# A study of the DASH algorithm for software property checking

Kasper Føns

20083881

May 30, 2014

### Table of contents

#### DASH<sub>call</sub>

Goal

ExtendFrontier - Supporting Interprocedural Analysis

#### Implementing ExtendFrontier

Description of InputConstraints
InputConstraints – Missing path constraint
Other problems and final implementation

#### Parallelizing Top-Down Interprocedural Analysis

Top-Down vs Bottom-Up Parallel analysis using BOLT Summaries and how to use them

Parallelizing Top-Down Interprocedural Analysis

Implementing ExtendFrontier

•0

DASH<sub>Call</sub>

```
void test(int x, int y)
 if(x > 0)
    y = 4;
    int q = sum(x, y);
    if(q == 5)
      if(x = 2)
        error;
 int sum(int i, int x)
   int s = i + x;
   return s;
```

•0 00 Goal

```
void test (int x, int y)
  if(x > 0)
                                       q := sum(x, y)
    int q = sum(x, y);
    if(q = 5)
       if(x = 2)
         error;
 int sum(int i, int x)
   int s = i + x;
   return s;
```

DASH<sub>Call</sub> Implementing ExtendFrontier

# Goal

•0 00

```
void test(int x, int y)
  if(x > 0)
                                      q := sum(x, y)
    int q = sum(x, y);
    if(q = 5)
       if(x = 2)
         error;
 int sum(int i, int x)
   int s = i + x;
   return s;
```

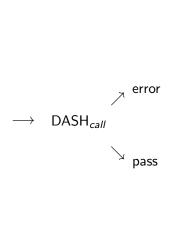
```
DASH<sub>call</sub>
```

Goal

DASH<sub>call</sub>

dua

```
void test(int x, int y)
  if(x > 0)
                                      q := sum(x, y)
    int q = sum(x, y);
    if(q = 5)
       if(x = 2)
         error;
 int sum(int i, int x)
   int s = i + x;
   return s;
```

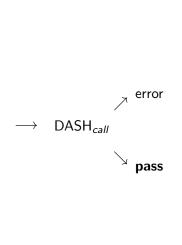


Goal

DASH<sub>call</sub>

Goa

```
void test(int x, int y)
  if(x > 0)
                                      q := sum(x, y)
    int q = sum(x, y);
    if(q = 5)
       if(x = 2)
         error;
 int sum(int i, int x)
   int s = i + x;
   return s;
```

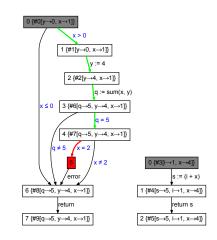


# Changes needed to DASH<sub>int</sub>

- RunTest
   Concrete execution
- ExecuteSymbolic Symbolic execution of traces
  - ConvertToRegionTrace...Generation of traces
- ExtendFrontierProcedure calls at the frontier

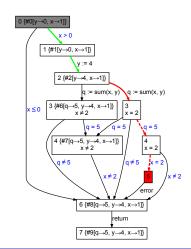
# Changes needed to DASH<sub>int</sub>

- RunTest Concrete execution
- ExecuteSymbolic Symbolic execution of traces
  - ► ConvertToRegionTrace. . . Generation of traces
- ExtendFrontierProcedure calls at the frontier



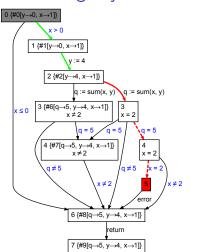
# Changes needed to DASH<sub>int</sub>

- RunTest Concrete execution
- ExecuteSymbolic Symbolic execution of traces
  - ConvertToRegionTrace. . . Generation of traces
- ExtendFrontierProcedure calls at the frontier



•0

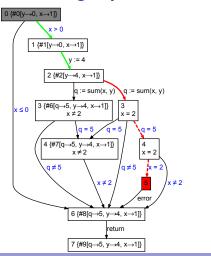
# Extending beyond the frontier



▶ Question: Can SUM generate output *q* when in some state specified by *w*?

•0

# Extending beyond the frontier



- ▶ Question: Can SUM generate output *q* when in some state specified by *w*?
- ► It is a reachability question for which DASH was designed for.

•0

# Extending beyond the frontier

```
\tau_w = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)
       (k-1,k) := \text{Frontier}(\tau_W)
       \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
            \phi' := S_k[e/x]
             \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi^I)
            if r = FAIL then
10.
                  t := m
11:
                  \rho := \mathsf{true}
12.
             else
13.
                  \rho := \mathsf{ComputeRefinePred}(m)
14:
                  t := UNSAT
15:
            end if
16: else
17:
             t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
                  \rho := \mathsf{RefinePred}(S, \tau_w)
19:
20:
             else
21:
                  \rho := \mathsf{true}
22.
             end if
23.
       end if
      return \langle t, \rho \rangle
```

- ► Question: Can SUM generate output *q* when in some state specified by *w*?
- ► It is a reachability question for which DASH was designed for.
- ExtendFrontier modifies the graph of the sub procedure and uses DASH to answer its question.

DASH<sub>Call</sub>

0

# ExtendFrontier – The Idea

1:  $\tau_w = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)$ 

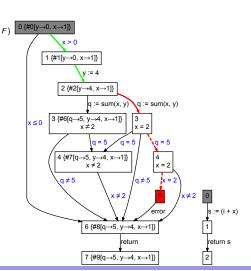
```
(k-1,k) := \text{Frontier}(\tau_W)
       \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{k-1}, S_k))
 5:
 6:
            \phi := InputContraints(S)
 7:
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi^I)
            if r = FAIL then
 9:
10:
                 t := m
11:
                 \rho := \mathsf{true}
12:
            else
13:
                 \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
                 \rho := \mathsf{RefinePred}(S, \tau_w)
19:
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
23.
      end if
24: return \langle t, \rho \rangle
```

DASH<sub>Call</sub>

0

### ExtendFrontier - The Idea

```
1: \tau_w = \langle S_0, S_1, \dots, S_n \rangle := \mathsf{GetWholeAbstractTrace}(\tau_o, F)
       (k-1,k) := \text{Frontier}(\tau_W)
      \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{\nu-1}, S_{\nu}) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi^I)
            if r = FAIL then
10.
                 t := m
11:
                 \rho := true
12.
            else
13.
                 \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                 \rho := RefinePred(S, \tau_w)
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
23.
       end if
24: return \langle t, \rho \rangle
```



DASH<sub>Call</sub>

0

#### ExtendFrontier - The Idea

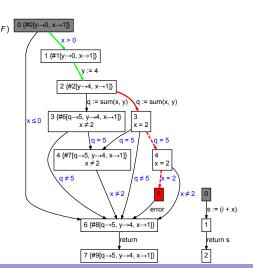
```
0 {#0[y→0, x→1]}
 1: \tau_w = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)
      (k-1,k) := Frontier(\tau_W)
                                                                                                              x > 0
     \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
      if Edge(S_{\nu-1}, S_{\nu}) \in CallReturn(E) then
                                                                                                       1 {#1[y→0, x→1]}
           let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
                                                                                                                     v := 4
           \phi := InputContraints(S)
                                                                                                             2 {#2[y→4, x→1]}
           \phi' := S_k[e/x]
           \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi^I)
                                                                                                                        a := sum(x, v)
                                                                                                                                         q := sum(x, y)
           if r = FAIL then
                                                                                                         3 {#6[q→5, y→4, x→1]}
10.
                t := m
                                                                                                                                         3
                                                                                                x ≤ 0
                                                                                                                                         x = 2
                                                                                                                     x ≠ 2
11:
                \rho := true
12.
           else
13.
                \rho := \mathsf{ComputeRefinePred}(m)
14:
                                                                                                            4 {#7[q→5, y→4, x→1]}
                t := UNSAT
15:
           end if
                                                                                                                       x \neq 2
16: else
                                                                                                            q \neq 5
                                                                                                                                          q ≠ 5
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
           if t = UNSAT then
                                                                                                                                                            x ≠ 2
                                                                                                                                x \neq 2
19:
                \rho := RefinePred(S, \tau_w)
20:
            else
21:
                \rho := \mathsf{true}
                                                                                                                       6 {#8[q→5, y→4, x→1]}
22.
           end if
23.
      end if
                                                                                                                                                                       return s
                                                                                                                                     return
24: return \langle t, \rho \rangle
                                                                                                                       7 {#9[q→5, y→4, x→1]}
```

DASH<sub>Call</sub>

0

### ExtendFrontier - The Idea

```
1: \tau_w = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)
      (k-1,k) := Frontier(\tau_W)
      \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
      if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi^I)
            if r = FAIL then
10.
                 t := m
11:
                 \rho := true
12.
            else
13.
                 \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                 \rho := RefinePred(S, \tau_w)
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
23.
      end if
24: return \langle t, \rho \rangle
```



DASH<sub>Call</sub>

0

# ExtendFrontier – The Idea

1:  $\tau_W = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)$ 

```
(k-1,k) := \text{Frontier}(\tau_W)
      \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \mathsf{ExecuteSymbolic}(\tau_{\mathsf{w}}, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma', \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{k-1}, S_k))
 5:
            \phi := InputContraints(S)
 7:
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
            if r = FAIL then
10:
                  t := m
11:
                  \rho := \mathsf{true}
12.
            else
13.
                  \rho := \mathsf{ComputeRefinePred}(m)
14:
                  t := UNSAT
15:
            end if
16: else
17:
             t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                  \rho := RefinePred(S, \tau_w)
20:
             else
21:
                  \rho := \mathsf{true}
22.
             end if
23.
       end if
24: return \langle t, \rho \rangle
```

### ExecuteSymbolic:

$$\phi_1 = x_0 > 0$$

$$\triangleright$$
  $S = \{x \mapsto x_0, y \mapsto 4\}$ 

$$\blacktriangleright \phi_2 = true$$

0

# ExtendFrontier – The Idea

1:  $\tau_W = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)$ 

```
(k-1,k) := \text{Frontier}(\tau_W)
       \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{k-1}, S_k))
 5:
            \phi := InputContraints(S)
 7:
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
            if r = FAIL then
10:
                 t := m
11:
                 \rho := \mathsf{true}
12.
            else
13.
                 \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                 \rho := RefinePred(S, \tau_w)
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
23.
       end if
24: return \langle t, \rho \rangle
```

### ExecuteSymbolic:

$$\phi_1 = x_0 > 0$$

$$\triangleright$$
  $S = \{x \mapsto x_0, y \mapsto 4\}$ 

$$ightharpoonup \phi_2 = true$$

0

# ExtendFrontier – The Idea

 $\tau_{w} = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)$ 

```
(k-1,k) := \text{Frontier}(\tau_W)
       \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
 7:
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
 9:
            if r = FAIL then
10:
                 t := m
11:
                 \rho := \mathsf{true}
12.
            else
13.
                 \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                 \rho := RefinePred(S, \tau_w)
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
23.
       end if
24: return \langle t, \rho \rangle
```

### ExecuteSymbolic:

$$\phi_1 = x_0 > 0$$

$$\triangleright$$
  $S = \{x \mapsto x_0, y \mapsto 4\}$ 

$$\bullet$$
  $\phi_2 = true$ 

0

### ExtendFrontier - The Idea

 $\tau_w = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)$ 

```
(k-1,k) := \text{Frontier}(\tau_W)
       \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
            if r = FAIL then
10.
                 t := m
11:
                 \rho := true
12.
            else
13.
                 \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                 \rho := RefinePred(S, \tau_w)
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
23.
       end if
24: return \langle t, \rho \rangle
```

#### ExecuteSymbolic:

$$\phi_1 = x_0 > 0$$

$$\triangleright$$
  $S = \{x \mapsto x_0, y \mapsto 4\}$ 

$$\blacktriangleright \phi_2 = true$$

$$\phi = \phi_{ic}$$

0

ExtendFrontier – Supporting Interprocedural Analysis

# ExtendFrontier - The Idea

 $\tau_{w} = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)$ 

```
(k-1,k) := \text{Frontier}(\tau_W)
       \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
            if r = FAIL then
10.
                 t := m
11:
                 \rho := true
12.
            else
13.
                 \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                 \rho := RefinePred(S, \tau_w)
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
23.
       end if
24: return \langle t, \rho \rangle
```

#### ExecuteSymbolic:

$$\phi_1 = x_0 > 0$$

$$\triangleright$$
  $S = \{x \mapsto x_0, y \mapsto 4\}$ 

$$\phi_2 = true$$

$$\phi = \phi_{ic}$$

$$\phi' = \phi_{ec}$$

Implementing ExtendFrontier

000

0000000

000

ExtendFrontier - Supporting Interprocedural Analysis

DASH<sub>Call</sub>

0

### ExtendFrontier - The Idea

```
\tau_{w} = \langle S_{0}, S_{1}, \dots, S_{n} \rangle := \text{GetWholeAbstractTrace}(\tau_{o}, F)
       (k-1,k) := \text{Frontier}(\tau_W)
      \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
            if r = FAIL then
10.
                 t := m
11:
                 \rho := true
12.
            else
13.
                 \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                 \rho := RefinePred(S, \tau_w)
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
23.
       end if
24: return \langle t, \rho \rangle
```

#### ExecuteSymbolic:

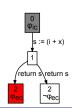
$$\phi_1 = x_0 > 0$$

$$\triangleright$$
  $S = \{x \mapsto x_0, y \mapsto 4\}$ 

$$\bullet$$
  $\phi_2 = true$ 

$$ightharpoonup \phi = \phi_{ic}$$





DASH<sub>Call</sub>

0

### ExtendFrontier - The Idea

```
\tau_{w} = \langle S_{0}, S_{1}, \dots, S_{n} \rangle := \text{GetWholeAbstractTrace}(\tau_{o}, F)
       (k-1,k) := \text{Frontier}(\tau_W)
      \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
            if r = FAIL then
10.
                  t := m
11:
                  \rho := \mathsf{true}
12.
            else
13.
                  \rho := \mathsf{ComputeRefinePred}(m)
14:
                  t := UNSAT
15:
            end if
16: else
17:
             t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                  \rho := RefinePred(S, \tau_w)
20:
             else
21:
                  \rho := \mathsf{true}
22.
             end if
23.
       end if
24: return \langle t, \rho \rangle
```

#### ExecuteSymbolic:

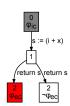
$$\phi_1 = x_0 > 0$$

$$\triangleright$$
  $S = \{x \mapsto x_0, y \mapsto 4\}$ 

$$\phi_2 = true$$

$$\phi = \phi_{ic}$$





DASH<sub>Call</sub>

0

### ExtendFrontier - The Idea

```
\tau_{w} = \langle S_{0}, S_{1}, \dots, S_{n} \rangle := \text{GetWholeAbstractTrace}(\tau_{o}, F)
       (k-1,k) := \text{Frontier}(\tau_W)
      \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
            if r = FAIL then
10.
                  t := m
11:
                  \rho := \mathsf{true}
12.
            else
13.
                  \rho := \mathsf{ComputeRefinePred}(m)
14:
                  t := UNSAT
15:
            end if
16: else
17:
             t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                  \rho := RefinePred(S, \tau_w)
20:
             else
21:
                  \rho := \mathsf{true}
22.
             end if
23.
       end if
24: return \langle t, \rho \rangle
```

#### ExecuteSymbolic:

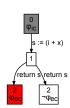
$$\phi_1 = x_0 > 0$$

$$\triangleright$$
  $S = \{x \mapsto x_0, y \mapsto 4\}$ 

$$\phi_2 = true$$

$$\phi = \phi_{ic}$$





# InputConstraints

```
\tau_w = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)
       (k-1,k) := \text{Frontier}(\tau_W)
       \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
 5:
            \phi := InputContraints(S)
            \phi' := S_k[e/x]
             \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi^I)
            if r = \text{FAIL} then
10.
                  t := m
11:
                  \rho := \mathsf{true}
12.
            else
13.
                  \rho := \mathsf{ComputeRefinePred}(m)
14:
                  t := UNSAT
15.
            end if
16: else
17:
             t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
                  \rho := \mathsf{RefinePred}(S, \tau_w)
19.
20:
             else
21:
                  \rho := \mathsf{true}
22.
             end if
23.
       end if
      return \langle t, \rho \rangle
```

- InputConstraints takes the symbolic map S as the only parameter. They write:
- The predicate φ corresponds to the constraints on Q's input variables which are computed directly from the symbolic map S (by the auxiliary function InputConstraints [...])
- ► When FAIL is reported, the test input *m* is returned unmodified.

Description of InputConstraints

# InputConstraints implementation

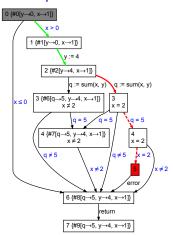
Given the below quote, how do we implement InputConstraints?

► The predicate φ corresponds to the constraints on Q's input variables which are computed directly from the symbolic map S (by the auxiliary function InputConstraints [...])

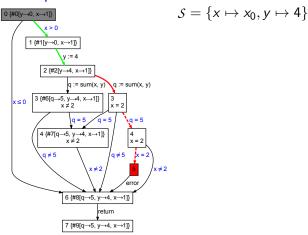
```
v := f(a_1, \ldots, a_n)
int f(int p_1, \ldots, \text{ int } p_n) \{\ldots\}
\phi_{ic} := InputConstraints(S)
     := \left( \bigwedge_{p_i \in \text{params}(Q)} p_i = \text{SymbolicEval}(a_i, \mathcal{S}) \right)
     :=p_1 = SymbolicEval(a_1, S) \land
         p_n = SymbolicEval(a_n, S)
```

Description of InputConstraints

### Example



# Example



Description of InputConstraints

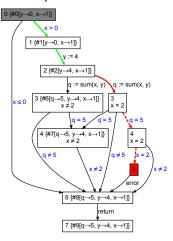
# Example

```
0 {#0[y→0, x→1]}
          1 {#1[y→0, x→1]}
                         y := 4
                2 {#2[y→4, x→1]}
                           q := sum(x, y) q := sum(x, y)
             3 {#6[q→5, y→4, x→1]}
x ≠ 2
    x ≤ 0
                                            x = 2
               4 {#7[q\rightarrow5, y\rightarrow4, x\rightarrow1]}
x \neq 2
                                                    x = 2
                                   x ≠ 2
                         6 {#8[q→5, y→4, x→1]]
                                        return
                         7 {#9[q→5, y→4, x→1]}
```

```
S = \{x \mapsto x_0, y \mapsto 4\}
int sum(int i, int x)
   int s = i + x;
   return s:
```

Description of InputConstraints

# Example



```
S = \{x \mapsto x_0, y \mapsto 4\}
int sum(int i, int x)
   int s = i + x:
   return s:
\phi_{ic} := InputConstraints(S)
     := \left( \bigwedge_{p_i \in \text{params}(Q)} p_i = \text{SymbolicEval}(a_i, S) \right)
```

### Example

```
0 {#0[y→0, x→1]}
          1 {#1[y→0, x→1]}
                2 {#2[y→4, x→1]}
                          q := sum(x, y) q := sum(x, y)
            3 {#6[q→5, y→4, x→1]}
x ≠ 2
   |x \le 0
                                           x = 2
               4 {#7[q\rightarrow5, y\rightarrow4, x\rightarrow1]}
                                  x ≠ 2
                         6 {#8[q→5, y→4, x→1]]
                                       return
                         7 {#9[q→5, y→4, x→1]}
```

```
S = \{x \mapsto x_0, y \mapsto 4\}
int sum(int i, int x)
   int s = i + x:
   return s:
\phi_{ic} := InputConstraints(S)
    :=\left(igwedge_{p_i\in\mathsf{params}(Q)}p_i=\mathsf{SymbolicEval}(a_i,\mathcal{S})
ight)
    := i = SymbolicEval(x, s) \land
        x = SymbolicEval(y, S)
```

Description of InputConstraints

# Example

```
0 {#0[y→0, x→1]}
          1 {#1[y→0, x→1]}
                2 {#2[y→4, x→1]}
                          q := sum(x, y) q := sum(x, y)
            3 {#6[q→5, y→4, x→1]}
x ≠ 2
    |x \le 0
               4 {#7[q\rightarrow5, y\rightarrow4, x\rightarrow1]}
                                  x ≠ 2
                         6 {#8[q→5, y→4, x→1]]
                                       return
                         7 {#9[q→5, y→4, x→1]}
```

```
S = \{x \mapsto x_0, y \mapsto 4\}
int sum(int i, int x)
   int s = i + x:
   return s:
\phi_{ic} := InputConstraints(S)
    :=\left(igwedge_{p_i\in\mathsf{params}(Q)}p_i=\mathsf{SymbolicEval}(a_i,\mathcal{S})
ight)
    := i = SymbolicEval(x, s) \land
        x = SymbolicEval(y, S)
    := i = x_0 \land x = 4
```

InputConstraints - Missing path constraint

# Missing path constraint examples – TestAbs

```
int abs(int a)
{
   if (a < 0)
     return -a;
   return a;
}</pre>
```

► Returns
-2147483648
when given as input.

InputConstraints - Missing path constraint

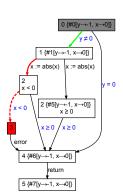
# Missing path constraint examples - TestAbs

InputConstraints - Missing path constraint

# Missing path constraint examples – TestAbs

Returns
-2147483648
when given as input.

```
void testabs(int x, int y)
{
    if(y \neq 0)
    {
        x = abs(x);
        if(x < 0)
        error;
    }
}</pre>
```



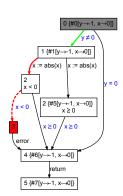
## Missing path constraint examples - TestAbs

```
int abs(int a)
{
    if (a < 0)
        return -a;
    return a;
}</pre>
```

► Returns
-2147483648
when given as input.

```
void testabs(int x, int y)
{
    if(y \neq 0)
    {
        x = abs(x);
        if(x < 0)
        error;
    }
}</pre>
```

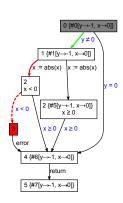
Solution returned by abs does not mention y.



## Missing path constraint examples - TestAbs

```
int abs(int a)
{
    if (a < 0)
        return -a;
    return a;
}</pre>
```

- ► Returns
  -2147483648
  when given as input.
- void testabs(int x, int y)
  {
   if(y \neq 0)
   {
   x = abs(x);
   if(x < 0)
   error;
   }
  }</pre>
  - Solution returned by abs does not mention *y*.
  - ► Call SAT solver again to find value for *y*!



## Missing path constraint examples – TestAbs2

```
int abs(int a)
{
   if (a < 0)
      return -a;
   return a;
}</pre>
```

► Returns
-2147483648
when given as input.

## Missing path constraint examples – TestAbs2

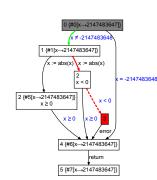
## Missing path constraint examples – TestAbs2

000000

```
int abs(int a)
  if (a < 0)
    return -a:
  return a;
```

Returns -2147483648when given as input.

```
void testabs2(int x)
  if (x \neq -2147483648)
    x = abs(x);
    if (x < 0)
       error:
```



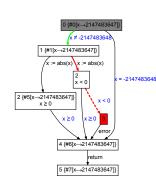
## Missing path constraint examples – TestAbs2

```
int abs(int a)
{
    if (a < 0)
        return -a;
    return a;
}</pre>
```

➤ Returns
-2147483648
when given as input.

```
void testabs2(int x)
{
   if(x \neq -2147483648)
   {
      x = abs(x);
      if(x < 0)
        error;
}</pre>
```

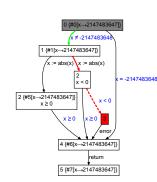
x = -2147483648, which is prohibited!



## Missing path constraint examples – TestAbs2

```
int abs(int a)
{
    if (a < 0)
        return -a;
    return a;
}</pre>
```

- ► Returns
  -2147483648
  when given as input.
- void testabs2(int x)
  {
   if(x \neq -2147483648)
   {
   x = abs(x);
   if(x < 0)
   error;
  }</pre>
  - x = -2147483648, which is prohibited!
  - Must include path constraint when analyzing abs.



# Adding the path constraint $\phi_1$

```
1: \tau_W = \langle S_0, S_1, \dots, S_n \rangle :=
       GetWholeAbstractTrace(\tau_0, F)
 2: (k-1,k) := \text{Frontier}(\tau_w)
 3: \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_W, P)
     if Edge(S_{k-1}, S_k) \in CallReturn(E) then
           let
       \langle \Sigma, \sigma^I, \rightarrow \rangle = \mathsf{GetProc}(\mathsf{Edge}(S_{k-1}, S_k))
                                                                      Include the path constraint \phi_1:
           \phi := InputContraints(S, \phi_1)
           \phi' := S_k[e/x]
           \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
                                                                      \phi_{ic} := InputConstraints(S, \phi_1)
           if r = FAIL then
10:
                t := m
                                                                             :=\left(igwedge_{p_i\in\mathsf{params}(Q)}p_i=\mathsf{SymbolicEval}(a_i,\mathcal{S})
ight)
11:
                \rho := \mathsf{true}
12:
           else
13:
                \rho := \mathsf{ComputeRefinePred}(m)
14:
                t := UNSAT
15:
           end if
16: else
17:
           t := IsSAT(\phi_1, S, \phi_2, P)
18:
           if t = UNSAT then
19.
                \rho := \mathsf{RefinePred}(S, \tau_w)
20.
           else
21:
                \rho := true
22.
           end if
23:
      end if
```

24: return  $\langle t, \rho \rangle$ 

DASH<sub>call</sub>

# Missing symbolic variables in solution

Implementing ExtendFrontier

```
void foo()
{
  int y = 4;
  int x = zero();
  if(x == y)
      error;
}

int zero()
{
  return 0;
}
```

## Missing symbolic variables in solution

0000000

```
void foo()
                                      y := 4
                                 1 {#1[y→4]}
   int y = 4;
                                 x := zero()\x := zero()
   int x = zero()
   if(x = y)
                               x = y
      error;
                                    2 {#4[y→4, x→0]}
                                         x \neq y
                                  x \neq v
int zero()
                             error
                           4 {#5[y→4, x→0]}
   return 0:
                                  return
                           5 {#6[y→4, x→0]}
```

### Missing symbolic variables in solution

```
void foo()
                                           y := 4
                                      1 {#1[y→4]}
   int y = 4;
                                      x := zero()\x := zero()
   int x = zero()
   if(x = y)
                                   x = y
       error;
                                         2 {#4[y-4, x-0]}
                                               x \neq v
                                       x \neq v
int zero()
                                 error
                               4 \{ \#5[y \rightarrow 4, x \rightarrow 0] \}
   return 0:
                                       return
                               5 {#6[y→4, x→0]}
```

▶ Input constraint does not include *y* since it is not part of the arguments to zero.

### Missing symbolic variables in solution

```
void foo()
                                      y := 4
                                 1 {#1[v→4]}
   int y = 4;
                                 x := zero()\x := zero()
   int x = zero()
   if(x = y)
                               x = y
      error;
                                    2 {#4[y-4, x-0]}
                                         x \neq v
                                  x \neq v
int zero()
                             error
                           4 {#5[y→4, x→0]}
   return 0:
                                  return
                           5 {#6[y→4, x→0]}
```

- ▶ Input constraint does not include y since it is not part of the arguments to zero.
- ► The analysis of zero does not know that y must be 4 and returns a solution where  $y \mapsto 0!$

### Missing symbolic variables in solution

```
void foo()
                                      y := 4
                                 1 {#1[v→4]}
   int y = 4;
                                 x := zero()\x := zero()
   int \times = zero()
   if(x = y)
                               x = y
      error;
                                    2 {#4[y→4, x→0]}
                                         x \neq v
                                  x≠v
                                        x≠v
int zero()
                             error
                           4 {#5[y→4, x→0]}
   return 0:
                                  return
                           5 {#6[y→4, x→0]}
```

- ▶ Input constraint does not include y since it is not part of the arguments to zero.
- ► The analysis of zero does not know that y must be 4 and returns a solution where  $y \mapsto 0$ !
- Must include all symbolic variables in constraint, such that zero knows how they are bound.

## Other problems with InputConstraints

► Variables used in the input constraint must be linked together with the exit constraint.

## Other problems with InputConstraints

▶ Variables used in the input constraint must be linked together with the exit constraint. → Need a two-step process when constructing the input constraint.

## Other problems with InputConstraints

- ▶ Variables used in the input constraint must be linked together with the exit constraint. → Need a two-step process when constructing the input constraint.
- ▶ What about renaming? When *x* is used both in the caller and called procedure?

## Other problems with InputConstraints

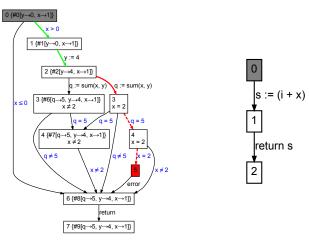
- ▶ Variables used in the input constraint must be linked together with the exit constraint. → Need a two-step process when constructing the input constraint.
- ▶ What about renaming? When x is used both in the caller and called procedure?  $\mapsto$  Rename external variables using a renamer  $\pi$ :  $\pi(a) = 1 \downarrow a$

# The fix to InputConstraints

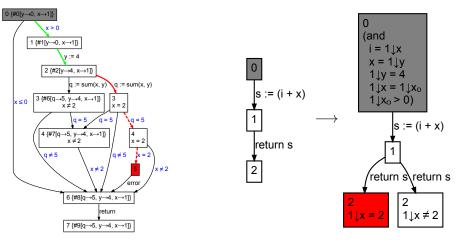
```
1: \tau_w = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_0, F)
      (k-1,k) := \text{Frontier}(\tau_w)
      \langle \phi_1, \mathcal{S}, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
      if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{k-1}, S_k))
          \phi := InputContraints(S, \phi_1)
          \phi' := S_k[e/x]
           \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi')
            if r = FAIL then
10:
                t := m
11:
                 \rho := \mathsf{true}
12.
            else
13.
                  \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19.
                  \rho := \mathsf{RefinePred}(S, \tau_w)
20.
21:
                  \rho := \mathsf{true}
22.
            end if
23.
       end if
24: return \langle t, \rho \rangle
```

$$egin{aligned} \phi_{\mathit{ic}} &:= \mathsf{InputConstraints}(\mathcal{S}, \phi_1) \ &:= \left(igwedge_{p_i \in \mathsf{params}(Q)} p_i = \pi(a_i)
ight) \land \ &\left(igwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e)
ight) \land \ &\pi(\phi_1) \end{aligned}$$

### Input- and exit-constraints on analyzed graph



### Input- and exit-constraints on analyzed graph



•00

#### Extract test input

Other problems and final implementation

DASH<sub>call</sub>

```
1: \tau_w = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)
  2: (k − 1, k) := Frontier(τ<sub>W</sub>)
  3: \langle \phi_1, S, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
            \phi := InputContraints(S, \phi_1)
            \phi' := S_k[e/x]
             \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi^I)
            if r = FAIL then
10.
                  t := m
11:
                  \rho := \mathsf{true}
12:
            else
13:
                  \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16: else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
            if t = UNSAT then
18:
                  \rho := RefinePred(S, \tau_w)
19:
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
23.
       end if
       return \langle t, \rho \rangle
```

### Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

▶ Initial symbolic variables  $v_0$  for all parameters of P'.

### Extract test input

- ▶ Initial symbolic variables  $v_0$  for all parameters of P'.
- ▶ Values for the variables mentioned in the input constraint  $\phi_{ic}$ :

### Extract test input

- Initial symbolic variables v₀ for all parameters of P'.
- Values for the variables mentioned in the input constraint *φ<sub>ic</sub>*:
  - Parameters and local variables for P.

### Extract test input

- Initial symbolic variables v₀ for all parameters of P'.
- Values for the variables mentioned in the input constraint *φ<sub>ic</sub>*:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.

### Extract test input

- Initial symbolic variables v₀ for all parameters of P'.
- ► Values for the variables mentioned in the input constraint  $\phi_{ic}$ :
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

### Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- ▶ Initial symbolic variables  $v_0$  for all parameters of P'.
- ► Values for the variables mentioned in the input constraint  $\phi_{ic}$ :
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

### Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- ► Initial symbolic variables  $v_0$  for all parameters of P'.
- ▶ Values for the variables mentioned in the input constraint  $\phi_{ic}$ :
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

### Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- ▶ Initial symbolic variables  $v_0$  for all parameters of P'.
- ► Values for the variables mentioned in the input constraint  $\phi_{ic}$ :
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

### Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

t:= m

- ► Initial symbolic variables v<sub>0</sub> for all parameters of P'.
- ▶ Values for the variables mentioned in the input constraint  $\phi_{ic}$ :
  - ► Parameters and local variables for *P*.
  - Initial symbolic variables for parameters for P.
  - ▶ Variables mentioned in the input constraint for P (if any).

#### Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- ► Initial symbolic variables  $v_0$  for all parameters of P'.
- Values for the variables mentioned in the input constraint φ<sub>ic</sub>:
  - Parameters and local variables for P.
  - ► Initial symbolic variables for parameters for *P*.
  - ► Variables mentioned in the input constraint for *P* (if any).

$$t := m \setminus \{v_0 \mid \forall v \in \mathsf{params}(P')\}$$

#### Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- ► Initial symbolic variables  $v_0$  for all parameters of P'.
- Values for the variables mentioned in the input constraint φ<sub>ic</sub>:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - ► Variables mentioned in the input constraint for *P* (if any).

$$t := \pi^{-1} \Big( m \setminus \{ v_0 \mid \forall v \in \mathsf{params}(P') \} \Big)$$

#### Extract test input

Goal: Extract test input for P when test input m for sub procedure P' is returned. m contains:

- ▶ Initial symbolic variables  $v_0$  for all parameters of P'.
- Values for the variables mentioned in the input constraint φ<sub>ic</sub>:
  - Parameters and local variables for P.
  - Initial symbolic variables for parameters for P.
  - Variables mentioned in the input constraint for P (if any).

$$t := \pi^{-1} \Big( m \setminus \{ v_0 \mid \forall v \in \mathsf{params}(P') \} \Big) \setminus$$

$$(\mathsf{locals}(P) \cup \mathsf{params}(P))$$

DASH<sub>call</sub>

### ComputeRefinePred

Goal: Extract refinement predicate  $\rho$ from proof m that shows that  $S_k$ cannot be reached in P'.

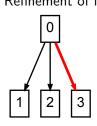
0.00

```
1: \tau_w = \langle S_0, S_1, \dots, S_n \rangle := \text{GetWholeAbstractTrace}(\tau_o, F)
  2: (k-1,k) := \text{Frontier}(\tau_W)
  3: \langle \phi_1, S, \phi_2 \rangle := \text{ExecuteSymbolic}(\tau_w, P)
       if Edge(S_{k-1}, S_k) \in CallReturn(E) then
            let \langle \Sigma, \sigma^I, \rightarrow \rangle = \text{GetProc}(\text{Edge}(S_{\nu-1}, S_{\nu}))
            \phi := InputContraints(S, \phi_1)
            \phi' := S_k[e/x]
            \langle r, m \rangle := \mathsf{DASH}(\langle \Sigma, \sigma^I \wedge \phi, \rightarrow \rangle, \neg \phi^I)
            if r = FAIL then
10.
                  t := m
11:
                 \rho := \mathsf{true}
12.
            else
13.
                  \rho := \mathsf{ComputeRefinePred}(m)
14:
                 t := UNSAT
15:
            end if
16.
       else
17:
            t := IsSAT(\phi_1, S, \phi_2, P)
18:
            if t = UNSAT then
19:
                  \rho := \text{RefinePred}(S, \tau_w)
20:
            else
21:
                 \rho := \mathsf{true}
22.
            end if
       end if
       return \langle t, \rho \rangle
```

### ComputeRefinePred

Goal: Extract refinement predicate  $\rho$  Refinement of initial region in P': from proof m that shows that  $S_k$ cannot be reached in P'.

▶ Predicate  $\neg p_i$  removes edge to some error region in P'

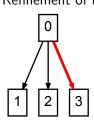


### ComputeRefinePred

Goal: Extract refinement predicate  $\rho$  Refinement of initial region in P': from proof m that shows that  $S_k$  cannot be reached in P'.

▶ Predicate  $\neg p_i$  removes edge to some error region in  $P' \mapsto a$  path to region  $S_k$  in P.

0.00



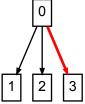
Other problems and final implementation

#### ComputeRefinePred

from proof m that shows that  $S_k$ cannot be reached in P'.

- ▶ Predicate  $\neg p_i$  removes edge to some error region in  $P' \mapsto a$ path to region  $S_k$  in P.
- ▶ The conjunction of these  $\neg p_1 \wedge \ldots \wedge \neg p_n$  removes all edges to error regions in P'

Goal: Extract refinement predicate  $\rho$  Refinement of initial region in P':



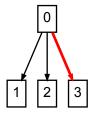
DASH<sub>call</sub>

## ComputeRefinePred

Goal: Extract refinement predicate from proof m that shows that  $S_k$  cannot be reached in P'.

- ▶ Predicate  $\neg p_i$  removes edge to some error region in  $P' \mapsto a$  path to region  $S_k$  in P.
- ▶ The conjunction of these  $\neg p_1 \land \ldots \land \neg p_n$  removes all edges to error regions in  $P' \mapsto$  every path to region  $S_k$  in P.

Goal: Extract refinement predicate  $\rho$  Refinement of initial region in P':



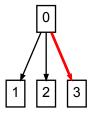
DASH<sub>call</sub>

## ComputeRefinePred

Goal: Extract refinement predicate  $\rho$  from proof m that shows that  $S_k$  cannot be reached in P'.

- ▶ Predicate  $\neg p_i$  removes edge to some error region in  $P' \mapsto a$  path to region  $S_k$  in P.
- ► The conjunction of these  $\neg p_1 \land \ldots \land \neg p_n$  removes all edges to error regions in  $P' \mapsto$  every path to region  $S_k$  in P.

Refinement of initial region in P':



$$\neg(\neg p_1\wedge\ldots\wedge\neg p_n)=p_1\vee\ldots\vee p_n$$

Other problems and final implementation

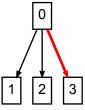
## ComputeRefinePred

Goal: Extract refinement predicate  $\rho$ from proof m that shows that  $S_k$ cannot be reached in P'.

- ▶ Predicate  $\neg p_i$  removes edge to some error region in  $P' \mapsto a$ path to region  $S_k$  in P.
- ▶ The conjunction of these  $\neg p_1 \wedge \ldots \wedge \neg p_n$  removes all edges to error regions in  $P' \mapsto$ every path to region  $S_k$  in P.

$$\rho := \bigvee_{\rho_i \in \mathsf{InitialRefines}(m)} \rho_i$$

Refinement of initial region in P':



$$\neg(\neg p_1 \wedge \ldots \wedge \neg p_n) = p_1 \vee \ldots \vee p_n$$

Parallelizing Top-Down Interprocedural Analysis

Other problems and final implementation

## ComputeRefinePred

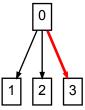
from proof m that shows that  $S_k$ cannot be reached in P'.

- ▶ Predicate  $\neg p_i$  removes edge to some error region in  $P' \mapsto a$ path to region  $S_k$  in P.
- The conjunction of these  $\neg p_1 \wedge \ldots \wedge \neg p_n$  removes all

edges to error regions in  $P' \mapsto$ every path to region  $S_k$  in P.

$$\rho := \left( \bigvee_{\alpha \in \mathsf{InitialRefiner}(m)} \rho_i \right)$$

Goal: Extract refinement predicate  $\rho$  Refinement of initial region in P':



$$\neg(\neg p_1\wedge\ldots\wedge\neg p_n)=p_1\vee\ldots\vee p_n$$

$$\begin{pmatrix} V & V & \rho_i \\ V & Q_i \in \text{InitialRefines}(m) \end{pmatrix} \begin{bmatrix} a_0/v_0, \dots, a_n/v_n \end{bmatrix} \mid v_i \in \text{params}(P')$$

Parallelizing Top-Down Interprocedural Analysis

Other problems and final implementation

## ComputeRefinePred

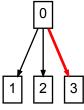
from proof m that shows that  $S_k$ cannot be reached in P'.

- ▶ Predicate  $\neg p_i$  removes edge to some error region in  $P' \mapsto a$ path to region  $S_k$  in P.
- The conjunction of these  $\neg p_1 \wedge \ldots \wedge \neg p_n$  removes all

edges to error regions in  $P' \mapsto$ every path to region  $S_k$  in P.

$$ho := \pi^{-1} \left( \left( \bigvee_{
ho_i \in \mathsf{InitialRefines}(m)} 
ho_i 
ight) \left[ a_0/v_0, \ldots, a_n/v_n 
ight] \mid v_i \in \mathsf{params}(P') 
ight)$$

Goal: Extract refinement predicate  $\rho$  Refinement of initial region in P':



$$\neg(\neg p_1\wedge\ldots\wedge\neg p_n)=p_1\vee\ldots\vee p_n$$

$$\left[a_0/v_0,\ldots,a_n/v_n
ight] \mid v_i \in \mathsf{params}(P')$$

000		000
Other problems and final implementa	ation	
ExtendFrontier implementation	2: 3:	$\begin{split} & \text{let } \langle \mathcal{S}_{k-1}, . \rangle = R\mathcal{S}_{k-1} \\ & \langle \phi, \mathcal{S} \rangle := \text{ExecuteSymbolic}(\tau_{c}, P, \varPsi, \mathcal{G}) \\ & op := \operatorname{Op}(\mathcal{S}_{k-1}, \mathcal{S}_{k}) \\ & \text{if } op \text{ matches } V := f(a_{0}, \ldots, a_{n}) \text{ then} \\ & \text{let } \langle p_{k}, . \rangle = \mathcal{S}_{k} \\ & P' := \operatorname{Lookup}(f, \varPsi) \\ & \pi := \operatorname{CreateVariableRenamer}(\operatorname{locals}(P) \cup \{v, v_{0} \mid \forall v \in \operatorname{params}(P)\}) \end{split}$
Renaming	8:	$\phi_{\mathit{ic}} := \left( igwedge_{v_i \in params(P')} v_i = \pi(a_i)  ight) \wedge \left( igwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e)  ight) \wedge \pi(\phi)$
► InputConstraints	9: 10: 11:	$\begin{array}{l} \phi_{ec} := \pi(\rho_k[@r/V]) \\ \mathcal{G}' := ReconstructGraphsAndInsertConstraints(\mathcal{G}, \phi_{ic}, \phi_{ec}, P') \\ \langle r, z \rangle = DashLoop(\mathcal{G}', \mathcal{P}, P') \end{array}$
<ul><li>ExitConstraints</li></ul>	12: 13:	$\begin{array}{l} \textbf{if } r = \text{FAIL } \textbf{then} \\ t := \pi^{-1} \Big( z \setminus \{ v_0 \mid \forall v \in params(P') \} \Big) \setminus \Big( locals(P) \cup params(P) \Big) \end{array}$

Implementing ExtendFrontier

```
► Extract
                                        24:
                                                     \rho := \mathsf{true}
    refinement
                                        25.
                                                 end if
                                             end if
    predicate
                                             return \langle t, \rho \rangle
```

14:

15:

16:

17: 18.

21.

22:

23:

19: else 20:

DASH<sub>call</sub>

Graph

construction

Extract test

input

 $\rho := \mathsf{true}$ 

t := UNSAT

 $t := IsSAT(\phi, P)$ 

if t = UNSAT then

 $\rho := \mathsf{RefinePred}(\tau_c)$ 

else

end if

else

Parallelizing Top-Down Interprocedural Analysis

 $\rho := \pi^{-1} \left( \left( \bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i \right) \left[ a_0 / v_0, \dots, a_n / v_n \right] \mid v_i \in \mathsf{params}(P') \right)$ 

000		000
Other problems and final implementa	ation	
ExtendFrontier implementation	2: 3:	let $\langle S_{k-1}, - \rangle = RS_{k-1}$ $\langle \phi, \zeta \rangle := \text{ExecuteSymbolic}(\tau_c, P, x, g)$ $op := \text{Op}(S_{k-1}, S_k)$ if $op \text{ matches } V := f(a_0, \dots, a_n) \text{ then}$ let $\langle P_k, - \rangle = S_k$ P' := Lookup(f, x) $\pi := \text{CreateVariableRenamer}(\text{locals}(P) \cup \{v, v_0 \mid \forall v \in \text{params}(P)\})$
Renaming	8:	$\phi_{\mathit{ic}} := \left( \bigwedge_{v_i \in params(P')} v_i = \pi(a_i) \right) \land \left( \bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \land \pi(\phi)$
1	9:	$\phi_{ec} := \pi( ho_k[\mathtt{@r/V}])$

 $g^{T}$  := ReconstructGraphsAndInsertConstraints $(g, \phi_{ic}, \phi_{ec}, P')$  $\langle r, z \rangle$  = DashLoop(g', p, P')

 $t := \pi^{-1} \left( z \setminus \{ v_0 \mid \forall v \in \mathsf{params}(P') \} \right) \setminus \left( \mathsf{locals}(P) \cup \mathsf{params}(P) \right)$ 

Implementing ExtendFrontier

11: 12:

13. 14:

15:

16:

17.

if r = FAIL then

 $\rho := \mathsf{true}$ 

t := UNSAT

else

DASH<sub>call</sub>

► InputConstraints

**ExitConstraints** 

construction

Graph

	000	000
Other problems and final imple	mentation	
ExtendFrontier implementation	2: 3:	$\begin{split} & \textbf{let} \ \langle S_{k-1}, \rangle = RS_{k-1} \\ & \langle \phi,S \rangle \coloneqq \textbf{ExecuteSymbolic}(\tau_c,P,\varPsi,g) \\ & op \coloneqq \textbf{Op}(S_{k-1},S_k) \\ & \textbf{if op matches } \textbf{V} \ \coloneqq \textbf{E}(a_0,\ldots,a_n) \ \textbf{then} \\ & \textbf{let} \ \langle \rho_k, \rangle = S_k \\ & P' \coloneqq \textbf{Lookup}(f,\varPsi) \\ & \pi \coloneqq \textbf{CreateVariableRenamer}(\textbf{locals}(P) \cup \{v,v_0 \mid \forall v \in \texttt{params}(P)\}) \end{split}$
<ul><li>Renaming</li></ul>	8:	$\phi_{ic} := \left( igwedge_{v_i \in params(P')} v_i = \pi(a_i) \right) \land \left( igwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \land \pi(\phi)$
	9:	$\phi_{ec} := \pi(\rho_k[@r/V])$

if r = FAIL then

 $\rho := \mathsf{true}$ 

t := UNSAT

 $t := IsSAT(\phi, P)$ 

if t = UNSAT then

 $\rho := \mathsf{true}$ 

 $\rho := \mathsf{RefinePred}(\tau_c)$ 

else

end if

else

end if end if

return  $\langle t, \rho \rangle$ 

g' := ReconstructGraphsAndInsertConstraints $(g, \phi_{ic}, \phi_{ec}, P')$  $\langle r, z \rangle$  = DashLoop(g', P, P')

 $t := \pi^{-1}(z \setminus \{v_0 \mid \forall v \in \mathsf{params}(P')\}) \setminus (\mathsf{locals}(P) \cup \mathsf{params}(P))$ 

 $\rho := \pi^{-1} \left( \left( \bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i \right) \left[ a_0 / v_0, \dots, a_n / v_n \right] \mid v_i \in \mathsf{params}(P') \right)$ 

Implementing ExtendFrontier

11: 12:

13: 14:

15:

16:

17: 18:

21.

23:

24:

25.

19: else 20:

DASH<sub>call</sub>

InputConstraints

**ExitConstraints** 

construction

Extract test

refinement

predicate

input

Extract

Graph

00•		000
Other problems and final implement	ation	
ExtendFrontier		let $\langle S_{k-1}, \rangle = RS_{k-1}$ $\langle \phi, \varsigma \rangle := \text{ExecuteSymbolic}(\tau_c, P, x, g)$
implementation	3:	$(\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5, \varphi_5, \varphi_5, \varphi_5, \varphi_5, \varphi_5, \varphi_5, \varphi_5$
'	5: 6:	$\begin{array}{l} \operatorname{let} \ \langle \rho_k, \rangle = S_k \\ P' := \operatorname{Lookup}(\mathfrak{f}, \mathscr{D}) \end{array}$
► Renaming	7: 8:	$\pi := CreateVariableRenamer(locals(P) \cup \{v, v_0 \mid \forall v \in params(P)\})$ $\phi_{ic} := \left(\bigwedge_{v_i \in params(P')} v_i = \pi(a_i)\right) \wedge \left(\bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e)\right) \wedge \pi(\phi)$
9	9:	$\phi_{ec} := \pi(\rho_k[@r/V])$
InputConstraints	10: 11:	$g' := ReconstructGraphsAndInsertConstraints(g, \phi_{ic}, \phi_{ec}, P')$ $\langle r, z \rangle = DashLoop(g', P, P')$

 $t := \pi^{-1} \left( z \setminus \{ v_0 \mid \forall v \in \mathsf{params}(P') \} \right) \setminus \left( \mathsf{locals}(P) \cup \mathsf{params}(P) \right)$ 

 $\rho := \pi^{-1} \left( \left( \bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i \right) \left[ a_0 / v_0, \dots, a_n / v_n \right] \mid v_i \in \mathsf{params}(P') \right)$ 

Implementing ExtendFrontier

0000000

11: 12:

14:

15:

16:

17:

**ExitConstraints** 

construction

Graph

```
18.
                                                end if
Extract test
                                       10 else
                                       20:
                                                t := IsSAT(\phi, P)
    input
                                       21.
                                                if t = \text{UNSAT} then
                                                    \rho := \mathsf{RefinePred}(\tau_c)
Extract
                                       23:
                                                else
                                       24:
                                                    \rho := \mathsf{true}
    refinement
                                       25.
                                                end if
                                            end if
    predicate
                                            return \langle t, \rho \rangle
```

if r = FAIL then

 $\rho := \mathsf{true}$ 

t := UNSAT

else

DASH<sub>call</sub>

000		000
Other problems and final implemen	tation	
ExtendFrontier implementation	2: 3:	$\begin{split} &\textbf{let } \langle S_{k-1}, \rangle = RS_{k-1} \\ &\langle \phi, \mathcal{S} \rangle := ExecuteSymbolic(\tau_{C}, P, x, \mathcal{G}) \\ &op := Op(S_{k-1}, S_k) \\ &\textbf{if } op \ \textbf{matches } \mathcal{V} := \mathbf{f}(a_0, \ldots, a_n) \ \textbf{then} \\ &\textbf{let } \langle p_k, \rangle = S_k \\ &P' := Lookup(\mathbf{f}, x) \\ &\pi := CreateVariableRenamer(locals(P) \cup \{v, v_0 \mid \forall v \in params(P)\}) \end{split}$
▶ Renaming	8:	$\phi_{ic} := \left( \bigwedge_{v_i \in params(P')} v_i = \pi(a_i) \right) \land \left( \bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \land \pi(\phi)$
► InputConstraints	9: 10: 11:	$\begin{array}{l} \phi_{ec} := \pi(\rho_k[@r/V]) \\ \mathcal{G}' := ReconstructGraphsAndInsertConstraints(\mathcal{G}, \phi_{ic}, \phi_{ec}, P') \\ \langle r, z \rangle = DashLoop(\mathcal{G}', \mathcal{P}, P') \end{array}$

 $t := \pi^{-1} \left( z \setminus \{ v_0 \mid \forall v \in \mathsf{params}(P') \} \right) \setminus \left( \mathsf{locals}(P) \cup \mathsf{params}(P) \right)$ 

 $\rho := \pi^{-1} \left( \left( \bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i \right) \left[ a_0 / v_0, \dots, a_n / v_n \right] \mid v_i \in \mathsf{params}(P') \right)$ 

Implementing ExtendFrontier

12:

14:

15:

16:

17: 18.

21:

23:

24:

25.

10 else 20:

```
end if
       predicate
                                               return \langle t, \rho \rangle
A study of the DASH algorithm for software property checking
```

**ExitConstraints** 

construction

Extract test

refinement

input

Extract

Graph

DASH<sub>call</sub>

Parallelizing Top-Down Interprocedural Analysis

if r = FAIL then

 $\rho := \mathsf{true}$ 

t := UNSAT

 $t := IsSAT(\phi, P)$ 

if t = UNSAT then  $\rho := \mathsf{RefinePred}(\tau_c)$ 

 $\rho := \mathsf{true}$ 

else

end if

else

end if

00	•	000
Other problems and final impleme	ntation	
ExtendFrontier implementation	2: 3:	$\begin{array}{l} \mathbf{let} \; \langle \mathcal{S}_{k-1}, . \rangle = R \mathcal{S}_{k-1} \\ \langle \phi, \mathcal{S} \rangle \coloneqq \mathbf{ExecuteSymbolic}(\tau_{\mathbf{C}}, P, \mathcal{I}, \mathcal{G}) \\ op \coloneqq Op(\mathcal{S}_{k-1}, \mathcal{S}_k) \\ \mathbf{if} \; op \; matches \; \mathbf{V} \coloneqq f(a_0, \ldots, a_n) \; then \\ \mathbf{let} \; \langle \rho_k, . \rangle = \mathcal{S}_k \\ P' \coloneqq Lookup(\mathbf{f}, \mathcal{I}) \\ \pi \coloneqq CreateVariableRenamer(locals(P) \cup \{v, v_0 \mid \forall v \in params(P)\}) \end{array}$
► Renaming	8:	$\phi_{ic} := \left( \bigwedge_{v_i \in params(P')} v_i = \pi(a_i) \right) \land \left( \bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \land \pi(\phi)$
► InputConstraints	9: 10: 11:	$\begin{aligned} \phi_{ec} &:= \pi(\rho_k[\mathfrak{G}r/V]) \\ \mathcal{G}' &:= ReconstructGraphsAndInsertConstraints(\mathcal{G}, \phi_{ic}, \phi_{ec}, P') \\ \mathcal{G}, z) &= DashLoop(\mathcal{G}', \mathcal{P}, P') \end{aligned}$

if r = FAIL then

 $\rho := \mathsf{true}$ 

t := UNSAT

 $t := IsSAT(\phi, P)$ 

if t = UNSAT then  $\rho := \text{RefinePred}(\tau_c)$ 

 $\rho := \mathsf{true}$ 

else

end if

else

end if end if

Implementing ExtendFrontier

12:

14:

15:

16:

17: 18:

21:

23:

24:

25.

19: else 20:

**ExitConstraints** 

construction

Extract test

refinement

input

Extract

Graph

```
predicate 27: return \langle t, \rho \rangle

A study of the DASH algorithm for software property checking
```

DASH<sub>call</sub>

Parallelizing Top-Down Interprocedural Analysis

00

 $t := \pi^{-1}(z \setminus \{v_0 \mid \forall v \in \mathsf{params}(P')\}) \setminus (\mathsf{locals}(P) \cup \mathsf{params}(P))$ 

 $\rho := \pi^{-1} \left( \left( \bigvee_{\rho_i \in \mathsf{InitialRefines}(z)} \rho_i \right) \left[ a_0 / v_0, \dots, a_n / v_n \right] \mid v_i \in \mathsf{params}(P') \right)$ 

000	000
Other problems and final implementation	
ExtendFrontier 2: 3: implementation 4: 5:	$\begin{split} & \textbf{let} \ \langle \mathcal{S}_{k-1}, \rangle = R\mathcal{S}_{k-1} \\ & \langle \phi,\mathcal{S} \rangle \coloneqq ExecuteSymbolic(\tau_{c},P,\mathscr{D},\mathscr{G}) \\ & o_{\mathcal{P}} \coloneqq Op(\mathcal{S}_{k-1},\mathcal{S}_{k}) \\ & \textbf{if op matches } V \ \coloneqq \ f(a_{0},\ldots,a_{n}) \ \textbf{then} \\ & \textbf{let} \ \langle \mathcal{P}_{k}, \rangle = \mathcal{S}_{k} \\ & \mathcal{P}' \coloneqq Lookup(f,\mathscr{D}) \\ & \pi \coloneqq CreateVariableRenamer(locals(P) \cup \{v,v_{0} \mid \forall v \in params(P)\}) \end{split}$

00

 $\phi_{ic} := \left( \bigwedge_{v_i \in \mathsf{params}(P')} v_i = \pi(a_i) \right) \land \left( \bigwedge_{(w \mapsto e) \in \mathcal{S}} \pi(w = e) \right) \land \pi(\phi)$ 

Implementing ExtendFrontier

```
\phi_{ec} := \pi(\rho_k[@r/V])
► InputConstraints
                                                               g^{\vec{f}} := \text{ReconstructGraphsAndInsertConstraints}(\mathcal{G}, \phi_{ic}, \phi_{ec}, P')
\langle r, z \rangle = \text{DashLoop}(\mathcal{G}', \mathcal{P}, P')
                                                   10:
                                                   11:
                                                   12:
                                                               if r = FAIL then
      ExitConstraints
                                                                     t := \pi^{-1} \left( z \setminus \{ v_0 \mid \forall v \in \mathsf{params}(P') \} \right) \setminus \left( \mathsf{locals}(P) \cup \mathsf{params}(P) \right)
                                                   13.
                                                   14:
                                                                     \rho := \mathsf{true}
Graph
                                                   15:
                                                               else
                                                   16:
                                                                     t := UNSAT
      construction
                                                                    \rho := \pi^{-1} \left( \left( \bigvee_{\rho_i \in \text{InitialRefines}(z)} \rho_i \right) \left[ a_0 / v_0, \dots, a_n / v_n \right] \mid v_i \in \text{params}(P') \right)
                                                   17.
                                                   18.
                                                                end if
Extract test
                                                   10 else
                                                   20:
                                                               t := IsSAT(\phi, P)
      input
                                                   21.
                                                               if t = \text{UNSAT} then
                                                                     \rho := RefinePred(\tau_c)
Extract
                                                   23:
                                                                else
                                                   24:
                                                                     \rho := true
      refinement
                                                   25.
                                                                end if
                                                          end if
```

return  $\langle t, \rho \rangle$ 

DASH<sub>call</sub>

Renaming

predicate

Does DASH scale to large programs?

Does DASH scale to large programs?

► DASH is single threaded

Does DASH scale to large programs?

- ► DASH is single threaded
- DASH does not cache prior results

Does DASH scale to large programs?

- ► DASH is single threaded
- DASH does not cache prior results

How can we improve the situation?

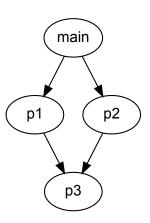
Does DASH scale to large programs?

- ► DASH is single threaded
- ▶ DASH does not cache prior results

How can we improve the situation? The article *Parallelizing Top-Down Interprocedural Analysis* presents BOLT:

- Top-Down Interprocedural analysis
- Uses summaries to avoid recomputation
- Is modular → multithreading support

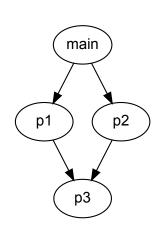
## Call graph



- ► Each procedure is a node
- Edges correspond to procedure calls

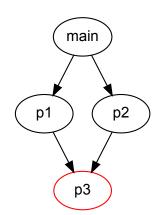
## Bottom-up interprocedural analysis

 Analyzes leafs first, assuming any input can be given

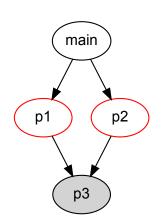


DASH<sub>call</sub>

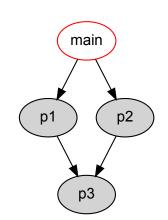
- Analyzes leafs first, assuming any input can be given
  - ▶ p3 analyzed first



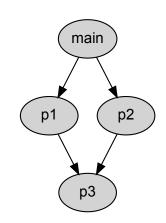
- Analyzes leafs first, assuming any input can be given
  - ▶ p3 analyzed first
  - ► Summery for *p*3 can be used to analyze *p*1 and *p*2



- Analyzes leafs first, assuming any input can be given
  - ▶ p3 analyzed first
  - ► Summery for p3 can be used to analyze p1 and p2
  - ► Summaries for *p*1 and *p*2 can be used to analyze *main*

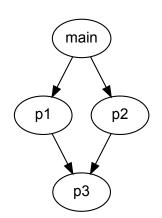


- Analyzes leafs first, assuming any input can be given
  - ▶ p3 analyzed first
  - ► Summery for p3 can be used to analyze p1 and p2
  - ► Summaries for *p*1 and *p*2 can be used to analyze *main*
- Callers of a procedure p<sub>i</sub> is decoupled from the analysis of the body of p<sub>i</sub> → easily parallelizable

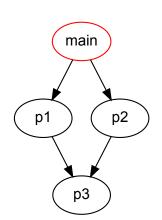


DASH<sub>call</sub>

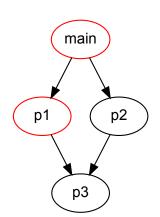
- Analyzes main procedure first.
- Analyze procedures in called context



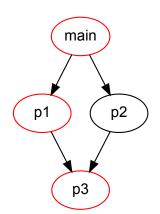
- Analyzes main procedure first.
- Analyze procedures in called context
  - ▶ analyze *main* procedure



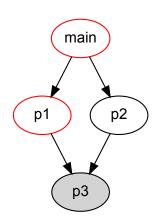
- ► Analyzes main procedure first.
- Analyze procedures in called context
  - ► analyze *main* procedure
  - ▶  $main \mapsto p1$



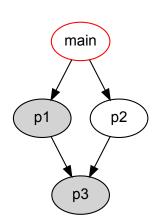
- ► Analyzes main procedure first.
- Analyze procedures in called context
  - ▶ analyze *main* procedure
  - ▶  $main \mapsto p1$
  - ▶  $p1 \mapsto p3$



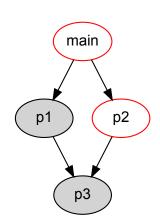
- ► Analyzes main procedure first.
- Analyze procedures in called context
  - ▶ analyze *main* procedure
  - ightharpoonup main  $\mapsto p1$
  - ▶  $p1 \mapsto p3$
  - Summary generated for p3



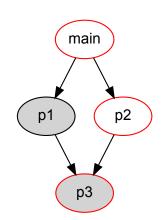
- Analyzes main procedure first.
- Analyze procedures in called context
  - ▶ analyze *main* procedure
  - ightharpoonup main  $\mapsto$  p1
  - ▶ p1 → p3
  - ▶ Summary generated for *p*3
  - Summary generated for p1



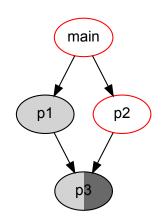
- Analyzes main procedure first.
- Analyze procedures in called context
  - ▶ analyze *main* procedure
  - ightharpoonup main  $\mapsto$  p1
  - ▶  $p1 \mapsto p3$
  - Summary generated for p3
  - Summary generated for p1
  - ▶  $main \mapsto p2$



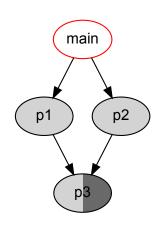
- Analyzes main procedure first.
- Analyze procedures in called context
  - ▶ analyze *main* procedure
  - ightharpoonup main  $\mapsto p1$
  - ▶  $p1 \mapsto p3$
  - Summary generated for p3
  - Summary generated for p1
  - ▶  $main \mapsto p2$
  - ▶ p2 → p3



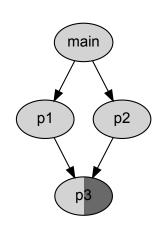
- ► Analyzes main procedure first.
- Analyze procedures in called context
  - ▶ analyze *main* procedure
  - ightharpoonup main  $\mapsto$  p1
  - ▶  $p1 \mapsto p3$
  - Summary generated for p3
  - Summary generated for p1
  - ▶  $main \mapsto p2$
  - p2 → p3
  - ▶ New summary generated for p3



- ► Analyzes main procedure first.
- Analyze procedures in called context
  - ▶ analyze *main* procedure
  - ightharpoonup main  $\mapsto$  p1
  - $ightharpoonup p1 \mapsto p3$
  - Summary generated for p3
  - Summary generated for p1
  - ▶  $main \mapsto p2$
  - p2 → p3
  - ▶ New summary generated for *p*3
  - ▶ Summary generated for *p*2



- ► Analyzes main procedure first.
- Analyze procedures in called context
  - ► analyze *main* procedure
  - ightharpoonup main  $\mapsto$  p1
  - p1 → p3
  - Summary generated for p3
  - Summary generated for p1
  - ▶  $main \mapsto p2$
  - p2 → p3
  - ▶ New summary generated for p3
  - Summary generated for p2
  - Analysis ends in main



DASH<sub>Call</sub>

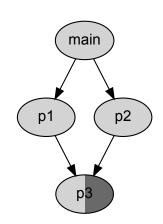
- Analyzes main procedure first.
- Analyze procedures in called context
  - ▶ analyze *main* procedure

$$ightharpoonup$$
 main  $\mapsto$  p1

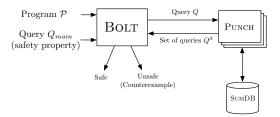
- Summary generated for p3
- Summary generated for p1

▶ 
$$main \mapsto p2$$

- ▶ New summary generated for *p*3
- Summary generated for p2
- Analysis ends in main
- ► Fine grained dependencies → difficult to parallelize



#### **BOLT**



- BOLT calls PUNCH with a query Q about a procedure P<sub>i</sub>
- PUNCH analyzes a single procedure.

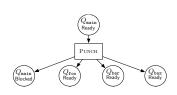
- Checks if SUMDB contains a summery that answers the query
- ▶ If not, it returns a set *Q*<sup>s</sup> to bolt containing the unanswered query
- ► If all answers are found, it answers the query *Q*

```
int foo(int p_foo);
int bar(int p bar);
int baz(int p baz);
main(int i, int j) {
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (i > -10)
    x = bar(i);
  else
    x = baz(j);
  v = x + 5;
  assert(y > 0);
```

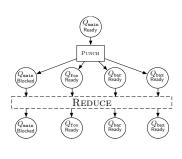
```
int foo(int p_foo);
int bar(int p bar);
int baz(int p baz);
main(int i, int j) {
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);
  v = x + 5;
  assert(y > 0);
```



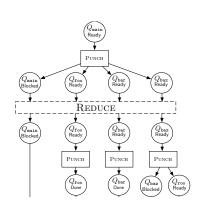
```
int foo(int p_foo);
int bar(int p bar);
int baz(int p baz);
main(int i, int j) {
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (i > -10)
    x = bar(i);
  else
    x = baz(j);
  v = x + 5;
  assert(y > 0);
```



```
int foo(int p_foo);
int bar(int p bar);
int baz(int p baz);
main(int i, int j) {
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);
  v = x + 5;
  assert(y > 0);
```

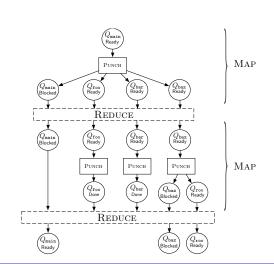


```
int foo(int p_foo);
int bar(int p bar);
int baz(int p baz);
main(int i, int j) {
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (i > -10)
    x = bar(i);
  else
    x = baz(j);
  v = x + 5;
  assert(y > 0);
```



## Parallel analysis using BOLT

```
int foo(int p_foo);
int bar(int p bar);
int baz(int p baz);
main(int i, int j) {
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (i > -10)
    x = bar(i);
  else
    x = baz(j);
  v = x + 5;
  assert(y > 0);
```



Parallelizing Top-Down Interprocedural Analysis

Summaries and how to use them

### **Summaries**

► Two kinds of summaries:

### **Summaries**

DASH<sub>call</sub>

- ► Two kinds of summaries:
  - Must summary:  $\langle \varphi_1 \stackrel{must}{\Longrightarrow}_{P_i} \varphi_2 \rangle$ All states starting in state  $\varphi_1$  in procedure  $P_i$  ends in a state  $\varphi_2$

#### **Summaries**

- ► Two kinds of summaries:
  - Must summary:  $\langle \varphi_1 \stackrel{must}{\Longrightarrow}_{P_i} \varphi_2 \rangle$ All states starting in state  $\varphi_1$  in procedure  $P_i$  ends in a state  $\varphi_2$
  - Not-may summary:  $\langle \varphi_1 \stackrel{\neg\textit{may}}{\Longrightarrow}_{P_i} \varphi_2 \rangle$ All states starting in state  $\varphi_1$  in procedure  $P_i$  cannot reach a state  $\varphi_2$

DASH<sub>call</sub>

### Queries and use of summaries

▶ Query:  $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_{P_i} \varphi_2 \rangle$ Can procedure  $P_i$  starting in state  $\varphi_1$  reach a state  $\varphi_2$ ?

Summaries and how to use them

### Queries and use of summaries

▶ Query:  $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_{P_i} \varphi_2 \rangle$ Can procedure  $P_i$  starting in state  $\varphi_1$  reach a state  $\varphi_2$ ? In DASH:  $\varphi_1$  = InputConstraints and  $\varphi_2 = S_k$ 

### Queries and use of summaries

- ▶ Query:  $\langle \varphi_1 \stackrel{!}{\Longrightarrow}_{P_i} \varphi_2 \rangle$ Can procedure  $P_i$  starting in state  $\varphi_1$  reach a state  $\varphi_2$ ? In DASH:  $\varphi_1 = \text{InputConstraints}$  and  $\varphi_2 = S_k$
- ▶ **Yes** if:  $S = \langle \hat{\varphi}_1 \stackrel{must}{\Longrightarrow}_{P_i} \hat{\varphi}_2 \rangle$  where  $\hat{\varphi}_1 \subseteq \varphi_1$  and  $\varphi_2 \cap \hat{\varphi}_2 \neq \emptyset$

### Queries and use of summaries

- ▶ Query:  $\langle \varphi_1 \stackrel{?}{\Longrightarrow}_{P_i} \varphi_2 \rangle$ Can procedure  $P_i$  starting in state  $\varphi_1$  reach a state  $\varphi_2$ ? In DASH:  $\varphi_1 = \text{InputConstraints}$  and  $\varphi_2 = S_k$
- ▶ **Yes** if:  $S = \langle \hat{\varphi}_1 \stackrel{\textit{must}}{\Longrightarrow}_{P_i} \hat{\varphi}_2 \rangle$  where  $\hat{\varphi}_1 \subseteq \varphi_1$  and  $\varphi_2 \cap \hat{\varphi}_2 \neq \emptyset$
- ▶ **No** if:  $S = \langle \hat{\varphi}_1 \stackrel{\neg may}{\Longrightarrow}_{P_i} \hat{\varphi}_2 \rangle$  where  $\varphi_1 \subseteq \hat{\varphi}_1$  and  $\varphi_2 \subseteq \hat{\varphi}_2$

DASH<sub>call</sub>

# Experimantal results

- ► Implemented DASH in BOLT
  - Single threaded analysis: 26 hours
  - Parallel analysis: 7 hours

# Experimantal results

- ► Implemented DASH in BOLT
  - ► Single threaded analysis: 26 hours
  - Parallel analysis: 7 hours
  - ► Average speedup: 3.71x