Proving Program Termination via Term Rewriting

Carsten Fuhs

Birkbeck, University of London

Advanced Course at the International School on Rewriting 2017
Eindhoven, The Netherlands
July 2017
1. Overview

2. Termination Analysis of Term Rewriting with Dependency Pairs

3. Haskell: a Pure Functional Language with Lazy Evaluation

4. Java: an Object-Oriented Imperative Language with Side Effects
Overview

Termination Analysis of Term Rewriting with Dependency Pairs

Haskell: a Pure Functional Language with Lazy Evaluation

Java: an Object-Oriented Imperative Language with Side Effects
Automated Termination Provers for Term Rewrite Systems

- AProVE (Aachen, ...)
- CiME3 (Paris)
- HOT (Cachan)
- Jambox (Amsterdam)
- Matchbox (Leipzig)
- Mu-Term (Valencia)
- MultumNonMulta (Kassel)
- NaTT (Innsbruck)
- THOR (Barcelona)
- TORPA (Eindhoven)
- TTT2 (Innsbruck)
- VMTL (Vienna)
- Wanda (Copenhagen)
- ...

- Powerful push-button termination analysis tools for Term Rewrite Systems (TRSs)
- Development spurred by annual International Termination Competition (termCOMP) since 2003
- termCOMP initially for term rewriting, now also C, Java, Haskell, Prolog
- Can we use tools for TRSs also for programming languages?
Automated Termination Provers for Term Rewrite Systems

- AProVE (Aachen, ...)
- CiME3 (Paris)
- HOT (Cachan)
- Jambox (Amsterdam)
- Matchbox (Leipzig)
- Mu-Term (Valencia)
- MultumNonMulta (Kassel)
- NaTT (Innsbruck)
- THOR (Barcelona)
- TORPA (Eindhoven)
- TTT2 (Innsbruck)
- VMTL (Vienna)
- Wanda (Copenhagen)

- Powerful push-button termination analysis tools for Term Rewrite Systems (TRSs)
- Development spurred by annual International Termination Competition (termCOMP) since 2003
- termCOMP initially for term rewriting, now also C, Java, Haskell, Prolog
- Can we use tools for TRSs also for programming languages?
Automated Termination Provers for Term Rewrite Systems

- AProVE (Aachen, \ldots)
- CiME3 (Paris)
- HOT (Cachan)
- Jambox (Amsterdam)
- Matchbox (Leipzig)
- Mu-Term (Valencia)
- MultumNonMulta (Kassel)
- NaTT (Innsbruck)
- THOR (Barcelona)
- TORPA (Eindhoven)
- TTT2 (Innsbruck)
- VMTL (Vienna)
- Wanda (Copenhagen)
- \ldots

- Powerful push-button termination analysis tools for Term Rewrite Systems (TRSs)
- Development spurred by annual International Termination Competition (termCOMP) since 2003
- termCOMP initially for term rewriting, now also C, Java, Haskell, Prolog
- Can we use tools for TRSs also for programming languages?
Termination Analysis for Programs – Why?

- push-button analysis
- does not need separate specification (hard to get right)
- termination is in most cases a desirable property
- non-termination can be security issue (Denial of Service)
- in 2011: PHP and Java issues with floating-point number parser
- value of termination analysis recognized by industry → Microsoft’s Terminator project
Termination Analysis for Programs – Why?

- push-button analysis
- does not need separate specification (hard to get right)
- termination is in most cases a desirable property
- non-termination can be security issue (Denial of Service)
- in 2011: PHP and Java issues with floating-point number parser
- value of termination analysis recognized by industry
  → Microsoft’s Terminator project
term rewriting is Turing-complete

- can represent inductive data structures (trees) in a natural way

**Idea 1**: port techniques from TRSs to each programming language

→ but: lots of repeated work

**Idea 2**: two-stage approach

- **front-end** for language-specific aspects, extracts TRS such that termination of TRS implies termination of the program
- **back-end**: reuse optimized off-the-shelf termination prover for TRSs

**This course**: How can we construct such a front-end?

- look at general principle
- look at two concrete programming languages as examples
  - Haskell (functional, lazy)
  - Java (imperative, object-oriented)
How can Term Rewriting Contribute?

- term rewriting is Turing-complete
- can represent inductive data structures (trees) in a natural way

**Idea 1**: port techniques from TRSs to each programming language
→ but: lots of repeated work

**Idea 2**: two-stage approach
- **front-end** for language-specific aspects, extracts TRS such that termination of TRS implies termination of the program
- **back-end**: reuse optimized off-the-shelf termination prover for TRSs

**This course**: How can we construct such a front-end?
- look at general principle
- look at two concrete programming languages as examples
  - Haskell (functional, lazy)
  - Java (imperative, object-oriented)
How can Term Rewriting Contribute?

- term rewriting is Turing-complete
- can represent inductive data structures (trees) in a natural way

**Idea 1:** port techniques from TRSs to each programming language
→ but: lots of repeated work

**Idea 2:** two-stage approach
- **front-end** for language-specific aspects, extracts TRS such that termination of TRS implies termination of the program
- **back-end:** reuse optimized off-the-shelf termination prover for TRSs

This course: How can we construct such a front-end?
- look at general principle
- look at two concrete programming languages as examples
  - Haskell (functional, lazy)
  - Java (imperative, object-oriented)
term rewriting is Turing-complete

can represent inductive data structures (trees) in a natural way

Idea 1: port techniques from TRSs to each programming language
→ but: lots of repeated work

Idea 2: two-stage approach
  - **front-end** for language-specific aspects, extracts **TRS** such that termination of TRS implies termination of the program
  - **back-end**: reuse optimized off-the-shelf termination prover for TRSs

This course: How can we construct such a front-end?
  - look at general principle
  - look at two concrete programming languages as examples
    - **Haskell** (functional, lazy)
    - **Java** (imperative, object-oriented)
1 Overview

2 Termination Analysis of Term Rewriting with Dependency Pairs

3 Haskell: a Pure Functional Language with Lazy Evaluation

4 Java: an Object-Oriented Imperative Language with Side Effects
Term rewriting: Evaluate terms by applying rules from $\mathcal{R}$

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

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

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

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

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

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

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

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

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

Show termination using Dependency Pairs
Example (Division)

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

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

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

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

Dependency Pairs (DPs) [Arts, Giesl, TCS ’00]

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

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

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

Dependency Pairs (DPs) [Arts, Giesl, TCS '00]

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

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

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

Dependency Pairs (DPs) [Arts, Giesl, TCS ’00]

- For TRS \( R \) build dependency pairs \( P \) (∼ function calls)
- Show: No ∞ call sequence with \( P \) (eval of \( P \)'s args via \( R \))
- Dependency Pair Framework [Giesl et al, JAR ’06] (simplified):
  while \( (P \neq \emptyset) \) do
Dependency Pairs (DPs) [Arts, Giesl, TCS ’00]

- For TRS \( R \) build dependency pairs \( P \) (\( \sim \) function calls)
- Show: No \( \infty \) call sequence with \( P \) (eval of \( P \)'s args via \( R \))
- Dependency Pair Framework [Giesl et al, JAR ’06] (simplified):
  while \( (P \neq \emptyset) \) do
    - find reduction pair \( (\preceq, \succ) \) with \( P \cup R \subseteq \preceq \)
Dependency Pairs (DPs) [Arts, Giesl, TCS ’00]

- For TRS $R$ build dependency pairs $P$ ($\sim$ function calls)
- Show: No $\infty$ call sequence with $P$ (eval of $P$’s args via $R$)
- Dependency Pair Framework [Giesl et al, JAR ’06] (simplified):
  while ($P \neq \emptyset$) do
    - find reduction pair ($\preceq$, $\succ$) with $P \cup R \subseteq \preceq$
    - delete $s \rightarrow t$ with $s \succ t$ from $P$ ($\succ$ well founded, not monotonic)
Dependency Pairs (DPs) [Arts, Giesl, TCS ’00]

- For TRS $\mathcal{R}$ build dependency pairs $\mathcal{P}$ ($\sim$ function calls)
- Show: No $\infty$ call sequence with $\mathcal{P}$ (eval of $\mathcal{P}$’s args via $\mathcal{R}$)
- Dependency Pair Framework [Giesl et al, JAR ’06] (simplified):
  while ($\mathcal{P} \neq \emptyset$) do
    - find reduction pair ($\preceq$, $\succ$) with $\mathcal{P} \cup \mathcal{R} \subseteq \preceq$
    - delete $s \rightarrow t$ with $s \succ t$ from $\mathcal{P}$ ($\succ$ well founded, *not monotonic*)
- Find ($\preceq$, $\succ$) automatically via SAT and SMT solving
Example (Constraints for Division)

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

\[ P = \begin{cases} \min^\#(s(x), s(y)) & \equiv \min^\#(x, y) \\ \text{quot}^\#(s(x), s(y)) & \equiv \min^\#(x, y) \\ \text{quot}^\#(s(x), s(y)) & \equiv \text{quot}^\#(\min(x, y), s(y)) \end{cases} \]

Use polynomial interpretation \([ \cdot ]\) over \(\mathbb{N}\) [Lankford '75] with

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

\(\bowtie\) (\(\preceq, \succeq\)) induced by \([ \cdot ]\) solves all term constraints

\(\bowtie\ P = \emptyset\)

\(\bowtie\) termination of division algorithm proved
Use polynomial interpretation $[\cdot]$ over $\mathbb{N}$ [Lankford ’75] with

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

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

$$[0] = 0 \quad \text{[s]}(x_1) = x_1 + 1$$

$\bowtie (\leadsto, \rhd)$ induced by $[\cdot]$ solves all term constraints

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

$\bowtie$ termination of division algorithm proved
Use polynomial interpretation \([ \cdot ]\) over \(\mathbb{N}\) [Lankford '75] with

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

\(\bowtie (\equiv, \succ)\) induced by \([ \cdot ]\) solves all term constraints

\(\bowtie P = \emptyset\)

\(\bowtie\) termination of division algorithm proved
execute program \textit{symbolically} from initial states of the program, handle language peculiarities here

\begin{verbatim}
f: if ...
    ...
    else
        ...
        g: while ...
    ...
\end{verbatim}
execute program **symbolically** from initial states of the program, handle language peculiarities here

f: if ...
    ...
else
    ...
    g: while ...
    ...
init(...)

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

extract TRS from cycles in the representation

if TRS terminates  \implies \text{any concrete program execution can use cycles only finitely often}  \implies \text{the program must terminate}
From Program to Term Rewriting, high-level

- execute program symbolically from initial states of the program, handle language peculiarities here

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

\rightarrow
\begin{align*}
\text{init}(...) \\
\downarrow \\
\text{f}(...) 
\end{align*}
```
From Program to Term Rewriting, high-level

- execute program **symbolically** from initial states of the program, handle language peculiarities here

\[ f : \text{if } \ldots \]
\[ \ldots \]
\[ \text{else} \]
\[ \ldots \]
\[ g : \text{while } \ldots \]
\[ \ldots \]

\[ \text{init}(\ldots) \]
\[ \downarrow \]
\[ f(\ldots) \]
\[ \ldots \]
\[ g(s) \]
execute program **symbolically** from initial states of the program, handle language peculiarities here

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

![Diagram](image)
execute program **symbolically** from initial states of the program, handle language peculiarities here

- use **generalization** of program states, get over-approximation of all possible program runs (≈ control-flow graph with extra info)
- related: Abstract Interpretation [Cousot and Cousot, *POPL '77*]
execute program **symbolically** from initial states of the program, handle language peculiarities here

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

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

**extract TRS from cycles** in the representation

\[
\begin{align*}
\text{f: if } & \ldots \\
& \ldots \\
\text{else } & \\
& \ldots \\
\text{g: while } & \ldots \\
& \ldots
\end{align*}
\]
execute program **symbolically** from initial states of the program, handle language peculiarities here

use **generalization** of program states, get over-approximation of all possible program runs ($\approx$ control-flow graph with extra info)

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

**extract TRS** from cycles in the representation

if TRS terminates

$\Rightarrow$ any **concrete program execution** can use cycles only finitely often

$\Rightarrow$ the program **must terminate**

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

$\text{init}(\ldots)$

$\downarrow$

$\text{f}(\ldots)$

$\quad$...$\quad$

$\text{g}(\vec{s}) \quad \text{instance of } \text{g}(\vec{s})$

$\ldots$

$\quad$...$\quad$

$\text{g}(\vec{t})$

$\quad$...$\quad$

$\text{g}(\vec{t})$
1 Overview

2 Termination Analysis of Term Rewriting with Dependency Pairs

3 Haskell: a Pure Functional Language with Lazy Evaluation

4 Java: an Object-Oriented Imperative Language with Side Effects
Haskell 98

- Widely used functional programming language
- Goal: analyze termination, reuse techniques for term rewrite systems

Approach

[Giesl, Raffelsieper, Schneider-Kamp, Swiderski, Thiemann, TOPLAS ’11]

- Translate from Haskell 98 to TRS
- Prove termination of the TRS using standard techniques for TRSs
  \[ \Rightarrow \text{Implies termination of the Haskell program!} \]
Haskell 98
- Widely used functional programming language
- Goal: analyze termination, reuse techniques for term rewrite systems

Approach
[Giesl, Raffelsieper, Schneider-Kamp, Swiderski, Thiemann, TOPLAS ’11]
- Translate from Haskell 98 to TRS
- Prove termination of the TRS using standard techniques for TRSs
⇒ Implies termination of the Haskell program!
From Haskell to TRSs

Challenges

- higher-order: functional variables, λ-abstractions, ...
  But: Standard framework for TRSs works on first-order terms

- lazy evaluation
  But: Standard TRS techniques consider all evaluation strategies

- polymorphic types
  But: TRSs are untyped

- usually not all Haskell functions terminate → ∞ data (streams)
  But: TRS techniques analyze termination of all terms
Syntax of Haskell

Data Structures

- data Nat = Z | S Nat
  - type constructor: Nat
  - data constructors: Z :: Nat, S :: Nat → Nat

- data List a = Nil | Cons a (List a)
  - type constructor: List of arity 1
  - data constructors: Nil :: List a, Cons :: a → (List a) → (List a)
Data Structures

- `data Nat = Z | S Nat`
  - type constructor: `Nat` of arity 0
  - data constructors: `Z :: Nat, S :: Nat → Nat`

- `data List a = Nil | Cons a (List a)`
  - type constructor: `List` of arity 1
  - data constructors: `Nil :: List a, Cons :: a → (List a) → (List a)`
Syntax of Haskell

Data Structures

- **data Nat = Z | S Nat**
  - type constructor: `Nat` of arity 0
  - data constructors: `Z :: Nat, S :: Nat → Nat`

- **data List a = Nil | Cons a (List a)**
  - type constructor: `List` of arity 1
  - data constructors: `Nil :: List a, Cons :: a → (List a) → (List a)`

Terms (well-typed)

- Variables: `x, y, ...`
- Function Symbols: constructors `(Z, S, Nil, Cons)` & defined `(from, take)`
- Applications `(t_1 t_2)`
  - `S Z` represents number 1
  - `Cons x Nil ≡ (Cons x) Nil` represents `[x]`
Syntax of Haskell

Data Structures

- data Nat = Z | S Nat
  - type constructor: Nat of arity 0
  - data constructors: Z :: Nat, S :: Nat → Nat
- data List a = Nil | Cons a (List a)
  - type constructor: List of arity 1
  - data constructors: Nil :: List a, Cons :: a → (List a) → (List a)

Types

- Type Variables: a, b, ...
- Applications of type constructors to types: List Nat, a → (List a), ...

S Z has type Nat
Cons x Nil has type List a
Syntax of Haskell

Data Structures
- `data Nat = Z | S Nat`
  - type constructor: `Nat` of arity 0
  - data constructors: `Z :: Nat`, `S :: Nat -> Nat`
- `data List a = Nil | Cons a (List a)`
  - type constructor: `List` of arity 1
  - data constructors: `Nil :: List a`, `Cons :: a -> (List a) -> (List a)`

Function Declarations (example)

```
from x = Cons x (from (S x))
```
```
take Z xs = Nil
```
```
take n Nil = Nil
```
```
take (S n) (Cons x xs) = Cons x (take n xs)
```
```
from :: Nat -> List Nat
```
```
take :: Nat -> (List a) -> (List a)
```
```
from x ≡ [x, x + 1, x + 2, ...]
```
```
take n [x_1, ..., x_n, ...] ≡ [x_1, ..., x_n]
```

Syntax of Haskell

Function Declarations (general)

\[ f \ell_1 \ldots \ell_n = r \]

- \( f \) is defined function symbol
- \( n \) is arity of \( f \)
- \( r \) is arbitrary term
- \( \ell_1 \ldots \ell_n \) are linear patterns (terms from constructors and variables)

Function Declarations (example)

from \( x = \text{Cons} \ x \ \text{from} \ (\text{S} \ x) \)  
\( \text{take} \ Z \ xs = \text{Nil} \)
\( \text{take} \ n \ \text{Nil} = \text{Nil} \)
\( \text{take} \ (\text{S} \ n) \ (\text{Cons} \ x \ xs) = \text{Cons} \ x \ (\text{take} \ n \ xs) \)

from :: \text{Nat} \to \text{List Nat}  
\( \text{take} :: \text{Nat} \to (\text{List} \ a) \to (\text{List} \ a) \)

from \( x \equiv \ [x, x + 1, x + 2, \ldots] \)  
\( \text{take} \ n \ [x_1, \ldots, x_n, \ldots] \equiv \ [x_1, \ldots, x_n] \)
Syntax of Haskell

Approach also works with

- built-in data structures
- type classes

All other Haskell constructs are eliminated by automatic transformations!

- lambda abstractions

- Conditions
- Local Declarations
- ...
Approach also works with
- built-in data structures
- type classes

All other Haskell constructs are eliminated by automatic transformations!
- lambda abstractions
  
  replace $\lambda u m \rightarrow \text{take } u \ (\text{from } m)$
  
  by $f$
  
  where $f \ u \ m = \text{take } u \ (\text{from } m)$

- Conditions
- Local Declarations
- ...
Approach also works with

- built-in data structures
- type classes

All other Haskell constructs are eliminated by automatic transformations!

- lambda abstractions

  replace \( m \to \text{take } u \text{ (from } m) \) by \( f \ u \)

  where \( f \ u \ m = \text{take } u \text{ (from } m) \)

- Conditions
- Local Declarations
- …
Syntax of Haskell

Approach also works with
- built-in data structures
- type classes

All other Haskell constructs are eliminated by automatic transformations!
- lambda abstractions
  - replace $\lambda t_1 \ldots t_n \rightarrow t$ with free variables $x_1, \ldots, x_m$
  - by $f x_1 \ldots x_m$
  - where $f x_1 \ldots x_m t_1 \ldots t_n = t$

- Conditions
- Local Declarations
- ...
Syntax of Haskell

Approach also works with
  - built-in data structures
  - type classes

All other Haskell constructs are eliminated by automatic transformations!
  - lambda abstractions
    replace \( t_1 \ldots t_n \rightarrow t \) with free variables \( x_1, \ldots, x_m \)
    by \( f x_1 \ldots x_m \)
    where \( f x_1 \ldots x_m t_1 \ldots t_n = t \)

- Conditions
- Local Declarations
- ...
from $x = \text{Cons} \ x \ (\text{from} \ (S \ x))$

take $Z \ xs = \text{Nil}$

take $n \ \text{Nil} = \text{Nil}$

take $(S \ n) \ (\text{Cons} \ x \ xs) = \text{Cons} \ x \ (\text{take} \ n \ xs)$

- **Evaluation Relation** $\rightarrow_H$

  from $Z$
from $x = \text{Cons } x (\text{from } (S \ x))$

take $Z \ xs = \text{Nil}$

take $n \ \text{Nil} = \text{Nil}$

take $(S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs)$

Evaluation Relation $\rightarrow_H$

from $Z$

evaluation position
from \( x = \text{Cons} \ x \ (\text{from} \ (S \ x)) \)  
take \( Z \ xs = \text{Nil} \)  
take \( n \ \text{Nil} = \text{Nil} \)  
take \( (S \ n) \ (\text{Cons} \ x \ xs) = \text{Cons} \ x \ (\text{take} \ n \ xs) \)

- **Evaluation Relation** \( \rightarrow_H \)

  \[
  \text{from} \ Z  \\
  \rightarrow_H \ \text{Cons} \ Z \ (\text{from} \ (S \ Z))
  \]

  *evaluation position*
from $x = \text{Cons } x (\text{from } (S \ x))$

take $Z \ xs = \text{Nil}$

take $n \ \text{Nil} = \text{Nil}$

take $(S \ n) (\text{Cons } x \ xs) = \text{Cons } x (\text{take } n \ xs)$

- Evaluation Relation $\rightarrow_{H}$

  from $Z$

  $\rightarrow_{H} \ \text{Cons } Z (\text{from } (S \ Z))$

  evaluation position
from \( x = \text{Cons} \ x \ (\text{from} \ (S \ x)) \)  
\( \text{take} \ Z \ xs = \text{Nil} \)  
\( \text{take} \ n \ \text{Nil} = \text{Nil} \)  
\( \text{take} \ (S \ n) \ (\text{Cons} \ x \ xs) = \text{Cons} \ x \ (\text{take} \ n \ xs) \)

- **Evaluation Relation** \( \rightarrow_H \)
  
  from \( Z \)
  
  \( \rightarrow_H \ \text{Cons} \ Z \ (\text{from} \ (S \ Z)) \)  
  
  \( \rightarrow_H \ \text{Cons} \ Z \ (\text{Cons} \ (S \ Z) \ (\text{from} \ (S \ (S \ Z)))) \)  
  
  *evaluation position*
from $x = \text{Cons } x (\text{from } (S \ x))$

\[
\begin{align*}
\text{take } Z \ xs &= \text{Nil} \\
\text{take } n \ \text{Nil} &= \text{Nil} \\
\text{take } (S \ n) (\text{Cons } x \ xs) &= \text{Cons } x (\text{take } n \ xs)
\end{align*}
\]

- **Evaluation Relation** $\rightarrow_H$

\[
\begin{align*}
\text{from } Z \\
\rightarrow_H \text{Cons } Z (\text{from } (S \ Z)) \\
\rightarrow_H \text{Cons } Z (\text{Cons } (S \ Z) \ (\text{from } (S (S \ Z)))) &\quad \text{evaluation position}
\end{align*}
\]
from $x = \text{Cons } x (\text{from } (S x))$  
\begin{align*}
take Z xs &= \text{Nil} \\
take n \text{Nil} &= \text{Nil} \\
take (S n) (\text{Cons } x xs) &= \text{Cons } x (\text{take } n xs)
\end{align*}

- **Evaluation Relation** $\rightarrow_H$

  \begin{align*}
  \text{from } Z \\
  \rightarrow_H \text{Cons } Z (\text{from } (S Z)) \\
  \rightarrow_H \text{Cons } Z (\text{Cons } (S Z) (\text{from } (S (S Z)))) & \quad \text{evaluation position} \\
  \rightarrow_H \ldots
\end{align*}
Semantics of Haskell

\[ \text{from } x = \text{Cons } x (\text{from } (S \ x)) \]
\[ \text{take } Z \ xs = \text{Nil} \]
\[ \text{take } n \ \text{Nil} = \text{Nil} \]
\[ \text{take } (S \ n) (\text{Cons } x \ xs) = \text{Cons } x (\text{take } n \ xs) \]

- **Evaluation Relation** \( \rightarrow_H \)

  \[
  \text{from } m \\
  \rightarrow_H \text{Cons } m (\text{from } (S \ m)) \\
  \rightarrow_H \text{Cons } m (\text{Cons } (S \ m) (\text{from } (S (S \ m)))) \\
  \rightarrow_H \ldots
  \]
from $x = \text{Cons} \ x \ (\text{from} \ (\text{S} \ x))$

\[
\begin{align*}
\text{take} \ Z \ xs & = \text{Nil} \\
\text{take} \ n \ \text{Nil} & = \text{Nil} \\
\text{take} \ (\text{S} \ n) \ (\text{Cons} \ x \ xs) & = \text{Cons} \ x \ (\text{take} \ n \ xs)
\end{align*}
\]

**Evaluation Relation** $\rightarrow_H$

\[
\text{take} \ (\text{S} \ Z) \ (\text{from} \ m)
\]
from $x = \text{Cons } x (\text{from } (S x))$

\begin{align*}
\text{take } Z \, xs &= \text{Nil} \\
\text{take } n \, \text{Nil} &= \text{Nil} \\
\text{take } (S \, n) (\text{Cons } x \, xs) &= \text{Cons } x (\text{take } n \, xs)
\end{align*}

Evaluation Relation $\rightarrow_H$

take $(S \, Z) (\text{from } m)$

\textit{evaluation position}
from \( x = \text{Cons} \ x \ (\text{from} \ (S \ x)) \) 

\( \text{take} \ Z \ xs = \text{Nil} \)

\( \text{take} \ n \ \text{Nil} = \text{Nil} \)

\( \text{take} \ (S \ n) \ (\text{Cons} \ x \ xs) = \text{Cons} \ x \ (\text{take} \ n \ xs) \)

**Evaluation Relation** \( \rightarrow^H \)

\( \text{take} \ (S \ Z) \ (\text{from} \ m) \)

\( \rightarrow^H \text{take} \ (S \ Z) \ (\text{Cons} \ m \ (\text{from} \ (S \ m))) \)

evaluation position
from $x = \text{Cons } x (\text{from} (S \ x))$

take $Z \ xs = \text{Nil}$

take $n \ \text{Nil} = \text{Nil}$

take $(S \ n) (\text{Cons } x \ xs) = \text{Cons } x (\text{take } n \ xs)$

- **Evaluation Relation** $\rightarrow_H$

  $\text{take} \ (S \ Z) \ (\text{from} \ m) \rightarrow_H \ \text{take} \ (S \ Z) \ (\text{Cons } m \ (\text{from} \ (S \ m)))$

  *evaluation position*
Semantics of Haskell

\[ \text{from } x = \text{Cons } x \left( \text{from } (\text{S } x) \right) \]

\[ \text{take } Z \, xs = \text{Nil} \]

\[ \text{take } n \, \text{Nil} = \text{Nil} \]

\[ \text{take } (\text{S } n) \, (\text{Cons } x \, xs) = \text{Cons } x \left( \text{take } n \, xs \right) \]

**Evaluation Relation** \( \rightarrow^*_H \)

\[ \text{take } (\text{S } Z) \left( \text{from } m \right) \]

\[ \rightarrow^*_H \, \text{take } (\text{S } Z) \left( \text{Cons } m \left( \text{from } (\text{S } m) \right) \right) \text{ evaluation position} \]

\[ \rightarrow^*_H \, \text{Cons } m \left( \text{take } Z \left( \text{from } (\text{S } m) \right) \right) \]
from $x = \text{Cons} \, x \,(\text{from} \,(S \, x))$  
\[\text{take} \, Z \, x \, s = \text{Nil}\]
\[\text{take} \, n \, \text{Nil} = \text{Nil}\]
\[\text{take} \,(S \, n) \,(\text{Cons} \, x \, x \, s) = \text{Cons} \, x \,(\text{take} \, n \, x \, s)\]

**Evaluation Relation** $\rightarrow_H$

\[\text{take} \,(S \, Z) \,(\text{from} \, m)\]
\[\rightarrow_H \text{take} \,(S \, Z) \,(\text{Cons} \, m \,(\text{from} \,(S \, m)))\]
\[\rightarrow_H \text{Cons} \, m \,(\text{take} \, Z \,(\text{from} \,(S \, m)))\]  

*evaluation position*
from $x = \text{Cons} \; x \; (\text{from} \; (S \; x))$

take $Z \; xs = \text{Nil}$

take $n \; \text{Nil} = \text{Nil}$

take $(S \; n) \; (\text{Cons} \; x \; xs) = \text{Cons} \; x \; (\text{take} \; n \; xs)$

- **Evaluation Relation** $\rightarrow_H$

  $\rightarrow_H \text{take} \; (S \; Z) \; (\text{from} \; m)$

  $\rightarrow_H \text{take} \; (S \; Z) \; (\text{Cons} \; m \; (\text{from} \; (S \; m)))$

  $\rightarrow_H \text{Cons} \; m \; (\text{take} \; Z \; (\text{from} \; (S \; m)))$

  $\rightarrow_H \text{Cons} \; m \; \text{Nil}$

  *evaluation position*
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of ground term $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow^*_H \ldots$
  - if $t \rightarrow^*_H (f \, t_1 \ldots t_n)$, $f$ defined, $n < \text{arity}(f)$, then $(f \, t_1 \ldots t_n \, t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow^*_H (c \, t_1 \ldots t_n)$, $c$ constructor, then $t_1, \ldots, t_n$ are also H-terminating.

- **H-Termination** of arbitrary term $t$ if
  $t \sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination of ground term** $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \cdots$
  - if $t \rightarrow_{H}^{*} (f \ t_{1} \ldots t_{n})$, $f$ defined, $n < \text{arity}(f)$,
    then $(f \ t_{1} \ldots t_{n} \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow_{H}^{*} (c \ t_{1} \ldots t_{n})$, $c$ constructor,
    then $t_{1},\ldots,t_{n}$ are also H-terminating.

- **H-Termination of arbitrary term** $t$ if
  
  $t_{\sigma}$ H-terminates for all substitutions $\sigma$ with H-terminating terms.
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination of ground term** $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_H \ldots$
  - if $t \rightarrow^*_H (f \ t_1 \ldots t_n)$, $f$ defined, $n < \text{arity}(f)$, then $(f \ t_1 \ldots t_n \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow^*_H (c \ t_1 \ldots t_n)$, $c$ constructor, then $t_1, \ldots, t_n$ are also H-terminating.

- **H-Termination of arbitrary term** $t$ if $t^\sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination of ground term** $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \ldots$
  - if $t \rightarrow_{H}^{*} (f \ t_{1} \ldots t_{n})$, $f$ defined, $n < \text{arity}(f)$,
    then $(f \ t_{1} \ldots t_{n} \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow_{H}^{*} (c \ t_{1} \ldots t_{n})$, $c$ constructor,
    then $t_{1}, \ldots, t_{n}$ are also H-terminating.

- **H-Termination of arbitrary term** $t$ if
  $t_{\sigma}$ H-terminates for all substitutions $\sigma$ with H-terminating terms.
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination of ground term** $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_H \ldots$
  - if $t \rightarrow^*_H (f\ t_1\ldots t_n)$, $f$ defined, $n < \text{arity}(f)$, then $(f\ t_1\ldots t_n\ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow^*_H (c\ t_1\ldots t_n)$, $c$ constructor, then $t_1,\ldots, t_n$ are also H-terminating.

- **H-Termination of arbitrary term** $t$ if
  $t\sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination of ground term** $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \ldots$
  - if $t \rightarrow^{*}_{H} (f \ t_{1} \ldots \ t_{n})$, $f$ defined, $n < \text{arity}(f)$, then $(f \ t_{1} \ldots \ t_{n} \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow^{*}_{H} (c \ t_{1} \ldots \ t_{n})$, $c$ constructor, then $t_{1}, \ldots, t_{n}$ are also H-terminating.

- **H-Termination of arbitrary term** $t$ if
  $t_{\sigma}$ H-terminates for all substitutions $\sigma$ with H-terminating terms.
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of ground term $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \ldots$
  - if $t \rightarrow^{*}_{H} (f \ t_{1} \ldots t_{n})$, $f$ defined, $n < \text{arity}(f)$, then $(f \ t_{1} \ldots t_{n} \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow^{*}_{H} (c \ t_{1} \ldots t_{n})$, $c$ constructor, then $t_{1}, \ldots, t_{n}$ are also H-terminating.

- **H-Termination** of arbitrary term $t$ if
  $t\sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.
**H-Termination**

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of ground term *t* if
  - *t* does not start infinite evaluation \( t \rightarrow_{H} \ldots \)
  - if \( t \rightarrow^{*}_{H} (f \ t_{1} \ldots t_{n}) \), *f* defined, \( n < \text{arity}(f) \), then \((f \ t_{1} \ldots t_{n} \ t')\) is also H-terminating if \( t' \) is H-terminating
  - if \( t \rightarrow^{*}_{H} (c \ t_{1} \ldots t_{n}) \), *c* constructor, then \( t_{1}, \ldots, t_{n} \) are also H-terminating.

- **H-Termination** of arbitrary term *t* if
  - *t*\( \sigma \) H-terminates for all substitutions \( \sigma \) with H-terminating terms.

- *x*
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of *ground* term $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \ldots$
  - if $t \rightarrow_{H}^{\star} (f \ t_1 \ldots t_n)$, $f$ defined, $n < \text{arity}(f)$, then $(f \ t_1 \ldots t_n \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow_{H}^{\star} (c \ t_1 \ldots t_n)$, $c$ constructor, then $t_1, \ldots, t_n$ are also H-terminating.

- **H-Termination** of *arbitrary* term $t$ if
  - $t\sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.

- $x$ is H-terminating
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of ground term $t$ if
  - $t$ does not start infinite evaluation $t \toH \ldots$
  - if $t \toH^* (f \ t_1 \ldots t_n)$, $f$ defined, $n < \text{arity}(f)$, then $(f \ t_1 \ldots t_n \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \toH^* (c \ t_1 \ldots t_n)$, $c$ constructor, then $t_1, \ldots, t_n$ are also H-terminating.

- **H-Termination** of arbitrary term $t$ if $t\sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.

- $x$ is H-terminating
  from
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of ground term $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \ldots$
  - if $t \rightarrow^{*}_{H} (f \ t_{1} \ldots t_{n})$, $f$ defined, $n < \text{arity}(f)$,
    then $(f \ t_{1} \ldots t_{n} \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow^{*}_{H} (c \ t_{1} \ldots t_{n})$, $c$ constructor,
    then $t_{1}, \ldots, t_{n}$ are also H-terminating.

- **H-Termination** of arbitrary term $t$ if
  $t\sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.

- $x$ is H-terminating
  - $\text{from}$ is not H-terminating ($\text{from Z}$ has infinite evaluation)
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of ground term $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \ldots$
  - if $t \rightarrow_{H}^{*} (f \ t_1 \ldots t_n)$, $f$ defined, $n < \text{arity}(f)$, then $(f \ t_1 \ldots t_n \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow_{H}^{*} (c \ t_1 \ldots t_n)$, $c$ constructor, then $t_1, \ldots, t_n$ are also H-terminating.

- **H-Termination** of arbitrary term $t$ if
  - $t\sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.

- $x$ is H-terminating
  - from is not H-terminating (from $Z$ has infinite evaluation)
  - take $u$ (from $m$)
H-Termination

Analyze **H-termination**:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of ground term $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \ldots$
  - if $t \rightarrow_{H}^{*} (f \ t_1 \ldots t_n)$, $f$ defined, $n < \text{arity}(f)$, then $(f \ t_1 \ldots t_n \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow_{H}^{*} (c \ t_1 \ldots t_n)$, $c$ constructor, then $t_1, \ldots, t_n$ are also H-terminating.
- **H-Termination** of arbitrary term $t$ if
  - $t_\sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.
- $x$ is H-terminating
  - from is not H-terminating (from $Z$ has infinite evaluation)
  - take $u$ (from $m$) is H-terminating
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of ground term $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \ldots$
  - if $t \rightarrow_{H}^{*} (f \ t_{1} \ldots t_{n})$, $f$ defined, $n < \text{arity}(f)$, then $(f \ t_{1} \ldots t_{n} \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow_{H}^{*} (c \ t_{1} \ldots t_{n})$, $c$ constructor, then $t_{1}, \ldots, t_{n}$ are also H-terminating.

- **H-Termination** of arbitrary term $t$ if
  $t_{\sigma}$ H-terminates for all substitutions $\sigma$ with H-terminating terms.

- $x$ is H-terminating
  - from is not H-terminating (from $Z$ has infinite evaluation)
  - take $u$ (from $m$) is H-terminating
  - Cons $u$ (from $m$)
H-Termination

Analyze H-termination:

If I only plug terminating arguments into my initial term, will the Haskell interpreter always give me an answer?

Formally:

- **H-Termination** of ground term $t$ if
  - $t$ does not start infinite evaluation $t \rightarrow_{H} \ldots$
  - if $t \rightarrow^{*}_{H} (f \ t_{1} \ldots t_{n})$, $f$ defined, $n < \text{arity}(f)$, then $(f \ t_{1} \ldots t_{n} \ t')$ is also H-terminating if $t'$ is H-terminating
  - if $t \rightarrow^{*}_{H} (c \ t_{1} \ldots t_{n})$, $c$ constructor, then $t_{1}, \ldots, t_{n}$ are also H-terminating.

- **H-Termination** of arbitrary term $t$ if $t\sigma$ H-terminates for all substitutions $\sigma$ with H-terminating terms.

- $x$ is H-terminating
  - from is not H-terminating (from $\text{Z}$ has infinite evaluation)
  - take $u$ (from $m$) is H-terminating
  - Cons $u$ (from $m$) is not H-terminating
From Haskell to Termination Graphs

\[
\text{from } x = \text{Cons } x \text{ (from } (S \ x))
\]

\[
\text{take } Z \ xs = \text{Nil}
\]

\[
\text{take } n \ \text{Nil} = \text{Nil}
\]

\[
\text{take } (S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs)
\]

**Goal:** Prove (H-)termination of initial term \(\text{take } u \ (\text{from } m)\)
From Haskell to Termination Graphs

from \( x = \text{Cons} \ x \ (\text{from} \ (S \ x)) \)  
\[ \text{take} \ Z \ xs = \text{Nil} \]
\[ \text{take} \ n \ \text{Nil} = \text{Nil} \]
\[ \text{take} \ (S \ n) \ (\text{Cons} \ x \ xs) = \text{Cons} \ x \ (\text{take} \ n \ xs) \]

**Goal:** Prove (H-)termination of initial term \( \text{take} \ u \ (\text{from} \ m) \)

**Naive approach**
- Use defining equations directly
- **fails**, since \( \text{from} \) is not terminating
- disregards Haskell’s evaluation strategy
From Haskell to Termination Graphs

\[
\text{from } x = \text{ Cons } x \ (\text{from } (S \ x))
\]

\[
\text{take } Z \ xs = \text{ Nil}
\]

\[
\text{take } n \ \text{Nil} = \text{ Nil}
\]

\[
\text{take } (S \ n) \ (\text{Cons } x \ xs) = \text{ Cons } x \ (\text{take } n \ xs)
\]

**Goal:** Prove (H-)termination of initial term \( \text{take } u \ (\text{from } m) \)

**Naive approach**
- Use defining equations directly
- **fails**, since \text{from} is not terminating
- disregards Haskell’s evaluation strategy

**Our approach** [Giesl et al, *TOPLAS ’11*]
- evaluate \text{initial term} a few steps
  \[\Rightarrow \text{termination graph} \ (\approx \text{abstract interpretation})\]
- our “abstract domain” for Haskell program states: \text{a single term}
- do not transform Haskell into TRS directly, but transform \text{termination graph} into TRS
From Haskell to Termination Graphs

\[
\text{from } x = \text{Cons } x \text{ (from } (S \ x)) \quad \text{take } Z \ xs = \text{Nil}
\]
\[
\text{take } n \text{ Nil } = \text{Nil}
\]
\[
\text{take } (S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs)
\]

\[
\text{take } u \ (\text{from } m)
\]

- begin with node marked with initial term
- 4 expansion rules to add children to leaves (more in paper)
- expansion rules try to \text{evaluate} terms
from \( x = \text{Cons} \, x \, (\text{from} \, (S \, x)) \)  
\[ \text{take} \, Z \, xs = \text{Nil} \]
\[ \text{take} \, n \, \text{Nil} = \text{Nil} \]
\[ \text{take} \, (S \, n) \, (\text{Cons} \, x \, xs) = \text{Cons} \, x \, (\text{take} \, n \, xs) \]

take \( u \) (from \( m \))

- **Case rule:**
  - **evaluation** has to continue with variable \( u \)
  - instantiate \( u \) by all possible constructor terms of correct type
From Haskell to Termination Graphs

\[
\text{from } x = \text{Cons } x \ (\text{from } (S \ x)) \quad \text{take } Z \ xs = \text{Nil} \\
\text{take } n \ \text{Nil} = \text{Nil} \\
\text{take } (S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs)
\]

**Case rule:**
- **evaluation** has to continue with variable \( u \)
- instantiate \( u \) by all possible constructor terms of correct type
from $x = \text{Cons } x \text{ (from } (S \ x))$  

\[ \text{take Z } xs = \text{Nil} \]

\[ \text{take } n \text{ Nil } = \text{Nil} \]

\[ \text{take } (S \ n) \text{ (Cons } x \ xs) = \text{Cons } x \text{ (take } n \ xs) \]
From Haskell to Termination Graphs

\[
\text{from } x = \text{ Cons } x (\text{ from } (S \, x)) \quad \text{take } Z \, xs = \text{ Nil} \\
\text{take } n \, \text{Nil} = \text{ Nil} \\
\text{take } (S \, n) (\text{Cons } x \, xs) = \text{Cons } x (\text{take } n \, xs)
\]

**Case**

- \( \text{take } u \, (\text{from } m) \)
- \( \text{take } Z \, (\text{from } m) \)
- \( \text{take } (S \, n) \, (\text{from } m) \)

\[
[u/\text{Z}] \\
[u/(S \, n)]
\]

**Eval** rule:

performs one **evaluation** step with \( \rightarrow_H \)
From Haskell to Termination Graphs

\[
\text{from } x = \text{Cons } x \text{ (from } (S \ x)\text{)} \quad \text{take } Z \ x s = \text{Nil} \\
\text{take } n \ \text{Nil} = \text{Nil} \\
\text{take } (S \ n) \ (\text{Cons } x \ x s) = \text{Cons } x \ (\text{take } n \ x s)
\]

- **Eval rule:**
  
  performs one evaluation step with \( \rightarrow_H \)
from $x = \text{Cons } x \ (\text{from } (S \ x))$  
take Z $xs = \text{Nil}$  
take $n \text{Nil} = \text{Nil}$  
take $(S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs)$

**Case** and **Eval** rule perform *narrowing*

w.r.t. Haskell’s evaluation strategy and types
From Haskell to Termination Graphs

from $x = \text{Cons } x \ (\text{from } (S \ x))$  

$\text{take } Z \ x s = \text{Nil}$

$\text{take } n \ \text{Nil} = \text{Nil}$

$\text{take } (S \ n) \ (\text{Cons } x \ x s) = \text{Cons } x \ (\text{take } n \ x s)$

- **ParSplit** rule:

  if head of term is a constructor like Cons or a variable, then continue with the parameters
from $x = \text{Cons } x (\text{from } (S \ x))$  

$\text{take } Z \, xs = \text{Nil}$

$\text{take } n \, \text{Nil} = \text{Nil}$

$\text{take } (S \ n) \, (\text{Cons } x \, xs) = \text{Cons } x \, (\text{take } n \, xs)$

- **ParSplit** rule:

  if head of term is a constructor like Cons or a variable, then continue with the parameters
From Haskell to Termination Graphs

\[ \text{from } x = \text{Cons } x (\text{from } (S \ x)) \quad \text{take } Z \ xs = \text{Nil} \]
\[ \text{take } n \ \text{Nil} = \text{Nil} \]
\[ \text{take } (S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs) \]

- one could continue with \textit{Case}, \textit{Eval}, \textit{ParSplit} \Rightarrow \text{infinite tree}
- Instead: \textit{Ins} rule to obtain finite graphs
From Haskell to Termination Graphs

\[ \text{from } x = \text{Cons } x \left( \text{from } (S \ x) \right) \]
\[ \text{take } Z \ xs = \text{Nil} \]
\[ \text{take } n \ \text{Nil} = \text{Nil} \]
\[ \text{take } (S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs) \]

**Ins rule:**
- if leaf \( t \) is instance of \( t' \), then add **instantiation edge** from \( t \) to \( t' \)
- one may re-use an existing node for \( t' \), if possible
from $x = \text{Cons}_x (\text{from} (S \, x))$  
$\text{take} \, Z \, xs = \text{Nil}$  
$\text{take} \, n \, \text{Nil} = \text{Nil}$  
$\text{take} \, (S \, n) \, (\text{Cons}_x \, xs) = \text{Cons}_x \, (\text{take} \, n \, xs)$

**Ins rule:**
- If leaf $t$ is instance of $t'$, then add instantiation edge from $t$ to $t'$
- One may re-use an existing node for $t'$, if possible
From Haskell to Termination Graphs

\[ \text{from } x = \text{Cons } x (\text{from } (S \ x)) \]
\[ \text{take } Z \ xs = \text{Nil} \]
\[ \text{take } n \ \text{Nil} = \text{Nil} \]
\[ \text{take } (S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs) \]

- \textbf{Ins rule:}
  - if leaf \( t \) is instance of \( t' \), then add \textit{instantiation edge} from \( t \) to \( t' \)
  - since instantiation is \([u/n, m/(S \ m)]\), add child nodes \( n \) and \( (S \ m) \)
From Haskell to Termination Graphs

\[ x = \text{Cons}(x, \text{from}(S\ s)) \quad \text{take}\ Z\ xs = \text{Nil} \]
\[ \text{take}\ n\ \text{Nil} = \text{Nil} \]
\[ \text{take}\ (S\ n)\ (\text{Cons}\ x\ xs) = \text{Cons}\ x\ (\text{take}\ n\ xs) \]

- **Ins** rule:
  - if leaf \( t \) is instance of \( t' \), then add instantiation edge from \( t \) to \( t' \)
  - since instantiation is \([u/n, m/(S\ m)]\), add child nodes \( n \) and \( (S\ m) \)
\[ \text{from } x = \text{Cons } x (\text{from } (S \ x)) \]
\[ \text{take } Z \ x s = \text{Nil} \]
\[ \text{take } n \ \text{Nil} = \text{Nil} \]
\[ \text{take } (S \ n) (\text{Cons } x \ x s) = \text{Cons } x (\text{take } n \ x s) \]

**ParSplit rule:**

if head of term is a constructor like \( S \),
then continue with the parameter
From Haskell to Termination Graphs

\[ x = \text{Cons } x \text{ (from } (\text{S } x)) \]

\[ \text{take } Z \text{ xs } = \text{Nil} \]

\[ \text{take } n \text{ Nil } = \text{Nil} \]

\[ \text{take } (\text{S } n) \text{ (Cons } x \text{ xs) } = \text{Cons } x \text{ (take } n \text{ xs)} \]

- **ParSplit** rule:
  - if head of term is a constructor like S, then continue with the parameter
From Haskell to Termination Graphs

\[ \text{from } x = \text{Cons } x (\text{from } (S \ x)) \]

\[ \text{take } Z \ xs = \text{Nil} \]

\[ \text{take } n \ Nil = \text{Nil} \]

\[ \text{take } (S \ n) (\text{Cons } x \ xs) = \text{Cons } x (\text{take } n \ xs) \]

**Termination Graph**

no expansion rule applicable to leaves anymore
From Haskell to Termination Graphs

\[ \text{from } x = \text{ Cons } x \ (\text{from } (S \ x)) \]
\[ \text{take } Z \ xs = \text{ Nil} \]
\[ \text{take } n \ \text{Nil} = \text{ Nil} \]
\[ \text{take } (S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs) \]

- **Termination Graph**
  - no expansion rule applicable to leaves anymore

- **Goal:** Prove H-termination of all terms in termination graph
Prove H-termination of all terms in termination graph

- a node is H-terminating if all its children are H-terminating
Prove H-termination of all terms in termination graph

- if node is not H-terminating, then a child is not H-terminating
Prove H-termination of all terms in termination graph

- not H-terminating node corresponds to infinite path in graph
Prove H-termination of all terms in termination graph

Case

- take $u$ (from $m$)
  - $[u/Z]$
  - $[u/(S n)]$

- take $Z$ (from $m$)
  - Nil

- take $(S n)$ (from $m$)
  - take $(S n)$ (Cons $m$ (from $(S m)$))
    - $Eval$
    - $Eval$
    - $ParSplit$
      - Cons $m$ (take $n$ (from $(S m)$))
        - $Ins$
        - take $n$ (from $(S m)$)
          - $ParSplit$
            - $S m$
              - $m$

- not H-terminating node corresponds to
  - Strongly Connected Component (SCC)
From Termination Graphs to DP Problems

- Prove H-termination of all terms for each SCC

- not H-terminating node corresponds to Strongly Connected Component (SCC)
Prove H-termination of all terms for each SCC

- not H-terminating node corresponds to Strongly Connected Component (SCC)
From Termination Graphs to DP Problems

- Every infinite path traverses an instantiation edge infinitely often
Every infinite path traverses an instantiation edge infinitely often

**DP path:**

path in SCC from node with incoming instantiation edge to node with outgoing instantiation edge
Every infinite path traverses a DP path infinitely often

**DP path:**
path in SCC from node with incoming instantiation edge to node with outgoing instantiation edge
Every infinite path traverses a DP path infinitely often

**Dependency Pairs:**

if there is a DP path from $s$ to $t$ marked with $\mu$, then generate the dependency pair $s \mu \rightarrow t$
Every infinite path traverses a DP path infinitely often
⇒ generate a dependency pair for every DP path

Dependency Pairs:
if there is a DP path from \( s \) to \( t \) marked with \( \mu \),
then generate the dependency pair \( s \mu \rightarrow t \)
Dependency Pair $\mathcal{P}$: \( \text{take}(S(n), \text{from}(m)) \rightarrow \text{take}(n, \text{from}(S(m))) \)

Rules $\mathcal{R}$: \( \emptyset \)

**Dependency Pairs:**

If there is a DP path from $s$ to $t$ marked with $\mu$, then generate the dependency pair $s \mu \rightarrow t$
Dependency Pair $\mathcal{P}$: $\text{take}(S(n), \text{from}(m)) \rightarrow \text{take}(n, \text{from}(S(m)))$

Rules $\mathcal{R}$: $\emptyset$ (rules for terms in instance-edge matcher)

**Dependency Pairs:**

if there is a DP path from $s$ to $t$ marked with $\mu$, then generate the dependency pair $s \mu \rightarrow t$
Dependency Pair $\mathcal{P}$: $\text{take}(S(n), \text{from}(m)) \rightarrow \text{take}(n, \text{from}(S(m)))$

Rules $\mathcal{R}$: $\emptyset$

Dependency Pairs:
if there is a DP path from $s$ to $t$ marked with $\mu$, then generate the dependency pair $s \mu \rightarrow t$
Dependency Pair $\mathcal{P}$: $\text{take}(S(n), \text{from}(m)) \rightarrow \text{take}(n, \text{from}(S(m)))$

Rules $\mathcal{R}$: $\emptyset$

- one DP per SCC
- no rules in $\mathcal{R}$ if no defined symbols to evaluate in rhs of DP
From Termination Graphs to DP Problems

- **Dependency Pair** $\mathcal{P}$: $\text{take}(S(n), \text{from}(m)) \rightarrow \text{take}(n, \text{from}(S(m)))$
- **Rules** $\mathcal{R}$: $\emptyset$

Termination easy to prove

- one DP per SCC
- no rules in $\mathcal{R}$ if no defined symbols to evaluate in rhs of DP
from $x = \text{Cons } x \ (\text{from } (S \ x))$  
take $Z \ xs = \text{Nil}$  
take $n \ \text{Nil} = \text{Nil}$  
take $(S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } n \ xs)$
from $x = \text{Cons } x \text{ (from } S x\text{)}$  
take $Z \ \text{xs} = \text{Nil}$

take $n \ \text{Nil} = \text{Nil}$

take $(S \ n) \ (\text{Cons } x \ \text{xs}) = \text{Cons } x \ (\text{take } (\text{p } (S \ n)) \ \text{xs})$

$p \ (S \ x) = x$
From Termination Graphs to DP Problems

\[ x = \text{Cons} \; x \; (\text{from} \; (S \; x)) \]
\[ \text{take} \; Z \; xs = \text{Nil} \]
\[ \text{take} \; n \; \text{Nil} = \text{Nil} \]
\[ p \; (S \; x) = x \]
\[ \text{take} \; (S \; n) \; (\text{Cons} \; x \; xs) = \text{Cons} \; x \; (\text{take} \; (p \; (S \; n)) \; xs) \]
From Termination Graphs to DP Problems

from $x = \text{Cons } x \ (\text{from } (S \ x))$  
take $Z \ xs = \text{Nil}$  
take $n \ \text{Nil} = \text{Nil}$  
take $(S \ n) \ (\text{Cons } x \ xs) = \text{Cons } x \ (\text{take } (p \ (S \ n)) \ xs)$

\[
p \ (S \ x) = x
\]
Dependency Pair \( \mathcal{P} \):
\[
\text{take}(S(n), \text{from}(m)) \rightarrow \text{take}(p(S(n)), \text{from}(S(m)))
\]
**Dependency Pair \( P \):**
\[
take(S(n), \text{from}(m)) \rightarrow \text{take}(p(S(n)), \text{from}(S(m)))
\]
From Termination Graphs to DP Problems

- **Dependency Pair $\mathcal{P}$:**
  \[
  \text{take}(S(n), \text{from}(m)) \rightarrow \text{take}(p(S(n)), \text{from}(S(m)))
  \]

- $\mathcal{R}$: rules for terms in instance-edge matcher

---

**Diagram:***

- **Case**
  - take $u$ (from $m$)
  - take $Z$ (from $m$)
  - Nil

- **Eval**
  - take $(S(n))$ (from $m$)
  - take $(S(n))$ (Cons $m$ (from $(S(m))$))
  - Cons $m$ (take $(p(S(n)))$ (from $(S(m))$))

- **ParSplit**
  - take $(p(S(n)))$ (from $(S(m))$)

- **Ins**
  - $m$

- **ParSplitEval**
  - $S m$

- **Eval**
  - $p(S(n))$
  - $n$
  - $m$

22/64
**Dependency Pair** $\mathcal{P}$:
\[
take(S(n), \text{from}(m)) \rightarrow take(p(S(n)), \text{from}(S(m)))
\]

**Rule path**
path from term in instance-edge matcher over *Eval* and *Case* nodes to non-*Eval* and non-*Case* node
From Termination Graphs to DP Problems

- **Dependency Pair $\mathcal{P}$:**
  \[
  \text{take}(S(n), \text{from}(m)) \rightarrow \text{take}(p(S(n)), \text{from}(S(m)))
  \]

- **Rules**
  If there is a rule path from $s$ to $t$ marked with $\mu$, then generate the rule $s\mu \rightarrow t$
From Termination Graphs to DP Problems

- **Dependency Pair** $\mathcal{P}$:
  \[ \text{take}(S(n), \text{from}(m)) \rightarrow \text{take}(p(S(n)), \text{from}(S(m))) \]

- **Rule** $\mathcal{R}$: $p(S(n)) \rightarrow n$

**Rules**

If there is a rule path from $s$ to $t$ marked with $\mu$, then generate the rule $s \mu \rightarrow t$
**Dependency Pair** \( \mathcal{P} \):
\[
take(S(n), \text{from}(m)) \rightarrow take(p(S(n)), \text{from}(S(m)))
\]

**Rule** \( \mathcal{R} \):
\[
p(S(n)) \rightarrow n
\]

termination easy to prove

**Rules**
if there is a rule path from \( s \) to \( t \) marked with \( \mu \), then generate the rule \( s \mu \rightarrow t \)
Experiments

Implementation in termination prover AProVE
(uses improved step termination graph \( \rightarrow \) DP problem)

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

Experiments on Haskell libraries

- FiniteMap, List, Monad, Prelude, Queue
- 300 seconds timeout
- AProVE shows H-Termination for 999 out of 1272 functions
The Tool Chain for Haskell

- Termination analysis of Haskell 98 via transformation to TRSs

```
Start Term
Haskell-Program → Termination Graph → DP Problems → Termination Tool for TRSs
```

- Language specifics are handled in transformation front-end
  ⇒ Apply TRS analysis back-end for several programming languages!
- Successful evaluation on Haskell 98 standard libraries

Details: [Giesl et al., TOPLAS ’11]

http://aprove.informatik.rwth-aachen.de/eval/Haskell
Conclusion of Part I

Analyze program termination in 2 steps:

- Program $\rightarrow$ term rewrite system
- Term rewrite system $\rightarrow$ termination proof

Termination analysis for languages other than Haskell:

- Logic programming: Prolog
  [van Raamsdonk, ICLP ’97; Schneider-Kamp et al, TOCL ’09; Giesl et al, PPDP ’12]
- Object-oriented programming: Java
  [Otto et al, RTA ’10] $\rightarrow$ tomorrow

AProVE web interface and Eclipse plug-in download at:

http://aprove.informatik.rwth-aachen.de/
Conclusion of Part I

Analyze program termination in 2 steps:

- Program $\rightarrow$ term rewrite system
- Term rewrite system $\rightarrow$ termination proof

Termination analysis for languages other than Haskell:

- Logic programming: Prolog
  [van Raamsdonk, ICLP ’97; Schneider-Kamp et al, TOCL ’09; Giesl et al, PPDP ’12]
- Object-oriented programming: Java
  [Otto et al, RTA ’10] $\rightarrow$ tomorrow

AProVE web interface and Eclipse plug-in download at:

http://aprove.informatik.rwth-aachen.de/
Conclusion of Part I

Analyze program termination in 2 steps:

- Program $\rightarrow$ term rewrite system
- Term rewrite system $\rightarrow$ termination proof

Termination analysis for languages other than Haskell:

- Logic programming: Prolog
- Object-oriented programming: Java
  [Otto et al, *RTA ’10*] $\rightarrow$ tomorrow

AProVE web interface and Eclipse plug-in download at:

http://aprove.informatik.rwth-aachen.de/
Consider the following Haskell program.

```haskell
data Nat = Z | S Nat
data List a = Nil | Cons a (List a)

myleNGTH Nil = Z
myleNGTH (Cons x xs) = S (myleNGTH xs)

mysum Nil = Z
mysum (Cons x xs) = plus x (mysum xs)

plus Z y = y
plus (S x) y = S (plus x y)
```
Question 1

Consider the start term `mylength x`.

(a) Is this start term H-terminating?

(b) Construct a termination graph for this start term.

(c) Extract a DP problem from the termination graph from part (b).

(d) Prove that this DP problem is “terminating”, i.e., that no infinite call sequences are possible.

(e) Check your solutions with the web interface (or a local installation) of the termination prover AProVE: http://aprove.informatik.rwth-aachen.de/ (note that AProVE preprocesses the termination graph before the step to DP problems so that the output will look slightly differently).
Question 2 (slightly harder/more interesting)

Consider the start term `mysum x`. Proceed with it as in Question 1.

*Hint:* One can draw instantiation edges also to nodes that are not yet present in the termination graph.

Question 3

What strengths and limitations do you expect this approach to termination proving of Haskell programs to have?
Proving Program Termination via Term Rewriting

1. Overview

2. Termination Analysis of Term Rewriting with Dependency Pairs

3. Haskell: a Pure Functional Language with Lazy Evaluation

4. Java: an Object-Oriented Imperative Language with Side Effects
Recipe for proving program termination by reusing TRS termination provers

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

Yesterday: Haskell, today: Java
Recipe for proving program termination by reusing TRS termination provers

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

Yesterday: Haskell, today: Java
Recipe for proving program termination by reusing TRS termination provers

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

Yesterday: Haskell, today: Java
Recipe for proving program termination by reusing TRS termination provers

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

Yesterday: Haskell, today: Java
Recipe for proving program termination by reusing TRS termination provers

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

Yesterday: Haskell, today: Java
Recipe for proving program termination by reusing TRS termination provers

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

Yesterday: Haskell, today: Java
Recipe for proving program termination by reusing TRS termination provers

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

Yesterday: Haskell, today: Java
Rewrite rules for Haskell programs in **standard** term rewriting
→ no predefined rules for addition, multiplication, etc.

**Drawbacks:**
- throws away domain knowledge about built-in data types like integers
- need to analyze recursive rules for `plus`, `times`, … over and over
- does not benefit from dedicated constraint solvers
  (SMT: SAT Modulo Theories) for arithmetic operations

**Solution:** use **constrained** term rewriting
Rewrite rules for Haskell programs in **standard** term rewriting

→ no predefined rules for addition, multiplication, etc.

**Drawbacks:**
- throws away domain knowledge about built-in data types like integers
- need to analyze recursive rules for `plus`, `times`, … over and over
- does not benefit from dedicated constraint solvers (SMT: SAT Modulo Theories) for arithmetic operations

**Solution:** use **constrained** term rewriting
Rewrite rules for Haskell programs in \textbf{standard} term rewriting
\[ \rightarrow \]\ no predefined rules for addition, multiplication, etc.

\textbf{Drawbacks:}

- throws away domain knowledge about built-in data types like integers
- need to analyze recursive rules for \texttt{plus}, \texttt{times}, \ldots over and over
- does not benefit from dedicated constraint solvers (SMT: SAT Modulo Theories) for arithmetic operations

\textbf{Solution:} use \textit{constrained} term rewriting
Beyond Classic TRSs for Programs

Rewrite rules for Haskell programs in standard term rewriting → no predefined rules for addition, multiplication, etc.

**Drawbacks:**
- throws away domain knowledge about built-in data types like integers
- need to analyze recursive rules for plus, times, ... over and over
- does not benefit from dedicated constraint solvers (SMT: SAT Modulo Theories) for arithmetic operations

**Solution:** use constrained term rewriting
Beyond Classic TRSs for Programs

Rewrite rules for Haskell programs in **standard** term rewriting
→ no predefined rules for addition, multiplication, etc.

**Drawbacks:**
- throws away domain knowledge about built-in data types like integers
- need to analyze recursive rules for `plus`, `times`, ... over and over
- does not benefit from dedicated constraint solvers (SMT: SAT Modulo Theories) for arithmetic operations

**Solution:** use **constrained** term rewriting
Beyond Classic TRSs for Programs

Rewrite rules for Haskell programs in **standard** term rewriting
→ no predefined rules for addition, multiplication, etc.

**Drawbacks:**
- throws away domain knowledge about built-in data types like integers
- need to analyze recursive rules for *plus*, *times*, … over and over
- does not benefit from dedicated constraint solvers (SMT: SAT Modulo Theories) for arithmetic operations

**Solution:** use **constrained term rewriting**
Constrained Term Rewriting, what’s that?

Term rewriting “with batteries included”

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

⇒ Term rewriting + SMT solving for automated reasoning

- General forms available, e.g., Logically Constrained TRSs [Kop, Nishida, FroCoS ’13]
- For program termination: use term rewriting with integers [Falke, Kapur, CADE ’09; Fuhs et al, RTA ’09; Giesl et al, JAR ’17]
Constrained Term Rewriting, what’s that?

Term rewriting “with batteries included”

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

⇒ Term rewriting + SMT solving for automated reasoning

- General forms available, e.g., Logically Constrained TRSs
  [Kop, Nishida, *FroCoS ’13*]
- For program termination: use term rewriting with integers
  [Falke, Kapur, *CADE ’09*; Fuhs et al, *RTA ’09*; Giesl et al, *JAR ’17*]
Constrained Term Rewriting, what’s that?

Term rewriting “with batteries included”

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

⇒ Term rewriting + SMT solving for automated reasoning

- General forms available, e.g., Logically Constrained TRSs
  [Kop, Nishida, FroCoS ’13]
- For program termination: use term rewriting with integers
  [Falke, Kapur, CADE ’09; Fuhs et al, RTA ’09; Giesl et al, JAR ’17]
Constrained Term Rewriting, what’s that?

Term rewriting “with batteries included”

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

⇒ Term rewriting + SMT solving for automated reasoning

- General forms available, e.g., Logically Constrained TRSs
  [Kop, Nishida, FroCoS ’13]
- For program termination: use term rewriting with integers
  [Falke, Kapur, CADE ’09; Fuhs et al, RTA ’09; Giesl et al, JAR ’17]
Constrained Term Rewriting, what’s that?

Term rewriting “with batteries included”

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

⇒ Term rewriting + SMT solving for automated reasoning

- General forms available, e.g., Logically Constrained TRSs [Kop, Nishida, FroCoS ’13]
- For program termination: use term rewriting with integers [Falke, Kapur, CADE ’09; Fuhs et al, RTA ’09; Giesl et al, JAR ’17]
Constrained Term Rewriting, what’s that?

Term rewriting “with batteries included”

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

⇒ Term rewriting + SMT solving for automated reasoning

- General forms available, e.g., Logically Constrained TRSs
  [Kop, Nishida, FroCoS ’13]
- For program termination: use term rewriting with integers
  [Falke, Kapur, CADE ’09; Fuhs et al, RTA ’09; Giesl et al, JAR ’17]
Constrained Term Rewriting, what’s that?

Term rewriting “with batteries included”

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

⇒ Term rewriting + SMT solving for automated reasoning

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

Consider a variation of the take-from program . . .

Example (Constrained Rewrite System)

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

Possible rewrite sequence:
Consider a variation of the take-from program . . .

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

\[\ell_0(2, 7)\]
Consider a variation of the take-from program . . .

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

\[
\begin{align*}
\ell_0(2, 7) &
\rightarrow \ell_1(2, 7, \text{Nil})
\end{align*}
\]
Consider a variation of the take-from program . . .

**Example (Constrained Rewrite System)**

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

Possible rewrite sequence:

\[
\ell_0(2, 7) \\
\quad \rightarrow \ell_1(2, 7, \text{Nil}) \\
\quad \rightarrow \ell_1(1, 8, \text{Cons}(7, \text{Nil}))
\]
Consider a variation of the take-from program . . .

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

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

Consider a variation of the take-from program . . .

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

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

Consider a variation of the take-from program . . .

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

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

Here 7, 8, . . . are predefined constants.
Papers on termination of imperative programs often about **integers** as data.
Papers on termination of imperative programs often about **integers** as data

Example (Imperative Program)

```
if (x ≥ 0)
    while (x ≠ 0)
        x = x − 1;
```

Does this program terminate?
Papers on termination of imperative programs often about **integers** as data

**Example (Imperative Program)**

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

Does this program terminate?

**Example (Equivalent Translation to Constrained Rewriting, cf. [McCarthy, *CACM ‘60]*)**

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

Example (Imperative Program)

\( \ell_0: \) if \((x \geq 0)\)
\( \ell_1: \) while \((x \neq 0)\)
\( \ell_2: \) \(x = x - 1;\)

Does this program terminate?

Example (Equivalent Translation to Constrained Rewriting, cf. [McCarthy, CACM '60])

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

Oh no!

\( \ell_1(-1) \rightarrow \ell_2(-1) \rightarrow \ell_1(-2) \rightarrow \ell_2(-2) \rightarrow \ell_1(-3) \rightarrow \cdots \)
Papers on termination of imperative programs often about **integers** as data

Example (Imperative Program)

\[
l_0: \text{ if } (x \geq 0) \\
l_1: \text{ while } (x \neq 0) \\
l_2: \quad x = x - 1;
\]

Does this program terminate?

Example (Equivalent Translation to Constrained Rewriting, cf. [McCarthy, CACM '60])

\[
l_0(x) \rightarrow l_1(x) \quad [x \geq 0] \\
l_0(x) \rightarrow l_3(x) \quad [x < 0] \\
l_1(x) \rightarrow l_2(x) \quad [x \neq 0] \\
l_2(x) \rightarrow l_1(x-1) \\
l_1(x) \rightarrow l_3(x) \quad [x = 0]
\]

Oh no! \[l_1(-1) \rightarrow l_2(-1) \rightarrow l_1(-2) \rightarrow l_2(-2) \rightarrow l_1(-3) \rightarrow \cdots\]

⇒ **Restrict initial states** to \(l_0(z)\) for \(z \in \mathbb{Z}\)
Papers on termination of imperative programs often about **integers** as data.

**Example (Imperative Program)**

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

Does this program terminate?

**Example (Equivalent Translation to Constrained Rewriting, cf. [McCarthy, CACM '60])**

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

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

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

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

**Example (Imperative Program)**

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

Does this program terminate?

**Example (Equivalent Translation to Constrained Rewriting, cf. [McCarthy, CACM '60])**

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

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

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

\Rightarrow \textbf{Find invariant } x \geq 0 \text{ at } \ell_1, \ell_2 \text{ (exercise)}
Java Challenges

Java: object-oriented imperative language

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

...
public class MyInt {

    // only wrap a primitive int
    private int val;

    // count "num" up to the value in "limit"
    public static void count(MyInt num, MyInt limit) {
        if (num == null || limit == null) {
            return;
        }
        // introduce sharing
        MyInt copy = num;
        while (num.val < limit.val) {
            copy.val++;
        }
    }
}
Tailor two-stage approach from Haskell analysis to Java [Otto et al, RTA ’10]

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

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

Implemented in the tool AProVE (→ web interface)

http://aprove.informatik.rwth-aachen.de/
Approach to Termination Analysis of Java

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

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

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

Implemented in the tool AProVE (→ web interface)

http://aprove.informatik.rwth-aachen.de/
Approach to Termination Analysis of Java

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

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

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

Implemented in the tool AProVE (∫ web interface)

http://aprove.informatik.rwth-aachen.de/
Approach to Termination Analysis of Java

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

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

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

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

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

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

Java Bytecode

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

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

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

Here: Java source code
Ingredients for the Abstract Domain

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

Heap predicates:
- Two references may be equal: $o_1 \equiv o_2$
- Two references may share: $o_1 \preceq o_2$
- Reference may have cycles: $o_1 !$

<table>
<thead>
<tr>
<th>03</th>
<th>num : $o_1$, limit : $o_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1$ : MyInt(?)</td>
<td></td>
</tr>
<tr>
<td>$o_2$ : MyInt(val = $i_1$)</td>
<td></td>
</tr>
<tr>
<td>$i_1$ : $[4, 80]$</td>
<td></td>
</tr>
</tbody>
</table>
Ingredients for the Abstract Domain

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

\textbf{Heap predicates:}
- Two references may be equal: \(o_1 =? o_2\)
- Two references may share: \(o_1 \%o_2\)
- Reference may have cycles: \(o_1 !\)

<table>
<thead>
<tr>
<th>03</th>
<th>num: (o_1), limit: (o_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(o_1): \texttt{MyInt(?)}</td>
<td></td>
</tr>
<tr>
<td>(o_2): \texttt{MyInt(val = i_1)}</td>
<td></td>
</tr>
<tr>
<td>(i_1): ([4, 80])</td>
<td></td>
</tr>
</tbody>
</table>
Ingredients for the Abstract Domain

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

Heap predicates:
- Two references may be equal: $o_1 =? o_2$
- Two references may share: $o_1 \lor o_2$
- Reference may have cycles: $o_1 !$

<table>
<thead>
<tr>
<th>03</th>
<th>num: $o_1$, limit: $o_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1$: MyInt(?)</td>
<td></td>
</tr>
<tr>
<td>$o_2$: MyInt(val = $i_1$)</td>
<td></td>
</tr>
<tr>
<td>$i_1$: [4, 80]</td>
<td></td>
</tr>
</tbody>
</table>
Building the Termination Graph

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

### A

<table>
<thead>
<tr>
<th></th>
<th>num: $o_1$, limit: $o_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1$</td>
<td>MyInt(?)</td>
</tr>
<tr>
<td>$o_2$</td>
<td>MyInt(?)</td>
</tr>
</tbody>
</table>
Building the Termination Graph

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

cond

X → Y

means: refine X with cond, then evaluate to Y; here combined for brevity (narrowing; Haskell: Case + Eval)
Building the Termination Graph

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

cond means: refine X with cond, then evaluate to Y; here combined for brevity (narrowing; Haskell: Case + Eval)
Building the Termination Graph

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

X → Y
means: evaluate X to Y
(Haskell: \textit{Eval})
Building the Termination Graph

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

    i_3 = i_1 + 1

    i_1 < i_2

    i_1 = null

    i_2 = null

    i_1 ≥ i_2

    i_3 = i_1 + 1
Building the Termination Graph

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

X is instance of Y (Haskell: Ins)
Termination Graphs

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

Using Termination Graphs for Termination Proofs

- every concrete Java computation corresponds to a computation path in the termination graph (related: DP paths for Haskell as suffixes of non-(H-)terminating computations)
- termination graph is called terminating
  - iff it has no infinite computation path
Termination Graphs

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

Using Termination Graphs for Termination Proofs

- every concrete Java computation corresponds to a computation path in the termination graph (related: DP paths for Haskell as suffixes of non-(H-)terminating computations)
- termination graph is called terminating iff it has no infinite computation path
For every class $C$ with $n$ fields, introduce an $n$-ary function symbol $C$

- term for $o_1$: $o_1$
- term for $o_2$: MyInt($i_2$)
- term for $o_3$: null
- term for $o_4$: $x$ (new variable)
- term for $i_1$: $i_1$ with side constraint $i_1 \geq 7$
  (invariant $i_1 \geq 7$ to be added to constrained rewrite rules for state Q)
public class A {
    int a;
}

public class B extends A {
    int b;
}

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

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

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

public class B extends A {
    int b;
}

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

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

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

public class A {
    int a;
}

public class B extends A {
    int b;
}

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

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

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

\[ i_3 = i_1 + 1 \]

\begin{align*}
\text{State } F: & \quad \ell_F( \text{jlO(MyInt(eoc, } i_1)), \text{jlO(MyInt(eoc, } i_2)) ) \\
\text{State } H: & \quad \ell_H( \text{jlO(MyInt(eoc, } i_1)), \text{jlO(MyInt(eoc, } i_2)) ) \\
\text{State } H: & \quad \ell_H( \text{jlO(MyInt(eoc, } i_1)), \text{jlO(MyInt(eoc, } i_2)) ) \\
\text{State } I: & \quad \ell_F( \text{jlO(MyInt(eoc, } i_1 + 1)), \text{jlO(MyInt(eoc, } i_2)) ) \\
\text{Termination easy to show (intuitively: } i_2 - i_1 \text{ decreases against bound 0) }
\end{align*}
State F: \( \ell_F( \text{jlo}(\text{MyInt}(\text{eoc}, i_1)), \text{jlo}(\text{MyInt}(\text{eoc}, i_2)) ) \)

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

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

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

Termination easy to show (intuitively: \( i_2 - i_1 \) decreases against bound 0)
From the Termination Graph to Terms and Rules

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

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

Termination easy to show (intuitively: \( i_2 - i_1 \) decreases against bound 0)
From the Termination Graph to Terms and Rules

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

\( i₃ = i₁ + 1 \)

\[ 6 \mid \text{num: o₁, limit: o₂, copy: o₁} \]

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

\[ 5 \mid \text{num: o₁, limit: o₂, copy: o₁} \]

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

- **State F:** \( \ell_F(\text{jLO(MyInt(eoc, i₁)), jLO(MyInt(eoc, i₂))} ) \)
  \[ \rightarrow \]
  **State H:** \( \ell_H(\text{jLO(MyInt(eoc, i₁)), jLO(MyInt(eoc, i₂))} ) \) \[ i₁ < i₂ \]

- **State H:** \( \ell_H(\text{jLO(MyInt(eoc, i₁)), jLO(MyInt(eoc, i₂))} ) \)

- **State I:** \( \ell_F(\text{jLO(MyInt(eoc, i₁ + 1)), jLO(MyInt(eoc, i₂))} ) \)

- Termination easy to show (intuitively: \( i₂ − i₁ \) decreases against bound 0)
From the Termination Graph to Terms and Rules

### State F:
\[ \ell_F(\ jL0(MyInt(eoc, i_1)), \ jL0(MyInt(eoc, i_2)) ) \]
\[ \rightarrow \]
State H: \[ \ell_H(\ jL0(MyInt(eoc, i_1)), \ jL0(MyInt(eoc, i_2)) ) \quad [i_1 < i_2] \]

### State H:
\[ \ell_H(\ jL0(MyInt(eoc, i_1)), \ jL0(MyInt(eoc, i_2)) ) \]
\[ \rightarrow \]
State I: \[ \ell_F(\ jL0(MyInt(eoc, i_1 + 1)), \ jL0(MyInt(eoc, i_2)) ) \]

Termination easy to show (intuitively: \( i_2 - i_1 \) decreases against bound 0)
From the Termination Graph to Terms and Rules

<table>
<thead>
<tr>
<th>5</th>
<th>num: o₁, limit: o₂, copy: o₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁: MyInt(val = i₁)</td>
<td></td>
</tr>
<tr>
<td>o₂: MyInt(val = i₂)</td>
<td></td>
</tr>
<tr>
<td>i₁: (−∞, ∞)</td>
<td></td>
</tr>
<tr>
<td>i₂: (−∞, ∞)</td>
<td></td>
</tr>
<tr>
<td>i₃ = i₁ + 1</td>
<td></td>
</tr>
</tbody>
</table>

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

State F: \[ \ell_F(\text{j}O(MyInt(eoc, i₁)), \text{j}O(MyInt(eoc, i₂))) \]
\[ \rightarrow \]
State H: \[ \ell_H(\text{j}O(MyInt(eoc, i₁)), \text{j}O(MyInt(eoc, i₂))) \]
\[ [i₁ < i₂] \]

State H: \[ \ell_H(\text{j}O(MyInt(eoc, i₁)), \text{j}O(MyInt(eoc, i₂))) \]
\[ \rightarrow \]
State I: \[ \ell_F(\text{j}O(MyInt(eoc, i₁ + 1)), \text{j}O(MyInt(eoc, i₂))) \]

Termination easy to show (intuitively: \( i₂ - i₁ \) decreases against bound 0)
Extensions

- **modular termination proofs and recursion**  
  [Brockschmidt et al, *RTA ’11*]
- proving **reachability** and **non-termination** (uses only termination graph)  
  [Brockschmidt et al, *FoVeOOS ’11*]
- proving termination with **cyclic data objects** (preprocessing in termination graph)  
  [Brockschmidt et al, *CAV ’12*]
- proving upper bounds for **time complexity** (abstracts terms to numbers)  
  [Frohn and Giesl, *iFM ’17*]
Extensions

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

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

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

- proving upper bounds for **time complexity** (abstracts terms to numbers) [Frohn and Giesl, *iFM ’17*]
Extensions

- **modular** termination proofs and **recursion** [Brockschmidt et al, *RTA ’11*]
- proving **reachability** and **non-termination** (uses only termination graph) [Brockschmidt et al, *FoVeOOS ’11*]
- proving termination with **cyclic data objects** (preprocessing in termination graph) [Brockschmidt et al, *CAV ’12*]
- proving upper bounds for time complexity (abstracts terms to numbers) [Frohn and Giesl, *iFM ’17*]
Extensions

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

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

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

- proving upper bounds for **time complexity** (abstracts terms to numbers)
  [Frohn and Giesl, *iFM ’17*]
Conclusion Part II

Java:
- Successful empirical evaluation of Java approach on Termination Problems Database, including Java classes (e.g., LinkedList)
- Approach also successful at Termination Competition (other tools like COSTA, Julia abstract data structures to numbers instead of terms)

Overall:
- Common theme for program analysis by rewriting:
  - handle language specifics in front-end
  - transitions between program states become rewrite rules for TRS termination back-end
- Haskell: single term as abstract domain to represent program state
- Java: more complex abstract domain, use constrained rewriting
Java:
- Successful empirical evaluation of Java approach on Termination Problems Database, including Java classes (e.g., LinkedList)
- Approach also successful at Termination Competition (other tools like COSTA, Julia abstract data structures to numbers instead of terms)

Overall:
- Common theme for program analysis by rewriting:
  - handle language specifics in **front-end**
  - transitions between program states become rewrite rules for **TRS termination back-end**
- Haskell: single term as abstract domain to represent program state
- Java: more complex abstract domain, use constrained rewriting
Question 4

Recall the imperative program fragment on slide 34 (the while loop counting down).

In the lecture we added the invariant $x \geq 0$ to the constrained rewrite system. Construct a termination graph for the program that also finds this invariant and extract a constrained rewrite system with the invariant from this termination graph.
Construct a termination graph for `length`, then extract the corresponding constrained rewrite system for the SCCs in the graph. Can you prove termination of the resulting constrained rewrite system?


Solutions for the Exercises
(a) Yes. Intuition: The recursive call to `mylength` is on a list that is shorter than the original (H-terminating!) list. Thus, the end of the list will eventually be reached, and the recursion ends in its base case.

(b)

(c) \( P : \text{mylength}(\text{Cons}(y, ys)) \rightarrow \text{mylength}(ys) \)

\( R : \emptyset \)
(d) We can prove termination via a linear polynomial interpretation \([\cdot]\) of the function symbols to \(\mathbb{N}\), such as:

\[
\begin{align*}
  \text{[myleft]}(x_1) &= x_1 \\
  \text{[Cons]}(x_1, x_2) &= x_1 + x_2 + 1
\end{align*}
\]

Alternatively, we could also use the embedding order or any path order.

(e) AProVE uses an adaption of the size-change termination principle [Lee, Jones, Ben-Amram, *POPL '01*] to term rewriting and dependency pairs [Thiemann, Giesl, *AAECC '05*] for the termination proof:

http://www.dcs.bbk.ac.uk/~carsten/isr2017/Ex1.html

Note that AProVE uses a slightly improved version of the step from termination graphs to DP problems. This can lead to simpler outputs than our translation from the lecture, in particular if Haskell terms with higher-order symbols are involved.

Further details: [Giesl et al, *TOPLAS '11*]
(a) Yes. The reason is that both \texttt{mysum} and \texttt{plus} are terminating since their recursive calls are on arguments that get smaller and smaller.
Solution for Question 2 (page 2)

(b)
(c) $\mathcal{P} : \text{mysum(Cons}(y, \text{ys})) \rightarrow \text{mysum}(\text{ys})$

$\text{plus}(S(u), v) \rightarrow \text{plus}(u, v)$

$\mathcal{R} : \emptyset$

(d) We can prove termination via a linear polynomial interpretation $[\cdot]$ of the function symbols to $\mathbb{N}$, such as:

$[\text{mysum}](x_1) = x_1$

$[\text{Cons}](x_1, x_2) = x_1 + x_2 + 1$

$[\text{plus}](x_1, x_2) = x_1$

$[S](x_1) = x_1 + 1$

Alternatively, we could also use the embedding order or any path order. (In general, more powerful techniques can be required for a successful termination proof of a Haskell program. However, the examples that we have considered in the exercises terminate for relatively straightforward reasons.)

(e) AProVE again uses the size-change termination principle for both DPs in $\mathcal{P}$ to prove termination:

http://www.dcs.bbk.ac.uk/~carsten/isr2017/Ex2.html
Solution for Question 3

Some strengths that one might expect (non-exhaustive list):

- support of user-defined data structures by representation as terms
- as termination tools for TRSs improve over time thanks to on-going development, so does this overall approach

Some weaknesses that one might expect (non-exhaustive list):

- support of built-in data structures (e.g., Integer) and their operations (e.g., +, *, ...) by terms over a finite signature and recursive rewrite rules on them is cumbersome; does not benefit from specialized program analysis techniques for built-in data structures, e.g., invariant synthesis (but: could improve using constrained rewriting with built-in data structures as translation target)
- termination back-end must prove termination of all terms; start term information is “lost in translation” (but: could include the path from initial node to SCCs in translated system and prove termination from only the start terms for the initial node in the resulting problem; would need TRS termination tools that benefit from this information)
Solution for Question 4 (page 1)

We get the following termination graph for the program:

```
A

<table>
<thead>
<tr>
<th>0</th>
<th>x : i_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_1 : (-∞, ∞)</td>
<td></td>
</tr>
</tbody>
</table>

i_1 ≥ 0

B

<table>
<thead>
<tr>
<th>1</th>
<th>x : i_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_1 : [0, ∞)</td>
<td></td>
</tr>
</tbody>
</table>

i_1 < 0

C

<table>
<thead>
<tr>
<th>3</th>
<th>x : i_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_1 : (-∞, -1]</td>
<td></td>
</tr>
</tbody>
</table>

i_1 ≠ 0

D

<table>
<thead>
<tr>
<th>2</th>
<th>x : i_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_1 : [1, ∞)</td>
<td></td>
</tr>
</tbody>
</table>

i_1 = 0

E

<table>
<thead>
<tr>
<th>3</th>
<th>x : i_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_1 : [0, 0]</td>
<td></td>
</tr>
</tbody>
</table>

F

<table>
<thead>
<tr>
<th>1</th>
<th>x : i_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_2 : [0, ∞)</td>
<td></td>
</tr>
</tbody>
</table>

i_2 = i_1 - 1
```
If we translate the whole termination graph, we get the following constrained rewrite rules:

\[
\begin{align*}
\ell_A(i_1) & \to \ell_B(i_1) \quad [i_1 \geq 0] \\
\ell_A(i_1) & \to \ell_C(i_1) \quad [i_1 < 0] \\
\ell_B(i_1) & \to \ell_D(i_1) \quad [i_1 \neq 0 \land i_1 \geq 0] \\
\ell_D(i_1) & \to \ell_B(i_1 - 1) \quad [i_1 \geq 1] \\
\ell_B(i_1) & \to \ell_E(i_1) \quad [i_1 = 0]
\end{align*}
\]

Apart from the different names for function symbols and variables, we get essentially the same result as on slide 34, with two differences:

- Instead of \( \ell_3 \), we now have the two different end-of-program symbols \( \ell_C \) and \( \ell_E \).
- The second-to-last rule has \( i_1 \geq 1 \) as its condition, which is stronger than \( i_1 \geq 0 \) (i.e., \( i_1 \geq 1 \) implies \( i_1 \geq 0 \)).
We get the following termination graph for the program:
The SCC of the graph gives rise to the following constrained rewrite rules:

\[
\begin{align*}
\ell_B(jlO(List(eoc, o_2)), i_1) & \rightarrow \ell_C(jlO(List(eoc, o_2)), i_1) \\
\ell_C(jlO(List(eoc, o_2)), i_1) & \rightarrow \ell_E(o_2, i_1) \\
\ell_E(o_2, i_1) & \rightarrow \ell_B(o_2, i_1 + 1)
\end{align*}
\]

The dependency pairs for these rules are identical to the rules (except that we may rename the defined function symbols). The following polynomial interpretation \([ \cdot ]\) to \(\mathbb{N}\) lets us conclude the termination proof:

\[
\begin{align*}
[\ell_B](o, i) &= o + 1 \\
[\ell_C](o, i) &= o \\
[\ell_E](o, i) &= o + 2 \\
[\text{List}](c, o) &= o + 3 \\
[jlO](c) &= c \\
[eoc] &= 0
\end{align*}
\]

In practice, tools like AProve will first apply techniques to simplify and combine the obtained rewrite rules. Here we may obtain this single rule:

\[
\ell(jlO(List(eoc, o_2)), i_1) \rightarrow \ell(o_2, i_1 + 1)
\]

Details: [Giesl et al, JAR '17]