CSC 6580
Spring 2020

Instructor: Stacy Prowell

Decidability (Again)

Hailstone Numbers

Given integer n_0>0, let a sequence of integers be defined as follows.

$$\begin{cases} n_{i+1} = 3n_i + 1 & \text{if } n_i \text{ is odd} \\ n_{i+1} = n_i/2 & \text{if } n_i \text{ is even} \end{cases}$$

This sequence terminates when $n_i = 1$.

The Collatz Conjecture: This sequence terminates for any positive integer n_0.

The number sequence is known as the "hailstone numbers."

```
def hailstone(start: int):
    print(start, end='')
    while start != 1:
        print(', ', end='')
        if start & 1 == 1:
            start = start * 3 + 1
        else:
            start //= 2
        print(start, end='')
```

```
>>> hailstone(23)
23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5,
16, 8, 4, 2, 1
```

Hailstone Numbers

The C program at right *terminates* iff the input is a terminating hailstone number. Otherwise it does *not terminate*.

It is *not known* if the Collatz conjecture is true. It is an open question.

Is termination of this program *decidable*?

```
#include <stdint.h>

void hailstone(uint64_t start) {
    while (start != 1) {
        if (start & 1) {
            start = start * 3 + 1;
        } else {
            start = start / 2;
        }
    }
}
```

Hailstone Numbers

Here is a test harness for the function.

Estimated time for various limits, *assuming* all values terminate!

All `uint16_t`: instant
All `uint32_t`: 47 minutes
All `uint64_t`: 393,000 years

But what if some do not terminate? How would we know? Keep waiting?

```
#include <stdint.h>

#define LIMIT 0xffff

extern void hailstone(uint64_t);

int main(int argc, char * argv[]) {
    for (uint64_t start = 1;
         start < LIMIT;
         ++start) {
        hailstone(start);
    }
}
```

Hailstone Numbers

It's decidable.

```c
#include <stdint.h>

#define LIMIT 0xffff

extern void hailstone(uint64_t);

int main(int argc, char * argv[]) {
    for (uint64_t start = 1;
         start < LIMIT;
         ++start) {
        hailstone(start);
    }
}
```

Hailstone Numbers

It's decidable.

This is basically a (very large) finite state machine.

In any sufficiently long sequence we will either (1) reach one and terminate, or (2) repeat a number and we will know this does not terminate. So with sufficient memory (a lot!) and sufficient time (a lot!) we can always decide.

```
#include <stdint.h>

#define LIMIT 0xffff

extern void hailstone(uint64_t);

int main(int argc, char * argv[]) {
    for (uint64_t start = 1;
         start < LIMIT;
         ++start) {
        hailstone(start);
    }
}
```

Why is this relevant?

Most things about most programs are decidable.

It is *hard* to write a concrete program whose termination is undecidable, and not just hard to figure out or unknown. Give it a try!

Even in the extreme example of the Collatz Conjecture, whose answer is *unknown* we are still decidable, even if the general Collatz Conjecture is *unprovable* or even *provably undecidable*.

A (Ridiculously) Short Introduction to the Very Important Technique of Program Slicing*

* Abridged version

A Program

Here's a short program.

Under what conditions does it terminate?

We can start to figure that out by constructing the *control slice.*

We pick a point in the program and a value (or values) we want, and then discard anything that does not affect those values at that point.

```
start:
  xor    eax, eax
  mov    rbx, START
.loop:
  add    rbx, STRIDE
  inc    rax
  cmp    rbx, END
  jne    .loop
  ret
```

A Program

If this sounds familiar, it has a lot in common with live variable analysis.

We want to know *when this program terminates*.

It terminates when we reach the return.

We reach the return when the branch is not taken.

```
start:
  xor     eax, eax
  mov     rbx, START
.loop:
  add     rbx, STRIDE
  inc     rax
  cmp     rbx, END
  jne     .loop            ; ZF must be set
  ret
```

A Program

Need to know when RBX is equal to END, so we only care about RBX. We select the statements that affect RBX.

This is like live variable analysis with only RAX live.

```
start:
  xor     eax, eax
  mov     rbx, START          ; needed
.loop:
  add     rbx, STRIDE         ; needed
  inc     rax
  cmp     rbx, END            ; rbx must be END
  jne     .loop               ; ZF must be set
  ret
```

A Program

These are the instructions that affect the program's control flow (which is all we care about). This is called the **control slice**.

```
  mov     rbx, START
.loop:
  add     rbx, STRIDE
  cmp     rbx, END
  jne     .loop
```

A Program

Let's simplify the variables.

We need to know when RBX = E.

Initially RBX = T. Assume the loop runs *n* times, then we add *n*S to RBX, but we do this for a 64-bit register. So we have:

$T + nS \cong E \pmod{2^{64}}$

```
  mov     rbx, T
.loop:
  add     rbx, S
  cmp     rbx, E
  jne     .loop
```

A Program

$T + nS = E \pmod{2^{64}}$

We know T, S, and E. If we can find an *n* that makes the above true, then the loop halts. If we cannot, then the loop doesn't halt.

We have the following.

$2^{64} \mid T + nS - E$

```
  mov    rbx, T
.loop:
  add    rbx, S
  cmp    rbx, E
  jne    .loop
```

A Program

$2^{64} \mid T + nS - E$

Is there an *n* that *satisfies* this expression? Better yet, is there a program that can find it for us?

```
  mov    rbx, T
.loop:
  add    rbx, S
  cmp    rbx, E
  jne    .loop
```

Satisfiability

Satisfiability

Given a *formula* is there some *model* (assignment of values to variables) that makes the formula *true*? If so, the formula is **satisfiable**. Otherwise it is not. (If all models make the formula true, the formula is **valid**.)

In propositional logic (variables, truth constants, and logical connectives but no quantifiers) satisfiability is *decidable*.

In first-order (predicate) logic (quantifiers) satisfiability is *not decidable*.

$\neg p \vee p$	$p \wedge \neg p$	$p \implies q$
valid	invalid / not satisfiable	satisfiable

Satisfiability

Boolean satisfiability (SAT) is the *first* problem proven to be NP-complete.

Check it by plugging in the answer. But finding the answer might be really hard...

SAT Solvers

Convert a Boolean expression to conjunctive normal form (CNF).

$$\neg p \lor (\neg q \land r) \Longrightarrow (\neg p \lor \neg q) \land (\neg p \lor r)$$

Now try to find an assignment that makes all the individual clauses true.

There can be multiple solutions.

$$\begin{gathered} p = \mathrm{T} \land q = \mathrm{F} \land r = \mathrm{T} \\ p = \mathrm{F} \land q = \mathrm{F} \land r = \mathrm{F} \\ p = \mathrm{F} \land q = \mathrm{T} \land r = \mathrm{F} \\ \vdots \end{gathered}$$

Which one a SAT solver chooses can be *random*, and vary from run to run. Some allow specifying a random seed so the result is repeatable.

Remember: They give you *a* solution, not necessarily *the* solution.

SAT Solvers

Lots of them. Some well-known ones are:

- MiniSat http://minisat.se/
- Glucose https://www.labri.fr/perso/lsimon/glucose/
- Lingeling http://fmv.jku.at/lingeling/

Also: PySAT https://pysathq.github.io/ provides a Python interface to a lot of different SAT solvers… the above three included.

Satisfiability Modulo Theories

Integers, Bit Arrays, and Other Stuff

Boolean satisfiability is find and all, but how do we use this to prove things about programs?

Well, start with a Boolean satisfiability problem.

$$(\neg p \vee \neg q) \wedge (\neg p \vee r)$$

Now replace some of the variables with expressions about integers.

$$(\neg(x \geq 15)) \vee \neg q) \wedge (\neg(x \geq 15)) \vee r)$$

These additions are called *background theories* and we can think of the result as a *tiered* satisfiability problem. Solve the outer problem, then this constrains the inner problems: *satisfiability modulo theories*.

Theories

Some theories are decidable.

- Pressburger Arithmetic

Some are not.

- Peano Arithmetic

Many are very useful for CS.

- Arrays, bit vectors, lists, ...

Z3

Z3

```
$ pip3 install z3-solver
```

Z3 is an open source SMT solver developed by Microsoft.

It has a nice Python interface.

https://ericpony.github.io/z3py-tutorial/guide-examples.htm

Z3

Let’s try a simple problem in Z3.

$$\neg p \vee (\neg q \wedge r)$$

```
from z3 import *
p = Bool(‘p’)
q = Bool(‘q’)
r = Bool(‘r’)
s = Solver()
s.add(Or(Not(p), And(Not(q), r)))
print(s.check())
print(s.model())
```

Z3

Let's try a simple problem in Z3.

$$\neg p \vee (\neg q \wedge r)$$

```
[p = False, q = False, r = True]
```

```
from z3 import *
p = Bool('p')
q = Bool('q')
r = Bool('r')
s = Solver()
s.add(Or(Not(p), And(Not(q), r)))
print(s.check())
print(s.model())
```

SMT-LIB

There is a standard language for SMT solvers, called SMT-LIB.

Work with Z3 using SMT-LIB online:
https://rise4fun.com/Z3

Find a tutorial guide here:
https://rise4fun.com/z3/tutorial

```
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)
(assert (or (not p) (and (not q) r)))
(check-sat)
(get-model)
```

Solving Termination with Z3

Solving

Here is an SMT-LIB program in Z3 for solving our earlier termination problem.

Recall that we need an *n* such that:

$2^{64} \mid T + nS - E$

Trying this when T = 7, S = 4, and E = 22… we find the model is *unsatisfiable*. The program runs forever.

```
(declare-fun n () Int)
(declare-fun T () Int)
(declare-fun S () Int)
(declare-fun E () Int)

; Our constraints for a solution.
(assert (= T 7))
(assert (= S 4))
(assert (= E 22))
(assert (> n 0))
(assert
  (= 0
    (mod
      (+ T (* n S) (- E)) (^ 2 64))))

; Solve
(check-sat)
(get-model)
```

Solving

Trying this again, but changing `E` = 23… we find the model is now *satisfiable*. But (when writing these slides) the value for *n* shows:

4,611,686,018,427,387,908

That's… awesome and all, but what about 4?

7 + 4*4 = 23

Hey, it found *a* solution. You need to add other constraints if you want the *first* value.

```
from z3 import *
n = Int('n')
T = Int('T')
S = Int('S')
E = Int('E')
s = Solver()
s.add(T == 7)
s.add(S == 4)
s.add(E == 23)
s.add(n > 0)
s.add(0 == ((T + n*S - E) % 2**64))
print(s.check())
print(s.model())
```

TERMINATOR

Byron Cook created a tool called TERMINATOR to prove termination of programs.

We can “reduce” lots of problems to termination. For instance, if the correct password is entered, the program terminates; if a particular string is printed, the program terminates; etc.

TERMINATOR tried to find a well-founded relation on variables of interest to prove termination--just like we did earlier.

There is a YouTube interview with him about this:
https://channel9.msdn.com/Shows/Going+Deep/Byron-Cook-Inside-Terminator

This is now part of T2: http://mmjb.github.io/T2/

Concolic Execution

Concrete Execution

How long will it take to find the error in testing if we just generate random inputs? This is *concrete execution.*

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

Symbolic Execution

We can *symbolically execute* the program to find the error.

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

Symbolic Execution

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

```
z = 2*y
x == 100000:
  100000 < 2*y  →  error
x != 100000:
  pass
```

This can be very hard and may take a long time for large programs.

Concolic Execution

1. Identify a set of variables as *inputs*.
2. Instrument the program to *trace* modifications to inputs or control flow changes.
3. Pick some initial input *values*.
4. Do:
    a. Execute the program on the input values.
    b. Symbolically re-execute the program *on the trace* and generate *constraints* on inputs and path conditions.
    c. Negate the last path condition (not already negated) to explore (potentially) an alternate path.
    d. Call a satisfiability solver on the constraints to generate a new input values. If not satisfied, go back and negate another path condition. If no more, halt.

Instrumentation? Pin

Pin supports dynamic instrumentation of binaries on Linux, Windows, and macOS.

https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

You *do not need* to re-compile the target program to instrument it; Pin is essentially Just-in-Time (JIT) instrumentation.

Concolic Execution

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

- The inputs are `x` and `y`
- Instrument the code
- Let `x = y = 1`.
- Execute `f(1,1)`
- The branch is not taken (`x != 100000`)

Concolic Execution

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

- Now symbolically execute
- Set `z = 2*y`
- Path condition is `x != 100000`

Concolic Execution

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

- Negate last path condition: `not(x != 100000)`
- Call SMT solver to get a valid `x` and `y`
- Get `x = 100000`, `y = 1`
- Execute `f(100000,1)`
- The outer branch is taken, but the inner branch is not

Concolic Execution

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

- Now symbolically execute
- Set `z = 2*y`
- Path conditions are now `x == 100000` and `x >= z`, which becomes `x >= 2*y`

Concolic Execution

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

- Negate last path condition:
  `not(x >= 2*y)`
- Run SMT solver to get valid `x` and `y`
- Get `x = 100000`, `y = 500001`
- Execute `f(100000,500001)`
- Both the outer and inner branch are now taken

Concolic Execution with Pin and Z3

https://triton.quarkslab.com/

Next time:
Last class!
Discussion of Final Exam