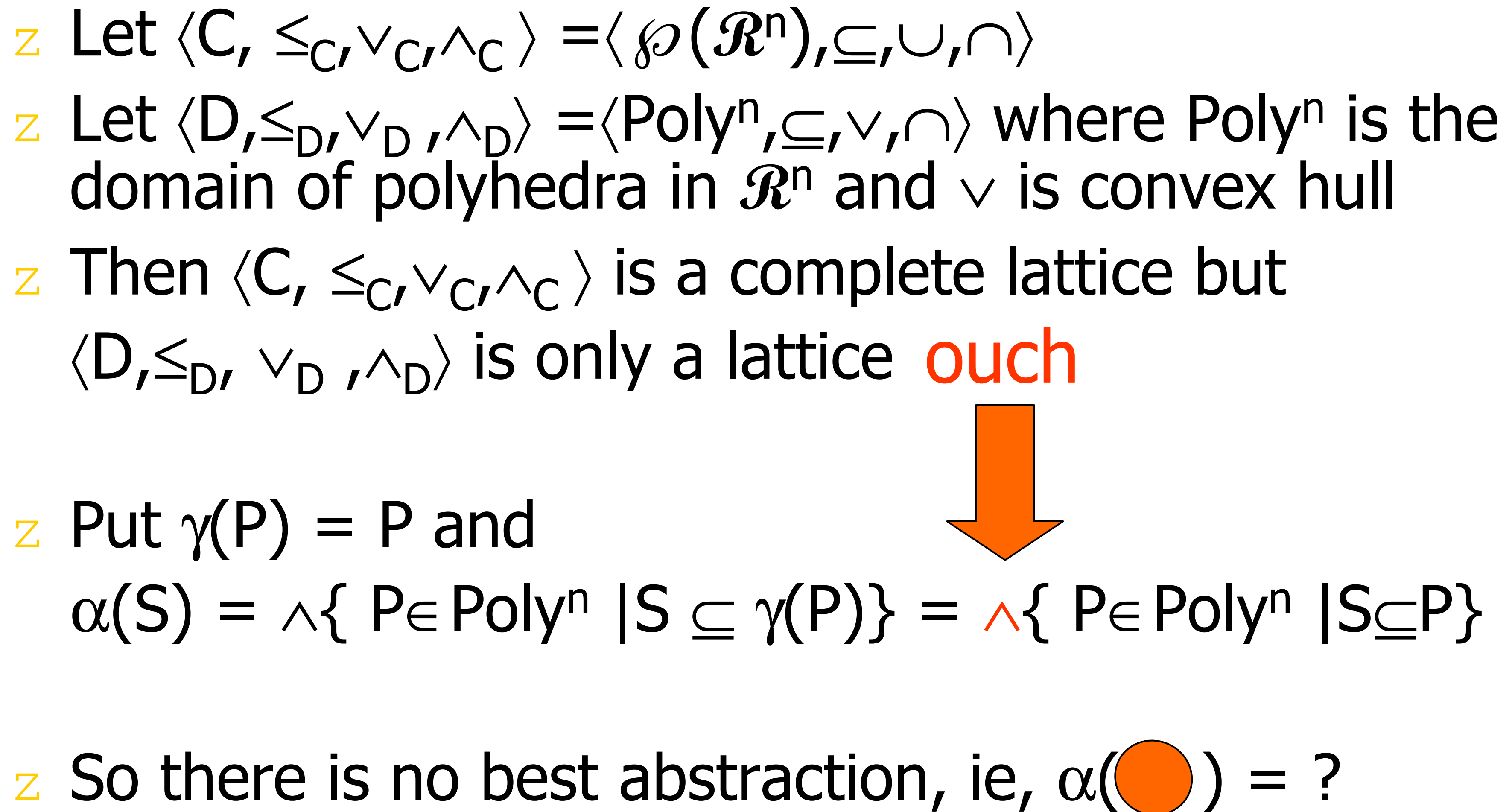
- The only space that contains all these polyhedra is a circle yet this is not polyhedral

Galois connections (reprise)

- Suppose $\langle D, \leq_D \rangle$ and $\langle C, \leq_C \rangle$ are posets.
- Then $\langle D, \alpha, C, \gamma \rangle$ is Galois connection whenever $\alpha(c) \leq_A d$ iff $c \leq_C \gamma(d)$.
- From this it follows that:
  - $\gamma \circ \alpha : C \to C$ is extensive, ie, $c \leq_C (\gamma \circ \alpha)(c)$ for all $c \in C$
  - $\alpha \circ \gamma : D \to D$ is reductive, ie, $(\alpha \circ \gamma)(d) \leq_D d$ for all $d \in D$
  - $\alpha : C \to D$ and $\gamma : D \to C$ are monotonic
  - $\alpha(c) = \wedge_D \{d \in D \mid c \leq_C \gamma(d)\}$
  - $\gamma(d) = \vee_C \{c \in C \mid \alpha(c) \leq_D d\}$
- Note that in the traces example $\langle \wp(\Sigma \times \Sigma), \subseteq, \cup, \cap \rangle$ and $\langle \wp(\Sigma^+), \subseteq, \cup, \cap \rangle$ were complete
- Other stuff (Knaster-Tarski) requires completeness

Absence of a Galois Connection

- Let $\langle C, \leq_C, \vee_C, \wedge_C \rangle = \langle \wp(\mathcal{R}^n), \subseteq, \cup, \cap \rangle$
- Let $\langle D, \leq_D, \vee_D, \wedge_D \rangle = \langle Poly^n, \subseteq, \vee, \cap \rangle$ where $Poly^n$ is the domain of polyhedra in $\mathcal{R}^n$ and $\vee$ is convex hull
- Then $\langle C, \leq_C, \vee_C, \wedge_C \rangle$ is a complete lattice but $\langle D, \leq_D, \vee_D, \wedge_D \rangle$ is only a lattice ouch



- Put $\gamma(P) = P$ and
  $\alpha(S) = \wedge\{ P \in Poly^n \mid S \subseteq \gamma(P)\} = \wedge\{ P \in Poly^n \mid S \subseteq P\}$
- So there is no best abstraction, ie, $\alpha(\bullet) = ?$

Sign analysis

- $\langle C, \leq_C, \vee_C, \wedge_C \rangle = \langle \wp(\mathcal{Z}), \subseteq, \cup, \cap \rangle$ is a complete lattice
- $\langle D, \leq_D, \vee_D, \wedge_D \rangle = \langle Sign, \leq, \vee, \wedge \rangle$ is a finite lattice where:
  - Sign = $\{\bot, +, -, T\}$
  - $\bot \leq d \leq T$ for all $d \in$ Sign
  - $\vee$:Sign×Sign→Sign and $\vee$:Sign×Sign→Sign are defined by:

$\vee$	$\bot$	+	-	T
$\bot$	$\bot$	+	-	T
+	+	+	T	T
-	-	T	-	T
T	T	T	T	T

$\wedge$	$\bot$	+	-	T
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
+	$\bot$	+	$\bot$	+
-	$\bot$	$\bot$	-	-
T	$\bot$	+	-	T

α and γ for Sign

- Concretisation $\gamma: Sign \rightarrow \wp(\mathcal{Z})$ is defined:
  - $\gamma(\bot) = \emptyset$
  - $\gamma(+) = \{n \in \mathcal{Z} \mid n > 0\}$
  - $\gamma(-) = \{n \in \mathcal{Z} \mid n < 0\}$
  - $\gamma(T) = \mathcal{Z}$
- Abstraction $\alpha: \wp(\mathcal{Z}) \rightarrow Sign$ is (directly) defined:
  - $\alpha(S) = \bot$ if $S = \emptyset$
  - $\alpha(S) = +$ else if $n > 0$ for all $n \in S$
  - $\alpha(S) = -$ else if $n < 0$ for all $n \in S$
  - $\alpha(S) = \mathcal{Z}$ otherwise

Galois connections

- Galois connection when $\alpha(S) \leq d$ iff $S \subseteq \gamma(d)$
- Galois connection holds by case analysis:
  - $\alpha(S) \leq +$ iff
  - $\alpha(S) \in \{\bot, +\}$ iff
  - $S = \emptyset$ *or* $n > 0$ for all $n \in S$ iff
  - $n > 0$ for all $n \in S$ iff
  - $S \subseteq \{n \in \mathcal{Z} \mid n > 0\}$ iff
  - $S \subseteq \gamma(+)$

Abstract multiplication for Sign: Safety and Optimality

*	⊥	+	-	T
⊥	⊥	⊥	⊥	⊥
+	⊥	+	-	T
-	⊥	-	+	T
T	⊥	T	T	T

- Safety requires that if $d_1, d_2 \in$ Sign, $n_1 \in \gamma(d_1)$ and $n_2 \in \gamma(d_2)$ then $n_1 * n_2 \in \gamma(d_1 * d_2)$
- Optimality requires that if $d_1, d_2 \in$ Sign then $d_1 * d_2 = \alpha(\{n_1 * n_2 \mid n_1 \in \gamma(d_1) \wedge n_2 \in \gamma(d_2)\})$
- Establish optimality by case analysis, ie,
  - $+ * - = \alpha(\{n_1 * n_2 \mid n_1 \in \gamma(+) \wedge n_2 \in \gamma(-)\}) =$
  - $\alpha(\{n_1 * n_2 \mid n_1 > 0 \wedge n_2 < 0\}) = \alpha(\{n \mid n < 0\}) = -$

An infinite domain that does not require widening

- Let $X = \{x_1, \ldots, x_n\}$ be a set of variables
- Let $Eqn_X = \{a_1x_1 + \ldots + a_nx_n = c \mid a_1, \ldots, a_n, c \in \mathcal{R}\}$
- Let $Eqns_X = \wp(Eqn_X)$ and let solns(S) denote the set of |X|–tuples of solutions to an $S \in Eqns_X$

- Suppose $X = \{x, y, z\}$ then $S_1, S_2 \in Eqns_X$ where:
  - $S_1 = \{x = 3, x + 2y = 1\}$
  - $S_2 = \{x = 3, z = 1, 2z + y = 1\}$
  - $solns(S_1) = \{\langle 3,-1,z\rangle \mid z \in \mathcal{R}\}$
  - $solns(S_2) = \{\langle 3,-1,1\rangle\}$

An ordering on $Eqns_X$

- Let $S, S' \in Eqns_X$.
  Then $S \leq S'$ iff $solns(S) \subseteq solns(S')$
- Is $\langle Eqns_X, \leq \rangle$ is a poset?
  - if $S_1 \leq S_2$ and $S_2 \leq S_3$ then $S_1 \leq S_3$ (transitivity)
  - $S \leq S$ for all $S \in Eqns_X$ (reflexivity)
  - if $S_1 \leq S_2$ and $S_2 \leq S_1$ then $S_1 = S_2$ (anti-symmetry)
- Counter-example to anti-symmetry:
  - Put $S_1 = \{x=1, y=2\}$ and $S_2 = \{x=1, x + y=3\}$
  - Then $solns(S_1) = \{\langle 1,2 \rangle\} = solns(S_2)$
  - But $S_1 \neq S_2$

Engineering a poset

- Introduce an equivalence relation $S \equiv S'$ iff $S \leq S'$ and $S' \leq S$
- Partition $Eqns_X$ into equivalent systems:
  - Put $[S]_{\equiv} = \{S' \in Eqns_X \mid S \equiv S'\}$
  - Put $[S]_{\equiv} \leq [S']_{\equiv}$ iff $S \leq S'$
  - Put $Aff_X = \{[S]_{\equiv} \mid S \in Eqns_X\}$
- Then $\langle Aff_X, \leq \rangle$ is a poset

- If $[S_1]_{\equiv} < [S_1]_{\equiv} < \ldots < [S_k]_{\equiv}$ then $k \leq n$, thus Aff_X does not contain an ∞ ascending chain

Symbolic Execution (Part V)

An antidote to Abstract Interpretation

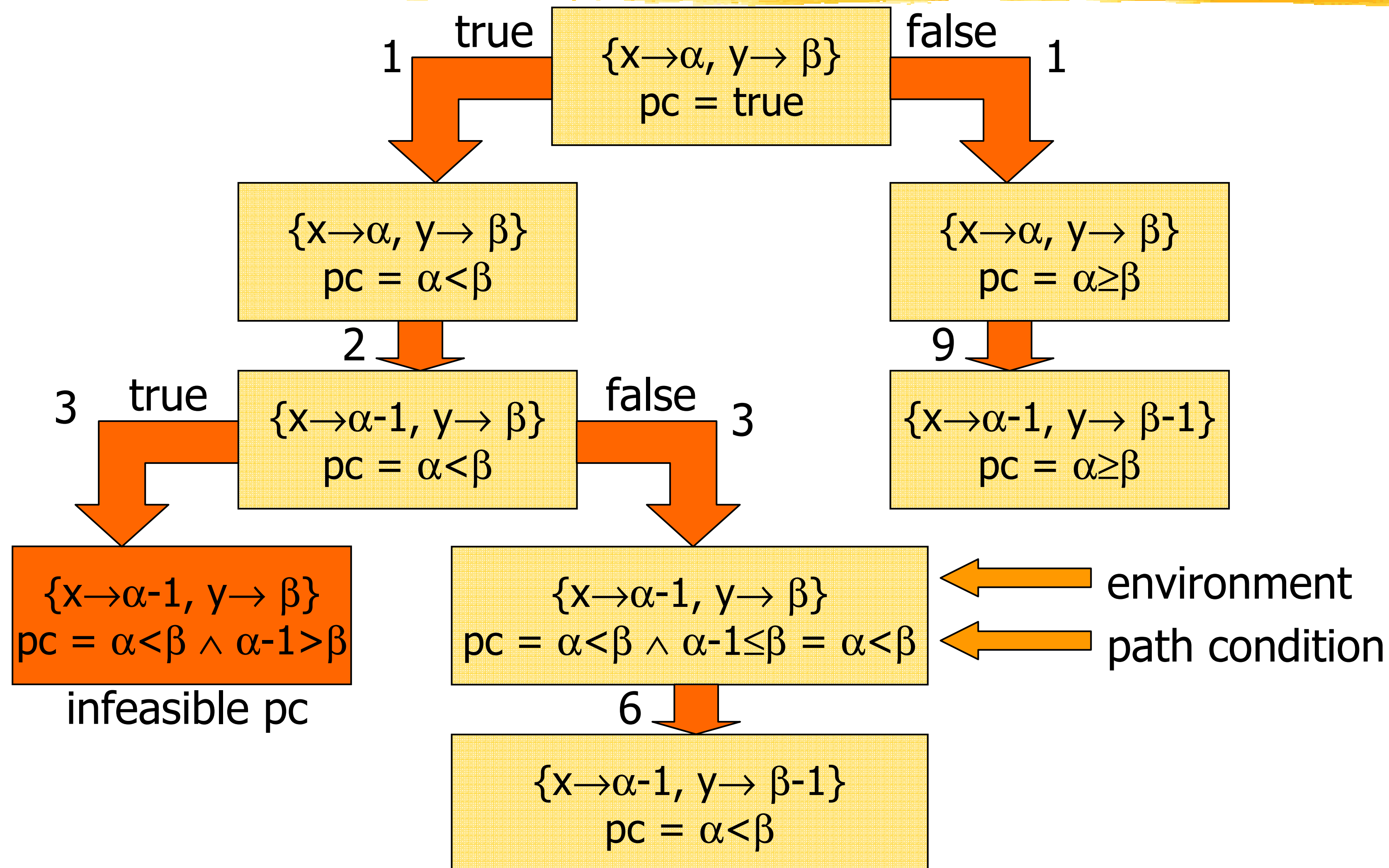
Symbolic Execution versus Abstract Interpretation

- Abstract Interpretation:
  - Generalisation of concrete semantics where the real data object are represented by abstractions
  - Loose information but can examine all paths
  - Conceived as generalisation of *automatic* data-flow analysis
- Symbolic Execution:
  - Generalisation of concrete semantics “where the real data objects are represented by arbitrary symbols”
    James C. King, CACM 19(7), 1976
  - Preserve information but cannot examine all paths
  - Conceived as generalisation of *manual* testing

Example of symbolic execution

```
foo (int x, int y)  {
    if (x < y)  {                  1
        x = x – 1;                 2
        if (x > y)                 3
            y = y + 1;             4
        else                       5
            y = y – 1;             6
    }                              7
    else                           8
        x = y + 1;                 9
}
```

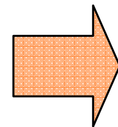
Representing values of variables with symbolic expressions



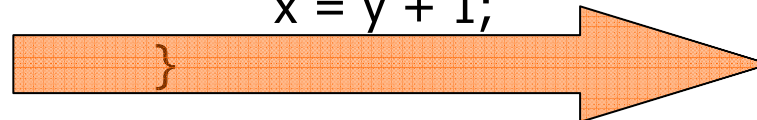
Applications

- King anticipated that the user would "examine the output produced and judge it correctness" of some branches of symbolic execution tree
- Dead-code elimination/strength reduction:

```
foo (int x, int y)  {
   if (x < y)  {
         x = x – 1;
         if (x > y)
            y = y + 1;
         else
            y = y – 1;
   }
   else
      x = y + 1;
}
```



```
foo (int x, int y)  {
   if (x < y)  {
         x = x – 1;
         if (x > y)
            ;
         else
            y = y – 1;
   }
   else
      x = y + 1;
}
```



Since $\alpha<\beta \Rightarrow \alpha\text{-}1\leq\beta$

```
foo (int x, int y)  {
   if (x < y)  {
         x = x – 1;
         y = y – 1;
   }
   else
      x = y + 1;
}
```

Test data generation

```
power (int x, int y)  {
    int w = abs(y);               1
    float z = 1;                  2
    while (w > 0) {               3
            z = z * x;            4
            w = w – 1;            5
    };                            6
    if (y < 0)                    7
        z = 1 / z;                8
    return z;                     9
 }
```

Finding test values that realise 1-2-3-4-5-3-7-9

partial path	*symbolic environment*	*pc*
ε	$\{x \rightarrow \alpha, y \rightarrow \beta, w \rightarrow \bot, z \rightarrow \bot\}$	true
1	$\{x \rightarrow \alpha, y \rightarrow \beta, w \rightarrow abs(\beta), z \rightarrow \bot\}$	true
1-2	$\{x \rightarrow \alpha, y \rightarrow \beta, w \rightarrow abs(\beta), z \rightarrow 1\}$	true
1-2-3	$\{x \rightarrow \alpha, y \rightarrow \beta, w \rightarrow abs(\beta), z \rightarrow 1\}$	$abs(\beta)>0$
1-2-3-4	$\{x \rightarrow \alpha, y \rightarrow \beta, w \rightarrow abs(\beta), z \rightarrow \alpha\}$	$abs(\beta)>0$
1-2-3-4-5	$\{x \rightarrow \alpha, y \rightarrow \beta, w \rightarrow abs(\beta)-1, z \rightarrow \alpha\}$	$abs(\beta)>0$
…-5-3	$\{x \rightarrow \alpha, y \rightarrow \beta, w \rightarrow abs(\beta)-1, z \rightarrow \alpha\}$	$(abs(\beta)>0 \wedge abs(\beta)-1 \leq 0)$ $= (abs(\beta)=1)$
…-5-3-7	$\{x \rightarrow \alpha, y \rightarrow \beta, w \rightarrow abs(\beta)-1, z \rightarrow \alpha\}$ $= \{x \rightarrow \alpha, y \rightarrow 1, w \rightarrow 0, z \rightarrow \alpha\}$	$(abs(\beta)=1 \wedge \beta \geq 0)$ $= (\beta=1)$
…-5-3-7-9	$\{x \rightarrow \alpha, y \rightarrow 1, w \rightarrow 0, z \rightarrow \alpha\}$	$\beta=1$

Symbolic evaluation systems

- A test datum only exists for a path if the pc is definitely satisfiable
- Satisfiability can be checked if the pc is:
  - a system of linear constraints over reals (Simplex)
  - a system of linear constraints over integers (Omega)
  - a system of polynomial equations (Gröbner)
  - a constraint satisfaction problem
- A theorem prover/computer algebra system can also simplify the symbolic environment/pc

Floyd-Hoare verification

- Suppose one as Floyd-style assume/assert predicates for a procedure
- Establish correctness on a per-function basis iff there are only a finite number of paths through the procedure:
  - Use the assume condition for the initial pc
  - For each path though the procedure, compute the final symbolic environment and pc
  - Substitute the local variables in the assert with their symbolic to obtain a symbolic predicate
  - Verify that pc entails the symbolic predicate

Handling loops (á la Hantler and King, ACM Computing Surveys, 8(3), 1976)

```
gcd (int x, int y)  {
    assume (x > 0 && y > 0);
    int a = x;
    int b = y;
    while (a ≠ b) {
cut ⇨   assert (a ≠ b &&
                  gcd(a,b)=gcd(x, y));
        if (a > b) a = a – b;
        else b = b – a;
    };
    assert (a = gcd(x, y))
    return a;
}
```

- Cut each loop
- Apply an "inductive genius" to add an invariant at each cut
- Verify first assert
- Interpret assert as assume to verify again
- Prover must know:
  - if a > b then gcd(a,b) = gcd(a-b,b)
  - gcd(a,b) = gcd(b,a)
  - gcd(a,a) = a

Handling loops (cont')

- Symbolic execution can check a loop invariant but not automatically synthesis one
- If the user does not supply an invariant, then loops can only examined to depth k:
  - Correctness is only guaranteed to depth k
  - Every loop is translated into a nest of k if statements
- Use a semi-algorithm for verification:
  - Search the symbolic execution tree breadth-first
  - Algorithm eventually halts if assert violated

Handling pointer-based data-structures

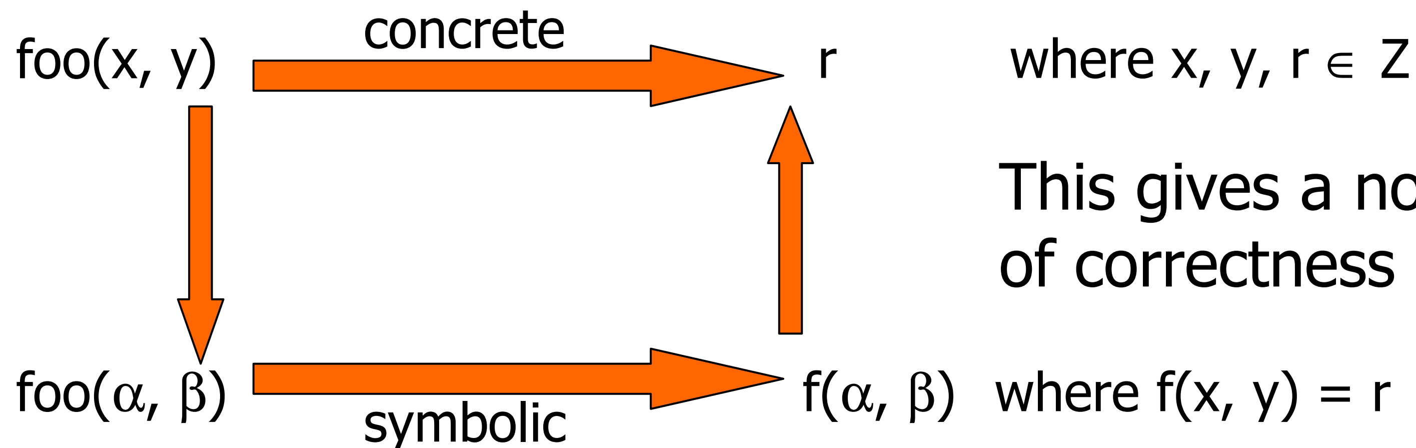
- Arrays and pointers mean that the number of variables (symbols) can be unbounded
- Lazy initialisation/JPF [Visser et al, TACAS'03]:
  - Pointer variables are initialised to ⊥, but can be later bound to:
    - null *or*
    - the symbolic address of a symbolic object that is type consistent *or*
    - the name of another pointer (unification)
  - When a pointer is assigned the value of another pointer (write), its value is merely updated to that the other pointer
  - When the contents of a pointer is actually needed, the name chain is followed. If last named pointer is bound to ⊥, then its ⊥ replaced with:
    - null *or*
    - the symbolic address of an existing type consistent object *or*
    - the symbolic address of a fresh type consistent object
  - Thus branching is performed on demand (also add analysis)
- Symbolic execution then enumerates all heaps up to size k [Lee, PhD Aarhus, November 2006]

Symbolic Execution versus Abstract Interpretation (reprise)

- Abstract Interpretation:
  - Generalisation of concrete semantics where the real data object are represented by abstractions

    (one instance of abstract interpretation is the concrete semantics)
- Symbolic Execution:
  - Generalisation of concrete semantics "where the real data objects are represented by arbitrary symbols"

    (any instance of a symbolic execution is the concrete semantics)



where x, y, r ∈ Z

This gives a notation of correctness

where f(x, y) = r