# Loop Summarization with Rational Vector Addition Systems

Jake Silverman

Zachary Kincaid

**Princeton University**

# The Why

Invariant generation techniques are effective

but can be unpredictable

# The Why

Invariant generation techniques are effective

but can be unpredictable

```
i = 0
while(i < 5) do
    i++
assert(i == 5)
```

**Polyhedron domain with widening / narrowing verifies assertion**

*D. Monniaux and J. Le Guen. Stratified Static Analysis Based on Variable Dependencies. in Proc: International Workshop on Numerical and Symbolic Abstract Domains (NSAD '11)

# The Why

Invariant generation techniques are effective

but can be unpredictable

```
i = 1
j = 0
while(i < 5) do
  j = j + i
  i++
assert(i == 5)
```

**Polyhedron domain with widening / narrowing fails to verify assertion**

**Not monotone: more information led to worse analysis**

*D. Monniaux and J. Le Guen. Stratified Static Analysis Based on Variable Dependencies. in Proc: International Workshop on Numerical and Symbolic Abstract Domains (NSAD '11)

# The Why

Invariant generation techniques are effective

but can be unpredictable

```
i = 1
j = 0
while(i < 5) do
  j = j + i
  i++
assert(i == 5)
```

**Polyhedron domain with widening / narrowing fails to verify assertion**

```
i = 0
j = 0
while(i < 1000) do
   i = i + step
   j = j + step
assert(i == j)
```

**Ultimate Automizer verifies assertion**

**Not monotone: more information led to worse analysis**

*D. Monniaux and J. Le Guen. Stratified Static Analysis Based on Variable Dependencies. in Proc: International Workshop on Numerical and Symbolic Abstract Domains (NSAD '11)

# The Why

Invariant generation techniques are effective

but can be unpredictable

```
i = 1
j = 0
while(i < 5) do
  j = j + i
  i++
assert(i == 5)
```

**Polyhedron domain with widening / narrowing fails to verify assertion**

```
assume(step < 2)
i = 0
j = 0
while(i < 1000) do
   i = i + step
   j = j + step
assert(i == j)
```

**Ultimate Automizer fails to verify assertion within 1 hour**

**Not monotone: more information led to worse analysis**

*D. Monniaux and J. Le Guen. Stratified Static Analysis Based on Variable Dependencies. in Proc: International Workshop on Numerical and Symbolic Abstract Domains (NSAD '11)

# The What

Want: invariant generation technique that is

predictable - can make theoretical guarantees about invariant quality (in particular, monotonicity)

precise - assertion verification capability comparable with state-of-the-art software model checkers

# The How

**Exploit compositionality to compute transition formula that over-approximates reachability relation of input**

$$\textbf{TR}[\![x := a]\!] \triangleq x' = a \wedge \bigwedge_{y \neq x} y' = y$$

$$\textbf{TR}[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2]\!] \triangleq b \wedge \textbf{TR}[\![S_1]\!] \vee \neg b \wedge \textbf{TR}[\![S_2]\!]$$

$$\textbf{TR}[\![S_1 ; S_2]\!] \triangleq \exists \overrightarrow{x''} . \textbf{TR}[\![S_1]\!][\overrightarrow{x''}/\overrightarrow{x'}] \wedge \textbf{TR}[\![S_2]\!][\overrightarrow{x''}/\overrightarrow{x}]$$

$$\textbf{TR}[\![\textbf{while } b \textbf{ do } S]\!] \triangleq (b \wedge \textbf{TR}[\![S]\!])^* \wedge \neg b[\overrightarrow{x'}/\overrightarrow{x}]$$

# The How

**Exploit compositionality to compute transition formula that over-approximates reachability relation of input**

$$\textbf{TR}[\![x := a]\!] \triangleq x' = a \wedge \bigwedge_{y \neq x} y' = y$$

$$\textbf{TR}[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2]\!] \triangleq b \wedge \textbf{TR}[\![S_1]\!] \vee \neg b \wedge \textbf{TR}[\![S_2]\!]$$

$$\textbf{TR}[\![S_1; S_2]\!] \triangleq \exists \overrightarrow{x''} . \textbf{TR}[\![S_1]\!][\overrightarrow{x''}/\overrightarrow{x'}] \wedge \textbf{TR}[\![S_2]\!][\overrightarrow{x''}/\overrightarrow{x}]$$

$$\textbf{TR}[\![\textbf{while } b \textbf{ do } S]\!] \triangleq (b \wedge \textbf{TR}[\![S]\!])^* \wedge \neg b[\overrightarrow{x'}/\overrightarrow{x}]$$

---

**Can encode loop-free segments without loss of information**

```
if(*) then
   x = x + 1
else
   x = x + 2
```

$$x' = x + 1 \vee x' = x + 2$$

# The How

**Exploit compositionality to compute transition formula that over-approximates reachability relation of input**

$$\textbf{TR}[\![x := a]\!] \triangleq x' = a \wedge \bigwedge_{y \neq x} y' = y$$

$$\textbf{TR}[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2]\!] \triangleq b \wedge \textbf{TR}[\![S_1]\!] \vee \neg b \wedge \textbf{TR}[\![S_2]\!]$$

$$\textbf{TR}[\![S_1; S_2]\!] \triangleq \exists \overrightarrow{x''} . \textbf{TR}[\![S_1]\!][\overrightarrow{x''}/\overrightarrow{x'}] \wedge \textbf{TR}[\![S_2]\!][\overrightarrow{x''}/\overrightarrow{x}]$$

$$\textbf{TR}[\![\textbf{while } b \textbf{ do } S]\!] \triangleq (b \wedge \textbf{TR}[\![S]\!])^* \wedge \neg b[\overrightarrow{x'}/\overrightarrow{x}]$$

**Can encode loop-free segments without loss of information**

```
if(*) then
   x = x + 1
else
   x = x + 2
```

$$x' = x + 1 \vee x' = x + 2$$

**Reachability relation of loops needs to be over-approximated**

# The How

**Exploit compositionality to compute transition formula that over-approximates reachability relation of input**

$$\mathbf{TR}[\![x := a]\!] \triangleq x' = a \wedge \bigwedge_{y \neq x} y' = y$$

$$\mathbf{TR}[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2]\!] \triangleq b \wedge \mathbf{TR}[\![S_1]\!] \vee \neg b \wedge \mathbf{TR}[\![S_2]\!]$$

$$\mathbf{TR}[\![S_1; S_2]\!] \triangleq \exists \overrightarrow{x''} \,.\, \mathbf{TR}[\![S_1]\!][\overrightarrow{x''}/\overrightarrow{x'}] \wedge \mathbf{TR}[\![S_2]\!][\overrightarrow{x''}/\overrightarrow{x}]$$

$$\mathbf{TR}[\![\textbf{while } b \textbf{ do } S]\!] \triangleq (b \wedge \mathbf{TR}[\![S]\!])^* \wedge \neg b[\overrightarrow{x'}/\overrightarrow{x}]$$

---

**Can encode loop-free segments without loss of information**

```
if(*) then
   x = x + 1
else
   x = x + 2
```

$$x' = x + 1 \vee x' = x + 2$$

---

**Reachability relation of loops needs to be over-approximated**

**If star operator is monotone, entire analysis in monotone**

# This talk

1) Predictable loop summarization using rational vector addition system with resets ($\mathbb{Q}$-VASR)

2) Precision improvement via capturing control flow using $\mathbb{Q}$-VASR with states ($\mathbb{Q}$-VASRS)

# $\mathbb{Q}$-VASR

Key property:

Reachability relation is LIRA-definable and computable in polytime

*C. Haase and S. Halfon. Integer vector addition systems with states. in Proc: International Workshop on Reachability Problems (RP ‘14)

# $\mathbb{Q}$-VASR

Key property:

Reachability relation is LIRA-definable and computable in polytime

**Finite set of transformers. Describes reset/inc to each dimension**

$$\left\{ \overbrace{\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ y \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \end{bmatrix}}^{T^1}, \overbrace{\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 10 \\ -1 \end{bmatrix}}^{T^2} \right\}$$

*C. Haase and S. Halfon. Integer vector addition systems with states. in Proc: International Workshop on Reachability Problems (RP ‘14)

# $\mathbb{Q}$-VASR

Key property:

Reachability relation is LIRA-definable and computable in polytime

**Finite set of transformers. Describes reset/inc to each dimension**

$$\left\{ \overbrace{\begin{bmatrix} x \\ y \end{bmatrix} \to \begin{bmatrix} 0 \\ y \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \end{bmatrix}}^{T^1}, \overbrace{\begin{bmatrix} x \\ y \end{bmatrix} \to \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 10 \\ -1 \end{bmatrix}}^{T^2} \right\}$$

**Corresponds to transition formula of form**

$$\bigvee_{i \in T} \bigwedge_{j \in vars} x'_j = \underbrace{r_{ij}}_{\{0,1\}} \cdot x_j + \underbrace{a_{ij}}_{\mathbb{Q}}$$

$$\begin{matrix} T^1 \\ T^2 \end{matrix} \boxed{\begin{matrix} (x' = 1 \wedge y' = y - 1) \vee \\ (x' = x + 10 \wedge y' = y - 1) \end{matrix}}$$

*C. Haase and S. Halfon. Integer vector addition systems with states. in Proc: International Workshop on Reachability Problems (RP '14)

# $\mathbb{Q}$-VASR

Key property:

Reachability relation is LIRA-definable and computable in polytime

**Finite set of transformers. Describes reset/inc to each dimension**

$$\left\{ \overbrace{\begin{bmatrix} x \\ y \end{bmatrix} \to \begin{bmatrix} 0 \\ y \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \end{bmatrix}}^{T^1}, \overbrace{\begin{bmatrix} x \\ y \end{bmatrix} \to \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 10 \\ -1 \end{bmatrix}}^{T^2} \right\}$$

5, 0.5

**Corresponds to transition formula of form**

$$\bigvee_{i \in T} \bigwedge_{j \in vars} x_j' = \underbrace{r_{ij}}_{\{0,1\}} \cdot x_j + \underbrace{a_{ij}}_{\mathbb{Q}}$$

$$\boxed{\begin{array}{ll} T^1 & (x' = 1 \wedge y' = y - 1) \vee \\ T^2 & (x' = x + 10 \wedge y' = y - 1) \end{array}}$$

*C. Haase and S. Halfon. Integer vector addition systems with states. in Proc: International Workshop on Reachability Problems (RP '14)

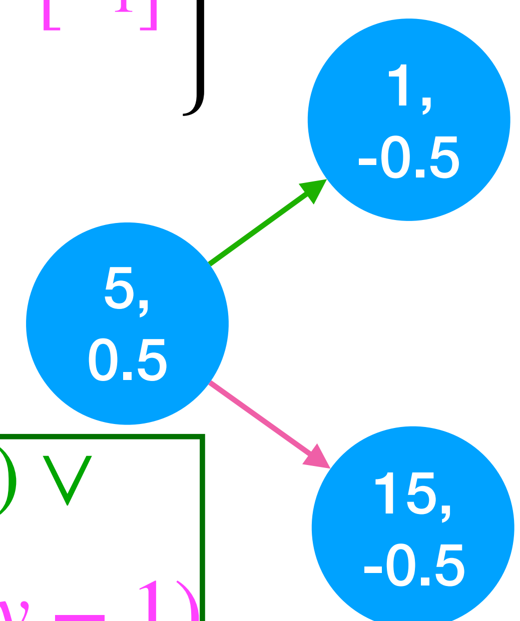
# $\mathbb{Q}$-VASR

Key property:

Reachability relation is LIRA-definable and computable in polytime

**Finite set of transformers. Describes reset/inc to each dimension**

$$\left\{ \overbrace{\begin{bmatrix}x\\y\end{bmatrix} \to \begin{bmatrix}0\\y\end{bmatrix} + \begin{bmatrix}1\\-1\end{bmatrix}}^{T^1}, \overbrace{\begin{bmatrix}x\\y\end{bmatrix} \to \begin{bmatrix}x\\y\end{bmatrix} + \begin{bmatrix}10\\-1\end{bmatrix}}^{T^2} \right\}$$




**Corresponds to transition formula of form**

$$\bigvee_{i\in T}\bigwedge_{j\in vars} x'_j = \underbrace{r_{ij}}_{\{0,1\}} \cdot x_j + \underbrace{a_{ij}}_{\mathbb{Q}}$$

$$\begin{array}{ll} T^1 & (x' = 1 \land y' = y - 1) \lor \\ T^2 & (x' = x + 10 \land y' = y - 1) \end{array}$$

*C. Haase and S. Halfon. Integer vector addition systems with states. in Proc: International Workshop on Reachability Problems (RP '14)

# $\mathbb{Q}$-VASR

Key property:

Reachability relation is LIRA-definable and computable in polytime

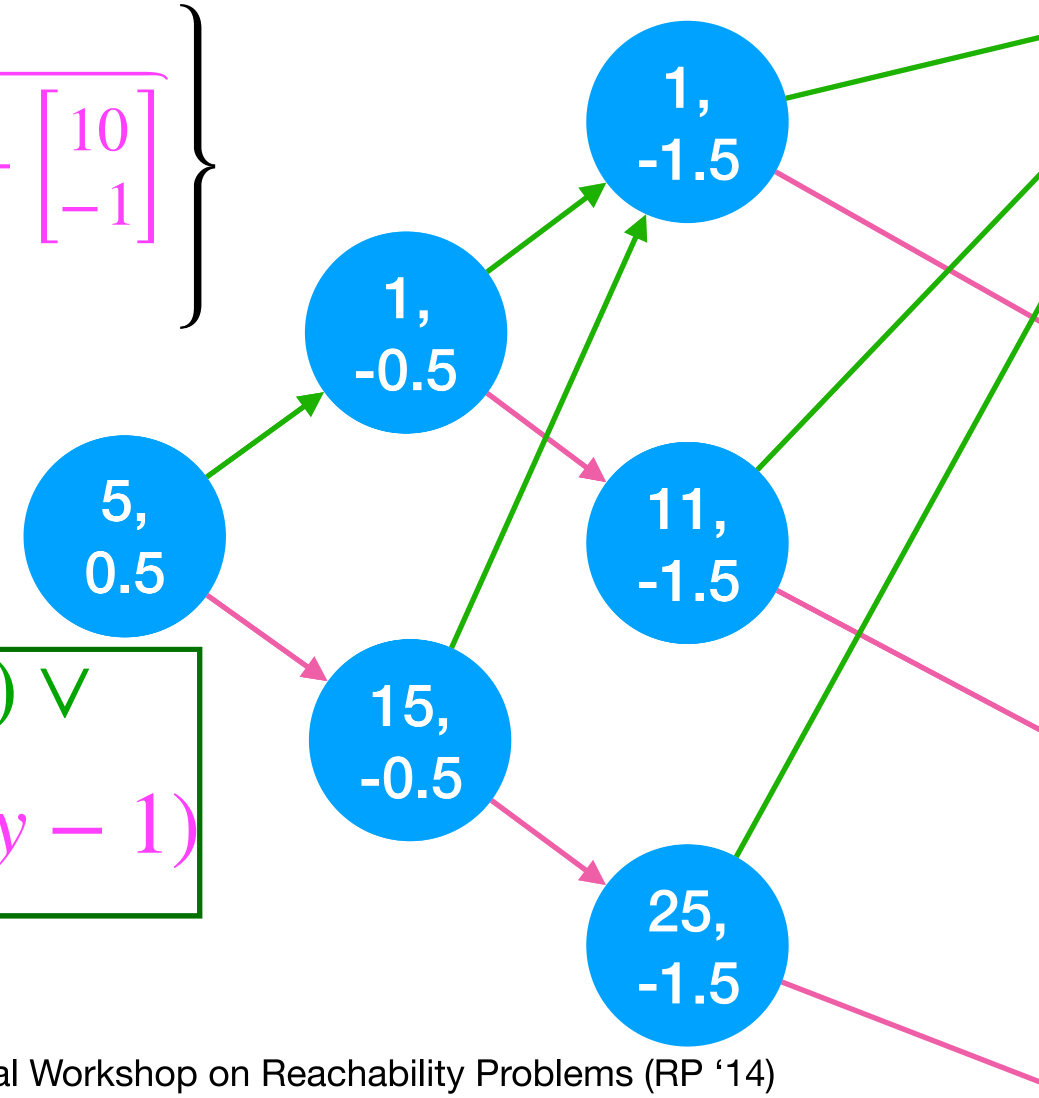
**Finite set of transformers. Describes reset/inc to each dimension**

$$\left\{ \overbrace{\begin{bmatrix}x\\y\end{bmatrix} \to \begin{bmatrix}0\\y\end{bmatrix} + \begin{bmatrix}1\\-1\end{bmatrix}}^{T^1}, \overbrace{\begin{bmatrix}x\\y\end{bmatrix} \to \begin{bmatrix}x\\y\end{bmatrix} + \begin{bmatrix}10\\-1\end{bmatrix}}^{T^2} \right\}$$

**Corresponds to transition formula of form**

$$\bigvee_{i\in T}\bigwedge_{j\in vars} x'_j = \underbrace{r_{ij}}_{\{0,1\}} \cdot x_j + \underbrace{a_{ij}}_{\mathbb{Q}}$$

$$\begin{array}{l} T^1 \\ T^2 \end{array} \boxed{\begin{array}{l} (x' = 1 \land y' = y - 1) \lor \\ (x' = x + 10 \land y' = y - 1) \end{array}}$$




*C. Haase and S. Halfon. Integer vector addition systems with states. in Proc: International Workshop on Reachability Problems (RP '14)

# Functional Queue

**Proof Goal:**

**Amortized constant time operations**

**Achieved by representing queue as two lists (front and back)**

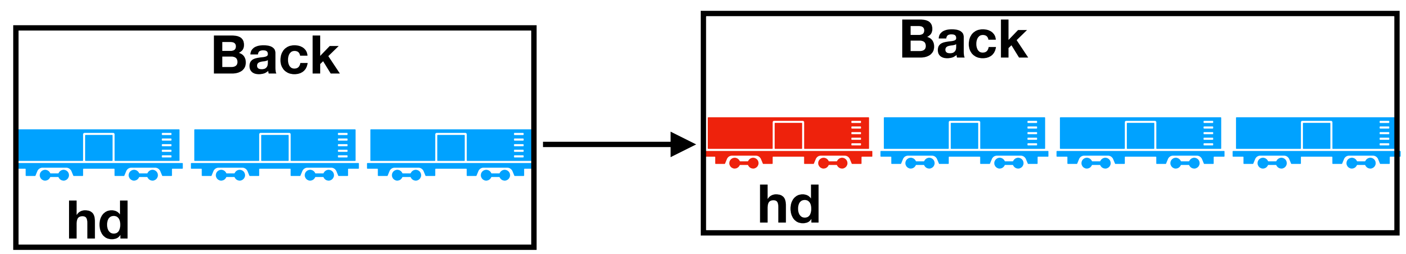
# Functional Queue

**Proof Goal:**

**Amortized constant time operations**

**Achieved by representing queue as two lists (front and back)**

**enqueue()**

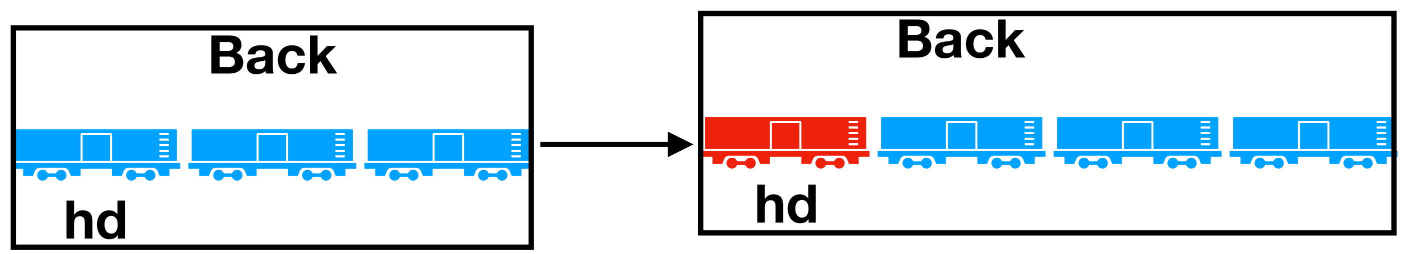


# Functional Queue

**Proof Goal:**
**Amortized constant time operations**

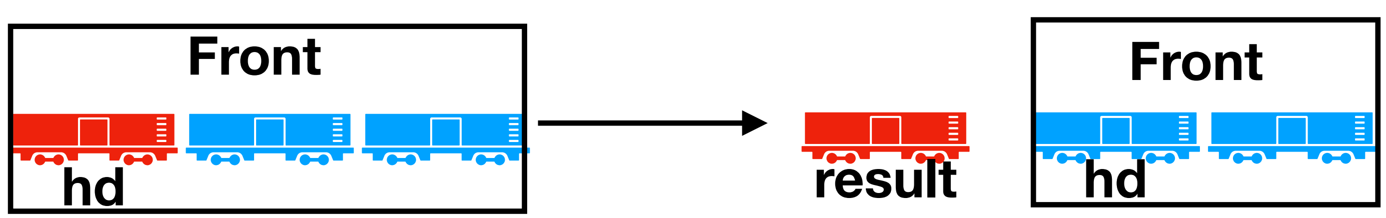
**Achieved by representing queue as two lists (front and back)**

**enqueue(****)**




**dequeue()**

1 **If Front is not empty**



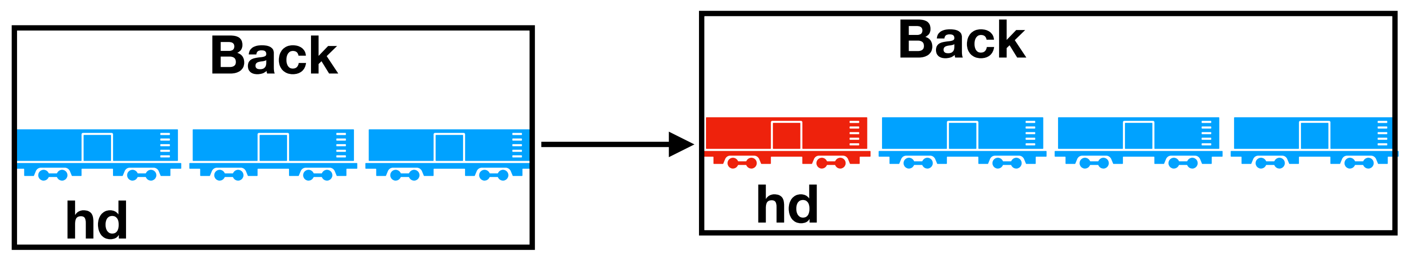
# Functional Queue

**Proof Goal:**

**Amortized constant time operations**

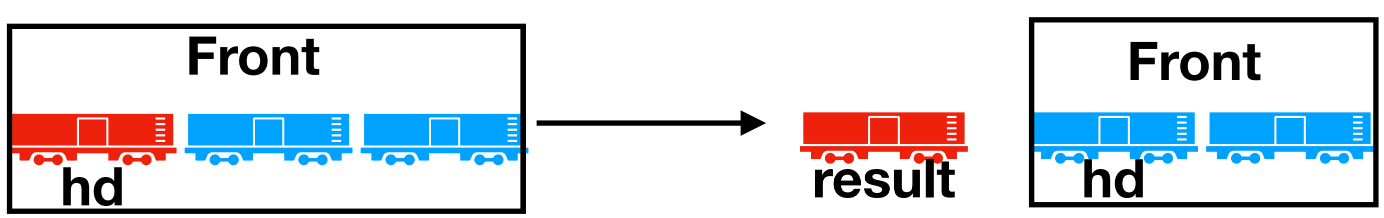
**Achieved by representing queue as two lists (front and back)**

**enqueue(**  **)**

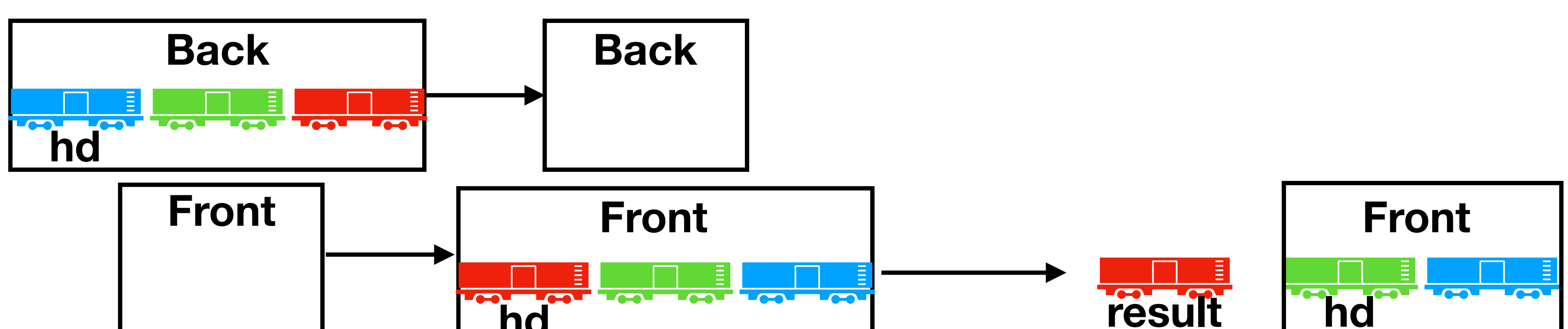



**dequeue()**

1. If Front is not empty




2. If Front is empty



# Functional Queue

```
procedure enqueue(elt):
   back_len := back_len + 1
   size := size + 1
   mem_ops := mem_ops + 1


procedure dequeue():
   if (front_len == 0) then
     //Reverse back, append to front
     while (back_len != 0) do
       front_len := front_len + 1
       back_len := back_len - 1
       mem_ops = mem_ops + 3
   front_len := front_len - 1
   size = size - 1
   mem_ops = mem_ops + 2
```

**Numeric abstraction reasoning about:**

- **length of back list**
- **length of front list**
- **total list size**
- **number of memory operations**

# Functional Queue

```
procedure enqueue(elt):
   back_len := back_len + 1
   size := size + 1
   mem_ops := mem_ops + 1


procedure dequeue():
   if (front_len == 0) then
     //Reverse back, append to front
     while (back_len != 0) do
       front_len := front_len + 1
       back_len := back_len - 1
       mem_ops = mem_ops + 3
   front_len := front_len - 1
   size = size - 1
   mem_ops = mem_ops + 2
```

**Most general harness**

```
procedure harness():
   nb_ops := 0
   while nondet() do
      nb_ops := nb_ops + 1
      if (size > 0 && nondet())
         enqueue()
      else
         dequeue()
```

# Functional Queue

```
procedure enqueue(elt):
   back_len := back_len + 1
   size := size + 1
   mem_ops := mem_ops + 1


procedure dequeue():
   if (front_len == 0) then
     //Reverse back, append to front
     while (back_len != 0) do
       front_len := front_len + 1
       back_len := back_len - 1
       mem_ops = mem_ops + 3
   front_len := front_len - 1
   size = size - 1
   mem_ops = mem_ops + 2
```

**Most general harness**

```
procedure harness():
   nb_ops := 0
   while nondet() do
      nb_ops := nb_ops + 1
      if (size > 0 && nondet())
         enqueue()
      else
         dequeue()
```

# Functional Queue Inner-Loop

```
while (back_len != 0) do
   front_len := front_len + 1
   back_len := back_len - 1
   mem_ops = mem_ops + 3
```

**Transition formula for single iteration**

$$back\_len \neq 0 \land \begin{pmatrix} \quad \texttt{front\_len'} = \texttt{front\_len} + 1 \\ \land \texttt{back\_len'} = \texttt{back\_len} - 1 \\ \land \texttt{mem\_ops'} = \texttt{mem\_ops} + 3 \\ \land \texttt{size'} = \texttt{size} \end{pmatrix}$$

# Functional Queue Inner-Loop

```
while (back_len != 0) do
   front_len := front_len + 1
   back_len := back_len - 1
   mem_ops = mem_ops + 3
```

**Transition formula for single iteration**

$$back\_len \neq 0 \land \begin{pmatrix} \texttt{front\_len'} = \texttt{front\_len} + 1 \\ \land\, \texttt{back\_len'} = \texttt{back\_len} - 1 \\ \land\, \texttt{mem\_ops'} = \texttt{mem\_ops} + 3 \\ \land\, \texttt{size'} = \texttt{size} \end{pmatrix}$$

**ℚ-VASR Abstraction**

$$V_{\textbf{deq}} = \left\{ \begin{bmatrix} front\_len \\ back\_len \\ mem\_ops \\ size \end{bmatrix} \rightarrow \begin{bmatrix} front\_len \\ back\_len \\ mem\_ops \\ size \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 3 \\ 0 \end{bmatrix} \right\}$$

# Functional Queue Inner-Loop

```
while (back_len != 0) do
   front_len := front_len + 1
   back_len := back_len - 1
   mem_ops = mem_ops + 3
```

**Transition formula for single iteration**

$$back\_len \neq 0 \land \begin{pmatrix} \texttt{front\_len'} = \texttt{front\_len} + 1 \\ \land\, \texttt{back\_len'} = \texttt{back\_len} - 1 \\ \land\, \texttt{mem\_ops'} = \texttt{mem\_ops} + 3 \\ \land\, \texttt{size'} = \texttt{size} \end{pmatrix}$$

**ℚ-VASR Abstraction**

$$V_{\textbf{deq}} = \left\{ \begin{bmatrix} front\_len \\ back\_len \\ mem\_ops \\ size \end{bmatrix} \to \begin{bmatrix} front\_len \\ back\_len \\ mem\_ops \\ size \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 3 \\ 0 \end{bmatrix} \right\}$$

**Reachability Relation**

$$\exists k \in \mathbb{N} \,.\, \begin{array}{l} front\_len' = front\_len + k \land \\ back\_len' = back\_len - k \land \\ mem\_ops' = mem\_ops + 3k \land \\ size' = size \end{array}$$

# Functional Queue

```
procedure harness():
  nb_ops := 0
  while nondet() do
    nb_ops := nb_ops + 1
    if (size > 0 && nondet())
      enqueue()
    else
      dequeue()
```

```
procedure enqueue(elt):
  back_len := back_len + 1
  size := size + 1
  mem_ops := mem_ops + 1
```

```
procedure dequeue():
  if (front_len == 0) then
```

$$back\_len' = 0 \land \exists k \in \mathbb{N}\,.\begin{pmatrix} front\_len' = front\_len + k \land \\ back\_len' = back\_len - k \land \\ mem\_ops' = mem\_ops + 3k \land \\ size' = size \end{pmatrix}$$

```
  front_len := front_len - 1
  size = size - 1
  mem_ops = mem_ops + 2
```

# Functional Queue

```
procedure harness():
   nb_ops := 0
   while nondet() do
      nb_ops := nb_ops + 1
      if (size > 0 && nondet())
         enqueue()
      else
         dequeue()
```

```
procedure enqueue(elt):
   back_len := back_len + 1
   size := size + 1
   mem_ops := mem_ops + 1
```

```
procedure dequeue():
   if (front_len == 0) then
```

$$\boxed{\begin{array}{r} back\_len' = 0 \wedge \\ \exists k \in \mathbb{N}\,. \end{array} \left( \begin{array}{l} front\_len' = front\_len + k \wedge \\ back\_len' = back\_len - k \wedge \\ mem\_ops' = mem\_ops + 3k \wedge \\ size' = size \end{array} \right)}$$

```
   front_len := front_len - 1
   size = size - 1
   mem_ops = mem_ops + 2
```

# Functional Queue

```
procedure harness():
   nb_ops := 0
   while nondet() do
      nb_ops := nb_ops + 1
      if (size > 0 && nondet())
         enqueue()
      else
         dequeue()
```

```
procedure enqueue(elt):
   back_len := back_len + 1
   size := size + 1
   mem_ops := mem_ops + 1
```

```
procedure dequeue():
   if (front_len == 0) then
```

$$back\_len' = 0 \wedge \exists k \in \mathbb{N}.\ \left( \begin{array}{l} front\_len' = front\_len + k \wedge \\ back\_len' = back\_len - k \wedge \\ mem\_ops' = mem\_ops + 3k \wedge \\ size' = size \end{array} \right)$$

```
   front_len := front_len - 1
   size = size - 1
   mem_ops = mem_ops + 2
```

$front\_len$ **can increase by arbitrary value**

$back\_len + front\_len$ **is always incremented or decremented by 1**

`enqueue:` $(back\_len + front\_len) + +$

`dequeue:` $(back\_len + front\_len) - -$

$$
V_{\textbf{har}} = \left\{ \begin{array}{l}
\underbrace{\begin{bmatrix} size \\ back\_len \\ mem\_ops + 3 * back\_len \\ back\_len + front\_len \\ nb\_ops \end{bmatrix} \rightarrow \begin{bmatrix} size \\ back\_len \\ mem\_ops + 3 * back\_len \\ back\_len + front\_len \\ nb\_ops \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 4 \\ 1 \\ 1 \end{bmatrix},}_{\textbf{enqueue}} \\
\underbrace{\begin{bmatrix} size \\ back\_len \\ mem\_ops + 3 * back\_len \\ back\_len + front\_len \\ nb\_ops \end{bmatrix} \rightarrow \begin{bmatrix} size \\ back\_len \\ mem\_ops + 3 * back\_len \\ back\_len + front\_len \\ nb\_ops \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \\ 2 \\ -1 \\ 1 \end{bmatrix},}_{\textbf{dequeue fast (conditional passed)}} \\
\underbrace{\begin{bmatrix} size \\ back\_len \\ mem\_ops + 3 * back\_len \\ back\_len + front\_len \\ nb\_ops \end{bmatrix} \rightarrow \begin{bmatrix} size \\ 0 \\ mem\_ops + 3 * back\_len \\ back\_len + front\_len \\ nb\_ops \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \\ 2 \\ -1 \\ 1 \end{bmatrix}}_{\textbf{dequeue slow (conditional taken)}}
\end{array} \right\}
$$

$mem\_ops$ **grows at most 4 times as quickly as** $nb\_ops$

# State Space Transformation

```
i = 0
while(*) do
   x = x + i + 2
   y = y + i
   i = i + 1
```

# State Space Transformation

```
i = 0
while(*) do
    x = x + i + 2
    y = y + i
    i = i + 1
```

**Transition formula for single iteration of loop**

$$x' = x + i + 2$$
$$y' = y + i$$
$$i' = i + 1$$

**Not representable as $\mathbb{Q}$-VASR**

# State Space Transformation

```
i = 0
while(*) do
   x = x + i + 2
   y = y + i
   i = i + 1
```

**Transition formula for single iteration of loop**

$$x' = x + i + 2$$
$$y' = y + i$$
$$i' = i + 1$$

**Not representable as $\mathbb{Q}$-VASR**

**Can always over-approximate transition formula as $\mathbb{Q}$-VASR by applying a lin. transformation**

# State Space Transformation

```
i = 0
while(*) do
   x = x + i + 2
   y = y + i
   i = i + 1
```

**Transition formula for single iteration of loop**

$$x' = x + i + 2$$
$$y' = y + i$$
$$i' = i + 1$$

**Not representable as ℚ-VASR**

$$\begin{array}{cc} & \begin{array}{ccc} \mathbf{x} & \mathbf{y} & \mathbf{i} \end{array} \\ \begin{array}{c} \textbf{Dim 1} \\ \textbf{Dim 2} \end{array} & \begin{bmatrix} 1 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

$$\left\{ \begin{bmatrix} x-y \\ i \end{bmatrix} \to \begin{bmatrix} x-y \\ i \end{bmatrix} + \begin{bmatrix} 2 \\ 1 \end{bmatrix} \right\}$$

**Can always over-approximate transition formula as ℚ-VASR by applying a lin. transformation**

# Predictable Analysis using $\mathbb{Q}$-VASR Abstractions

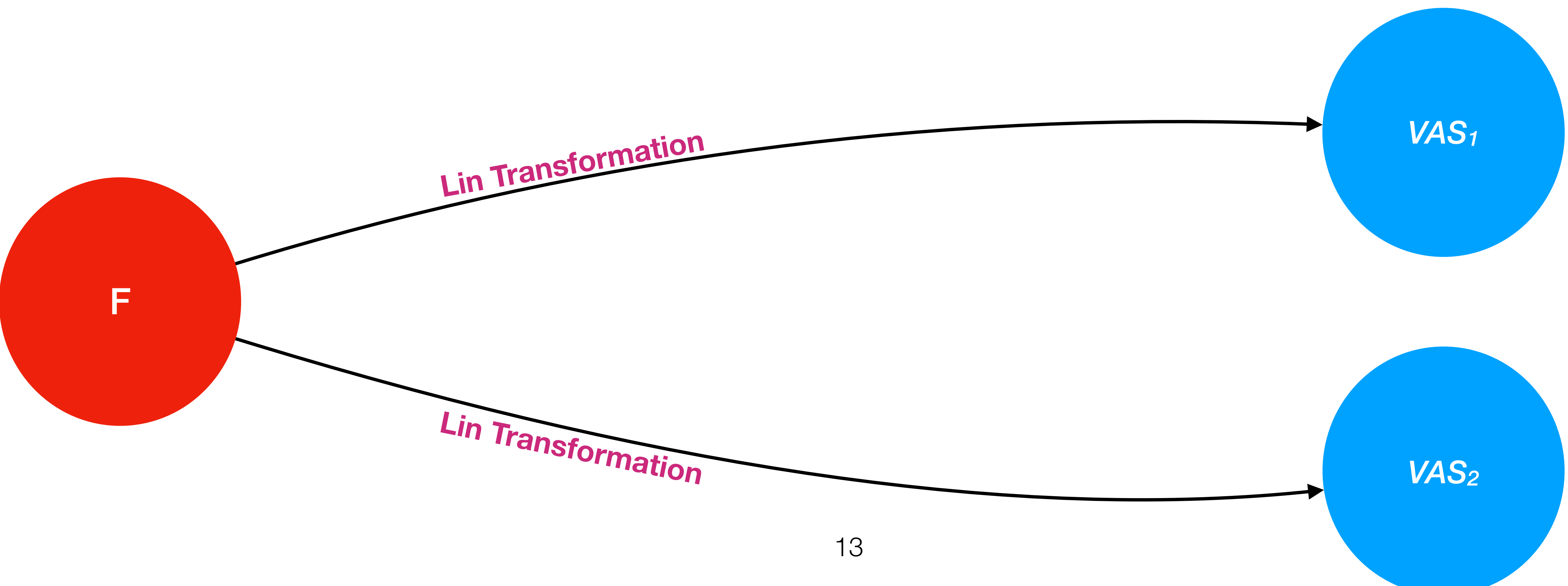
Key Result:

For any LRA transition formula $F$, we can compute a best $\mathbb{Q}$-VASR abstraction of $F$

# Predictable Analysis using $\mathbb{Q}$-VASR Abstractions

Key Result:

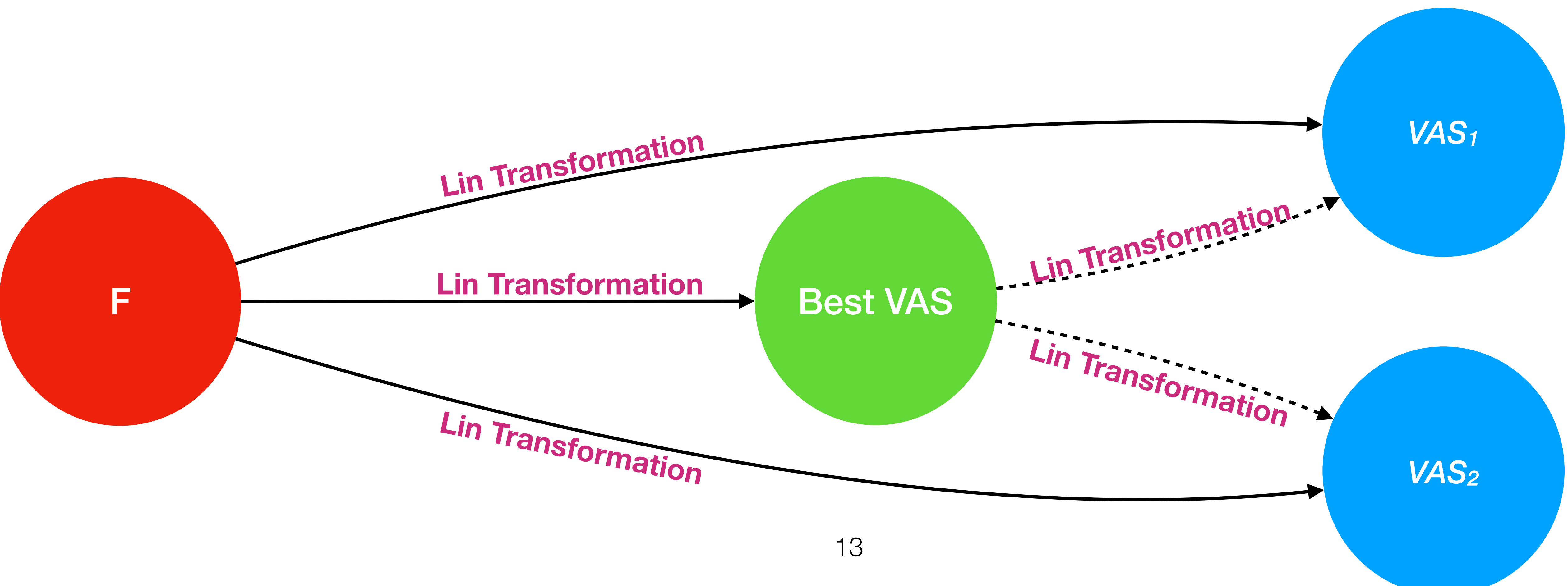
For any LRA transition formula $F$, we can compute a best $\mathbb{Q}$-VASR abstraction of $F$



# Predictable Analysis using $\mathbb{Q}$-VASR Abstractions

Key Result:

For any LRA transition formula $F$, we can compute a best $\mathbb{Q}$-VASR abstraction of $F$



# Computing Best $\mathbb{Q}$-VASR Abstractions

1 **Convert transition formula to DNF**

$$\mathbf{DNF}(F) = C_1 \vee C_2 \vee \ldots C_n$$

# Computing Best $\mathbb{Q}$-VASR Abstractions

1 Convert transition formula to DNF

2 Compute best $\mathbb{Q}$-VASR for each LRA cube

Key contribution

$$VAS\_ABS(C_1) \quad VAS\_ABS(C_2) \quad VAS\_ABS(C_n)$$

$$\textbf{DNF}(F) = C_1 \vee C_2 \vee \ldots C_n$$

# Computing Best $\mathbb{Q}$-VASR Abstractions

1. **Convert transition formula to DNF**
2. **Compute best $\mathbb{Q}$-VASR for each LRA cube**

   Key contribution
3. **Compute best common abstraction of all $\mathbb{Q}$-VASR abstractions**

   Key contribution

$$VAS\_ABS(F)$$

$$\boxed{VAS\_ABS(C_1) \sqcup VAS\_ABS(C_2) \sqcup VAS\_ABS(C_n)}$$

$$\textbf{DNF}(F) = C_1 \vee C_2 \vee \ldots C_n$$

# Computing Best $\mathbb{Q}$-VASR Abstractions

1. Convert transition formula to DNF
2. Compute best $\mathbb{Q}$-VASR for each LRA cube
   Key contribution
3. Compute best common abstraction of all $\mathbb{Q}$-VASR abstractions
   Key contribution

$$VAS\_ABS(F)$$

$$\boxed{VAS\_ABS(C_1) \bigsqcup VAS\_ABS(C_2) \bigsqcup VAS\_ABS(C_n)}$$

$$\textbf{DNF}(F) = C_1 \vee C_2 \vee \ldots C_n$$

1. Step 2 can only compute best $\mathbb{Q}$-VASR for LRA cube
2. Can use SMT solver to enumerate DNF lazily

# This talk

1) Predictable loop summarization using rational vector addition system with resets ($\mathbb{Q}$-VASR)

2) Precision improvement via capturing control flow using $\mathbb{Q}$-VASR with states ($\mathbb{Q}$-VASRS)

# $\mathbb{Q}$-VASRS Abstractions Example

```
int x = 0, i = 0
while(*) do
   if(i%2 == 0)
     i = i + 1
   else
      x = x + 1
      i = i + 1
```

# $\mathbb{Q}$-VASRS Abstractions Example

```
int x = 0, i = 0
while(*) do
   if(i%2 == 0)
     i = i + 1
   else
      x = x + 1
      i = i + 1
```

$$\left\{ \begin{bmatrix} i \\ x \end{bmatrix} \rightarrow \begin{bmatrix} i+1 \\ x \end{bmatrix}, \begin{bmatrix} i \\ x \end{bmatrix} \rightarrow \begin{bmatrix} i+1 \\ x+1 \end{bmatrix} \right\}$$

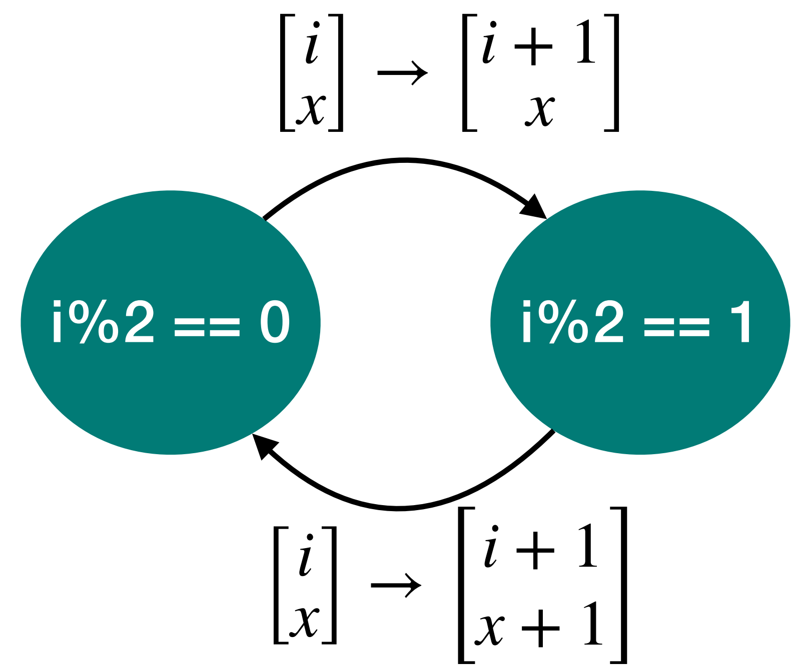
**A Best $\mathbb{Q}$-VASR Abstraction**
**Cannot show $2x \leq i$**

# $\mathbb{Q}$-VASRS Abstractions Example

```
int x = 0, i = 0
while(*) do
   if(i%2 == 0)
     i = i + 1
   else
      x = x + 1
      i = i + 1
```

$$\left\{ \begin{bmatrix} i \\ x \end{bmatrix} \to \begin{bmatrix} i+1 \\ x \end{bmatrix}, \begin{bmatrix} i \\ x \end{bmatrix} \to \begin{bmatrix} i+1 \\ x+1 \end{bmatrix} \right\}$$

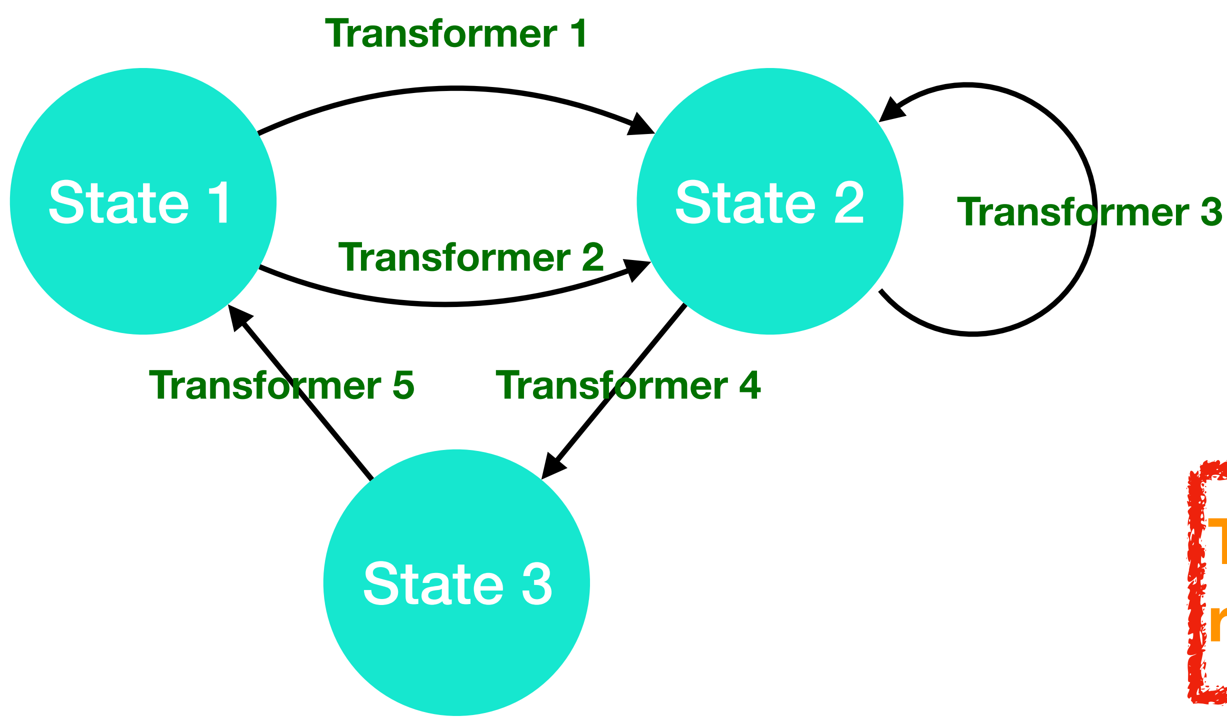
**A Best $\mathbb{Q}$-VASR Abstraction**
**Cannot show 2x ≤ i**




**$\mathbb{Q}$-VASRS Abstraction**

$\mathbb{Q}$-VASRS Abstraction can prove that loop maintains invariant 2x ≤ i

# ℚ-VASRS

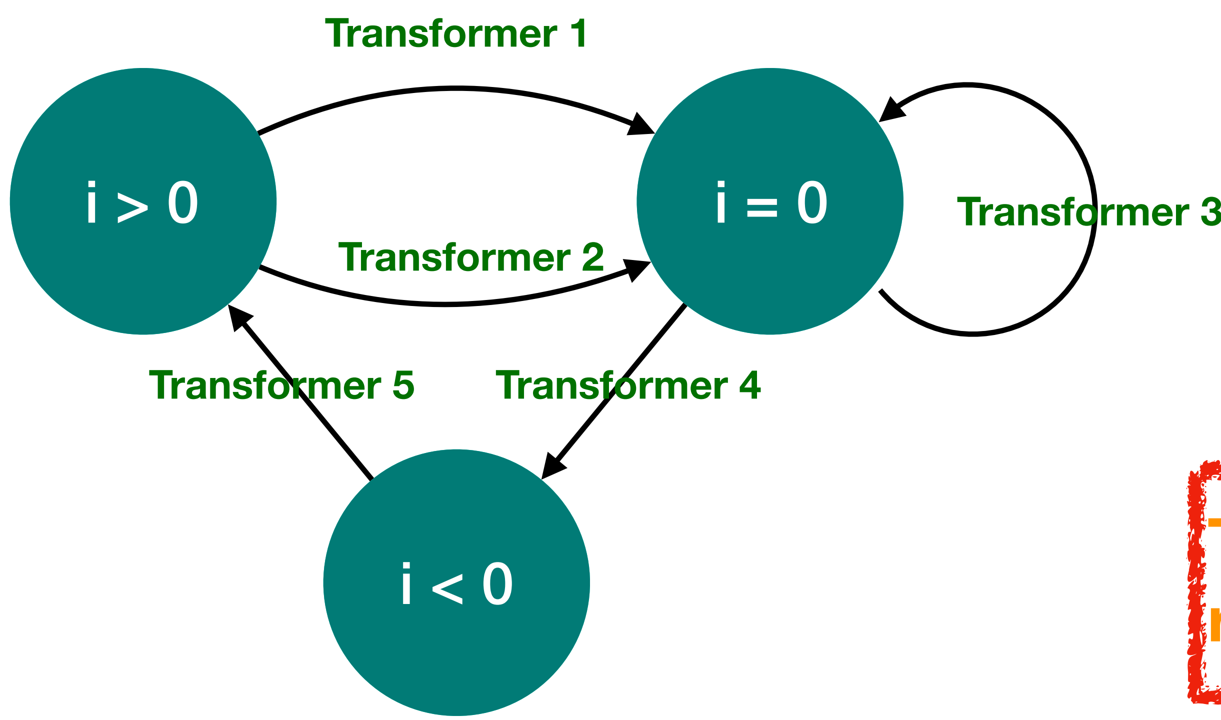



**Reachability relation defined by sequences of transformers that form paths through graph.**

**Transition formula representing reachability relation computable in polytime***

*C. Haase and S. Halfon. Integer vector addition systems with states. in Proc: International Workshop on Reachability Problems (RP '14)

# ℚ-VASRS




**Reachability relation defined by sequences of transformers that form paths through graph.**

**Transition formula representing reachability relation computable in polytime***

**Predicate Q-VASRS:**
**Control States are predicates over program variables.**
**Predicates must partition state space.**

*C. Haase and S. Halfon. Integer vector addition systems with states. in Proc: International Workshop on Reachability Problems (RP '14)

# Best $\mathbb{Q}$-VASRS Abstractions

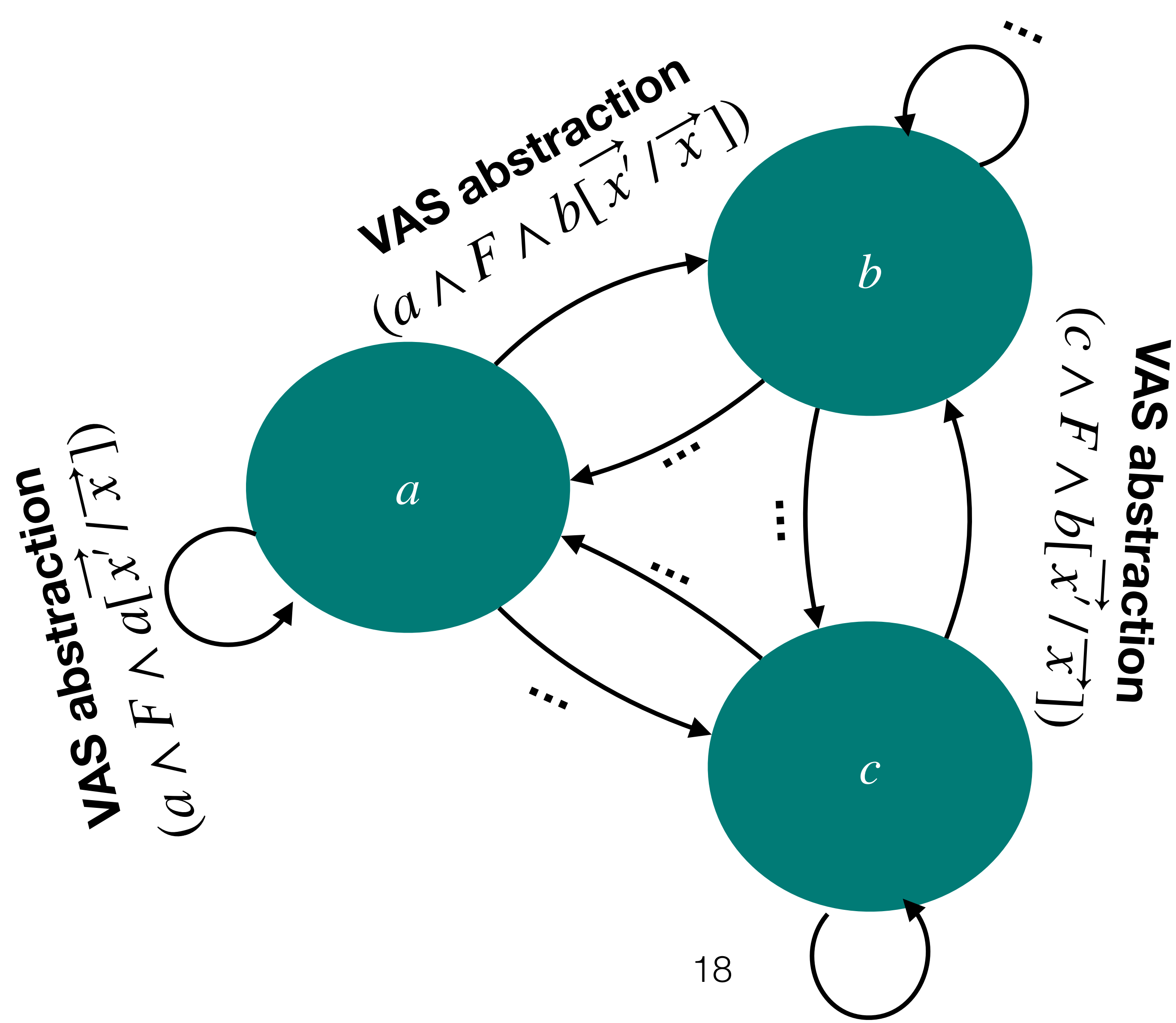
Key Result:

Can compute best $\mathbb{Q}$-VASRS abstraction of input LRA formula $F$ with a fixed set of predicates

# Best ℚ-VASRS Abstractions

Key Result:

Can compute best ℚ-VASRS abstraction of input LRA formula $F$ with a fixed set of predicates



# Predictable $\mathbb{Q}$-VASRS Abstractions

**Finer set of predicates => potentially more precise abstraction**

**No best set of predicates, must settle for a good one**

# Predictable $\mathbb{Q}$-VASRS Abstractions

**Finer set of predicates => potentially more precise abstraction**

**No best set of predicates, must settle for a good one**

**Need monotonicity**

**That if $F \models G$, then $Predicates(F)$ is at least as fine as $Predicates(G)$**

# Predictable $\mathbb{Q}$-VASRS Abstractions

**Finer set of predicates => potentially more precise abstraction**

**No best set of predicates, must settle for a good one**

**Need monotonicity**

**That if $F \models G$, then $Predicates(F)$ is at least as fine as $Predicates(G)$**

**Solution**

**Use connected components of topological closure of $\exists x' . F$ as predicates**

# Predictable $\mathbb{Q}$-VASRS Abstractions

**Finer set of predicates => potentially more precise abstraction**

**No best set of predicates, must settle for a good one**

**Need monotonicity**

**That if $F \models G$, then $Predicates(F)$ is at least as fine as $Predicates(G)$**

**Solution**

**Use connected components of topological closure of $\exists x' . F$ as predicates**

$F$ $\quad$ $F \models G$ $\quad$ $G$

# Predictable $\mathbb{Q}$-VASRS Abstractions

**Finer set of predicates => potentially more precise abstraction**

**No best set of predicates, must settle for a good one**

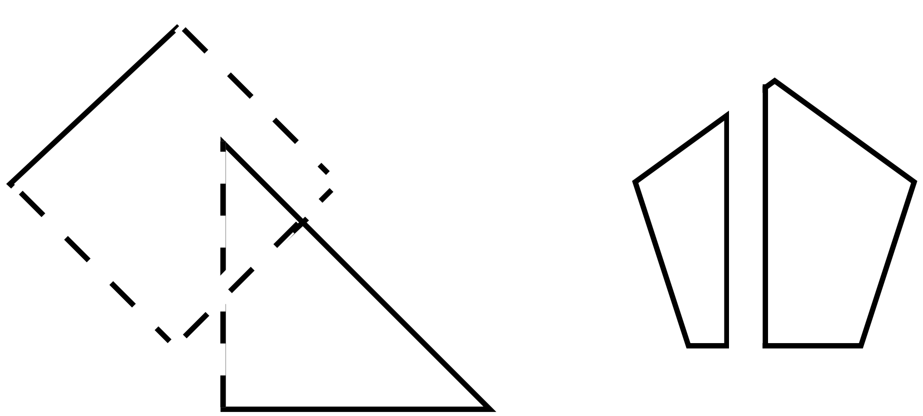
**Need monotonicity**

**That if $F \models G$, then $Predicates(F)$ is at least as fine as $Predicates(G)$**

**Solution**

**Use connected components of topological closure of $\exists x' \,.\, F$ as predicates**

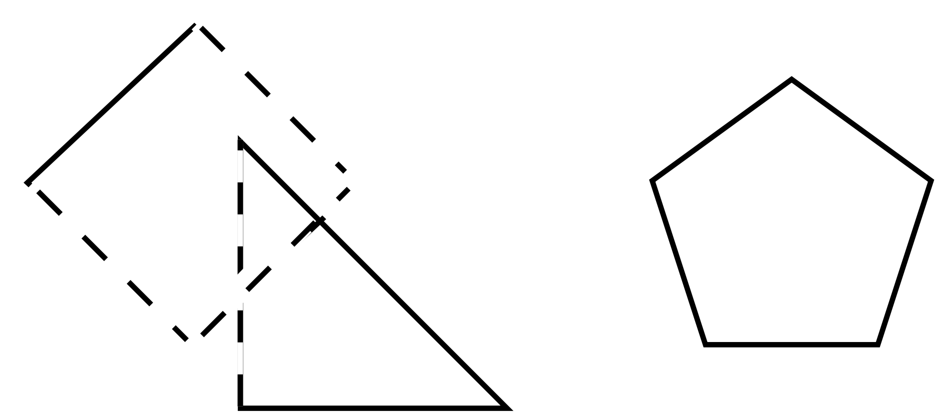
$F$



$F \models G$

1 View formula as finite union of convex polyhedra

$G$



# Predictable $\mathbb{Q}$-VASRS Abstractions

**Finer set of predicates => potentially more precise abstraction**

**No best set of predicates, must settle for a good one**

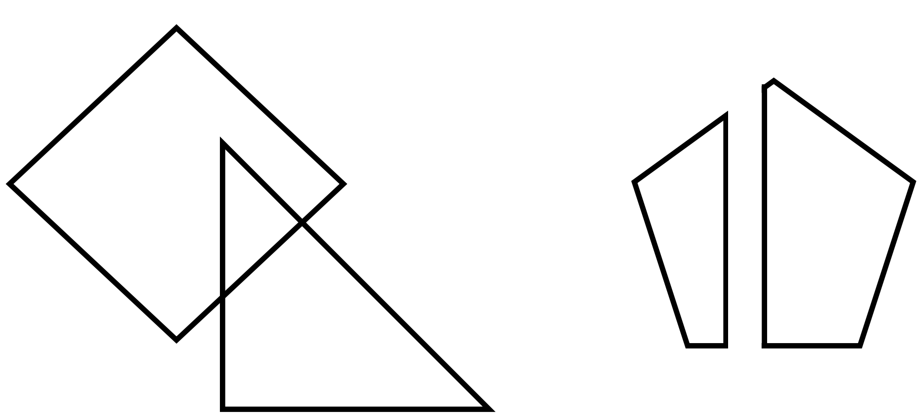
**Need monotonicity**

**That if $F \models G$, then $Predicates(F)$ is at least as fine as $Predicates(G)$**

**Solution**

**Use connected components of topological closure of $\exists x' \,.\, F$ as predicates**

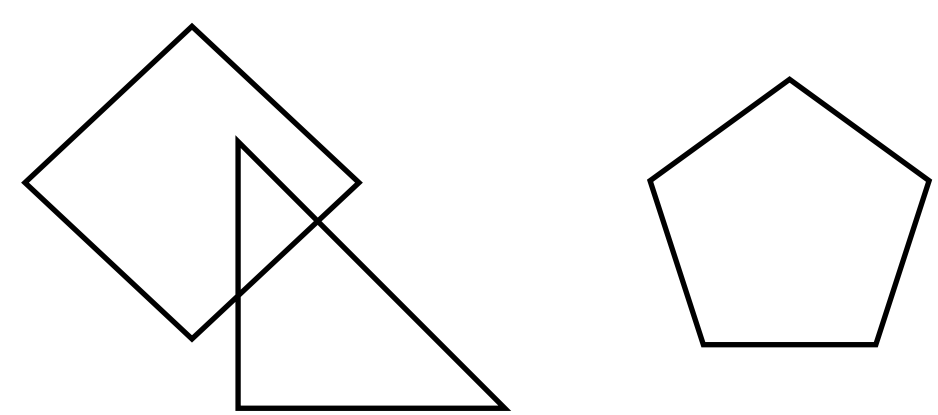
$F$



$F \models G$

1. View formula as finite union of convex polyhedra
2. Convert strict inequalities to non-strict

$G$



# Predictable $\mathbb{Q}$-VASRS Abstractions

**Finer set of predicates => potentially more precise abstraction**

**No best set of predicates, must settle for a good one**

**Need monotonicity**

**That if $F \models G$, then $Predicates(F)$ is at least as fine as $Predicates(G)$**

**Solution**

**Use connected components of topological closure of $\exists x' . F$ as predicates**

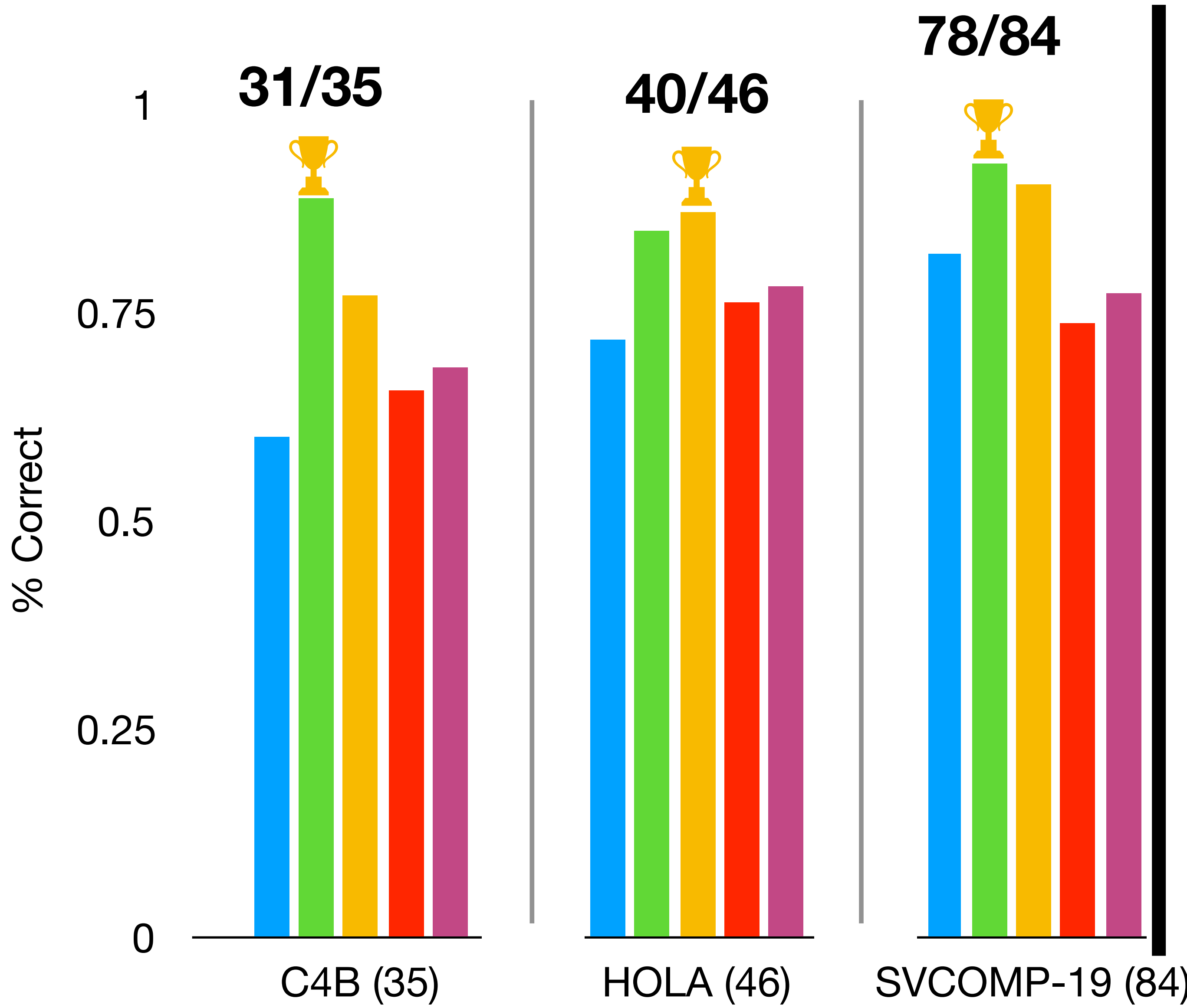
$F$

$F \models G$

1. View formula as finite union of convex polyhedra
2. Convert strict inequalities to non-strict
3. Compute largest connect components

$G$

# Evaluation




| | C4B (35) | HOLA (46) | SVCOMP-19 (84) |
|---|---|---|---|
| ℚ-VASR | 29 S | 50 S | 73 S |
| ℚ-VASRS | 33 S | 65 S | 107 S |
| CRA | 30 S | 56 S | 87 S |
| SeaHorn | 2431 S | 2112 S | 3038 S |
| UltAuto | 3974 S | 3003 S | 6933 S |

**Accuracy** **Runtime**

ℚ-VASR ℚ-VASRS CRA SeaHorn UltAuto

**Results newer than paper version:**
**ℚ-VASR and ℚ-VASRS faster after optimization**
**ℚ-VASR passes two more cases after bug fix**

**Timeout: 300 Seconds per case**
**SVCOMP-19 restricted to safe integer benchmarks from loops category**
**Most accurate tool in any given suite does not subsume all others**

# Summary

- Developed predictable and compositional program analysis with $\mathbb{Q}$-VASR

- Extended analysis with $\mathbb{Q}$-VASR with states to capture control flow information

- Shown improvements in both accuracy and speed over state-of-art-tools while providing guarantees about invariant quality