Automated Termination and Complexity Analysis of Programs

Carsten Fuhs

Birkbeck, University of London

EuroProofNet Summer School on Verification Technology, Systems & Applications 2022

Saarbrücken, Germany

5 & 7 September 2022

https://www.dcs.bbk.ac.uk/~carsten/vtsa2022/
Two approaches:

Dynamic analysis:
- Run the program on example inputs (testing).
- Goal: find errors
- Requires good choice of test cases
- In general, no guarantee for absence of errors

Static analysis:
- Analyse the program text without actually running the program.
- Can prove (verify) correctness of the program
- Important for safety-critical applications
- Motivating example: first flight of Ariane 5 rocket in 1996
  
  https://www.youtube.com/watch?v=PK_yguLapgA
  
  https://en.wikipedia.org/wiki/Ariane_5_Flight_501

Manual static analysis requires high effort and expertise

⇒ For broad applicability: Build automatic tools for static analysis!
Quality Assurance for Software by Program Analysis

Two approaches:

- **Dynamic analysis:**
  Run the program on example inputs (testing).
  + goal: find errors
  - requires good choice of test cases
  - in general no guarantee for absence of errors

- **Static analysis:**
  Analyse the program text without actually running the program.
  + can prove (verify) correctness of the program
  - → important for safety-critical applications
  - → motivating example: first flight of Ariane 5 rocket in 1996
    https://www.youtube.com/watch?v=PK_yguLapgA
    https://en.wikipedia.org/wiki/Ariane_5_Flight_501
  - manual static analysis requires high effort and expertise
  ⇒ for broad applicability:
    Build automatic tools for static analysis!
Quality Assurance for Software by Program Analysis

Two approaches:

- **Dynamic analysis:**
  Run the program on example inputs (testing).
  - goal: find errors
  - requires good choice of test cases
  - in general no guarantee for absence of errors

- **Static analysis:**
  Analyse the program text without actually running the program.
  - can prove (verify) correctness of the program
    - important for safety-critical applications
    - motivating example: first flight of Ariane 5 rocket in 1996
      - manual static analysis requires high effort and expertise
      ⇒ for broad applicability:

      Build automatic tools for static analysis!
For the user (programmer): Use static analysis tools as “black boxes”.

- Partial Correctness: will my program always produce the right result?
  - Assertions by the programmer: `assert x > 0`
  - Equivalence: Do two programs always produce the same result?
  - Confluence: Does my program always produce the same result? Confluence is a property that establishes the global determinism of a computation despite possible local non-determinism.
  - Correctness of refactoring: does the order of applying compiler optimisation rules matter?
For the user (programmer): Use static analysis tools as “black boxes”.

What properties of programs do we want to analyse?
For the user (programmer): Use static analysis tools as “black boxes”.

What properties of programs do we want to analyse?

- **Partial Correctness**
  - will my program always produce the right result?
For the user (programmer): Use static analysis tools as “black boxes”.

What properties of programs do we want to analyse?

- **Partial Correctness**
  → will my program always produce the right result?

- **Assertions by the programmer.**
  
  ```
  assert x > 0
  ```
  
  → will this always be true?
For the user (programmer): Use static analysis tools as “black boxes”.

What properties of programs do we want to analyse?

- **Partial Correctness**
  → will my program always produce the right result?

- **Assertions by the programmer.** assert \( x > 0 \)
  → will this always be true?

- **Equivalence.** Do two programs always produce the same result?
  → correctness of refactoring
For the user (programmer): Use static analysis tools as “black boxes”.

What properties of programs do we want to analyse?

- **Partial Correctness**
  - will my program always produce the right result?

- **Assertions by the programmer.**
  - assert $x > 0$
  - will this always be true?

- **Equivalence.** Do two programs always produce the same result?
  - correctness of refactoring

- **Confluence.** For languages with non-deterministic rules/commands:
  - Does my program always produce the same result?

  Confluence is a property that establishes the global determinism of a computation despite possible local non-determinism.

  [Hristakiev, PhD thesis ’17]

  - does the order of applying compiler optimisation rules matter?
- Memory Safety
  - are my memory accesses always legal?
    
    ```c
    int* x = NULL; *x = 42;
    ```
  - undefined behaviour!
  - memory safety matters: Heartbleed (OpenSSL attack)
Memory Safety
- are my memory accesses always legal?
  - `int* x = NULL; *x = 42;`
  - undefined behaviour!
  - memory safety matters: Heartbleed (OpenSSL attack)

Termination
- will my program give an output for all inputs in finitely many steps?
Static analysis: the user’s perspective (2/2)

- **Memory Safety**
  - are my memory accesses always legal?
    - `int* x = NULL; *x = 42;`
  - undefined behaviour!
  - memory safety matters: Heartbleed (OpenSSL attack)

- **Termination**
  - will my program give an output for all inputs in finitely many steps?

- **(Quantitative) Resource Use aka Complexity**
  - how many steps will my program need in the worst case? (runtime complexity)
  - how large can my data become? (size complexity)
Static analysis: the user’s perspective (2/2)

- **Memory Safety**
  - are my memory accesses always legal?
    
    ```c
    int* x = NULL; *x = 42;
    ```
  - undefined behaviour!
  - memory safety matters: Heartbleed (OpenSSL attack)

- **Termination**
  - will my program give an output for all inputs in finitely many steps?

- **(Quantitative) Resource Use aka Complexity**
  - how many steps will my program need in the worst case? (runtime complexity)
  - how large can my data become? (size complexity)

**Note:** All these properties are **undecidable**!
⇒ use automatable sufficient criteria in practice
Program analysis tool developed in Aachen, London, Innsbruck, ...
Program analysis tool developed in Aachen, London, Innsbruck, ...

- Fully automated, hundreds of techniques for termination, time complexity bounds, ...

Termination
Complexity
Non-Termination
Program analysis tool developed in Aachen, London, Innsbruck, ...  
Fully automated, hundreds of techniques for termination, time complexity bounds, ...  
Highly configurable via strategy language
Program analysis tool developed in Aachen, London, Innsbruck, ...

- Fully automated, hundreds of techniques for termination, time complexity bounds, ...
- Highly configurable via strategy language
- Proofs usually have many steps → construct proof tree
Program analysis tool developed in Aachen, London, Innsbruck, ...

- Fully automated, hundreds of techniques for termination, time complexity bounds, ...
- Highly configurable via strategy language
- Proofs usually have many steps → construct proof tree
- Founding tool of Termination Competition, since 2004
- Initially: analyse termination of term rewrite systems (TRSs), later also complexity bounds
• Program analysis tool developed in Aachen, London, Innsbruck, …
• Fully automated, hundreds of techniques for termination, time complexity bounds, …
• Highly configurable via strategy language
• Proofs usually have many steps → construct proof tree
• Founding tool of Termination Competition, since 2004
• Initially: analyse termination of term rewrite systems (TRSs), later also complexity bounds
• Since 2006 more input languages: Prolog, Haskell, Java, C (via LLVM)
Program analysis tool developed in Aachen, London, Innsbruck, …
Fully automated, hundreds of techniques for termination, time complexity bounds, …
Highly configurable via strategy language
Proofs usually have many steps → construct proof tree
Founding tool of Termination Competition, since 2004
Initially: analyse termination of term rewrite systems (TRSs), later also complexity bounds
Since 2006 more input languages: Prolog, Haskell, Java, C (via LLVM)
  1 dedicated program analysis by symbolic execution and abstraction
Program analysis tool developed in Aachen, London, Innsbruck, ...
- Fully automated, hundreds of techniques for termination, time complexity bounds, ...
- Highly configurable via strategy language
- Proofs usually have many steps → construct proof tree
- Founding tool of Termination Competition, since 2004
- Initially: analyse termination of term rewrite systems (TRSs), later also complexity bounds
- Since 2006 more input languages: Prolog, Haskell, Java, C (via LLVM)
  1. dedicated program analysis by symbolic execution and abstraction
  2. extract rewrite system
Program analysis tool developed in Aachen, London, Innsbruck, ... 
Fully automated, hundreds of techniques for termination, time complexity bounds, ...
Highly configurable via strategy language
Proofs usually have many steps \( \rightarrow \) construct proof tree
Founding tool of Termination Competition, since 2004
Initially: analyse termination of term rewrite systems (TRSs), later also complexity bounds
Since 2006 more input languages: Prolog, Haskell, Java, C (via LLVM)
1. dedicated program analysis by symbolic execution and abstraction
2. extract rewrite system
3. termination of rewrite system \( \Rightarrow \) termination of program

![Diagram showing the flow of information from frontends to backend]

Frontends

Symbolic Execution Graph

TRS

Termination
Complexity
Non-Termination

Backend
Program analysis tool developed in Aachen, London, Innsbruck, ...  
Fully automated, hundreds of techniques for termination, time complexity bounds, ...  
Highly configurable via strategy language  
Proofs usually have many steps $\Rightarrow$ construct proof tree  
Founding tool of Termination Competition, since 2004  
Initially: analyse termination of term rewrite systems (TRSs), later also complexity bounds  
Since 2006 more input languages: Prolog, Haskell, Java, C (via LLVM)

1. dedicated program analysis by symbolic execution and abstraction
2. extract constrained rewrite system (constraints in integer arithmetic)
3. termination of constrained rewrite system $\Rightarrow$ termination of program
What is Static Program Analysis About?

**Goal:** (Automatically) prove whether a given program $P$ has (un)desirable property

**Approach:** Often in two phases
What is Static Program Analysis About?

Goal: (Automatically) prove whether a given program $P$ has (un)desirable property

Approach: Often in two phases

Front-End
- Input: Program in Java, C, Prolog, Haskell, ...
- Output: Mathematical representation amenable to automated analysis (usually some kind of transition system)
- Often over-approximation, preserves the property of interest
What is Static Program Analysis About?

**Goal:** (Automatically) prove whether a given program $P$ has (un)desirable property

**Approach:** Often in two phases

**Front-End**
- Input: Program in Java, C, Prolog, Haskell, …
- Output: Mathematical representation amenable to automated analysis (usually some kind of transition system)
- Often over-approximation, preserves the property of interest

**Back-End**
- Performs the analysis of the desired property
  ⇒ Result carries over to original program
I. Termination Analysis
Why Analyse Termination?

Program
Input handler
Mathematical proof
Biological process

Variations of the same problem:

2 special case of
3 can be interpreted as
4 probabilistic version of

2011: PHP and Java issues with floating-point number parser
http://www.exploringbinary.com/java-hangs-when-converting-2-2250738585072012e-308/
Why Analyse Termination?

1. **Program**: produces result
Why Analyse Termination?

1. **Program**: produces result
2. **Input handler**: system reacts
Why Analyse Termination?

1. **Program**: produces result
2. **Input handler**: system reacts
3. **Mathematical proof**: the induction is valid

Additional notes:

- Variations of the same problem:
  - Special case of...
  - Can be interpreted as...
  - Probabilistic version of...

- PHP and Java issues with floating-point number parser:
Why Analyse Termination?

1. **Program**: produces result
2. **Input handler**: system reacts
3. **Mathematical proof**: the induction is valid
4. **Biological process**: reaches a stable state
Why Analyse Termination?

1. **Program**: produces result
2. **Input handler**: system reacts
3. **Mathematical proof**: the induction is valid
4. **Biological process**: reaches a stable state

Variations of the same problem:

- 2. special case of 1
- 3. can be interpreted as 1
- 4. probabilistic version of 1
Why Analyse Termination?

1. **Program**: produces result
2. **Input handler**: system reacts
3. **Mathematical proof**: the induction is valid
4. **Biological process**: reaches a stable state

Variations of the same problem:

2. special case of 1
3. can be interpreted as 1
4. probabilistic version of 1

2011: PHP and Java issues with floating-point number parser

The Bad News

Theorem (Turing 1936)

*The question if a given program terminates on a fixed input is undecidable.*
The Bad News

**Theorem (Turing 1936)**

*The question if a given program terminates on a fixed input is undecidable.*

- We want to solve the (harder) question if a given program terminates on all inputs.
The Bad News

Theorem (Turing 1936)

*The question if a given program terminates on a fixed input is undecidable.*

- We want to solve the (harder) question if a given program terminates on *all* inputs.
- That’s not even semi-decidable!
The Bad News

**Theorem (Turing 1936)**

*The question if a given program terminates on a fixed input is undecidable.*

- We want to solve the (harder) question if a given program terminates on **all** inputs.
- That’s not even semi-decidable!
- But, fear not . . .
Termination Analysis, Classically

Turing 1949

“Finally the checker has to verify that the process comes to an end. [...] This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.”
“Finally the checker has to verify that the process comes to an end. [...] This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.”

1. Find ranking function $f$ (“quantity”)
Termination Analysis, Classically

Turing 1949

"Finally the checker has to verify that the process comes to an end. [...] This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops."

1. Find ranking function $f$ ("quantity")
2. Prove $f$ to have a lower bound ("vanish when the machine stops")
Termination Analysis, Classically

Turing 1949

“Finally the checker has to verify that the process comes to an end. [...] This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.”

1. Find ranking function $f$ (“quantity”)
2. Prove $f$ to have a lower bound (“vanish when the machine stops”)
3. Prove that $f$ decreases over time
Finally the checker has to verify that the process comes to an end. [...] This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.

1. Find ranking function $f$ (“quantity”)
2. Prove $f$ to have a lower bound (“vanish when the machine stops”)
3. Prove that $f$ decreases over time

Example (Termination can be simple)

\[
\textbf{while } x > 0:\ \\
\quad x = x - 1
\]
Question: Does program $P$ terminate?

Answer:

1. $\phi$ satisfiable, model $M$ (e.g., $a = 3$, $b = 1$, $c = 1$): $\Rightarrow P$ terminating, $M$ fills in the gaps in the termination proof

2. $\phi$ unsatisfiable: $\Rightarrow$ termination status of $P$ unknown $\Rightarrow$ try a different template (proof technique)

In practice:

- Encode only one proof step $\Rightarrow$ try to prove only part of the program terminating
- Repeat until the whole program is proved terminating
Question: Does program $P$ terminate?

Approach: Encode termination proof \textit{template} to logical constraint $\varphi$, ask SMT solver

Answer:

1. If $\varphi$ is satisfiable, model $M$ (e.g., $a = 3$, $b = 1$, $c = 1$):
   \[\Rightarrow P \text{ terminating, } M \text{ fills in the gaps in the termination proof}\]

2. If $\varphi$ is unsatisfiable:
   \[\Rightarrow \text{termination status of } P \text{ unknown}\]
   \[\Rightarrow \text{try a different template (proof technique)}\]

In practice:

Encode only one proof step → try to prove only part of the program terminating

Repeat until the whole program is proved terminating
Question: Does program $P$ terminate?

Approach: Encode termination proof template to logical constraint $\varphi$, ask SMT solver

$\rightarrow$ SMT = SATisfiability Modulo Theories, solve constraints like

$$b > 0 \land (4ab - 7b^2 > 1 \lor 3a + c \geq b^3)$$
Question: Does program $P$ terminate?

Approach: Encode termination proof template to logical constraint $\varphi$, ask SMT solver

$\rightarrow$ SMT = SATisfiability Modulo Theories, solve constraints like

$$b > 0 \land (4ab - 7b^2 > 1 \lor 3a + c \geq b^3)$$

Answer:
Question: Does program $P$ terminate?

Approach: Encode termination proof template to logical constraint $\varphi$, ask SMT solver

$\text{SMT} = \text{SATisfiability Modulo Theories}$, solve constraints like

$$b > 0 \land (4ab - 7b^2 > 1 \lor 3a + c \geq b^3)$$

Answer:

1. $\varphi$ satisfiable, model $M$ (e.g., $a = 3, b = 1, c = 1$):
   $\Rightarrow P$ terminating, $M$ fills in the gaps in the termination proof
Question: Does program $P$ terminate?

Approach: Encode termination proof template to logical constraint $\varphi$, ask SMT solver

$\rightarrow$ \textbf{SMT} = \textbf{SAT}isfiability \textbf{Modulo} \textbf{T}heories, solve constraints like

$$b > 0 \land (4ab - 7b^2 > 1 \lor 3a + c \geq b^3)$$

Answer:

1. $\varphi$ satisfiable, model $M$ (e.g., $a = 3, b = 1, c = 1$):
   $\Rightarrow$ $P$ terminating, $M$ fills in the gaps in the termination proof

2. $\varphi$ unsatisfiable:
   $\Rightarrow$ termination status of $P$ unknown
   $\Rightarrow$ try a different template (proof technique)
Question: Does program $P$ terminate?

Approach: Encode termination proof template to logical constraint $\varphi$, ask SMT solver

$\rightarrow \text{SMT = SATisfiability Modulo Theories, solve constraints like}$

$$b > 0 \land (4ab - 7b^2 > 1 \lor 3a + c \geq b^3)$$

Answer:

1. $\varphi$ satisfiable, model $M$ (e.g., $a = 3, b = 1, c = 1$):
   $\Rightarrow P$ terminating, $M$ fills in the gaps in the termination proof

2. $\varphi$ unsatisfiable:
   $\Rightarrow$ termination status of $P$ unknown
   $\Rightarrow$ try a different template (proof technique)

In practice:

- Encode only one proof step at a time
  $\rightarrow$ try to prove only part of the program terminating
- Repeat until the whole program is proved terminating
The Rest of Today’s Session

Termination proving in the back-end

1. Term Rewrite Systems (TRSs)
2. Imperative Programs (as Integer Transition Systems, ITSs)
3. Both together! Logically Constrained Term Rewrite Systems
The Rest of Today’s Session

Termination proving in the back-end
1. Term Rewrite Systems (TRSs)
2. Imperative Programs (as Integer Transition Systems, ITSs)
3. Both together! Logically Constrained Term Rewrite Systems

Processing practical programming languages in the front-end
4. Java
5. C (via LLVM)
I.1 Termination Analysis of Term Rewrite Systems
What’s Term Rewriting?

Syntactic approach for reasoning in equational first-order logic

Core functional programming language without many restrictions

- Features of "real" FP:
  - first-order (usually)
  - no fixed evaluation strategy
  - non-determinism!
  - no fixed order of rules to apply (Haskell: top to bottom)
    - non-determinism!
  - untyped (unless you really want types)
  - no pre-defined data structures (integers, arrays, ...)

14/173
What’s Term Rewriting?

Syntactic approach for reasoning in equational first-order logic
What’s Term Rewriting?

Syntactic approach for reasoning in equational first-order logic

Core functional programming language without many restrictions (and features) of “real” FP:
What’s Term Rewriting?

Syntactic approach for reasoning in equational first-order logic

Core functional programming language without many restrictions (and features) of “real” FP:

- first-order (usually)
- no fixed evaluation strategy → non-determinism!
- no fixed order of rules to apply (Haskell: top to bottom) → non-determinism!
- untyped (unless you really want types)
- no pre-defined data structures (integers, arrays, ...
Represent natural numbers by terms (inductively defined data structure):

\[0, \, s(0), \, s(s(0)), \, \ldots\]
Represent natural numbers by terms (inductively defined data structure):

\[ 0, \ s(0), \ s(s(0)), \ldots \]

Example (A Term Rewrite System (TRS) for Division)

\[ \mathcal{R} = \begin{cases} 
  \text{minus}(x, 0) & \rightarrow \ x \\
  \text{minus}(s(x), s(y)) & \rightarrow \ \text{minus}(x, y) \\
  \text{quot}(0, s(y)) & \rightarrow \ 0 \\
  \text{quot}(s(x), s(y)) & \rightarrow \ s(\text{quot}(\text{minus}(x, y), s(y))) 
\end{cases} \]
Show Me an Example!

Represent natural numbers by terms (inductively defined data structure):

\[ 0, \ s(0), \ s(s(0)), \ldots \]

**Example (A Term Rewrite System (TRS) for Division)**

\[
\mathcal{R} = \begin{cases}
    \text{minus}(x, 0) & \rightarrow x \\
    \text{minus}(s(x), s(y)) & \rightarrow \text{minus}(x, y) \\
    \text{quot}(0, s(y)) & \rightarrow 0 \\
    \text{quot}(s(x), s(y)) & \rightarrow s(\text{quot}(\text{minus}(x, y), s(y)))
\end{cases}
\]

Calculation:

\[
\text{minus}(s(s(0)), s(0)) \rightarrow_{\mathcal{R}} \text{minus}(s(0), 0) \rightarrow_{\mathcal{R}} s(0)
\]
Why Care about Termination of Term Rewriting?

- Termination needed by theorem provers
Why Care about Termination of Term Rewriting?

- Termination needed by theorem provers
- Translate program $P$ with inductive data structures (trees) to TRS, represent data structures as terms
  $\Rightarrow$ Termination of TRS implies termination of $P$
Why Care about Termination of Term Rewriting?

- Termination needed by theorem provers

- Translate program \( P \) with inductive data structures (trees) to TRS, represent data structures as terms

  \( \Rightarrow \) Termination of TRS implies termination of \( P \)

- Logic programming: Prolog
  
Why Care about Termination of Term Rewriting?

- Termination needed by theorem provers

- Translate program $P$ with inductive data structures (trees) to TRS, represent data structures as terms

\[ \Rightarrow \text{Termination of TRS implies termination of } P \]

- Logic programming: Prolog

- (Lazy) functional programming: Haskell [Giesl et al, *TOPLAS ’11*]
Why Care about Termination of Term Rewriting?

- Termination needed by theorem provers

- Translate program $P$ with inductive data structures (trees) to TRS, represent data structures as terms

  $\Rightarrow$ Termination of TRS implies termination of $P$

- Logic programming: Prolog
  [van Raamsdonk, *ICLP* ’97; Schneider-Kamp et al, *TOCL* ’09; Giesl et al, *PPDP* ’12]


- Object-oriented programming: Java [Otto et al, *RTA* ’10]
Example (Division)

\[ \mathcal{R} = \left\{ \begin{array}{l}
\text{minus}(x, 0) \rightarrow x \\
\text{minus}(s(x), s(y)) \rightarrow \text{minus}(x, y) \\
\text{quot}(0, s(y)) \rightarrow 0 \\
\text{quot}(s(x), s(y)) \rightarrow s(\text{quot}(\text{minus}(x, y), s(y)))
\end{array} \right. \]

Term rewriting: Evaluate terms by applying rules from \( \mathcal{R} \)

\[ \text{minus}(s(s(0)), s(0)) \rightarrow_{\mathcal{R}} \text{minus}(s(0), 0) \rightarrow_{\mathcal{R}} s(0) \]
Example (Division)

\[ \mathcal{R} = \begin{cases} 
\text{minus}(x, 0) & \rightarrow x \\
\text{minus}(s(x), s(y)) & \rightarrow \text{minus}(x, y) \\
\text{quot}(0, s(y)) & \rightarrow 0 \\
\text{quot}(s(x), s(y)) & \rightarrow s(\text{quot}(\text{minus}(x, y), s(y))) 
\end{cases} \]

Term rewriting: Evaluate terms by applying rules from \( \mathcal{R} \)

\[ \text{minus}(s(s(0)), s(0)) \rightarrow_{\mathcal{R}} \text{minus}(s(0), 0) \rightarrow_{\mathcal{R}} s(0) \]

Termination: No infinite evaluation sequences \( t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} t_3 \rightarrow_{\mathcal{R}} \ldots \)
Example (Division)

\[ R = \begin{cases} 
\text{minus}(x, 0) & \rightarrow x \\
\text{minus}(s(x), s(y)) & \rightarrow \text{minus}(x, y) \\
\text{quot}(0, s(y)) & \rightarrow 0 \\
\text{quot}(s(x), s(y)) & \rightarrow s(\text{quot}(\text{minus}(x, y), s(y)))
\end{cases} \]

Term rewriting: Evaluate terms by applying rules from \( R \)

\[ \text{minus}(s(s(0)), s(0)) \rightarrow_R \text{minus}(s(0), 0) \rightarrow_R s(0) \]

Termination: No infinite evaluation sequences \( t_1 \rightarrow_R t_2 \rightarrow_R t_3 \rightarrow_R \ldots \)

Show termination using Dependency Pairs
Example (Division)

\[ \mathcal{R} = \left\{ \begin{array}{c}
\text{minus}(x, 0) \rightarrow x \\
\text{minus}(s(x), s(y)) \rightarrow \text{minus}(x, y) \\
\text{quot}(0, s(y)) \rightarrow 0 \\
\text{quot}(s(x), s(y)) \rightarrow s(\text{quot}(\text{minus}(x, y), s(y)))
\end{array} \right\} \]

Dependency Pairs [Arts, Giesl, TCS ’00]
### Example (Division)

\[
\mathcal{R} = \begin{cases} 
\text{minus}(x, 0) & \rightarrow x \\
\text{minus}(s(x), s(y)) & \rightarrow \text{minus}(x, y) \\
\text{quot}(0, s(y)) & \rightarrow 0 \\
\text{quot}(s(x), s(y)) & \rightarrow s(\text{quot}(\text{minus}(x, y), s(y)))
\end{cases}
\]

\[
\mathcal{DP} = \begin{cases} 
\text{minus}^\#(s(x), s(y)) & \rightarrow \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \rightarrow \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \rightarrow \text{quot}^\#(\text{minus}(x, y), s(y))
\end{cases}
\]

### Dependency Pairs [Arts, Giesl, TCS '00]

- For TRS \( \mathcal{R} \) build dependency pairs \( \mathcal{DP} \) \((\sim \text{function calls})\)
- Show: No \( \infty \) call sequence with \( \mathcal{DP} \) (eval of \( \mathcal{DP} \)'s args via \( \mathcal{R} \))
Example (Division)

\[ R = \begin{cases} 
  \text{minus}(x, 0) & \rightarrow x \\
  \text{minus}(s(x), s(y)) & \rightarrow \text{minus}(x, y) \\
  \text{quot}(0, s(y)) & \rightarrow 0 \\
  \text{quot}(s(x), s(y)) & \rightarrow s(\text{quot}(\text{minus}(x, y), s(y))) 
\end{cases} \]

\[ DP = \begin{cases} 
  \text{minus}^#(s(x), s(y)) & \rightarrow \text{minus}^#(x, y) \\
  \text{quot}^#(s(x), s(y)) & \rightarrow \text{minus}^#(x, y) \\
  \text{quot}^#(s(x), s(y)) & \rightarrow \text{quot}^#(\text{minus}(x, y), s(y)) 
\end{cases} \]

Dependency Pairs [Arts, Giesl, TCS ’00]

- For TRS \( R \) build dependency pairs \( DP \) (\( \sim \) function calls)
- Show: No infinite call sequence with \( DP \) (eval of \( DP \)'s args via \( R \))
- Dependency Pair Framework [Giesl et al, JAR ’06] (simplified):
Example (Division)

\[ \mathcal{R} = \begin{cases} 
\text{minus}(x, 0) & \rightarrow & x \\
\text{minus}(s(x), s(y)) & \rightarrow & \text{minus}(x, y) \\
\text{quot}(0, s(y)) & \rightarrow & 0 \\
\text{quot}(s(x), s(y)) & \rightarrow & s(\text{quot}(\text{minus}(x, y), s(y))) 
\end{cases} \]

\[ \mathcal{DP} = \begin{cases} 
\text{minus}^\#(s(x), s(y)) & \rightarrow & \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \rightarrow & \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \rightarrow & \text{quot}^\#(\text{minus}(x, y), s(y)) 
\end{cases} \]

Dependency Pairs [Arts, Giesl, TCS ’00]

- For TRS \( \mathcal{R} \) build dependency pairs \( \mathcal{DP} \) (∼ function calls)
- Show: No \( \infty \) call sequence with \( \mathcal{DP} \) (eval of \( \mathcal{DP} \)'s args via \( \mathcal{R} \))
- Dependency Pair Framework [Giesl et al, JAR ’06] (simplified): while \( \mathcal{DP} \neq \emptyset \) :
Example (Division)

\[ R = \begin{cases} 
\text{minus}(x, 0) & \sim x \\
\text{minus}(s(x), s(y)) & \sim \text{minus}(x, y) \\
\text{quot}(0, s(y)) & \sim 0 \\
\text{quot}(s(x), s(y)) & \sim s(\text{quot}(\text{minus}(x, y), s(y))) 
\end{cases} \]

\[ DP = \begin{cases} 
\text{minus}^\#(s(x), s(y)) & \sim \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \sim \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \sim \text{quot}^\#(\text{minus}(x, y), s(y)) 
\end{cases} \]

Dependency Pairs [Arts, Giesl, TCS ’00]

- For TRS \( R \) build dependency pairs \( DP \) \((\sim \text{function calls})\)
- Show: No \( \infty \) call sequence with \( DP \) (eval of \( DP \)'s args via \( R \))
- Dependency Pair Framework [Giesl et al, JAR ’06] (simplified):

  \[ \text{while } DP \neq \emptyset : \]
  - find well-founded order \( \succ \) with \( DP \cup R \subseteq \succeq \)
Example (Division)

\[ R = \begin{cases} 
\text{minus}(x, 0) & \leadsto x \\
\text{minus}(s(x), s(y)) & \leadsto \text{minus}(x, y) \\
\text{quot}(0, s(y)) & \leadsto 0 \\
\text{quot}(s(x), s(y)) & \leadsto s(\text{quot}(\text{minus}(x, y), s(y))) 
\end{cases} \]

\[ \mathcal{DP} = \begin{cases} 
\text{minus}^\#(s(x), s(y)) & \leadsto \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \leadsto \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \leadsto \text{quot}^\#(\text{minus}(x, y), s(y)) 
\end{cases} \]

Dependency Pairs [Arts, Giesl, TCS ’00]

- For TRS \( R \) build dependency pairs \( \mathcal{DP} \) (~ function calls)
- Show: No \( \infty \) call sequence with \( \mathcal{DP} \) (eval of \( \mathcal{DP} \)'s args via \( R \))
- Dependency Pair Framework [Giesl et al, JAR ’06] (simplified):
  - while \( \mathcal{DP} \neq \emptyset \):
    - find well-founded order \( \succ \) with \( \mathcal{DP} \cup R \subseteq \succ \)
    - delete \( s \rightarrow t \) with \( s \succ t \) from \( \mathcal{DP} \)
**Example (Division)**

\[
\mathcal{R} = \begin{cases} 
\text{minus}(x, 0) & \sim x \\
\text{minus}(s(x), s(y)) & \sim \text{minus}(x, y) \\
\text{quot}(0, s(y)) & \sim 0 \\
\text{quot}(s(x), s(y)) & \sim s(\text{quot}(\text{minus}(x, y), s(y)))
\end{cases}
\]

\[
\mathcal{DP} = \begin{cases} 
\text{minus}^\#(s(x), s(y)) & \sim \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \sim \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \sim \text{quot}^\#(\text{minus}(x, y), s(y))
\end{cases}
\]

**Dependency Pairs** [Arts, Giesl, *TCS '00*]

- For TRS \( \mathcal{R} \) build dependency pairs \( \mathcal{DP} \) (\( \sim \) function calls)
- Show: No \( \infty \) call sequence with \( \mathcal{DP} \) (eval of \( \mathcal{DP} \)'s args via \( \mathcal{R} \))
- Dependency Pair Framework [Giesl et al, *JAR '06*] (simplified):
  - **while** \( \mathcal{DP} \neq \emptyset \):
    - find well-founded order \( \succ \) with \( \mathcal{DP} \cup \mathcal{R} \subseteq \succ \)
    - delete \( s \rightarrow t \) with \( s \succ t \) from \( \mathcal{DP} \)
  - Find \( \succ \) automatically and efficiently
Get $\succ$ via polynomial interpretations $[\cdot]$ over $\mathbb{N}$ [Lankford ’75]

**Example**

\[
\text{minus}(s(x), s(y)) \preceq \text{minus}(x, y)
\]
Get \( \succcurlyeq \) via polynomial interpretations \([ \cdot ]\) over \( \mathbb{N} \) [Lankford ’75]

**Example**

\[
\text{minus}(s(x), s(y)) \succcurlyeq \text{minus}(x, y)
\]

Use \([ \cdot ]\) with

- \([\text{minus}](x_1, x_2) = x_1\)
- \([s](x_1) = x_1 + 1\)
Polynomial Interpretations

Get $\succ$ via polynomial interpretations $[\cdot]$ over $\mathbb{N}$ [Lankford '75]

Example

$\forall x, y. \; x + 1 = [\text{minus}(s(x), s(y))] \geq [\text{minus}(x, y)] = x$

Use $[\cdot]$ with

- $[\text{minus}](x_1, x_2) = x_1$
- $[s](x_1) = x_1 + 1$

Extend to terms:

- $[x] = x$
- $[f(t_1, \ldots, t_n)] = [f]([t_1], \ldots, [t_n])$

$\succ$ boils down to $>$ over $\mathbb{N}$
Example (Constraints for Division)

\[ \mathcal{R} = \begin{cases} 
\text{minus}(x, 0) & \implies x \\
\text{minus}(s(x), s(y)) & \implies \text{minus}(x, y) \\
\text{quot}(0, s(y)) & \implies 0 \\
\text{quot}(s(x), s(y)) & \implies s(\text{quot}(\text{minus}(x, y), s(y))) 
\end{cases} \]

\[ \mathcal{DP} = \begin{cases} 
\text{minus}^\#(s(x), s(y)) & \implies \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \implies \text{minus}^\#(x, y) \\
\text{quot}^\#(s(x), s(y)) & \implies \text{quot}^\#(\text{minus}(x, y), s(y)) 
\end{cases} \]
Example (Constraints for Division)

\[ \mathcal{R} = \begin{cases} 
\text{minus}(x, 0) & \rightarrow x \\
\text{minus}(s(x), s(y)) & \rightarrow \text{minus}(x, y) \\
\text{quot}(0, s(y)) & \rightarrow 0 \\
\text{quot}(s(x), s(y)) & \rightarrow s(\text{quot}(\text{minus}(x, y), s(y))) 
\end{cases} \]

\[ \mathcal{D}\mathcal{P} = \begin{cases} 
\text{minus}^#(s(x), s(y)) & \succ \text{minus}^#(x, y) \\
\text{quot}^#(s(x), s(y)) & \succ \text{minus}^#(x, y) \\
\text{quot}^#(s(x), s(y)) & \succ \text{quot}^#(\text{minus}(x, y), s(y)) 
\end{cases} \]

Use interpretation \([ \cdot ]\) over \(\mathbb{N}\) with

\[ [\text{quot}^#](x_1, x_2) = x_1 \]
\[ [\text{minus}^#](x_1, x_2) = x_1 \]
\[ [0] = 0 \]
\[ [\text{quot}](x_1, x_2) = x_1 + x_2 \]
\[ [\text{minus}](x_1, x_2) = x_1 \]
\[ [s](x_1) = x_1 + 1 \]

\(\succ\) order solves all constraints
Example (Constraints for Division)

\[ R = \begin{cases} 
\text{minus}(x, 0) & x \\
\text{minus}(s(x), s(y)) & \text{minus}(x, y) \\
\text{quot}(0, s(y)) & 0 \\
\text{quot}(s(x), s(y)) & s(\text{quot}(\text{minus}(x, y), s(y))) 
\end{cases} \]

\[ DP = \begin{cases} 
\text{minus}^#(x_1, x_2) = x_1 \\
\text{minus}(x_1, x_2) = x_1 + x_2 \\
0 = 0 \\
\text{quot}(x_1, x_2) = x_1 + 1 \\
\text{quot}^#(x_1, x_2) = x_1 \\
\text{s}(x_1) = x_1 + 1 
\end{cases} \]

\( \bowtie \) order solves all constraints

\( \bowtie \) \( DP = \emptyset \)

\( \bowtie \) termination of division algorithm proved \( \square \)
Remark

Polynomial interpretations play several roles for program analysis:

Use interpretation $\cdot$ over $\mathbb{N}$ with

$$
\begin{align*}
[\text{quot}^\#](x_1, x_2) &= x_1 \\
[\text{minus}^\#](x_1, x_2) &= x_1 \\
[0] &= 0
\end{align*}
$$

$$
\begin{align*}
[\text{quot}](x_1, x_2) &= x_1 + x_2 \\
[\text{minus}](x_1, x_2) &= x_1 \\
[s](x_1) &= x_1 + 1
\end{align*}
$$

$\bowtie$ order solves all constraints

$\bowtie$ $\mathcal{DP} = \emptyset$

$\bowtie$ termination of division algorithm proved

□

Remark

Polynomial interpretations play several roles for program analysis:

- Ranking function: $[\text{quot}^\#]$ and $[\text{minus}^\#]$

Use interpretation $[\cdot]$ over $\mathbb{N}$ with

$$
[\text{quot}^\#](x_1, x_2) = x_1 \\
[\text{minus}^\#](x_1, x_2) = x_1 \\
[0] = 0
$$

- $[\text{quot}](x_1, x_2) = x_1 + x_2$
- $[\text{minus}](x_1, x_2) = x_1$
- $[s](x_1) = x_1 + 1$

- order solves all constraints
- $\mathcal{DP} = \emptyset$
- termination of division algorithm proved
Remark

Polynomial interpretations play several roles for program analysis:

- Ranking function: \([\text{quot}^\#]\) and \([\text{minus}^\#]\)
- Summary: \([\text{quot}]\) and \([\text{minus}]\)

Use interpretation \([\cdot]\) over \(\mathbb{N}\) with

\[
\begin{align*}
[\text{quot}^\#](x_1, x_2) &= x_1 \\
[\text{minus}^\#](x_1, x_2) &= x_1 \\
[0] &= 0 \\
\end{align*}
\]

\[
\begin{align*}
[\text{quot}](x_1, x_2) &= x_1 + x_2 \\
[\text{minus}](x_1, x_2) &= x_1 \\
[s](x_1) &= x_1 + 1 \\
\end{align*}
\]

\(\bowtie\) order solves all constraints

\(\bowtie\) \(D \mathcal{P} = \emptyset\)

\(\bowtie\) termination of division algorithm proved
Remark

Polynomial interpretations play several roles for program analysis:

- Ranking function: \([\text{quot}^\#]\) and \([\text{minus}^\#]\)
- Summary: \([\text{quot}]\) and \([\text{minus}]\)
- Abstraction (aka norm) for data structures: \([0]\) and \([s]\)

Use interpretation \([\cdot]\) over \(\mathbb{N}\) with

\[
\begin{align*}
[\text{quot}^\#](x_1, x_2) &= x_1 & [\text{quot}](x_1, x_2) &= x_1 + x_2 \\
[\text{minus}^\#](x_1, x_2) &= x_1 & [\text{minus}](x_1, x_2) &= x_1 \\
[0] &= 0 & [s](x_1) &= x_1 + 1
\end{align*}
\]

\(\bowtie\) order solves all constraints
\(\bowtie\) \(\mathcal{DP} = \emptyset\)
\(\bowtie\) termination of division algorithm proved
Task: Solve \( \text{minus}(s(x), s(y)) \preceq \text{minus}(x, y) \)
Task: Solve $\text{minus}(s(x), s(y)) \preceq \text{minus}(x, y)$

1. Fix template polynomials with **parametric coefficients**, get interpretation template:

   $[\text{minus}](x, y) = a_m + b_m x + c_m y, \quad [s](x) = a_s + b_s x$
Task: Solve $\text{minus}(s(x), s(y)) \preceq \text{minus}(x, y)$

1. Fix template polynomials with \textbf{parametric coefficients},
   get interpretation template:
   \[
   \text{[minus]}(x, y) = a_m + b_m x + c_m y, \quad [s](x) = a_s + b_s x
   \]

2. From term constraint to polynomial constraint:
   \[
   s \preceq t \iff [s] \geq [t]
   \]
   Here: $\forall x, y. \ (a_s b_m + a_s c_m) + (b_s b_m - b_m) x + (b_s c_m - c_m) y \geq 0$
Task: Solve \( \text{minus}(s(x), s(y)) \preceq \text{minus}(x, y) \)

1. Fix template polynomials with \textbf{parametric coefficients},
get interpretation template:
\[
[\text{minus}](x, y) = a_m + b_m x + c_m y, \quad [s](x) = a_s + b_s x
\]

2. From term constraint to polynomial constraint:
\[
s \preceq t \iff [s] \geq [t]
\]
Here: \( \forall x, y. \ (a_s b_m + a_s c_m) + (b_s b_m - b_m) x + (b_s c_m - c_m) y \geq 0 \)

3. Eliminate \( \forall x, y \) \textbf{by absolute positiveness criterion}
[Hong, Jakuš, \textit{JAR '98}]:
Here: \( a_s b_m + a_s c_m \geq 0 \land b_s b_m - b_m \geq 0 \land b_s c_m - c_m \geq 0 \)
Task: Solve \( \text{minus}(s(x), s(y)) \preceq \text{minus}(x, y) \)

1. Fix template polynomials with **parametric coefficients**, get interpretation template:
   \[
   [\text{minus}](x, y) = a_m + b_m x + c_m y, \quad [s](x) = a_s + b_s x
   \]

2. From term constraint to polynomial constraint:
   \[
   s \preceq t \iff [s] \geq [t]
   \]
   Here: \( \forall x, y. \quad (a_s b_m + a_s c_m) + (b_s b_m - b_m) x + (b_s c_m - c_m) y \geq 0 \)

3. Eliminate \( \forall x, y \) by **absolute positiveness criterion** [Hong, Jakuš, JAR '98]:
   Here: \( a_s b_m + a_s c_m \geq 0 \land b_s b_m - b_m \geq 0 \land b_s c_m - c_m \geq 0 \)
Automation

Task: Solve \( \text{minus}(s(x), s(y)) \preceq \text{minus}(x, y) \)

1. Fix template polynomials with parametric coefficients, get interpretation template:
\[
[\text{minus}](x, y) = a_m + b_m x + c_m y, \quad [s](x) = a_s + b_s x
\]

2. From term constraint to polynomial constraint:
\[
s \preceq t \rightsquigarrow [s] \geq [t]
\]
Here:
\[
\forall x, y. \ (a_s b_m + a_s c_m) + (b_s b_m - b_m) x + (b_s c_m - c_m) y \geq 0
\]

3. Eliminate \( \forall x, y \) by absolute positiveness criterion
[Hong, Jakuš, JAR '98]:
Here:
\[
a_s b_m + a_s c_m \geq 0 \land b_s b_m - b_m \geq 0 \land b_s c_m - c_m \geq 0
\]
Task: Solve \( \text{minus}(s(x), s(y)) \preceq \text{minus}(x, y) \)

1. Fix template polynomials with parametric coefficients, get interpretation template:

\[
[\text{minus}](x, y) = a_m + b_m x + c_m y, \quad [s](x) = a_s + b_s x
\]

2. From term constraint to polynomial constraint:

\[
s \succeq t \iff [s] \succeq [t]
\]

Here:

\[
\forall x, y. \ (a_s b_m + a_s c_m) + (b_s b_m - b_m) x + (b_s c_m - c_m) y \geq 0
\]

3. Eliminate \( \forall x, y \) by absolute positiveness criterion [Hong, Jakuš, JAR '98]:

Here:

\[
a_s b_m + a_s c_m \geq 0 \land b_s b_m - b_m \geq 0 \land b_s c_m - c_m \geq 0
\]

Non-linear constraints, even for linear interpretations
Task: Solve \( \text{minus}(s(x), s(y)) \preceq \text{minus}(x, y) \)

1. Fix template polynomials with parametric coefficients, get interpretation template:
\[
[\text{minus}](x, y) = a_m + b_m x + c_m y, \quad [s](x) = a_s + b_s x
\]

2. From term constraint to polynomial constraint:
\[
s \preceq t \iff [s] \geq [t]
\]
Here:
\[
\forall x, y. \ (a_s b_m + a_s c_m) + (b_s b_m - b_m) x + (b_s c_m - c_m) y \geq 0
\]

3. Eliminate \( \forall x, y \) by absolute positiveness criterion [Hong, Jakuš, JAR ’98]:
Here:
\[
a_s b_m + a_s c_m \geq 0 \land b_s b_m - b_m \geq 0 \land b_s c_m - c_m \geq 0
\]

Non-linear constraints, even for linear interpretations

Task: Show satisfiability of non-linear constraints over \( \mathbb{N} \) (\( \rightarrow \) SMT solver!)

\( \bowtie \) Prove termination of given term rewrite system
Polynomials with negative coefficients and max-operator
[Hirokawa, Middeldorp, IC '07; Fuhs et al, SAT '07, RTA '08]
- can model behaviour of functions more closely:
  \[ [\text{pred}](x_1) = \max(x_1 - 1, 0) \]
- automation via encoding to non-linear constraints, more complex Boolean structure
Extensions of Polynomial Interpretations

- Polynomials with **negative coefficients** and **max-operator**
  [Hirokawa, Middeldorp, *IC ’07*; Fuhs et al, *SAT ’07, RTA ’08*]
  - can model behaviour of functions more closely:
    \[ \text{pred}(x_1) = \max(x_1 - 1, 0) \]
  - automation via encoding to non-linear constraints, more complex Boolean structure

- Polynomials over \( \mathbb{Q}^+ \) and \( \mathbb{R}^+ \) [Lucas, *RAIRO ’05*]
  - non-integer coefficients increase proving power
Extensions of Polynomial Interpretations

- Polynomials with **negative coefficients** and **max-operator** [Hirokawa, Middeldorp, *IC ’07*; Fuhs et al, *SAT ’07, RTA ’08*]
  - can model behaviour of functions more closely:
    \[\text{pred}(x_1) = \max(x_1 - 1, 0)\]
  - automation via encoding to non-linear constraints, more complex Boolean structure

- Polynomials over \(\mathbb{Q}^+\) and \(\mathbb{R}^+\) [Lucas, *RAIRO ’05*]
  - non-integer coefficients increase proving power

- **Matrix interpretations** [Endrullis, Waldmann, Zantema, *JAR ’08*]
  - linear interpretation to vectors over \(\mathbb{N}^k\), coefficients are matrices
  - useful for deeply nested terms
  - automation: constraints with more complex atoms
  - generalisation to tuple interpretations [Yamada, *JAR ’22*]
Extensions of Polynomial Interpretations

- Polynomials with **negative coefficients** and **max-operator**
  [Hirokawa, Middeldorp, *IC ’07*; Fuhs et al, *SAT ’07, RTA ’08*]
  - can model behaviour of functions more closely:
    \[[\text{pred}](x_1) = \max(x_1 - 1, 0)\]
  - automation via encoding to non-linear constraints, more complex Boolean structure

- Polynomials over \(\mathbb{Q}^+\) and \(\mathbb{R}^+\) [Lucas, *RAIRO ’05*]
  - non-integer coefficients increase proving power

- **Matrix interpretations** [Endrullis, Waldmann, Zantema, *JAR ’08*]
  - linear interpretation to vectors over \(\mathbb{N}^k\), coefficients are matrices
  - useful for deeply nested terms
  - automation: constraints with more complex atoms
  - several flavours: plus-times-semiring, max-plus-semiring [Koprowski, Waldmann, *Acta Cyb. ’09*], ...
  - generalisation to tuple interpretations [Yamada, *JAR ’22*]

- ...
Path orders: based on precedences on function symbols

- Knuth-Bendix Order [Knuth, Bendix, CPAA ’70]
  → polynomial time algorithm [Korovin, Voronkov, IC ’03]
  → SMT encoding [Zankl, Hirokawa, Middeldorp, JAR ’09]

- Lexicographic Path Orders [Kamin, Lévy, Unpublished Manuscript ’80]
  → SAT encoding [Codish et al, JAR ’11]

- Recursive Path Orders [Dershowitz, Manna, CACM ’79; Dershowitz, TCS ’82]
  → SMT encoding [Yamada, Kusakari, Sakabe, SCP ’15]
(SAT and) SMT Solving for Path Orders

Path orders: based on precedences on function symbols

- **Knuth-Bendix Order** [Knuth, Bendix, *CPAA ’70*]
  → polynomial time algorithm [Korovin, Voronkov, *IC ’03*]
  → SMT encoding [Zankl, Hirokawa, Middeldorp, *JAR ’09*]

- **Lexicographic Path Orders** [Kamin, Lévy, *Unpublished Manuscript ’80*]
  and **Recursive Path Orders** [Dershowitz, Manna, *CACM ’79;* Dershowitz, *TCS ’82*]
  → SAT encoding [Codish et al, *JAR ’11*]
Path orders: based on precedences on function symbols

- Knuth-Bendix Order [Knuth, Bendix, CPAA ’70] → polynomial time algorithm [Korovin, Voronkov, IC ’03] → SMT encoding [Zankl, Hirokawa, Middeldorp, JAR ’09]

- Lexicographic Path Orders [Kamin, Lévy, Unpublished Manuscript ’80] and Recursive Path Orders [Dershowitz, Manna, CACM ’79; Dershowitz, TCS ’82] → SAT encoding [Codish et al, JAR ’11]

- Weighted Path Order [Yamada, Kusakari, Sakabe, SCP ’15] → SMT encoding
Further Techniques and Settings for TRSs

- Proving non-termination (an infinite run is possible)

Specific rewrite strategies: innermost, outermost, context-sensitive rewriting [Lucas, *ACM Comput. Surv. '20*], ...

Higher-order rewriting: functional variables, higher types, β-reduction
  \[
  \text{map} (F, \text{Cons}(x, xs)) \rightarrow \text{Cons}(F(x), \text{map}(F, xs))
  \]
  [Kop, PhD thesis '12]

Probabilistic term rewriting: Positive/Strong Almost Sure Termination [Avanzini, Dal Lago, Yamada, *SCP '20*]

Complexity analysis
  [Hirokawa, Moser, *IJCAR '08*; Noschinski, Emmes, Giesl, *JAR '13*; ...]

Can re-use termination machinery to infer and prove statements like "runtime complexity of this TRS is in $O(n^3)$" – more in Session 2!
Further Techniques and Settings for TRSs

- Proving **non-termination** (an infinite run is possible)
  

- Specific **rewrite strategies**: innermost, outermost, context-sensitive rewriting [Lucas, *ACM Comput. Surv. ’20*], ...
Further Techniques and Settings for TRSs

- Proving non-termination (an infinite run is possible)

- Specific rewrite strategies: innermost, outermost, context-sensitive rewriting [Lucas, *ACM Comput. Surv. ’20*], ...

- Higher-order rewriting: functional variables, higher types, \(\beta\)-reduction
  \[
  \text{map}(F, \text{Cons}(x, xs)) \rightarrow \text{Cons}(F(x), \text{map}(F, xs))
  \]
  [Kop, *PhD thesis ’12*]
Further Techniques and Settings for TRSs

- Proving non-termination (an infinite run is possible) [Giesl, Thiemann, Schneider-Kamp, *FroCoS ’05*; Payet, *TCS ’08*; Zankl et al, *SOFSEM ’10*; Emmes, Enger, Giesl, *IJCAR ’12*; ...]

- Specific rewrite strategies: innermost, outermost, context-sensitive rewriting [Lucas, *ACM Comput. Surv. ’20*], ... 

- Higher-order rewriting: functional variables, higher types, β-reduction

  \[ \text{map}(F, \text{Cons}(x, xs)) \rightarrow \text{Cons}(F(x), \text{map}(F, xs)) \]

  [Kop, *PhD thesis ’12*]

Further Techniques and Settings for TRSs

- Proving non-termination (an infinite run is possible) [Giesl, Thiemann, Schneider-Kamp, *FroCoS ’05*; Payet, *TCS ’08*; Zankl et al, *SOFSEM ’10*; Emmes, Enger, Giesl, *IJCAR ’12*; ...]

- Specific rewrite strategies: innermost, outermost, context-sensitive rewriting [Lucas, *ACM Comput. Surv. ’20*], ...

- Higher-order rewriting: functional variables, higher types, \( \beta \)-reduction
  \[
  \text{map}(F, \text{Cons}(x, xs)) \rightarrow \text{Cons}(F(x), \text{map}(F, xs))
  \]
  [Kop, *PhD thesis ’12*]


- Complexity analysis
  [Hirokawa, Moser, *IJCAR ’08*; Noschinski, Emmes, Giesl, *JAR ’13*; ...]
  Can re-use termination machinery to infer and prove statements like “runtime complexity of this TRS is in \( O(n^3) \)”
  \[ \rightarrow \text{more in Session 2!} \]
SMT Solvers *from* Termination Analysis

Annual SMT-COMP, division QF\_NIA (Quantifier-Free Non-linear Integer Arithmetic)

<table>
<thead>
<tr>
<th>Year</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>Barcelogic-QF_NIA</td>
</tr>
<tr>
<td>2010</td>
<td>MiniSmt</td>
</tr>
<tr>
<td>2011</td>
<td>AProVE</td>
</tr>
<tr>
<td>2012</td>
<td><em>no</em> QF_NIA</td>
</tr>
<tr>
<td>2013</td>
<td><em>no</em> SMT-COMP</td>
</tr>
<tr>
<td>2014</td>
<td>AProVE</td>
</tr>
<tr>
<td>2015</td>
<td>AProVE</td>
</tr>
<tr>
<td>2016</td>
<td>Yices</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Annual SMT-COMP, division QF_NIA (Quantifier-Free Non-linear Integer Arithmetic)

<table>
<thead>
<tr>
<th>Year</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>Barcelogic-QF_NIA</td>
</tr>
<tr>
<td>2010</td>
<td>MiniSmt (spin-off of TTT2)</td>
</tr>
<tr>
<td>2011</td>
<td>AProVE</td>
</tr>
<tr>
<td>2012</td>
<td>no QF_NIA</td>
</tr>
<tr>
<td>2013</td>
<td>no SMT-COMP</td>
</tr>
<tr>
<td>2014</td>
<td>AProVE</td>
</tr>
<tr>
<td>2015</td>
<td>AProVE</td>
</tr>
<tr>
<td>2016</td>
<td>Yices</td>
</tr>
</tbody>
</table>

⇒ *Termination provers* can also be successful SMT solvers!
Annual SMT-COMP, division QF_NIA (Quantifier-Free Non-linear Integer Arithmetic)

<table>
<thead>
<tr>
<th>Year</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>Barcelogic-QF_NIA</td>
</tr>
<tr>
<td>2010</td>
<td>MiniSmt (spin-off of TTT2)</td>
</tr>
<tr>
<td>2011</td>
<td>AProVE</td>
</tr>
<tr>
<td>2012</td>
<td>no QF_NIA</td>
</tr>
<tr>
<td>2013</td>
<td>no SMT-COMP</td>
</tr>
<tr>
<td>2014</td>
<td>AProVE</td>
</tr>
<tr>
<td>2015</td>
<td>AProVE</td>
</tr>
<tr>
<td>2016</td>
<td>Yices</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

⇒ Termination provers can also be successful SMT solvers!

(disclaimer: Z3 participated only hors concours)
termCOMP 2022 participants:

- AProVE (RWTH Aachen, Birkbeck U London, U Innsbruck, ...)
- iRankFinder (UC Madrid)
- LoAT (RWTH Aachen)
- Matchbox (HTWK Leipzig)
- Mu-Term (UP Valencia)
- MultumNonMulta (BA Saarland)
- NaTT (AIST Tokyo)
- NTI+cTI (U Réunion)
- SOL (Gunma U)
- TcT (U Innsbruck, INRIA Sophia Antipolis)
- TTT2 (U Innsbruck)
- Ultimate Automizer (U Freiburg)
- Wanda (RU Nijmegen)
Benchmark set: Termination Problem DataBase (TPDB)
https://termination-portal.org/wiki/TPDB
→ 1000s of termination and complexity problems
Benchmark set: Termination Problem DataBase (TPDB)
https://termination-portal.org/wiki/TPDB
→ 1000s of termination and complexity problems

Timeout: 300 seconds
Benchmark set: Termination Problem DataBase (TPDB)
https://termination-portal.org/wiki/TPDB
→ 1000s of termination and complexity problems
Timeout: 300 seconds
Run on StarExec platform [Stump, Sutcliffe, Tinelli, IJCAR ’14]
Benchmark set: Termination Problem DataBase (TPDB)
https://termination-portal.org/wiki/TPDB
→ 1000s of termination and complexity problems

Timeout: 300 seconds

Run on StarExec platform [Stump, Sutcliffe, Tinelli, IJCAR ’14]

Categories for proving (non-)termination and for inferring upper/lower complexity bounds for different programming languages
Benchmark set: Termination Problem DataBase (TPDB)
https://termination-portal.org/wiki/TPDB
→ 1000s of termination and complexity problems

Timeout: 300 seconds

Run on StarExec platform [Stump, Sutcliffe, Tinelli, IJCAR ’14]

Categories for proving (non-)termination and for inferring upper/lower complexity bounds for different programming languages

Part of the Olympic Games at the Federated Logic Conference
Web interfaces available:

- **AProVE**: [https://aprove.informatik.rwth-aachen.de/interface](https://aprove.informatik.rwth-aachen.de/interface)
- **TTT2**: [http://colo6-c703.uibk.ac.at/ ttt2/web/](http://colo6-c703.uibk.ac.at/ttt2/web/)
Web interfaces available:

- AProVE: https://aprove.informatik.rwth-aachen.de/interface
- iRankFinder: http://irankfinder.loopkiller.com:8081/
- Mu-Term:
- TTT2: http://colo6-c703.uibk.ac.at/TTT2/web/

Input format for termination of TRSs:

```
(VAR x y)
(RULES
  plus(0, y) -> y
  plus(s(x), y) -> s(plus(x, y))
)
```
I.2 Termination Analysis of Programs on Integers
Papers on termination of imperative programs often about **integers** as data
Papers on termination of imperative programs often about **integers** as data

**Example (Imperative Program)**

\[
\begin{align*}
\text{if } (x \geq 0) \\
\text{while } (x \neq 0) \\
& x = x - 1;
\end{align*}
\]

Does this program terminate? (\(x\) ranges over \(\mathbb{Z}\))

\[\Rightarrow\text{Restrict initial states to } \ell_0(z) \text{ for } z \in \mathbb{Z}\]

\[\Rightarrow \text{Find invariant } x \geq 0 \text{ at } \ell_1, \ell_2 \text{ (exercise)}\]
Papers on termination of imperative programs often about **integers** as data

**Example (Imperative Program)**

\[ \ell_0: \text{ if } (x \geq 0) \]
\[ \ell_1: \text{ while } (x \neq 0) \]
\[ \ell_2: \quad x = x - 1; \]

Does this program terminate? (\(x\) ranges over \(\mathbb{Z}\))

**Example (Equivalent Translation to an Integer Transition System, cf. [McCarthy, CACM '60])**

\[ \ell_0(x) \rightarrow \ell_1(x) \quad [x \geq 0] \]
\[ \ell_0(x) \rightarrow \ell_3(x) \quad [x < 0] \]
\[ \ell_1(x) \rightarrow \ell_2(x) \quad [x \neq 0] \]
\[ \ell_2(x) \rightarrow \ell_1(x - 1) \]
\[ \ell_1(x) \rightarrow \ell_3(x) \quad [x = 0] \]
Papers on termination of imperative programs often about integers as data.

Example (Imperative Program)

\[\ell_0: \text{if } (x \geq 0)\]
\[\ell_1: \text{while } (x \neq 0)\]
\[\ell_2: \quad x = x - 1;\]

Does this program terminate?

(x ranges over \(\mathbb{Z}\))

Example (Equivalent Translation to an Integer Transition System, cf. [McCarthy, CACM '60])

\[
\begin{align*}
\ell_0(x) & \rightarrow \ell_1(x) \quad [x \geq 0] \\
\ell_0(x) & \rightarrow \ell_3(x) \quad [x < 0] \\
\ell_1(x) & \rightarrow \ell_2(x) \quad [x \neq 0] \\
\ell_2(x) & \rightarrow \ell_1(x - 1) \\
\ell_1(x) & \rightarrow \ell_3(x) \quad [x = 0]
\end{align*}
\]

Oh no! \(\ell_1(-1) \rightarrow \ell_2(-1) \rightarrow \ell_1(-2) \rightarrow \ell_2(-2) \rightarrow \ell_1(-3) \rightarrow \cdots\)
Papers on termination of imperative programs often about integers as data

Example (Imperative Program)

\[ \ell_0: \quad \text{if } (x \geq 0) \]
\[ \ell_1: \quad \text{while } (x \neq 0) \]
\[ \ell_2: \quad x = x - 1; \]

Does this program terminate?
(x ranges over \( \mathbb{Z} \))

Example (Equivalent Translation to an Integer Transition System, cf. [McCarthy, CACM '60])

\[ \ell_0(x) \rightarrow \ell_1(x) \ [x \geq 0] \]
\[ \ell_0(x) \rightarrow \ell_3(x) \ [x < 0] \]
\[ \ell_1(x) \rightarrow \ell_2(x) \ [x \neq 0] \]
\[ \ell_2(x) \rightarrow \ell_1(x - 1) \]
\[ \ell_1(x) \rightarrow \ell_3(x) \ [x = 0] \]

Oh no!
\[ \ell_1(-1) \rightarrow \ell_2(-1) \rightarrow \ell_1(-2) \rightarrow \ell_2(-2) \rightarrow \ell_1(-3) \rightarrow \cdots \]

\[ \Rightarrow \text{Restrict initial states to } \ell_0(z) \text{ for } z \in \mathbb{Z} \]
Papers on termination of imperative programs often about **integers** as data.

**Example (Imprecise Program)**

\[
\begin{align*}
\ell_0 &: \quad \textbf{if} \ (x \geq 0) \\
\ell_1 &: \quad \textbf{while} \ (x \neq 0) \\
\ell_2 &: \quad x = x - 1;
\end{align*}
\]

Does this program terminate? (\(x\) ranges over \(\mathbb{Z}\))

**Example (Equivalent Translation to an Integer Transition System, cf. [McCarthy, CACM ’60])**

\[
\begin{align*}
\ell_0(x) &\rightarrow \ell_1(x) \quad [x \geq 0] \\
\ell_0(x) &\rightarrow \ell_3(x) \quad [x < 0] \\
\ell_1(x) &\rightarrow \ell_2(x) \quad [x \neq 0] \\
\ell_2(x) &\rightarrow \ell_1(x - 1) \\
\ell_1(x) &\rightarrow \ell_3(x) \quad [x = 0]
\end{align*}
\]

**Obsidian**

\[
\ell_1(-1) \rightarrow \ell_2(-1) \rightarrow \ell_1(-2) \rightarrow \ell_2(-2) \rightarrow \ell_1(-3) \rightarrow \cdots
\]

⇒ **Restrict initial states** to \(\ell_0(z)\) for \(z \in \mathbb{Z}\)

⇒ **Find invariant** \(x \geq 0\) at \(\ell_1, \ell_2\) (exercise)
Papers on termination of imperative programs often about **integers** as data.

Example (Imperative Program)

\[ \ell_0: \text{if } (x \geq 0) \]
\[ \ell_1: \text{while } (x \neq 0) \]
\[ \ell_2: \quad x = x - 1; \]

Does this program terminate?  
\((x \text{ ranges over } \mathbb{Z})\)

Example (Equivalent Translation to an Integer Transition System, cf. [McCarthy, CACM '60])

\[
\begin{align*}
\ell_0(x) & \rightarrow \ell_1(x) \quad [x \geq 0] \\
\ell_0(x) & \rightarrow \ell_3(x) \quad [x < 0] \\
\ell_1(x) & \rightarrow \ell_2(x) \quad [x \neq 0 \land x \geq 0] \\
\ell_2(x) & \rightarrow \ell_1(x - 1) \quad [x \geq 0] \\
\ell_1(x) & \rightarrow \ell_3(x) \quad [x = 0 \land x \geq 0]
\end{align*}
\]

Oh no!  
\[
\ell_1(-1) \rightarrow \ell_2(-1) \rightarrow \ell_1(-2) \rightarrow \ell_2(-2) \rightarrow \ell_1(-3) \rightarrow \cdots
\]

\(\Rightarrow\) **Restrict initial states to** \(\ell_0(z)\) for \(z \in \mathbb{Z}\)

\(\Rightarrow\) **Find invariant** \(x \geq 0\) at \(\ell_1, \ell_2\) (exercise)
Proving Termination with Invariants

Example (Transition system with invariants)

\[
\begin{align*}
\ell_0(x) & \rightarrow \ell_1(x) \quad [x \geq 0] \\
\ell_1(x) & \rightarrow \ell_2(x) \quad [x \neq 0 \land x \geq 0] \\
\ell_2(x) & \rightarrow \ell_1(x - 1) \quad [x \geq 0] \\
\ell_1(x) & \rightarrow \ell_3(x) \quad [x = 0 \land x \geq 0]
\end{align*}
\]

Prove termination by ranking function \([\cdot]\) with \([\ell_0](x) = [\ell_1](x) = \cdots = x\)
Example (Transition system with invariants)

\[ \ell_0(x) \succ \ell_1(x) \quad [x \geq 0] \]

\[ \ell_1(x) \succ \ell_2(x) \quad [x \neq 0 \land x \geq 0] \]

\[ \ell_2(x) \succ \ell_1(x - 1) \quad [x \geq 0] \]

\[ \ell_1(x) \succ \ell_3(x) \quad [x = 0 \land x \geq 0] \]

Prove termination by ranking function [·] with \([\ell_0](x) = [\ell_1](x) = \cdots = x\)
Example (Transition system with invariants)

\[ \ell_0(x) \succneq \ell_1(x) \quad [x \geq 0] \]
\[ \ell_1(x) \succneq \ell_2(x) \quad [x \neq 0 \land x \geq 0] \]
\[ \ell_2(x) \succ \ell_1(x - 1) \quad [x \geq 0] \]
\[ \ell_1(x) \succneq \ell_3(x) \quad [x = 0 \land x \geq 0] \]

Prove termination by ranking function \([ \cdot ]\) with \([\ell_0](x) = [\ell_1](x) = \cdots = x\)

Automate search using parametric ranking function:

\[ [\ell_0](x) = a_0 + b_0 \cdot x, \quad [\ell_1](x) = a_1 + b_1 \cdot x, \quad \ldots \]
Example (Transition system with invariants)

\[
\begin{align*}
\ell_0(x) & \succsim \ell_1(x) \quad [x \geq 0] \\
\ell_1(x) & \succsim \ell_2(x) \quad [x \neq 0 \land x \geq 0] \\
\ell_2(x) & \succsim \ell_1(x - 1) \quad [x \geq 0] \\
\ell_1(x) & \succsim \ell_3(x) \quad [x = 0 \land x \geq 0]
\end{align*}
\]

Prove termination by ranking function \([ \cdot ]\) with \([\ell_0](x) = [\ell_1](x) = \cdots = x\)

Automate search using parametric ranking function:

\[
[\ell_0](x) = a_0 + b_0 \cdot x, \quad [\ell_1](x) = a_1 + b_1 \cdot x, \quad \ldots
\]

Constraints here:

\[
\begin{align*}
x \geq 0 & \implies a_2 + b_2 \cdot x > a_1 + b_1 \cdot (x - 1) \quad \text{“decrease …”} \\
x \geq 0 & \implies a_2 + b_2 \cdot x \geq 0 \quad \text{“… against a bound”}
\end{align*}
\]
Proving Termination with Invariants

Example (Transition system with invariants)

\[
\begin{align*}
\ell_0(x) & \succsim \ell_1(x) \quad [x \geq 0] \\
\ell_1(x) & \succsim \ell_2(x) \quad [x \neq 0 \land x \geq 0] \\
\ell_2(x) & \succsim \ell_1(x - 1) \quad [x \geq 0] \\
\ell_1(x) & \succsim \ell_3(x) \quad [x = 0 \land x \geq 0]
\end{align*}
\]

Prove termination by ranking function \([ \cdot ]\) with \([\ell_0](x) = [\ell_1](x) = \cdots = x\)

Automate search using parametric ranking function:

\[
[\ell_0](x) = a_0 + b_0 \cdot x, \quad [\ell_1](x) = a_1 + b_1 \cdot x, \quad \ldots
\]

Constraints here:

\[
\begin{align*}
x \geq 0 & \implies a_2 + b_2 \cdot x > a_1 + b_1 \cdot (x - 1) \quad \text{“decrease …”} \\
x \geq 0 & \implies a_2 + b_2 \cdot x \geq 0 \quad \text{“… against a bound”}
\end{align*}
\]

Use Farkas’ Lemma to eliminate \(\forall x\), solver for linear constraints gives model for \(a_i, b_i\).
Example (Transition system with invariants)

\[ \ell_0(x) \succcurlyeq \ell_1(x) \quad [x \geq 0] \]
\[ \ell_1(x) \succcurlyeq \ell_2(x) \quad [x \neq 0 \land x \geq 0] \]
\[ \ell_2(x) \succcurlyeq \ell_1(x - 1) \quad [x \geq 0] \]
\[ \ell_1(x) \succcurlyeq \ell_3(x) \quad [x = 0 \land x \geq 0] \]

Prove termination by ranking function \([ \cdot \)] with \([\ell_0](x) = [\ell_1](x) = \cdots = x\)

Automate search using parametric ranking function:

\[ [\ell_0](x) = a_0 + b_0 \cdot x, \quad [\ell_1](x) = a_1 + b_1 \cdot x, \quad \ldots \]

Constraints here:

\[
\begin{align*}
x \geq 0 & \quad \Rightarrow \quad a_2 + b_2 \cdot x > a_1 + b_1 \cdot (x - 1) \quad \text{“decrease …”} \\
x \geq 0 & \quad \Rightarrow \quad a_2 + b_2 \cdot x \geq 0 \quad \text{“… against a bound”}
\end{align*}
\]

Use Farkas’ Lemma to eliminate \(\forall x\), solver for linear constraints gives model for \(a_i, b_i\).

More: [Podelski, Rybalchenko, VMCAI ’04, Alias et al, SAS ’10]
Proving Termination with Invariants

Example (Transition system with invariants)

\[ \ell_0(x) \rightarrow \ell_1(x) \quad [x \geq 0] \]
\[ \ell_1(x) \rightarrow \ell_2(x) \quad [x \neq 0 \land x \geq 0] \]
\[ \ell_1(x) \rightarrow \ell_3(x) \quad [x = 0 \land x \geq 0] \]

Prove termination by ranking function \([ \cdot ]\) with \([\ell_0](x) = [\ell_1](x) = \cdots = x\)

Automate search using parametric ranking function:

\[ [\ell_0](x) = a_0 + b_0 \cdot x, \quad [\ell_1](x) = a_1 + b_1 \cdot x, \quad \ldots \]

Constraints here:

\[ x \geq 0 \quad \Rightarrow \quad a_2 + b_2 \cdot x > a_1 + b_1 \cdot (x - 1) \quad \text{“decrease . . .”} \]
\[ x \geq 0 \quad \Rightarrow \quad a_2 + b_2 \cdot x \geq 0 \quad \text{“. . . against a bound”} \]

Use Farkas’ Lemma to eliminate \(\forall x\), solver for linear constraints gives model for \(a_i, b_i\).

More: [Podelski, Rybalchenko, VMCAI ’04, Alias et al, SAS ’10]
Searching for Invariants Using SMT

Termination prover needs to find invariants for programs on integers

− → abstract interpretation

− → more about this in a few minutes!

By counterexample-based reasoning + safety prover:

Terminator

− → prove termination of single program runs

− → termination argument often generalises

... also cooperating with removal of terminating rules

T2

[Brockschmidt, Cook, Fuhs, CAV '13; Brockschmidt et al, TACAS '16]

Using Max-SMT

[Larraz, Oliveras, Rodríguez-Carbonell, Rubio, FMCAD '13]

Nowadays all SMT-based!
Searching for Invariants Using SMT

Termination prover needs to find invariants for programs on integers

- Statically before the translation
  [Otto et al, *RTA ’10*; Ströder et al, *JAR ’17, …*]
  → abstract interpretation [Cousot, Cousot, *POPL ’77*]
  → more about this in a few minutes!
Searching for Invariants Using SMT

Termination prover needs to find invariants for programs on integers

- Statically before the translation
  → abstract interpretation [Cousot, Cousot, *POPL ’77*
  → more about this in a few minutes!

- By counterexample-based reasoning + safety prover: **Terminator**
  [Cook, Podelski, Rybalchenko, *CAV ’06, PLDI ’06*]
  → prove termination of single program **runs**
  → termination argument often generalises
Searching for Invariants Using SMT

Termination prover needs to find invariants for programs on integers

- Statically before the translation
  [Otto et al, RTA ’10; Ströder et al, JAR ’17, …]
  → abstract interpretation [Cousot, Cousot, POPL ’77]
  → more about this in a few minutes!

- By counterexample-based reasoning + safety prover: Terminator
  [Cook, Podelski, Rybalchenko, CAV ’06, PLDI ’06]
  → prove termination of single program runs
  → termination argument often generalises

- … also cooperating with removal of terminating rules (as for TRSs): T2
  [Brockschmidt, Cook, Fuhs, CAV ’13; Brockschmidt et al, TACAS ’16]
Searching for Invariants Using SMT

Termination prover needs to find invariants for programs on integers

- Statically before the translation
  → abstract interpretation [Cousot, Cousot, *POPL ’77*]
  → more about this in a few minutes!

- By counterexample-based reasoning + safety prover: **Terminator**
  [Cook, Podelski, Rybalchenko, *CAV ’06, PLDI ’06*]
  → prove termination of single program **runs**
  → termination argument often generalises

- … also cooperating with removal of terminating **rules** (as for TRSs):
  **T2** [Brockschmidt, Cook, Fuhs, *CAV ’13*; Brockschmidt et al, *TACAS ’16*]

- Using Max-SMT
  [Larraz, Oliveras, Rodríguez-Carbonell, Rubio, *FMCAD ’13*]
Searching for Invariants Using SMT

Termination prover needs to find invariants for programs on integers

- Statically before the translation
  
  → abstract interpretation [Cousot, Cousot, *POPL ‘77*]
  → more about this in a few minutes!

- By counterexample-based reasoning + safety prover: Terminator
  
  [Cook, Podelski, Rybalchenko, *CAV ‘06, PLDI ‘06*]
  → prove termination of single program runs
  → termination argument often generalises

- … also cooperating with removal of terminating rules (as for TRSs): T2
  
  [Brockschmidt, Cook, Fuhs, *CAV ‘13*; Brockschmidt et al, *TACAS ‘16*]

- Using Max-SMT
  
  [Larraz, Oliveras, Rodríguez-Carbonell, Rubio, *FMCAD ‘13*]

Nowadays all SMT-based!
Extensions

- Proving non-termination (infinite run is possible from initial states)
Extensions

- Proving **non**-termination (infinite run is possible from initial states)

- Complexity bounds
Extensions

- Proving **non**-termination (infinite run is possible **from initial states**) 

- Complexity bounds 

- CTL* model checking for **infinite** state systems based on termination and non-termination provers
  [Cook, Khlaaf, Piterman, *JACM ’17*]
Extensions

- Proving non-termination (infinite run is possible from initial states) [Gupta et al, POPL ’08, Brockschmidt et al, FoVeOOS ’11, Chen et al, TACAS ’14, Larraz et al, CAV ’14, Cook et al, FMCAD ’14, ...]

- Complexity bounds [Alias et al, SAS ’10, Hoffmann, Shao, JFP ’15, Brockschmidt et al, TOPLAS ’16, ...]

- CTL* model checking for infinite state systems based on termination and non-termination provers [Cook, Khlaaf, Piterman, JACM ’17]

- Beyond sequential programs on integers:
  - structs/classes [Berdine et al, CAV ’06; Otto et al, RTA ’10; ...]
  - arrays (pointer arithmetic) [Ströder et al, JAR ’17, ...]
  - multi-threaded programs [Cook et al, PLDI ’07, ...]
  - ...

Why Care about Termination of Term Rewriting?

Termination needed by theorem provers

Translate program $P$ with inductive data structures (trees) to TRS, represent data structures as terms

$\Rightarrow$ Termination of TRS implies termination of $P$

- Logic programming: Prolog
  [van Raamsdonk, *ICLP ’97*; Schneider-Kamp et al, *TOCL ’09*;
  Giesl et al, *PPDP ’12*]

- (Lazy) functional programming: Haskell
  [Giesl et al, *TOPLAS ’11*]

- Object-oriented programming: Java
  [Otto et al, *RTA ’10*]
Beyond Classic TRSs for Program Analysis

So far, so good . . .
but do we *really* want to represent 1000000 as $s(s(s(\ldots)))$?!
So far, so good . . .
but do we really want to represent 1000000 as $s(s(s(...)))$?!
So far, so good . . .
but do we *really* want to represent 1000000 as \(s(s(s(...)))\)?!

**Drawbacks:**
- throws away domain knowledge about built-in data types like integers
Beyond Classic TRSs for Program Analysis

So far, so good . . .
but do we really want to represent 1000000 as \( s(s(s(...))) \)⁉

**Drawbacks:**

- throws away domain knowledge about built-in data types like integers
- need to analyse recursive rules for `minus`, `quot`, . . . over and over
So far, so good . . .
but do we really want to represent 1000000 as $s(s(s(...)))$?!

**Drawbacks:**
- throws away domain knowledge about built-in data types like integers
- need to analyse recursive rules for \texttt{minus}, \texttt{quot}, \ldots over and over
- does not benefit from dedicated constraint solvers (e.g., SMT solvers) for arithmetic operations in programs
So far, so good . . .
but do we really want to represent 1000000 as \( s(s(s(...))) \)?!

**Drawbacks:**
- throws away domain knowledge about built-in data types like integers
- need to analyse recursive rules for `minus`, `quot`, . . . over and over
- does not benefit from dedicated constraint solvers (e.g., SMT solvers) for arithmetic operations in programs

Solution: use **constrained term rewriting**
Constrained Term Rewriting, What’s That?

Term rewriting “with batteries included”
- first-order
- no fixed evaluation strategy
- no fixed order of rules to apply
Constrained Term Rewriting, What’s That?

Term rewriting “with batteries included”

- first-order
- no fixed evaluation strategy
- no fixed order of rules to apply
- typed

...with pre-defined data structures (integers, arrays, bitvectors, ...), usually from SMT-LIB theories

rewrite rules with SMT constraints

⇒ Term rewriting + SMT solving for automated reasoning

General forms available, e.g., Logically Constrained TRSs

[Kop, Nishida, FroCoS '13]

For program termination: use term rewriting with integers

[Falke, Kapur, CADE '09; Fuhs et al, RTA '09; Giesl et al, JAR '17]
Constrained Term Rewriting, What’s That?

Term rewriting “with batteries included”

- first-order
- no fixed evaluation strategy
- no fixed order of rules to apply
- typed
- with pre-defined data structures (integers, arrays, bitvectors, ...), usually from SMT-LIB theories
Constrained Term Rewriting, What’s That?

Term rewriting “with batteries included”

- first-order
- no fixed evaluation strategy
- no fixed order of rules to apply
- typed
- with pre-defined data structures (integers, arrays, bitvectors, ...), usually from SMT-LIB theories
- rewrite rules with SMT constraints

⇒ Term rewriting + SMT solving for automated reasoning

General forms available, e.g., Logically Constrained TRSs

[Kop, Nishida, FroCoS '13]

For program termination: use term rewriting with integers

[Falke, Kapur, CADE '09; Fuhs et al, RTA '09; Giesl et al, JAR '17]
Constrained Term Rewriting, What’s That?

Term rewriting “with batteries included”

- first-order
- no fixed evaluation strategy
- no fixed order of rules to apply
- typed
- with pre-defined data structures (integers, arrays, bitvectors, ...), usually from SMT-LIB theories
- rewrite rules with SMT constraints

⇒ Term rewriting + SMT solving for automated reasoning
Term rewriting “with batteries included”

- first-order
- no fixed evaluation strategy
- no fixed order of rules to apply
- typed
- with pre-defined data structures (integers, arrays, bitvectors, ...), usually from SMT-LIB theories
- rewrite rules with SMT constraints

⇒ Term rewriting + SMT solving for automated reasoning

- General forms available, e.g., Logically Constrained TRSs
  [Kop, Nishida, *FroCoS ’13*]
Constrained Term Rewriting, What’s That?

Term rewriting “with batteries included”
- first-order
- no fixed evaluation strategy
- no fixed order of rules to apply
- typed
- with pre-defined data structures (integers, arrays, bitvectors, ...), usually from SMT-LIB theories
- rewrite rules with SMT constraints

⇒ Term rewriting + SMT solving for automated reasoning

- General forms available, e.g., Logically Constrained TRSs [Kop, Nishida, *FroCoS ‘13]*
- For program termination: use term rewriting with integers [Falke, Kapur, *CADE ’09*; Fuhs et al, *RTA ’09*; Giesl et al, *JAR ’17*]
### Example (Constrained Rewrite System)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell_0(n, r) \rightarrow \ell_1(n, r, \text{Nil}) )</td>
<td></td>
</tr>
<tr>
<td>( \ell_1(n, r, xs) \rightarrow \ell_1(n - 1, r + 1, \text{Cons}(r, xs)) )</td>
<td>([n &gt; 0])</td>
</tr>
<tr>
<td>( \ell_1(n, r, xs) \rightarrow \ell_2(xs) )</td>
<td>([n = 0])</td>
</tr>
</tbody>
</table>
### Example (Constrained Rewrite System)

<table>
<thead>
<tr>
<th>Rewrite Rule</th>
<th>Description</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell_0(n, r) \rightarrow \ell_1(n, r, \text{Nil}) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \ell_1(n, r, xs) \rightarrow \ell_1(n - 1, r + 1, \text{Cons}(r, xs)) )</td>
<td>([n &gt; 0])</td>
<td></td>
</tr>
<tr>
<td>( \ell_1(n, r, xs) \rightarrow \ell_2(xs) )</td>
<td>([n = 0])</td>
<td></td>
</tr>
</tbody>
</table>

Possible rewrite sequence:

\[ \ell_0(2, 7) \]
Example (Constrained Rewrite System)

\[
\begin{align*}
\ell_0(n, r) & \rightarrow \ell_1(n, r, \text{Nil}) \\
\ell_1(n, r, xs) & \rightarrow \ell_1(n - 1, r + 1, \text{Cons}(r, xs)) \quad [n > 0] \\
\ell_1(n, r, xs) & \rightarrow \ell_2(xs) \quad [n = 0]
\end{align*}
\]

Possible rewrite sequence:

\[
\begin{align*}
\ell_0(2, 7) \\
& \rightarrow \ell_1(2, 7, \text{Nil})
\end{align*}
\]
Constrained Rewriting by Example

Example (Constrained Rewrite System)

\[ 
\ell_0(n, r) \rightarrow \ell_1(n, r, \text{Nil}) \\
\ell_1(n, r, xs) \rightarrow \ell_1(n - 1, r + 1, \text{Cons}(r, xs)) \quad [n > 0] \\
\ell_1(n, r, xs) \rightarrow \ell_2(xs) \quad [n = 0] 
\]

Possible rewrite sequence:

\[ 
\ell_0(2, 7) \\
\rightarrow \ell_1(2, 7, \text{Nil}) \\
\rightarrow \ell_1(1, 8, \text{Cons}(7, \text{Nil})) 
\]
### Example (Constrained Rewrite System)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_0(n, r)$ $\rightarrow$ $\ell_1(n, r, \text{Nil})$</td>
<td></td>
</tr>
<tr>
<td>$\ell_1(n, r, xs)$ $\rightarrow$ $\ell_1(n - 1, r + 1, \text{Cons}(r, xs))$</td>
<td>$[n &gt; 0]$</td>
</tr>
<tr>
<td>$\ell_1(n, r, xs)$ $\rightarrow$ $\ell_2(xs)$</td>
<td>$[n = 0]$</td>
</tr>
</tbody>
</table>

Possible rewrite sequence:

\[
\ell_0(2, 7) \\
\rightarrow \ell_1(2, 7, \text{Nil}) \\
\rightarrow \ell_1(1, 8, \text{Cons}(7, \text{Nil})) \\
\rightarrow \ell_1(0, 9, \text{Cons}(8, \text{Cons}(7, \text{Nil})))
\]
Example (Constrained Rewrite System)

\[
\begin{align*}
\ell_0(n, r) & \rightarrow \ell_1(n, r, \text{Nil}) \\
\ell_1(n, r, xs) & \rightarrow \ell_1(n - 1, r + 1, \text{Cons}(r, xs)) \quad [n > 0] \\
\ell_1(n, r, xs) & \rightarrow \ell_2(xs) \quad [n = 0]
\end{align*}
\]

Possible rewrite sequence:

\[
\begin{align*}
\ell_0(2, 7) & \\
& \rightarrow \ell_1(2, 7, \text{Nil}) \\
& \rightarrow \ell_1(1, 8, \text{Cons}(7, \text{Nil})) \\
& \rightarrow \ell_1(0, 9, \text{Cons}(8, \text{Cons}(7, \text{Nil}))) \\
& \rightarrow \ell_2(\text{Cons}(8, \text{Cons}(7, \text{Nil})))
\end{align*}
\]
Constrained Rewriting by Example

**Example (Constrained Rewrite System)**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Rule</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_0(n, r)$</td>
<td>$\rightarrow \ell_1(n, r, \text{Nil})$</td>
<td></td>
</tr>
<tr>
<td>$\ell_1(n, r, xs)$</td>
<td>$\rightarrow \ell_1(n - 1, r + 1, \text{Cons}(r, xs))$</td>
<td>$[n &gt; 0]$</td>
</tr>
<tr>
<td>$\ell_1(n, r, xs)$</td>
<td>$\rightarrow \ell_2(xs)$</td>
<td>$[n = 0]$</td>
</tr>
</tbody>
</table>

Possible rewrite sequence:

\[
\ell_0(2, 7) \rightarrow \ell_1(2, 7, \text{Nil}) \rightarrow \ell_1(1, 8, \text{Cons}(7, \text{Nil})) \rightarrow \ell_1(0, 9, \text{Cons}(8, \text{Cons}(7, \text{Nil}))) \rightarrow \ell_2(\text{Cons}(8, \text{Cons}(7, \text{Nil})))
\]

Here $7, 8, \ldots$ are predefined constants.
Example (Constrained Rewrite System)

\[\ell_0(n, r) \rightarrow \ell_1(n, r, \text{Nil})\]
\[\ell_1(n, r, xs) \rightarrow \ell_1(n - 1, r + 1, \text{Cons}(r, xs)) \quad [n > 0]\]
\[\ell_1(n, r, xs) \rightarrow \ell_2(xs) \quad [n = 0]\]

Possible rewrite sequence:

\[\ell_0(2, 7)\]
\[\rightarrow \ell_1(2, 7, \text{Nil})\]
\[\rightarrow \ell_1(1, 8, \text{Cons}(7, \text{Nil}))\]
\[\rightarrow \ell_1(0, 9, \text{Cons}(8, \text{Cons}(7, \text{Nil})))\]
\[\rightarrow \ell_2(\text{Cons}(8, \text{Cons}(7, \text{Nil})))\]

Here 7, 8, \ldots are predefined constants.

Termination proof: reuse techniques for TRSs and integer programs
Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last ~ 20 years
Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last \( \sim 20 \) years
- Term rewriting: handles **inductive data structures** well
Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last $\sim 20$ years
- Term rewriting: handles **inductive data structures** well
- Imperative programs on integers: need to consider **reachability/safety** and **invariants**
Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last $\sim 20$ years

Term rewriting: handles **inductive data structures** well

Imperative programs on integers: need to consider **reachability/safety** and **invariants**

Since a few years cross-fertilisation
Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last \( \sim 20 \) years
- Term rewriting: handles \textbf{inductive data structures} well
- Imperative programs on integers: need to consider \textbf{reachability/safety} and \textbf{invariants}
- Since a few years cross-fertilisation
- Constrained term rewriting: best of both worlds as back-end language
Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last \(\sim\) 20 years

Term rewriting: handles \textbf{inductive data structures} well

Imperative programs on integers: need to consider \textbf{reachability/safety} and \textbf{invariants}

Since a few years cross-fertilisation

Constrained term rewriting: best of both worlds as back-end language

Proof search heavily relies on SMT solving for automation
Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last \( \sim 20 \) years
- Term rewriting: handles **inductive data structures** well
- Imperative programs on integers: need to consider **reachability/safety** and **invariants**
- Since a few years cross-fertilisation
- Constrained term rewriting: best of both worlds as back-end language
- Proof search heavily relies on SMT solving for automation
- Needs of termination analysis have also led to better SMT solvers
Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last \( \sim \) 20 years
- Term rewriting: handles **inductive data structures** well
- Imperative programs on integers: need to consider **reachability/safety** and **invariants**
- Since a few years cross-fertilisation
- Constrained term rewriting: best of both worlds as back-end language
- Proof search heavily relies on SMT solving for automation
- Needs of termination analysis have also led to better SMT solvers
- More information . . .

http://termination-portal.org
Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last ∼ 20 years
- Term rewriting: handles **inductive data structures** well
- Imperative programs on integers: need to consider **reachability/safety** and **invariants**
- Since a few years cross-fertilisation
- Constrained term rewriting: best of both worlds as back-end language
- Proof search heavily relies on SMT solving for automation
- Needs of termination analysis have also led to better SMT solvers
- More information . . .

  [http://termination-portal.org](http://termination-portal.org)

**Behind (almost) every successful termination prover . . .**
Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last $\sim 20$ years
- Term rewriting: handles **inductive data structures** well
- Imperative programs on integers: need to consider **reachability/safety** and **invariants**
- Since a few years cross-fertilisation
- Constrained term rewriting: best of both worlds as back-end language
- Proof search heavily relies on SMT solving for automation
- Needs of termination analysis have also led to better SMT solvers
- More information ... 

http://termination-portal.org

**Behind (almost) every successful termination prover ...**

**... there is a powerful SAT / SMT solver!**
I.3 Termination Analysis of Java programs
execute program **symbolically** from initial states of the program, handle language peculiarities here (→ Java: sharing, cyclicity analysis)

```plaintext
f: if ...
   ...
else
   ...
   g: while ...
   ...
```
execute program symbolically from initial states of the program, handle language peculiarities here (→ Java: sharing, cyclicity analysis)

```plaintext
f: if ... init(...) ... 
else ... 
   g: while ... ...
```
execute program symbolically from initial states of the program, handle language peculiarities here (→ Java: sharing, cyclicity analysis)

\[
\begin{align*}
\text{f: if} & \quad \ldots \\
\quad & \ldots \\
\quad \text{else} \\
\quad & \ldots \\
\quad \text{g: while} & \quad \ldots \\
\quad & \ldots \\
\end{align*}
\]

init(...) ↓ f(...)

init(...)
execute program **symbolically** from initial states of the program, handle language peculiarities here (→ Java: sharing, cyclicity analysis)

\[
f: \text{if } \ldots \\
\ldots \\
\text{else} \\
\ldots \\
g: \text{while } \ldots \\
\ldots
\]
execute program symbolically from initial states of the program, handle language peculiarities here (Java: sharing, cyclicity analysis)

\[ \text{init}(\ldots) \]
\[ \text{f}(\ldots) \]
\[ \text{else} \]
\[ \text{g: while } \ldots \]
\[ \ldots \]
\[ \text{g}(s) \]
\[ \text{g}(t) \]
execute program **symbolically** from initial states of the program, handle language peculiarities here (→ Java: sharing, cyclicity analysis)

use **generalisation** of program states, get **over-approximation** of all possible program runs (≈ control-flow graph with extra info)

closely related: Abstract Interpretation [Cousot and Cousot, *POPL '77*]

\[
\begin{align*}
f &: \text{if } \ldots \\
& \quad \ldots \\
& \quad \text{else} \\
& \quad \ldots \\
& \quad g &: \text{while } \ldots \\
& \quad \ldots
\end{align*}
\]

```
init(...)  
\downarrow  
f(...)  
\downarrow  
g(s)  
\downarrow  
g(t)  
\longleftarrow \text{instance of } g(s)
```

\[
\begin{align*}
\ldots
\end{align*}
\]
execute program symbolically from initial states of the program, handle language peculiarities here (→ Java: sharing, cyclicity analysis)

use generalisation of program states, get over-approximation of all possible program runs (≈ control-flow graph with extra info)

closely related: Abstract Interpretation [Cousot and Cousot, POPL ’77]

extract TRS from cycles in the representation
execute program **symbolically** from initial states of the program, handle language peculiarities here (→ Java: sharing, cyclicity analysis)

use **generalisation** of program states, get **over-approximation** of all possible program runs (≈ control-flow graph with extra info)

closely related: Abstract Interpretation [Cousot and Cousot, *POPL ’77*]

**extract TRS** from cycles in the representation

if TRS terminates

⇒ any concrete program execution can use cycles only finitely often

⇒ the program **must** terminate

\[
\begin{align*}
f &: \text{if } \ldots \\
& \quad \text{...} \\
& \quad \text{else} \\
& \quad \text{...} \\
& \quad g &: \text{while } \ldots \\
& \quad \text{...} \\
\end{align*}
\]
Recipe for proving program termination by reusing TRS termination provers

1. Decide on suitable symbolic representation of abstract program states
   (abstract domain)
   - What data objects can we represent as terms?
2. Execute program symbolically from its initial states
3. Use generalisation of program states to get closed finite representation
   (symbolic execution graph, abstract interpretation)
4. Extract rewrite rules that "over-approximate" program executions in strongly-connected components of graph
5. Prove termination of these rewrite rules
   ⇒ implies termination of program from initial states
Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states
  (**abstract domain**)
  \[\rightarrow\] here: what data objects can we represent as terms?
Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (abstract domain)
  → here: what data objects can we represent as terms?
- Execute program symbolically from its initial states
Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states
  (*abstract domain*)
  → here: what data objects can we represent as terms?
- Execute program *symbolically* from its initial states
- Use *generalisation* of program states to get closed finite representation (symbolic execution graph, abstract interpretation)
Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (abstract domain)
  → here: what data objects can we represent as terms?
- Execute program symbolically from its initial states
- Use generalisation of program states to get closed finite representation (symbolic execution graph, abstract interpretation)
- Extract rewrite rules that “over-approximate” program executions in strongly-connected components of graph
Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (abstract domain)
  → here: what data objects can we represent as terms?
- Execute program symbolically from its initial states
- Use generalisation of program states to get closed finite representation (symbolic execution graph, abstract interpretation)
- Extract rewrite rules that “over-approximate” program executions in strongly-connected components of graph
- Prove termination of these rewrite rules
  ⇒ implies termination of program from initial states
Java Challenges

Java: object-oriented imperative language
- sharing and aliasing (several references to the same object)
- side effects
- cyclic data objects (e.g., list.next == list)
- object-orientation with inheritance
- ...

...
public class MyInt {

    // only wrap a primitive int
    private int val;

    // count "num" up to the value in "limit"
    public static void count(MyInt num, MyInt limit) {
        if (num == null || limit == null) {
            return;
        }
        // introduce sharing
        MyInt copy = num;
        while (num.val < limit.val) {
            copy.val++;
        }
    }
}

Does **count** terminate for all inputs? Why (not)?
(Assume that **num** and **limit** are not references to the same object.)
Approach to Termination Analysis of Java

Tailor two-stage approach to Java [Otto et al, RTA ’10]
Approach to Termination Analysis of Java

Tailor two-stage approach to Java [Otto et al, RTA ’10]

**Back-end:** From rewrite system to termination proof
- Constrained term rewriting with integers [Giesl et al, JAR ’17]
- Termination techniques for rewriting and for integers can be integrated
Approach to Termination Analysis of Java

Tailor two-stage approach to Java [Otto et al, RTA ’10]

**Back-end:** From rewrite system to termination proof
- Constrained term rewriting with integers [Giesl et al, JAR ’17]
- Termination techniques for rewriting and for integers can be integrated

**Front-end:** From Java to constrained rewrite system
- Build *symbolic execution graph* that over-approximates all runs of Java program (abstract interpretation)
- Symbolic execution graph has *invariants* for integers and heap object shape (trees?)
- Extract rewrite system from symbolic execution graph
Approach to Termination Analysis of Java

Tailor two-stage approach to Java [Otto et al, RTA ’10]

**Back-end:** From rewrite system to termination proof
- Constrained term rewriting with integers [Giesl et al, JAR ’17]
- Termination techniques for rewriting and for integers can be integrated

**Front-end:** From Java to constrained rewrite system
- Build **symbolic execution graph** that over-approximates all runs of Java program (abstract interpretation)
- Symbolic execution graph has **invariants** for integers and heap object shape (trees?)
- Extract rewrite system from symbolic execution graph

Implemented in the tool **AProVE** (→ web interface)

http://aprove.informatik.rwth-aachen.de/
[Otto et al, *RTA ’10*] describe their technique for compiled Java programs: Java Bytecode
[Otto et al, RTA ’10] describe their technique for compiled Java programs: **Java Bytecode**

- desugared machine code for a (virtual) stack machine, still has all the (relevant) information from source code
- input for Java interpreter and for many program analysis tools
- somewhat inconvenient for presentation, though . . .
[Otto et al, RTA ’10] describe their technique for converting compiled Java programs: **Java Bytecode**

- desugared machine code for a (virtual) stack machine, still has all the (relevant) information from source input for Java interpreter and for many program analysis tools
- somewhat inconvenient for presentation, though

```java
00: aload_0
01: ifnull 8
04: aload_1
05: ifnonnull 9
08: return
09: aload_0
10: astore_2
11: aload_0
12: getfield val
15: aload_1
16: getfield val
19: if_icmpge 35
22: aload_2
23: aload_2
24: getfield val
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```
[Otto et al, RTA ’10] describe their technique for compiled Java programs: Java Bytecode

- desugared machine code for a (virtual) stack machine, still has all the (relevant) information from source code
- input for Java interpreter and for many program analysis tools
- somewhat inconvenient for presentation, though . . .

Here: Java source code
Ingredients for the Abstract Domain

1. program counter value (line number)
2. values of variables (treating int as $\mathbb{Z}$)
3. over-approximating info on possible variable values
   - integers: use intervals, e.g. $x \in [4, 7]$ or $y \in [0, \infty)$
   - heap memory with objects, **no sharing** unless stated otherwise
   - MyInt(?): maybe null, maybe a MyInt object

**Heap predicates:**
- Two references may be equal: $o_1 =? o_2$

<table>
<thead>
<tr>
<th>$\emptyset 3$</th>
<th>num: $o_1$, limit: $o_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1$: MyInt()?</td>
<td></td>
</tr>
<tr>
<td>$o_2$: MyInt(val = $i_1$)</td>
<td></td>
</tr>
<tr>
<td>$i_1$: [4, 80]</td>
<td></td>
</tr>
</tbody>
</table>
Ingredients for the Abstract Domain

1. program counter value (line number)
2. values of variables (treating int as $\mathbb{Z}$)
3. over-approximating info on possible variable values
   - integers: use intervals, e.g. $x \in [4, 7]$ or $y \in [0, \infty)$
   - heap memory with objects, no sharing unless stated otherwise
   - MyInt(?): maybe null, maybe a MyInt object

Heap predicates:
- Two references may be equal: $o_1 =? o_2$
- Two references may share: $o_1 \neq / o_2$

<table>
<thead>
<tr>
<th>$o_3$</th>
<th>num: $o_1$, limit: $o_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1$: MyInt(?)</td>
<td></td>
</tr>
<tr>
<td>$o_2$: MyInt(val = $i_1$)</td>
<td></td>
</tr>
<tr>
<td>$i_1$: [4, 80]</td>
<td></td>
</tr>
</tbody>
</table>
Ingredients for the Abstract Domain

1. program counter value (line number)
2. values of variables (treating int as \( \mathbb{Z} \))
3. over-approximating info on possible variable values
   - integers: use intervals, e.g. \( x \in [4, 7] \) or \( y \in [0, \infty) \)
   - heap memory with objects, **no sharing** unless stated otherwise
   - MyInt(?): maybe null, maybe a MyInt object

**Heap predicates:**
- Two references may be equal: \( o_1 =? o_2 \)
- Two references may share: \( o_1 \backslash/ o_2 \)
- Reference may have cycles: \( o_1 ! \)

\[
\begin{array}{c|c}
03 & \text{num: } o_1, \text{limit: } o_2 \\
\hline
o_1: \text{MyInt(?)} \\
o_2: \text{MyInt(val = } i_1) \\
i_1: [4, 80]
\end{array}
\]
Building the Symbolic Execution Graph

```java
public class MyInt {
    private int val;
    static void count(MyInt num, MyInt limit) {
        if (num == null || limit == null)
            return;
        MyInt copy = num;
        while (num.val < limit.val)
            copy.val++;
    }
}
```

A

<table>
<thead>
<tr>
<th>1</th>
<th>num: o₁, limit: o₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁ : MyInt(?)</td>
<td></td>
</tr>
<tr>
<td>o₂ : MyInt(?)</td>
<td></td>
</tr>
</tbody>
</table>
Building the Symbolic Execution Graph

public class MyInt {
    private int val;
    static void count(MyInt num, MyInt limit) {
        if (num == null || limit == null)
            return;
        MyInt copy = num;
        while (num.val < limit.val)
            copy.val++;
    }
}

**cond** means: refine X with **cond**, then evaluate to Y; here combined for brevity (narrowing)
Building the Symbolic Execution Graph

public class MyInt {
    private int val;
    static void count(MyInt num, MyInt limit) {
        1: if (num == null || limit == null) return;
        2: MyInt copy = num;
        3: while (num.val < limit.val) copy.val++;
        4: }
    }

\[
\begin{array}{|c|c|c|}
\hline
1 & num : o_1, limit : o_2 \\
\hline
| o_1 : MyInt(?) \\
| o_2 : MyInt(?) \\
\hline
2 & num : o_1, limit : o_2 \\
\hline
| o_1 : MyInt(val = i_1) \\
| o_2 : MyInt(?) \\
| i_1 : (-\infty, \infty) \\
\hline
3 & num : o_1, limit : o_2 \\
\hline
| o_1 : null \\
| o_2 : MyInt(?) \\
\hline
4 & num : o_1, limit : o_2 \\
\hline
| o_1 : MyInt(val = i_1) \\
| o_2 : MyInt(val = i_2) \\
| i_1 : (-\infty, \infty) \\
| i_2 : (-\infty, \infty) \\
\hline
\end{array}
\]

\[X \xrightarrow{\text{cond}} Y\]

means: refine X with \textit{cond}, then evaluate to Y; here combined for brevity (narrowing)
Building the Symbolic Execution Graph

public class MyInt {
    private int val;
    static void count(MyInt num, MyInt limit) {
        1: if (num == null || limit == null) return;
        2: MyInt copy = num;
        3: while (num.val < limit.val) copy.val++;
        4: }
    }

X → Y
means: evaluate X to Y
public class MyInt {
    private int val;
    static void count(MyInt num, MyInt limit) {
        if (num == null || limit == null) return;
        MyInt copy = num;
        while (num.val < limit.val) copy.val++;
    }
}
public class MyInt {
    private int val;
    static void count(MyInt num, MyInt limit) {
        if (num == null || limit == null) { return; }
        MyInt copy = num;
        while (num.val < limit.val) { copy.val++;
        } }
    }

    public class MyInt {
        private int val;
        static void count(MyInt num, MyInt limit) {
            if (num == null || limit == null) { return; }
            MyInt copy = num;
            while (num.val < limit.val) { copy.val++;
            } }
    }

    MyInt count(MyInt num, MyInt limit) {
        MyInt copy = num;
        while (num.val < limit.val) { copy.val++;
        } return copy;
    }
public class MyInt {
    private int val;
    static void count(MyInt num, MyInt limit) {
        if (num == null || limit == null) return;
        MyInt copy = num;
        while (num.val < limit.val) copy.val++;
    }
}
Symbolic Execution Graphs

- symbolic over-approximation of all computations (abstract interpretation)
- expand nodes until all leaves correspond to program ends
- by suitable generalisation steps (widening), one can always get a finite symbolic execution graph
- state $s_1$ is instance of state $s_2$ if all concrete states described by $s_1$ are also described by $s_2$
Symbolic Execution Graphs

- symbolic over-approximation of all computations (abstract interpretation)
- expand nodes until all leaves correspond to program ends
- by suitable generalisation steps (widening), one can always get a finite symbolic execution graph
- state $s_1$ is instance of state $s_2$
  - if all concrete states described by $s_1$ are also described by $s_2$

Using Symbolic Execution Graphs for Termination Proofs

- every concrete Java computation corresponds to a computation path in the symbolic execution graph
- symbolic execution graph is called terminating iff it has no infinite computation path
Transformation of Objects to Terms (1/2)

For every class \(C\) with \(n\) fields, introduce an \(n\)-ary function symbol \(C\)

- **term** for \(o_1\): \(o_1\)
- **term** for \(o_2\): \(\text{MyInt}(i_2)\)
- **term** for \(o_3\): \(\text{null}\)
- **term** for \(o_4\): \(x\) (new variable)
- **term** for \(i_1\): \(i_1\) with side constraint \(i_1 \geq 7\)
  
  (add invariant \(i_1 \geq 7\) to constrained rewrite rules from state Q)
Dealing with subclasses:

```java
google class A {
    int a;
}
google class B extends A {
    int b;
}
...
A x = new A();
x.a = 1;
B y = new B();
y.a = 2;
y.b = 3;
```
Dealing with subclasses:

- for every class C with $n$ fields, introduce $(n + 1)$-ary function symbol $C$
- first argument: part of the object corresponding to subclasses of C
- **term** for $x$: $A(eoc, 1)$
  $\rightarrow eoc$ for end of class
- **term** for $y$: $A(B(eoc, 3), 2)$

```java
public class A {
    int a;
}

public class B extends A {
    int b;
}
...
A x = new A();
x.a = 1;

B y = new B();
y.a = 2;
y.b = 3;
```
public class A {
    int a;
}

public class B extends A {
    int b;
}

...  
A x = new A();
x.a = 1;

B y = new B();
y.a = 2;
y.b = 3;

Dealing with **subclasses**:  
- for every class C with *n* fields, introduce \((n + 1)\)-ary function symbol \(C\)
- first argument: part of the object corresponding to subclasses of \(C\)
- *term* for \(x\): \(jlO(A(eoc, 1))\)  
  \(\rightarrow\) \(eoc\) for end of class  
- *term* for \(y\): \(jlO(A(B(eoc, 3), 2))\)
- every class extends \(\text{Object!}\)
  \(\rightarrow jlO \equiv \text{java.lang.Object}\)
From the Symbolic Execution Graph to Terms and Rules

<table>
<thead>
<tr>
<th>5</th>
<th>num: o₁,limit: o₂, copy: o₁</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>o₁ : MyInt(val = i₁)</td>
</tr>
<tr>
<td></td>
<td>o₂ : MyInt(val = i₂)</td>
</tr>
<tr>
<td></td>
<td>i₁ : (−∞, ∞)</td>
</tr>
<tr>
<td></td>
<td>i₂ : (−∞, ∞)</td>
</tr>
</tbody>
</table>

\[ i₃ = i₁ + 1 \]

<table>
<thead>
<tr>
<th>6</th>
<th>num: o₁,limit: o₂, copy: o₁</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>o₁ : MyInt(val = i₁)</td>
</tr>
<tr>
<td></td>
<td>o₂ : MyInt(val = i₂)</td>
</tr>
<tr>
<td></td>
<td>i₁ : (−∞, ∞)</td>
</tr>
<tr>
<td></td>
<td>i₂ : (−∞, ∞)</td>
</tr>
</tbody>
</table>

\[ i₃ = i₁ + 1 \]

<table>
<thead>
<tr>
<th>5</th>
<th>num: o₁,limit: o₂, copy: o₁</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>o₁ : MyInt(val = i₁)</td>
</tr>
<tr>
<td></td>
<td>o₂ : MyInt(val = i₂)</td>
</tr>
<tr>
<td></td>
<td>i₁ : (−∞, ∞)</td>
</tr>
<tr>
<td></td>
<td>i₂ : (−∞, ∞)</td>
</tr>
</tbody>
</table>

Termination easy to show (intuitively: \( i₂ - i₁ \) decreases against bound 0)
State F: \[\ell_F(\ j\text{LO}(\text{MyInt}(@eoc, i_1)), \ j\text{LO}(\text{MyInt}(@eoc, i_2))) \]

State H: \[\ell_H(\ j\text{LO}(\text{MyInt}(@eoc, i_1)), \ j\text{LO}(\text{MyInt}(@eoc, i_2))) \]
From the Symbolic Execution Graph to Terms and Rules

State F: \( \ell_F( j\text{LO}(\text{MyInt}(\text{eoc}, i_1)), j\text{LO}(\text{MyInt}(\text{eoc}, i_2)) ) \)

\[ i_1 < i_2 \]

\[ \rightarrow \]

State H: \( \ell_H( j\text{LO}(\text{MyInt}(\text{eoc}, i_1)), j\text{LO}(\text{MyInt}(\text{eoc}, i_2)) ) \)

\[ i_1 < i_2 \]
From the Symbolic Execution Graph to Terms and Rules

<table>
<thead>
<tr>
<th>5</th>
<th>num: o₁, limit: o₂, copy: o₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁: MyInt(val = i₁)</td>
<td></td>
</tr>
<tr>
<td>o₂: MyInt(val = i₂)</td>
<td></td>
</tr>
<tr>
<td>i₁: (−∞, ∞)</td>
<td></td>
</tr>
<tr>
<td>i₂: (−∞, ∞)</td>
<td></td>
</tr>
</tbody>
</table>

\[ i₃ = i₁ + 1 \]

<table>
<thead>
<tr>
<th>6</th>
<th>num: o₁, limit: o₂, copy: o₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁: MyInt(val = i₁)</td>
<td></td>
</tr>
<tr>
<td>o₂: MyInt(val = i₂)</td>
<td></td>
</tr>
<tr>
<td>i₁: (−∞, ∞)</td>
<td></td>
</tr>
<tr>
<td>i₂: (−∞, ∞)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>num: o₁, limit: o₂, copy: o₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁: MyInt(val = i₁)</td>
<td></td>
</tr>
<tr>
<td>o₂: MyInt(val = i₂)</td>
<td></td>
</tr>
<tr>
<td>i₁: (−∞, ∞)</td>
<td></td>
</tr>
<tr>
<td>i₂: (−∞, ∞)</td>
<td></td>
</tr>
</tbody>
</table>

- **State F:** \[ ℓ_F( \ jLO(MyInt(eoc, i₁)), \ jLO(MyInt(eoc, i₂)) ) \]  
  \[ \rightarrow \]  
- **State H:** \[ ℓ_H( \ jLO(MyInt(eoc, i₁)), \ jLO(MyInt(eoc, i₂)) ) \]  
  \[ [i₁ < i₂] \]
- **State H:** \[ ℓ_H( \ jLO(MyInt(eoc, i₁)), \ jLO(MyInt(eoc, i₂)) ) \]  
- **State I:** \[ ℓ_F( \ jLO(MyInt(eoc, i₁ + 1)), \ jLO(MyInt(eoc, i₂)) ) \]
From the Symbolic Execution Graph to Terms and Rules

State F: \( \ell_F\left( j\text{LO}(\text{MyInt}(\text{eoc}, i_1)), j\text{LO}(\text{MyInt}(\text{eoc}, i_2)) \right) \)

\[ \rightarrow \]

State H: \( \ell_H\left( j\text{LO}(\text{MyInt}(\text{eoc}, i_1)), j\text{LO}(\text{MyInt}(\text{eoc}, i_2)) \right) \)

\[ [i_1 < i_2] \]

State H: \( \ell_H\left( j\text{LO}(\text{MyInt}(\text{eoc}, i_1)), j\text{LO}(\text{MyInt}(\text{eoc}, i_2)) \right) \)

\[ \rightarrow \]

State I: \( \ell_F\left( j\text{LO}(\text{MyInt}(\text{eoc}, i_1 + 1)), j\text{LO}(\text{MyInt}(\text{eoc}, i_2)) \right) \)
From the Symbolic Execution Graph to Terms and Rules

\[ i_3 = i_1 + 1 \]

\[
\begin{array}{c|c|c|c|c}
5 & \text{num: } o_1, \text{limit: } o_2, \text{copy: } o_1 \\
\hline
o_1: \text{MyInt}(\text{val} = i_3) \\
o_2: \text{MyInt}(\text{val} = i_2) \\
i_3: (\mathbb{R}, \mathbb{R}) \\
i_2: (\mathbb{R}, \mathbb{R}) \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
6 & \text{num: } o_1, \text{limit: } o_2, \text{copy: } o_1 \\
\hline
o_1: \text{MyInt}(\text{val} = i_1) \\
o_2: \text{MyInt}(\text{val} = i_2) \\
i_1: (\mathbb{R}, \mathbb{R}) \\
i_2: (\mathbb{R}, \mathbb{R}) \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
5 & \text{num: } o_1, \text{limit: } o_2, \text{copy: } o_1 \\
\hline
o_1: \text{MyInt}(\text{val} = i_1) \\
o_2: \text{MyInt}(\text{val} = i_2) \\
i_1: (\mathbb{R}, \mathbb{R}) \\
i_2: (\mathbb{R}, \mathbb{R}) \\
\end{array}
\]

- State F: \( \ell_F(\ j\text{O}(\text{MyInt}(\text{eoc}, i_1)), \ j\text{O}(\text{MyInt}(\text{eoc}, i_2)) ) \)

\[ \rightarrow \]

- State H: \( \ell_H(\ j\text{O}(\text{MyInt}(\text{eoc}, i_1)), \ j\text{O}(\text{MyInt}(\text{eoc}, i_2)) ) \) \( [i_1 < i_2] \)

- State H: \( \ell_H(\ j\text{O}(\text{MyInt}(\text{eoc}, i_1)), \ j\text{O}(\text{MyInt}(\text{eoc}, i_2)) ) \)

\[ \rightarrow \]

- State I: \( \ell_F(\ j\text{O}(\text{MyInt}(\text{eoc}, i_1 + 1)), \ j\text{O}(\text{MyInt}(\text{eoc}, i_2)) ) \)

- Termination easy to show (intuitively: \( i_2 - i_1 \) decreases against bound 0)
modular termination proofs and recursion
[Brockschmidt et al, RTA ’11]
Extensions

- **modular** termination proofs and **recursion**
  [Brockschmidt et al, *RTA* ’11]

- proving **reachability** and **non-termination** (uses only symbolic execution graph)
  [Brockschmidt et al, *FoVeOOS* ’11]
Extensions

- **modular** termination proofs and **recursion**
  [Brockschmidt et al, *RTA ’11*]

- proving **reachability** and **non-termination** (uses only symbolic execution graph)
  [Brockschmidt et al, *FoVeOOS ’11*]

- proving termination with **cyclic data objects** (preprocessing in symbolic execution graph)
  [Brockschmidt et al, *CAV ’12*]
Extensions

- **modular** termination proofs and **recursion**
  [Brockschmidt et al, *RTA '11*]

- proving **reachability** and **non-termination** (uses only symbolic execution graph)
  [Brockschmidt et al, *FoVeOOS '11*]

- proving termination with **cyclic data objects** (preprocessing in symbolic execution graph)
  [Brockschmidt et al, *CAV '12*]

- proving upper bounds for **time complexity** (abstracts terms to numbers)
  [Frohn and Giesl, *iFM '17*]
So far: Java as a memory-safe object-oriented language
→ out-of-bounds memory accesses in Java: well-defined exceptions
From Java to C

- So far: Java as a memory-safe object-oriented language
  → out-of-bounds memory accesses in Java: well-defined exceptions
- Now: C as a systems programming language with pointer arithmetic and no guarantees of memory safety
  → out-of-bounds memory accesses in C: undefined behaviour
From Java to C

- So far: Java as a memory-safe object-oriented language
  → out-of-bounds memory accesses in Java: well-defined exceptions

- Now: C as a systems programming language with pointer arithmetic and **no** guarantees of memory safety
  → out-of-bounds memory accesses in C: **undefined behaviour**
    - replacing all files on the computer with cat GIFs
So far: Java as a memory-safe object-oriented language
→ out-of bounds memory accesses in Java: well-defined exceptions

Now: C as a systems programming language with pointer arithmetic and no guarantees of memory safety
→ out-of bounds memory accesses in C: undefined behaviour
  - replacing all files on the computer with cat GIFs
  - information leaks (Heartbleed)
So far: Java as a memory-safe object-oriented language
   → out-of-bounds memory accesses in Java: well-defined exceptions

Now: C as a systems programming language with pointer arithmetic and no guarantees of memory safety
   → out-of-bounds memory accesses in C: undefined behaviour
      - replacing all files on the computer with cat GIFs
      - information leaks (Heartbleed)
      - non-termination
So far: Java as a memory-safe object-oriented language
→ out-of-bounds memory accesses in Java: well-defined exceptions

Now: C as a systems programming language with pointer arithmetic and no guarantees of memory safety
→ out-of-bounds memory accesses in C: undefined behaviour
  - replacing all files on the computer with cat GIFs
  - information leaks (Heartbleed)
  - non-termination
  - ...
From Java to C

- So far: Java as a memory-safe object-oriented language
  → out-of-bounds memory accesses in Java: well-defined exceptions

- Now: C as a systems programming language with pointer arithmetic and no guarantees of memory safety
  → out-of-bounds memory accesses in C: *undefined behaviour*
    - replacing all files on the computer with cat GIFs
    - information leaks (Heartbleed)
    - non-termination
    - ...

⇒ C programs must be memory safe as a precondition for termination!
From Java to C

- So far: Java as a memory-safe object-oriented language → out-of-bounds memory accesses in Java: well-defined exceptions
- Now: C as a systems programming language with pointer arithmetic and no guarantees of memory safety → out-of-bounds memory accesses in C: **undefined behaviour**
  - replacing all files on the computer with cat GIFs
  - information leaks (Heartbleed)
  - **non-termination**
  - ...

⇒ C programs must be memory safe as a precondition for termination!

- Use case: programs on strings represented as char arrays whose last element has 0 as entry (“0-terminated strings”)
From Java to C

- So far: Java as a memory-safe object-oriented language
  → out-of-bounds memory accesses in Java: well-defined exceptions
- Now: C as a systems programming language with pointer arithmetic
  and no guarantees of memory safety
  → out-of-bounds memory accesses in C: **undefined behaviour**
    - replacing all files on the computer with cat GIFs
    - information leaks (Heartbleed)
    - **non-termination**
    - ...

⇒ C programs must be memory safe as a precondition for termination!
- Use case: programs on strings represented as char arrays whose last element has 0 as entry ("0-terminated strings")
- Tailor two-stage approach to C [Ströder et al, JAR ’17]
Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*(++s));
    return s - str;
}
```
Precondition: \texttt{str} points to allocated 0-terminated string

Is this program \textit{memory-safe} and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*(++s));
    return s-str;
}
```

No memory access outside allocated memory!
Precondition: \texttt{str} points to allocated 0-terminated string

Is this program \texttt{memory-safe} and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*(++s));
    return s-str;
}
```

No memory access outside allocated memory!

(precondition for termination)
Precondition: \texttt{str} points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*(++s));
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!
Motivation

Precondition: `str` points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*(++s));
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!
Precondition: `str` points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*(++s));
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!
Motivation

Precondition: `str` points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*(++s));
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognize!
Motivation

Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating? No! (violation of memory safety)

```c
int strlen(char* str) {
    char* s = str;
    while(*(++s));
    return s-str;
}
```

Bugs w.r.t. pointers are hard to recognise!
Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while((s)++);
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognize!
Motivation

Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while((s)++);
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!
Precondition: \texttt{str} points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while((*s)++);
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!
Precondition: \texttt{str} points to allocated 0-terminated string

Is this program memory-safe and terminating?

\begin{verbatim}
int strlen(char* str) {
    char* s = str;
    while((*s)++);
    return s-str;
}
\end{verbatim}
Precondition: `str` points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while((*s)++);
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!
Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating? No!

(int strlen(char* str) {
    char* s = str;
    while((s++));
    return s-str;
}

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!  

3 0

str
s
Precondition: \( \text{str} \) points to allocated 0-terminated string

Is this program memory-safe and terminating? \textbf{No!} (non-terminating – for unbounded integers)

```c
int strlen(char* str) {
    char* s = str;
    while(*s++);
    return s-str;
}
```

No memory access outside allocated memory! (precondition for termination)

Bugs w.r.t. pointers are hard to recognise!
Motivation

Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*s++);
    return s-str;
}
```

No memory access outside allocated memory!
(precondition for termination)

Bugs w.r.t. pointers are hard to recognise!
Motivation

Precondition: \texttt{str} points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*s++);
    return s-str;
}
```

Bugs w.r.t. pointers are hard to recognise!
Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*s++);
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognize!
Motivation

Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*s++);
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognize!
Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating?

```c
int strlen(char* str) {
    char* s = str;
    while(*s++);
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!
Precondition: \( \text{str} \) points to allocated 0-terminated string

Is this program memory-safe and terminating? Yes! But…

```c
int strlen(char* str) {
    char* s = str;
    while(*s++);
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognize!
Motivation

Precondition: str points to allocated 0-terminated string

Is this program memory-safe and terminating? Yes!

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!
Motivation

Precondition: \( \text{str} \) points to allocated 0-terminated string

Is this program memory-safe and terminating? Yes!

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

No memory access outside allocated memory!

Bugs w.r.t. pointers are hard to recognise!
Precondition: \texttt{str} points to allocated 0-terminated string

Is this program memory-safe and terminating? \textbf{Yes!}

\textbf{How to prove this automatically?}

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

Bugs w.r.t. pointers are hard to recognise!
Overview
Overview

compile

Symbolic Execution

prove memory safety

Integer Transition System

synthesize

prove termination

original program is memory-safe and terminating
Overview

Compile

Symbolic Execution Graph

Prove memory safety

Original program is memory-safe and terminating

Integer Transition System

Synthesise
Overview

Compile C

Symbolic Execution Graph

Prove memory safety

synthesize

Integer Transition System

prove termination

Original program is memory-safe and terminating
Overview

compile

Symbolic Execution Graph

prove memory safety

Integer Transition System

prove termination

synthesise
Overview

original program is memory-safe and terminating

compile

prove memory safety

prove termination

synthesize

Symbolic Execution Graph

Integer Transition System

C

LLVM

55/173
original program is memory-safe and terminating

prove termination

Symbolic Execution Graph

synthesize

prove memory safety

compile

Integer Transition System

Overview
original program is memory-safe and terminating

compile

prove memory safety

prove termination

synthesize

Symbolic Execution Graph
over-approximate operations
From Program to Symbolic Execution Graph (1/2)

- over-approximate operations
- inference rules for each instruction
over-approximate operations

inference rules for each instruction

refinement
• over-approximate operations
• inference rules for each instruction
• refinement
• generalisation
- over-approximate operations
- inference rules for each instruction
- refinement
- generalisation
- reduce reasoning to SMT
int strlen(char* str) {
  char* s = str;
  while(*s) s++;
  return s-str;
}

int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}

```c

```
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}

\[\text{str} \quad \text{if} \quad \text{str} = \text{end} \quad 0\]
\[\text{s} \quad \text{if} \quad \text{s} \neq 0 \quad \text{...} \quad 0\]
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
Overview

original program is memory-safe and terminating

compile

prove memory safety

Synthesize

prove termination

Symbolic Execution Graph

Integer Transition System

C

LLVM
Non-termination $\leadsto$ infinite run through graph
- Non-termination $\Rightarrow$ infinite run through graph
- Express graph traversal (SCCs)
Non-termination $\leadsto$ infinite run through graph

Express graph traversal (SCCs)

by Integer Transition System (ITS)
Non-termination $\leadsto$ infinite run through graph

Express graph traversal (SCCs) by Integer Transition System (ITS)

ITS terminating $\implies$ C program terminating
Function symbols: abstract states
Function symbols: abstract states
Arguments: variables occurring in states
- Function symbols: abstract states
- Arguments: variables occurring in states

![Diagram](attachment:image.png)
• Function symbols: abstract states
• Arguments: variables occurring in states

\[ \ell_A( ) \]
Function symbols: abstract states

Arguments: variables occurring in states

\[
\ell_A(str)
\]
Function symbols: abstract states
Arguments: variables occurring in states

\[ \ell_A(\text{str}, u_{\text{end}}) \]
Function symbols: abstract states
Arguments: variables occurring in states

\[ \ell_A(\text{str}, u_{end}, s) \]
Function symbols: abstract states
Arguments: variables occurring in states

\[ \ell_A(\text{str}, u_{end}, s) \rightarrow \ell_B(\quad ) \]
Function symbols: abstract states
Arguments: variables occurring in states

\[ \ell_A(str, u_{end}, s) \rightarrow \ell_B(str) \]
Function symbols: abstract states
Arguments: variables occurring in states

\[ \ell_A(\text{str}, u_{\text{end}}, s) \rightarrow \ell_B(\text{str}, u_{\text{end}}) \]
Function symbols: abstract states
Arguments: variables occurring in states

\[
\ell_A(\text{str}, \text{u}_{end}, s) \rightarrow \ell_B(\text{str}, \text{u}_{end}, s + 1)
\]
Function symbols: abstract states

Arguments: variables occurring in states
Resulting ITS (after automated simplification):
Resulting ITS (after automated simplification):

\[
\ell(x, y) \quad \xrightarrow{x<y} \quad \ell(x + 1, y)
\]
Resulting ITS (after automated simplification):

\[ \ell(x,y) \xrightarrow{x < y} \ell(x+1,y) \]

\[ x \quad \cdots \quad y \]

\[ \downarrow \quad \downarrow \]

\[ \cdots \quad 0 \]
Resulting ITS (after automated simplification):

\[ x < y \]
\[ \ell(x, y) \rightarrow \ell(x + 1, y) \]

Automatic termination proof by any termination prover
So far: assume that LLVM bitcode is essentially “the same” as C code

But: LLVM bitcode is much closer to assembly than C

Let’s look at the details of the actual analysis
original program is memory-safe and terminating

C

compiler

Symbolic Execution Graph

prove memory safety

prove termination

Integer Transition System

synthesize
The Low-Level Virtual Machine Framework

- LLVM used for compiler optimisation and verification
The Low-Level Virtual Machine Framework

- LLVM used for compiler optimisation and verification
- Close to assembly language
The Low-Level Virtual Machine Framework

- LLVM used for compiler optimisation and verification
- Close to assembly language
- Still structured: functions, data structures, type safety
The Low-Level Virtual Machine Framework

- LLVM used for compiler optimisation and verification
- Close to assembly language
- Still structured: functions, data structures, type safety
- Single Static Assignment (SSA)
The Low-Level Virtual Machine Framework

- LLVM used for compiler optimisation and verification
- Close to assembly language
- Still structured: functions, data structures, type safety
- Single Static Assignment (SSA)

Caveat: user-defined data structures (structs) in LLVM are still work in progress for AProVE
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

LLVM Code (simplified)

```c
define i32 strlen(i8* str) {
    entry:
    0: c0 = load i8* str
    1: c0zero = icmp eq i8 c0, 0
    br i1 c0zero, label done, label loop
loop:
    0: olds = phi i8* [str,entry], [s,loop]
    1: s = getelementptr i8* olds, i32 1
    2: c = load i8* s
    3: czero = icmp eq i8 c, 0
    br i1 czero, label done, label loop
done:
    0: sfin = phi i8* [str,entry], [s,loop]
    1: sfinint = ptrtoint i8* sfin to i32
    2: strint = ptrtoint i8* str to i32
    3: size = sub i32 sfinint, strint
    4: ret i32 size
}
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

LLVM Code (simplified)

```assembly
define i32 strlen(i8* str) {
    entry:
        0: c0 = load i8* str
        1: c0zero = icmp eq i8 c0, 0
        2: br i1 c0zero, label done, label loop
    loop:
        0: olds = phi i8* [str,entry],[s,loop]
        1: s = getelementptr i8* olds, i32 1
        2: c = load i8* s
        3: czero = icmp eq i8 c, 0
        4: br i1 czero, label done, label loop
    done:
        0: sfin = phi i8* [str,entry],[s,loop]
        1: sfinint = ptrtoint i8* sfin to i32
        2: strint = ptrtoint i8* str to i32
        3: size = sub i32 sfinint, strint
        4: ret i32 size
}
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

LLVM Code (simplified)

```llvm
define i32 strlen(i8* str) {
    entry:
        0: c0 = load i8* str
        1: c0zero = icmp eq i8 c0, 0
        2: br i1 c0zero, label done, label loop
    loop:
        0: olds = phi i8* [str,entry],[s,loop]
        1: s = getelementptr i8* olds, i32 1
        2: c = load i8* s
        3: czero = icmp eq i8 c, 0
        4: br i1 czero, label done, label loop
    done:
        0: sfin = phi i8* [str,entry],[s,loop]
        1: sfinint = ptrtoint i8* sfin to i32
        2: strint = ptrtoint i8* str to i32
        3: size = sub i32 sfinint, strint
        4: ret i32 size
}
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

LLVM Code (simplified)

```llvm
define i32 strlen(i8* str) {
  entry:
  0: c0 = load i8* str
  1: c0zero = icmp eq i8 c0, 0
  2: br i1 c0zero, label done, label loop
  loop:
  0: olds = phi i8* [str,entry],[s,loop]
  1: s = getelementptr i8* olds, i32 1
  done:
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

LLVM Code (simplified)

```assembly
define i32 strlen(i8* str) {
    entry:
      0: c0 = load i8* str
      1: c0zero = icmp eq i8 c0, 0
      2: br i1 c0zero, label done, label loop
    loop:
      0: olds = phi i8* [str,entry],[s,loop]
      1: s = getelementptr i8* olds, i32 1
      2: c = load i8* s
    done:
      0: sfin = phi i8* [str,entry],[s,loop]
      1: sfinint = ptrtoint i8* sfin to i32
      2: strint = ptrtoint i8* str to i32
      3: size = sub i32 sfinint, strint
      4: ret i32 size
}
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

LLVM Code (simplified)
```
define i32 strlen(i8* str) {
  entry:
      0: c0 = load i8* str
      1: c0zero = icmp eq i8 c0, 0
      2: br i1 c0zero, label done, label loop
  loop:
      0: olds = phi i8* [str,entry],[s,loop]
      1: s = getelementptr i8* olds, i32 1
      2: c = load i8* s
      3: czero = icmp eq i8 c, 0
          br i1 czero, label done, label loop
  done:
      0: sfin = phi i8* [str,entry],[s,loop]
      1: sfinint = ptrtoint i8* sfin to i32
      2: strint = ptrtoint i8* str to i32
      3: size = sub i32 sfinint, strint
      4: ret i32 size
}
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

LLVM Code (simplified)

```llvm
define i32 strlen(i8* str) {
  entry:
  0: c0 = load i8* str
  1: c0zero = icmp eq i8 c0, 0
  2: br i1 c0zero, label done, label loop
  loop:
  0: olds = phi i8* [str,entry],[s,loop]
  1: s = getelementptr i8* olds, i32 1
  2: c = load i8* s
  3: czero = icmp eq i8 c, 0
  4: br i1 czero, label done, label loop
  done:
  0: sfin = phi i8* [str,entry],[s,loop]
  1: sfinint = ptrtoint i8* sfin to i32
  2: strint = ptrtoint i8* str to i32
  3: size = sub i32 sfinint, strint
  4: ret i32 size
}
```
Example C Program

```c
int strlen(char* str) {
    char* s = str;
    while(*s) s++;
    return s-str;
}
```

**LLVM Code (simplified)**

```llvm
define i32 strlen(i8* str) {
  entry:
    0: c0 = load i8* str
    1: c0zero = icmp eq i8 c0, 0
    2: br i1 c0zero, label done, label loop
  loop:
    0: olds = phi i8* [str,entry],[s,loop]
    1: s = getelementptr i8* olds, i32 1
    2: c = load i8* s
    3: czero = icmp eq i8 c, 0
    4: br i1 czero, label done, label loop
  done:
    0: sfin = phi i8* [str,entry],[s,loop]
    1: sfinint = ptrtoint i8* sfin to i32
    2: strint = ptrtoint i8* str to i32
    3: size = sub i32 sfinint, strint
    4: ret i32 size
}
```
Overview

original program is memory-safe and terminating

compile

prove memory safety

Symbolic Execution Graph

prove termination

Integer Transition System

synthesize
Abstract domain:
Abstract domain:

- represent system configurations as states
Abstract domain:

- represent system configurations as states
- represent operations as edges
Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations
Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations

Initial State:
Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations
  - program position \( pos \): previous block, current block, line number

Initial State:
Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations
  - program position $pos$: previous block, current block, line number

Initial State:

\[
pos = (\varepsilon, \text{entry}, 0)\]
From LLVM to Symbolic Execution Graph

Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations
  - program position $pos$: previous block, current block, line number
  - allocation list $AL$

Initial State:

$$pos = (\varepsilon, \text{entry}, 0)$$
Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations
  - program position $pos$: previous block, current block, line number
  - allocation list $AL$

Initial State:

$pos = (\varepsilon, \text{entry}, 0)$

$AL = \{\text{alloc} (\text{str}, u_{end})\}$
From LLVM to Symbolic Execution Graph

Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations
  - program position pos: previous block, current block, line number
  - allocation list AL
  - points to map PT

Initial State:

\[
\begin{align*}
pos &= (\varepsilon, \text{entry}, 0) \\
AL &= \{ \text{alloc}(\text{str}, u_{\text{end}}) \}
\end{align*}
\]
Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations
  - program position $pos$: previous block, current block, line number
  - allocation list $AL$
  - points to map $PT$

Initial State:

\[
\begin{align*}
pos &= (\varepsilon, \text{entry}, 0) \\
AL &= \{\text{alloc(str, } u_{\text{end}})\} \\
PT &= \{u_{\text{end}} \mapsto_{\text{i8}} 0\}
\end{align*}
\]
Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations

  - program position $pos$: previous block, current block, line number
  - allocation list $AL$
  - points to map $PT$
  - knowledge base $KB$

Initial State:

\[
\begin{align*}
pos &= (\varepsilon, \text{entry}, 0) \\
AL &= \{ \text{alloc}(\text{str}, u_{\text{end}}) \} \\
PT &= \{ u_{\text{end}} \mapsto_{i8} 0 \}
\end{align*}
\]
Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations

- program position $pos$: previous block, current block, line number
- allocation list $AL$
- points to map $PT$
- knowledge base $KB$

Initial State:

$$pos = (\varepsilon, \text{entry}, 0)$$
$$AL = \{ alloc(str, u_{end}) \}$$
$$PT = \{ u_{end} \mapsto _{i8} 0 \}$$
$$KB = \emptyset$$
Abstract domain:

- represent system configurations as states
- represent operations as edges
- abstract states stand for sets of configurations

- program position $pos$: previous block, current block, line number
- allocation list $AL$
- points to map $PT$
- knowledge base $KB$

Initial State:

$pos = (\varepsilon, \text{entry}, 0)$
$AL = \{ alloc(\text{str}, u_{end}) \}$
$PT = \{ u_{end} \rightarrow^{i8} 0 \}$
$KB = \emptyset$

formal semantics for states:
Separation Logic [O’Hearn, Reynolds, Yang, CSL ’01]
over-approximate program states and operations
From LLVM to Symbolic Execution Graph

- over-approximate program states and operations
- inference rules for each instruction
From LLVM to Symbolic Execution Graph

- over-approximate program states and operations
- inference rules for each instruction
- refinement
From LLVM to Symbolic Execution Graph

- over-approximate program states and operations
- inference rules for each instruction
- refinement
- generalisation
From LLVM to Symbolic Execution Graph

- over-approximate program states and operations
- inference rules for each instruction
- refinement
- generalisation
- automation via SMT solving (SAT Modulo Theories)
define i32 strlen(i8* str) {
entry:
  0: c0 = load i8* str
  ...

define i32 strlen(i8* str) {
  entry:
      0: c0 = load i8* str
      ...

define i32 strlen(i8* str) {
entry:
  0: c0 = load i8* str
  ...
}

\[
\begin{align*}
\text{Initial state:} & \\
pos &= (\varepsilon, \text{entry}, 0) \\
AL &= \{ \text{alloc}(\text{str}, u_{end}) \} \\
PT &= \{ u_{end} \mapsto_{i8} 0 \} \\
KB &= \emptyset
\end{align*}
\]
define i32 strlen(i8* str) {
    entry:
      0: c0 = load i8* str
      ...

    Initial state:
    pos = (ε, entry, 0)
    AL = {alloc(str, u_end)}
    PT = {u_end ↦ i8 0}
    KB = ∅

    Evaluation
    pos = (ε, entry, 1)
    AL = {alloc(str, u_end)}
    PT = {u_end ↦ i8 0,
          str ↦ i8 c0}
    KB = ∅
define i32 strlen(i8* str) {
  entry:
    0: c0 = load i8* str
    ...

  Initial state:
  pos = (ε, entry, 0)
  AL = \{ alloc(str, u_{end} ) \}
  PT = \{ u_{end} \rightarrow_{i8} 0 \}
  KB = \emptyset

  Evaluation
  Memory access: check allocation!
... entry:
  0: c0 = load i8* str
  1: c0zero = icmp eq i8 c0, 0
...

\[ pos = (\varepsilon, \text{entry}, 1) \]
\[ AL = \{ \text{alloc}(\text{str}, u_{\text{end}}) \} \]
\[ PT = \{ u_{\text{end}} \mapsto_{\text{i8}} 0, \]
\[ \quad \text{str} \mapsto_{\text{i8}} c0 \} \]
\[ KB = \emptyset \]
From LLVM to Symbolic Execution Graph

... entry:
0: c0 = load i8* str
1: c0zero = icmp eq i8 c0, 0
...

pos = (ε, entry, 1)
AL = \{ alloc(str, u_{end}) \}
PT = \{ u_{end} \mapsto_{i8} 0, 
          str \mapsto_{i8} c0 \}
KB = \emptyset

\[
c0 = 0 \quad \text{or} \quad c0 \neq 0
\]

Refinement
entry:
0: \texttt{c0 = load i8* str}
1: \texttt{c0zero = icmp eq i8 c0, 0}

\[
\begin{align*}
\text{pos} &= (\varepsilon, \text{entry}, 1) \\
\text{AL} &= \{\text{alloc}(\text{str}, \text{u}_{\text{end}})\} \\
\text{PT} &= \{\text{u}_{\text{end}} \rightarrow_{i8} 0, \\
& \quad \text{str} \rightarrow_{i8} \text{c0}\} \\
\text{KB} &= \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{pos} &= (\varepsilon, \text{entry}, 1) \\
\text{AL} &= \{\text{alloc}(\text{str}, \text{u}_{\text{end}})\} \\
\text{PT} &= \{\text{u}_{\text{end}} \rightarrow_{i8} 0, \\
& \quad \text{str} \rightarrow_{i8} \text{c0}\} \\
\text{KB} &= \{\text{c0} = 0\}
\end{align*}
\]

\[
\begin{align*}
\text{pos} &= (\varepsilon, \text{entry}, 1) \\
\text{AL} &= \{\text{alloc}(\text{str}, \text{u}_{\text{end}})\} \\
\text{PT} &= \{\text{u}_{\text{end}} \rightarrow_{i8} 0, \\
& \quad \text{str} \rightarrow_{i8} \text{c0}\} \\
\text{KB} &= \{\text{c0} \neq 0\}
\end{align*}
\]
... loop:
  0: oldds = phi i8* [str, entry], [s, loop]
  1: s = getelementptr i8* oldds, i32 1
...

pos = (loop, loop, 0)
AL = \{ alloc(str, uend) \}
PT = \{ uend \rightarrow_{i8} 0,
    str \rightarrow_{i8} c0, s \rightarrow_{i8} c \}
KB = \{ c \neq 0, s = oldds + 1,
    c0 \neq 0, oldds = str \}
From LLVM to Symbolic Execution Graph

... loop:
  0: olds = phi i8* [str,entry],[s,loop]
  1: s = getelementptr i8* olds, i32 1
...

\[
pos = (\text{loop}, \text{loop}, 0)\]
\[
AL = \{ alloc(str, u_{end}) \} \]
\[
PT = \{ u_{end} \mapsto_{i8} 0, \\
    \quad \text{str} \mapsto_{i8} c0, s \mapsto_{i8} c \} \]
\[
KB = \{ c \neq 0, s = olds + 1, \\
    \quad c0 \neq 0, olds = str \} \]
From LLVM to Symbolic Execution Graph

... loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1
...

pos = (loop,loop,0)
AL = \{ alloc(str,u_end) \}
PT = \{ u_end ←i8 0,
  str ←i8 c0, s ←i8 c,
  olds ←i8 v \}
KB = \{ c ≠ 0, s = olds + 1,
  c0 ≠ 0, olds = str \}
From LLVM to Symbolic Execution Graph

... loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1
...

\[
\begin{align*}
pos &= (loop, loop, 0) \\
AL &= \{ alloc(str, u_{end}) \} \\
PT &= \{ u_{end} \leftarrow_{i8} 0, \\
& \quad \text{str} \leftarrow_{i8} c0, \text{s} \leftarrow_{i8} c \} \\
KB &= \{ c \neq 0, \text{s} = \text{olds} + 1, \\
& \quad \text{c0} \neq 0, \text{olds} = \text{str} \}
\end{align*}
\]

Generalisation
From LLVM to Symbolic Execution Graph

... loop:
  0: olds = phi i8* [str,entry],[s,loop]
  1: s = getelementptr i8* olds, i32 1
...

pos = (loop, loop, 0)
AL = \{ alloc(str, u_{end}) \}
PT = \{ u_{end} \rightarrow_{i8} 0,
        str \rightarrow_{i8} c0, s \rightarrow_{i8} c \}
KB = \{ c \neq 0, s = olds + 1,
       c0 \neq 0, olds = str \}

Generalisation (to obtain finite graph)
From LLVM to Symbolic Execution Graph

... loop:
  0: olds = phi i8* [str,entry],[s,loop]
  1: s = getelementptr i8* olds, i32 1
...

\[
\begin{align*}
pos &= (\text{loop, loop, 0}) \\
\text{AL} &= \{\text{alloc}(\text{str, u} \_ \text{end})\} \\
\text{PT} &= \{\text{u} \_ \text{end} \leftarrow i8 0, \\
& \quad \text{str} \leftarrow i8 \text{c}0, \text{s} \leftarrow i8 \text{c} \}
\end{align*}
\]

\[
\begin{align*}
\text{KB} &= \{c \neq 0, \text{s} = \text{olds} + 1, \\
& \quad \text{c}0 \neq 0, \text{olds} = \text{str}\}
\end{align*}
\]
From LLVM to Symbolic Execution Graph

... loop:
  0: olds = phi i8* [str, entry], [s, loop]
  1: s = getelementptr i8* olds, i32 1
...

\[ pos = (\text{loop, loop, 0}) \]
\[ AL = \{ \text{alloc(str, u\_end)} \} \]
\[ PT = \{ u\_end \leftarrow i8 0, \]
\[ \quad \text{str} \leftarrow i8 c0, s \leftarrow i8 c, \]
\[ \quad \text{olds} \leftarrow i8 v \} \]
\[ KB = \{ c \neq 0, s = \text{olds} + 1, \]
\[ \quad c0 \neq 0, \text{olds} = \text{str} \} \]
... loop:  
0: olds = phi i8* [str,entry],[s,loop]  
1: s = getelementptr i8* olds, i32 1  
...

pos = (loop, loop, 0)  
AL = \{ alloc(str, u\text{end}) \}  
PT = \{ u\text{end} \rightarrow_{i8} 0, 
    str \rightarrow_{i8} c0, s \rightarrow_{i8} c \}  
KB = \{ c \neq 0, s = olds + 1, 
    c0 \neq 0, olds = str \}
From LLVM to Symbolic Execution Graph

... loop:
  0: olds = phi i8* [str, entry], [s, loop]
  1: s = getelementptr i8* olds, i32 1

\[
\begin{align*}
pos &= (\text{loop}, \text{loop}, 0) \\
AL &= \{ \text{alloc}(\text{str}, u_{end}) \} \\
PT &= \{ u_{end} \leftarrow_{i8} 0, \\
 & \quad \text{str} \leftarrow_{i8} c0, s \leftarrow_{i8} c, \\
 & \quad \text{olds} \leftarrow_{i8} v \} \\
KB &= \{ c \neq 0, s = \text{olds} + 1, \\
 & \quad c0 \neq 0, \text{olds} = \text{str} \}
\end{align*}
\]

Generalisation

\[
\begin{align*}
pos &= (\text{loop}, \text{loop}, 0) \\
AL &= \{ \text{alloc}(\text{str}, u_{end}) \} \\
PT &= \{ u_{end} \leftarrow_{i8} 0, \\
 & \quad \text{str} \leftarrow_{i8} c0, \\
 & \quad \} \\
KB &= \{ c \neq 0, v \neq 0, \\
 & \quad s = \text{olds} + 1, c0 \neq 0, \\
 & \quad \text{olds} = \text{str} + 1 \}
\end{align*}
\]
From LLVM to Symbolic Execution Graph

... loop:
    0: olds = phi i8* [str,entry],[s,loop]
    1: s = getelementptr i8* olds, i32 1
...

\[
\begin{align*}
    \text{pos} &= (\text{loop,loop,0}) \\
    \text{AL} &= \{ \text{alloc(str,u}_\text{end}) \} \\
    \text{PT} &= \{ u\text{end} \mapsto \text{i8} 0, \text{str} \mapsto \text{i8} c0, s \mapsto \text{i8} c, \text{olds} \mapsto \text{i8} v \} \\
    \text{KB} &= \{ c \neq 0, v \neq 0, s = \text{olds} + 1, c0 \neq 0, \text{olds} = \text{str} \}
\end{align*}
\]
From LLVM to Symbolic Execution Graph

... loop:
  0: olds = phi i8* [str,entry],[s,loop]
  1: s = getelementptr i8* olds, i32 1
...

pos = (loop, loop, 0)
AL = \{ alloc(str, u_{end}) \}
PT = \{ u_{end} \mapsto_{i8} 0, 
       str \mapsto_{i8} c0, s \mapsto_{i8} c, 
       olds \mapsto_{i8} v \}
KB = \{ c \neq 0, s = olds + 1, 
       c0 \neq 0, olds = str \}

pos = (loop, loop, 0)
AL = \{ alloc(str, u_{end}) \}
PT = \{ u_{end} \mapsto_{i8} 0, 
       str \mapsto_{i8} c0, s \mapsto_{i8} c, 
       olds \mapsto_{i8} v \}
... loop:
0: olds = phi i8* [str, entry], [s, loop]
1: s = getelementptr i8* olds, i32 1
...
From LLVM to Symbolic Execution Graph

...  
loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1
...

```
pos = (loop,loop,0)
AL = \{alloc(str,u_end)\}
PT = \{u_end \rightarrow_{i8} 0,
  str \rightarrow_{i8} c0, s \rightarrow_{i8} c, 
  olds \rightarrow_{i8} v\}
KB = \{c \neq 0, s = olds + 1,
  c0 \neq 0, olds = str\}
```

```generalisation```
pos = (loop,loop,0)
AL = \{alloc(str,u_end)\}
PT = \{u_end \rightarrow_{i8} 0,
  str \rightarrow_{i8} c0, s \rightarrow_{i8} c, 
  olds \rightarrow_{i8} v\}
KB = \{c \neq 0, v \neq 0,
  s = olds + 1, c0 \neq 0,
  olds = str + 1\}
```
From LLVM to Symbolic Execution Graph

... loop:
  0: olds = phi i8* [str,entry],[s,loop]
  1: s = getelementptr i8* olds, i32 1
  ...

\[
\begin{align*}
opos &= (\text{loop,loop,0}) \\
\text{AL} &= \{ \text{alloc}(\text{str}, u_{\text{end}}) \} \\
\text{PT} &= \{ u_{\text{end}} \leftarrow_{i8} 0, \\
&\quad \text{str} \leftarrow_{i8} c\theta, s \leftarrow_{i8} c, \\
&\quad \text{olds} \leftarrow_{i8} v \} \\
\text{KB} &= \{ c \neq 0, s = \text{olds} + 1, \\
&\quad c\theta \neq 0, \text{olds} = \text{str} \}
\end{align*}
\]

\[
\begin{align*}
opos &= (\text{loop,loop,0}) \\
\text{AL} &= \{ \text{alloc}(\text{str}, u_{\text{end}}) \} \\
\text{PT} &= \{ u_{\text{end}} \leftarrow_{i8} 0, \\
&\quad \text{str} \leftarrow_{i8} c\theta, s \leftarrow_{i8} c, \\
&\quad \text{olds} \leftarrow_{i8} v \} \\
\text{KB} &= \{ c \neq 0, v \neq 0, \\
&\quad s = \text{olds} + 1, c\theta \neq 0, \\
&\quad \text{olds} = \text{str} + 1 \}
\end{align*}
\]
... loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1
...

\[
\begin{align*}
pos &= (\text{loop, loop, 0}) \\
AL &= \{ alloc(str, u_{end}) \} \\
PT &= \{ u_{end} \mapsto i8 0, \\
& \quad \text{str} \mapsto i8 \ c0, s \mapsto i8 c, \\
& \quad \text{olds} \mapsto i8 v \} \\
KB &= \{ c \neq 0, s = \text{olds} + 1, \\
& \quad c0 \neq 0, \text{olds} = \text{str} \}
\end{align*}
\]

\[
\begin{align*}
pos &= (\text{loop, loop, 0}) \\
AL &= \{ alloc(str, u_{end}) \} \\
PT &= \{ u_{end} \mapsto i8 0, \\
& \quad \text{str} \mapsto i8 \ c0, s \mapsto i8 c, \\
& \quad \text{olds} \mapsto i8 v \} \\
KB &= \{ c \neq 0, v \neq 0, \\
& \quad s = \text{olds} + 1, c0 \neq 0, \\
& \quad \text{olds} = \text{str} + 1 \}
\end{align*}
\]
From LLVM to Symbolic Execution Graph

... loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1
...

pos = (loop, loop, 0)
AL = {alloc(str, u\text{end})}
PT = {u\text{end} \rightarrow i8 0,
      str \rightarrow i8 c0, s \rightarrow i8 c,
      olds \rightarrow i8 v}
KB = {c \neq 0, v \neq 0,
      s = olds + 1, c0 \neq 0,
      olds = str + 1}

Generalisation

pos = (loop, loop, 0)
AL = {alloc(str, u\text{end})}
PT = {u\text{end} \rightarrow i8 0,
      str \rightarrow i8 c0, s \rightarrow i8 c,
      olds \rightarrow i8 v}
KB = {c \neq 0, v \neq 0,
      s = olds + 1, c0 \neq 0,
      olds = str}
From LLVM to Symbolic Execution Graph

... loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1
...

\[
\begin{align*}
pos &= (\text{loop,loop,0}) \\
AL &= \{\text{alloc} (\text{str,u}_{\text{end}})\} \\
PT &= \{\text{u}_{\text{end}} \rightarrow \text{i8} 0, \\
    \text{str} \rightarrow \text{i8} \text{c0}, \text{s} \rightarrow \text{i8} \text{c}, \\
    \text{olds} \rightarrow \text{i8} \text{v}\} \\
KB &= \{c \neq 0, s = \text{olds} + 1, \\
    \text{c0} \neq 0, \text{olds} = \text{str}\}
\end{align*}
\]

\[
x = y \iff x \geq y \land x \leq y
\]
From LLVM to Symbolic Execution Graph

... loop:
  0: olds = phi i8* [str,entry],[s,loop]
  1: s = getelementptr i8* olds, i32 1
...

pos = (loop, loop, 0)
AL = \{ alloc(str, u_{end}) \}
PT = \{ u_{end} \mapsto_{i8} 0,
         str \mapsto_{i8} c0, s \mapsto_{i8} c,
         olds \mapsto_{i8} v \}
KB = \{ c \neq 0, s = olds + 1,
        c0 \neq 0, olds = str \}

pos = (loop, loop, 0)
AL = \{ alloc(str, u_{end}) \}
PT = \{ u_{end} \mapsto_{i8} 0,
         str \mapsto_{i8} c0, s \mapsto_{i8} c,
         olds \mapsto_{i8} v \}
KB = \{ c \neq 0, v \neq 0,
        s = olds + 1, c0 \neq 0,
        olds \geq str, \}
loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1

pos = (loop, loop, 0)
AL = {alloc(str, u_end)}
x_1 \rightarrow_{ty} y_1 \land 
x_2 \rightarrow_{ty} y_2 \land 
y_1 \neq y_2

PT = \{u_{end} \leftarrow_{i8} 0, 
\quad str \leftarrow_{i8} c0, s \leftarrow_{i8} c, 
\quad olds \leftarrow_{i8} v\}
KB = \{c \neq 0, v \neq 0, 
\quad s = olds + 1, c0 \neq 0, 
\quad olds \geq str,\}
... loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1
...

pos = (loop,loop,0)
AL = {alloc(str,u_end)}

\[
x_1 \rightarrow_{ty} y_1 \land \\
x_2 \rightarrow_{ty} y_2 \land \\
y_1 \neq y_2
\]

\[\implies x_1 \neq x_2\]

PT = \{u_end \leftarrow_{i8} 0, \\
str \leftarrow_{i8} c0, s \leftarrow_{i8} c, \\
olds \leftarrow_{i8} v\}

KB = \{c \neq 0, v \neq 0, \\
s = olds + 1, c0 \neq 0, \\
olds \geq str, \}

\]
From LLVM to Symbolic Execution Graph

... loop:
  0: olds = phi i8* [str,entry],[s,loop]
  1: s = getelementptr i8* olds, i32 1

pos = (loop, loop, 0)
AL = \{ alloc(str, u_{end}) \}
\begin{align*}
x_1 & \dashv_{ty} y_1 \land \\
x_2 & \dashv_{ty} y_2 \land \implies x_1 \neq x_2 \\
y_1 & \neq y_2
\end{align*}

Check whether
\[
x_1 \lt x_2 \text{ or } x_1 \gt x_2
\]
holds!

\[
PT = \{ u_{end} \dashv_{i8} 0, \\
str \dashv_{i8} c0, s \dashv_{i8} c, \\
olds \dashv_{i8} v \}
\]

\[
KB = \{ c \neq 0, v \neq 0, \\
s = olds + 1, c0 \neq 0, \\
olds \geq str, \}
\]
From LLVM to Symbolic Execution Graph

... loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1
...

pos = (loop, loop, 0)
AL = \{ alloc(str, u_end) \}
PT = \{ u_end \rightarrow i8 0,
str \rightarrow i8 c0, s \rightarrow i8 c,
olds \rightarrow i8 v \}
KB = \{ c \neq 0, s = olds + 1, 
c0 \neq 0, olds = str \}

pos = (loop, loop, 0)
AL = \{ alloc(str, u_end) \}
PT = \{ u_end \rightarrow i8 0,
str \rightarrow i8 c0, s \rightarrow i8 c,
olds \rightarrow i8 v \}
KB = \{ c \neq 0, v \neq 0, 
s = olds + 1, c0 \neq 0, 
olds \geq str, s \neq u_end \}
From LLVM to Symbolic Execution Graph

... loop:
0: olds = phi i8* [str,entry],[s,loop]
1: s = getelementptr i8* olds, i32 1
...

\[pos = (\text{loop}, \text{loop}, 0)\]

\[AL = \{ alloc(\text{str}, \text{u}_{\text{end}}) \}\]

\[PT = \{ u_{\text{end}} \mapsto_{i8} 0, \]
\[\text{str} \mapsto_{i8} c0, \text{s} \mapsto_{i8} c, \]
\[\text{olds} \mapsto_{i8} v \}\]

\[KB = \{ c \neq 0, s = \text{olds} + 1, \]
\[c0 \neq 0, \text{olds} = \text{str} \}\]

Generalisation
From LLVM to Symbolic Execution Graph

\[
pos = (\text{loop}, \text{loop}, 0) \\
AL = \{ \text{alloc}(\text{str}, u_{end}) \} \\
PT = \{ u_{end} \leftarrow 0, \str \leftarrow c_0, s \leftarrow c, \olds \leftarrow v \} \\
KB = \{ c \neq 0, v \neq 0, s = \olds + 1, c_0 \neq 0, \olds \geq \text{str}, s < u_{end} \} \]
From LLVM to Symbolic Execution Graph

\[ pos = (\text{loop}, \text{loop}, 0) \]
\[ AL = \{ \text{alloc} (\text{str}, u_{end}) \} \]
\[ PT = \{ u_{end} \rightarrow_{i8} 0, \]
\[ \text{str} \rightarrow_{i8} c0, \text{s} \rightarrow_{i8} c, \]
\[ \text{olds} \rightarrow_{i8} v \} \]
\[ KB = \{ c \neq 0, v \neq 0, \]
\[ s = \text{olds} + 1, c0 \neq 0, \]
\[ \text{olds} \geq \text{str}, s < u_{end} \} \]
original program is memory-safe and terminating

compile

prove memory safety

Symbolic Execution Graph

prove termination

synthesize

Integer Transition System
Non-termination $\leadsto$ infinite run through graph

Express graph traversal (strongly connected components)

by Integer Transition System (ITS)

ITS terminating $\implies$ C program terminating
Function symbols: abstract states
Function symbols: abstract states
Arguments: variables occurring in states
Function symbols: abstract states

Arguments: variables occurring in states

\[ \text{pos} = (\varepsilon, \text{entry}, 1) \]
\[ AL = \{ alloc(str, u_{end}) \} \]
\[ PT = \{ u_{end} \mapsto_{i8} 0, \]
\[ \phantom{PT} \quad \text{str} \mapsto_{i8} c0 \} \]
\[ KB = \emptyset \]

\[ \text{pos} = (\varepsilon, \text{entry}, 1) \]
\[ AL = \{ alloc(str, u_{end}) \} \]
\[ PT = \{ u_{end} \mapsto_{i8} 0, \]
\[ \phantom{PT} \quad \text{str} \mapsto_{i8} c0 \} \]
\[ KB = \{ c0 \neq 0 \} \]
Function symbols: abstract states

Arguments: variables occurring in states

\[
\ell_B( )
\]

**B**

\[
pos = (\varepsilon, \text{entry}, 1) \\
AL = \{alloc(str, u_{end})\} \\
PT = \{u_{end} \leftrightarrow_{i8} 0, \\
str \leftrightarrow_{i8} c0\} \\
KB = \emptyset
\]

**D**

\[
pos = (\varepsilon, \text{entry}, 1) \\
AL = \{alloc(str, u_{end})\} \\
PT = \{u_{end} \leftrightarrow_{i8} 0, \\
str \leftrightarrow_{i8} c0\} \\
KB = \{c0 \neq 0\}
\]
Function symbols: abstract states

Arguments: variables occurring in states

\[ pos = (\varepsilon, \text{entry}, 1) \]
\[ AL = \{ \text{alloc}(\text{str}, u_{\text{end}}) \} \]
\[ PT = \{ u_{\text{end}} \leftarrow_{i8} 0, \]
\[ \text{str} \leftarrow_{i8} c\emptyset \} \]
\[ KB = \emptyset \]

\[ \ell_B(\text{str}) \]
Function symbols: abstract states
Arguments: variables occurring in states

\[ \ell_B(\text{str}, u_{end}) \]
Function symbols: abstract states

Arguments: variables occurring in states

\[ \ell_B(str, u_{end}, c0) \]
- Function symbols: abstract states
- Arguments: variables occurring in states

\[ \ell_B(\text{str}, \text{u}_{\text{end}}, \text{c}0) \rightarrow \ell_D(\ ) \]
Function symbols: abstract states
Arguments: variables occurring in states

\[ pos = (\varepsilon, \text{entry}, 1) \]
\[ AL = \{ \text{alloc}(\text{str}, u_{end}) \} \]
\[ PT = \{ u_{end} \leftarrow_{i8} 0, \]
\[ \quad \text{str} \leftarrow_{i8} c0 \} \]
\[ KB = \emptyset \]

\[ pos = (\varepsilon, \text{entry}, 1) \]
\[ AL = \{ \text{alloc}(\text{str}, u_{end}) \} \]
\[ PT = \{ u_{end} \leftarrow_{i8} 0, \]
\[ \quad \text{str} \leftarrow_{i8} c0 \} \]
\[ KB = \{ c0 \neq 0 \} \]

\[ \ell_B(\text{str}, u_{end}, c0) \quad \longrightarrow \quad \ell_D(\text{str} \quad \quad ) \]
Function symbols: abstract states

Arguments: variables occurring in states

\[
\begin{align*}
pos &= (\varepsilon, \text{entry}, 1) \\
AL &= \{\text{alloc}(\text{str}, u_{\text{end}})\} \\
PT &= \{u_{\text{end}} \mapsto_{i8} 0, \\
    \text{str} \mapsto_{i8} c0\} \\
KB &= \emptyset
\end{align*}
\]

\[
\begin{align*}
\ell_B(\text{str}, u_{\text{end}}, c0) &\quad\longrightarrow\quad \ell_D(\text{str}, u_{\text{end}})
\end{align*}
\]
Function symbols: abstract states
Arguments: variables occurring in states

\[
\begin{align*}
\ell_B : \text{str, u}_{\text{end}}, c0 \quad &\longrightarrow \quad \ell_D (\text{str, u}_{\text{end}}, c0) \\
B &\quad \text{pos} = (\varepsilon, \text{entry}, 1) \\
&\quad AL = \{ \text{alloc(str, u}_{\text{end}}) \} \\
&\quad PT = \{ u_{\text{end}} \leftarrow_{i8} 0, \\
&\quad \quad \text{str} \leftarrow_{i8} c0 \} \\
&\quad KB = \emptyset \\
D &\quad \text{pos} = (\varepsilon, \text{entry}, 1) \\
&\quad AL = \{ \text{alloc(str, u}_{\text{end}}) \} \\
&\quad PT = \{ u_{\text{end}} \leftarrow_{i8} 0, \\
&\quad \quad \text{str} \leftarrow_{i8} c0 \} \\
&\quad KB = \{ c0 \neq 0 \}
\end{align*}
\]
From Symb. Exec. Graph to Integer Transition Systems (2/3)

- Function symbols: abstract states
- Arguments: variables occurring in states

\[ \ell_B(\text{str}, u_{\text{end}}, c0) \xrightarrow{c0 \neq 0} \ell_D(\text{str}, u_{\text{end}}, c0) \]
Resulting ITS (after automated simplification):
Resulting ITS (after automated simplification):

\[ \ell(x, y) \xrightarrow{x < y} \ell(x + 1, y) \]
Resulting ITS (after automated simplification):

\[ \ell(x, y) \xrightarrow{x < y} \ell(x + 1, y) \]

\[ x \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \quad y \]

\[ \begin{array}{c}
0 \\
\vdots \\
\end{array} \]
Resulting ITS (after automated simplification):

\[ l(x, y) \xrightarrow{x<y} l(x + 1, y) \]

Automatic termination proof by any termination prover
Overview

original program is memory-safe and terminating

compile

prove termination

prove memory safety

synthesize

Symbolic Execution Graph

Integer Transition System
Experimental Results

- implemented in AProVE
  
  http://aprove.informatik.rwth-aachen.de/
Experimental Results

- implemented in AProVE
  http://aprove.informatik.rwth-aachen.de/

- demo category of SV-COMP 2014 (TACAS):
  https://sv-comp.sosy-lab.org/
Experimental Results

- implemented in AProVE
  [http://aprove.informatik.rwth-aachen.de/](http://aprove.informatik.rwth-aachen.de/)

- demo category of SV-COMP 2014 (TACAS):
  [https://sv-comp.sosy-lab.org/](https://sv-comp.sosy-lab.org/)
  5 participants, most points for AProVE

- C category of termCOMP 2014 (IJCAR):
  AProVE winner

- termination category of SV-COMP 2015 (TACAS):
  6 participants, AProVE winner

- termination category of SV-COMP 2016 (TACAS):
  3 participants, AProVE winner

- SV-COMP 2022 (TACAS):
  3 participants, AProVE second (after UltimateAutomizer)

- termCOMP 2022 (IJCAR):
  2 participants, AProVE winner
Experimental Results

- implemented in AProVE
  
  http://aprove.informatik.rwth-aachen.de/

- demo category of SV-COMP 2014 (TACAS):
  
  https://sv-comp.sosy-lab.org/

  5 participants, most points for AProVE

- C category of termCOMP 2014 (IJCAR):

- termination category of SV-COMP 2015 (TACAS):

- termination category of SV-COMP 2016 (TACAS):

- SV-COMP 2022 (TACAS):

  3 participants, AProVE second

  (after UltimateAutomizer)

- termCOMP 2022 (IJCAR):

  2 participants, AProVE winner
Experimental Results

- implemented in AProVE
  
  http://aprove.informatik.rwth-aachen.de/

- demo category of SV-COMP 2014 (TACAS):
  
  https://sv-comp.sosy-lab.org/
  
  5 participants, most points for AProVE

- C category of termCOMP 2014 (IJCAR):
  
  3 participants, AProVE winner

- termination category of SV-COMP 2015 (TACAS):
  
  6 participants, AProVE winner

- termination category of SV-COMP 2016 (TACAS):
  
  3 participants, AProVE winner

- SV-COMP 2022 (TACAS):
  
  3 participants, AProVE second (after UltimateAutomizer)

- termCOMP 2022 (IJCAR):
  
  2 participants, AProVE winner
Experimental Results

- implemented in AProVE
  http://aprove.informatik.rwth-aachen.de/

- demo category of SV-COMP 2014 (TACAS):
  https://sv-comp.sosy-lab.org/
  5 participants, most points for AProVE

- C category of termCOMP 2014 (IJCAR):
  3 participants, AProVE winner
Experimental Results

- implemented in AProVE
  http://aprove.informatik.rwth-aachen.de/

  5 participants, most points for AProVE

- C category of termCOMP 2014 (IJCAR):
  3 participants, AProVE winner

- termination category of SV-COMP 2015 (TACAS):
  6 participants, AProVE winner

- termination category of SV-COMP 2016 (TACAS):
  3 participants, AProVE winner

...
Experimental Results

- implemented in AProVE
  http://aprove.informatik.rwth-aachen.de/

- demo category of SV-COMP 2014 (TACAS):
  https://sv-comp.sosy-lab.org/
  5 participants, most points for AProVE

- C category of termCOMP 2014 (IJCAR):
  3 participants, AProVE winner

- termination category of SV-COMP 2015 (TACAS):
  6 participants, AProVE winner

- termination category of SV-COMP 2016 (TACAS):
  3 participants, AProVE winner

  
  SV-COMP 2022 (TACAS): 3 participants, AProVE second
  (after UltimateAutomizer)

- termCOMP 2022 (IJCAR): 2 participants, AProVE winner
Extensions

Beyond strlen:

- support malloc + free
Extensions

Beyond strlen:

- support malloc + free
- improved generalisation heuristic, can handle strcpy
Extensions

Beyond strlen:

- support malloc + free
- improved generalisation heuristic, can handle strcpy
- function calls (also recursive)
Beyond strlen:

- support malloc + free
- improved generalisation heuristic, can handle strcpy
- function calls (also recursive)
- soundness proved wrt formal Vellvm semantics from [Zhao et al, *POPL '12*]
Beyond strlen:

- support malloc + free
- improved generalisation heuristic, can handle strcpy
- function calls (also recursive)
- soundness proved wrt formal Vellvm semantics from [Zhao et al, *POPL ’12*
- non-termination analysis [Hensel, Mensendieck, Giesl, *TACAS ’22*]
Beyond `strlen`:

- Support `malloc` + `free`
- Improved generalisation heuristic, can handle `strcpy`
- Function calls (also recursive)
- Soundness proved wrt formal Vellvm semantics from [Zhao et al, *POPL ’12*]
- Non-termination analysis [Hensel, Mensendieck, Giesl, *TACAS ’22*]
- Termination and complexity wrt bitvector semantics (so far: `int = ℤ`) [Hensel et al, *JLAMP ’22*]
Conclusion: Termination of C / LLVM programs

original program is memory-safe and terminating

prove termination

SYMBOLIC EXECUTION GRAPH

prove memory safety

SYNTHESISE

INTEGER TRANSITION SYSTEM

compile
Front-Ends for Haskell and Prolog

**Haskell** [Giesl et al, *TOPLAS ’11*]
- lazy evaluation
- polymorphic types
- higher-order
Front-Ends for Haskell and Prolog

**Haskell** [Giesl et al, *TOPLAS ’11*]
- lazy evaluation
- polymorphic types
- higher-order
⇒ abstract domain: a single term; extract (non-constrained) TRS
Front-Ends for Haskell and Prolog

**Haskell** [Giesl et al, *TOPLAS ’11*]
- lazy evaluation
- polymorphic types
- higher-order

⇒ abstract domain: a single term; extract (non-constrained) TRS

**Prolog** [Schneider-Kamp et al, *TOCL ’09*; Giesl et al, *PPDP ’12*]
- backtracking
- uses unification instead of matching
- extra-logical language features (e.g., cut)
Front-Ends for Haskell and Prolog

**Haskell** [Giesl et al, *TOPLAS ‘11*]

- lazy evaluation
- polymorphic types
- higher-order

$\Rightarrow$ abstract domain: a single term; extract (non-constrained) TRS

**Prolog** [Schneider-Kamp et al, *TOCL ‘09*; Giesl et al, *PPDP ‘12*]

- backtracking
- uses unification instead of matching
- extra-logical language features (e.g., cut)

$\Rightarrow$ abstract domain based on equivalent **linear** Prolog semantics [Ströder et al, *LOPSTR ‘11*], tracks which variables are for ground terms vs arbitrary terms
Conclusion: Termination Analysis for Programs

- Termination proving for TRSs and ITSs driven by SMT solvers
Conclusion: Termination Analysis for Programs

- Termination proving for TRSs and ITSs driven by SMT solvers
- Constrained rewriting: Term rewriting + pre-defined primitive data structures
Conclusion: Termination Analysis for Programs

- Termination proving for TRSs and ITSs driven by SMT solvers
- Constrained rewriting: Term rewriting + pre-defined primitive data structures
- Common theme for analysis of program termination by (constrained) rewriting:
  - handle language specifics in \textbf{front-end}
  - transitions between program states become (constrained) rewrite rules for \textbf{termination back-end}
Conclusion: Termination Analysis for Programs

- Termination proving for TRSs and ITSs driven by SMT solvers
- Constrained rewriting: Term rewriting + pre-defined primitive data structures
- Common theme for analysis of program termination by (constrained) rewriting:
  - handle language specifics in **front-end**
  - transitions between program states become (constrained) rewrite rules for **termination back-end**
- Works across paradigms: Java, C, Haskell, Prolog
Given: Program $P$.

Session 1: Does $P$ terminate at all?

Session 2: How many steps may $P$ take until it terminates?
II.1 Complexity Analysis for Programs on Integers
What Do You Mean by Complexity?

Literature uses many alternative names:
- (Computational/Algorithmic) complexity analysis
- (Computational) cost analysis
- Resource analysis
- Static profiling
- ...

Resource:
- Number of evaluation steps
- Number of network requests
- Peak memory use
- Battery power
- ...

**Given:** Program $P$.

**Task:** Provide upper/lower bounds on the resource use of running $P$ as a function of the input (size) in the worst case
Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage

 Specifications: What guarantees can we make to the API's user?

> "The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking)."

[Documentation](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)

→ Computational cost as a non-functional requirement!

Profiling: Which parts of the code need most runtime as inputs grow larger?

Smart contracts: Bound execution cost (as "gas", i.e., money)

More: see Section 1.1.2 of PhD thesis by Alicia Merayo Corcoba

Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[9x252] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[33x233] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[106x233] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2044x2206] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2061x2206] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2172x2206] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2033x2206] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2033x2194] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2033x2181] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2033x2167] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[4033x4154] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[4033x4141] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[4033x4127] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[4033x4114] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[4033x4100] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[4044x4088] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[4061x4088] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2033x2074] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2033x2061] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2033x2047] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2033x2034] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2024x2025] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2028x2021] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2325x2038] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2305x2021] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2024x2025] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2028x2021] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2109x2021] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---

[2109x2021] Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

---
Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks

  → related DARPA project: *Space/Time Analysis for Cybersecurity*
  
  [https://www.darpa.mil/program/space-time-analysis-for-cybersecurity](https://www.darpa.mil/program/space-time-analysis-for-cybersecurity)

---

### Specifications: What guarantees can we make to the API’s user?

- "The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking)."

  [https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)

  → computational cost as a non-functional requirement!

---

### Profiling:

Which parts of the code need most runtime as inputs grow larger?

### Smart contracts:

- **Bound execution cost (as "gas", i.e., money)**

More: see Section 1.1.2 of PhD thesis by Alicia Merayo Corcoba


---

---
Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks
  → related DARPA project: *Space/Time Analysis for Cybersecurity*
  
  https://www.darpa.mil/program/space-time-analysis-for-cybersecurity

- **Embedded devices**: Bound memory usage
Why Care About Computational Cost, Anyway?

- Mobile devices: Bound energy usage
- Security: Denial of Service attacks
  \[\rightarrow\] related DARPA project: *Space/Time Analysis for Cybersecurity*
  
  https://www.darpa.mil/program/space-time-analysis-for-cybersecurity

- Embedded devices: Bound memory usage
- Specifications: What guarantees can we make to the API’s user?

---

Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks
  → related DARPA project: *Space/Time Analysis for Cybersecurity*
  https://www.darpa.mil/program/space-time-analysis-for-cybersecurity
- **Embedded devices**: Bound memory usage
- **Specifications**: What guarantees can we make to the API’s user?
  “The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires \(O(n)\) time. All of the other operations run in linear time (roughly speaking).”
  https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
  → computational cost as a non-functional requirement!
Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks
  - related DARPA project: *Space/Time Analysis for Cybersecurity*
    - [https://www.darpa.mil/program/space-time-analysis-for-cybersecurity](https://www.darpa.mil/program/space-time-analysis-for-cybersecurity)
- **Embedded devices**: Bound memory usage
- **Specifications**: What guarantees can we make to the API’s user?
  “The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking).”
  - [https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)
  - computational cost as a non-functional requirement!
- **Profiling**:
  Which parts of the code need most runtime as inputs grow larger?
Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks
  - → related DARPA project: *Space/Time Analysis for Cybersecurity*
    https://www.darpa.mil/program/space-time-analysis-for-cybersecurity
- **Embedded devices**: Bound memory usage
- **Specifications**: What guarantees can we make to the API’s user?
  “The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding \( n \) elements requires \( O(n) \) time. All of the other operations run in linear time (roughly speaking).”
  https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
  → computational cost as a non-functional requirement!
- **Profiling**:
  Which parts of the code need most runtime as inputs grow larger?
- **Smart contracts**: Bound execution cost (as “gas”, i.e., money)
Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks
  → related DARPA project: *Space/Time Analysis for Cybersecurity*
  https://www.darpa.mil/program/space-time-analysis-for-cybersecurity
- **Embedded devices**: Bound memory usage
- **Specifications**: What guarantees can we make to the API’s user?
  “The *size*, *isEmpty*, *get*, *set*, *iterator*, and *listIterator* operations run in constant time. The *add* operation runs in amortized constant time, that is, adding *n* elements requires $O(n)$ time. All of the other operations run in linear time (roughly speaking).”
  https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
  → computational cost as a non-functional requirement!

- **Profiling**:
  Which parts of the code need most runtime as inputs grow larger?
- **Smart contracts**: Bound execution cost (as “gas”, i.e., money)
- **More**: see Section 1.1.2 of PhD thesis by Alicia Merayo Corcoba¹

Question: Write a Python function that returns the sum $1 + 2 + \cdots + n$. 

```python
def sum1(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
        i = i + 1
    return r

def sum2(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
    return r

def sum3(n):
    r = 0
    i = 1
    while i <= n:
        j = 0
        while j < i:
            r = r + 1
            j = j + 1
        i = i + 1
    return r

def sum4(n):
    return n*(n+1)//2
```
**Question:** Write a Python function that returns the sum $1 + 2 + \cdots + n$.

```python
def sum1(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
        i = i + 1
    return r
```

Runtime in $O(f(n))$ means:

- the program needs at most about $f(n)$ steps for an input of "size" $n$.
- the runtime is "of order $f(n)$".
**Question:** Write a Python function that returns the sum $1 + 2 + \cdots + n$.

```python
def sum1(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
        i = i + 1
    return r
```

The runtime in $O(n)$ means: the program needs at most about $f(n)$ steps for an input of "size" $n$, the runtime is "of order $f(n)$".
**Question:** Write a Python function that returns the sum $1 + 2 + \cdots + n$.

```python
def sum1(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
    return r
```

**runtime in $O(f(n))$ means:**

- the program needs at most about $f(n)$ steps for an input of “size” $n$
- the runtime is “of order $f(n)$”
Question: Write a Python function that returns the sum $1 + 2 + \cdots + n$.

```python
def sum1(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
        i = i + 1
    return r

def sum2(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
    return r

def sum3(n):
    r = 0
    i = 1
    while i <= n:
        j = 0
        while j < i:
            r = r + 1
            j = j + 1
        i = i + 1
    return r

def sum4(n):
    return n*(n+1)//2
```

Runtime in $O(f(n))$ means:

- the program needs at most about $f(n)$ steps for an input of "size" $n$
- the runtime is "of order $f(n)$"
Show Me Some Examples!

**Question:** Write a Python function that returns the sum $1 + 2 + \cdots + n$.

```python
def sum1(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
        i = i + 1
    return r

def sum2(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
        return r

def sum3(n):
    r = 0
    i = 1
    while i <= n:
        j = 0
        while j < i:
            r = r + 1
            j = j + 1
        i = i + 1
    return r

def sum4(n):
    return n*(n+1)//2
```

**Runtime in $O(f(n))$:**

- The program needs at most **about $f(n)$ steps** for an input of “size” $n$.
- The runtime is “of order $f(n)$”
**Question:** Write a Python function that returns the sum $1 + 2 + \cdots + n$.

```
def sum1(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
        i = i + 1
    return r

def sum2(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
        return r

def sum3(n):
    r = 0
    i = 1
    while i <= n:
        j = 0
        while j < i:
            r = r + 1
            j = j + 1
        i = i + 1
    return r

def sum4(n):
    return n*(n+1)//2
```

Runtime in $O(f(n))$ means:

- the program needs at most about $f(n)$ steps for an input of “size” $n$
- the runtime is “of order $f(n)$”
**Question:** Write a Python function that returns the sum $1 + 2 + \cdots + n$.

```python
def sum1(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
        i = i + 1
    return r

def sum2(n):
    r = 0
    i = 1
    while i <= n:
        r = r + i
    return r

def sum3(n):
    r = 0
    i = 1
    while i <= n:
        j = 0
        while j < i:
            r = r + 1
            j = j + 1
        i = i + 1
    return r

def sum4(n):
    return n*(n+1)//2
```

**runtime in $O(f(n))$ means:**

- the program needs at most **about $f(n)$ steps** for an input of “size” $n$
- the **runtime** is “of order $f(n)$”
**Question:** Write a Python function that returns the sum $1 + 2 + \cdots + n$.

<table>
<thead>
<tr>
<th>def sum1(n):</th>
<th>def sum2(n):</th>
<th>def sum3(n):</th>
</tr>
</thead>
<tbody>
<tr>
<td>r = 0</td>
<td>r = 0</td>
<td>r = 0</td>
</tr>
<tr>
<td>i = 1</td>
<td>i = 1</td>
<td>i = 1</td>
</tr>
<tr>
<td>while i &lt;= n:</td>
<td>while i &lt;= n:</td>
<td>while i &lt;= n:</td>
</tr>
<tr>
<td>r = r + i</td>
<td>r = r + i</td>
<td>r = r + 1</td>
</tr>
<tr>
<td>i = i + 1</td>
<td></td>
<td>j = j + 1</td>
</tr>
<tr>
<td>return r</td>
<td>return r</td>
<td>i = i + 1</td>
</tr>
</tbody>
</table>

**runtime in $O(f(n))$ means:**

- the program needs at most about $f(n)$ steps for an input of “size” $n$
- the runtime is “of order $f(n)$”

<table>
<thead>
<tr>
<th>def sum4(n):</th>
</tr>
</thead>
<tbody>
<tr>
<td>return n*(n+1)//2</td>
</tr>
</tbody>
</table>
**Question:** Write a Python function that returns the sum $1 + 2 + \cdots + n$.

<table>
<thead>
<tr>
<th>def sum1(n): r = 0</th>
<th>def sum2(n): r = 0</th>
<th>def sum3(n): r = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 1</td>
<td>i = 1</td>
<td>i = 1</td>
</tr>
<tr>
<td>while i &lt;= n:</td>
<td>while i &lt;= n:</td>
<td>while i &lt;= n:</td>
</tr>
<tr>
<td>r = r + i</td>
<td>r = r + i</td>
<td>j = 0</td>
</tr>
<tr>
<td>i = i + 1</td>
<td>return r</td>
<td>while j &lt; i:</td>
</tr>
<tr>
<td>return r</td>
<td></td>
<td>r = r + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>j = j + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i = i + 1</td>
</tr>
</tbody>
</table>

**runtime in $O(f(n))$ means:**

- the program needs at most about $f(n)$ steps for an input of “size” $n$
- the runtime is “of order $f(n)$”

<table>
<thead>
<tr>
<th>def sum4(n):</th>
<th>89/173</th>
</tr>
</thead>
<tbody>
<tr>
<td>return n*(n+1)//2</td>
<td></td>
</tr>
</tbody>
</table>
Is There a Tool that Finds such Bounds Automatically?

- Fully automatic open-source tool KoAT: https://github.com/s-falke/kittel-koat

Journal paper about the automated analysis implemented in KoAT:

Experiments: http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity-Journal
Is There a Tool that Finds such Bounds Automatically?

- Fully automatic open-source tool KoAT:
  
  https://github.com/s-falke/kittel-koat

- Journal paper about the automated analysis implemented in KoAT:
  
  M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, J. Giesl, *Analyzing runtime and size complexity of integer programs* 
Is There a Tool that Finds such Bounds Automatically?

- Fully automatic open-source tool KoAT:
  
  https://github.com/s-falke/kittel-koat

- Journal paper about the automated analysis implemented in KoAT:
  M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, J. Giesl, *Analyzing runtime and size complexity of integer programs*

- Experiments:
  
  http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity-Journal
How Can We Make the Computer Do the Work for Us?

Idea: **Countdown**.
For each loop find a **ranking function** $f$ on the variables:
expression that gets smaller each time round the loop, but never reaches 0.
How Can We Make the Computer Do the Work for Us?

Idea: **Countdown.**
For each loop find a *ranking function* $f$ on the variables:
expression that gets smaller each time round the loop, but never reaches 0.
⇒ Gives us a bound on the number of times we go through the loop
Idea: **Countdown.**

For each loop find a **ranking function** \( f \) on the variables: expression that gets smaller each time round the loop, but never reaches 0.

⇒ Gives us a bound on the **number of times** we go through the loop

Termination analysis tools find ranking functions automatically!
How Can We Make the Computer Do the Work for Us?

Idea: **Countdown.**
For each loop find a *ranking function* $f$ on the variables: expression that gets smaller each time round the loop, but never reaches 0. ⇒ Gives us a bound on the number of times we go through the loop

Termination analysis tools find ranking functions automatically!

```python
def twoLoops1(x, z):
    while x > 0:
        x = x - 1
    while z > 0:
        z = z - 1
```

Loop 1: ranking function $x$
Loop 2: ranking function $z$
⇒ runtime in $O(x + z)$

```python
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1
```

Loop 1: ranking function $x$
Loop 2: ranking function $z$
⇒ runtime in $O(\ldots)$ oops.

Best runtime bound: $O(x^2 + z)$
Idea: **Countdown.**

For each loop find a **ranking function** \( f \) on the variables:

expression that gets smaller each time round the loop, but never reaches 0.

\[ \Rightarrow \] Gives us a bound on the **number of times** we go through the loop

Termination analysis tools find ranking functions automatically!

```python
def twoLoops1(x, z):
    while x > 0:
        x = x - 1

    while z > 0:
        z = z - 1
```

Loop 1: ranking function \( x \)
Idea: **Countdown.**
For each loop find a ranking function $f$ on the variables: expression that gets smaller each time round the loop, but never reaches 0.  
⇒ Gives us a bound on the number of times we go through the loop

Termination analysis tools find ranking functions automatically!

```python
def twoLoops1(x, z):
    while x > 0:
        x = x - 1

    while z > 0:
        z = z - 1
```

Loop 1: ranking function $x$
Loop 2: ranking function $z$
How Can We Make the Computer Do the Work for Us?

Idea: **Countdown**.
For each loop find a **ranking function** $f$ on the variables:
expression that gets smaller each time round the loop, but never reaches 0.
$\Rightarrow$ Gives us a bound on the **number of times** we go through the loop
Termination analysis tools find ranking functions automatically!

```python
def twoLoops1(x, z):
    while x > 0:
        x = x - 1
    while z > 0:
        z = z - 1

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1
```

Loop 1: ranking function $x$
Loop 2: ranking function $z$
$\Rightarrow$ runtime in $O(x + z)$
Idea: **Countdown**.
For each loop find a **ranking function** $f$ on the variables: expression that gets smaller each time round the loop, but never reaches 0.
$\implies$ Gives us a bound on the number of times we go through the loop
Termination analysis tools find ranking functions automatically!

```
def twoLoops1(x, z):
    while x > 0:
        x = x - 1
    while z > 0:
        z = z - 1

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1
```

Loop 1: ranking function $x$
Loop 2: ranking function $z$
$\implies$ runtime in $\mathcal{O}(x + z)$
Idea: **Countdown.**
For each loop find a **ranking function** $f$ on the variables: expression that gets smaller each time round the loop, but never reaches 0. 
⇒ Gives us a bound on the **number of times** we go through the loop 
Termination analysis tools find ranking functions automatically!

```python
def twoLoops1(x, z):
    while x > 0:
        x = x - 1
    while z > 0:
        z = z - 1

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1
```

Loop 1: ranking function $x$
Loop 2: ranking function $z$
⇒ runtime in $O(x + z)$
### Idea: **Countdown.**

For each loop find a **ranking function** \( f \) on the variables: expression that gets smaller each time round the loop, but never reaches 0.

⇒ Gives us a bound on the **number of times** we go through the loop

Termination analysis tools find ranking functions automatically!

```python
def twoLoops1(x, z):
    while x > 0:
        x = x - 1
    while z > 0:
        z = z - 1

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1
```

Loop 1: ranking function \( x \)
Loop 2: ranking function \( z \)

⇒ runtime in \( \mathcal{O}(x + z) \)
How Can We Make the Computer Do the Work for Us?

Idea: **Countdown.**
For each loop find a **ranking function** $f$ on the variables:
expression that gets smaller each time round the loop, but never reaches 0.
⇒ Gives us a bound on the number of times we go through the loop
Termination analysis tools find ranking functions automatically!

```python
def twoLoops1(x, z):
    while x > 0:
        x = x - 1
    while z > 0:
        z = z - 1

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1
```

Loop 1: ranking function $x$
Loop 2: ranking function $z$
⇒ runtime in $O(x + z)$

⇒ runtime in $O(x^2 + z)$
How Can We Make the Computer Do the Work for Us?

Idea: **Countdown.**
For each loop find a **ranking function** $f$ on the variables: expression that gets smaller each time round the loop, but never reaches 0.
⇒ Gives us a bound on the **number of times** we go through the loop
Termination analysis tools find ranking functions automatically!

```python
def twoLoops1(x, z):
    while x > 0:
        x = x - 1
    while z > 0:
        z = z - 1

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1
```

Loop 1: ranking function $x$
Loop 2: ranking function $z$
⇒ runtime in $\mathcal{O}(x + z)$

Loop 1: ranking function $x$
Loop 2: ranking function $z$
⇒ runtime in ... oops.
Idea: **Countdown.**
For each loop find a **ranking function** $f$ on the variables: expression that gets smaller each time round the loop, but never reaches 0.
⇒ Gives us a bound on the **number of times** we go through the loop
Termination analysis tools find ranking functions automatically!

```python
def twoLoops1(x, z):
    while x > 0:
        x = x - 1
    while z > 0:
        z = z - 1

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1
```

Loop 1: ranking function $x$
Loop 2: ranking function $z$
⇒ runtime in $O(x + z)$

⇒ runtime in ... oops.

Best runtime bound: $O(x^2 + z)$
How Can we Fix our Approach?

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$
How Can we Fix our Approach?

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$

Problem:
Loop 1 writes to $z$. In Loop 2, $z$ is much larger than its initial value $z_0$!
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1

Loop 1: ranking function \( f_1(x, z) = x \)
Loop 2: ranking function \( f_2(x, z) = z \)

Problem:
Loop 1 writes to \( z \). In Loop 2, \( z \) is much larger than its initial value \( z_0 \)!
Now an oracle tells us:

Now an oracle tells us:

\( z \) is at most \( z_0 + x \cdot 2^{0} \), and \( x = 0 \).

\[ f_2(0, z_0 + x \cdot 2^{0}) = z_0 + x \cdot 2^{0} \]
gives runtime bound for Loop 2: \( O(z_0 + x \cdot 2^{0}) \)

Data size influences runtime.
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$

Problem:
Loop 1 writes to $z$. In Loop 2, $z$ is much larger than its initial value $z_0$!

Now an oracle tells us:

*Oh, when you reach Loop 2, $z$ is at most $z_0 + x_0^2$, and $x$ is 0.*
How Can we Fix our Approach?

```python
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1
```

| Loop 1: ranking function \( f_1(x, z) = x \) |
| Loop 2: ranking function \( f_2(x, z) = z \) |

**Problem:**
Loop 1 writes to \( z \). In Loop 2, \( z \) is much larger than its initial value \( z_0 \)!

Now an oracle tells us:

*Oh, when you reach Loop 2, \( z \) is at most \( z_0 + x_0^2 \), and \( x \) is 0.*

So:

1. we can make at most \( f_2(x, z) = z \) steps in Loop 2
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1

Loop 1: ranking function \( f_1(x, z) = x \)
Loop 2: ranking function \( f_2(x, z) = z \)

Problem:
Loop 1 writes to \( z \). In Loop 2, \( z \) is much larger than its initial value \( z_0 \)!
Now an oracle tells us:

Oh, when you reach Loop 2, \( z \) is at most \( z_0 + x^2_0 \), and \( x \) is 0.

So:
1. we can make at most \( f_2(x, z) = z \) steps in Loop 2
2. when we enter Loop 2, we know \( z \leq z_0 + x^2_0 \) and \( x = 0 \)
How Can we Fix our Approach?

def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$

Problem:
Loop 1 writes to $z$. In Loop 2, $z$ is much larger than its initial value $z_0$!

Now an oracle tells us:

Oh, when you reach Loop 2, $z$ is at most $z_0 + x_0^2$, and $x$ is 0.

So:

1. we can make at most $f_2(x, z) = z$ steps in Loop 2
2. when we enter Loop 2, we know $z \leq z_0 + x_0^2$ and $x = 0$

$\Rightarrow f_2(0, z_0 + x_0^2) = z_0 + x_0^2$
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$

Problem:
Loop 1 writes to $z$. In Loop 2, $z$ is much larger than its initial value $z_0$!
Now an oracle tells us:

Oh, when you reach Loop 2, $z$ is at most $z_0 + x_0^2$, and $x$ is 0.

So:

1. we can make at most $f_2(x, z) = z$ steps in Loop 2
2. when we enter Loop 2, we know $z \leq z_0 + x_0^2$ and $x = 0$

$\Rightarrow f_2(0, z_0 + x_0^2) = z_0 + x_0^2$ gives runtime bound for Loop 2: $O(z_0 + x_0^2)$
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$

**Problem:**
Loop 1 writes to $z$. In Loop 2, $z$ is much larger than its initial value $z_0$!

Now an oracle tells us:

*Oh, when you reach Loop 2, $z$ is at most $z_0 + x_0^2$, and $x$ is 0.*

So:

1. we can make at most $f_2(x, z) = z$ steps in Loop 2
2. when we enter Loop 2, we know $z \leq z_0 + x_0^2$ and $x = 0$

$\Rightarrow f_2(0, z_0 + x_0^2) = z_0 + x_0^2$ gives runtime bound for Loop 2: $O(z_0 + x_0^2)$

**Data size influences runtime.**
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    # (*)
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    # (*)
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$

Wanted: automatic oracle to tell how big $z$ can be at $(*).$
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    # (*)
    while z > 0:
        z = z - 1

Loop 1: ranking function \( f_1(x, z) = x \)

Loop 2: ranking function \( f_2(x, z) = z \)

**Wanted:** automatic oracle to tell how big \( z \) can be at \((*)\).

**We know:**

1. each time round Loop 1, \( x \) goes down by 1, from \( x_0 \) until 0
How Can We Build such an Oracle for Size Bounds?

```python
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    # (*)
    while z > 0:
        z = z - 1

Loop 1: ranking function \( f_1(x, z) = x \)

Loop 2: ranking function \( f_2(x, z) = z \)
```

**Wanted:** automatic oracle to tell how big \( z \) can be at \((*)\).

We know:

1. each time round Loop 1, \( x \) goes down by 1, from \( x_0 \) until 0
   \[ \implies \text{in Loop 1: } x \leq x_0 \]
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    # (*)
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$
Loop 2: ranking function $f_2(x, z) = z$

**Wanted:** automatic oracle to tell how big $z$ can be at (*).

We know:

1. each time round Loop 1, $x$ goes down by 1, from $x_0$ until 0
   ⇒ in Loop 1: $x \leq x_0$

2. each time round Loop 1, $z$ goes up by $x$ ($\leq x_0$)
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    # (*)
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$

**Wanted:** automatic oracle to tell how big $z$ can be at (*).

We know:

1. each time round Loop 1, $x$ goes down by 1, from $x_0$ until 0
   \[ \Rightarrow \text{in Loop 1: } x \leq x_0 \]
2. each time round Loop 1, $z$ goes up by $x$ ($\leq x_0$)
3. we run through Loop 1 at most $f_1(x_0, z_0) = x_0$ times
How Can We Build such an Oracle for Size Bounds?

```python
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    # (*)
while z > 0:
    z = z - 1
```

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$

Wanted: automatic oracle to tell how big $z$ can be at (*)

We know:

1. each time round Loop 1, $x$ goes down by 1, from $x_0$ until 0  
   $\Rightarrow$ in Loop 1: $x \leq x_0$

2. each time round Loop 1, $z$ goes up by $x$ ($\leq x_0$)

3. we run through Loop 1 at most $f_1(x_0, z_0) = x_0$ times

$\Rightarrow$ at (*), $z$ will be at most $z_0 + x_0 \cdot x_0 = z_0 + x_0^2$!
def twoLoops2(x, z):
    while x > 0:
        x = x - 1
        z = z + x
    # (*)
    while z > 0:
        z = z - 1

Loop 1: ranking function $f_1(x, z) = x$

Loop 2: ranking function $f_2(x, z) = z$

**Wanted:** automatic oracle to tell how big $z$ can be at (*)

We know:

1. each time round Loop 1, $x$ goes down by 1, from $x_0$ until 0
   $\Rightarrow$ in Loop 1: $x \leq x_0$

2. each time round Loop 1, $z$ goes up by $x$ ($\leq x_0$)

3. we run through Loop 1 at most $f_1(x_0, z_0) = x_0$ times

$\Rightarrow$ at (*), $z$ will be at most $z_0 + x_0 \cdot x_0 = z_0 + x_0^2$!

**Runtime influences data size.**
Example (List program)

Input: List x

\( \ell_0: \) List y = null

\( \ell_1: \) while x ≠ null do

\hspace{1em} y = new List(x.val, y)

\hspace{1em} x = x.next

\hspace{2em} done

\hspace{1em} List z = y

\( \ell_2: \) while z ≠ null do

\hspace{1em} List u = z.next

\( \ell_3: \) while u ≠ null do

\hspace{2em} z.val += u.val

\hspace{2em} u = u.next

\hspace{2em} done

\hspace{1em} z = z.next

\hspace{2em} done

Example (Integer abstraction)

Input: int x

\( \ell_0: \) int y = 0

\( \ell_1: \) while x > 0 do

\hspace{1em} y = y + 1

\hspace{1em} x = x - 1

\hspace{2em} done

\hspace{1em} int z = y

\( \ell_2: \) while z > 0 do

\hspace{1em} int u = z - 1

\( \ell_3: \) while u > 0 do

\hspace{2em} skip

\hspace{2em} u = u - 1

\hspace{2em} done

\hspace{1em} z = z - 1

\hspace{2em} done
Example (List program)

Input: List $x$

$l_0$: List $y = \text{null}$

$l_1$: while $x \neq \text{null}$ do
  $y = \text{new List}(x\.val, y)$
  $x = x\.next$

done

List $z = y$

$l_2$: while $z \neq \text{null}$ do
  List $u = z\.next$

$l_3$: while $u \neq \text{null}$ do
  $z\.val += u\.val$
  $u = u\.next$

done

$z = z\.next$

done

---

$x = [3, 1, 5] \bowtie$

$y = [5, 1, 3] \bowtie$

$z = [5 + 1 + 3, 1 + 3, 3]$
Example (List program)

Input: List x

ℓ₀: List y = null

ℓ₁: while x ≠ null do
    y = new List(x.val, y)
    x = x.next
    done

List z = y

ℓ₂: while z ≠ null do
    List u = z.next

ℓ₃: while u ≠ null do
    z.val += u.val
    u = u.next
    done

z = z.next

done

Example (Integer abstraction)

Input: int x

ℓ₀: int y = 0

ℓ₁: while x > 0 do
    y = y + 1
    x = x - 1
    done

int z = y

ℓ₂: while z > 0 do
    int u = z - 1

ℓ₃: while u > 0 do
    skip
    u = u - 1
    done

z = z - 1

done
Control flow graph:

Example (Integer abstraction)

Input: int x

\[ \ell_0: \text{int } y = 0 \]

\[ \ell_1: \text{while } x > 0 \text{ do} \]

\[ y = y + 1 \]
\[ x = x - 1 \]

\[ \text{done} \]

\[ \text{int } z = y \]

\[ \ell_2: \text{while } z > 0 \text{ do} \]

\[ \text{int } u = z - 1 \]

\[ \ell_3: \text{while } u > 0 \text{ do} \]

\[ \text{skip} \]
\[ u = u - 1 \]

\[ \text{done} \]

\[ z = z - 1 \]

\[ \text{done} \]
What Does the Problem Look Like?

- **Programs** as Integer Transition Systems:
  - Locations $L$: $l_0$ start
  - Variables $V$
  - Transitions $T$: Formula over pre- $(x, y, \ldots)$, post-variables $(x', y', \ldots)$

  e.g., $t_5 = (l_3, u \leq 0 \land z > 0 \land z' = z - 1, l_2)$

  for $l_3(u, x, y, z) \rightarrow l_2(u', x', y', z')$ [$u \leq 0 \land z > 0 \land z' = z - 1 \land u' = u \land x' = x \land y' = y$]
What Do the Problem and the Solution Look Like?

- **Programs** as Integer Transition Systems:
  - Locations $L$: $l_0$ start
  - Variables $V$
  - Transitions $T$: Formula over pre- $(x, y, \ldots)$, post-variables $(x', y', \ldots)$
    
    e.g., $t_5 = (l_3, u \leq 0 \land z > 0 \land z' = z - 1, l_2)$
    
    for $l_3(u, x, y, z) \rightarrow l_2(u', x', y', z')$ $[u \leq 0 \land z > 0 \land z' = z - 1 \land u' = u \land x' = x \land y' = y]$  

- **Runtime complexity**:
  - $R(t)$ upper bound on number of uses of $t \in T$ in execution
  - $R(t)$ monotonic function in $V$, e.g. $|x|^2 + |y| + 1$
  - $R(t)$ expresses bound in input values
What Do the Problem and the Solution Look Like?

- **Programs** as Integer Transition Systems:
  - Locations $\mathcal{L}$: $l_0$ start
  - Variables $\mathcal{V}$
  - Transitions $\mathcal{T}$: Formula over pre- $(x, y, \ldots)$, post-variables $(x', y', \ldots)$

  e.g., $t_5 = (l_3, u \leq 0 \land z > 0 \land z' = z - 1, l_2)$
  for $l_3(u, x, y, z) \rightarrow l_2(u', x', y', z') \ [u \leq 0 \land z > 0 \land z' = z - 1 \land u' = u \land x' = x \land y' = y]$

- **Runtime complexity**:
  - $R(t)$ upper bound on number of uses of $t \in \mathcal{T}$ in execution
  - $R(t)$ monotonic function in $\mathcal{V}$, e.g. $|x|^2 + |y| + 1$
  - $R(t)$ expresses bound in input values

- **Size complexity**:
  - $S(t, v')$ upper bound on size of $v \in \mathcal{V}$ after using $t \in \mathcal{T}$
  - $S(t, v')$ monotonic function in $\mathcal{V}$
  - $S(t, v')$ expresses bound in input values
And in the Example?

\[ \ell_0 \quad t_0: \quad y = 0 \]

\[ \ell_1 \quad t_1: \quad \text{if}(x > 0) \]
\[ y = y + 1 \]
\[ x = x - 1 \]

\[ \ell_2 \quad t_2: \quad \text{if}(x \leq 0) \]
\[ z = y \]

\[ \ell_2 \quad t_3: \quad \text{if}(z > 0) \]
\[ u = z - 1 \]

\[ \ell_3 \quad t_4: \quad \text{if}(u > 0) \]
\[ \text{if}(z > 0) \]
\[ u = u - 1 \]

Overall runtime is bounded by

\[ R(t_1) + \ldots + R(t_5) = 3 + 4 \cdot |x| + |x|^2. \]
And in the Example?

Goal: find complexity bounds w.r.t. the sizes of the input variables

\[ t_0: \ y = 0 \]
\[ t_1: \ \text{if}(x > 0) \]
\[ y = y + 1 \]
\[ x = x - 1 \]
\[ t_2: \ \text{if}(x \leq 0) \]
\[ z = y \]

\[ t_3: \ \text{if}(z > 0) \]
\[ u = z - 1 \]

\[ t_4: \ \text{if}(u > 0) \]
\[ \text{if}(z > 0) \]
\[ u = u - 1 \]

\[ t_5: \ \text{if}(u \leq 0) \]
\[ \text{if}(z > 0) \]
\[ z = z - 1 \]
And in the Example?

Goal: find complexity bounds w.r.t. the sizes of the input variables

- Runtime bound function $\mathcal{R}(t)$:
  bound on number of times that transition $t$ occurs in executions
  
  e.g., $\mathcal{R}(t_1) = |x|$,  
  $\mathcal{R}(t_4) = |x| + |x|^2$
And in the Example?

Goal: find complexity bounds w.r.t. the \textit{sizes} of the input variables

- **Runtime bound function** $\mathcal{R}(t)$: bound on number of times that transition $t$ occurs in executions
  - e.g., $\mathcal{R}(t_1) = |x|$, $\mathcal{R}(t_4) = |x| + |x|^2$

- **Size bound function** $\mathcal{S}(t, v')$: bound on $|v|$ after using transition $t$ in program executions
  - e.g. $\mathcal{S}(t_1, y') = |x|$
And in the Example?

**Goal:** find complexity bounds w.r.t. the sizes of the input variables

- **Runtime bound function** $\mathcal{R}(t)$:
  bound on number of times that transition $t$ occurs in executions
  
  e.g., $\mathcal{R}(t_1) = |x|$, 
  \[ \mathcal{R}(t_4) = |x| + |x|^2 \]

- **Size bound function** $S(t, v')$:
  bound on $|v|$ after using transition $t$ in program executions
  
  e.g. $S(t_1, y') = |x|$

Overall runtime is bounded by $\mathcal{R}(t_1) + \ldots + \mathcal{R}(t_5) = 3 + 4 \cdot |x| + |x|^2$. 
How Do You Know?
Runtime Bounds I

\[ t_0: \quad y = 0 \]

\[ t_1: \quad \text{if} (x > 0) \]
\[ y = y + 1 \]
\[ x = x - 1 \]

\[ t_2: \quad \text{if} (x \leq 0) \]
\[ z = y \]

\[ t_3: \quad \text{if} (z > 0) \]
\[ u = z - 1 \]

\[ t_4: \quad \text{if} (u > 0) \]
\[ \text{if} (z > 0) \]
\[ u = u - 1 \]
Polynomial ranking function (PRF): \( P : \mathcal{L} \rightarrow \mathbb{Z}[\mathcal{V}] \) with

1. **no increase**
   No transition increases

2. **decrease**
   At least one decreases

3. **bounded**
   Bounded from below by 1

---

Runtime Bounds I (PRFs)

\[ t_0 : \ y = 0 \]

\[ t_1 : \ 
   \text{if}(x > 0) \\
   y = y + 1 \\
   x = x - 1 \]

\[ t_2 : \ 
   \text{if}(x \leq 0) \\
   z = y \]

\[ t_3 : \ 
   \text{if}(z > 0) \\
   u = z - 1 \]

\[ t_4 : \ 
   \text{if}(u > 0) \\
   \text{if}(z > 0) \\
   u = u - 1 \]
Runtime Bounds I (PRFs)

Polynomial ranking function (PRF): 
\( \mathcal{P} : \mathcal{L} \to \mathbb{Z}[\mathcal{V}] \) with

1. no increase
   No transition increases
2. decrease
   At least one decreases
3. bounded
   Bounded from below by 1

Example (PRF I)
\[ \mathcal{P}_1(\ell) = x \quad \text{for all } \ell \in \mathcal{L} \]
Runtime Bounds I (PRFs)

Polynomial ranking function (PRF): \( P : \mathcal{L} \rightarrow \mathbb{Z}[\mathcal{V}] \) with

1. **no increase**
   
   No transition increases

2. **decrease**
   
   At least one decreases

3. **bounded**
   
   Bounded from below by 1

**Example (PRF I)**

\[ P_1(\ell) = x \quad \text{for all } \ell \in \mathcal{L} \]

no increase on any transition
t1 decreases, bounded
Polynomial ranking function (PRF):
\[ \mathcal{P} : \mathcal{L} \rightarrow \mathbb{Z}[\mathcal{V}] \] with

1. **no increase**
   - No transition increases

2. **decrease**
   - At least one decreases

3. **bounded**
   - Bounded from below by 1

**Key idea:** decreasing \( t \) used at most \( \mathcal{P}(\ell_0) \) times
Polynomial ranking function (PRF):
\( P : \mathcal{L} \rightarrow \mathbb{Z}[V] \) with

1. no increase
   - No transition increases

2. decrease
   - At least one decreases

3. bounded
   - Bounded from below by 1

Key idea: decreasing \( t \) used at most \( \mathcal{P}(\ell_0) \) times

\[ \mathcal{R}(t) \leq [\mathcal{P}(\ell_0)] \]

[ \(-\) \equiv “make monotonic (on \( \mathbb{N} \)”]
Polynomial ranking function (PRF): \( P : \mathcal{L} \to \mathbb{Z}[\mathcal{V}] \) with

1. **no increase**
   - No transition increases

2. **decrease**
   - At least one decreases

3. **bounded**
   - Bounded from below by 1

**Key idea:** decreasing \( t \) used at most \( \mathcal{P}(\ell_0) \) times

\[ \rightarrow \mathcal{R}(t) \leq [\mathcal{P}(\ell_0)] \]

\([−]\) ≡ “make monotonic (on \( \mathbb{N} \))”
Runtime Bounds I (PRFs for Complexity)

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]

\[ t_0: \quad y = 0 \]

\[ t_1: \quad \text{if } (x > 0) \]
\[ \quad y = y + 1 \]
\[ \quad x = x - 1 \]

\[ t_2: \quad \text{if } (x \leq 0) \]
\[ \quad z = y \]

\[ t_3: \quad \text{if } (z > 0) \]
\[ \quad u = z - 1 \]

\[ t_4: \quad \text{if } (u > 0) \]
\[ \quad \text{if } (z > 0) \]
\[ \quad u = u - 1 \]

Polynomial ranking function (PRF):
\[ P: \mathcal{L} \to \mathbb{Z}[\mathcal{V}] \] with

1. **no increase**
   - No transition increases
2. **decrease**
   - At least one decreases
3. **bounded**
   - Bounded from below by 1

**Example (PRF II)**

\[ P_2(\ell_0) = 1 \]
\[ P_2(\ell) = 0 \quad \text{for all } \ell \in \mathcal{L} \setminus \{\ell_0\} \]

**no increase** on any transition
\[ t_0 \] decreases, bounded
Polynomial ranking function (PRF):

\[ P : \mathcal{L} \to \mathbb{Z}[\mathcal{V}] \] with

1. no increase
   - No transition increases
2. decrease
   - At least one decreases
3. bounded
   - Bounded from below by 1

Example (PRF III)

\[ P_3(l) = 1 \] for all \( l \in \{l_0, l_1\} \)
\[ P_3(l) = 0 \] for all \( l \in \{l_2, l_3\} \)

no increase on any transition

\( t_2 \) decreases, bounded
Size Bounds

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]

- \( t_0: \ y = 0 \)
- \( t_1: \text{if}(x > 0)\)
  - \( y = y + 1 \)
  - \( x = x - 1 \)
- \( t_2: \text{if}(x \leq 0)\)
  - \( z = y \)

Second loop depends on \( z \)

- \( t_3: \text{if}(z > 0)\)
  - \( u = z - 1 \)
- \( t_4: \text{if}(u > 0)\)
  - \( \text{if}(z > 0)\)
    - \( u = u - 1 \)
- \( t_5: \text{if}(u \leq 0)\)
  - \( \text{if}(z > 0)\)
    - \( z = z - 1 \)
Size Bounds

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]

- \( t_0 \): \( y = 0 \)
- \( t_1 \): \( \text{if} (x > 0) \)
  \[ y = y + 1 \]
  \[ x = x - 1 \]
- \( t_2 \): \( \text{if} (x \leq 0) \)
  \[ z = y \]
  \[ S(t_2, z') \]

Second loop depends on \( z \)

- \( t_3 \): \( \text{if} (z > 0) \)
  \[ u = z - 1 \]
- \( t_4 \): \( \text{if} (u > 0) \)
  \[ \text{if} (z > 0) \]
  \[ u = u - 1 \]
Size Bounds

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]

\[ t_0: \quad y = 0 \]

\[ t_1: \quad \text{if}(x > 0) \]
\[ \quad y = y + 1 \]
\[ \quad x = x - 1 \]

\[ t_2: \quad \text{if}(x \leq 0) \]
\[ \quad z = y \]

Second loop depends on \( z \)
\[ \leftrightarrow \] Compute \( S(t_2, z') \)
\[ \ldots \text{which depends on } y \text{ after } t_0, t_1 \]

\[ t_5: \quad \text{if}(u \leq 0) \]
\[ \quad \text{if}(z > 0) \]
\[ \quad z = z - 1 \]

\[ t_3: \quad \text{if}(z > 0) \]
\[ \quad u = z - 1 \]

\[ t_4: \quad \text{if}(u > 0) \]
\[ \quad \text{if}(z > 0) \]
\[ \quad u = u - 1 \]
\[
\mathcal{R}(t_0) = 1 \\
\mathcal{R}(t_1) = |x| \\
\mathcal{R}(t_2) = 1
\]

\[t_0: \quad y = 0\]

\[t_1: \quad \text{if}(x > 0) \quad y = y + 1 \quad x = x - 1\]

\[t_2: \quad \text{if}(x \leq 0) \quad z = y\]

\[t_3: \quad \text{if}(u \leq 0) \quad \text{if}(z > 0) \quad z = z - 1 \quad \text{if}(z > 0) \quad u = z - 1\]

\[t_4: \quad \text{if}(u > 0) \quad \text{if}(z > 0) \quad u = u - 1\]
Size Bounds: Local

\[ \mathcal{R}(t_0) = 1 \]
\[ \mathcal{R}(t_1) = |x| \]
\[ \mathcal{R}(t_2) = 1 \]

\( t_0: \ y = 0 \)
\( t_1: \ if(x > 0) \)
\[ y = y + 1 \]
\[ x = x - 1 \]
\( t_2: \ if(x \leq 0) \)
\[ z = y \]

Result Variable Graph:
- Nodes \( |t, v'| \), labels \( S_l(t, v') \)
- Change of \( v \) in one use of \( t \):
  \[ t \implies S_l(t, v')(\mathcal{V}) \geq v' \]

0 ≥ \( |t_0, y'| \)
$R(t_0) = 1$
$R(t_1) = |x|$
$R(t_2) = 1$

$t_0$: $y = 0$

$t_1$: if $(x > 0)$
- $y = y + 1$
- $x = x - 1$

$t_2$: if $(x \leq 0)$
- $z = y$

$t_3$: if $(z > 0)$
- $u = z - 1$

$t_4$: if $(u > 0)$
- if $(z > 0)$
- $u = u - 1$

$t_5$: if $(u \leq 0)$

Result Variable Graph:

- Nodes $|t, v'|$, labels $S_l(t, v')$
- Change of $v$ in one use of $t$:

\[ t \implies S_l(t, v')(V) \geq v' \]
\( R(t_0) = 1 \)
\( R(t_1) = |x| \)
\( R(t_2) = 1 \)

- \( t_0: \ y = 0 \)
- \( t_1: \ \text{if}(x > 0) \)
  \( y = y + 1 \)
  \( x = x - 1 \)
- \( t_2: \ \text{if}(x \leq 0) \)
  \( z = y \)
- \( t_3: \ \text{if}(z > 0) \)
  \( u = z - 1 \)
- \( t_4: \ \text{if}(u > 0) \)
  \( u = u - 1 \)

- \( t_5: \ \text{if}(u \leq 0) \)
  \( \text{if}(z > 0) \)
  \( z = z - 1 \)

0 \( \geq |t_0, y'| \)

\(|y| + 1 \geq |t_1, y'| \)

\(|y| \geq |t_2, z'| \)

Result Variable Graph:
- Nodes \(|t, v'|\), labels \( S_l(t, v')\)

Change of \( v \) in one use of \( t \):

\[ t \implies S_l(t, v')(\mathcal{V}) \geq v' \]
\( R(t_0) = 1 \)
\( R(t_1) = |x| \)
\( R(t_2) = 1 \)

\[ t_0: \quad y = 0 \]

\[ t_1: \quad \text{if}(x > 0) \]
\[ y = y + 1 \]
\[ x = x - 1 \]

\[ t_2: \quad \text{if}(x \leq 0) \]
\[ z = y \]

\[ t_3: \quad \text{if}(z > 0) \]
\[ u = z - 1 \]

\[ t_4: \quad \text{if}(u > 0) \]
\[ \text{if}(z > 0) \]
\[ u = u - 1 \]

\[ t_5: \quad \text{if}(u \leq 0) \]

\[ 0 \geq |t_0, y'| \]

\[ |y| + 1 \geq |t_1, y'| \]

\[ |y| \geq |t_2, z'| \]

**Result Variable Graph:**

- **Nodes** \( |t, v'| \), labels \( S_l(t, v') \)
- Change of \( v \) in one use of \( t \):
  \[ t \quad \implies \quad S_l(t, v')(\mathcal{V}) \geq v' \]

- **Edges:**
  Flow of information
Size Bounds: Local

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]

\[ t_0: \ y = 0 \]
\[ t_1: \ \text{if}(x > 0) \]
\[ \quad y = y + 1 \]
\[ \quad x = x - 1 \]
\[ t_2: \ \text{if}(x \leq 0) \]
\[ \quad z = y \]

\[ t_3: \ \text{if}(z > 0) \]
\[ \quad u = z - 1 \]
\[ t_4: \ \text{if}(u > 0) \]
\[ \quad \text{if}(z > 0) \]
\[ \quad u = u - 1 \]

\[ 0 \geq |t_0, y'| \]
\[ \downarrow \]
\[ |y| + 1 \geq |t_1, y'| \]
\[ |y| \geq |t_2, z'| \]

Result Variable Graph:

- Nodes \(|t, v'|\), labels \(S_l(t, v')\)
- Change of \(v\) in one use of \(t\):
  \[ t \implies S_l(t, v')(V) \geq v' \]
- Edges:
  Flow of information
\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]

\begin{align*}
\ell_0 & \quad t_0: \quad y = 0 \\
\ell_1 & \quad t_1: \quad \text{if } (x > 0) \quad y = y + 1, \quad x = x - 1 \\
\ell_2 & \quad t_2: \quad \text{if } (x \leq 0) \quad z = y \\
\ell_3 & \quad t_5: \quad \text{if } (u \leq 0) \quad \text{if } (z > 0) \quad z = z - 1 \\
\ell_4 & \quad t_4: \quad \text{if } (u > 0) \quad \text{if } (z > 0) \quad u = u - 1 \\
\end{align*}

\[ 0 \geq |t_0, y'| \]
\[ \downarrow R \]
\[ |y| + 1 \geq |t_1, y'| \]
\[ |y| \geq |t_2, z'| \]

**Result Variable Graph:**
- Nodes \(|t, v'|\), labels \(S_l(t, v')\)
- Change of \(v\) in \textit{one use} of \(t\):
  \[ t \implies S_l(t, v')(V) \geq v' \]
- Edges:
  Flow of information
Size Bounds: Local

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]

\[ t_0: \quad y = 0 \]
\[ t_1: \quad \text{if}(x > 0) \]
\[ \quad y = y + 1 \]
\[ \quad x = x - 1 \]
\[ t_2: \quad \text{if}(x \leq 0) \]
\[ \quad z = y \]

\[ t_3: \quad \text{if}(z > 0) \]
\[ \quad u = z - 1 \]
\[ t_4: \quad \text{if}(u > 0) \]
\[ \quad \text{if}(z > 0) \]
\[ \quad u = u - 1 \]

\[ 0 \geq |t_0, y'| \]
\[ |y| + 1 \geq |t_1, y'| \]
\[ |y| \geq |t_2, z'| \]

Result Variable Graph:

- Nodes \(|t, v'|\), labels \(S_l(t, v')\)
- Change of \(v\) in one use of \(t\):
  \[ t \rightarrow S_l(t, v')(\mathcal{V}) \geq v' \]

- Edges:
  Flow of information
\( R(t_0) = 1 \)
\( R(t_1) = |x| \)
\( R(t_2) = 1 \)

\[ \begin{align*}
  t_0: & \quad y = 0 \\
  t_1: & \quad \text{if}(x > 0) \\
    & \quad y = y + 1 \\
    & \quad x = x - 1 \\
  t_2: & \quad \text{if}(x \leq 0) \\
    & \quad z = y \\
  t_3: & \quad \text{if}(u > 0) \\
    & \quad u = u - 1 \\
  t_4: & \quad \text{if}(u > 0) \\
    & \quad \text{if}(z > 0) \\
    & \quad u = u - 1 \\
  t_5: & \quad \text{if}(u \leq 0) \\
    & \quad \text{if}(z > 0) \\
    & \quad z = z - 1 \\
\end{align*} \]

Result Variable Graph:
- Nodes \(|t, v'|\), labels \(S_l(t, v')\)
- Change of \(v\) in one use of \(t\):
  \[ t \implies S_l(t, v')(\mathcal{V}) \geq v' \]
- Edges: Flow of information
\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]

**Computing \( S(t, v') \):**

**Result Variable Graph:**

- **Nodes** \( |t, v'| \), **labels** \( S_l(t, v') \)
- Change of \( v \) in **one use** of \( t \):

\[ t \implies S_l(t, v')(\mathcal{V}) \geq v' \]

- **Edges:**
  - Flow of information
Size Bounds: Global

\[ \mathcal{R}(t_0) = 1 \quad \mathcal{S}(t_0, y') = 0 \]
\[ \mathcal{R}(t_1) = |x| \]
\[ \mathcal{R}(t_2) = 1 \]

Computing \( \mathcal{S}(t, v') \):

- No cycles: \( \mathcal{S}_l \)

Result Variable Graph:

- Nodes \( |t, v'| \), labels \( \mathcal{S}_l(t, v') \)

Change of \( v \) in one use of \( t \):

\[ t \implies \mathcal{S}_l(t, v')(\mathcal{V}) \geq v' \]

- Edges:
  Flow of information
Size Bounds: Global

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]

\[ S(t_0, y') = 0 \]
\[ S(t_1, y') = |x| \]

Computing \( S(t, v') \):
- No cycles: \( S_l \)
- Cycles: Combine \( R, S_l \)
  - if \( S_l \approx v + c, c \in \mathbb{Z} \):
    \[ S(t, v') = S(\tilde{t}, v') + R(t) \cdot c \]
    \( \tilde{t} \) predecessor of \( t \)

Result Variable Graph:
- Nodes \( |t, v'| \), labels \( S_l(t, v') \)
  - Change of \( v \) in one use of \( t \):
    \[ t \implies S_l(t, v')(V) \geq v' \]
- Edges:
  - Flow of information
Size Bounds: Global

\[ \mathcal{R}(t_0) = 1 \quad \mathcal{S}(t_0, y') = 0 \]
\[ \mathcal{R}(t_1) = |x| \quad \mathcal{S}(t_1, y') = |x| \]
\[ \mathcal{R}(t_2) = 1 \quad \mathcal{S}(t_2, z') = |x| \]

Result Variable Graph:
- Nodes \(|t, v'|\), labels \(S_l(t, v')\)
- Change of \(v\) in one use of \(t\):
  \[ t \implies S_l(t, v')(V) \geq v' \]
- Edges:
  Flow of information

Computing \(\mathcal{S}(t, v')\):
- No cycles: \(S_l (+ \text{ propagation})\)
- Cycles: Combine \(\mathcal{R}, S_l\)
  - if \(S_l \approx v + c, c \in \mathbb{Z}\):
    \[ \mathcal{S}(t, v') = \mathcal{S}(%5x, v') + \mathcal{R}(t) \cdot c \]
    \(\tilde{t}\) predecessor of \(t\)
Size Bounds: Global

\[ R(t_0) = 1 \quad S(t_0, y') = 0 \]
\[ R(t_1) = |x| \quad S(t_1, y') = |x| \]
\[ R(t_2) = 1 \quad S(t_2, z') = |x| \]

Computing \( S(t, v') \):
- No cycles: \( S_l (+ \text{ propagation}) \)
- Cycles: Combine \( R, S_l \)
  - if \( S_l \approx v + c, c \in \mathbb{Z} \):
    \[ S(t, v') = S(\tilde{t}, v') + R(t) \cdot c \]
    \( \tilde{t} \) predecessor of \( t \)
  - More complex: See paper

Result Variable Graph:
- Nodes \(|t, v'|\), labels \( S_l(t, v') \)
- Change of \( v \) in one use of \( t \):
  \[ t \implies S_l(t, v')(\mathcal{V}) \geq v' \]
- Edges:
  Flow of information
\[ \mathcal{R}(t_0) = 1 \quad \mathcal{S}(t_0, y') = 0 \]
\[ \mathcal{R}(t_1) = |x| \quad \mathcal{S}(t_1, y') = |x| \]
\[ \mathcal{R}(t_2) = 1 \quad \mathcal{S}(t_2, z') = |x| \]

**Example (PRF IV)**

Consider only \( T_1 = \{ t_3, t_4, t_5 \} \)

- \( t_5: \) if \( u \leq 0 \)
  - if \( z > 0 \)
  - \( z = z - 1 \)
- \( t_4: \) if \( u > 0 \)
  - if \( z > 0 \)
  - \( u = u - 1 \)
- \( t_3: \) if \( z > 0 \)
  - \( u = z - 1 \)
Runtime Bounds II: Modularity

\( R(t_0) = 1 \)
\( R(t_1) = |x| \)
\( R(t_2) = 1 \)

\( S(t_0, y') = 0 \)
\( S(t_1, y') = |x| \)
\( S(t_2, z') = |x| \)

Example (PRF IV)
Consider only \( \mathcal{T}_1 = \{t_3, t_4, t_5\} \)

\begin{align*}
  t_5: & \quad \text{if}(u \leq 0) \\
       & \quad \text{if}(z > 0) \\
       & \quad z = z - 1

  t_3: & \quad \text{if}(z > 0) \\
       & \quad u = z - 1

  t_4: & \quad \text{if}(u > 0) \\
       & \quad \text{if}(z > 0) \\
       & \quad u = u - 1
\end{align*}
Example (PRF IV)

Consider only $T_1 = \{t_3, t_4, t_5\}$

$P_4(\ell_2) = P_4(\ell_3) = z$

no increase on transitions $T_1$

$t_5$ decreases, bounded
\[ \mathcal{R}(t_0) = 1 \quad \mathcal{S}(t_0, y') = 0 \]
\[ \mathcal{R}(t_1) = |x| \quad \mathcal{S}(t_1, y') = |x| \]
\[ \mathcal{R}(t_2) = 1 \quad \mathcal{S}(t_2, z') = |x| \]

Example (PRF IV)

Consider only \( \mathcal{T}_1 = \{t_3, t_4, t_5\} \)

\[ \mathcal{P}_4(\ell_2) = \mathcal{P}_4(\ell_3) = z \]

**no increase** on transitions \( \mathcal{T}_1 \)

\( t_5 \) decreases, bounded

\( \quad \rightarrow \textbf{When} \ \mathcal{T}_1 \ \text{reached, then } z \ \text{steps:} \)

\( t_5: \quad \text{if}(u \leq 0) \)
\( \quad \text{if}(z > 0) \)
\( \quad z = z - 1 \)

\( t_4: \quad \text{if}(u > 0) \)
\( \quad \text{if}(z > 0) \)
\( \quad u = u - 1 \)

\( t_3: \quad \text{if}(z > 0) \)
\( \quad u = z - 1 \)

\( t_2: \text{if}(z > 0) \quad \)
Runtime Bounds II: Modularity

\[ \mathcal{R}(t_0) = 1 \]
\[ \mathcal{R}(t_1) = |x| \]
\[ \mathcal{R}(t_2) = 1 \]

**Example (PRF IV)**

Consider only \( \mathcal{T}_1 = \{t_3, t_4, t_5\} \)

\[ \mathcal{P}_4(\ell_2) = \mathcal{P}_4(\ell_3) = z \]

**no increase on transitions** \( \mathcal{T}_1 \)

\( t_5 \) **decreases, bounded**

\( \rightarrow \textbf{When} \ \mathcal{T}_1 \ \text{reached, then} \ z \ \text{steps:} \)

\[ \mathcal{T}_1 \ \text{reached} \ \mathcal{R}(t_2) = 1 \text{ time} \]
\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]
\[ S(t_0, y') = 0 \]
\[ S(t_1, y') = |x| \]
\[ S(t_2, z') = |x| \]

**Example (PRF IV)**

Consider only \( \mathcal{T}_1 = \{t_3, t_4, t_5\} \)

\[ P_4(l_2) = P_4(l_3) = z \]

**no increase on transitions** \( \mathcal{T}_1 \)

\( t_5 \) decreases, bounded

\( \leftarrow \) **When** \( \mathcal{T}_1 \) reached, then \( z \) **steps**:

\( \mathcal{T}_1 \) reached \( R(t_2) = 1 \) time

\( z \) has size \( S(t_2, y') = |x| \)
\[ \mathcal{R}(t_0) = 1 \quad \mathcal{S}(t_0, y') = 0 \]
\[ \mathcal{R}(t_1) = |x| \quad \mathcal{S}(t_1, y') = |x| \]
\[ \mathcal{R}(t_2) = 1 \quad \mathcal{S}(t_2, z') = |x| \]

\[ \mathcal{R}(t_5) = |x| \]

**Example (PRF IV)**

Consider only \( \mathcal{T}_1 = \{t_3, t_4, t_5\} \)

\[ \mathcal{P}_4(\ell_2) = \mathcal{P}_4(\ell_3) = z \]

no increase on transitions \( \mathcal{T}_1 \)

\( t_5 \) decreases, bounded

\[ \leftarrow \text{When } \mathcal{T}_1 \text{ reached, then } z \text{ steps:} \]

\( T_1 \text{ reached } \mathcal{R}(t_2) = 1 \text{ time} \)

\( z \text{ has size } \mathcal{S}(t_2, y') = |x| \)

\[ \leftarrow \mathcal{R}(t_5) = \mathcal{R}(t_2) \cdot \mathcal{S}(t_2, y') = 1 \cdot |x| \]
**Runtime Bounds II: Modularity**

\[
\mathcal{R}(t_0) = 1 \quad \mathcal{S}(t_0, y') = 0 \\
\mathcal{R}(t_1) = |x| \quad \mathcal{S}(t_1, y') = |x| \\
\mathcal{R}(t_2) = 1 \quad \mathcal{S}(t_2, z') = |x| \\
\mathcal{R}(t_5) = |x|
\]

**Example (PRF V)**

Consider only \( \mathcal{T}_2 = \{t_3, t_4\} \)

\[
\mathcal{P}_4(\ell_2) = 1 \quad \mathcal{P}_4(\ell_3) = 0
\]

**no increase on transitions** \( \mathcal{T}_2 \)

\( t_3 \) decreases, bounded

\[
t_5: \quad \text{if}(u \leq 0) \quad \text{if}(z > 0) \\
\quad z = z - 1
\]

\[
t_3: \quad \text{if}(z > 0) \\
\quad u = z - 1
\]

\[
t_4: \quad \text{if}(u > 0) \quad \text{if}(z > 0) \\
\quad u = u - 1
\]
Runtime Bounds II: Modularity

\[ R(t_0) = 1 \quad S(t_0, y') = 0 \]
\[ R(t_1) = |x| \quad S(t_1, y') = |x| \]
\[ R(t_2) = 1 \quad S(t_2, z') = |x| \]

\[ R(t_5) = |x| \]

Example (PRF V)

Consider only \( T_2 = \{ t_3, t_4 \} \)

\[ P_4(\ell_2) = 1 \quad P_4(\ell_3) = 0 \]

no increase on transitions \( T_2 \)

\( t_3 \) decreases, bounded

\[ \text{When } T_2 \text{ reached, then 1 step:} \]

\[ t_5: \quad \text{if}(u \leq 0) \]
\[ \quad \text{if}(z > 0) \]
\[ \quad z = z - 1 \]

\[ t_3: \quad \text{if}(z > 0) \]
\[ \quad u = z - 1 \]

\[ t_4: \quad \text{if}(u > 0) \]
\[ \quad \text{if}(z > 0) \]
\[ \quad u = u - 1 \]
Runtime Bounds II: Modularity

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]
\[ S(t_0, y') = 0 \]
\[ S(t_1, y') = |x| \]
\[ S(t_2, z') = |x| \]
\[ S(t_5, y') = 0 \]

\[ P_4(\ell_2) = 1 \quad P_4(\ell_3) = 0 \]

Example (PRF V)

Consider only \( \mathcal{T}_2 = \{t_3, t_4\} \)

no increase on transitions \( \mathcal{T}_2 \)

\( t_3 \) decreases, bounded

\( \leftarrow \text{When } \mathcal{T}_2 \text{ reached, then } 1 \text{ step:} \)

\( \mathcal{T}_2 \text{ reached} \)

\[ R(t_2) = 1 \text{ time and} \]
\[ R(t_5) = |x| \text{ times} \]
Runtime Bounds II: Modularity

\[ \mathcal{R}(t_0) = 1 \quad \mathcal{S}(t_0, y') = 0 \]
\[ \mathcal{R}(t_1) = |x| \quad \mathcal{S}(t_1, y') = |x| \]
\[ \mathcal{R}(t_2) = 1 \quad \mathcal{S}(t_2, z') = |x| \]
\[ \mathcal{R}(t_3) = |x| + 1 \]
\[ \mathcal{R}(t_5) = |x| \]

Example (PRF V)
Consider only \( \mathcal{T}_2 = \{t_3, t_4\} \)
\[ \mathcal{P}_4(\ell_2) = 1 \quad \mathcal{P}_4(\ell_3) = 0 \]

no increase on transitions \( \mathcal{T}_2 \)
\( t_3 \) decreases, bounded

\[ \mathcal{R}(t_2) = 1 \text{ time and} \quad \mathcal{R}(t_5) = |x| \text{ times} \]

\[ \mathcal{R}(t_3) = \mathcal{R}(t_2) \cdot 1 + \mathcal{R}(t_5) \cdot 1 \]
\[ = 1 \cdot 1 + |x| \cdot 1 \]
Runtime Bounds II: Modularity

\[ \mathcal{R}(t_0) = 1 \]
\[ \mathcal{R}(t_1) = |x| \]
\[ \mathcal{R}(t_2) = 1 \]
\[ \mathcal{R}(t_3) = |x| + 1 \]
\[ \mathcal{R}(t_5) = |x| \]

\[ \mathcal{S}(t_0, y') = 0 \]
\[ \mathcal{S}(t_1, y') = |x| \]
\[ \mathcal{S}(t_2, z') = |x| \]

**Example (PRF VI)**

Consider only \( \mathcal{T}_3 = \{t_4\} \)

\[ \mathcal{P}_5(\ell_3) = u \]

no increase on transitions \( \mathcal{T}_3 \)

\( t_4 \) decreases, bounded

\( t_4: \quad \textbf{if}(u > 0) \Rightarrow \ell_3 \)
\[ \textbf{if}(z > 0) \]
\[ u = u - 1 \]
Runtime Bounds II: Modularity

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]
\[ R(t_3) = |x| + 1 \]
\[ R(t_5) = |x| \]

\[ S(t_0, y') = 0 \]
\[ S(t_1, y') = |x| \]
\[ S(t_2, z') = |x| \]

Example (PRF VI)

Consider only \( \mathcal{T}_3 = \{t_4\} \)

\[ \mathcal{P}_5(\ell_3) = u \]

**no increase** on transitions \( \mathcal{T}_3 \)

\( t_4 \) decreases, bounded

→ **When** \( \mathcal{T}_3 \) reached, then \( u \) **steps**:

\[ t_4: \quad \text{if}(u > 0) \]
\[ \quad \text{if}(z > 0) \]
\[ \quad u = u - 1 \]
Runtime Bounds II: Modularity

\[ \mathcal{R}(t_0) = 1 \]
\[ \mathcal{R}(t_1) = |x| \]
\[ \mathcal{R}(t_2) = 1 \]
\[ \mathcal{R}(t_3) = |x| + 1 \]
\[ \mathcal{R}(t_5) = |x| \]

\[ S(t_0, y') = 0 \]
\[ S(t_1, y') = |x| \]
\[ S(t_2, z') = |x| \]

Example (PRF VI)

Consider only \( T_3 = \{t_4\} \)

\[ \mathcal{P}_5(\ell_3) = u \]

no increase on transitions \( T_3 \)

\( t_4 \) decreases, bounded

\( \rightarrow \) When \( T_3 \) reached, then \( u \) steps:

\[ T_3 \text{ reached } \mathcal{R}(t_3) = |x| + 1 \text{ times} \]
Runtime Bounds II: Modularity

\[ \mathcal{R}(t_0) = 1 \]
\[ \mathcal{R}(t_1) = |x| \]
\[ \mathcal{R}(t_2) = 1 \]
\[ \mathcal{R}(t_3) = |x| + 1 \]
\[ \mathcal{R}(t_5) = |x| \]

\[ S(t_0, y') = 0 \]
\[ S(t_1, y') = |x| \]
\[ S(t_2, z') = |x| \]

Example (PRF VI)

Consider only \( \mathcal{T}_3 = \{t_4\} \)

\[ P_5(\ell_3) = u \]

no increase on transitions \( \mathcal{T}_3 \)

\( t_4 \) decreases, bounded

\[ \text{When } \mathcal{T}_3 \text{ reached, then } u \text{ steps:} \]

\( \mathcal{T}_3 \) reached \( \mathcal{R}(t_3) = |x| + 1 \) times

\( u \) has size \( S(t_3, u') \)
Runtime Bounds II: Modularity

\[ R(t_0) = 1 \]
\[ R(t_1) = |x| \]
\[ R(t_2) = 1 \]
\[ R(t_3) = |x| + 1 \]
\[ R(t_5) = |x| \]

\[ S(t_0, y') = 0 \]
\[ S(t_1, y') = |x| \]
\[ S(t_2, z') = |x| \]
\[ S(t_3, u') = |x| \]

**Example (PRF VI)**

Consider only \( T_3 = \{t_4\} \)

\[ P_5(\ell_3) = u \]

no increase on transitions \( T_3 \)

\( t_4 \) decreases, bounded

\[ \text{When } T_3 \text{ reached, then } u \text{ steps:} \]
\[ T_3 \text{ reached } R(t_3) = |x| + 1 \text{ times} \]
\[ u \text{ has size } S(t_3, u') = |x| \]
Runtime Bounds II: Modularity

\[ \mathcal{R}(t_0) = 1 \quad \mathcal{S}(t_0, y') = 0 \]
\[ \mathcal{R}(t_1) = |x| \quad \mathcal{S}(t_1, y') = |x| \]
\[ \mathcal{R}(t_2) = 1 \quad \mathcal{S}(t_2, z') = |x| \]
\[ \mathcal{R}(t_3) = |x| + 1 \quad \mathcal{S}(t_3, u') = |x| \]
\[ \mathcal{R}(t_4) = |x|^2 + |x| \]
\[ \mathcal{R}(t_5) = |x| \]

Example (PRF VI)
Consider only \( \mathcal{T}_3 = \{t_4\} \)
\[ \mathcal{P}_5(\ell_3) = u \]

no increase on transitions \( \mathcal{T}_3 \)
t\( t_4 \) decreases, bounded

\( \leftarrow \) When \( \mathcal{T}_3 \) reached, then \( u \) steps:
\( \mathcal{T}_3 \) reached \( \mathcal{R}(t_3) = |x| + 1 \) times
\( u \) has size \( \mathcal{S}(t_3, u') = |x| \)

\[ \leftarrow \mathcal{R}(t_4) = \mathcal{R}(t_3) \cdot \mathcal{S}(t_3, u') \]
\[ = (|x| + 1) \cdot |x| \]
TimeBounds: Procedure

**TimeBounds**(\(\mathcal{R}, \mathcal{S}\))

**Input:** Runtime bounds \(\mathcal{R}\), Size bounds \(\mathcal{S}\)

\(\mathcal{T}' \leftarrow \{ t \in \mathcal{T} \mid \mathcal{R}(t) \) unbounded\}

\(\mathcal{P} \leftarrow \text{synthPRF}(\mathcal{T}')\)

\(\mathcal{L}_\downarrow \leftarrow \text{entryLocations}(\mathcal{T}')\)

\(\mathcal{T}_\ell \leftarrow \text{leadingTo}(\ell, \mathcal{T} \setminus \mathcal{T}')\)

\(\mathcal{R}' \leftarrow \mathcal{R}\)

for all \(t \in \mathcal{T}'\) decreasing under \(\mathcal{P}\) do

\[
\mathcal{R}'(t) \leftarrow \sum_{\ell \in \mathcal{L}_\downarrow, t \in \mathcal{T}_\ell} \mathcal{R}(\tilde{t}) \cdot [\mathcal{P}(\ell)](\mathcal{S}(\tilde{t}, v'_1), \ldots, \mathcal{S}(\tilde{t}, v'_n))
\]

end for

**Output:** \(\mathcal{R}'\)
SizeBounds: Procedure

SizeBoundsTriv($R, S, C$)

**Input:** Runtime bounds $R$, Size bounds $S$, $C = \{|t, v'|\}$

- $T_t \leftarrow \text{leadingTo}(t, \mathcal{T})$
- $S' \leftarrow S$
- $S'(t, v') \leftarrow \max\{S_1(t, v')(S(\tilde{t}, v'_1), \ldots, S(\tilde{t}, v'_n)) | \tilde{t} \in T_t\}$

**Output:** $S'$
SizeBounds: Procedure

SizeBoundsTriv(\(R, S, C\))

**Input:** Runtime bounds \(R\), Size bounds \(S\), \(C = \{|t, v'|\}\)

\(T_t \leftarrow \text{leadingTo}(t, T)\)

\(S' \leftarrow S\)

\(S'(t, v') \leftarrow \max\{S_l(t, v')(S(\tilde{t}, v'_1), \ldots, S(\tilde{t}, v'_n)) | \tilde{t} \in T_t\}\)

**Output:** \(S'\)

SizeBoundsNonTriv(\(R, S, C\))

Case \(C\) non-trivial Strongly Connected Component: See paper
AlternatingCompl: Overall Procedure

\begin{align*}
\text{AlternatingCompl}(\mathcal{T}, \mathcal{V})
\end{align*}

**Input:** Program of transitions $\mathcal{T}$, variables $\mathcal{V}$
- $\mathcal{R} \leftarrow \text{unboundedTimeCompl}(\mathcal{T})$
- $\mathcal{S} \leftarrow \text{unboundedSizeCompl}(\mathcal{T}, \mathcal{V})$

\textbf{while} $\mathcal{R}, \mathcal{S}$ have unbounded elements \textbf{do}
- $\mathcal{R} \leftarrow \text{TimeBounds}(\mathcal{R}, \mathcal{S})$
- \textbf{for all} $C$ SCC of $\text{RVG}(\mathcal{T}, \mathcal{V})$ \textbf{do}
  - $\mathcal{S} \leftarrow \text{SizeBounds}(\mathcal{R}, \mathcal{S}, C)$
\textbf{end for}
\textbf{end while}

**Output:** $\mathcal{R}, \mathcal{S}$
Are There Other Techniques and Tools?

- Using techniques from termination proving: ABC\(^2\), AProVE, CoFloCo\(^3\), COSTA/PUBS\(^4\), Loopus\(^5\), Rank\(^6\), TcT\(^7\), ...
Are There Other Techniques and Tools?

- Using techniques from termination proving: ABC\textsuperscript{2}, AProVE, CoFloCo\textsuperscript{3}, COSTA/PUBS\textsuperscript{4}, Loopus\textsuperscript{5}, Rank\textsuperscript{6}, TcT\textsuperscript{7}, ...
- Using invariant generation: SPEED\textsuperscript{8}

\textsuperscript{2}R. Blanc, T. Henzinger, L. Kovács: \textit{ABC: Algebraic Bound Computation for Loops}, LPAR (Dakar) '10  
\textsuperscript{3}A. Flores-Montoya and R. Hähnle: \textit{Resource Analysis of Complex Programs with Cost Equations}, APLAS '14  
\textsuperscript{5}M. Sinn, F. Zuleger, H. Veith: \textit{A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis}, CAV '14  
\textsuperscript{6}C. Alias, A. Darte, P. Feautrier, L. Gonnord: \textit{Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs}, SAS '10  
\textsuperscript{7}M. Avanzini, G. Moser, M. Schaper: \textit{TcT: Tyrolean Complexity Tool}, TACAS '16  
\textsuperscript{8}S. Gulwani, K. Mehro, T. Chilimbi: \textit{SPEED: precise and efficient static estimation of program computational complexity}, POPL '09
Are There Other Techniques and Tools?

- Using techniques from termination proving: ABC², AProVE, CoFloCo³, COSTA/PUBS⁴, Loopus⁵, Rank⁶, TcT⁷, ... 
- Using invariant generation: SPEED⁸ 
- Using type-based amortised analysis:⁹ RAML¹⁰, ... 

² R. Blanc, T. Henzinger, L. Kovács: ABC: Algebraic Bound Computation for Loops, LPAR (Dakar) ’10
³ A. Flores-Montoya and R. Hähnle: Resource Analysis of Complex Programs with Cost Equations, APLAS ’14
⁵ M. Sinn, F. Zuleger, H. Veith: A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis, CAV ’14
⁶ C. Alias, A. Darte, P. Feautrier, L. Gonnord: Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs, SAS ’10
⁷ M. Avanzini, G. Moser, M. Schaper: TcT: Tyrolean Complexity Tool, TACAS ’16
⁸ S. Gulwani, K. Mehro, T. Chilimbi: SPEED: precise and efficient static estimation of program computational complexity, POPL ’09
⁹ J. Hoffmann, S. Jost: Two decades of automatic amortized resource analysis, MSCS ’22
¹⁰ J. Hoffmann, K. Aehlig, M. Hofmann: Resource Aware ML, CAV ’12
Prototype: KoAT, using Microsoft’s SMT solver Z3 (Z3 on github: https://github.com/Z3Prover/z3) to find PRFs, size bounds, ...

<table>
<thead>
<tr>
<th>Tool</th>
<th>log n</th>
<th>n</th>
<th>log n</th>
<th>n</th>
<th>log n</th>
<th>n</th>
<th>log n</th>
<th>n</th>
<th>log n</th>
<th>n</th>
<th>log n</th>
<th>n</th>
<th>log n</th>
<th>n</th>
<th>log n</th>
</tr>
</thead>
<tbody>
<tr>
<td>KoAT</td>
<td>131</td>
<td>0</td>
<td>167</td>
<td>0</td>
<td>78</td>
<td>7</td>
<td>3</td>
<td>18</td>
<td>285</td>
<td>0.7 s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoFloCo</td>
<td>117</td>
<td>0</td>
<td>153</td>
<td>0</td>
<td>66</td>
<td>9</td>
<td>2</td>
<td>0</td>
<td>342</td>
<td>1.3 s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loopus</td>
<td>117</td>
<td>0</td>
<td>130</td>
<td>0</td>
<td>49</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>383</td>
<td>0.2 s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KoAT-TACAS'14</td>
<td>118</td>
<td>0</td>
<td>127</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>391</td>
<td>1.1 s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUBS</td>
<td>109</td>
<td>4</td>
<td>127</td>
<td>6</td>
<td>24</td>
<td>8</td>
<td>0</td>
<td>7</td>
<td>404</td>
<td>0.8 s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rank</td>
<td>56</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>608</td>
<td>0.1 s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Timeout 60 s

Time is average runtime for successful proof

103/173
Prototype: KoAT, using Microsoft’s SMT solver Z3 (Z3 on github: https://github.com/Z3Prover/z3) to find PRFs, size bounds, ...

682 examples, taken from

- prior evaluations (of ABC, Loopus, PUBS/COSTA, Rank, SPEED)
- termination benchmarks (of T2, AProVE)
- examples from our article describing the techniques
Prototype: KoAT, using Microsoft’s SMT solver Z3 (Z3 on github: https://github.com/Z3Prover/z3) to find PRFs, size bounds, ...

682 examples, taken from
   - prior evaluations (of ABC, Loopus, PUBS/COSTA, Rank, SPEED)
   - termination benchmarks (of T2, AProVE)
   - examples from our article describing the techniques

<table>
<thead>
<tr>
<th>Tool</th>
<th>1</th>
<th>log n</th>
<th>n</th>
<th>n log n</th>
<th>n^2</th>
<th>n^3</th>
<th>n&gt;^3</th>
<th>EXP</th>
<th>No res.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>KoAT</td>
<td>131</td>
<td>0</td>
<td>167</td>
<td>0</td>
<td>78</td>
<td>7</td>
<td>3</td>
<td>18</td>
<td></td>
<td>0.7 s</td>
</tr>
<tr>
<td>CoFloCo</td>
<td>117</td>
<td>0</td>
<td>153</td>
<td>0</td>
<td>66</td>
<td>9</td>
<td>2</td>
<td>0</td>
<td>342</td>
<td>1.3 s</td>
</tr>
<tr>
<td>Loopus</td>
<td>117</td>
<td>0</td>
<td>130</td>
<td>0</td>
<td>49</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>383</td>
<td>0.2 s</td>
</tr>
<tr>
<td>KoAT-TACAS’14</td>
<td>118</td>
<td>0</td>
<td>127</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>391</td>
<td>1.1 s</td>
</tr>
<tr>
<td>PUBS</td>
<td>109</td>
<td>4</td>
<td>127</td>
<td>6</td>
<td>24</td>
<td>8</td>
<td>0</td>
<td>7</td>
<td>404</td>
<td>0.8 s</td>
</tr>
<tr>
<td>Rank</td>
<td>56</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>608</td>
<td>0.1 s</td>
</tr>
</tbody>
</table>

- timeout 60 s
- *Time* is average runtime for successful proof
Comparing KoAT directly to other tools (wrt asymptotic bounds)

<table>
<thead>
<tr>
<th>Compared tool</th>
<th>more precise</th>
<th>less precise</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoFloCo</td>
<td>31</td>
<td>80</td>
</tr>
<tr>
<td>KoAT-TACAS’14</td>
<td>0</td>
<td>118</td>
</tr>
<tr>
<td>PUBS</td>
<td>46</td>
<td>134</td>
</tr>
<tr>
<td>Loopus</td>
<td>16</td>
<td>117</td>
</tr>
<tr>
<td>Rank</td>
<td>5</td>
<td>327</td>
</tr>
</tbody>
</table>

⇒ each tool has its own strengths and weaknesses
Extensions implemented:

- Recursion (beyond tail recursion)
Extensions implemented:

- Recursion (beyond tail recursion)
- Exponential data (e.g., `while x > 0 do y = 2 \cdot y; x --; done`)

Other cost measures (e.g., network traffic, energy usage, ...)

- annotate transitions with cost
- (so far: each transition costs 1)
Extensions implemented:

- Recursion (beyond tail recursion)
- Exponential data (e.g., \texttt{while }x > 0 \texttt{ do } y = 2 \cdot y; x \leftarrow -; \texttt{ done})
- Exponential calls (e.g., \( f(n) = f(n - 1) + f(n - 2) \))
So, Is That Everything?

Extensions implemented:

- Recursion (beyond tail recursion)
- Exponential data (e.g., \texttt{while }x > 0 \texttt{ do } y = 2 \cdot y; \ x \ -= \ -; \ \texttt{done})
- Exponential calls (e.g., \( f(n) = f(n - 1) + f(n - 2) \))
- Methods handled independently, composing results at call sites
Extensions implemented:

- Recursion (beyond tail recursion)
- Exponential data (e.g., `while x > 0 do y = 2 \cdot y; x ← −; done`)
- Exponential calls (e.g., \( f(n) = f(n - 1) + f(n - 2) \))
- Methods handled independently, composing results at call sites
- Other cost measures (e.g., network traffic, energy usage, \ldots)
- \( \rightarrow \) annotate transitions with cost of transition
  (so far: each transition costs 1)
Extensions implemented:

- Recursion (beyond tail recursion)
- Exponential data (e.g., `while x > 0 do y = 2 \cdot y; x--; done`)
- Exponential calls (e.g., `f(n) = f(n - 1) + f(n - 2)`)
- Methods handled independently, composing results at call sites
- Other cost measures (e.g., network traffic, energy usage, ...)
  → annotate transitions with cost of transition
  (so far: each transition costs 1)

http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity-Journal
Precise handling of loops with computable complexity in the KoAT approach\(^{11}\)

\(^{11}\)N. Lommen, F. Meyer, J. Giesl: *Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops*, IJCAR '22
Precise handling of loops with computable complexity in the KoAT approach\textsuperscript{11}

Inference of \textit{lower} bounds for worst-case runtime complexity\textsuperscript{12}: LoAT\textsuperscript{13}

\textsuperscript{11} N. Lommen, F. Meyer, J. Giesl: \textit{Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops}, IJCAR '22

\textsuperscript{12} F. Frohn, M. Naaf, M. Brockschmidt, J. Giesl: \textit{Inferring Lower Runtime Bounds for Integer Programs}, TOPLAS '20

\textsuperscript{13} F. Frohn, J. Giesl: \textit{Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)}, IJCAR '22
Where Can I Learn More? Current Developments

- Precise handling of loops with computable complexity in the KoAT approach\textsuperscript{11}
- Inference of lower bounds for worst-case runtime complexity\textsuperscript{12}: LoAT\textsuperscript{13}
- Cost analysis for Java programs via Integer Transition Systems\textsuperscript{14}

\textsuperscript{11} N. Lommen, F. Meyer, J. Giesl: Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops, IJCAR '22
\textsuperscript{12} F. Frohn, M. Naaf, M. Brockschmidt, J. Giesl: Inferring Lower Runtime Bounds for Integer Programs, TOPLAS '20
\textsuperscript{13} F. Frohn, J. Giesl: Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description), IJCAR '22
\textsuperscript{14} F. Frohn, J. Giesl: Complexity Analysis for Java with AProVE, iFM '17
Where Can I Learn More? Current Developments

- Precise handling of loops with computable complexity in the KoAT approach\(^{11}\)
- Inference of lower bounds for worst-case runtime complexity\(^{12}\): LoAT\(^{13}\)
- Cost analysis for Java programs via Integer Transition Systems\(^{14}\)
- Cost analysis for probabilistic programs\(^{15,16,17}\)

\(^{11}\)N. Lommen, F. Meyer, J. Giesl: Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops, IJCAR ’22

\(^{12}\)F. Frohn, M. Naaf, M. Brockschmidt, J. Giesl: Inferring Lower Runtime Bounds for Integer Programs, TOPLAS ’20

\(^{13}\)F. Frohn, J. Giesl: Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description), IJCAR ’22

\(^{14}\)F. Frohn, J. Giesl: Complexity Analysis for Java with AProVE, iFM ’17

\(^{15}\)P. Wang, H. Fu, A. Goharshady, K. Chatterjee, X. Qin, W. Shi: Cost analysis of nondeterministic probabilistic programs, PLDI ’19

\(^{16}\)F. Meyer, M. Hark, J. Giesl: Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes, TACAS ’21

\(^{17}\)L. Leutgeb, G. Moser, F. Zuleger: Automated Expected Amortised Cost Analysis of Probabilistic Data Structures, CAV ’22
Key insights:

- Data size influences runtime
- Runtime influences data size
- *Other influences minor*
Key insights:

- Data size influences runtime
- Runtime influences data size
- *Other influences minor*

Solution:

- Alternating size/runtime analysis
- Modularity by using *only* these results
II.2 Complexity Analysis for Term Rewriting
What is *Term Rewriting*?

(1) Core functional programming language without many restrictions (and features) of “real” FP:

Example (Term Rewrite System (TRS) \( R \))

\[
\text{double}(0) \rightarrow 0 \\
\text{double}(s(x)) \rightarrow s(s(\text{double}(s(x))))
\]

Compute “double of 3 is 6”:

\[
\text{double}(s(s(s(s(0))))) \rightarrow R \\
\text{double}(s(s(s(s(0))))) \rightarrow R \\
\text{double}(s(s(s(s(0))))) \rightarrow R \\
\text{double}(s(s(s(s(0))))) \rightarrow R
\]
What is Term Rewriting?

(1) Core functional programming language without many restrictions (and features) of “real” FP:
   - first-order (usually)
   - no fixed evaluation strategy
   - untyped
   - no pre-defined data structures (integers, arrays, . . .)
What is Term Rewriting?

(1) Core functional programming language without many restrictions (and features) of “real” FP:
- first-order (usually)
- no fixed evaluation strategy
- untyped
- no pre-defined data structures (integers, arrays, . . .)

(2) Syntactic approach for reasoning in equational first-order logic

Example (Term Rewrite System (TRS) R)

double(0) \rightarrow 0
double(s(x)) \rightarrow s(s(double(x)))

Compute “double of 3 is 6”:
double(s(s(s(0)))) \rightarrow 6

in 4 steps with − → R
What is *Term Rewriting*?

(1) Core functional programming language

without many restrictions (and features) of “real” FP:

- first-order (usually)
- no fixed evaluation strategy
- untyped
- no pre-defined data structures (integers, arrays, ...)

(2) Syntactic approach for reasoning in equational first-order logic

---

**Example (Term Rewrite System (TRS) \( \mathcal{R} \))**

- \( \text{double}(0) \rightarrow 0 \)
- \( \text{double}(s(x)) \rightarrow s(s(\text{double}(x))) \)
What is *Term Rewriting*?

(1) Core functional programming language without many restrictions (and features) of “real” FP:
- first-order (usually)
- no fixed evaluation strategy
- untyped
- no pre-defined data structures (integers, arrays, . . .)

(2) Syntactic approach for reasoning in equational first-order logic

Example (Term Rewrite System (TRS) $\mathcal{R}$)

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double(0)</code> $\rightarrow$ 0</td>
</tr>
<tr>
<td><code>double(s(x))</code> $\rightarrow$ <code>s(s(double(x)))</code></td>
</tr>
</tbody>
</table>

Compute “double of 3 is 6”:

`double(s(s(s(0))))`
What is *Term Rewriting*?

(1) Core functional programming language
without many restrictions (and features) of “real” FP:
- first-order (usually)
- no fixed evaluation strategy
- untyped
- no pre-defined data structures (integers, arrays, . . .)

(2) Syntactic approach for reasoning in equational first-order logic

Example (Term Rewrite System (TRS) $\mathcal{R}$)

- $\text{double}(0) \rightarrow 0$
- $\text{double}(s(x)) \rightarrow s(s(\text{double}(x)))$

Compute “double of 3 is 6”:

\[ \text{double}(s(s(s(0)))) \rightarrow_{\mathcal{R}} s(s(\text{double}(s(s(0))))) \]
What is Term Rewriting?

(1) Core functional programming language without many restrictions (and features) of “real” FP:
- first-order (usually)
- no fixed evaluation strategy
- untyped
- no pre-defined data structures (integers, arrays, . . .)

(2) Syntactic approach for reasoning in equational first-order logic

Example (Term Rewrite System (TRS) $\mathcal{R}$)

$$
\begin{align*}
\text{double}(0) & \rightarrow 0 \\
\text{double}(s(x)) & \rightarrow s(s(\text{double}(x)))
\end{align*}
$$

Compute “double of 3 is 6”:

$$
\begin{align*}
\text{double}(s(s(s(0)))) & \\
\rightarrow_{\mathcal{R}} s(s(\text{double}(s(s(0))))) & \\
\rightarrow_{\mathcal{R}} s(s(s(s(s(\text{double}(s(0)))))))
\end{align*}
$$
What is **Term Rewriting**?

(1) Core functional programming language
without many restrictions (and features) of “real” FP:
- first-order (usually)
- no fixed evaluation strategy
- untyped
- no pre-defined data structures (integers, arrays, . . .)

(2) Syntactic approach for reasoning in equational first-order logic

**Example (Term Rewrite System (TRS) \( \mathcal{R} \))**

\[
\begin{align*}
\text{double}(0) & \rightarrow 0 \\
\text{double}(s(x)) & \rightarrow s(s(\text{double}(x)))
\end{align*}
\]

Compute “double of 3 is 6”:

\[
\begin{align*}
double(s(s(s(0)))) & \\
\rightarrow_\mathcal{R} & s(s(s(s(\text{double}(s(s(0))))))) \\
\rightarrow_\mathcal{R} & s(s(s(s(s(s(s(s(0))))))))) \\
\rightarrow_\mathcal{R} & s(s(s(s(s(s(s(s(s(s(0)))))))))
\end{align*}
\]
What is Term Rewriting?

(1) Core functional programming language
without many restrictions (and features) of “real” FP:
- first-order (usually)
- no fixed evaluation strategy
- untyped
- no pre-defined data structures (integers, arrays, ...)

(2) Syntactic approach for reasoning in equational first-order logic

Example (Term Rewrite System (TRS) $\mathcal{R}$)

Compute “double of 3 is 6”:

- $\text{double}(0) \rightarrow 0$
- $\text{double}(s(x)) \rightarrow s(s(\text{double}(x)))$
- $\text{double}(s(s(s(0)))) \rightarrow s(s(\text{double}(s(s(0)))))$
- $\text{double}(s(s(s(s(0))))) \rightarrow s(s(\text{double}(s(s(s(s(0)))))))$
- $\text{double}(s(s(s(s(s(0)))))) \rightarrow s(s(s(s(s(s(0)))))))$
What is *Term Rewriting*?

(1) Core functional programming language without many restrictions (and features) of “real” FP:
- first-order (usually)
- no fixed evaluation strategy
- untyped
- no pre-defined data structures (integers, arrays, . . .)

(2) Syntactic approach for reasoning in equational first-order logic

**Example (Term Rewrite System (TRS) \( \mathcal{R} \))**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double(0)</code> → 0</td>
<td></td>
</tr>
<tr>
<td><code>double(s(x))</code> → s(s(double(x)))`</td>
<td></td>
</tr>
</tbody>
</table>

Compute “double of 3 is 6”:

\[
\begin{align*}
double(s(s(s(0)))) & \rightarrow_{\mathcal{R}} s(s(double(s(s(0)))))) \\
& \rightarrow_{\mathcal{R}} s(s(s(s(s(double(s(0)))))))) \\
& \rightarrow_{\mathcal{R}} s(s(s(s(s(s(s(double(0)))))))) \\
& \rightarrow_{\mathcal{R}} s(s(s(s(s(s(s(0)))))))
\end{align*}
\]

in 4 steps with \( \rightarrow_{\mathcal{R}} \)
What is *Term Rewriting*?

(1) Core functional programming language without many restrictions (and features) of “real” FP:
- first-order (usually)
- no fixed evaluation strategy
- untyped
- no pre-defined data structures (integers, arrays, ...)

(2) Syntactic approach for reasoning in equational first-order logic

Example (Term Rewrite System (TRS) $\mathcal{R}$)

- $\text{double}(0) \rightarrow 0$
- $\text{double}(s(x)) \rightarrow s(s(\text{double}(x)))$

Compute “double of 3 is 6”:

$\text{double}(s^3(0))$
$\rightarrow_{\mathcal{R}} s^2(\text{double}(s^2(0)))$
$\rightarrow_{\mathcal{R}} s^4(\text{double}(s(0)))$
$\rightarrow_{\mathcal{R}} s^6(\text{double}(0))$
$\rightarrow_{\mathcal{R}} s^6(0)$

in 4 steps with $\rightarrow_{\mathcal{R}}$
What is \textit{Complexity} of Term Rewriting?

\textbf{Given:} TRS \( \mathcal{R} \) (e.g., \{ \texttt{double}(0) \rightarrow 0, \texttt{double}(\texttt{s}(x)) \rightarrow \texttt{s(s(double}(x))) \})
What is *Complexity* of Term Rewriting?

**Given:** TRS $\mathcal{R}$ (e.g., \{ double($0$) $\rightarrow$ 0, double($s(x)$) $\rightarrow$ $s(s(double(x)))$ \})

**Question:** How long can a $\rightarrow_\mathcal{R}$ sequence from a term of size $n$ become? (worst case)
What is Complexity of Term Rewriting?

Given: TRS $\mathcal{R}$ (e.g., \{ double(0) $\rightarrow$ 0, double(s(x)) $\rightarrow$ s(s(double(x))) \})

Question: How long can a $\rightarrow_{\mathcal{R}}$ sequence from a term of size $n$ become? (worst case)

Here: Does $\mathcal{R}$ have complexity $\Theta(n)$?
What is *Complexity* of Term Rewriting?

**Given:** TRS $\mathcal{R}$ (e.g., $\{ \text{double}(0) \rightarrow 0, \text{double}(s(x)) \rightarrow s(s(\text{double}(x))) \}$)

**Question:** How long can a $\rightarrow \mathcal{R}$ sequence from a term of size $n$ become? *(worst case)*

**Here:** Does $\mathcal{R}$ have complexity $\Theta(n)$?

(1) **Yes!**
What is *Complexity* of Term Rewriting?

**Given:** TRS $\mathcal{R}$ (e.g., \{ double(0) $\rightarrow$ 0, double(s(x)) $\rightarrow$ s(s(double(x))) \})

**Question:** How long can a $\rightarrow_{\mathcal{R}}$ sequence from a term of size $n$ become? (worst case)

**Here:** Does $\mathcal{R}$ have complexity $\Theta(n)$?

(1) Yes!

$$\text{double}(s^{n-2}(0)) \rightarrow_{\mathcal{R}}^{n-1} s^{2n-4}(0)$$
What is *Complexity* of Term Rewriting?

**Given:** TRS $\mathcal{R}$ (e.g., \{ \text{double}(0) \to 0, \text{double}(s(x)) \to s(s(\text{double}(x))) \})

**Question:** How long can a $\to_{\mathcal{R}}$ sequence from a term of size $n$ become?  
(worst case)

**Here:** Does $\mathcal{R}$ have complexity $\Theta(n)$?

(1) Yes!

$$\text{double}(s^{n-2}(0)) \to_{\mathcal{R}}^{n-1} s^{2n-4}(0)$$

- **basic terms** $f(t_1, \ldots, t_n)$ with $t_i$ constructor terms allow only $n$ steps
What is Complexity of Term Rewriting?

**Given:** TRS \( \mathcal{R} \) (e.g., \{ double(0) → 0, double(s(x)) → s(s(double(x))) \})

**Question:** How long can a \( \rightarrow \mathcal{R} \) sequence from a term of size \( n \) become? (worst case)

**Here:** Does \( \mathcal{R} \) have complexity \( \Theta(n) \)?

(1) Yes!

\[
\text{double}(s^{n-2}(0)) \rightarrow_{\mathcal{R}}^{n-1} s^{2n-4}(0)
\]

- **basic terms** \( f(t_1, \ldots, t_n) \) with \( t_i \) constructor terms allow only \( n \) steps
- **runtime complexity** \( rc_{\mathcal{R}}(n) \): basic terms as start terms
What is *Complexity* of Term Rewriting?

**Given:** TRS $\mathcal{R}$ (e.g., \{ double(0) → 0, double(s(x)) → s(s(double(x))) \})

**Question:** How long can a $\rightarrow_{\mathcal{R}}$ sequence from a term of size $n$ become? (worst case)

**Here:** Does $\mathcal{R}$ have complexity $\Theta(n)$?

(1) Yes!

$$\text{double}(s^{n-2}(0)) \rightarrow_{\mathcal{R}}^{n-1} s^{2n-4}(0)$$

- basic terms $f(t_1, \ldots, t_n)$ with $t_i$ constructor terms allow only $n$ steps
- runtime complexity $rc_{\mathcal{R}}(n)$: basic terms as start terms
- $rc_{\mathcal{R}}(n)$ for program analysis
What is Complexity of Term Rewriting?

Given: TRS $\mathcal{R}$ (e.g., $\{ \text{double}(0) \rightarrow 0, \text{double}(s(x)) \rightarrow s(s(\text{double}(x))) \}$)

Question: How long can a $\rightarrow_\mathcal{R}$ sequence from a term of size $n$ become? (worst case)

Here: Does $\mathcal{R}$ have complexity $\Theta(n)$?

(1) Yes!

$$\text{double}(s^{n-2}(0)) \rightarrow_{\mathcal{R}}^{n-1} s^{2n-4}(0)$$

- basic terms $f(t_1, \ldots, t_n)$ with $t_i$ constructor terms allow only $n$ steps
- runtime complexity $rc_{\mathcal{R}}(n)$: basic terms as start terms
- $rc_{\mathcal{R}}(n)$ for program analysis

(2) No!
What is *Complexity* of Term Rewriting?

**Given:** TRS $\mathcal{R}$ (e.g., $\{ \text{double}(0) \rightarrow 0, \text{double}(s(x)) \rightarrow s(s(\text{double}(x))) \}$)

**Question:** How long can a $\rightarrow_{\mathcal{R}}$ sequence from a term of size $n$ become? (worst case)

**Here:** Does $\mathcal{R}$ have complexity $\Theta(n)$?

1. **Yes!**
   
   $\text{double}(s^{n-2}(0)) \rightarrow_{\mathcal{R}}^{n-1} s^{2n-4}(0)$
   
   *basic terms* $f(t_1, \ldots, t_n)$ with $t_i$ *constructor terms* allow only $n$ steps

   *runtime complexity* $rc_{\mathcal{R}}(n)$: basic terms as start terms

   *rc$_{\mathcal{R}}(n)$ for program analysis*

2. **No!**

   $\text{double}^3(s(0)) \rightarrow_{\mathcal{R}}^2 \text{double}^2(s^2(0)) \rightarrow_{\mathcal{R}}^3 \text{double}(s^4(0)) \rightarrow_{\mathcal{R}}^5 s^8(0)$ in 10 steps
What is *Complexity* of Term Rewriting?

**Given:** TRS $\mathcal{R}$ (e.g., \{ double(0) $\to$ 0, double(s(x)) $\to$ s(s(double(x))) \})

**Question:** How long can a $\to_{\mathcal{R}}$ sequence from a term of size $n$ become? (worst case)

**Here:** Does $\mathcal{R}$ have complexity $\Theta(n)$?

1) **Yes!**

   \[
   \text{double}(s^{n-2}(0)) \rightarrow_{\mathcal{R}}^{n-1} s^{2n-4}(0)
   \]

   - basic terms $f(t_1, \ldots, t_n)$ with $t_i$ constructor terms allow only $n$ steps
   - runtime complexity $rc_{\mathcal{R}}(n)$: basic terms as start terms
   - $rc_{\mathcal{R}}(n)$ for program analysis

2) **No!**

   \[
   \text{double}^3(s(0)) \rightarrow_{\mathcal{R}}^2 \text{double}^2(s^2(0)) \rightarrow_{\mathcal{R}}^3 \text{double}(s^4(0)) \rightarrow_{\mathcal{R}}^5 s^8(0)
   \]

   - $\text{double}^{n-2}(s(0))$ allows $\Theta(2^n)$ many steps to $s^{2n-2}(0)$
What is *Complexity* of Term Rewriting?

**Given:** TRS $\mathcal{R}$ (e.g., \{ $\text{double}(0) \rightarrow 0$, $\text{double}(s(x)) \rightarrow s(s(\text{double}(x)))$ \})

**Question:** How long can a $\rightarrow_{\mathcal{R}}$ sequence from a term of size $n$ become? (worst case)

**Here:** Does $\mathcal{R}$ have complexity $\Theta(n)$?

1. **Yes!**
   \[
   \text{double}(s^{n-2}(0)) \rightarrow_{\mathcal{R}}^{n-1} s^{2n-4}(0)
   \]
   - basic terms $f(t_1, \ldots, t_n)$ with $t_i$ constructor terms allow only $n$ steps
   - runtime complexity $rc_{\mathcal{R}}(n)$: basic terms as start terms
   - $rc_{\mathcal{R}}(n)$ for program analysis

2. **No!**
   \[
   \text{double}^3(s(0)) \rightarrow_{\mathcal{R}}^2 \text{double}^2(s^2(0)) \rightarrow_{\mathcal{R}}^3 \text{double}(s^4(0)) \rightarrow_{\mathcal{R}}^5 s^8(0)
   \]
   - $\text{double}^{n-2}(s(0))$ allows $\Theta(2^n)$ many steps to $s^{2n-2}(0)$
   - derivational complexity $dc_{\mathcal{R}}(n)$: no restrictions on start terms
What is *Complexity* of Term Rewriting?

**Given:** TRS $\mathcal{R}$ (e.g., \{ $\text{double}(0) \rightarrow 0$, $\text{double}(s(x)) \rightarrow s(s(\text{double}(x)))$ \})

**Question:** How long can a $\rightarrow_{\mathcal{R}}$ sequence from a term of size $n$ become? (worst case)

**Here:** Does $\mathcal{R}$ have complexity $\Theta(n)$?

(1) Yes!

$$\text{double}(s^{n-2}(0)) \rightarrow_{\mathcal{R}}^{n-1} s^{2n-4}(0)$$

- basic terms $f(t_1, \ldots, t_n)$ with $t_i$ constructor terms allow only $n$ steps
- runtime complexity $rc_{\mathcal{R}}(n)$: basic terms as start terms
- $rc_{\mathcal{R}}(n)$ for program analysis

(2) No!

$$\text{double}^3(s(0)) \rightarrow_{\mathcal{R}}^2 \text{double}^2(s^2(0)) \rightarrow_{\mathcal{R}}^3 \text{double}(s^4(0)) \rightarrow_{\mathcal{R}}^5 s^8(0)$$ in 10 steps

- $\text{double}^{n-2}(s(0))$ allows $\Theta(2^n)$ many steps to $s^{2n-2}(0)$
- derivational complexity $dc_{\mathcal{R}}(n)$: no restrictions on start terms
- $dc_{\mathcal{R}}(n)$ for equational reasoning: cost of solving the word problem $\mathcal{E} \models s \equiv t$ by rewriting $s$ and $t$ via an equivalent convergent TRS $\mathcal{R}_E$
Complexity Analysis for TRSs: Overview

1. Introduction
2. Automatically Finding Upper Bounds
3. Automatically Finding Lower Bounds
4. Transformational Techniques
5. Analysing Program Complexity via TRS Complexity
6. Current Developments
1989: Derivational complexity introduced, linked to termination proofs\(^\dagger\)

\[^\dagger]\text{D. Hofbauer, C. Lautemann: } \textit{Termination proofs and the length of derivations, RTA ’89}
A Short Timeline (1/2)

1989: Derivational complexity introduced, linked to termination proofs\(^{18}\)

2001: Techniques for polynomial upper complexity bounds\(^{19}\)

\[^{18}\text{D. Hofbauer, C. Lautemann: } Termination proofs and the length of derivations, RTA '89}\]

\[^{19}\text{G. Bonfante, A. Cichon, J. Marion, and H. Touzet: } Algorithms with polynomial interpretation termination proof, JFP '01}\]
A Short Timeline (1/2)

1989: Derivational complexity introduced, linked to termination proofs

2001: Techniques for polynomial upper complexity bounds

2008: Runtime complexity introduced with first analysis techniques

---

18 D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA ’89

19 G. Bonfante, A. Cichon, J. Marion, and H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP ’01

20 N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR ’08
A Short Timeline (1/2)

1989: Derivational complexity introduced, linked to termination proofs\textsuperscript{18}

2001: Techniques for polynomial upper complexity bounds\textsuperscript{19}

2008: Runtime complexity introduced with first analysis techniques\textsuperscript{20}

2008: First automated tools to find complexity bounds: TcT\textsuperscript{21}, CaT\textsuperscript{22}

\textsuperscript{18} D. Hofbauer, C. Lautemann: \textit{Termination proofs and the length of derivations}, RTA '89

\textsuperscript{19} G. Bonfante, A. Cichon, J. Marion, and H. Touzet: \textit{Algorithms with polynomial interpretation termination proof}, JFP '01

\textsuperscript{20} N. Hirokawa, G. Moser: \textit{Automated complexity analysis based on the dependency pair method}, IJCAR '08


\textsuperscript{22} M. Korp, C. Sternagel, H. Zankl, A. Middeldorp: \textit{Tyrolean Termination Tool 2}, RTA '09, \url{http://cl-informatik.uibk.ac.at/software/cat/}
1989: Derivational complexity introduced, linked to termination proofs

2001: Techniques for polynomial upper complexity bounds

2008: Runtime complexity introduced with first analysis techniques

2008: First automated tools to find complexity bounds: TcT, CaT

2008: First complexity analysis categories in the Termination Competition

http://termination-portal.org/wiki/Termination_Competition

---

18 D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA ’89

19 G. Bonfante, A. Cichon, J. Marion, and H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP ’01

20 N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR ’08


A Short Timeline (1/2)

1989: Derivational complexity introduced, linked to termination proofs\textsuperscript{18}
2001: Techniques for polynomial upper complexity bounds\textsuperscript{19}
2008: Runtime complexity introduced with first analysis techniques\textsuperscript{20}
2008: First automated tools to find complexity bounds: TcT\textsuperscript{21}, CaT\textsuperscript{22}

First complexity analysis categories in the Termination Competition

http://termination-portal.org/wiki/Termination_Competition

\textsuperscript{18} D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA ’89
\textsuperscript{19} G. Bonfante, A. Cichon, J. Marion, and H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP ’01
\textsuperscript{20} N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR ’08
\textsuperscript{22} M. Korp, C. Sternagel, H. Zankl, A. Middeldorp: *Tyrolean Termination Tool 2*, RTA ’09, http://cl-informatik.uibk.ac.at/software/cat/
2022: Termination Competition 2022 with complexity analysis tools AProVE\textsuperscript{23}, TcT in August 2022

https://termcomp.github.io/Y2022

Some Definitions

Definition (Derivation Height $\text{dh}$)

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a relation $\rightarrow$, the \textit{derivation height} is:

$$\text{dh}(t, \rightarrow) = \sup \{ n \mid \exists t'. t \rightarrow^n t' \}$$

If $t$ starts an infinite $\rightarrow$-sequence, we set $\text{dh}(t, \rightarrow) = \omega$.
Some Definitions

**Definition (Derivation Height $dh$)**

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a relation $\rightarrow$, the **derivation height** is:

$$dh(t, \rightarrow) = \sup \{ n \mid \exists t'. t \rightarrow^n t' \}$$

If $t$ starts an infinite $\rightarrow$-sequence, we set $dh(t, \rightarrow) = \omega$.

$dh(t, \rightarrow)$: length of the longest $\rightarrow$-sequence from $t$. 

**Definition (Derivational Complexity $dc_R$)**

For a TRS $R$, the **derivational complexity** is:

$$dc_R(n) = \sup \{ dh(t, \rightarrow_R) \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{V}), |t| \leq n \}$$

$dc_R(n)$: length of the longest $\rightarrow_R$-sequence from a term of size at most $n$.
Some Definitions

Definition (Derivation Height $dh$)

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a relation $\rightarrow$, the derivation height is:

$$dh(t, \rightarrow) = \sup \{ n \mid \exists t'. t \rightarrow^n t' \}$$

If $t$ starts an infinite $\rightarrow$-sequence, we set $dh(t, \rightarrow) = \omega$.

$dh(t, \rightarrow)$: length of the longest $\rightarrow$-sequence from $t$.

Example: $dh(\text{double}(s(s(s(0)))), \rightarrow_{\mathcal{R}}) = 4$
**Some Definitions**

**Definition (Derivation Height \(dh\))**

For a term \(t \in \mathcal{T}(\mathcal{F}, \mathcal{V})\) and a relation \(\rightarrow\), the **derivation height** is:

\[
dh(t, \rightarrow) = \sup \{ n \mid \exists t'. t \rightarrow^n t' \}
\]

If \(t\) starts an infinite \(\rightarrow\)-sequence, we set \(dh(t, \rightarrow) = \omega\).

\(dh(t, \rightarrow)\): length of the longest \(\rightarrow\)-sequence from \(t\).

**Example:** \(dh(\text{double}(s(s(s(0)))), \rightarrow_\mathcal{R}) = 4\)

**Definition (Derivational Complexity \(dc\))**

For a TRS \(\mathcal{R}\), the **derivational complexity** is:

\[
dc_\mathcal{R}(n) = \sup \{ dh(t, \rightarrow_\mathcal{R}) \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{V}), |t| \leq n \}
\]
Some Definitions

Definition (Derivation Height $dh$)

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a relation $\rightarrow$, the **derivation height** is:

$$ dh(t, \rightarrow) = \sup \{ n \mid \exists t'. t \rightarrow^n t' \} $$

If $t$ starts an infinite $\rightarrow$-sequence, we set $dh(t, \rightarrow) = \omega$.

$dh(t, \rightarrow)$: length of the longest $\rightarrow$-sequence from $t$.

**Example:** $dh(\text{double}(s(s(s(0)))), \rightarrow_{\mathcal{R}}) = 4$

Definition (Derivational Complexity $dc$)

For a TRS $\mathcal{R}$, the **derivational complexity** is:

$$ dc_{\mathcal{R}}(n) = \sup \{ dh(t, \rightarrow_{\mathcal{R}}) \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{V}), |t| \leq n \} $$

$dc_{\mathcal{R}}(n)$: length of the longest $\rightarrow_{\mathcal{R}}$-sequence from a term of size at most $n$. 
Some Definitions

**Definition (Derivation Height \( dh \))**

For a term \( t \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \) and a relation \( \rightarrow \), the **derivation height** is:

\[
dh(t, \rightarrow) = \sup \{ n \mid \exists t'. t \rightarrow^n t' \}
\]

If \( t \) starts an infinite \( \rightarrow \)-sequence, we set \( \dh(t, \rightarrow) = \omega \).

\( \dh(t, \rightarrow) \): length of the longest \( \rightarrow \)-sequence from \( t \).

**Example:** \( \dh(\text{double}(\text{s}(\text{s}(\text{s}(\text{0}))))), \rightarrow_{\mathcal{R}}) = 4 \)

**Definition (Derivational Complexity \( dc \))**

For a TRS \( \mathcal{R} \), the **derivational complexity** is:

\[
dc_{\mathcal{R}}(n) = \sup \{ \dh(t, \rightarrow_{\mathcal{R}}) \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{V}), |t| \leq n \}
\]

\( dc_{\mathcal{R}}(n) \): length of the longest \( \rightarrow_{\mathcal{R}} \)-sequence from a term of size at most \( n \)

**Example:** For \( \mathcal{R} \) for \text{double}, we have \( dc_{\mathcal{R}}(n) \in \Theta(2^n) \).
The Bad News for automation:

For a given TRS $R$, the following questions are undecidable:

$dc_R(n) = \omega$ for some $n$? ($\rightarrow$ termination!)

$dc_R(n)$ polynomially bounded?

Goal: find approximations for derivational complexity

Initial focus: find upper bounds

$dc_R(n) \in O(...)$

A. Schnabl and J. G. Simonsen: The exact hardness of deciding derivational and runtime complexity, CSL '11
The Bad News for automation:

For a given TRS $\mathcal{R}$, the following questions are undecidable:

- $dc_{\mathcal{R}}(n) = \omega$ for some $n$? ($\rightarrow$ termination!)
The Bad News for automation:

For a given TRS $\mathcal{R}$, the following questions are undecidable:

- $d_{c\mathcal{R}}(n) = \omega$ for some $n$? ($\rightarrow$ termination!)
- $d_{c\mathcal{R}}(n)$ polynomially bounded?\(^{24}\)

\(^{24}\)A. Schnabl and J. G. Simonsen: *The exact hardness of deciding derivational and runtime complexity*, CSL ’11
The Bad News for automation:

For a given TRS $\mathcal{R}$, the following questions are undecidable:

- $dc_{\mathcal{R}}(n) = \omega$ for some $n$? ($\rightarrow$ termination!)
- $dc_{\mathcal{R}}(n)$ polynomially bounded?

Goal: find approximations for derivational complexity

Initial focus: find upper bounds

\[ dc_{\mathcal{R}}(n) \in \mathcal{O}(...) \]

---

\[ ^{24} \text{A. Schnabl and J. G. Simonsen: } The \text{ exact hardness of deciding derivational and runtime complexity, CSL '11} \]
**Example (double)**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double(0)</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>double(s(x))</code></td>
<td><code>s(s(double(x)))</code></td>
</tr>
</tbody>
</table>

Show \( dc_R(n) < \omega \) by termination proof with reduction order \( \succ \) on terms.

Get \( \succ \) via polynomial interpretation over \( \mathbb{N} \):

\[
\ell \succ r \iff [\ell] \succ [r]
\]

Example:

\[
\begin{align*}
[\text{double}] & (x) = 3 \cdot x, \\
[s] & (x) = x + 1,
\end{align*}
\]

Extend to terms:

\[
\begin{align*}
[x] & = x, \\
[f(t_1, \ldots, t_n)] & = [f([t_1], \ldots, [t_n])]
\end{align*}
\]

Automated search for \([\cdot]\) via SAT or SMT solving.

---

D. Lankford: Canonical algebraic simplification in computational logic, U Texas '75

C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, H. Zankl: SAT solving for termination analysis with polynomial interpretations, SAT '07

C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, A. Rubio: SAT modulo linear arithmetic for solving polynomial constraints, JAR '12
Example (double)

\[
\begin{align*}
double(0) & \succ 0 \\
double(s(x)) & \succ s(s(double(x)))
\end{align*}
\]

Show \(dc_R(n) < \omega\) by termination proof with reduction order \(\succ\) on terms.
Example (double)

\[
\begin{align*}
\text{double}(0) & \succ 0 \\
\text{double}(s(x)) & \succ s(s(\text{double}(x)))
\end{align*}
\]

Show \( dc_{\mathcal{R}}(n) < \omega \) by termination proof with reduction order \( \succ \) on terms. Get \( \succ \) via polynomial interpretation\(^{25}\) \([\cdot]\) over \(\mathbb{N}\): \( l \succ r \iff [l] \succ [r] \)

\(^{25}\)D. Lankford: *Canonical algebraic simplification in computational logic*, U Texas ’75
Example (double)

\[
\begin{align*}
\text{double}(0) & \succ 0 \\
\text{double}(s(x)) & \succ s(s(\text{double}(x)))
\end{align*}
\]

Show \( dc_R(n) < \omega \) by termination proof with reduction order \( \succ \) on terms. Get \( \succ \) via polynomial interpretation\(^{25}\) \([ \cdot ]\) over \( \mathbb{N} \): \( \ell \succ r \iff [\ell] \succ [r] \)

Example: \( [\text{double}](x) = 3 \cdot x \), \( [s](x) = x + 1 \), \( [0] = 1 \)

---

\(^{25}\) D. Lankford: *Canonical algebraic simplification in computational logic*, U Texas ’75
Example (double)

\[
\begin{align*}
\text{double}(0) & \succ 0 \\
\text{double}(s(x)) & \succ s(s(\text{double}(x)))
\end{align*}
\]

Show \( dc_\mathcal{R}(n) < \omega \) by termination proof with reduction order \( \succ \) on terms. Get \( \succ \) via polynomial interpretation\(^{25}\) \([\cdot]\) over \( \mathbb{N} \): \( \ell \succ r \iff [\ell] \succ [r] \)

Example: \([\text{double}](x) = 3 \cdot x\), \([s](x) = x + 1\), \([0] = 1\)

Extend to terms:

- \([x] = x\)
- \([f(t_1, \ldots, t_n)] = [f][[t_1], \ldots, [t_n]]\)

\(^{25}\) D. Lankford: *Canonical algebraic simplification in computational logic*, U Texas ’75
Derivational Complexity from Polynomial Interpretations (1/2)

Example (double)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{double}(0)</td>
<td>\succ</td>
<td>0</td>
</tr>
<tr>
<td>\text{double}(s(x))</td>
<td>\succ</td>
<td>s(s(\text{double}(x)))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>\succ</td>
<td>1</td>
</tr>
<tr>
<td>3 \cdot x + 3</td>
<td>\succ</td>
<td>3 \cdot x + 2</td>
</tr>
</tbody>
</table>

Show \( dc_R(n) < \omega \) by \textbf{termination proof} with reduction order \( \succ \) on terms. Get \( \succ \) via \textbf{polynomial interpretation} \[ \cdot \] over \( \mathbb{N} \): \( \ell \succ r \iff \llbracket \ell \rrbracket \succ \rrbracket r \rrbracket \)

Example: \( \llbracket \text{double} \rrbracket (x) = 3 \cdot x \), \( \llbracket s \rrbracket (x) = x + 1 \), \( \llbracket 0 \rrbracket = 1 \)

Extend to terms:

- \( \llbracket x \rrbracket = x \)
- \( \llbracket f(t_1, \ldots, t_n) \rrbracket = \llbracket f \rrbracket (\llbracket t_1 \rrbracket, \ldots, \llbracket t_n \rrbracket) \)

---

\(^{25}\) D. Lankford: \textit{Canonical algebraic simplification in computational logic}, U Texas ’75
Derivational Complexity from Polynomial Interpretations (1/2)

Example (double)

\[
\begin{align*}
\text{double}(0) & \succ 0 & 3 & > 1 \\
\text{double}(s(x)) & \succ s(s(\text{double}(x))) & 3 \cdot x + 3 & > 3 \cdot x + 2
\end{align*}
\]

Show \( dc_R(n) < \omega \) by termination proof with reduction order \( \succ \) on terms. Get \( \succ \) via polynomial interpretation\(^{25}\) \([\cdot]\) over \( \mathbb{N} \): \( \ell \succ r \iff [\ell] \succ [r] \)

Example: \([\text{double}](x) = 3 \cdot x\), \([s](x) = x + 1\), \([0] = 1\)

Extend to terms:

- \([x] = x\)
- \([f(t_1, \ldots, t_n)] = [f([t_1], \ldots, [t_n])]\)

Automated search for \([\cdot]\) via SAT\(^{26}\) or SMT\(^{27}\) solving

---

\(^{25}\)D. Lankford: *Canonical algebraic simplification in computational logic*, U Texas ’75

\(^{26}\)C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, H. Zankl: *SAT solving for termination analysis with polynomial interpretations*, SAT ’07

\(^{27}\)C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, A. Rubio: *SAT modulo linear arithmetic for solving polynomial constraints*, JAR ’12
### Example (double)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double(0)</code></td>
<td>$\succequiv 0$</td>
<td>$3 \succequiv 1$</td>
</tr>
<tr>
<td><code>double(s(x))</code></td>
<td>$\succequiv s(s(double(x)))$</td>
<td>$3 \cdot x + 3 \succequiv 3 \cdot x + 2$</td>
</tr>
</tbody>
</table>

**Example:**

\[
\text{[double]}(x) = 3 \cdot x, \quad [s](x) = x + 1, \quad [0] = 1
\]

This proves more than just termination...
Derivational Complexity from Polynomial Interpretations (2/2)

Example (double)

\[
\begin{align*}
\text{double}(0) & \succ 0 & 3 & > 1 \\
\text{double}(s(x)) & \succ s(s(\text{double}(x))) & 3 \cdot x + 3 & > 3 \cdot x + 2
\end{align*}
\]

Example: \([\text{double}] (x) = 3 \cdot x, \quad [s] (x) = x + 1, \quad [0] = 1\]

This proves more than just termination...

Theorem (Upper bounds for \(dc_\mathcal{R}(n)\) from polynomial interpretations\(^{28}\))

- Termination proof for TRS \(\mathcal{R}\) with polynomial interpretation

\[\Rightarrow dc_\mathcal{R}(n) \in 2^{2^{\mathcal{O}(n)}}\]

---

\(^{28}\) D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA ’89
Example (double)

\[
\begin{align*}
\text{double}(0) & \succ 0 & 3 & > 1 \\
\text{double}(s(x)) & \succ s(s(\text{double}(x))) & 3 \cdot x + 3 & > 3 \cdot x + 2
\end{align*}
\]

Example:

\[
\begin{align*}
[\text{double}](x) = 3 \cdot x, & \quad [s](x) = x + 1, & \quad [0] = 1
\end{align*}
\]

This proves more than just termination...

Theorem (Upper bounds for \(dc_R(n)\) from polynomial interpretations\(^{28}\))

- Termination proof for TRS \(R\) with polynomial interpretation
  \[
  \Rightarrow dc_R(n) \in 2^{O(n)}
  \]
- Termination proof for TRS \(R\) with linear polynomial interpretation
  \[
  \Rightarrow dc_R(n) \in 2^{O(n)}
  \]

\(^{28}\) D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA ’89
Termination proof for TRS $\mathcal{R}$ with …

- matchbounds\(^{29}\)
  \[ \Rightarrow dc_\mathcal{R}(n) \in \mathcal{O}(n) \]
- arctic matrix interpretations\(^{30}\)
  \[ \Rightarrow dc_\mathcal{R}(n) \in \mathcal{O}(n) \]

---

\(^{29}\) A. Geser, D. Hofbauer, J. Waldmann: *Match-bounded string rewriting systems*, AAECC ’04

\(^{30}\) A. Koprowski, J. Waldmann: *Max/plus tree automata for termination of term rewriting*, Acta Cyb. ’09
Termination proof for TRS $\mathcal{R}$ with...

- matchbounds\(^{29}\)
  \[ \Rightarrow dc_{\mathcal{R}}(n) \in \mathcal{O}(n) \]
- arctic matrix interpretations\(^{30}\)
  \[ \Rightarrow dc_{\mathcal{R}}(n) \in \mathcal{O}(n) \]
- triangular matrix interpretation\(^{31}\)
  \[ \Rightarrow dc_{\mathcal{R}}(n) \text{ is at most polynomial} \]
- matrix interpretation of spectral radius\(^{32}\) \(\leq 1\)
  \[ \Rightarrow dc_{\mathcal{R}}(n) \text{ is at most polynomial} \]

---

\(^{29}\) A. Geser, D. Hofbauer, J. Waldmann: *Match-bounded string rewriting systems*, AAECC '04

\(^{30}\) A. Koprowski, J. Waldmann: *Max/plus tree automata for termination of term rewriting*, Acta Cyb. '09

\(^{31}\) G. Moser, A. Schnabl, J. Waldmann: *Complexity analysis of term rewriting based on matrix and context dependent interpretations*, FSTTCS '08

\(^{32}\) F. Neurauter, H. Zankl, A. Middeldorp: *Revisiting matrix interpretations for polynomial derivational complexity of term rewriting*, LPAR (Yogyakarta) '10
Derivational Complexity from Termination Proofs (1/2)

Termination proof for TRS $\mathcal{R}$ with …

- matchbounds\(^{29}\) \[ \Rightarrow dc_\mathcal{R}(n) \in O(n) \]
- arctic matrix interpretations\(^{30}\) \[ \Rightarrow dc_\mathcal{R}(n) \in O(n) \]
- triangular matrix interpretation\(^{31}\) \[ \Rightarrow dc_\mathcal{R}(n) \text{ is at most polynomial} \]
- matrix interpretation of spectral radius\(^{32}\) $\leq 1$ \[ \Rightarrow dc_\mathcal{R}(n) \text{ is at most polynomial} \]
- standard matrix interpretation\(^{33}\) \[ \Rightarrow dc_\mathcal{R}(n) \text{ is at most exponential} \]

\(^{29}\) A. Geser, D. Hofbauer, J. Waldmann: *Match-bounded string rewriting systems*, AAECC ’04

\(^{30}\) A. Koprowski, J. Waldmann: *Max/plus tree automata for termination of term rewriting*, Acta Cyb. ’09

\(^{31}\) G. Moser, A. Schnabl, J. Waldmann: *Complexity analysis of term rewriting based on matrix and context dependent interpretations*, FSTTCS ’08

\(^{32}\) F. Neurauter, H. Zankl, A. Middeldorp: *Revisiting matrix interpretations for polynomial derivational complexity of term rewriting*, LPAR (Yogyakarta) ’10

\(^{33}\) J. Endrullis, J. Waldmann, and H. Zantema: *Matrix interpretations for proving termination of term rewriting*, JAR ’08
Termination proof for TRS $\mathcal{R}$ with...

- lexicographic path order$^{34}$ \quad $\Rightarrow$ \quad $dc_{\mathcal{R}}(n)$ is at most multiple recursive$^{35}$

---

$^{34}$ S. Kamin, J.-J. Lévy: *Two generalizations of the recursive path ordering*, U Illinois '80

$^{35}$ A. Weiermann: *Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths*, TCS '95
Termination proof for TRS $\mathcal{R}$ with . . .

- lexicographic path order$^{34}$ \implies $d_{c\mathcal{R}}(n)$ is at most multiple recursive$^{35}$
- Dependency Pairs method$^{36}$ with dependency graphs and usable rules \implies $d_{c\mathcal{R}}(n)$ is at most primitive recursive$^{37}$

---

$^{34}$ S. Kamin, J.-J. Lévy: *Two generalizations of the recursive path ordering*, U Illinois '80
$^{35}$ A. Weiermann: *Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths*, TCS '95
$^{36}$ T. Arts, J. Giesl: *Termination of term rewriting using dependency pairs*, TCS '00
$^{37}$ G. Moser, A. Schnabl: *The derivational complexity induced by the dependency pair method*, LMCS '11
Termination proof for TRS $\mathcal{R}$ with . . .

- lexicographic path order\(^{34}\) $\Rightarrow d_{c\mathcal{R}}(n)$ is at most multiple recursive\(^{35}\)
- Dependency Pairs method\(^{36}\) with dependency graphs and usable rules $\Rightarrow d_{c\mathcal{R}}(n)$ is at most primitive recursive\(^{37}\)
- Dependency Pairs framework\(^{38}\)\(^{39}\) with dependency graphs, reduction pairs, subterm criterion $\Rightarrow d_{c\mathcal{R}}(n)$ is at most multiple recursive\(^{40}\)

---

\(^{34}\) S. Kamin, J.-J. Lévy: Two generalizations of the recursive path ordering, U Illinois ’80
\(^{35}\) A. Weiermann: Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths, TCS ’95
\(^{36}\) T. Arts, J. Giesl: Termination of term rewriting using dependency pairs, TCS ’00
\(^{37}\) G. Moser, A. Schnabl: The derivational complexity induced by the dependency pair method, LMCS ’11
\(^{38}\) J. Giesl, R. Thiemann, P. Schneider-Kamp, S. Falke: Mechanizing and improving dependency pairs, JAR ’06
\(^{39}\) N. Hirokawa and A. Middeldorp: Tyrolean Termination Tool: Techniques and features, IC ’07
\(^{40}\) G. Moser, A. Schnabl: Termination proofs in the dependency pair framework may induce multiple recursive derivational complexity, RTA ’11
So far: upper bounds for derivational complexity

Definition (Basic Term)

For defined symbols $D$ and constructor symbols $C$, the term $f(t_1, ..., t_n)$ is in the set $T_{basic}$ of basic terms iff $f \in D$ and $t_1, ..., t_n \in T(C, V)$.

Definition (Runtime Complexity)

For a TRS $R$, the runtime complexity is:

$$rc_R(n) = \sup \{ dh(t, \rightarrow_R) \mid t \in T_{basic}, |t| \leq n \}$$

$rc_R(n)$: like derivational complexity... but for basic terms only!

---

N. Hirokawa, G. Moser: Automated complexity analysis based on the dependency pair method, IJCAR '08
Runtime Complexity

- So far: upper bounds for derivational complexity
- But: derivational complexity counter-intuitive, often infeasible
Runtime Complexity

- So far: upper bounds for derivational complexity
- But: derivational complexity counter-intuitive, often infeasible
- Wanted: complexity of evaluation of double on data: \(\text{double}(s^n(0))\)

---

Definition (Basic Term 41)

For defined symbols \(D\) and constructor symbols \(C\), the term \(f(t_1, \ldots, t_n)\) is in the set \(T_{\text{basic}}\) of basic terms iff \(f \in D\) and \(t_1, \ldots, t_n \in T(C, V)\).

Definition (Runtime Complexity 41)

For a TRS \(R\), the runtime complexity is:

\[
rc_R(n) = \sup \{ dh(t, \rightarrow_R) \mid t \in T_{\text{basic}}, |t| \leq n \}
\]

\(rc_R(n)\): like derivational complexity...

...but for basic terms only!

---

N. Hirokawa, G. Moser: Automated complexity analysis based on the dependency pair method, IJCAR '08
Runtime Complexity

- So far: upper bounds for derivational complexity
- But: derivational complexity counter-intuitive, often infeasible
- Wanted: complexity of evaluation of \texttt{double on data}: \texttt{double(s^n(0))}

\noindent \textbf{Definition (Basic Term\textsuperscript{41})}

For \texttt{defined symbols} $\mathcal{D}$ and \texttt{constructor symbols} $\mathcal{C}$, the term $f(t_1, \ldots, t_n)$ is in the set $\mathcal{T}_{\text{basic}}$ of \texttt{basic terms} iff $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$.

\textsuperscript{41}N. Hirokawa, G. Moser: \textit{Automated complexity analysis based on the dependency pair method}, IJCAR ’08
Runtime Complexity

- So far: upper bounds for derivational complexity
- But: derivational complexity counter-intuitive, often infeasible
- Wanted: complexity of evaluation of double on data: \(\text{double}(s^n(0))\)

### Definition (Basic Term\(^{41}\))

For defined symbols \(D\) and constructor symbols \(C\), the term

\[
f(t_1, \ldots, t_n)
\]

is in the set \(T_{\text{basic}}\) of basic terms iff \(f \in D\) and \(t_1, \ldots, t_n \in T(C, V)\).

### Definition (Runtime Complexity \(rc^{41}\))

For a TRS \(\mathcal{R}\), the runtime complexity is:

\[
rc_{\mathcal{R}}(n) = \sup \{ \text{dh}(t, \rightarrow_{\mathcal{R}}) \mid t \in T_{\text{basic}}, |t| \leq n \}
\]

---

\(^{41}\) N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR ’08
Runtime Complexity

- So far: upper bounds for derivational complexity
- But: derivational complexity counter-intuitive, often infeasible
- Wanted: complexity of evaluation of `double` on data: `double(s^n(0))`

**Definition (Basic Term\textsuperscript{41})**

For defined symbols $\mathcal{D}$ and constructor symbols $\mathcal{C}$, the term

$$f(t_1, \ldots, t_n)$$

is in the set $\mathcal{T}_{\text{basic}}$ of basic terms iff $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$.

**Definition (Runtime Complexity $rc$\textsuperscript{41})**

For a TRS $\mathcal{R}$, the runtime complexity is:

$$rc_{\mathcal{R}}(n) = \sup \{ dh(t, \rightarrow_{\mathcal{R}}) \mid t \in \mathcal{T}_{\text{basic}}, |t| \leq n \}$$

$rc_{\mathcal{R}}(n)$: like derivational complexity... but for basic terms only!

\textsuperscript{41} N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR ’08
Runtime Complexity from Polynomial Interpretations

Polynomial interpretations can induce upper bounds to runtime complexity:\textsuperscript{42}

**Definition (Strongly linear polynomial, restricted interpretation)**

- Polynomial $p$ is **strongly linear** iff
  \[ p(x_1, \ldots, x_n) = x_1 + \cdots + x_n + a \text{ for some } a \in \mathbb{N}. \]
- Polynomial interpretation $[\cdot]$ is **restricted** iff
  for all constructor symbols $f$, $[f](x_1, \ldots, x_n)$ is strongly linear.

Idea: $[t] \leq c \cdot |t|$ for fixed $c \in \mathbb{N}$.

\textsuperscript{42} G. Bonfante, A. Cichon, J. Marion, H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP '01
Runtime Complexity from Polynomial Interpretations

Polynomial interpretations can induce upper bounds to runtime complexity: ⁴²

**Definition (Strongly linear polynomial, restricted interpretation)**

- Polynomial $p$ is **strongly linear** iff
  $$p(x_1, \ldots, x_n) = x_1 + \cdots + x_n + a$$
  for some $a \in \mathbb{N}$.
- Polynomial interpretation $[\cdot]$ is **restricted** iff
  for all constructor symbols $f$, $[f](x_1, \ldots, x_n)$ is strongly linear.

**Idea:** $[t] \leq c \cdot |t|$ for fixed $c \in \mathbb{N}$.

**Theorem (Upper bounds for $rc_R(n)$ from restricted interpretations)**

Termination proof for TRS $R$ with **restricted** interpretation $[\cdot]$ of degree at most $d$ for $[f]$  

$$\Rightarrow rc_R(n) \in \mathcal{O}(n^d)$$

---

⁴² G. Bonfante, A. Cichon, J. Marion, H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP '01
Runtime Complexity from Polynomial Interpretations

Polynomial interpretations can induce upper bounds to runtime complexity:\(^{42}\)

**Definition (Strongly linear polynomial, restricted interpretation)**
- Polynomial \( p \) is **strongly linear** iff
  \[ p(x_1, \ldots, x_n) = x_1 + \cdots + x_n + a \] for some \( a \in \mathbb{N} \).
- Polynomial interpretation \([\cdot]\) is **restricted** iff
  for all constructor symbols \( f \), \( [f](x_1, \ldots, x_n) \) is strongly linear.

Idea: \([t] \leq c \cdot |t|\) for fixed \( c \in \mathbb{N} \).

**Theorem (Upper bounds for \( rc_{\mathbb{R}}(n) \) from restricted interpretations)**

Termination proof for TRS \( \mathbb{R} \) with **restricted** interpretation \([\cdot]\) of degree at most \( d \) for \([f]\)

\[ \Rightarrow rc_{\mathbb{R}}(n) \in \mathcal{O}(n^d) \]

**Example:** \([\text{double}](x) = 3 \cdot x\), \([\text{s}](x) = x + 1\), \([0] = 1\) is restricted, degree 1

\[ \Rightarrow rc_{\mathbb{R}}(n) \in \mathcal{O}(n) \] for TRS \( \mathbb{R} \) for **double**

---

\(^{42}\) G. Bonfante, A. Cichon, J. Marion, H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP ’01
Dependency Tuples for *Innermost* Runtime Complexity irc

Here: innermost rewriting (≈ call-by-value)

Example (reverse)

<table>
<thead>
<tr>
<th>Function</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>app(nil, y)</td>
<td>→ y</td>
<td>app(add(n, x), y) → add(n, app(x, y))</td>
</tr>
<tr>
<td>reverse(nil)</td>
<td>→ nil</td>
<td>reverse(add(n, x)) → app(reverse(x), add(n, nil))</td>
</tr>
</tbody>
</table>

For rule ℓ → r, eval of ℓ costs 1 + eval of all function calls in r together.
Dependency Tuples for *Innermost* Runtime Complexity irc

Here: innermost rewriting (≈ call-by-value)

**Example (reverse)**

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{app}(\text{nil}, y)) → (y)</td>
<td>(\text{app}(\text{add}(n, x), y)) → (\text{add}(n, \text{app}(x, y)))</td>
</tr>
<tr>
<td>(\text{reverse}(\text{nil})) → (\text{nil})</td>
<td>(\text{reverse}(\text{add}(n, x))) → (\text{app}(\text{reverse}(x), \text{add}(n, \text{nil})))</td>
</tr>
</tbody>
</table>

For rule \(\ell \rightarrow r\), eval of \(\ell\) costs \(1 +\) eval of all function calls in \(r\) **together**.

---

43 L. Noschinski, F. Emmes, J. Giesl: *Analyzing innermost runtime complexity of term rewriting by dependency pairs*, JAR '13
Dependency Tuples for *Innermost* Runtime Complexity irc

Here: innermost rewriting (≈ call-by-value)

Example (reverse)

\[
\begin{align*}
\text{app}(\text{nil}, y) & \rightarrow y \\
\text{reverse}(\text{nil}) & \rightarrow \text{nil}
\end{align*}
\]

\[
\begin{align*}
\text{app}(\text{add}(n, x), y) & \rightarrow \text{add}(n, \text{app}(x, y)) \\
\text{reverse}(\text{add}(n, x)) & \rightarrow \text{app}(\text{reverse}(x), \text{add}(n, \text{nil}))
\end{align*}
\]

For rule \( \ell \rightarrow r \), eval of \( \ell \) costs 1 + eval of all function calls in \( r \) together:

Example (Dependency Tuples\(^43\) for reverse)

\[
\begin{align*}
\text{app}(\text{nil}, y) & \rightarrow \text{Com}_0 \\
\text{app}(\text{add}(n, x), y) & \rightarrow \text{Com}_1(\text{app}(x, y)) \\
\text{reverse}(\text{nil}) & \rightarrow \text{Com}_0 \\
\text{reverse}(\text{add}(n, x)) & \rightarrow \text{Com}_2(\text{app}(\text{reverse}(x), \text{add}(n, \text{nil})), \text{reverse}(x))
\end{align*}
\]

- Function calls to count marked with \#
- Compound symbols \( \text{Com}_k \) group function calls together

---

\(^43\) L. Noschinski, F. Emmes, J. Giesl: *Analyzing innermost runtime complexity of term rewriting by dependency pairs*, JAR '13
### Example (reverse, Dependency Tuples for reverse)

\[
\begin{align*}
  \text{app}^{\#}(\text{nil}, y) & \rightarrow \text{Com}_0 \\
  \text{app}^{\#}(\text{add}(n, x), y) & \rightarrow \text{Com}_1(\text{app}^{\#}(x, y)) \\
  \text{reverse}^{\#}(\text{nil}) & \rightarrow \text{Com}_0 \\
  \text{reverse}^{\#}(\text{add}(n, x)) & \rightarrow \text{Com}_2(\text{app}^{\#}(\text{reverse}(x), \text{add}(n, \text{nil})), \text{reverse}^{\#}(x)) \\
  \text{app}(\text{nil}, y) & \rightarrow y \\
  \text{app}(\text{add}(n, x), y) & \rightarrow \text{add}(n, \text{app}(x, y)) \\
  \text{reverse}(\text{nil}) & \rightarrow \text{nil} \\
  \text{reverse}(\text{add}(n, x)) & \rightarrow \text{app}(\text{reverse}(x), \text{add}(n, \text{nil}))
\end{align*}
\]
Polynomial Interpretations for Dependency Tuples

Example (reverse, Dependency Tuples for reverse)

\[
\begin{align*}
\text{app}(\text{nil}, y) & \rightarrow \text{Com}_0 \\
\text{app}(\text{add}(n, x), y) & \rightarrow \text{Com}_1(\text{app}(x, y)) \\
\text{reverse}(\text{nil}) & \rightarrow \text{Com}_0 \\
\text{reverse}(\text{add}(n, x)) & \rightarrow \text{Com}_2(\text{app}(\text{reverse}(x), \text{add}(n, \text{nil})), \text{reverse}(x)) \\
\text{app}(\text{nil}, y) & \rightarrow y \\
\text{app}(\text{add}(n, x), y) & \rightarrow \text{add}(n, \text{app}(x, y)) \\
\text{reverse}(\text{nil}) & \rightarrow \text{nil} \\
\text{reverse}(\text{add}(n, x)) & \rightarrow \text{app}(\text{reverse}(x), \text{add}(n, \text{nil}))
\end{align*}
\]

Use interpretation \([ \cdot ]\) with \([\text{Com}_k](x_1, \ldots, x_k) = x_1 + \cdots + x_k\) and

\[
\begin{align*}
[\text{nil}] &= 0 \\
[\text{app}](x_1, x_2) &= x_1 + x_2 \\
[\text{app}^#](x_1, x_2) &= x_1 + 1 \\
[\text{add}](x_1, x_2) &= x_2 + 1 \quad (\leq \text{restricted interpret.}) \\
[\text{reverse}](x_1) &= x_1 \quad (\text{bounds helper fct. result size}) \\
[\text{reverse}^#](x_1) &= x_1^2 + x_1 + 1 \quad (\text{complexity of fct.})
\end{align*}
\]

to show \([\ell] \geq [r]\) for all rules and \([\ell] \geq 1 + [r]\) for all Dependency Tuples

Maximum degree of \([ \cdot ]\) is 2 \(\Rightarrow \operatorname{irc}_R(n) \in \mathcal{O}(n^2)\)
Related Techniques

- Dependency Tuples are an adaptation of Dependency Pairs (DPs) from termination analysis to complexity analysis, allow for **incremental** complexity proofs with several techniques.
Related Techniques

- Dependency Tuples are an adaptation of Dependency Pairs (DPs) from termination analysis to complexity analysis, allow for **incremental** complexity proofs with several techniques.

- Further adaptation of DPs (incomparable): Weak (Innermost) Dependency Pairs for (innermost) runtime complexity\(^{44}\)

\(^{44}\) N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR '08
Related Techniques

- Dependency Tuples are an adaptation of Dependency Pairs (DPs) from termination analysis to complexity analysis, allow for **incremental** complexity proofs with several techniques
- Further adaptation of DPs (incomparable): Weak (Innermost) Dependency Pairs for (innermost) runtime complexity\textsuperscript{44}
- Extensions by polynomial path orders\textsuperscript{45}, usable replacement maps\textsuperscript{46}, a combination framework for complexity analysis\textsuperscript{47}, …

\textsuperscript{44} N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR ’08
\textsuperscript{45} M. Avanzini, G. Moser: *Dependency pairs and polynomial path orders*, RTA ’09
\textsuperscript{46} N. Hirokawa, G. Moser: *Automated complexity analysis based on context-sensitive rewriting*, RTA-TLCA ’14
\textsuperscript{47} M. Avanzini, G. Moser: *A combination framework for complexity*, IC ’16
How about Lower Bounds for Complexity?

Why lower bounds?
- Get tight bounds with upper bounds
- Can indicate implementation bugs
- Security: single query can trigger Denial of Service

Here: Two techniques for finding lower bounds

F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder: Lower bounds for runtime complexity of term rewriting, JAR '17
How about Lower Bounds for Complexity?

Why lower bounds?
- get tight bounds with upper bounds
- can indicate implementation bugs
- security: single query can trigger Denial of Service

Graph showing runtime vs input size with upper bound, worst case, best case, and lower bound curves.

F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder: Lower bounds for runtime complexity of term rewriting, JAR '17
How about Lower Bounds for Complexity?

Why lower bounds?
- get **tight bounds** with upper bounds
- can indicate implementation bugs
- security: single query can trigger Denial of Service

Here: Two techniques for finding lower bounds\(^{48}\) inspired by proving non-termination

---

\(^{48}\) F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder: *Lower bounds for runtime complexity of term rewriting*, JAR ’17
Finding Lower Bounds by Induction

(1) Induction technique, inspired by non-looping non-termination\(^{49}\)
Finding Lower Bounds by Induction

(1) Induction technique, inspired by non-looping non-termination\footnote{F. Emmes, T. Enger, J. Giesl: Proving non-looping non-termination automatically, IJCAR ’12} non-termination

- Generate infinite family $\mathcal{T}_{\text{witness}}$ of basic terms as witnesses in

$$\forall n \in \mathbb{N}. \ \exists t_n \in \mathcal{T}_{\text{witness}}. \ |t_n| \leq q(n) \ \land \ \text{dh}(t_n, \rightarrow_{\mathcal{R}}) \geq p(n)$$

to conclude $r_{c_{\mathcal{R}}}(n) \in \Omega(p'(n))$. 

Finding Lower Bounds by Induction

(1) Induction technique, inspired by non-looping non-termination\textsuperscript{49}

- Generate infinite family $\mathcal{T}_{\text{witness}}$ of basic terms as witnesses in

$$\forall n \in \mathbb{N}. \exists t_n \in \mathcal{T}_{\text{witness}}. |t_n| \leq q(n) \land \text{dh}(t_n, \rightarrow_R) \geq p(n)$$


to conclude $r_{c_R}(n) \in \Omega(p'(n))$.

- Constructor terms for arguments can be built recursively after type inference: $0, s(0), s(s(0)), \ldots$ (here $q(n) = n + 1$, often linear)

---

\textsuperscript{49} F. Emmes, T. Enger, J. Giesl: \textit{Proving non-looping non-termination automatically}, IJCAR ’12
Finding Lower Bounds by Induction

(1) Induction technique, inspired by non-looping non-termination\textsuperscript{49}

- Generate infinite family $T_{\text{witness}}$ of basic terms as witnesses in
\[
\forall n \in \mathbb{N}. \ \exists t_n \in T_{\text{witness}}. \ |t_n| \leq q(n) \ \land \ \text{dh}(t_n, \rightarrow_R) \geq p(n)
\]
to conclude $\text{rc}_R(n) \in \Omega(p'(n))$.
- Constructor terms for arguments can be built recursively after type inference: $0, s(0), s(s(0)), \ldots$ (here $q(n) = n + 1$, often linear)
- Evaluate $t_n$ by narrowing, get rewrite sequences with recursive calls

\textsuperscript{49} F. Emmes, T. Enger, J. Giesl: Proving non-looping non-termination automatically, IJCAR ’12
Finding Lower Bounds by Induction

(1) Induction technique, inspired by non-looping non-termination\(^{49}\)

- Generate infinite family \(\mathcal{T}_{\text{witness}}\) of basic terms as witnesses in

\[
\forall n \in \mathbb{N}. \ \exists t_n \in \mathcal{T}_{\text{witness}}. \ |t_n| \leq q(n) \ \land \ dh(t_n, \rightarrow_{\mathcal{R}}) \geq p(n)
\]

to conclude \(rc_{\mathcal{R}}(n) \in \Omega(p'(n))\).

- Constructor terms for arguments can be built recursively after type inference: \(0, s(0), s(s(0)), \ldots\) (here \(q(n) = n + 1\), often linear)

- Evaluate \(t_n\) by narrowing, get rewrite sequences with recursive calls

- Speculate polynomial \(p(n)\) based on values for \(n = 0, 1, \ldots, k\)

\(^{49}\) F. Emmes, T. Enger, J. Giesl: Proving non-looping non-termination automatically, IJCAR '12
Finding Lower Bounds by Induction

(1) Induction technique, inspired by non-looping non-termination\textsuperscript{49}

- Generate infinite family $T_{\text{witness}}$ of basic terms as witnesses in

$$\forall n \in \mathbb{N}. \ \exists t_n \in T_{\text{witness}}. \ |t_n| \leq q(n) \land \text{dh}(t_n, \rightarrow_R) \geq p(n)$$

- Constructor terms for arguments can be built recursively after type inference: $0, s(0), s(s(0)), \ldots$ (here $q(n) = n + 1$, often linear)

- Evaluate $t_n$ by narrowing, get rewrite sequences with recursive calls

- Speculate polynomial $p(n)$ based on values for $n = 0, 1, \ldots, k$

- Prove rewrite lemma $t_n \rightarrow_R^{\geq p(n)} t'_n$ inductively

\textsuperscript{49} F. Emmes, T. Enger, J. Giesl: \textit{Proving non-looping non-termination automatically}, IJCAR ’12
Finding Lower Bounds by Induction

(1) Induction technique, inspired by **non-looping** non-termination\(^{49}\)

- Generate infinite family \(\mathcal{T}_{\text{witness}}\) of basic terms as witnesses in

\[
\forall n \in \mathbb{N}. \ \exists t_n \in \mathcal{T}_{\text{witness}}. \ |t_n| \leq q(n) \land \text{dh}(t_n, \rightarrow_R) \geq p(n)
\]


to conclude \(rc_R(n) \in \Omega(p'(n))\).

- Constructor terms for arguments can be built recursively after type inference:

\(0, s(0), s(s(0)), \ldots\) (here \(q(n) = n + 1\), often linear)

- Evaluate \(t_n\) by narrowing, get rewrite sequences with recursive calls

- Speculate polynomial \(p(n)\) based on values for \(n = 0, 1, \ldots, k\)

- Prove rewrite lemma \(t_n \rightarrow \overset{\geq p(n)}{R} t'_n\) inductively

- Get lower bound for \(rc_R(n)\) from \(p(n)\) in rewrite lemma and \(q(n)\)

---

\(^{49}\) F. Emmes, T. Enger, J. Giesl: *Proving non-looping non-termination automatically*, IJCAR ’12
Example (quicksort)

\[
\begin{align*}
qs(\text{nil}) & \rightarrow \text{nil} \\
qs(\text{cons}(x, xs)) & \rightarrow qs(\text{low}(x, xs)) ++ \text{cons}(x, qs(\text{low}(x, xs))) \\
\text{low}(x, \text{nil}) & \rightarrow \text{nil} \\
\text{low}(x, \text{cons}(y, ys)) & \rightarrow \text{if}(x \leq y, x, \text{cons}(y, ys)) \\
\text{if}(\text{tt}, x, \text{cons}(y, ys)) & \rightarrow \text{low}(x, ys) \\
\text{if}(\text{ff}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{low}(x, ys)) \\
& \ldots
\end{align*}
\]
## Example (quicksort)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qs(nil)</code></td>
<td><code>nil</code></td>
</tr>
<tr>
<td><code>qs(cons(x, xs))</code></td>
<td><code>qs(low(x, xs)) ++ cons(x, qs(low(x, xs)))</code></td>
</tr>
<tr>
<td><code>low(x, nil)</code></td>
<td><code>nil</code></td>
</tr>
<tr>
<td><code>low(x, cons(y, ys))</code></td>
<td><code>if(x ≤ y, x, cons(y, ys))</code></td>
</tr>
<tr>
<td><code>if(tt, x, cons(y, ys))</code></td>
<td><code>low(x, ys)</code></td>
</tr>
<tr>
<td><code>if(ff, x, cons(y, ys))</code></td>
<td><code>cons(y, low(x, ys))</code></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Speculate and prove rewrite lemma:

\[
qs(\text{cons}(\text{zero}, \ldots, \text{cons}(\text{zero}, \text{nil}))) \rightarrow 3n^2 + 2n + 1 \quad \text{cons}(\text{zero}, \ldots, \text{cons}(\text{zero}, \text{nil}))
\]
Example (quicksort)

- \( qs(\text{nil}) \rightarrow \text{nil} \)
- \( qs(\text{cons}(x, xs)) \rightarrow qs(\text{low}(x, xs)) ++ \text{cons}(x, qs(\text{low}(x, xs))) \)
- \( \text{low}(x, \text{nil}) \rightarrow \text{nil} \)
- \( \text{low}(x, \text{cons}(y, ys)) \rightarrow \text{if}(x \leq y, x, \text{cons}(y, ys)) \)
- \( \text{if}(\text{tt}, x, \text{cons}(y, ys)) \rightarrow \text{low}(x, ys) \)
- \( \text{if}(\text{ff}, x, \text{cons}(y, ys)) \rightarrow \text{cons}(y, \text{low}(x, ys)) \)

... 

Speculate and prove rewrite lemma:

- \( qs(\text{cons}(\text{zero}, \ldots, \text{cons}(\text{zero}, \text{nil}))) \rightarrow 3n^2 + 2n + 1 \) \( \text{cons}(\text{zero}, \ldots, \text{cons}(\text{zero}, \text{nil})) \)
- \( qs(\text{cons}^n(\text{zero}, \text{nil})) \rightarrow 3n^2 + 2n + 1 \) \( \text{cons}(\text{zero}, \ldots, \text{cons}(\text{zero}, \text{nil})) \)
### Example (quicksort)

<table>
<thead>
<tr>
<th>Function</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(qs(\text{nil}))</td>
<td>(\rightarrow \text{nil})</td>
</tr>
<tr>
<td>(qs(\text{cons}(x, xs)))</td>
<td>(\rightarrow qs(\text{low}(x, xs)) + \text{cons}(x, qs(\text{low}(x, xs))))</td>
</tr>
<tr>
<td>(\text{low}(x, \text{nil}))</td>
<td>(\rightarrow \text{nil})</td>
</tr>
<tr>
<td>(\text{low}(x, \text{cons}(y, ys)))</td>
<td>(\rightarrow \text{if}(x \leq y, x, \text{cons}(y, ys)))</td>
</tr>
<tr>
<td>(\text{if}(\text{tt}, x, \text{cons}(y, ys)))</td>
<td>(\rightarrow \text{low}(x, ys))</td>
</tr>
<tr>
<td>(\text{if}(\text{ff}, x, \text{cons}(y, ys)))</td>
<td>(\rightarrow \text{cons}(y, \text{low}(x, ys)))</td>
</tr>
</tbody>
</table>

\[\ldots\]

### Speculate and prove rewrite lemma:

\[
qs(\text{cons}(\text{zero}, \ldots, \text{cons}(\text{zero}, \text{nil}))) \rightarrow 3n^2 + 2n + 1 \quad \text{cons}(\text{zero}, \ldots, \text{cons}(\text{zero}, \text{nil}))
\]

\[
qs(\text{cons}^n(\text{zero}, \text{nil})) \rightarrow 3n^2 + 2n + 1 \quad \text{cons}(\text{zero}, \ldots, \text{cons}(\text{zero}, \text{nil}))
\]

From \(|qs(\text{cons}^n(\text{zero}, \text{nil}))| = 2n + 2\) we get

\[
r_{c_R}(2n + 2) \geq 3n^2 + 2n + 1
\]
Finding Lower Bounds by Induction: Example

Example (quicksort)

\[
\begin{align*}
qs(\text{nil}) & \rightarrow \text{nil} \\
qs(\text{cons}(x, xs)) & \rightarrow qs(\text{low}(x, xs)) ++ \text{cons}(x, qs(\text{low}(x, xs))) \\
\text{low}(x, \text{nil}) & \rightarrow \text{nil} \\
\text{low}(x, \text{cons}(y, ys)) & \rightarrow \text{if}(x \leq y, x, \text{cons}(y, ys)) \\
\text{if}(\texttt{tt}, x, \text{cons}(y, ys)) & \rightarrow \text{low}(x, ys) \\
\text{if}(\texttt{ff}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{low}(x, ys)) \\
& \ldots
\end{align*}
\]

Speculate and prove rewrite lemma:

\[
\begin{align*}
qs(\text{cons}(\texttt{zero}, \ldots, \text{cons}(\texttt{zero}, \text{nil}))) & \rightarrow 3n^2 + 2n + 1 \quad \text{cons}(\texttt{zero}, \ldots, \text{cons}(\texttt{zero}, \text{nil})) \\
qs(\text{cons}^n(\texttt{zero}, \text{nil})) & \rightarrow 3n^2 + 2n + 1 \quad \text{cons}(\texttt{zero}, \ldots, \text{cons}(\texttt{zero}, \text{nil}))
\end{align*}
\]

From \(|qs(\text{cons}^n(\texttt{zero}, \text{nil}))| = 2n + 2\) we get 
\[rc_R(2n + 2) \geq 3n^2 + 2n + 1\] and \(rc_R(n) \in \Omega(n^2)\).
(2) Decreasing loops, inspired by \textbf{looping} non-termination with

\[
\begin{align*}
s & \rightarrow^+_R C[s\sigma] \rightarrow^+_R C[C\sigma[s\sigma^2]] \rightarrow^+_R \cdots
\end{align*}
\]

Example: \( f(y) \rightarrow f(s(y)) \) has loop \( f(y) \rightarrow^+_R f(s(y)) \) with \( \sigma(y) = 0 \).
(2) Decreasing loops, inspired by \textbf{looping} non-termination with

\[ s \rightarrow^+ \mathcal{R} C[s\sigma] \rightarrow^+ \mathcal{R} C[C\sigma[s\sigma^2]] \rightarrow^+ \mathcal{R} \cdots \]

\textbf{Example:} \( f(y) \rightarrow f(s(y)) \) has loop \( f(y) \rightarrow^+ \mathcal{R} f(s(y)) \) with \( \sigma(y) = 0 \).

Intuition for \textit{linear} lower bounds:

some fixed context \( D \) is \textbf{removed} in an argument of recursive call, other arguments may grow, sequence can be repeated (loop)
(2) Decreasing loops, inspired by **looping** non-termination with

\[ s \rightarrow_{\mathcal{R}}^+ C[s\sigma] \rightarrow_{\mathcal{R}}^+ C[C\sigma[s\sigma^2]] \rightarrow_{\mathcal{R}}^+ \cdots \]

**Example:** \( f(y) \rightarrow f(s(y)) \) has loop \( f(y) \rightarrow_{\mathcal{R}}^+ f(s(y)) \) with \( \sigma(y) = 0 \).

Intuition for **linear** lower bounds:

some fixed context \( D \) is **removed** in an argument of recursive call, other arguments may grow, sequence can be repeated (loop)

**Example:** \( \text{plus}(s(x), y) \rightarrow \text{plus}(x, s(y)) \) has decreasing loop

\[ \text{plus}(s(x), y) \rightarrow_{\mathcal{R}}^+ \text{plus}(x, s(y)) \] with \( D[x] = s(x) \)
Decreasing loops, inspired by **looping** non-termination with
\[ s \rightarrow^+ \mathcal{R} C[s\sigma] \rightarrow^+ \mathcal{R} C[C\sigma[s\sigma^2]] \rightarrow^+ \mathcal{R} \ldots \]

**Example:** \( f(y) \rightarrow f(s(y)) \) has loop \( f(y) \rightarrow^+ \mathcal{R} f(s(y)) \) with \( \sigma(y) = 0 \).

Intuition for **linear** lower bounds:

some fixed context \( D \) is **removed** in an argument of recursive call, other
arguments may grow, sequence can be repeated (loop)

**Example:** \( \text{plus}(s(x), y) \rightarrow \text{plus}(x, s(y)) \) has decreasing loop
\[ \text{plus}(s(x), y) \rightarrow^+ \mathcal{R} \text{plus}(x, s(y)) \] with \( D[x] = s(x) \)

for *base term* \( s = \text{plus}(x, y) \), *pumping substitution* \( \theta = [x \mapsto s(x)] \), and
*result substitution* \( \sigma = [y \mapsto s(y)] \):
\[ s\theta \rightarrow^+ \mathcal{R} C[s\sigma] \]

Implies \( rc(n) \in \Omega(n)! \)
Exponential lower bounds: several “compatible” parallel recursive calls:

- **Example:** $\text{fib}(s(s(n))) \rightarrow \text{plus}(\text{fib}(s(n)), \text{fib}(n))$ has 2 decreasing loops:
  
  \[
  \text{fib}(s(s(n))) \rightarrow^+ \mathcal{R} C[\text{fib}(s(n))] \quad \text{and} \quad \text{fib}(s(s(n))) \rightarrow^+ \mathcal{R} C[\text{fib}(n)]
  \]

  Implies $rc(n) \in \Omega(2^n)$!
Exponential lower bounds: several “compatible” parallel recursive calls:

- **Example:** \( \text{fib}(s(s(n))) \rightarrow \text{plus}(\text{fib}(s(n)), \text{fib}(n)) \) has 2 decreasing loops:

  \[
  \text{fib}(s(s(n))) \rightarrow^+ \mathcal{R} C[\text{fib}(s(n))] \quad \text{and} \quad \text{fib}(s(s(n))) \rightarrow^+ \mathcal{R} C[\text{fib}(n)]
  \]

  Implies \( \text{rc}(n) \in \Omega(2^n)! \)

- **(Non-)Example:** \( \text{tr}(\text{node}(x, y)) \rightarrow \text{node}(\text{tr}(x), \text{tr}(y)) \)

  Has linear complexity. But:

  \[
  \text{tr}(\text{node}(x, y)) \rightarrow^+ \mathcal{R} C[\text{tr}(x)] \quad \text{and} \quad \text{tr}(\text{node}(x, y)) \rightarrow^+ \mathcal{R} C[\text{tr}(y)]
  \]

  are not compatible (their pumping substitutions do not commute).
Exponential lower bounds: several “compatible” parallel recursive calls:

- **Example:** \( \text{fib}(s(s(n))) \rightarrow \text{plus}(\text{fib}(s(n)), \text{fib}(n)) \) has 2 decreasing loops:

  \[
  \text{fib}(s(s(n))) \rightarrow^+ \mathcal{R} \text{C}[\text{fib}(s(n))] \quad \text{and} \quad \text{fib}(s(s(n))) \rightarrow^+ \mathcal{R} \text{C}[\text{fib}(n)]
  \]

  Implies \( rc(n) \in \Omega(2^n) \)

- **(Non-)Example:** \( \text{tr}(\text{node}(x, y)) \rightarrow \text{node}(\text{tr}(x), \text{tr}(y)) \)

  Has linear complexity. But:

  \[
  \text{tr}(\text{node}(x, y)) \rightarrow^+ \mathcal{R} \text{C}[\text{tr}(x)] \quad \text{and} \quad \text{tr}(\text{node}(x, y)) \rightarrow^+ \mathcal{R} \text{C}[\text{tr}(y)]
  \]

  are not compatible (their pumping substitutions do not commute).

Automation for decreasing loops: **narrowing**.
Benefits of Induction Technique:

- Can find **non-linear** polynomial lower bounds
- Also works on non-left-linear TRSs
Benefits of Induction Technique:
- Can find **non-linear** polynomial lower bounds
- Also works on non-left-linear TRSs

Benefits of Decreasing Loops:
- Does not rely as much on heuristics
- Computationally more lightweight
Benefits of Induction Technique:
- Can find **non-linear** polynomial lower bounds
- Also works on non-left-linear TRSs

Benefits of Decreasing Loops:
- Does not rely as much on heuristics
- Computationally more lightweight

⇒ First try decreasing loops, then induction technique
Benefits of Induction Technique:
- Can find \textbf{non-linear} polynomial lower bounds
- Also works on non-left-linear TRSs

Benefits of Decreasing Loops:
- Does not rely as much on heuristics
- Computationally more lightweight

⇒ First try decreasing loops, then induction technique

Both techniques can be adapted to innermost runtime complexity!
A Landscape of Complexity Properties and Transformations

idc, irc: like dc, rc, but for *innermost* rewriting
A Landscape of Complexity Properties and Transformations

idc, irc: like dc, rc, but for *innermost* rewriting

TRS

dc

rc

idc

irc

\\[50\] F. Frohn, J. Giesl: *Analyzing runtime complexity via innermost runtime complexity*, LPAR ’17
A Landscape of Complexity Properties and Transformations

idc, irc: like dc, rc, but for *innermost* rewriting

---

50 F. Frohn, J. Giesl: *Analyzing runtime complexity via innermost runtime complexity*, LPAR ’17

51 C. Fuhs: *Transforming Derivational Complexity of Term Rewriting to Runtime Complexity*, FroCoS ’19
The big picture:

- **Have:** Tool for automated analysis of runtime complexity $\text{rc}_\mathcal{R}$
Transforming Derivational Complexity to Runtime Complexity

The big picture:

- **Have:** Tool for automated analysis of runtime complexity $\text{rc}_R$
- **Want:** Tool for automated analysis of derivational complexity $\text{dc}_R$
The big picture:

- **Have:** Tool for automated analysis of runtime complexity $r_{cR}$
- **Want:** Tool for automated analysis of derivational complexity $d_{cR}$
- **Idea:**
  
  “$r_{cR}$ analysis tool + transformation on TRS $R = d_{cR}$ analysis tool”
The big picture:

- **Have:** Tool for automated analysis of runtime complexity $rc_{\mathcal{R}}$
- **Want:** Tool for automated analysis of derivational complexity $dc_{\mathcal{R}}$
- **Idea:**
  
  "$rc_{\mathcal{R}}$ analysis tool + transformation on TRS $\mathcal{R} = dc_{\mathcal{R}}$ analysis tool"

- **Benefits:**
  
  - Get analysis of derivational complexity “for free”
  - Progress in runtime complexity analysis automatically improves derivational complexity analysis
program transformation such that runtime complexity of transformed TRS is **identical** to derivational complexity of original TRS
program transformation such that runtime complexity of transformed TRS is \textit{identical} to derivational complexity of original TRS

transformation correct also from idc to irc
From dc to rc: Results

- program transformation such that runtime complexity of transformed TRS is *identical* to derivational complexity of original TRS
- transformation correct also from idc to irc
- implemented in program analysis tool AProVE
From dc to rc: Results

- program transformation such that runtime complexity of transformed TRS is identical to derivational complexity of original TRS
- transformation correct also from idc to irc
- implemented in program analysis tool AProVE
- evaluated successfully on TPDB\textsuperscript{52} relative to state of the art TcT

\textsuperscript{52}Termination Problem DataBase, standard benchmark source for annual Termination Competition (termCOMP) with 1000s of problems, [http://termination-portal.org/wiki/TPDB](http://termination-portal.org/wiki/TPDB)
From dc to rc: Transformation

Issue:
- Runtime complexity assumes basic terms as start terms
- We want to analyse complexity for arbitrary terms
From dc to rc: Transformation

Issue:
- Runtime complexity assumes basic terms as start terms
- We want to analyse complexity for arbitrary terms

Idea:
- Introduce constructor symbol $c_f$ for defined symbol $f$
From dc to rc: Transformation

**Issue:**
- Runtime complexity assumes **basic** terms as start terms
- We want to analyse complexity for **arbitrary** terms

**Idea:**
- Introduce **constructor symbol** $c_f$ for **defined symbol** $f$
- Add **generator rewrite rules** $G$ to reconstruct arbitrary term with $f$
  from basic term with $c_f$
From $dc$ to $rc$: Transformation

**Issue:**
- Runtime complexity assumes **basic** terms as start terms
- We want to analyse complexity for **arbitrary** terms

**Idea:**
- Introduce **constructor symbol** $c_f$ for **defined symbol** $f$
- Add **generator rewrite rules** $G$ to reconstruct arbitrary term with $f$ from basic term with $c_f$

Represent

\[
t = \text{double}(\text{double}(\text{double}(s(0))))
\]
From dc to rc: Transformation

Issue:
- Runtime complexity assumes basic terms as start terms
- We want to analyse complexity for arbitrary terms

Idea:
- Introduce constructor symbol $c_f$ for defined symbol $f$
- Add generator rewrite rules $G$ to reconstruct arbitrary term with $f$ from basic term with $c_f$

Represent
\[ t = \text{double} (\text{double} (\text{double} (\text{s} (0)))) \]
by basic variant

\[ \text{bv}(t) = \text{enc}_{\text{double}} ( \text{c}_{\text{double}} (\text{c}_{\text{double}} (\text{s} (0)))) \]
From dc to rc: Transformation

Issue:
- Runtime complexity assumes basic terms as start terms
- We want to analyse complexity for arbitrary terms

Idea:
- Introduce constructor symbol $c_f$ for defined symbol $f$
- Add generator rewrite rules $G$ to reconstruct arbitrary term with $f$
  from basic term with $c_f$

Represent
$$t = \text{double}(\text{double}(\text{double}(s(0))))$$
by basic variant
$$\text{bv}(t) = \text{enc}_{\text{double}}(c_{\text{double}}(c_{\text{double}}(s(0))))$$

**Example (Generator rules $G$)**

- $\text{enc}_{\text{double}}(x) \rightarrow \text{double}(\text{argenc}(x))$
- $\text{enc}_0 \rightarrow 0$
- $\text{enc}_s(x) \rightarrow s(\text{argenc}(x))$
- $\text{argenc}(c_{\text{double}}(x)) \rightarrow \text{double}(\text{argenc}(x))$
- $\text{argenc}(0) \rightarrow 0$
- $\text{argenc}(s(x)) \rightarrow s(\text{argenc}(x))$
From \( dc \) to \( rc \): Transformation

**Issue:**
- Runtime complexity assumes **basic** terms as start terms
- We want to analyse complexity for **arbitrary** terms

**Idea:**
- Introduce constructor symbol \( c_f \) for defined symbol \( f \)
- Add generator rewrite rules \( G \) to reconstruct arbitrary term with \( f \) from basic term with \( c_f \)

Represent

\[
t = \text{double}(\text{double}(\text{double}(\text{s}(0)))))
\]

by **basic** variant

\[
bv(t) = \text{enc}_{\text{double}}(c_{\text{double}}(c_{\text{double}}(\text{s}(0)))))
\]

Then:
- \( bv(t) \) is **basic** term, size \( |t| \)

---

**Example (Generator rules \( G \))**

\[
\begin{align*}
\text{enc}_{\text{double}}(x) & \rightarrow \text{double}(\text{argenc}(x)) \\
\text{enc}_0 & \rightarrow 0 \\
\text{enc}_s(x) & \rightarrow \text{s}(\text{argenc}(x)) \\
\text{argenc}(c_{\text{double}}(x)) & \rightarrow \text{double}(\text{argenc}(x)) \\
\text{argenc}(0) & \rightarrow 0 \\
\text{argenc}(s(x)) & \rightarrow \text{s}(\text{argenc}(x))
\end{align*}
\]
From dc to rc: Transformation

**Issue:**
- Runtime complexity assumes **basic** terms as start terms
- We want to analyse complexity for **arbitrary** terms

**Idea:**
- Introduce **constructor symbol** $c_f$ for **defined symbol** $f$
- Add **generator rewrite rules** $G$ to reconstruct arbitrary term with $f$ from basic term with $c_f$

Represent

$t = \text{double(}\text{double(}\text{double(s(0))))}$

by **basic variant**

$bv(t) = \text{enc}_\text{double}(c_\text{double}(c_\text{double}(s(0))))$

Then:
- $bv(t)$ is **basic** term, size $|t|$
- $bv(t) \rightarrow^*_G t$

**Example (Generator rules $G$)**

- $\text{enc}_\text{double}(x) \rightarrow \text{double(}\text{argenc}(x))$
- $\text{enc}_0 \rightarrow 0$
- $\text{enc}_s(x) \rightarrow s(\text{argenc}(x))$
- $\text{argenc}(c_\text{double}(x)) \rightarrow \text{double(}\text{argenc}(x))$
- $\text{argenc}(0) \rightarrow 0$
- $\text{argenc}(s(x)) \rightarrow s(\text{argenc}(x))$
Issue:

- $\rightarrow_{R \cup G}$ has extra rewrite steps not present in $\rightarrow_R$
- may change complexity
General Case: Relative Rewriting

**Issue:**
- $\rightarrow_{\mathcal{R} \cup \mathcal{G}}$ has extra rewrite steps not present in $\rightarrow_{\mathcal{R}}$
- may change complexity

**Solution:**
- add $\mathcal{G}$ as relative rewrite rules:
  - $\rightarrow_{\mathcal{G}}$ steps are **not counted** for complexity analysis!
- transform $\mathcal{R}$ to $\mathcal{R}/\mathcal{G}$ ($\rightarrow_{\mathcal{R}}$ steps are counted, $\rightarrow_{\mathcal{G}}$ steps are not)
General Case: Relative Rewriting

Issue:
- $\rightarrow R \cup G$ has extra rewrite steps not present in $\rightarrow R$
- may change complexity

Solution:
- add $G$ as relative rewrite rules:
  - $\rightarrow G$ steps are not counted for complexity analysis!
- transform $R$ to $R/G$ ($\rightarrow R$ steps are counted, $\rightarrow G$ steps are not)
- more generally: transform $R/S$ to $R/(S \cup G)$
  (input may contain relative rules $S$, too)
From \( dc \) to \( rc \): Correctness

**Theorem (Derivational Complexity via Runtime Complexity)**

Let \( R/S \) be a relative TRS, let \( G \) be the generator rules for \( R/S \). Then

1. \( dc_{R/S}(n) = rc_{R/(S∪G)}(n) \) (arbitrary rewrite strategies)
2. \( idc_{R/S}(n) = irc_{R/(S∪G)}(n) \) (innermost rewriting)

Note: equalities hold also non-asymptotically!
Experiments on TPDB, compare with state of the art in TcT:

- upper bounds $\text{idc}$: both AProVE and TcT with transformation are stronger than standard TcT

- upper bounds $\text{dc}$: TcT stronger than AProVE and TcT with transformation, but AProVE still solves some new examples

- lower bounds $\text{idc}$ and $\text{dc}$: heuristics do not seem to benefit much
Experiments on TPDB, compare with state of the art in TcT:

- upper bounds $\text{idc}$: both AProVE and TcT with transformation are stronger than standard TcT

- upper bounds $\text{dc}$: TcT stronger than AProVE and TcT with transformation, but AProVE still solves some new examples

- lower bounds $\text{idc}$ and $\text{dc}$: heuristics do not seem to benefit much

$\Rightarrow$ Transformation-based approach should be part of the portfolio of analysis tools for derivational complexity
Possible applications

- compiler simplifications
- SMT solver preprocessing

Start terms may have nested defined symbols, so $d_{cR}$ is appropriate
Possible applications
- compiler simplifications
- SMT solver preprocessing

Start terms may have nested defined symbols, so $dc_R$ is appropriate

Go between derivational and runtime complexity
- So far: encode full term universe $T$ via basic terms $T_{\text{basic}}$
- Generalise: write relative rules to generate arbitrary set $U$ of terms “between” basic and all terms ($T_{\text{basic}} \subseteq U \subseteq T$).
Possible applications

- compiler simplifications
- SMT solver preprocessing

Start terms may have nested defined symbols, so $d_{CR}$ is appropriate.

Go between derivational and runtime complexity

- So far: encode full term universe $T$ via basic terms $T_{basic}$
- Generalise: write relative rules to generate arbitrary set $U$ of terms “between” basic and all terms ($T_{basic} \subseteq U \subseteq T$).

Want to adapt techniques from runtime complexity analysis to derivational complexity! How?

- (Useful) adaptation of Dependency Pairs?
- Abstractions to numbers?
- …
A Landscape of Complexity Properties and Transformations

idc, irc: like dc, rc, but for *innermost* rewriting

FroCoS'19

dc → rc

FroCoS'19

idc → irc

TRS

LPAR'17

Rec. ITS irc

ITS irc
A Landscape of Complexity Properties and Transformations

idc, irc: like dc, rc, but for innermost rewriting

M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, J. Giesl: Complexity analysis for term rewriting by integer transition systems, FroCoS '17
A Landscape of Complexity Properties and Transformations

idc, irc: like dc, rc, but for innermost rewriting

TRS

dc \rightarrow rc \rightarrow \text{FroCoS'19} \rightarrow \text{LPAR'17} \rightarrow \text{Rec. ITS irc} \rightarrow \text{ITS irc} \rightarrow \text{FroCoS'17}

idc \rightarrow irc \rightarrow \text{FroCoS'19} \rightarrow \text{LPAR'17} \rightarrow \text{Rec. ITS irc} \rightarrow \text{ITS irc} \rightarrow \text{FroCoS'17}

\[53\] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, J. Giesl: Complexity analysis for term rewriting by integer transition systems, FroCoS '17
Recently significant progress in complexity analysis tools for Integer Transition Systems (ITSs):

- CoFloCo\textsuperscript{54}
- KoAT\textsuperscript{55}
- PUBS\textsuperscript{56}

Goal: use these tools to find upper bounds for TRS complexity

\textsuperscript{54} A. Flores-Montoya, R. Hähnle: \textit{Resource analysis of complex programs with cost equations}, APLAS '14, https://github.com/aeflores/CoFloCo


Analysing irc of Insertion Sort by Hand: Bottom-Up

Example

\[
\begin{align*}
\text{isort}(\text{nil}, ys) & \rightarrow ys \\
\text{isort}(\text{cons}(x, xs), ys) & \rightarrow \text{isort}(xs, \text{insert}(x, ys)) \\
\text{insert}(x, \text{nil}) & \rightarrow \text{cons}(x, \text{nil}) \\
\text{insert}(x, \text{cons}(y, ys)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys)) \\
\text{if}(\text{true}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{insert}(x, ys)) \\
\text{if}(\text{false}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(x, \text{cons}(y, ys)) \\
\text{gt}(0, y) & \equiv \text{false} \\
\text{gt}(s(x), 0) & \equiv \text{true} \\
\text{gt}(s(x), s(y)) & \rightarrow \text{gt}(x, y)
\end{align*}
\]
Analysing irc of Insertion Sort by Hand: Bottom-Up

Example

\[
\begin{align*}
\text{isort}(\text{nil}, ys) & \rightarrow ys \\
\text{isort}(\text{cons}(x, xs), ys) & \rightarrow \text{isort}(xs, \text{insert}(x, ys)) \\
\text{insert}(x, \text{nil}) & \rightarrow \text{cons}(x, \text{nil}) \\
\text{insert}(x, \text{cons}(y, ys)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys)) \\
\text{if}(\text{true}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{insert}(x, ys)) \\
\text{if}(\text{false}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(x, \text{cons}(y, ys)) \\
\text{gt}(0, y) & \equiv \text{false} \\
\text{gt}(s(x), 0) & \equiv \text{true} \\
\text{gt}(s(x), s(y)) & \rightarrow \text{gt}(x, y)
\end{align*}
\]

Note: innermost reduction strategy
Analysing irc of Insertion Sort by Hand: Bottom-Up

Example

\[
\begin{align*}
isort(nil, ys) & \rightarrow ys \\
isort(\text{cons}(x, xs), ys) & \rightarrow isort(xs, \text{insert}(x, ys)) \\
isort(xs, ys) & \rightarrow isort(xs, insert(x, ys)) \\
insert(x, nil) & \rightarrow \text{cons}(x, nil) \\
insert(x, \text{cons}(y, ys)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys)) \\
\text{if}(\text{true}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{insert}(x, y)) \\
\text{if}(\text{false}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(x, \text{cons}(y, y)) \\
\text{gt}(0, y) & \equiv \text{false} \\
\text{gt}(s(x), 0) & \equiv \text{true} \\
\text{gt}(s(x), s(y)) & \equiv \text{gt}(x, y)
\end{align*}
\]

- \text{rt}(\text{gt}(x, y)) \in O(1) \quad (\text{"\rightarrow" for relative rules})

Note: innermost reduction strategy
Analysing irc of Insertion Sort by Hand: Bottom-Up

Example

\[
\begin{align*}
\text{isort}(\text{nil}, ys) & \rightarrow ys \\
\text{isort}(\text{cons}(x, xs), ys) & \rightarrow \text{isort}(xs, \text{insert}(x, ys)) \\
\text{insert}(x, \text{nil}) & \rightarrow \text{cons}(x, \text{nil}) \\
\text{insert}(x, \text{cons}(y, ys)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys)) \\
\text{if}(\text{true}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{insert}(x, ys)) \\
\text{if}(\text{false}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(x, \text{cons}(y, ys)) \\
\text{gt}(0, y) & \equiv \text{false} \\
\text{gt}(s(x), 0) & \equiv \text{true} \\
\text{gt}(s(x), s(y)) & \equiv \text{gt}(x, y)
\end{align*}
\]

- \( \text{rt}(\text{gt}(x, y)) \in O(1) \) ("\( \overrightarrow{\text{--}} \)" for relative rules)
- \( \text{rt}(\text{insert}(x, ys)) \in O(\text{length}(ys)) \)

Note: innermost reduction strategy
Analysing irc of Insertion Sort by Hand: Bottom-Up

Example

\[
\begin{align*}
isort(\text{nil}, \text{ys}) & \rightarrow \text{ys} \\
isort(\text{cons}(x, xs), \text{ys}) & \rightarrow \text{isort}(xs, \text{insert}(x, \text{ys})) \\
\text{insert}(x, \text{nil}) & \rightarrow \text{cons}(x, \text{nil}) \\
\text{insert}(x, \text{cons}(y, \text{ys})) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, \text{ys})) \\
\text{if}(\text{true}, x, \text{cons}(y, \text{ys})) & \rightarrow \text{cons}(y, \text{insert}(x, \text{ys})) \\
\text{if}(\text{false}, x, \text{cons}(y, \text{ys})) & \rightarrow \text{cons}(x, \text{cons}(y, \text{ys})) \\
\text{gt}(0, y) & \rightarrow \text{false} \\
\text{gt}(s(x), 0) & \rightarrow \text{true} \\
\text{gt}(s(x), s(y)) & \rightarrow \text{gt}(x, y)
\end{align*}
\]

- \( \text{rt}(\text{gt}(x, y)) \in \mathcal{O}(1) \) ("\( \overrightleftharpoons \)" for relative rules)
- \( \text{rt}(\text{insert}(x, y)) \in \mathcal{O}(\text{length}(ys)) \)
- \( \text{rt}(\text{isort}(xs, ys)) \in \mathcal{O}(\text{length}(xs) \cdot \ldots) \)

Note: innermost reduction strategy
Analysing irc of Insertion Sort by Hand: Bottom-Up

Example

\[
\begin{align*}
\text{isort}(\text{nil}, ys) & \rightarrow ys \\
\text{isort}(\text{cons}(x, xs), ys) & \rightarrow \text{isort}(xs, \text{insert}(x, ys)) \\
\text{insert}(x, \text{nil}) & \rightarrow \text{cons}(x, \text{nil}) \\
\text{insert}(x, \text{cons}(y, ys)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys)) \\
\text{if}(\text{true}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{insert}(x, ys)) \\
\text{if}(\text{false}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(x, \text{cons}(y, ys)) \\
\text{gt}(0, y) & \quad \equiv \quad \text{false} \\
\text{gt}(\text{s}(x), 0) & \quad \equiv \quad \text{true} \\
\text{gt}(\text{s}(x), \text{s}(y)) & \quad \equiv \quad \text{gt}(x, y)
\end{align*}
\]

- \(\text{rt}(\text{gt}(x, y)) \in \mathcal{O}(1)\) ("\(\rightarrow\)" for relative rules)
- \(\text{rt}(\text{insert}(x, ys)) \in \mathcal{O}(\text{length}(ys))\)
- \(\text{rt}(\text{isort}(xs, ys)) \in \mathcal{O}(\text{length}(xs) \cdot (\text{length}(xs) + \text{length}(ys)))\)

Note: innermost reduction strategy
Example

\[
\begin{align*}
\text{isort}(\text{nil}, y) & \rightarrow y \\
\text{isort}(\text{cons}(x, xs), y) & \rightarrow \text{isort}(xs, \text{insert}(x, y)) \\
\text{insert}(x, \text{nil}) & \rightarrow \text{cons}(x, \text{nil}) \\
\text{insert}(x, \text{cons}(y, y)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, y)) \\
\text{if}(\text{true}, x, \text{cons}(y, y)) & \rightarrow \text{cons}(y, \text{insert}(x, y)) \\
\text{if}(\text{false}, x, \text{cons}(y, y)) & \rightarrow \text{cons}(x, \text{cons}(y, y)) \\
\text{gt}(0, y) & \equiv \text{false} \\
\text{gt}(s(x), 0) & \equiv \text{true} \\
\text{gt}(s(x), s(y)) & \equiv \text{gt}(x, y)
\end{align*}
\]

- the recursive \text{isort} rule is at most applied linearly often
Using Dependency Tuples: Top-Down

Example

\[
\begin{align*}
\text{isort}(\text{nil}, ys) & \rightarrow ys \\
\text{isort}(\text{cons}(x, xs), ys) & \rightarrow \text{isort}(xs, \text{insert}(x, ys)) \\
\text{insert}(x, \text{nil}) & \rightarrow \text{cons}(x, \text{nil}) \\
\text{insert}(x, \text{cons}(y, ys)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys)) \\
\text{if}(\text{true}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{insert}(x, ys)) \\
\text{if}(\text{false}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(x, \text{cons}(y, ys)) \\
\text{gt}(0, y) & \equiv \text{false} \\
\text{gt}(s(x), 0) & \equiv \text{true} \\
\text{gt}(s(x), s(y)) & \equiv \text{gt}(x, y)
\end{align*}
\]

- the recursive \text{isort} rule is at most applied linearly often
- the recursive \text{insert} rule is at most applied quadratically often
Using Dependency Tuples: Top-Down

Example

- $\text{isort}(\text{nil}, ys) \rightarrow ys$
- $\text{isort}(\text{cons}(x, xs), ys) \rightarrow \text{isort}(xs, \text{insert}(x, ys))$
- $\text{insert}(x, \text{nil}) \rightarrow \text{cons}(x, \text{nil})$
- $\text{insert}(x, \text{cons}(y, ys)) \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys))$
- $\text{if}(\text{true}, x, \text{cons}(y, ys)) \rightarrow \text{cons}(y, \text{insert}(x, ys))$
- $\text{if}(\text{false}, x, \text{cons}(y, ys)) \rightarrow \text{cons}(x, \text{cons}(y, ys))$
- $\text{gt}(0, y) \rightarrow \text{false}$
- $\text{gt}(\text{s}(x), 0) \rightarrow \text{true}$
- $\text{gt}(\text{s}(x), \text{s}(y)) \rightarrow \text{gt}(x, y)$

- the recursive $\text{isort}$ rule is at most applied linearly often
- the recursive $\text{insert}$ rule is at most applied quadratically often
  - note: requires reasoning about $\text{isort}$, $\text{insert}$, and $\text{if}$ rules!
Using Dependency Tuples: Top-Down

Example

\[
\begin{align*}
isort(\text{nil}, ys) & \rightarrow ys \\
isort(\text{cons}(x, xs), ys) & \rightarrow isort(xs, \text{insert}(x, ys)) \\
isort(x, \text{nil}) & \rightarrow \text{cons}(x, \text{nil}) \\
isort(x, \text{cons}(y, ys)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys)) \\
isort(\text{true}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{insert}(x, ys)) \\
isort(\text{false}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(x, \text{cons}(y, ys)) \\
\text{gt}(0, y) & \equiv \text{false} \\
\text{gt}(s(x), 0) & \equiv \text{true} \\
\text{gt}(s(x), s(y)) & \equiv \text{gt}(x, y)
\end{align*}
\]

- the recursive \textit{isort} rule is at most applied linearly often
- the recursive \textit{insert} rule is at most applied quadratically often
  - note: requires reasoning about \textit{isort}, \textit{insert}, and \textit{if} rules!
  - found via quadratic polynomial interpretation
Using Dependency Tuples: Top-Down

Example

\[
\begin{align*}
isort(nil, ys) & \rightarrow ys \\
isort(cons(x, xs), ys) & \rightarrow isort(xs, insert(x, ys)) \\
insert(x, nil) & \rightarrow cons(x, nil) \\
insert(x, cons(y, ys)) & \rightarrow if(gt(x, y), x, cons(y, ys)) \\
if(true, x, cons(y, ys)) & \rightarrow cons(y, insert(x, ys)) \\
if(false, x, cons(y, ys)) & \rightarrow cons(x, cons(y, ys)) \\
\end{align*}
\]

- the recursive isort rule is at most applied linearly often
- the recursive insert rule is at most applied quadratically often
  - note: requires reasoning about isort, insert, and if rules!
  - found via quadratic polynomial interpretation
- the recursive if rule is applied as often as the recursive insert rule
Example

\[
\text{isort}(\text{nil}, ys) \quad \rightarrow \quad ys
\]
\[
\text{isort}(\text{cons}(x, xs), ys) \quad \rightarrow \quad \text{isort}(xs, \text{insert}(x, ys))
\]
\[
\text{insert}(x, \text{nil}) \quad \rightarrow \quad \text{cons}(x, \text{nil})
\]
\[
\text{insert}(x, \text{cons}(y, ys)) \quad \rightarrow \quad \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys))
\]
\[
\text{if}(\text{true}, x, \text{cons}(y, ys)) \quad \rightarrow \quad \text{cons}(y, \text{insert}(x, ys))
\]
\[
\text{if}(\text{false}, x, \text{cons}(y, ys)) \quad \rightarrow \quad \text{cons}(x, \text{cons}(y, ys))
\]
\[
\text{gt}(0, y) \quad \equiv \quad \text{false}
\]
\[
\text{gt}(s(x), 0) \quad \equiv \quad \text{true}
\]
\[
\text{gt}(s(x), s(y)) \quad \equiv \quad \text{gt}(x, y)
\]

1. abstract terms to integers
Example

\[
isort(xs', ys) \Rightarrow ys \quad | \quad xs' = 1
\]

\[
isort(\text{cons}(x, xs), ys) \Rightarrow isort(xs, \text{insert}(x, ys))
\]

\[
\text{insert}(x, \text{nil}) \Rightarrow \text{cons}(x, \text{nil})
\]

\[
\text{insert}(x, \text{cons}(y, ys)) \Rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys))
\]

\[
\text{if}(\text{true}, x, \text{cons}(y, ys)) \Rightarrow \text{cons}(y, \text{insert}(x, ys))
\]

\[
\text{if}(\text{false}, x, \text{cons}(y, ys)) \Rightarrow \text{cons}(x, \text{cons}(y, ys))
\]

\[
\text{gt}(0, y) \Rightarrow \text{false}
\]

\[
\text{gt}(s(x), 0) \Rightarrow \text{true}
\]

\[
\text{gt}(s(x), s(y)) \Rightarrow \text{gt}(x, y)
\]

\[\text{abstract terms to integers}\]
Example

\[
\begin{align*}
\text{isort}(xs', ys) & \xrightarrow{1} ys & | & xs' = 1 \\
\text{isort}(xs', ys) & \xrightarrow{1} \text{isort}(xs, \text{insert}(x, ys)) & | & xs' = 1 + x + xs \\
\text{insert}(x, \text{nil}) & \rightarrow \text{cons}(x, \text{nil}) \\
\text{insert}(x, \text{cons}(y, ys)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys)) \\
\text{if}(\text{true}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{insert}(x, ys)) \\
\text{if}(\text{false}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(x, \text{cons}(y, ys)) \\
\text{gt}(0, y) & \equiv \text{false} \\
\text{gt}(s(x), 0) & \equiv \text{true} \\
\text{gt}(s(x), s(y)) & \equiv \text{gt}(x, y)
\end{align*}
\]

abstract terms to integers
Bird’s Eye View of the Transformation

Example

\[
\begin{align*}
\text{isort}(xs', ys) & \xrightarrow{1} ys & \text{if } xs' = 1 \\
\text{isort}(xs', ys) & \xrightarrow{1} \text{isort}(xs, \text{insert}(x, ys)) & \text{if } xs' = 1 + x + xs \\
\text{insert}(x, ys') & \xrightarrow{1} 2 + x & \text{if } ys' = 1 \\
\text{insert}(x, \text{cons}(y, ys)) & \rightarrow \text{if}(\text{gt}(x, y), x, \text{cons}(y, ys)) \\
\text{if}(\text{true}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(y, \text{insert}(x, ys)) \\
\text{if}(\text{false}, x, \text{cons}(y, ys)) & \rightarrow \text{cons}(x, \text{cons}(y, ys)) \\
\text{gt}(0, y) & \equiv \text{false} \\
\text{gt}(s(x), 0) & \equiv \text{true} \\
\text{gt}(s(x), s(y)) & \equiv \text{gt}(x, y)
\end{align*}
\]

abstract terms to integers
Example

\[
\begin{align*}
\text{isort}(xs', ys) & \xrightarrow{1} ys & | & xs' = 1 \\
\text{isort}(xs', ys) & \xrightarrow{1} \text{isort}(xs, \text{insert}(x, ys)) & | & xs' = 1 + x + xs \\
\text{insert}(x, ys') & \xrightarrow{1} 2 + x & | & ys' = 1 \\
\text{insert}(x, ys') & \xrightarrow{1} \text{if}(\text{gt}(x, y), x, ys') & | & ys' = 1 + y + ys \\
\text{if}(b, x, ys') & \xrightarrow{1} 1 + y + \text{insert}(x, ys) & | & b = 1 \land ys' = 1 + y + ys \\
\text{if}(b, x, ys') & \xrightarrow{1} 1 + ys' & | & b = 1 \land ys' = 1 + y + ys \\
\text{gt}(x', y') & \xrightarrow{0} 1 & | & x' = 1 \\
\text{gt}(x', y') & \xrightarrow{0} 1 & | & x' = 1 + x \land y' = 1 \\
\text{gt}(x', y') & \xrightarrow{0} \text{gt}(x, y) & | & x' = 1 + x \land y' = 1 + y
\end{align*}
\]

abstract terms to integers
### Bird's Eye View of the Transformation

**Example**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transformation</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{isort}(xs', ys))</td>
<td>(\rightarrow ys)</td>
<td>(xs' = 1)</td>
</tr>
<tr>
<td>(\text{isort}(xs', ys))</td>
<td>(\rightarrow \text{isort}(xs, \text{insert}(x, ys)))</td>
<td>(xs' = 1 + x + xs)</td>
</tr>
<tr>
<td>(\text{insert}(x, ys'))</td>
<td>(\rightarrow 2 + x)</td>
<td>(ys' = 1)</td>
</tr>
<tr>
<td>(\text{insert}(x, ys'))</td>
<td>(\rightarrow \text{if}(\text{gt}(x, y), x, ys'))</td>
<td>(ys' = 1 + y + ys)</td>
</tr>
<tr>
<td>(\text{if}(b, x, ys'))</td>
<td>(\rightarrow 1 + y + \text{insert}(x, ys))</td>
<td>(b = 1 \wedge ys' = 1 + y + ys)</td>
</tr>
<tr>
<td>(\text{if}(b, x, ys'))</td>
<td>(\rightarrow 1 + ys')</td>
<td>(b = 1 \wedge ys' = 1 + y + ys)</td>
</tr>
<tr>
<td>(\text{gt}(x', y'))</td>
<td>(\rightarrow 1)</td>
<td>(x' = 1)</td>
</tr>
<tr>
<td>(\text{gt}(x', y'))</td>
<td>(\rightarrow 1)</td>
<td>(x' = 1 + x \wedge y' = 1)</td>
</tr>
<tr>
<td>(\text{gt}(x', y'))</td>
<td>(\rightarrow \text{gt}(x, y))</td>
<td>(x' = 1 + x \wedge y' = 1 + y)</td>
</tr>
</tbody>
</table>

1. abstract terms to integers
   - \([c](x_1, \ldots, x_n) = 1 + x_1 + \cdots + x_n\) for constructors \(c\)
   - note: variables range over \(\mathbb{N}\)
   - just + and ·
Bird’s Eye View of the Transformation

Example

\[
\begin{align*}
\text{isort}(xs', ys) & \xrightarrow{1} \text{ys} \quad | \quad xs' = 1 \\
\text{isort}(xs', ys) & \xrightarrow{1} \text{isort}(xs, \text{insert}(x, ys)) \quad | \quad xs' = 1 + x + xs \\
\text{insert}(x, ys') & \xrightarrow{1} 2 + x \quad | \quad ys' = 1 \\
\text{insert}(x, ys') & \xrightarrow{1} \text{if}(\text{gt}(x, y), x, ys') \quad | \quad ys' = 1 + y + ys \\
\text{if}(b, x, ys') & \xrightarrow{1} 1 + y + \text{insert}(x, ys) \quad | \quad b = 1 \land ys' = 1 + y + ys \\
\text{if}(b, x, ys') & \xrightarrow{1} 1 + ys' \quad | \quad b = 1 \land ys' = 1 + y + ys \\
\text{gt}(x', y') & \xrightarrow{0} 1 \quad | \quad x' = 1 \\
\text{gt}(x', y') & \xrightarrow{0} 1 \quad | \quad x' = 1 + x \land y' = 1 \\
\text{gt}(x', y') & \xrightarrow{0} \text{gt}(x, y) \quad | \quad x' = 1 + x \land y' = 1 + y
\end{align*}
\]

1. abstract terms to integers
   - \([c](x_1, \ldots, x_n) = 1 + x_1 + \cdots + x_n\) for constructors \(c\)
   - note: variables range over \(\mathbb{N}\)
   - just + and ·

2. analyse result size for bottom-SCC (Strongly Connected Component) of call graph using standard ITS tools
Call Graph & Bottom SCCs

- isort
- insert
- if
- gt
### Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Left</th>
<th>Right</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isort(xs', ys)</code></td>
<td>$\rightarrow$</td>
<td><code>ys</code></td>
<td>$xs' = 1$</td>
</tr>
<tr>
<td><code>isort(xs', ys)</code></td>
<td>$\rightarrow$</td>
<td><code>isort(xs, insert(x, ys))</code></td>
<td>$xs' = 1 + x + xs$</td>
</tr>
<tr>
<td><code>insert(x, ys')</code></td>
<td>$\rightarrow$</td>
<td><code>2 + x</code></td>
<td>$ys' = 1$</td>
</tr>
<tr>
<td><code>insert(x, ys')</code></td>
<td>$\rightarrow$</td>
<td><code>if(gt(x, y), x, ys')</code></td>
<td>$ys' = 1 + y + ys$</td>
</tr>
<tr>
<td><code>if(b, x, ys')</code></td>
<td>$\rightarrow$</td>
<td><code>1 + y + insert(x, ys)</code></td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
</tr>
<tr>
<td><code>if(b, x, ys')</code></td>
<td>$\rightarrow$</td>
<td><code>1 + ys'</code></td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
</tr>
<tr>
<td><code>gt(x', y')</code></td>
<td>$\rightarrow$</td>
<td><code>1</code></td>
<td>$x' = 1$</td>
</tr>
<tr>
<td><code>gt(x', y')</code></td>
<td>$\rightarrow$</td>
<td><code>1</code></td>
<td>$x' = 1 + x \land y' = 1$</td>
</tr>
<tr>
<td><code>gt(x', y')</code></td>
<td>$\rightarrow$</td>
<td><code>gt(x, y)</code></td>
<td>$x' = 1 + x \land y' = 1 + y$</td>
</tr>
</tbody>
</table>

1. **abstract terms to integers**
   - $[c](x_1, \ldots, x_n) = 1 + x_1 + \cdots + x_n$ for constructors $c$
   - note: variables range over $\mathbb{N}$
   - just $+$ and $\cdot$

2. **analyse result size for bottom-SCC using standard ITS tools**
isort\((xs', y)\) $\xrightarrow{1} y$ | $xs' = 1$

isort\((xs', y)\) $\xrightarrow{1} \text{isort}(xs, \text{insert}(x, y))$ | $xs' = 1 + x + xs$

insert\((x, y)\) $\xrightarrow{1} 2 + x$ | $ys' = 1$

insert\((x, y)\) $\xrightarrow{1} \text{if}(\text{gt}(x, y), x, y)$ | $ys' = 1 + y + ys$

if\((b, x, y)\) $\xrightarrow{1} 1 + y + \text{insert}(x, y)$ | $b = 1 \land ys' = 1 + y + ys$

if\((b, x, y)\) $\xrightarrow{1} 1 + y$ | $b = 1 \land ys' = 1 + y + ys$

gt\((x', y')\) $\xrightarrow{0} 1$ | $x' = 1$

gt\((x', y')\) $\xrightarrow{0} 1$ | $x' = 1 + x \land y' = 1$

abstract terms to integers
- $[c](x_1, \ldots, x_n) = 1 + x_1 + \cdots + x_n$ for constructors $c$
- note: variables range over $\mathbb{N}$
- just + and ·

analyse result size for bottom-SCC using standard ITS tools
### Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{isort}(xs', ys) \xrightarrow{1} ys$</td>
<td>$xs' = 1$</td>
<td></td>
</tr>
<tr>
<td>$\text{isort}(xs', ys) \xrightarrow{1} \text{isort}(xs, \text{insert}(x, ys))$</td>
<td>$xs' = 1 + x + xs$</td>
<td></td>
</tr>
<tr>
<td>$\text{insert}(x, ys') \xrightarrow{1} 2 + x$</td>
<td>$ys' = 1$</td>
<td></td>
</tr>
<tr>
<td>$\text{insert}(x, ys') \xrightarrow{1} \text{if}(\text{gt}(x, y), x, ys')$</td>
<td>$ys' = 1 + y + ys$</td>
<td></td>
</tr>
<tr>
<td>$\text{if}(b, x, ys') \xrightarrow{1} 1 + y + \text{insert}(x, ys)$</td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
<td></td>
</tr>
<tr>
<td>$\text{if}(b, x, ys') \xrightarrow{1} 1 + ys'$</td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
<td></td>
</tr>
<tr>
<td>$\text{gt}(x', y') \xrightarrow{0} 1$</td>
<td>$x' = 1$</td>
<td></td>
</tr>
<tr>
<td>$\text{gt}(x', y') \xrightarrow{0} 1$</td>
<td>$x' = 1 + x \land y' = 1$</td>
<td></td>
</tr>
<tr>
<td>$\text{gt}(x', y') \xrightarrow{0} \text{gt}(x, y)$</td>
<td>$x' = 1 + x \land y' = 1 + y$</td>
<td></td>
</tr>
</tbody>
</table>

1. abstract terms to integers
   - $[c](x_1, \ldots, x_n) = 1 + x_1 + \cdots + x_n$ for constructors $c$
   - note: variables range over $\mathbb{N}$
   - just $+$ and $\cdot$

2. analyse result size for bottom-SCC using standard ITS tools

3. analyse runtime of bottom-SCC using standard ITS tools
abstract terms to integers

- $[c](x_1, \ldots, x_n) = 1 + x_1 + \cdots + x_n$ for constructors $c$
- note: variables range over $\mathbb{N}$
- just $+$ and $\cdot$

analyse result size for bottom-SCC using standard ITS tools
analyse runtime of bottom-SCC using standard ITS tools
Abstracting Terms to Integers: Pitfalls
## Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

### Example

- $h(x) \rightarrow f(g(x))$
- $f(x) \rightarrow f(x)$
- $g(a) \overset{\equiv}{\rightarrow} g(a)$

### Definition

$N$ is a terminating variant of $S$ iff $N$ terminates and every $N$-normal form is an $S$-normal form.
Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

Example

\[ h(x) \rightarrow f(g(x)) \quad f(x) \rightarrow f(x) \quad g(a) \overline{\rightarrow} g(a) \]

innermost rewriting: \[ h(x) \rightarrow f(g(x)) \rightarrow f(g(x)) \rightarrow \ldots \]
### Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

#### Example

\[
\begin{align*}
    h(x) & \rightarrow f(g(x)) & f(x) & \rightarrow f(x) & g(a) & \equiv g(a) \\
    \text{innermost rewriting:} & h(x) & \rightarrow f(g(x)) & \rightarrow f(g(x)) & \rightarrow \ldots & O(\infty)
\end{align*}
\]
Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

**Example**

\[
\begin{align*}
h(x) & \rightarrow f(g(x)) & f(x) & \rightarrow f(x) & g(a) & \rightarrow g(a) \\
\text{innermost rewriting:} & h(x) & \rightarrow f(g(x)) & \rightarrow f(g(x)) & \rightarrow & \ldots & \mathcal{O}(\infty)
\end{align*}
\]

- Just ground rewriting?
### Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

#### Example

<table>
<thead>
<tr>
<th>Innermost rewriting:</th>
<th>ground rewriting:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h(x) \rightarrow f(g(x))$</td>
<td>$h(a) \rightarrow f(g(a))$</td>
</tr>
<tr>
<td>$f(x) \rightarrow f(x)$</td>
<td>$f(g(a)) \rightarrow ...$</td>
</tr>
<tr>
<td>$g(a) \xrightarrow{=} g(a)$</td>
<td>$O(\infty)$</td>
</tr>
</tbody>
</table>

- Just ground rewriting?
# Terminating Variants

<table>
<thead>
<tr>
<th><strong>Term Rewriting</strong></th>
<th><strong>Integer Transition Systems</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

## Example

\[
\begin{align*}
 h(x) & \rightarrow f(g(x)) & f(x) & \rightarrow f(x) & g(a) & \implies g(a) \\
\text{innermost rewriting:} & h(x) & \rightarrow f(g(x)) & \rightarrow f(g(x)) & \rightarrow \ldots & \mathcal{O}(\infty) \\
\text{ground rewriting:} & h(a) & \rightarrow f(g(a)) & \implies f(g(a)) & \implies \ldots & \mathcal{O}(1)
\end{align*}
\]

- Just ground rewriting?
### Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

#### Example

- **h(x) → f(g(x))**
- **f(x) → f(x)**
- **g(a) → g(a)**

**innermost rewriting:**

\[
\begin{align*}
  h(x) & \rightarrow f(g(x)) \\
  f(g(x)) & \rightarrow f(g(x)) \\
  & \ldots \\
\end{align*}
\]

\[O(\infty)\]

**ground rewriting:**

\[
\begin{align*}
  h(a) & \rightarrow f(g(a)) \\
  f(g(a)) & \rightarrow f(g(a)) \\
  & \ldots \\
\end{align*}
\]

\[O(1)\]

- **Just ground rewriting?**
- **Add terminating variant of relative rules!**
Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

Example

\[ h(x) \rightarrow f(g(x)) \quad f(x) \rightarrow f(x) \quad g(a) \longrightarrow g(a) \]

innermost rewriting: \[ h(x) \rightarrow f(g(x)) \rightarrow f(g(x)) \rightarrow \ldots \quad O(\infty) \]

ground rewriting: \[ h(a) \rightarrow f(g(a)) \longrightarrow f(g(a)) \longrightarrow \ldots \quad O(1) \]

- Just ground rewriting?
- Add terminating variant of relative rules!

Definition

\( N \) is a terminating variant of \( S \) iff \( N \) terminates and every \( N \)-normal form is an \( S \)-normal form.
### Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

#### Example

- **h(x) → f(g(x))**  
  - **f(x) → f(x)**  
  - **g(a) → g(a)**  
  - **g(a) → a**

  **innermost rewriting:**  
  - **h(x) → f(g(x)) → f(g(x)) → ...**  
  - **O(∞)**

  **ground rewriting:**  
  - **h(a) → f(g(a)) → f(g(a)) → ...**  
  - **O(1)**

- Just ground rewriting?
- Add terminating variant of relative rules!

#### Definition

\( \mathcal{N} \) is a terminating variant of \( S \) iff \( \mathcal{N} \) terminates and every \( \mathcal{N} \)-normal form is an \( S \)-normal form.
Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

**Example**

\[
\begin{align*}
    h(x) & \rightarrow f(g(x)) \\
    f(x) & \rightarrow f(x) \\
    g(a) & \rightarrow g(a) \\
    g(a) & \rightarrow a
\end{align*}
\]

innermost rewriting:

\[
\begin{align*}
    h(x) & \rightarrow f(g(x)) \rightarrow f(g(x)) \rightarrow \ldots \\
    \mathcal{O}(\infty)
\end{align*}
\]

ground rewriting:

\[
\begin{align*}
    h(a) & \rightarrow f(g(a)) \rightarrow f(g(a)) \rightarrow \ldots \\
    \mathcal{O}(1)
\end{align*}
\]

with terminating variant:

\[
\begin{align*}
    h(a) & \rightarrow f(g(a)) \rightarrow f(a) \rightarrow f(a) \rightarrow \ldots \\
\end{align*}
\]

- Just ground rewriting?
- Add terminating variant of relative rules!

**Definition**

\(\mathcal{N}\) is a terminating variant of \(S\) iff \(\mathcal{N}\) terminates and every \(\mathcal{N}\)-normal form is an \(S\)-normal form.
Terminating Variants

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>start terms may have variables</td>
<td>ground start terms only</td>
</tr>
</tbody>
</table>

Example

\[
\begin{align*}
  h(x) & \rightarrow f(g(x)) & f(x) & \rightarrow f(x) & g(a) & \rightarrow g(a) & g(a) & \rightarrow a \\
  \text{innermost rewriting:} & h(x) & \rightarrow f(g(x)) & \rightarrow f(g(x)) & \rightarrow \ldots & \mathcal{O}(\infty) \\
  \text{ground rewriting:} & h(a) & \rightarrow f(g(a)) & \rightarrow f(g(a)) & \rightarrow \ldots & \mathcal{O}(1) \\
  \text{with terminating variant:} & h(a) & \rightarrow f(g(a)) & \rightarrow f(a) & \rightarrow f(a) & \rightarrow \ldots & \mathcal{O}(\infty)
\end{align*}
\]

- Just ground rewriting?
- Add terminating variant of relative rules!

Definition

\( \mathcal{N} \) is a terminating variant of \( S \) iff \( \mathcal{N} \) terminates and every \( \mathcal{N} \)-normal form is an \( S \)-normal form.
Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

Example

\[ f(x) \rightarrow f(g(a)) \quad \text{and} \quad g(b(a)) \rightarrow a \]

Definition

A TRS is completely defined iff its ground normal forms do not contain defined symbols.

TRS not completely defined? ⇝ Add suitable terminating variant!
Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

Example

\[
\begin{align*}
  f(x) &\rightarrow f(g(a)) & g(b(a)) &\rightarrow a
\end{align*}
\]

original TRS:

\[
\begin{align*}
  f(a) &\rightarrow f(g(a)) & f(g(a)) &\rightarrow \ldots
\end{align*}
\]
Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

Example

\[
\begin{align*}
  f(x) & \rightarrow f(g(a)) \\
  g(b(a)) & \rightarrow a
\end{align*}
\]

Original TRS:

\[
\begin{align*}
  f(a) & \rightarrow f(g(a)) \\
  f(g(a)) & \rightarrow f(g(a)) \rightarrow \ldots
\end{align*}
\]

\(O(\infty)\)
Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

**Example**

\[
\begin{align*}
  f(x) & \rightarrow f(g(a)) \\
  g(b(a)) & \rightarrow a
\end{align*}
\]

**original TRS:**

\[
\begin{align*}
  f(a) & \rightarrow f(g(a)) \\
  f(g(a)) & \rightarrow f(g(a)) & \ldots & O(\infty)
\end{align*}
\]

**resulting ITS:**

\[
\begin{align*}
  f(1) & \xrightarrow{1} f(g(1))
\end{align*}
\]
### Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

#### Example

<table>
<thead>
<tr>
<th>Original TRS:</th>
<th>Resulting ITS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) \rightarrow f(g(a)) )</td>
<td>( f(1) \rightarrow f(g(1)) )</td>
</tr>
<tr>
<td>( g(b(a)) \rightarrow a )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

\( O(\infty) \)

**Definition**

A TRS is completely defined iff its ground normal forms do not contain defined symbols.

A TRS not completely defined?

\( \rightarrow \)

Add suitable terminating variant!

\( O(\infty) \)
Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

Example

\[ f(x) \rightarrow f(g(a)) \quad g(b(a)) \rightarrow a \]

original TRS: \[ f(a) \rightarrow f(g(a)) \rightarrow f(g(a)) \rightarrow \ldots \quad O(\infty) \]

resulting ITS: \[ f(1) \stackrel{1}{\rightarrow} f(g(1)) \quad O(1) \]

Definition

A TRS is completely defined iff its ground normal forms do not contain defined symbols.
Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

**Example**

\[
f(x) \rightarrow f(g(a)) \quad \quad g(b(a)) \rightarrow a \quad \quad g(x) \rightarrow a
\]

original TRS:
\[
f(a) \rightarrow f(g(a)) \rightarrow f(g(a)) \rightarrow \ldots \quad O(\infty)
\]

resulting ITS:
\[
f(1) \rightarrow f(g(1)) \quad \quad O(1)
\]

**Definition**

A TRS is completely defined iff its ground normal forms do not contain defined symbols.

TRS not completely defined? \(\rightsquigarrow\) Add suitable terminating variant!
Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

**Example**

<table>
<thead>
<tr>
<th>Original TRS:</th>
<th>Resulting ITS:</th>
<th>ITS after completion:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) \rightarrow f(g(a)) )</td>
<td>( f(1) \rightarrow f(g(1)) )</td>
<td>( f(1) \rightarrow f(g(1)) \rightarrow a )</td>
</tr>
<tr>
<td>( g(b(a)) \rightarrow a )</td>
<td></td>
<td>( f(1) \rightarrow f(g(1)) \rightarrow a )</td>
</tr>
<tr>
<td>( g(x) \rightarrow a )</td>
<td></td>
<td>( f(1) \rightarrow f(g(1)) \rightarrow a )</td>
</tr>
</tbody>
</table>

**Definition**

A TRS is completely defined iff its ground normal forms do not contain defined symbols.

TRS not completely defined? \( \mapsto \) Add suitable terminating variant!
Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

Example

\[ f(x) \rightarrow f(g(a)) \quad g(b(a)) \rightarrow a \quad g(x) \rightarrow a \]

original TRS:

\[
\begin{align*}
    f(a) & \rightarrow f(g(a)) \rightarrow f(g(a)) \rightarrow \ldots \\
    & \in \bigO(\infty)
\end{align*}
\]

resulting ITS:

\[
\begin{align*}
    f(1) & \rightarrow f(g(1)) \\
    & \in \bigO(1)
\end{align*}
\]

ITS after completion:

\[
\begin{align*}
    f(1) & \rightarrow f(g(1)) \rightarrow f(1) \rightarrow f(g(1)) \rightarrow \ldots \\
    & \in \bigO(\infty)
\end{align*}
\]

Definition

A TRS is completely defined iff its ground normal forms do not contain defined symbols.

TRS not completely defined? \(\bowtie\) Add suitable terminating variant!
Ensuring Complete Definedness

<table>
<thead>
<tr>
<th>Term Rewriting</th>
<th>Integer Transition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary matchers</td>
<td>integer substitutions only</td>
</tr>
</tbody>
</table>

Example

\[
\begin{align*}
  f(x) & \rightarrow f(g(a)) \\
  g(b(a)) & \rightarrow a \\
  g(x) & \rightarrow a
\end{align*}
\]

original TRS:  
\[
\begin{align*}
  f(a) & \rightarrow f(g(a)) \rightarrow f(g(a)) \rightarrow \ldots \\
  O(\infty)
\end{align*}
\]

resulting ITS:  
\[
\begin{align*}
  f(1) & \rightarrow f(g(1)) \\
  O(1)
\end{align*}
\]

ITS after completion:  
\[
\begin{align*}
  f(1) & \rightarrow f(g(1)) \rightarrow f(1) \rightarrow f(g(1)) \rightarrow \ldots \\
  O(\infty)
\end{align*}
\]

Definition

A TRS is completely defined iff its well-typed ground normal forms do not contain defined symbols.

TRS not completely defined? \(\curvearrowright\) Add suitable terminating variant!
## Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isort(xs', ys)  \rightarrow  ys</code></td>
<td><code>xs' = 1</code></td>
</tr>
<tr>
<td><code>isort(xs', ys)  \rightarrow  isort(xs, insert(x, ys))</code></td>
<td><code>xs' = 1 + x + xs</code></td>
</tr>
<tr>
<td><code>insert(x, ys')  \rightarrow  2 + x</code></td>
<td><code>ys' = 1</code></td>
</tr>
<tr>
<td><code>insert(x, ys')  \rightarrow  if(b, x, ys')</code></td>
<td><code>ys' = 1 + y + ys \land b \leq 1</code></td>
</tr>
<tr>
<td><code>if(b, x, ys')  \rightarrow  1 + y + insert(x, ys)</code></td>
<td><code>b = 1 \land ys' = 1 + y + ys</code></td>
</tr>
<tr>
<td><code>if(b, x, ys')  \rightarrow  1 + ys'</code></td>
<td><code>b = 1 \land ys' = 1 + y + ys</code></td>
</tr>
</tbody>
</table>

1. abstract terms to integers
2. analyse result size for bottom-SCC using standard ITS tools
3. analyse runtime of bottom-SCC using standard ITS tools
Call Graph & Bottom SCCs

isort

insert

if
Call Graph & Bottom SCCs

isort

insert

if

if
### Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{isort}(xs', ys) \downarrow 1 \rightarrow ys )</td>
<td>( xs' = 1 )</td>
<td></td>
</tr>
<tr>
<td>( \text{isort}(xs', ys) \downarrow 1 \rightarrow \text{isort}(xs, \text{insert}(x, ys)) )</td>
<td>( xs' = 1 + x + xs )</td>
<td></td>
</tr>
<tr>
<td>( \text{insert}(x, ys') \downarrow 1 \rightarrow 2 + x )</td>
<td>( ys' = 1 )</td>
<td></td>
</tr>
<tr>
<td>( \text{insert}(x, ys') \downarrow 1 \rightarrow \text{if}(b, x, ys') )</td>
<td>( ys' = 1 + y + ys \land b \leq 1 )</td>
<td></td>
</tr>
<tr>
<td>( \text{if}(b, x, ys') \downarrow 1 \rightarrow 1 + y + \text{insert}(x, ys) )</td>
<td>( b = 1 \land ys' = 1 + y + ys )</td>
<td></td>
</tr>
<tr>
<td>( \text{if}(b, x, ys') \downarrow 1 \rightarrow 1 + ys' )</td>
<td>( b = 1 \land ys' = 1 + y + ys )</td>
<td></td>
</tr>
</tbody>
</table>

1. abstract terms to integers
2. analyse result size for bottom-SCC using standard ITS tools
3. analyse runtime of bottom-SCC using standard ITS tools
Bird’s Eye View

Example

\[
\text{isort}(xs', ys) \overset{1}{\rightarrow} ys \quad | \quad xs' = 1
\]
\[
\text{isort}(xs', ys) \overset{1}{\rightarrow} \text{isort}(xs, \text{insert}(x, ys)) \quad | \quad xs' = 1 + x + xs
\]
\[
\text{insert}(x, ys') \overset{1}{\rightarrow} 2 + x \quad | \quad ys' = 1
\]
\[
\text{insert}(x, ys') \overset{1}{\rightarrow} \text{if}(b, x, ys') \quad | \quad ys' = 1 + y + ys \land b \leq 1
\]
\[
\text{if}(b, x, ys') \overset{1}{\rightarrow} 1 + y + \text{insert}(x, ys) \quad | \quad b = 1 \land ys' = 1 + y + ys
\]
\[
\text{if}(b, x, ys') \overset{1}{\rightarrow} 1 + ys' \quad | \quad b = 1 \land ys' = 1 + y + ys
\]

1 abstract terms to integers
2 analyse result size for bottom-SCC using standard ITS tools
3 analyse runtime of bottom-SCC using standard ITS tools
Analyse Size Using Standard ITS Tools
Using Runtime Analysis to Compute Size Bounds

**Idea:** time bound for \texttt{insert} in transformed rules gives size bound for \texttt{insert} in original rules

**Example**

- \texttt{insert}(x, ys') \xrightarrow{1} 2 + x \quad | \quad ys' = 1
- \texttt{insert}(x, ys') \xrightarrow{1} \text{if}(b, x, ys') \quad | \quad ys' = 1 + y + ys \land b \leq 1
- \texttt{if}(b, x, ys') \xrightarrow{1} 1 + y + \texttt{insert}(x, ys) \quad | \quad b = 1 \land ys' = 1 + y + ys
- \texttt{if}(b, x, ys') \xrightarrow{1} 1 + ys' \quad | \quad b = 1 \land ys' = 1 + y + ys
Using Runtime Analysis to Compute Size Bounds

**Idea:** time bound for `insert` in transformed rules gives size bound for `insert` in original rules

**Example**

\[
\begin{align*}
\text{insert}(x, ys') & \rightarrow 1 \rightarrow 2 + x & | & ys' = 1 \\
\text{insert}(x, ys') & \rightarrow 1 \rightarrow \text{if}(b, x, ys') & | & ys' = 1 + y + ys \land b \leq 1 \\
\text{if}(b, x, ys') & \rightarrow 1 \rightarrow 1 + y + \text{insert}(x, ys) & | & b = 1 \land ys' = 1 + y + ys \\
\text{if}(b, x, ys') & \rightarrow 1 \rightarrow 1 + ys' & | & b = 1 \land ys' = 1 + y + ys
\end{align*}
\]

**Idea:** move “integer context” to weights
Using Runtime Analysis to Compute Size Bounds

**Idea:** time bound for \texttt{insert} in transformed rules gives size bound for \texttt{insert} in original rules

**Example**

\begin{align*}
\text{insert}(x, ys') & \xrightarrow{2+x} & 2 + x & \mid y = 1 \\
\text{insert}(x, ys') & \xrightarrow{1} & \text{if}(b, x, ys') & \mid y = 1 + y + ys \land b \leq 1 \\
\text{if}(b, x, ys') & \xrightarrow{1} & 1 + y + \text{insert}(x, ys) & \mid b = 1 \land ys' = 1 + y + ys \\
\text{if}(b, x, ys') & \xrightarrow{1} & 1 + y s' & \mid b = 1 \land ys' = 1 + y + ys
\end{align*}

**Idea:** move “integer context” to weights
**Idea:** time bound for \texttt{insert} in transformed rules gives size bound for \texttt{insert} in original rules

**Example**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Size Bound</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{insert}(x, ys') \xrightarrow{2+x} 2 + x</td>
<td>ys' = 1</td>
<td></td>
</tr>
<tr>
<td>\texttt{insert}(x, ys') \xrightarrow{0} \texttt{if}(b, x, ys')</td>
<td>ys' = 1 + y + ys \land b \leq 1</td>
<td></td>
</tr>
<tr>
<td>\texttt{if}(b, x, ys') \xrightarrow{1} 1 + y + \texttt{insert}(x, ys)</td>
<td>b = 1 \land ys' = 1 + y + ys</td>
<td></td>
</tr>
<tr>
<td>\texttt{if}(b, x, ys') \xrightarrow{1} 1 + ys'</td>
<td>b = 1 \land ys' = 1 + y + ys</td>
<td></td>
</tr>
</tbody>
</table>

**Idea:** move “integer context” to weights
Using Runtime Analysis to Compute Size Bounds

**Idea:** time bound for `insert` in transformed rules gives size bound for `insert` in original rules

**Example**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transformation</th>
<th>New Value</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(x, ys')</code></td>
<td>$\frac{2+x}{2}$</td>
<td>$2 + x$</td>
<td>$ys' = 1$</td>
</tr>
<tr>
<td><code>insert(x, ys')</code></td>
<td>$\frac{0}{1+y}$</td>
<td><code>if(b, x, ys')</code></td>
<td>$ys' = 1 + y + ys \land b \leq 1$</td>
</tr>
<tr>
<td><code>if(b, x, ys')</code></td>
<td>$\frac{1+y}{1}$</td>
<td>$1 + y + \text{insert}(x, ys)$</td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
</tr>
<tr>
<td><code>if(b, x, ys')</code></td>
<td>$\frac{1}{1}$</td>
<td>$1 + ys'$</td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
</tr>
</tbody>
</table>

**Idea:** move “integer context” to weights
Using Runtime Analysis to Compute Size Bounds

Idea: time bound for insert in transformed rules gives size bound for insert in original rules

Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Right-hand side</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert (x, ys') (\xrightarrow{2+x}) (2 + x) (\mid ys' = 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert (x, ys') (\xrightarrow{0}) if ((b, x, ys')) (\mid ys' = 1 + y + ys \land b \leq 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if ((b, x, ys')) (\xrightarrow{1+y}) (1 + y + \text{insert}(x, ys)) (\mid b = 1 \land ys' = 1 + y + ys)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if ((b, x, ys')) (\xrightarrow{1+ys'}) (1 + ys') (\mid b = 1 \land ys' = 1 + y + ys)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Idea: move “integer context” to weights
Using Runtime Analysis to Compute Size Bounds

Idea: time bound for *insert* in transformed rules gives size bound for *insert* in original rules

Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Time Bound</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(x, ys')</code></td>
<td>$2+x$</td>
<td>$ys' = 1$</td>
</tr>
<tr>
<td><code>insert(x, ys')</code></td>
<td>$0$</td>
<td>$ys' = 1 + y + ys \land b \leq 1$</td>
</tr>
<tr>
<td><code>if(b, x, ys')</code></td>
<td>$1+y$</td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
</tr>
<tr>
<td><code>if(b, x, ys')</code></td>
<td>$1+ys'$</td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
</tr>
</tbody>
</table>

Idea: move “integer context” to weights $\bowtie \text{sz}(\text{insert}(x, ys')) \leq 1 + x + ys'$
Using Runtime Analysis to Compute Size Bounds

**Idea:** time bound for `insert` in transformed rules gives size bound for `insert` in original rules

**Example**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transformation</th>
<th>Size Bound</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(x, y)</code></td>
<td><code>2 + x</code></td>
<td><code>2 + x</code></td>
<td><code>ys' = 1</code></td>
</tr>
</tbody>
</table>
| `insert(x, y)` | `0` | `if(b, x, y)` | `ys' = 1 + y + y 
| `if(b, x, y)` | `1 + y` | `1 + y + insert(x, y)` | `b = 1 \land y \leq 1` |
| `if(b, x, y)` | `1 + y` | `1 + y` | `b = 1 \land y \leq 1` |

**Idea:** move “integer context” to weights \( \sim sz(insert(x, y')) \leq 1 + x + y's' \)

**Example**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transformation</th>
<th>Size Bound</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f(x)</code></td>
<td><code>1</code></td>
<td><code>2 + x \cdot f(x - 1)</code></td>
<td><code>x &gt; 0</code></td>
</tr>
</tbody>
</table>
Using Runtime Analysis to Compute Size Bounds

**Idea:** time bound for `insert` in transformed rules gives size bound for `insert` in original rules

### Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transformation</th>
<th>Size Bound</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(x, ys')</code></td>
<td>$2+x \rightarrow 2+x$</td>
<td>$ys' = 1$</td>
<td></td>
</tr>
<tr>
<td><code>insert(x, ys')</code></td>
<td>$0 \rightarrow \text{if}(b, x, ys')$</td>
<td>$ys' = 1 + y + ys \land b \leq 1$</td>
<td></td>
</tr>
<tr>
<td><code>if(b, x, ys')</code></td>
<td>$1+y \rightarrow 1+y + \text{insert}(x, ys)$</td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
<td></td>
</tr>
<tr>
<td><code>if(b, x, ys')</code></td>
<td>$1+ys' \rightarrow 1+ys'$</td>
<td>$b = 1 \land ys' = 1 + y + ys$</td>
<td></td>
</tr>
</tbody>
</table>

### Idea: move “integer context” to weights

$\bowtie \text{sz}(\text{insert}(x, ys')) \leq 1 + x + ys'$

### Example

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transformation</th>
<th>Size Bound</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$</td>
<td>$1 \rightarrow 2 + x \cdot f(x - 1)$</td>
<td>$x &gt; 0$</td>
<td></td>
</tr>
</tbody>
</table>

**Idea:** use accumulator
Using Runtime Analysis to Compute Size Bounds

**Idea:** time bound for `insert` in transformed rules gives size bound for `insert` in original rules

**Example**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transformation</th>
<th>Size Bound</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(x, ys')</code></td>
<td><code>2 + x</code></td>
<td><code>2 + x</code></td>
<td><code>ys' = 1</code></td>
</tr>
<tr>
<td><code>insert(x, ys')</code></td>
<td><code>0</code></td>
<td><code>if (b, x, ys')</code></td>
<td><code>ys' = 1 + y + ys ∧ b ≤ 1</code></td>
</tr>
<tr>
<td><code>if (b, x, ys')</code></td>
<td><code>1 + y</code></td>
<td><code>1 + y + insert(x, ys)</code></td>
<td><code>b = 1 ∧ ys' = 1 + y + ys</code></td>
</tr>
<tr>
<td><code>if (b, x, ys')</code></td>
<td><code>1 + ys'</code></td>
<td><code>1 + ys'</code></td>
<td><code>b = 1 ∧ ys' = 1 + y + ys</code></td>
</tr>
</tbody>
</table>

**Idea:** move “integer context” to weights \( \sim sz(insert(x, ys')) \leq 1 + x + ys' \)

**Example**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transformation</th>
<th>Result</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f(x)</code></td>
<td><code>1</code></td>
<td><code>2 + x \cdot f(x - 1)</code></td>
<td><code>x &gt; 0</code></td>
</tr>
<tr>
<td><code>f(x, acc)</code></td>
<td><code>acc \cdot 2</code></td>
<td><code>2 + x \cdot f(x - 1, acc \cdot x)</code></td>
<td><code>x &gt; 0</code></td>
</tr>
</tbody>
</table>

**Idea:** use accumulator
## Example

<table>
<thead>
<tr>
<th>Term</th>
<th>Rule</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{isort}(xs', ys)</td>
<td>\rightarrow 1 \rightarrow ys</td>
<td>\quad \quad</td>
</tr>
<tr>
<td>\text{isort}(xs', ys)</td>
<td>\rightarrow \text{isort}(xs, \text{insert}(x, ys))</td>
<td>\quad \quad</td>
</tr>
<tr>
<td>\text{insert}(x, ys')</td>
<td>\rightarrow 2 + x</td>
<td>\quad \quad</td>
</tr>
<tr>
<td>\text{insert}(x, ys')</td>
<td>\rightarrow \text{if}(b, x, ys')</td>
<td>\quad \quad</td>
</tr>
<tr>
<td>\text{if}(b, x, ys')</td>
<td>\rightarrow 1 + y + \text{insert}(x, y_{}</td>
<td>\quad \quad</td>
</tr>
<tr>
<td>\text{if}(b, x, ys')</td>
<td>\rightarrow 1 + y_{}'</td>
<td>\quad \quad</td>
</tr>
</tbody>
</table>

1. abstract terms to integers
2. analyse result size for bottom-SCC using standard ITS tools
3. analyse runtime of bottom-SCC using standard ITS tools
### Example

\[
isort(xs', ys) \xrightarrow{1} ys \quad | \
xs' = 1
\]
\[
isort(xs', ys) \xrightarrow{1} \text{isort}(xs, \text{insert}(x, ys)) \quad | \
xs' = 1 + x + xs
\]

1. abstract terms to integers
2. analyse result size for bottom-SCC using standard ITS tools
3. analyse runtime of bottom-SCC using standard ITS tools
Analyse Runtime Using Standard Tools
Removing Nested Function Calls

**Example**

\[
\begin{align*}
isort(xs', ys) & \xrightarrow{1} ys \quad | \quad xs' = 1 \\
isort(xs', ys) & \xrightarrow{1} \text{isort}(xs, \text{insert}(x, ys)) \quad | \quad xs' = 1 + x + xs
\end{align*}
\]

- \(\text{sz(insert}(x, ys)) \leq 1 + x + ys\)
- \(\text{rt(insert}(x, ys)) \leq 2 \cdot ys\)
Removing Nested Function Calls

Example

\[
\begin{align*}
isort(xs', ys) & \overset{1}{\Rightarrow} ys & | & xs' = 1 \\
isort(xs', ys) & \overset{1}{\Rightarrow} isort(xs, \text{insert}(x, ys)) & | & xs' = 1 + x + xs
\end{align*}
\]

- \( sz(\text{insert}(x, ys)) \leq 1 + x + ys \)
- \( rt(\text{insert}(x, ys)) \leq 2 \cdot ys \)
- add costs of nested function call
Removing Nested Function Calls

Example

\[
\begin{align*}
\text{isort}(xs', ys) & \xrightarrow{1} ys & | & xs' = 1 \\
\text{isort}(xs', ys) & \xrightarrow{1+2\cdot ys} \text{isort}(xs, \text{insert}(x, ys)) & | & xs' = 1 + x + xs
\end{align*}
\]

- \(\text{sz}(\text{insert}(x, ys)) \leq 1 + x + ys\)
- \(\text{rt}(\text{insert}(x, ys)) \leq 2 \cdot ys\)
- add costs of nested function call
Removing Nested Function Calls

Example

\[
\begin{align*}
\text{isort}(xs', ys) & \xrightarrow{1} ys & | & xs' = 1 \\
\text{isort}(xs', ys) & \xrightarrow{1+2\cdot ys} \text{isort}(xs, \text{insert}(x, ys)) & | & xs' = 1+x+xs
\end{align*}
\]

- \(\text{sz}(\text{insert}(x, ys)) \leq 1 + x + ys\)
- \(\text{rt}(\text{insert}(x, ys)) \leq 2 \cdot ys\)
- add costs of nested function call
- replace nested function call by fresh variable \(x_f\)
## Removing Nested Function Calls

### Example

| isort($xs', ys$) | $\rightarrow$ | $ys$ | $|$ | $xs' = 1$ |
|------------------|---------------|------|---|----------------|
| isort($xs', ys$) | $\frac{1+2\cdot ys}{2}$ | $\text{isort}(xs, x_f)$ | $|$ | $xs' = 1 + x + xs$ |

- $\text{sz}(\text{insert}(x, ys)) \leq 1 + x + ys$
- $\text{rt}(\text{insert}(x, ys)) \leq 2 \cdot ys$
- add costs of nested function call
- replace nested function call by fresh variable $x_f$
Removing Nested Function Calls

**Example**

\[
\begin{align*}
\text{isort}(x's', y's) & \xrightarrow{1} y's & | & x's' = 1 \\
\text{isort}(x's', y's) & \xrightarrow{1+2\cdot y's} \text{isort}(x's, x_f) & | & x's' = 1 + x + x's
\end{align*}
\]

- \(\text{sz(insert}(x, y's)) \leq 1 + x + y's\)
- \(\text{rt(insert}(x, y's)) \leq 2 \cdot y's\)
- add costs of nested function call
- replace nested function call by fresh variable \(x_f\)
- add constraint \(\text{“}x_f \leq \text{size bound}”\)
Removing Nested Function Calls

Example

\[
\text{isort}(xs', ys) \xrightarrow{1} ys \\
\text{isort}(xs', ys) \xrightarrow{1+2\cdot ys} \text{isort}(xs, x_f) \\
\text{sz}(\text{insert}(x, ys)) \leq 1 + x + ys \\
\text{rt}(\text{insert}(x, ys)) \leq 2 \cdot ys \\
\text{add costs of nested function call} \\
\text{replace nested function call by fresh variable } x_f \\
\text{add constraint } "x_f \leq \text{size bound}" \\
\]
Removing Nested Function Calls

Example

\[
\begin{align*}
\text{isort}(xs', ys) & \xrightarrow{1} ys \\
\text{isort}(xs', ys) & \xrightarrow{1+2\cdot ys} \text{isort}(xs, xf) \\
& \quad | \quad xs' = 1 \\
& \quad | \quad xs' = 1 + x + xs \land xf \leq 1 + x + ys
\end{align*}
\]

- \(sz(insert(x, ys)) \leq 1 + x + ys\)
- \(rt(insert(x, ys)) \leq 2 \cdot ys\)
- add costs of nested function call
- replace nested function call by fresh variable \(xf\)
- add constraint "\(xf \leq \) size bound"
- \(rt(isort(xs', ys)) \leq O(xs'^2 + xs' \cdot ys)\)
Removing Nested Function Calls

### Example

\[
\text{isort}(xs',ys) \xrightarrow{1} ys \\
\text{isort}(xs',ys) \xrightarrow{1+2\cdot ys} \text{isort}(xs,x_f) \\
| \quad xs' = 1 \\
| \quad xs' = 1 + x + xs \land x_f \leq 1 + x + ys
\]

- \( sz(\text{insert}(x,ys)) \leq 1 + x + ys \)
- \( rt(\text{insert}(x,ys)) \leq 2 \cdot ys \)
- add costs of nested function call
- replace nested function call by fresh variable \( x_f \)
- add constraint “\( x_f \leq \) size bound”
- \( rt(\text{isort}(xs',ys)) \leq O(xs'^2 + xs' \cdot ys) \)
- similar techniques to eliminate outer function calls
Removing Nested Function Calls

**Example**

\[
\begin{aligned}
\text{isort}(xs', ys) & \xrightarrow{1} ys \\
\text{isort}(xs', ys) & \xrightarrow{1 + 2 \cdot ys} \text{isort}(xs, xf) \\
\end{aligned}
\]

\[
\begin{aligned}
| \text{xs'} = 1 & | \text{xs'} = 1 + x + x \text{s} \land xf \leq 1 + x + ys \\
\end{aligned}
\]

- \( \text{sz}(\text{insert}(x, ys)) \leq 1 + x + ys \)
- \( \text{rt}(\text{insert}(x, ys)) \leq 2 \cdot ys \)
- add costs of nested function call
- replace nested function call by fresh variable \( xf \)
- add constraint “\( xf \leq \text{size bound} \)”

\( \text{rt}(\text{isort}(xs', ys)) \leq \mathcal{O}(xs'^2 + xs' \cdot ys) \)

- similar techniques to eliminate outer function calls
  
  \[
  \text{times}(s(x), y) \rightarrow \text{plus}(\text{times}(x, y), y)
  \]
Removing Nested Function Calls

Example

\[
\begin{align*}
\text{isort}(xs', ys) & \overset{1}{\rightarrow} ys & | & xs' = 1 \\
\text{isort}(xs', ys) & \overset{1+2\cdot ys}{\rightarrow} \text{isort}(xs, xf) & | & xs' = 1 + x + xs \land xf \leq 1 + x + ys
\end{align*}
\]

- \(sz(\text{insert}(x, ys)) \leq 1 + x + ys\)
- \(rt(\text{insert}(x, ys)) \leq 2 \cdot ys\)
- add costs of nested function call
- replace nested function call by fresh variable \(xf\)
- add constraint \("xf \leq \text{size bound}"\)

\[
\text{rt(isort}(xs', ys)) \leq \mathcal{O}(xs'^2 + xs' \cdot ys)
\]

- similar techniques to eliminate outer function calls \(\rightarrow\) see paper!

\[
\text{times}(s(x), y) \rightarrow \text{plus}(\text{times}(x, y), y)
\]
Experiments

ITS tools CoFloCo, KoAT, and PUBS used as back-ends.
Experiments

ITS tools CoFloCo, KoAT, and PUBS used as back-ends.

Results on the TPDB (922 examples):
Experiments

ITS tools CoFloCo, KoAT, and PUBS used as back-ends.

Results on the TPDB (922 examples):

- AProVE + ITS back-end finds better bounds than AProVE & TcT for 127 TRSs
- transformation a useful additional inference technique for upper bounds
Abstraction from terms to integers

Modular bottom-up approach using standard ITS tools

Approach complements and improves state of the art

Note: abstraction **hard-coded** to term size

⇒ Future work: more flexible approach?
app(nil, y) → y  |  app(add(n, x), y) → add(n, app(x, y))
reverse(nil) → nil  |  reverse(add(n, x)) → app(reverse(x), add(n, nil))
shuffle(nil) → nil  |  shuffle(add(n, x)) → add(n, shuffle(reverse(x))))
AProVE finds (tight) upper bound $O(n^4)$ for $dc_R$: 

\[
\begin{align*}
\text{app}(\text{nil}, y) &\rightarrow y & \text{app}(\text{add}(n, x), y) &\rightarrow \text{add}(n, \text{app}(x, y)) \\
\text{reverse}(\text{nil}) &\rightarrow \text{nil} & \text{reverse}(\text{add}(n, x)) &\rightarrow \text{app}(\text{reverse}(x), \text{add}(n, \text{nil})) \\
\text{shuffle}(\text{nil}) &\rightarrow \text{nil} & \text{shuffle}(\text{add}(n, x)) &\rightarrow \text{add}(n, \text{shuffle}(\text{reverse}(x))))
\end{align*}
\]
AProVE finds (tight) upper bound $O(n^4)$ for $dc_R$:

1. Add generator rules $\mathcal{G}$, so analyse $rc_{R/\mathcal{G}}$ instead (FroCoS’19)
app(nil, y) → y | app(add(n, x), y) → add(n, app(x, y))
reverse(nil) → nil | reverse(add(n, x)) → app(reverse(x), add(n, nil))
shuffle(nil) → nil | shuffle(add(n, x)) → add(n, shuffle(reverse(x))))

AProVE finds (tight) upper bound $O(n^4)$ for $dc_{\mathcal{R}}$:

1. Add generator rules $\mathcal{G}$, so analyse $rc_{\mathcal{R}/\mathcal{G}}$ instead (FroCoS’19)
2. Detect: innermost is worst case here, analyse $irc_{\mathcal{R}/\mathcal{G}}$ instead (LPAR’17)
AProVE finds (tight) upper bound $\mathcal{O}(n^4)$ for $\text{dc}_R$:

1. Add generator rules $G$, so analyse $\text{rc}_{R/G}$ instead (FroCoS’19)
2. Detect: innermost is worst case here, analyse $\text{irc}_{R/G}$ instead (LPAR’17)
3. Transform TRS to Recursive Integer Transition System (RITS), analyse complexity of RITS instead (FroCoS’17)
AProVE finds (tight) upper bound $\mathcal{O}(n^4)$ for $\text{dc}_R$:

1. Add generator rules $\mathcal{G}$, so analyse $\text{rc}_{R/\mathcal{G}}$ instead (FroCoS’19)
2. Detect: innermost is worst case here, analyse $\text{irc}_{R/\mathcal{G}}$ instead (LPAR’17)
3. Transform TRS to Recursive Integer Transition System (RITS), analyse complexity of RITS instead (FroCoS’17)
4. ITS tools CoFloCo and KoAT find upper bounds for runtime and size of individual RITS functions, combine to complexity of RITS
app(nil, y) → y          app(add(n, x), y) → add(n, app(x, y))
reverse(nil) → nil        reverse(add(n, x)) → app(reverse(x), add(n, nil))
shuffle(nil) → nil        shuffle(add(n, x)) → add(n, shuffle(reverse(x))))

AProVE finds (tight) upper bound $O(n^4)$ for $dc_\mathcal{R}$:
1. Add generator rules $\mathcal{G}$, so analyse $rc_\mathcal{R}/\mathcal{G}$ instead (FroCoS’19)
2. Detect: innermost is worst case here, analyse $irc_\mathcal{R}/\mathcal{G}$ instead (LPAR’17)
3. Transform TRS to Recursive Integer Transition System (RITS), analyse complexity of RITS instead (FroCoS’17)
4. ITS tools CoFloCo and KoAT find upper bounds for runtime and size of individual RITS functions, combine to complexity of RITS
5. Upper bound $O(n^4)$ for RITS complexity carries over to $dc_\mathcal{R}$ of input!

AProVE finds lower bound $\Omega(n^3)$ for $dc_\mathcal{R}$ using induction technique.
Automated tools for TRS Complexity at the Termination Competition 2022:

- AProVE: https://aprove.informatik.rwth-aachen.de/
- TcT: https://tcs-informatik.uibk.ac.at/tools/tct/

For TcT Web, use only VAR and RULES entries in the text format and configure other aspects (e.g., start terms) in the web interface.
Automated tools for TRS Complexity at the Termination Competition 2022:

- AProVE: https://aprove.informatik.rwth-aachen.de/
- TcT: https://tcs-informatik.uibk.ac.at/tools/tct/

Web interfaces available:

- AProVE: https://aprove.informatik.rwth-aachen.de/interface
- TcT: http://colo6-c703.uibk.ac.at/tct/tct-trs/

---

57 For TcT Web, use only VAR and RULES entries in the text format and configure other aspects (e.g., start terms) in the web interface.
Automated tools for TRS Complexity at the Termination Competition 2022:

- **AProVE**: [https://aprove.informatik.rwth-aachen.de/](https://aprove.informatik.rwth-aachen.de/)
- **TcT**: [https://tcs-informatik.uibk.ac.at/tools/tct/](https://tcs-informatik.uibk.ac.at/tools/tct/)

Web interfaces available:

- **AProVE**: [https://aprove.informatik.rwth-aachen.de/interface](https://aprove.informatik.rwth-aachen.de/interface)
- **TcT**: [http://colo6-c703.uibk.ac.at/tct/tct-trs/](http://colo6-c703.uibk.ac.at/tct/tct-trs/)

Input format for runtime complexity:\(^{57}\)

```
(VAR x y)
(GOAL COMPLEXITY)
(STARTTERM CONSTRUCTOR-BASED)
(RULES
  plus(0, y) -> y
  plus(s(x), y) -> s(plus(x, y))
)
```

---

\(^{57}\)For TcT Web, use only VAR and RULES entries in the text format and configure other aspects (e.g., start terms) in the web interface.
Innermost runtime complexity:

(VAR x y)
(GOAL COMPLEXITY)
(STARTTERM CONSTRUCTOR-BASED)
(STRATEGY INNERMOST)
(RULES
   plus(0, y) -> y
   plus(s(x), y) -> s(plus(x, y))
)
Derivational complexity:

(VAR x y)
(GOAL COMPLEXITY)
(STARTTERM UNRESTRICTED)
(RULES
   plus(0, y) -> y
   plus(s(x), y) -> s(plus(x, y))
)
Innermost derivational complexity:

(VAR x y)
(GOAL COMPLEXITY)
(STARTTERM UNRESTRICTED)
(STRATEGY INNERMOST)
(RULES
   plus(0, y) -> y
   plus(s(x), y) -> s(plus(x, y))
 )
A Landscape of Complexity Properties and Transformations

idc, irc: like dc, rc, but for *innermost* rewriting

**TRS**

FroCoS'19

dc → rc

FroCoS'19

idc → irc

LPAR'17

Rec. ITS irc

FroCoS'17

ITS irc

FroCoS'17

FroCoS'17
A Landscape of Complexity Properties and Transformations

idc, irc: like dc, rc, but for *innermost* rewriting

**OCaml**

- dc \(\rightarrow\) FroCoS'19 \(\rightarrow\) rc

**Java**

- idc \(\rightarrow\) FroCoS'19 \(\rightarrow\) irc

**Prolog**

- TRS

- FroCoS'17

- LPAR'17

- Rec. ITS irc

- ITS irc
A Landscape of Complexity Properties and Transformations

idc, irc: like dc, rc, but for *innermost* rewriting

---


59 G. Moser, M. Schaper: *From Jinja bytecode to term rewriting: A complexity reflecting transformation*, IC ’18

60 J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, C. Fuhs: *Symbolic evaluation graphs and term rewriting: A general methodology for analyzing logic programs*, PPDP ’12
Complexity analysis for functional programs (OCaml) by translation to term rewriting
Complexity analysis for functional programs (OCaml) by translation to term rewriting

Challenge for translation to TRS: OCaml is higher-order – functions can take functions as arguments: $\text{map}(F, xs)$
Complexity analysis for functional programs (OCaml) by translation to term rewriting

Challenge for translation to TRS: OCaml is higher-order – functions can take functions as arguments: \( \text{map}(F, xs) \)

Solution:
- Defunctionalisation to: \( \text{a}(\text{a}(\text{map}, F), xs) \)
- Analyse start term with non-functional parameter types, then partially evaluate functions to instantiate higher-order variables
- Further program transformations
  \[ \Rightarrow \text{First-order TRS } \mathcal{R} \text{ with } rc_{\mathcal{R}}(n) \text{ an upper bound for the complexity of the OCaml program} \]
Complexity analysis for Prolog programs and for Java programs by translation to term rewriting
Complexity analysis for Prolog programs and for Java programs by translation to term rewriting

Common ideas:

- Analyse program via symbolic execution and generalisation (a form of abstract interpretation\(^{61}\))
- Deal with language specifics in program analysis
- Extract TRS \( R \) such that \( rc_R(n) \) is provably at least as high as runtime of program on input of size \( n \)
- Can represent tree structures of program as terms in TRS!

\(^{61}\) P. Cousot, R. Cousot: *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, POPL ’77
Current Developments

- amortised complexity analysis for term rewriting\textsuperscript{62}

\textsuperscript{62} G. Moser, M. Schneckenreither: \textit{Automated amortised resource analysis for term rewrite systems}, SCP '20
Current Developments

- **amortised** complexity analysis for term rewriting\(^{62}\)
- **probabilistic** term rewriting \(\rightarrow\) upper bounds on **expected runtime**\(^{63}\)

---

\(^{62}\) G. Moser, M. Schneckenreither: *Automated amortised resource analysis for term rewrite systems*, SCP '20

\(^{63}\) M. Avanzini, U. Dal Lago, A. Yamada: *On probabilistic term rewriting*, SCP '20
amortised complexity analysis for term rewriting\textsuperscript{62}

probabilistic term rewriting $\rightarrow$ upper bounds on expected runtime\textsuperscript{63}

complexity analysis for logically constrained rewriting with built-in data types from SMT theories (integers, booleans, arrays, . . . )\textsuperscript{64}

\textsuperscript{62} G. Moser, M. Schneckenreither: Automated amortised resource analysis for term rewrite systems, SCP ’20

\textsuperscript{63} M. Avanzini, U. Dal Lago, A. Yamada: On probabilistic term rewriting, SCP ’20

\textsuperscript{64} S. Winkler, G. Moser: Runtime complexity analysis of logically constrained rewriting, LOPSTR ’20
Current Developments

- **amortised** complexity analysis for term rewriting\(^{62}\)
- **probabilistic** term rewriting \(\rightarrow\) upper bounds on expected runtime\(^{63}\)
- complexity analysis for **logically constrained rewriting** with built-in data types from SMT theories (integers, booleans, arrays, . . . )\(^{64}\)
- direct analysis of complexity for **higher-order term rewriting**\(^{65}\)

---

\(^{62}\) G. Moser, M. Schneckenreither: *Automated amortised resource analysis for term rewrite systems*, SCP '20

\(^{63}\) M. Avanzini, U. Dal Lago, A. Yamada: *On probabilistic term rewriting*, SCP '20

\(^{64}\) S. Winkler, G. Moser: *Runtime complexity analysis of logically constrained rewriting*, LOPSTR '20

\(^{65}\) C. Kop, D. Vale: *Tuple interpretations for higher-order rewriting*, FSCD '21
Current Developments

- **amortised** complexity analysis for term rewriting\(^{62}\)
- **probabilistic** term rewriting → upper bounds on *expected runtime*\(^{63}\)
- complexity analysis for **logically constrained rewriting** with built-in data types from SMT theories (integers, booleans, arrays, . . .)\(^{64}\)
- direct analysis of complexity for **higher-order term rewriting**\(^{65}\)
- analysis of **parallel**-innermost runtime complexity\(^{66}\)

---

\(^{62}\) G. Moser, M. Schneckenreither: *Automated amortised resource analysis for term rewrite systems*, SCP '20

\(^{63}\) M. Avanzini, U. Dal Lago, A. Yamada: *On probabilistic term rewriting*, SCP '20

\(^{64}\) S. Winkler, G. Moser: *Runtime complexity analysis of logically constrained rewriting*, LOPSTR '20

\(^{65}\) C. Kop, D. Vale: *Tuple interpretations for higher-order rewriting*, FSCD '21

\(^{66}\) T. Baudon, C. Fuhs, L. Gonnord: *Analysing parallel complexity of term rewriting*, LOPSTR '22
III. Termination and Complexity

Proof Certification
Certification: Who Watches the Watchers?

- Termination and complexity analysis tools are large, e.g., AProVE has several 100,000s LOC – most likely with bugs!
Certification: Who Watches the Watchers?

- Termination and complexity analysis tools are large, e.g., AProVE has several 100,000s LOC – most likely with bugs!
- Observation in early Termination Competitions: some tools disagreed on YES / NO for termination
Certification: Who Watches the Watchers?

- Termination and complexity analysis tools are large, e.g., AProVE has several 100,000s LOC – most likely with bugs!
- Observation in early Termination Competitions: some tools disagreed on YES / NO for termination
- **Step 1**: Require human-readable proof output. But: can be large!
Certification: Who Watches the Watchers?

- Termination and complexity analysis tools are large, e.g., AProVE has several 100,000s LOC – most likely with bugs!
- Observation in early Termination Competitions: some tools disagreed on YES / NO for termination
- **Step 1**: Require human-readable proof output. But: can be large!
- **Step 2**: Machine-readable XML proof output, can be certified independently by **trustworthy** tools based on Coq and Isabelle
Certification: Who Watches the Watchers?

- Termination and complexity analysis tools are large, e.g., AProVE has several 100,000s LOC – most likely with bugs!
- Observation in early Termination Competitions: some tools disagreed on YES / NO for termination
- **Step 1**: Require human-readable proof output. But: can be large!
- **Step 2**: Machine-readable XML proof output, can be certified independently by trustworthy tools based on Coq and Isabelle
- ~ 2007/8: projects A3PAT\(^{67}\), CoLoR\(^{68}\), IsaFoR\(^{69}\) formalise term rewriting, termination, proof techniques → automatic proof checkers

---

\(^{67}\) E. Contejean, P. Courtieu, J. Forest, O. Pons, X. Urbain: Automated Certified Proofs with CiME3, RTA ’11

\(^{68}\) F. Blanqui, A. Koprowski: CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates, MSCS ’11

\(^{69}\) R. Thiemann, C. Sternagel: Certification of Termination Proofs using CeTA, TPHOLs ’09
Certification: Who Watches the Watchers?

- Termination and complexity analysis tools are large, e.g., AProVE has several 100,000s LOC – most likely with bugs!
- Observation in early Termination Competitions: some tools disagreed on YES / NO for termination
- **Step 1**: Require human-readable proof output. But: can be large!
- **Step 2**: Machine-readable XML proof output, can be certified independently by trustworthy tools based on Coq and Isabelle

~ 2007/8: projects A3PAT\(^67\), CoLoR\(^68\), IsaFoR\(^69\) formalise term rewriting, termination, proof techniques → automatic proof checkers
- performance bottleneck: computations in theorem prover

---

\(^67\) E. Contejean, P. Courtieu, J. Forest, O. Pons, X. Urbain: *Automated Certified Proofs with CiME3*, RTA ’11

\(^68\) F. Blanqui, A. Koprowski: *CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates*, MSCS ’11

\(^69\) R. Thiemann, C. Sternagel: *Certification of Termination Proofs using CeTA*, TPHOLs ’09
Certification: Who Watches the Watchers?

- Termination and complexity analysis tools are large, e.g., AProVE has several 100,000s LOC – most likely with bugs!
- Observation in early Termination Competitions: some tools disagreed on YES / NO for termination
- **Step 1**: Require human-readable proof output. But: can be large!
- **Step 2**: Machine-readable XML proof output, can be certified independently by **trustworthy** tools based on Coq and Isabelle
- ~ 2007/8: projects A3PAT\(^{67}\), CoLoR\(^{68}\), IsaFoR\(^{69}\) formalise term rewriting, termination, proof techniques → automatic proof checkers
- performance bottleneck: computations in theorem prover
- solution: extract source code (Haskell, OCaml, ...) for proof checker → CeTA tool from IsaFoR

---

\(^{67}\) E. Contejean, P. Courtieu, J. Forest, O. Pons, X. Urbain: *Automated Certified Proofs with CiME3*, RTA ’11

\(^{68}\) F. Blanqui, A. Koprowski: *CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates*, MSCS ’11

\(^{69}\) R. Thiemann, C. Sternagel: *Certification of Termination Proofs using CeTA*, TPHOLs ’09
CeTA can certify proofs for...
CeTA can certify proofs for...

- termination of TRSs (several flavours), ITSs, and LLVM programs

\[^{70}\text{M. Haslbeck, R. Thiemann: An Isabelle/HOL formalization of AProVE’s termination method for LLVM IR, CPP ’21}\]
Proof Certification with CeTA

http://cl-informatik.uibk.ac.at/isafor/

CeTA can certify proofs for...

- termination of TRSs (several flavours), ITSs, and LLVM programs
- non-termination for TRSs

---

70 M. Haslbeck, R. Thiemann: *An Isabelle/HOL formalization of AProVE’s termination method for LLVM IR*, CPP ’21
Proof Certification with CeTA

http://cl-informatik.uibk.ac.at/isafor/

CeTA can certify proofs for...

- termination of TRSs (several flavours), ITSs, and LLVM programs\textsuperscript{70}
- non-termination for TRSs
- upper bounds for complexity

\textsuperscript{70}M. Haslbeck, R. Thiemann: An Isabelle/HOL formalization of AProVE’s termination method for LLVM IR, CPP ’21
CeTA can certify proofs for...

- termination of TRSs (several flavours), ITSs, and LLVM programs\textsuperscript{70}
- non-termination for TRSs
- upper bounds for complexity
- confluence and non-confluence proofs for TRSs

\textsuperscript{70} M. Haslbeck, R. Thiemann: An Isabelle/HOL formalization of AProVE’s termination method for LLVM IR, CPP ’21
CeTA can certify proofs for...

- termination of TRSs (several flavours), ITSs, and LLVM programs\textsuperscript{70}
- non-termination for TRSs
- upper bounds for complexity
- confluence and non-confluence proofs for TRSs
- safety: invariants for ITSs\textsuperscript{71}

\textsuperscript{70} M. Haslbeck, R. Thiemann: \textit{An Isabelle/HOL formalization of AProVE’s termination method for LLVM IR}, CPP ’21

Proof Certification with CeTA

http://cl-informatik.uibk.ac.at/isafor/

CeTA can certify proofs for...
- termination of TRSs (several flavours), ITSs, and LLVM programs
- non-termination for TRSs
- upper bounds for complexity
- confluence and non-confluence proofs for TRSs
- safety: invariants for ITSs

If certification unsuccessful:
CeTA indicates which part of the proof it could not follow

---

70 M. Haslbeck, R. Thiemann: An Isabelle/HOL formalization of AProVE’s termination method for LLVM IR, CPP ’21
termCOMP with Certification (✓) (1/2)
Let's zoom in . . .

Termination of Rewriting

Progress: 100%, CPU Time: 85d 8:05:33, Node Time: 34d 3:4

TRS Standard 54200 54199

1. AProVE21
   ✓ 1. AProVE21
2. NaTT 2.3.2
3. ttt2-1.20
   ✓ 2. ttt2-1.20
4. muterm 6.0.3
   ✓ 3. NaTT 1.6.2
5. NTI_22

SRS Standard 54202 54201

1. matchbox-2022-07-22
   ✓ 1. matchbox-2022-07-22
2. MnM3.19c
3. AProVE21
   ✓ 2. AProVE21
4. ttt2-1.20
   ✓ 3. ttt2-1.20
5. NaTT 2.3.2
   ✓ 4. NaTT 1.6.2
6. muterm 6.0.3
Let’s zoom in . . .

⇒ proof certification is competitive!
Termination and complexity analysis: active fields of research
Termination and Complexity: Conclusion

- Termination and complexity analysis: active fields of research
- Push-button tools to prove (non-)termination and to infer upper and lower complexity bounds available
Termination and complexity analysis: active fields of research

- Push-button tools to prove (non-)termination and to infer upper and lower complexity bounds available

- Cross-fertilisation between techniques for different formalisms (integer transition systems, functional programs, . . .)
Termination and complexity analysis: active fields of research

Push-button tools to prove (non-)termination and to infer upper and lower complexity bounds available

Cross-fertilisation between techniques for different formalisms (integer transition systems, functional programs, ...)

Certification helps raise trust in automatically found proofs of (non-)termination and complexity bounds
Termination and complexity analysis: active fields of research

Push-button tools to prove (non-)termination and to infer upper and lower complexity bounds available

Cross-fertilisation between techniques for different formalisms (integer transition systems, functional programs, . . .)

Certification helps raise trust in automatically found proofs of (non-)termination and complexity bounds

Thanks a lot for your attention!


