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

Proving Program Termination via Term Rewriting

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)
- $\mathsf{T_TT_2}$ (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)
- $\mathsf{T_TT_2}$ (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)
- $\mathsf{T_TT_2}$ (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?**

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
  - `http://www.exploringbinary.com/php-hangs-on-numeric-value-2-2250738585072011e-308/`
  - `http://www.exploringbinary.com/java-hangs-when-converting-2-2250738585072012e-308/`
- 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
  - `http://www.exploringbinary.com/php-hangs-on-numeric-value-2-2250738585072011e-308/`
  - `http://www.exploringbinary.com/java-hangs-when-converting-2-2250738585072012e-308/`
- value of termination analysis recognized by industry
  $\rightarrow$ Microsoft's Terminator project

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
$\rightarrow$ 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
$\rightarrow$ 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
$\rightarrow$ 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
$\rightarrow$ 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)

Example (Division)

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

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

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

Example (Division)

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

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

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

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

Example (Division)

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

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

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

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

Example (Division)

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

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

Example (Division)

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

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

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

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

Example (Division)

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

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

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

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

Example (Division)

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

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

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

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

Example (Division)

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

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

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

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

Example (Division)

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

$$\mathcal{P} = \left\{ \begin{array}{rcl} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \overset{\succ}{(\sim)} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \overset{\succ}{(\sim)} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \overset{\succ}{(\sim)} & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{array} \right.$$

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

- For TRS $\mathcal{R}$ build dependency pairs $\mathcal{P}$ ($\sim$ function calls)
- Show: No ∞ 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 $(\succsim, \succ)$ with $\mathcal{P} \cup \mathcal{R} \subseteq \succsim$
  - delete $s \to t$ with $s \succ t$ from $\mathcal{P}$ ($\succ$ well founded, *not monotonic*)

Example (Division)

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

$$\mathcal{P} = \left\{ \begin{array}{rcl} \mathsf{minus}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \underset{(\sim)}{\succ} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \underset{(\sim)}{\succ} & \mathsf{minus}^\sharp(x, y) \\ \mathsf{quot}^\sharp(\mathsf{s}(x), \mathsf{s}(y)) & \underset{(\sim)}{\succ} & \mathsf{quot}^\sharp(\mathsf{minus}(x, y), \mathsf{s}(y)) \end{array} \right.$$

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

- For TRS $\mathcal{R}$ build dependency pairs $\mathcal{P}$ (∼ function calls)
- Show: No ∞ 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 $(\succsim, \succ)$ with $\mathcal{P} \cup \mathcal{R} \subseteq \succsim$
  - delete $s \to t$ with $s \succ t$ from $\mathcal{P}$ ($\succ$ well founded, *not monotonic*)
- Find $(\succsim, \succ)$ automatically via SAT and SMT solving

Example (Constraints for Division)

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

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

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

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

$\curvearrowright$ $(\succsim, \succ)$ induced by $[\,\cdot\,]$ solves all term constraints

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

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

Example (Constraints for Division)

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

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

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

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

$\curvearrowright$ $(\succsim, \succ)$ induced by $[\,\cdot\,]$ solves all term constraints

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

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

Example (Constraints for Division)

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

$$\mathcal{P} = \left\{ \vphantom{\begin{array}{c} a \\ a \\ a \end{array}} \right.$$

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

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

$\curvearrowright$ $(\succsim, \succ)$ induced by $[\,\cdot\,]$ solves all term constraints

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

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

From Program to Term Rewriting, high-level

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

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

From Program to Term Rewriting, high-level

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

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

init(...)

From Program to Term Rewriting, high-level

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

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

$$\begin{array}{c}\mathsf{init}(\ldots)\\ \downarrow\\ \mathsf{f}(\ldots)\end{array}$$

From Program to Term Rewriting, high-level

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

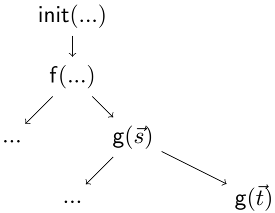
```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```

$$
\begin{array}{ccc}
 & \mathsf{init}(\dots) & \\
 & \downarrow & \\
 & \mathsf{f}(\dots) & \\
\swarrow & & \searrow \\
\dots & & \mathsf{g}(\vec{s})
\end{array}
$$

From Program to Term Rewriting, high-level

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

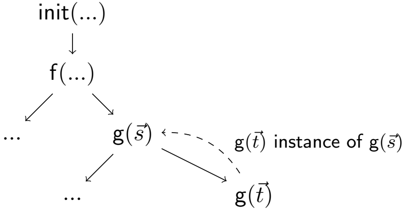
```
f : if ...
        ...
    else
        ...
        g : while ...
                  ...
```



From Program to Term Rewriting, high-level

- 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*]

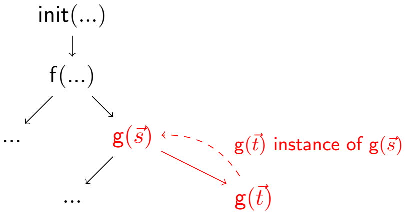
```
f : if ...
        ...
    else
        ...
        g : while ...
                ...
```



From Program to Term Rewriting, high-level

- 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

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



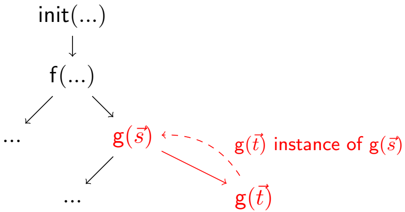
From Program to Term Rewriting, high-level

- 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**

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



Here: Haskell

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

$\Rightarrow$ Implies termination of the Haskell program!

Here: Haskell

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

$\Rightarrow$ 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 $\rightarrow \infty$ data (streams)
  But: TRS techniques analyze termination of all terms

Data Structures

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

Data Structures

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

Syntax of Haskell

Data Structures

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

Terms (well-typed)

- Variables: x, y, $\ldots$
- Function Symbols: constructors (Z, S, Nil, Cons) & defined (from, take)
- Applications $(t_1\ t_2)$
  - S Z represents number 1
  - Cons x Nil $\equiv$ (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 $\to$ Nat
- data List a = Nil | Cons a (List a)
  - type constructor: List of arity 1
  - data constructors: Nil :: List a, Cons :: $a \to$ (List a) $\to$ (List a)

Types

- Type Variables: a, b, ...
- Applications of type constructors to types: List Nat, $a \to$ (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 $\to$ Nat
- data List a = Nil | Cons a (List a)
  - type constructor: List of arity 1
  - data constructors: Nil :: List a, Cons :: $a \to$ (List a) $\to$ (List a)

Function Declarations (example)

$$\mathsf{from}\,x = \mathsf{Cons}\,x\,(\mathsf{from}\,(\mathsf{S}\,x))$$

$$\mathsf{from} :: \mathsf{Nat} \to \mathsf{List\,Nat}$$

$$\mathsf{from}\ x \ \equiv\ [x, x+1, x+2, \ldots]$$

$$\begin{aligned} \mathsf{take}\,\mathsf{Z}\,xs &= \mathsf{Nil} \\ \mathsf{take}\,n\,\mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{Cons}\,x\,xs) &= \mathsf{Cons}\,x\,(\mathsf{take}\,n\,xs) \end{aligned}$$

$$\mathsf{take} :: \mathsf{Nat} \to (\mathsf{List}\,a) \to (\mathsf{List}\,a)$$

$$\mathsf{take}\ n\ [x_1, \ldots, x_n, \ldots] \ \equiv\ [x_1, \ldots, x_n]$$

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)

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

$$\mathsf{from} :: \mathsf{Nat} \to \mathsf{List\ Nat}$$

$$\mathsf{from}\ x \equiv [x, x+1, x+2, \ldots]$$

$$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$$
$$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$$
$$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)$$

$$\mathsf{take} :: \mathsf{Nat} \to (\mathsf{List}\ a) \to (\mathsf{List}\ a)$$

$$\mathsf{take}\ n\ [x_1, \ldots, x_n, \ldots] \equiv [x_1, \ldots, x_n]$$

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
- ...

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	$\backslash\, u\ m \to \mathsf{take}\, u\, (\mathsf{from}\, m)$
by	f
where	$\mathsf{f}\ u\ m = \ \mathsf{take}\, u\, (\mathsf{from}\, m)$

- 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	$\backslash \quad m \to \mathsf{take}\, u\, (\mathsf{from}\, m)$
by	$\mathsf{f}\ u$
where	$\mathsf{f}\ u\ m = \mathsf{take}\, u\, (\mathsf{from}\, m)$

- 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 $\backslash\, t_1 \ldots t_n \to t$ with free variables $x_1, \ldots, x_m$
  by $\mathsf{f}\, x_1 \ldots x_m$
  where $\mathsf{f}\, x_1 \ldots x_m\, t_1 \ldots t_n = t$

- 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 $\backslash\, t_1 \ldots t_n \rightarrow t$ with free variables $x_1, \ldots, x_m$
  by $\mathsf{f}\; x_1 \ldots x_m$
  where $\mathsf{f}\; x_1 \ldots x_m\, t_1 \ldots t_n = t$

- Conditions
- Local Declarations
- ...

Semantics of Haskell

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

$$\begin{aligned}
\mathsf{take}\ \mathsf{Z}\ xs &= \mathsf{Nil}\\
\mathsf{take}\ n\ \mathsf{Nil} &= \mathsf{Nil}\\
\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) &= \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)
\end{aligned}$$

- **Evaluation Relation** $\rightarrow_{\mathsf{H}}$

$$\mathsf{from}\ \mathsf{Z}$$

Semantics of Haskell

$\mathsf{from}\, x = \mathsf{Cons}\, x\, (\mathsf{from}\, (\mathsf{S}\, x))$

$\mathsf{take}\, \mathsf{Z}\, xs = \mathsf{Nil}$
$\mathsf{take}\, n\, \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\, (\mathsf{S}\, n)\, (\mathsf{Cons}\, x\, xs) = \mathsf{Cons}\, x\, (\mathsf{take}\, n\, xs)$

- **Evaluation Relation** $\rightarrow_{\mathsf{H}}$

from Z

evaluation position

Semantics of Haskell

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

$$\begin{aligned} \mathsf{take}\ \mathsf{Z}\ xs &= \mathsf{Nil} \\ \mathsf{take}\ n\ \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) &= \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs) \end{aligned}$$

- **Evaluation Relation $\to_{\mathsf{H}}$**

$$\begin{aligned} & \mathsf{from}\ \mathsf{Z} \\ \to_{\mathsf{H}}\ & \mathsf{Cons}\ \mathsf{Z}\ (\mathsf{from}\ (\mathsf{S}\ \mathsf{Z})) \end{aligned}$$

evaluation position

Semantics of Haskell

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

$$\begin{aligned}\mathsf{take}\ \mathsf{Z}\ xs &= \mathsf{Nil}\\ \mathsf{take}\ n\ \mathsf{Nil} &= \mathsf{Nil}\\ \mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) &= \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)\end{aligned}$$

- **Evaluation Relation** $\to_{\mathsf{H}}$

$$\begin{aligned}&\mathsf{from}\ \mathsf{Z}\\ \to_{\mathsf{H}}\ &\mathsf{Cons}\ \mathsf{Z}\ (\mathsf{from}\ (\mathsf{S}\ \mathsf{Z}))\end{aligned}$$

evaluation position

Semantics of Haskell

$$\mathsf{from}\, x = \mathsf{Cons}\, x\, (\mathsf{from}\, (\mathsf{S}\, x))$$

$$\begin{aligned} \mathsf{take}\, \mathsf{Z}\, xs &= \mathsf{Nil} \\ \mathsf{take}\, n\, \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{take}\, (\mathsf{S}\, n)\, (\mathsf{Cons}\, x\, xs) &= \mathsf{Cons}\, x\, (\mathsf{take}\, n\, xs) \end{aligned}$$

- **Evaluation Relation $\to_{\mathsf{H}}$**

$$\begin{aligned} & \mathsf{from}\, \mathsf{Z} \\ \to_{\mathsf{H}}\ & \mathsf{Cons}\, \mathsf{Z}\, (\mathsf{from}\, (\mathsf{S}\, \mathsf{Z})) \\ \to_{\mathsf{H}}\ & \mathsf{Cons}\, \mathsf{Z}\, (\mathsf{Cons}\, (\mathsf{S}\, \mathsf{Z})\, (\mathsf{from}\, (\mathsf{S}\, (\mathsf{S}\, \mathsf{Z})))) \end{aligned}$$

evaluation position

Semantics of Haskell

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

$$\begin{aligned}\mathsf{take}\ \mathsf{Z}\ xs &= \mathsf{Nil}\\ \mathsf{take}\ n\ \mathsf{Nil} &= \mathsf{Nil}\\ \mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) &= \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)\end{aligned}$$

- **Evaluation Relation $\rightarrow_{\mathsf{H}}$**

$$\begin{aligned} & \mathsf{from}\ \mathsf{Z}\\ \rightarrow_{\mathsf{H}}\ & \mathsf{Cons}\ \mathsf{Z}\ (\mathsf{from}\ (\mathsf{S}\ \mathsf{Z}))\\ \rightarrow_{\mathsf{H}}\ & \mathsf{Cons}\ \mathsf{Z}\ (\mathsf{Cons}\ (\mathsf{S}\ \mathsf{Z})\ (\mathsf{from}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z})))) \qquad \textit{evaluation position}\end{aligned}$$

$$\mathsf{from}\ x = \mathsf{Cons}\ x\ (\mathsf{from}\ (\mathsf{S}\ x)) \qquad \begin{aligned} \mathsf{take}\ \mathsf{Z}\ xs &= \mathsf{Nil} \\ \mathsf{take}\ n\ \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) &= \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs) \end{aligned}$$

- **Evaluation Relation** $\rightarrow_{\mathsf{H}}$

$$\begin{aligned} & \mathsf{from}\ \mathsf{Z} \\ \rightarrow_{\mathsf{H}}\ & \mathsf{Cons}\ \mathsf{Z}\ (\mathsf{from}\ (\mathsf{S}\ \mathsf{Z})) \\ \rightarrow_{\mathsf{H}}\ & \mathsf{Cons}\ \mathsf{Z}\ (\mathsf{Cons}\ (\mathsf{S}\ \mathsf{Z})\ (\mathsf{from}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z})))) \qquad \textit{evaluation position} \\ \rightarrow_{\mathsf{H}}\ & \ldots \end{aligned}$$

Semantics of Haskell

$$\mathsf{from}\, x = \mathsf{Cons}\, x\, (\mathsf{from}\, (\mathsf{S}\, x))$$

$$\begin{aligned} \mathsf{take}\, \mathsf{Z}\, xs &= \mathsf{Nil} \\ \mathsf{take}\, n\, \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{take}\, (\mathsf{S}\, n)\, (\mathsf{Cons}\, x\, xs) &= \mathsf{Cons}\, x\, (\mathsf{take}\, n\, xs) \end{aligned}$$

- **Evaluation Relation** $\to_{\mathsf{H}}$

$$\begin{aligned} & \mathsf{from}\, m \\ \to_{\mathsf{H}}\ & \mathsf{Cons}\, m\, (\mathsf{from}\, (\mathsf{S}\, m)) \\ \to_{\mathsf{H}}\ & \mathsf{Cons}\, m\, (\mathsf{Cons}\, (\mathsf{S}\, m)\, (\mathsf{from}\, (\mathsf{S}\, (\mathsf{S}\, m)))) \\ \to_{\mathsf{H}}\ & \ldots \end{aligned}$$

Semantics of Haskell

$$\mathsf{from}\, x = \mathsf{Cons}\, x\, (\mathsf{from}\, (\mathsf{S}\, x))$$

$$\begin{aligned}\mathsf{take}\,\mathsf{Z}\, xs &= \mathsf{Nil}\\ \mathsf{take}\, n\, \mathsf{Nil} &= \mathsf{Nil}\\ \mathsf{take}\,(\mathsf{S}\, n)\,(\mathsf{Cons}\, x\, xs) &= \mathsf{Cons}\, x\,(\mathsf{take}\, n\, xs)\end{aligned}$$

- **Evaluation Relation** $\rightarrow_{\mathsf{H}}$

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

Semantics of Haskell

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

$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$
$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)$

- **Evaluation Relation** $\to_{\mathsf{H}}$

$\mathsf{take}\ (\mathsf{S}\ \mathsf{Z})\ (\mathsf{from}\ m)$ *evaluation position*

Semantics of Haskell

$$\mathsf{from}\,x = \mathsf{Cons}\,x\,(\mathsf{from}\,(\mathsf{S}\,x))$$

$$\begin{aligned}\mathsf{take}\,\mathsf{Z}\,xs &= \mathsf{Nil}\\ \mathsf{take}\,n\,\mathsf{Nil} &= \mathsf{Nil}\\ \mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{Cons}\,x\,xs) &= \mathsf{Cons}\,x\,(\mathsf{take}\,n\,xs)\end{aligned}$$

- **Evaluation Relation** $\to_{\mathsf{H}}$

$$\begin{aligned} & \mathsf{take}\,(\mathsf{S}\,\mathsf{Z})\,(\mathsf{from}\,m) \\ \to_{\mathsf{H}}\ & \mathsf{take}\,(\mathsf{S}\,\mathsf{Z})\,(\mathsf{Cons}\,m\,(\mathsf{from}\,(\mathsf{S}\,m)))\end{aligned}$$

evaluation position

$$\mathsf{from}\; x = \mathsf{Cons}\; x\; (\mathsf{from}\; (\mathsf{S}\; x)) \qquad \begin{aligned} \mathsf{take}\; \mathsf{Z}\; xs &= \mathsf{Nil} \\ \mathsf{take}\; n\; \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{take}\; (\mathsf{S}\; n)\; (\mathsf{Cons}\; x\; xs) &= \mathsf{Cons}\; x\; (\mathsf{take}\; n\; xs) \end{aligned}$$

- **Evaluation Relation** $\to_{\mathsf{H}}$

$$\begin{aligned} & \mathsf{take}\; (\mathsf{S}\; \mathsf{Z})\; (\mathsf{from}\; m) \\ \to_{\mathsf{H}}\; & \mathsf{take}\; (\mathsf{S}\; \mathsf{Z})\; (\mathsf{Cons}\; m\; (\mathsf{from}\; (\mathsf{S}\; m))) \end{aligned}$$

evaluation position

Semantics of Haskell

$$\mathsf{from}\; x = \mathsf{Cons}\; x\; (\mathsf{from}\; (\mathsf{S}\; x))$$

$$\begin{aligned}
\mathsf{take}\; \mathsf{Z}\; xs &= \mathsf{Nil} \\
\mathsf{take}\; n\; \mathsf{Nil} &= \mathsf{Nil} \\
\mathsf{take}\; (\mathsf{S}\; n)\; (\mathsf{Cons}\; x\; xs) &= \mathsf{Cons}\; x\; (\mathsf{take}\; n\; xs)
\end{aligned}$$

- **Evaluation Relation $\rightarrow_{\mathsf{H}}$**

$$\begin{aligned}
& \mathsf{take}\; (\mathsf{S}\; \mathsf{Z})\; (\mathsf{from}\; m) \\
\rightarrow_{\mathsf{H}}\; & \mathsf{take}\; (\mathsf{S}\; \mathsf{Z})\; (\mathsf{Cons}\; m\; (\mathsf{from}\; (\mathsf{S}\; m))) \qquad \textit{evaluation position} \\
\rightarrow_{\mathsf{H}}\; & \mathsf{Cons}\; m\; (\mathsf{take}\; \mathsf{Z}\; (\mathsf{from}\; (\mathsf{S}\; m)))
\end{aligned}$$

$$\mathsf{from}\, x = \mathsf{Cons}\, x\, (\mathsf{from}\, (\mathsf{S}\, x)) \qquad \begin{aligned} \mathsf{take}\, \mathsf{Z}\, xs &= \mathsf{Nil} \\ \mathsf{take}\, n\, \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{take}\, (\mathsf{S}\, n)\, (\mathsf{Cons}\, x\, xs) &= \mathsf{Cons}\, x\, (\mathsf{take}\, n\, xs) \end{aligned}$$

- **Evaluation Relation** $\to_{\mathsf{H}}$

$$\begin{aligned} & \mathsf{take}\, (\mathsf{S}\,\mathsf{Z})\, (\mathsf{from}\, m) \\ \to_{\mathsf{H}}\ & \mathsf{take}\, (\mathsf{S}\,\mathsf{Z})\, (\mathsf{Cons}\, m\, (\mathsf{from}\, (\mathsf{S}\, m))) \\ \to_{\mathsf{H}}\ & \mathsf{Cons}\, m\, (\mathsf{take}\, \mathsf{Z}\, (\mathsf{from}\, (\mathsf{S}\, m))) \end{aligned}$$

evaluation position

$$\mathsf{from}\, x = \mathsf{Cons}\, x\, (\mathsf{from}\, (\mathsf{S}\, x)) \qquad \begin{aligned} \mathsf{take}\, \mathsf{Z}\, xs &= \mathsf{Nil} \\ \mathsf{take}\, n\, \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{take}\, (\mathsf{S}\, n)\, (\mathsf{Cons}\, x\, xs) &= \mathsf{Cons}\, x\, (\mathsf{take}\, n\, xs) \end{aligned}$$

- **Evaluation Relation $\to_{\mathsf{H}}$**

$$\begin{aligned} & \mathsf{take}\, (\mathsf{S}\, \mathsf{Z})\, (\mathsf{from}\, m) \\ \to_{\mathsf{H}}\; & \mathsf{take}\, (\mathsf{S}\, \mathsf{Z})\, (\mathsf{Cons}\, m\, (\mathsf{from}\, (\mathsf{S}\, m))) \\ \to_{\mathsf{H}}\; & \mathsf{Cons}\, m\, (\mathsf{take}\, \mathsf{Z}\, (\mathsf{from}\, (\mathsf{S}\, m))) \qquad \textit{evaluation position} \\ \to_{\mathsf{H}}\; & \mathsf{Cons}\, m\, \mathsf{Nil} \end{aligned}$$

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 $t \to_{\mathsf{H}} \ldots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$, f defined, $n < \mathrm{arity}(f)$, then $(f\ t_1 \ldots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \ldots t_n)$, 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 σ 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 $t \to_{\mathsf{H}} \ldots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$, f defined, $n < \mathrm{arity}(f)$, then $(f\ t_1 \ldots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \ldots t_n)$, 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 σ 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 $t \to_{\mathsf{H}} \dots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \dots t_n)$, f defined, $n < \mathrm{arity}(f)$, then $(f\ t_1 \dots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \dots t_n)$, c constructor, then $t_1, \dots, t_n$ are also H-terminating.
- **H-Termination** of arbitrary term t if
  $t\sigma$ H-terminates for all substitutions σ 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 $t \to_{\mathsf{H}} \ldots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$, f defined, $n < \mathrm{arity}(f)$,
    then $(f\ t_1 \ldots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \ldots t_n)$, 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 σ with H-terminating terms.

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 $t \to_{\mathsf{H}} \ldots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$, f defined, $n < \mathrm{arity}(f)$, then $(f\ t_1 \ldots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \ldots t_n)$, 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 σ 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 $t \to_{\mathsf{H}} \dots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \dots t_n)$, f defined, $n < \mathrm{arity}(f)$, then $(f\ t_1 \dots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \dots t_n)$, c constructor, then $t_1, \dots, t_n$ are also H-terminating.
- **H-Termination** of arbitrary term t if
  $t\sigma$ H-terminates for all substitutions σ with H-terminating terms.

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 $t \to_{\mathsf{H}} \dots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \dots t_n)$, f defined, $n < \mathrm{arity}(f)$,
    then $(f\ t_1 \dots t_n\, t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \dots t_n)$, c constructor,
    then $t_1, \dots, t_n$ are also H-terminating.
- **H-Termination** of arbitrary term t if
  $t\sigma$ H-terminates for all substitutions σ 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 $t \to_{\mathsf{H}} \ldots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$, f defined, $n < \mathrm{arity}(f)$,
    then $(f\ t_1 \ldots t_n\, t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \ldots t_n)$, 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 σ 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 $t \to_{\mathsf{H}} \dots$
  - if $t \to_{\mathsf{H}}^* (f\ t_1 \dots t_n)$, f defined, $n < \mathrm{arity}(f)$, then $(f\ t_1 \dots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^* (c\ t_1 \dots t_n)$, c constructor, then $t_1, \dots, t_n$ are also H-terminating.
- **H-Termination** of arbitrary term t if $t\sigma$ H-terminates for all substitutions σ 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 $t \to_{\mathsf{H}} \ldots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$, f defined, $n < \mathrm{arity}(f)$,
    then $(f\ t_1 \ldots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \ldots t_n)$, 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 σ 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 $t \to_{\mathsf{H}} \ldots$
  - if $t \to_{\mathsf{H}}^* (f\ t_1 \ldots t_n)$, f defined, $n < \mathrm{arity}(f)$,
    then $(f\ t_1 \ldots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^* (c\ t_1 \ldots t_n)$, 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 σ with H-terminating terms.
- x is H-terminating
  from is not H-terminating (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 $t \to_{\mathsf{H}} \ldots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \ldots t_n)$, f defined, $n < \mathrm{arity}(f)$,
    then $(f\ t_1 \ldots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \ldots t_n)$, 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 σ 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 $t \to_{\mathsf{H}} \ldots$
  - if $t \to_{\mathsf{H}}^* (f\ t_1 \ldots t_n)$, f defined, $n < \mathrm{arity}(f)$,
    then $(f\ t_1 \ldots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^* (c\ t_1 \ldots t_n)$, 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 σ 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 $t \to_{\mathsf{H}} \dots$
  - if $t \to_{\mathsf{H}}^{*} (f\ t_1 \dots t_n)$, f defined, $n < \mathrm{arity}(f)$, then $(f\ t_1 \dots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^{*} (c\ t_1 \dots t_n)$, c constructor, then $t_1, \dots, t_n$ are also H-terminating.
- **H-Termination** of arbitrary term t if $t\sigma$ H-terminates for all substitutions σ 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)

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 $t \to_{\mathsf{H}} \dots$
  - if $t \to_{\mathsf{H}}^* (f\ t_1 \dots t_n)$, f defined, $n < \mathrm{arity}(f)$,
    then $(f\ t_1 \dots t_n\ t')$ is also H-terminating if t' is H-terminating
  - if $t \to_{\mathsf{H}}^* (c\ t_1 \dots t_n)$, c constructor,
    then $t_1, \dots, t_n$ are also H-terminating.
- **H-Termination** of arbitrary term t if
  $t\sigma$ H-terminates for all substitutions σ 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) is not H-terminating

From Haskell to Termination Graphs

$\mathsf{from}\,x = \mathsf{Cons}\,x\,(\mathsf{from}\,(\mathsf{S}\,x))$

$\mathsf{take}\,\mathsf{Z}\,xs = \mathsf{Nil}$
$\mathsf{take}\,n\,\mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{Cons}\,x\,xs) = \mathsf{Cons}\,x\,(\mathsf{take}\,n\,xs)$

Goal: Prove (H-)termination of initial term $\mathsf{take}\,u\,(\mathsf{from}\,m)$

From Haskell to Termination Graphs

$\mathsf{from}\,x = \mathsf{Cons}\,x\,(\mathsf{from}\,(\mathsf{S}\,x))$

$\mathsf{take}\,\mathsf{Z}\,xs = \mathsf{Nil}$
$\mathsf{take}\,n\,\mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{Cons}\,x\,xs) = \mathsf{Cons}\,x\,(\mathsf{take}\,n\,xs)$

Goal: Prove (H-)termination of initial term $\mathsf{take}\,u\,(\mathsf{from}\,m)$

Naive approach

- Use defining equations directly
- fails, since from is not terminating
- disregards Haskell's evaluation strategy

From Haskell to Termination Graphs

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

$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$
$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)$

Goal: Prove (H-)termination of initial term take u (from m)

Naive approach

- Use defining equations directly
- fails, since from is not terminating
- disregards Haskell's evaluation strategy

Our approach [Giesl et al, *TOPLAS '11*]

- evaluate **initial term** a few steps
  $\Rightarrow$ termination graph ($\approx$ abstract interpretation)
- our "abstract domain" for Haskell program states: *a single term*
- do not transform Haskell into TRS directly,
  but transform termination graph into TRS

From Haskell to Termination Graphs

$\mathsf{from}\, x = \mathsf{Cons}\, x\, (\mathsf{from}\, (\mathsf{S}\, x))$

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

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

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

$\boxed{\mathsf{take}\, u\, (\mathsf{from}\, m)}$

- begin with node marked with initial term
- 4 expansion rules to add children to leaves (more in paper)
- expansion rules try to *evaluate* terms

From Haskell to Termination Graphs

$\mathsf{from}\, x = \mathsf{Cons}\, x\, (\mathsf{from}\, (\mathsf{S}\, x))$

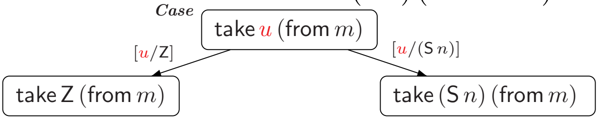
$\mathsf{take}\, \mathsf{Z}\, xs = \mathsf{Nil}$
$\mathsf{take}\, n\, \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\, (\mathsf{S}\, n)\, (\mathsf{Cons}\, x\, xs) = \mathsf{Cons}\, x\, (\mathsf{take}\, n\, xs)$

$\boxed{\mathsf{take}\, u\, (\mathsf{from}\, m)}$

- ***Case* rule**:
  - evaluation has to continue with variable u
  - instantiate u by all possible constructor terms of correct type

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

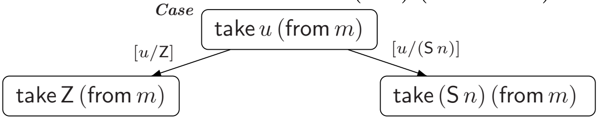
$$\begin{aligned}\mathsf{take}\ \mathsf{Z}\ xs &= \mathsf{Nil}\\ \mathsf{take}\ n\ \mathsf{Nil} &= \mathsf{Nil}\\ \mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) &= \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)\end{aligned}$$



- *Case* rule:
  - evaluation has to continue with variable u
  - instantiate u by all possible constructor terms of correct type

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

$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$
$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)$



- **Main Property of Termination Graphs**:

  A node is H-terminating if all its children are H-terminating.

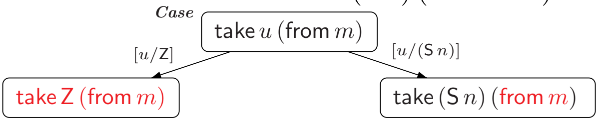
From Haskell to Termination Graphs

$\mathsf{from}\, x = \mathsf{Cons}\, x\, (\mathsf{from}\, (\mathsf{S}\, x))$

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

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

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

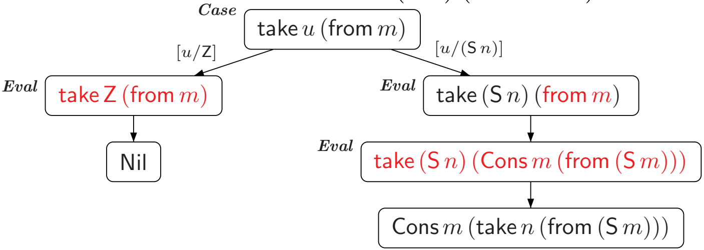


- ***Eval*** rule:

  performs one evaluation step with $\to_{\mathsf{H}}$

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

$$\begin{aligned}\mathsf{take}\ \mathsf{Z}\ xs &= \mathsf{Nil}\\ \mathsf{take}\ n\ \mathsf{Nil} &= \mathsf{Nil}\\ \mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) &= \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)\end{aligned}$$



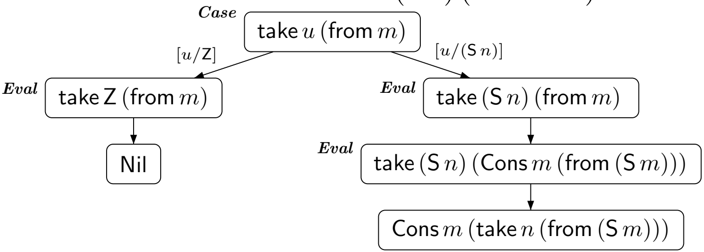
- ***Eval*** **rule:**

  performs one evaluation step with $\to_{\mathsf{H}}$

From Haskell to Termination Graphs

$\mathsf{from}\,x = \mathsf{Cons}\,x\,(\mathsf{from}\,(\mathsf{S}\,x))$

$\mathsf{take}\,\mathsf{Z}\,xs = \mathsf{Nil}$
$\mathsf{take}\,n\,\mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{Cons}\,x\,xs) = \mathsf{Cons}\,x\,(\mathsf{take}\,n\,xs)$

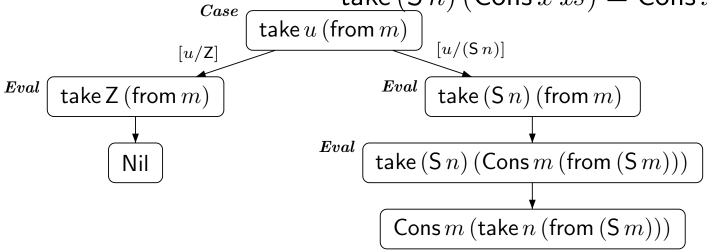


Case and ***Eval*** rule perform *narrowing*
w.r.t. Haskell's evaluation strategy and types

From Haskell to Termination Graphs

$\mathsf{from}\,x = \mathsf{Cons}\,x\,(\mathsf{from}\,(\mathsf{S}\,x))$

$\mathsf{take}\,\mathsf{Z}\,xs = \mathsf{Nil}$
$\mathsf{take}\,n\,\mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{Cons}\,x\,xs) = \mathsf{Cons}\,x\,(\mathsf{take}\,n\,xs)$



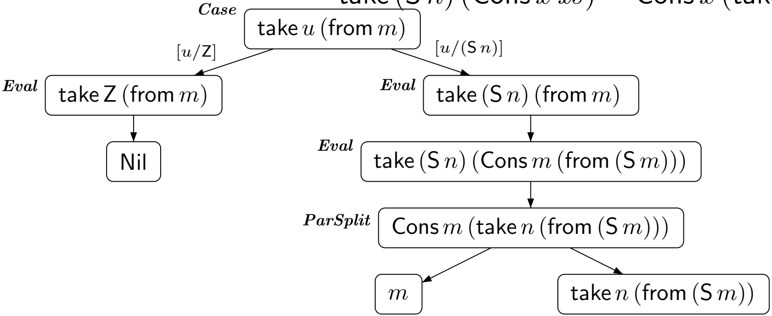
- ***ParSplit*** rule:

  if head of term is a constructor like Cons or a variable,
  then continue with the parameters

From Haskell to Termination Graphs

$\mathsf{from}\; x = \mathsf{Cons}\; x\; (\mathsf{from}\; (\mathsf{S}\; x))$

$\mathsf{take}\; \mathsf{Z}\; xs = \mathsf{Nil}$
$\mathsf{take}\; n\; \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\; (\mathsf{S}\; n)\; (\mathsf{Cons}\; x\; xs) = \mathsf{Cons}\; x\; (\mathsf{take}\; n\; xs)$



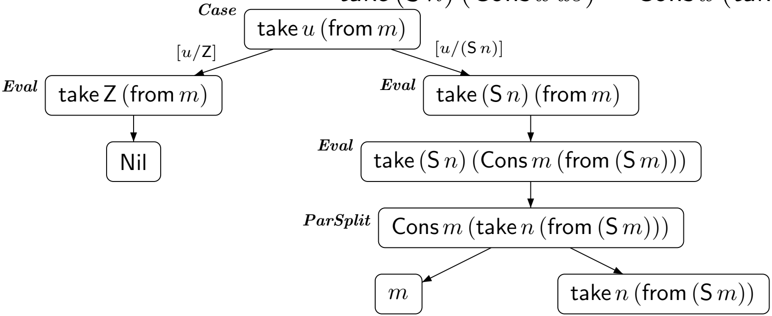
- ***ParSplit*** rule:

  if head of term is a constructor like Cons or a variable,
  then continue with the parameters

From Haskell to Termination Graphs

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

$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$
$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)$

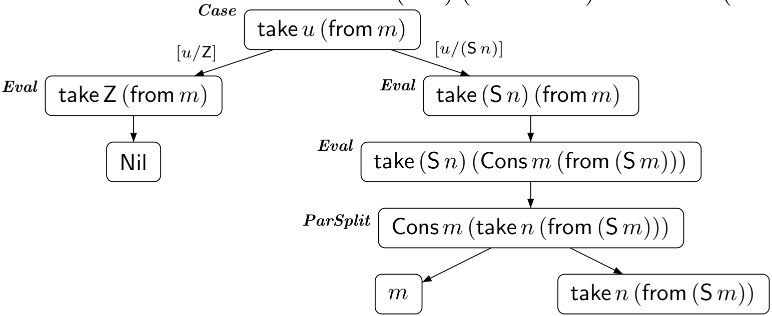


- one could continue with ***Case***, ***Eval***, ***ParSplit***
  $\Rightarrow$ infinite tree
- Instead: ***Ins*** rule to obtain finite graphs

From Haskell to Termination Graphs

$$\mathsf{from}\; x = \mathsf{Cons}\; x\; (\mathsf{from}\; (\mathsf{S}\; x))$$

$$\begin{aligned} \mathsf{take}\; \mathsf{Z}\; xs &= \mathsf{Nil} \\ \mathsf{take}\; n\; \mathsf{Nil} &= \mathsf{Nil} \\ \mathsf{take}\; (\mathsf{S}\; n)\; (\mathsf{Cons}\; x\; xs) &= \mathsf{Cons}\; x\; (\mathsf{take}\; n\; xs) \end{aligned}$$

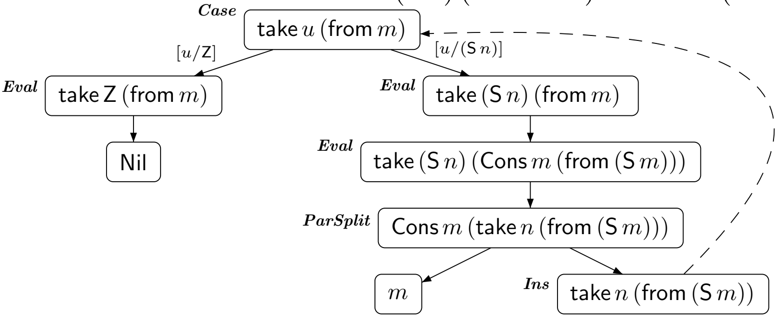


- ***Ins* rule:**
  - if leaf t is instance of t', then add instantiation edge from t to t'
  - one may re-use an existing node for t', if possible

From Haskell to Termination Graphs

$$\mathsf{from}\; x = \mathsf{Cons}\; x\; (\mathsf{from}\; (\mathsf{S}\; x))$$

$$\begin{aligned}\mathsf{take}\;\mathsf{Z}\; xs &= \mathsf{Nil}\\ \mathsf{take}\; n\;\mathsf{Nil} &= \mathsf{Nil}\\ \mathsf{take}\;(\mathsf{S}\; n)\;(\mathsf{Cons}\; x\; xs) &= \mathsf{Cons}\; x\;(\mathsf{take}\; n\; xs)\end{aligned}$$

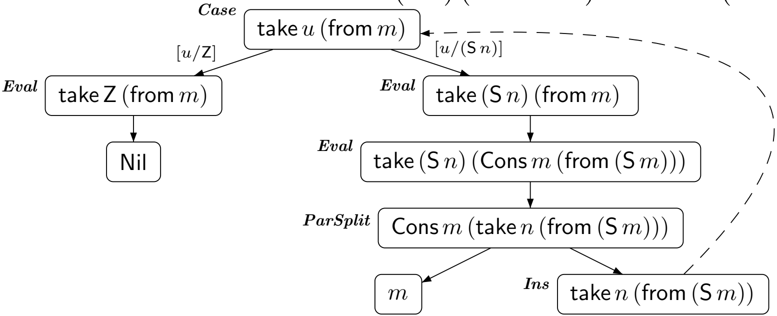


- ***Ins*** rule:
  - if leaf t is instance of t', then add instantiation edge from t to t'
  - one may re-use an existing node for t', if possible

From Haskell to Termination Graphs

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

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

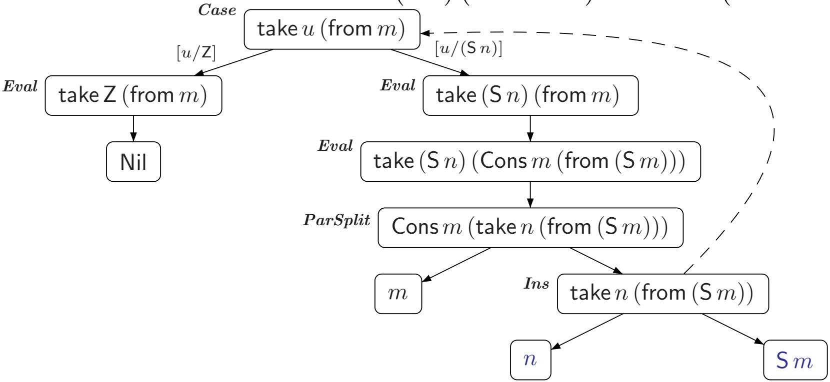


- ***Ins*** rule:
  - if leaf t is instance of t', then add instantiation edge from t to t'
  - since instantiation is $[u/n,\ m/(\text{S}\ m)]$, add child nodes n and $(\text{S}\ m)$

From Haskell to Termination Graphs

$\mathsf{from}\; x = \mathsf{Cons}\; x\; (\mathsf{from}\; (\mathsf{S}\; x))$

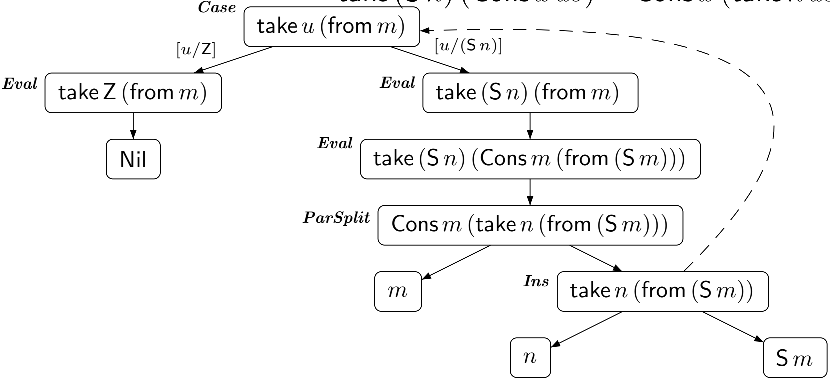
$\mathsf{take}\; \mathsf{Z}\; xs = \mathsf{Nil}$
$\mathsf{take}\; n\; \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\; (\mathsf{S}\; n)\; (\mathsf{Cons}\; x\; xs) = \mathsf{Cons}\; x\; (\mathsf{take}\; n\; xs)$



- ***Ins* rule:**
  - if leaf t is instance of t', then add instantiation edge from t to t'
  - since instantiation is $[u/n,\; m/(\mathsf{S}\; m)]$, add child nodes n and $(\mathsf{S}\; m)$

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

$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$
$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)$

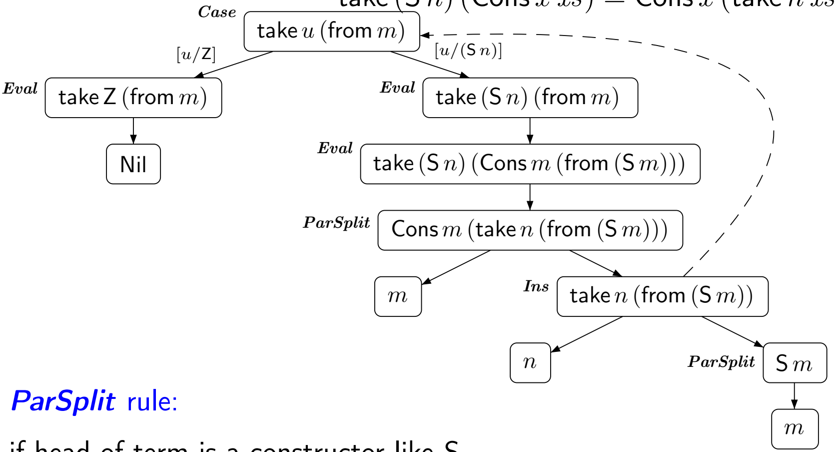


- ***ParSplit*** rule:

  if head of term is a constructor like S,
  then continue with the parameter

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

$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$
$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)$

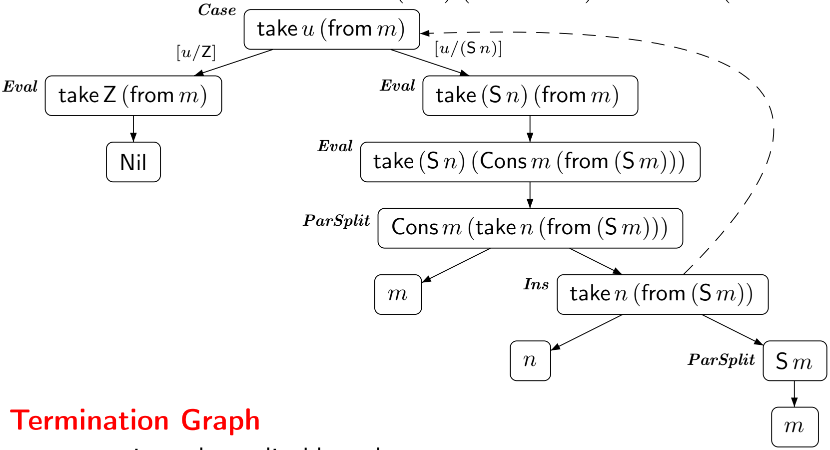


- ***ParSplit*** rule:

  if head of term is a constructor like S,
  then continue with the parameter

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

$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$
$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)$



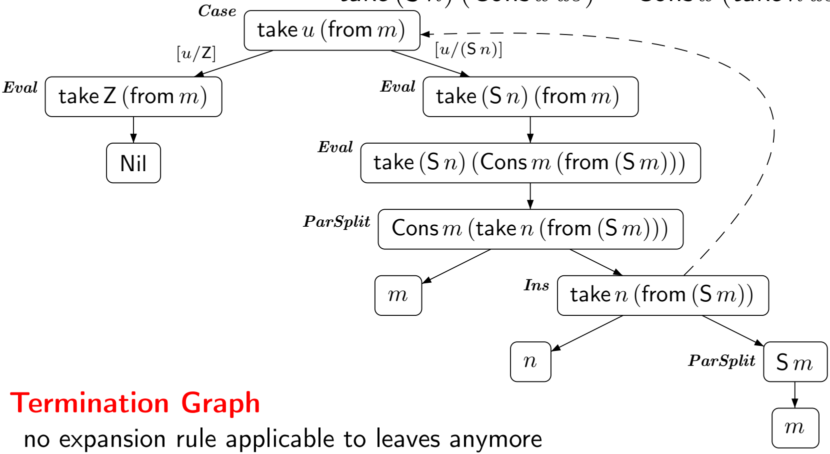
- **Termination Graph**

  no expansion rule applicable to leaves anymore

From Haskell to Termination Graphs

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

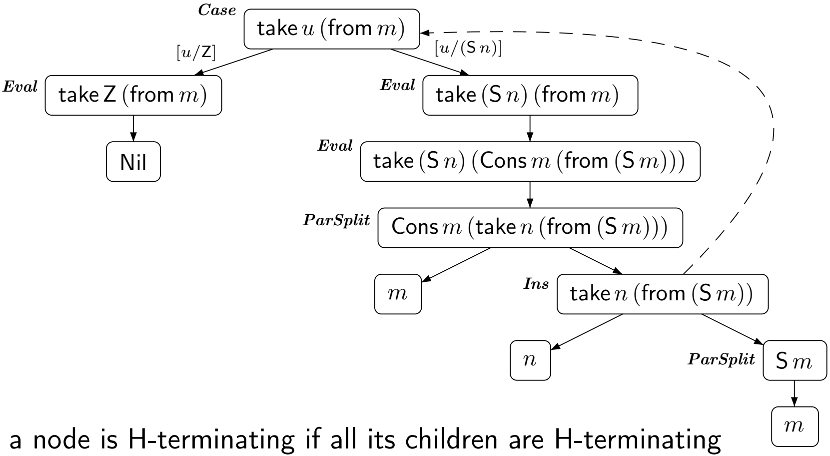
$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$
$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$
$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ n\ xs)$



- **Termination Graph**
  no expansion rule applicable to leaves anymore
- **Goal:** Prove H-termination of all terms in termination graph

From Termination Graphs to DP Problems

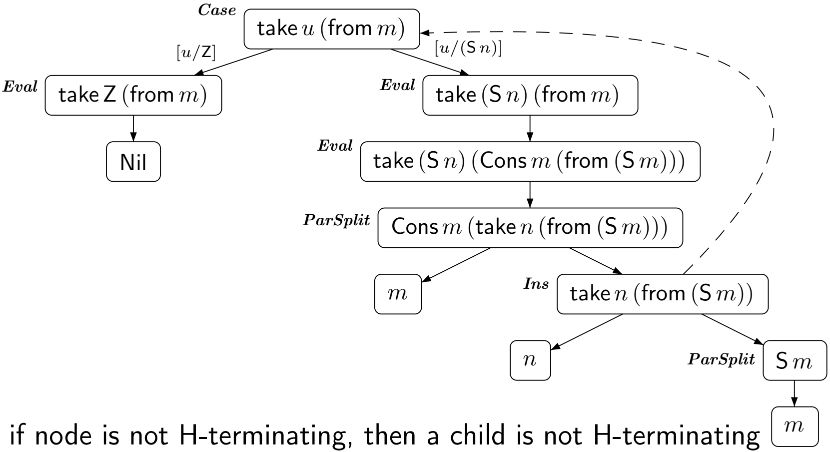
- Prove H-termination of all terms in termination graph



- a node is H-terminating if all its children are H-terminating

From Termination Graphs to DP Problems

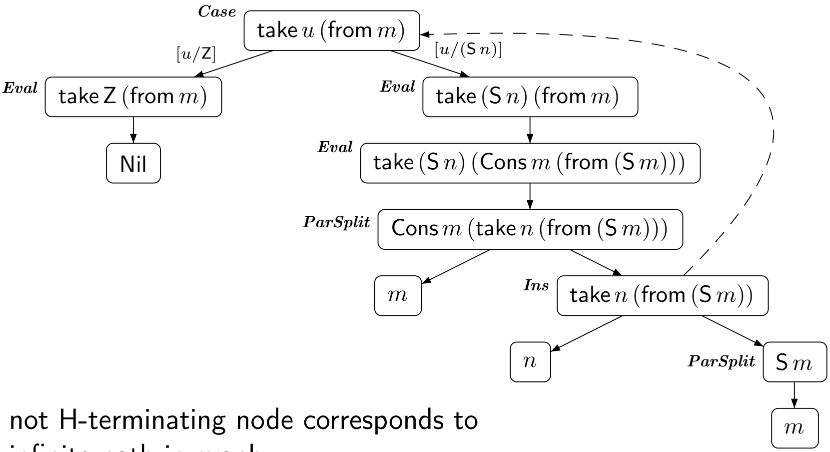
- Prove H-termination of all terms in termination graph



- if node is not H-terminating, then a child is not H-terminating

From Termination Graphs to DP Problems

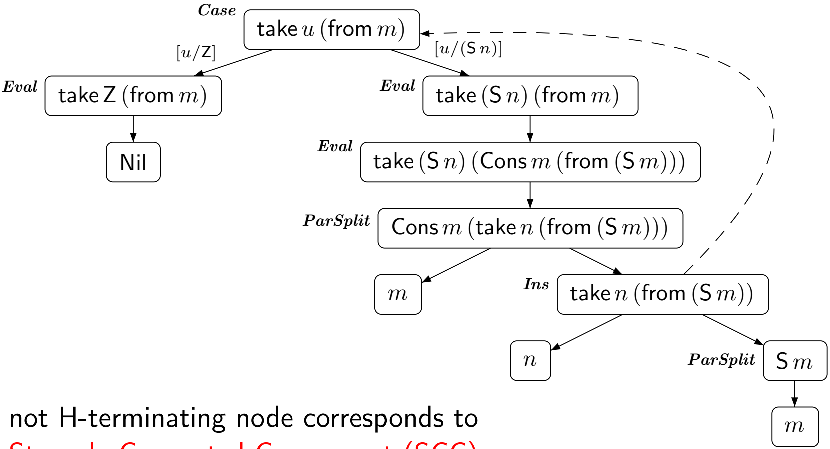
- Prove H-termination of all terms in termination graph



- not H-terminating node corresponds to infinite path in graph

From Termination Graphs to DP Problems

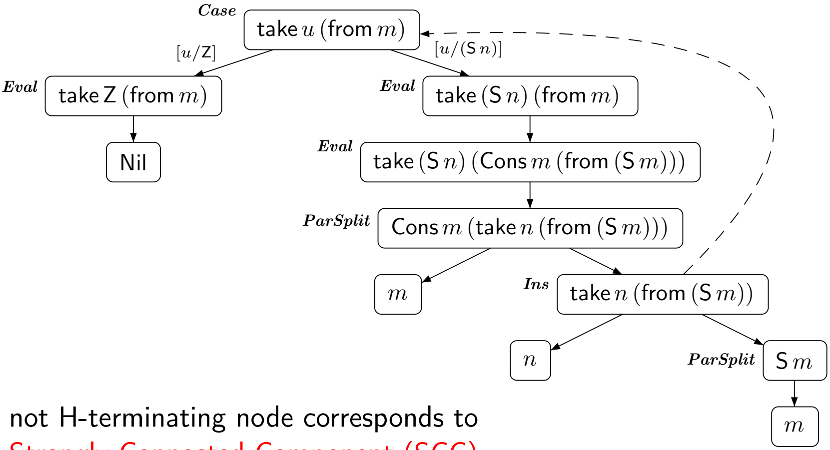
- Prove H-termination of all terms in termination graph



- not H-terminating node corresponds to Strongly Connected Component (SCC)

From Termination Graphs to DP Problems

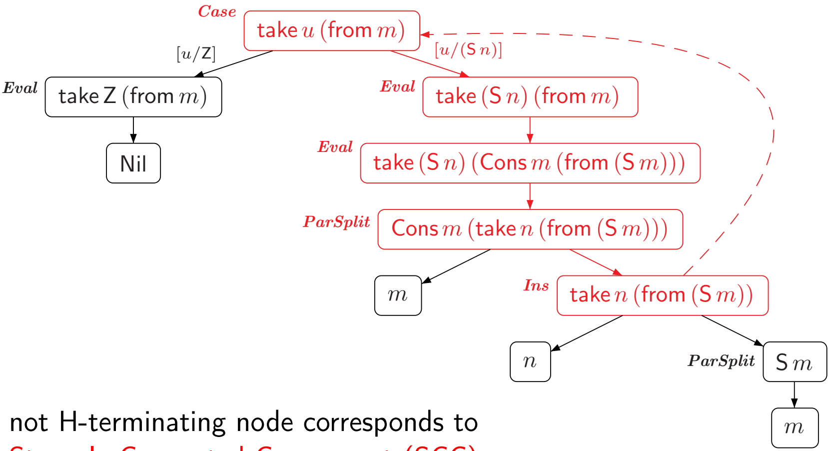
- Prove H-termination of all terms for each SCC



- not H-terminating node corresponds to Strongly Connected Component (SCC)

From Termination Graphs to DP Problems

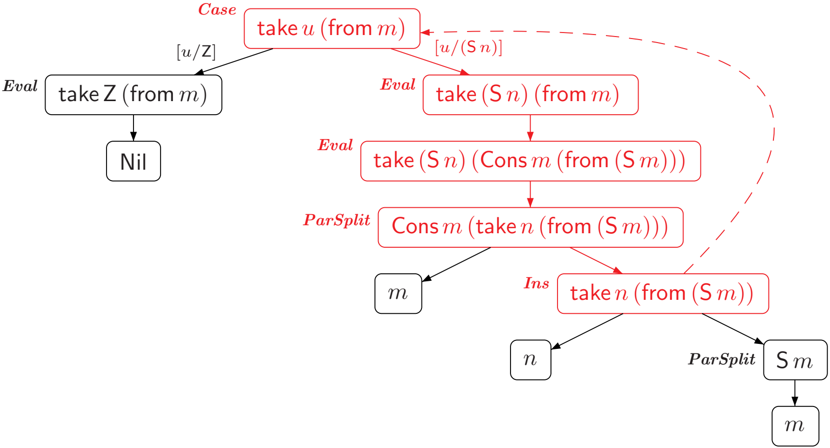
- Prove H-termination of all terms for each SCC



- not H-terminating node corresponds to Strongly Connected Component (SCC)

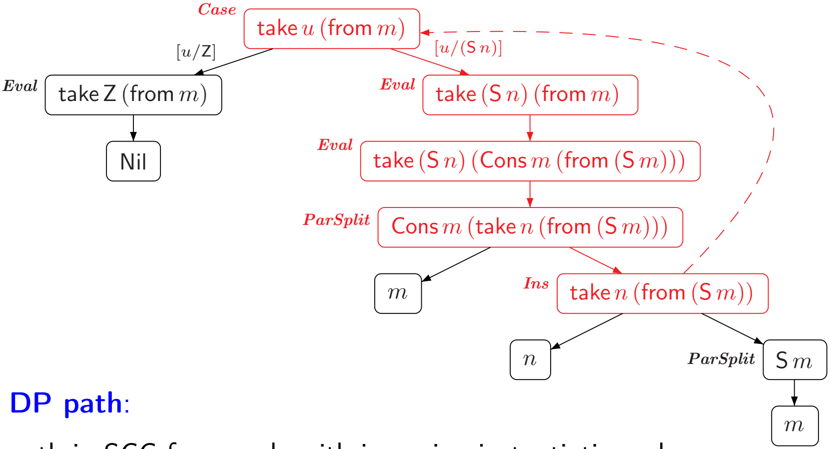
From Termination Graphs to DP Problems

- Every infinite path traverses an instantiation edge infinitely often



From Termination Graphs to DP Problems

- Every infinite path traverses an instantiation edge infinitely often

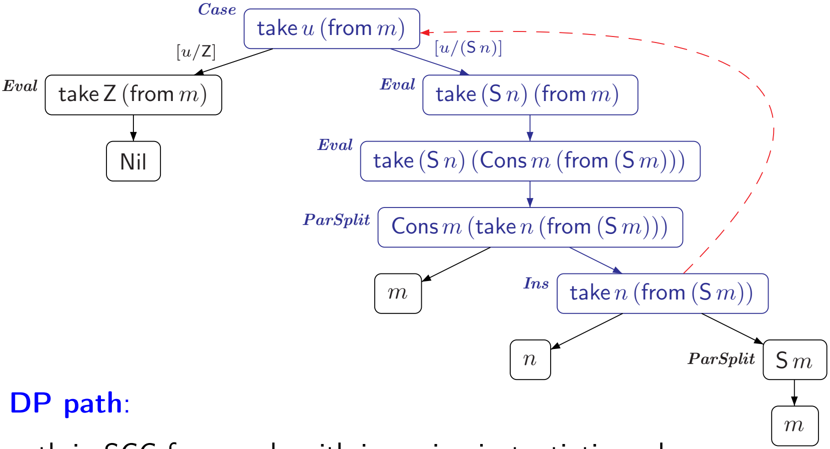


- **DP path**:

  path in SCC from node with incoming instantiation edge
  to node with outgoing instantiation edge

From Termination Graphs to DP Problems

- Every infinite path traverses a DP path infinitely often

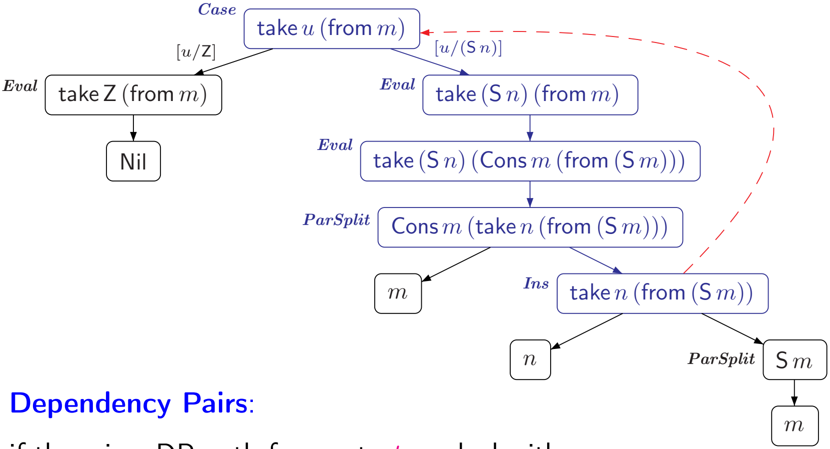


- **DP path**:

  path in SCC from node with incoming instantiation edge
  to node with outgoing instantiation edge

From Termination Graphs to DP Problems

- Every infinite path traverses a DP path infinitely often

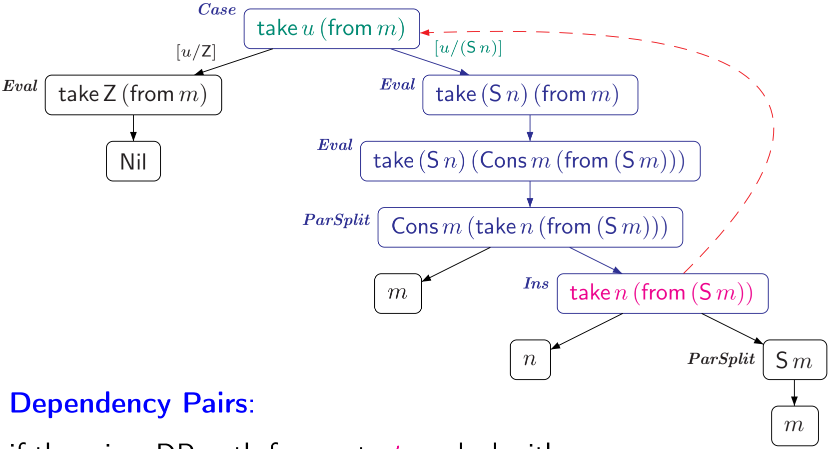


- **Dependency Pairs**:

  if there is a DP path from s to t marked with μ,
  then generate the dependency pair $s\,\mu \to t$

From Termination Graphs to DP Problems

- Every infinite path traverses a DP path infinitely often
  $\Rightarrow$ generate a dependency pair for every DP path

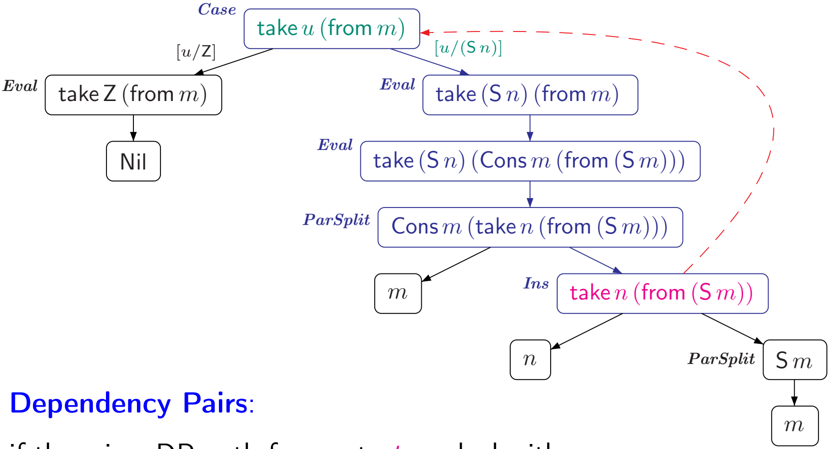


- **Dependency Pairs**:

  if there is a DP path from s to t marked with μ,
  then generate the dependency pair $s\ \mu \to t$

From Termination Graphs to DP Problems

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

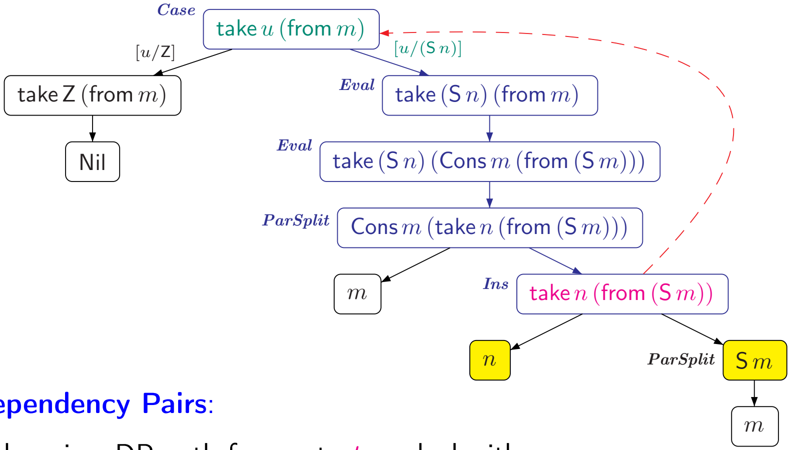


- **Dependency Pairs**:

  if there is a DP path from s to t marked with μ,
  then generate the dependency pair $s\,\mu \to t$

From Termination Graphs to DP Problems

- Dependency Pair $\mathcal{P}$: $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(n, \mathsf{from}(\mathsf{S}(m)))$
  Rules $\mathcal{R}$: $\emptyset$ (rules for terms in instance-edge matcher)

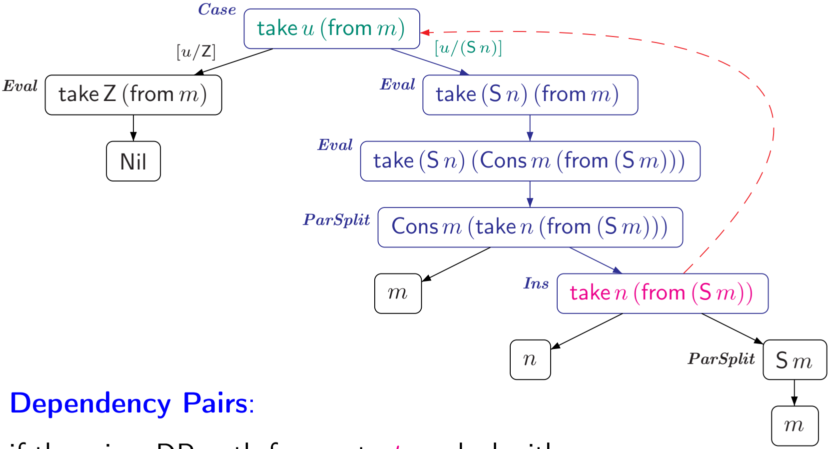


- **Dependency Pairs**:

  if there is a DP path from s to t marked with μ,
  then generate the dependency pair $s\,\mu \to t$

From Termination Graphs to DP Problems

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

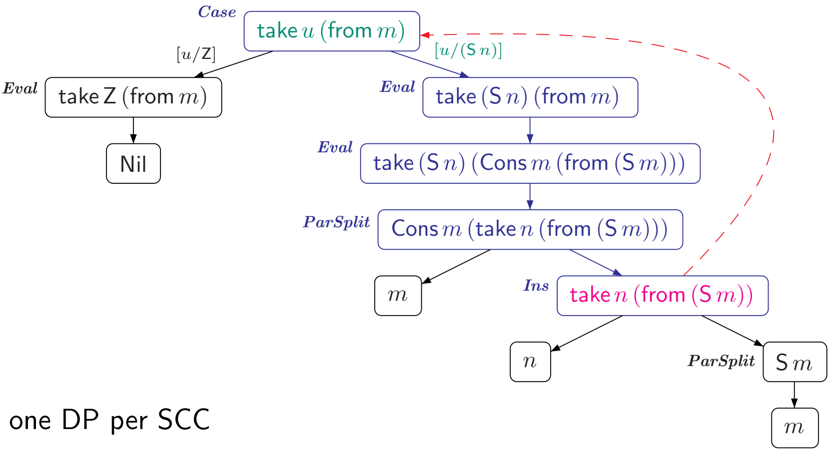


- **Dependency Pairs**:

  if there is a DP path from s to t marked with μ,
  then generate the dependency pair $s\,\mu \to t$

From Termination Graphs to DP Problems

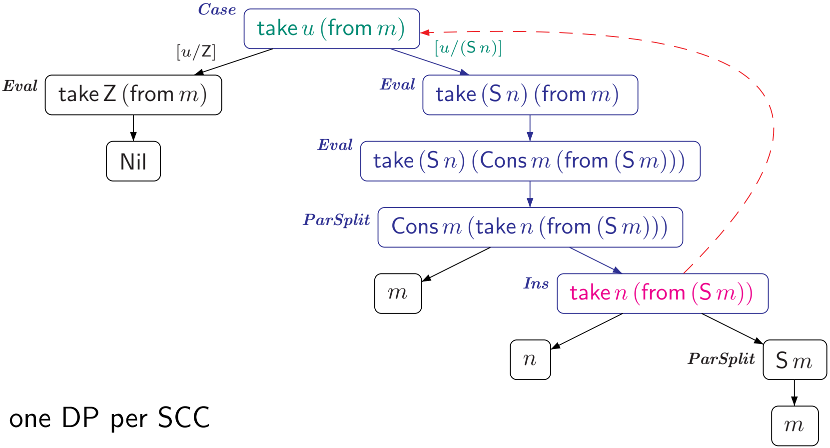
- Dependency Pair $\mathcal{P}$: $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(n, \mathsf{from}(\mathsf{S}(m)))$
  Rules $\mathcal{R}$: $\emptyset$



- 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 $\mathcal{P}$: $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(n, \mathsf{from}(\mathsf{S}(m)))$
  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

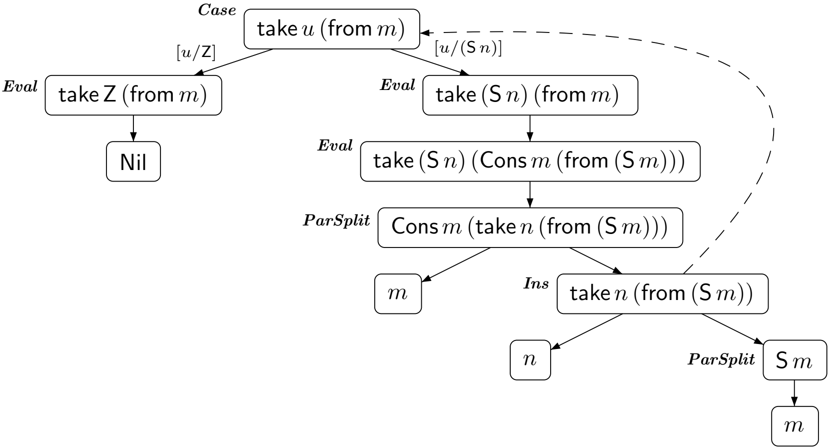
From Termination Graphs to DP Problems

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

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

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

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



From Termination Graphs to DP Problems

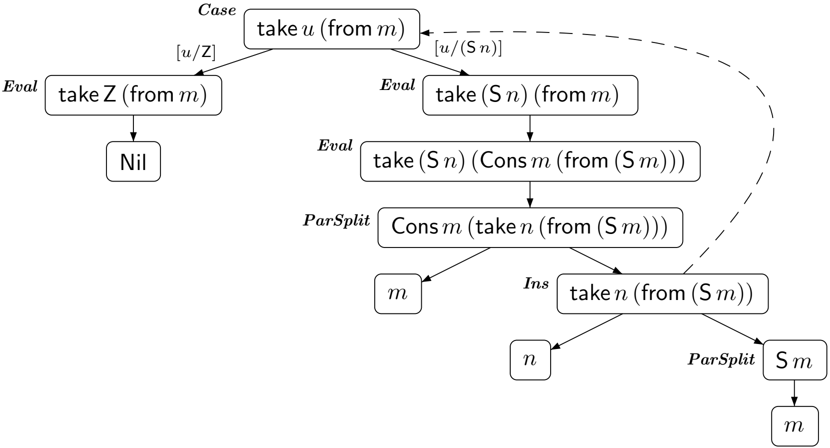
$\mathsf{from}\ x = \mathsf{Cons}\ x\ (\mathsf{from}\ (\mathsf{S}\ x))$

$\mathsf{p}\ (\mathsf{S}\ x) = x$

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

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

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



From Termination Graphs to DP Problems

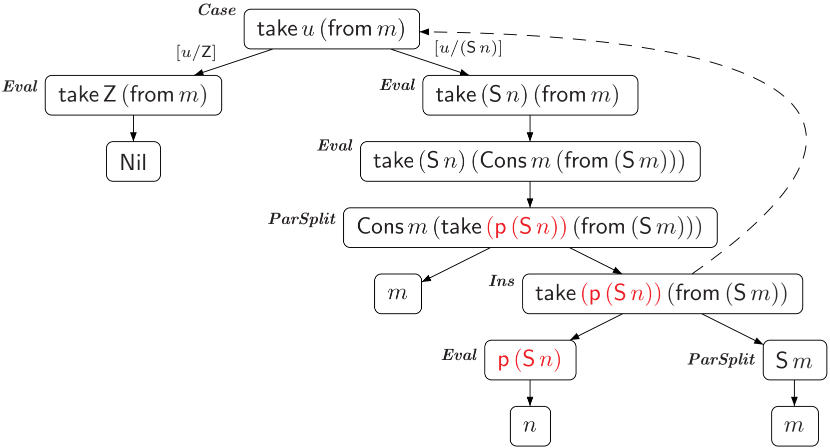
$\mathsf{from}\ x = \mathsf{Cons}\ x\ (\mathsf{from}\ (\mathsf{S}\ x))$

$\mathsf{p}\ (\mathsf{S}\ x) = x$

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

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

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



From Termination Graphs to DP Problems

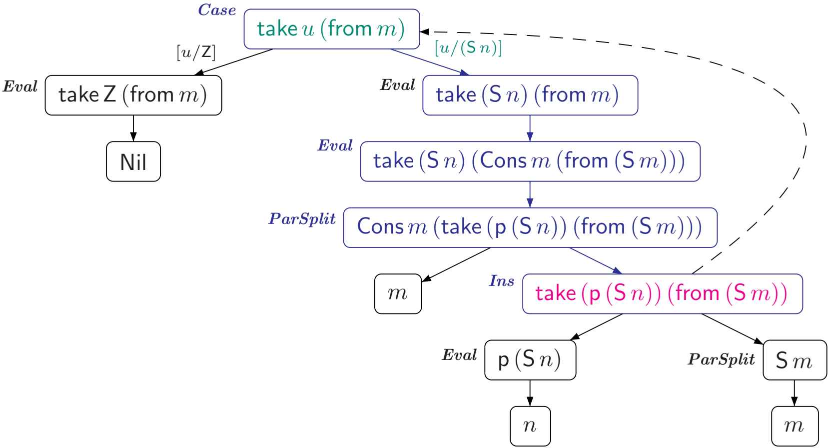
$\mathsf{from}\ x = \mathsf{Cons}\ x\ (\mathsf{from}\ (\mathsf{S}\ x))$

$\mathsf{p}\ (\mathsf{S}\ x) = x$

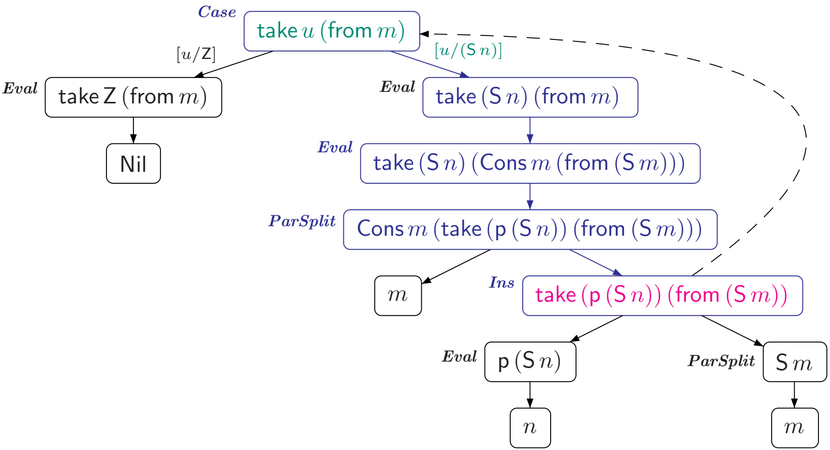
$\mathsf{take}\ \mathsf{Z}\ xs = \mathsf{Nil}$

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

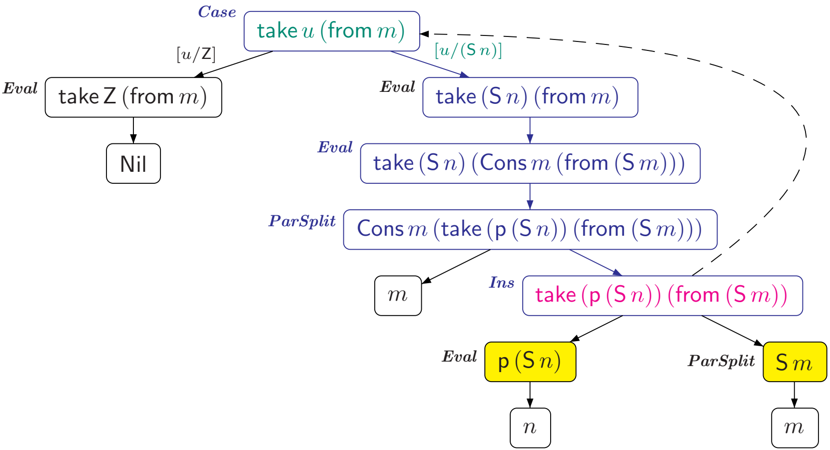
$\mathsf{take}\ (\mathsf{S}\ n)\ (\mathsf{Cons}\ x\ xs) = \mathsf{Cons}\ x\ (\mathsf{take}\ (\mathsf{p}\ (\mathsf{S}\ n))\ xs)$



- **Dependency Pair $\mathcal{P}$:**
  $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(\mathsf{p}(\mathsf{S}(n)), \mathsf{from}(\mathsf{S}(m)))$

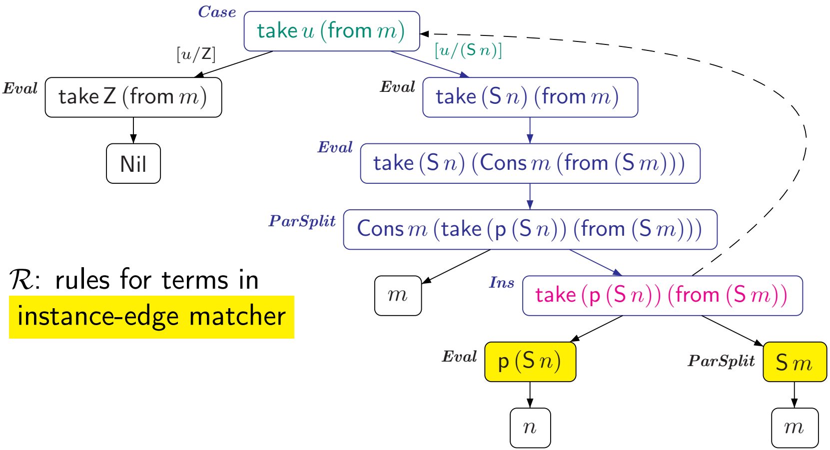


- **Dependency Pair $\mathcal{P}$:**
  $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(\mathsf{p}(\mathsf{S}(n)), \mathsf{from}(\mathsf{S}(m)))$



From Termination Graphs to DP Problems

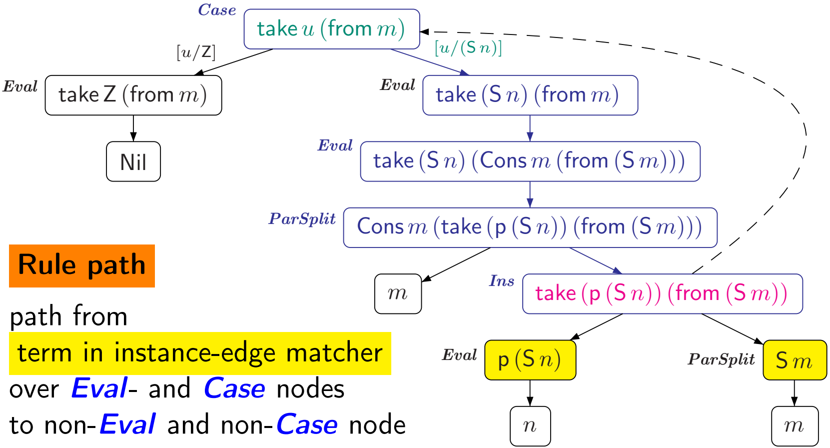
- **Dependency Pair $\mathcal{P}$:**
  $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(\mathsf{p}(\mathsf{S}(n)), \mathsf{from}(\mathsf{S}(m)))$



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

From Termination Graphs to DP Problems

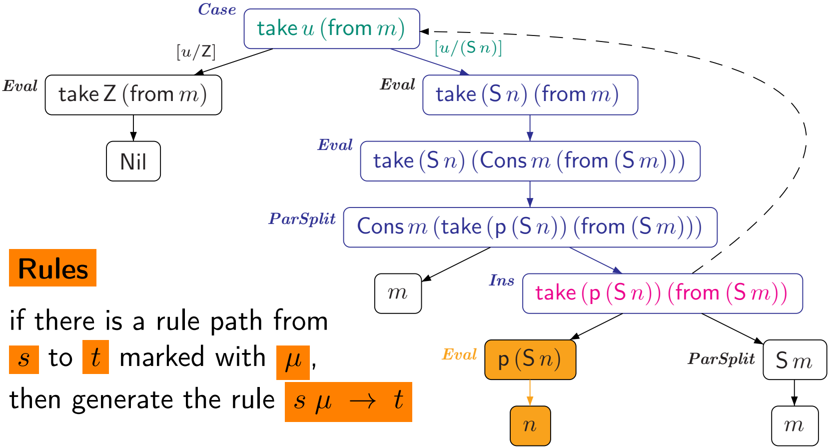
- **Dependency Pair $\mathcal{P}$:**
  $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(\mathsf{p}(\mathsf{S}(n)), \mathsf{from}(\mathsf{S}(m)))$



- **Rule path**

  path from
  term in instance-edge matcher
  over ***Eval***- and ***Case*** nodes
  to non-***Eval*** and non-***Case*** node

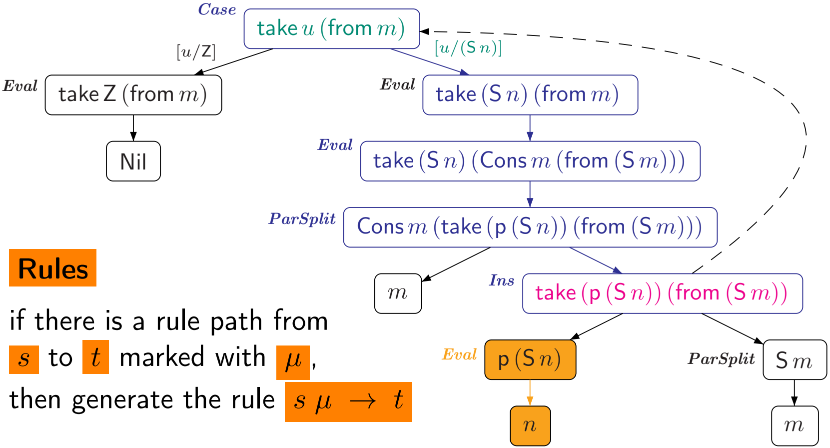
- **Dependency Pair $\mathcal{P}$:**
  $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(\mathsf{p}(\mathsf{S}(n)), \mathsf{from}(\mathsf{S}(m)))$



- **Rules**

  if there is a rule path from s to t marked with μ,
  then generate the rule $s\,\mu \to t$

- **Dependency Pair $\mathcal{P}$:**
  $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(\mathsf{p}(\mathsf{S}(n)), \mathsf{from}(\mathsf{S}(m)))$
- Rule $\mathcal{R}$: $\mathsf{p}(\mathsf{S}(n)) \ \to \ n$

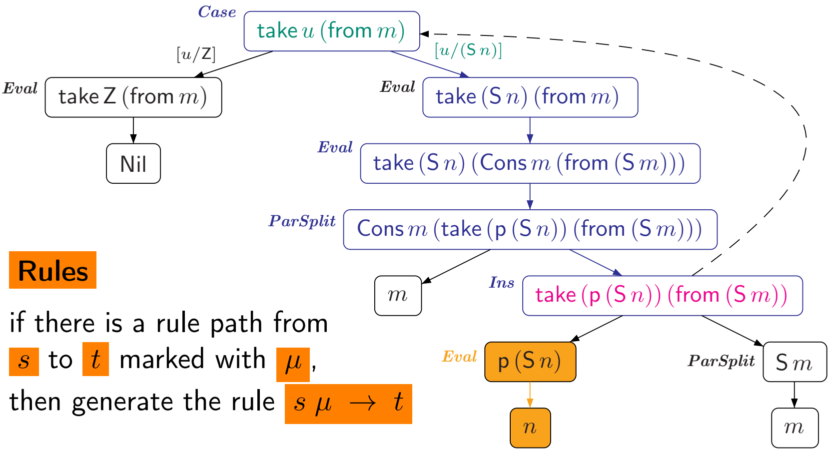


- **Rules**

  if there is a rule path from s to t marked with μ,
  then generate the rule $s\,\mu \ \to \ t$

From Termination Graphs to DP Problems

- **Dependency Pair $\mathcal{P}$:**
  $\mathsf{take}(\mathsf{S}(n), \mathsf{from}(m)) \to \mathsf{take}(\mathsf{p}(\mathsf{S}(n)), \mathsf{from}(\mathsf{S}(m)))$
- Rule $\mathcal{R}$: $\mathsf{p}(\mathsf{S}(n)) \;\to\; n$ termination easy to prove



- **Rules**

  if there is a rule path from s to t marked with μ,
  then generate the rule $s\,\mu \;\to\; t$

Experiments

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

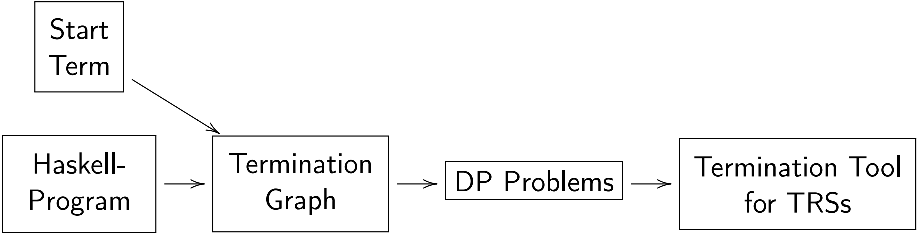
http://aprove.informatik.rwth-aachen.de/

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



- Language specifics are handled in transformation front-end

$\Rightarrow$ 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 → term rewrite system
- Term rewrite system → 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*] → **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
  [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/

Haskell Exercises I

Consider the following Haskell program.

```
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).

Haskell Exercises III

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?

Recap: from Haskell to Term Rewriting for Termination

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\rightarrow$ 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
  $\Rightarrow$ implies termination of program from initial states

Yesterday: Haskell, today: Java

Recap: from Haskell to Term Rewriting for Termination

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\rightarrow$ 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
  $\Rightarrow$ implies termination of program from initial states

Yesterday: Haskell, today: Java

Recap: from Haskell to Term Rewriting for Termination

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\rightarrow$ 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
  $\Rightarrow$ implies termination of program from initial states

Yesterday: Haskell, today: Java

Recap: from Haskell to Term Rewriting for Termination

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\rightarrow$ 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
  $\Rightarrow$ implies termination of program from initial states

Yesterday: Haskell, today: Java

Recap: from Haskell to Term Rewriting for Termination

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\rightarrow$ 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
  $\Rightarrow$ implies termination of program from initial states

Yesterday: Haskell, today: Java

Recap: from Haskell to Term Rewriting for Termination

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\rightarrow$ 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
  $\Rightarrow$ implies termination of program from initial states

Yesterday: Haskell, today: Java

Recap: from Haskell to Term Rewriting for Termination

Recipe for proving program termination by reusing TRS termination provers

- Decide on suitable symbolic representation of abstract program states (**abstract domain**)
  $\rightarrow$ 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
  $\Rightarrow$ implies termination of program from initial states

Yesterday: Haskell, today: Java

Beyond Classic TRSs for Programs

Rewrite rules for Haskell programs in **standard** term rewriting
$\rightarrow$ 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
$\rightarrow$ 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
$\rightarrow$ 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
$\rightarrow$ 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
$\rightarrow$ 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
$\rightarrow$ 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

$\Rightarrow$ 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

$\Rightarrow$ 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

$\Rightarrow$ 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

$\Rightarrow$ 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

$\Rightarrow$ 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

$\Rightarrow$ 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

$\Rightarrow$ 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)

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

Possible rewrite sequence:

Constrained Rewriting by Example

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

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

$$\ell_0(2, 7)$$

Constrained Rewriting by Example

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

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

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

Constrained Rewriting by Example

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

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

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

Constrained Rewriting by Example

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

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

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

Constrained Rewriting by Example

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

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

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

Constrained Rewriting by Example

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

Example (Constrained Rewrite System)

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

Possible rewrite sequence:

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

Here 7, 8, $\ldots$ 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)

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

Does this program terminate?

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

Example (Imperative Program)

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

Does this program terminate?

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

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

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

Example (Imperative Program)

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

Does this program terminate?

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

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

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

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

Example (Imperative Program)

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

Does this program terminate?

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

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

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

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

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

Example (Imperative Program)

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

Does this program terminate?

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

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

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

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

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

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

Example (Imperative Program)

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

Does this program terminate?

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

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

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

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

$\Rightarrow$ Find **invariant** $x \geq 0$ at ℓ_1, ℓ_2 (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
- . . .

Java Example

```
public class MyInt {

  // only wrap a primitive int
  private int val;

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

Does `count` terminate for all inputs? Why (not)?

(You may assume that `num` and `limit` are not references to the same object.)

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 ($\rightarrow$ 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 ($\rightarrow$ 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 ($\rightarrow$ web interface)

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

Java: Source Code vs Bytecode

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

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

Java: Source Code vs Bytecode

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

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

Java: Source Code vs Bytecode

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

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

```
00: aload_0
01: ifnull 8
04: aload_1
05: ifnonnull 9
08: return
09: aload_0
10: astore_2
11: aload_0
12: getfield val
15: aload_1
16: getfield val
19: if_icmpge 35
22: aload_2
23: aload_2
24: getfield val
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

Java: Source Code vs Bytecode

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

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

Here: **Java source code**

Ingredients for the Abstract Domain

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

Heap predicates:

- Two references may be equal: $o_1 =^? o_2$
- Two references may share: $o_1 \searrow\!\!\swarrow o_2$
- Reference may have cycles: $o_1\,!$

03	$\mathtt{num} : o_1, \mathtt{limit} : o_2$
$o_1 : \mathtt{MyInt}(?)$ $o_2 : \mathtt{MyInt}(\mathtt{val} = i_1)$ $i_1 : [4, 80]$	

Ingredients for the Abstract Domain

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

Heap predicates:

- Two references may be equal: $o_1 =^? o_2$
- Two references may share: $o_1 \searrow\!\!\swarrow o_2$
- Reference may have cycles: o_1 !

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

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. $\mathtt{x} \in [4,\ 7]$ or $\mathtt{y} \in [0,\ \infty)$
   - heap memory with objects, **no sharing** unless stated otherwise
   - `MyInt`(?): maybe `null`, maybe a `MyInt` object

   Heap predicates:

   - Two references may be equal: $o_1 =^? o_2$
   - Two references may share: $o_1 \searrow\!\!\swarrow o_2$
   - Reference may have cycles: $o_1\,!$

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

Building the Termination Graph

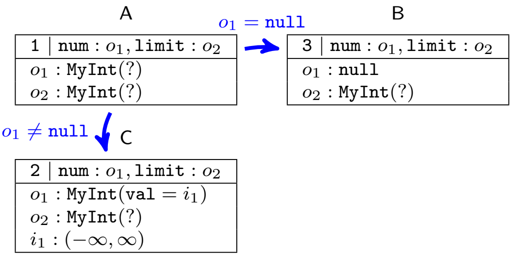
```
   public class MyInt {
     private int val;
     static void count(MyInt num,
          MyInt limit) {
1:      if (num == null
2:          || limit == null)
3:        return;
4:      MyInt copy = num;
5:      while (num.val < limit.val)
6:        copy.val++;
7: } }
```

A

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

Building the Termination Graph

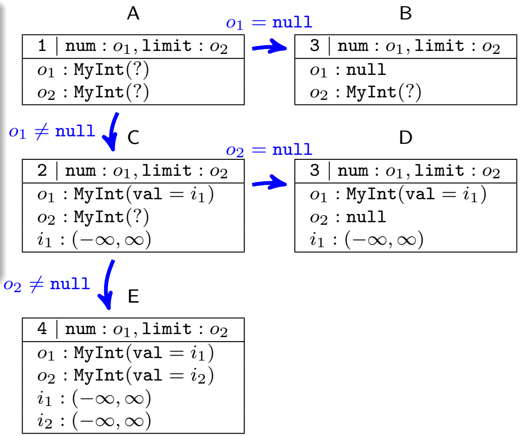
```
public class MyInt {
  private int val;
  static void count(MyInt num,
         MyInt limit) {
    if (num == null
        || limit == null)
      return;
    MyInt copy = num;
    while (num.val < limit.val)
      copy.val++;
} }
```



$$X \xrightarrow{cond} Y$$

means: refine X with $cond$, then evaluate to Y; here combined for brevity (narrowing; Haskell: $\boldsymbol{Case} + \boldsymbol{Eval}$)

```
  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++;
} }
```

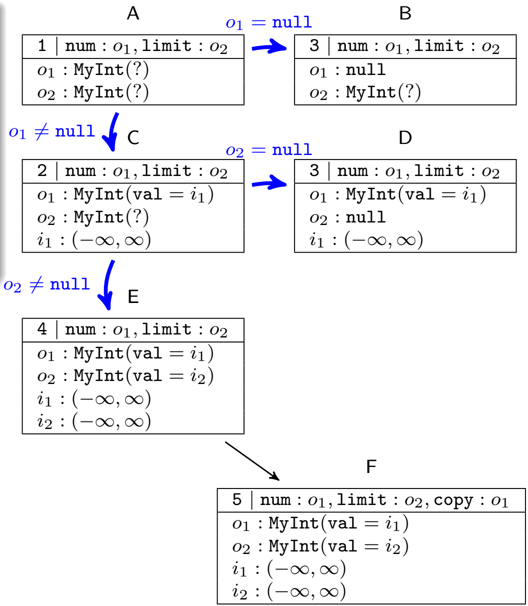


$\mathsf{X} \xrightarrow{cond} \mathsf{Y}$

means: refine X with $cond$, then evaluate to Y; here combined for brevity (narrowing; Haskell: $\boldsymbol{Case} + \boldsymbol{Eval}$)

Building the Termination Graph

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

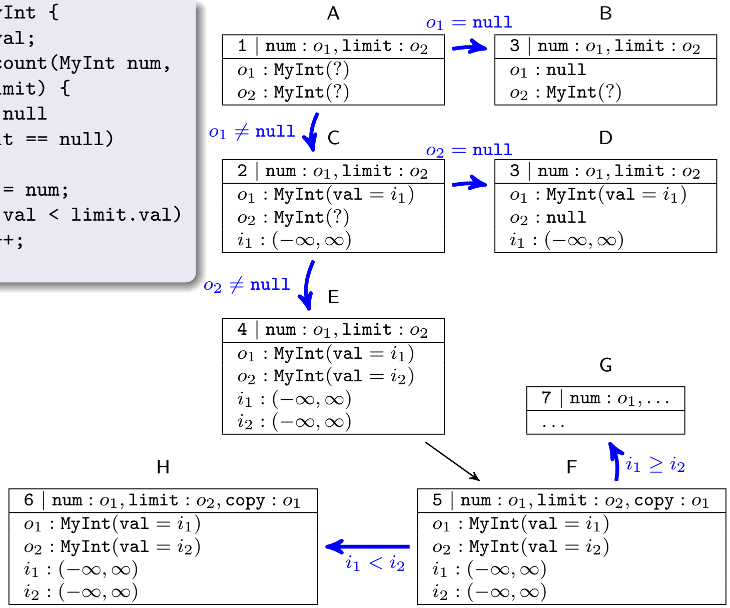


$X \longrightarrow Y$

means: evaluate X to Y
(Haskell: ***Eval***)

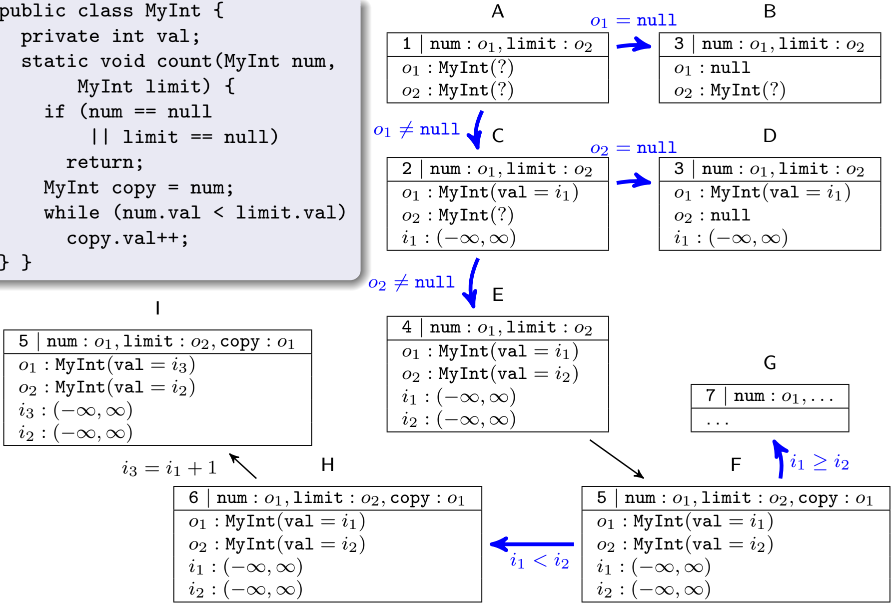
Building the Termination Graph

```
   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++;
   } }
```

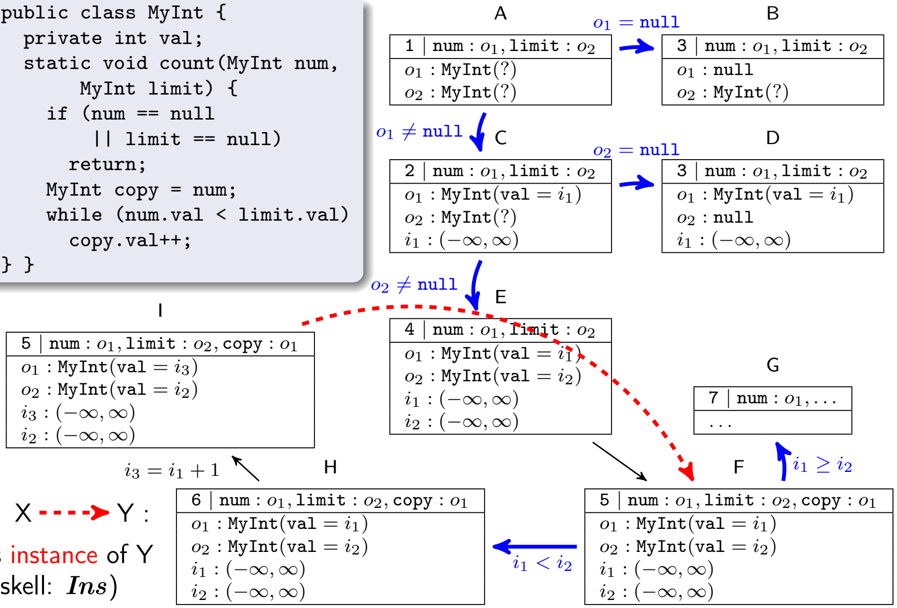


Building the Termination Graph

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



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



X $\dashrightarrow$ Y :

X is instance of Y

(Haskell: *Ins*)

From Java to Termination Graphs

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

From Java to Termination Graphs

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

Transformation of Objects to Terms

Q

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

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

- term for o_1: o_1
- term for o_2: MyInt(i_2)
- term for o_3: null
- term for o_4: x (new variable)
- term for i_1: i_1 with **side constraint** $i_1 \geq 7$
  (invariant $i_1 \geq 7$ to be added to constrained rewrite rules for state Q)

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

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

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

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

Transformation of Objects to Terms

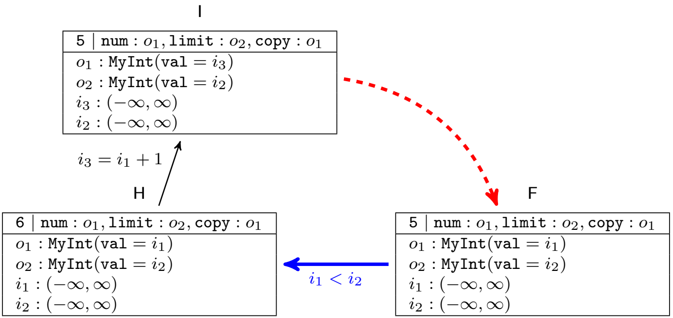
```
public class A {
  int a;
}

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`: $\mathsf{jlO}(\mathsf{A}(\mathsf{eoc}, 1))$
  $\to$ eoc for end of class
- term for `y`: $\mathsf{jlO}(\mathsf{A}(\mathsf{B}(\mathsf{eoc}, 3), 2))$
- every class extends `Object`!
  ($\to$ jlO $\equiv$ `java.lang.Object`)

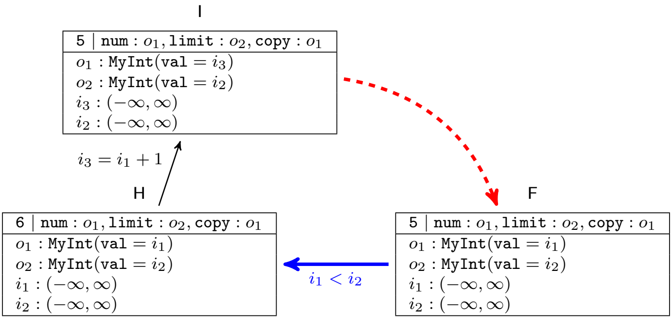


- State F: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$

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

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

From the Termination Graph to Terms and Rules



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

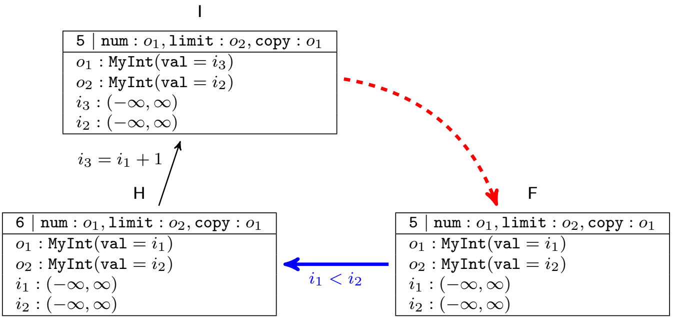
  State H: $\ell_{\mathsf{H}}(\ \mathsf{jIO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jIO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$

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

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

- Termination easy to show (intuitively: $i_2 - i_1$ decreases against bound 0)

From the Termination Graph to Terms and Rules



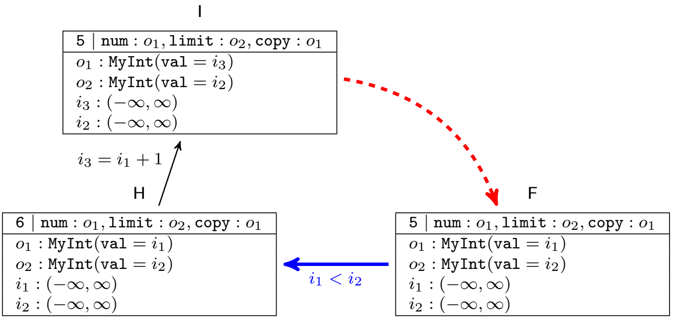
- State F: $\ell_{\mathsf{F}}($ jlO(MyInt(eoc, i_1)), jlO(MyInt(eoc, i_2)) $)$

  $\rightarrow$

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

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



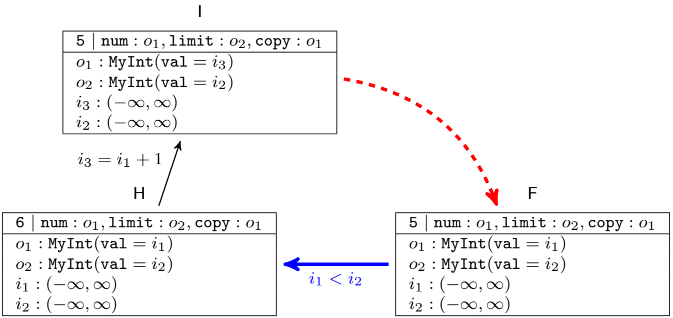
- State F: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$

  $\rightarrow$

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

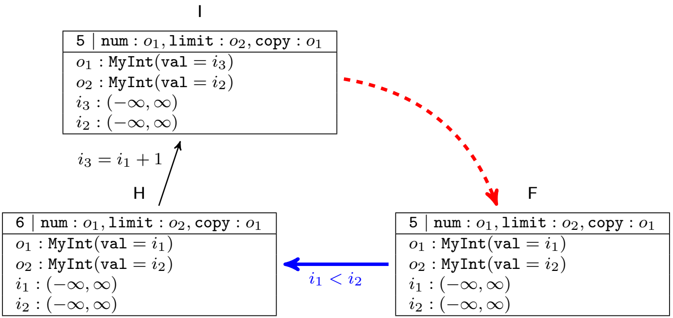
  State I: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1 + 1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$
- Termination easy to show (intuitively: $i_2 - i_1$ decreases against bound 0)

From the Termination Graph to Terms and Rules



- State F: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$
  $\rightarrow$
  State H: $\ell_{\mathsf{H}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$ $\quad [i_1 < i_2]$
- State H: $\ell_{\mathsf{H}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$
  $\rightarrow$
  State I: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1 + 1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ \ )$
- Termination easy to show (intuitively: $i_2 - i_1$ decreases against bound 0)

From the Termination Graph to Terms and Rules



- State F: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$
  $\rightarrow$
  State H: $\ell_{\mathsf{H}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$ $[i_1 < i_2]$
- State H: $\ell_{\mathsf{H}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$
  $\rightarrow$
  State I: $\ell_{\mathsf{F}}(\ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_1 + 1)),\ \ \mathsf{jlO}(\mathsf{MyInt}(\mathsf{eoc}, i_2))\ )$
- Termination easy to show (intuitively: $i_2 - i_1$ decreases against bound 0)

Extensions

- **modular** termination proofs and **recursion** [Brockschmidt et al, *RTA '11*]
- proving **reachability** and **non-termination** (uses only 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

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

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.

Java Exercises II

Question 5

```
public class List {
  private List next;

  public static int length(List xs) {
    int res = 0;                            // 1
    while (xs != null) {                    // 2
      xs = xs.next;                         // 3
      res++;                                // 4
    }
    return res;                             // 5
  }
}
```

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?

References I

T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *RTA '11*, pages 155–170, 2011.

M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *CAV '12*, pages 105–122, 2012a.

M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. In *FoVeOOS '11*, pages 123–141, 2012b.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252, 1977.

S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *CADE '09*, pages 277–293, 2009.

F. Frohn and J. Giesl. Complexity analysis for Java with AProVE. In *iFM '17*, 2017. To appear.

C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *RTA '09*, pages 32–47, 2009.

J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37 (3):155–203, 2006.

References III

J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):1–39, 2011. See also http://aprove.informatik.rwth-aachen.de/eval/Haskell/.

J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting: A general methodology for analyzing logic programs. In *PPDP '12*, pages 1–12, 2012.

J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

C. Kop and N. Nishida. Term rewriting with logical constraints. In *FroCoS '13*, pages 343–358, 2013.

D. S. Lankford. Canonical algebraic simplification in computational logic. Technical Report ATP-25, University of Texas, 1975.

C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL '01*, pages 81–92, 2001.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4): 184–195, 1960.

C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *RTA '10*, pages 259–276, 2010.

References V

P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1):1–52, 2009.

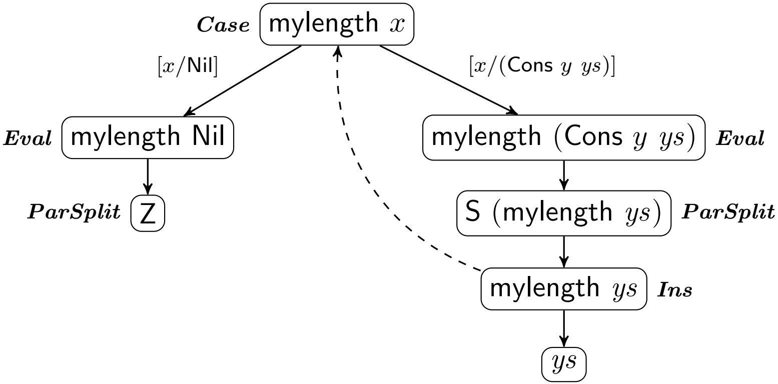
R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.

F. van Raamsdonk. Translating logic programs into conditional rewriting systems. In *ICLP '97*, pages 168–182, 1997.

Solution for Question 1 (page 1)

(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) $\mathcal{P}$: mylength(Cons(y, ys)) $\to$ mylength(ys)
$\mathcal{R}$: $\emptyset$

Solution for Question 1 (page 2)

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

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

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.

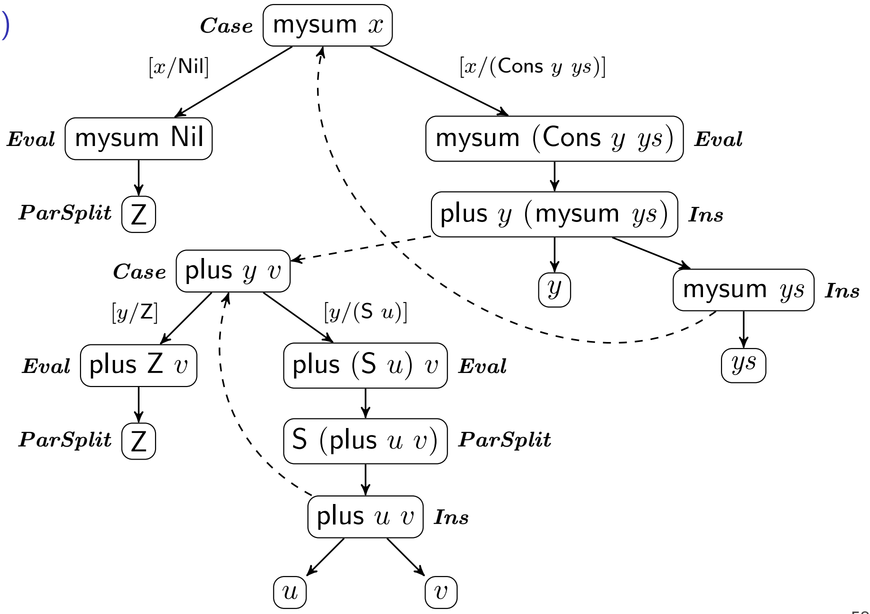
Further details: [Giesl et al, *TOPLAS '11*]

Solution for Question 2 (page 1)

(a) Yes. The reason is that both `mysum` and `plus` are terminating since their recursive calls are on arguments that get smaller and smaller.

Solution for Question 2 (page 2)

(b)



Solution for Question 2 (page 3)

(c) $\mathcal{P}$: $\mathsf{mysum}(\mathsf{Cons}(y, ys)) \to \mathsf{mysum}(ys)$
$\mathsf{plus}(\mathsf{S}(u), v) \to \mathsf{plus}(u, v)$
$\mathcal{R}$: $\emptyset$

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

$$[\mathsf{mysum}](x_1) = x_1 \qquad [\mathsf{Cons}](x_1, x_2) = x_1 + x_2 + 1$$
$$[\mathsf{plus}](x_1, x_2) = x_1 \qquad [\mathsf{S}](x_1) = x_1 + 1$$

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):

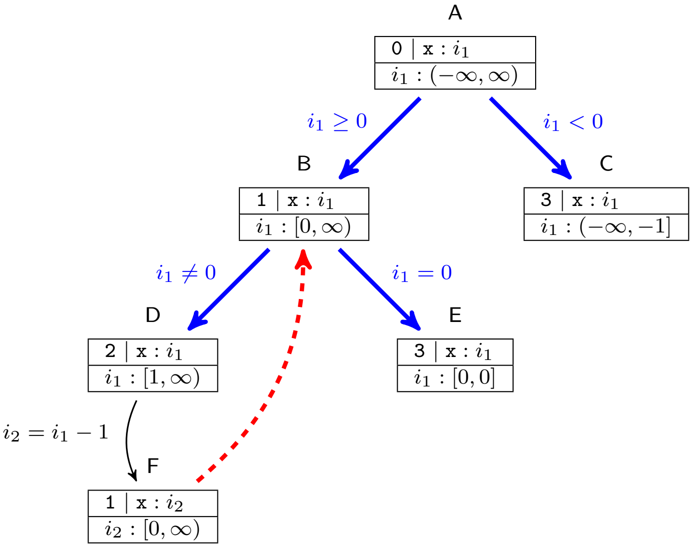
- support of user-defined data structures by representation as terms
- as termination tools for TRSs improve over time thanks to on-going development, so does this overall approach

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:



Solution for Question 4 (page 2)

If we translate the whole termination graph, we get the following constrained rewrite rules:

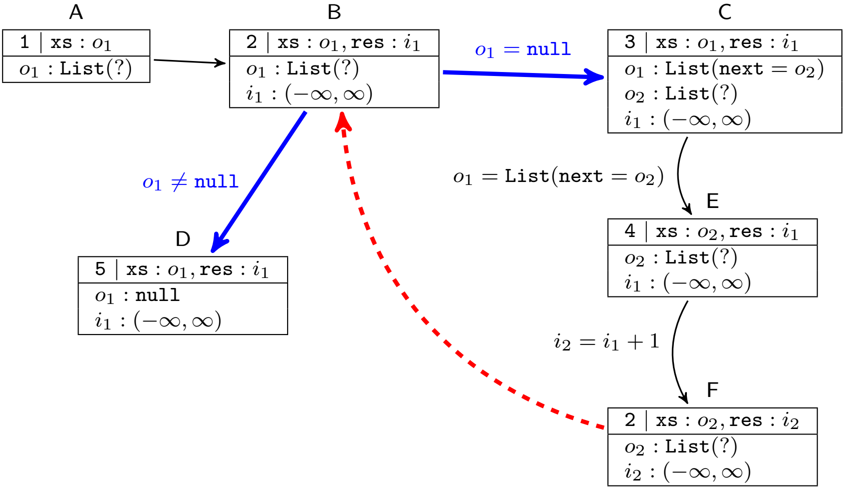
$$\begin{array}{lcll}
\ell_{\mathsf{A}}(i_1) & \to & \ell_{\mathsf{B}}(i_1) & [i_1 \geq 0] \\
\ell_{\mathsf{A}}(i_1) & \to & \ell_{\mathsf{C}}(i_1) & [i_1 < 0] \\
\ell_{\mathsf{B}}(i_1) & \to & \ell_{\mathsf{D}}(i_1) & [i_1 \neq 0 \wedge i_1 \geq 0] \\
\ell_{\mathsf{D}}(i_1) & \to & \ell_{\mathsf{B}}(i_1 - 1) & [i_1 \geq 1] \\
\ell_{\mathsf{B}}(i_1) & \to & \ell_{\mathsf{E}}(i_1) & [i_1 = 0]
\end{array}$$

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 ℓ_3, we now have the two different end-of-program symbols ℓ_{C} and ℓ_{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 $i_1 \geq 0$).

Solution for Question 5 (page 1)

We get the following termination graph for the program:



Solution for Question 5 (page 2)

The SCC of the graph gives rise to the following constrained rewrite rules:

$$\begin{array}{rcl} \ell_{\mathsf{B}}(\mathsf{jlO}(\mathsf{List}(\mathsf{eoc}, o_2)), i_1) & \to & \ell_{\mathsf{C}}(\mathsf{jlO}(\mathsf{List}(\mathsf{eoc}, o_2)), i_1) \\ \ell_{\mathsf{C}}(\mathsf{jlO}(\mathsf{List}(\mathsf{eoc}, o_2)), i_1) & \to & \ell_{\mathsf{E}}(o_2, i_1) \\ \ell_{\mathsf{E}}(o_2, i_1) & \to & \ell_{\mathsf{B}}(o_2, i_1 + 1) \end{array}$$

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:

$$\begin{array}{rclcrcl} [\ell_{\mathsf{B}}](o, i) & = & o + 1 & \qquad & [\ell_{\mathsf{C}}](o, i) & = & o \\ [\ell_{\mathsf{E}}](o, i) & = & o + 2 & & [\mathsf{List}](c, o) & = & o + 3 \\ [\mathsf{jlO}](c) & = & c & & [\mathsf{eoc}] & = & 0 \end{array}$$

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

$$\ell(\mathsf{jlO}(\mathsf{List}(\mathsf{eoc}, o_2)), i_1) \quad \to \quad \ell(o_2, i_1 + 1)$$

Details: [Giesl et al, *JAR '17*]