REMEMBER HOARE/FLOYD PROOFS?
CAN WE AUTOMATE THEM?
We need annotations!

Can we come up with them automatically?

We need to verify those annotations: decision procedures

Dafny’s backend (Z3) does this.

Lots of other examples ...

Alternatively, the annotations can be correct-by-construction
LET'S SEE TAKE THE FIRST BABY STEPS TOWARDS AUTOMATION ...
STEP 1: HOUDINI

[Rustan Leino and Cormac Flanagan 2001]
ONCE UPON A TIME ...
ESC/Java architecture

annotated program

translator

verification condition

automatic theorem prover

counterexample

post-processor

warning message

"valid"
What about Legacy code?
Annotation assistant
Annotation assistant
Annotation assistant
Annotation assistant
Annotation assistant
Annotation assistant
Annotation assistant

Houdini

The great ESC wizard!
Annotation assistant

Unannotated Java program

Inference engine

Annotated Java program

ESC/Java

Warning messages
Basically, the idea is as follows:
Basically, the idea is as follows:

Fix a \textit{finite} set of candidate formulas.
Basically, the idea is as follows:

Fix a **finite** set of candidate formulas.

<table>
<thead>
<tr>
<th>Type of f</th>
<th>Candidate invariants for f</th>
</tr>
</thead>
<tbody>
<tr>
<td>integral type</td>
<td>//@ invariant f cmp expr;</td>
</tr>
<tr>
<td>reference type</td>
<td>//@ invariant f != null;</td>
</tr>
<tr>
<td>array type</td>
<td>//@ invariant f != null;</td>
</tr>
<tr>
<td></td>
<td>//@ invariant \nonnullelements(f);</td>
</tr>
<tr>
<td></td>
<td>//@ invariant (\forall int i; 0 &lt;= i &amp;&amp; i &lt; expr \implies f[i] != null);</td>
</tr>
<tr>
<td></td>
<td>//@ invariant f.length cmp expr;</td>
</tr>
<tr>
<td>boolean</td>
<td>//@ invariant f == false;</td>
</tr>
<tr>
<td></td>
<td>//@ invariant f == true;</td>
</tr>
</tbody>
</table>
Basically, the idea is as follows:

Fix a finite set of candidate formulas.
Basically, the idea is as follows:

Fix a finite set of possible formulas (our atoms).

Let $\Phi$ be the set of formulas formed as any conjunction of these formulas.
Basicall, the idea is as follows:

Fix a finite set of possible formulas (our atoms).

Let $\Phi$ be the set of formulas formed as any conjunction of these formulas.

for each edge $u \rightarrow v$ labeled with command $c$

$$\text{Annotation}(v) = \{ \phi \in \Phi \mid \text{Annotation}(u) \vdash \text{wp}(c, \phi) \}$$

until $\text{Annotation}$ doesn’t change
IN A NOT SO DISTANT PAST ...

[Sharma and Aiken 2014]
The basic idea is the same, but the scope is smaller:
The basic idea is the same, but the scope is smaller:

we just want to automatically discover loop invariants.
The basic idea is the same, but the scope is smaller:

we just want to automatically discover loop invariants.

The discoverable invariants range over a finite set of formulas.
The basic idea is the same, but the scope is smaller:

we just want to automatically discover loop invariants.

The discoverable invariants range over a finite set of formulas.

We define this set through a first guess and a finite set of possible moves.
The basic idea is the same, but the scope is smaller:

we just want to automatically discover loop invariants.

The discoverable invariants range over a finite set of formulas.

We define this set through a first guess and a finite set of possible moves.

It is more straightforward to check a candidate:
The basic idea is the same, but the scope is smaller:

we just want to automatically discover loop invariants.

The discoverable invariants range over a finite set of formulas.

We define this set through a first guess and a finite set of possible moves.

It is more straightforward to check a candidate:

\[(\text{pre } \implies I) \land (\{I\} \text{ Body } \{I\}) \land (I \land \neg C \implies \text{ post})\]
INVARIANT INFERENCE USING RANDOMIZED SEARCH
Use a customized Metropolis-Hastings (randomized sampling) algorithm to search for an invariant in the space of moves.
INVARIANT INFEERENCE USING RANDOMIZED SEARCH

Use a customized Metropolis-Hastings (randomized sampling) algorithm to search for an invariant in the space of moves.

1. Select an initial candidate

2. Repeat (millions of times)
   • Propose a random modification and evaluate cost
   • If (cost decreased)
     {accept}
   • If (cost increased)
     {with some probability accept anyway}
Use a customized Metropolis-Hastings (randomized sampling) algorithm to search for an invariant in the space of moves.

1. Select an initial candidate

2. Repeat (millions of times)
   - Propose a random modification and evaluate cost
   - If ( cost decreased )
     { accept }
   - If ( cost increased )
     { with some probability accept anyway }
Use a customized Metropolis-Hastings (randomized sampling) algorithm to search for an invariant in the space of moves.
Use a customized Metropolis-Hastings (randomized sampling) algorithm to search for an invariant in the space of moves.

Appropriate Set of moves: symmetric and ergodic
Use a customized Metropolis-Hastings (randomized sampling) algorithm to search for an invariant in the space of moves.

Appropriate Set of moves: symmetric and ergodic

It guarantees convergence
INVARIANT INFERENCE USING RANDOMIZED SEARCH

Use a customized Metropolis-Hastings (randomized sampling) algorithm to search for an invariant in the space of moves.

Appropriate Set of moves: symmetric and ergodic

Useful cost function: use a concrete set of good (G) and bad (B) states, and state-pairs (Z)

\[ c_V(C) = \sum_{g \in G} \sum_{b \in B} (-C(g) \cdot -C(b) + C(g) \cdot C(b)) + \sum_{g \in G} -C(g) + \sum_{b \in B} C(b) + \sum_{(s,t) \in Z} C(s) \cdot -C(t) \]
INVARIANT INFERENCE USING RANDOMIZED SEARCH

Use a customized Metropolis-Hastings (randomized sampling) algorithm to search for an invariant in the space of moves.

Appropriate Set of moves: symmetric and ergodic

advantage: anything with a non-zero cost is definitely not an invariant, so it can be thrown out before going to solves!

$$c_V(C) = \sum_{g \in G} \sum_{b \in B} (-C(g) \cdot -C(b) + C(g) \cdot C(b)) + \sum_{g \in G} -C(g) + \sum_{b \in B} C(b) + \sum_{(s,t) \in Z} C(s) \cdot -C(t)$$
SET OF MOVES

It depends on what type of invariant you want to infer:
It depends on what type of invariant you want to infer:

**Numerical Invariants:**

\[ \bigvee \bigwedge_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \left( \sum_{k=1}^{n} w_{k}^{(i,j)} x_k \leq d^{(i,j)} \right) \]
It depends on what type of invariant you want to infer:

**Numerical Invariants:**

$$\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \left( \sum_{k=1}^{n} w_{k}^{(i,j)} x_k \leq d^{(i,j)} \right)$$

Fix a **finite** set of coefficients and constants (from the code).
It depends on what type of invariant you want to infer:

**Numerical Invariants:**

\[
\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \left( \sum_{k=1}^{n} w_{k}^{(i,j)} x_k \leq d^{(i,j)} \right)
\]

Fix a finite set of coefficients and constants (from the code).

Up to 10 conjuncts, and if required up to 10 disjuncts.
It depends on what type of invariant you want to infer:

**Numerical Invariants:**

$$\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \left( \sum_{k=1}^{n} w_{i,j}^{(i,j)} x_k \leq d_{i,j}^{(i,j)} \right)$$

Fix a **finite** set of coefficients and constants (from the code).

Up to 10 conjuncts, and if required up to 10 disjuncts.

Select **uniformly at random** a $k, i, j$, all coefficients, and all constants. Then take a move with probability $1/3$: 
SET OF MOVES

It depends on what type of invariant you want to infer:

**Numerical Invariants:**

\[
\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \left( \sum_{k=1}^{n} w_{k}^{(i,j)} x_k \leq d^{(i,j)} \right)
\]

Fix a finite set of coefficients and constants (from the code).

Up to 10 conjuncts, and if required up to 10 disjuncts.

Select uniformly at random a k, i, j, all coefficients, and all constants. Then take a move with probability 1/3:

- randomly select a coefficient, and update \(d^{(i,j)}\)
It depends on what type of invariant you want to infer:

**Numerical Invariants:**

\[ \bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \left( \sum_{k=1}^{n} w_{k}^{(i,j)} x_k \leq d^{(i,j)} \right) \]

Fix a *finite* set of coefficients and constants (from the code).

Up to 10 conjuncts, and if required up to 10 disjuncts.

Select *uniformly at random* a \( k, i, j, \) all coefficients, and all constants. Then take a move with probability 1/3:

- randomly select a *coefficient*, and update \( d^{(i,j)} \)
- randomly select a *constant*, and update: \( w_{k}^{(i,j)}\)
It depends on what type of invariant you want to infer:

**Numerical Invariants:**

\[
\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \left( \sum_{k=1}^{n} w_{k}^{(i,j)} x_k \leq d^{(i,j)} \right)
\]

Fix a finite set of coefficients and constants (from the code).

Up to 10 conjuncts, and if required up to 10 disjuncts.

Select uniformly at random a \( k, i, j \), all coefficients, and all constants. Then take a move with probability 1/3:

- randomly select a coefficient, and update \( d^{(i,j)} \)
- randomly select a constant, and update: \( w_{k}^{(i,j)} \)
- with some probability \( p \), change all \( w^{(i,j)} \) and \( d^{(i,j)} \), and with probability \( (1-p) \) remove the inequality entirely.
WHAT DO WE LEARN?

One can imagine then invariants for arrays, strings, ...
WHAT DO WE LEARN?

One can imagine then invariants for arrays, strings, ...

The moral of the story is:
WHAT DO WE LEARN?

One can imagine then invariants for arrays, strings, ...

The moral of the story is:

☐ If you setup a finite search space for your all admirable invariants, then you can, well, search it.
☐ You need decision procedures as your oracle.
☐ One can try to avoid using them until necessary.
WHAT DO WE LEARN?

One can imagine then invariants for arrays, strings, ...

The moral of the story is:

☐ If you setup a finite search space for your all admirable invariants, then you can, well, search it.
☐ You need decision procedures as your oracle.
☐ One can try to avoid using them until necessary.

You can design your own clever state-space and your own clever search, and possibly compete with these people!
STEP 2: CONSTANT PROPAGATION
WHAT DO WE LEARN?

What if I am not happy with a finite set of invariant choices?

```plaintext
z = 3
x = 1
while (x > 0) {
    if (x = 1) then y = 7
    else y = z + 4
    x = 3
    print y
}
```
WHAT DO WE LEARN?

What if I am not happy with a finite set of invariant choices?

```
z = 3
x = 1
while (x > 0) {
    if (x = 1) then y = 7
    else y = z + 4
    x = 3
    print y
}
```

What if I am not greedy, and all I care about is to determine whether a variable has a constant value at any given program point.
What if I am not happy with a finite set of invariant choices?

```plaintext
z = 3
x = 1
while (x > 0) {
    if (x = 1) then y = 7
    else y = z + 4
    x = 3
    print y
}
```

What if I am not greedy, and all I care about is to determine whether a variable has a constant value at any given program point.

My annotations are then of the form: \( v = c \) (\( v \in V, c \in \mathbb{Z} \))
What if I am not happy with a finite set of invariant choices?

\[ z = 3 \]
\[ x = 1 \]
while \( x > 0 \) {
  if \( x = 1 \) then \( y = 7 \)
  else \( y = z + 4 \)

\[ x = 3 \]
print \( y \)
}

What if I am not greedy, and all I care about is to determine whether a variable has a constant value at any given program point.

My annotations are then of the form:

\[ v = c \ (v \in V, c \in \mathbb{Z}) \]

\[ v = c \ (v \in V, c \in \mathbb{Z} \cup \{*\}) \]
What do we learn?

What if I am not happy with a finite set of invariant choices?

```plaintext
z = 3
x = 1
while (x > 0) {
    if (x = 1) then y = 7
    else y = z + 4
    x = 3
    print y
}
```
What if I am not happy with a *finite* set of invariant choices?

\[
\begin{align*}
z &= 3 \\
x &= 1 \\
\text{while } (x > 0) \{ \\
  &\quad \text{if } (x = 1) \text{ then } y = 7 \\
  &\quad \quad \text{else } y = z + 4 \\
  &\quad \quad x = 3 \\
  &\quad \quad \text{print } y \\
\}
\end{align*}
\]

What if I am not greedy, and all I care about is to determine whether a variable has a *constant value* at any given program point.
WHAT DO WE LEARN?

What if I am not happy with a finite set of invariant choices?

\[
z = 3 \\
x = 1 \\
\text{while } (x > 0) \{ \\
\quad \text{if } (x = 1) \text{ then } y = 7 \\
\quad \quad \text{else } y = z + 4 \\
\quad x = 3 \\
\quad \text{print } y \\
\}\]

What if I am not greedy, and all I care about is to determine whether a variable has a constant value at any given program point.

My annotations are then of the form: \[ v = c \ (v \in V, c \in \mathbb{Z}) \]
What if I am not happy with a finite set of invariant choices?

\[
\begin{align*}
z &= 3 \\
x &= 1 \\
\text{while } (x > 0) \{ \\
& \quad \text{if } (x = 1) \text{ then } y = 7 \\
& \quad \quad \text{else } y = z + 4 \\
& \quad x = 3 \\
& \quad \text{print } y \\
\}
\end{align*}
\]

What if I am not greedy, and all I care about is to determine whether a variable has a constant value at any given program point.

My annotations are then of the form: \( v = c \cdot (v \in V, c \in \mathbb{Z}) \)

\[
v = c \cdot (v \in V, c \in \mathbb{Z} \cup \{\ast\})
\]
Here is a natural algorithm you would naturally choose to run:

```plaintext
z = 3
x = 1

while (x > 0) {
    if (x = 1) then y = 7
    else y = z + 4
    x = 3
    print y
}
```
Here is a natural algorithm you would naturally choose to run:

[x=0, y=0, z=0]
  z = 3
  x = 1
  while (x > 0) {
    if (x = 1) then y = 7
    else y = z + 4
    x = 3
  }
  print y
Here is a natural algorithm you would naturally choose to run:

```
[x=0, y=0, z=0]
z = 3
[x=0, y=0, z=3]
x = 1

while (x > 0) {
    if (x = 1) then y = 7
    else y = z + 4

    x = 3
}
print y
```
Here is a natural algorithm you would naturally choose to run:

\[
\begin{align*}
[x=0, y=0, z=0] \\
\quad z &= 3 \\
[x=0, y=0, z=3] \\
\quad x &= 1 \\
[x=1, y=0, z=3] \\
\quad \text{while } (x > 0) \{ \\
\quad \quad \text{if } (x = 1) \text{ then } y &= 7 \\
\quad \quad \quad \text{else } y &= z + 4 \\
\quad \quad x &= 3 \\
\quad \text{print } y \\
\quad \}\end{align*}
\]
Here is a natural algorithm you would naturally choose to run:

```plaintext
[x=0, y=0, z=0]
  z = 3
[x=0, y=0, z=3]
  x = 1
[x=1, y=0, z=3]
  while (x > 0) {
  [x=1, y=0, z=3]
    if (x = 1) then y = 7
    else y = z + 4
    x = 3
  print y
  }
```
CONSTANT PROPAGATION

Here is a natural algorithm you would naturally choose to run:

[x=0, y=0, z=0]
  z = 3
[x=0, y=0, z=3]
  x = 1
[x=1, y=0, z=3]
  while (x > 0) {
[x=1, y=0, z=3]
    if (x = 1) then y = 7  [x=1, y=7, z=3]
      else y = z + 4

    x = 3

  print y

}
Here is a natural algorithm you would naturally choose to run:

```plaintext
[x=0, y=0, z=0]
  z = 3
[x=0, y=0, z=3]
  x = 1
[x=1, y=0, z=3]
  while (x > 0) {
[x=1, y=0, z=3]
    if (x = 1) then y = 7  [x=1, y=7, z=3]
    else y = z + 4
[x=1, y=7, z=3]
  x = 3
  print y
[x=1, y=7, z=3]
}
```
Here is a natural algorithm you would naturally choose to run:

```plaintext
[x=0, y=0, z=0]
z = 3
[x=0, y=0, z=3]
x = 1
[x=1, y=0, z=3]
while (x > 0) {
    [x=1, y=0, z=3]
    if (x = 1) then y = 7   [x=1, y=7, z=3]
    else y = z + 4
    [x=1, y=7, z=3]
x = 3
    [x=3, y=7, z=3]
    print y
}
```
CONSTANT PROPAGATION

Here is a natural algorithm you would naturally choose to run:

```
[x=0, y=0, z=0]
    z = 3
[x=0, y=0, z=3]
    x = 1
[x=1, y=0, z=3]
    while (x > 0) {
        if (x = 1) then y = 7  [x=1, y=7, z=3]
        else y = z + 4
        [x=1, y=7, z=3]
            x = 3
        [x=3, y=7, z=3]
            print y
        [x=3, y=7, z=3]
    }
```
Here is a natural algorithm you would naturally choose to run:

\[
\begin{align*}
[x=0, y=0, z=0] \\
& z = 3 \\
[x=0, y=0, z=3] \\
& x = 1 \\
[x=1, y=0, z=3] \\
& \text{while (} x > 0 \text{) } \{ \\
& \quad [x=*, y=*, z=3] \\
& \quad \quad \text{if (} x = 1 \text{) then } y = 7 \quad [x=1, y=7, z=3] \\
& \quad \quad \quad \quad \text{else } y = z + 4 \\
& \quad \quad [x=1, y=7, z=3] \\
& \quad \quad \quad \quad x = 3 \\
& \quad \quad [x=3, y=7, z=3] \\
& \quad \quad \quad \quad \text{print } y \\
& \quad \quad [x=3, y=7, z=3] \\
& \} \\
\end{align*}
\]
CONSTANT PROPAGATION

Here is a natural algorithm you would naturally choose to run:

```plaintext
[x=0, y=0, z=0]
z = 3
[x=0, y=0, z=3]
x = 1
[x=1, y=0, z=3]
    while (x > 0) {
        [x=*, y=*, z=3]
            if (x = 1) then y = 7  [x=1, y=7, z=3]
            else y = z + 4
        [x=1, y=7, z=3]
x = 3
        [x=3, y=7, z=3]
    print y
[x=3, y=7, z=3]
}
```
Here is a natural algorithm you would naturally choose to run:

\[
\begin{align*}
&[x=0, y=0, z=0] \\
&\quad z = 3 \\
&[x=0, y=0, z=3] \\
&\quad x = 1 \\
&[x=1, y=0, z=3] \\
&\quad \text{while } (x > 0) \{ \\
&\quad \quad [x=*, y=*, z=3] \\
&\quad \quad \quad \text{if } (x = 1) \text{ then } y = 7 \quad [x=1, y=7, z=3] \\
&\quad \quad \quad \quad \text{else } y = z + 4 \quad [x=*, y=7, z=3] \\
&\quad \quad [x=1, y=7, z=3] \\
&\quad \quad \quad x = 3 \\
&\quad [x=3, y=7, z=3] \\
&\quad \quad \quad \text{print } y \\
&\quad [x=3, y=7, z=3] \\
&\quad \}\}
\end{align*}
\]
Here is a natural algorithm you would naturally choose to run:

\[
[x=0, y=0, z=0] \\
\quad z = 3 \\
[x=0, y=0, z=3] \\
\quad x = 1 \\
[x=1, y=0, z=3] \\
\quad \text{while } (x > 0) \{ \\
[x=*, y=*, z=3] \\
\quad \text{if } (x = 1) \text{ then } y = 7 \quad [x=1, y=7, z=3] \\
\quad \text{else } y = z + 4 \quad [x=*, y=7, z=3] \\
[x=*, y=7, z=3] \\
\quad x = 3 \\
[x=3, y=7, z=3] \\
\quad \text{print } y \\
[x=3, y=7, z=3] \\
\}\]
Here is a natural algorithm you would naturally choose to run:

\[
\begin{align*}
\text{[x=0, y=0, z=0] } \\
& \quad \text{z = 3} \\
\text{[x=0, y=0, z=3] } \\
& \quad \text{x = 1} \\
\text{[x=1, y=0, z=3] } \\
& \quad \text{while (x > 0) { } } \\
\text{[x=*, y=*, z=3] } \\
& \quad \text{if (x = 1) then y = 7 } \quad \text{[x=1, y=7, z=3]} \\
& \quad \quad \text{else y = z + 4 } \quad \text{[x=*, y=7, z=3]} \\
\text{[x=*, y=7, z=3] } \\
& \quad \text{x = 3} \\
\text{[x=3, y=7, z=3] } \\
& \quad \text{print y} \\
\text{[x=3, y=7, z=3] } \\
\end{align*}
\]
Here is a natural algorithm you would naturally choose to run:

\[
\begin{align*}
&\text{[x=0, y=0, z=0]} \\
&\quad \quad z = 3 \\
&\text{[x=0, y=0, z=3]} \\
&\quad \quad x = 1 \\
&\text{[x=1, y=0, z=3]} \\
&\quad \quad \text{while (x > 0) {}} \\
&\text{[x=*, y=*, z=3]} \\
&\quad \quad \quad \text{if (x = 1) then y = 7} \quad [x=1, y=7, z=3] \\
&\quad \quad \quad \quad \text{else y = z + 4} \quad [x=*, y=7, z=3] \\
&\text{[x=*, y=7, z=3]} \\
&\quad \quad x = 3 \\
&\text{[x=3, y=7, z=3]} \\
&\quad \quad \text{print y} \\
&\text{[x=3, y=7, z=3]} \\
\end{align*}
\]
CONSTANT PROPAGATION

More formally:
CONSTANT PROPAGATION

More formally: \( v = c \ (v \in V, c \in \mathbb{Z} \cup \{\ast\} \cup \{?\}) \)
CONSTANT PROPAGATION

More formally: \[ v = c \quad (v \in V, c \in \mathbb{Z} \cup \{\ast\} \cup \{\text{?}\}) \]

This is done over the control flow graph (CFG).
CONSTANT PROPAGATION

More formally: \[ v = c \ (v \in V, c \in \mathbb{Z} \cup \{\ast\} \cup \{?\}) \]

This is done over the control flow graph (CFG).

Initially everything is unknown everywhere other than at the entry of the graph, where everything is 0.
CONSTANT PROPAGATION

More formally: \[ v = c \ (v \in V, c \in \mathbb{Z} \cup \{\ast\} \cup \{?\}) \]

This is done over the control flow graph (CFG).

Initially everything is unknown everywhere other than at the entry of the graph, where everything is 0.

There is a precise function that defines how every command transforms the information from its source to its destination.
CONSTANT PROPAGATION

More formally: 
\[ v = c \ (v \in V, c \in \mathbb{Z} \cup \{\ast\} \cup \{?\}) \]

This is done over the control flow graph (CFG).

Initially everything is unknown everywhere other than at the entry of the graph, where everything is 0.

There is a precise function that defines how every command transforms the information from its source to its destination.

It is well-defined how to combine information whenever there is a join in the CFG.
CONSTANT PROPAGATION

More formally: \[ v = c \ (v \in V, c \in \mathbb{Z} \cup \{\ast\} \cup \{?\}) \]

This is done over the control flow graph (CFG).

Initially everything is unknown everywhere other than at the entry of the graph, where everything is 0.

There is a precise function that defines how every command transforms the information from its source to its destination.

It is well-defined how to combine information whenever there is a join in the CFG.

The algorithm iterates, updating each node, until nothing is changed.
WHY DOES THIS TERMINATE?