Midterm

- Grades and solutions are (and have been) on Moodle
- The midterm was hard[er than I thought]
  - grades will be scaled
- I gave everyone a 10 bonus point
  (already included in your total)

max:	98
mean:	71
min:	45
standard deviation:	13

I will pass graded midterms back at end of today's class

Projects, etc.

- Thank you for the project updates
- Everyone who submitted should have gotten a response
- If you didn't submit, why not?
- Final report due Dec 7, 11:59 PM
- Homework 3 is up, due Nov 29

questions?

Path-Based Static Analysis

Static analysis we know

- We've looked at static (and dynamic) analysis that:
  - identifies invariants
  - describes a method's effect
  - maps inputs to outputs

Example

```
int increment(int num) {
  print(num);
  print("Have a nice day");
  return num+1;
}
```

What can we tell, statically, about the method's effects?

return value > num
return value 1 more than num

Dynamic analysis too

- Daikon can tell you (sometimes complex) relationships between variables
- Temporal relationships are also possible
  for example:
  - .close() is always preceded by .open()
  - .close() is never followed by .open()

Problems with dynamic analysis

- Unsound: A property is not guaranteed to be true
  - .close() is never followed by .open():
  maybe we simply never say an .open() after a .close()
- Incomplete: We may never observe some property
  - If we never see a .open(), how can we know that must be followed by .close()?

Static analysis

- Can static analysis alleviate these problems?
- Is static analysis sound?
- Is static analysis complete?
- Well, maybe. But it's hard!
  - summaries can be hard to compute
  - analysis must account for all paths through the method
  - summary language generally must be very expressive

Another approach: path-based

- An alternative to summaries is to perform path-based analysis
- Analyze just one path through the method at a time
- This approach is conceptually simpler
  - and often simpler to implement

Example

```
void myRead(File f) throws BadException {
  if (f.exists()) {
    f.open();
    print(f.readLine());
  } else {
    throw new BadException("f does not exist");
}
```

What can we tell, statically, about when the exception is thrown?

Only if f.exists() == false

Larger example

```
void myRead(File f) throws BadException {
  if (today() == day.MONDAY) {
    if (f.exists()) {
      f.open();
      print(f.readLine());
    } else {
      throw new BadException("f !exist");a
  } else {
    print("fake line");
  }
}
```

Issues

- There can be a lot of paths:
  n conditionals → up to 2^n paths
- There can be A LOT of paths:
  loops, recursive functions, etc.
- Let's ignore these issues for now (just for now)

Finite State Properties

- Let's use FSMs to describe (specify) a class:
- Two states: Open and Closed
- An Open file can be closed
- A Closed file can be opened
- Other transitions are errors

First, simple algorithm

For each path:

track the transitions and states through the FSM

Example with simple algorithm

assume we start in Close

```
void myRead(boolean dump, File f) {
  int x = 1;
  if (dump) {   // explore TRUE branch
    x = 0;
    f.open();
    f.write(DATA);
  }
  if (dump && x==1)   // explore TRUE branch
    f.close();
}
```

What went wrong?

assume we start in Close

```
void myRead(boolean dump, File f) {
  int x = 1;
  if (dump) {   // explore TRUE branch
    x = 0;
    f.open();
    f.write(DATA);
  }
  if (dump && x==1)   // explore TRUE branch
    f.close();
}
```

This path is not possible!

Second algorithm

- Keep track of the branch decisions on paths
- Create an "abstract state," which is a combination: <file state, predicate>
  - predicate is a conjunction of all the branch conditions observed on the path
- If the predicate is false, we know the path is impossible

Example with second algorithm

assume we start in Close

```
void myRead(boolean dump, File f) {
  int x = 1;
  if (dump) {   // explore TRUE branch
    x = 0;
    f.open();
    f.write(DATA);
  }
  if (dump && x==1)   // explore TRUE branch
    f.close();
}
```

Still not enough!

- Keeping track of just predicates, can eliminate some bad paths.

What paths can we eliminate?

assume we start in Close

```
void myRead(boolean dump, File f) {
  int x = 1;
  if (dump) {   // explore TRUE branch
    x = 0;
    f.open();
    f.write(DATA);
  }
  if (!dump && x==1) // explore TRUE branch
    f.close();
}
```

Still not enough!

- Keeping track of just predicates, can eliminate some bad paths.
- To eliminate more, we need to keep track of relevant variable values.

Third algorithm

- Examine all branch predicates and keep track of all variables in those predicates
  - dump, x
- Keep track of the branch decisions on paths
- Create an "abstract state," which is a combination: <file state, larger predicate>
  - larger predicate is a conjunction of all the branch conditions observed on the path with variables' values
- If the predicate is false, we know the path is impossible

Example with third algorithm

assume we start in Close

```
void myRead(boolean dump, File f) {
  int x = 1;
  if (dump) {   // If we explore TRUE branch here
    x = 0;
    f.open();
    f.write(DATA);
  }
  if (dump && x==1)   // we won't explore TRUE branch here
    f.close();
}
```

In practice

- This can actually work
  - except those unresolved issues with loops and recursion
- Requires:
  - A theorem prover: something that can deduce whether a predicate is false
  - A way of accurately modeling branch predicates
    - A hard problem in general. Why?
      - because branch predicates can be arbitrary code and we know arbitrary code can be undecidable!
    - But many predicates are easy in practice

So does this really work?

- For very small programs, sure.
- But for large program, there are simply too many paths
- So in practice, this approach has not scaled. The exponential blow up in paths does not allow applying this to large programs.

Where does it work?

- Single method analysis
- Small class analysis
- Small modules?

The program can be large, but if you analyze small modules, it can be helpful.

Can we do better?

- If we only care about a particular property,
  - such as can open be followed by open
- Then many paths may be irrelevant

```
void tests(int x, int y) {
  if (x == 5) x++; else --x;
  if (y == 6) new File().open();
    else new File().close();
}
```

Do we care about value of x and its predicates?

Key question

- So we want a compromise:

  naïve approach was not enough,
  but keeping track of all predicates was too much

- How can we model only the predicates relevant to the property we care about?

Idea

- Give up on analyzing one path at a time
- Instead, analyze all paths at once
- When paths split, keep track of them all
- When paths join
  - join all abstract states with the same information
  - this limits the number of possible abstract states by the number of FSM states
- In other words, keep track of the predicates, but now we'll have AND and OR of the predicates

Why does it work

- In essence, we are trying to note relevant correlations between predicates and states

```
void method() {
  if (q) flag = 1;
  else flag = 0;
  …
  if (q) …
  else …
}
```

common pattern, as are more elaborate variations

OK, back to loops and recursion

- Consider the following example

```
foo(x, y) {
  if (x == 0) return; open(y);
  close(y);
  foo(x-1, y);
}
```

Recursive constraints

- Like any static analysis, recursion and looping introduces recursive constraints
- If we have an initial estimate of what to track, we can iteratively improve it
- Typically, each time around the loop will not add a new constraint. There is a finite number of constraints, and the solution space is finite.

What else is hard with path analysis?

- Aliasing is two variables pointing to the same object
- Aliasing can be very tricky

```
void method(boolean b) {
  if (b) … else …
  d = b;
  if (d) … else …
}
```

What if d = function(b)?
Could be anything

Another aliasing example

```
void method() {
  File f = new File(PATH);
  File myFile = f;
  myFile.open();
  List l = new LinkedList();
  while (l.isEmpty())
    l.add(myFile);
  File g = (File) l.get(0);
  g.close();
}
```

Is f open or closed at the end?

What if you have multiple values

- For example, suppose we are dealing with 3 files, all at once.
- One solution is to run our analysis 3 times, once per each file.
- Have to resolve which aliases map to that file.
- Must compute all predicate information for those aliases.

ESP

- Error Detection via Scalable Program Analysis
- Sound: everything it returns is true
- Incomplete: won't return all true things
- Verified file handling in gcc:
  - 140K lines of code
  - 600+ file manipulation calls
- Advantage: strong guarantee
  - Not "I didn't find any bugs," but
  - Proof that the program will always correctly handle files, regardless of input

http://www.microsoft.com/windows/cse/pa_projects.mspx

ESP experience

- Was originally a university research project
- Went on to become a production tool within Microsoft
- Used on many core Windows projects
- Very successful

- But used mostly as a "bug finder," not prover
- Reason: alias analysis was not precise enough to limit mistakes on truly large programs

ESP is simple, but…

- Even simpler than we discussed:
  - very simple model of program state
  - only reasons about paths

- But, the complexity is hidden:
  - theorem prover
  - alias analysis

- Also, requires the entire program and cannot be used on a module in isolation

Summary

- ESP can prove the absence of certain types of bugs in a program:
  - for example, closing a closed file
- Recursion, large number of paths, aliasing make the problem very complex
- Successful tool, used in industry at Microsoft

Midterm

- Grades and solutions are (and have been) on Moodle
- The midterm was hard[er than I thought]
  - grades will be scaled
- I gave everyone a 10 bonus point

  (already included in your total)

max:	98
mean:	71
min:	45
standard deviation:	13

I will pass back graded midterms now