Abstract Interpretation

Simple Language (from the programming projects)

<expr> ::= **const** | **id** [only consider integer vars/consts; in the project also do float]

| <expr> **+** <expr> | <expr> **-** <expr>

| <expr> ***** <expr> | <expr> **/** <expr>

| **(** <expr> **)**

<cond> ::= **true** | **false** | <expr> **<** <expr> [also **<=**, **>**, **>=**, ==, **!=**]

| <cond> **&&** <cond> | <cond> **||** <cond>

| **!** <cond> | **(** <cond> **)**

Abstract Memory State (we will just say “Abstract State”)

Abstract state: a map $\boldsymbol{\sigma}_a$ from vars to abstract values

A summarization of many possible concrete states

$\boldsymbol{\sigma}_a$: Vars $\rightarrow$ { *Neg, Zero, Pos, AnyInt* }

Vars is the set of all variable names in the program

In a concrete (non-abstract) state σ we map to { *0, -1, 1, -2, 2, …* }

Here we use an abstraction of this set of concrete values

$\sigma_a(\text{id})$ = *Neg*: represents all concrete states with $\sigma(\text{id}) < 0$

$\sigma_a(\text{id})$ = *Zero*: represents all concrete states with $\sigma(\text{id}) = 0$

$\sigma_a(\text{id})$ = *Pos*: represents all concrete states with $\sigma(\text{id}) > 0$

$\sigma_a(\text{id})$ = *AnyInt*: represents all concrete states

For illustration, we will use this abstraction to prove the absence of “division by zero” errors statically, without running the program

Abstract Evaluation

Abstract evaluation relation for arithmetic expressions: triples <**ae**, σ_a> → v_a

- **ae** is a parse subtree derived from <expr>
- σ_a is an abstract state
- v_a is an abstract value $\in$ { *Neg*, *Zero*, *Pos*, *AnyInt* }

Meaning of <**ae**, σ_a> → v_a : the evaluation of **ae** from any concrete state σ abstracted by σ_a, if it completes successfully, will produce a concrete value *v* abstracted by v_a

Example: <**x+y+1**, [x↦*Pos*, y↦*Pos*]> → *Pos*

Example: <**x*y-1**, [x↦*Zero*, y↦*Pos*]> → *Neg*

Evaluation for Arithmetic Expressions

Syntax: **id** | **const** | <expr> **+** <expr> | …

$$\boxed{\langle \textbf{const}, \sigma_a \rangle \rightarrow \mathit{Pos}}$$

if const.lexval is a positive integer; similarly for *Zero* and *Neg*

$$\boxed{\langle \textbf{id}, \sigma_a \rangle \rightarrow \sigma_a(\mathrm{id})}$$

static error if σ_a(id) is undefined; use of uninitialized variable

$$\boxed{\frac{\langle ae_1, \sigma_a \rangle \rightarrow v_{a1} \quad \langle ae_2, \sigma_a \rangle \rightarrow v_{a2}}{\langle ae_1 + ae_2, \sigma_a \rangle \rightarrow v_a}}$$

$$v_a = v_{a1} +_a v_{a2}$$

Here we use abstract addition operator $+_a$ working on abstract values

Evaluation for Arithmetic Expressions

$+_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*Neg*	*Neg*	*AnyInt*	*AnyInt*
Zero		*Zero*	*Pos*	*AnyInt*
Pos			*Pos*	*AnyInt*
AnyInt				*AnyInt*

second operand

first operand

$-_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg				
Zero				
Pos				
AnyInt				

Let’s try this first ourselves; don’t look at next slide yet

Evaluation for Arithmetic Expressions

$+_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*Neg*	*Neg*	*AnyInt*	*AnyInt*
Zero		*Zero*	*Pos*	*AnyInt*
Pos			*Pos*	*AnyInt*
AnyInt				*AnyInt*

second operand

first operand

$-_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*AnyInt*	*Neg*	*Neg*	*AnyInt*
Zero	*Pos*	*Zero*	*Neg*	*AnyInt*
Pos	*Pos*	*Pos*	*AnyInt*	*AnyInt*
AnyInt	*AnyInt*	*AnyInt*	*AnyInt*	*AnyInt*

Evaluation for Arithmetic Expressions

Let’s try this first ourselves; don’t look at next slide yet

$*_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg				
Zero				
Pos				
AnyInt				

second operand

first operand

$/_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg				
Zero				
Pos				
AnyInt				

Semantics of concrete /: treat as reals, then round toward zero

Evaluation for Arithmetic Expressions

$*_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*Pos*	*Zero*	*Neg*	*AnyInt*
Zero		*Zero*	*Zero*	*Zero*
Pos			*Pos*	*AnyInt*
AnyInt				*AnyInt*

second operand

first operand

$/_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*AnyInt*	⚠	*AnyInt*	⚠
Zero	*Zero*	⚠	*Zero*	⚠
Pos	*AnyInt*	⚠	*AnyInt*	⚠
AnyInt	*AnyInt*	⚠	*AnyInt*	⚠

Example:
Pos $/_a$ Pos
5 / 3 = 1
2/ 3 = 0
To represent both outcomes we use *AnyInt*

⚠ Abstract operation is undefined: cannot guarantee the absence of run-time division-by-zero error

Integers vs Floats

second operand

first operand

$/_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*AnyInt*	⚠	*AnyInt*	⚠
Zero	*Zero*	⚠	*Zero*	⚠
Pos	*AnyInt*	⚠	*AnyInt*	⚠
AnyInt	*AnyInt*	⚠	*AnyInt*	⚠

second operand

first operand

$/_a$	*Neg*	*Zero*	*Pos*	*AnyFloat*
Neg	*Pos*	⚠	*Neg*	⚠
Zero	*Zero*	⚠	*Zero*	⚠
Pos	*Neg*	⚠	*Pos*	⚠
AnyFloat	*AnyFloat*	⚠	*AnyFloat*	⚠

Division Example

int x = 3; [x↦*Pos*]
int z = -x; [x↦*Pos*, z↦*Neg*]
int w = x - z + 5; [x↦*Pos*, z↦*Neg*, w↦*Pos*]
w = w / x; [x↦*Pos*, z↦*Neg*, w↦*AnyInt*]
x = x / w; static checking error: w may be 0 [but not really…]

We could choose to be less conservative: only complain if we are sure that the second operand is zero

second operand

first operand

$/_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*AnyInt*	⚠	*AnyInt*	*AnyInt*
Zero	*Zero*	⚠	*Zero*	*Zero*
Pos	*AnyInt*	⚠	*AnyInt*	*AnyInt*
AnyInt	*AnyInt*	⚠	*AnyInt*	*AnyInt*

Integers vs Floats: Less Conservative

second operand

first operand

$/_a$	Neg	Zero	Pos	AnyInt
Neg	AnyInt	⚠	AnyInt	AnyInt
Zero	Zero	⚠	Zero	Zero
Pos	AnyInt	⚠	AnyInt	AnyInt
AnyInt	AnyInt	⚠	AnyInt	AnyInt

second operand

first operand

$/_a$	Neg	Zero	Pos	AnyFloat
Neg	Pos	⚠	Neg	AnyFloat
Zero	Zero	⚠	Zero	Zero
Pos	Neg	⚠	Pos	AnyFloat
AnyFloat	AnyFloat	⚠	AnyFloat	AnyFloat

Trade-Offs in Algorithm Design

More conservative version: if it does not report an error, we are guaranteed that every execution will not have div-by-zero error

Less conservative version: if it does report an error, we are guaranteed that every execution will have an div-by-zero error

- This will avoid false warnings, but will also miss some programs with run-time div-by-zero errors

This is an example of a typical trade-off in the design of static checking algorithms

Evaluation for Boolean Expressions

<cond> ::= **true** | **false** | <expr> **<** <expr> [also **<=**, **>**, **>=**, **==**, **!=**]

| <cond> **&&** <cond> | <cond> **||** <cond>

| **!** <cond> | **(** <cond> **)**

Concrete: <**be**, σ> → *v*

v is a value from *{ true, false }*

Abstract: <**be**, σ_a> → *AnyBool*

For now, keep it simple: statically, assume that at run time, both *true* and *false* are possible. Do not look inside these expressions and do not check. We will revisit this later.

Statements: $<s, \sigma_a> \rightarrow \sigma'_a$

$$\boxed{<\textbf{skip}, \sigma_a> \rightarrow \sigma_a}$$

$$\frac{<ae, \sigma_a> \rightarrow v_a}{<\textbf{id=}ae, \sigma_a> \rightarrow \sigma_a[id \mapsto v_a]}$$

$$\frac{<s_1, \sigma_a> \rightarrow \sigma_{a1} \quad <s_2, \sigma_a> \rightarrow \sigma_{a2}}{<\textbf{if (}be\textbf{)}\ s_1\ \textbf{else}\ s_2, \sigma_a> \rightarrow \sigma'_a} \qquad \sigma'_a = merge(\sigma_{a1},\sigma_{a2})$$

$$\frac{<s_1, \sigma_a> \rightarrow \sigma_{a1}}{<\textbf{if (}be\textbf{)}\ s_1, \sigma_a> \rightarrow \sigma'_a} \qquad \sigma'_a = merge(\sigma_a,\sigma_{a1})$$

Merging of Abstract States

Do it variable-by-variable:

1) If the variable is defined in both abstract states: use this table

merge	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*Neg*	*AnyInt*	*AnyInt*	*AnyInt*
Zero		*Zero*	*AnyInt*	*AnyInt*
Pos			*Pos*	*AnyInt*
AnyInt				*AnyInt*

2) If the variable is defined in only one abstract state: undefined in the merged state [this will allow us to catch uninitialized variables; details later]
3) If the variable is undefined in both abstract states: remains undefined in the merged state

Example of Merging

merge	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*Neg*	*AnyInt*	*AnyInt*	*AnyInt*
Zero		*Zero*	*AnyInt*	*AnyInt*
Pos			*Pos*	*AnyInt*
AnyInt				*AnyInt*

	resulting state:
x = 1;	[x↦*Pos*]
y = -2;	[x↦*Pos*, y↦*Neg*]
if (…)	
z = x+1;	[x↦*Pos*, y↦*Neg, z*↦*Pos*]
else	
z = x-y;	[x↦*Pos*, y↦*Neg, z*↦*Pos*]
	[x↦*Pos*, y↦*Neg, z*↦*Pos*]

Example of Merging

merge	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*Neg*	*AnyInt*	*AnyInt*	*AnyInt*
Zero		*Zero*	*AnyInt*	*AnyInt*
Pos			*Pos*	*AnyInt*
AnyInt				*AnyInt*

	resulting state:
x = 1;	[x↦*Pos*]
y = -2;	[x↦*Pos*, y↦*Neg*]
if (...)	
z = x+1;	[x↦*Pos*, y↦*Neg*, *z*↦*Pos*]
else	
z = x+y;	[x↦*Pos*, y↦*Neg*, *z*↦*AnyInt*]
	[x↦*Pos*, y↦*Neg*, *z*↦*AnyInt*]

Loops: < **while (be)** s, σ_a >→ σ'_a

We abstract the loop condition as "don't know; could be true or false''; need to consider all possible executions of the loop

0 iterations: $\sigma'_a = \sigma_a$

1 iteration: if <s, σ_a> → σ_{a1} , then $\sigma'_a = \sigma_{a1}$

2 iterations: if <s, σ_{a1}> → σ_{a2}, then $\sigma'_a = \sigma_{a2}$ and so on

σ'_a = merge(σ_a, σ_{a1}, σ_{a2}, σ_{a3}, …): infinite number of σ_{ak}

But: σ'_a can be computed in a finite number of steps

$\sigma'_{a0} = \sigma_a$

σ'_{a1} = merge(σ'_{a0}, σ_{a1})

σ'_{a2} = merge(σ'_{a1}, σ_{a2})

σ'_{a3} = merge(σ'_{a2}, σ_{a3}) and so on

This converges: after a while we have $\sigma'_{ak} = \sigma'_{a(k-1)}$ [details omitted]

Interpreter for the Abstract Semantics

If we implement an interpreter, we get a static checker for division-by-zero errors and use-before-initialization errors

Code implementation (e.g., for the programming projects)

```
AbstractValue abs_eval(TreeNode n, AbstractState s) { …
  if (n is a plus expression) return
     abs_plus(abs_eval(left subexpr, s), abs_eval(right subexpr, s));
}
```

or, in a more object-oriented style

```
class BinaryExpr {
  AbstractValue abs_eval(AbstractState s) { …
    if (this is a plus expression)
      return abs_plus(expr1.abs_eval(s), expr2.abs_eval(s));
  }
}
```

Interpreter for the Abstract Semantics

Code implementation for if-then-else

```
abs_exec(TreeNode n, AbstractState s) {
  AbstractState s2 = s.copy(); // create a new table and copy all data
  abs_exec(then part, s); // updates s
  abs_exec(else part, s2); // updates s2
  abs_merge(s, s2); // merge s2 into s; changes s
}
```

Code implementation for while loop

```
AbstractState abs_exec(TreeNode n, AbstractState as) {
  // in a loop, abs_exec(body, current state) and
  // merge the current state $\sigma_{ak}$ into the result state $\sigma'_{ak}$. stop after
  // convergence is seen with $\sigma'_{ak} = \sigma'_{a(k-1)}$ and then return $\sigma'_{ak}$
}
```

Project 4

Goal: modify Project 3 to do checking for possible division by zero and use of uninitialized vars [but not inside conditional expressions; will do in Project 5]

Code changes will be minimal: if you have code to do real interpretation, it is not hard to change it to do abstract interpretation

- State now contains abstract values
- Expressions are evaluated using the abstract operators
- If-then, if-then-else, and while-loops are changed as described in the last few slides

Project 4

Implementation detail: Integers vs Floats

Use more refined versions of the abstract values: set *{ NegInt, ZeroInt, PosInt, AnyInt, NegFloat, ZeroFloat, PosFloat, AnyFloat }*

- Easier to handle division (has different semantics for ints vs floats)
- Printing for testing/debugging/grading: print one of those 8 strings [e.g., **not** Neg, NEG, Neg_Int, NEGINT, … **but exactly** NegInt]

Printing:

- For statement **print expr;** abstractly evaluate the expression and then println its abstract value [one of those 8 strings]
- Do not print the program
- Do not print the abstract state

Project 4

Static checking

Check 1: division by zero – report error if the second operand of division is *ZeroInt*, *AnyInt*, *ZeroFloat*, *AnyFloat*
[this is the "more conservative" approach from earlier; could result in false warnings a.k.a. false positives]

Check 2: use of uninitialized variable – error if a variable is used in an expression but there is no value for the variable in the abstract state

Uninitialized Variables

Example 1:

int x; int y = x;

when we try abs_eval(x) we will not find x in state

Example 2:

int x; if (…) { x = 1; } else {x = -2; } int y = x;

state after true branch [x↦*Pos*], state after false branch [x↦*Neg*], state after merge [x↦*AnyInt*], checking is fine for **int y = x;**

Example 3:

int x; int z = 2; if (…) { x = 1; } else { … } int y = x;

state after true branch [x↦*Pos*, z↦*Pos*], state after false branch [z↦*Pos*], state after merge [z↦*Pos*], error for **int y = x;**

Todo: at home, try an example with a while-loop

Project 5: Boolean Expressions Refined

<cond> ::= **true** | **false** | <expr> **<** <expr> [also **<=**, **>**, **>=**, ==, **!=**]

| <cond> **&&** <cond> | <cond> **||** <cond>

| **!** <cond> | **(** <cond> **)**

Abstract: <**be**, σ_a> → v_a where $v_a \in$ { *True, False, AnyBool* }

$\&\&_a$	*True*	*False*	*AnyBool*
True	*True*	*False*	*AnyBool*
False		*False*	*False*
AnyBool			*AnyBool*

Similarly for || and !

Short-Circuit Evaluation

Abstract: <**be**, σ_a> → v_a where $v_a \in$ { *True, False, AnyBool* }

Our abstract evaluation should "simulate" what happens in concrete evaluations. For example, consider **&&**

Case 1: first operand evaluates to *True* [i.e., in all concrete executions, the first operand will evaluate to true and the second operand will definitely be evaluated]. So, in Project 5, evaluate the second operand and use its value as the result of &&

Case 2: first operand evaluates to *False* [i.e., in all concrete executions, the first operand will evaluate to false and the second operand will definitely **not** be evaluated]. So, in Project 5, do **not** evaluate the second operand and just produce *False*

Short-Circuit Evaluation

Case 3: first operand evaluates to *AnyBool* [i.e., in some concrete executions, the first operand could possibly evaluate to true and in those cases the second operand will be evaluated]. So, in Project 5, evaluate abstractly the second operand and then produce *AnyBool* $\mathbf{\&\&}_{a}$ that value

Operator ||: do something similar, but suitable for OR

Comparisons

<cond> ::= … | <expr> **<** <expr> [also **<=**, **>**, **>=**, **==**, **!=**]

$<_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*AnyBool*	*True*	*True*	*AnyBool*
Zero	*False*	*False*	*True*	*AnyBool*
Pos	*False*	*False*	*AnyBool*	*AnyBool*
AnyInt	*AnyBool*	*AnyBool*	*AnyBool*	*AnyBool*

$==_a$	*Neg*	*Zero*	*Pos*	*AnyInt*
Neg	*AnyBool*	*False*	*False*	*AnyBool*
Zero		*True*	*False*	*AnyBool*
Pos			*AnyBool*	*AnyBool*
AnyInt				*AnyBool*

In reality, will have comparisons for { *NegInt*, *ZeroInt*, *PosInt*, *AnyInt*} and separately for { *NegFloat*, *ZeroFloat*, *PosFloat*, *AnyFloat*}, since we assume that the input program successfully passed typechecking

If-Then-Else with Dead Code Errors

Code implementation for if-then-else

```
abs_exec(TreeNode n, AbstractState s) {
  AbstractBool cond = abs_eval(condition, s);
  // case 1: statically guaranteed to be true; else part is dead code
  if (cond == True) { terminate with static error (dead code) }
  // case 2: statically guaranteed to be false; then part is dead code
  if (cond == False) { terminate with static error (dead code) }
  // case 3: do not know statically because cond == AnyBool
  AbstractState s2 = s.copy();
  abs_exec(then part, s); // updates s
  abs_exec(else part, s2); // updates s2
  abs_merge(s, s2); // merge s2 into s; changes s
}
```

If-Then with Dead Code Error

Code implementation for if-then

```
abs_exec(TreeNode n, AbstractState s) {
  AbstractBool cond = abs_eval(condition, s);
  // case 1: statically guaranteed to be false; then part is dead code
  if (cond == False) { terminate with static error (dead code) }
  // case 2: statically guaranteed to be true; no merging needed
  if (cond == True) { abs_exec(then part, s); return; }
  // case 3: do not know statically because cond == AnyBool
  AbstractState s2 = s.copy();
  abs_exec(then part, s2); // updates s2
  abs_merge(s, s2); // merge s2 into s; changes s
}
```

While-Do with Dead Code Error

Code implementation for while-do loop

```
abs_exec(TreeNode n, AbstractState s) {
  AbstractBool cond = abs_eval(condition, s);
  // case 1: statically guaranteed to be false; loop body is dead code
  if (cond == False) { terminate with static error (dead code) }
  // case 2: statically guaranteed to be true; at least one iteration;
  // modify the state for one iteration and continue with general case
  if (cond == True) { abs_exec(loop body, s); }
  // general case: process exactly how you did in Project 4;
  // do not re-evaluate the loop condition
  …
}
```

Why Case 2?

Example:

int x; int y = 0;

while (y < 100) { x = y; y = y+1; }

print x;

Project 3: successful completion, x is 99

Project 4: x is uninitialized in the abstract state immediately before the loop; the analysis thinks that there could be zero iterations of the loop and complains about uninitialized x at the print

Project 5: since *ZeroInt* $<_a$ *PosInt*, case 2 applies. The state is changed to include an initial value for x. Then the loop is processed as usual, starting from that modified state. The final value for x is *AnyInt*.

Project 5 without case 2: same as Project 4.

Food for thought: what would happen if we had **int y = 1;** instead?

Uninitialized Variables

Example 1:

```
int x; int y = x;
```

when we try abs_eval(x) we will not find x in state

Example 2:

```
int x = readint; int z; if (x > 0) { … } else { z = 1; } int y = z;
```

error reported for uninitialized z

Example 3:

```
int p = 1; int q = 2; int z; if (p<q) { z = 3; } else { … } int y = z;
```

error reported for uninitialized z, but at run time there is no error

Example 4:

```
int p = readint; int q = 1; int r; while(q<p) { q = q+1; r = q*q; }  int y = r;
```

Uninitialized Variables in Java

Example 1: `int x; int y = x;`

Result: both our checker and javac complain

Example 2:

```
int x = readint; int z;
if (x > 0) { x = 0; } else { z = 1; }
int y = z;
```

vs.

```
int x = (new Scanner(System.in)).nextInt();
int z;
if (x > 0) { x = 0; } else { z = 1; }
int y = z;
```

Result: both our checker and javac complain

Uninitialized Variables in Java

Example 3:

```
int x = 1; int y = 2; int z = x + y;
int w;
if (z > 0) { w = 3; }
int v = w;
```

Result: javac incorrectly complains; our checker correctly accepts

Example 4: let us add “final” (write-once) in the Java code

```
final int x = 1; final int y = 2; final int z = x + y;
int w;
if (z > 0) { w = 3; }
int v = w;
```

Result: javac correctly accepts; our checker correctly accepts

Uninitialized Variables in Java

Example 5:

```
final int x = 1; final int y = 2; int z = x + y; int w;
if (z > 0) { w = 3; }
int v = w;
int p; z = -z;
if (z > 0) { p = 4; }
int q = p;
```

Result: javac complains about **w** and **p**; our checker complains about **p**

Example 6:

```
final int x = 1; final int y = 2; int w;
if (x < y) { w = 3; }
int v = w;
```

Result: javac correctly accepts; our checker incorrectly complains