Automated Reasoning for Static Program Analysis

Carsten Fuhs

Birkbeck, University of London

SAT / SMT / AR Summer School 2024

Nancy, France

29 June 2024

https://www.dcs.bbk.ac.uk/~carsten/satsmtar2024/

Quality Assurance for Software by Program Analysis

Two approaches:

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

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:

Use automatic reasoning for static analysis!

Static Analysis: the User's Perspective (1/2)

For the user (programmer): Use static analysis tools as "black boxes".

Static Analysis: the User's Perspective (1/2)

For the user (programmer): Use static analysis tools as "black boxes".

What properties of programs do we want to analyse?

Static Analysis: the User's Perspective (1/2)

For the user (programmer): Use static analysis tools as "black boxes".

What properties of programs do we want to analyse?

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

Static Analysis: the User's Perspective (1/2)

For the user (programmer): Use static analysis tools as "black boxes".

What properties of programs do we want to analyse?

- **Termination**
  $\longrightarrow$ will my program give an output for all inputs in finitely many steps?
- **(Quantitative) Resource Use** aka **Complexity**
  $\longrightarrow$ how many steps will my program need in the worst case? (runtime complexity)
  $\longrightarrow$ how large can my data become? (size complexity)

Static Analysis: the User's Perspective (1/2)

For the user (programmer): Use static analysis tools as "black boxes".

What properties of programs do we want to analyse?

- **Termination**
  $\longrightarrow$ will my program give an output for all inputs in finitely many steps?
- **(Quantitative) Resource Use** aka **Complexity**
  $\longrightarrow$ how many steps will my program need in the worst case? (runtime complexity)
  $\longrightarrow$ how large can my data become? (size complexity)
- **Equivalence**. Do **two** programs always produce the same result?
  $\longrightarrow$ correctness of refactoring
  $\longrightarrow$ translation validation for compilers

Static Analysis: the User's Perspective (1/2)

For the user (programmer): Use static analysis tools as "black boxes".

What properties of programs do we want to analyse?

- **Termination**
  $\longrightarrow$ will my program give an output for all inputs in finitely many steps?
- **(Quantitative) Resource Use** aka **Complexity**
  $\longrightarrow$ how many steps will my program need in the worst case? (runtime complexity)
  $\longrightarrow$ how large can my data become? (size complexity)
- **Equivalence**. Do **two** programs always produce the same result?
  $\longrightarrow$ correctness of refactoring
  $\longrightarrow$ translation validation for compilers
- **Confluence**. For languages with non-deterministic rules/commands:
  Does **one** 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*]

  $\longrightarrow$ does the order of applying compiler optimisation rules matter?

Static Analysis: the User's Perspective (1/2)

For the user (programmer): Use static analysis tools as "black boxes".

What properties of programs do we want to analyse?

- **Termination**
  $\longrightarrow$ will my program give an output for all inputs in finitely many steps?
- **(Quantitative) Resource Use** aka **Complexity**
  $\longrightarrow$ how many steps will my program need in the worst case? (runtime complexity)
  $\longrightarrow$ how large can my data become? (size complexity)
- **Equivalence**. Do **two** programs always produce the same result?
  $\longrightarrow$ correctness of refactoring
  $\longrightarrow$ translation validation for compilers
- **Confluence**. For languages with non-deterministic rules/commands:
  Does **one** 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*]

  $\longrightarrow$ does the order of applying compiler optimisation rules matter?

Static Analysis: the User's Perspective (1/2)

For the user (programmer): Use static analysis tools as "black boxes".

What properties of programs do we want to analyse?

- **Termination**
  $\longrightarrow$ will my program give an output for all inputs in finitely many steps?
- **(Quantitative) Resource Use** aka **Complexity**
  $\longrightarrow$ how many steps will my program need in the worst case? (runtime complexity)
  $\longrightarrow$ how large can my data become? (size complexity)
- **Equivalence**. Do **two** programs always produce the same result?
  $\longrightarrow$ correctness of refactoring
  $\longrightarrow$ translation validation for compilers
- **Confluence**. For languages with non-deterministic rules/commands: Does **one** 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*]

  $\longrightarrow$ does the order of applying compiler optimisation rules matter?

Ask me in the coffee break!

Static analysis: the user's perspective (2/2)

Safety properties.

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

Static analysis: the user's perspective (2/2)

Safety properties.

- **Partial Correctness**

  → will my program always produce the right result?
- **Assertions by the programmer.**

```
assert x > 0;
```

  → will this always be true?

Static analysis: the user's perspective (2/2)

Safety properties.

- **Partial Correctness**
  $\longrightarrow$ will my program always produce the right result?
- **Assertions by the programmer**.

```
assert x > 0;
```

  $\longrightarrow$ will this always be true?
- **Memory Safety**
  $\longrightarrow$ are my memory accesses always legal?

```
int* x = NULL;  *x = 42;
```

Static analysis: the user's perspective (2/2)

Safety properties.

- **Partial Correctness**

  $\longrightarrow$ will my program always produce the right result?
- **Assertions by the programmer**.

  ```
  assert x > 0;
  ```

  $\longrightarrow$ will this always be true?
- **Memory Safety**

  $\longrightarrow$ are my memory accesses always legal?

  ```
  int* x = NULL;  *x = 42;
  ```

  $\longrightarrow$ undefined behaviour!

Static analysis: the user's perspective (2/2)

Safety properties.

- **Partial Correctness**

  $\longrightarrow$ will my program always produce the right result?

- **Assertions by the programmer.**

  ```
  assert x > 0;
  ```

  $\longrightarrow$ will this always be true?

- **Memory Safety**

  $\longrightarrow$ are my memory accesses always legal?

  ```
  int* x = NULL;  *x = 42;
  ```

  $\longrightarrow$ undefined behaviour!

  $\longrightarrow$ replacing all files on the computer with cat GIFs

Static analysis: the user's perspective (2/2)

Safety properties.

- **Partial Correctness**

  → will my program always produce the right result?

- **Assertions by the programmer**.

  ```
  assert x > 0;
  ```

  → will this always be true?

- **Memory Safety**

  → are my memory accesses always legal?

  ```
  int* x = NULL;  *x = 42;
  ```

  → undefined behaviour!

  → replacing all files on the computer with cat GIFs

  → information leaks (Heartbleed OpenSSL attack)

Static analysis: the user's perspective (2/2)

Safety properties.

- **Partial Correctness**
  $\longrightarrow$ will my program always produce the right result?
- **Assertions by the programmer**.

```
assert x > 0;
```

  $\longrightarrow$ will this always be true?
- **Memory Safety**
  $\longrightarrow$ are my memory accesses always legal?

```
int* x = NULL;  *x = 42;
```

  $\longrightarrow$ undefined behaviour!
  $\longrightarrow$ replacing all files on the computer with cat GIFs
  $\longrightarrow$ information leaks (Heartbleed OpenSSL attack)
  $\longrightarrow$ **non-termination**

Static analysis: the user's perspective (2/2)

Safety properties.

- **Partial Correctness**
  → will my program always produce the right result?
- **Assertions by the programmer.**
  ```
  assert x > 0;
  ```
  → will this always be true?
- **Memory Safety**
  → are my memory accesses always legal?
  ```
  int* x = NULL;  *x = 42;
  ```
  → undefined behaviour!
  → replacing all files on the computer with cat GIFs
  → information leaks (Heartbleed OpenSSL attack)
  → **non-termination**

Note: All these properties are **undecidable**!
⇒ use automatable sufficient criteria in practice

APROVE ... since 2001

- Program analysis tool developed in Aachen, London, Innsbruck, ...

AProVE . . . since 2001

- Program analysis tool developed in Aachen, London, Innsbruck, . . .
- Fully automated, hundreds of techniques for **termination**, **time complexity bounds**, . . .

Termination

Complexity

Non-Termination

APROVE . . . since 2001

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

Termination

Complexity

Non-Termination

APROVE . . . since 2001

- 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 $\longrightarrow$ construct **proof tree**

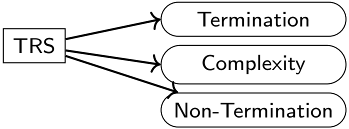
Termination

Complexity

Non-Termination

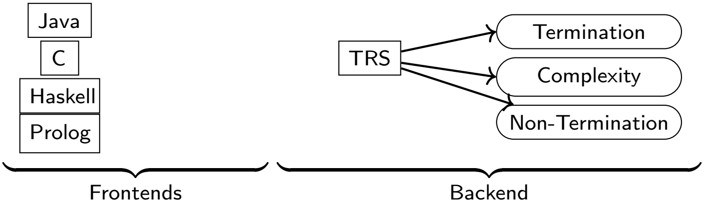
APROVE . . . since 2001

- 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 $\longrightarrow$ construct **proof tree**
- Founding tool of Termination Competition, since 2004
- Initially: analyse **termination** of **term rewrite systems (TRSs)**, later also complexity bounds



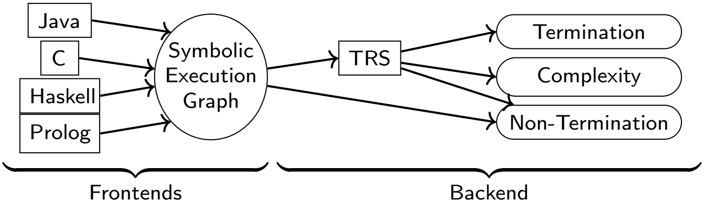
AProVE . . . since 2001

- 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 $\longrightarrow$ 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)



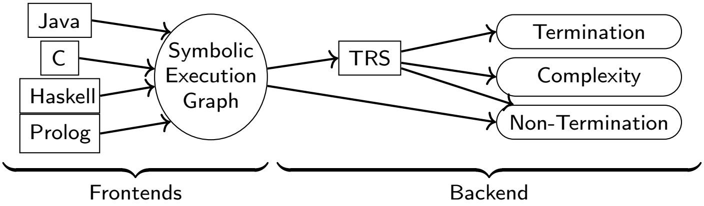
AProVE . . . since 2001

- 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 $\longrightarrow$ 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



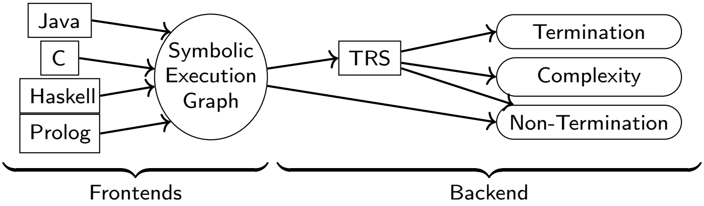
AProVE . . . since 2001

- 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 $\longrightarrow$ 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



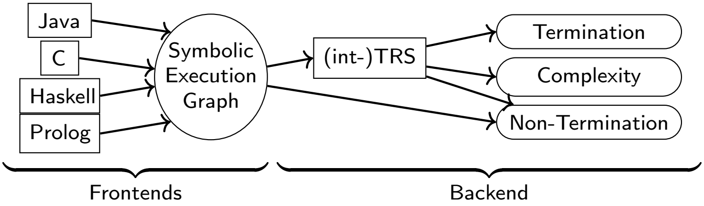
AProVE ... since 2001

- 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 $\longrightarrow$ 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



AProVE ... since 2001

- 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 $\longrightarrow$ 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

$\Rightarrow$ Result carries over to original program

I. Termination Analysis

Why Analyse Termination?

Why Analyse Termination?

1. **Program**: produces result (no spec needed!)

Why Analyse Termination?

1. **Program**: produces result (no spec needed!)
2. **Input handler**: system reacts

Why Analyse Termination?

1. **Program**: produces result (no spec needed!)
2. **Input handler**: system reacts
3. **Mathematical proof**: the induction is valid

Why Analyse Termination?

1. **Program**: produces result (no spec needed!)
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 (no spec needed!)
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 (no spec needed!)
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

- http://www.exploringbinary.com/php-hangs-on-numeric-value-2-2250738585072011e-308/
- http://www.exploringbinary.com/java-hangs-when-converting-2-2250738585072012e-308/

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.
Here again he should be assisted by the programmer giving a further definite
assertion to be verified. 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."

Termination Analysis, Classically

Turing 1949

Finally the checker has to verify that the process comes to an end.
Here again he should be assisted by the programmer giving a further definite
assertion to be verified. 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. Here again he should be assisted by the programmer giving a further definite assertion to be verified. 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")
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. Here again he should be assisted by the programmer giving a further definite assertion to be verified. 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")
2. Prove f to have a **lower bound** ("vanish when the machine stops")
3. Prove that f **decreases** over time

Termination Analysis, Classically

Turing 1949

Finally the checker has to verify that the process comes to an end.
Here again he should be assisted by the programmer giving a further definite
assertion to be verified. 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")
2. Prove f to have a **lower bound** ("vanish when the machine stops")
3. Prove that f **decreases** over time

Example (Does this program terminate for all $x \in \mathbb{Z}$?)

while $x > 0$:

$x = x - 1$

Termination Analysis, in the Era of Automated Reasoning

Question: Does program P terminate?

Termination Analysis, in the Era of Automated Reasoning

Question: Does program P terminate?

Approach: Encode termination proof **template** to logical constraint φ,
ask SMT solver

Termination Analysis, in the Era of Automated Reasoning

Question: Does program P terminate?

Approach: Encode termination proof **template** to logical constraint φ, ask SMT solver

$\rightarrow$ **SMT** = **SAT**isfiability **M**odulo **T**heories, solve constraints like

$$b > 0 \quad \wedge \quad (4\,a\,b - 7\,b^2 > 1 \quad \vee \quad 3\,a + c \geq b^3)$$

Question: Does program P terminate?

Approach: Encode termination proof **template** to logical constraint φ,
ask SMT solver

$\longrightarrow$ **SMT** = **SAT**isfiability **M**odulo **T**heories, solve constraints like

$$b > 0 \quad \wedge \quad (4\,a\,b - 7\,b^2 > 1 \quad \vee \quad 3\,a + c \geq b^3)$$

Answer:

Termination Analysis, in the Era of Automated Reasoning

Question: Does program P terminate?

Approach: Encode termination proof **template** to logical constraint φ, ask SMT solver

$\longrightarrow$ **SMT** = **SAT**isfiability **M**odulo **T**heories, solve constraints like

$$b > 0 \quad \wedge \quad (4\,a\,b - 7\,b^2 > 1 \quad \vee \quad 3\,a + c \geq b^3)$$

Answer:

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

Termination Analysis, in the Era of Automated Reasoning

Question: Does program P terminate?

Approach: Encode termination proof **template** to logical constraint φ, ask SMT solver

$\longrightarrow$ **SMT** = **SAT**isfiability **M**odulo **T**heories, solve constraints like

$$b > 0 \quad \wedge \quad (4\,a\,b - 7\,b^2 > 1 \quad \vee \quad 3\,a + c \geq b^3)$$

Answer:

1. φ satisfiable, model M (e.g., $a = 3, b = 1, c = 1$):
   $\Rightarrow$ P terminating, M fills in the gaps in the termination proof
2. φ unsatisfiable:
   $\Rightarrow$ termination status of P unknown
   $\Rightarrow$ try a different template (proof technique)

Termination Analysis, in the Era of Automated Reasoning

Question: Does program P terminate?

Approach: Encode termination proof **template** to logical constraint φ,
ask SMT solver

$\longrightarrow$ **SMT** = **SAT**isfiability **M**odulo **T**heories, solve constraints like

$$b > 0 \quad \wedge \quad (4\,a\,b - 7\,b^2 > 1 \quad \vee \quad 3\,a + c \geq b^3)$$

Answer:

1. φ satisfiable, model M (e.g., $a = 3, b = 1, c = 1$):
   $\Rightarrow$ P terminating, M fills in the gaps in the termination proof
2. φ unsatisfiable:
   $\Rightarrow$ termination status of P unknown
   $\Rightarrow$ try a different template (proof technique)

In practice:

- Encode only one proof **step** at a time
  $\longrightarrow$ try to prove only **part** of the program terminating
- **Repeat** until the whole program is proved terminating

Termination Proving in the Back-End and in the Front-End

Back-End:

1. Term Rewrite Systems (TRSs)
2. Imperative Programs (as Integer Transition Systems, ITSs)
3. Both together! Logically Constrained Term Rewrite Systems

Termination Proving in the Back-End and in the Front-End

Back-End:

1. Term Rewrite Systems (TRSs)
2. Imperative Programs (as Integer Transition Systems, ITSs)
3. Both together! Logically Constrained Term Rewrite Systems

Front-End: processing practical programming languages
Example: Java

I.1 Termination Analysis of Term Rewrite Systems

What's Term Rewriting?

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 $\longrightarrow$ non-determinism!
- no fixed order of rules to apply (Haskell: top to bottom) $\longrightarrow$ non-determinism!
- untyped (unless you really want types)
- no pre-defined data structures (integers, arrays, ...)

Show Me an Example!

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

$$0,\ \mathsf{s}(0),\ \mathsf{s}(\mathsf{s}(0)),\ \ldots$$

Show Me an Example!

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

$$0,\ \mathsf{s}(0),\ \mathsf{s}(\mathsf{s}(0)),\ \ldots$$

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

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, 0) & \to & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}(x, y) \\ \mathsf{quot}(0, \mathsf{s}(y)) & \to & 0 \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

Show Me an Example!

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

$$0, \; \mathsf{s}(0), \; \mathsf{s}(\mathsf{s}(0)), \; \ldots$$

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

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathsf{minus}(x, 0) & \to & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}(x, y) \\ \mathsf{quot}(0, \mathsf{s}(y)) & \to & 0 \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{array} \right.$$

Calculation:

$$\mathsf{minus}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(0)) \quad \to_{\mathcal{R}} \quad \mathsf{minus}(\mathsf{s}(0), 0) \quad \to_{\mathcal{R}} \quad \mathsf{s}(0)$$

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
    [van Raamsdonk, *ICLP '97*; Schneider-Kamp et al, *TOCL '09*; Giesl et al, *PPDP '12*]

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*]

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*]

Example (Division)

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

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

$$\mathsf{minus}(\mathsf{s}(\mathsf{s}(\mathsf{0})), \mathsf{s}(\mathsf{0})) \rightarrow_{\mathcal{R}} \mathsf{minus}(\mathsf{s}(\mathsf{0}), \mathsf{0}) \rightarrow_{\mathcal{R}} \mathsf{s}(\mathsf{0})$$

Example (Division)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \to & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \to & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

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

$$\mathsf{minus}(\mathsf{s}(\mathsf{s}(\mathsf{0})), \mathsf{s}(\mathsf{0})) \to_{\mathcal{R}} \mathsf{minus}(\mathsf{s}(\mathsf{0}), \mathsf{0}) \to_{\mathcal{R}} \mathsf{s}(\mathsf{0})$$

Termination: No infinite evaluation sequences $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} t_3 \to_{\mathcal{R}} \ldots$

Example (Division)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \to & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \to & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

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

$$\mathsf{minus}(\mathsf{s}(\mathsf{s}(\mathsf{0})), \mathsf{s}(\mathsf{0})) \to_{\mathcal{R}} \mathsf{minus}(\mathsf{s}(\mathsf{0}), \mathsf{0}) \to_{\mathcal{R}} \mathsf{s}(\mathsf{0})$$

Termination: No infinite evaluation sequences $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} t_3 \to_{\mathcal{R}} \dots$
Show termination using Dependency Pairs

Example (Division)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \to & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \to & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

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

Example (Division)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \to & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \to & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

$$\mathcal{DP} = \begin{cases} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{cases}$$

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

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

Example (Division)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \to & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \to & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

$$\mathcal{DP} = \begin{cases} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{cases}$$

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

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

Example (Division)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \to & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \to & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

$$\mathcal{DP} = \begin{cases} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \to & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{cases}$$

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

- For TRS $\mathcal{R}$ build dependency pairs $\mathcal{DP}$ (∼ function calls)
- Show: No ∞ 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)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \succsim & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \succsim & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

$$\mathcal{DP} = \begin{cases} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{cases}$$

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

- For TRS $\mathcal{R}$ build dependency pairs $\mathcal{DP}$ ($\sim$ function calls)
- Show: No ∞ 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 $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$

Example (Division)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \succsim & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \succsim & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

$$\mathcal{DP} = \begin{cases} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \overset{\succ}{(\sim)} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \overset{\succ}{(\sim)} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \overset{\succ}{(\sim)} & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{cases}$$

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

- For TRS $\mathcal{R}$ build dependency pairs $\mathcal{DP}$ (∼ function calls)
- Show: No ∞ 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 $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
  - delete $s \longrightarrow t$ with $s \succ t$ from $\mathcal{DP}$

Example (Division)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \succsim & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \succsim & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

$$\mathcal{DP} = \begin{cases} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \overset{\succ}{(\sim)} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \overset{\succ}{(\sim)} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \overset{\succ}{(\sim)} & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{cases}$$

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

- For TRS $\mathcal{R}$ build dependency pairs $\mathcal{DP}$ (∼ function calls)
- Show: No ∞ 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 $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
  - delete $s \to t$ with $s \succ t$ from $\mathcal{DP}$
- Find $\succ$ automatically and efficiently

Polynomial Interpretations

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

Example

$$\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \;\; \gtrsim \;\; \mathsf{minus}(x, y)$$

Polynomial Interpretations

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

Example

$$\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \quad \succsim \quad \mathsf{minus}(x, y)$$

Use $[\,\cdot\,]$ with

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

Polynomial Interpretations

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

Example

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

Use $[\,\cdot\,]$ with

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

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)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \succsim & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \succsim & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

$$\mathcal{DP} = \begin{cases} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \underset{(\sim)}{\succ} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \underset{(\sim)}{\succ} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \underset{(\sim)}{\succ} & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{cases}$$

Example (Constraints for Division)

$$\mathcal{R} = \begin{cases} \mathsf{minus}(x, \mathsf{0}) & \succsim & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \succsim & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{cases}$$

$$\mathcal{DP} = \begin{cases} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \succ & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \succ & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \succ & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{cases}$$

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

$$\begin{aligned} [\mathsf{quot}^\sharp](x_1, x_2) &= x_1 & \qquad [\mathsf{quot}](x_1, x_2) &= x_1 + x_2 \\ [\mathsf{minus}^\sharp](x_1, x_2) &= x_1 & \qquad [\mathsf{minus}](x_1, x_2) &= x_1 \\ [\mathsf{0}] &= 0 & \qquad [\mathsf{s}](x_1) &= x_1 + 1 \end{aligned}$$

$\curvearrowright$ order solves all constraints

Example (Constraints for Division)

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathsf{minus}(x, \mathsf{0}) & \succsim & x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{minus}(x, y) \\ \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) & \succsim & \mathsf{0} \\ \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) & \succsim & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \end{array} \right.$$

$$\mathcal{DP} = \left\{ \right.$$

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

$$\begin{array}{rclcrcl} [\mathsf{quot}^\sharp](x_1, x_2) & = & x_1 & & [\mathsf{quot}](x_1, x_2) & = & x_1 + x_2 \\ [\mathsf{minus}^\sharp](x_1, x_2) & = & x_1 & & [\mathsf{minus}](x_1, x_2) & = & x_1 \\ [\mathsf{0}] & = & 0 & & [\mathsf{s}](x_1) & = & x_1 + 1 \end{array}$$

$\curvearrowright$ order solves all constraints

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

$\curvearrowright$ termination of division algorithm proved $\square$

Remark

Polynomial interpretations play several roles for program analysis:

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

$$\begin{aligned} [\mathsf{quot}^\sharp](x_1, x_2) &= x_1 & [\mathsf{quot}](x_1, x_2) &= x_1 + x_2 \\ [\mathsf{minus}^\sharp](x_1, x_2) &= x_1 & [\mathsf{minus}](x_1, x_2) &= x_1 \\ [\mathsf{0}] &= 0 & [\mathsf{s}](x_1) &= x_1 + 1 \end{aligned}$$

$\curvearrowright$ order solves all constraints

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

$\curvearrowright$ termination of division algorithm proved $\square$

Remark

Polynomial interpretations play several roles for program analysis:

- Ranking function: $[\mathsf{quot}^\sharp]$ and $[\mathsf{minus}^\sharp]$

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

$$
\begin{aligned}
[\mathsf{quot}^\sharp](x_1, x_2) &= x_1 & \qquad [\mathsf{quot}](x_1, x_2) &= x_1 + x_2 \\
[\mathsf{minus}^\sharp](x_1, x_2) &= x_1 & \qquad [\mathsf{minus}](x_1, x_2) &= x_1 \\
[0] &= 0 & \qquad [\mathsf{s}](x_1) &= x_1 + 1
\end{aligned}
$$

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

Remark

Polynomial interpretations play several roles for program analysis:

- Ranking function: $[\mathsf{quot}^\sharp]$ and $[\mathsf{minus}^\sharp]$
- Summary: $[\mathsf{quot}]$ and $[\mathsf{minus}]$

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

$$
\begin{aligned}
[\mathsf{quot}^\sharp](x_1, x_2) &= x_1 & \qquad [\mathsf{quot}](x_1, x_2) &= x_1 + x_2 \\
[\mathsf{minus}^\sharp](x_1, x_2) &= x_1 & \qquad [\mathsf{minus}](x_1, x_2) &= x_1 \\
[\mathsf{0}] &= 0 & \qquad [\mathsf{s}](x_1) &= x_1 + 1
\end{aligned}
$$

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

Remark

Polynomial interpretations play several roles for program analysis:

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

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

$$
\begin{aligned}
[\mathsf{quot}^\sharp](x_1, x_2) &= x_1 & \qquad [\mathsf{quot}](x_1, x_2) &= x_1 + x_2 \\
[\mathsf{minus}^\sharp](x_1, x_2) &= x_1 & \qquad [\mathsf{minus}](x_1, x_2) &= x_1 \\
[\mathsf{0}] &= 0 & \qquad [\mathsf{s}](x_1) &= x_1 + 1
\end{aligned}
$$

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

Automation

Task: Solve $\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \succsim \mathsf{minus}(x, y)$

Automation

Task: Solve $\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \succsim \mathsf{minus}(x, y)$

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

$$[\mathsf{minus}](x, y) = a_{\mathsf{m}} + b_{\mathsf{m}}\, x + c_{\mathsf{m}}\, y, \quad [\mathsf{s}](x) = a_{\mathsf{s}} + b_{\mathsf{s}}\, x$$

Automation

Task: Solve $\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \gtrsim \mathsf{minus}(x, y)$

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

$$[\mathsf{minus}](x, y) = a_{\mathsf{m}} + b_{\mathsf{m}}\, x + c_{\mathsf{m}}\, y, \quad [\mathsf{s}](x) = a_{\mathsf{s}} + b_{\mathsf{s}}\, x$$

2. From term constraint to polynomial constraint:

$$s \gtrsim t \;\curvearrowright\; [s] \geq [t]$$

Here: $\forall x, y.\; (a_{\mathsf{s}}\, b_{\mathsf{m}} + a_{\mathsf{s}}\, c_{\mathsf{m}}) + (b_{\mathsf{s}}\, b_{\mathsf{m}} - b_{\mathsf{m}})\, x + (b_{\mathsf{s}}\, c_{\mathsf{m}} - c_{\mathsf{m}})\, y \geq 0$

Automation

Task: Solve $\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \succsim \mathsf{minus}(x, y)$

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

$$[\mathsf{minus}](x, y) = a_{\mathsf{m}} + b_{\mathsf{m}}\, x + c_{\mathsf{m}}\, y, \quad [\mathsf{s}](x) = a_{\mathsf{s}} + b_{\mathsf{s}}\, x$$

2. From term constraint to polynomial constraint:

$$s \succsim t \;\curvearrowright\; [s] \geq [t]$$

Here: $\forall x, y.\; (a_{\mathsf{s}}\, b_{\mathsf{m}} + a_{\mathsf{s}}\, c_{\mathsf{m}}) + (b_{\mathsf{s}}\, b_{\mathsf{m}} - b_{\mathsf{m}})\, x + (b_{\mathsf{s}}\, c_{\mathsf{m}} - c_{\mathsf{m}})\, y \geq 0$

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

Here: $a_{\mathsf{s}}\, b_{\mathsf{m}} + a_{\mathsf{s}}\, c_{\mathsf{m}} \geq 0 \wedge b_{\mathsf{s}}\, b_{\mathsf{m}} - b_{\mathsf{m}} \geq 0 \wedge b_{\mathsf{s}}\, c_{\mathsf{m}} - c_{\mathsf{m}} \geq 0$

Automation

Task: Solve $\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \succsim \mathsf{minus}(x, y)$

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

$$[\mathsf{minus}](x, y) = a_{\mathsf{m}} + b_{\mathsf{m}}\, x + c_{\mathsf{m}}\, y, \quad [\mathsf{s}](x) = a_{\mathsf{s}} + b_{\mathsf{s}}\, x$$

2. From term constraint to polynomial constraint:

$$s \succsim t \;\curvearrowright\; [s] \geq [t]$$

Here: $\forall x, y.\ (a_{\mathsf{s}}\, b_{\mathsf{m}} + a_{\mathsf{s}}\, c_{\mathsf{m}}) + (b_{\mathsf{s}}\, b_{\mathsf{m}} - b_{\mathsf{m}})\, x + (b_{\mathsf{s}}\, c_{\mathsf{m}} - c_{\mathsf{m}})\, y \geq 0$

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

Here: $a_{\mathsf{s}}\, b_{\mathsf{m}} + a_{\mathsf{s}}\, c_{\mathsf{m}} \geq 0 \;\wedge\; b_{\mathsf{s}}\, b_{\mathsf{m}} - b_{\mathsf{m}} \geq 0 \;\wedge\; b_{\mathsf{s}}\, c_{\mathsf{m}} - c_{\mathsf{m}} \geq 0$

Automation

Task: Solve $\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \succsim \mathsf{minus}(x, y)$

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

$$[\mathsf{minus}](x, y) = a_\mathsf{m} + b_\mathsf{m}\, x + c_\mathsf{m}\, y, \quad [\mathsf{s}](x) = a_\mathsf{s} + b_\mathsf{s}\, x$$

2. From term constraint to polynomial constraint:

$$s \succsim t \;\curvearrowright\; [s] \geq [t]$$

Here: $\forall x, y.\; (a_\mathsf{s}\, b_\mathsf{m} + a_\mathsf{s}\, c_\mathsf{m}) + (b_\mathsf{s}\, b_\mathsf{m} - b_\mathsf{m})\, x + (b_\mathsf{s}\, c_\mathsf{m} - c_\mathsf{m})\, y \geq 0$

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

Here: $a_\mathsf{s}\, b_\mathsf{m} + a_\mathsf{s}\, c_\mathsf{m} \geq 0 \;\wedge\; b_\mathsf{s}\, b_\mathsf{m} - b_\mathsf{m} \geq 0 \;\wedge\; b_\mathsf{s}\, c_\mathsf{m} - c_\mathsf{m} \geq 0$

Automation

Task: Solve $\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \succsim \mathsf{minus}(x, y)$

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

$$[\mathsf{minus}](x, y) = a_{\mathsf{m}} + b_{\mathsf{m}}\, x + c_{\mathsf{m}}\, y, \quad [\mathsf{s}](x) = a_{\mathsf{s}} + b_{\mathsf{s}}\, x$$

2. From term constraint to polynomial constraint:

$$s \succsim t \ \curvearrowright\ [s] \geq [t]$$

Here: $\forall x, y.\ (a_{\mathsf{s}}\, b_{\mathsf{m}} + a_{\mathsf{s}}\, c_{\mathsf{m}}) + (b_{\mathsf{s}}\, b_{\mathsf{m}} - b_{\mathsf{m}})\, x + (b_{\mathsf{s}}\, c_{\mathsf{m}} - c_{\mathsf{m}})\, y \geq 0$

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

Here: $a_{\mathsf{s}}\, b_{\mathsf{m}} + a_{\mathsf{s}}\, c_{\mathsf{m}} \geq 0 \wedge b_{\mathsf{s}}\, b_{\mathsf{m}} - b_{\mathsf{m}} \geq 0 \wedge b_{\mathsf{s}}\, c_{\mathsf{m}} - c_{\mathsf{m}} \geq 0$

Non-linear constraints, even for linear interpretations

Automation

Task: Solve $\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \succsim \mathsf{minus}(x, y)$

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

$$[\mathsf{minus}](x, y) = a_{\mathsf{m}} + b_{\mathsf{m}}\, x + c_{\mathsf{m}}\, y, \quad [\mathsf{s}](x) = a_{\mathsf{s}} + b_{\mathsf{s}}\, x$$

2. From term constraint to polynomial constraint:

$$s \succsim t \;\curvearrowright\; [s] \geq [t]$$

   Here: $\forall x, y.\; (a_{\mathsf{s}}\, b_{\mathsf{m}} + a_{\mathsf{s}}\, c_{\mathsf{m}}) + (b_{\mathsf{s}}\, b_{\mathsf{m}} - b_{\mathsf{m}})\, x + (b_{\mathsf{s}}\, c_{\mathsf{m}} - c_{\mathsf{m}})\, y \geq 0$

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

   Here: $a_{\mathsf{s}}\, b_{\mathsf{m}} + a_{\mathsf{s}}\, c_{\mathsf{m}} \geq 0 \wedge b_{\mathsf{s}}\, b_{\mathsf{m}} - b_{\mathsf{m}} \geq 0 \wedge b_{\mathsf{s}}\, c_{\mathsf{m}} - c_{\mathsf{m}} \geq 0$

   Non-linear constraints, even for linear interpretations

Task: Show satisfiability of non-linear constraints over $\mathbb{N}$ ($\longrightarrow$ SMT solver!)
$\curvearrowright$ Prove termination of given term rewrite system

Non-Linear Constraint Solving

Satisfiability of non-linear SMT formulas over $\mathbb{N}$ undecidable (Hilbert's 10th problem)

- Restrict **unknowns** to finite domain $\{0, \ldots, n\}$
- Problem NP-complete

Non-Linear Constraint Solving

Satisfiability of non-linear SMT formulas over $\mathbb{N}$ undecidable (Hilbert's 10th problem)

- Restrict **unknowns** to finite domain $\{0, \ldots, n\}$
- Problem NP-complete

Approach [Fuhs et al, *SAT '07*]

- Encode non-linear SMT formula to pure SAT
  $\longrightarrow$ bit-blasting for QF_NIA
- Use SAT solver to get solution
- Eager Approach to SMT, but any SMT solver will do!
- Observation: if a model over $\mathbb{N}$ exists, usually small n suffices (e.g., $n = 3$)

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: [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: $[\mathsf{pred}](x_1) = \max(x_1 - 1, 0)$
  - 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
  - SMT-based automation [Fuhs et al, *AISC '08*; Zankl, Middeldorp, *LPAR '10*; Borralleras et al, *JAR '12*]

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: $[\mathsf{pred}](x_1) = \max(x_1 - 1, 0)$
  - 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
  - SMT-based automation [Fuhs et al, *AISC '08*; Zankl, Middeldorp, *LPAR '10*; Borralleras et al, *JAR '12*]

- 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 [Kop, Vale, *FSCD '21*; 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: $[\mathsf{pred}](x_1) = \max(x_1 - 1, 0)$
  - 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
  - SMT-based automation [Fuhs et al, *AISC '08*; Zankl, Middeldorp, *LPAR '10*; Borralleras et al, *JAR '12*]

- 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 [Kop, Vale, *FSCD '21*; Yamada, *JAR '22*]

- ...

SAT and SMT Solving for Path Orders

Path orders: based on precedences on function symbols

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

SAT and SMT Solving for Path Orders

Path orders: based on precedences on function symbols

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

- Lexicographic Path Order [Kamin, Lévy, *Unpublished Manuscript '80*] and Recursive Path Order [Dershowitz, Manna, *CACM '79*; Dershowitz, *TCS '82*]
  $\longrightarrow$ SAT encoding [Codish et al, *JAR '11*]

SAT and SMT Solving for Path Orders

Path orders: based on precedences on function symbols

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

- Lexicographic Path Order [Kamin, Lévy, *Unpublished Manuscript '80*] and Recursive Path Order [Dershowitz, Manna, *CACM '79*; Dershowitz, *TCS '82*]
  $\longrightarrow$ SAT encoding [Codish et al, *JAR '11*]

- Weighted Path Order [Yamada, Kusakari, Sakabe, *SCP '15*]
  $\longrightarrow$ SMT encoding

Automation of the Order Search (1/2)

Dependency Pair Framework (simplified):

while $\mathcal{DP} \neq \emptyset$ **:**

- find well-founded order $\succ$ with $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
- delete $s \longrightarrow t$ with $s \succ t$ from $\mathcal{DP}$

Automation of the Order Search (1/2)

Dependency Pair Framework (simplified):

while $\mathcal{DP} \neq \emptyset$ **:**

- **find well-founded order** $\succ$ **with** $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
- delete $s \to t$ with $s \succ t$ from $\mathcal{DP}$

Automation of the Order Search (1/2)

Dependency Pair Framework (simplified):

while $\mathcal{DP} \neq \emptyset$ **:**

- **find well-founded order** $\succ$ **with** $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
- delete $s \longrightarrow t$ with $s \succ t$ from $\mathcal{DP}$

Implementation

- Launch **several concurrent instances** of the order search.

Automation of the Order Search (1/2)

Dependency Pair Framework (simplified):

while $\mathcal{DP} \neq \emptyset$ **:**

- **find well-founded order** $\succ$ **with** $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
- delete $s \to t$ with $s \succ t$ from $\mathcal{DP}$

Implementation

- Launch **several concurrent instances** of the order search.
- Each one uses different parameters (e.g., type of order, degree, max. coefficient, ...).

Automation of the Order Search (1/2)

Dependency Pair Framework (simplified):

while $\mathcal{DP} \neq \emptyset$ **:**

- **find well-founded order** $\succ$ **with** $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
- delete $s \to t$ with $s \succ t$ from $\mathcal{DP}$

Implementation

- Launch **several concurrent instances** of the order search.
- Each one uses different parameters (e.g., type of order, degree, max. coefficient, ...).
- SAT/SMT solver launched as external process on file/stdin.

Automation of the Order Search (1/2)

Dependency Pair Framework (simplified):

while $\mathcal{DP} \neq \emptyset$ **:**

- **find well-founded order** $\succ$ **with** $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
- delete $s \longrightarrow t$ with $s \succ t$ from $\mathcal{DP}$

Implementation

- Launch **several concurrent instances** of the order search.
- Each one uses different parameters (e.g., type of order, degree, max. coefficient, . . .).
- SAT/SMT solver launched as external process on file/stdin.
- First **SATISFIABLE** answer wins, kill all other instances.

Automation of the Order Search (1/2)

Dependency Pair Framework (simplified):

while $\mathcal{DP} \neq \emptyset$ **:**

- **find well-founded order** $\succ$ **with** $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
- delete $s \to t$ with $s \succ t$ from $\mathcal{DP}$

Implementation

- Launch **several concurrent instances** of the order search.
- Each one uses different parameters (e.g., type of order, degree, max. coefficient, ...).
- SAT/SMT solver launched as external process on file/stdin.
- First **SATISFIABLE** answer wins, kill all other instances.
- If internal timeout elapses (or everyone says **UNSATISFIABLE**):
  $\longrightarrow$ kill all search instances; retry with larger search space.

Automation of the Order Search (1/2)

Dependency Pair Framework (simplified):

while $\mathcal{DP} \neq \emptyset$ **:**

- **find well-founded order** $\succ$ **with** $\mathcal{DP} \cup \mathcal{R} \subseteq \succsim$
- delete $s \to t$ with $s \succ t$ from $\mathcal{DP}$

Implementation

- Launch **several concurrent instances** of the order search.
- Each one uses different parameters (e.g., type of order, degree, max. coefficient, . . .).
- SAT/SMT solver launched as external process on file/stdin.
- First **SATISFIABLE** answer wins, kill all other instances.
- If internal timeout elapses (or everyone says **UNSATISFIABLE**):
  $\longrightarrow$ kill all search instances; retry with larger search space.
- In addition: try non-SAT/SMT-based techniques
  $\longrightarrow$ decompose problem into Strongly Connected Components, prove non-termination, . . .

Automation of the Order Search (2/2)

Requirements on SAT/SMT solver:

- return model **quickly** (at most 5–10 seconds)
- performance for unsatisfiable instances not really important

Automation of the Order Search (2/2)

Requirements on SAT/SMT solver:

- return model **quickly** (at most 5–10 seconds)
- performance for unsatisfiable instances not really important

Current SAT solver of choice in AProVE: **MiniSat 2.2** [Eén, Sörensson, *SAT '03*]
(version from around 2008; finds models quickly)

Survey among tool authors (Aug/Sep 2022):
https://lists.rwth-aachen.de/hyperkitty/list/termtools@lists.rwth-aachen.de/thread/FNDNU5Y7TGXYXX34YWKFO2ICSRT6M3ME/

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*; . . .]

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*], ...

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 [Kop, *PhD thesis '12*]

$$\mathsf{map}(F, \mathsf{Cons}(x, xs)) \to \mathsf{Cons}(F(x), \mathsf{map}(F, xs))$$

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 [Kop, *PhD thesis '12*]

$$\mathsf{map}(F, \mathsf{Cons}(x, xs)) \to \mathsf{Cons}(F(x), \mathsf{map}(F, xs))$$

- **Probabilistic** term rewriting: (Positive/Strong) Almost Sure Termination
  [Avanzini, Dal Lago, Yamada, *SCP '20*; Kassing, Giesl, *CADE '23*]

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 [Kop, *PhD thesis '12*]

$$\mathsf{map}(F, \mathsf{Cons}(x, xs)) \to \mathsf{Cons}(F(x), \mathsf{map}(F, xs))$$

- **Probabilistic** term rewriting: (Positive/Strong) Almost Sure Termination
  [Avanzini, Dal Lago, Yamada, *SCP '20*; Kassing, Giesl, *CADE '23*]
- **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 $\mathcal{O}(n^3)$"

SMT Solvers *from* Termination Analysis

Annual SMT-COMP, division QF_NIA (Quantifier-Free Non-linear Integer Arithmetic)

Year	Winner
2009	Barcelogic-QF_NIA
2010	MiniSmt
2011	AProVE
2012	*no QF_NIA*
2013	*no SMT-COMP*
2014	AProVE
2015	AProVE
2016	Yices
...	...

SMT Solvers *from* Termination Analysis

Annual SMT-COMP, division QF_NIA (Quantifier-Free Non-linear Integer Arithmetic)

Year	Winner
2009	Barcelogic-QF_NIA
2010	MiniSmt (spin-off of $\mathsf{T_TT_2}$)
2011	AProVE
2012	*no QF_NIA*
2013	*no SMT-COMP*
2014	AProVE
2015	AProVE
2016	Yices
...	...

⇒ Termination provers can also be successful SMT solvers!

SMT Solvers *from* Termination Analysis

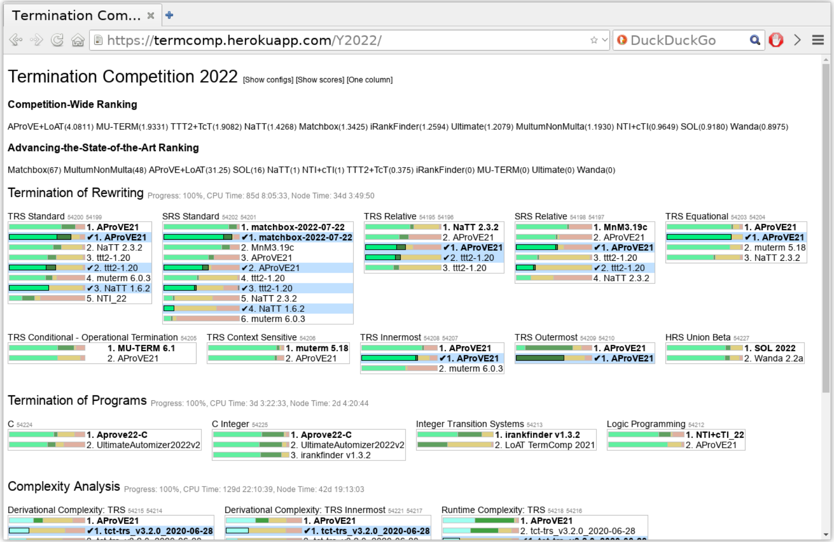
Annual SMT-COMP, division QF_NIA (Quantifier-Free Non-linear Integer Arithmetic)

Year	Winner
2009	Barcelogic-QF_NIA
2010	MiniSmt (spin-off of $\mathsf{T_TT_2}$)
2011	AProVE
2012	*no QF_NIA*
2013	*no SMT-COMP*
2014	AProVE
2015	AProVE
2016	Yices
...	...

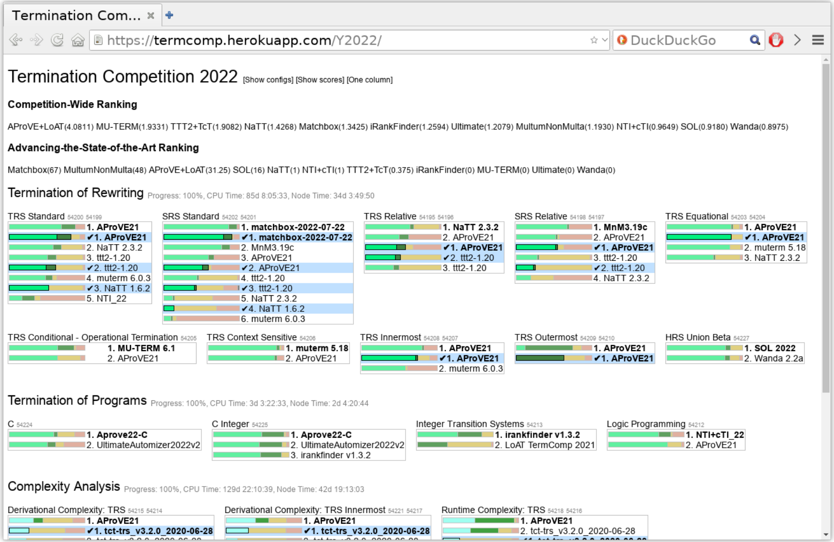
⇒ Termination provers can also be successful SMT solvers!

(disclaimer: Z3 participated only *hors concours*)

The Termination Competition (termCOMP) (1/3)



The Termination Competition (termCOMP) (1/3)



https://termination-portal.org/wiki/Termination_Competition

The Termination Competition (termCOMP) (2/3)

termCOMP 2022 participants (2024 similar):

- 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)
- $\mathsf{T_TT_2}$ (U Innsbruck)
- Ultimate Automizer (U Freiburg)
- Wanda (RU Nijmegen)

The Termination Competition (termCOMP) (3/3)

- Benchmark set: Termination Problem DataBase (TPDB)
  https://termination-portal.org/wiki/TPDB
  $\longrightarrow$ 1000s of termination and complexity problems

The Termination Competition (termCOMP) (3/3)

- Benchmark set: Termination Problem DataBase (TPDB)
  https://termination-portal.org/wiki/TPDB
  $\longrightarrow$ 1000s of termination and complexity problems
- Timeout: 300 seconds

The Termination Competition (termCOMP) (3/3)

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

The Termination Competition (termCOMP) (3/3)

- Benchmark set: Termination Problem DataBase (TPDB)
  https://termination-portal.org/wiki/TPDB
  $\longrightarrow$ 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

The Termination Competition (termCOMP) (3/3)

- Benchmark set: Termination Problem DataBase (TPDB)
  https://termination-portal.org/wiki/TPDB
  $\longrightarrow$ 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

Input for Automated Tools

Web interfaces available:

- AProVE: https://aprove.informatik.rwth-aachen.de/interface
- iRankFinder: http://irankfinder.loopkiller.com:8081/
- Mu-Term: http://zenon.dsic.upv.es/muterm/index.php/web-interface/
- $\mathrm{T_TT_2}$: http://colo6-c703.uibk.ac.at/ttt2/web/

Input for Automated Tools

Web interfaces available:

- AProVE: https://aprove.informatik.rwth-aachen.de/interface
- iRankFinder: http://irankfinder.loopkiller.com:8081/
- Mu-Term: http://zenon.dsic.upv.es/muterm/index.php/web-interface/
- $\mathrm{T_TT_2}$: 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)

if $(\mathsf{x} \geq 0)$
  while $(\mathsf{x} \neq 0)$
    $\mathsf{x} = \mathsf{x} - 1;$

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

Papers on termination of imperative programs often about **integers** as data

Example (Imperative Program)

ℓ_0: **if** $(\mathsf{x} \geq 0)$
ℓ_1: **while** $(\mathsf{x} \neq 0)$
ℓ_2: $\mathsf{x} = \mathsf{x} - 1;$

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

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

$$\begin{array}{lcll}
\ell_0(x) & \rightarrow & \ell_1(x) & [x \geq 0] \\
\ell_0(x) & \rightarrow & \ell_3(x) & [x < 0] \\
\ell_1(x) & \rightarrow & \ell_2(x) & [x \neq 0] \\
\ell_2(x) & \rightarrow & \ell_1(x-1) & \\
\ell_1(x) & \rightarrow & \ell_3(x) & [x = 0]
\end{array}$$

Papers on termination of imperative programs often about **integers** as data

Example (Imperative Program)

ℓ_0: **if** $(x \geq 0)$
ℓ_1: **while** $(x \neq 0)$
ℓ_2: $x = x - 1;$

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

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

$$\begin{array}{lcll}
\ell_0(x) & \to & \ell_1(x) & [x \geq 0] \\
\ell_0(x) & \to & \ell_3(x) & [x < 0] \\
\ell_1(x) & \to & \ell_2(x) & [x \neq 0] \\
\ell_2(x) & \to & \ell_1(x-1) & \\
\ell_1(x) & \to & \ell_3(x) & [x = 0]
\end{array}$$

Oh no! $\ell_1(-1) \to \ell_2(-1) \to \ell_1(-2) \to \ell_2(-2) \to \ell_1(-3) \to \cdots$

Papers on termination of imperative programs often about **integers** as data

Example (Imperative Program)

ℓ_0: **if** $(x \geq 0)$
ℓ_1:   **while** $(x \neq 0)$
ℓ_2: $x = x - 1;$

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

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

$$
\begin{array}{lclr}
\ell_0(x) & \to & \ell_1(x) & [x \geq 0] \\
\ell_0(x) & \to & \ell_3(x) & [x < 0] \\
\ell_1(x) & \to & \ell_2(x) & [x \neq 0] \\
\ell_2(x) & \to & \ell_1(x-1) & \\
\ell_1(x) & \to & \ell_3(x) & [x = 0]
\end{array}
$$

Oh no! $\ell_1(-1) \to \ell_2(-1) \to \ell_1(-2) \to \ell_2(-2) \to \ell_1(-3) \to \cdots$

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

Papers on termination of imperative programs often about **integers** as data

Example (Imperative Program)

ℓ_0: **if** $(\mathsf{x} \geq 0)$
ℓ_1: **while** $(\mathsf{x} \neq 0)$
ℓ_2: $\mathsf{x} = \mathsf{x} - 1;$

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

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

$$\begin{array}{lcll} \ell_0(x) & \to & \ell_1(x) & [x \geq 0] \\ \ell_0(x) & \to & \ell_3(x) & [x < 0] \\ \ell_1(x) & \to & \ell_2(x) & [x \neq 0] \\ \ell_2(x) & \to & \ell_1(x-1) & \\ \ell_1(x) & \to & \ell_3(x) & [x = 0] \end{array}$$

Oh no! $\ell_1(-1) \to \ell_2(-1) \to \ell_1(-2) \to \ell_2(-2) \to \ell_1(-3) \to \cdots$

⇒ **Restrict initial states** to $\ell_0(z)$ for $z \in \mathbb{Z}$
⇒ Find **invariant** $x \geq 0$ at ℓ_1, ℓ_2 (exercise)

Papers on termination of imperative programs often about **integers** as data

Example (Imperative Program)

```
ℓ0:  if (x ≥ 0)
ℓ1:      while (x ≠ 0)
ℓ2:          x = x − 1;
```

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

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

$$\begin{array}{lcll} \ell_0(x) & \to & \ell_1(x) & [x \geq 0] \\ \ell_0(x) & \to & \ell_3(x) & [x < 0] \\ \ell_1(x) & \to & \ell_2(x) & [x \neq 0 \land x \geq 0] \\ \ell_2(x) & \to & \ell_1(x-1) & [x \geq 0] \\ \ell_1(x) & \to & \ell_3(x) & [x = 0 \land x \geq 0] \end{array}$$

Oh no! $\ell_1(-1) \to \ell_2(-1) \to \ell_1(-2) \to \ell_2(-2) \to \ell_1(-3) \to \cdots$

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

⇒ Find **invariant** $x \geq 0$ at ℓ_1, ℓ_2 (exercise)

Proving Termination with Invariants

Example (Transition system with invariants)

$$
\begin{array}{lcll}
\ell_0(x) & \to & \ell_1(x) & [x \geq 0] \\
\ell_1(x) & \to & \ell_2(x) & [x \neq 0 \wedge x \geq 0] \\
\ell_2(x) & \to & \ell_1(x-1) & [x \geq 0] \\
\ell_1(x) & \to & \ell_3(x) & [x = 0 \wedge x \geq 0]
\end{array}
$$

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

Proving Termination with Invariants

Example (Transition system with invariants)

$$
\begin{array}{lcll}
\ell_0(x) & \succsim & \ell_1(x) & [x \geq 0] \\
\ell_1(x) & \succsim & \ell_2(x) & [x \neq 0 \land x \geq 0] \\
\ell_2(x) & \succ & \ell_1(x-1) & [x \geq 0] \\
\ell_1(x) & \succsim & \ell_3(x) & [x = 0 \land x \geq 0]
\end{array}
$$

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

Proving Termination with Invariants

Example (Transition system with invariants)

$$\begin{array}{lcll} \ell_0(x) & \succsim & \ell_1(x) & [x \geq 0] \\ \ell_1(x) & \succsim & \ell_2(x) & [x \neq 0 \land x \geq 0] \\ \ell_2(x) & \succ & \ell_1(x-1) & [x \geq 0] \\ \ell_1(x) & \succsim & \ell_3(x) & [x = 0 \land x \geq 0] \end{array}$$

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

Automate search using parametric ranking function:

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

Proving Termination with Invariants

Example (Transition system with invariants)

$$\begin{array}{llll} \ell_0(x) & \succsim & \ell_1(x) & [x \geq 0] \\ \ell_1(x) & \succsim & \ell_2(x) & [x \neq 0 \land x \geq 0] \\ \ell_2(x) & \succ & \ell_1(x-1) & [x \geq 0] \\ \ell_1(x) & \succsim & \ell_3(x) & [x = 0 \land x \geq 0] \end{array}$$

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:

$$x \geq 0 \quad \Rightarrow \quad a_2 + b_2 \cdot x > a_1 + b_1 \cdot (x-1) \quad \text{"decrease \ldots"}$$
$$x \geq 0 \quad \Rightarrow \quad a_2 + b_2 \cdot x \geq 0 \quad \text{"\ldots against a bound"}$$

Proving Termination with Invariants

Example (Transition system with invariants)

$$\begin{array}{llll} \ell_0(x) & \succsim & \ell_1(x) & [x \geq 0] \\ \ell_1(x) & \succsim & \ell_2(x) & [x \neq 0 \land x \geq 0] \\ \ell_2(x) & \succ & \ell_1(x-1) & [x \geq 0] \\ \ell_1(x) & \succsim & \ell_3(x) & [x = 0 \land x \geq 0] \end{array}$$

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:

$$\begin{array}{lcll} x \geq 0 & \Rightarrow & a_2 + b_2 \cdot x > a_1 + b_1 \cdot (x-1) & \text{"decrease \ldots"} \\ x \geq 0 & \Rightarrow & a_2 + b_2 \cdot x \geq 0 & \text{"\ldots against a bound"} \end{array}$$

Use Farkas' Lemma to eliminate $\forall x$, solver for **linear** constraints gives model for a_i, b_i.

Proving Termination with Invariants

Example (Transition system with invariants)

$$
\begin{array}{llll}
\ell_0(x) & \succsim & \ell_1(x) & [x \geq 0] \\
\ell_1(x) & \succsim & \ell_2(x) & [x \neq 0 \land x \geq 0] \\
\ell_2(x) & \succ & \ell_1(x-1) & [x \geq 0] \\
\ell_1(x) & \succsim & \ell_3(x) & [x = 0 \land x \geq 0]
\end{array}
$$

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:

$$
\begin{array}{lll l}
x \geq 0 & \Rightarrow & a_2 + b_2 \cdot x > a_1 + b_1 \cdot (x-1) & \text{“decrease \ldots”} \\
x \geq 0 & \Rightarrow & a_2 + b_2 \cdot x \geq 0 & \text{“\ldots against a bound”}
\end{array}
$$

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)

$$\begin{array}{lcll} \ell_0(x) & \to & \ell_1(x) & [x \geq 0] \\ \ell_1(x) & \to & \ell_2(x) & [x \neq 0 \land x \geq 0] \\ & & & \\ \ell_1(x) & \to & \ell_3(x) & [x = 0 \land x \geq 0] \end{array}$$

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:

$$\begin{array}{lcll} x \geq 0 & \Rightarrow & a_2 + b_2 \cdot x > a_1 + b_1 \cdot (x - 1) & \text{"decrease \ldots"} \\ x \geq 0 & \Rightarrow & a_2 + b_2 \cdot x \geq 0 & \text{"\ldots\ against a bound"} \end{array}$$

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

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*, ...]
  $\longrightarrow$ abstract interpretation [Cousot, Cousot, *POPL '77*]
  $\longrightarrow$ 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 [Otto et al, *RTA '10*; Ströder et al, *JAR '17*, . . .]
  $\longrightarrow$ abstract interpretation [Cousot, Cousot, *POPL '77*]
  $\longrightarrow$ more about this in a few minutes!

- By counterexample-based reasoning + safety prover: **Terminator**
  [Cook, Podelski, Rybalchenko, *CAV '06, PLDI '06*]
  $\longrightarrow$ prove termination of single program **runs**
  $\longrightarrow$ 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*, . . .]
  $\longrightarrow$ abstract interpretation [Cousot, Cousot, *POPL '77*]
  $\longrightarrow$ more about this in a few minutes!
- By counterexample-based reasoning + safety prover: **Terminator**
  [Cook, Podelski, Rybalchenko, *CAV '06, PLDI '06*]
  $\longrightarrow$ prove termination of single program **runs**
  $\longrightarrow$ 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 [Otto et al, *RTA '10*; Ströder et al, *JAR '17*, ...]
  $\longrightarrow$ abstract interpretation [Cousot, Cousot, *POPL '77*]
  $\longrightarrow$ more about this in a few minutes!

- By counterexample-based reasoning + safety prover: **Terminator**
  [Cook, Podelski, Rybalchenko, *CAV '06, PLDI '06*]
  $\longrightarrow$ prove termination of single program **runs**
  $\longrightarrow$ 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: **VeryMax**
  [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 [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*]

- Using Max-SMT: **VeryMax**
  [Larraz, Oliveras, Rodríguez-Carbonell, Rubio, *FMCAD '13*]

Nowadays all SMT-based!

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*, . . .]

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*, ...]

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*]

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 $\mathsf{s}(\mathsf{s}(\mathsf{s}(...)))$?!

Beyond Classic TRSs for Program Analysis

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

Drawbacks:

Beyond Classic TRSs for Program Analysis

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

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

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
- does not benefit from dedicated constraint solvers
  (e.g., SMT solvers) for arithmetic operations in programs

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
- 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

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

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

$\Rightarrow$ Term rewriting + SMT solving for automated reasoning

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

$\Rightarrow$ Term rewriting + SMT solving for automated reasoning

- Constrained rewriting known at least since [Vorobyov, *RTA '89*]

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

$\Rightarrow$ Term rewriting + SMT solving for automated reasoning

- Constrained rewriting known at least since [Vorobyov, *RTA '89*]
- 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

$\Rightarrow$ Term rewriting + SMT solving for automated reasoning

- Constrained rewriting known at least since [Vorobyov, *RTA '89*]
- 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*]

Logically Constrained TRSs: Adoption

Analysis techniques for Logically Constrained TRSs:

- Termination [Kop, *WST '13*; Nishida, Winkler, *VSTTE '18*]
- Complexity [Winkler, Moser, *LOPSTR '20*]
- Equivalence [Fuhs, Kop, Nishida, *TOCL '17*; Ciobâcă, Lucanu, Buruiana, *JLAMP '23*]
- Confluence [Schöpf, Middeldorp, *CADE '23*; Schöpf, Mitterwallner, Middeldorp, *IJCAR '24*]
- Reachability / Safety [Ciobâcă, Lucanu, *IJCAR '18*; Kojima, Nishida, *JLAMP '23*]

Constrained Rewriting by Example

Example (Constrained Rewrite System)

$$
\begin{array}{rcll}
\ell_0(n, r) & \to & \ell_1(n, r, \mathsf{Nil}) & \\
\ell_1(n, r, xs) & \to & \ell_1(n-1, r+1, \mathsf{Cons}(r, xs)) & [n > 0] \\
\ell_1(n, r, xs) & \to & \ell_2(xs) & [n = 0]
\end{array}
$$

Constrained Rewriting by Example

Example (Constrained Rewrite System)

$$
\begin{array}{rcll}
\ell_0(n, r) & \to & \ell_1(n, r, \mathsf{Nil}) & \\
\ell_1(n, r, xs) & \to & \ell_1(n-1, r+1, \mathsf{Cons}(r, xs)) & [n > 0] \\
\ell_1(n, r, xs) & \to & \ell_2(xs) & [n = 0]
\end{array}
$$

Possible rewrite sequence:

$$\ell_0(2, 7)$$

Constrained Rewriting by Example

Example (Constrained Rewrite System)

$$
\begin{array}{rcll}
\ell_0(n, r) & \to & \ell_1(n, r, \mathsf{Nil}) & \\
\ell_1(n, r, xs) & \to & \ell_1(n-1, r+1, \mathsf{Cons}(r, xs)) & [n > 0] \\
\ell_1(n, r, xs) & \to & \ell_2(xs) & [n = 0]
\end{array}
$$

Possible rewrite sequence:

$$
\begin{array}{l}
\quad \ell_0(2, 7) \\
\to \ell_1(2, 7, \mathsf{Nil})
\end{array}
$$

Constrained Rewriting by Example

Example (Constrained Rewrite System)

$$
\begin{array}{rcll}
\ell_0(n, r) & \to & \ell_1(n, r, \mathsf{Nil}) & \\
\ell_1(n, r, xs) & \to & \ell_1(n-1, r+1, \mathsf{Cons}(r, xs)) & [n > 0] \\
\ell_1(n, r, xs) & \to & \ell_2(xs) & [n = 0]
\end{array}
$$

Possible rewrite sequence:

$$
\begin{array}{l}
\quad \ell_0(2, 7) \\
\to \ell_1(2, 7, \mathsf{Nil}) \\
\to \ell_1(1, 8, \mathsf{Cons}(7, \mathsf{Nil}))
\end{array}
$$

Constrained Rewriting by Example

Example (Constrained Rewrite System)

$$
\begin{array}{rcll}
\ell_0(n, r) & \rightarrow & \ell_1(n, r, \mathsf{Nil}) & \\
\ell_1(n, r, xs) & \rightarrow & \ell_1(n-1, r+1, \mathsf{Cons}(r, xs)) & [n > 0] \\
\ell_1(n, r, xs) & \rightarrow & \ell_2(xs) & [n = 0]
\end{array}
$$

Possible rewrite sequence:

$$
\begin{array}{l}
\quad \ell_0(2, 7) \\
\rightarrow \ell_1(2, 7, \mathsf{Nil}) \\
\rightarrow \ell_1(1, 8, \mathsf{Cons}(7, \mathsf{Nil})) \\
\rightarrow \ell_1(0, 9, \mathsf{Cons}(8, \mathsf{Cons}(7, \mathsf{Nil})))
\end{array}
$$

Constrained Rewriting by Example

Example (Constrained Rewrite System)

$$\begin{array}{rcll} \ell_0(n, r) & \to & \ell_1(n, r, \mathsf{Nil}) & \\ \ell_1(n, r, xs) & \to & \ell_1(n-1, r+1, \mathsf{Cons}(r, xs)) & [n > 0] \\ \ell_1(n, r, xs) & \to & \ell_2(xs) & [n = 0] \end{array}$$

Possible rewrite sequence:

$$\begin{array}{l} \quad \ell_0(2, 7) \\ \to \ell_1(2, 7, \mathsf{Nil}) \\ \to \ell_1(1, 8, \mathsf{Cons}(7, \mathsf{Nil})) \\ \to \ell_1(0, 9, \mathsf{Cons}(8, \mathsf{Cons}(7, \mathsf{Nil}))) \\ \to \ell_2(\mathsf{Cons}(8, \mathsf{Cons}(7, \mathsf{Nil}))) \end{array}$$

Constrained Rewriting by Example

Example (Constrained Rewrite System)

$$
\begin{array}{rcll}
\ell_0(n, r) & \rightarrow & \ell_1(n, r, \mathsf{Nil}) & \\
\ell_1(n, r, xs) & \rightarrow & \ell_1(n-1, r+1, \mathsf{Cons}(r, xs)) & [n > 0] \\
\ell_1(n, r, xs) & \rightarrow & \ell_2(xs) & [n = 0]
\end{array}
$$

Possible rewrite sequence:

$$
\begin{array}{l}
\quad \ell_0(2, 7) \\
\rightarrow \ell_1(2, 7, \mathsf{Nil}) \\
\rightarrow \ell_1(1, 8, \mathsf{Cons}(7, \mathsf{Nil})) \\
\rightarrow \ell_1(0, 9, \mathsf{Cons}(8, \mathsf{Cons}(7, \mathsf{Nil}))) \\
\rightarrow \ell_2(\mathsf{Cons}(8, \mathsf{Cons}(7, \mathsf{Nil})))
\end{array}
$$

Here 7, 8, $\ldots$ are predefined constants.

Constrained Rewriting by Example

Example (Constrained Rewrite System)

$$
\begin{array}{rcll}
\ell_0(n, r) & \to & \ell_1(n, r, \mathsf{Nil}) & \\
\ell_1(n, r, xs) & \to & \ell_1(n-1, r+1, \mathsf{Cons}(r, xs)) & [n > 0] \\
\ell_1(n, r, xs) & \to & \ell_2(xs) & [n = 0]
\end{array}
$$

Possible rewrite sequence:

$$
\begin{array}{l}
\quad \ell_0(2, 7) \\
\to \ell_1(2, 7, \mathsf{Nil}) \\
\to \ell_1(1, 8, \mathsf{Cons}(7, \mathsf{Nil})) \\
\to \ell_1(0, 9, \mathsf{Cons}(8, \mathsf{Cons}(7, \mathsf{Nil}))) \\
\to \ell_2(\mathsf{Cons}(8, \mathsf{Cons}(7, \mathsf{Nil})))
\end{array}
$$

Here 7, 8, $\ldots$ are predefined constants.

Termination proof: reuse techniques for TRSs and integer programs

Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last ~ 25 years

Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last $\sim$ 25 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 $\sim$ 25 years
- Term rewriting: handles **inductive data structures** well
- Imperative programs on integers: need to consider **reachability/safety** and **invariants**

Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last $\sim$ 25 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 ~ 25 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

Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last $\sim$ 25 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

Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last $\sim$ 25 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 $\sim$ 25 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 $\sim$ 25 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 . . .

Conclusion: Termination Proving Back-Ends

- Automated termination analysis for term rewriting and for imperative programs developed in parallel over the last $\sim$ 25 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

Front-End: from Program to Constrained Term Rewriting, high-level

- execute program **symbolically** from initial states of the program, handle language peculiarities here ($\longrightarrow$ Java: sharing, cyclicity analysis)

```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```

Front-End: from Program to Constrained Term Rewriting, high-level

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

init(...)

```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```

Front-End: from Program to Constrained Term Rewriting, high-level

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

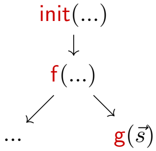
```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```

$\mathsf{init}(...)$
$\downarrow$
$\mathsf{f}(...)$

Front-End: from Program to Constrained Term Rewriting, high-level

- execute program **symbolically** from initial states of the program, handle language peculiarities here ($\longrightarrow$ Java: sharing, cyclicity analysis)

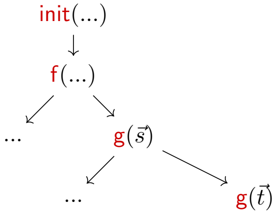
```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```



Front-End: from Program to Constrained Term Rewriting, high-level

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

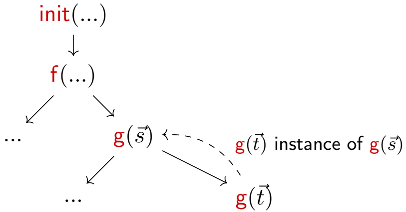
```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```



Front-End: from Program to Constrained Term Rewriting, high-level

- execute program **symbolically** from initial states of the program, handle language peculiarities here ($\longrightarrow$ Java: sharing, cyclicity analysis)
- use **generalisation** of program states, get **over-approximation** of all possible program runs ($\approx$ control-flow graph with extra info)
- closely related: Abstract Interpretation

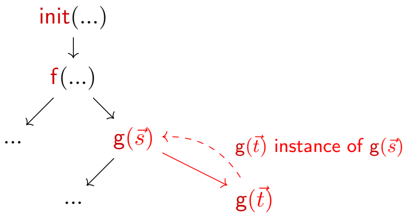
```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```



Front-End: from Program to Constrained Term Rewriting, high-level

- execute program **symbolically** from initial states of the program, handle language peculiarities here ($\longrightarrow$ Java: sharing, cyclicity analysis)
- use **generalisation** of program states, get **over-approximation** of all possible program runs ($\approx$ control-flow graph with extra info)
- closely related: Abstract Interpretation
- **extract TRS** from cycles in the representation

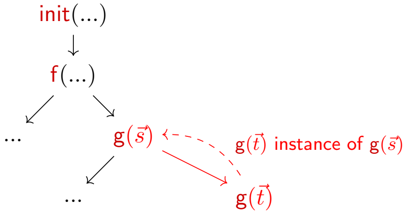
```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```



Front-End: from Program to Constrained Term Rewriting, high-level

- execute program **symbolically** from initial states of the program, handle language peculiarities here ($\longrightarrow$ Java: sharing, cyclicity analysis)
- use **generalisation** of program states, get **over-approximation** of all possible program runs ($\approx$ control-flow graph with extra info)
- closely related: Abstract Interpretation
- **extract TRS** from cycles in the representation
- if TRS terminates
  $\Rightarrow$ any **concrete program execution** can use cycles only finitely often
  $\Rightarrow$ the program **must terminate**

```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```



Application: Termination Analysis of Programs

Recipe for proving program termination by reusing TRS termination provers

Application: Termination Analysis of Programs

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\longrightarrow$ here: what data objects can we represent as terms?

Application: Termination Analysis of Programs

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\longrightarrow$ here: what data objects can we represent as terms?
- Execute program **symbolically** from its initial states

Application: Termination Analysis of Programs

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\longrightarrow$ 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)

Application: Termination Analysis of Programs

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\longrightarrow$ 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

Application: Termination Analysis of Programs

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\longrightarrow$ 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
  $\Rightarrow$ 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
- ...

Java Example

```
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 ($\longrightarrow$ web interface)

http://aprove.informatik.rwth-aachen.de/

Java: Source Code vs Bytecode

[Otto et al, *RTA '10*] describe their technique for **compiled** Java programs: **Java Bytecode**

Java: Source Code vs 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 ...

Java: Source Code vs Bytecode

```
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 **cor** ;: **Java Bytecode**

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

Java: Source Code vs 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 . . .

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: $o_1 =^? o_2$

`03`	num : o_1, `limit` : o_2
o_1 : `MyInt`(?)	
o_2 : `MyInt`(val $= i_1$)	
i_1 : $[4, 80]$	

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: $o_1 \searrow\!\!\swarrow o_2$

`03`	`num` : o_1, `limit` : o_2
o_1 : `MyInt`(?)	
o_2 : `MyInt`(`val` $= i_1$)	
i_1 : $[4, 80]$	

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: $o_1 \searrow\!\!\swarrow o_2$
   - Reference may have cycles: $o_1\,!$

`03` | num : o_1, `limit` : o_2
o_1 : `MyInt`(?)
o_2 : `MyInt`(val $= i_1$)
i_1 : $[4, 80]$

Building the Symbolic Execution Graph

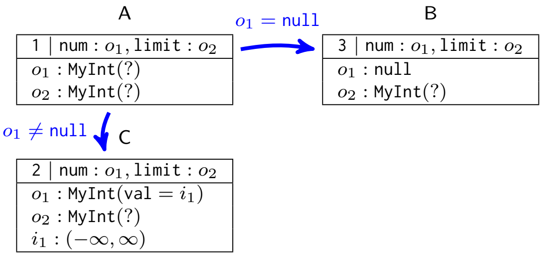
```
   public class MyInt {
     private int val;
     static void count(MyInt num, MyInt limit) {
1:     if (num == null
2:         || limit == null)
3:       return;
4:     MyInt copy = num;
5:     while (num.val < limit.val)
6:       copy.val++;
7: } }
```

A

1	num : o_1, limit : o_2
o_1 : MyInt$(?)$	
o_2 : MyInt$(?)$	

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++;
} }
```

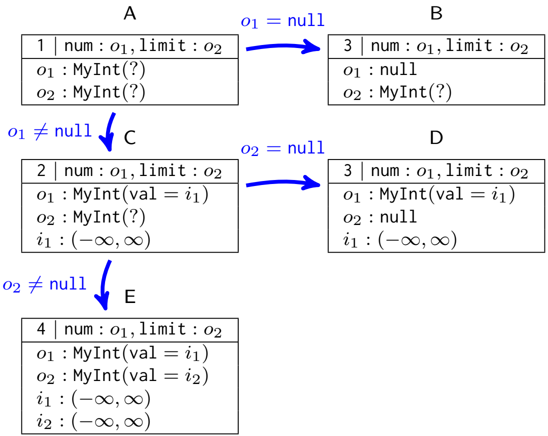


$$\text{X} \xrightarrow{cond} \text{Y}$$

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
2:         || limit == null)
3:       return;
4:     MyInt copy = num;
5:     while (num.val < limit.val)
6:       copy.val++;
7: } }
```

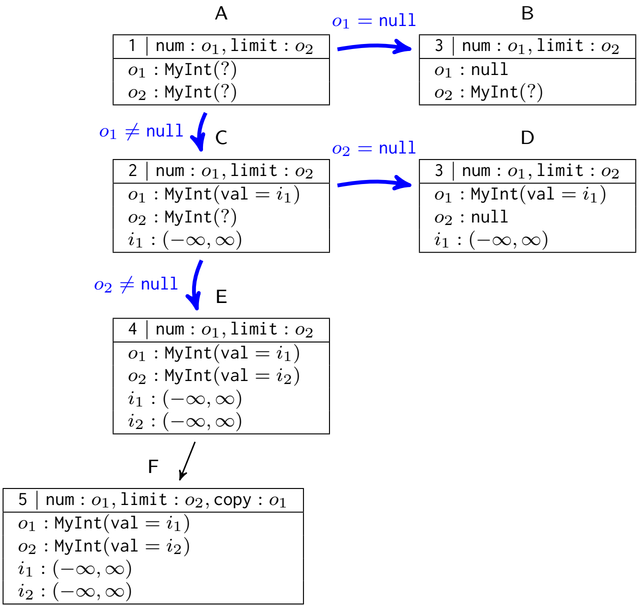


$$X \xrightarrow{cond} Y$$

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
2:         || limit == null)
3:       return;
4:     MyInt copy = num;
5:     while (num.val < limit.val)
6:       copy.val++;
7: } }
```

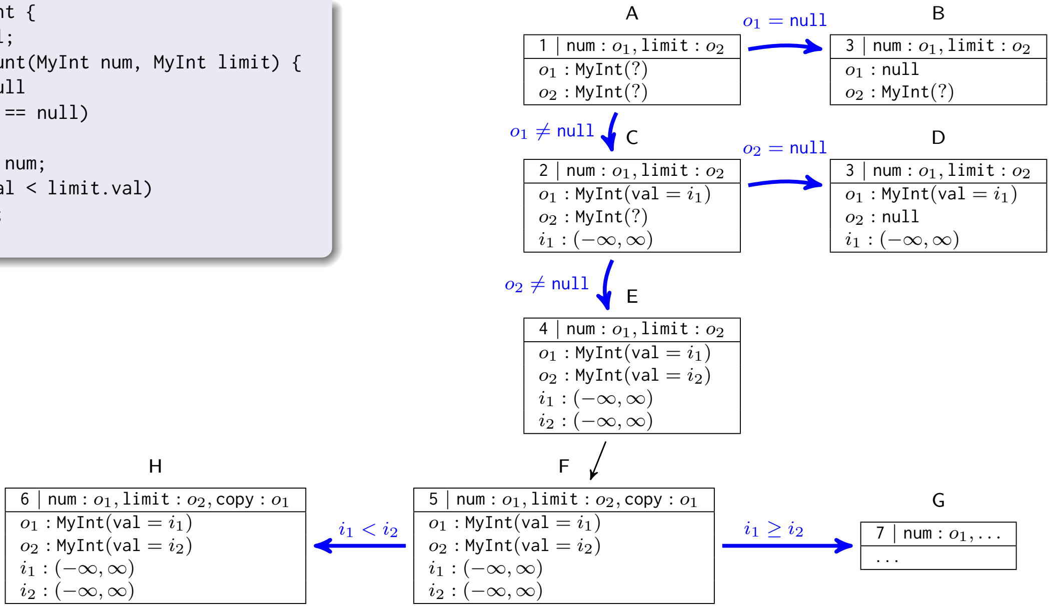


X ⟶ Y

means: evaluate X to Y

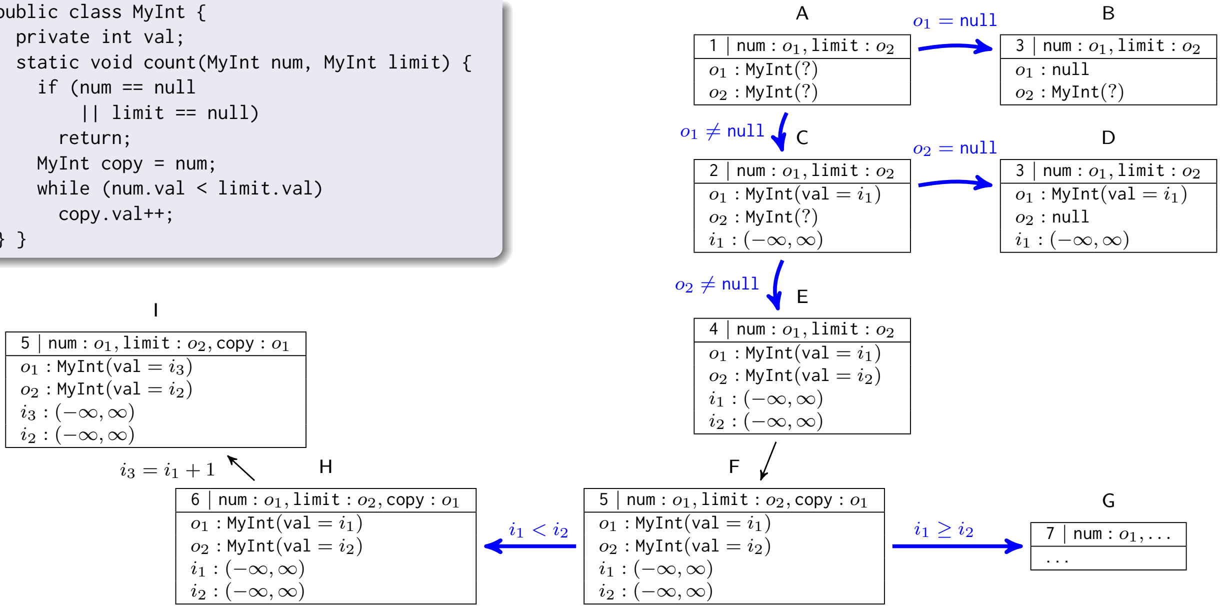
Building the Symbolic Execution Graph

```
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++;
} }
```

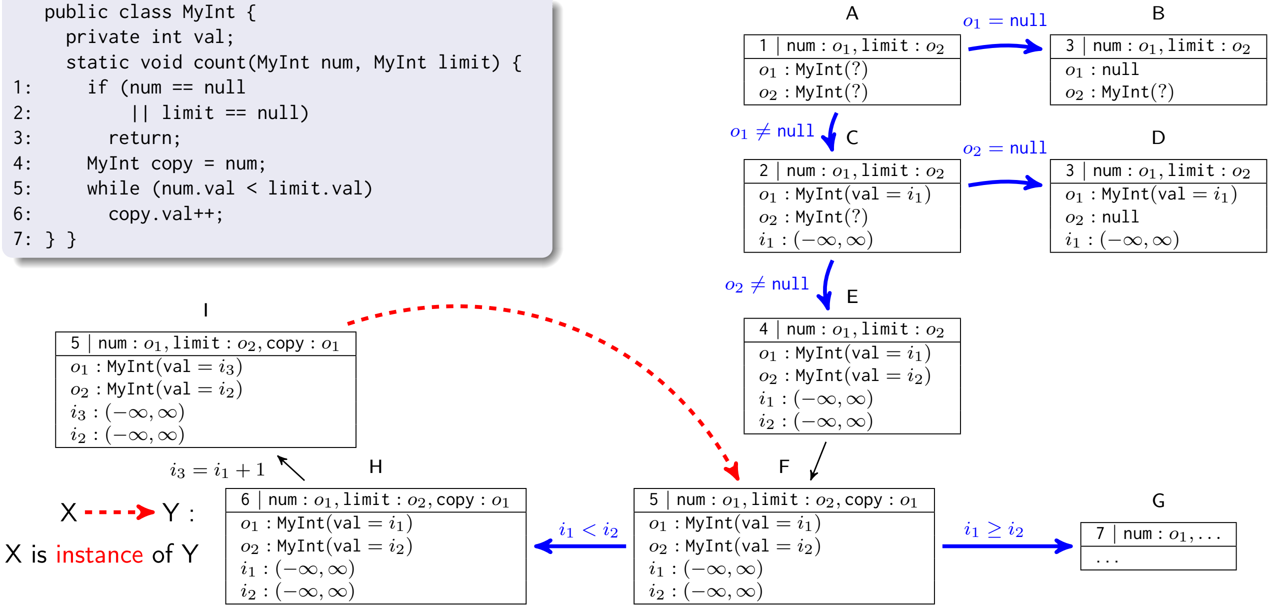


Building the Symbolic Execution Graph

```
   public class MyInt {
     private int val;
     static void count(MyInt num, MyInt limit) {
1:     if (num == null
2:         || limit == null)
3:       return;
4:     MyInt copy = num;
5:     while (num.val < limit.val)
6:       copy.val++;
7: } }
```



```
   public class MyInt {
     private int val;
     static void count(MyInt num, MyInt limit) {
1:     if (num == null
2:         || limit == null)
3:       return;
4:     MyInt copy = num;
5:     while (num.val < limit.val)
6:       copy.val++;
7: } }
```



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

From Java to Symbolic Execution Graphs

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)

Q

16 | num : o_1, limit : o_2, x : o_3, y : o_4, z : i_1
o_1 : MyInt$(?)$
o_2 : MyInt(val $= i_2$)
o_3 : null
o_4 : MyList$(?)$
o_4 !
i_1 : $[7, \infty)$
i_2 : $(-\infty, \infty)$

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

- **term** for o_1: o_1
- **term** for o_2: MyInt(i_2)
- **term** for o_3: 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)

Transformation of Objects to Terms (2/2)

```
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**:

Transformation of Objects to Terms (2/2)

```
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`: $\mathsf{A}(\mathsf{eoc}, 1)$

  $\longrightarrow$ eoc for end of class
- **term** for `y`: $\mathsf{A}(\mathsf{B}(\mathsf{eoc}, 3), 2)$

Transformation of Objects to Terms (2/2)

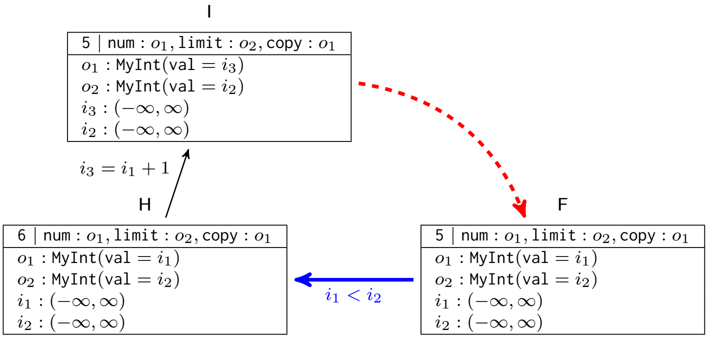
```
public class A {
  int a;
}

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**:

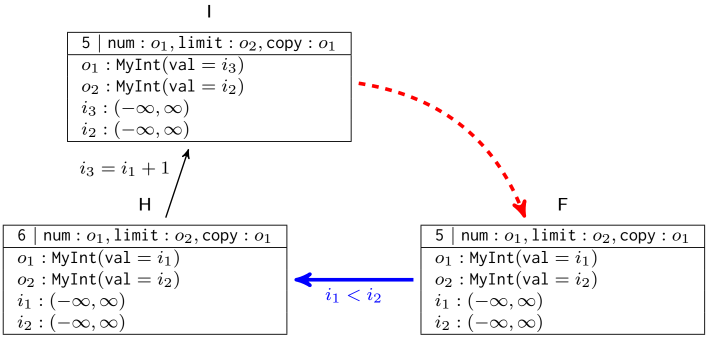
- for every class `C` with n fields, introduce $(n+1)$-ary function symbol C
- first argument: part of the object corresponding to subclasses of `C`
- **term** for `x`: $\mathsf{jlO}(\mathsf{A}(\mathsf{eoc}, 1))$

  $\longrightarrow$ eoc for end of class
- **term** for `y`: $\mathsf{jlO}(\mathsf{A}(\mathsf{B}(\mathsf{eoc}, 3), 2))$
- every class extends `Object`!
  ($\longrightarrow$ jlO $\equiv$ `java.lang.Object`)

From the Symbolic Execution Graph to Terms and Rules



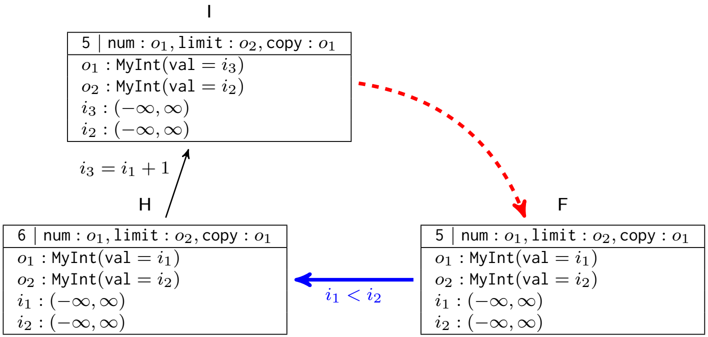
From the Symbolic Execution Graph to Terms and Rules



- State F: $\ell_F(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$

  State H: $\ell_H(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$

From the Symbolic Execution Graph to Terms and Rules

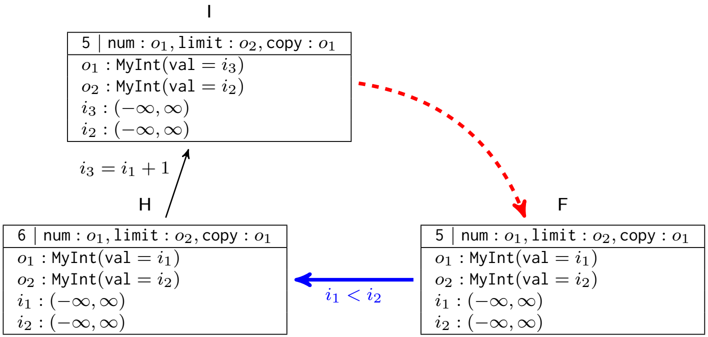


- State F: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$

  $\longrightarrow$

  State H: $\ell_{\mathsf{H}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$ $\quad [i_1 < i_2]$

From the Symbolic Execution Graph to Terms and Rules



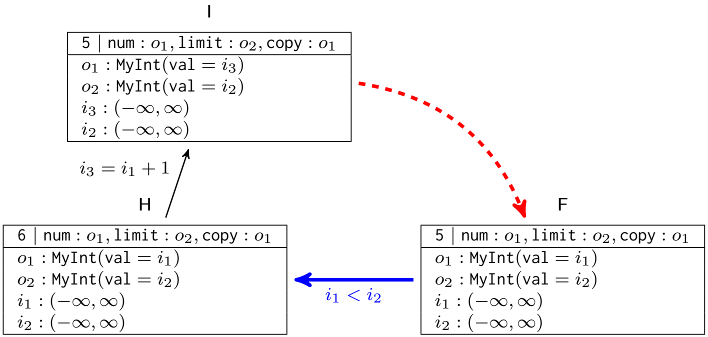
- State F: $\ell_F(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$

  $\longrightarrow$

  State H: $\ell_H(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$ $[i_1 < i_2]$

- State H: $\ell_H(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$

  State I: $\ell_F(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1 + 1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$



- State F: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$

  $\longrightarrow$

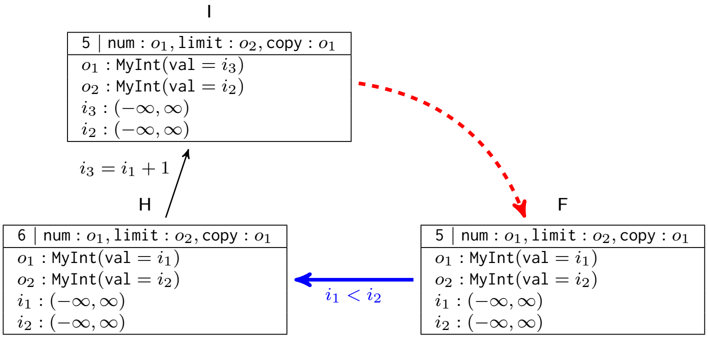
  State H: $\ell_{\mathsf{H}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )\qquad [i_1 < i_2]$

- State H: $\ell_{\mathsf{H}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$

  $\longrightarrow$

  State I: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1 + 1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$

From the Symbolic Execution Graph to Terms and Rules



- State F: $\ell_F(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$

  $\longrightarrow$

  State H: $\ell_H(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$ $[i_1 < i_2]$
- State H: $\ell_H(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$

  $\longrightarrow$

  State I: $\ell_F(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1 + 1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$
- Termination easy to show (intuitively: $i_2 - i_1$ decreases against bound 0)

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*]

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

⇒ 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)

⇒ abstract domain based on equivalent **linear** Prolog semantics [Ströder et al, *LOPSTR '11*], tracks which variables are for ground terms vs arbitrary terms

Front-End for LLVM

LLVM [Ströder et al, *JAR '17*]

- LLVM bitcode: intermediate language of LLVM compiler framework
- `clang` compiler has prominent frontend for C
- challenges: memory safety, pointer arithmetic

Front-End for LLVM

LLVM [Ströder et al, *JAR '17*]

- LLVM bitcode: intermediate language of LLVM compiler framework
- `clang` compiler has prominent frontend for C
- challenges: memory safety, pointer arithmetic

⇒ abstract domain tracks information about allocated memory and its content; extract Integer Transition System

Front-End for LLVM

LLVM [Ströder et al, *JAR '17*]

- LLVM bitcode: intermediate language of LLVM compiler framework
- `clang` compiler has prominent frontend for C
- challenges: memory safety, pointer arithmetic

⇒ abstract domain tracks information about allocated memory and its content; extract Integer Transition System

Extensions:

- bitvector `int` semantics [Hensel et al, *JLAMP '18*]
- linked lists [Hensel, Giesl, *CADE '23*]

Conclusion: Termination Analysis for Programs

- Termination proving for (LC)TRSs driven by SMT solvers

Conclusion: Termination Analysis for Programs

- Termination proving for (LC)TRSs driven by SMT solvers
- Constrained rewriting: Term rewriting + pre-defined primitive data structures

Conclusion: Termination Analysis for Programs

- Termination proving for (LC)TRSs 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**

Conclusion: Termination Analysis for Programs

- Termination proving for (LC)TRSs 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

II. Complexity Analysis

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
- ...

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
- ...

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

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
  $\longrightarrow$ related DARPA project: *Space/Time Analysis for Cybersecurity*
  https://www.darpa.mil/program/space-time-analysis-for-cybersecurity

Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks
  $\longrightarrow$ 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
  $\longrightarrow$ 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
  $\longrightarrow$ 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
  $\longrightarrow$ computational cost as a non-functional requirement!

Why Care About Computational Cost, Anyway?

- **Mobile devices**: Bound energy usage
- **Security**: Denial of Service attacks
  $\longrightarrow$ 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
  $\longrightarrow$ 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
  $\longrightarrow$ 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
  $\longrightarrow$ 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
  $\longrightarrow$ 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
  $\longrightarrow$ 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[1]

[1] A. Merayo Corcoba: *Resource analysis of integer and abstract programs*, PhD thesis, U Complutense Madrid, 2022

Show Me Some Examples!

Question: Write a Python function that returns the sum $1 + 2 + \cdots + n$.

Show Me Some Examples!

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
```

Show Me Some Examples!

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
```

runtime in $\mathcal{O}(f(n))$ means:

- for an input of "size" n, the program needs at most about $f(n)$ steps
- the runtime is "of order $f(n)$"

Show Me Some Examples!

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
```

$\mathcal{O}(n)$

runtime in $\mathcal{O}(f(n))$ means:

- for an input of "size" n, the program needs at most about $f(n)$ steps
- the runtime is "of order $f(n)$"

Show Me Some Examples!

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
```

$\mathcal{O}(n)$

```
def sum2(n):
  r = 0
  i = 1
  while i <= n:
    r = r + i

  return r
```

runtime in $\mathcal{O}(f(n))$ means:

- for an input of "size" n, the program needs at most about $f(n)$ steps
- the runtime is "of order $f(n)$"

Show Me Some Examples!

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
```

$\mathcal{O}(n)$

```
def sum2(n):
  r = 0
  i = 1
  while i <= n:
    r = r + i

  return r
```

$\mathcal{O}(\infty)$

runtime in $\mathcal{O}(f(n))$ means:

- for an input of "size" n, the program needs at most about $f(n)$ steps
- the runtime is "of order $f(n)$"

Show Me Some Examples!

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
```

$\mathcal{O}(n)$

```
def sum2(n):
  r = 0
  i = 1
  while i <= n:
    r = r + i

  return r
```

$\mathcal{O}(\infty)$

```
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
```

runtime in $\mathcal{O}(f(n))$ means:

- for an input of "size" n, the program needs at most about $f(n)$ steps
- the runtime is "of order $f(n)$"

Show Me Some Examples!

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
```

$\mathcal{O}(n)$

```
def sum2(n):
  r = 0
  i = 1
  while i <= n:
    r = r + i

  return r
```

$\mathcal{O}(\infty)$

```
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
```

$\mathcal{O}(n^2)$

runtime in $\mathcal{O}(f(n))$ means:

- for an input of "size" n, the program needs at most about $f(n)$ steps
- the runtime is "of order $f(n)$"

Show Me Some Examples!

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
```

$\mathcal{O}(n)$

```
def sum2(n):
  r = 0
  i = 1
  while i <= n:
    r = r + i

  return r
```

$\mathcal{O}(\infty)$

```
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
```

$\mathcal{O}(n^2)$

```
def sum4(n):
  return n*(n+1)//2
```

runtime in $\mathcal{O}(f(n))$ means:

- for an input of "size" n, the program needs at most about $f(n)$ steps
- the runtime is "of order $f(n)$"

Show Me Some Examples!

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
```

$\mathcal{O}(n)$

```
def sum2(n):
  r = 0
  i = 1
  while i <= n:
    r = r + i

  return r
```

$\mathcal{O}(\infty)$

```
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
```

$\mathcal{O}(n^2)$

$\mathcal{O}(1)$

```
def sum4(n):
  return n*(n+1)//2
```

runtime in $\mathcal{O}(f(n))$ means:

- for an input of "size" n, the program needs at most about $f(n)$ steps
- the runtime is "of order $f(n)$"

Is There a Tool that Finds such Bounds Automatically?

- Fully automatic open-source tool KoAT:

  https://github.com/s-falke/kittel-koat

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
  ACM Transactions on Programming Languages and Systems 38 (4), pp. 1 – 50, 2016.

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
  ACM Transactions on Programming Languages and Systems 38 (4), pp. 1 – 50, 2016.

- 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 goes below 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 goes below 0.

$\Rightarrow$ Gives us a bound on the number of times we go through the loop

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 goes below 0.

$\Rightarrow$ 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 goes below 0.

$\Rightarrow$ 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
```

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 goes below 0.

$\Rightarrow$ 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
```

Loop 1: ranking function x

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 goes below 0.

$\Rightarrow$ 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
```

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 goes below 0.

$\Rightarrow$ 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
```

Loop 1: ranking function x

Loop 2: ranking function z

$\Rightarrow$ 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 goes below 0.

$\Rightarrow$ 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

$\Rightarrow$ 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 goes below 0.

$\Rightarrow$ 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

$\Rightarrow$ runtime in $\mathcal{O}(x + z)$

Loop 1: ranking function x

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 goes below 0.

⇒ 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

⇒ runtime in $\mathcal{O}(x + z)$

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 goes below 0.

$\Rightarrow$ 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

$\Rightarrow$ runtime in $\mathcal{O}(x + z)$

Loop 1: ranking function x
Loop 2: ranking function z

$\Rightarrow$ runtime in

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 goes below 0.

$\Rightarrow$ 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

$\Rightarrow$ runtime in $\mathcal{O}(x + z)$

Loop 1: ranking function x
Loop 2: ranking function z

$\Rightarrow$ runtime in ... oops.

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 goes below 0.

$\Rightarrow$ 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

$\Rightarrow$ runtime in $\mathcal{O}(x + z)$

Loop 1: ranking function x
Loop 2: ranking function z

$\Rightarrow$ runtime in … oops.
Best runtime bound: $\mathcal{O}(x^2 + z)$

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!

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:

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 $z_0 + x_0^2$.

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 $z_0 + x_0^2$.

So:

1. we can make at most $f_2(x, z) = z$ steps in Loop 2

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 $z_0 + x_0^2$.

So:

1. we can make at most $f_2(x, z) = z$ steps in Loop 2
2. when we enter Loop 2, we know $z \leq z_0 + x_0^2$

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 $z_0 + x_0^2$.

So:

1. we can make at most $f_2(x,z) = z$ steps in Loop 2
2. when we enter Loop 2, we know $z \leq z_0 + x_0^2$

$\Rightarrow f_2(\ldots, z_0 + x_0^2) = z_0 + x_0^2$

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 $z_0 + x_0^2$.

So:

1. we can make at most $f_2(x, z) = z$ steps in Loop 2
2. when we enter Loop 2, we know $z \leq z_0 + x_0^2$

$\Rightarrow$ $f_2(..., z_0 + x_0^2) = z_0 + x_0^2$ gives runtime bound for Loop 2: $\mathcal{O}(z_0 + x_0^2)$

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 $z_0 + x_0^2$.

So:

1. we can make at most $f_2(x, z) = z$ steps in Loop 2
2. when we enter Loop 2, we know $z \leq z_0 + x_0^2$

$\Rightarrow$ $f_2(..., z_0 + x_0^2) = z_0 + x_0^2$ gives runtime bound for Loop 2: $\mathcal{O}(z_0 + x_0^2)$

Data size influences runtime.

How Can We Build such an Oracle for Size Bounds?

```
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 Build such an Oracle for Size Bounds?

```
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 (*).

How Can We Build such an Oracle for Size Bounds?

```
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?

```
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: $x \leq x_0$

How Can We Build such an Oracle for Size Bounds?

```
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 ($\leq x_0$)

How Can We Build such an Oracle for Size Bounds?

```
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: $x \le x_0$
2. each time round Loop 1, z goes up by x ($\le x_0$)
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?

```
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 $z_0 + x_0 \cdot x_0 \ = \ z_0 + x_0^2$!

How Can We Build such an Oracle for Size Bounds?

```
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.

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], . . .

---

[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

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, D.Zanardini: *Cost analysis of object-oriented bytecode programs*, TCS '12

[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

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], ...
- Using invariant generation: SPEED[8]

---

[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

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, D.Zanardini: *Cost analysis of object-oriented bytecode programs*, TCS '12

[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

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], ...
- Using invariant generation: SPEED[8]
- Using abstract interpretation: Infer[9]

---

[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

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, D.Zanardini: *Cost analysis of object-oriented bytecode programs*, TCS '12

[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] E. Çiçek, M. Bouaziz, S. Cho, D. Distefano: *Static Resource Analysis at Scale (Extended Abstract)*, SAS '20

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], ...
- Using invariant generation: SPEED[8]
- Using abstract interpretation: Infer[9]
- Using type-based amortised analysis:[10] RAML[11], ...

---

[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

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, D.Zanardini: *Cost analysis of object-oriented bytecode programs*, TCS '12

[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] E. Çiçek, M. Bouaziz, S. Cho, D. Distefano: *Static Resource Analysis at Scale (Extended Abstract)*, SAS '20

[10] J. Hoffmann, S. Jost: *Two decades of automatic amortized resource analysis*, MSCS '22

[11] J. Hoffmann, K. Aehlig, M. Hofmann: *Resource Aware ML*, CAV '12

Current Developments

- Precise handling of loops with computable complexity in the KoAT approach[12]

[12] N. Lommen, F. Meyer, J. Giesl: *Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops*, IJCAR '22

Current Developments

- Precise handling of loops with computable complexity in the KoAT approach[12]
- Inference of **lower** bounds for worst-case runtime complexity[13]: LoAT[14]

---

[12] N. Lommen, F. Meyer, J. Giesl: *Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops*, IJCAR '22

[13] F. Frohn, M. Naaf, M. Brockschmidt, J. Giesl: *Inferring Lower Runtime Bounds for Integer Programs*, TOPLAS '20

[14] F. Frohn, J. Giesl: *Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)*, IJCAR '22

Current Developments

- Precise handling of loops with computable complexity in the KoAT approach[12]
- Inference of **lower** bounds for worst-case runtime complexity[13]: LoAT[14]
- Cost analysis for Java programs via Integer Transition Systems[15]

---

[12] N. Lommen, F. Meyer, J. Giesl: *Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops*, IJCAR '22

[13] F. Frohn, M. Naaf, M. Brockschmidt, J. Giesl: *Inferring Lower Runtime Bounds for Integer Programs*, TOPLAS '20

[14] F. Frohn, J. Giesl: *Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)*, IJCAR '22

[15] F. Frohn, J. Giesl: *Complexity Analysis for Java with AProVE*, iFM '17

Current Developments

- Precise handling of loops with computable complexity in the KoAT approach[12]
- Inference of **lower** bounds for worst-case runtime complexity[13]: LoAT[14]
- Cost analysis for Java programs via Integer Transition Systems[15]
- Cost analysis for **probabilistic** programs[16][17][18]

---

[12] N. Lommen, F. Meyer, J. Giesl: *Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops*, IJCAR '22

[13] F. Frohn, M. Naaf, M. Brockschmidt, J. Giesl: *Inferring Lower Runtime Bounds for Integer Programs*, TOPLAS '20

[14] F. Frohn, J. Giesl: *Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)*, IJCAR '22

[15] F. Frohn, J. Giesl: *Complexity Analysis for Java with AProVE*, iFM '17

[16] P. Wang, H. Fu, A. Goharshady, K. Chatterjee, X. Qin, W. Shi: *Cost analysis of nondeterministic probabilistic programs*, PLDI '19

[17] F. Meyer, M. Hark, J. Giesl: *Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes*, TACAS '21

[18] L. Leutgeb, G. Moser, F. Zuleger: *Automated Expected Amortised Cost Analysis of Probabilistic Data Structures*, CAV '22

Complexity of Integer Programs: What to Take Home?

Key insights:

- Data size influences runtime
- Runtime influences data size
- *Other influences minor*

Complexity of Integer Programs: What to Take Home?

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:

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

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) $\mathcal{R}$)

$$\mathsf{double}(\mathsf{0}) \longrightarrow \mathsf{0}$$

$$\mathsf{double}(\mathsf{s}(x)) \longrightarrow \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$$

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) $\mathcal{R}$)

$$\mathsf{double}(\mathsf{0}) \to \mathsf{0}$$

$$\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$$

Compute "double of 3 is 6":

$$\mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))$$

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) $\mathcal{R}$)

$$\mathsf{double}(\mathsf{0}) \to \mathsf{0}$$

$$\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$$

Compute "double of 3 is 6":

$$\begin{array}{ll} & \mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))) \end{array}$$

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) $\mathcal{R}$)

$$\mathsf{double}(\mathsf{0}) \to \mathsf{0}$$

$$\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$$

Compute "double of 3 is 6":

$$\begin{array}{ll} & \mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{s}(\mathsf{0})))))) \end{array}$$

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) $\mathcal{R}$)

$$\mathsf{double}(\mathsf{0}) \longrightarrow \mathsf{0}$$
$$\mathsf{double}(\mathsf{s}(x)) \longrightarrow \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$$

Compute "double of 3 is 6":

$$\begin{array}{rl} & \mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{s}(\mathsf{0})))))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{0}))))))) \end{array}$$

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) $\mathcal{R}$)

$$\mathsf{double}(\mathsf{0}) \to \mathsf{0}$$

$$\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$$

Compute "double of 3 is 6":

$$\begin{array}{ll} & \mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{s}(\mathsf{0})))))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{0}))))))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))))) \end{array}$$

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) $\mathcal{R}$)

$$\mathsf{double}(\mathsf{0}) \to \mathsf{0}$$

$$\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$$

Compute "double of 3 is 6":

$$\begin{array}{rl} & \mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{s}(\mathsf{0})))))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{double}(\mathsf{0}))))))) \\ \to_{\mathcal{R}} & \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))))) \end{array}$$

in 4 steps with $\to_{\mathcal{R}}$

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) $\mathcal{R}$)

$$\mathsf{double}(\mathsf{0}) \to \mathsf{0}$$

$$\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$$

Compute "double of 3 is 6":

$$\begin{aligned} & \mathsf{double}(\mathsf{s}^3(\mathsf{0})) \\ \to_{\mathcal{R}} \quad & \mathsf{s}^2(\mathsf{double}(\mathsf{s}^2(\mathsf{0}))) \\ \to_{\mathcal{R}} \quad & \mathsf{s}^4(\mathsf{double}(\mathsf{s}(\mathsf{0}))) \\ \to_{\mathcal{R}} \quad & \mathsf{s}^6(\mathsf{double}(\mathsf{0})) \\ \to_{\mathcal{R}} \quad & \mathsf{s}^6(\mathsf{0}) \end{aligned}$$

in 4 steps with $\to_{\mathcal{R}}$

What is *Complexity* of Term Rewriting?

Given: TRS $\mathcal{R}$ (e.g., $\{\ \mathsf{double}(\mathsf{0}) \to \mathsf{0},\ \ \mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))\ \}$)

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)

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 $\Theta(n)$?

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!**

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!**

$$\mathsf{double}(\mathsf{s}^{n-2}(\mathsf{0})) \to_{\mathcal{R}}^{n-1} \mathsf{s}^{2n-4}(\mathsf{0})$$

What is *Complexity* of Term Rewriting?

Given: TRS $\mathcal{R}$ (e.g., $\{$ $\mathsf{double}(\mathsf{0}) \to \mathsf{0}$, $\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$ $\}$)

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!**

$$\mathsf{double}(\mathsf{s}^{n-2}(\mathsf{0})) \to_{\mathcal{R}}^{n-1} \mathsf{s}^{2n-4}(\mathsf{0})$$

- **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) $\to$ 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!**

$$\mathsf{double}(\mathsf{s}^{n-2}(\mathsf{0})) \to_{\mathcal{R}}^{n-1} \mathsf{s}^{2n-4}(\mathsf{0})$$

- **basic terms** $f(t_1, \ldots, t_n)$ with t_i constructor terms allow only n steps
- **runtime complexity** $\mathrm{rc}_{\mathcal{R}}(n)$: basic terms as start terms

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!**

$$\mathsf{double}(\mathsf{s}^{n-2}(\mathsf{0})) \to_{\mathcal{R}}^{n-1} \mathsf{s}^{2n-4}(\mathsf{0})$$

- **basic terms** $f(t_1, \ldots, t_n)$ with t_i constructor terms allow only n steps
- **runtime complexity** $\mathrm{rc}_{\mathcal{R}}(n)$: basic terms as start terms
- $\mathrm{rc}_{\mathcal{R}}(n)$ for **program analysis**

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!**

$$\mathsf{double}(\mathsf{s}^{n-2}(\mathsf{0})) \to_{\mathcal{R}}^{n-1} \mathsf{s}^{2n-4}(\mathsf{0})$$

- **basic terms** $f(t_1, \ldots, t_n)$ with t_i constructor terms allow only n steps
- **runtime complexity** $\mathrm{rc}_{\mathcal{R}}(n)$: basic terms as start terms
- $\mathrm{rc}_{\mathcal{R}}(n)$ for **program analysis**

(2) **No!**

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!**

$$\mathsf{double}(\mathsf{s}^{n-2}(0)) \to_{\mathcal{R}}^{n-1} \mathsf{s}^{2n-4}(0)$$

- **basic terms** $f(t_1, \ldots, t_n)$ with t_i constructor terms allow only n steps
- **runtime complexity** $\mathrm{rc}_{\mathcal{R}}(n)$: basic terms as start terms
- $\mathrm{rc}_{\mathcal{R}}(n)$ for **program analysis**

(2) **No!**

$$\mathsf{double}^3(\mathsf{s}(0)) \to_{\mathcal{R}}^{2} \mathsf{double}^2(\mathsf{s}^2(0)) \to_{\mathcal{R}}^{3} \mathsf{double}(\mathsf{s}^4(0)) \to_{\mathcal{R}}^{5} \mathsf{s}^8(0) \text{ in 10 steps}$$

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!**

$$\mathsf{double}(\mathsf{s}^{n-2}(\mathsf{0})) \to_{\mathcal{R}}^{n-1} \mathsf{s}^{2n-4}(\mathsf{0})$$

- **basic terms** $f(t_1,\ldots,t_n)$ with t_i constructor terms allow only n steps
- **runtime complexity** $\mathrm{rc}_{\mathcal{R}}(n)$: basic terms as start terms
- $\mathrm{rc}_{\mathcal{R}}(n)$ for **program analysis**

(2) **No!**

$$\mathsf{double}^3(\mathsf{s}(\mathsf{0})) \to_{\mathcal{R}}^{2} \mathsf{double}^2(\mathsf{s}^2(\mathsf{0})) \to_{\mathcal{R}}^{3} \mathsf{double}(\mathsf{s}^4(\mathsf{0})) \to_{\mathcal{R}}^{5} \mathsf{s}^8(\mathsf{0}) \text{ in 10 steps}$$

- $\mathsf{double}^{n-2}(\mathsf{s}(\mathsf{0}))$ allows $\Theta(2^n)$ many steps to $\mathsf{s}^{2^{n-2}}(\mathsf{0})$

What is *Complexity* of Term Rewriting?

Given: TRS $\mathcal{R}$ (e.g., $\{$ $\mathsf{double}(\mathsf{0}) \to \mathsf{0}$, $\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$ $\}$)

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!**

$$\mathsf{double}(\mathsf{s}^{n-2}(\mathsf{0})) \to_{\mathcal{R}}^{n-1} \mathsf{s}^{2n-4}(\mathsf{0})$$

- **basic terms** $f(t_1, \ldots, t_n)$ with t_i constructor terms allow only n steps
- **runtime complexity** $\mathrm{rc}_{\mathcal{R}}(n)$: basic terms as start terms
- $\mathrm{rc}_{\mathcal{R}}(n)$ for **program analysis**

(2) **No!**

$$\mathsf{double}^3(\mathsf{s}(\mathsf{0})) \to_{\mathcal{R}}^{2} \mathsf{double}^2(\mathsf{s}^2(\mathsf{0})) \to_{\mathcal{R}}^{3} \mathsf{double}(\mathsf{s}^4(\mathsf{0})) \to_{\mathcal{R}}^{5} \mathsf{s}^8(\mathsf{0}) \text{ in 10 steps}$$

- $\mathsf{double}^{n-2}(\mathsf{s}(\mathsf{0}))$ allows $\Theta(2^n)$ many steps to $\mathsf{s}^{2^{n-2}}(\mathsf{0})$
- **derivational complexity** $\mathrm{dc}_{\mathcal{R}}(n)$: no restrictions on start terms

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!**

$$\mathsf{double}(\mathsf{s}^{n-2}(\mathsf{0})) \to_{\mathcal{R}}^{n-1} \mathsf{s}^{2n-4}(\mathsf{0})$$

- **basic terms** $f(t_1, \ldots, t_n)$ with t_i constructor terms allow only n steps
- **runtime complexity** $\mathrm{rc}_{\mathcal{R}}(n)$: basic terms as start terms
- $\mathrm{rc}_{\mathcal{R}}(n)$ for **program analysis**

(2) **No!**

$$\mathsf{double}^3(\mathsf{s}(\mathsf{0})) \to_{\mathcal{R}}^{2} \mathsf{double}^2(\mathsf{s}^2(\mathsf{0})) \to_{\mathcal{R}}^{3} \mathsf{double}(\mathsf{s}^4(\mathsf{0})) \to_{\mathcal{R}}^{5} \mathsf{s}^8(\mathsf{0}) \text{ in 10 steps}$$

- $\mathsf{double}^{n-2}(\mathsf{s}(\mathsf{0}))$ allows $\Theta(2^n)$ many steps to $\mathsf{s}^{2^{n-2}}(\mathsf{0})$
- **derivational complexity** $\mathrm{dc}_{\mathcal{R}}(n)$: no restrictions on start terms
- $\mathrm{dc}_{\mathcal{R}}(n)$ 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 $\mathcal{R}_{\mathcal{E}}$

Complexity Analysis for TRSs: Overview

A Short Timeline (1/2)

1989: Derivational complexity introduced, linked to termination proofs[19]

[19] D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA '89

A Short Timeline (1/2)

1989: Derivational complexity introduced, linked to termination proofs[19]

2001: Techniques for polynomial upper complexity bounds[20]

[19] D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA '89

[20] 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[19]

2001: Techniques for polynomial upper complexity bounds[20]

2008: Runtime complexity introduced with first analysis techniques[21]

---

[19] D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA '89

[20] G. Bonfante, A. Cichon, J. Marion, and H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP '01

[21] 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[19]

2001: Techniques for polynomial upper complexity bounds[20]

2008: Runtime complexity introduced with first analysis techniques[21]

2008: First automated tools to find complexity bounds: TcT[22], CaT[23]

---

[19] D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA '89

[20] G. Bonfante, A. Cichon, J. Marion, and H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP '01

[21] N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR '08

[22] M. Avanzini, G. Moser, M. Schaper: *TcT: Tyrolean Complexity Tool*, TACAS '16, `https://tcs-informatik.uibk.ac.at/tools/tct/`

[23] M. Korp, C. Sternagel, H. Zankl, A. Middeldorp: *Tyrolean Termination Tool 2*, RTA '09, `http://cl-informatik.uibk.ac.at/software/cat/`

A Short Timeline (1/2)

1989: Derivational complexity introduced, linked to termination proofs[19]

2001: Techniques for polynomial upper complexity bounds[20]

2008: Runtime complexity introduced with first analysis techniques[21]

2008: First automated tools to find complexity bounds: TcT[22], CaT[23]

2008: First complexity analysis categories in the Termination Competition (termCOMP)

[19] D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA '89

[20] G. Bonfante, A. Cichon, J. Marion, and H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP '01

[21] N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR '08

[22] M. Avanzini, G. Moser, M. Schaper: *TcT: Tyrolean Complexity Tool*, TACAS '16, https://tcs-informatik.uibk.ac.at/tools/tct/

[23] M. Korp, C. Sternagel, H. Zankl, A. Middeldorp: *Tyrolean Termination Tool 2*, RTA '09, http://cl-informatik.uibk.ac.at/software/cat/

A Short Timeline (1/2)

1989: Derivational complexity introduced, linked to termination proofs[19]

2001: Techniques for polynomial upper complexity bounds[20]

2008: Runtime complexity introduced with first analysis techniques[21]

2008: First automated tools to find complexity bounds: TcT[22], CaT[23]

2008: First complexity analysis categories in the Termination Competition (termCOMP)

...

[19] D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA '89

[20] G. Bonfante, A. Cichon, J. Marion, and H. Touzet: *Algorithms with polynomial interpretation termination proof*, JFP '01

[21] N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR '08

[22] M. Avanzini, G. Moser, M. Schaper: *TcT: Tyrolean Complexity Tool*, TACAS '16, https://tcs-informatik.uibk.ac.at/tools/tct/

[23] M. Korp, C. Sternagel, H. Zankl, A. Middeldorp: *Tyrolean Termination Tool 2*, RTA '09, http://cl-informatik.uibk.ac.at/software/cat/

Some Definitions

Definition (Derivation Height dh)

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a relation $\longrightarrow$, the **derivation height** is:

$$\mathrm{dh}(t, \longrightarrow) \;=\; \sup \{\, n \mid \exists t'.\, t \longrightarrow^{n} t' \,\}$$

If t starts an infinite $\longrightarrow$-sequence, we set $\mathrm{dh}(t, \longrightarrow) = \omega$.

Some Definitions

Definition (Derivation Height dh)

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a relation $\longrightarrow$, the **derivation height** is:

$$\mathrm{dh}(t, \longrightarrow) \;=\; \sup \{\, n \mid \exists t'.\, t \longrightarrow^{n} t' \,\}$$

If t starts an infinite $\longrightarrow$-sequence, we set $\mathrm{dh}(t, \longrightarrow) = \omega$.

$\mathrm{dh}(t, \longrightarrow)$: length of the longest $\longrightarrow$-sequence from t.

Some Definitions

Definition (Derivation Height dh)

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a relation $\longrightarrow$, the **derivation height** is:

$$\mathrm{dh}(t, \longrightarrow) \;=\; \sup \{\, n \mid \exists t'.\, t \longrightarrow^{n} t' \,\}$$

If t starts an infinite $\longrightarrow$-sequence, we set $\mathrm{dh}(t, \longrightarrow) = \omega$.

$\mathrm{dh}(t, \longrightarrow)$: length of the longest $\longrightarrow$-sequence from t.

Example: $\mathrm{dh}(\,\mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))),\ \longrightarrow_{\mathcal{R}}\,) = 4$

Some Definitions

Definition (Derivation Height dh)

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a relation $\longrightarrow$, the **derivation height** is:

$$\mathrm{dh}(t, \longrightarrow) \;=\; \sup \{\, n \mid \exists t'.\, t \longrightarrow^n t' \,\}$$

If t starts an infinite $\longrightarrow$-sequence, we set $\mathrm{dh}(t, \longrightarrow) = \omega$.

$\mathrm{dh}(t, \longrightarrow)$: length of the longest $\longrightarrow$-sequence from t.

Example: $\mathrm{dh}(\,\mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))),\ \longrightarrow_{\mathcal{R}}\,) = 4$

Definition (Derivational Complexity dc)

For a TRS $\mathcal{R}$, the **derivational complexity** is:

$$\mathrm{dc}_{\mathcal{R}}(n) \;=\; \sup \{\, \mathrm{dh}(t, \longrightarrow_{\mathcal{R}}) \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{V}), |t| \le n \,\}$$

Some Definitions

Definition (Derivation Height dh)

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a relation $\to$, the **derivation height** is:

$$\mathrm{dh}(t, \to) \;=\; \sup \{\, n \mid \exists t'.\, t \to^{n} t' \,\}$$

If t starts an infinite $\to$-sequence, we set $\mathrm{dh}(t, \to) = \omega$.

$\mathrm{dh}(t, \to)$: length of the longest $\to$-sequence from t.

Example: $\mathrm{dh}(\,\mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))),\ \to_{\mathcal{R}}\,) = 4$

Definition (Derivational Complexity dc)

For a TRS $\mathcal{R}$, the **derivational complexity** is:

$$\mathrm{dc}_{\mathcal{R}}(n) \;=\; \sup \{\, \mathrm{dh}(t, \to_{\mathcal{R}}) \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{V}), |t| \leq n \,\}$$

$\mathrm{dc}_{\mathcal{R}}(n)$: length of the longest $\to_{\mathcal{R}}$-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 $\longrightarrow$, the **derivation height** is:

$$\mathrm{dh}(t, \longrightarrow) \;=\; \sup \{ \, n \mid \exists t'. \, t \longrightarrow^{n} t' \, \}$$

If t starts an infinite $\longrightarrow$-sequence, we set $\mathrm{dh}(t, \longrightarrow) = \omega$.

$\mathrm{dh}(t, \longrightarrow)$: length of the longest $\longrightarrow$-sequence from t.

Example: $\mathrm{dh}(\, \mathsf{double}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))), \; \longrightarrow_{\mathcal{R}} \,) = 4$

Definition (Derivational Complexity dc)

For a TRS $\mathcal{R}$, the **derivational complexity** is:

$$\mathrm{dc}_{\mathcal{R}}(n) \;=\; \sup \{ \, \mathrm{dh}(t, \longrightarrow_{\mathcal{R}}) \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{V}), |t| \leq n \, \}$$

$\mathrm{dc}_{\mathcal{R}}(n)$: length of the longest $\longrightarrow_{\mathcal{R}}$-sequence from a term of size at most n

Example: For $\mathcal{R}$ for double, we have $\mathrm{dc}_{\mathcal{R}}(n) \in \Theta(2^n)$.

Upper Bounds

The Bad News for automation:

Upper Bounds

The Bad News for automation:

For a given TRS $\mathcal{R}$, the following questions are undecidable:

- $\mathrm{dc}_{\mathcal{R}}(n) = \omega$ for some n? ($\longrightarrow$ termination!)

Upper Bounds

The Bad News for automation:

For a given TRS $\mathcal{R}$, the following questions are undecidable:

- $\mathrm{dc}_{\mathcal{R}}(n) = \omega$ for some n? ($\longrightarrow$ termination!)
- $\mathrm{dc}_{\mathcal{R}}(n)$ polynomially bounded?[24]

[24] A. Schnabl and J. G. Simonsen: *The exact hardness of deciding derivational and runtime complexity*, CSL '11

Upper Bounds

The Bad News for automation:

For a given TRS $\mathcal{R}$, the following questions are undecidable:

- $\mathrm{dc}_{\mathcal{R}}(n) = \omega$ for some n? ($\longrightarrow$ termination!)
- $\mathrm{dc}_{\mathcal{R}}(n)$ polynomially bounded?[24]

Goal: find **approximations** for derivational complexity

Initial focus: find upper bounds

$$\mathrm{dc}_{\mathcal{R}}(n) \in \mathcal{O}(...)$$

[24] A. Schnabl and J. G. Simonsen: *The exact hardness of deciding derivational and runtime complexity*, CSL '11

Derivational Complexity from Polynomial Interpretations (1/2)

Example (double)

$$\begin{aligned} \mathsf{double}(\mathsf{0}) &\to \mathsf{0} \\ \mathsf{double}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \end{aligned}$$

Derivational Complexity from Polynomial Interpretations (1/2)

Example (double)

$$\begin{aligned} \mathsf{double}(\mathsf{0}) &\succ \mathsf{0} \\ \mathsf{double}(\mathsf{s}(x)) &\succ \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \end{aligned}$$

Show $\mathrm{dc}_{\mathcal{R}}(n) < \omega$ by **termination proof** with reduction order $\succ$ on terms.

Derivational Complexity from Polynomial Interpretations (1/2)

Example (double)

$$\begin{aligned}\mathsf{double}(\mathsf{0}) &\succ \mathsf{0}\\ \mathsf{double}(\mathsf{s}(x)) &\succ \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))\end{aligned}$$

Show $\mathrm{dc}_{\mathcal{R}}(n) < \omega$ by **termination proof** with reduction order $\succ$ on terms.
Get $\succ$ via **polynomial interpretation**[25] $[\,\cdot\,]$ over $\mathbb{N}$: $\ell \succ r \iff [\ell] \succ [r]$

[25] D. Lankford: *Canonical algebraic simplification in computational logic*, U Texas '75

Example (double)

$$\begin{aligned} \mathsf{double}(\mathsf{0}) &\succ \mathsf{0} \\ \mathsf{double}(\mathsf{s}(x)) &\succ \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \end{aligned}$$

Show $\mathrm{dc}_{\mathcal{R}}(n) < \omega$ 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: $[\mathsf{double}](x) = 3 \cdot x, \quad [\mathsf{s}](x) = x + 1, \quad [\mathsf{0}] = 1$

[25] D. Lankford: *Canonical algebraic simplification in computational logic*, U Texas '75

Derivational Complexity from Polynomial Interpretations (1/2)

Example (double)

$$\mathsf{double}(\mathsf{0}) \succ \mathsf{0}$$
$$\mathsf{double}(\mathsf{s}(x)) \succ \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$$

Show $\mathrm{dc}_{\mathcal{R}}(n) < \omega$ 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: $[\mathsf{double}](x) = 3 \cdot x$, $[\mathsf{s}](x) = x + 1$, $[\mathsf{0}] = 1$

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)

double(0)	$\succ$	0	3	$>$	1
double(s(x))	$\succ$	s(s(double(x)))	$3 \cdot x + 3$	$>$	$3 \cdot x + 2$

Show $\mathrm{dc}_{\mathcal{R}}(n) < \omega$ 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: $[\mathsf{double}](x) = 3 \cdot x$, $[\mathsf{s}](x) = x + 1$, $[\mathsf{0}] = 1$

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)

$$\begin{array}{rcl|rcl} \mathsf{double}(\mathsf{0}) & \succ & \mathsf{0} & 3 & > & 1 \\ \mathsf{double}(\mathsf{s}(x)) & \succ & \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) & 3 \cdot x + 3 & > & 3 \cdot x + 2 \end{array}$$

Show $\mathrm{dc}_{\mathcal{R}}(n) < \omega$ 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: $[\mathsf{double}](x) = 3 \cdot x,\quad [\mathsf{s}](x) = x + 1,\quad [\mathsf{0}] = 1$

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

Derivational Complexity from Polynomial Interpretations (2/2)

Example (double)

$$\begin{aligned} \mathsf{double}(\mathsf{0}) &\succ \mathsf{0} \\ \mathsf{double}(\mathsf{s}(x)) &\succ \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \end{aligned} \qquad \begin{aligned} 3 &> 1 \\ 3 \cdot x + 3 &> 3 \cdot x + 2 \end{aligned}$$

Example: $[\mathsf{double}](x) = 3 \cdot x$, $[\mathsf{s}](x) = x + 1$, $[\mathsf{0}] = 1$

This proves more than just termination...

Derivational Complexity from Polynomial Interpretations (2/2)

Example (double)

$\mathsf{double}(\mathsf{0}) \succ \mathsf{0}$		$3 > 1$	
$\mathsf{double}(\mathsf{s}(x)) \succ \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))$		$3 \cdot x + 3 > 3 \cdot x + 2$	

Example: $[\mathsf{double}](x) = 3 \cdot x, \quad [\mathsf{s}](x) = x + 1, \quad [\mathsf{0}] = 1$

This proves more than just termination...

Theorem (Upper bounds for $\mathrm{dc}_{\mathcal{R}}(n)$ from polynomial interpretations[28])

- *Termination proof for TRS $\mathcal{R}$ with **polynomial** interpretation*

$$\Rightarrow \mathrm{dc}_{\mathcal{R}}(n) \in 2^{2^{\mathcal{O}(n)}}$$

[28] D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA '89

Derivational Complexity from Polynomial Interpretations (2/2)

Example (double)

double(0)	$\succ$	0	3	$>$	1
double(s(x))	$\succ$	s(s(double(x)))	$3 \cdot x + 3$	$>$	$3 \cdot x + 2$

Example: [double]$(x) = 3 \cdot x$, [s]$(x) = x + 1$, [0] $= 1$

This proves more than just termination...

Theorem (Upper bounds for $\mathrm{dc}_{\mathcal{R}}(n)$ from polynomial interpretations[28])

- *Termination proof for TRS $\mathcal{R}$ with **polynomial** interpretation*

  $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n) \in 2^{2^{\mathcal{O}(n)}}$
- *Termination proof for TRS $\mathcal{R}$ with **linear polynomial** interpretation*

  $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n) \in 2^{\mathcal{O}(n)}$

[28] D. Hofbauer, C. Lautemann: *Termination proofs and the length of derivations*, RTA '89

Derivational Complexity from Termination Proofs (1/2)

Termination proof for TRS $\mathcal{R}$ with ...

- matchbounds[29] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n) \in \mathcal{O}(n)$
- arctic matrix interpretations[30] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n) \in \mathcal{O}(n)$

[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

Derivational Complexity from Termination Proofs (1/2)

Termination proof for TRS $\mathcal{R}$ with ...

- matchbounds[29] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n) \in \mathcal{O}(n)$
- arctic matrix interpretations[30] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n) \in \mathcal{O}(n)$
- triangular matrix interpretation[31] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ is at most polynomial
- matrix interpretation of spectral radius[32] ≤ 1 $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ 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] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n) \in \mathcal{O}(n)$
- arctic matrix interpretations[30] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n) \in \mathcal{O}(n)$
- triangular matrix interpretation[31] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ is at most polynomial
- matrix interpretation of spectral radius[32] ≤ 1 $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ is at most polynomial
- standard matrix interpretation[33] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ 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

Derivational Complexity from Termination Proofs (2/2)

Termination proof for TRS $\mathcal{R}$ with ...

- Lexicographic Path Order[34] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ 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

Derivational Complexity from Termination Proofs (2/2)

Termination proof for TRS $\mathcal{R}$ with ...

- Lexicographic Path Order[34] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ is at most multiple recursive[35]
- Dependency Pairs method[36] with dependency graphs and usable rules $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ 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

Derivational Complexity from Termination Proofs (2/2)

Termination proof for TRS $\mathcal{R}$ with ...

- Lexicographic Path Order[34] $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ is at most multiple recursive[35]
- Dependency Pairs method[36] with dependency graphs and usable rules $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ is at most primitive recursive[37]
- Dependency Pairs framework[38][39] with dependency graphs, reduction pairs, subterm criterion $\Rightarrow \mathrm{dc}_{\mathcal{R}}(n)$ 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

Runtime Complexity

- So far: upper bounds for derivational complexity

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**: $\mathsf{double}(\mathsf{s}^n(\mathsf{0}))$

Runtime Complexity

- So far: upper bounds for derivational complexity
- But: derivational complexity counter-intuitive, often infeasible
- Wanted: complexity of evaluation of double **on data**: $\mathsf{double}(\mathsf{s}^n(0))$

Definition (Basic Term[41])

For defined symbols $\mathcal{D}$ and constructor symbols $\mathcal{C}$, the term

$$f(t_1, \ldots, t_n)$$

is in the set $\mathcal{T}_{\mathrm{basic}}$ of **basic terms** iff $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$.

[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**: $\mathsf{double}(\mathsf{s}^n(\mathsf{0}))$

Definition (Basic Term[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[41])

For a TRS $\mathcal{R}$, the **runtime complexity** is:

$$\mathrm{rc}_{\mathcal{R}}(n) \;=\; \sup \{ \, \mathrm{dh}(t, \to_{\mathcal{R}}) \mid t \in \mathcal{T}_{\text{basic}}, |t| \leq n \, \}$$

[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[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[41])

For a TRS $\mathcal{R}$, the **runtime complexity** is:

$$\mathrm{rc}_{\mathcal{R}}(n) \;=\; \sup \{ \, \mathrm{dh}(t, \to_{\mathcal{R}}) \mid t \in \mathcal{T}_{\text{basic}}, |t| \leq n \, \}$$

$\mathrm{rc}_{\mathcal{R}}(n)$: like derivational complexity... but for basic terms only!

[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:[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 $c \in \mathbb{N}$.

[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 $p(x_1, \dots, x_n) = x_1 + \cdots + x_n + a$ for some $a \in \mathbb{N}$.
- Polynomial interpretation $[\,\cdot\,]$ is **restricted** iff for all constructor symbols f, $[f](x_1, \dots, x_n)$ is strongly linear.

Idea: $[t] \leq c \cdot |t|$ for fixed $c \in \mathbb{N}$.

Theorem (Upper bounds for $\mathrm{rc}_\mathcal{R}(n)$ from restricted interpretations)

Termination proof for TRS $\mathcal{R}$ with ***restricted*** *interpretation $[\,\cdot\,]$ of degree at most d for $[f]$*

$$\Rightarrow \mathrm{rc}_\mathcal{R}(n) \in \mathcal{O}(n^d)$$

[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 $\mathrm{rc}_{\mathcal{R}}(n)$ from restricted interpretations)

*Termination proof for TRS $\mathcal{R}$ with **restricted** interpretation $[\,\cdot\,]$ of degree at most d for $[f]$*
$\Rightarrow \mathrm{rc}_{\mathcal{R}}(n) \in \mathcal{O}(n^d)$

Example: $[\mathsf{double}](x) = 3 \cdot x$, $[\mathsf{s}](x) = x + 1$, $[\mathsf{0}] = 1$ is restricted, degree 1
$\Rightarrow \mathrm{rc}_{\mathcal{R}}(n) \in \mathcal{O}(n)$ for TRS $\mathcal{R}$ 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 ($\approx$ call-by-value)

Example (reverse)

$\mathsf{app}(\mathsf{nil}, y) \to y$	$\mathsf{app}(\mathsf{add}(n, x), y) \to \mathsf{add}(n, \mathsf{app}(x, y))$
$\mathsf{reverse}(\mathsf{nil}) \to \mathsf{nil}$	$\mathsf{reverse}(\mathsf{add}(n, x)) \to \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil}))$

Dependency Tuples for *Innermost* Runtime Complexity irc

Here: innermost rewriting ($\approx$ call-by-value)

Example (reverse)

$$\begin{array}{rcl|rcl} \mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\ \mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \end{array}$$

For rule $\ell \to r$, eval of ℓ 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)

$$\begin{array}{rcl|rcl} \mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\ \mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \end{array}$$

For rule $\ell \to r$, eval of ℓ costs 1 + eval of all function calls in r **together**:

Example (Dependency Tuples[43] for reverse)

$$\begin{array}{rcl} \mathsf{app}^\sharp(\mathsf{nil}, y) & \to & \mathsf{Com}_0 \\ \mathsf{app}^\sharp(\mathsf{add}(n, x), y) & \to & \mathsf{Com}_1(\mathsf{app}^\sharp(x, y)) \\ \mathsf{reverse}^\sharp(\mathsf{nil}) & \to & \mathsf{Com}_0 \\ \mathsf{reverse}^\sharp(\mathsf{add}(n, x)) & \to & \mathsf{Com}_2(\mathsf{app}^\sharp(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})), \mathsf{reverse}^\sharp(x)) \end{array}$$

- Function calls to count marked with $\sharp$
- Compound symbols 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)

$$
\begin{aligned}
\mathsf{app}^\sharp(\mathsf{nil}, y) &\rightarrow \mathsf{Com}_0 \\
\mathsf{app}^\sharp(\mathsf{add}(n, x), y) &\rightarrow \mathsf{Com}_1(\mathsf{app}^\sharp(x, y)) \\
\mathsf{reverse}^\sharp(\mathsf{nil}) &\rightarrow \mathsf{Com}_0 \\
\mathsf{reverse}^\sharp(\mathsf{add}(n, x)) &\rightarrow \mathsf{Com}_2(\mathsf{app}^\sharp(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})), \mathsf{reverse}^\sharp(x))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{app}(\mathsf{nil}, y) &\rightarrow y & \mathsf{app}(\mathsf{add}(n, x), y) &\rightarrow \mathsf{add}(n, \mathsf{app}(x, y)) \\
\mathsf{reverse}(\mathsf{nil}) &\rightarrow \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) &\rightarrow \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil}))
\end{aligned}
$$

Polynomial Interpretations for Dependency Tuples

Example (reverse, Dependency Tuples for reverse)

$$
\begin{aligned}
\mathsf{app}^\sharp(\mathsf{nil}, y) &\to \mathsf{Com}_0 \\
\mathsf{app}^\sharp(\mathsf{add}(n, x), y) &\to \mathsf{Com}_1(\mathsf{app}^\sharp(x, y)) \\
\mathsf{reverse}^\sharp(\mathsf{nil}) &\to \mathsf{Com}_0 \\
\mathsf{reverse}^\sharp(\mathsf{add}(n, x)) &\to \mathsf{Com}_2(\mathsf{app}^\sharp(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})), \mathsf{reverse}^\sharp(x))
\end{aligned}
$$

$\mathsf{app}(\mathsf{nil}, y) \to y$	$\mathsf{app}(\mathsf{add}(n, x), y) \to \mathsf{add}(n, \mathsf{app}(x, y))$
$\mathsf{reverse}(\mathsf{nil}) \to \mathsf{nil}$	$\mathsf{reverse}(\mathsf{add}(n, x)) \to \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil}))$

Use interpretation $[\,\cdot\,]$ with $[\mathsf{Com}_k](x_1, \ldots, x_k) = x_1 + \cdots + x_k$ and

$[\mathsf{nil}] = 0$	$[\mathsf{add}](x_1, x_2) = x_2 + 1$	($\leq$ restricted interpretation)
$[\mathsf{app}](x_1, x_2) = x_1 + x_2$	$[\mathsf{reverse}](x_1) = x_1$	(bounds helper function's result size)
$[\mathsf{app}^\sharp](x_1, x_2) = x_1 + 1$	$[\mathsf{reverse}^\sharp](x_1) = x_1^2 + x_1 + 1$	(complexity of function)

to show $[\ell] \geq [r]$ for all rules and $[\ell] \geq 1 + [r]$ for all Dependency Tuples

Maximum degree of $[f^\sharp]$ is 2 $\Rightarrow \mathrm{irc}_{\mathcal{R}}(n) \in \mathcal{O}(n^2)$

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]

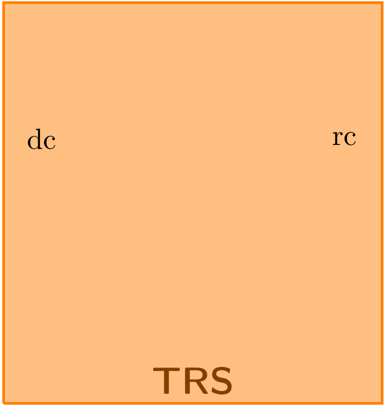
[44] N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR '08

Related Techniques

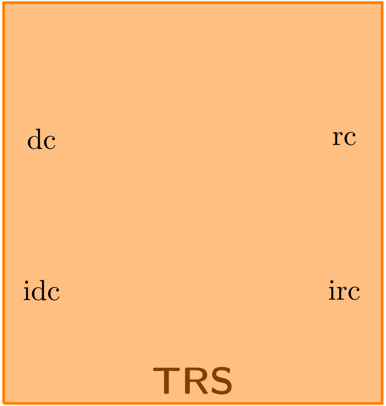
- Dependency Tuples are an adaptation of Dependency Pairs (DPs) from termination analysis to complexity analysis, allow for **incremental** complexity proofs with several techniques
- Further adaptation of DPs (incomparable): Weak (Innermost) Dependency Pairs for (innermost) runtime complexity[44]
- Extensions by polynomial path orders[45], usable replacement maps[46], a combination framework for complexity analysis[47], ...

[44] N. Hirokawa, G. Moser: *Automated complexity analysis based on the dependency pair method*, IJCAR '08
[45] M. Avanzini, G. Moser: *Dependency pairs and polynomial path orders*, RTA '09
[46] N. Hirokawa, G. Moser: *Automated complexity analysis based on context-sensitive rewriting*, RTA-TLCA '14
[47] M. Avanzini, G. Moser: *A combination framework for complexity*, IC '16

A Landscape of Complexity Properties and Transformations

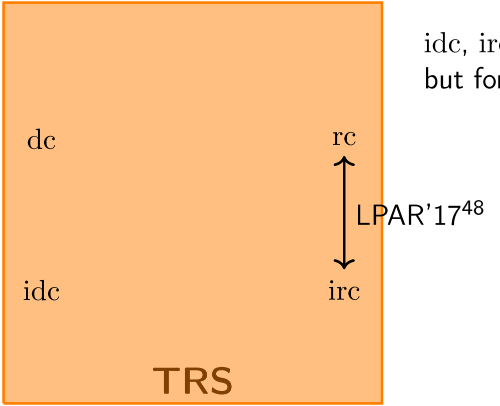


A Landscape of Complexity Properties and Transformations



idc, irc: like dc, rc,
but for *innermost* rewriting

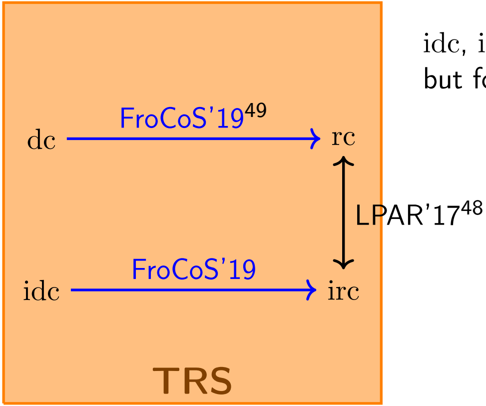
A Landscape of Complexity Properties and Transformations



idc, irc: like dc, rc,
but for *innermost* rewriting

[48] 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

[48] F. Frohn, J. Giesl: *Analyzing runtime complexity via innermost runtime complexity*, LPAR '17
[49] C. Fuhs: *Transforming Derivational Complexity of Term Rewriting to Runtime Complexity*, FroCoS '19

Transforming Derivational Complexity to Runtime Complexity

The big picture:

- **Have:** Tool for automated analysis of runtime complexity $\mathrm{rc}_{\mathcal{R}}$

Transforming Derivational Complexity to Runtime Complexity

The big picture:

- **Have:** Tool for automated analysis of runtime complexity $\mathrm{rc}_{\mathcal{R}}$
- **Want:** Tool for automated analysis of derivational complexity $\mathrm{dc}_{\mathcal{R}}$

Transforming Derivational Complexity to Runtime Complexity

The big picture:

- **Have:** Tool for automated analysis of runtime complexity $\mathrm{rc}_{\mathcal{R}}$
- **Want:** Tool for automated analysis of derivational complexity $\mathrm{dc}_{\mathcal{R}}$
- **Idea:**

"$\mathrm{rc}_{\mathcal{R}}$ analysis tool + transformation on TRS $\mathcal{R}$ = $\mathrm{dc}_{\mathcal{R}}$ analysis tool"

Transforming Derivational Complexity to Runtime Complexity

The big picture:

- **Have:** Tool for automated analysis of runtime complexity $\mathrm{rc}_{\mathcal{R}}$
- **Want:** Tool for automated analysis of derivational complexity $\mathrm{dc}_{\mathcal{R}}$
- **Idea:**

  "$\mathrm{rc}_{\mathcal{R}}$ analysis tool + transformation on TRS $\mathcal{R}$ = $\mathrm{dc}_{\mathcal{R}}$ analysis tool"

- **Benefits:**
  - Get analysis of derivational complexity "for free"
  - Progress in runtime complexity analysis automatically improves derivational complexity analysis

From dc to rc: Results

- program transformation such that runtime complexity of transformed TRS is **identical** to derivational complexity of original TRS

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

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[50] relative to state of the art TcT

[50] Termination Problem DataBase, standard benchmark source for annual Termination Competition (termCOMP) with 1000s of problems, `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** $\mathcal{G}$ 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** $\mathcal{G}$ to reconstruct arbitrary term with f from basic term with c_f

Represent

$$t = \mathsf{double}(\mathsf{double}(\mathsf{double}(\mathsf{s}(\mathsf{0}))))$$

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** $\mathcal{G}$ to reconstruct arbitrary term with f from basic term with c_f

Represent

$$t = \mathsf{double}(\mathsf{double}(\mathsf{double}(\mathsf{s}(\mathsf{0}))))$$

by **basic variant**

$$\mathrm{bv}(t) = \mathsf{enc}_{\mathsf{double}}(\mathsf{c}_{\mathsf{double}}(\mathsf{c}_{\mathsf{double}}(\mathsf{s}(\mathsf{0}))))$$

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** $\mathcal{G}$ to reconstruct arbitrary term with f from basic term with c_f

Represent

$$t = \mathsf{double}(\mathsf{double}(\mathsf{double}(\mathsf{s}(\mathsf{0}))))$$

by **basic variant**

$$\mathrm{bv}(t) = \mathsf{enc}_{\mathsf{double}}(\mathsf{c}_{\mathsf{double}}(\mathsf{c}_{\mathsf{double}}(\mathsf{s}(\mathsf{0}))))$$

Example (Generator rules $\mathcal{G}$)

$$\begin{aligned}
\mathsf{enc}_{\mathsf{double}}(x) &\to \mathsf{double}(\mathsf{argenc}(x))\\
\mathsf{enc}_{\mathsf{0}} &\to \mathsf{0}\\
\mathsf{enc}_{\mathsf{s}}(x) &\to \mathsf{s}(\mathsf{argenc}(x))\\
\mathsf{argenc}(\mathsf{c}_{\mathsf{double}}(x)) &\to \mathsf{double}(\mathsf{argenc}(x))\\
\mathsf{argenc}(\mathsf{0}) &\to \mathsf{0}\\
\mathsf{argenc}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{argenc}(x))
\end{aligned}$$

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** $\mathcal{G}$ to reconstruct arbitrary term with f from basic term with c_f

Represent

$$t = \mathsf{double}(\mathsf{double}(\mathsf{double}(\mathsf{s}(\mathsf{0}))))$$

by **basic variant**

$$\mathrm{bv}(t) = \mathsf{enc}_{\mathsf{double}}(\mathsf{c}_{\mathsf{double}}(\mathsf{c}_{\mathsf{double}}(\mathsf{s}(\mathsf{0}))))$$

Then:

- $\mathrm{bv}(t)$ is **basic** term, size $|t|$

Example (Generator rules $\mathcal{G}$)

$$\begin{aligned}
\mathsf{enc}_{\mathsf{double}}(x) &\to \mathsf{double}(\mathsf{argenc}(x))\\
\mathsf{enc}_{\mathsf{0}} &\to \mathsf{0}\\
\mathsf{enc}_{\mathsf{s}}(x) &\to \mathsf{s}(\mathsf{argenc}(x))\\
\mathsf{argenc}(\mathsf{c}_{\mathsf{double}}(x)) &\to \mathsf{double}(\mathsf{argenc}(x))\\
\mathsf{argenc}(\mathsf{0}) &\to \mathsf{0}\\
\mathsf{argenc}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{argenc}(x))
\end{aligned}$$

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** $\mathcal{G}$ to reconstruct arbitrary term with f from basic term with c_f

Represent

$$t = \mathsf{double}(\mathsf{double}(\mathsf{double}(\mathsf{s}(\mathsf{0}))))$$

by **basic variant**

$$\mathrm{bv}(t) = \mathsf{enc}_{\mathsf{double}}(\mathsf{c}_{\mathsf{double}}(\mathsf{c}_{\mathsf{double}}(\mathsf{s}(\mathsf{0}))))$$

Then:

- $\mathrm{bv}(t)$ is **basic** term, size $|t|$
- $\mathrm{bv}(t) \to_{\mathcal{G}}^{*} t$

Example (Generator rules $\mathcal{G}$)

$$\begin{aligned}
\mathsf{enc}_{\mathsf{double}}(x) &\to \mathsf{double}(\mathsf{argenc}(x)) \\
\mathsf{enc}_{\mathsf{0}} &\to \mathsf{0} \\
\mathsf{enc}_{\mathsf{s}}(x) &\to \mathsf{s}(\mathsf{argenc}(x)) \\
\mathsf{argenc}(\mathsf{c}_{\mathsf{double}}(x)) &\to \mathsf{double}(\mathsf{argenc}(x)) \\
\mathsf{argenc}(\mathsf{0}) &\to \mathsf{0} \\
\mathsf{argenc}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{argenc}(x))
\end{aligned}$$

General Case: Relative Rewriting

Issue:

- $\longrightarrow_{\mathcal{R} \cup \mathcal{G}}$ has extra rewrite steps not present in $\longrightarrow_{\mathcal{R}}$
- may change complexity

General Case: Relative Rewriting

Issue:

- $\to_{\mathcal{R} \cup \mathcal{G}}$ has extra rewrite steps not present in $\to_{\mathcal{R}}$
- may change complexity

Solution:

- add $\mathcal{G}$ as **relative** rewrite rules:
  $\to_{\mathcal{G}}$ steps are **not counted** for complexity analysis!
- transform $\mathcal{R}$ to $\mathcal{R}/\mathcal{G}$ ($\to_{\mathcal{R}}$ steps are counted, $\to_{\mathcal{G}}$ steps are not)

General Case: Relative Rewriting

Issue:

- $\to_{\mathcal{R} \cup \mathcal{G}}$ has extra rewrite steps not present in $\to_{\mathcal{R}}$
- may change complexity

Solution:

- add $\mathcal{G}$ as **relative** rewrite rules:
  $\to_{\mathcal{G}}$ steps are **not counted** for complexity analysis!
- transform $\mathcal{R}$ to $\mathcal{R}/\mathcal{G}$ ($\to_{\mathcal{R}}$ steps are counted, $\to_{\mathcal{G}}$ steps are not)
- more generally: transform $\mathcal{R}/\mathcal{S}$ to $\mathcal{R}/(\mathcal{S} \cup \mathcal{G})$
  (input may contain relative rules $\mathcal{S}$, too)

From dc to rc: Correctness

Theorem (Derivational Complexity via Runtime Complexity)

Let $\mathcal{R}/\mathcal{S}$ be a relative TRS, let $\mathcal{G}$ be the generator rules for $\mathcal{R}/\mathcal{S}$. Then

1. $\mathrm{dc}_{\mathcal{R}/\mathcal{S}}(n) = \mathrm{rc}_{\mathcal{R}/(\mathcal{S} \cup \mathcal{G})}(n)$ *(arbitrary rewrite strategies)*
2. $\mathrm{idc}_{\mathcal{R}/\mathcal{S}}(n) = \mathrm{irc}_{\mathcal{R}/(\mathcal{S} \cup \mathcal{G})}(n)$ *(innermost rewriting)*

Note: equalities hold also non-asymptotically!

From (i)dc to (i)rc: Experiments

Experiments on TPDB, compare with state of the art in TcT:

- upper bounds idc: both AProVE and TcT with transformation are stronger than standard TcT
- upper bounds dc: TcT stronger than AProVE and TcT with transformation, but AProVE still solves some new examples
- lower bounds idc and dc: heuristics do not seem to benefit much

From (i)dc to (i)rc: Experiments

Experiments on TPDB, compare with state of the art in TcT:

- upper bounds idc: both AProVE and TcT with transformation are stronger than standard TcT
- upper bounds dc: TcT stronger than AProVE and TcT with transformation, but AProVE still solves some new examples
- lower bounds idc and dc: heuristics do not seem to benefit much

$\Rightarrow$ Transformation-based approach should be part of the portfolio of analysis tools for derivational complexity

Derivational Complexity: Applications and Extensions

- **Possible applications**
  - compiler simplifications
  - SMT solver preprocessing

  Start terms may have nested defined symbols, so $\mathrm{dc}_{\mathcal{R}}$ is appropriate

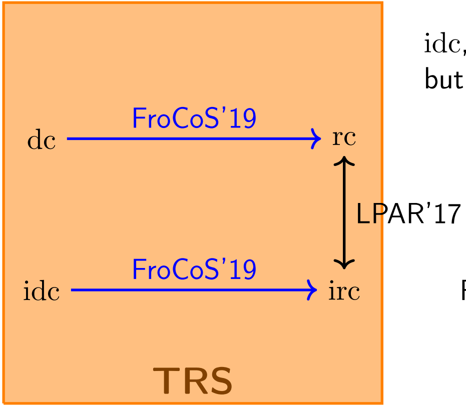
Derivational Complexity: Applications and Extensions

- **Possible applications**
  - compiler simplifications
  - SMT solver preprocessing

  Start terms may have nested defined symbols, so $\mathrm{dc}_{\mathcal{R}}$ is appropriate

- Go **between** derivational and runtime complexity
  - So far: encode *full* term universe $\mathcal{T}$ via basic terms $\mathcal{T}_{\mathrm{basic}}$
  - Generalise: write relative rules to generate **arbitrary** set $\mathcal{U}$ of terms "between" basic and all terms ($\mathcal{T}_{\mathrm{basic}} \subseteq \mathcal{U} \subseteq \mathcal{T}$).

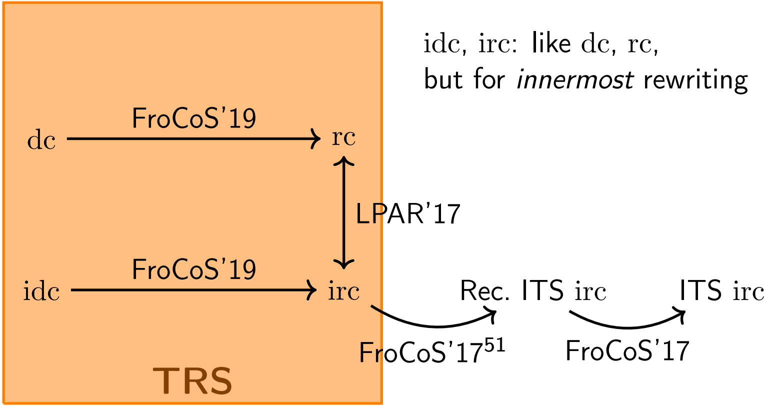
A Landscape of Complexity Properties and Transformations



idc, irc: like dc, rc,
but for *innermost* rewriting

Rec. ITS irc ITS irc

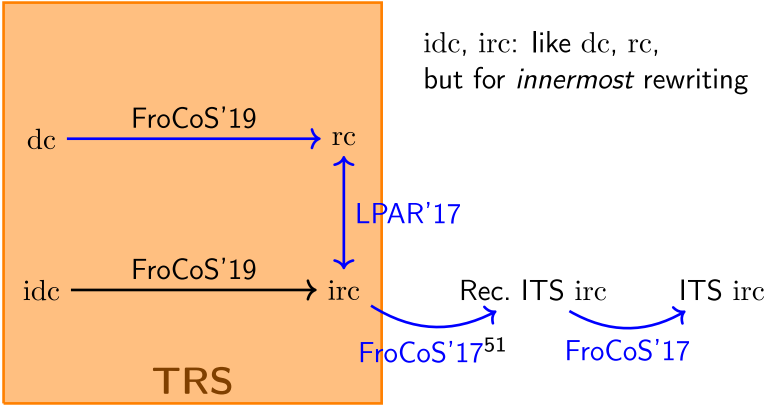
A Landscape of Complexity Properties and Transformations



idc, irc: like dc, rc,
but for *innermost* rewriting

[51] 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

[51] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, J. Giesl: *Complexity analysis for term rewriting by integer transition systems*, FroCoS '17

Derivational_Complexity_Full_Rewriting/AG01/#3.12, TPDB

$$
\begin{array}{rcl|rcl}
\mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\
\mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \\
\mathsf{shuffle}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{shuffle}(\mathsf{add}(n, x)) & \to & \mathsf{add}(n, \mathsf{shuffle}(\mathsf{reverse}(x)))
\end{array}
$$

$$
\begin{array}{rcl|rcl}
\mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\
\mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \\
\mathsf{shuffle}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{shuffle}(\mathsf{add}(n, x)) & \to & \mathsf{add}(n, \mathsf{shuffle}(\mathsf{reverse}(x)))
\end{array}
$$

AProVE finds (tight) upper bound $\mathcal{O}(n^4)$ for $\mathrm{dc}_{\mathcal{R}}$:

Derivational_Complexity_Full_Rewriting/AG01/#3.12, TPDB

$$\begin{array}{rcl|rcl} \mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\ \mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \\ \mathsf{shuffle}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{shuffle}(\mathsf{add}(n, x)) & \to & \mathsf{add}(n, \mathsf{shuffle}(\mathsf{reverse}(x))) \end{array}$$

AProVE finds (tight) upper bound $\mathcal{O}(n^4)$ for $\mathrm{dc}_{\mathcal{R}}$:

1. Add generator rules $\mathcal{G}$, so analyse $\mathrm{rc}_{\mathcal{R}/\mathcal{G}}$ instead (FroCoS'19)

Derivational_Complexity_Full_Rewriting/AG01/#3.12, TPDB

$$\begin{array}{rcl|rcl} \mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\ \mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \\ \mathsf{shuffle}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{shuffle}(\mathsf{add}(n, x)) & \to & \mathsf{add}(n, \mathsf{shuffle}(\mathsf{reverse}(x))) \end{array}$$

AProVE finds (tight) upper bound $\mathcal{O}(n^4)$ for $\mathrm{dc}_{\mathcal{R}}$:

1. Add generator rules $\mathcal{G}$, so analyse $\mathrm{rc}_{\mathcal{R}/\mathcal{G}}$ instead (FroCoS'19)
2. Detect: innermost is worst case here, analyse $\mathrm{irc}_{\mathcal{R}/\mathcal{G}}$ instead (LPAR'17)

$$\begin{array}{rcl|rcl} \mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\ \mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \\ \mathsf{shuffle}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{shuffle}(\mathsf{add}(n, x)) & \to & \mathsf{add}(n, \mathsf{shuffle}(\mathsf{reverse}(x))) \end{array}$$

AProVE finds (tight) upper bound $\mathcal{O}(n^4)$ for $\mathrm{dc}_{\mathcal{R}}$:

1. Add generator rules $\mathcal{G}$, so analyse $\mathrm{rc}_{\mathcal{R}/\mathcal{G}}$ instead (FroCoS'19)
2. Detect: innermost is worst case here, analyse $\mathrm{irc}_{\mathcal{R}/\mathcal{G}}$ instead (LPAR'17)
3. Transform TRS to Recursive Integer Transition System (RITS), analyse complexity of RITS instead (FroCoS'17)

Derivational_Complexity_Full_Rewriting/AG01/#3.12, TPDB

$$\begin{array}{rcl|rcl} \mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\ \mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \\ \mathsf{shuffle}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{shuffle}(\mathsf{add}(n, x)) & \to & \mathsf{add}(n, \mathsf{shuffle}(\mathsf{reverse}(x))) \end{array}$$

AProVE finds (tight) upper bound $\mathcal{O}(n^4)$ for $\mathrm{dc}_{\mathcal{R}}$:

1. Add generator rules $\mathcal{G}$, so analyse $\mathrm{rc}_{\mathcal{R}/\mathcal{G}}$ instead (FroCoS'19)
2. Detect: innermost is worst case here, analyse $\mathrm{irc}_{\mathcal{R}/\mathcal{G}}$ 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

$$\begin{array}{rcl|rcl} \mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\ \mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \\ \mathsf{shuffle}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{shuffle}(\mathsf{add}(n, x)) & \to & \mathsf{add}(n, \mathsf{shuffle}(\mathsf{reverse}(x))) \end{array}$$

AProVE finds (tight) upper bound $\mathcal{O}(n^4)$ for $\mathrm{dc}_{\mathcal{R}}$:

1. Add generator rules $\mathcal{G}$, so analyse $\mathrm{rc}_{\mathcal{R}/\mathcal{G}}$ instead (FroCoS'19)
2. Detect: innermost is worst case here, analyse $\mathrm{irc}_{\mathcal{R}/\mathcal{G}}$ 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 $\mathcal{O}(n^4)$ for RITS complexity carries over to $\mathrm{dc}_{\mathcal{R}}$ of input!

Derivational_Complexity_Full_Rewriting/AG01/#3.12, TPDB

$$\begin{array}{rcl|rcl} \mathsf{app}(\mathsf{nil}, y) & \to & y & \mathsf{app}(\mathsf{add}(n, x), y) & \to & \mathsf{add}(n, \mathsf{app}(x, y)) \\ \mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n, x)) & \to & \mathsf{app}(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) \\ \mathsf{shuffle}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{shuffle}(\mathsf{add}(n, x)) & \to & \mathsf{add}(n, \mathsf{shuffle}(\mathsf{reverse}(x))) \end{array}$$

AProVE finds (tight) upper bound $\mathcal{O}(n^4)$ for $\mathrm{dc}_{\mathcal{R}}$:

1. Add generator rules $\mathcal{G}$, so analyse $\mathrm{rc}_{\mathcal{R}/\mathcal{G}}$ instead (FroCoS'19)
2. Detect: innermost is worst case here, analyse $\mathrm{irc}_{\mathcal{R}/\mathcal{G}}$ 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 $\mathcal{O}(n^4)$ for RITS complexity carries over to $\mathrm{dc}_{\mathcal{R}}$ of input!

AProVE finds lower bound $\Omega(n^3)$ for $\mathrm{dc}_{\mathcal{R}}$.[52]

[52] F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder: *Lower bounds for runtime complexity of term rewriting*, JAR '17

Input for Automated Tools (1/4)

Automated tools for TRS Complexity at recent Termination Competitions:

- AProVE: https://aprove.informatik.rwth-aachen.de/
- TcT: https://tcs-informatik.uibk.ac.at/tools/tct/

[53] 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.

Input for Automated Tools (1/4)

Automated tools for TRS Complexity at recent Termination Competitions:

- 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/`

---

[53] 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.

Input for Automated Tools (1/4)

Automated tools for TRS Complexity at recent Termination Competitions:

- 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/

Input format for runtime complexity:[53]

```
(VAR x y)
(GOAL COMPLEXITY)
(STARTTERM CONSTRUCTOR-BASED)
(RULES
  plus(0, y) -> y
  plus(s(x), y) -> s(plus(x, y))
)
```

[53] 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.

Input for Automated Tools (2/4)

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))
)
```

Input for Automated Tools (3/4)

Derivational complexity:

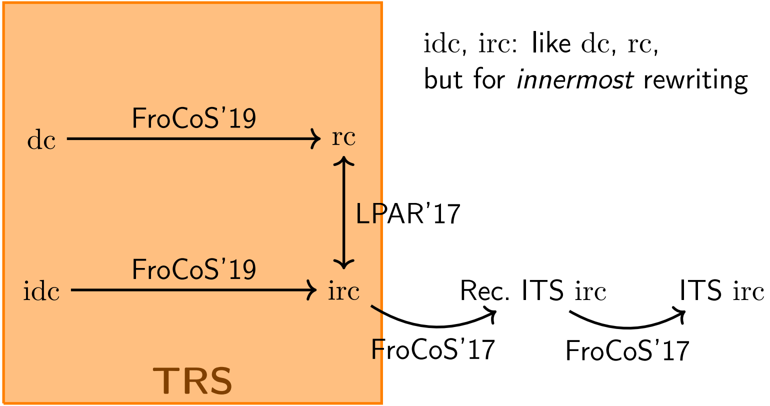
```
(VAR x y)
(GOAL COMPLEXITY)
(STARTTERM UNRESTRICTED)
(RULES
  plus(0, y) -> y
  plus(s(x), y) -> s(plus(x, y))
)
```

Input for Automated Tools (4/4)

Innermost derivational complexity:

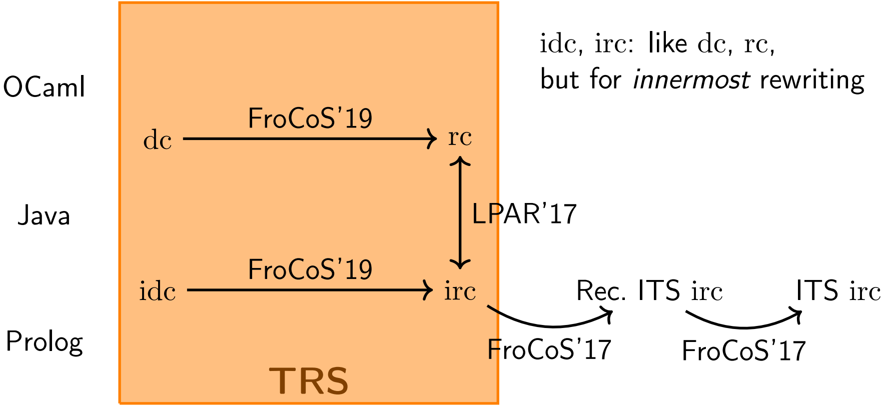
```
(VAR x y)
(GOAL COMPLEXITY)
(STARTTERM UNRESTRICTED)
(STRATEGY INNERMOST)
(RULES
  plus(0, y) -> y
  plus(s(x), y) -> s(plus(x, y))
)
```

A Landscape of Complexity Properties and Transformations



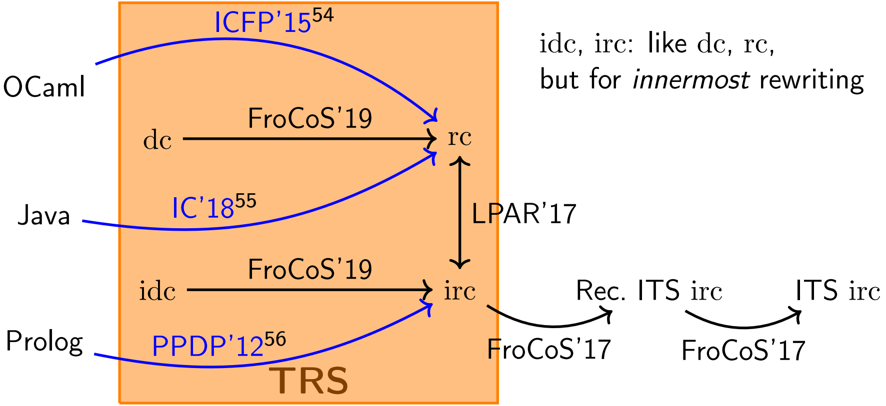
idc, irc: like dc, rc,
but for *innermost* rewriting

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

54 M. Avanzini, U. Dal Lago, G. Moser: *Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order*, ICFP '15

55 G. Moser, M. Schaper: *From Jinja bytecode to term rewriting: A complexity reflecting transformation*, IC '18

56 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

Program Complexity Analysis via Term Rewriting: OCaml

Complexity analysis for functional programs (OCaml) by translation to term rewriting

Program Complexity Analysis via Term Rewriting: OCaml

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: map(F, xs)

Program Complexity Analysis via Term Rewriting: OCaml

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: $\mathsf{map}(F, xs)$

Solution:

- Defunctionalisation to: $\mathsf{a}(\mathsf{a}(\mathsf{map}, F), xs)$
- Analyse start term with non-functional parameter types, then partially evaluate functions to instantiate higher-order variables
- Further program transformations

$\Rightarrow$ First-order TRS $\mathcal{R}$ with $\mathrm{rc}_{\mathcal{R}}(n)$ an upper bound for the complexity of the OCaml program

Program Complexity Analysis via Term Rewriting: Prolog and Java

Complexity analysis for Prolog programs and for Java programs by translation to term rewriting

Program Complexity Analysis via Term Rewriting: Prolog and Java

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[57])
- Deal with language specifics in program analysis
- Extract TRS $\mathcal{R}$ such that $\mathrm{rc}_{\mathcal{R}}(n)$ 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!

[57] P. Cousot, R. Cousot: *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, POPL '77

- **amortised** complexity analysis for term rewriting[58]

[58] G. Moser, M. Schneckenreither: *Automated amortised resource analysis for term rewrite systems*, SCP '20

Current Developments

- **amortised** complexity analysis for term rewriting[58]
- **probabilistic** term rewriting $\longrightarrow$ upper bounds on **expected runtime**[59]

[58] G. Moser, M. Schneckenreither: *Automated amortised resource analysis for term rewrite systems*, SCP '20
[59] M. Avanzini, U. Dal Lago, A. Yamada: *On probabilistic term rewriting*, SCP '20

Current Developments

- **amortised** complexity analysis for term rewriting[58]
- **probabilistic** term rewriting $\longrightarrow$ upper bounds on **expected runtime**[59]
- complexity analysis for **logically constrained rewriting** with built-in data types from SMT theories (integers, booleans, arrays, ...)[60]

---

[58] G. Moser, M. Schneckenreither: *Automated amortised resource analysis for term rewrite systems*, SCP '20
[59] M. Avanzini, U. Dal Lago, A. Yamada: *On probabilistic term rewriting*, SCP '20
[60] S. Winkler, G. Moser: *Runtime complexity analysis of logically constrained rewriting*, LOPSTR '20

Current Developments

- **amortised** complexity analysis for term rewriting[58]
- **probabilistic** term rewriting $\longrightarrow$ upper bounds on **expected runtime**[59]
- complexity analysis for **logically constrained rewriting** with built-in data types from SMT theories (integers, booleans, arrays, ...)[60]
- direct analysis of complexity for **higher-order term rewriting**[61]

---

[58] G. Moser, M. Schneckenreither: *Automated amortised resource analysis for term rewrite systems*, SCP '20
[59] M. Avanzini, U. Dal Lago, A. Yamada: *On probabilistic term rewriting*, SCP '20
[60] S. Winkler, G. Moser: *Runtime complexity analysis of logically constrained rewriting*, LOPSTR '20
[61] C. Kop, D. Vale: *Tuple interpretations for higher-order rewriting*, FSCD '21

Current Developments

- **amortised** complexity analysis for term rewriting[58]
- **probabilistic** term rewriting $\longrightarrow$ upper bounds on **expected runtime**[59]
- complexity analysis for **logically constrained rewriting** with built-in data types from SMT theories (integers, booleans, arrays, ...)[60]
- direct analysis of complexity for **higher-order term rewriting**[61]
- analysis of **parallel**-innermost runtime complexity[62]

[58] G. Moser, M. Schneckenreither: *Automated amortised resource analysis for term rewrite systems*, SCP '20
[59] M. Avanzini, U. Dal Lago, A. Yamada: *On probabilistic term rewriting*, SCP '20
[60] S. Winkler, G. Moser: *Runtime complexity analysis of logically constrained rewriting*, LOPSTR '20
[61] C. Kop, D. Vale: *Tuple interpretations for higher-order rewriting*, FSCD '21
[62] 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
- $\sim$ 2007/8: projects A3PAT[63], CoLoR[64], IsaFoR[65] formalise term rewriting, termination, proof techniques $\longrightarrow$ automatic proof checkers

[63] E. Contejean,P. Courtieu, J. Forest, O. Pons, X. Urbain: *Automated Certified Proofs with CiME3*, RTA '11
[64] 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
[65] 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
- $\sim$ 2007/8: projects A3PAT[63], CoLoR[64], IsaFoR[65] formalise term rewriting, termination, proof techniques $\longrightarrow$ automatic proof checkers
- performance bottleneck: computations in theorem prover

[63] E. Contejean, P. Courtieu, J. Forest, O. Pons, X. Urbain: *Automated Certified Proofs with CiME3*, RTA '11

[64] 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

[65] 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
- $\sim$ 2007/8: projects A3PAT[63], CoLoR[64], IsaFoR[65] formalise term rewriting, termination, proof techniques $\longrightarrow$ automatic proof checkers
- performance bottleneck: computations in theorem prover
- solution: extract source code (Haskell, OCaml, ...) for proof checker
  $\longrightarrow$ CeTA tool from IsaFoR

[63] E. Contejean,P. Courtieu, J. Forest, O. Pons, X. Urbain: *Automated Certified Proofs with CiME3*, RTA '11

[64] 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

[65] R. Thiemann, C. Sternagel: *Certification of Termination Proofs using CeTA*, TPHOLs '09

Proof Certification with CeTA

http://cl-informatik.uibk.ac.at/isafor/

CeTA can certify proofs for...

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[66]

[66] 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[66]
- non-termination for TRSs

[66] 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[66]
- non-termination for TRSs
- upper bounds for complexity

[66] 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[66]
- non-termination for TRSs
- upper bounds for complexity
- confluence and non-confluence proofs for TRSs

[66] 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[66]
- non-termination for TRSs
- upper bounds for complexity
- confluence and non-confluence proofs for TRSs
- safety: invariants for ITSs[67]

[66] M. Haslbeck, R. Thiemann: *An Isabelle/HOL formalization of AProVE's termination method for LLVM IR*, CPP '21

[67] M. Brockschmidt, S. Joosten, R. Thiemann, A. Yamada: *Certifying Safety and Termination Proofs for Integer Transition Systems*, CADE '17

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[66]
- non-termination for TRSs
- upper bounds for complexity
- confluence and non-confluence proofs for TRSs
- safety: invariants for ITSs[67]

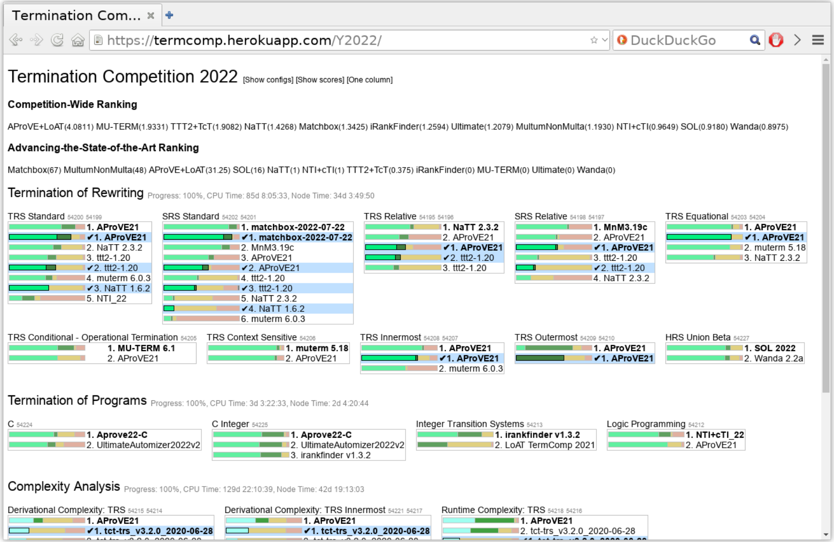
If certification unsuccessful:
CeTA indicates **which part** of the proof it could not follow

---

[66] M. Haslbeck, R. Thiemann: *An Isabelle/HOL formalization of AProVE's termination method for LLVM IR*, CPP '21

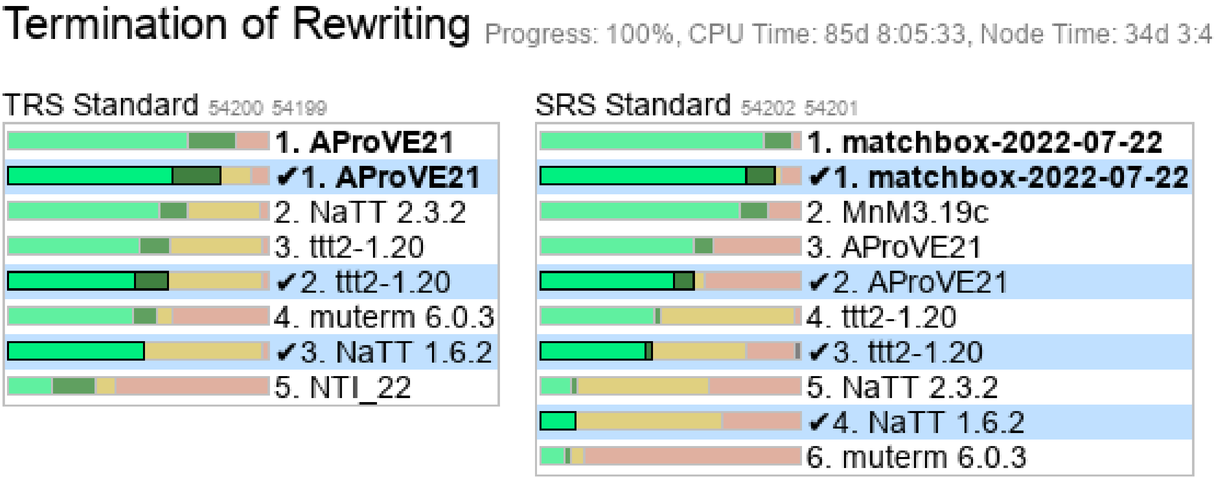
[67] M. Brockschmidt, S. Joosten, R. Thiemann, A. Yamada: *Certifying Safety and Termination Proofs for Integer Transition Systems*, CADE '17

termCOMP with Certification (✓) (1/2)



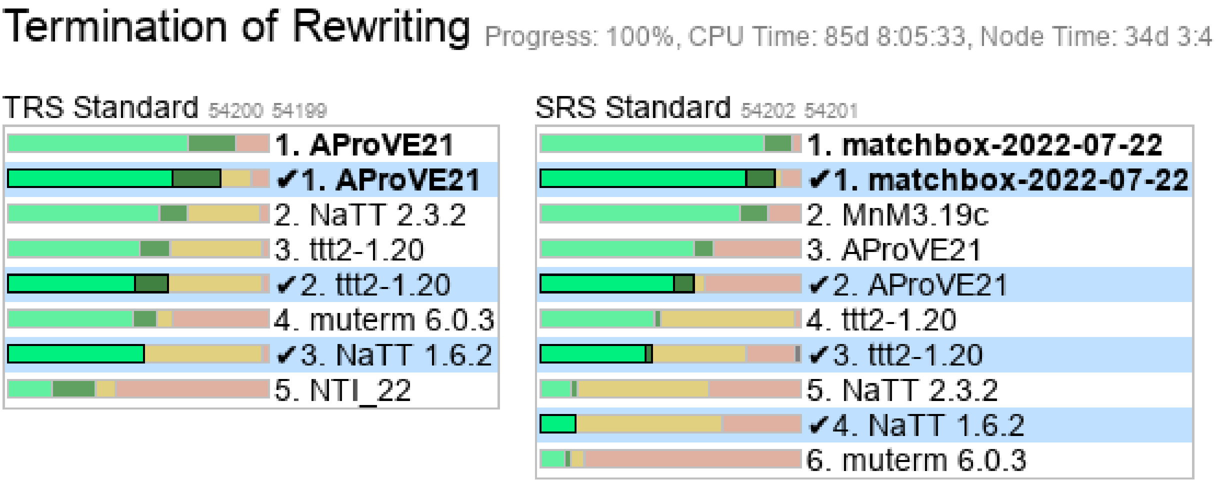
termCOMP with Certification (✓) (2/2)

Let's zoom in . . .



termCOMP with Certification (✓) (2/2)

Let's zoom in . . .



⇒ proof certification is competitive!

Termination and Complexity: Conclusion

- 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 – SAT/SMT solvers find the proof steps!

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 – SAT/SMT solvers find the proof steps!
- Cross-fertilisation between techniques for different formalisms (integer transition systems, functional programs, ...)

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 – SAT/SMT solvers find the proof steps!
- 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: 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 – SAT/SMT solvers find the proof steps!
- 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!

References I

- E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.
- C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS '10*, pages 117–133, 2010.
- T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
- M. Avanzini and G. Moser. Dependency pairs and polynomial path orders. In *RTA '09*, pages 48–62, 2009.
- M. Avanzini and G. Moser. A combination framework for complexity. *Information and Computation*, 248:22–55, 2016.
- M. Avanzini, G. Moser, and M. Schaper. TcT: Tyrolean Complexity Tool. In *TACAS '16*, pages 407–423, 2016.

References II

- M. Avanzini, U. Dal Lago, and A. Yamada. On probabilistic term rewriting. *Science of Computer Programming*, 185, 2020.
- T. Baudon, C. Fuhs, and L. Gonnord. Analysing parallel complexity of term rewriting. In *LOPSTR '22*, pages 3–23, 2022.
- J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV '06*, pages 386–400, 2006.
- R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: algebraic bound computation for loops. In *LPAR (Dakar) '10*, pages 103–118, 2010.
- F. Blanqui and A. Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011.
- G. Bonfante, A. Cichon, J. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001.

- C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, 2012.
- M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *RTA '11*, pages 155–170, 2011.
- M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *CAV '12*, pages 105–122, 2012a.
- M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. In *FoVeOOS '11*, pages 123–141, 2012b.
- M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *CAV '13*, pages 413–429, 2013.
- M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. T2: temporal property verification. In *TACAS '16*, pages 387–393, 2016a.

References IV

- M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Transactions on Programming Languages and Systems*, 38 (4), 2016b.
- M. Brockschmidt, S. J. C. Joosten, R. Thiemann, and A. Yamada. Certifying safety and termination proofs for integer transition systems. In *CADE '17*, pages 454–471, 2017.
- H.-Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. W. O'Hearn. Proving nontermination via safety. In *TACAS '14*, pages 156–171, 2014.
- E. Çiçek, M. Bouaziz, S. Cho, and D. Distefano. Static resource analysis at scale (extended abstract). In *SAS '20*, pages 3–6. Springer, 2020.
- Ș. Ciobâcă and D. Lucanu. A coinductive approach to proving reachability properties in logically constrained term rewriting systems. In *IJCAR '18*, pages 295–311, 2018.
- Ș. Ciobâcă, D. Lucanu, and A. Buruiana. Operationally-based program equivalence proofs using LCTRSs. *Journal of Logical and Algebraic Methods in Programming*, 135:100894, 2023.

References V

- M. Codish, J. Giesl, P. Schneider-Kamp, and R. Thiemann. SAT solving for termination proofs with recursive path orders and dependency pairs. *Journal of Automated Reasoning*, 49(1):53–93, 2012.
- E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Automated certified proofs with CiME3. In *RTA '11*, pages 21–30, 2011.
- B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *CAV '06*, pages 415–418, 2006a.
- B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI '06*, pages 415–426, 2006b.
- B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI '07*, pages 320–330, 2007.
- B. Cook, C. Fuhs, K. Nimkar, and P. W. O'Hearn. Disproving termination with overapproximation. In *FMCAD '14*, pages 67–74, 2014.

References VI

- B. Cook, H. Khlaaf, and N. Piterman. Verifying increasingly expressive temporal logics for infinite-state systems. *Journal of the ACM*, 64(2):15:1–15:39, 2017.
- P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252, 1977.
- N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3): 279–301, 1982.
- N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
- F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In *IJCAR '12*, pages 225–240, 2012.
- J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2–3):195–220, 2008.

References VII

- S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *CADE '09*, pages 277–293, 2009.
- A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *APLAS '14*, pages 275–295, 2014.
- F. Frohn and J. Giesl. Analyzing runtime complexity via innermost runtime complexity. In *Proc. LPAR '17*, pages 249–268, 2017a.
- F. Frohn and J. Giesl. Complexity analysis for Java with AProVE. In *iFM '17*, pages 85–101, 2017b.
- F. Frohn and J. Giesl. Proving non-termination and lower runtime bounds with loat (system description). In *IJCAR '22*, pages 712–722, 2022.
- F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder. Lower bounds for runtime complexity of term rewriting. *Journal of Automated Reasoning*, 59(1):121–163, 2017.

References VIII

- F. Frohn, M. Naaf, M. Brockschmidt, and J. Giesl. Inferring lower runtime bounds for integer programs. *ACM Transactions on Programming Languages and Systems*, 42(3):13:1–13:50, 2020.
- C. Fuhs. Transforming derivational complexity of term rewriting to runtime complexity. In *FroCoS '19*, pages 348–364, 2019.
- C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT '07*, pages 340–354, 2007.
- C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal termination. In *RTA '08*, pages 110–125, 2008a.
- C. Fuhs, R. Navarro-Marset, C. Otto, J. Giesl, S. Lucas, and P. Schneider-Kamp. Search techniques for rational polynomial orders. In *AISC '08*, pages 109–124, 2008b.
- C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *RTA '09*, pages 32–47, 2009.

References IX

- C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM Transactions on Computational Logic*, 18(2):14:1–14:50, 2017.
- A. Geser, D. Hofbauer, and J. Waldmann. Match-bounded string rewriting systems. *Applicable Algebra in Engineering, Communication and Computing*, 15(3–4):149–171, 2004.
- J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *FroCoS '05*, pages 216–231, 2005.
- J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
- J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):1–39, 2011. See also `http://aprove.informatik.rwth-aachen.de/eval/Haskell/`.

References X

- J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting: A general methodology for analyzing logic programs. In *PPDP '12*, pages 1–12, 2012.
- J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58 (1):3–31, 2017.
- S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL '09*, pages 127–139, 2009.
- A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *POPL '08*, pages 147–158, 2008.
- M. W. Haslbeck and R. Thiemann. An Isabelle/HOL formalization of AProVE's termination method for LLVM IR. In *CPP '21*, pages 238–249, 2021.

References XI

- J. Hensel and J. Giesl. Proving termination of C programs with lists. In *CADE '23*, pages 266–285, 2023.
- J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *Journal of Logical and Algebraic Methods in Programming*, 97:105–130, 2018.
- N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.
- N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *IJCAR '08*, pages 364–379, 2008.
- N. Hirokawa and G. Moser. Automated complexity analysis based on context-sensitive rewriting. In *RTA-TLCA '14*, pages 257–271, 2014.
- D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *RTA '89*, pages 167–177, 1989.

References XII

- J. Hoffmann and S. Jost. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science*, pages 1–31, 2022.
- J. Hoffmann and Z. Shao. Type-based amortized resource analysis with integers and arrays. *Journal of Functional Programming*, 25, 2015.
- J. Hoffmann, K. Aehlig, and M. Hofmann. Resource aware ML. In *CAV '12*, pages 781–786, 2012.
- H. Hong and D. Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21 (1):23–38, 1998.
- I. S. Hristakiev. *Confluence Analysis for a Graph Programming Language*. PhD thesis, University of York, 2009.
- S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, Urbana, IL, USA, 1980.

References XIII

- J. Kassing and J. Giesl. Proving almost-sure innermost termination of probabilistic term rewriting using dependency pairs. In *CADE '23*, pages 344–364, 2023.
- D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.
- M. Kojima and N. Nishida. Reducing non-occurrence of specified runtime errors to all-path reachability problems of constrained rewriting. *Journal of Logical and Algebraic Methods in Programming*, 135:100903, 2023.
- C. Kop. *Higher Order Termination*. PhD thesis, VU Amsterdam, 2012.
- C. Kop. Termination of LCTRSs. In *WST '13*, pages 59–63, 2013.
- C. Kop and N. Nishida. Term rewriting with logical constraints. In *FroCoS '13*, pages 343–358, 2013.
- C. Kop and D. Vale. Tuple interpretations for higher-order complexity. In *FSCD '21*, pages 31:1–31:22, 2021.

References XIV

- A. Koprowski and J. Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybernetica*, 19(2):357–392, 2009.
- K. Korovin and A. Voronkov. Orienting rewrite rules with the Knuth-Bendix order. *Information and Computation*, 183(2):165–186, 2003.
- M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *RTA '09*, pages 295–304, 2009.
- D. S. Lankford. Canonical algebraic simplification in computational logic. Technical Report ATP-25, University of Texas, 1975.
- D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination of imperative programs using Max-SMT. In *FMCAD '13*, pages 218–225, 2013.
- L. Leutgeb, G. Moser, and F. Zuleger. Automated expected amortised cost analysis of probabilistic data structures. In *CAV '22, Part II*, pages 70–91, 2022.

References XV

- N. Lommen, F. Meyer, and J. Giesl. Automatic complexity analysis of integer programs via triangular weakly non-linear loops. In *IJCAR '22*, pages 734–754, 2022.
- S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO - Theoretical Informatics and Applications*, 39(3):547–586, 2005.
- S. Lucas. Context-sensitive rewriting. *ACM Computing Surveys*, 53(4):78:1–78:36, 2020.
- J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.
- A. Merayo Corcoba. *Resource analysis of integer and abstract programs*. PhD thesis, Universidad Complutense de Madrid, 2022.
- F. Meyer, M. Hark, and J. Giesl. Inferring expected runtimes of probabilistic integer programs using expected sizes. In *TACAS '21, Part I*, pages 250–269, 2021.
- G. Moser and M. Schaper. From Jinja bytecode to term rewriting: A complexity reflecting transformation. *Information and Computation*, 261:116–143, 2018.

References XVI

- G. Moser and A. Schnabl. The derivational complexity induced by the dependency pair method. *Logical Methods in Computer Science*, 7(3), 2011a.
- G. Moser and A. Schnabl. Termination proofs in the dependency pair framework may induce multiple recursive derivational complexity. In *RTA '11*, pages 235–250, 2011b.
- G. Moser and M. Schneckenreither. Automated amortised resource analysis for term rewrite systems. *Science of Computer Programming*, 185, 2020.
- G. Moser, A. Schnabl, and J. Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *FSTTCS '08*, pages 304–315, 2008.
- M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl. Complexity analysis for term rewriting by integer transition systems. In *FroCoS '17*, pages 132–150, 2017.
- F. Neurauter, H. Zankl, and A. Middeldorp. Revisiting matrix interpretations for polynomial derivational complexity of term rewriting. In *LPAR (Yogyakarta) '10*, pages 550–564, 2010.

References XVII

- N. Nishida and S. Winkler. Loop detection by logically constrained term rewriting. In *VSTTE '18*, pages 309–321, 2018.
- L. Noschinski, F. Emmes, and J. Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013.
- C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *RTA '10*, pages 259–276, 2010.
- É. Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theoretical Computer Science*, 403(2-3), 2008.
- A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI '04*, pages 239–251, 2004.
- A. Schnabl and J. G. Simonsen. The exact hardness of deciding derivational and runtime complexity. In *CSL '11*, pages 481–495, 2011.

References XVIII

- P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1):1–52, 2009.
- J. Schöpf and A. Middeldorp. Confluence criteria for logically constrained rewrite systems. In *CADE '23*, pages 474–490, 2023.
- M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV '14*, pages 745–761, 2014.
- T. Ströder, F. Emmes, P. Schneider-Kamp, J. Giesl, and C. Fuhs. A linear operational semantics for termination and complexity analysis of ISO Prolog. In *LOPSTR '11*, pages 237–252, 2012.
- T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning*, 58(1):33–65, 2017.
- A. Stump, G. Sutcliffe, and C. Tinelli. Starexec: A cross-community infrastructure for logic solving. In *IJCAR '14*, pages 367–373, 2014. `https://www.starexec.org/`.

References XIX

- R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *TPHOLs '09*, pages 452–468, 2009.
- A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.
- A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.
- F. van Raamsdonk. Translating logic programs into conditional rewriting systems. In *ICLP '97*, pages 168–182, 1997.
- S. G. Vorobyov. Conditional rewrite rule systems with built-in arithmetic and induction. In *RTA '89*, pages 492–512, 1989.
- P. Wang, H. Fu, A. K. Goharshady, K. Chatterjee, X. Qin, and W. Shi. Cost analysis of nondeterministic probabilistic programs. In *PLDI '19*, pages 204–220, 2019.

References XX

- A. Weiermann. Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths. *Theoretical Computer Science*, 139(1&2):355–362, 1995.
- S. Winkler and G. Moser. Runtime complexity analysis of logically constrained rewriting. In *LOPSTR '20*, pages 37–55, 2020.
- A. Yamada. Tuple interpretations for termination of term rewriting. *Journal of Automated Reasoning*, 66(4):667–688, 2022.
- A. Yamada, K. Kusakari, and T. Sakabe. A unified ordering for termination proving. *Science of Computer Programming*, 111:110–134, 2015.
- H. Zankl and A. Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *LPAR (Dakar) '10*, pages 481–500, 2010.
- H. Zankl, N. Hirokawa, and A. Middeldorp. KBO orientability. *Journal of Automated Reasoning*, 43(2):173–201, 2009.

References XXI

H. Zankl, C. Sternagel, D. Hofbauer, and A. Middeldorp. Finding and certifying loops. In *SOFSEM '10*, pages 755–766, 2010.