Modular Automatic Complexity Analysis of Recursive Integer Programs

Nils Lommen and Jürgen Giesl

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

main(x, y):
  while $x > 0$ do
    $x \leftarrow x - 1;\ y \leftarrow y + \text{fact}(x);$

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

main(x, y):
  while $x > 0$ do
    $x \leftarrow x - 1;\ y \leftarrow y + \mathrm{fact}(x);$

fact(a):	if $a = 0$	then	return 1;
	else if $a > 0$	then	return $a \cdot \mathrm{fact}(a-1)$;

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
```

- Does this program terminate over $\mathbb{Z}$?

```
fact(a):  if a = 0        then  return 1;
          else if a > 0   then  return a · fact(a − 1);
```

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
```

- Does this program terminate over $\mathbb{Z}$?
- How often do we execute the loops?

```
fact(a):  if a = 0        then  return 1;
          else if a > 0   then  return a · fact(a − 1);
```

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

$\text{main}(x, y)$:
  while $x > 0$ do
    $x \leftarrow x - 1;\ y \leftarrow y + \text{fact}(x);$
  $x \leftarrow 1;$

- Does this program terminate over $\mathbb{Z}$?
- How often do we execute the loops?

$\text{fact}(a)$: if $a = 0$ then return 1;
else if $a > 0$ then return $a \cdot \text{fact}(a - 1)$;

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

$$
\begin{array}{l}
\text{main}(x, y): \\
\quad \text{while } x > 0 \text{ do} \\
\quad\quad x \leftarrow x - 1;\ y \leftarrow y + \text{fact}(x); \\
\quad x \leftarrow 1; \\
\quad \text{while } x < y \text{ do} \\
\quad\quad x \leftarrow 3 \cdot x;\ y \leftarrow 2 \cdot y;
\end{array}
$$

- Does this program terminate over $\mathbb{Z}$?
- How often do we execute the loops?

$$
\begin{array}{llll}
\text{fact}(a): & \text{if } a = 0 & \text{then} & \text{return } 1; \\
 & \text{else if } a > 0 & \text{then} & \text{return } a \cdot \text{fact}(a - 1);
\end{array}
$$

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

$\text{main}(x, y)$:
  while $x > 0$ do
    $x \leftarrow x - 1;\ y \leftarrow y + \text{fact}(x);$
  $x \leftarrow 1;$
  while $x < y$ do
    $x \leftarrow 3 \cdot x;\ y \leftarrow 2 \cdot y;$

- Does this program terminate over $\mathbb{Z}$?
- How often do we execute the loops?
  - Solution: Use KoAT!

$\text{fact}(a)$: if $a = 0$ then return 1;
else if $a > 0$ then return $a \cdot \text{fact}(a - 1)$;

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- Does this program terminate over $\mathbb{Z}$?
- How often do we execute the loops?
  - Solution: Use KoAT!
  - Open-source complexity analysis tool for **I**nteger **T**ransition **S**ystems (ITS) with Function Calls

```
fact(a):  if a = 0       then  return 1;
          else if a > 0  then  return a · fact(a − 1);
```

Motivation

Goal: Infer (upper) runtime bounds for "real-world" programs

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- Does this program terminate over $\mathbb{Z}$?
- How often do we execute the loops?
  - Solution: Use KoAT!
  - Open-source complexity analysis tool for **I**nteger **T**ransition **S**ystems (ITS) with Function Calls

Contribution: Extend modular complexity analysis approach by function calls

```
fact(a):  if a = 0       then  return 1;
          else if a > 0  then  return a · fact(a − 1);
```

General Architecture

- Preprocess Programs:

General Architecture

- Preprocess Programs:
  - Remove never executed loops

General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron

General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \dots + a_d \cdot x_d \leq c$

General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...

General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?

General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?

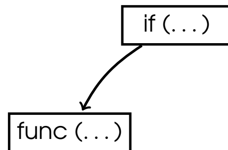
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?

if (...)

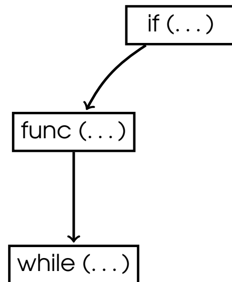
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?



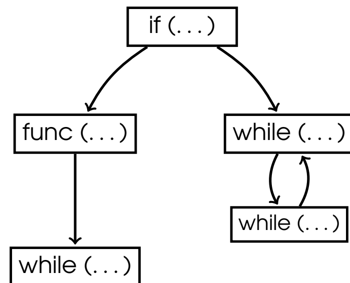
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?



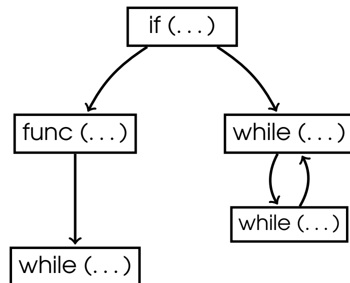
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?



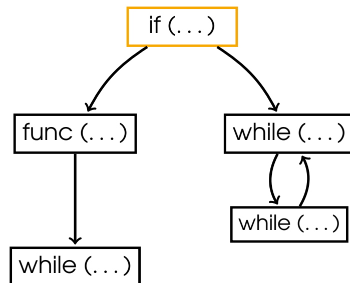
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more . . .
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?
- Analyze program blocks one after another



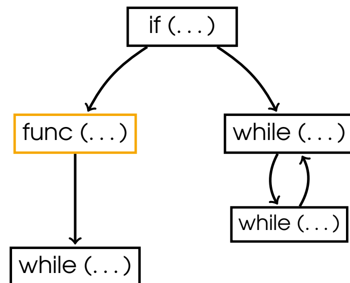
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?
- Analyze program blocks one after another



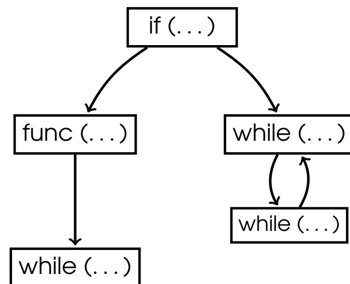
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?
- Analyze program blocks one after another



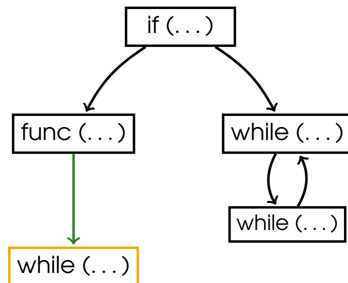
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?
- Analyze program blocks one after another
- Propagate information from "previous" blocks to "later" blocks via *size bounds*



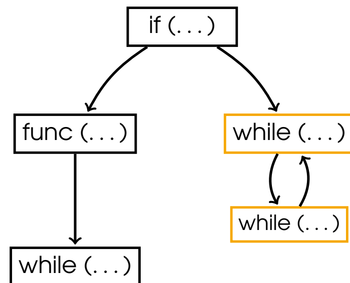
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?
- Analyze program blocks one after another
- Propagate information from "previous" blocks to "later" blocks via *size bounds*



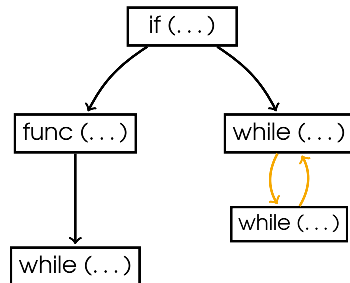
General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?
- Analyze program blocks one after another
- Propagate information from "previous" blocks to "later" blocks via *size bounds*



General Architecture

- Preprocess Programs:
  - Remove never executed loops
  - Invariants by Apron
    - Polyhedral domains: $a_1 \cdot x_1 + \cdots + a_d \cdot x_d \leq c$
  - many more ...
- Time bounds: How many executions?
- Size bounds: What is the value of a variable after evaluating a certain program part?
- Analyze program blocks one after another
- Propagate information from "previous" blocks to "later" blocks via *size bounds*



Automatically Computing Runtime Bounds

Step 1: Analyzing the first loop in the main-function

main(x, y):
  while $x > 0$ do
    $x \leftarrow x - 1;\ \ y \leftarrow y + \text{fact}(x);$
  $x \leftarrow 1;$
  while $x < y$ do
    $x \leftarrow 3 \cdot x;\ \ y \leftarrow 2 \cdot y;$

Automatically Computing Runtime Bounds

Step 1: Analyzing the first loop in the main-function

main(x, y):
  while $x > 0$ do
    $x \leftarrow x - 1;\ y \leftarrow y + \text{fact}(x);$
  $x \leftarrow 1;$
  while $x < y$ do
    $x \leftarrow 3 \cdot x;\ y \leftarrow 2 \cdot y;$

- **Modular** analysis of recursive programs:

Step 1: Analyzing the first loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - **Goal**: Find a linear *ranking function* via SMT-solvers to bound the loop iterations.

Automatically Computing Runtime Bounds

Step 1: Analyzing the first loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - **Goal**: Find a linear *ranking function* via SMT-solvers to bound the loop iterations.
  - **On our example** (first loop):

Automatically Computing Runtime Bounds

Step 1: Analyzing the first loop in the main-function

$\text{main}(x, y)$:
$\quad \text{while } x > 0 \text{ do}$
$\quad\quad x \leftarrow x - 1;\; y \leftarrow y + \text{fact}(x);$
$\quad x \leftarrow 1;$
$\quad \text{while } x < y \text{ do}$
$\quad\quad x \leftarrow 3 \cdot x;\; y \leftarrow 2 \cdot y;$

- **Modular** analysis of recursive programs:
  - **Goal**: Find a linear *ranking function* via SMT-solvers to bound the loop iterations.
  - **On our example** (first loop):
    - Ranking Function: $\text{rf} = \langle x \rangle$

Automatically Computing Runtime Bounds

Step 1: Analyzing the first loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - **Goal**: Find a linear *ranking function* via SMT-solvers to bound the loop iterations.
  - **On our example** (first loop):
    - Ranking Function: $\mathrm{rf} = \langle x \rangle$
    - The ranking function is bounded from below by $x > 0$.

Automatically Computing Runtime Bounds

Step 1: Analyzing the first loop in the main-function

main(x, y):
  while $x > 0$ do
    $x \leftarrow x - 1;\ y \leftarrow y + \mathrm{fact}(x);$
  $x \leftarrow 1;$
  while $x < y$ do
    $x \leftarrow 3 \cdot x;\ y \leftarrow 2 \cdot y;$

- **Modular** analysis of recursive programs:
  - **Goal**: Find a linear *ranking function* via SMT-solvers to bound the loop iterations.
  - **On our example** (first loop):
    - Ranking Function: $\mathrm{rf} = \langle x \rangle$
    - The ranking function is bounded from below by $x > 0$.
    - In each iteration, x is strictly decreased:

$$x \leftarrow x - 1$$

Automatically Computing Runtime Bounds

Step 1: Analyzing the first loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - **Goal**: Find a linear *ranking function* via SMT-solvers to bound the loop iterations.
  - **On our example** (first loop):
    - Ranking Function: $\mathrm{rf} = \langle x \rangle$
    - The ranking function is bounded from below by $x > 0$.
    - In each iteration, x is strictly decreased:

$$x \leftarrow x - 1$$

Overall Runtime Bound: $x + \cdots$

Step 2: Analyzing the factorial-function $\mathrm{fact}(x)$:

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

Overall Runtime Bound: $x + \cdots$

Step 2: Analyzing the factorial-function $\mathrm{fact}(x)$:

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function $\text{fact}(x)$:

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function fact(x):

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: $\text{rf} = \langle a \rangle$

Overall Runtime Bound: $x + \cdots$

Step 2: Analyzing the factorial-function fact(x):

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: rf $= \langle a \rangle$
  - 2-dimensional Ranking Function:
    $\rightsquigarrow$ Solve Recurrence for Runtime Bound

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function $\text{fact}(x)$:

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: $\text{rf} = \langle a \rangle$
  - 2-dimensional Ranking Function:
    $\rightsquigarrow$ Solve Recurrence for Runtime Bound
  - Lift *local runtime bound*:

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function fact(x):

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: $\mathrm{rf} = \langle a \rangle$
  - 2-dimensional Ranking Function:
    $\rightsquigarrow$ Solve Recurrence for Runtime Bound
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function fact(x):

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: $\text{rf} = \langle a \rangle$
  - 2-dimensional Ranking Function:
    $\rightsquigarrow$ Solve Recurrence for Runtime Bound
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable a before executing the subprogram?

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function fact(x):

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: $\text{rf} = \langle a \rangle$
  - 2-dimensional Ranking Function:
    $\rightsquigarrow$ Solve Recurrence for Runtime Bound
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable a before executing the subprogram?

runtime bound of first loop · size(a)

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function fact(x):

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: $\text{rf} = \langle a \rangle$
  - 2-dimensional Ranking Function:
    $\rightsquigarrow$ Solve Recurrence for Runtime Bound
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable a before executing the subprogram?

$$x \cdot \text{size}(a)$$

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function fact(x):

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: rf $= \langle a \rangle$
  - 2-dimensional Ranking Function:
    $\rightsquigarrow$ Solve Recurrence for Runtime Bound
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable a before executing the subprogram?

$$x \cdot x$$

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function fact(x):

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: $\text{rf} = \langle a \rangle$
  - 2-dimensional Ranking Function:
    $\rightsquigarrow$ Solve Recurrence for Runtime Bound
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable a before executing the subprogram?

$$x \cdot x = x^2$$

Overall Runtime Bound: $x + \cdots$

Automatically Computing Runtime Bounds

Step 2: Analyzing the factorial-function fact(x):

```
fact(a):
  if a = 0 then
    return 1;
  else if a > 0 then
    return a · fact(a − 1);
```

- **Modular** analysis of recursive programs:
  - Ranking Function: $\text{rf} = \langle a \rangle$
  - 2-dimensional Ranking Function:
    $\rightsquigarrow$ Solve Recurrence for Runtime Bound
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable a before executing the subprogram?

$$x \cdot x = x^2$$

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Size Bounds

Step 3: Analyzing the size of y in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = ???
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Size Bounds

Step 3: Analyzing the size of y in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = ???
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Size Bounds

Step 3: Analyzing the size of y in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = ???
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Change-accumulated size bounds

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Size Bounds

Step 3: Analyzing the size of y in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = ???
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Change-accumulated size bounds
    - Value after n loop iterations:

$$size(y) + n \cdot size(fact(x))$$

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Size Bounds

Step 3: Analyzing the size of y in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1; y ← y + fact(x);
  x ← 1;
  # size(y) = ???
  while x < y do
    x ← 3 · x; y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Change-accumulated size bounds
    - Value after n loop iterations:

$$size(y) + n \cdot size(fact(x))$$

  - Lift *local size bound*:

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Size Bounds

Step 3: Analyzing the size of y in the main-function

$\text{main}(x, y)$:
    while $x > 0$ do
        $x \leftarrow x - 1;\ y \leftarrow y + \text{fact}(x);$
    $x \leftarrow 1;$
    $\#\ \mathit{size}(y) = ???$
    while $x < y$ do
        $x \leftarrow 3 \cdot x;\ y \leftarrow 2 \cdot y;$

- **Modular** analysis of recursive programs:
  - Change-accumulated size bounds
    - Value after n loop iterations:

$$\mathit{size}(y) + n \cdot \mathit{size}(\text{fact}(x))$$

  - Lift *local size bound*:
    - How often do we execute the subprogram?

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Size Bounds

Step 3: Analyzing the size of y in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = ???
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Change-accumulated size bounds
    - Value after n loop iterations:

$$\mathit{size}(y) + n \cdot \mathit{size}(\mathrm{fact}(x))$$

  - Lift *local size bound*:
    - How often do we execute the subprogram?
    - How large is $\mathrm{fact}(x)$?

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Size Bounds

Step 3: Analyzing the size of y in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = ???
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Change-accumulated size bounds
    - Value after n loop iterations:

$$size(y) + n \cdot size(fact(x))$$

  - Lift *local size bound*:
    - How often do we execute the subprogram?
    - How large is fact(x)?

$$|y| + x \cdot x^x$$

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Size Bounds

Step 3: Analyzing the size of y in the main-function

```
main(x, y):
   while x > 0 do
      x ← x − 1;  y ← y + fact(x);
   x ← 1;
   # size(y) =  |y| + x · x^x
   while x < y do
      x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Change-accumulated size bounds
    - Value after n loop iterations:

$$size(y) + n \cdot size(fact(x))$$

  - Lift *local size bound*:
    - How often do we execute the subprogram?
    - How large is fact(x)?

$$|y| + x \cdot x^x$$

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
   while x > 0 do
      x ← x − 1;  y ← y + fact(x);
   x ← 1;
   # size(y) = |y| + x · x^x
   while x < y do
      x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = |y| + x · x^x
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR '26)

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = |y| + x · x^x
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR '26)
  - Lift *local runtime bound*:

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = |y| + x · x^x
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR '26)
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = |y| + x · x^x
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR '26)
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable y before executing the subprogram?

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

main(x, y):
  while $x > 0$ do
    $x \leftarrow x - 1;\ y \leftarrow y + \text{fact}(x);$
  $x \leftarrow 1;$
  # $size(y) = |y| + x \cdot x^x$
  while $x < y$ do
    $x \leftarrow 3 \cdot x;\ y \leftarrow 2 \cdot y;$

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR '26)
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable y before executing the subprogram?

$1 \cdot \log(\text{size}(y))$

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = |y| + x · x^x
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR '26)
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable y before executing the subprogram?

$$1 \cdot \log(\mathit{size}(y)) = \log(|y| + x \cdot x^x)$$

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = |y| + x · x^x
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR '26)
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable y before executing the subprogram?

$$1 \cdot \log(\mathit{size}(y)) = \log(|y| + x \cdot x^x)$$
$$\leq 1 + \log(|y|) + \log(x \cdot x^x)$$

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = |y| + x · x^x
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR '26)
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable y before executing the subprogram?

$$
\begin{aligned}
1 \cdot \log(\mathit{size}(y)) &= \log(|y| + x \cdot x^x) \\
&\leq 1 + \log(|y|) + \log(x \cdot x^x) \\
&= 1 + \log(|y|) + (x+1) \cdot \log(x)
\end{aligned}
$$

Overall Runtime Bound: $x + x^2 + \cdots$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = |y| + x · x^x
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR '26)
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable y before executing the subprogram?

$$
\begin{aligned}
1 \cdot \log(\mathit{size}(y)) &= \log(|y| + x \cdot x^x) \\
&\leq 1 + \log(|y|) + \log(x \cdot x^x) \\
&= 1 + \log(|y|) + (x+1) \cdot \log(x)
\end{aligned}
$$

Overall Runtime Bound: $x + x^2 + 1 + \log(|y|) + (x+1) \cdot \log(x)$

Automatically Computing Runtime Bounds

Step 4: Analyzing the second loop in the main-function

```
main(x, y):
  while x > 0 do
    x ← x − 1;  y ← y + fact(x);
  x ← 1;
  # size(y) = |y| + x · x^x
  while x < y do
    x ← 3 · x;  y ← 2 · y;
```

- **Modular** analysis of recursive programs:
  - Loop has runtime $\log(|y|)$ (JAR ’26)
  - Lift *local runtime bound*:
    - How often do we execute the subprogram?
    - How large is the variable y before executing the subprogram?

$$
\begin{aligned}
1 \cdot \log(\mathit{size}(y)) &= \log(|y| + x \cdot x^x) \\
&\leq 1 + \log(|y|) + \log(x \cdot x^x) \\
&= 1 + \log(|y|) + (x+1) \cdot \log(x)
\end{aligned}
$$

Overall Runtime Bound: $x + x^2 + 1 + \log(|y|) + (x+1) \cdot \log(x) \in \mathcal{O}(x^2 + \log(|y|))$

Live Demo

```
(GOAL COMPLEXITY)
(STARTTERM (FUNCTIONSYMBOLS l0))
(RETURNLOCATIONS (FUNCTIONSYMBOLS f2))
(VAR x y a)
(RULES
  l0(x,y,a) -> l1(x,y,a)
  l1(x,y,a) -> l1(x-1,y+f1[a|(x,y,a)<-(x,y,x)],a) :|: x > 0
  l1(x,y,a) -> l2(1,y,a) :|: x <= 0
  l2(x,y,a) -> l2(3*x,2*y,a) :|: x < y && x > 0

  f1(x,y,a) -> f2(x,y,1) :|: a = 0
  f1(x,y,a) -> f2(x,y,a*f1[a|(x,y,a)<-(x,y,a - 1)]) :|: a > 0)
```

https://koat.verify.rwth-aachen.de

Evaluation of our Implementation in KoAT2

- Complexity_ITS consisting of 838 benchmarks from TPDB
- 45 new benchmarks with function calls

Evaluation of our Implementation in KoAT2

- Complexity_ITS consisting of 838 benchmarks from TPDB
- 45 new benchmarks with function calls
- $\mathrm{AVG}^+(\mathrm{s})$ average successful runtime vs $\mathrm{AVG}(\mathrm{s})$ average runtime

	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$< \infty$	$\mathrm{AVG}^+(\mathrm{s})$	$\mathrm{AVG}(\mathrm{s})$
CoFloCo	125	0	234	95	10	464	3.19	16.38
KoAT1	132	0	231	108	16	499	0.64	8.06
KoAT2	128	11	281	121	28	583	3.45	21.78

Evaluation of our Implementation in KoAT2

- Complexity_ITS consisting of 838 benchmarks from TPDB
- 45 new benchmarks with function calls
- $\mathrm{AVG}^{+}(\mathrm{s})$ average successful runtime vs $\mathrm{AVG}(\mathrm{s})$ average runtime

	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$< \infty$	$\mathrm{AVG}^{+}(\mathrm{s})$	$\mathrm{AVG}(\mathrm{s})$
CoFloCo	125	0	234	95	10	464	3.19	16.38
KoAT1	132	0	231	108	16	499	0.64	8.06
KoAT2	128	11	281	121	28	583	3.45	21.78

- At most 652 benchmarks might terminate

Evaluation of our Implementation in KoAT2

- Complexity_ITS consisting of 838 benchmarks from TPDB
- 45 new benchmarks with function calls
- $\mathrm{AVG}^+(\mathrm{s})$ average successful runtime vs $\mathrm{AVG}(\mathrm{s})$ average runtime

	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$< \infty$	$\mathrm{AVG}^+(\mathrm{s})$	$\mathrm{AVG}(\mathrm{s})$	succ. rate
CoFloCo	125	0	234	95	10	464	3.19	16.38	71%
KoAT1	132	0	231	108	16	499	0.64	8.06	76%
KoAT2	128	11	281	121	28	583	3.45	21.78	89%

- At most 652 benchmarks might terminate
- KoAT2 solves 89% of benchmarks which might terminate

Conclusion

KoAT uses

Conclusion

KoAT uses

- a modular approach to compute runtime bounds combining

Conclusion

KoAT uses

- a modular approach to compute runtime bounds combining
  - novel ranking functions for programs
    with function calls

Conclusion

KoAT uses

- a modular approach to compute runtime bounds combining
  - novel ranking functions for programs with function calls
  - a procedure to handle twn-loops (IJCAR '22, JAR '26)

Conclusion

KoAT uses

- a modular approach to compute runtime bounds combining
  - novel ranking functions for programs with function calls
  - a procedure to handle twn-loops (IJCAR '22, JAR '26)
- a modular approach to compute size bounds combining

Conclusion

KoAT uses

- a modular approach to compute runtime bounds combining
  - novel ranking functions for programs with function calls
  - a procedure to handle twn-loops (IJCAR '22, JAR '26)
- a modular approach to compute size bounds combining
  - novel change-accumulated size bounds for recursive programs

Conclusion

KoAT uses

- a modular approach to compute runtime bounds combining
  - novel ranking functions for programs with function calls
  - a procedure to handle twn-loops (IJCAR '22, JAR '26)
- a modular approach to compute size bounds combining
  - novel change-accumulated size bounds for recursive programs
  - a procedure using closed-forms (FroCoS '23, JAR '26)

Conclusion

KoAT uses

- a modular approach to compute runtime bounds combining
  - novel ranking functions for programs with function calls
  - a procedure to handle twn-loops (IJCAR '22, JAR '26)
- a modular approach to compute size bounds combining
  - novel change-accumulated size bounds for recursive programs
  - a procedure using closed-forms (FroCoS '23, JAR '26)

`https://koat.verify.rwth-aachen.de`

Conclusion

KoAT uses

- a modular approach to compute runtime bounds combining
  - novel ranking functions for programs with function calls
  - a procedure to handle twn-loops (IJCAR '22, JAR '26)
- a modular approach to compute size bounds combining
  - novel change-accumulated size bounds for recursive programs
  - a procedure using closed-forms (FroCoS '23, JAR '26)

`https://koat.verify.rwth-aachen.de`

Conclusion

KoAT uses

- a modular approach to compute runtime bounds combining
  - novel ranking functions for programs with function calls
  - a procedure to handle twn-loops (IJCAR '22, JAR '26)
- a modular approach to compute size bounds combining
  - novel change-accumulated size bounds for recursive programs
  - a procedure using closed-forms (FroCoS '23, JAR '26)

`https://koat.verify.rwth-aachen.de`

Thank You!