Procedure-Modular Termination Analysis

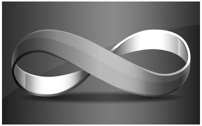
Cristina David
Daniel Kroening
Peter Schrammel

International Workshop on Termination, WST 2018

Motivation

Termination bugs:

- Hanging apps
- Zombie worker threads
- Denial-of-service attacks



State-of-the-art in termination analysis:

- Tremendous progress in recent years, but
  - Focus on small programs with complex termination arguments
  - SV-COMP Termination category (before 2017): avg. 20 lines per benchmark!
- Many tools use mathematical instead of machine numbers

Objective:

- Step towards practical termination analysis of large programs

Overview

Approach:

- Summary-based, context-sensitive, interprocedural
- Lexicographic ranking functions
- Universal termination and sufficient preconditions

2LS: static analysis tool for C programs:

- Synthesis-based program analysis
- Bit-precise reasoning

Evaluation:

- Benchmarks that are more than two orders of magnitude larger

TOPLAS 2018
Chen, David, Kroening, Schrammel, Wachter
Bit-Precise Procedure-Modular Termination Analysis

Overview

Termination Analyses

Universal termination:

- Result: terminating / potentially non-term. / non-terminating
- Decision problem

Conditional termination:

- Result: sufficient precondition for termination
- Inference problem

Example

(cf. http://www.netlib.org/clapack/cblas/sasum.c)

```
int h(int *sx, int n, int incx) {
  int nincx = n * incx;
  int stemp = 0;
  for (int i=0; incx<0 ? i >= nincx : i <= nincx; i+=incx) {
    stemp += sx[i];
  }
  return stemp;
}
```

Example

(cf. http://www.netlib.org/clapack/cblas/sasum.c)

```
unsigned f(int *sx, int n, unsigned incx) {
  unsigned s = *sx;
  if(incx>0) s = h(sx, n, incx);
  return s;
}
int h(int *sx, int n, int incx) {
  int nincx = n * incx;
  int stemp = 0;
  for (int i=0; incx<0 ? i >= nincx : i <= nincx; i+=incx) {
    stemp += sx[i];
  }
  return stemp;
}
```

Example

(cf. http://www.netlib.org/clapack/cblas/sasum.c)

```
unsigned f(int *sx, int n, unsigned incx) {
  unsigned s = *sx;
  if(incx>0) s = h(sx, n, incx);
  return s;
}
int h(int *sx, int n, int incx) {
  int nincx = n * incx;
  int stemp = 0;
  for (int i=0; incx<0 ? i >= nincx : i <= nincx; i+=incx) {
    stemp += sx[i];
  }
  return stemp;
}
```

- Why context-sensitive?

Example

(cf. http://www.netlib.org/clapack/cblas/sasum.c)

```
unsigned f(int *sx, int n, unsigned incx) {
  unsigned s = *sx;
  if(incx>0) s = h(sx, n, incx);
  return s;
}
int h(int *sx, int n, int incx) {
  int nincx = n * incx;
  int stemp = 0;
  for (int i=0; incx<0 ? i >= nincx : i <= nincx; i+=incx) {
    stemp += sx[i];
  }
  return stemp;
}
```

- Why context-sensitive?
- Why modular?

Example

(cf. http://www.netlib.org/clapack/cblas/sasum.c)

```
unsigned f(int *sx, int n, unsigned incx) {
  unsigned s = *sx;
  if(incx>0) s = h(sx, n, incx);
  return s;
}
int h(int *sx, int n, int incx) {
  int nincx = n * incx;
  int stemp = 0;
  for (int i=0; incx<0 ? i >= nincx : i <= nincx; i+=incx) {
    stemp += sx[i];
  }
  return stemp;
}
```

- Why context-sensitive?
- Why modular?
- Why lexicographic ranking functions?

4	7	1	2

Example

(cf. http://www.netlib.org/clapack/cblas/sasum.c)

```
unsigned f(int *sx, int n, unsigned incx) {
  unsigned s = *sx;
  if(incx>0) s = h(sx, n, incx);
  return s;
}
int h(int *sx, int n, int incx) {
  int nincx = n * incx;
  int stemp = 0;
  for (int i=0; incx<0 ? i >= nincx : i <= nincx; i+=incx) {
    stemp += sx[i];
  }
  return stemp;
}
```

- Why context-sensitive?
- Why modular?
- Why lexicographic ranking functions?

4	7	1	1

Example

(cf. http://www.netlib.org/clapack/cblas/sasum.c)

```
unsigned f(int *sx, int n, unsigned incx) {
  unsigned s = *sx;
  if(incx>0) s = h(sx, n, incx);
  return s;
}
int h(int *sx, int n, int incx) {
  int nincx = n * incx;
  int stemp = 0;
  for (int i=0; incx<0 ? i >= nincx : i <= nincx; i+=incx) {
    stemp += sx[i];
  }
  return stemp;
}
```

- Why context-sensitive?
- Why modular?
- Why lexicographic ranking functions?

4	7	1	0

Example

(cf. http://www.netlib.org/clapack/cblas/sasum.c)

```
unsigned f(int *sx, int n, unsigned incx) {
   unsigned s = *sx;
   if(incx>0) s = h(sx, n, incx);
   return s;
}
int h(int *sx, int n, int incx) {
   int nincx = n * incx;
   int stemp = 0;
   for (int i=0; incx<0 ? i >= nincx : i <= nincx; i+=incx) {
      stemp += sx[i];
   }
   return stemp;
}
```

- Why context-sensitive?
- Why modular?
- Why lexicographic ranking functions?

4	7	0	9

Example

(cf. http://www.netlib.org/clapack/cblas/sasum.c)

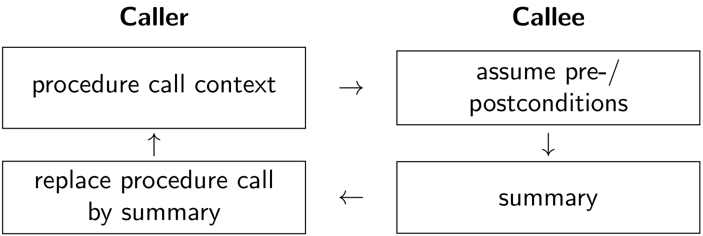
```
unsigned f(int *sx, int n, unsigned incx) {
  unsigned s = *sx;
  if(incx>0) s = h(sx, n, incx);
  return s;
}
int h(int *sx, int n, int incx) {
  int nincx = n * incx;
  int stemp = 0;
  for (int i=0; incx<0 ? i >= nincx : i <= nincx; i+=incx) {
    stemp += sx[i];
  }
  return stemp;
}
```

- Why context-sensitive?
- Why modular?
- Why lexicographic ranking functions?

3	8	4	2

Summary-Based Interprocedural Analysis

- **Context:** relation over inputs and outputs from the caller's perspective
- **Summary:** relation over inputs and outputs from the callee's perspective



Example

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}
```

```
unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

Example — Universal Termination

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}
```

```
unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

→ analyse f

↳ $CallCtx^{o}_{h_0}$ $(1 \leq z \leq M \wedge 0 \leq w \leq M)$

→ analyse h w.r.t. $CallCtx^{o}_{h_0}$ $(1 \leq y \leq M \wedge 0 \leq x \leq M)$

↳ $Inv^{o}_{h_0}$ $(0 \leq x \leq M \wedge 1 \leq y \leq M)$, $Sum^{o}_{h_0}$ $(10 \leq x \leq M \wedge 1 \leq y \leq M)$

↳ Inv^{o}_{f} $(true)$, Sum^{o}_{f} $(true)$

←

Example — Universal Termination

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}

unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

→ analyse f

↳ $CallCtx^{o}_{h_0}$ $(1 \le z \le M \wedge 0 \le w \le M)$

→ analyse h w.r.t. $CallCtx^{o}_{h_0}$ $(1 \le y \le M \wedge 0 \le x \le M)$

↳ $Inv^{o}_{h_0}$ $(0 \le x \le M \wedge 1 \le y \le M)$, $Sum^{o}_{h_0}$ $(10 \le x \le M \wedge 1 \le y \le M)$

↳ Inv^{o}_{f} $(true)$, Sum^{o}_{f} $(true)$

←

→ analyse f

→ analyse h

↳ RR_h $(-x > -x')$

h terminating

↳ RR_f $(true)$

f terminating ←

Example — Sufficient Preconditions

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}

unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

→ analyse f

↳ $CallCtx^{o}_{h_0}$ $(true)$

→ analyse h w.r.t. $CallCtx^{o}_{h_0}$ $(true)$

↳ $Inv^{o}_{h_0}$ $(true)$, $Sum^{o}_{h_0}$ $(10 \leq x \leq M \wedge 0 \leq y \leq M)$

↳ Inv^{o}_{f} $(true)$, Sum^{o}_{f} $(true)$

←

→ analyse f

↳ $CallCtx^{u}_{h_0}$

→ analyse h w.r.t. $CallCtx^{u}_{h_0}$

↳ $Inv^{u}_{h_0}$, RR_{h_0} $(-x > x)$, $Sum^{u}_{h_0}$

$Precond^{u}_{h}$ $(1 \leq y \leq M)$

↳ Inv^{u}_{f}, RR_{f} $(true)$, Sum^{u}_{f}

← $Precond^{u}_{f}$ $(true)$

Synthesis-Based Program Analysis

2LS: A Static Analyser for C Programs

`http://www.cprover.org/2LS`

Built on CPROVER framework:

- Bit-precise analysis (including floating-point arithmetic)
- Reduction to propositional encoding and SAT solving

Concepts used:

- Logical specification of program analysis problems
- Template-based synthesis

Program Encoding

Non-recursive programs with multiple procedures

Procedure f: $(\quad Init(\boldsymbol{x}^{in}, \boldsymbol{x}) \quad , \quad Trans(\boldsymbol{x}, \boldsymbol{x}') \quad , \quad Out(\boldsymbol{x}, \boldsymbol{x}^{out}) \quad)$

Program Encoding

Non-recursive programs with multiple procedures

Procedure f: ($Init(\boldsymbol{x}^{in}, \boldsymbol{x})$, $Trans(\boldsymbol{x}, \boldsymbol{x}')$, $Out(\boldsymbol{x}, \boldsymbol{x}^{out})$)

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0)
    w = h(z);

  return w;
}
```

$$
\begin{aligned}
& w_0 = 0 \\
\wedge\ & g_4 = z > 0 \\
\wedge\ & h_0((z),(r_{h_0})) \wedge w_1 = r_{h_0} \\
\wedge\ & w_2^{\phi} = g_4 ? w_1 : w_0 \\
\wedge\ & r_h = x_1^{\phi}
\end{aligned}
$$

```
unsigned h(unsigned y) {
  unsigned x;
  for (x=0;

      x<10;
      x+=y);
  return x;
}
```

$$
\begin{aligned}
& g_0 = true \\
\wedge\ & x_0 = 0 \\
\wedge\ & g_1 = g_0 \wedge x_1^{\phi} = (ls_3 ? x_3^{lb} : x_0) \\
\wedge\ & g_2 = (x_1^{\phi} < 10 \wedge g_1) \\
\wedge\ & x_2 = x_1^{\phi} + y \\
\wedge\ & r_h = x_1^{\phi}
\end{aligned}
$$

Program Encoding

Non-recursive programs with multiple procedures

Procedure f: ($Init(\boldsymbol{x}^{in}, \boldsymbol{x})$, $Trans(\boldsymbol{x}, \boldsymbol{x}')$, $Out(\boldsymbol{x}, \boldsymbol{x}^{out})$)

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0)
    w = h(z);

  return w;
}
```

$$
\begin{aligned}
& w_0 = 0 \\
\wedge\ & g_4 = z > 0 \\
\wedge\ & h_0((z),(r_{h_0})) \wedge w_1 = r_{h_0} \\
\wedge\ & w_2^{\phi} = g_4 ? w_1 : w_0 \\
\wedge\ & r_h = x_1^{\phi}
\end{aligned}
$$

```
unsigned h(unsigned y) {
  unsigned x;
  for (x=0;

      x<10;
      x+=y);
  return x;
}
```

$$
\begin{aligned}
& g_0 = true \\
\wedge\ & x_0 = 0 \\
\wedge\ & g_1 = g_0 \wedge x_1^{\phi} = (ls_3 ? x_3^{lb} : x_0) \\
\wedge\ & g_2 = (x_1^{\phi} < 10 \wedge g_1) \\
\wedge\ & x_2 = x_1^{\phi} + y \\
\wedge\ & r_h = x_1^{\phi}
\end{aligned}
$$

Logical Specification of Program Analysis Problems

Safety verification:

$$\begin{aligned}\exists_2 Inv. \quad & \forall \boldsymbol{x}^{in}, \boldsymbol{x}, \boldsymbol{x}'. \\ & \big(Init(\boldsymbol{x}^{in}, \boldsymbol{x}) \implies Inv(\boldsymbol{x})\big) \wedge \\ & \big(Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies Inv(\boldsymbol{x}')\big) \wedge \\ & \big(Inv(\boldsymbol{x}) \implies \neg Err(\boldsymbol{x})\big)\end{aligned}$$

(Blass and Gurevich '87, Grebenshchikov et al '12, David et al '15, ...)

Logical Specification of Program Analysis Problems

Safety verification:

$$\begin{aligned}\exists_2 Inv.\quad & \forall \boldsymbol{x}^{in}, \boldsymbol{x}, \boldsymbol{x}'.\\ & \big(Init(\boldsymbol{x}^{in}, \boldsymbol{x}) \implies Inv(\boldsymbol{x})\big) \wedge \\ & (Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies Inv(\boldsymbol{x}')) \wedge \\ & (Inv(\boldsymbol{x}) \implies \neg Err(\boldsymbol{x}))\end{aligned}$$

Invariant inference:

$$\begin{aligned}\min Inv.\quad & \forall \boldsymbol{x}, \boldsymbol{x}'.\\ & (Init(\boldsymbol{x}) \implies Inv(\boldsymbol{x})) \wedge \\ & (Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies Inv(\boldsymbol{x}'))\end{aligned}$$

(Blass and Gurevich '87, Grebenshchikov et al '12, David et al '15, ...)

Logical Specification of Program Analysis Problems

Safety verification:

$$\begin{aligned}\exists_2 Inv.\quad & \forall \boldsymbol{x}^{in}, \boldsymbol{x}, \boldsymbol{x}'.\\ & (Init(\boldsymbol{x}^{in}, \boldsymbol{x}) \Longrightarrow Inv(\boldsymbol{x})) \wedge\\ & (Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow Inv(\boldsymbol{x}')) \wedge\\ & (Inv(\boldsymbol{x}) \Longrightarrow \neg Err(\boldsymbol{x}))\end{aligned}$$

Invariant inference:

$$\begin{aligned}\min Inv.\quad & \forall \boldsymbol{x}, \boldsymbol{x}'.\\ & (Init(\boldsymbol{x}) \Longrightarrow Inv(\boldsymbol{x})) \wedge\\ & (Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow Inv(\boldsymbol{x}'))\end{aligned}$$

Termination, . . .

(Blass and Gurevich '87, Grebenshchikov et al '12, David et al '15, ...)

Template-Based Synthesis

Reduction to first-order logic via templates, e.g. safety verification:

$$\begin{aligned} \exists_2 Inv.\forall \boldsymbol{x}, \boldsymbol{x}'1. \quad & (Init(\boldsymbol{x}) \implies Inv(\boldsymbol{x})) \wedge \\ & (Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies Inv(\boldsymbol{x}')) \wedge \\ & (Inv(\boldsymbol{x}) \implies \neg Err(\boldsymbol{x})) \end{aligned}$$

Template-Based Synthesis

Reduction to first-order logic via templates, e.g. safety verification:

$$\begin{aligned}\exists \boldsymbol{d}.\quad \forall \boldsymbol{x}, \boldsymbol{x}'.\quad & (Init(\boldsymbol{x}) \Longrightarrow \mathcal{T}(\boldsymbol{x}, \boldsymbol{d})) \wedge \\ & (\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathcal{T}(\boldsymbol{x}', \boldsymbol{d})) \wedge \\ & (\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \Longrightarrow \neg Err(\boldsymbol{x}))\end{aligned}$$

where $\boldsymbol{d}$ are template parameters.

(Graf & Saïdi CAV'97, . . . , Reps et al, . . . Brauer et al, . . . , Srivastava et al, . . .)

Template-Based Synthesis

Reduction to first-order logic via templates, e.g. invariant inference:

$$\begin{aligned} \min \boldsymbol{d}. \quad \forall \boldsymbol{x}, \boldsymbol{x}'. \quad & (\mathit{Init}(\boldsymbol{x}) \Longrightarrow \mathcal{T}(\boldsymbol{x}, \boldsymbol{d})) \wedge \\ & (\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathcal{T}(\boldsymbol{x}', \boldsymbol{d})) \end{aligned}$$

where $\boldsymbol{d}$ are template parameters.

(Graf & Saïdi CAV'97, . . . , Reps et al, . . . Brauer et al, . . . , Srivastava et al, . . .)

Template-Based Synthesis

Reduction to first-order logic via templates, e.g. invariant inference:

$$\min \boldsymbol{d}. \quad \forall \boldsymbol{x}, \boldsymbol{x}'. \quad \begin{array}{l}(\mathit{Init}(\boldsymbol{x}) \Longrightarrow \mathcal{T}(\boldsymbol{x}, \boldsymbol{d})) \wedge \\ (\mathcal{T}(\boldsymbol{x}, \boldsymbol{d}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathcal{T}(\boldsymbol{x}', \boldsymbol{d}))\end{array}$$

where $\boldsymbol{d}$ are template parameters.

(Graf & Saïdi CAV'97, . . . , Reps et al, . . . Brauer et al, . . . , Srivastava et al, . . .)

- Abstract interpretation with large-block transformers
- Abstract domains implemented implicitly
- Drawback: Difficult to make efficient
  - We use strategy iteration (e.g. Gawlitza & Seidl, FMSD'14)

Invariants, Summaries, Calling Contexts

- **Invariant** *Inv*:

$$\begin{aligned} \min\ Inv^o.\forall \boldsymbol{x}^{in}, \boldsymbol{x}, \boldsymbol{x}'. \quad & Init(\boldsymbol{x}^{in}, \boldsymbol{x}) \Longrightarrow Inv^o(\boldsymbol{x}) \\ \wedge \quad & Inv^o(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow Inv^o(\boldsymbol{x}') \end{aligned}$$

Invariants, Summaries, Calling Contexts

- **Invariant** *Inv*:

$$\begin{aligned} \min\ Inv^o.\forall \mathbf{x}^{in}, \mathbf{x}, \mathbf{x}'.\quad & Init(\mathbf{x}^{in}, \mathbf{x}) \implies Inv^o(\mathbf{x}) \\ \wedge\quad & Inv^o(\mathbf{x}) \wedge Trans(\mathbf{x}, \mathbf{x}') \implies Inv^o(\mathbf{x}') \end{aligned}$$

- **Summary** Sum^o:

$$\begin{aligned} & \min\ Sum^o.\forall \mathbf{x}^{in}, \mathbf{x}, \mathbf{x}', \mathbf{x}^{out}. \\ & \quad Init(\mathbf{x}^{in}, \mathbf{x}) \wedge Inv^o(\mathbf{x}') \wedge Out(\mathbf{x}', \mathbf{x}^{out}) \implies Sum^o(\mathbf{x}^{in}, \mathbf{x}^{out}) \end{aligned}$$

Invariants, Summaries, Calling Contexts

- **Invariant** *Inv*:

$$\begin{aligned} \min Inv^o.\forall \boldsymbol{x}^{in}, \boldsymbol{x}, \boldsymbol{x}'. \quad & Init(\boldsymbol{x}^{in}, \boldsymbol{x}) \Longrightarrow Inv^o(\boldsymbol{x}) \\ \wedge \quad & Inv^o(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow Inv^o(\boldsymbol{x}') \end{aligned}$$

- **Summary** Sum^o:

$$\begin{aligned} & \min Sum^o.\forall \boldsymbol{x}^{in}, \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}^{out}. \\ & \quad Init(\boldsymbol{x}^{in}, \boldsymbol{x}) \wedge Inv^o(\boldsymbol{x}') \wedge Out(\boldsymbol{x}', \boldsymbol{x}^{out}) \Longrightarrow Sum^o(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \end{aligned}$$

- **Calling context** $CallCtx^o_{h_i}$ for procedure call h at call site i:

$$\begin{aligned} & \min CallCtx^o_{h_i}.\forall \boldsymbol{x}, \boldsymbol{x}', {\boldsymbol{x}^{p\_in}}_i, {\boldsymbol{x}^{p\_out}}_i : \\ & \quad Inv^o(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow CallCtx^o_{h_i}({\boldsymbol{x}^{p\_in}}_i, {\boldsymbol{x}^{p\_out}}_i) \end{aligned}$$

Termination Arguments, Preconditions

- Given Inv^o, **termination argument** RR:

$$\exists RR . \forall \boldsymbol{x}, \boldsymbol{x}' : Inv^o(\boldsymbol{x}) \land Trans(\boldsymbol{x}, \boldsymbol{x}') \implies RR(\boldsymbol{x}, \boldsymbol{x}')$$

with e.g. $RR(\boldsymbol{x}, \boldsymbol{x}') = r(\boldsymbol{x}) > r(\boldsymbol{x}') \land r(\boldsymbol{x}) > 0$

Termination Arguments, Preconditions

- Given Inv^o, **termination argument** RR:

$$\exists RR.\forall \boldsymbol{x}, \boldsymbol{x}' : Inv^o(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies RR(\boldsymbol{x}, \boldsymbol{x}')$$

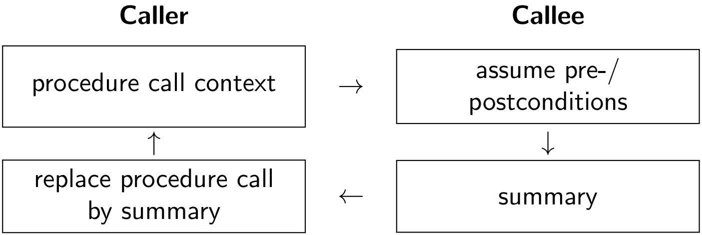
  with e.g. $RR(\boldsymbol{x}, \boldsymbol{x}') = r(\boldsymbol{x}) > r(\boldsymbol{x}') \wedge r(\boldsymbol{x}) > 0$

- **Sufficient precondition for termination** $Precond^u$ ...

Interprocedural Analysis

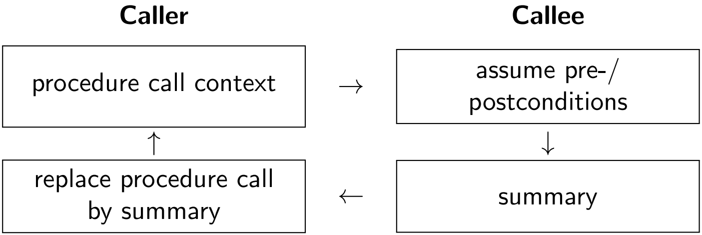
Summary-Based Interprocedural Analysis

- **Context:** relation over inputs and outputs from the caller's perspective
- **Summary:** relation over inputs and outputs from the callee's perspective



Summary-Based Interprocedural Analysis

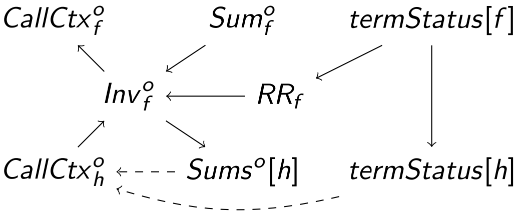
- **Context:** relation over inputs and outputs from the caller's perspective
- **Summary:** relation over inputs and outputs from the callee's perspective



Cyclically dependent predicates

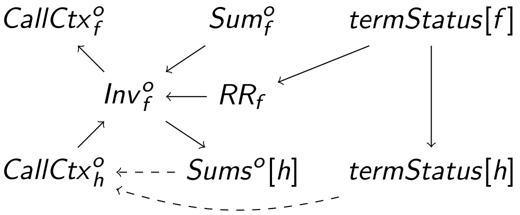
Universal Termination

Cyclically dependent predicates:



Decomposition of the Verification Problem

Cyclically dependent predicates:



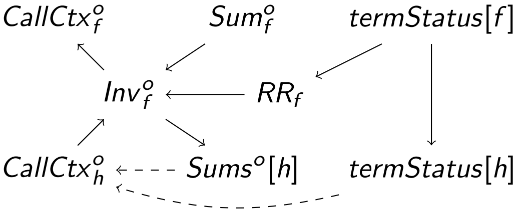
Decomposition into a sequence of subproblems

- Classical approach: Follow the call graph top-down

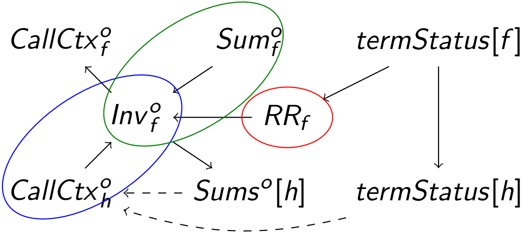
Soundness of the decomposition by

- soundness of the individual subproblems
- soundness of the combination of the subproblem results
- induction over the call graph traversal algorithm

Universal Termination



Universal Termination



1. Calling contexts of procedure calls h in f
2. Recurse
3. Invariants and summary of procedure f
4. Termination argument for procedure f
5. Determine termination status of f

Example – Bubble Sort

```
void sort(int n, int a[])
{
  __CPROVER_assume(n>=0);

  for(int x=0; x<n-1; x++)
    for(int y=0; y<n-x-1; y++) // g
      if(a[y]>a[y+1])
        swap(a[y], a[y+1]);
}
```

Example – Bubble Sort

```
void sort(int n, int a[])
{
  __CPROVER_assume(n>=0);

  for(int x=0; x<n-1; x++)
    for(int y=0; y<n-x-1; y++) // g
      if(a[y]>a[y+1])
        swap(a[y], a[y+1]);
}
```

Invariants: $(\mathit{true} \implies 0 \leq x \leq 2^{31} - 2) \wedge (g \implies 0 \leq y \leq 2^{31} - 2)$

Example – Bubble Sort

```
void sort(int n, int a[])
{
  __CPROVER_assume(n>=0);

  for(int x=0; x<n-1; x++)
    for(int y=0; y<n-x-1; y++) // g
      if(a[y]>a[y+1])
        swap(a[y], a[y+1]);
}
```

Invariants: $(true \implies 0 \leq x \leq 2^{31} - 2) \wedge (g \implies 0 \leq y \leq 2^{31} - 2)$

Termination arguments:

$$(\neg g \wedge (x < n - 1) \wedge \neg g' \implies (-1 \cdot (x - x') + 0 \cdot (y - y')) > 0) \wedge (g \wedge g' \implies (-1 \cdot (y - y') > 0)).$$

Lexicographic Linear Ranking Functions

Lexicographic ranking function $(R_n, R_{n-1}, ..., R_1)$:

$$
\begin{aligned}
&\exists \Delta > 0, i \in [1, n] : \forall \boldsymbol{x}, \boldsymbol{x}' : Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \\
&\quad R_i(\boldsymbol{x}) > 0 && \text{(Bounded)} \\
\wedge &\quad R_i(\boldsymbol{x}) - R_i(\boldsymbol{x}') > \Delta && \text{(Decreasing)} \\
\wedge &\quad \forall j > i : R_j(\boldsymbol{x}) - R_j(\boldsymbol{x}') \geq 0 && \text{(Unaffecting)}
\end{aligned}
$$

Lexicographic Linear Ranking Functions

Lexicographic ranking function $(R_n, R_{n-1}, ..., R_1)$:

$$
\begin{aligned}
&\exists \Delta > 0, i \in [1, n] : \forall \boldsymbol{x}, \boldsymbol{x}' : Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies \\
&\quad R_i(\boldsymbol{x}) > 0 && \text{(Bounded)} \\
&\wedge \quad R_i(\boldsymbol{x}) - R_i(\boldsymbol{x}') > \Delta && \text{(Decreasing)} \\
&\wedge \quad \forall j > i : R_j(\boldsymbol{x}) - R_j(\boldsymbol{x}') \geq 0 && \text{(Unaffecting)}
\end{aligned}
$$

Lexicographic Linear Ranking Functions

Lexicographic ranking function $(R_n, R_{n-1}, ..., R_1)$:

$$
\begin{aligned}
&\exists \Delta > 0, i \in [1, n] : \forall \boldsymbol{x}, \boldsymbol{x}' : Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \\
&\quad R_i(\boldsymbol{x}) > 0 && \text{(Bounded)} \\
&\wedge \quad R_i(\boldsymbol{x}) - R_i(\boldsymbol{x}') > 0 && \text{(Decreasing)} \\
&\wedge \quad \forall j > i : R_j(\boldsymbol{x}) - R_j(\boldsymbol{x}') \geq 0 && \text{(Unaffecting)}
\end{aligned}
$$

Lexicographic Linear Ranking Functions

Lexicographic ranking function $(R_n, R_{n-1}, ..., R_1)$:

$$\begin{array}{lll}\exists \Delta > 0, i \in [1, n] : \forall \boldsymbol{x}, \boldsymbol{x}' : Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies & & \\ & R_i(\boldsymbol{x}) > 0 & \text{(Bounded)} \\ \wedge & R_i(\boldsymbol{x}) - R_i(\boldsymbol{x}') > 0 & \text{(Decreasing)} \\ \wedge & \forall j > i : R_j(\boldsymbol{x}) - R_j(\boldsymbol{x}') \geq 0 & \text{(Unaffecting)}\end{array}$$

Lexicographic ranking function $(R_n, R_{n-1}, ..., R_1)$ over bitvectors:

$$\exists RR^n . \forall \boldsymbol{x}, \boldsymbol{x}' : Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \implies RR^n(\boldsymbol{x}, \boldsymbol{x}')$$

with

$$RR^n(\boldsymbol{x}, \boldsymbol{x}') = \bigvee_{i=1}^{n} (R_i(\boldsymbol{x}) - R_i(\boldsymbol{x}') > 0 \wedge \bigwedge_{j=i+1}^{n} (R_j(\boldsymbol{x}) - R_j(\boldsymbol{x}') \geq 0))$$

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x,y),(x',y')) = \ldots$

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x,y),(x',y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$$Trans((x, y), (x', y')) = \ldots$$

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x, y), (x', y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x, y), (x', y')) = \ldots$

$((x, y), (x', y')) = ((1, 100), (0, 100))$

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x, y), (x', y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x,y),(x',y')) = \ldots$

$((x,y),(x',y')) = ((1,100),(0,100))$

RR^1 corresponds to (x)

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x,y),(x',y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x,y),(x',y')) = \ldots$

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x,y),(x',y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x, y), (x', y')) = \ldots$

$((x, y), (x', y')) = ((1, 1), (1001, 2))$

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x, y), (x', y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x, y), (x', y')) = \ldots$

$((x, y), (x', y')) = ((1, 1), (1001, 2))$
RR^1 infeasible

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x, y), (x', y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x, y), (x', y')) = \dots$

1. Start with $RR = \mathit{false}$, $n = 1$ components
2. Find values for $((x, y), (x', y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x, y), (x', y')) = \ldots$

$((x, y), (x', y')) = ((1, 99), (0, 100))$

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x, y), (x', y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x, y), (x', y')) = \ldots$

$((x, y), (x', y')) = ((1, 99), (0, 100))$

RR^2 corresponds to $(-y, x)$

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x, y), (x', y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x, y), (x', y')) = \dots$

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x, y), (x', y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

Lexicographic Linear Ranking Functions

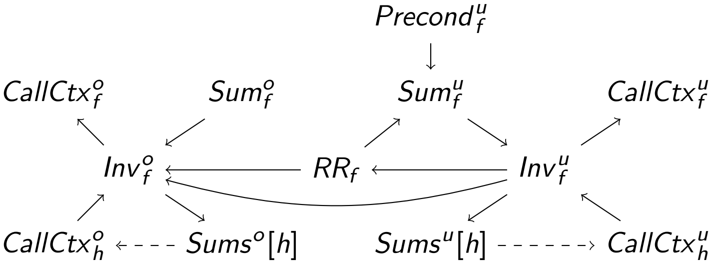
```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$Trans((x,y),(x',y')) = \ldots$

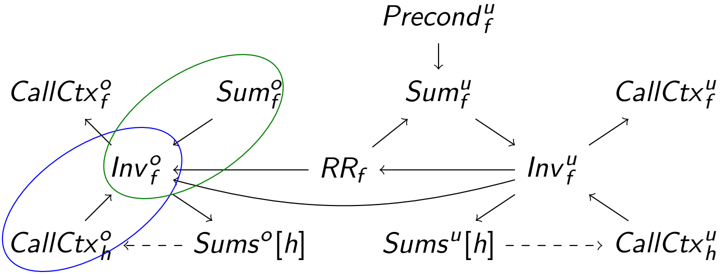
$((x,y),(x',y'))$ no more values
RR^2 corresponds to $(-y, x)$

1. Start with $RR = false$, $n = 1$ components
2. Find values for $((x,y),(x',y'))$ that do not satisfy RR
   - If fails then RR is a valid termination argument
3. Find values for parameters of ranking function components
   - If fails then increase n
   - Otherwise instantiate RR
4. Repeat from 2.

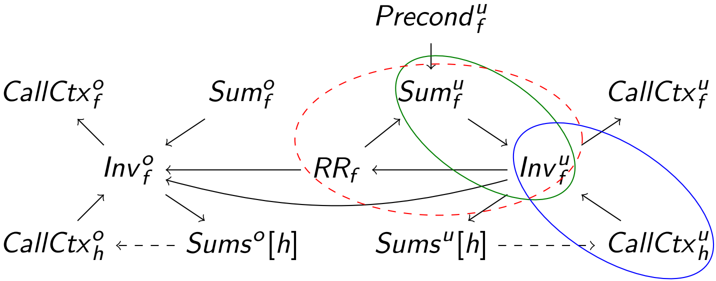
Sufficient Preconditions for Termination



Sufficient Preconditions for Termination

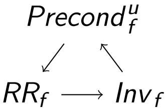


Sufficient Preconditions for Termination



1. . . .
2. Under-approximating calling contexts of procedure calls h in f
3. Recurse
4. Termination argument and sufficient preconditions for procedure f
5. Under-approximating invariants and summary of procedure f

Sufficient Preconditions for Termination



false is trivial sufficient precondition
Want to find "maximal" (weakest) solutions

Sufficient Preconditions for Termination

$$\begin{array}{ccc} & Precond_f^u & \\ \swarrow & & \nwarrow \\ RR_f & \longrightarrow & Inv_f \end{array}$$

false is trivial sufficient precondition
Want to find "maximal" (weakest) solutions

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}
```

```
unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

(cf. Cook et al '08)

Sufficient Preconditions for Termination

$$Precond^u_f \quad RR_f \longrightarrow Inv_f$$

false is trivial sufficient precondition
Want to find "maximal" (weakest) solutions

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}
```

```
unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

1. Pick a valid input as candidate precondition, e.g. $y=0$

(cf. Cook et al '08)

Sufficient Preconditions for Termination

false is trivial sufficient precondition
Want to find “maximal” (weakest) solutions

$$Precond^u_f \swarrow \quad \nwarrow \quad RR_f \longrightarrow Inv_f$$

```
unsigned f(unsigned z) {
   unsigned w = 0;
   if(z>0) w = h(z);
   return w;
}

unsigned h(unsigned y) {
   unsigned x;
   for (x=0; x<10; x+=y);
   return x;
}
```

1. Pick a valid input as candidate precondition, e.g. $y{=}0$
2. Compute invariant $(x{=}0 \wedge y{=}0)$
   and termination argument RR_f (not found)

(cf. Cook et al ’08)

Sufficient Preconditions for Termination

$$Precond_f^u \qquad RR_f \longrightarrow Inv_f$$

false is trivial sufficient precondition
Want to find "maximal" (weakest) solutions

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}
```

```
unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

1. Pick a valid input as candidate precondition, e.g. $y{=}21$

(cf. Cook et al '08)

Sufficient Preconditions for Termination

$$\begin{array}{ccc} & Precond_f^u & \\ \swarrow & & \nwarrow \\ RR_f & \longrightarrow & Inv_f \end{array}$$

false is trivial sufficient precondition
Want to find "maximal" (weakest) solutions

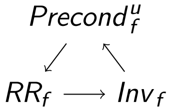
```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}
```

```
unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

1. Pick a valid input as candidate precondition, e.g. $y{=}21$
2. Compute invariant $(0 \leq x \leq 42 \wedge y{=}21)$
   and termination argument RR_f $(-x > -x')$

(cf. Cook et al '08)

Sufficient Preconditions for Termination



false is trivial sufficient precondition
Want to find "maximal" (weakest) solutions

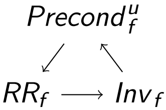
```
unsigned f(unsigned z) {
    unsigned w = 0;
    if(z>0) w = h(z);
    return w;
}

unsigned h(unsigned y) {
    unsigned x;
    for (x=0; x<10; x+=y);
    return x;
}
```

1. Pick a valid input as candidate precondition, e.g.
2. Compute invariant $(0 \leq x \leq 42 \wedge y = 21)$ and termination argument RR_f $(-x > -x')$
3. Compute partial precondition $Precond' = (1 \leq y \leq M)$ under-approximating all inputs that terminate under RR_f

(cf. Cook et al '08)

Sufficient Preconditions for Termination

false is trivial sufficient precondition
Want to find "maximal" (weakest) solutions




```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}
```

```
unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

1. Pick a valid input as candidate precondition, e.g.
   - If no such input found, $Precond_f^u = (1 \leq y \leq M)$
2. Compute invariant $(0 \leq x \leq 42 \wedge y{=}21)$ and termination argument RR_f $(-x > -x')$
3. Compute partial precondition $Precond' = (1 \leq y \leq M)$ under-approximating all inputs that terminate under RR_f

(cf. Cook et al '08)

Sufficient Preconditions for Termination

$Precond^u_f$

$\swarrow \qquad \nwarrow$

$RR_f \longrightarrow Inv_f$

false is trivial sufficient precondition

Want to find "maximal" (weakest) solutions

```
unsigned f(unsigned z) {
  unsigned w = 0;
  if(z>0) w = h(z);
  return w;
}

unsigned h(unsigned y) {
  unsigned x;
  for (x=0; x<10; x+=y);
  return x;
}
```

1. Pick a valid input as candidate precondition, e.g.
   - If no such input found, $Precond^u_f = (1 \leq y \leq M)$
2. Compute invariant $(0 \leq x \leq 42 \wedge y{=}21)$ and termination argument RR_f $(-x > -x')$
3. Compute partial precondition $Precond' = (1 \leq y \leq M)$ under-approximating all inputs that terminate under RR_f

(cf. Cook et al '08)

Sufficient Preconditions for Termination

Under-approximations:

- Compute a sufficient precondition for termination by computing a necessary precondition for non-termination
- Given RR, solve:

$$\begin{array}{ll} \min\ \mathit{Precond}^{\widetilde{u}}, \mathit{Inv}^{\widetilde{u}} : \forall \boldsymbol{x}^{in}, \boldsymbol{x}, \boldsymbol{x}' : & \\ \quad \mathit{Precond}^{\widetilde{u}}(\boldsymbol{x}^{in}) \wedge \mathit{Init}(\boldsymbol{x}^{in}, \boldsymbol{x}) \Longrightarrow \mathit{Inv}^{\widetilde{u}}(\boldsymbol{x}) & \\ \wedge \quad \mathit{Inv}^{\widetilde{u}}(\boldsymbol{x}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathit{Inv}^{\widetilde{u}}(\boldsymbol{x}') \wedge \neg RR(\boldsymbol{x}, \boldsymbol{x}') \ . & \end{array}$$

- $\mathit{Precond}^{u} := \neg \mathit{Precond}^{\widetilde{u}}$

(cf. Cook et al '08)

Evaluation

Evaluation

Hypotheses:

- Modular termination analysis is fast
- Modular termination analysis is precise
- 2LS outperforms existing termination analysis tools
- 2LS' analysis is bit-precise
- 2LS computes usable preconditions for termination

Benchmarks:

- Product line benchmarks from SV-COMP
  (597 benchmarks, 1.6 MLOC)
- Non-trivial procedural structure
  (on average 67 procedures, 5.5 loops)

Modular termination analysis is fast

	expected	**2LS IPTA**	2LS MTA	TAN	Ultimate
terminating	264	249	26	18	50
non-terminating	333	320	333	3	324
potentially non-terminating	—	14	1	425	0
timed out	—	**14**	237	150	43
errors	—	0	0	1	180
total run time (h)	—	**58.7**	119.6	92.8	23.9

Timeout: 1800 s per benchmark

Modular termination analysis is precise

	expected	**2LS IPTA**	2LS MTA	TAN	Ultimate
terminating	264	**249**	26	18	50
non-terminating	333	**320**	333	3	324
potentially non-terminating	—	**14**	1	425	0
timed out	—	14	237	150	43
errors	—	0	0	1	180
total run time (h)	—	58.7	119.6	92.8	23.9

Comparison with existing termination analysers

	expected	**2LS IPTA**	2LS MTA	TAN	Ultimate
terminating	264	**249**	26	18	50
non-terminating	333	**320**	333	3	324
potentially non-terminating	—	**14**	1	425	0
timed out	—	**14**	237	150	43
errors	—	**0**	0	1	180
total run time (h)	—	**58.7**	119.6	92.8	23.9

We also tried:

- AProVE, Loopus, FuncTion, HipTNT, ARMC, T2, KiTTeL

Mathematical vs. Machine Integers

```
void f00(unsigned n) {
  for(unsigned x=0; x<=n; x++);
}
```

Mathematical ✓
Machine ×

```
void f01(unsigned x) {
  while(x>=10) x++;
}
```

Mathematical ×
Machine ✓

Benchmarks from Loopus tool (Sinn et al CAV'14):

	2LS	Loopus
terminating	2	15
potentially non-term.	9	0
timed out / unknown	4	0

Rationals vs. Floating-Point

```
void f00(float x) {
  __CPROVER_assume(
    FLT_MIN<=x && x<=FLT_MAX);
  while(x>0.0f)
    x *= 0.9f;
}
```

Rationals	×
Floating-point	×

```
void f01(float x) {
  __CPROVER_assume(
    FLT_MIN<=x && x<=FLT_MAX);
  while(x>0.0f)
    x *= 0.1f;
}
```

Rationals	×
Floating-point	✓

(with round-to-nearest/ties-to-even rounding mode)

SV-COMP Termination Category

1119 LOC on average in 2018! *

	2017				2018			
	total	term	nont	bug	total	term	nont	bug
Ultimate	1272	813	459	0	1725	1110	615	0
CPAChecker	821	396	425	4	1193	685	508	0
AProVE	520	458	62	0	906	838	68	0
2LS	927	580	347	34	1365	727	638	1

	2017	2018
Ultimate	36h	61h
CPAChecker	119h	150h
AProVE	167h	228h
2LS	16h	28h

* Thanks to Jera Hensel and other SV-COMP participants.

Preconditions for Termination

(From: http://www.netlib.org/clapack/cblas/sasum.c)

```
int f(int *sx, int n, int incx) {
  int  nincx = n * incx;
  int stemp=0;
  for (int i=0; incx<0 ? i >= nincx: i<= nincx;
       i+=incx) {
    stemp += sx[i-1];
  }
  return stemp;
}
```

Sufficient precondition for termination: `incx != 0`

Conclusions

Limitations and Future Work

Language features:

- Recursion
- Dynamically allocated data structures
- Strings and arrays
- Concurrency

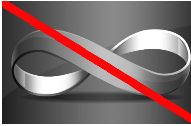
Analysis precision:

- Template refinement

Analyses and applications:

- Non-termination (Malík et al, TACAS'18)
- Sufficient preconditions for non-termination
- Cost analysis

Wrap-up



Summary:

- Interprocedural termination analysis using synthesis-based program analysis approach
- Evaluation on larger benchmarks

Ongoing work:

- Extend to dynamically allocated data structures
- Template refinement

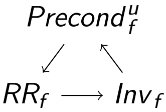
TOPLAS 2018
Chen, David, Kroening, Schrammel, Wachter
Bit-Precise Procedure-Modular Termination Analysis

2LS analysis tool for C programs: `www.cprover.org/2LS`

- Safety verification and refutation (SAS'15, TACAS'16, ATVA'17)
- (Non-)termination (ASE'15, TOPLAS'18, TACAS'18)

Extra Slides

Sufficient Preconditions for Termination



false is trivial sufficient precondition
Want to find "maximal" (weakest) solutions

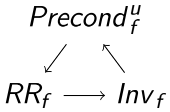
Bootstrapping:

1. Pick a valid input as candidate precondition
   - If no such input found, then candidate precondition is a sufficient precondition $Precond^u_f$
2. Compute invariant and termination argument RR_f under this precondition
   - If does not terminate, exclude input from candidate precondition and repeat from 1.
3. Compute partial precondition $Precond'$ under-approximating all inputs that terminate under RR_f
4. Add $Precond'$ to candidate precondition, repeat from 1.

(cf. Cook et al '08)

Sufficient Preconditions for Termination

false is trivial sufficient precondition
Want to find "maximal" (weakest) solutions



Bootstrapping:

1. Pick a valid input as candidate precondition
   - If no such input found, then candidate precondition is a sufficient precondition $Precond_f^u$
2. Compute invariant and termination argument RR_f under this precondition
   - If does not terminate, exclude input from candidate precondition and repeat from 1.
3. Compute partial precondition $Precond'$ under-approximating all inputs that terminate under RR_f
4. Add $Precond'$ to candidate precondition, repeat from 1.

(cf. Cook et al '08)

Implementation Details

Solver backend:

- One SAT solver instance per procedure (plus helper solvers)
- Many incremental calls

Summary re-use:

- Only recurse if available summaries $Sums[h]$ are incompatible with $CallCtx_h$ at current call site

Issues:

- Bitvector extension
- Coefficient refinement
- Bounding iterations (where sound) and
  number of lexicographic components