Automated Static Program Analysis via Constrained Term Rewriting

Carsten Fuhs

Birkbeck, University of London

15th International Symposium on Frontiers of Combining Systems (FroCoS 2025)

Reykjavík, Iceland

30th September 2025

Outline

Outline

1. Static Program Analysis
2. Constrained Term Rewriting
3. Proving Termination of Logically Constrained Simply-typed Term Rewriting Systems
4. Proving Termination of Scala Programs by Constrained Term Rewriting

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
    $\longrightarrow$ important for safety-critical applications
    $\longrightarrow$ 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
  $\Rightarrow$ 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 (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**

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

- **Assertions by the programmer**.

```
assert x > 0;
```

  $\longrightarrow$ 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**

  $\longrightarrow$ 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)

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**
  $\longrightarrow$ 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**

Note: All these properties are **undecidable**!
$\Rightarrow$ use automatable sufficient criteria in practice

How to approach static program analysis?

Goal: (Automatically) prove whether a given program P has (un)desirable property
Approach: Often in two phases

How to approach static program analysis?

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: Intermediate representation amenable to automated analysis
- Often over-approximation, preserves the property of interest

How to approach static program analysis?

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: Intermediate representation amenable to automated analysis
- Often over-approximation, preserves the property of interest

Back-End

- Performs the analysis of the desired property

$\Rightarrow$ Result carries over to original program

How to approach static program analysis?

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: Intermediate representation amenable to automated analysis
- Often over-approximation, preserves the property of interest

Back-End

- Performs the analysis of the desired property

$\Rightarrow$ Result carries over to original program

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ Result for $\mathcal{R}_P$ + proof

How to approach static program analysis?

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: Intermediate representation amenable to automated analysis
- Often over-approximation, preserves the property of interest

Back-End

- Performs the analysis of the desired property

$\Rightarrow$ Result carries over to original program

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ Result for $\mathcal{R}_P$ + proof

Outline

Choosing an intermediate representation

Inspiration from proving program termination:

Choosing an intermediate representation

Inspiration from proving program termination:

Early translations in the literature use Term Rewriting Systems (TRSs):

- Prolog [van Raamsdonk, *ICLP '97*; Giesl et al, *PPDP '12*]
- Haskell [Giesl et al, *TOPLAS '11*]
- ...

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
- no fixed order of rules to apply (Haskell, Scala: top to bottom)
- untyped
- no pre-defined data structures (integers, arrays, ...)

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
- no fixed order of rules to apply (Haskell, Scala: top to bottom)
- untyped
- no pre-defined data structures (integers, arrays, ...)

$\Rightarrow$ Programs over user-defined algebraic data types

Summing up natural numbers

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

Summing up natural numbers

Numbers: $\mathsf{0}$, $\mathsf{s}(\mathsf{0})$, $\mathsf{s}(\mathsf{s}(\mathsf{0}))$, ...

Rules:

$$
\begin{aligned}
\mathsf{sum}(\mathsf{0}) &\to \mathsf{0} \\
\mathsf{sum}(\mathsf{s}(x)) &\to \mathsf{plus}(\mathsf{s}(x), \mathsf{sum}(x)) \\
\mathsf{plus}(\mathsf{0}, y) &\to y \\
\mathsf{plus}(\mathsf{s}(x), y) &\to \mathsf{s}(\mathsf{plus}(x, y))
\end{aligned}
$$

Summing up natural numbers

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

Rules:

$$\begin{array}{rcl} \mathsf{sum}(\mathsf{0}) & \to & \mathsf{0} \\ \mathsf{sum}(\mathsf{s}(x)) & \to & \mathsf{plus}(\mathsf{s}(x), \mathsf{sum}(x)) \\ \mathsf{plus}(\mathsf{0}, y) & \to & y \\ \mathsf{plus}(\mathsf{s}(x), y) & \to & \mathsf{s}(\mathsf{plus}(x, y)) \end{array}$$

Then we can compute, e.g., $1 + 1 = 2$ as

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

Summing up natural numbers

Numbers: $\mathsf{0}$, $\mathsf{s}(\mathsf{0})$, $\mathsf{s}(\mathsf{s}(\mathsf{0}))$, ...

Rules:

$$\begin{array}{rcl} \mathsf{sum}(\mathsf{0}) & \to & \mathsf{0} \\ \mathsf{sum}(\mathsf{s}(x)) & \to & \mathsf{plus}(\mathsf{s}(x), \mathsf{sum}(x)) \\ \mathsf{plus}(\mathsf{0}, y) & \to & y \\ \mathsf{plus}(\mathsf{s}(x), y) & \to & \mathsf{s}(\mathsf{plus}(x, y)) \end{array}$$

Then we can compute, e.g., $1 + 1 = 2$ as

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

Integer arithmetic possible with more complex recursive rules.

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

Drawbacks:

- throws away domain knowledge about built-in data types like integers
- need to analyse recursive rules for plus, ... over and over

Beyond classic TRSs for program analysis

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

Drawbacks:

- throws away domain knowledge about built-in data types like integers
- need to analyse recursive rules for plus, . . . 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 $\mathsf{s}(\mathsf{s}(\mathsf{s}(...)))$?!

Drawbacks:

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

Solution: use **constrained term rewriting**

What's constrained term rewriting?

Term rewriting "with batteries included"

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

What's constrained term rewriting?

Term rewriting "with batteries included"

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

What's constrained term rewriting?

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 (SMT: SAT Modulo Theories)

What's constrained term rewriting?

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 (SMT: SAT Modulo Theories)
- rewrite rules with SMT constraints

What's constrained term rewriting?

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 (SMT: SAT Modulo Theories)
- rewrite rules with SMT constraints

$\Rightarrow$ Programs over user-defined algebraic data types and pre-defined data types

What's constrained term rewriting?

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 (SMT: SAT Modulo Theories)
- rewrite rules with SMT constraints

$\Rightarrow$ Programs over user-defined algebraic data types and pre-defined data types

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

What's constrained term rewriting?

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 (SMT: SAT Modulo Theories)
- rewrite rules with SMT constraints

$\Rightarrow$ Programs over user-defined algebraic data types and pre-defined data types

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

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

What's constrained term rewriting?

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 (SMT: SAT Modulo Theories)
- rewrite rules with SMT constraints

$\Rightarrow$ Programs over user-defined algebraic data types and pre-defined data types

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

- Constrained rewriting known at least since [Vorobyov, *RTA '89*]
- Consolidated as **Logically Constrained TRSs (LCTRSs)** [Kop, Nishida, *FroCoS '13*]

What's constrained term rewriting?

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 (SMT: SAT Modulo Theories)
- rewrite rules with SMT constraints

$\Rightarrow$ Programs over user-defined algebraic data types and pre-defined data types

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

- Constrained rewriting known at least since [Vorobyov, *RTA '89*]
- Consolidated as **Logically Constrained TRSs (LCTRSs)** [Kop, Nishida, *FroCoS '13*]
- Adoption of LCTRSs by the community: 83 citations so far (Google Scholar, Sep 2025)

Logically Constrained TRSs: adoption

Analysis techniques for Logically Constrained TRSs:

Logically Constrained TRSs: adoption

Analysis techniques for Logically Constrained TRSs:

- Termination
  [Kop, *WST '13*; Nishida, Winkler, *VSTTE '18*; Matsumi, Nishida, Kojima, Shin, *WST'23*]

Logically Constrained TRSs: adoption

Analysis techniques for Logically Constrained TRSs:

- Termination
  [Kop, *WST '13*; Nishida, Winkler, *VSTTE '18*; Matsumi, Nishida, Kojima, Shin, *WST'23*]
- Complexity [Winkler, Moser, *LOPSTR '20*]

Logically Constrained TRSs: adoption

Analysis techniques for Logically Constrained TRSs:

- Termination
  [Kop, *WST '13*; Nishida, Winkler, *VSTTE '18*; Matsumi, Nishida, Kojima, Shin, *WST'23*]
- Complexity [Winkler, Moser, *LOPSTR '20*]
- Equivalence [Fuhs, Kop, Nishida, *TOCL '17*; Ciobâcă, Lucanu, Buruiana, *JLAMP '23*]

Logically Constrained TRSs: adoption

Analysis techniques for Logically Constrained TRSs:

- Termination
  [Kop, *WST '13*; Nishida, Winkler, *VSTTE '18*; Matsumi, Nishida, Kojima, Shin, *WST'23*]
- 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*]

Logically Constrained TRSs: adoption

Analysis techniques for Logically Constrained TRSs:

- Termination
  [Kop, *WST '13*; Nishida, Winkler, *VSTTE '18*; Matsumi, Nishida, Kojima, Shin, *WST'23*]
- 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)

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

Constrained rewriting by example

Example (Constrained Rewrite System)

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

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:

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

Constrained rewriting by example

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

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

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)

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

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.

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

Outline

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/

Existing translations for termination

Proving program termination:

Program P in language L → Intermediate representation $\mathcal{R}_P$ → YES/NO/MAYBE for $\mathcal{R}_P$ + proof

Existing translations for termination

Proving program termination:

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ YES/NO/MAYBE for $\mathcal{R}_P$ + proof

Many translations to (constrained) rewriting in the literature

- Prolog [van Raamsdonk, *ICLP '97*], [Giesl et al, *PPDP '12*]
- Java [Otto et al, *RTA '10*]
- Haskell [Giesl et al, *TOPLAS '11*]
- LLVM [Ströder et al, *JAR '17*]
- C [Fuhs, Kop, Nishida, *TOCL '17*]
- Jinja [Moser, Schaper, *IC '18*]
- Scala [Milovančević, Fuhs, Kunčak, *WPTE '25*]
- ...

Proving call-by-value termination of constrained higher-order rewriting

Proving program termination:

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ YES/NO/MAYBE for $\mathcal{R}_P$ + proof

Proving call-by-value termination of constrained higher-order rewriting

Proving program termination:

Program P in language L → Intermediate representation $\mathcal{R}_P$ → YES/NO/MAYBE for $\mathcal{R}_P$ + proof

Here: use intermediate representation for **higher-order functional programs**

Proving call-by-value termination of constrained higher-order rewriting

Proving program termination:

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ YES/NO/MAYBE for $\mathcal{R}_P$ + proof

Constrained higher-order rewriting . . .

Here: use intermediate representation for **higher-order functional programs**

Proving program termination:

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ YES/NO/MAYBE for $\mathcal{R}_P$ + proof

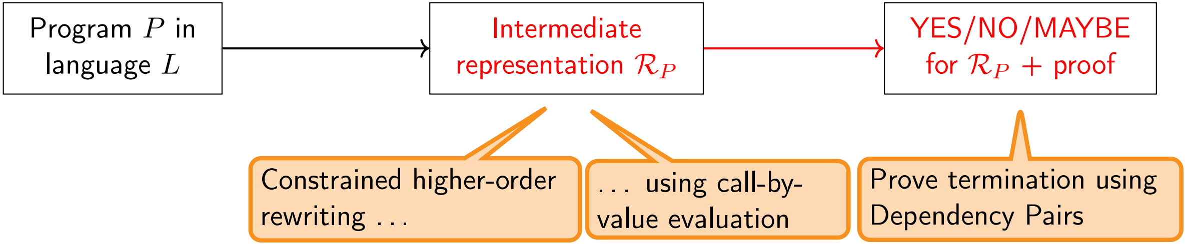
Constrained higher-order rewriting . . .

. . . using call-by-value evaluation

Here: use intermediate representation for **higher-order functional programs**

Proving call-by-value termination of constrained higher-order rewriting

Proving program termination:



Here: use intermediate representation for **higher-order functional programs**

Our intermediate representation: LCSTRSs

Ideal intermediate representation should

- be good for automated reasoning (no "lost in encoding")
- express language features directly

Our intermediate representation: LCSTRSs

Ideal intermediate representation should

- be good for automated reasoning (no “lost in encoding”)
- express language features directly

What is available?

Our intermediate representation: LCSTRSs

Ideal intermediate representation should

- be good for automated reasoning (no "lost in encoding")
- express language features directly

$\mathsf{length}\ \mathsf{nil} \to \mathsf{zero}$	$\mathsf{length}\ (\mathsf{cons}\ x\ xs) \to \mathsf{s}\ (\mathsf{length}\ xs)$
$\mathsf{plus}\ x\ \mathsf{zero} \to x$	$\mathsf{plus}\ x\ (\mathsf{s}\ y) \to \mathsf{s}\ (\mathsf{plus}\ x\ y)$

What is available?

- Term Rewriting Systems aka TRSs: functions on algebraic data structures

Our intermediate representation: LCSTRSs

Ideal intermediate representation should

- be good for automated reasoning (no "lost in encoding")
- express language features directly

$$\mathsf{length}\ \mathsf{nil} \to \mathsf{zero} \qquad \mathsf{length}\ (\mathsf{cons}\ x\ xs) \to \mathsf{s}\ (\mathsf{length}\ xs)$$
$$\mathsf{plus}\ x\ \mathsf{zero} \to x \qquad \mathsf{plus}\ x\ (\mathsf{s}\ y) \to \mathsf{s}\ (\mathsf{plus}\ x\ y)$$

What is available?

- Term Rewriting Systems aka TRSs: functions on algebraic data structures
- Integer Transition Systems aka ITSs: functions/statements on integer data + arithmetic

$$\mathsf{gcd}\ m\ n \to \mathsf{gcd}\ (-m)\ n\ [m < 0] \qquad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ m\ (-n)\ [n < 0]$$
$$\mathsf{gcd}\ m\ 0 \to m\ [m \geq 0] \qquad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ n\ (m \bmod n)\ [m \geq 0 \land n > 0]$$

Our intermediate representation: LCSTRSs

Ideal intermediate representation should

- be good for automated reasoning (no "lost in encoding")
- express language features directly

$$\mathsf{length}\ \mathsf{nil} \to \mathsf{zero} \qquad \mathsf{length}\ (\mathsf{cons}\ x\ xs) \to \mathsf{s}\ (\mathsf{length}\ xs)$$
$$\mathsf{plus}\ x\ \mathsf{zero} \to x \qquad \mathsf{plus}\ x\ (\mathsf{s}\ y) \to \mathsf{s}\ (\mathsf{plus}\ x\ y)$$

What is available?

- Term Rewriting Systems aka TRSs: functions on algebraic data structures
- Integer Transition Systems aka ITSs: functions/statements on integer data + arithmetic

$$\mathsf{gcd}\ m\ n \to \mathsf{gcd}\ (-m)\ n\ [m < 0] \qquad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ m\ (-n)\ [n < 0]$$
$$\mathsf{gcd}\ m\ 0 \to m\ [m \geq 0] \qquad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$$

- Logically Constrained TRSs aka LCTRSs [Kop, Nishida, *FroCoS '13*]:
  TRSs + ITSs + arbitrary logical theories (arrays, bitvectors, ...)

Our intermediate representation: LCSTRSs

Ideal intermediate representation should

- be good for automated reasoning (no "lost in encoding")
- express language features directly

$$\mathsf{length}\ \mathsf{nil} \to \mathsf{zero} \quad \Big| \quad \mathsf{length}\ (\mathsf{cons}\ x\ xs) \to \mathsf{s}\ (\mathsf{length}\ xs)$$
$$\mathsf{plus}\ x\ \mathsf{zero} \to x \quad \Big| \quad \mathsf{plus}\ x\ (\mathsf{s}\ y) \to \mathsf{s}\ (\mathsf{plus}\ x\ y)$$

What is available?

- Term Rewriting Systems aka TRSs: functions on algebraic data structures
- Integer Transition Systems aka ITSs: functions/statements on integer data + arithmetic

$$\mathsf{gcd}\ m\ n \to \mathsf{gcd}\ (-m)\ n\ [m < 0] \quad \Big| \quad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ m\ (-n)\ [n < 0]$$
$$\mathsf{gcd}\ m\ 0 \to m\ [m \geq 0] \quad \Big| \quad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$$

- Logically Constrained TRSs aka LCTRSs [Kop, Nishida, *FroCoS '13*]:
  TRSs + ITSs + arbitrary logical theories (arrays, bitvectors, . . .)
- **Logically Constrained Simply-typed TRSs aka LCSTRSs** [Guo, Kop, *ESOP '24*]:
  LCTRSs + higher-order functions (but no λ)

$$\mathsf{gcdlist} : \mathsf{intlist} \to \mathsf{int}, \quad \mathsf{fold} : (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{intlist} \to \mathsf{int}$$
$$\mathsf{gcdlist} \to \mathsf{fold}\ \mathsf{gcd}\ 0 \quad \Big| \quad \mathsf{fold}\ f\ y\ \mathsf{nil} \to y \quad \Big| \quad \mathsf{fold}\ f\ y\ (\mathsf{cons}\ x\ l) \to f\ x\ (\mathsf{fold}\ f\ y\ l)$$

Call-by-value (cbv) and innermost rewriting for LCSTRSs

Evaluating with an LCSTRS

$$\begin{aligned} \mathsf{fact}\ 0 &\to 1 \\ \mathsf{fact}\ x &\to x * \mathsf{fact}\ (x-1)\ [x > 0] \\ \mathsf{g}\ x &\to \mathsf{g}\ (\mathsf{fact}\ {-1}) \end{aligned}$$

Call-by-value (cbv) and innermost rewriting for LCSTRSs

Evaluating with an LCSTRS

$$\begin{aligned}\mathsf{fact}\ 0 &\to 1\\ \mathsf{fact}\ x &\to x * \mathsf{fact}\ (x-1)\ [x>0]\\ \mathsf{g}\ x &\to \mathsf{g}\ (\mathsf{fact}\ -1)\end{aligned}$$

Cbv rewriting

Proper subterms of redex:

ground values

$$\begin{aligned} &\mathsf{g}\ \underline{(\mathsf{fact}\ 1)} \xrightarrow{\text{v}} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0})\\ \xrightarrow{\text{v}}\ &\mathsf{g}\ \underline{(1*1)} \xrightarrow{\text{v}} \underline{\mathsf{g}\ 1}\\ \xrightarrow{\text{v}}\ &\mathsf{g}\ (\mathsf{fact}\ -1) \not\xrightarrow{\text{v}}\end{aligned}$$

Evaluating with an LCSTRS

$$\mathsf{fact}\ 0 \to 1$$
$$\mathsf{fact}\ x \to x * \mathsf{fact}\ (x-1)\ [x > 0]$$
$$\mathsf{g}\ x \to \mathsf{g}\ (\mathsf{fact}\ {-1})$$

Cbv rewriting

Proper subterms of redex:
ground values

$$\mathsf{g}\ \underline{(\mathsf{fact}\ 1)} \xrightarrow{\mathrm{v}} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0})$$
$$\xrightarrow{\mathrm{v}} \mathsf{g}\ \underline{(1 * 1)} \xrightarrow{\mathrm{v}} \underline{\mathsf{g}\ 1}$$
$$\xrightarrow{\mathrm{v}} \mathsf{g}\ (\mathsf{fact}\ {-1}) \not\xrightarrow{\mathrm{v}}$$

- Cbv: used in programming languages
- Want to prove termination of cbv rewriting!

Call-by-value (cbv) and innermost rewriting for LCSTRSs

Evaluating with an LCSTRS

$\mathsf{fact}\ 0 \to 1$
$\mathsf{fact}\ x \to x * \mathsf{fact}\ (x-1)\ [x > 0]$
$\mathsf{g}\ x \to \mathsf{g}\ (\mathsf{fact}\ {-1})$

Cbv rewriting

Proper subterms of redex: ground values

$$\mathsf{g}\ \underline{(\mathsf{fact}\ 1)} \xrightarrow{\mathrm{v}} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \xrightarrow{\mathrm{v}} \mathsf{g}\ \underline{(1*1)} \xrightarrow{\mathrm{v}} \underline{\mathsf{g}\ 1} \xrightarrow{\mathrm{v}} \mathsf{g}\ (\mathsf{fact}\ {-1}) \not\xrightarrow{\mathrm{v}}$$

- Cbv: used in programming languages
- Want to prove termination of cbv rewriting!
- Literature on termination: focus on innermost rewriting

Call-by-value (cbv) and innermost rewriting for LCSTRSs

Evaluating with an LCSTRS

$$\begin{aligned} \mathsf{fact}\ 0 &\to 1 \\ \mathsf{fact}\ x &\to x * \mathsf{fact}\ (x-1)\ [x > 0] \\ \mathsf{g}\ x &\to \mathsf{g}\ (\mathsf{fact}\ {-1}) \end{aligned}$$

Cbv rewriting

Proper subterms of redex: ground values

$$\begin{aligned} & \mathsf{g}\ \underline{(\mathsf{fact}\ 1)} \xrightarrow{\mathrm{v}} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \\ \xrightarrow{\mathrm{v}}\ & \mathsf{g}\ \underline{(1 * 1)} \xrightarrow{\mathrm{v}} \underline{\mathsf{g}\ 1} \\ \xrightarrow{\mathrm{v}}\ & \mathsf{g}\ (\mathsf{fact}\ {-1}) \not\xrightarrow{\mathrm{v}} \end{aligned}$$

Innermost rewriting

Proper subterms of redex: normal forms

$$\begin{aligned} & \mathsf{g}\ \underline{(\mathsf{fact}\ 1)} \xrightarrow{\mathrm{i}} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \\ \xrightarrow{\mathrm{i}}\ & \mathsf{g}\ \underline{(1 * 1)} \xrightarrow{\mathrm{i}} \underline{\mathsf{g}\ 1} \\ \xrightarrow{\mathrm{i}}\ & \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} \xrightarrow{\mathrm{i}} \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} \\ \xrightarrow{\mathrm{i}}\ & \ldots \end{aligned}$$

- Cbv: used in programming languages
- Want to prove termination of cbv rewriting!
- Literature on termination: focus on innermost rewriting

Call-by-value (cbv) and innermost rewriting for LCSTRSs

Evaluating with an LCSTRS

$$\begin{aligned} &\mathsf{fact}\ 0 \to 1 \\ &\mathsf{fact}\ x \to x * \mathsf{fact}\ (x-1)\ [x > 0] \\ &\mathsf{g}\ x \to \mathsf{g}\ (\mathsf{fact}\ {-1}) \end{aligned}$$

Cbv rewriting

Proper subterms of redex: ground values

$$\begin{array}{llll} & \mathsf{g}\ \underline{(\mathsf{fact}\ 1)} & \xrightarrow{\mathrm{v}} & \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \\ \xrightarrow{\mathrm{v}} & \mathsf{g}\ \underline{(1 * 1)} & \xrightarrow{\mathrm{v}} & \underline{\mathsf{g}\ 1} \\ \xrightarrow{\mathrm{v}} & \mathsf{g}\ (\mathsf{fact}\ {-1}) & \not\xrightarrow{\mathrm{v}} & \end{array}$$

Innermost rewriting

Proper subterms of redex: normal forms

$$\begin{array}{llll} & \mathsf{g}\ \underline{(\mathsf{fact}\ 1)} & \xrightarrow{\mathrm{i}} & \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \\ \xrightarrow{\mathrm{i}} & \mathsf{g}\ \underline{(1 * 1)} & \xrightarrow{\mathrm{i}} & \underline{\mathsf{g}\ 1} \\ \xrightarrow{\mathrm{i}} & \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} & \xrightarrow{\mathrm{i}} & \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} \\ \xrightarrow{\mathrm{i}} & \dots & & \end{array}$$

- Cbv: used in programming languages
- Want to prove termination of cbv rewriting!
- Literature on termination: focus on innermost rewriting
- Solution [Fuhs, Guo, Kop, *FSCD '25*]: mention $x : \mathsf{int}$ in constraint
  $\Rightarrow$ LCTRS semantics says: all x occurring in constraint must be instantiated by theory values!
- int is inextensible theory sort

Call-by-value (cbv) and innermost rewriting for LCSTRSs

Evaluating with an LCSTRS

$$\mathsf{fact}\ 0 \to 1$$
$$\mathsf{fact}\ x \to x * \mathsf{fact}\ (x-1)\ [x > 0]$$
$$\mathsf{g}\ x \to \mathsf{g}\ (\mathsf{fact}\ {-1})$$

Cbv rewriting

Proper subterms of redex: ground values

$$\mathsf{g}\ \underline{(\mathsf{fact}\ 1)} \xrightarrow{\mathrm{v}} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \xrightarrow{\mathrm{v}} \mathsf{g}\ \underline{(1 * 1)} \xrightarrow{\mathrm{v}} \underline{\mathsf{g}\ 1} \xrightarrow{\mathrm{v}} \mathsf{g}\ (\mathsf{fact}\ {-1}) \not\xrightarrow{\mathrm{v}}$$

Innermost rewriting

Proper subterms of redex: normal forms

$$\mathsf{g}\ \underline{(\mathsf{fact}\ 1)} \xrightarrow{\mathrm{i}} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \xrightarrow{\mathrm{i}} \mathsf{g}\ \underline{(1 * 1)} \xrightarrow{\mathrm{i}} \underline{\mathsf{g}\ 1} \xrightarrow{\mathrm{i}} \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} \xrightarrow{\mathrm{i}} \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} \xrightarrow{\mathrm{i}} \dots$$

- Cbv: used in programming languages
- Want to prove termination of cbv rewriting!
- Literature on termination: focus on innermost rewriting
- Solution [Fuhs, Guo, Kop, *FSCD '25*]: mention $x : \mathsf{int}$ in constraint
  $\Rightarrow$ LCTRS semantics says: all x occurring in constraint must be instantiated by theory values!
- int is inextensible theory sort

$$\mathsf{fact}\ 0 \to 1$$
$$\mathsf{fact}\ x \to x * \mathsf{fact}\ (x-1)\ [x > 0]$$
$$\mathsf{g}\ x \to \mathsf{g}\ (\mathsf{fact}\ {-1})\quad [x \equiv x]$$

Call-by-value (cbv) and innermost rewriting for LCSTRSs

Evaluating with an LCSTRS

$$\mathsf{fact}\ 0 \to 1$$
$$\mathsf{fact}\ x \to x * \mathsf{fact}\ (x-1)\ [x > 0]$$
$$\mathsf{g}\ x \to \mathsf{g}\ (\mathsf{fact}\ {-1})$$

Cbv rewriting

Proper subterms of redex: ground values

$$\begin{aligned} & \mathsf{g}\ \underline{(\mathsf{fact}\ 1)} && \xrightarrow{\text{v}} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \\ \xrightarrow{\text{v}}\ & \mathsf{g}\ \underline{(1 * 1)} && \xrightarrow{\text{v}} \underline{\mathsf{g}\ 1} \\ \xrightarrow{\text{v}}\ & \mathsf{g}\ (\mathsf{fact}\ {-1}) && \not\xrightarrow{\text{v}} \end{aligned}$$

Innermost rewriting

Proper subterms of redex: normal forms

$$\begin{aligned} & \mathsf{g}\ \underline{(\mathsf{fact}\ 1)} && \xrightarrow{\text{i}} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \\ \xrightarrow{\text{i}}\ & \mathsf{g}\ \underline{(1 * 1)} && \xrightarrow{\text{i}} \underline{\mathsf{g}\ 1} \\ \xrightarrow{\text{i}}\ & \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} && \xrightarrow{\text{i}} \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} \\ \xrightarrow{\text{i}}\ & \dots \end{aligned}$$

- Cbv: used in programming languages
- Want to prove termination of cbv rewriting!
- Literature on termination: focus on innermost rewriting
- Solution [Fuhs, Guo, Kop, *FSCD '25*]: mention $x : \mathsf{int}$ in constraint
  $\Rightarrow$ LCTRS semantics says: all x occurring in constraint must be instantiated by theory values!
- int is inextensible theory sort

$$\mathsf{fact}\ 0 \to 1$$
$$\mathsf{fact}\ x \to x * \mathsf{fact}\ (x-1)\ [x > 0]$$
$$\mathsf{g}\ x \to \mathsf{g}\ (\mathsf{fact}\ {-1})\qquad [x \equiv x]$$

$\Rightarrow$ Terminates also for innermost rewriting!

Call-by-value (cbv) and innermost rewriting for LCSTRSs

Evaluating with an LCSTRS

$$\mathsf{fact}\ 0 \to 1$$
$$\mathsf{fact}\ x \to x * \mathsf{fact}\ (x-1)\ [x > 0]$$
$$\mathsf{g}\ x \to \mathsf{g}\ (\mathsf{fact}\ {-1})$$

Cbv rewriting

Proper subterms of redex: ground values

$$\mathsf{g}\ \underline{(\mathsf{fact}\ 1)} \xrightarrow{v} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \xrightarrow{v} \mathsf{g}\ \underline{(1*1)} \xrightarrow{v} \underline{\mathsf{g}\ 1} \xrightarrow{v} \mathsf{g}\ (\mathsf{fact}\ {-1}) \not\xrightarrow{v}$$

Innermost rewriting

Proper subterms of redex: normal forms

$$\mathsf{g}\ \underline{(\mathsf{fact}\ 1)} \xrightarrow{i} \mathsf{g}\ (1 * \underline{\mathsf{fact}\ 0}) \xrightarrow{i} \mathsf{g}\ \underline{(1*1)} \xrightarrow{i} \underline{\mathsf{g}\ 1} \xrightarrow{i} \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} \xrightarrow{i} \underline{\mathsf{g}\ (\mathsf{fact}\ {-1})} \xrightarrow{i} \dots$$

- Cbv: used in programming languages
- Want to prove termination of cbv rewriting!
- Literature on termination: focus on innermost rewriting
- Solution [Fuhs, Guo, Kop, *FSCD '25*]: mention $x : \mathsf{int}$ in constraint
  $\Rightarrow$ LCTRS semantics says: all x occurring in constraint must be instantiated by theory values!
- int is inextensible theory sort

$$\mathsf{fact}\ 0 \to 1$$
$$\mathsf{fact}\ x \to x * \mathsf{fact}\ (x-1)\ [x > 0]$$
$$\mathsf{g}\ x \to \mathsf{g}\ (\mathsf{fact}\ {-1})\quad [x \equiv x]$$

$\Rightarrow$ Terminates also for innermost rewriting!

Note: transformation enforces cbv only for theory sorts

Static Dependency Pairs

$\mathcal{R}_{\mathsf{gcd}}$

$\mathsf{gcdlist} \to \mathsf{fold}\ \mathsf{gcd}\ 0$

$\mathsf{fold}\ f\ y\ \mathsf{nil} \to y\ \ [y \equiv y] \quad | \quad \mathsf{fold}\ f\ y\ (\mathsf{cons}\ x\ l) \to f\ x\ (\mathsf{fold}\ f\ y\ l)\ \ [x \equiv x \wedge y \equiv y]$

$\mathsf{gcd}\ m\ n \to \mathsf{gcd}\ (-m)\ n\ [m < 0 \wedge n \equiv n] \quad | \quad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ m\ (-n)\ [n < 0 \wedge m \equiv m]$

$\mathsf{gcd}\ m\ 0 \to m\ [m \geq 0] \quad | \quad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

Static Dependency Pairs

$\mathcal{R}_{\mathsf{gcd}}$

$\mathsf{gcdlist} \to \mathsf{fold}\ \mathsf{gcd}\ 0$

$\mathsf{fold}\ f\ y\ \mathsf{nil} \to y\ \ [y \equiv y] \quad | \quad \mathsf{fold}\ f\ y\ (\mathsf{cons}\ x\ l) \to f\ x\ (\mathsf{fold}\ f\ y\ l)\ \ [x \equiv x \wedge y \equiv y]$

$\mathsf{gcd}\ m\ n \to \mathsf{gcd}\ (-m)\ n\ [m < 0 \wedge n \equiv n] \quad | \quad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ m\ (-n)\ [n < 0 \wedge m \equiv m]$

$\mathsf{gcd}\ m\ 0 \to m\ [m \geq 0] \quad | \quad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

Prove termination by Static Dependency Pairs for LCSTRSs [Guo, Hagens, Kop, Vale, *MFCS '24*]

- For LCSTRS $\mathcal{R}$ build dependency pairs $\mathcal{P} = \mathsf{SDP}(\mathcal{R}_{\mathsf{gcd}})$ (∼ function calls)

Static Dependency Pairs

$\mathcal{R}_{\text{gcd}}$

$\mathsf{gcdlist} \to \mathsf{fold}\ \mathsf{gcd}\ 0$

$\mathsf{fold}\ f\ y\ \mathsf{nil} \to y\ \ [y \equiv y] \quad | \quad \mathsf{fold}\ f\ y\ (\mathsf{cons}\ x\ l) \to f\ x\ (\mathsf{fold}\ f\ y\ l)\ \ [x \equiv x \wedge y \equiv y]$

$\mathsf{gcd}\ m\ n \to \mathsf{gcd}\ (-m)\ n\ [m < 0 \wedge n \equiv n] \quad | \quad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ m\ (-n)\ [n < 0 \wedge m \equiv m]$

$\mathsf{gcd}\ m\ 0 \to m\ [m \geq 0] \quad | \quad \mathsf{gcd}\ m\ n \to \mathsf{gcd}\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

Prove termination by Static Dependency Pairs for LCSTRSs [Guo, Hagens, Kop, Vale, *MFCS '24*]

- For LCSTRS $\mathcal{R}$ build dependency pairs $\mathcal{P} = \mathsf{SDP}(\mathcal{R}_{\text{gcd}})$ ($\sim$ function calls)
- Show: No ∞ call sequence with $\mathcal{P}$ (eval of $\mathcal{P}$'s args via $\mathcal{R}$)

Static Dependency Pairs

$\mathcal{R}_{\text{gcd}}$

$\text{gcdlist} \to \text{fold gcd } 0$

$\text{fold } f\ y \text{ nil} \to y \quad [y \equiv y] \quad | \quad \text{fold } f\ y\ (\text{cons } x\ l) \to f\ x\ (\text{fold } f\ y\ l) \quad [x \equiv x \wedge y \equiv y]$

$\text{gcd } m\ n \to \text{gcd } (-m)\ n\ [m < 0 \wedge n \equiv n] \quad | \quad \text{gcd } m\ n \to \text{gcd } m\ (-n) \quad [n < 0 \wedge m \equiv m]$

$\text{gcd } m\ 0 \to m \quad [m \geq 0] \quad | \quad \text{gcd } m\ n \to \text{gcd } n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

Prove termination by Static Dependency Pairs for LCSTRSs [Guo, Hagens, Kop, Vale, *MFCS '24*]

- For LCSTRS $\mathcal{R}$ build dependency pairs $\mathcal{P} = \text{SDP}(\mathcal{R}_{\text{gcd}})$ ($\sim$ function calls)
- Show: No ∞ call sequence with $\mathcal{P}$ (eval of $\mathcal{P}$'s args via $\mathcal{R}$)

$\text{SDP}(\mathcal{R}_{\text{gcd}})$

$\text{gcdlist}^\sharp\ l' \Rightarrow \text{fold}^\sharp \text{ gcd } 0\ l' \quad | \quad \text{gcd}^\sharp\ m\ n \Rightarrow \text{gcd}^\sharp\ (-m)\ n\ [m < 0 \wedge n \equiv n]$

$\text{gcdlist}^\sharp\ l' \Rightarrow \text{gcd}^\sharp\ m'\ n' \quad | \quad \text{gcd}^\sharp\ m\ n \Rightarrow \text{gcd}^\sharp\ m\ (-n)\ [n < 0 \wedge m \equiv m]$

$\text{fold}^\sharp\ f\ y\ (\text{cons } x\ l) \Rightarrow \text{fold}^\sharp\ f\ y\ l\ [x \equiv x \wedge y \equiv y] \quad | \quad \text{gcd}^\sharp\ m\ n \Rightarrow \text{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

Dependency Pair Framework

- Works on DP problems $(\mathcal{P}, \mathcal{R})$
- DP framework:

$S := \{(\mathsf{SDP}(\mathcal{R}), \mathcal{R})\}$
while $S = S' \uplus \{(\mathcal{P}, \mathcal{R})\}$
$\qquad S := S' \cup \rho(\mathcal{P}, \mathcal{R})$ for a DP processor ρ
print "YES"

Dependency Pair Framework

- Works on DP problems $(\mathcal{P}, \mathcal{R})$
- DP framework:

$S := \{(\mathsf{SDP}(\mathcal{R}), \mathcal{R})\}$
while $S = S' \uplus \{(\mathcal{P}, \mathcal{R})\}$
$\qquad S := S' \cup \rho(\mathcal{P}, \mathcal{R})$ for a DP processor ρ
print "YES"

Existing DP processors for LCSTRSs [Guo, Hagens, Kop, Vale, *MFCS '24*]

- Graph processor
- Subterm criterion processor
- Integer mapping processor

Dependency Pair Framework

- Works on DP problems $(\mathcal{P}, \mathcal{R})$
- DP framework:

$S := \{(\mathsf{SDP}(\mathcal{R}), \mathcal{R})\}$
while $S = S' \uplus \{(\mathcal{P}, \mathcal{R})\}$
    $S := S' \cup \rho(\mathcal{P}, \mathcal{R})$ for a DP processor ρ
print "YES"

Existing DP processors for LCSTRSs [Guo, Hagens, Kop, Vale, *MFCS '24*]

- Graph processor
- Subterm criterion processor
- Integer mapping processor

New **innermost** DP processors for LCSTRSs [Fuhs, Guo, Kop, *FSCD '25*]

- Usable rules processor
- Reduction pair processor with usable rules wrt argument filtering
- Chaining processor

Also for **compositional termination analysis** via universal computability!

Existing DP processors for LCSTRSs

$\mathcal{P}$

(1) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{fold}^\sharp\ \mathsf{gcd}\ 0\ l'$

(2) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{gcd}^\sharp\ m'\ n'$

(3) $\mathsf{fold}^\sharp\ f\ y\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{fold}^\sharp\ f\ y\ l\ [x \equiv x \wedge y \equiv y]$

(4) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ (-m)\ n\ [m < 0 \wedge n \equiv n]$

(5) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ m\ (-n)\ [n < 0 \wedge m \equiv m]$

(6) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

$\mathcal{P}$

(1) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{fold}^\sharp\ \mathsf{gcd}\ 0\ l'$

(2) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{gcd}^\sharp\ m'\ n'$

(3) $\mathsf{fold}^\sharp\ f\ y\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{fold}^\sharp\ f\ y\ l\ [x \equiv x \wedge y \equiv y]$

(4) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ (-m)\ n\ [m < 0 \wedge n \equiv n]$

(5) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ m\ (-n)\ [n < 0 \wedge m \equiv m]$

(6) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

$\mathcal{R}$

$\ldots$

$\mathcal{P}$

(1) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{fold}^\sharp\ \mathsf{gcd}\ 0\ l'$

(2) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{gcd}^\sharp\ m'\ n'$

(3) $\mathsf{fold}^\sharp\ f\ y\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{fold}^\sharp\ f\ y\ l\ [x \equiv x \wedge y \equiv y]$

(4) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ (-m)\ n\ [m < 0 \wedge n \equiv n]$

(5) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ m\ (-n)\ [n < 0 \wedge m \equiv m]$

(6) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

$\mathcal{R}$

...

- **Dependency Graph**:
  which calls may follow one another?

$\mathcal{P}$

(1) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{fold}^\sharp\ \mathsf{gcd}\ 0\ l'$

(2) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{gcd}^\sharp\ m'\ n'$

(3) $\mathsf{fold}^\sharp\ f\ y\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{fold}^\sharp\ f\ y\ l\ [x \equiv x \wedge y \equiv y]$

(4) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ (-m)\ n\ [m < 0 \wedge n \equiv n]$

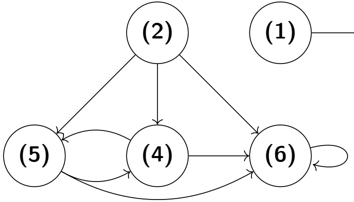
(5) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ m\ (-n)\ [n < 0 \wedge m \equiv m]$

(6) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

$\mathcal{R}$

$\ldots$

- **Dependency Graph**:
  which calls may follow one another?
- Approximation
  [Guo, Hagens, Kop, Vale, *MFCS '24*]:



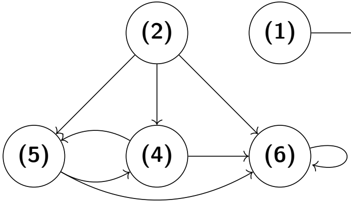
$\mathcal{P}$

(1) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{fold}^\sharp\ \mathsf{gcd}\ 0\ l'$
(2) $\mathsf{gcdlist}^\sharp\ l' \Rightarrow \mathsf{gcd}^\sharp\ m'\ n'$
(3) $\mathsf{fold}^\sharp\ f\ y\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{fold}^\sharp\ f\ y\ l\ [x \equiv x \wedge y \equiv y]$
(4) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ (-m)\ n\ [m < 0 \wedge n \equiv n]$
(5) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ m\ (-n)\ [n < 0 \wedge m \equiv m]$
(6) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

$\mathcal{R}$

. . .

- **Dependency Graph**: which calls may follow one another?
- Approximation [Guo, Hagens, Kop, Vale, *MFCS '24*]:



- **Graph processor**: decompose $\mathcal{P}$ into non-trivial Strongly Connected Components

$\mathcal{P}$

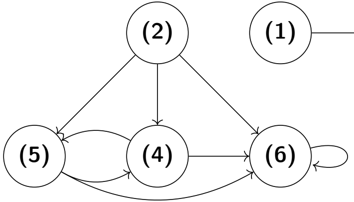
(1) $\text{gcdlist}^\sharp\ l' \Rightarrow \text{fold}^\sharp\ \text{gcd}\ 0\ l'$
(2) $\text{gcdlist}^\sharp\ l' \Rightarrow \text{gcd}^\sharp\ m'\ n'$
(3) $\text{fold}^\sharp\ f\ y\ (\text{cons}\ x\ l) \Rightarrow \text{fold}^\sharp\ f\ y\ l\ [x \equiv x \wedge y \equiv y]$

(4) $\text{gcd}^\sharp\ m\ n \Rightarrow \text{gcd}^\sharp\ (-m)\ n\ [m < 0 \wedge n \equiv n]$
(5) $\text{gcd}^\sharp\ m\ n \Rightarrow \text{gcd}^\sharp\ m\ (-n)\ [n < 0 \wedge m \equiv m]$
(6) $\text{gcd}^\sharp\ m\ n \Rightarrow \text{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

$\mathcal{R}$

$\ldots$

- **Dependency Graph**: which calls may follow one another?
- Approximation [Guo, Hagens, Kop, Vale, *MFCS '24*]:



- **Graph processor**: decompose $\mathcal{P}$ into non-trivial Strongly Connected Components
- Here:

$$(\{\mathbf{(3)}\}, \mathcal{R})$$
$$(\{\mathbf{(6)}\}, \mathcal{R})$$
$$(\{\mathbf{(4)}, \mathbf{(5)}\}, \mathcal{R})$$

$\mathcal{P}$

(3) $\text{fold}^\sharp\ f\ y\ (\text{cons}\ x\ l) \Rightarrow \text{fold}^\sharp\ f\ y\ l\ [x \equiv x \wedge y \equiv y]$

$\mathcal{R}$

$\ldots$

$\mathcal{P}$

(3) $\mathsf{fold}^\sharp\ f\ y\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{fold}^\sharp\ f\ y\ l\ [x \equiv x \wedge y \equiv y]$

$\mathcal{R}$

$\ldots$

Subterm criterion processor [Guo, Hagens, Kop, Vale, *MFCS '24*]

- Detect structural decrease in argument
- Use projection $\nu(\mathsf{fold}^\sharp) = 3$
- Get $\mathsf{cons}\ x\ l \rhd l$

$\Rightarrow$ Remove **(3)**

$\Rightarrow$ $(\emptyset, \mathcal{R})$ deleted by graph processor

$\mathcal{P}$

(6) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \land n > 0]$

$\mathcal{R}$

$\ldots$

$\mathcal{P}$

(6) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

$\mathcal{R}$

$\ldots$

Integer mapping processor [Guo, Hagens, Kop, Vale, *MFCS '24*]

- Detect integer value decrease in argument
- Use projection $\nu(\mathsf{gcd}^\sharp) = 2$
- Get $m \geq 0 \wedge n > 0 \models n > m \bmod n$
  and $m \geq 0 \wedge n > 0 \models n \geq 0$

$\Rightarrow$ Remove **(6)**

$\Rightarrow$ $(\emptyset, \mathcal{R})$ deleted by graph processor

$\mathcal{P}$

(6) $\mathsf{gcd}^\sharp\ m\ n \Rightarrow \mathsf{gcd}^\sharp\ n\ (m \bmod n)\ [m \geq 0 \wedge n > 0]$

$\mathcal{R}$

$\ldots$

Integer mapping processor [Guo, Hagens, Kop, Vale, *MFCS '24*]

- Detect integer value decrease in argument
- Use projection $\nu(\mathsf{gcd}^\sharp) = 2$
- Get $m \geq 0 \wedge n > 0 \models n > m \bmod n$
  and $m \geq 0 \wedge n > 0 \models n \geq 0$

$\Rightarrow$ Remove **(6)**

$\Rightarrow$ $(\emptyset, \mathcal{R})$ deleted by graph processor

$(\{\mathbf{(4)}, \mathbf{(5)}\}, \mathcal{R})$ handled by integer mapping processor + graph processor

$\Rightarrow$ termination of $\mathcal{R}_{\mathsf{gcd}}$ proved!

New DP processors for LCSTRSs

Usable rules processor

$\mathcal{R}_{\mathsf{dfoldr}}$

drop : int → alist → alist
dfoldr : (a → b → b) → b → int → alist → b

$$
\begin{array}{lcll}
\mathsf{drop}\ n\ l & \to & l & [n \leq 0] \\
\mathsf{drop}\ n\ \mathsf{nil} & \to & \mathsf{nil} & [n \equiv n] \\
\mathsf{drop}\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{drop}\ (n-1)\ l & [n > 0]
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{dfoldr}\ f\ y\ n\ \mathsf{nil} & \to & y & [n \equiv n] \\
\mathsf{dfoldr}\ f\ y\ n\ (\mathsf{cons}\ x\ l) & \to & f\ x\ (\mathsf{dfoldr}\ f\ y\ n\ (\mathsf{drop}\ n\ l)) & [n \equiv n]
\end{array}
$$

Usable rules processor

$\mathcal{R}_{\text{dfoldr}}$

drop : int → alist → alist
dfoldr : (a → b → b) → b → int → alist → b

$$
\begin{array}{lcll}
\text{drop}\ n\ l & \to & l & [n \le 0] \\
\text{drop}\ n\ \text{nil} & \to & \text{nil} & [n \equiv n] \\
\text{drop}\ n\ (\text{cons}\ x\ l) & \to & \text{drop}\ (n-1)\ l & [n > 0] \\
\text{dfoldr}\ f\ y\ n\ \text{nil} & \to & y & [n \equiv n] \\
\text{dfoldr}\ f\ y\ n\ (\text{cons}\ x\ l) & \to & f\ x\ (\text{dfoldr}\ f\ y\ n\ (\text{drop}\ n\ l)) & [n \equiv n]
\end{array}
$$

- Troublesome DP problem:

$$(\ \{ \ \text{dfoldr}^\sharp\ f\ y\ n\ (\text{cons}\ x\ l) \Rightarrow \text{dfoldr}^\sharp\ f\ y\ n\ (\text{drop}\ n\ l)\ [n \equiv n] \ \},\ \mathcal{R}_{\text{dfoldr}} \ )$$

Usable rules processor

$\mathcal{R}_{\text{dfoldr}}$

drop : int → alist → alist
dfoldr : (a → b → b) → b → int → alist → b

$$
\begin{array}{lcll}
\text{drop}\ n\ l & \to & l & [n \leq 0] \\
\text{drop}\ n\ \text{nil} & \to & \text{nil} & [n \equiv n] \\
\text{drop}\ n\ (\text{cons}\ x\ l) & \to & \text{drop}\ (n-1)\ l & [n > 0] \\
\text{dfoldr}\ f\ y\ n\ \text{nil} & \to & y & [n \equiv n] \\
\text{dfoldr}\ f\ y\ n\ (\text{cons}\ x\ l) & \to & f\ x\ (\text{dfoldr}\ f\ y\ n\ (\text{drop}\ n\ l)) & [n \equiv n]
\end{array}
$$

- Troublesome DP problem:

$$(\ \{ \ \text{dfoldr}^\sharp\ f\ y\ n\ (\text{cons}\ x\ l) \Rightarrow \text{dfoldr}^\sharp\ f\ y\ n\ (\text{drop}\ n\ l)\ [n \equiv n] \ \}, \ \mathcal{R}_{\text{dfoldr}} \ )$$

- Reduction pair processor can show $\text{cons}\ x\ l \succ \text{drop}\ n\ l$
- But cannot show $\text{dfoldr}\ f\ y\ n\ (\text{cons}\ x\ l) \succsim f\ x\ (\text{dfoldr}\ f\ y\ n\ (\text{drop}\ n\ l))[n \equiv n]$

Usable rules processor

$\mathcal{R}_{\mathsf{dfoldr}}$

drop : int → alist → alist
dfoldr : (a → b → b) → b → int → alist → b

$$
\begin{array}{llll}
\mathsf{drop}\ n\ l & \to & l & [n \le 0] \\
\mathsf{drop}\ n\ \mathsf{nil} & \to & \mathsf{nil} & [n \equiv n] \\
\mathsf{drop}\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{drop}\ (n-1)\ l & [n > 0] \\
\mathsf{dfoldr}\ f\ y\ n\ \mathsf{nil} & \to & y & [n \equiv n] \\
\mathsf{dfoldr}\ f\ y\ n\ (\mathsf{cons}\ x\ l) & \to & f\ x\ (\mathsf{dfoldr}\ f\ y\ n\ (\mathsf{drop}\ n\ l)) & [n \equiv n]
\end{array}
$$

- Troublesome DP problem:

$$(\ \{ \ \mathsf{dfoldr}^\sharp\ f\ y\ n\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{dfoldr}^\sharp\ f\ y\ n\ (\mathsf{drop}\ n\ l)\ [n \equiv n] \ \}, \ \mathcal{R}_{\mathsf{dfoldr}} \ )$$

- Reduction pair processor can show $\mathsf{cons}\ x\ l \succ \mathsf{drop}\ n\ l$
- But cannot show $\mathsf{dfoldr}\ f\ y\ n\ (\mathsf{cons}\ x\ l) \succsim f\ x\ (\mathsf{dfoldr}\ f\ y\ n\ (\mathsf{drop}\ n\ l))[n \equiv n]$
- **Usable rules processor**: keep only **usable** rules, called from DPs
- Here: rules for drop

Usable rules processor

$\mathcal{R}$

drop : int → alist → alist
dfoldr : (a → b → b) → b → int → alist → b

$$\begin{array}{lcll} \mathsf{drop}\ n\ l & \to & l & [n \le 0] \\ \mathsf{drop}\ n\ \mathsf{nil} & \to & \mathsf{nil} & [n \equiv n] \\ \mathsf{drop}\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{drop}\ (n-1)\ l & [n > 0] \end{array}$$

- Troublesome DP problem:

$$(\ \{\ \ \mathsf{dfoldr}^\sharp\ f\ y\ n\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{dfoldr}^\sharp\ f\ y\ n\ (\mathsf{drop}\ n\ l)\ [n \equiv n]\ \ \},\ \ \mathcal{R}_{\mathsf{dfoldr}}\ \ )$$

- Reduction pair processor can show $\mathsf{cons}\ x\ l \succ \mathsf{drop}\ n\ l$
- But cannot show $\mathsf{dfoldr}\ f\ y\ n\ (\mathsf{cons}\ x\ l) \succsim f\ x\ (\mathsf{dfoldr}\ f\ y\ n\ (\mathsf{drop}\ n\ l))[n \equiv n]$
- **Usable rules processor**: keep only **usable** rules, called from DPs
- Here: rules for drop

Reduction pair processor with usable rules wrt an argument filtering

$\mathcal{R}_{\mathsf{dfoldl}}$

$$
\begin{array}{lcll}
\mathsf{drop}\ n\ l & \to & l & [n \leq 0] \\
\mathsf{drop}\ n\ \mathsf{nil} & \to & \mathsf{nil} & [n \equiv n] \\
\mathsf{drop}\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{drop}\ (n-1)\ l & [n > 0] \\
\\
\mathsf{dfoldl}\ f\ y\ n\ \mathsf{nil} & \to & y & [n \equiv n] \\
\mathsf{dfoldl}\ f\ y\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{dfoldl}\ f\ (f\ y\ x)\ n\ (\mathsf{drop}\ n\ l) & [n \equiv n]
\end{array}
$$

Reduction pair processor with usable rules wrt an argument filtering

$\mathcal{R}_{\mathsf{dfoldl}}$

$$
\begin{array}{lcll}
\mathsf{drop}\ n\ l & \to & l & [n \leq 0] \\
\mathsf{drop}\ n\ \mathsf{nil} & \to & \mathsf{nil} & [n \equiv n] \\
\mathsf{drop}\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{drop}\ (n-1)\ l & [n > 0] \\
\mathsf{dfoldl}\ f\ y\ n\ \mathsf{nil} & \to & y & [n \equiv n] \\
\mathsf{dfoldl}\ f\ y\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{dfoldl}\ f\ (f\ y\ x)\ n\ (\mathsf{drop}\ n\ l) & [n \equiv n]
\end{array}
$$

- Troublesome DP problem:

$$
(\ \{\ \mathsf{dfoldl}^{\sharp}\ f\ y\ n\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{dfoldl}^{\sharp}\ f\ (f\ y\ x)\ n\ (\mathsf{drop}\ n\ l)\ [n \equiv n]\ \},\ \mathcal{R}_{\mathsf{dfoldl}}\ )
$$

Reduction pair processor with usable rules wrt an argument filtering

$\mathcal{R}_{\mathsf{dfoldl}}$

$$
\begin{array}{lcll}
\mathsf{drop}\ n\ l & \to & l & [n \le 0] \\
\mathsf{drop}\ n\ \mathsf{nil} & \to & \mathsf{nil} & [n \equiv n] \\
\mathsf{drop}\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{drop}\ (n-1)\ l & [n > 0] \\
\mathsf{dfoldl}\ f\ y\ n\ \mathsf{nil} & \to & y & [n \equiv n] \\
\mathsf{dfoldl}\ f\ y\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{dfoldl}\ f\ (f\ y\ x)\ n\ (\mathsf{drop}\ n\ l) & [n \equiv n]
\end{array}
$$

- Troublesome DP problem:

$$
(\ \{\ \mathsf{dfoldl}^\sharp\ f\ y\ n\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{dfoldl}^\sharp\ f\ (f\ y\ x)\ n\ (\mathsf{drop}\ n\ l)\ [n \equiv n]\ \},\ \mathcal{R}_{\mathsf{dfoldl}}\ )
$$

- All rules are usable!

Reduction pair processor with usable rules wrt an argument filtering

$\mathcal{R}_{\mathsf{dfoldl}}$

$$
\begin{array}{lcll}
\mathsf{drop}\ n\ l & \to & l & [n \leq 0] \\
\mathsf{drop}\ n\ \mathsf{nil} & \to & \mathsf{nil} & [n \equiv n] \\
\mathsf{drop}\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{drop}\ (n-1)\ l & [n > 0] \\
\mathsf{dfoldl}\ f\ y\ n\ \mathsf{nil} & \to & y & [n \equiv n] \\
\mathsf{dfoldl}\ f\ y\ n\ (\mathsf{cons}\ x\ l) & \to & \mathsf{dfoldl}\ f\ (f\ y\ x)\ n\ (\mathsf{drop}\ n\ l) & [n \equiv n]
\end{array}
$$

- Troublesome DP problem:

$$(\ \{ \ \mathsf{dfoldl}^{\sharp}\ f\ y\ n\ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{dfoldl}^{\sharp}\ f\ (f\ y\ x)\ n\ (\mathsf{drop}\ n\ l)\ [n \equiv n] \ \},\ \mathcal{R}_{\mathsf{dfoldl}} \ )$$

- All rules are usable!
- **Reduction pair processor with usable rules wrt argument filtering**:

  temporarily disregard arguments, calculate usable rules, use reduction pair (HORPO, ...)

Reduction pair processor with usable rules wrt an argument filtering

$\mathcal{R}_{\text{dfoldl}}$

$$
\begin{array}{lcll}
\text{drop}\ n\ l & \to & l & [n \leq 0] \\
\text{drop}\ n\ \text{nil} & \to & \text{nil} & [n \equiv n] \\
\text{drop}\ n\ (\text{cons}\ x\ l) & \to & \text{drop}\ (n-1)\ l & [n > 0] \\
\\
\text{dfoldl}\ f\ y\ n\ \text{nil} & \to & y & [n \equiv n] \\
\text{dfoldl}\ f\ y\ n\ (\text{cons}\ x\ l) & \to & \text{dfoldl}\ f\ (f\ y\ x)\ n\ (\text{drop}\ n\ l) & [n \equiv n]
\end{array}
$$

- Troublesome DP problem:

$$(\ \{ \ \text{dfoldl}^\sharp\ f\ y\ n\ (\text{cons}\ x\ l) \Rightarrow \text{dfoldl}^\sharp\ f\ (f\ y\ x)\ n\ (\text{drop}\ n\ l)\ [n \equiv n] \ \}, \ \mathcal{R}_{\text{dfoldl}} \ )$$

- All rules are usable!
- **Reduction pair processor with usable rules wrt argument filtering**:

  temporarily disregard arguments, calculate usable rules, use reduction pair (HORPO, ...)
- $\mathrm{regard}(\text{dfoldl}^\sharp) = \{4\}$ $\quad\Rightarrow\quad$ use first-order RPO!

Reduction pair processor with usable rules wrt an argument filtering

$\mathcal{R}_{\text{dfoldl}}$

$$
\begin{array}{lcll}
\text{drop}\ n\ l & \to & l & [n \leq 0] \\
\text{drop}\ n\ \text{nil} & \to & \text{nil} & [n \equiv n] \\
\text{drop}\ n\ (\text{cons}\ x\ l) & \to & \text{drop}\ (n-1)\ l & [n > 0] \\
\text{dfoldl}\ f\ y\ n\ \text{nil} & \to & y & [n \equiv n] \\
\text{dfoldl}\ f\ y\ n\ (\text{cons}\ x\ l) & \to & \text{dfoldl}\ f\ (f\ y\ x)\ n\ (\text{drop}\ n\ l) & [n \equiv n]
\end{array}
$$

- Troublesome DP problem:

$$(\ \{ \ \text{dfoldl}^\sharp \quad (\text{cons}\ x\ l) \Rightarrow \text{dfoldl}^\sharp \quad (\text{drop}\ n\ l)\ [n \equiv n] \ \},\ \mathcal{R}_{\text{dfoldl}} \ )$$

- All rules are usable!
- **Reduction pair processor with usable rules wrt argument filtering**:

  temporarily disregard arguments, calculate usable rules, use reduction pair (HORPO, . . .)
- $\mathrm{regard}(\text{dfoldl}^\sharp) = \{4\} \quad \Rightarrow$ use first-order RPO!

Reduction pair processor with usable rules wrt an argument filtering

$\mathcal{R}_{\mathsf{dfoldl}}$

$$
\begin{array}{lll}
\mathsf{drop}\ n\ l & \to\ l & [n \leq 0] \\
\mathsf{drop}\ n\ \mathsf{nil} & \to\ \mathsf{nil} & [n \equiv n] \\
\mathsf{drop}\ n\ (\mathsf{cons}\ x\ l) & \to\ \mathsf{drop}\ (n-1)\ l & [n > 0]
\end{array}
$$

- Troublesome DP problem:

$$(\ \{ \ \mathsf{dfoldl}^\sharp \ (\mathsf{cons}\ x\ l) \Rightarrow \mathsf{dfoldl}^\sharp \ (\mathsf{drop}\ n\ l)\ [n \equiv n] \ \},\ \mathcal{R}_{\mathsf{dfoldl}} \ )$$

- All rules are usable!
- **Reduction pair processor with usable rules wrt argument filtering**:

  temporarily disregard arguments, calculate usable rules, use reduction pair (HORPO, . . .)
- $\mathrm{regard}(\mathsf{dfoldl}^\sharp) = \{4\} \quad \Rightarrow$ use first-order RPO!

Chaining processor

(S)DPs from imperative program [Fuhs, Kop, Nishida, *TOCL '17*]

```
def fact(x):
  z = 1            # fact
  i = 1            # u1
  while i <= x:    # u2
    z = z * i      # u3
    i = i + 1      # u4
                   # u5
```

Chaining processor

(S)DPs from imperative program [Fuhs, Kop, Nishida, *TOCL '17*]

```
def fact(x):
  z = 1            # fact
  i = 1            # u1
  while i <= x:    # u2
    z = z * i      # u3
    i = i + 1      # u4
                   # u5
```

$$
\begin{array}{lcll}
\mathsf{fact}^\sharp\ x & \Rightarrow & \mathsf{u_1}^\sharp\ x\ 1 & [x \equiv x] \\
\mathsf{u_1}^\sharp\ x\ z & \Rightarrow & \mathsf{u_2}^\sharp\ x\ z\ 1 & [x \equiv x \wedge z \equiv z] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_3}^\sharp\ x\ z\ i & [i \leq x \wedge z \equiv z] \\
\mathsf{u_3}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_4}^\sharp\ x\ (z * i)\ i & [x \equiv x \wedge z \equiv z \wedge i \equiv i] \\
\mathsf{u_4}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_2}^\sharp\ x\ z\ (i+1) & [x \equiv x \wedge z \equiv z \wedge i \equiv i] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_5}^\sharp\ x\ z & [\neg(i \leq x) \wedge z \equiv z]
\end{array}
$$

- Automated translations $\Rightarrow$ DPs with many small steps
- Can be hard to analyse!

Chaining processor

(S)DPs from imperative program [Fuhs, Kop, Nishida, *TOCL '17*]

```
def fact(x):
  z = 1            # fact
  i = 1            # u1
  while i <= x:    # u2
    z = z * i      # u3
    i = i + 1      # u4
                   # u5
```

$$
\begin{array}{lcll}
\mathsf{fact}^\sharp\ x & \Rightarrow & \mathsf{u_1}^\sharp\ x\ 1 & [x \equiv x] \\
\mathsf{u_1}^\sharp\ x\ z & \Rightarrow & \mathsf{u_2}^\sharp\ x\ z\ 1 & [x \equiv x \wedge z \equiv z] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_3}^\sharp\ x\ z\ i & [i \leq x \wedge z \equiv z] \\
\mathsf{u_3}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_4}^\sharp\ x\ (z * i)\ i & [x \equiv x \wedge z \equiv z \wedge i \equiv i] \\
\mathsf{u_4}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_2}^\sharp\ x\ z\ (i+1) & [x \equiv x \wedge z \equiv z \wedge i \equiv i] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_5}^\sharp\ x\ z & [\neg(i \leq x) \wedge z \equiv z]
\end{array}
$$

- Automated translations $\Rightarrow$ DPs with many small steps
- Can be hard to analyse!
- **Chaining processor**: remove intermediate symbols $\mathsf{u_1}^\sharp$

Chaining processor

(S)DPs from imperative program [Fuhs, Kop, Nishida, *TOCL '17*]

```
def fact(x):
  z = 1              # fact
  i = 1              # u1
  while i <= x:      # u2
    z = z * i        # u3
    i = i + 1        # u4
                     # u5
```

$$
\begin{array}{lcll}
\mathsf{fact}^\sharp\ x & \Rightarrow & & \\
 & & \mathsf{u_2}^\sharp\ x\ 1\ 1 & [x \equiv x] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_3}^\sharp\ x\ z\ i & [i \leq x \wedge z \equiv z] \\
\mathsf{u_3}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_4}^\sharp\ x\ (z * i)\ i & [x \equiv x \wedge z \equiv z \wedge i \equiv i] \\
\mathsf{u_4}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_2}^\sharp\ x\ z\ (i+1) & [x \equiv x \wedge z \equiv z \wedge i \equiv i] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_5}^\sharp\ x\ z & [\neg(i \leq x) \wedge z \equiv z]
\end{array}
$$

- Automated translations $\Rightarrow$ DPs with many small steps
- Can be hard to analyse!
- **Chaining processor**: remove intermediate symbols $\mathsf{u_1}^\sharp$, $\mathsf{u_3}^\sharp$

Chaining processor

(S)DPs from imperative program [Fuhs, Kop, Nishida, *TOCL '17*]

```
def fact(x):
  z = 1            # fact
  i = 1            # u1
  while i <= x:    # u2
    z = z * i      # u3
    i = i + 1      # u4
                   # u5
```

$$
\begin{array}{llll}
\mathsf{fact}^\sharp\ x & \Rightarrow & & \\
 & & \mathsf{u_2}^\sharp\ x\ 1\ 1 & [x \equiv x] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & & \\
 & & \mathsf{u_4}^\sharp\ x\ (z * i)\ i & [i \leq x \wedge x \equiv x \wedge z \equiv z \wedge i \equiv i] \\
\mathsf{u_4}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_2}^\sharp\ x\ z\ (i+1) & [x \equiv x \wedge z \equiv z \wedge i \equiv i] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_5}^\sharp\ x\ z & [\neg(i \leq x) \wedge z \equiv z]
\end{array}
$$

- Automated translations $\Rightarrow$ DPs with many small steps
- Can be hard to analyse!
- **Chaining processor**: remove intermediate symbols $\mathsf{u_1}^\sharp$, $\mathsf{u_3}^\sharp$, $\mathsf{u_4}^\sharp$

Chaining processor

(S)DPs from imperative program [Fuhs, Kop, Nishida, *TOCL '17*]

```
def fact(x):
  z = 1               # fact
  i = 1               # u1
  while i <= x:       # u2
    z = z * i         # u3
    i = i + 1         # u4
                      # u5
```

$$\mathsf{fact}^\sharp\ x \Rightarrow \mathsf{u_2}^\sharp\ x\ 1\ 1 \quad [x \equiv x]$$

$$\mathsf{u_2}^\sharp\ x\ z\ i \Rightarrow \mathsf{u_2}^\sharp\ x\ z\ (i+1) \quad [i \le x \wedge x \equiv x \wedge z \equiv z \wedge i \equiv i]$$

$$\mathsf{u_2}^\sharp\ x\ z\ i \Rightarrow \mathsf{u_5}^\sharp\ x\ z \quad [\neg(i \le x) \wedge z \equiv z]$$

- Automated translations $\Rightarrow$ DPs with many small steps
- Can be hard to analyse!
- **Chaining processor**: remove intermediate symbols $\mathsf{u_1}^\sharp$, $\mathsf{u_3}^\sharp$, $\mathsf{u_4}^\sharp$

Chaining processor

(S)DPs from imperative program [Fuhs, Kop, Nishida, *TOCL '17*]

```
def fact(x):
  z = 1             # fact
  i = 1             # u1
  while i <= x:     # u2
    z = z * i       # u3
    i = i + 1       # u4
                    # u5
```

$$
\begin{array}{lcll}
\mathsf{fact}^\sharp\ x & \Rightarrow & \mathsf{u_2}^\sharp\ x\ 1\ 1 & [x \equiv x] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_2}^\sharp\ x\ z\ (i+1) & [i \leq x \wedge z \equiv z] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_5}^\sharp\ x\ z & [\neg(i \leq x) \wedge z \equiv z]
\end{array}
$$

- Automated translations ⇒ DPs with many small steps
- Can be hard to analyse!
- **Chaining processor**: remove intermediate symbols $\mathsf{u_1}^\sharp$, $\mathsf{u_3}^\sharp$, $\mathsf{u_4}^\sharp$

Chaining processor

(S)DPs from imperative program [Fuhs, Kop, Nishida, *TOCL '17*]

```
def fact(x):
  z = 1            # fact
  i = 1            # u1
  while i <= x:    # u2
    z = z * i      # u3
    i = i + 1      # u4
                   # u5
```

$$\begin{array}{lcll}
\mathsf{fact}^\sharp\ x & \Rightarrow & \mathsf{u_2}^\sharp\ x\ 1\ 1 & [x \equiv x] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_2}^\sharp\ x\ z\ (i+1) & [i \leq x \wedge z \equiv z] \\
\mathsf{u_2}^\sharp\ x\ z\ i & \Rightarrow & \mathsf{u_5}^\sharp\ x\ z & [\neg(i \leq x) \wedge z \equiv z]
\end{array}$$

- Automated translations $\Rightarrow$ DPs with many small steps
- Can be hard to analyse!
- **Chaining processor**: remove intermediate symbols $\mathsf{u_1}^\sharp$, $\mathsf{u_3}^\sharp$, $\mathsf{u_4}^\sharp$
- Integer mapping processor + graph processor prove termination

Compositional analysis from Universal Computability

- Goal: compositional **open-world** program analysis

Compositional analysis from Universal Computability

- Goal: compositional **open-world** program analysis
- For termination analysis: Universal Computability [Guo, Hagens, Kop, Vale, *MFCS '24*]

Compositional analysis from Universal Computability

- Goal: compositional **open-world** program analysis
- For termination analysis: Universal Computability [Guo, Hagens, Kop, Vale, *MFCS '24*]
- Analyse LCSTRS for use in context of larger program

Compositional analysis from Universal Computability

- Goal: compositional **open-world** program analysis
- For termination analysis: Universal Computability [Guo, Hagens, Kop, Vale, *MFCS '24*]
- Analyse LCSTRS for use in context of larger program
- Usable rules + reduction pair processor available for innermost (and cbv) rewriting!

Implementation

- Implementation in open-source tool Cora: https://github.com/hezzel/cora/
- HORPO as reduction pair
- Z3 as SMT solver

Experiments (1/3)

Experiments using 60 seconds timeout

Experiments (1/3)

Experiments using 60 seconds timeout

275 inputs: integer TRSs + λ-free HO-TRSs from TPDB + own benchmarks

Experiments (1/3)

Experiments using 60 seconds timeout

275 inputs: integer TRSs + λ-free HO-TRSs from TPDB + own benchmarks

Cora (innermost/cbv) v Cora (full) [Guo, Hagens, Kop, Vale, *MFCS '24*]

	Termination			**Universal Computability**		
	Full	Innermost	Call-by-value	Full	Innermost	Call-by-value
Total yes	171	179	182	155	179	182

Experiments (2/3)

117 integer TRSs: Cora v AProVE [Giesl et al, *JAR '17*] [Fuhs et al, *RTA '09*]

	Cora innermost	Cora call-by-value	AProVE innermost
Total yes	72	73	102

Experiments (2/3)

117 integer TRSs: Cora v AProVE [Giesl et al, *JAR '17*] [Fuhs et al, *RTA '09*]

	Cora innermost	Cora call-by-value	AProVE innermost
Total yes	72	73	102

- AProVE has strong reduction pair processor with polynomial interpretations and usable rules
- AProVE can handle rules $\mathsf{f}(x) \to \mathsf{g}(x > 0, x)$, $\mathsf{g}(\mathfrak{t}, x) \to r_1$, $\mathsf{g}(\mathfrak{f}, x) \to r_2$ well

Experiments (3/3)

140 λ-free HO-TRSs: Cora v WANDA [Kop, *FSCD '20*]

	Cora innermost / call-by-value	WANDA full termination
Total yes	79	105

Experiments (3/3)

140 λ-free HO-TRSs: Cora v WANDA [Kop, *FSCD '20*]

	Cora innermost / call-by-value	WANDA full termination
Total yes	79	105

- WANDA: Polynomial interpretations, dynamic DPs, delegation to first-order termination tool, ...

Conclusion: call-by-value termination analysis of LCSTRSs

- Transformation for analysis of LCSTRSs with call-by-value via innermost strategy
- Three new processors: usable rules, reduction pair with temporary argument filtering, chaining
- Improved open-world termination analysis

Conclusion: call-by-value termination analysis of LCSTRSs

- Transformation for analysis of LCSTRSs with call-by-value via innermost strategy
- Three new processors: usable rules, reduction pair with temporary argument filtering, chaining
- Improved open-world termination analysis
- Implementation: `https://github.com/hezzel/cora/`
- Evaluation page: `https://www.cs.ru.nl/~cynthiakop/experiments/fscd25/`

Conclusion: call-by-value termination analysis of LCSTRSs

- Transformation for analysis of LCSTRSs with call-by-value via innermost strategy
- Three new processors: usable rules, reduction pair with temporary argument filtering, chaining
- Improved open-world termination analysis
- Implementation: https://github.com/hezzel/cora/
- Evaluation page: https://www.cs.ru.nl/~cynthiakop/experiments/fscd25/
- FSCD 2025 paper:

An Innermost DP Framework for Constrained Higher-Order Rewriting

Carsten Fuhs
Birkbeck, University of London, UK

Liye Guo
Radboud University, Nijmegen, The Netherlands

Cynthia Kop
Radboud University, Nijmegen, The Netherlands

Outline

1. Static Program Analysis
2. Constrained Term Rewriting
3. Proving Termination of Logically Constrained Simply-typed Term Rewriting Systems
4. Proving Termination of Scala Programs by Constrained Term Rewriting

Background: Stainless verifier for Scala programs

- Open-source deductive verifier for Scala: https://stainless.epfl.ch
- Tutorial: https://epfl-lara.github.io/asplos2022tutorial/
- Verification approach requires functions to be **terminating**

Termination proving in Stainless

```scala
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

Termination proving in Stainless

```
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

```
// Scala function
def f(t: Formula): Boolean = t match

  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

Termination proving in Stainless

```
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

- Manual measure annotations

```
// Scala function
def f(t: Formula): Boolean = t match

  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

Termination proving in Stainless

```
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

- Manual measure annotations

```
// Scala function
def f(t: Formula): Boolean = t match
  decreases(t.size) // written by the user
  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

Termination proving in Stainless

```
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

- Manual measure annotations

```
// Scala function
def f(t: Formula): Boolean = t match

  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

Termination proving in Stainless

```
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

```
// Scala function
def f(t: Formula): Boolean = t match

  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

- Manual measure annotations
- Type-based measure inference [Hamza, Voirol, Kunčak, *OOPSLA '19*]

Termination proving in Stainless

```
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

```
// Scala function
def f(t: Formula): Boolean = t match

  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

- Manual measure annotations
- Type-based measure inference [Hamza, Voirol, Kunčak, *OOPSLA '19*]
- Measure transfer (between similar functions) [Milovančević, Fuhs, Bucev, Kunčak, *iFM '24*]

Termination proving in Stainless

```
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

```
// Scala function
def f(t: Formula): Boolean = t match

  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

- Manual measure annotations
- Type-based measure inference [Hamza, Voirol, Kunčak, *OOPSLA '19*]
- Measure transfer (between similar functions) [Milovančević, Fuhs, Bucev, Kunčak, *iFM '24*]
- **Here:** translate to LCTRS, call termination prover [Milovančević, Fuhs, Kunčak, *WPTE '25*]

Termination proving in Stainless

```
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

```
// Scala function
def f(t: Formula): Boolean = t match

  case True ⇒ true
  case False ⇒ false
  case Not(t) ⇒ if f(t) then false else true
  case Imply(l, r) ⇒ f(Not(l)) || f(r)
```

- Manual measure annotations
- Type-based measure inference [Hamza, Voirol, Kunčak, *OOPSLA '19*]
- Measure transfer (between similar functions) [Milovančević, Fuhs, Bucev, Kunčak, *iFM '24*]
- **Here:** translate to LCTRS, call termination prover [Milovančević, Fuhs, Kunčak, *WPTE '25*]
  $\Rightarrow$ (simplified) LCTRS:

$$
\begin{array}{rll|rll}
\mathsf{f}(\mathsf{True}) & \to \mathfrak{t} & & \mathsf{f}(\mathsf{False}) & \to \mathfrak{f} & \\
\mathsf{f}(\mathsf{Not}(n)) & \to \mathsf{f}_1(\mathsf{Not}(n), \mathsf{f}(n)) & & \mathsf{f}(\mathsf{Imply}(l,r)) & \to \mathsf{f}_2(\mathsf{Imply}(l,r), \mathsf{f}(\mathsf{Not}(l))) & \\
\mathsf{f}_1(\mathsf{Not}(n), tmp_1) & \to \mathfrak{f} & [tmp_1] & \mathsf{f}_2(\mathsf{Imply}(l,r), tmp_2) & \to \mathfrak{t} & [tmp_2] \\
\mathsf{f}_1(\mathsf{Not}(n), tmp_1) & \to \mathfrak{t} & [\neg tmp_1] & \mathsf{f}_2(\mathsf{Imply}(l,r), tmp_2) & \to \mathsf{f}(r) & [\neg tmp_2]
\end{array}
$$

Termination proving in Stainless

```scala
// Scala data type
sealed abstract class Formula
case object True extends Formula
case object False extends Formula
case class Not(p: Formula) extends Formula
case class Imply(l: Formula, r: Formula) extends Formula
```

```scala
// Scala function
def f(t: Formula): Boolean = t match

  case True => true
  case False => false
  case Not(t) => if f(t) then false else true
  case Imply(l, r) => f(Not(l)) || f(r)
```

- Manual measure annotations
- Type-based measure inference [Hamza, Voirol, Kunčak, *OOPSLA '19*]
- Measure transfer (between similar functions) [Milovančević, Fuhs, Bucev, Kunčak, *iFM '24*]
- **Here:** translate to LCTRS, call termination prover [Milovančević, Fuhs, Kunčak, *WPTE '25*]

  $\Rightarrow$ (simplified) LCTRS:

$$\begin{aligned} \mathsf{f}(\mathsf{True}) &\to \mathfrak{t} & \\ \mathsf{f}(\mathsf{Not}(n)) &\to \mathsf{f}_1(\mathsf{Not}(n), \mathsf{f}(n)) & \\ \mathsf{f}_1(\mathsf{Not}(n), tmp_1) &\to \mathfrak{f} & [tmp_1] \\ \mathsf{f}_1(\mathsf{Not}(n), tmp_1) &\to \mathfrak{t} & [\neg tmp_1] \end{aligned}$$

$$\begin{aligned} \mathsf{f}(\mathsf{False}) &\to \mathfrak{f} & \\ \mathsf{f}(\mathsf{Imply}(l,r)) &\to \mathsf{f}_2(\mathsf{Imply}(l,r), \mathsf{f}(\mathsf{Not}(l))) & \\ \mathsf{f}_2(\mathsf{Imply}(l,r), tmp_2) &\to \mathfrak{t} & [tmp_2] \\ \mathsf{f}_2(\mathsf{Imply}(l,r), tmp_2) &\to \mathsf{f}(r) & [\neg tmp_2] \end{aligned}$$

$\Rightarrow$ easy to prove terminating

Experiments with Stainless and AProVE

Benchmarks from work on autograding by equivalence proving
[Milovančević, Kunčak, *PLDI '23*; Milovančević et al, *ESOP '25*]

Experiments with Stainless and AProVE

Benchmarks from work on autograding by equivalence proving
[Milovančević, Kunčak, *PLDI '23*; Milovančević et al, *ESOP '25*]

Name	LOC	#	Inference	Transfer	**LCTRS**	Total Proven
formula	59	37	0	27	**24**	28
sigma	10	704	0	678	**0**	678
prime	21	22	0	5	**14**	14
gcd	9	41	0	22	**15**	27

Experiments with Stainless and AProVE

Benchmarks from work on autograding by equivalence proving
[Milovančević, Kunčak, *PLDI '23*; Milovančević et al, *ESOP '25*]

Name	LOC	#	Inference	Transfer	**LCTRS**	Total Proven
formula	59	37	0	27	**24**	28
sigma	10	704	0	678	**0**	678
prime	21	22	0	5	**14**	14
gcd	9	41	0	22	**15**	27

- Benchmark selection cannot be handled by measure inference
- *sigma* benchmark has higher-order arguments, outside of the scope of AProVE's Integer TRSs
- Measure transfer and termination proofs via constrained rewriting complement each other well

Limitations / future work

- Translation represents bounded Scala integers as unbounded. But:

```
def overflow_fun(i: Int, n: Int): Int =
  if i ≤ n then overflow_fun(i + 1, n) else i
```

Limitations / future work

- Translation represents bounded Scala integers as unbounded. But:

  def overflow_fun(i: Int, n: Int): Int =
    if i $\leq$ n **then** overflow_fun(i + 1, n) **else** i

- Translation does not yet support higher-order functions (solution: use LCSTRSs)

Limitations / future work

- Translation represents bounded Scala integers as unbounded. But:

  def overflow_fun(i: Int, n: Int): Int =
    if i $\leq$ n **then** overflow_fun(i + 1, n) **else** i

- Translation does not yet support higher-order functions (solution: use LCSTRSs)
- Support for lazy values missing

Limitations / future work

- Translation represents bounded Scala integers as unbounded. But:

```
def overflow_fun(i: Int, n: Int): Int =
  if i ≤ n then overflow_fun(i + 1, n) else i
```

- Translation does not yet support higher-order functions (solution: use LCSTRSs)
- Support for lazy values missing
- Actual encoding of Scala's line-by-line pattern matching more complex: patterns may overlap. Solution: one function symbol per Scala pattern case:

$$\begin{aligned} \mathsf{f}_{12}(s, \mathsf{True}) &\to \mathsf{f}_{21}(s, \mathsf{True}) \\ \mathsf{f}_{12}(s, \mathsf{False}) &\to \mathsf{f}_{21}(s, \mathsf{False}) \\ \mathsf{f}_{12}(s, \mathsf{Not}(s\_p)) &\to \mathsf{f}_{21}(s, \mathsf{Not}(s\_p)) \\[1em] \mathsf{f}_{12}(s, \mathsf{Imply}(s\_l, s\_r)) &\to \mathsf{f}_{13}(s, \mathsf{Imply}(s\_l, s\_r)) \end{aligned}$$

$\Rightarrow$ optimise via pattern differences [Nishida, Kojima, Nakamura, *FroCoS '25*]

Limitations / future work

- Translation represents bounded Scala integers as unbounded. But:

  def overflow_fun(i: Int, n: Int): Int =
  if i $\leq$ n **then** overflow_fun(i + 1, n) **else** i

- Translation does not yet support higher-order functions (solution: use LCSTRSs)
- Support for lazy values missing
- Actual encoding of Scala's line-by-line pattern matching more complex: patterns may overlap. Solution: one function symbol per Scala pattern case:

$$\mathsf{f}_{12}(s, \mathsf{True}) \to \mathsf{f}_{21}(s, \mathsf{True})$$
$$\mathsf{f}_{12}(s, \mathsf{False}) \to \mathsf{f}_{21}(s, \mathsf{False})$$
$$\mathsf{f}_{12}(s, \mathsf{Not}(s\_p)) \to \mathsf{f}_{21}(s, \mathsf{Not}(s\_p))$$

$$\mathsf{f}_{12}(s, \mathsf{Imply}(s\_l, s\_r)) \to \mathsf{f}_{13}(s, \mathsf{Imply}(s\_l, s\_r))$$

  $\Rightarrow$ optimise via pattern differences [Nishida, Kojima, Nakamura, *FroCoS '25*]

- ...

Conclusion: from Scala to LCTRSs for termination analysis

- Integration of translation to LCTRSs into the Stainless analysis pipeline
- Improvement over existing termination proof techniques in Stainless
- Work in progress

Conclusion: from Scala to LCTRSs for termination analysis

- Integration of translation to LCTRSs into the Stainless analysis pipeline
- Improvement over existing termination proof techniques in Stainless
- Work in progress
- WPTE 2025 paper:

Proving Termination of Scala Programs by Constrained Term Rewriting

Dragana Milovančević
EPFL, Station 14
CH-1015 Lausanne, Switzerland
dragana.milovancevic@epfl.ch

Carsten Fuhs
Birkbeck, University of London
United Kingdom
c.fuhs@bbk.ac.uk

Viktor Kunčak
EPFL, Station 14
CH-1015 Lausanne, Switzerland
viktor.kuncak@epfl.ch

Take Home Messages

- **Static analysis workflow:**

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ Result for $\mathcal{R}_P$ + proof

Take Home Messages

- **Static analysis workflow:**

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ Result for $\mathcal{R}_P$ + proof

- **Constrained rewriting:** a versatile intermediate representation for static program analysis

Take Home Messages

- **Static analysis workflow:**

$$\boxed{\text{Program } P \text{ in language } L} \longrightarrow \boxed{\text{Intermediate representation } \mathcal{R}_P} \longrightarrow \boxed{\text{Result for } \mathcal{R}_P + \text{proof}}$$

- **Constrained rewriting:** a versatile intermediate representation for static program analysis
- **Formalisms:** LC(S)TRSs = Logically Constrained (Simply-typed) Term Rewriting Systems

Take Home Messages

- **Static analysis workflow:**

$$\boxed{\text{Program } P \text{ in language } L} \longrightarrow \boxed{\text{Intermediate representation } \mathcal{R}_P} \longrightarrow \boxed{\text{Result for } \mathcal{R}_P + \text{proof}}$$

- **Constrained rewriting:** a versatile intermediate representation for static program analysis
- **Formalisms:** LC(S)TRSs = Logically Constrained (Simply-typed) Term Rewriting Systems
- **Properties:** Termination, complexity, equivalence, safety/reachability, confluence, ...

Take Home Messages

- **Static analysis workflow:**

$$\boxed{\text{Program } P \text{ in language } L} \longrightarrow \boxed{\text{Intermediate representation } \mathcal{R}_P} \longrightarrow \boxed{\text{Result for } \mathcal{R}_P + \text{proof}}$$

- **Constrained rewriting:** a versatile intermediate representation for static program analysis
- **Formalisms:** LC(S)TRSs = Logically Constrained (Simply-typed) Term Rewriting Systems
- **Properties:** Termination, complexity, equivalence, safety/reachability, confluence, . . .
- **Tools:** Cora, CRaris, crest, Ctrl, RMT, TcT, . . .

Take Home Messages

- **Static analysis workflow:**

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ Result for $\mathcal{R}_P$ + proof

- **Constrained rewriting:** a versatile intermediate representation for static program analysis
- **Formalisms:** LC(S)TRSs = Logically Constrained (Simply-typed) Term Rewriting Systems
- **Properties:** Termination, complexity, equivalence, safety/reachability, confluence, ...
- **Tools:** Cora, CRaris, crest, Ctrl, RMT, TcT, ...

Can *your* programs be translated to LC(S)TRSs?

Take Home Messages

- **Static analysis workflow:**

Program P in language L $\longrightarrow$ Intermediate representation $\mathcal{R}_P$ $\longrightarrow$ Result for $\mathcal{R}_P$ + proof

- **Constrained rewriting:** a versatile intermediate representation for static program analysis
- **Formalisms:** LC(S)TRSs = Logically Constrained (Simply-typed) Term Rewriting Systems
- **Properties:** Termination, complexity, equivalence, safety/reachability, confluence, ...
- **Tools:** Cora, CRaris, crest, Ctrl, RMT, TcT, ...

Can *your* programs be translated to LC(S)TRSs?

Thanks a lot for your attention!