CSCI 1951Q

Topics in Programming Languages: Program Analysis

Example #1: Type checking

Typescript

```
interface Printable {
    print(): void;
}

class A implements Printable {
    print() { console.log("A") }
}

class B {}


function printList<T extends Printable>(l: T[]) {
    for (let t of l) {
        t.print();
        t.send();
    }
}

printList([new A()])
printList([new B()])
```

Example #2: Type inference

Basic inference in C++

```cpp
std::vector<std::string> v = {"a", "b"};
                    auto v = {"a", "b"};
```

Polymorphic inference in OCaml

```ocaml
let apply_left f (x, y) =
  (f x, y)

val apply_left :
   ('a → 'b) →
    'a * 'c  →
    'b * 'c
```

Flow-sensitive inference in Typescript

```typescript
function unwrap<T>(
  x: {success: T} | {error: string}
): T {
    if ("error" in x) {
        // x: {error: string}
        throw new Error(x.error);
    } else {
        // x: {success: T}
        return x.success;
    }
}
```

Example #3: Pointer analysis

Ownership in Rust

```rust
let mut v = vec![1, 2, 3];
let n = &v[0];
v.push(0);
println!("{n}");
```

```
error[E0502]: cannot
borrow `v` as mutable
because it is also
borrowed as immutable
```

Bug-finding in Java w/ Infer

```java
class C {
  int getLength(
    @Nullable String s
  ) {
    return s.length();
  }
}
```

```
error: object s is
annotated with @Nullable
and is dereferenced
without a null check
```

Example #4: Dataflow optimizations

```rust
#[unsafe(no_mangle)]
pub fn sum_two_A(
  v: &[i32]
) -> i32 {
  assert!(v.len() > 1);
  let x = v[0];
  let y = v[1];
  x + y
}

#[unsafe(no_mangle)]
pub fn sum_two_B(
  v: &[i32]
) -> i32 {
  let x = v[0];
  let y = v[1];
  x + y
}
```

```asm
sum_two_A:
        cmp     rsi, 1
        jbe     .PANIC
        mov     eax, dword ptr [rdi + 4]
        add     eax, dword ptr [rdi]
        ret
.PANIC:
        ; ... panic code ..

sum_two_B:
        push    rax
        cmp     rsi, 1
        je      .PANIC_1
        test    rsi, rsi
        je      .PANIC_2
        mov     eax, dword ptr [rdi + 4]
        add     eax, dword ptr [rdi]
        pop     rcx
        ret
.PANIC_1:
        ; ... panic code ...
.PANIC_2:
        ; ... panic code ...
```

https://godbolt.org/z/e6ehGY93Y

Example #5: Coverage-guided fuzzing

```rust
let path = args().skip(1).next().unwrap();
let contents = fs::read_to_string(&path).unwrap();
if let Some("a") = contents.get(0..1) {
  if let Some("b") == contents.get(1..2) {
    if let Some("c") = contents.get(2..3) {
      panic!("Invariant violated!");
    }
  }
}
```

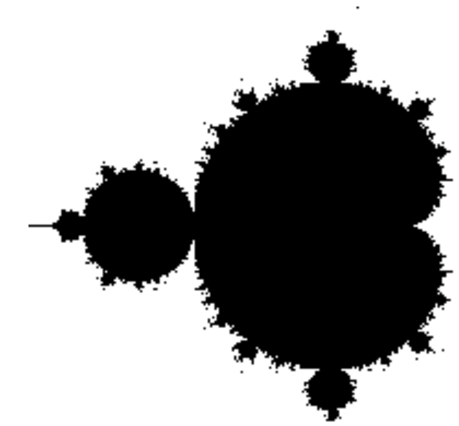
```
cargo afl fuzz -i in -o out target/release/example @@
```

```
    american fuzzy lop ++4.33c {default} (target/release/foobar) [explore]
┌─ process timing ────────────────────────────────────┬─ overall results ───┐
│        run time : 0 days, 0 hrs, 0 min, 14 sec      │  cycles done : 18   │
│   last new find : 0 days, 0 hrs, 0 min, 12 sec      │ corpus count : 8    │
│last saved crash : 0 days, 0 hrs, 0 min, 2 sec       │saved crashes : 2    │
│ last saved hang : none seen yet                     │  saved hangs : 0    │
├─ cycle progress ──────────────────┬─ map coverage───┴─────────────────────┤
│  now processing : 6.26 (75.0%)    │    map density : 26.61% / 38.71%      │
│  runs timed out : 0 (0.00%)       │ count coverage : 1.85 bits/tuple      │
├─ stage progress ──────────────────┼─ findings in depth ───────────────────┤
│  now trying : havoc               │ favored items : 8 (100.00%)           │
│ stage execs : 132/400 (33.00%)    │  new edges on : 8 (100.00%)           │
│ total execs : 31.5k               │ total crashes : 44 (2 saved)          │
│  exec speed : 2223/sec            │  total tmouts : 0 (0 saved)           │
```

crash.txt:

```
xyz
```

Example #6: JIT compilation



```
cargo build --release
time (./target/release/mandelbrot 16000 > /dev/null)
3.38s user 0.03s system 461% cpu 0.737 total
```

```
time (java -Xint Mandelbrot 16000 > /dev/null)
279.99s user 1.19s system 947% cpu 29.683 total

time (java Mandelbrot 16000 > /dev/null)
10.83s user 0.07s system 918% cpu 1.187 total
```

```
time (node --jitless mandelbrot.js 16000 > /dev/null)
414.00s user 4.47s system 867% cpu 48.237 total

time (node mandelbrot.js 16000 > /dev/null)
13.21s user 0.11s system 929% cpu 1.433 total
```

Example #7: Parallelization & Differentiation

```python
def mean(X):
    acc = 0
    for n in X:
        acc += n
    return acc / len(X)
```

```python
def means(X):
    acc = []
    for row in X:
        acc.append(mean(row))
    return acc
```

```python
mean = lambda X: X.sum() / len(X)
means = jit(lambda X: jnp.stack([mean(row) for row in X]))
means.trace(X).lower().as_text()
```

```
module @jit__lambda_ attributes {mhlo.num_partitions = 1 : i32, mhlo.num_replicas = 1 : i32} {
  func.func public @main(%arg0: tensor<3x3xf32>) -> (tensor<3xf32> {jax.result_info = ""}) {
    %0 = stablehlo.slice %arg0 [0:1, 0:3] : (tensor<3x3xf32>) -> tensor<1x3xf32>
    %1 = stablehlo.reshape %0 : (tensor<1x3xf32>) -> tensor<3xf32>
    ...
```

```python
loss = lambda X: means(X).sum()
grad(loss)(X)
```

```
Array([[0.33333334, 0.33333334, 0.33333334],
       ...
```

Program analysis is about algorithms to answer fundamental questions about program behavior.

Does this program have undefined behavior?

When is this data no longer needed?

What kind of values can this variable take?

What’s the hottest code path in my program?

Program analysis draw

Complexity theory
Which analyses are feasible?

Lar
Whi

Systems programming
Which analyses make fast code?

So
Wh

Human factors
Which analyses are usa

Meta 3.8 ★
$323K - $522K/yr Total Pay
$405K Median
Compiler Engineer

Apple 4.1 ★
$253K - $421K/yr Total Pay
$323K Median
Compiler Engineer

NVIDIA 4.6 ★
$238K - $371K/yr Total Pay
$295K Median
Compiler Engineer

The computability theory foundations of program analysis

A refresher on the halting problem

Let $M(D)$ be a Turing machine that take a description of another Turing machine D as input.

M cannot have the following behavior:

$M(D)$ outputs true if D halts on all inputs, and false otherwise.

Informal proof: let F be the Turing machine which computes $M(F)$ and loops forever if $M(F)$ says F halts.

If $M(F)$ says F halts, then F does not halt.
If $M(F)$ says F does not halt, then F halts.

Therefore, M cannot exist.

The halting problem is <u>undecidable</u>*.

*There does not exist an algorithm which can compute a correct answer.

Rice's theorem generalizes the halting problem

Let $M(D)$ be a Turing machine that take a description of another Turing machine D as input.

Let P be a **non-trivial** and **extensional** property of a Turing machine.

M cannot have the following behavior:

$M(D)$ outputs true if $P(D)$, and false otherwise.

non-trivial: there are machines both with and without P
(trivial: P = all machines)

extensional: P does not distinguish between machines with identical input/output behavior.
(non-extensional: P = the set of machines with 3 states)

Informal proof of Rice's theorem by reduction to the halting problem

Assume $R(D)$ decides whether D satisfies P. We want to decide whether D halts.

Let F be a Turing machine which satisfies P.

Construct the machine $D'(x)$ which calls $D(x)$ and returns $F(x)$.

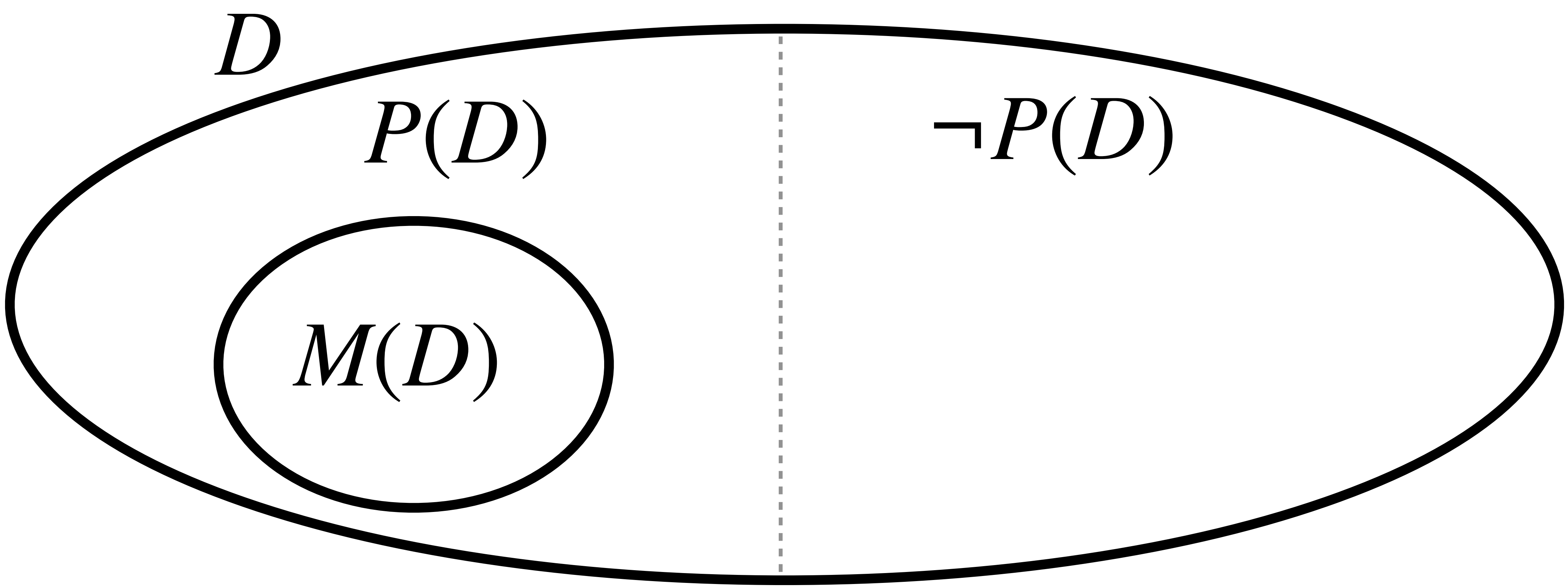
Then $R(D')$ outputs true iff D halts.

Every interesting program property is undecidable!

...and yet...?

Option 1: Approximate analysis

Design algorithms that are wrong in a predictable way



$$M(D) \implies P(D)$$

Example: $P(D)$ = D halts.

M: true = D definitely halts
false = D might halt

Example: $P(D)$ = D has no undefined behavior.

M: true = D definitely no UB
false = D maybe UB

Option 2: Dynamic analysis

Make educated guesses based on past program executions

now

Time

Does this variable always have the same type?	Assume it always will and produce specialized code w/ guards
Is this virtual method call always going to the same method?	Assume it always will and use a direct method call w/ guards
Has this input covered new branches?	Assume it will find more and keep modifying it

Option 3: Domain-specific analysis

Rice's theorem is only true for Turing-complete languages

No halting problem if you don't have general recursion!

OCaml:

```
let rec f = fun (s : string) -> f s in
f "Hello"
```

[infinite loop]

Dhall:

```
let f = \(s : Text) -> f s in
f "Hello"
```

```
Error: Unbound variable: f
```

Option 3: Domain-specific analysis

Rice's theorem is only true for Turing-complete languages

```sql
SELECT * FROM orders
WHERE total_amount > 100
  AND customer_id = 123
```

✅ →

```sql
SELECT * FROM orders
WHERE customer_id = 123
  AND total_amount > 100
```

```python
orders = [
  row for row in orders
  if row.customer_id == 123 \
  and row.total_amount > 100]
```

❌ →

```python
orders = [
  row for row in orders
  if row.total_amount > 100 \
  and row.customer_id == 123]
```

```python
@dataclass
class Row:
  customer_id: Id
  total_amount: float
```

```python
@dataclass
class Id:
  id: int

  def __eq__(self, value):
    print("I'm being equated!!")
    return self.id == int(value)
```

Course info

cel.cs.brown.edu/csci-1951q-f25/