Lecture 25 : Predicate abstraction, contd.

Converting C code to Boolean programs

Ex

```
int main() {
    int x, y;
    y=1;
    x=1;
    if (y>x)
        y--;
    else
        y++;
    assert(y>x);
}
```

```
int main() {
bool b0 := *;
b0 := *;
b0 := *;
if ( b0 ) {
    b0 := *
} else {
    b0 := *
}
assert ( b0 )
}
```

We don't know if x was initialized with a value ≤1 or >1.

b_0 tracks whether y>x

b_0 can be initialized to either true / false because of undefined semantics in C.

Easy question: what to fill in here?

Question: What is the impact of the statement $y--$ on the indicator variable b_0 which tracks $y > x$ if b_0 was originally true?

assume $(y > x)$ $\{y > x\}$

$y := y - 1$

$\{$ $y \geq x$ $\}$ ← ① What is the strongest postcondition?

$y \geq x \iff \underline{y+1 > x}$

② If $y \geq x$, then can b_0 be true? ✓

③ If $y \geq x$, then can b_0 be false? ✓

④ Effectively, the statement $y--$ under the condition b_0 causes an update $b_0 = *$.

Question: What is the impact of the statement

y++ on the indicator variable b_0 which

tracks $y > x$ if b_0 was originally false?

Answer: $b_0 = *$

Question: What is the impact of the statement $x := 1$

on the indicator variable b_0?

Answer: $b_0 = *$

Objection! Oh, but we know that $y = 1$

↑

But the analysis algorithm doesn't know it.

Follow up question

Question: What is the impact of the statement `y++` on the indicator variable b_0 which tracks $y > x$ if b_0 was originally false?

Answer: $b_0 = *$ ← How does get an SMT solver to come up with this?

$b_0 = \text{true}$

$b_0 = \text{false}$

Two easier questions:

Q1. Is it possible for b_0' (value after update) to be true?

Ask the SMT solver: "Are there old values of x & y such that $\underbrace{y \not> x}_{(b_0 = \text{false})}$ & such that $\underbrace{y+1 > x}_{(b_0' = \text{true})}$?"

$\exists x . \exists y . \; y \not> x$ and $y+1 > x$?

Z3 will say: Yes.

Q2: Is it possible for b_0' to be false?

$\exists x. \exists y.$ $\underbrace{y \not> x}_{b_0 = \text{false}}$ $\underbrace{y+1 \not> x}_{b_0' = \text{false}}$

Z3 will say: yes!

Thus, we update b_0 as $b_0 := \overbrace{\text{true} \mid \text{false}}^{*}$

Dogma: What makes verification hard is:

1. Unbounded loops, &
2. Unbounded data.

← If all loops bounded, then only finitely many paths through program.

If variables stuck to values 1, 2, .. 10

& n variables, k lines of code,

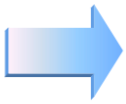
your state space contains only $\underbrace{k \cdot 10^n}$ elements.

Note: The worry is not that this path will crash. It is that this path

Question

```
int main() {
    int x, y;
    y=1;
    x=1;
    if (y>x)
        y--;
    else
        y++;
    assert(y>x);
}
```

Predicate: $y>x$



```
main() {
    bool b0; // y>x
    b0=*;
    b0=*;
    if (b0)
        b0=*;
    else
        b0=*;
    assert(b0);
}
```

rash.

this path might so crash.

How to automate?

(1) Collect the path predicates.

No. The original program never goes into the then branch.

$y_1 = 1$ ← "First" version of the variable y.

$x_1 = 1$

assume $(y_1 > x_1)$

$y_2 = y_1 - 1$ ← "Second version" of the variable y

assert $(y_2 \not> x_1)$

(2) We ask z3: $\exists x_1 \exists y_1 \exists y_2$. s.t. $y_1 = 1$ and $x_1 = 1$ and $y_1 > x_1$ and $y_2 = y_1 - 1$ and $y_2 \not> x_1$?

z3 says: No!

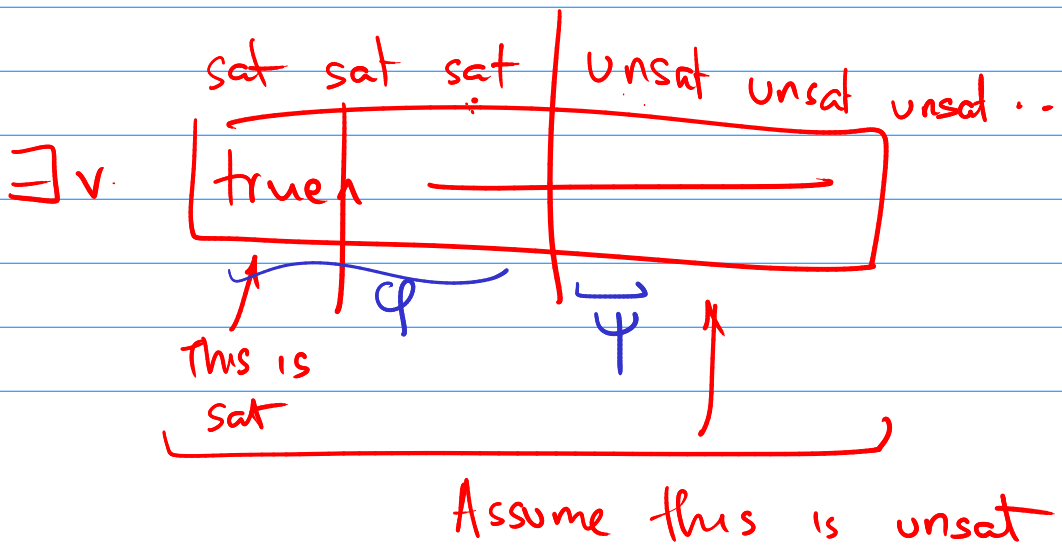
In general, to check feasibility of a path, $\pi = c_1; c_2; c_3; \ldots c_n$

we ask: $\exists \vec{v}$ $\quad true \wedge a_1 \wedge a_2 \wedge a_3 \wedge \ldots \wedge a_n$

$\vec{v}$: vector of variables. (SSA)

If whole constraint is infeasible, then look at shortest infeasible prefix.

The path is feasible iff the SMT solver says satisfiable



Craig's Interpolation Theorem

For all formulas φ, ψ in FOL

If $\varphi \wedge \psi$ is unsat

then $\exists$ a formula χ ← called an "interpolant"

s.t. ① $\varphi \Rightarrow \chi$

② $\chi \wedge \psi$ is unsat

→ Prove that $\varphi \wedge \psi$ is unsat

③ χ is only made of symbols common to φ & ψ.

Hopefully χ is much smaller than φ.

Longest feasible prefix φ

Ex $\exists x_1, y_1, y_2 \cdot$ s.t. $y_1 = 1$ and $x_1 = 1$ and $y_1 > x_1$ and $y_2 = y_1 - 1$ and $y_2 \not> x_1$

ψ

Here, the interpolant ψ is just φ itself

Short infeasible prefix

```
int main() {
    int x, y;
    y=1;
    x=1;
    if (y>x)
        y--;
    else
        y++;
    assert(y>x);
}
```

```
main() {
    bool b0; // y>x
    b0=*;
    b0=*;
    if (b0)
        b0=*;
    else
        b0=*;
    assert(b0);
}
```

φ

Predicate: $y>x$

ψ

Conceptually if $(b_0 \wedge b_x)$

Shortest infeasible prefix

Longest feasible prefix

(Conditions apply) Because $\varphi \Rightarrow x$ Now, at this point b_x = true

High level question: What is it about φ that makes $\varphi \wedge \psi$ infeasible?

Answer: Interpolant χ (fn of both φ, ψ)

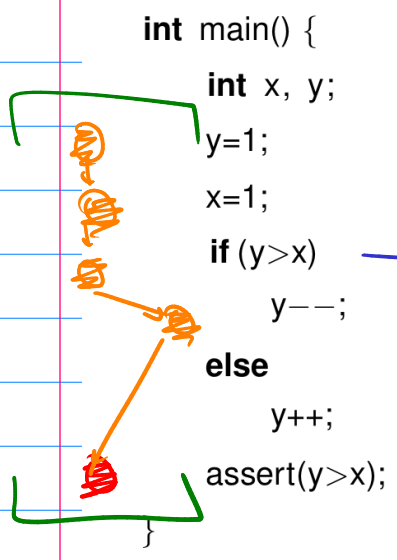
Now, add b_χ to the abstraction.

Solution: Instead of just tracking

$b_0 \equiv$ "$y > x$",

also track χ = "$x=1$ and $y=1$".

$b_1 \equiv$ "$x=1$" $b_2 \equiv$ "$y=1$"



$b_0 := *, b_1 := *, b_2 := x$

$b_0 := *, b_1 := *, b_2 := \text{true}$

$b_0 := \text{false}, b_1 := \text{true}, b_2 := \text{true}$

if (b_0) {

$b_0 := \_\_\_, b_1 := \_\_\_, b_2 := \_\_\_$

} else {

$b_0 := \_\_\_, b_1 := \_\_\_, b_2 := \_\_\_$

}

assert (b_0)

This was the counterexample path in the old abstraction.