Karlsruhe Institute of Technology

Static Program Checking

Specification Inference

Automated Software Analysis Group, Institute of Theoretical Informatics

Jun.-prof. Dr. Mana Taghdiri

Monday – July 10, 2014

Statistical justification

- Only reports those invariants that are statistically justified
  - Not the ones that happen to be true by chance
- Depends on the set of values obtained at a program point
- Example 1:
  - In an entire test suite, a program point was executed only 3 times with x = 7, -42, 22
  - The invariants x != 0, x <= 22, x >= -42 will be generated
- Example 2:
  - For 0 < y < 10, 0 < z < 10, given 3 pairs <y, z>, the invariant y != z can be inferred.
  - Might be more reliable if true for 10000 pairs
- Solution?

Statistical justification

- Only reports those invariants that are statistically justified
  - Not the ones that happen to be true by chance
- Depends on the set of values obtained at a program point
- Example 1:
  - In an entire test suite, a program point was executed only 3 times with x = 7, -42, 22
  - The invariants x != 0, x <= 22, x >= -42 will be generated
- Example 2:
  - For 0 < y < 10, 0 < z < 10, given 3 pairs <y, z>, the invariant y != z can be inferred.
  - Might be more reliable if true for 10000 pairs
- Solution 1:
  - Ask for a better test suite
    - But how to generate an ideal test suite?

Statistical justification

- Daikon's solution:
  - For each detected invariant, it computes the probability that the invariant might appear by chance in a random input
  - If the probability is less than a user-provided threshold, then property is not just by chance, and is reported
  - It assumes a distribution and performs a statistical analysis
  - Because actual distribution of variable values is unknown, the computed probability is not absolute, but the exact value is not so important; the order of magnitude is important
  - Daikon uses uniform distribution of values, not a guarantee, but a good measure
  - User's threshold must be very low.
    - Because Daikon checks for millions of invariants

Statistical justification

- Daikon has a probability computation for each invariant
- Example
  - Suppose variable x takes values in a range with size r (based on our samples) containing 0
  - We have s sample values of x
  - Suppose in all samples x != 0
  - Probability of this invariant?

Statistical justification

- Daikon has a probability computation for each invariant
- Example
  - Suppose variable x takes values in a range with size r (based on our samples) containing 0
  - Suppose in all samples x != 0
  - Assuming uniform distribution, probability of x != 0 in one sample = 1 – 1/r
  - Given s sample values of x, probability = $(1 - 1/r)^s$
  - If this probability is less than the threshold, then report the invariant
- More precisely:
  - In an entire test suite, a point was executed only 3 times with x = 7, -42, 22
  - The invariants x != 0, x <= 22, x >= -42 will be generated
  - Probability of nonzero = $(1 - 1/65)^3 = 0.94$
  - So x != 0 will not be reported

Statistical justification

- The statistical heuristic is not a guarantee. So Daikon also outputs the number of values (samples) that support an invariant, so the user can decide

- Problematic case: repeated values
  - A variable is not changed in a loop, but recorded repeatedly at the loop entry
  - Then number of samples is artificially high
  - We don’t like properties derived based on that

Example revisited – add array elements

$$
\begin{array}{l}
i, s := 0, 0; \\
\textbf{do } i \neq n \rightarrow \\
\qquad i, s := i + 1, s + b[i] \\
\textbf{od}
\end{array}
$$

Precondition: $\mathsf{n} \geq 0$
Postcondition: $\mathsf{s} = (\sum \mathsf{j} : 0 \leq \mathsf{j} < \mathsf{n} : \mathsf{b}[\mathsf{j}])$
Loop invariant: $0 \leq \mathsf{i} \leq \mathsf{n}$ and $\mathsf{s} = (\sum \mathsf{j} : 0 \leq \mathsf{j} < \mathsf{i} : \mathsf{b}[\mathsf{j}])$

- Instrumentation at the program entry, the loop head, and program exit.
- Ran on 100 randomly-generated arrays of length 7-13 with elements from -100 to 100.

Example – add array elements

```
15.1.1:::ENTER                    100 samples
  N = size(B)                       (7 values)
  N in [7..13]                      (7 values)
  B                                 (100 values)
    All elements in [-100..100]     (200 values)

15.1.1:::EXIT                     100 samples
  N = I = orig(N) = size(B)         (7 values)
  B = orig(B)                       (100 values)
  S = sum(B)                        (96 values)
  N in [7..13]                      (7 values)
  B                                 (100 values)
    All elements in [-100..100]     (200 values)

15.1.1:::LOOP                     1107 samples
  N = size(B)                       (7 values)
  S = sum(B[0..I-1])                (452 values)
  N in [7..13]                      (7 values)
  I in [0..13]                      (14 values)
  I <= N                            (77 values)
  B                                 (100 values)
    All elements in [-100..100]     (200 values)
  B[0..I-1]                         (985 values)
    All elements in [-100..100]     (200 values)
```

Example – add array elements

```
sum(B) in [-556..539]              (96 values)     *
B[0] nonzero in [-99..96]          (79 values)     *
B[-1] in [-88..99]                 (80 values)     *

N != B[-1]                         (99 values)     *
B[0] != B[-1]                      (100 values)    *
```

- additional invariants if we don't eliminate repeated values

- these invariants are reported for the loop, but nowhere else in the code, although B doesn't change in the program

Repeated values – solutions

- Always:
  - Every sample should be counted, no matter what
- Changed value:
  - A sample is considered if its value is different from last time this program point was examined (doesn’t account for recomputations that result in the same value)
- Assignment:
  - Sample contributes to confidence computation if variable has been assigned since last time seen at this program point. (implemented in Daikon) – takes engineering effort to implement
- Random:
  - If value has changed, then consider it. Otherwise, consider it with a probability of ½

Statistical confidence – bottom line

- An invariant is reported if:
  - There are enough samples that contribute to computing confidence (based on the picked strategy)
  - The computed confidence is higher than the user-specified threshold

Comparable variables – example

```
// Return the sum of the elements of array b, which has length n.
long array_sum(int * b, long n) {
    long s = 0;
    for (int i=0; i<n; i++)
        s = s + b[i];
    return s;
}
```

Unconstrained: all scalars (array elements, indices, addresses)

Source type: (i, elements of b) (s, n)

Coerced: same as unconstrained

Lackwit: ?

Comparable variables – example

```
// Return the sum of the elements of array b, which has length n.
long array_sum(int * b, long n) {
    long s = 0;
    for (int i=0; i<n; i++)
        s = s + b[i];
    return s;
}
```

Unconstrained: all scalars (array elements, indices, addresses)

Source type: (i, elements of b) (s, n)

Coerced: same as unconstrained

Lackwit: (i, n) (s, elements of b)
** if b[i] > 0, then i <= s, but this can't be inferred by lackwit types
** doesn't occur much in practice

Experimental results – comparable vars

Comparability	Other vars	Ratio
Unconstrained	8.8	1.00
Source types	4.5	.51
Coerced types	5.1	.58
Lackwit	0.6	.06

Gives average number of variables to which each variable is comparable, and the ratios between each method and the unconstrained method

Experimental results – effect on invariants

Comparability	Invariants total	Invariants binary	Time
Unconstrained	100%	100%	100%
Source types	78%	61%	91%
Coerced types	78%	74%	96%
Lackwit	55%	27%	75%

Percentage of total and binary invariants reported, and time to compute all invariants, compared to unconstrained comparability

Qualitative analysis of invariants:
Those not reported by Lackwit are all irrelevant for common programming tasks
Example: other techniques produce x < y (for char * pointers) – not useful

How to handle data structures?

- Sample invariants:

```
PriorityQueue:::CLASS
  prio.closure(next).rank is sorted by <=

void PriorityQueue.insert():::EXIT
  size(prio.closure(next)) = size(orig(prio.closure(next))) + 1
```

Data structure invariants

- Pointers are difficult only for recursive data structures where the system may have to traverse arbitrarily many links
  - O.w. a pointer is just a record with two fields (address, and content)

- Invariants are
  - Local: true in objects with a fixed distance from the current variable
    - node.left.parent = node
    - The instrumenter records object fields up to a certain specified depth
  - Global: involves an arbitrary-size collection of objects
    - mytree is sorted
    - num < size(myList)
    - Must explicitly represent the collection

How to handle data structures?

- Linearization
  - Instrumented code traverses data structures and records them explicitly as arrays in program traces
  - Example invariant: mytree is sorted

linearization

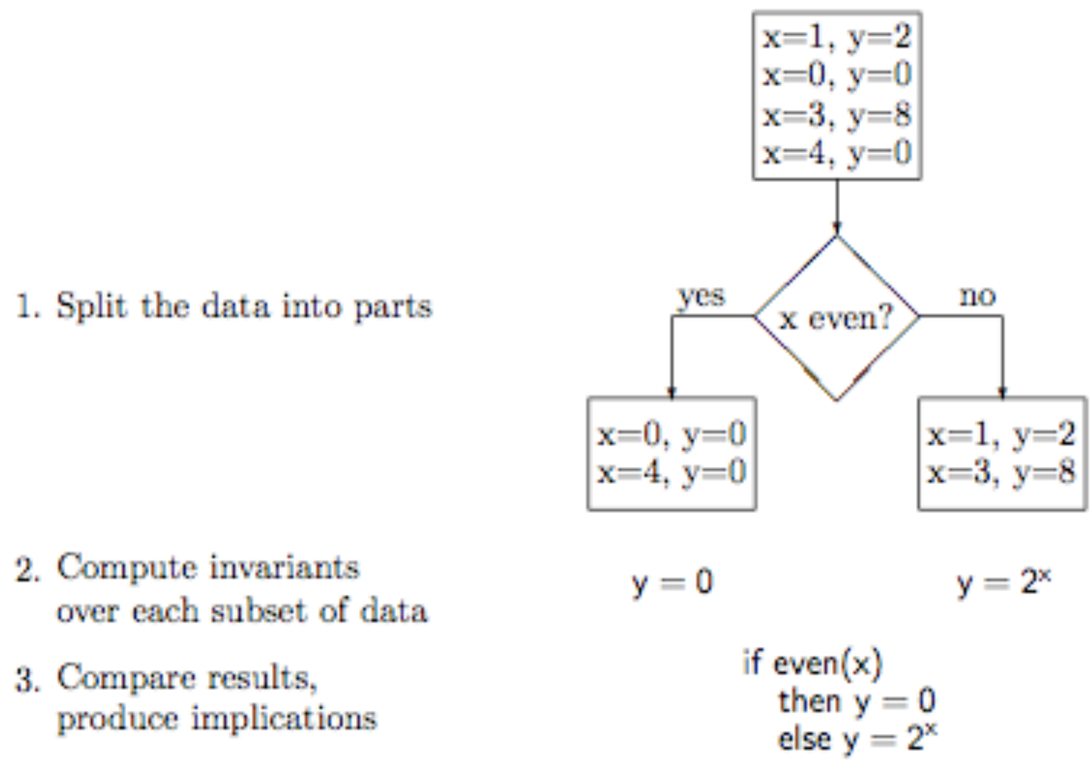
- Linearization involves
  - Selecting a root
    - Current program variables
  - Determining a field for traversing the data structure
    - Fields that point to objects of the same type (next)
    - If there are multiple options (e.g. prev), then makes multiple arrays
    - Also for combinations of fields
      - (in-order, pre-order and post-order of left and right in a tree)
  - Selecting which fields of the visited objects should be written in trace file
    - Fields with non-recursive types are written out
  - Also records special attributes
    - Is cyclic, is a DAG, is a tree
    - This kind of information must be discovered by instrumenter. Is lost after linearization

Conditional invariants

- Many important invariants are not universal
  - p.left.value < p.right.value is true if p, p.right and p.left are non null
  - If arg < 0 then result = -arg else result = arg (absolute value)
  - If x \in orig(list) then size(list) = size(orig(list)) – 1 (list deletion)

Conditional invariants

- Many important invariants are not universal
  - p.left.value < p.right.value is true if p, p.right and p.left are non null
  - If arg < 0 then result = -arg else result = arg (absolute value)
  - If x \in orig(list) then size(list) = size(orig(list)) – 1 (list deletion)



Trace splitting

- What predicates to use for splitting?
- How many of the split candidates to use?
- How to combine them?
- Example
  - For two splitting predicates p and q, there are at least 13 potential subparts of data trace:

Trace splitting

- What predicates to use for splitting?
- How many of the split candidates to use?
- How to combine them?
- Example
  - For two splitting predicates p and q, there are at least 13 potential subparts of data trace:
    - Whole trace (condition true)
    - Four subtraces (p, !p, q, !q)
    - Eight subtraces (p & q, !(p & q), p & !q, !(p & !q),
    - !p & q, !(!p & q), !p & !q, !(!p & !q)
- Daikon uses single-level splitting
  - True, and p, !p, q, !q

Splitting policy

- A static analysis policy:
  - splitting conditions based on analysis of the program's source code
  - Daikon currently implements this policy
  - Uses conditions used for branches in program
    - (if statements and pure boolean member functions)
- A special values policy:
  - compares a variable to preselected values chosen
    - Statically (such as null, zero, or literals in the source code) or
- A policy based on exceptions to detect invariants:
  - tracks variable values that violate potential invariants, rather than immediately discarding the falsified invariant
  - If the number of falsifying samples is moderate, those samples can be separately processed, resulting in a nearly-true invariant plus an invariant over the exceptions
- A programmer-directed policy:
  - allows a user to select splitting conditions a priori

Experiments on data structures

class	relevant	implied	irrelevant	precision
LinkedList	317	11	1	96%
OrderedList	201	5	5	95%
StackLi	184	8	1	95%
StackAr	159	0	0	100%
QueueAr	500	0	0	100%
ListNode	46	1	1	95%
LinkedListItr	185	8	0	95%

Redundant (implied) invariant detection was not implemented for this experiment

Experiments on data structures

class	relevant	missing	recall
LinkedList	317	3	99%
OrderedList	201	5	98%
StackLi	184	0	100%
StackAr	159	0	100%
QueueAr	500	10	98%
ListNode	46	0	100%
LinkedListItr	185	2	99%

These are all textbook data structures, so we know the exact set of invariants

Sample invariants

```
void LinkedList.insert(Object x, LinkedListItr p):::EXIT
  x = header.next.element
  if (p != null && p.current != null)
    then size(orig(header.next.closure(next))) = size(header.next.closure(next)) - 1
    else header.closure(next) = orig(header.closure(next))

boolean LinkedList.isEmpty():::EXIT
  if (header.next == null)
    then return = true
    else return = false

void LinkedList.remove(Object x):::EXIT
  size(header.next.closure(next)) <= size(orig(header.next.closure(next)))
  MISSING: if (findPrevious(s) != null)
    then size(header.next.closure(next)) = size(orig(header.next.closure(next))) - 1
    else size(header.next.closure(next)) = size(orig(header.next.closure(next)))
```

Final words

- Experiments show the invariants are
  - Accurate, useful, and efficient to generate
- Scalability
  - Even relatively small test suites are enough for detecting good invariants
- Ease of use
  - Still uses a lot of memory (internal data structures)
  - Usability can be improved (to cope with all generated invariants)

Static summary computation

- Summaries specify how a procedure behaves
  - Computed in the Alloy language
  - Are safe (sound) abstractions
    - It is guaranteed that the summaries account for all procedure executions

- This summarization technique
  - Is fully automatic
  - Requires absolutely no annotations/guidance/additional information from the user

- Goal: cost-effectiveness
  - The summarization must be fast
    - Is used as a small phase of a bigger bug finding technique
    - Is linear in the size of the code
  - Accuracy is not so important
    - Produces as accurate summaries as possible using a lightweight technique

Syntactic summaries

- Summarize the behavior of a procedure as a symbolic relationship between pre and post states
- Summaries are declarative formulas in a subset of Alloy
  - Doesn't include quantifiers
  - Doesn't include set comprehension

- Provide both an upper and a lower bound on the final values of fields, return value, and allocated objects

relational expr ⊆ field'/variable' ⊆ relational expr

- The result can sometimes be precise

field'/variable' = relational expr

Example – Precise Spec

```
Job nullifyMove(Entry e1, Entry e2) {
    e1.job = e2.job;
    e2.job = null;
    return e1.job;
}
```

Example – Precise Spec

```
Job nullifyMove(Entry e1, Entry e2) {
    e1.job = e2.job;
    e2.job = null;
    return e1.job;
}
```

The summary is correct even when e1 and e2 are aliased.

relational override

job' = job ++ (e1 → e2.job) ++ (e2 → null)

$ret = e1.(job ++ (e1 → e2.job) ++ (e2 → null))

Example – Imprecise Spec

In a list of jobs, returns the first one with n predecessors

```
Entry findFirst(Entry e, int n) {
  Entry c = e;
  while ((c != null) && (c.job.predsNum != n)) {
     c = c.next;
  }
  return c;
}
```

Return value is reachable from e

$\$ret \subseteq$ e.*next

$\$ret \supseteq \varnothing$

Example – Imprecise Spec

c.f != d

```
Entry findFirst(Entry e, int n) {
  Entry c = e;
  while ((c != null) && (c.job. predsNum != n)) {
     c = c.next;
  }
  return c;
}
```

Return value is reachable from e
And it is either null, or its f field equals d

$ret $\subseteq$ (e.*next & (null + f.d))

$ret $\supseteq \varnothing$

Example – Imprecise Spec

```
Entry findFirst(Entry e, int n) {
  Entry c = e;
  while ((c != null) && (c.job. predsNum != n)) {
     c = c.next;
  }
  return c;
}
```

Return value is reachable from e
And it is either null, or its
job.predsNum field equals n

$\$ret \subseteq (e.{*}next \ \& \ (null + job.predsNum.n))$

$\$ret \supseteq \emptyset$

Why is this imprecise?

Example – Imprecise Spec

```
void JobList.init() {
 Entry c = this.head;
 while (c != null) {
    c.job.scheduled = 0;
    c = c.next;
 }
}
```

•The scheduled field of any job reachable from this list may be changed to 0

•The scheduled field of all other jobs remain unchanged

scheduled’ $\subseteq$ scheduled + (this.head.*next.job $\rightarrow$ 0)

scheduled’ $\supseteq$ scheduled - (this.head.*next.job $\rightarrow$ univ)

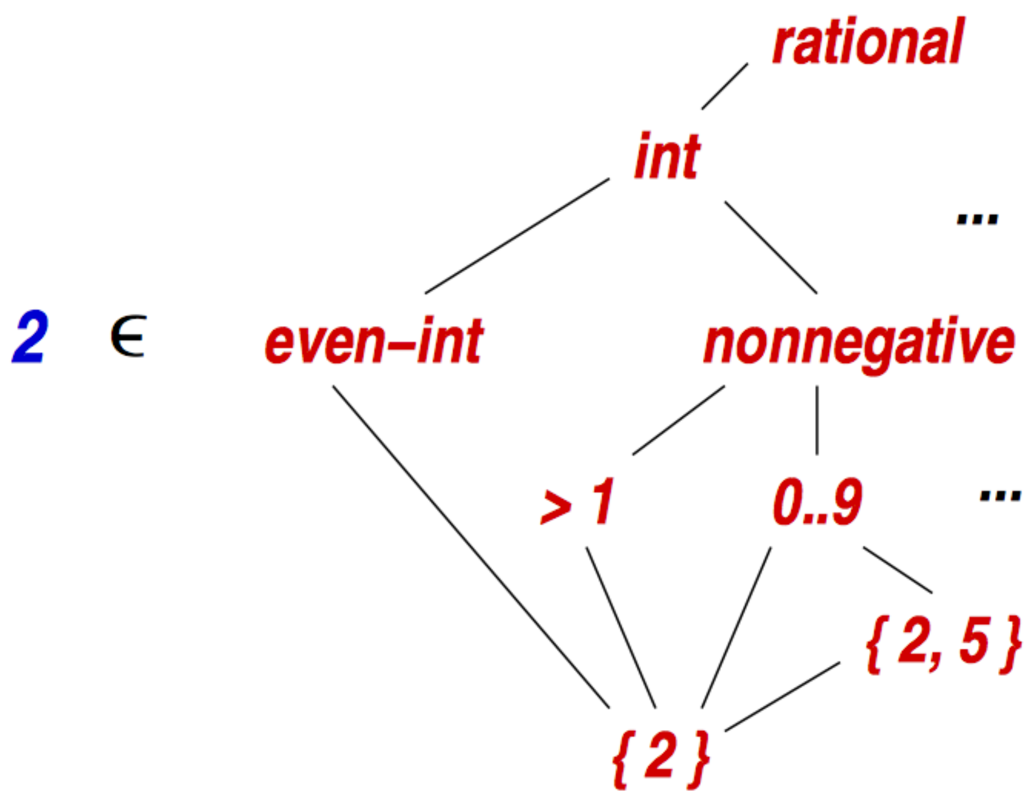
Approach: abstract interpretation

- Study aspects of the concrete (but more complicated) executions by looking at corresponding properties of abstract (and simpler) executions

- Example: for the abstract domain of $\{(+), (-), (+/-)\}$
  - $-1515 * 17 \Rightarrow -(+) * (+) \Rightarrow (-) * (+) \Rightarrow (-)$
  - $-1515 + 17 \Rightarrow -(+) + (+) \Rightarrow (-) + (+) \Rightarrow (+/-)$

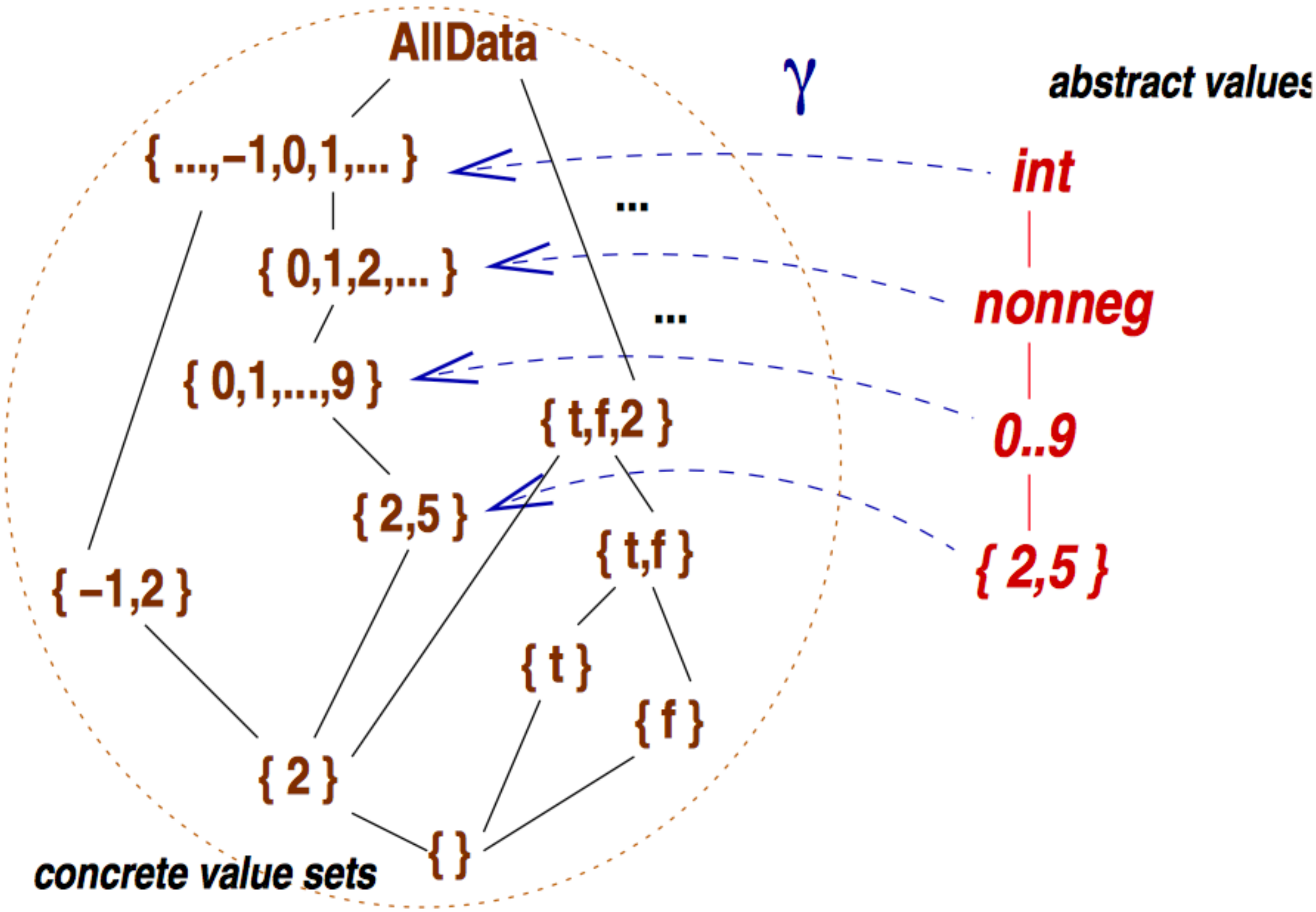
- Abstract interpretation approximates program behavior by replacing the concrete domain of computation and its concrete operations with an abstract domain and abstract operations.

Value abstractions

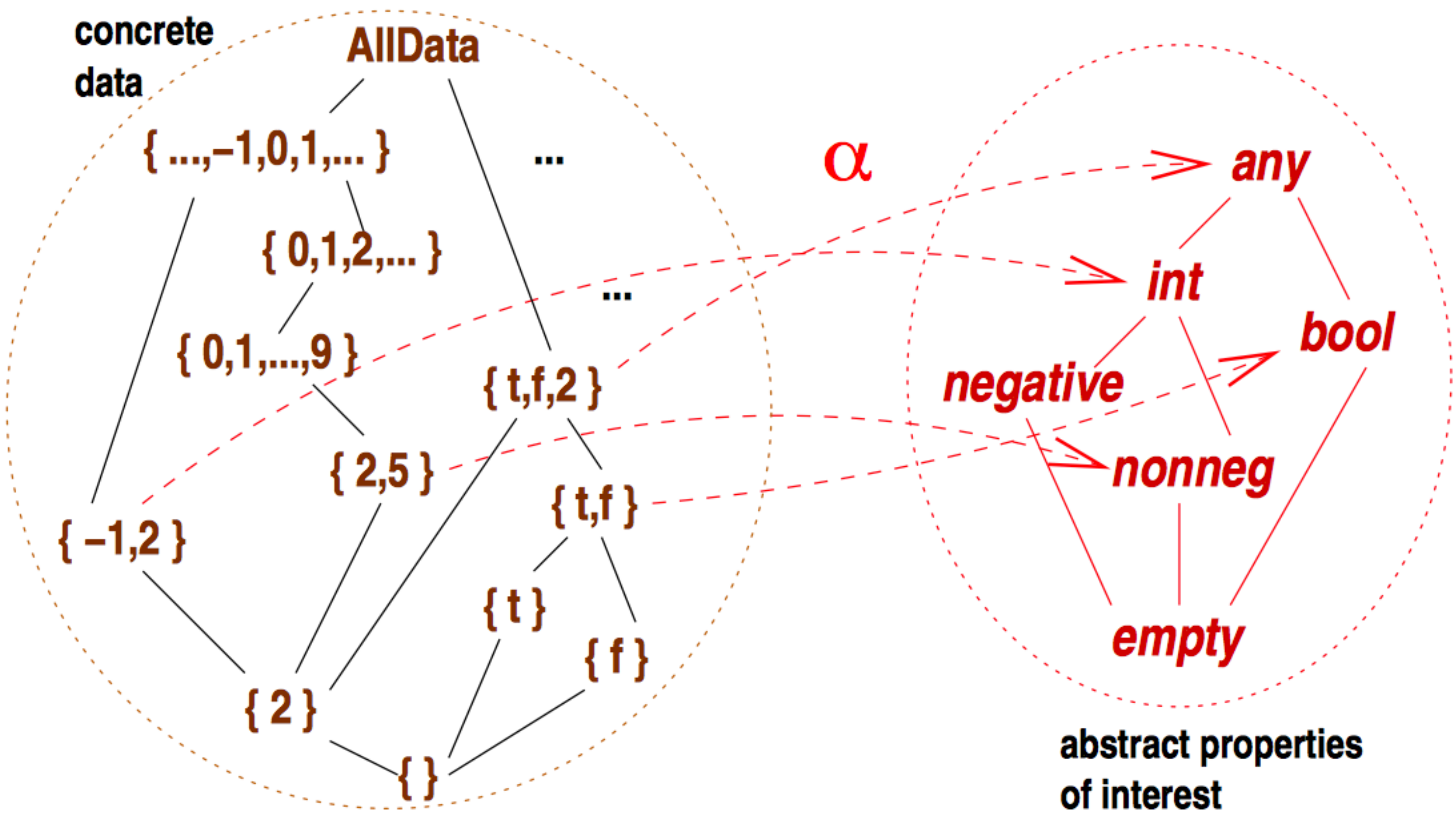


All the properties listed on the right are abstractions of 2; the upwards lines denote $\sqsubseteq$, a loss of precision.

Concretization function



Abstraction function



Function α maps each set to the abstract value that best describes it.

Abstract interpretation

- Consists of
  - A concrete domain S, and an abstract domain A
  - An abstraction function alpha, and a concretization function gamma
  - Alpha and gamma form a Galois connection
    - S \in gamma(alpha(S))
    - A = alpha(gamma(A))
- Is defined over an ordered set (lattice)
  - a partially ordered set where any two elements have a unique least upper bound (join) and a greatest lower bound (meet)
  - Examples:
    - The set {1, 2, 3} and the subset relation
      - Bounded with a bottom and a top
    - The set of natural numbers and the less-than relation
      - Unbounded with a bottom

Technique: Abstract Interpretation

Abstract domain = <Lower bound, Upper bound>

E^{ℓ}, E^{u}: (Var ∪ Field ∪ Type) → Relational Expr

Partial order

$$\langle E^{\ell}_1, E^{u}_1\rangle \sqsubseteq \langle E^{\ell}_2, E^{u}_2\rangle \Leftrightarrow$$
$$\forall x, (E^{\ell}_1(x) \supseteq E^{\ell}_2(x)) \wedge (E^{u}_1(x) \subseteq E^{u}_2(x))$$

Lattice join

$$\langle E^{\ell}_1, E^{u}_1\rangle \sqcup \langle E^{\ell}_2, E^{u}_2\rangle = \langle \lambda x.\ E^{\ell}_1(x)\ \&\ E^{\ell}_2(x),$$
$$\lambda x.\ E^{u}_1(x) + E^{u}_2(x) \rangle$$

Approach

- Is an abstract interpretation
  - Flow sensitive (order of statements is important)
  - Context sensitive (calling context is important)
- Abstract domain
  - Relational expressions in Alloy
- Symbolic execution
  - Pre-state
    - Each type, variable, field is represented by a relation constant
  - Execution
    - Keeps two expressions for each type, variable, field:
      - Lower bound: tuples that occur in all executions (must side-effects)
      - Upper bound: tuples that occur in some executions (may side-efffects)
    - As the code updates the values, the relational expressions are updated
  - Final summary is an over-approximation of the behavior

(lb **in** x’) **and** (x’ in ub)

Example

```
1. x.f = y;
2. if (c == 1)
3.    z.g = y;
4. else
5.    z.g = x;
6. end
```

0. x, y, z, f, g, c
1. $[f] = f \mathbin{++} x \rightarrow y$

3. $[g] = g \mathbin{++} z \rightarrow y$

5. $[g] = g \mathbin{++} z \rightarrow x$
6. $[g] \subseteq g + z \rightarrow y + z \rightarrow x$
6. $[g] \supseteq g - z \rightarrow \mathrm{univ}$

Program constructs

- Object allocation
  - Allocated objects are represented by fresh unary relations
- Call sites
  - Context-sensitive summaries
  - Context consists of variables, fields, types whose summaries are accurate
  - Generate a template summary for each context of a procedure using symbolic constants for accessed fields, variables, types
  - Instantiate it at each call site using relational expressions of fields, variables, and types at that point
- Loops
  - Use the loops condition to get more precise abstractions of the body
    - At the entry point, the relational encoding of the condition is intersected with expressions for the condition’s variables to remove all tuples that violate the condition
    - Then abstract the body by computing fixpoint
    - Then intersect the negation of loop condition with the final values of condition’s variables

Technique: Abstract Interpretation

Widenings:

- x + x.r + .. + $x.r^{(k)}$ in upper bound goes to x.*r
  x & x.r & .. & $x.r^{(k)}$ in lower bound goes to {}
- upper bound with more than n operators goes to univ
  lower bound with more than n operators goes to {}
- $(m+1)^{th}$ allocation of a type goes to
  a symbolic set of objects with unspecified cardinality
- simplification rules to shorten the exprs