Automated Termination Analysis

Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

VTSA '12, Saarbrücken, Germany

Overview

I. Termination of Term Rewriting

1. Termination of Term Rewrite Systems
2. Non-Termination of Term Rewrite Systems
3. Complexity of Term Rewrite Systems
4. Termination of Integer Term Rewrite Systems

II. Termination of Programs

1. Termination of Functional Programs (Haskell) (ACM TOPLAS '11)
2. Termination of Logic Programs (Prolog)
3. Termination of Imperative Programs (Java)

Automated Termination Tools for TRSs

- AProVE *(Aachen)*
- CARIBOO *(Nancy)*
- CiME *(Orsay)*
- Jambox *(Amsterdam)*
- Matchbox *(Leipzig)*
- MU-TERM *(Valencia)*
- MultumNonMulta *(Kassel)*
- TEPARLA *(Eindhoven)*
- Termptation *(Barcelona)*
- TORPA *(Eindhoven)*
- TPA *(Eindhoven)*
- TTT *(Innsbruck)*
- VMTL *(Vienna)*

- Annual *International Competition of Termination Tools*
- well-developed field
- active research
- powerful techniques & tools
- **But:** What about application in practice?
- **Goal:** TRS-techniques for programming languages

Termination of Functional Programs

- first-order languages with strict evaluation strategy
  (Walther, 94), (Giesl, 95), (Lee, Jones, Ben-Amram, 01)
- ensuring termination (e.g., by typing)
  (Telford & Turner, 00), (Xi, 02), (Abel, 04), (Barthe et al, 04) etc.
- outermost termination of untyped first-order rewriting
  (Fissore, Gnaedig, Kirchner, 02)
- automated technique for small HASKELL-like language
  (Panitz & Schmidt-Schauss, 97)

- do not work on full existing languages
- no use of TRS-techniques (stand-alone methods)

Termination of Functional Programs

- first-order languages with strict evaluation strategy
  (Walther, 94), (Giesl, 95), (Lee, Jones, Ben-Amram, 01)
- ensuring termination (e.g., by typing)
  (Telford & Turner, 00), (Xi, 02), (Abel, 04), (Barthe et al, 04) etc.
- outermost termination of untyped first-order rewriting
  (Fissore, Gnaedig, Kirchner, 02)
- automated technique for small HASKELL-like language
  (Panitz & Schmidt-Schauss, 97)

- **new approach to use TRS-techniques for termination of** HASKELL
- based on *(Panitz & Schmidt-Schauss, 97)*, but:
  - works on full HASKELL-language
  - allows to integrate modern TRS-techniques and TRS-tools

HASKELL

- one of the most popular functional languages
- using TRS-techniques for HASKELL is challenging:
  - HASKELL has a lazy evaluation strategy.
    For TRSs, one proves termination of *all* reductions.
  - HASKELL's equations are handled from top to bottom.
    For TRSs, *any* rule may be used for rewriting.
  - HASKELL has polymorphic types.
    TRSs are *untyped*.
  - In HASKELL-programs, often only some functions terminate.
    TRS-methods try to prove termination of *all* terms.
  - HASKELL is a higher-order language.
    Most automatic TRS-methods only handle *first-order* rewriting.

Syntax of HASKELL

Data Structures

- data Nats = Z | S Nats

  type constructor: Nats of arity 0

  data constructors: Z :: Nats

  S :: Nats $\to$ Nats

- 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 Nats = Z | S Nats

  type constructor: Nats of arity 0

  data constructors: Z :: Nats

  S :: Nats $\to$ Nats

- 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 Nats, $a \to (\text{List } a)$, ...

S Z	has type	Nats
Cons x Nil	has type	List a

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)

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

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

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

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

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

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

Syntax of HASKELL

Extension of our approach for

- type classes
- built-in data structures

All other HASKELL-constructs: eliminated by automatic transformation

- Lambda Abstractions

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

Syntax of HASKELL

Extension of our approach for

- type classes
- built-in data structures

All other HASKELL-constructs: eliminated by automatic transformation

- Lambda Abstractions

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

- Conditions
- Local Declarations
- ...

Semantics and Termination 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 $\to_{\mathsf{H}}$**

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

evaluation position

Semantics and Termination 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 and Termination 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** $\to_{\mathsf{H}}$

$$\begin{array}{ll} & \mathsf{take}\ (\mathsf{S}\ \mathsf{Z})\ (\mathsf{from}\ m) \\ \to_{\mathsf{H}} & \mathsf{take}\ (\mathsf{S}\ \mathsf{Z})\ (\mathsf{Cons}\ m\ (\mathsf{from}\ (\mathsf{S}\ m))) \\ \to_{\mathsf{H}} & \mathsf{Cons}\ m\ (\mathsf{take}\ \mathsf{Z}\ (\mathsf{from}\ (\mathsf{S}\ m))) \\ \to_{\mathsf{H}} & \mathsf{Cons}\ m\ \mathsf{Nil} \end{array}$$

evaluation position

Semantics and Termination 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** $\to_{\mathsf{H}}$
- **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, $n < \text{arity}(f)$, 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.
- "from" not H-terminating ("from Z" has infinite evaluation) "take u (from m)" is H-terminating

Proving Termination of HASKELL

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

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

- **Goal:** Prove termination of *start term* "$\mathsf{take}\ u\ (\mathsf{from}\ m)$"
- **Naive approach:**
  - take defining equations of take and from as TRS
  - fails, since from is not terminating
  - disregards HASKELL's lazy evaluation strategy
- **Our approach:**
  - evaluate start term a few steps $\Rightarrow$ termination graph
  - do not transform HASKELL into TRS directly,
    but transform termination graph into TRS

From HASKELL to Termination Graphs

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

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

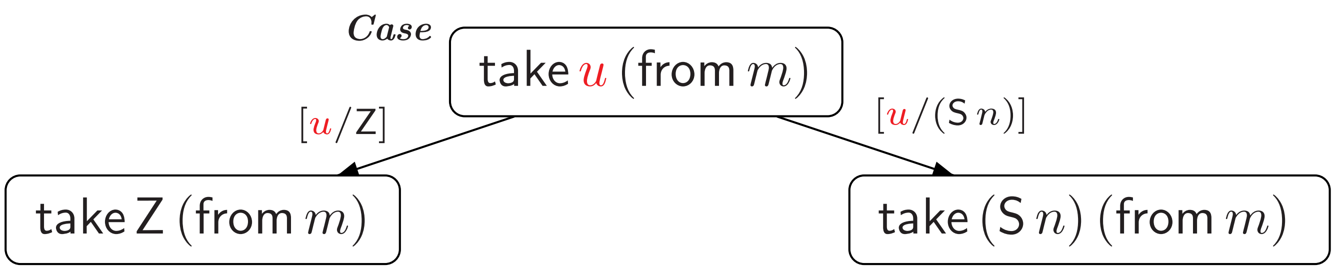
$\mathsf{take}\; u\; (\mathsf{from}\; m)$

- begin with node marked with start term
- 5 expansion rules to add children to leaves
- expansion rules try to *evaluate* terms

From HASKELL to Termination Graphs

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

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

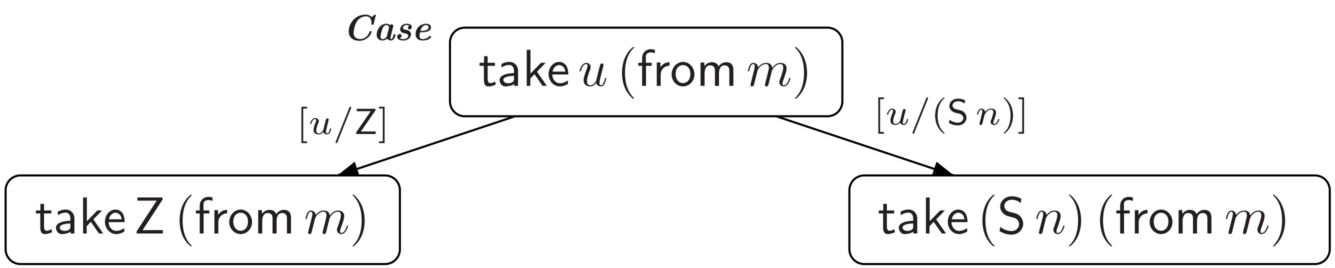


- ***Case*** rule:
  - evaluation has to continue with variable u
  - instantiate u by all possible constructor terms of correct type

From HASKELL to Termination Graphs

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

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



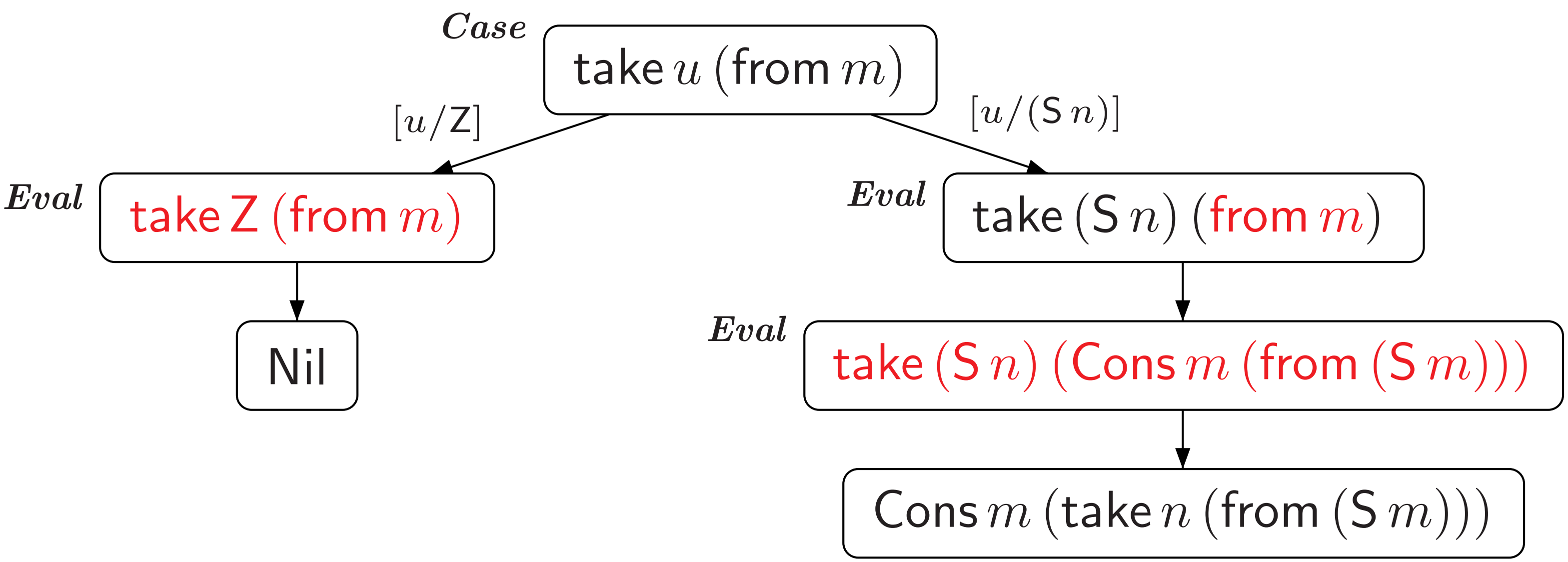
- **Main Property of Termination Graphs:**

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

From HASKELL to Termination Graphs

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

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



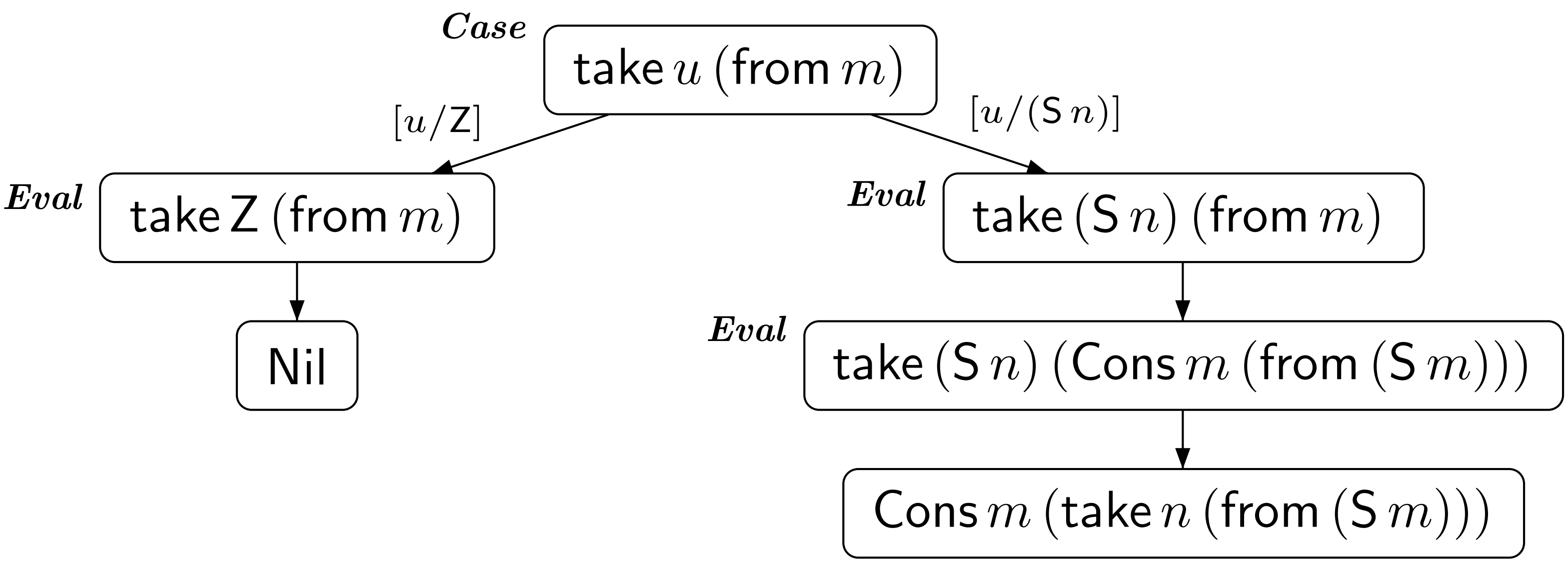
- ***Eval*** rule:

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

From HASKELL to Termination Graphs

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

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

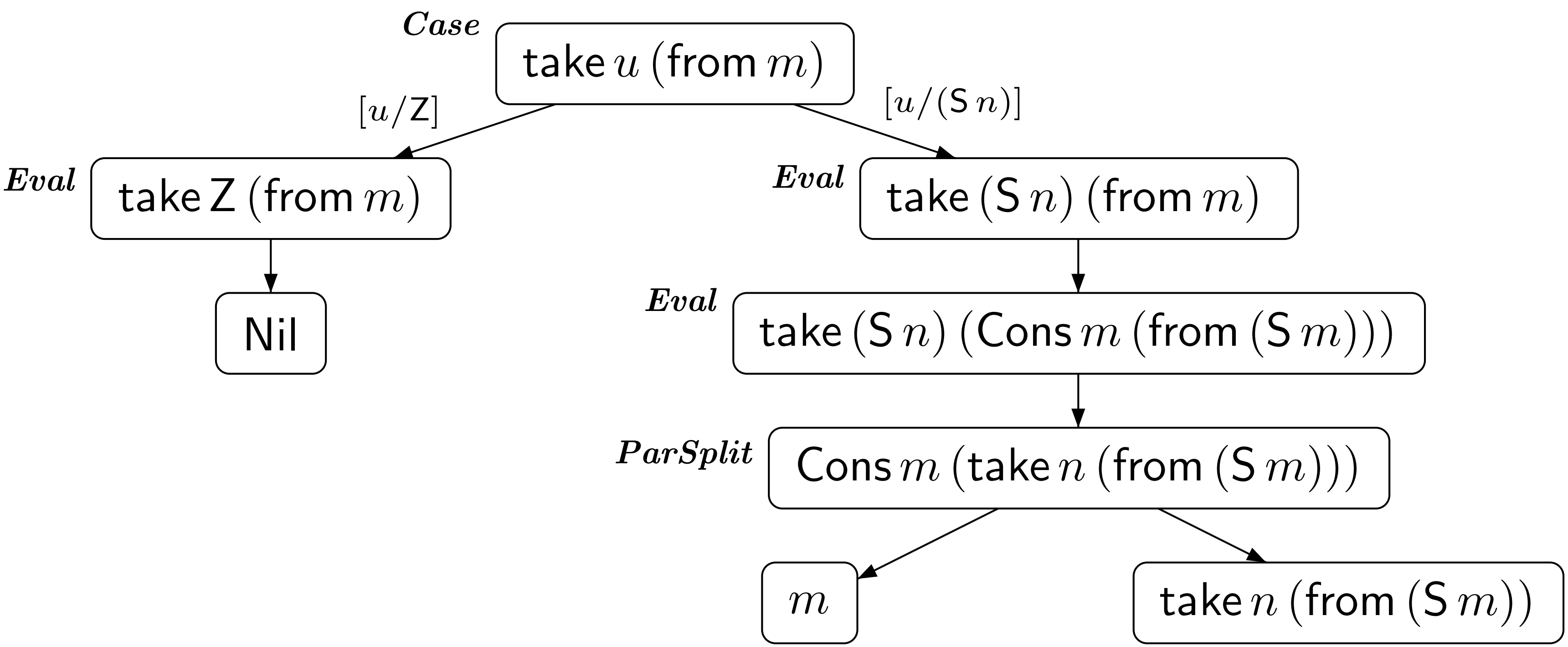


Case and ***Eval*** rule perform *narrowing*
w.r.t. HASKELL's evaluation strategy and types

From HASKELL to Termination Graphs

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

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



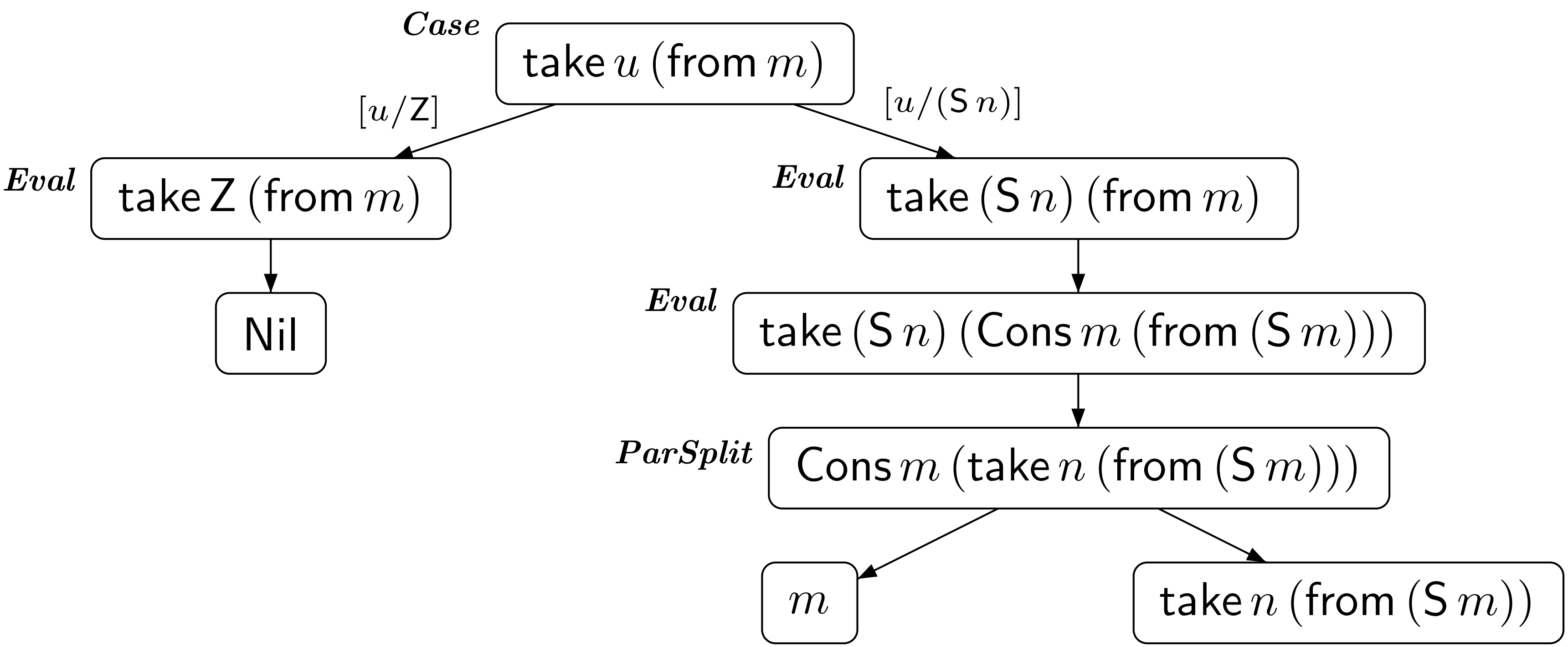
- ***ParSplit*** rule:

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

From HASKELL to Termination Graphs

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

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

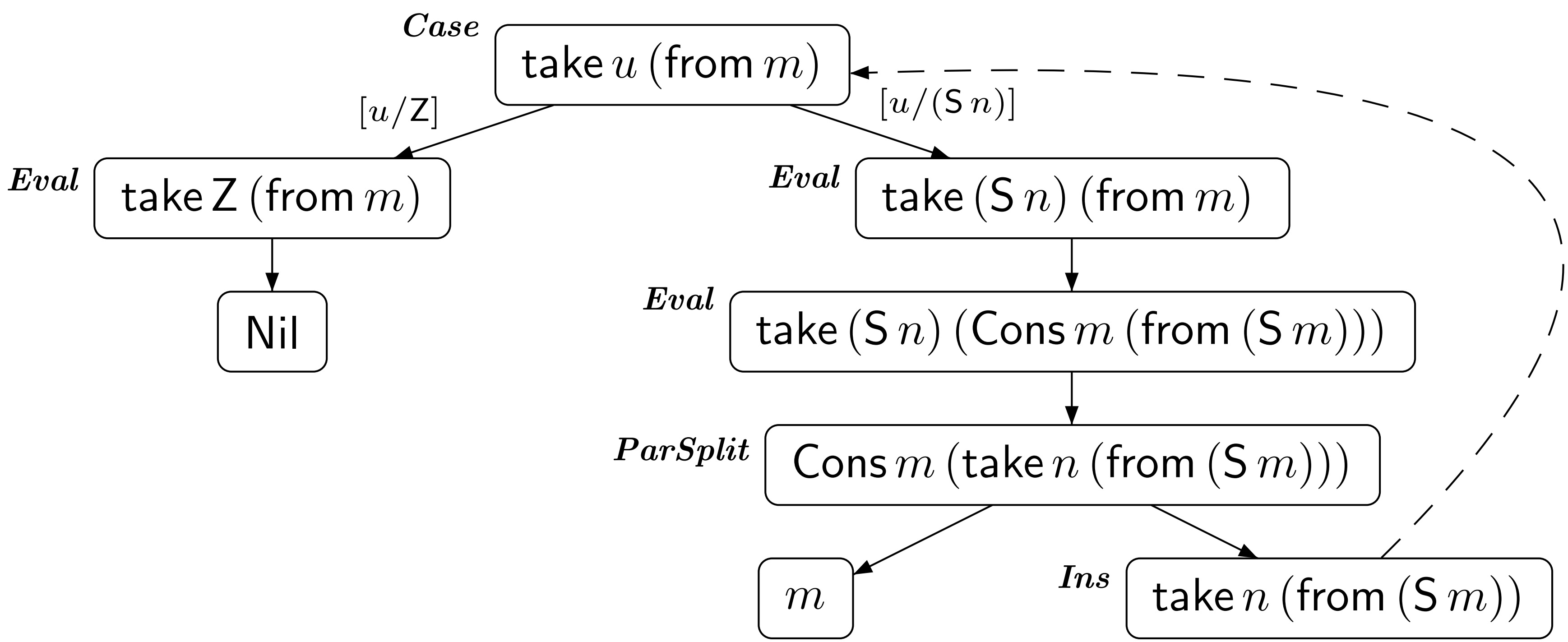


- one could continue with ***Case***, ***Eval***, ***ParSplit***
  $\Rightarrow$ infinite tree
- Instead: ***Ins*** rule to obtain finite graphs

From HASKELL to Termination Graphs

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

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

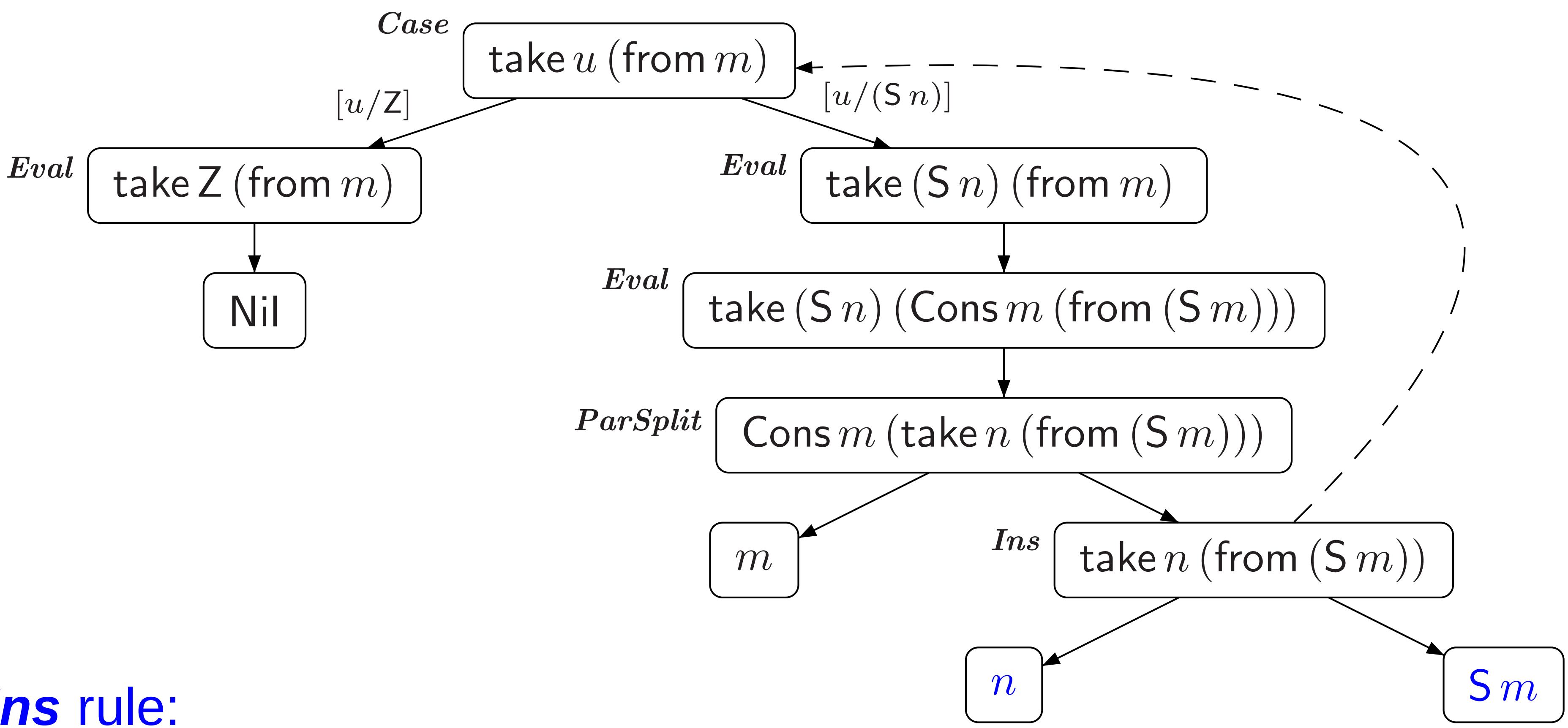


- ***Ins*** **rule:**
  - if leaf t is instance of t', then add instantiation edge from t to t'
  - one may re-use an existing node for t', if possible

From HASKELL to Termination Graphs

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

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

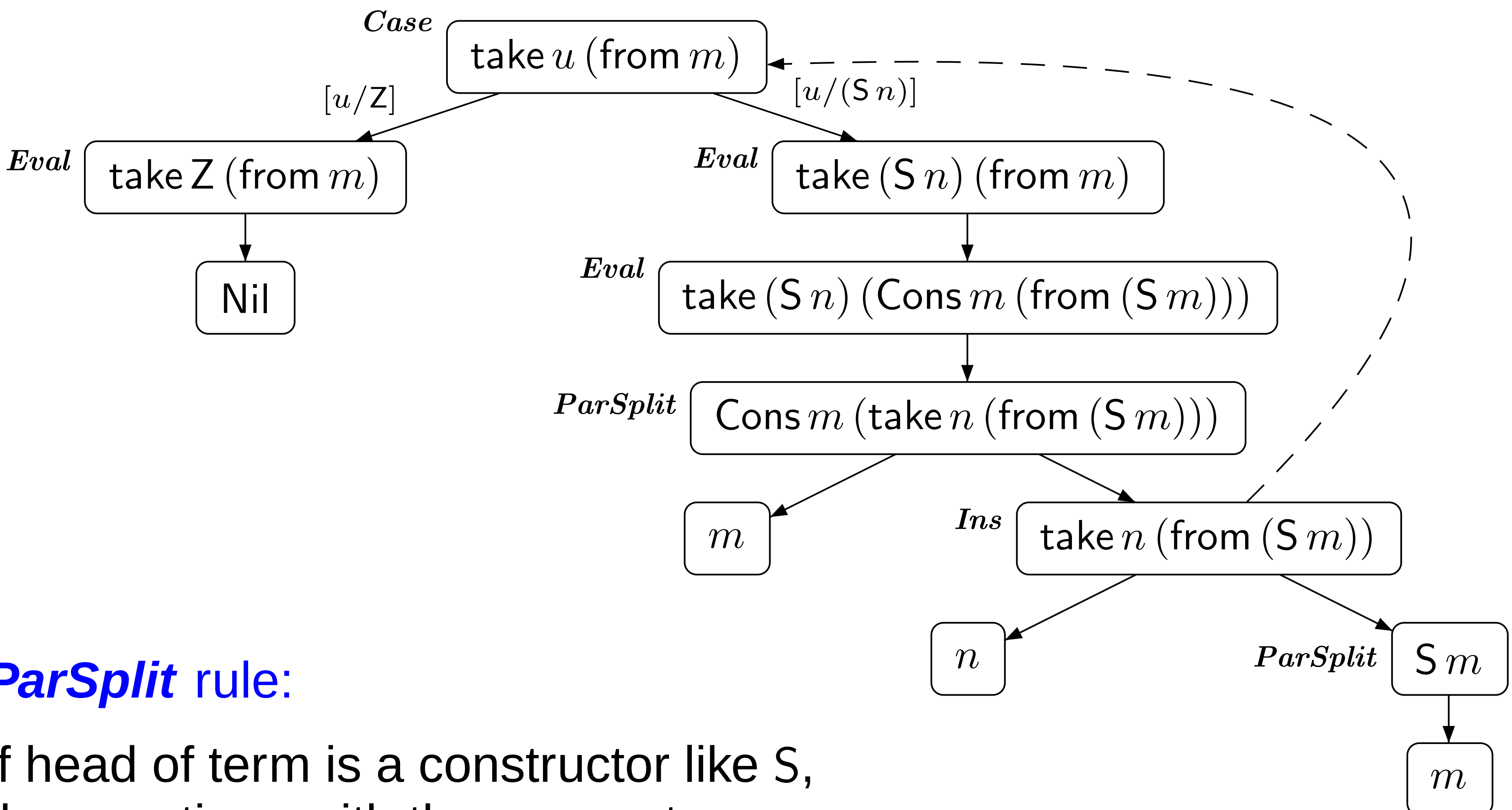


- ***Ins* rule:**
  - if leaf t is instance of t', then add instantiation edge from t to t'
  - since instantiation is $[u/n,\ m/(\mathsf{S}\ m)]$, add child nodes n and $(\mathsf{S}\ m)$

From HASKELL to Termination Graphs

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

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



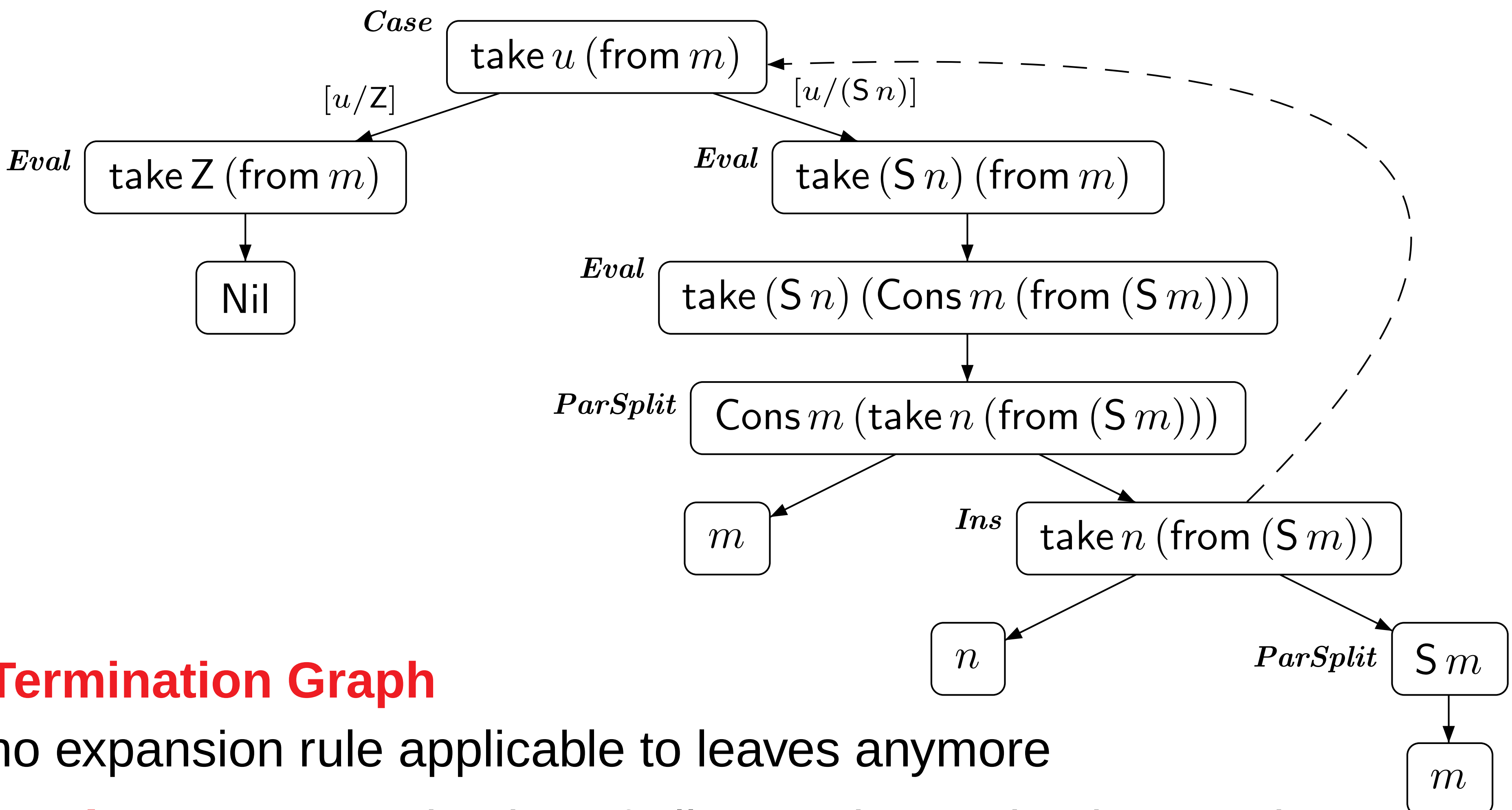
- ***ParSplit*** rule:

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

From HASKELL to Termination Graphs

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

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

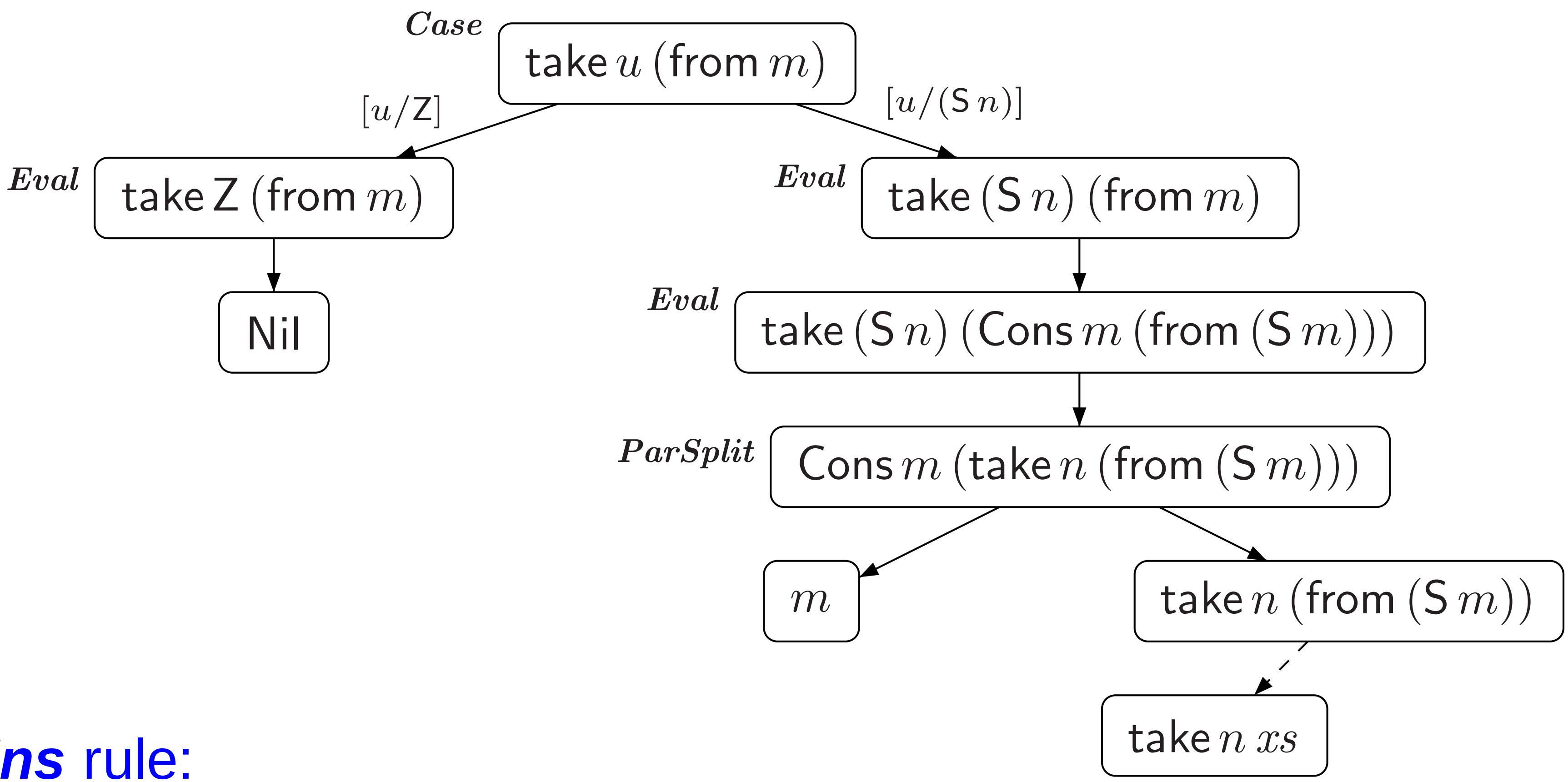


- **Termination Graph**
  no expansion rule applicable to leaves anymore
- **Goal:** Prove H-termination of all terms in termination graph

From HASKELL to Termination Graphs

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

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

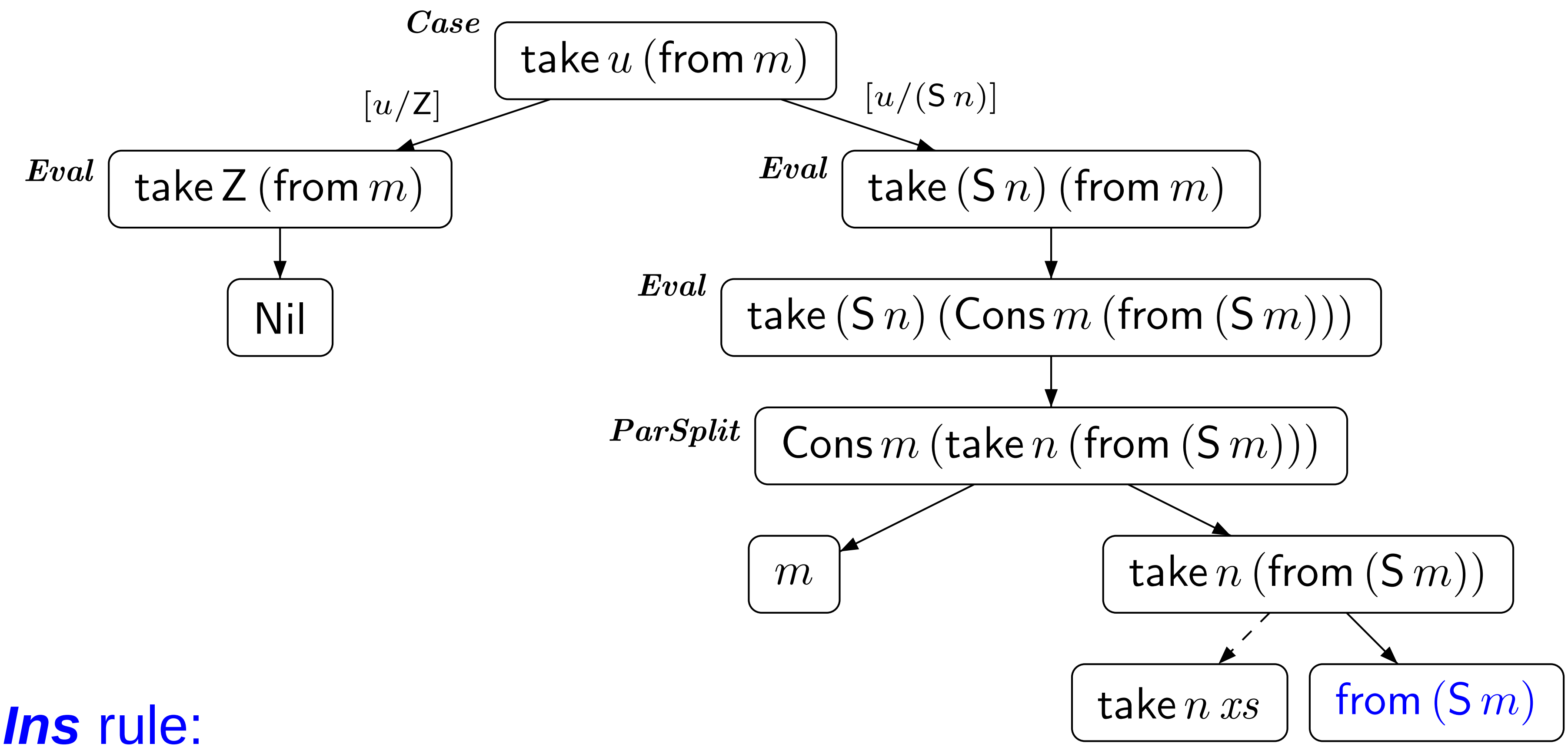


- ***Ins*** **rule:**
  - if leaf t is instance of t', then add instantiation edge from t to t'
  - introduces *indeterminism*

From HASKELL to Termination Graphs

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

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

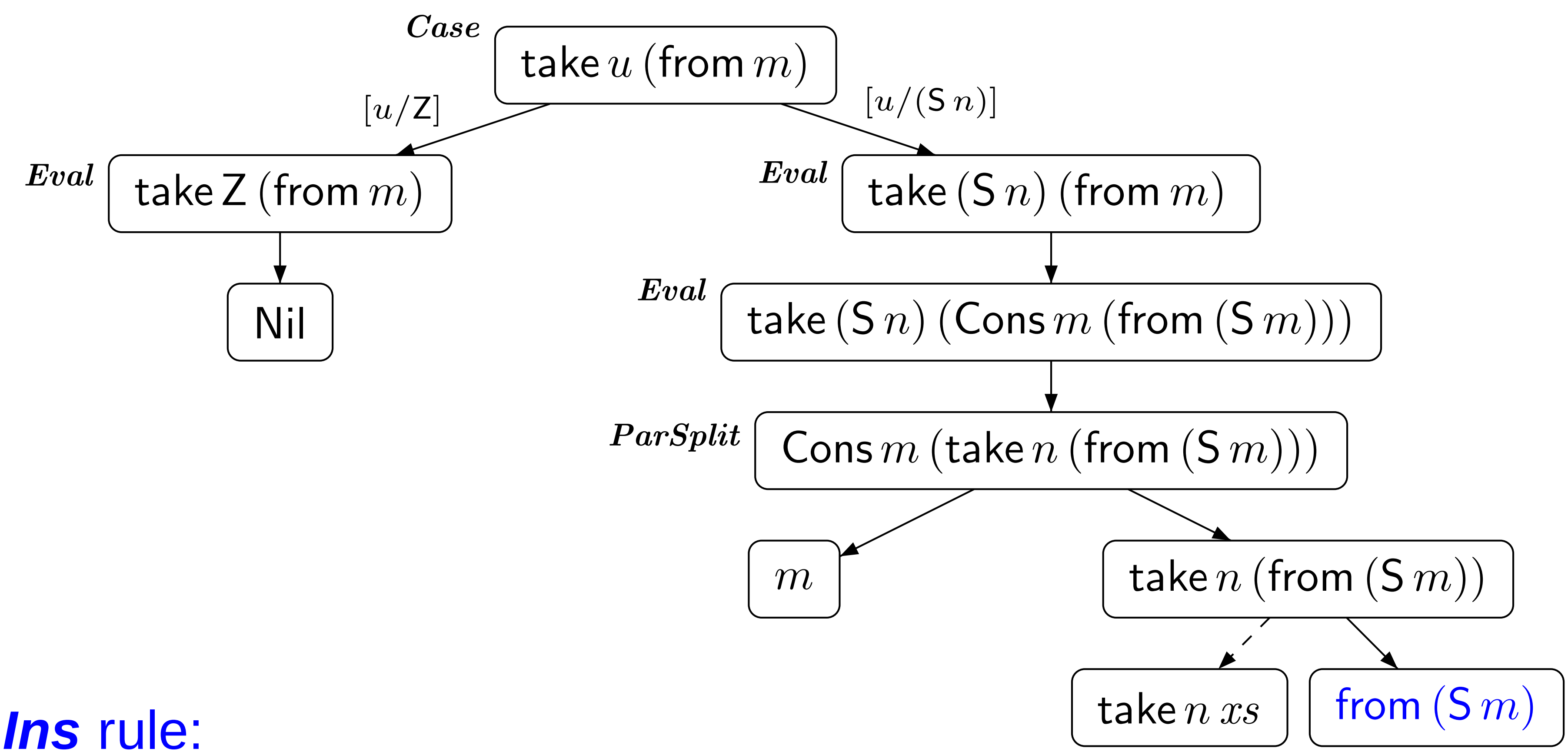


- ***Ins*** **rule:**
  - if leaf t is instance of t', then add instantiation edge from t to t'
  - since instantiation is $[xs/\mathsf{from}\ (\mathsf{S}\ m)]$, add child node $\mathsf{from}\ (\mathsf{S}\ m)$

From HASKELL to Termination Graphs

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

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



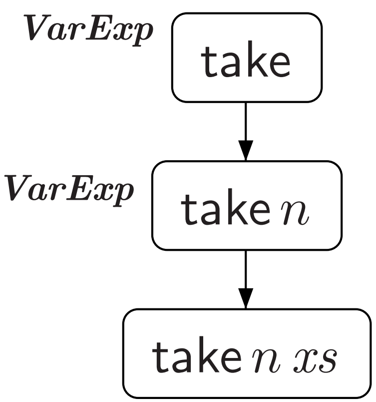
- ***Ins*** rule:
  - if leaf t is instance of t', then add instantiation edge from t to t'
  - proving H-termination of all terms in termination graph fails!

From HASKELL to Termination Graphs

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

Expansion Rules

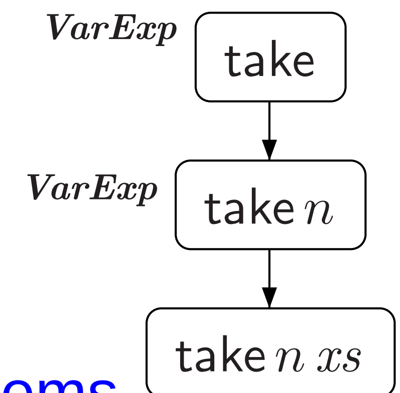
- ***Case***
- ***Eval***
- ***ParSplit***
- ***Ins***
- ***VarExp***



- ***VarExp*** rule:
  - if function is applied to too few arguments,
    then add fresh variable as additional argument

From Termination Graphs to TRSs

- Termination graphs can be obtained for any start term
- **Goal:** Prove H-termination of all terms in termination graph
- **First Approach:**
  Transform termination graph into TRS
  $\Rightarrow$ disadvantageous



- **Better Approach:**
  Transform termination graph into DP problems
- **Dependency Pairs**
  - powerful & popular termination technique for TRSs
  - DP framework allows integration & combination of *any* TRS-termination technique

Dependency Pair Framework

- Apply the general idea of problem solving for termination analysis
  - transform problems into simpler sub-problems repeatedly until all problems are solved
- What *objects* do we work on, i.e., what are the "*problems*"?
  - DP problems $(\mathcal{P}, \mathcal{R})$

    $\mathcal{P}$ *dependency pairs*

    $\mathcal{R}$ *rules*
- What *techniques* do we use for transformation?
  - DP processors: $Proc(\,(\mathcal{P}, \mathcal{R})\,) = \{(\mathcal{P}_1, \mathcal{R}_1), \ldots, (\mathcal{P}_n, \mathcal{R}_n)\}$
- When is a problem *solved*?
  - $(\mathcal{P}, \mathcal{R})$ is *finite* iff there is no infinite $(\mathcal{P}, \mathcal{R})$-chain

    $s_1\sigma_1 \;\to_{\mathcal{P}}\; t_1\sigma_1 \;\to_{\mathcal{R}}^*\; s_2\sigma_2 \;\to_{\mathcal{P}}\; t_2\sigma_2 \;\to_{\mathcal{R}}^*\; \ldots$ where $s_i \to t_i \in \mathcal{P}$

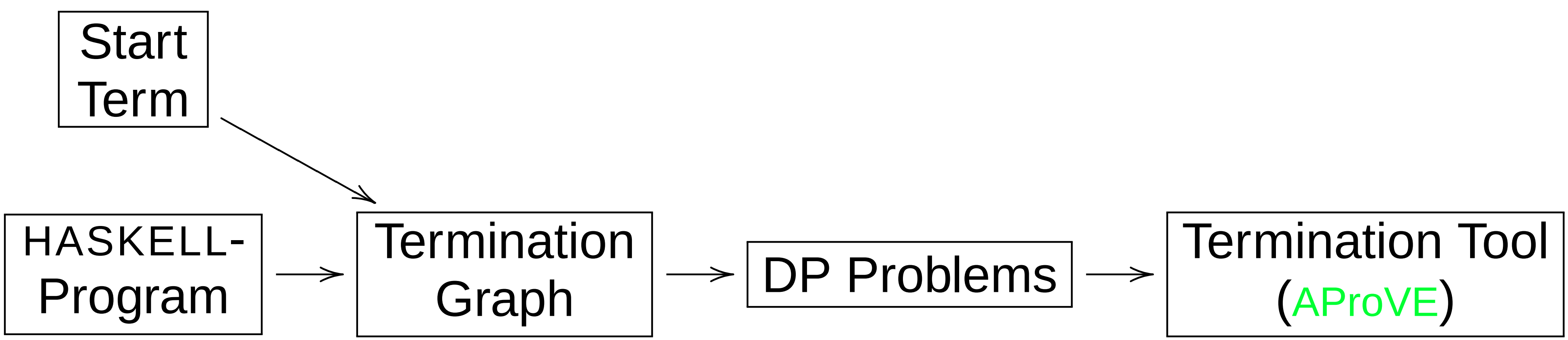
Dependency Pair Framework

Termination of TRS $\mathcal{R}$

- construct initial DP problem $(DP(\mathcal{R}), \mathcal{R})$
- TRS $\mathcal{R}$ is terminating iff initial DP problem is finite
- use DP framework to prove that initial DP problem is finite

Termination of HASKELL

- generate termination graph for start term
- construct initial DP problems from termination graph
- start term is H-terminating if initial DP problems are finite
- use DP framework to prove that initial DP problems are finite

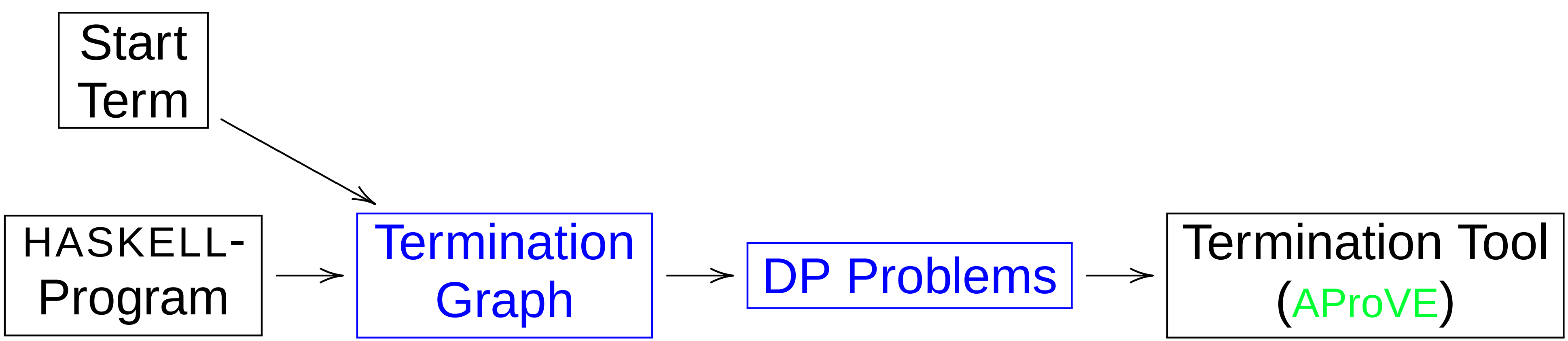


Dependency Pair Framework

How to construct DP problems from termination graph?

Termination of HASKELL

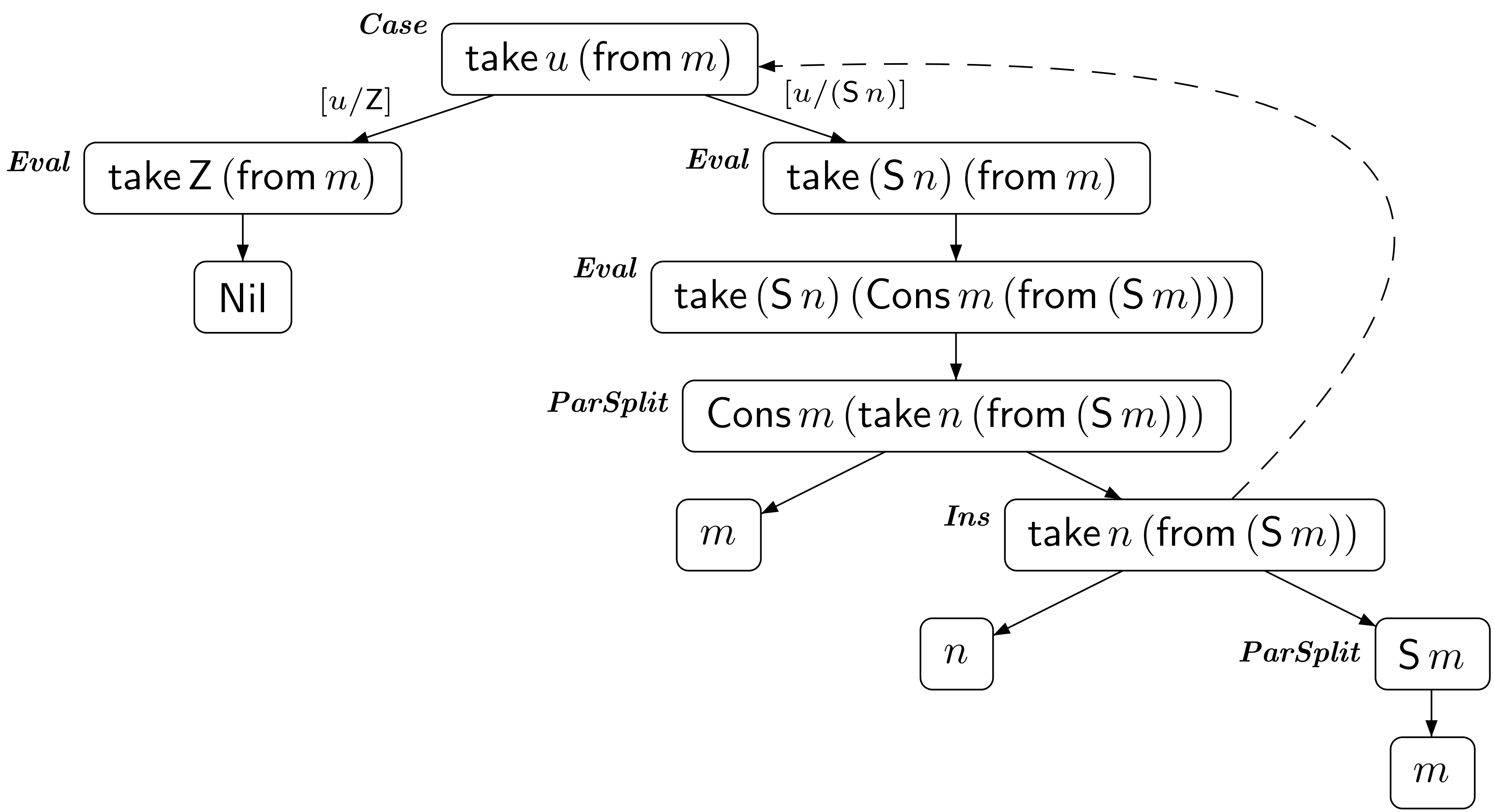
- generate termination graph for start term
- construct initial DP problems from termination graph
- start term is H-terminating if initial DP problems are finite
- use DP framework to prove that initial DP problems are finite



From Termination Graphs to DP Problems

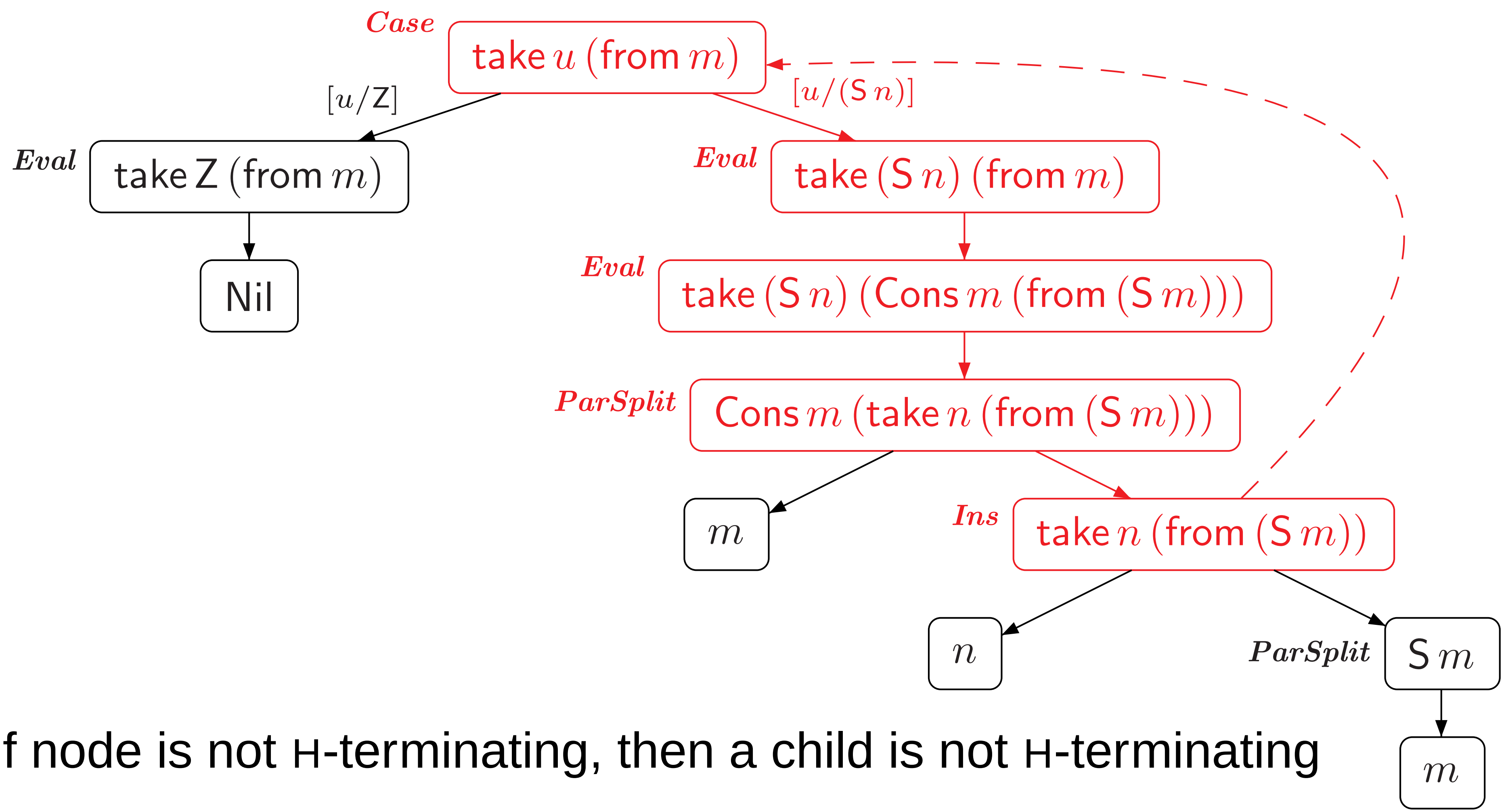
- higher-order terms can be represented as applicative first-order terms

  "$x\ y$" becomes "$\mathsf{ap}(x, y)$"



From Termination Graphs to DP Problems

- **Goal:** Prove H-termination of all terms for each SCC

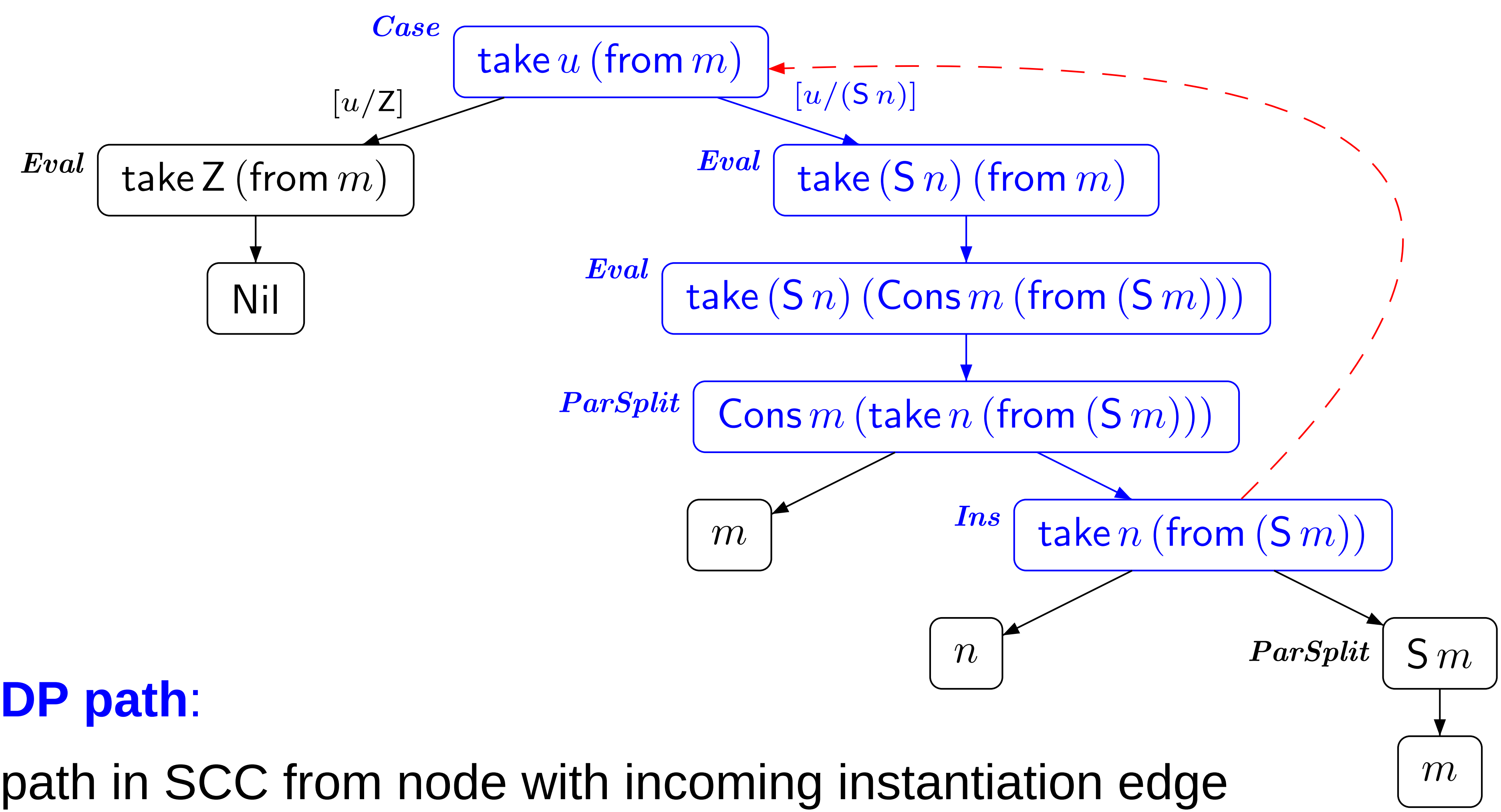


- if node is not H-terminating, then a child is not H-terminating
- not H-terminating node corresponds to SCC

From Termination Graphs to DP Problems

- every infinite path traverses a DP path infinitely often

  ⇒ generate a dependency pair for every DP path



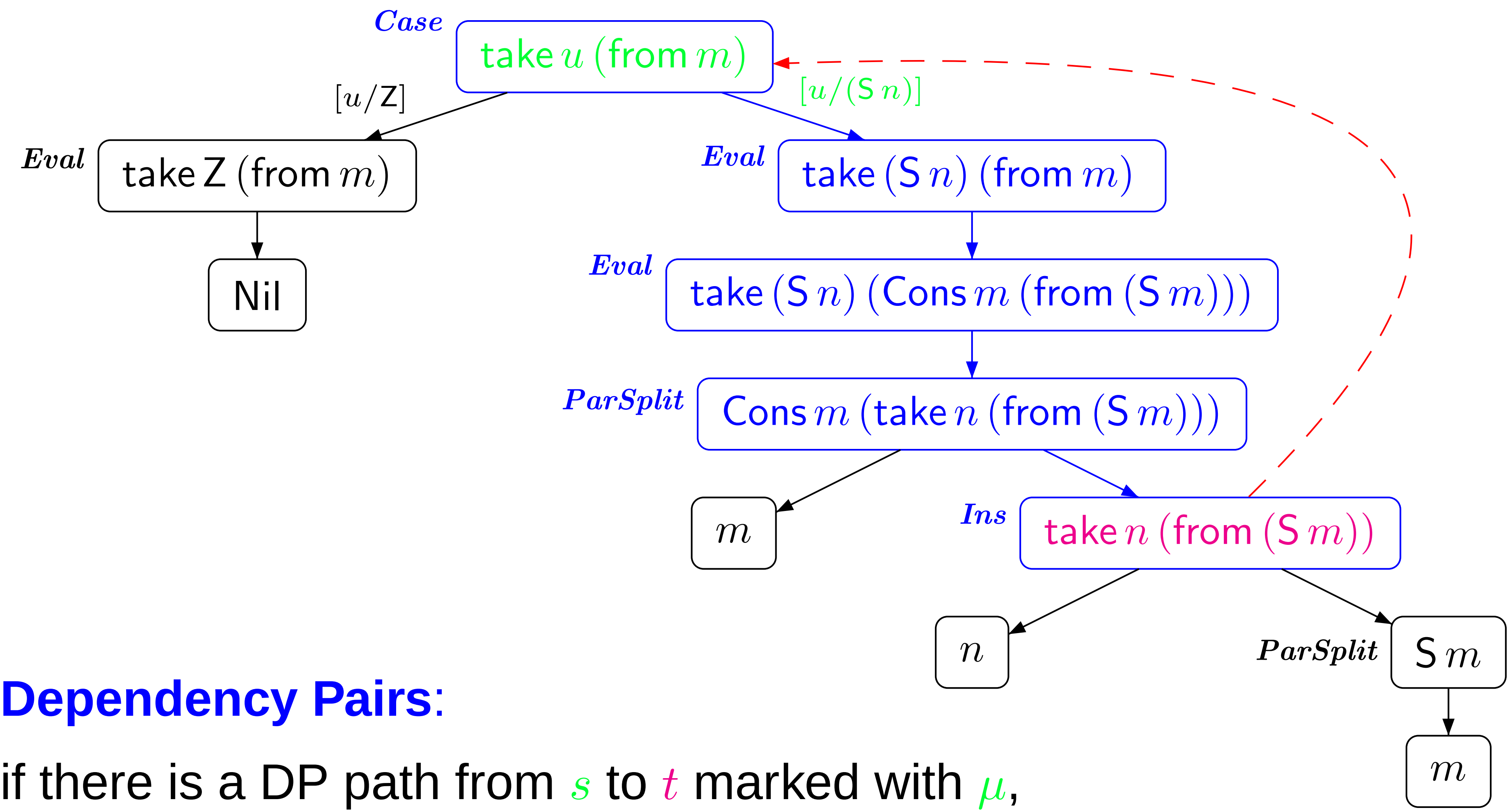
- **DP path**:

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

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 $\mathcal{R}$: $\varnothing$ termination is easy to prove



- **Dependency Pairs**:

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

Generating infinite $(\mathcal{P}, \mathcal{R})$-chains

Term in graph not terminating

$\curvearrowright$ s_1 not terminating

DP path from s_1 to t_1 marked with μ_1

$\curvearrowright$ $s_1 \; \tau_1$ not terminating

$\curvearrowright$ $s_1 \; (\tau_1 \downarrow_\mathsf{H})$ not terminating

$\curvearrowright$ $s_1 \; \mu_1 \; \sigma_1$ not terminating

$\curvearrowright$ $t_1 \; \sigma_1$ not terminating

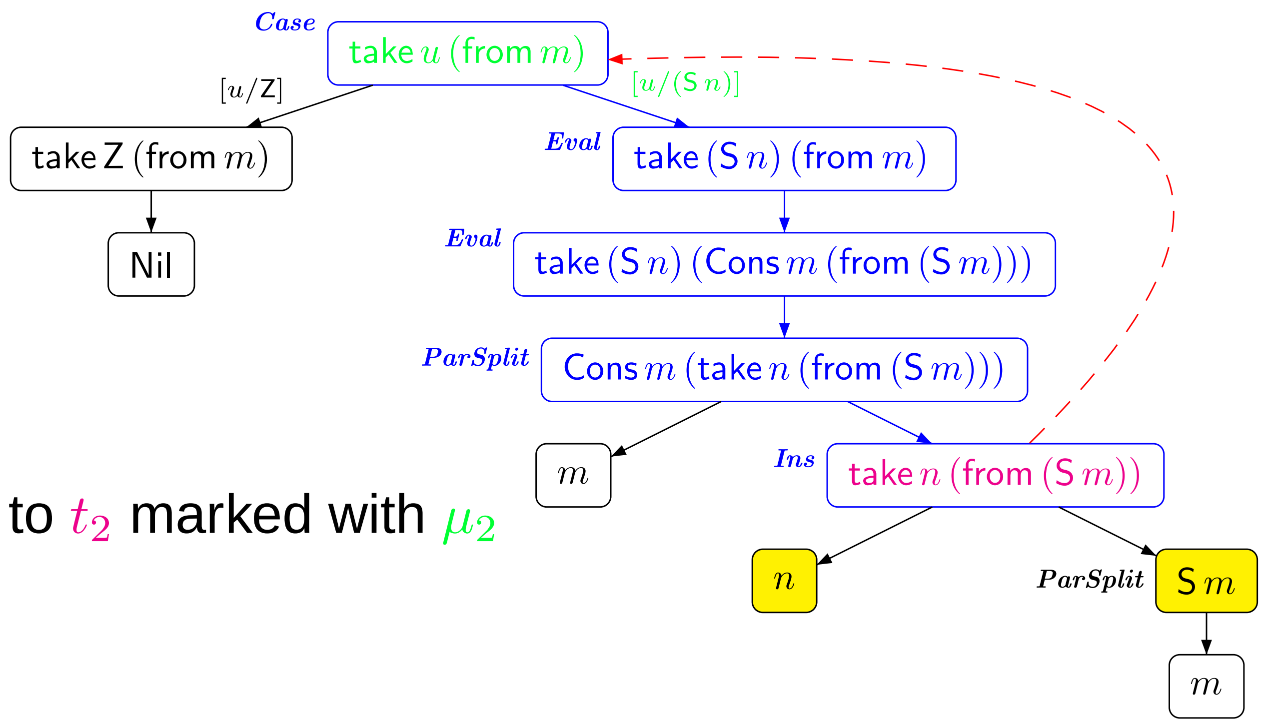
$\curvearrowright$ $s_2 \; \tau_2$ not terminating

DP path from s_2 to t_2 marked with μ_2

$\curvearrowright$ $s_2 \; (\tau_2 \downarrow_\mathsf{H})$ not terminating

$\curvearrowright$ $s_2 \; \mu_2 \; \sigma_2$ not terminating

$\curvearrowright$ $t_2 \; \sigma_2$ not terminating



$$s_1 \, \mu_1 \, \sigma_1 \to_{\mathcal{P}} \underbrace{t_1 \sigma_1}_{s_2 \, \tau_2} \quad \to_{\mathcal{R}}^* \quad \underbrace{s_2 \, \mu_2 \, \sigma_2}_{s_2 \, (\tau_2 \downarrow_\mathsf{H})} \to_{\mathcal{P}} t_2 \sigma_2$$

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

$\mathcal{R} = \varnothing$ if no defined symbol in matcher

From Termination Graphs to DP Problems

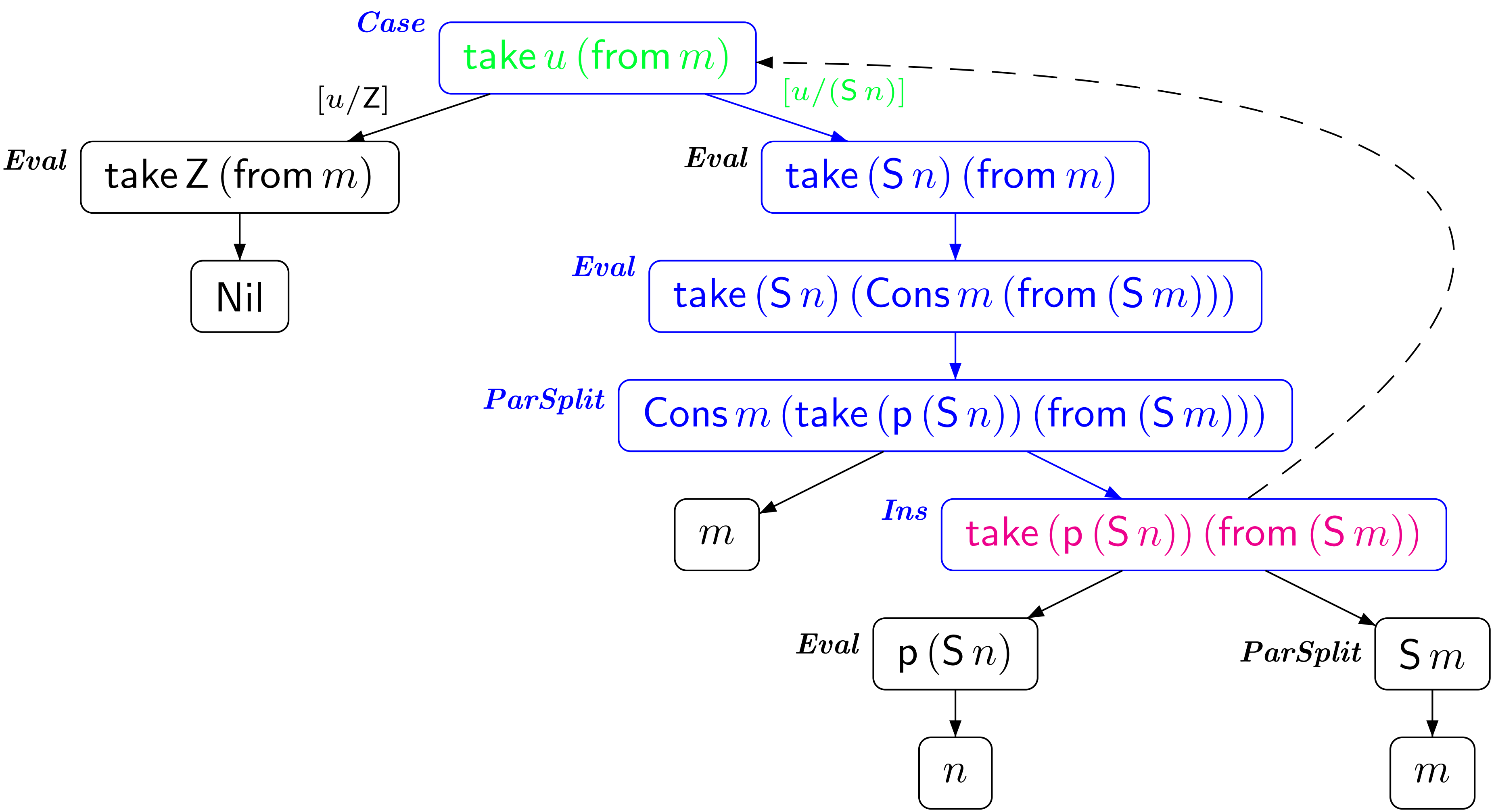
$\mathsf{from}\ x = \mathsf{Cons}\ x\ (\mathsf{from}\ (\mathsf{S}\ x))$

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

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

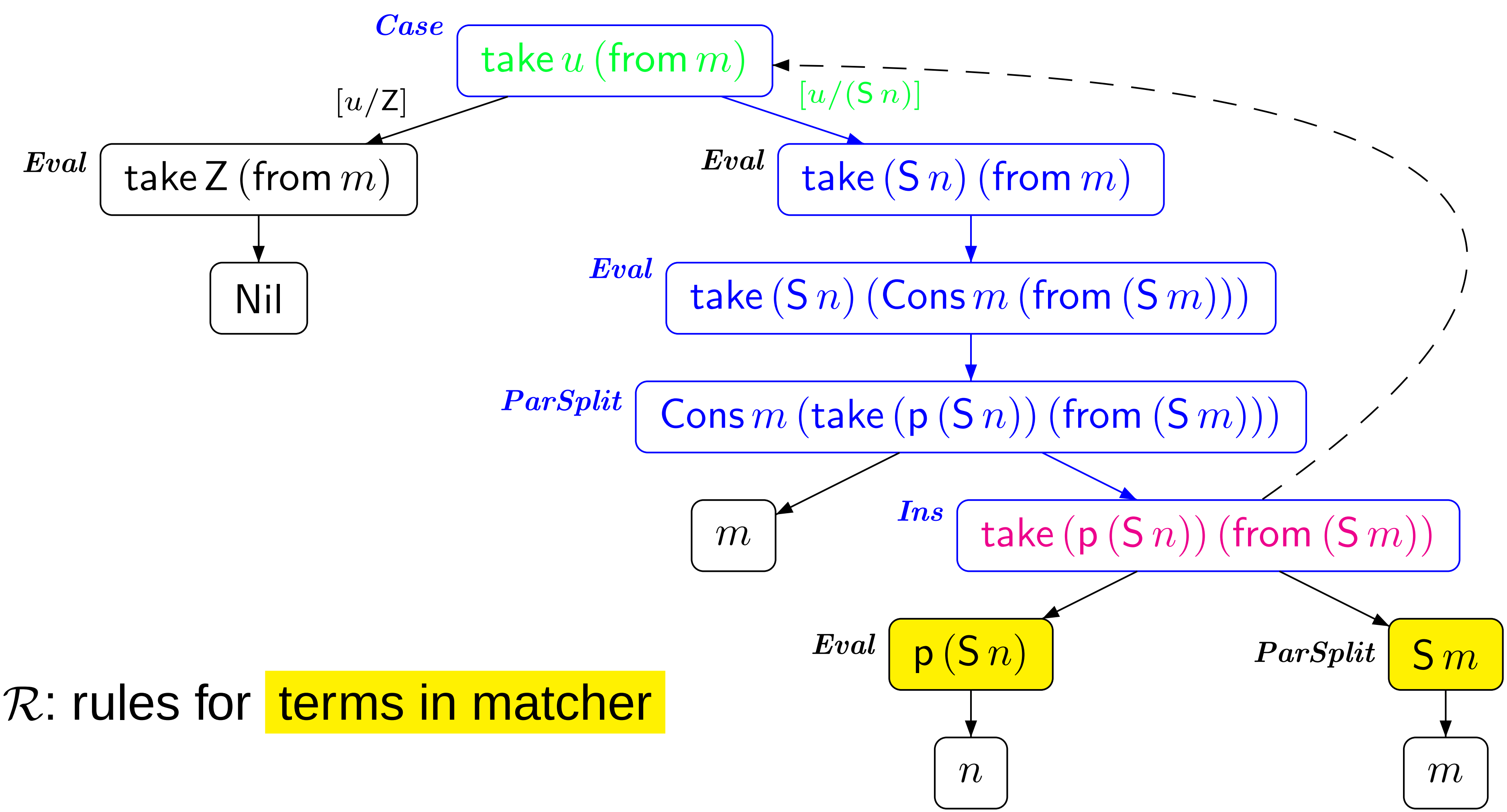
$\mathsf{take}\ n\ \mathsf{Nil} = \mathsf{Nil}$

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



From Termination Graphs to DP Problems

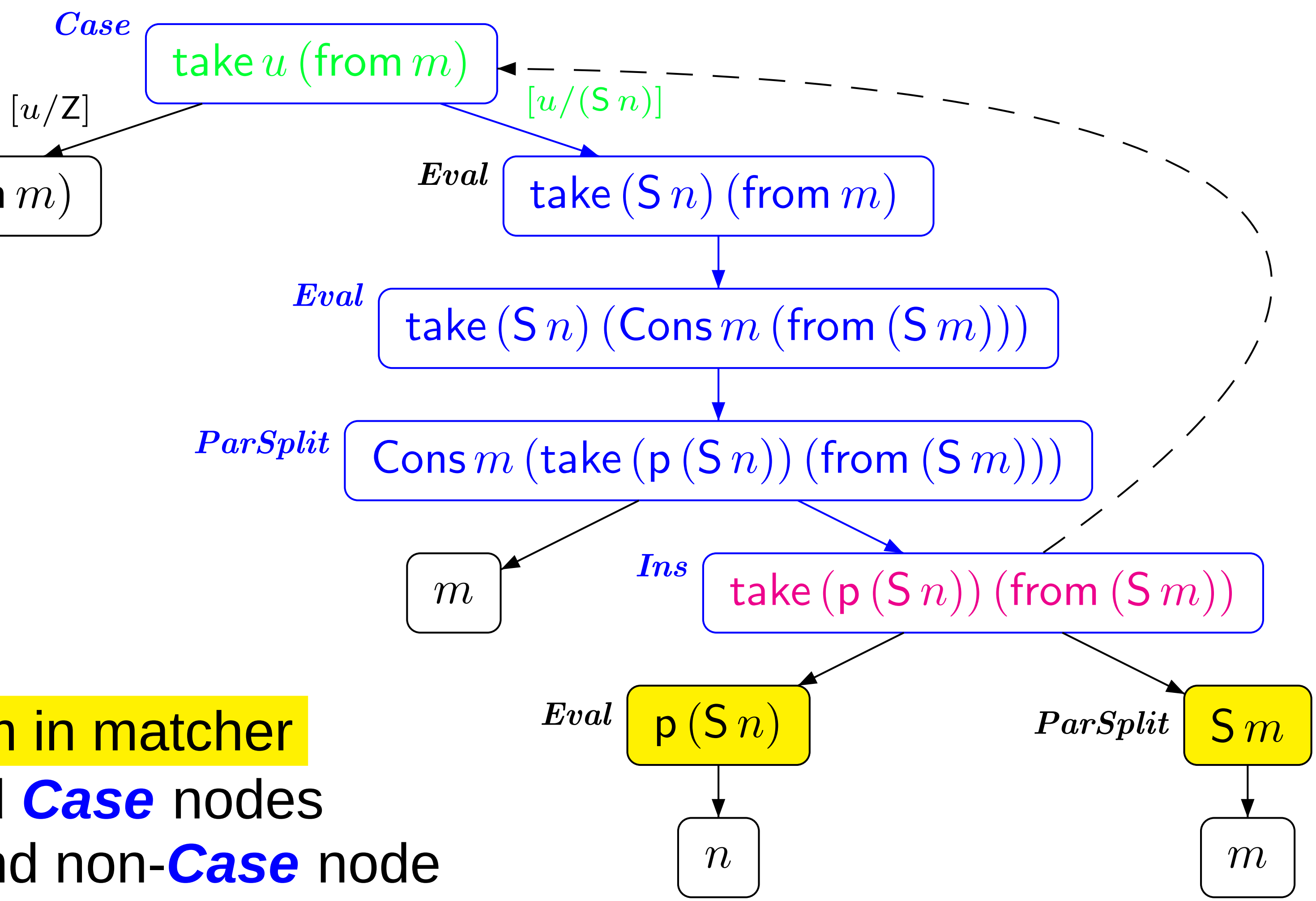
- **Dependency Pair $\mathcal{P}$:** take (S n) (from m) $\rightarrow$ take (p (S n)) (from (S m))



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

From Termination Graphs to DP Problems

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



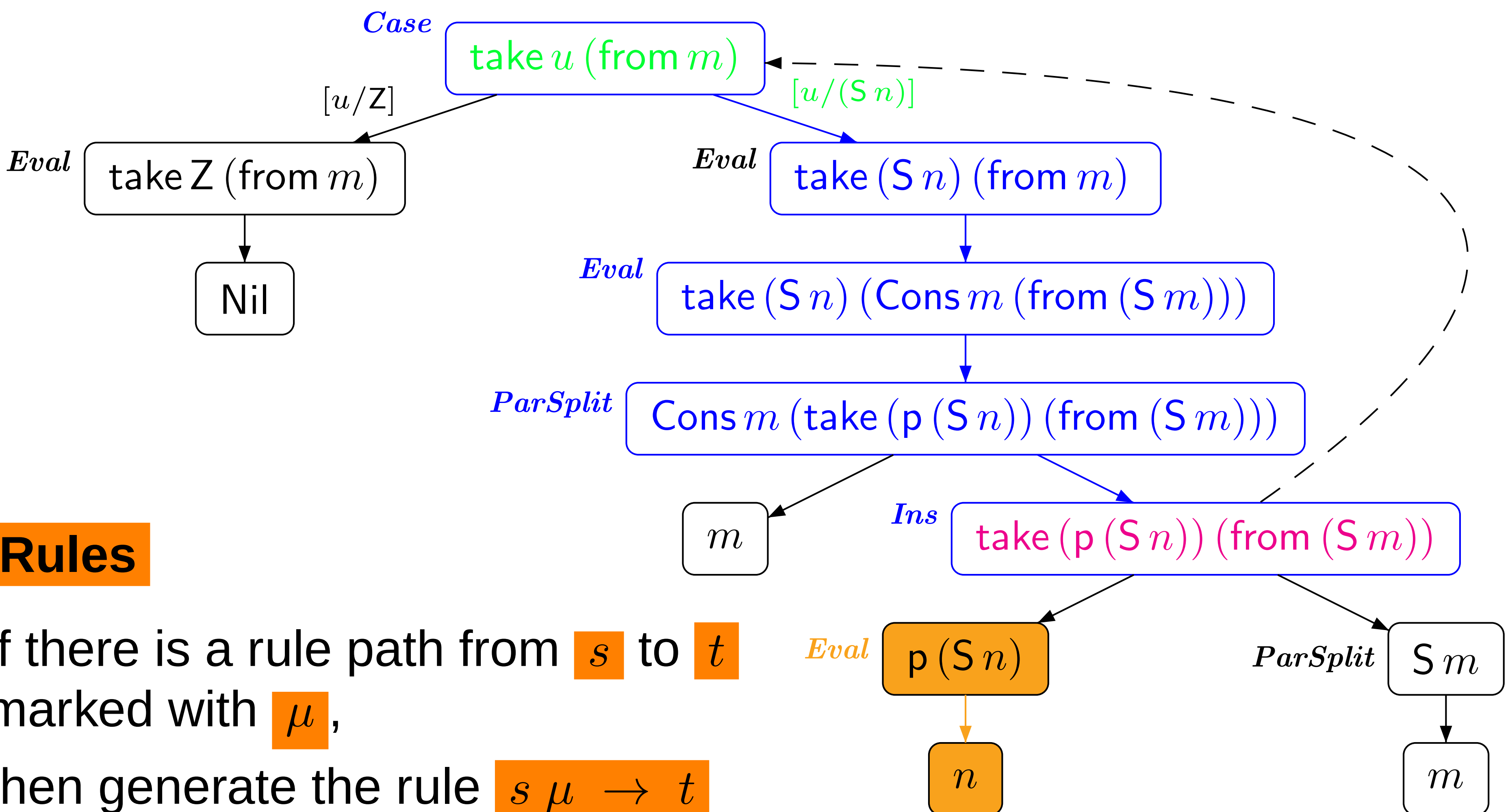
- **Rule path**

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

From Termination Graphs to DP Problems

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

  Rule $\mathcal{R}$: p (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 $s\,\mu \to t$

From Termination Graphs to DP Problems

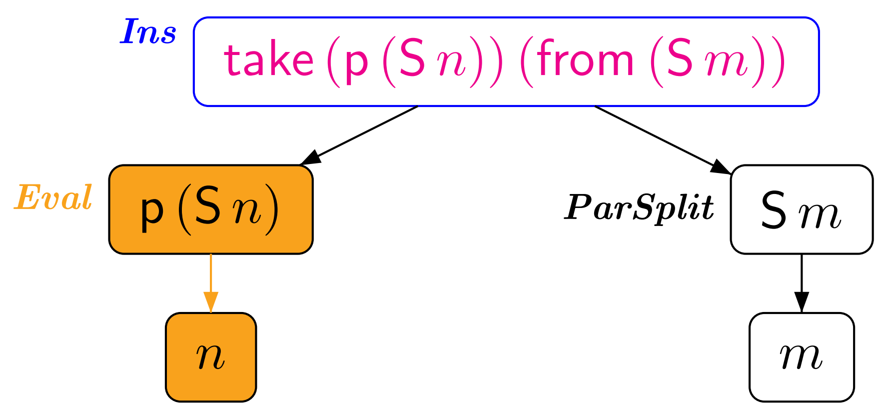
- **Dependency Pair $\mathcal{P}$:** $\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{from}\,m) \to \mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\,n))\,(\mathsf{from}\,(\mathsf{S}\,m))$

  Rule $\mathcal{R}$: $\mathsf{p}\,(\mathsf{S}\,n) \to n$

- **Improvement:** evaluate rhs of DP as much as possible

$$
\begin{aligned}
& \mathbf{ev}(\,\mathsf{take}\;(\,\mathsf{p}\,(\mathsf{S}\,n)\,)\;(\mathsf{from}\;(\,\mathsf{S}\,m\,))\,) \\
= \;& \mathsf{take}\;\mathbf{ev}(\,\mathsf{p}\,(\mathsf{S}\,n)\,)\;(\mathsf{from}\;\mathbf{ev}(\,\mathsf{S}\,m\,)) \\
= \;& \mathsf{take}\;n\;(\mathsf{from}\;(\,\mathsf{S}\,m\,))
\end{aligned}
$$

- $\mathbf{ev}(\,t\,)$: term reachable from t by traversing ***Eval***-nodes
  traverses subterms of ***ParSplit***- and ***Ins***-nodes



From Termination Graphs to DP Problems

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

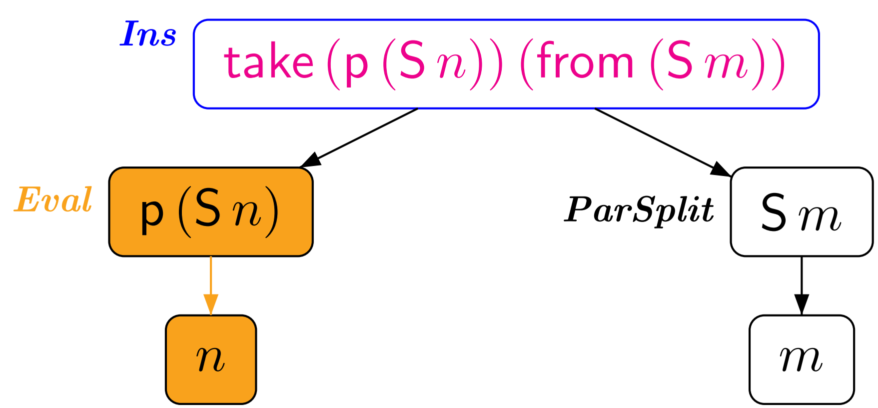
- **Improvement:** evaluate rhs of DP as much as possible

$$\mathbf{ev}(\text{ take } (\text{ p } (\text{S } n)\ )\ (\text{from } (\text{ S } m\ ))\ )$$
$$= \text{take } \mathbf{ev}(\text{ p } (\text{S } n)\ )\ (\text{from } \mathbf{ev}(\text{ S } m\ ))$$
$$= \text{take } n\ (\text{from } (\text{ S } m\ ))$$

- $\mathbf{ev}(\,t\,)$: term reachable from t by traversing ***Eval***-nodes
  traverses subterms of ***ParSplit***- and ***Ins***-nodes

- **Rules**

  only needed for terms
  where computation of $\mathbf{ev}$ stopped

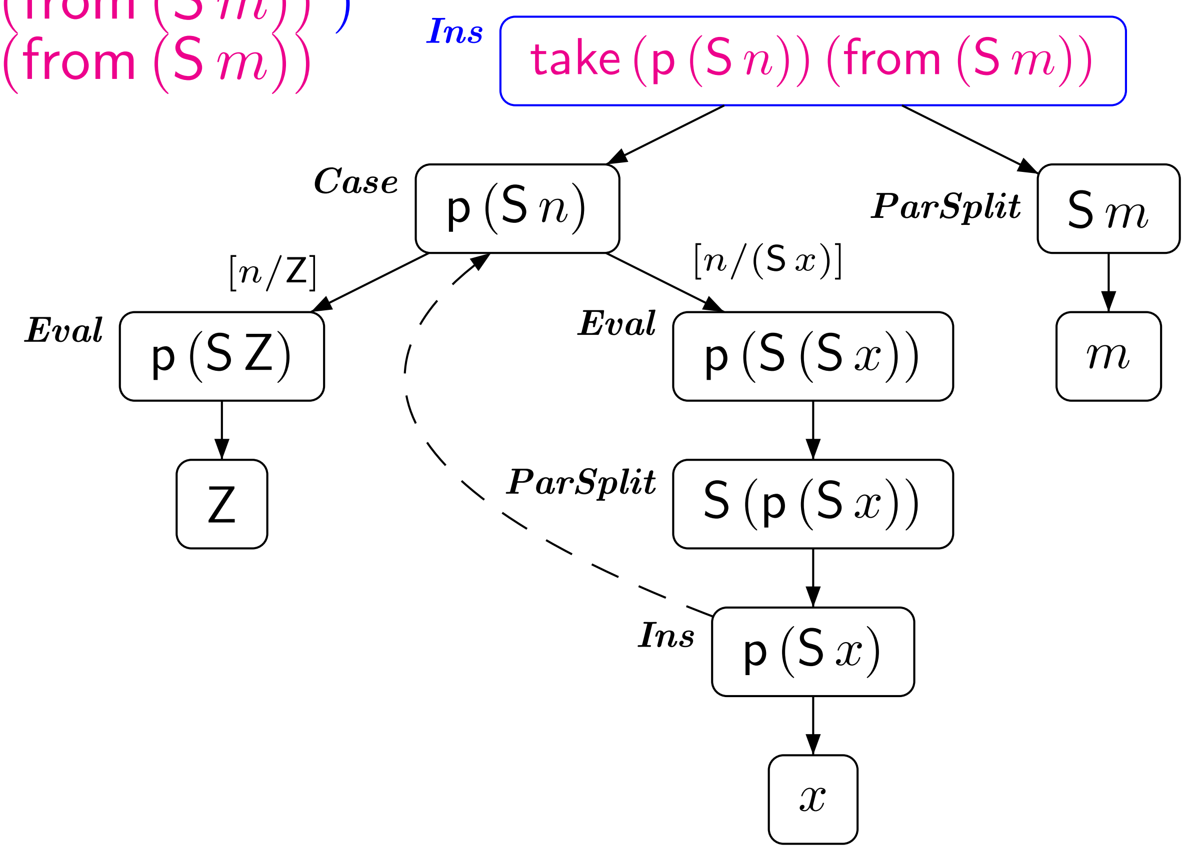


From Termination Graphs to DP Problems

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

- **Improvement:** evaluate rhs of DP as much as possible

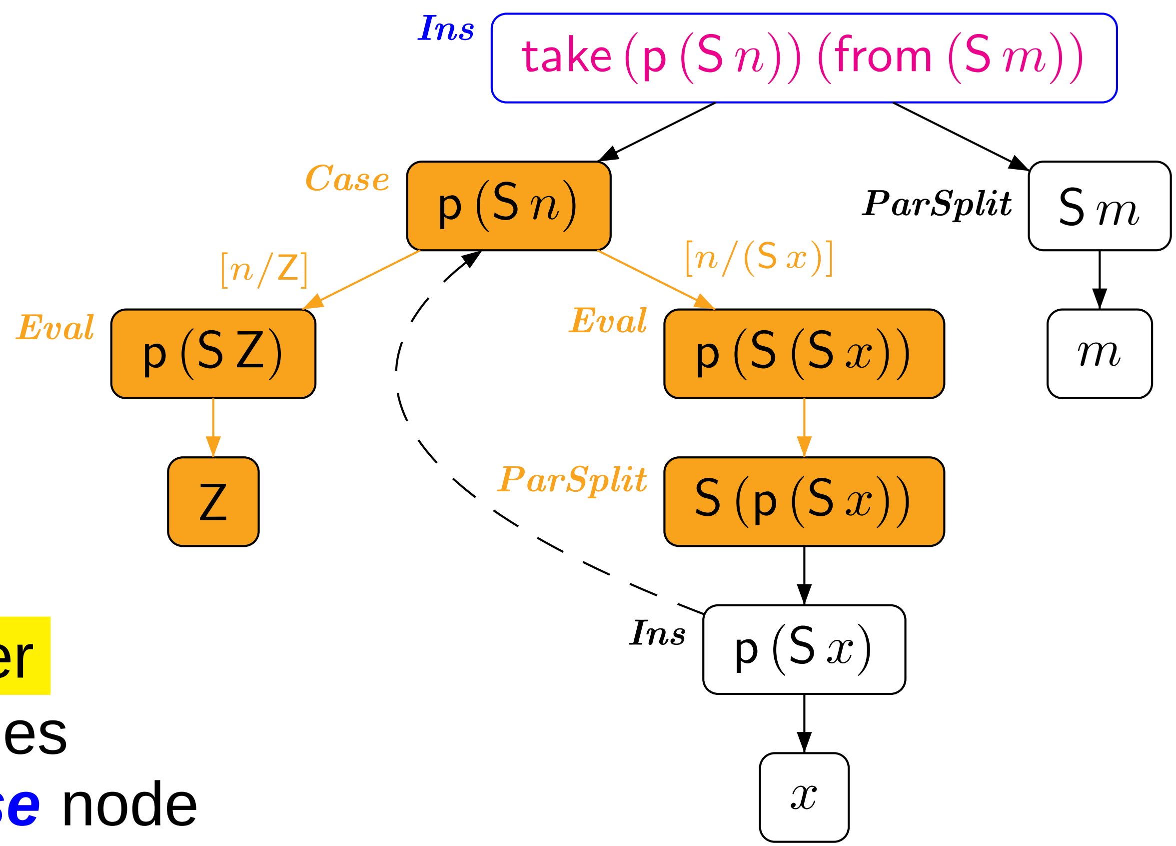
$$\begin{array}{ll} & \mathbf{ev}(\ \text{take}\,(\text{p}\,(\text{S}\,n))\,(\text{from}\,(\text{S}\,m))\ ) \\ = & \text{take}\,(\text{p}\,(\text{S}\,n))\,(\text{from}\,(\text{S}\,m)) \end{array}$$



$$\begin{array}{l} \text{p}\,(\text{S}\,\text{Z}) = \text{Z} \\ \text{p}\,(\text{S}\,x) = \text{S}\,(\text{p}\,x) \end{array}$$

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 matcher
  over ***Eval***- and ***Case*** nodes
  to non-***Eval*** and non-***Case*** node

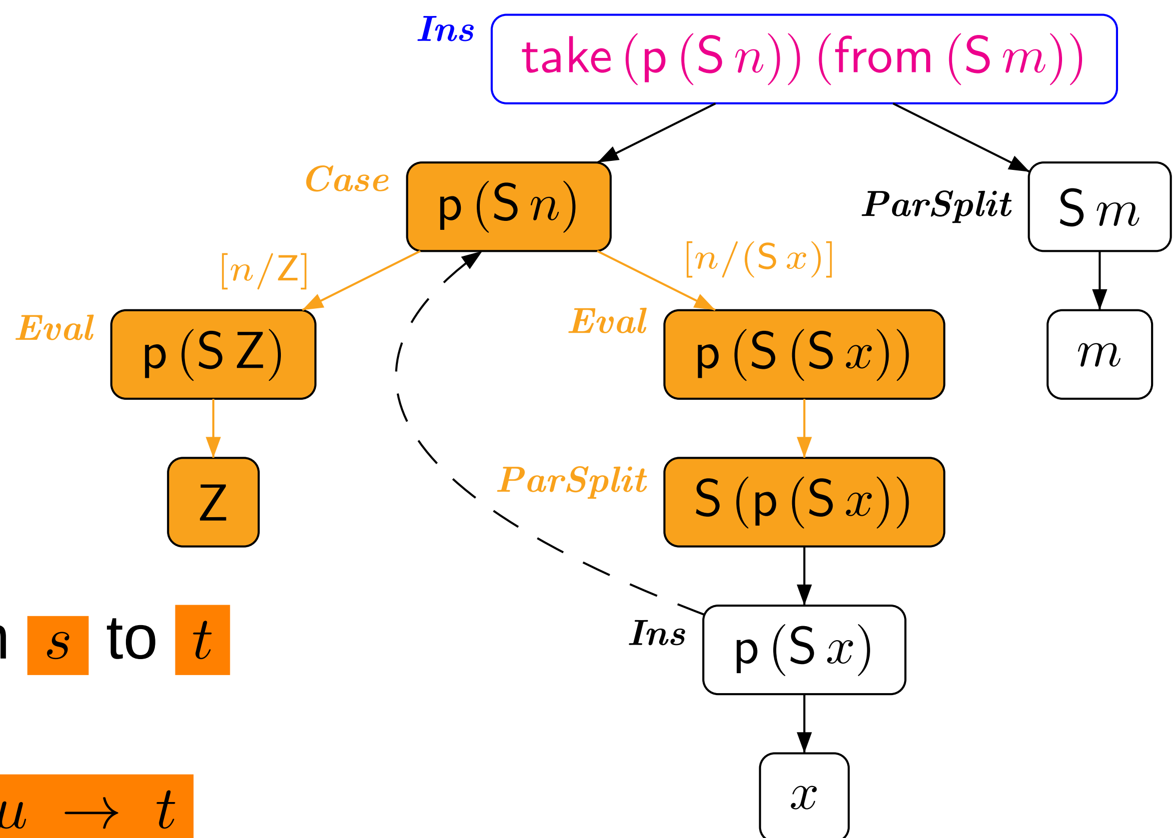
From Termination Graphs to DP Problems

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

  Rules $\mathcal{R}$:

  $\mathsf{p}\,(\mathsf{S}\,\mathsf{Z}) \to \mathsf{Z}$

  $\mathsf{p}\,(\mathsf{S}\,(\mathsf{S}\,x)) \to \mathsf{S}\,(\mathsf{p}\,(\mathsf{S}\,x))$



- **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

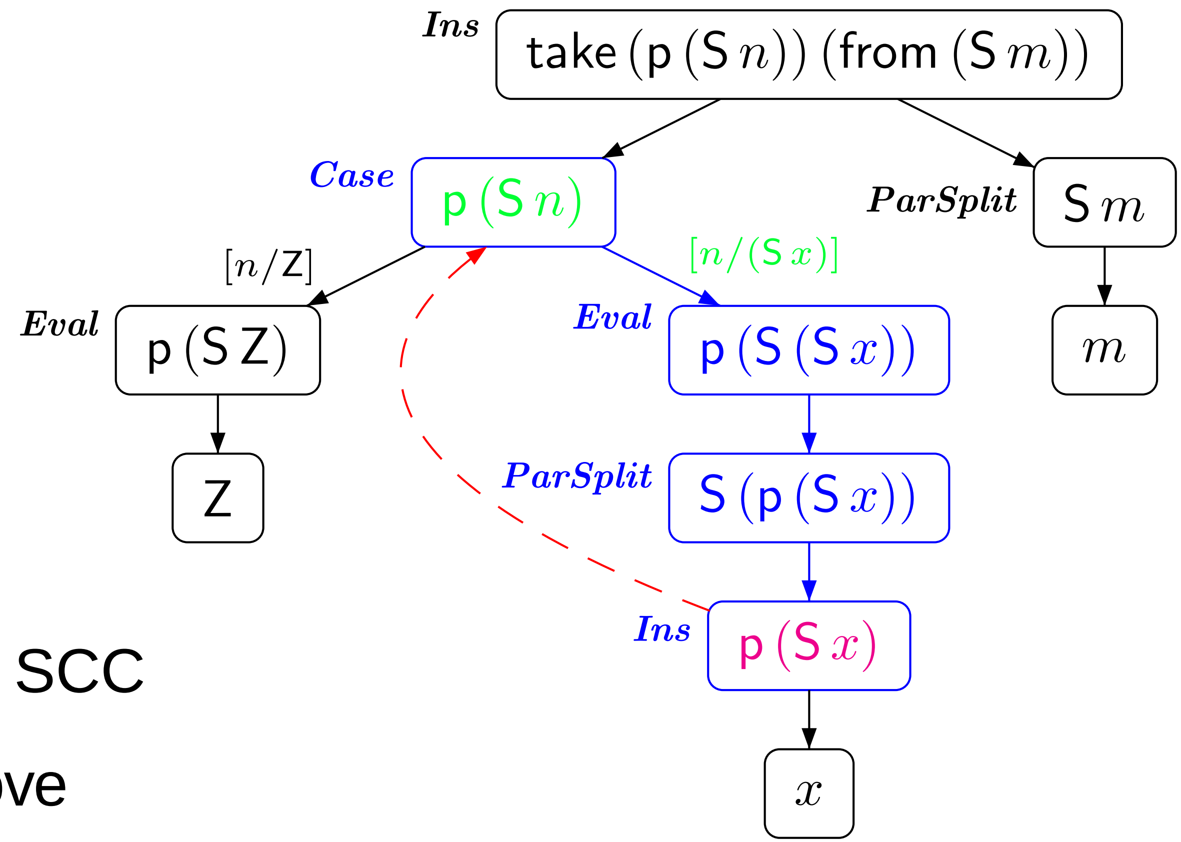
- **Dependency Pair $\mathcal{P}$:** $\mathsf{take}\,(\mathsf{S}\,n)\,(\mathsf{from}\,m) \;\to\; \mathsf{take}\,(\mathsf{p}\,(\mathsf{S}\,n))\,(\mathsf{from}\,(\mathsf{S}\,m))$

  Rules $\mathcal{R}$: $\mathsf{p}\,(\mathsf{S}\,\mathsf{Z}) \;\to\; \mathsf{Z}$

  $\mathsf{p}\,(\mathsf{S}\,(\mathsf{S}\,x)) \;\to\; \mathsf{S}\,(\mathsf{p}\,(\mathsf{S}\,x))$

- **Dependency Pair $\mathcal{P}$:** $\mathsf{p}\,(\mathsf{S}\,(\mathsf{S}\,x)) \to \mathsf{p}\,(\mathsf{S}\,x)$

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



- one DP problem for each SCC
- termination is easy to prove

Termination of HASKELL-Programs

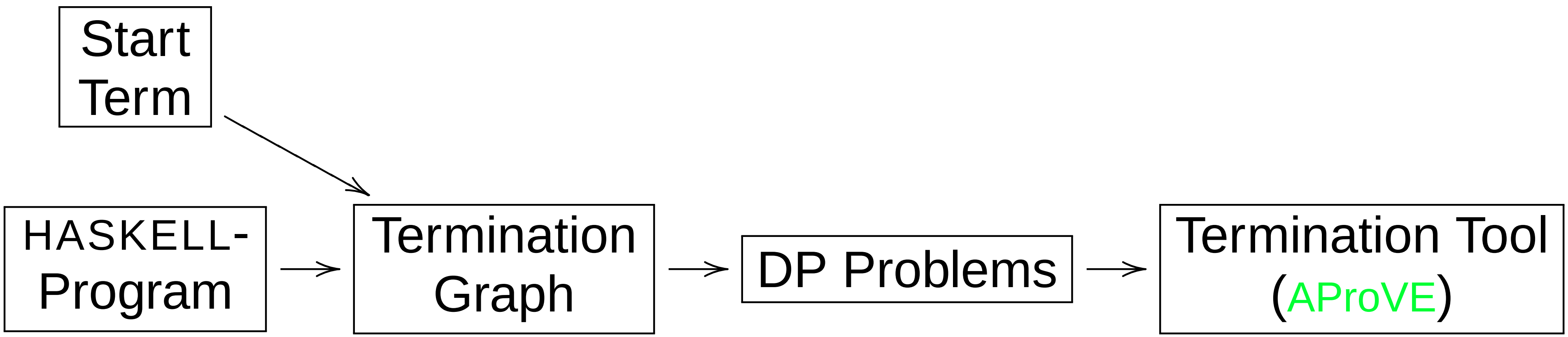
- **New approach in order to use TRS-techniques for HASKELL**
  - generate termination graph for given start term
  - extract DP problems from termination graph
  - prove finiteness of DP problems by existing TRS-techniques

- **Implemented in AProVE**
  - accepts full HASKELL 98 language
  - successfully evaluated with standard HASKELL-libraries

	FiniteMap	List	Maybe	Monad	Prelude	Queue	Total
YES	256	166	9	69	489	5	994
TOTAL	321	174	9	80	692	5	1281

Termination of HASKELL-Programs

- **New approach in order to use TRS-techniques for HASKELL**



- **Implemented in AProVE**
  - accepts full HASKELL 98 language
  - successfully evaluated with standard HASKELL-libraries

	FiniteMap	List	Maybe	Monad	Prelude	Queue	Total
YES	256	166	9	69	489	5	994
TOTAL	321	174	9	80	692	5	1281

Overview

I. Termination of Term Rewriting

1. Termination of Term Rewrite Systems
2. Non-Termination of Term Rewrite Systems
3. Complexity of Term Rewrite Systems
4. Termination of Integer Term Rewrite Systems

II. Termination of Programs

1. Termination of Functional Programs (Haskell)
2. Termination of Logic Programs (Prolog) (PPDP '12)
3. Termination of Imperative Programs (Java)

Termination of Logic Programming Languages

- well-developed field *(De Schreye & Decorte, 94) etc.*
- **direct approaches:** work directly on the logic program
  - cTI *(Mesnard et al)*
  - TerminWeb *(Codish et al)*
  - TermiLog *(Lindenstrauss et al)*
  - Polytool *(Nguyen, De Schreye, Giesl, Schneider-Kamp)*

  TRS-techniques can be adapted to work *directly* on the LP
- **transformational approaches:** transform LP to TRS
  - TALP *(Ohlebusch et al)*
  - AProVE *(Giesl et al)*
- only for *definite* LP (without cut)
- not for real PROLOG

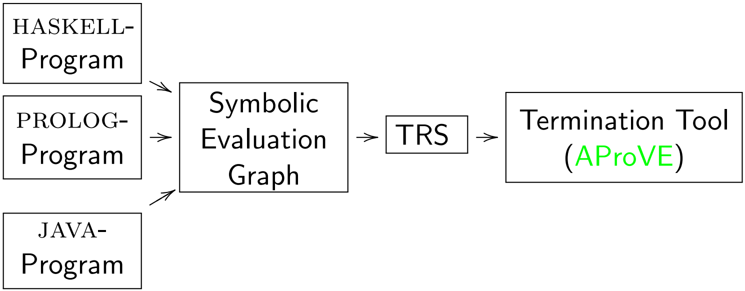
Termination of Logic Programming Languages

- analyzing PROLOG is challenging due to cuts etc.
- **New approach**
  - Frontend
    - evaluate PROLOG a few steps $\Rightarrow$ symbolic evaluation graph

      graph captures evaluation strategy due to cuts etc.
    - transform symbolic evaluation graph $\Rightarrow$ TRS
  - Backend
    - prove termination of the resulting TRS
      (using existing techniques & tools)
- implemented in **AProVE**
  - successfully evaluated on PROLOG-collections with cuts
  - most powerful termination tool for PROLOG
    (winner of *termination competition* for PROLOG)

Termination of Logic Programming Languages

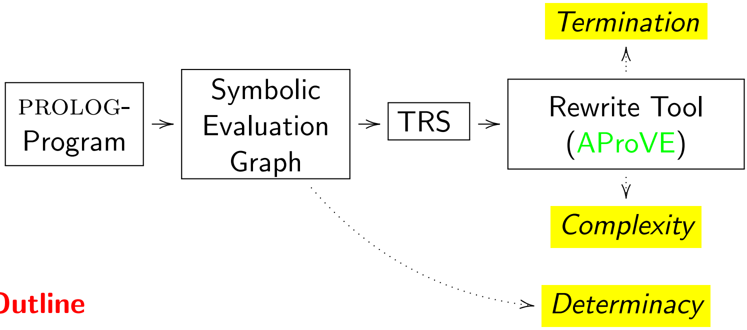
- analyzing PROLOG is challenging due to cuts etc.



- implemented in **AProVE**
  - successfully evaluated on PROLOG-collections with cuts
  - most powerful termination tool for PROLOG (winner of *termination competition* for PROLOG)

Symbolic Evaluation Graphs and Term Rewriting

General methodology for analyzing PROLOG programs



Outline

- linear operational semantics of PROLOG
- from PROLOG to symbolic evaluation graphs
- from symbolic evaluation graphs to TRSs for termination analysis
- from symbolic evaluation graphs to TRSs for complexity analysis
- determinacy analysis

$$
\begin{aligned}
\mathsf{star}(XS, [\,]) &\mathrel{:\!-} !. && (1)\\
\mathsf{star}([\,], ZS) &\mathrel{:\!-} !, \mathsf{eq}(ZS, [\,]). && (2)\\
\mathsf{star}(XS, ZS) &\mathrel{:\!-} \mathsf{app}(XS, YS, ZS), \mathsf{star}(XS, YS). && (3)\\
\mathsf{app}([\,], YS, YS). && && (4)\\
\mathsf{app}([X \mid XS], YS, [X \mid ZS]) &\mathrel{:\!-} \mathsf{app}(XS, YS, ZS). && (5)\\
\mathsf{eq}(X, X). && && (6)
\end{aligned}
$$

- $\mathsf{star}(t_1, t_2)$ holds iff t_2 results from concatenation of t_1 $(t_2 \in (t_1)^*)$
  - $\mathsf{star}([1, 2], [\,])$ holds
  - $\mathsf{star}([1, 2], [1, 2])$ holds, since $\mathsf{app}([1, 2], [\,], [1, 2]), \mathsf{star}([1, 2], [\,])$ hold
  - $\mathsf{star}([1, 2], [1, 2, 1, 2])$ holds, etc.

- cut in clause (2) needed for *termination*. Otherwise:

  $\mathsf{star}([\,], t)$ would lead to

  $\mathsf{app}([\,], YS, t), \mathsf{star}([\,], YS)$ would lead to

  $\mathsf{star}([\,], t)$

$$
\begin{aligned}
\mathsf{star}(XS, [\,]) &\text{ :- } !. && (1)\\
\mathsf{star}([\,], ZS) &\text{ :- } !, \mathsf{eq}(ZS, [\,]). && (2)\\
\mathsf{star}(XS, ZS) &\text{ :- } \mathsf{app}(XS, YS, ZS), \mathsf{star}(XS, YS). && (3)\\
\mathsf{app}([\,], YS, YS). & && (4)\\
\mathsf{app}([X \mid XS], YS, [X \mid ZS]) &\text{ :- } \mathsf{app}(XS, YS, ZS). && (5)\\
\mathsf{eq}(X, X). & && (6)
\end{aligned}
$$

- **state:** $(G_1 \mid \ldots \mid G_n)$ with current goal G_1 and next goals $G_2, \ldots, G_n$
- **goal:** $(t_1, \ldots, t_k)$ query or
  $(t_1, \ldots, t_k)^c$ query labeled by clause c used for next resolution
- **inference rules:**
  - Case
  - Eval
  - Back
  - Cut
  - Suc

$$
\begin{array}{rl}
\mathsf{star}([1,2],[\,]) & \vdash_{\textsc{Case}}\\
\mathsf{star}([1,2],[\,])^{(1)} \mid \mathsf{star}([1,2],[\,])^{(2)} \mid \mathsf{star}([1,2],[\,])^{(3)} & \vdash_{\textsc{Eval}}\\
!_1 \mid \mathsf{star}([1,2],[\,])^{(2)} \mid \mathsf{star}([1,2],[\,])^{(3)} & \vdash_{\textsc{Cut}}\\
\square & \vdash_{\textsc{Suc}}\\
\varepsilon &
\end{array}
$$

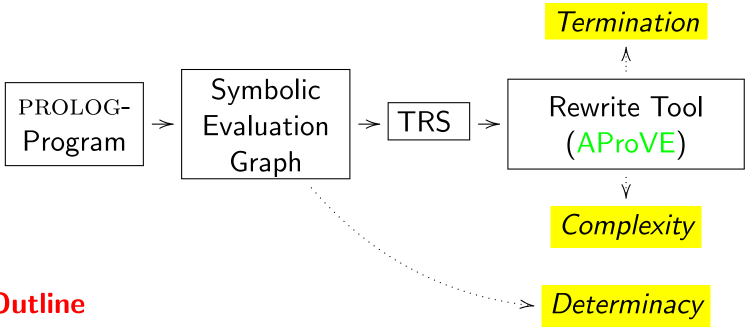
$$
\begin{aligned}
\mathsf{star}(XS, []) &\ \text{:-}\ !. && (1)\\
\mathsf{star}([], ZS) &\ \text{:-}\ !, \mathsf{eq}(ZS, []). && (2)\\
\mathsf{star}(XS, ZS) &\ \text{:-}\ \mathsf{app}(XS, YS, ZS), \mathsf{star}(XS, YS). && (3)\\
\mathsf{app}([], YS, YS). & && (4)\\
\mathsf{app}([X \mid XS], YS, [X \mid ZS]) &\ \text{:-}\ \mathsf{app}(XS, YS, ZS). && (5)\\
\mathsf{eq}(X, X). & && (6)
\end{aligned}
$$

- **state:** $(G_1 \mid \ldots \mid G_n)$ with current goal G_1 and next goals $G_2, \ldots, G_n$
- *linear* semantics, since state contains all backtracking information
  $\Rightarrow$ evaluation is a sequence of states, not a search tree
- suitable for extension to abstract states

$$
\begin{aligned}
\mathsf{star}([1,2],[]) &\quad \vdash_{\textsc{Case}}\\
\mathsf{star}([1,2],[])^{(1)} \mid \mathsf{star}([1,2],[])^{(2)} \mid \mathsf{star}([1,2],[])^{(3)} &\quad \vdash_{\textsc{Eval}}\\
!_1 \mid \mathsf{star}([1,2],[])^{(2)} \mid \mathsf{star}([1,2],[])^{(3)} &\quad \vdash_{\textsc{Cut}}\\
\Box &\quad \vdash_{\textsc{Suc}}\\
\varepsilon &
\end{aligned}
$$

Symbolic Evaluation Graphs and Term Rewriting

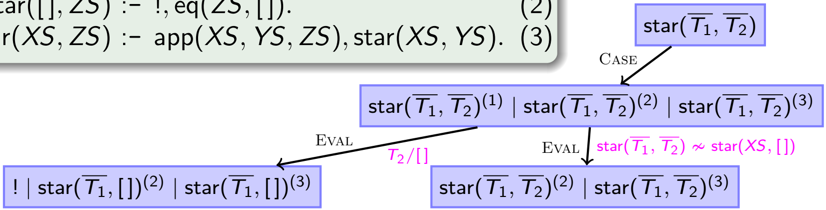
General methodology for analyzing PROLOG programs



Outline

- linear operational semantics of PROLOG
- from PROLOG to symbolic evaluation graphs
- from symbolic evaluation graphs to TRSs for termination analysis
- from symbolic evaluation graphs to TRSs for complexity analysis
- determinacy analysis

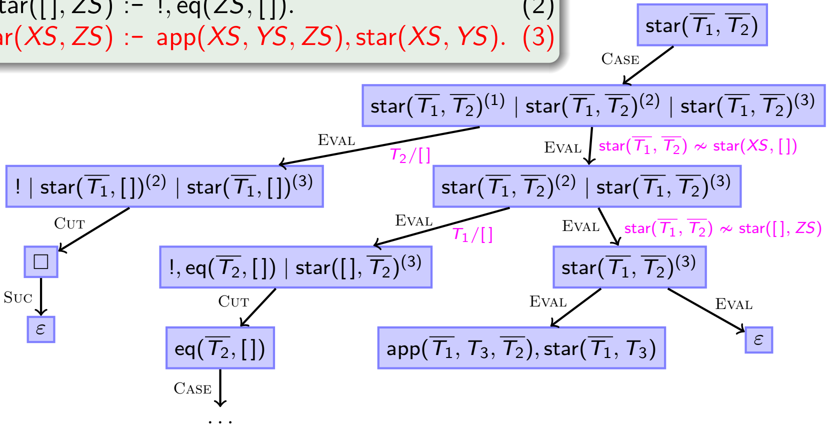
```
star(XS, []) :- !.                              (1)
star([], ZS) :- !, eq(ZS, []).                  (2)
star(XS, ZS) :- app(XS, YS, ZS), star(XS, YS).  (3)
```



- **symbolic evaluation graph:** all evaluations for a *class* of queries
- **class of queries** $\mathcal{Q}_m^p$ described by *predicate p* and *moding m*

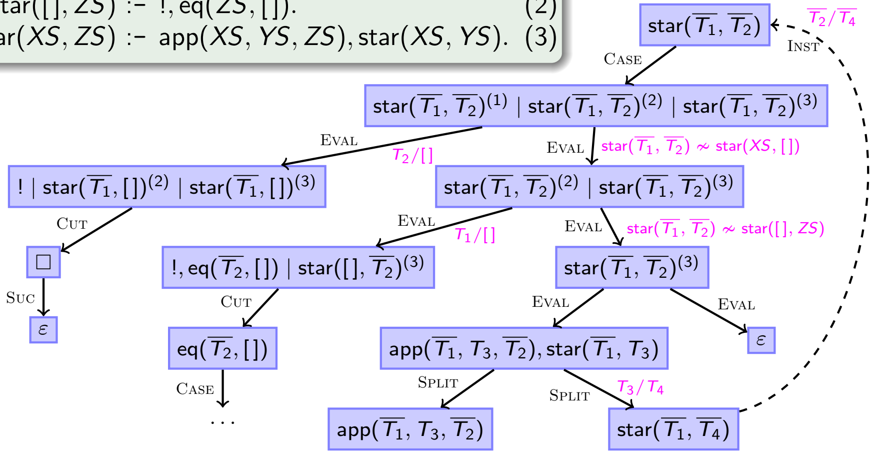
  Example: $\mathcal{Q}_m^{\text{star}} = \{\text{star}(t_1, t_2) \mid t_1, t_2 \text{ are ground}\}$.
- **abstract state:** stands for *set* of concrete states
  - state with *abstract* variables $T_1, T_2, \ldots$ representing arbitrary terms
  - constraints on the terms represented by $T_1, T_2, \ldots$
    - groundness constraints: $\overline{T_1}$, $\overline{T_2}$
    - unification constraints: star($\overline{T_1}$, $\overline{T_2}$) $\nsim$ star(XS, [])

```
star(XS, [])  :- !.                              (1)
star([], ZS)  :- !, eq(ZS, []).                  (2)
star(XS, ZS)  :- app(XS, YS, ZS), star(XS, YS).  (3)
```



- **abstract state:** stands for *set* of concrete states
  - state with *abstract* variables $T_1, T_2, \ldots$ representing arbitrary terms
  - constraints on the terms represented by $T_1, T_2, \ldots$
    - groundness constraints: $\overline{T_1}$, $\overline{T_2}$
    - unification constraints: star($\overline{T_1}$, $\overline{T_2}$) $\nsim$ star(XS, [])

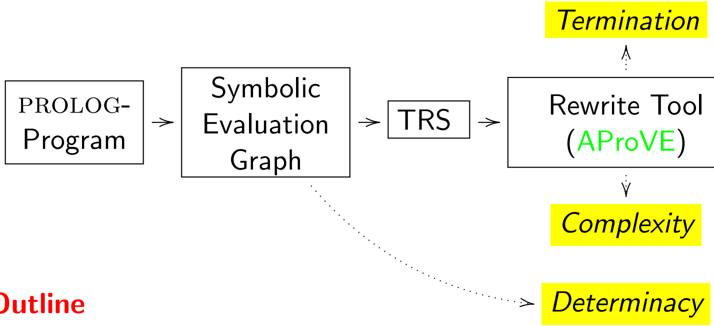
```
star(XS, []) :- !.                          (1)
star([], ZS) :- !, eq(ZS, []).              (2)
star(XS, ZS) :- app(XS, YS, ZS), star(XS, YS).   (3)
```



- INST: connection to previous state if current state is an *instance*
- SPLIT: split away first atom from a query
  - fresh variables in SPLIT's second successor
  - approximate first atom's answer substitution by *groundness analysis*

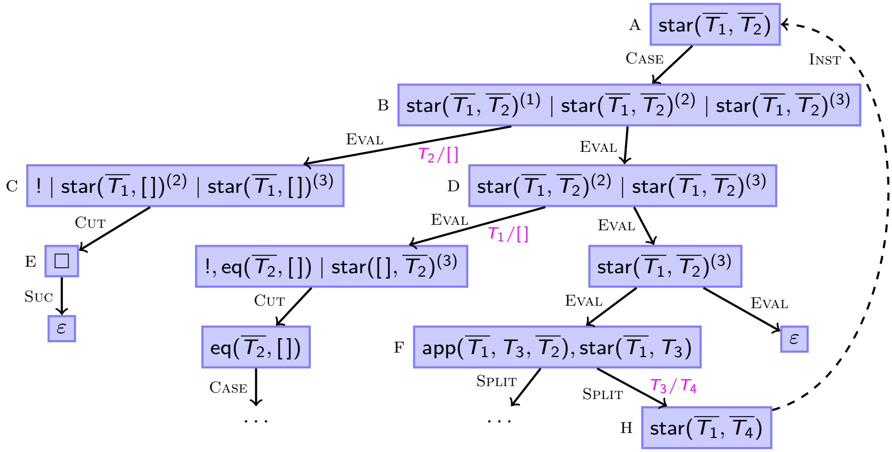
Symbolic Evaluation Graphs and Term Rewriting

General methodology for analyzing PROLOG programs



Outline

- linear operational semantics of PROLOG
- from PROLOG to symbolic evaluation graphs
- from symbolic evaluation graphs to TRSs for termination analysis
- from symbolic evaluation graphs to TRSs for complexity analysis
- determinacy analysis



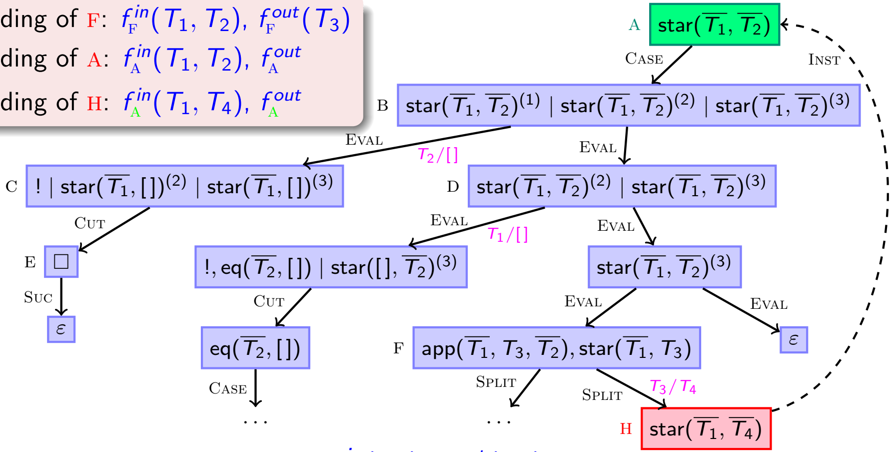
- **Aim:** show termination of concrete states represented by graph
- **Solution:** synthesize TRS from the graph
  - TRS captures all evaluations that are crucial for termination behavior
  - existing rewrite tools can show termination of TRS

    ⇒ prove termination of original PROLOG program

Encoding of F: $f_F^{in}(T_1, T_2),\ f_F^{out}(T_3)$

Encoding of A: $f_A^{in}(T_1, T_2),\ f_A^{out}$

Encoding of H: $f_A^{in}(T_1, T_4),\ f_A^{out}$

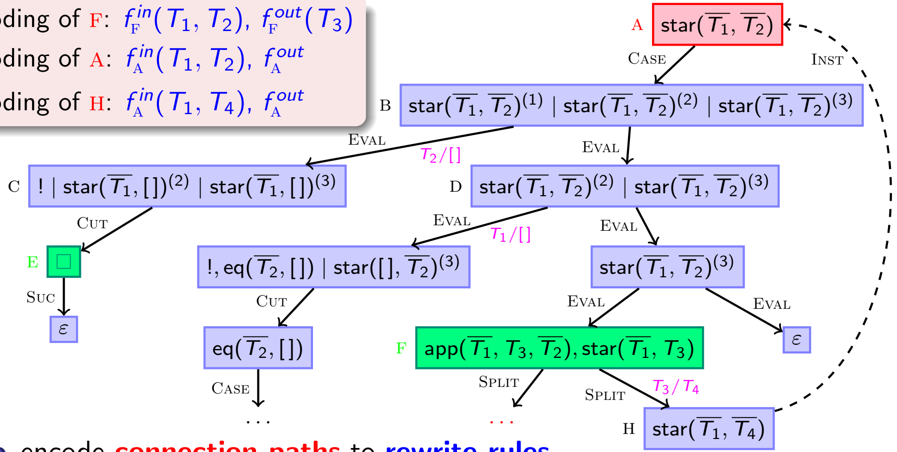


- encode **state** s to **terms** $f_s^{in}(\ldots),\ f_s^{out}(\ldots)$
  - arguments of f_s^{in}: abstract ground variables of s $(\overline{T_1}, \overline{T_2}, \ldots)$
  - arguments of f_s^{out}: remaining abstract variables of s which are made ground by every answer substitution of s (*groundness analysis*)
- for state s with Inst edge to s': use $f_{s'}^{in},\ f_{s'}^{out}$ instead of $f_s^{in},\ f_s^{out}$

Encoding of F: $f_F^{in}(T_1, T_2),\ f_F^{out}(T_3)$

Encoding of A: $f_A^{in}(T_1, T_2),\ f_A^{out}$

Encoding of H: $f_A^{in}(T_1, T_4),\ f_A^{out}$

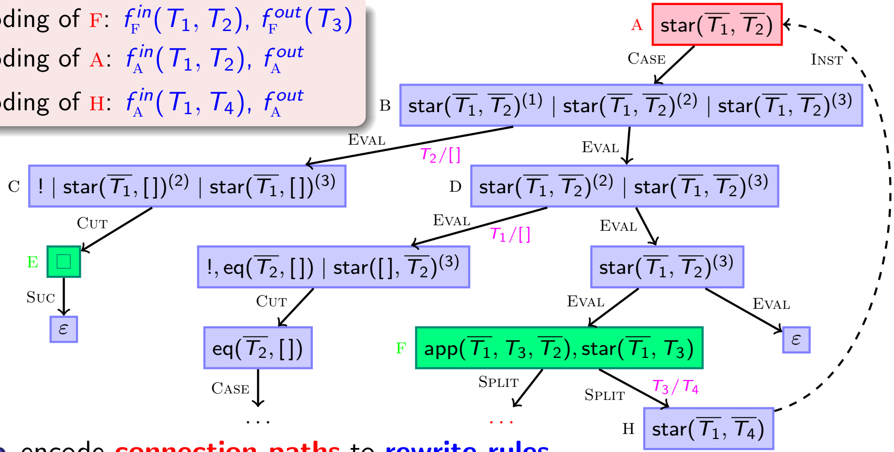


- encode **connection paths** to **rewrite rules**
- **connection path**:
  - start state = root, successor of Inst, or successor of Split but no Inst or Split node itself
  - end state = Inst, Split, Suc node, or successor of Inst node
  - connection path may not traverse end nodes except Suc nodes

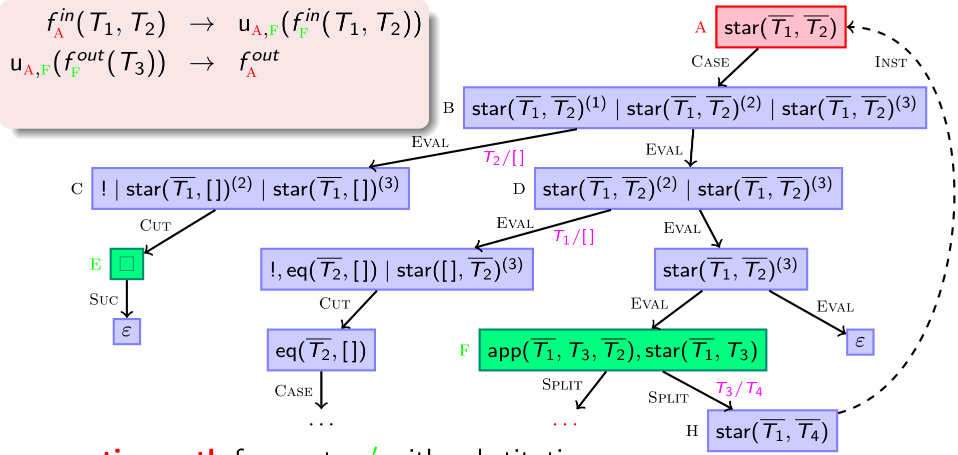
Encoding of F: $f_{\mathrm{F}}^{in}(T_1, T_2),\ f_{\mathrm{F}}^{out}(T_3)$

Encoding of A: $f_{\mathrm{A}}^{in}(T_1, T_2),\ f_{\mathrm{A}}^{out}$

Encoding of H: $f_{\mathrm{A}}^{in}(T_1, T_4),\ f_{\mathrm{A}}^{out}$



- encode **connection paths** to **rewrite rules**
- **connection path**: cover all ways through graph except
  - Inst edges (are covered by the encoding of terms)
  - Split edges (will be covered by extra Split rules later)
  - parts without cycles or Suc nodes (irrelevant for termination behavior)



connection path from s to s' with substitution σ:

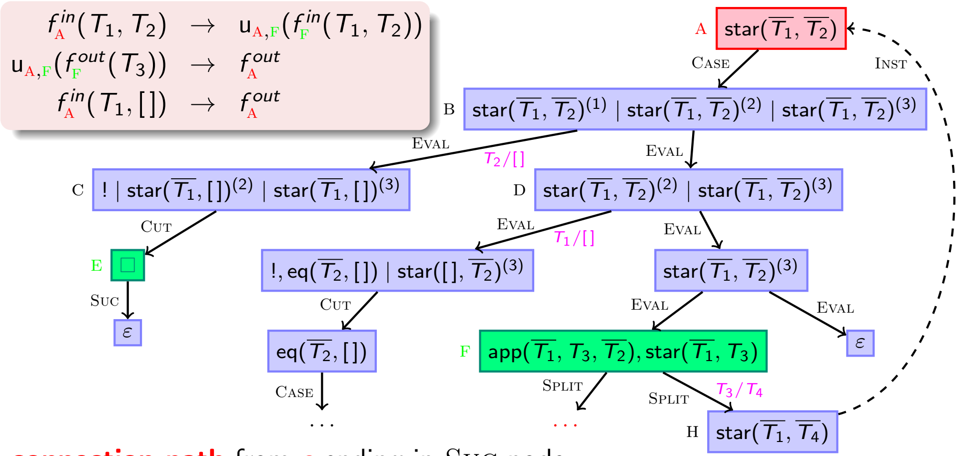
$f^{in}_{s}(\ldots)\sigma$ evaluates to $f^{out}_{s}(\ldots)\sigma$ if
$f^{in}_{s'}(\ldots)$ evaluates to $f^{out}_{s'}(\ldots)$

$f^{in}_{A}(T_1, T_2)$ evaluates to f^{out}_{A} if
$f^{in}_{F}(T_1, T_2)$ evaluates to $f^{out}_{F}(T_3)$

rewrite rules:

$$f^{in}_{s}(\ldots)\sigma \rightarrow u_{s,s'}(f^{in}_{s'}(\ldots))$$
$$u_{s,s'}(f^{out}_{s'}(\ldots)) \rightarrow f^{out}_{s}(\ldots)\sigma$$

$$f^{in}_{A}(T_1, T_2) \rightarrow u_{A,F}(f^{in}_{F}(T_1, T_2))$$
$$u_{A,F}(f^{out}_{F}(T_3)) \rightarrow f^{out}_{A}$$



connection path from s ending in Suc node:

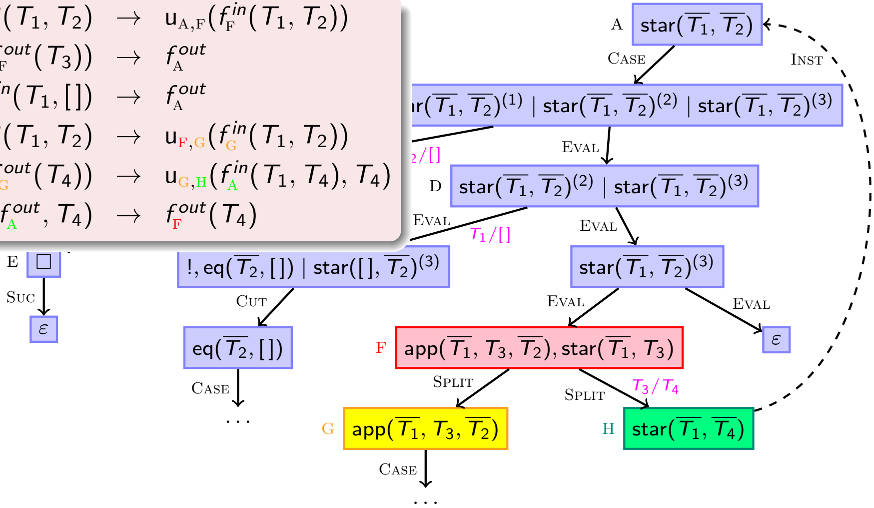
$f_s^{in}(\ldots)\sigma$ evaluates to $f_s^{out}(\ldots)\sigma$ $\qquad$ $f_{\mathrm{A}}^{in}(T_1, [\,])$ evaluates to f_{A}^{out}

intuition:

$f_{\mathrm{A}}^{in}(T_1, T_2)$ evaluates to f_{A}^{out} $\qquad$ if $T_2 \in (T_1)^*$

$f_{\mathrm{F}}^{in}(T_1, T_2)$ evaluates to $f_{\mathrm{F}}^{out}(T_3)$ $\qquad$ if $T_1 \neq [\,]$, $T_2 \neq [\,]$, T_3 is T_2 without prefix T_1, $T_3 \in (T_1)^*$

$$
\begin{aligned}
f^{in}_{A}(T_1, T_2) &\rightarrow u_{A,F}(f^{in}_{F}(T_1, T_2)) \\
u_{A,F}(f^{out}_{F}(T_3)) &\rightarrow f^{out}_{A} \\
f^{in}_{A}(T_1, []) &\rightarrow f^{out}_{A} \\
f^{in}_{F}(T_1, T_2) &\rightarrow u_{F,G}(f^{in}_{G}(T_1, T_2)) \\
u_{F,G}(f^{out}_{G}(T_4)) &\rightarrow u_{G,H}(f^{in}_{A}(T_1, T_4), T_4) \\
u_{G,H}(f^{out}_{A}, T_4) &\rightarrow f^{out}_{F}(T_4)
\end{aligned}
$$



Split node s with successors s_1 and s_1:

$f^{in}_{s}(\ldots)\sigma$ evaluates to $f^{out}_{s}(\ldots)\sigma$ if

$f^{in}_{s_1}(\ldots)\sigma$ evaluates to $f^{out}_{s_1}(\ldots)\sigma$ and

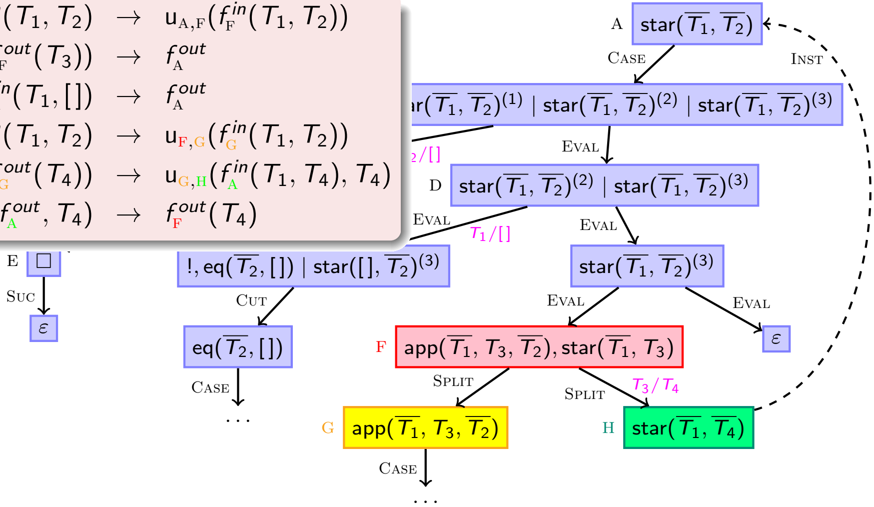
$f^{in}_{s_2}(\ldots)$ evaluates to $f^{out}_{s_2}(\ldots)$

$f^{in}_{F}(T_1, T_2)$ evaluates to $f^{out}_{F}(T_4)$ if

$f^{in}_{G}(T_1, T_2)$ evaluates to $f^{out}_{G}(T_4)$ and

$f^{in}_{A}(T_1, T_4)$ evaluates to f^{out}_{A}

$$
\begin{aligned}
f^{in}_{A}(T_1, T_2) &\rightarrow u_{A,F}(f^{in}_{F}(T_1, T_2))\\
u_{A,F}(f^{out}_{F}(T_3)) &\rightarrow f^{out}_{A}\\
f^{in}_{A}(T_1, []) &\rightarrow f^{out}_{A}\\
f^{in}_{F}(T_1, T_2) &\rightarrow u_{F,G}(f^{in}_{G}(T_1, T_2))\\
u_{F,G}(f^{out}_{G}(T_4)) &\rightarrow u_{G,H}(f^{in}_{A}(T_1, T_4), T_4)\\
u_{G,H}(f^{out}_{A}, T_4) &\rightarrow f^{out}_{F}(T_4)
\end{aligned}
$$



intuition:

$f^{in}_{F}(T_1, T_2)$ evaluates to $f^{out}_{F}(T_4)$ if $T_1 \neq []$, $T_2 \neq []$, T_4 is T_2 without prefix T_1, $T_4 \in (T_1)^*$

$f^{in}_{G}(T_1, T_2)$ evaluates to $f^{out}_{G}(T_4)$ if $T_1 \neq []$, $T_2 \neq []$, T_4 is T_2 without prefix T_1

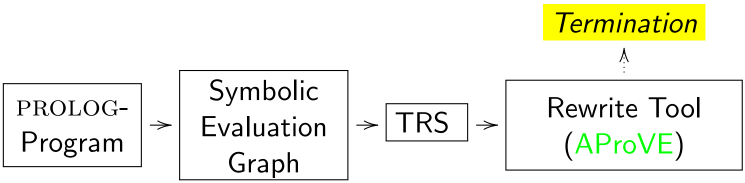
$f^{in}_{A}(T_1, T_4)$ evaluates to f^{out}_{A} if $T_4 \in (T_1)^*$

$$
\begin{aligned}
\mathsf{star}(XS, [\,]) &\text{ :- } !. \\
\mathsf{star}([\,], ZS) &\text{ :- } !, \mathsf{eq}(ZS, [\,]). \\
\mathsf{star}(XS, ZS) &\text{ :- } \mathsf{app}(XS, YS, ZS), \mathsf{star}(XS, YS). \\
\mathsf{app}([\,], YS, YS)&. \\
\mathsf{app}([X \mid XS], YS, [X \mid ZS]) &\text{ :- } \mathsf{app}(XS, YS, ZS). \\
\mathsf{eq}(X, X)&.
\end{aligned}
$$

$$
\begin{aligned}
f^{in}_{A}(T_1, T_2) &\rightarrow u_{A,F}(f^{in}_{F}(T_1, T_2)) \\
u_{A,F}(f^{out}_{F}(T_3)) &\rightarrow f^{out}_{A} \\
f^{in}_{A}(T_1, [\,]) &\rightarrow f^{out}_{A} \\
f^{in}_{F}(T_1, T_2) &\rightarrow u_{F,G}(f^{in}_{G}(T_1, T_2)) \\
u_{F,G}(f^{out}_{G}(T_4)) &\rightarrow u_{G,H}(f^{in}_{A}(T_1, T_4), T_4) \\
u_{G,H}(f^{out}_{A}, T_4) &\rightarrow f^{out}_{F}(T_4) \\
f^{in}_{G}([T_5 \mid T_6], [T_5 \mid T_7]) &\rightarrow u_{G,I}(f^{in}_{I}(T_6, T_7)) \\
u_{G,I}(f^{out}_{I}(T_3)) &\rightarrow f^{out}_{G}(T_3) \\
f^{in}_{I}([T_8 \mid T_9], [T_8 \mid T_{10}]) &\rightarrow u_{I,K}(f^{in}_{I}(T_9, T_{10})) \\
u_{I,K}(f^{out}_{I}(T_3)) &\rightarrow f^{out}_{I}(T_3) \\
f^{in}_{I}([\,], T_3) &\rightarrow f^{out}_{I}(T_3)
\end{aligned}
$$

- existing TRS tools prove termination automatically
- original PROLOG program terminates

Symbolic Evaluation Graphs and Term Rewriting

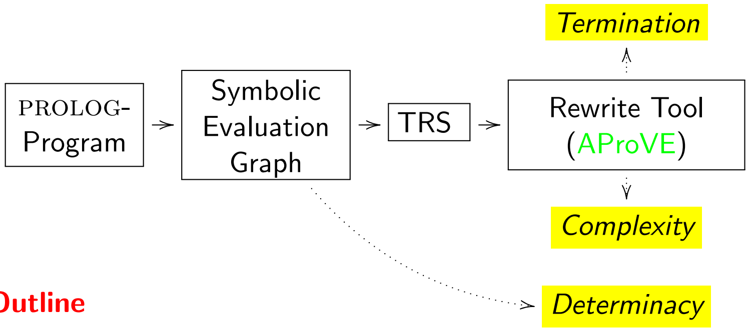


implemented in tool **AProVE**

- most powerful tool for termination of definite logic programs
- only tool for termination of non-definite PROLOG programs
- winner of *termination competition* for PROLOG
  (proves 342 of 477 examples, average runtime 6.5 s per example)

Symbolic Evaluation Graphs and Term Rewriting

General methodology for analyzing PROLOG programs



Outline

- linear operational semantics of PROLOG
- from PROLOG to symbolic evaluation graphs
- from symbolic evaluation graphs to TRSs for termination analysis
- from symbolic evaluation graphs to TRSs for complexity analysis
- determinacy analysis

Complexity for Logic Programs

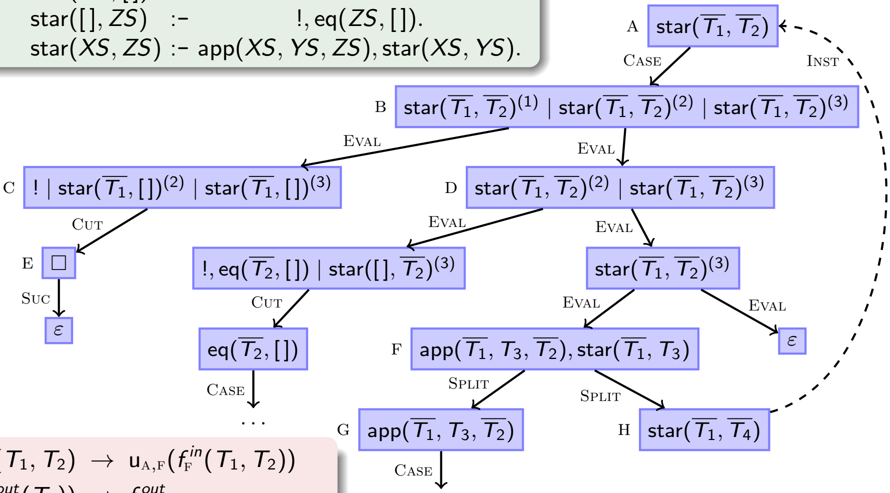
Program $\mathcal{P}$, Class of queries $\mathcal{Q}_m^{\mathsf{p}}$

- $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{p}}}$ maps $n \in \mathbb{N}$ to longest evaluation starting with $Q \in \mathcal{Q}_m^{\mathsf{p}}$, where $|Q|_m \leq n$
- $|Q|_m$: number of variables and function symbols on *input positions*
- corresponds to number of unification attempts

- $\mathcal{R}$ has linear complexity for class $\mathcal{Q}_m^{\mathsf{p}}$ if $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{p}}}(n) \in \mathcal{O}(n)$
  $\mathcal{R}$ has quadratic complexity for class $\mathcal{Q}_m^{\mathsf{p}}$ if $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{p}}}(n) \in \mathcal{O}(n^2)$ etc.
- Example (star-program): has linear complexity
- Goal: Re-use existing methodology for termination analysis to analyze complexity as well

$\mathcal{P}$:

```
star(XS, [])  :-                          !.
star([], ZS)  :-               !, eq(ZS, []).
star(XS, ZS)  :- app(XS, YS, ZS), star(XS, YS).
```



$$
\begin{aligned}
f_{\mathrm{A}}^{in}(T_1, T_2) &\rightarrow \mathsf{u}_{\mathrm{A,F}}(f_{\mathrm{F}}^{in}(T_1, T_2))\\
\mathsf{u}_{\mathrm{A,F}}(f_{\mathrm{F}}^{out}(T_3)) &\rightarrow f_{\mathrm{A}}^{out}\\
f_{\mathrm{A}}^{in}(T_1, []) &\rightarrow f_{\mathrm{A}}^{out}\\
f_{\mathrm{F}}^{in}(T_1, T_2) &\rightarrow \mathsf{u}_{\mathrm{F,G}}(f_{\mathrm{G}}^{in}(T_1, T_2))\\
\mathsf{u}_{\mathrm{F,G}}(f_{\mathrm{G}}^{out}(T_4)) &\rightarrow \mathsf{u}_{\mathrm{G,H}}(f_{\mathrm{A}}^{in}(T_1, T_4), T_4)\\
\mathsf{u}_{\mathrm{G,H}}(f_{\mathrm{A}}^{out}, T_4) &\rightarrow f_{\mathrm{F}}^{out}(T_4)
\end{aligned}
$$

- generate symbolic evaluation graph
- generate TRS from graph
- determine complexity of TRS by existing tool

$\mathcal{P}$:
$$\begin{aligned} &\mathsf{star}(XS, []) \;\text{:-}\; !.\\ &\mathsf{star}([], ZS) \;\text{:-}\; !, \mathsf{eq}(ZS, []).\\ &\mathsf{star}(XS, ZS) \;\text{:-}\; \mathsf{app}(XS, YS, ZS), \mathsf{star}(XS, YS).\end{aligned}$$

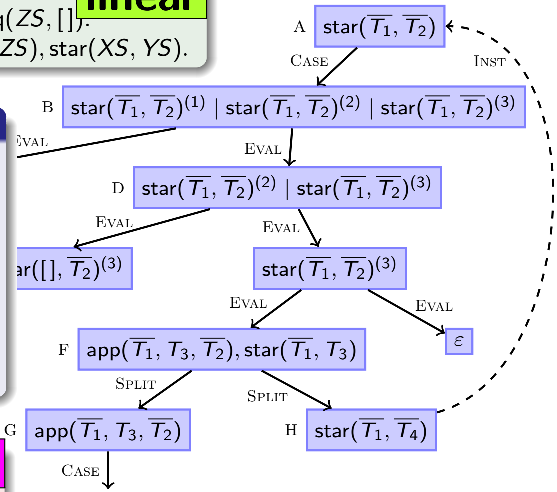
linear

Correct!

- depends on SPLIT's successor G
- in $\mathcal{P}$: repeat evaluation of H for every answer of G (*backtracking*)
- in TRS: evaluate H once (choose G's answer *non-deterministically*)
- Here: G is *deterministic* (has only one answer)

$$\begin{aligned} f^{in}_{A}(T_1, T_2) &\to u_{A,F}(f^{in}_{F}(\ldots\\ u_{A,F}(f^{out}_{F}(T_3)) &\to f^{out}_{A}\\ f^{in}_{A}(T_1, []) &\to f^{out}_{A}\\ f^{in}_{F}(T_1, T_2) &\to u_{F,G}(f^{in}_{G}(T_1, T_2))\\ u_{F,G}(f^{out}_{G}(T_4)) &\to u_{G,H}(f^{in}_{A}(T_1, T_4), T_4)\\ u_{G,H}(f^{out}_{A}, T_4) &\to f^{out}_{F}(T_4)\end{aligned}$$

linear



- generate symbolic evaluation graph
- generate TRS from graph
- determine complexity of TRS by existing tool
- infer that $\mathcal{P}$ has the same complexity

$$\mathcal{P}: \quad \text{sublist}(X, Y) \text{ :- } \text{app}(P, U, Y),\ \text{app}(V, X, P). \quad (1)$$
$$\text{app}([\,], YS, YS). \quad (2)$$
$$\text{app}([X \mid XS], YS, [X \mid ZS]) \text{ :- } \text{app}(XS, YS, ZS) \quad (3)$$

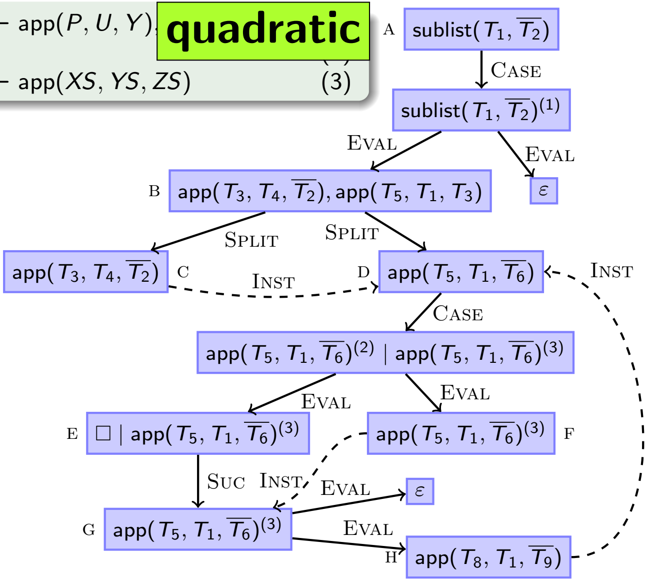
Evaluation of sublist

- $\mathcal{Q}_m^{\text{sublist}} = \{\text{sublist}(t_1, t_2) \mid t_2 \text{ ground}\}$
- computes all sublists of Y (by *backtracking*)
- $\mathcal{P}$:
  - linear many possibilities to split Y into P and U
  - for each possible P, linear evaluation of app(V, X, P)

$\mathcal{P}$: sublist(X, Y) :- app(P, U, Y), **quadratic**

app$([\,], YS, YS)$.

app$([X \,|\, XS], YS, [X \,|\, ZS])$:- app(XS, YS, ZS) (3)



$f^{in}_{\mathrm{B}}(T_2) \;\to\;$ u **linear**

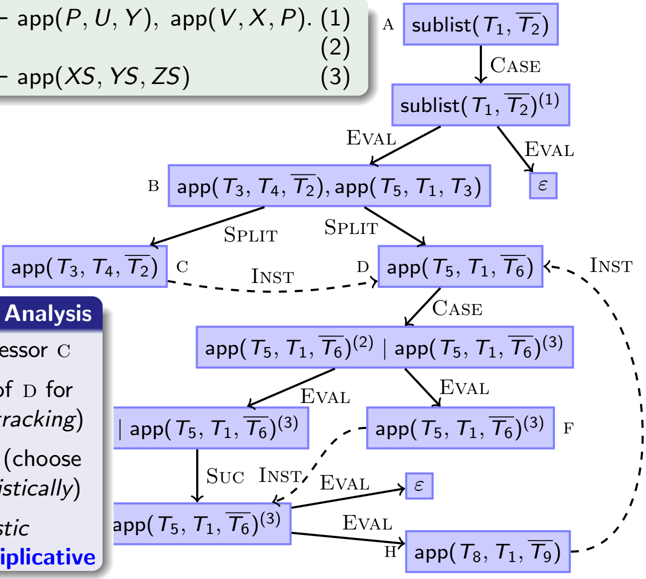
$\mathrm{u}_{\mathrm{B,C}}(f^{out}_{\mathrm{D}}(\ldots)) \;\to\; \mathrm{u}_{\mathrm{C,D}}(f^{in}_{\mathrm{D}}(\ldots))$

$\mathrm{u}_{\mathrm{C,D}}(f^{out}_{\mathrm{D}}(\ldots)) \;\to\; f^{out}_{\mathrm{B}}(\ldots)$

- generate symbolic evaluation graph and TRS
- determine complexity of TRS by existing tool
- ~~infer that $\mathcal{P}$ has the same complexity~~

$\mathcal{P}$:

$$\begin{array}{lr} \text{sublist}(X, Y) \text{ :- } \text{app}(P, U, Y),\ \text{app}(V, X, P). & (1) \\ \text{app}([\,], YS, YS). & (2) \\ \text{app}([X \mid XS], YS, [X \mid ZS]) \text{ :- } \text{app}(XS, YS, ZS) & (3) \end{array}$$



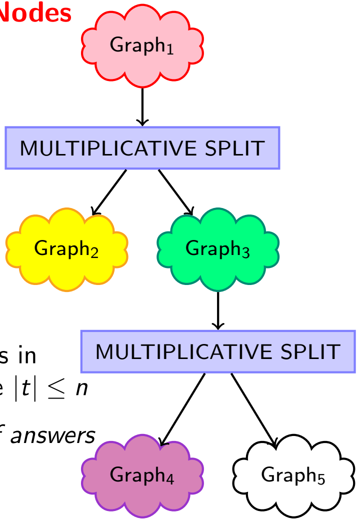
Correctness of Complexity Analysis

- depends on SPLIT's successor C
- in $\mathcal{P}$: repeat evaluation of D for every answer of C (*backtracking*)
- in TRS: evaluate D once (choose C's answer *non-deterministically*)
- Here: C is *not deterministic*
  ⇒ SPLIT node B is **multiplicative**

$$\begin{array}{rcl} f_{\text{B}}^{in}(T_2) & \to & \text{u}_{\text{B,C}}(f_{\text{D}}^{in}(T_2)) \\ \text{u}_{\text{B,C}}(f_{\text{D}}^{out}(\dots)) & \to & \text{u}_{\text{C,D}}(f_{\text{D}}^{in}(\dots)) \\ \text{u}_{\text{C,D}}(f_{\text{D}}^{out}(\dots)) & \to & f_{\text{B}}^{out}(\dots) \end{array}$$

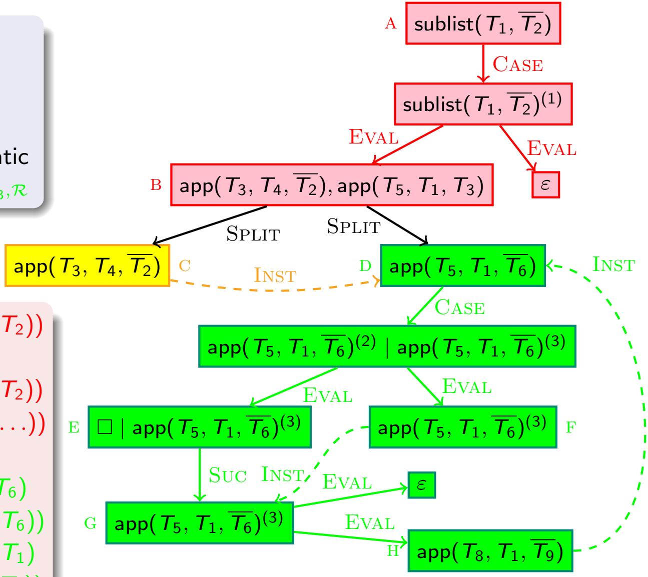
- generate symbolic evaluation graph and TRS
- determine complexity of TRS by existing tool
- ~~infer that $\mathcal{P}$ has the same complexity~~

Decompose Graph by Multiplicative Split Nodes



- generate symbolic evaluation graph
- generate separate TRSs $\mathcal{R}_1, \ldots, \mathcal{R}_5$ for parts up to multiplicative SPLIT nodes (no multiplicative SPLIT node may reach itself)
- determine $irc_{\mathcal{R}_1,\mathcal{R}}$, ..., $irc_{\mathcal{R}_5,\mathcal{R}}$ separately
  - maps $n \in \mathbb{N}$ to maximal number of $\mathcal{R}_i$-steps in evaluation starting with basic term t, where $|t| \leq n$
  - upper bound for *runtime* and for *number of answers*
- combine complexities
  - multiply complexities for children of multiplicative SPLITs
  - add complexities of parents of multiplicative SPLITs
  - $irc_{\mathcal{R}_1,\mathcal{R}} + irc_{\mathcal{R}_2,\mathcal{R}} \cdot (irc_{\mathcal{R}_3,\mathcal{R}} + irc_{\mathcal{R}_4,\mathcal{R}} \cdot irc_{\mathcal{R}_5,\mathcal{R}})$

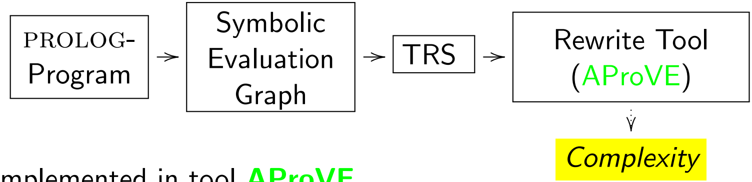
- $irc_{\mathcal{R}_1,\mathcal{R}}$: constant
- $irc_{\mathcal{R}_2,\mathcal{R}}$: linear
- $irc_{\mathcal{R}_3,\mathcal{R}}$: linear
- complexity of $\mathcal{P}$: quadratic
  $irc_{\mathcal{R}_1,\mathcal{R}} + irc_{\mathcal{R}_2,\mathcal{R}} \cdot irc_{\mathcal{R}_3,\mathcal{R}}$



$$
\begin{aligned}
f^{in}_{\mathrm{A}}(T_2) &\to \mathrm{u}_{\mathrm{A,B}}(f^{in}_{\mathrm{B}}(T_2))\\
\mathrm{u}_{\mathrm{A,B}}(f^{out}_{\mathrm{B}}(\ldots)) &\to f^{out}_{\mathrm{A}}(T_1)\\
f^{in}_{\mathrm{B}}(T_2) &\to \mathrm{u}_{\mathrm{B,C}}(f^{in}_{\mathrm{D}}(T_2))\\
\mathrm{u}_{\mathrm{B,C}}(f^{out}_{\mathrm{D}}(\ldots)) &\to \mathrm{u}_{\mathrm{C,D}}(f^{in}_{\mathrm{D}}(\ldots))\\
\mathrm{u}_{\mathrm{C,D}}(f^{out}_{\mathrm{D}}(\ldots)) &\to f^{out}_{\mathrm{B}}(\ldots)\\
f^{in}_{\mathrm{D}}(T_6) &\to f^{out}_{\mathrm{D}}([\,], T_6)\\
f^{in}_{\mathrm{D}}(T_6) &\to \mathrm{u}_{\mathrm{D,G}}(f^{in}_{\mathrm{G}}(T_6))\\
\mathrm{u}_{\mathrm{D,G}}(f^{out}_{\mathrm{G}}(\ldots)) &\to f^{out}_{\mathrm{D}}(T_5, T_1)\\
f^{in}_{\mathrm{D}}(T_6) &\to \mathrm{u}_{\mathrm{D,F}}(f^{in}_{\mathrm{G}}(T_6))\\
\mathrm{u}_{\mathrm{D,F}}(\ldots) &\to f^{out}_{\mathrm{D}}(T_5, T_1)\\
f^{in}_{\mathrm{G}}([T_7 \mid T_9]) &\to \mathrm{u}_{\mathrm{G,H}}(\ldots)\\
\mathrm{u}_{\mathrm{G,H}}(\ldots) &\to f^{out}_{\mathrm{G}}([T_7 \mid T_8])
\end{aligned}
$$

- generate graph and TRSs $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$
- determine $irc_{\mathcal{R}_1,\mathcal{R}}$, $irc_{\mathcal{R}_2,\mathcal{R}}$, $irc_{\mathcal{R}_3,\mathcal{R}}$
- infer complexity of $\mathcal{P}$

Symbolic Evaluation Graphs and Term Rewriting



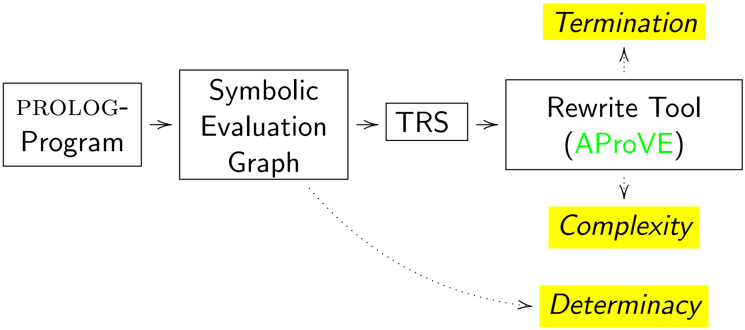
implemented in tool **AProVE**

- only tool for complexity of non-well-moded or non-definite programs
- experiments on all 477 programs of *TPDB*

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \cdot 2^n)$	**bounds**	**time**
CASLOG	1	21	4	**3**	29	14.8
CiaoPP	3	19	4	**3**	29	11.7
AProVE	**54**	**117**	**37**	0	**208**	**10.6**

Symbolic Evaluation Graphs and Term Rewriting

General methodology for analyzing PROLOG programs



Outline

- linear operational semantics of PROLOG
- from PROLOG to symbolic evaluation graphs
- from symbolic evaluation graphs to TRSs for termination analysis
- from symbolic evaluation graphs to TRSs for complexity analysis
- determinacy analysis

Criterion for determinacy of s

If s reaches SUC node s', then there is no path from s' to a SUC node.

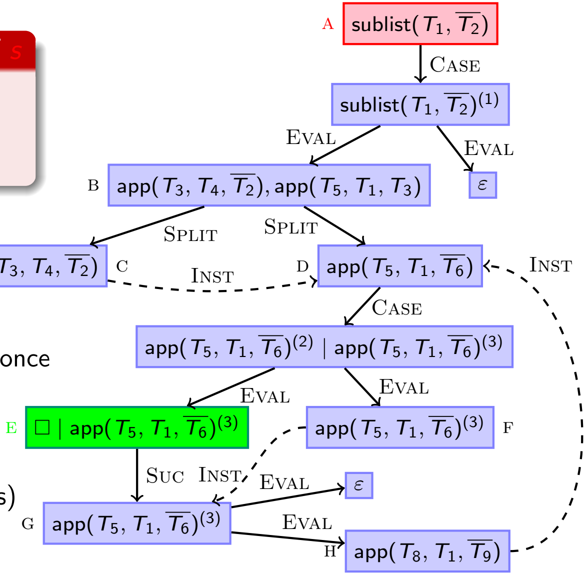
- query deterministic iff

  it generates at most one answer substitution at most once

  - for program analysis
  - for complexity analysis (non-multiplicative SPLITs)

- successful evaluation ⇒

  path to SUC node in symbolic evaluation graph

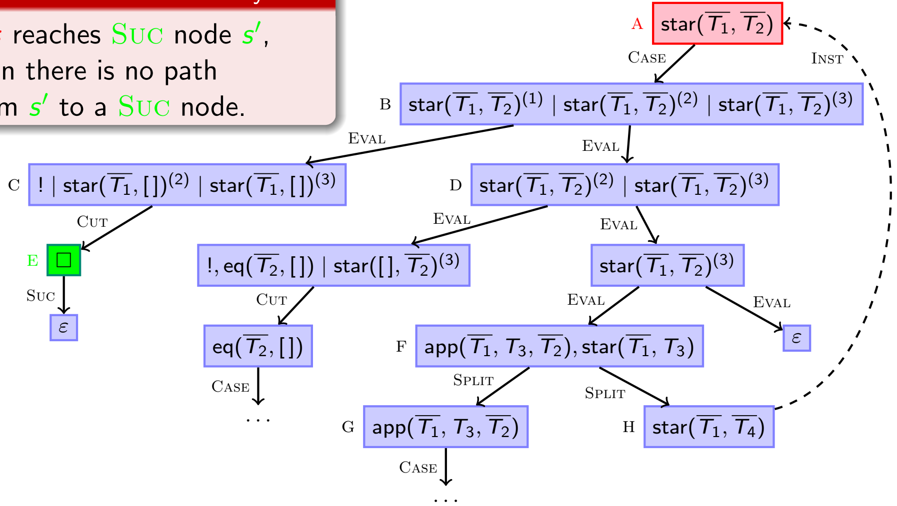


- C not deterministic

  ⇒ SPLIT node B multiplicative
- A not deterministic

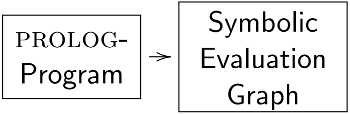
Criterion for determinacy of s

If s reaches Suc node s', then there is no path from s' to a Suc node.



- G is deterministic
  $\Rightarrow$ Split node F not multiplicative
- A is deterministic

Symbolic Evaluation Graphs and Term Rewriting



implemented in tool **AProVE**

- experiments on 300 definite programs:
  CiaoPP: 132, AProVE: 69
- experiments on 177 non-definite programs:
  CiaoPP: 61, AProVE: 92
- only first step, but substantial addition to existing determinacy analyses
  (AProVE succeeds on 78 examples where CiaoPP fails)
- strong enough for complexity analysis

Overview

I. Termination of Term Rewriting

1. Termination of Term Rewrite Systems
2. Non-Termination of Term Rewrite Systems
3. Complexity of Term Rewrite Systems
4. Termination of Integer Term Rewrite Systems

II. Termination of Programs

1. Termination of Functional Programs (Haskell)
2. Termination of Logic Programs (Prolog)
3. Termination of Imperative Programs (Java) (RTA '10 & '11, CAV '12)

Termination of Imperative Programs

Direct Approaches

- Synthesis of Linear Ranking Functions
  (Colon & Sipma, 01), (Podelski & Rybalchenko, 04), . . .
- Terminator: Termination Analysis by Abstraction & Model Checking
  (Cook, Podelski, Rybalchenko et al., since 05)
- Julia & COSTA: Termination Analysis of JAVA BYTECODE
  (Spoto, Mesnard, Payet, 10),
  (Albert, Arenas, Codish, Genaim, Puebla, Zanardini, 08)
- . . .

- used at Microsoft for verifying Windows device drivers
- no use of TRS-techniques (stand-alone methods)

Termination of Imperative Programs

Rewrite-Based Approach

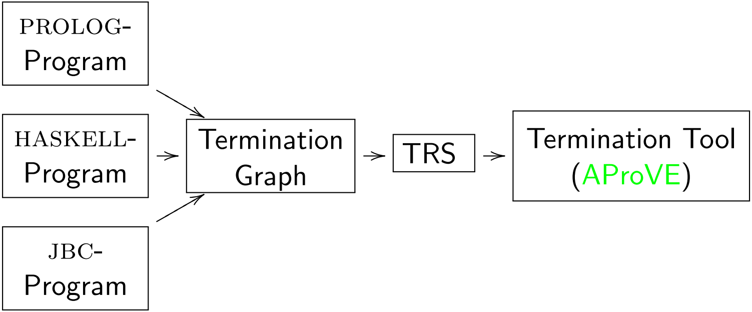
- analyze JAVA BYTECODE (JBC) instead of JAVA
- using TRS-techniques for JBC is challenging
  - sharing and aliasing
  - side effects
  - cyclic data objects
  - object-orientation
  - recursion
  - ...

Termination of Imperative Programs

- **New approach**
  - Frontend
    - evaluate JBC a few steps ⇒ termination graph

      termination graph captures side effects, sharing, cyclic data objects etc.
    - transform termination graph ⇒ TRS
  - Backend
    - prove termination of the resulting TRS
      (using existing techniques & tools)
- implemented in **AProVE**
  - successfully evaluated on JBC-collection
  - competitive termination tool for JBC

Termination of Imperative Programs



- implemented in **AProVE**
  - successfully evaluated on JBC-collection
  - competitive termination tool for JBC

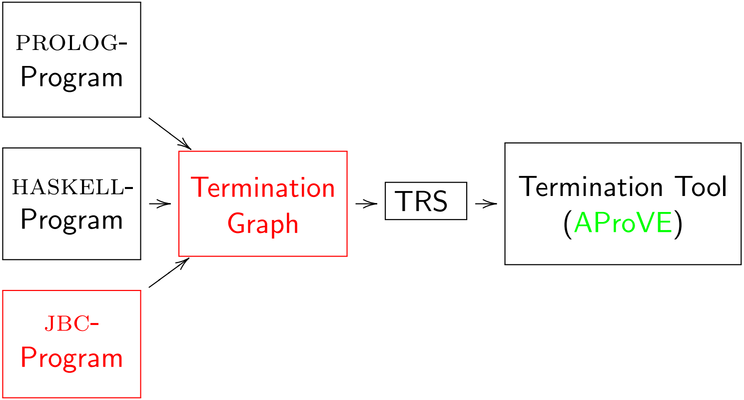
Termination of Imperative Programs

- other techniques:
  abstract objects to numbers
  - `IntList`-object representing $[0, 1, 2]$
    is abstracted to length 3

```
public class IntList {
  int value;
  IntList next;
}
```

- our technique:
  abstract objects to terms
  - introduce function symbol for every class
  - `IntList`-object representing $[0, 1, 2]$
    is abstracted to term: $\mathsf{IntList}(0,\ \mathsf{IntList}(1,\ \mathsf{IntList}(2, \mathsf{null})\,)\,)$
  - TRS-techniques generate suitable orders to compare arbitrary terms
  - particularly powerful on user-defined data types
  - powerful on pre-defined data types by using Integer TRSs

From JBC to Termination Graphs



Example

```
00: aload_0       // load num to opstack
01: ifnull 8      // jump to line 8 if top
                  // of opstack is null
04: aload_1       // load limit
05: ifnonnull 9   // jump if not null
08: return
09: aload_0       // load num
10: astore_2      // store into copy
11: aload_0       // load num
12: getfield val  // load field val
15: aload_1       // load limit
16: getfield val  // load field val
19: if_icmpge 35  // jump if
                  // num.val >= limit.val
22: aload_2       // load copy
23: aload_2       // load copy
24: getfield val  // load field val
27: iconst_1      // load constant 1
28: iadd          // add copy.val and 1
29: putfield val  // store into copy.val
32: goto 11
35: return
```

```
public class Int {
  // only wrap a primitive int
  private int val;

  // count up to the value
  // in "limit"
  public static void count(
      Int num, Int limit) {

    if (num == null
        || limit == null) {
        return;
    }

    // introduce sharing
    Int copy = num;

    while (num.val < limit.val) {
        copy.val++;
    }
  }
}
```

Abstract States of the JVM

```
00: aload_0        // load num to opstack
01: ifnull 8       // jump to line 8 if top
                   // of opstack is null
04: aload_1        // load limit
05: ifnonnull 9    // jump if not null
08: return
09: aload_0        // load num
10: astore_2       // store into copy
11: aload_0        // load num
12: getfield val   // load field val
15: aload_1        // load limit
16: getfield val   // load field val
19: if_icmpge 35   // jump if
                   // num.val >= limit.val
22: aload_2        // load copy
23: aload_2        // load copy
24: getfield val   // load field val
27: iconst_1       // load constant 1
28: iadd           // add copy.val and 1
29: putfield val   // store into copy.val
32: goto 11
35: return
```

$\texttt{ifnull 8} \mid \texttt{n}: o_1, \texttt{l}: o_2 \mid o_1$

$o_1 = \texttt{Int}(\texttt{val} = i_1) \quad i_1 = (-\infty, \infty)$

$o_2 = \texttt{Int}(?)$

4 components

1. next program instruction
2. values of local variables (value of `num` is *reference* o_1)
3. values on the operand stack
4. information about the heap
   - object at address o_2 is `null` or of type `Int`
   - object at o_1 has type `Int`, `val`-field has value i_1
   - i_1 is an arbitrary integer
   - no sharing

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     ⋮
19: if_icmpge 35
     ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

aload_0 | n : o_1 , l : o_2 | ε
o_1 = Int(?) o_2 = Int(?)

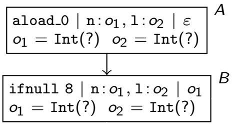
A

State *A*:

- do all calls of count terminate?
- num and limit are arbitrary, but distinct Int-objects

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     ⋮
19: if_icmpge 35
     ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

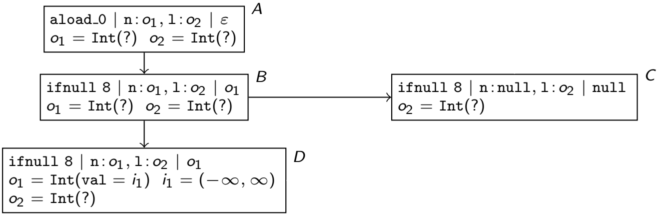


State *B*:

- "aload_0" loads value of num on operand stack
- *A* connected to *B* by *evaluation edge*

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     ⋮
19: if_icmpge 35
     ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

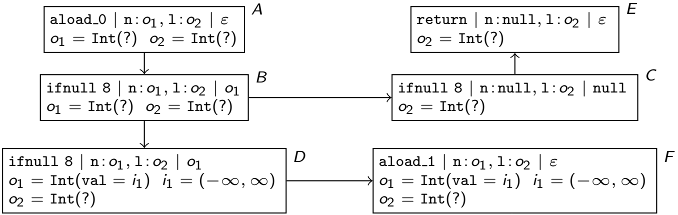


States *C* and *D*:

- "ifnull 8" needs to know whether o_1 is null
- *refine* information about heap (*refinement edges*)

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
      ⋮
19: if_icmpge 35
      ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

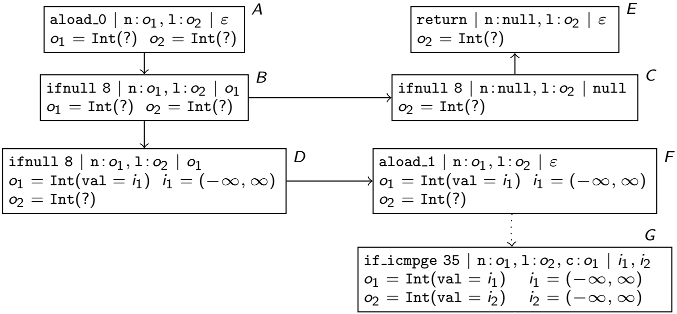


States E and F:

- evaluate "`ifnull 8`" in C and D
- *evaluation edges*

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     ⋮
19: if_icmpge 35
     ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

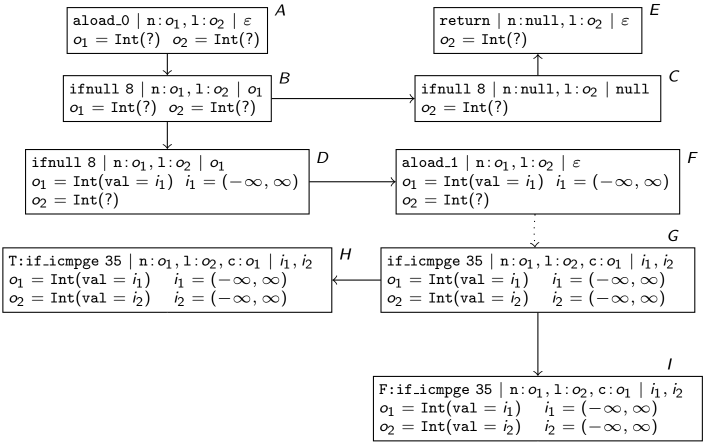


State G:

- in state F, check if `limit` is `null` analogously
- aliasing in G: `num` and `copy` point to the same address o_1
- `val`-fields of `num` and `limit` pushed on operand stack

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     :
19: if_icmpge 35
     :
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

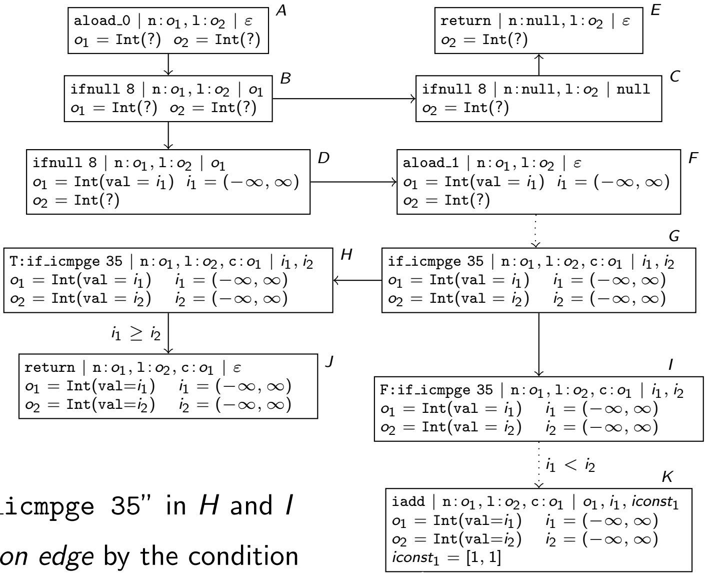


States *H* and *I*:

- "`if_icmpge 35`" needs to know whether $i_1 \geq i_2$
- *refine* information about heap (*refinement edges*)

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
     ⋮
19: if_icmpge 35
     ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

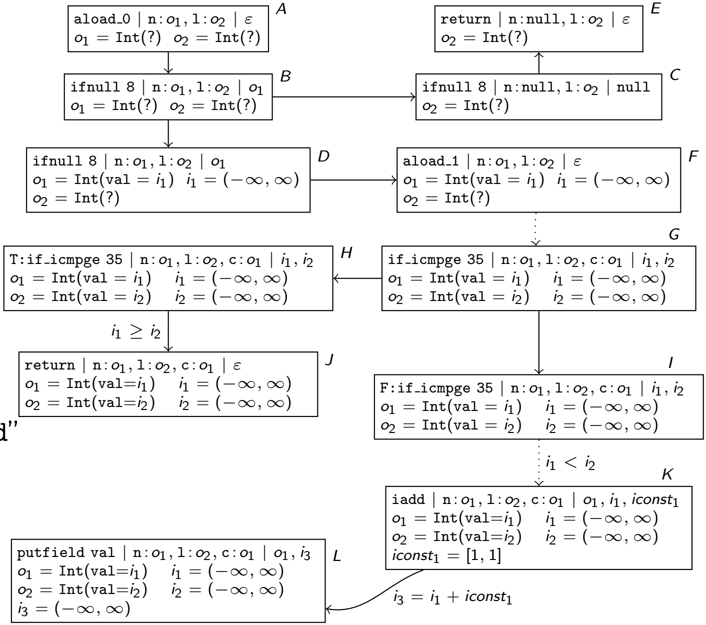


States J and K:

- evaluate "`if_icmpge 35`" in H and I
- label *evaluation edge* by the condition
- `val`-field of copy and integer variable with value 1 on operand stack

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
      ⋮
19: if_icmpge 35
      ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```

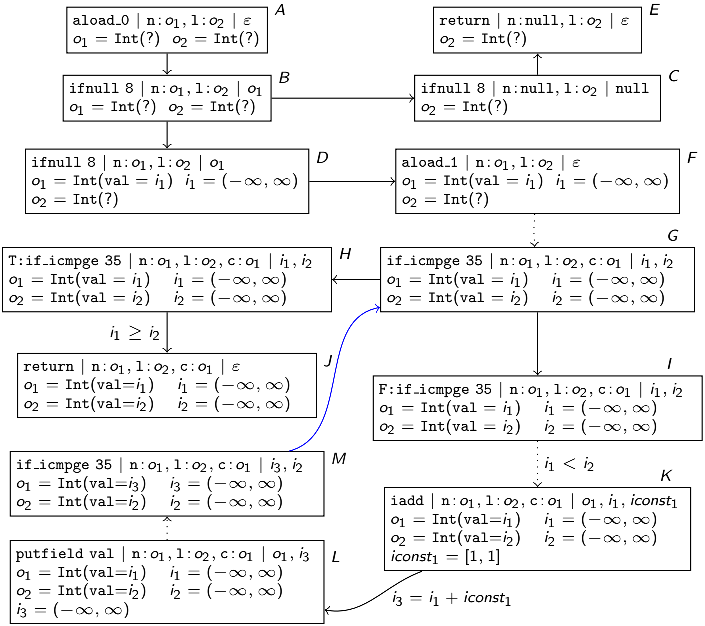


State *L*:

- evaluate "iadd"
- new variable i_3
- label edge by connection

From JBC to Termination Graphs

```
00: aload_0
01: ifnull 8
04: aload_1
    ⋮
19: if_icmpge 35
    ⋮
27: iconst_1
28: iadd
29: putfield val
32: goto 11
35: return
```



State *M*:

- again reaches "if_icmpge"
- *M* *instance* of *G*
- *instantiation edge*

From JBC to Termination Graphs

Termination Graphs

- expand nodes until all leaves correspond to program ends
- by appropriate generalization steps,
  one always reaches a *finite* termination graph
- state s_1 is *instance* of s_2 iff
  every concrete state described by s_1 is also described by s_2

Using Termination Graphs for Termination Proofs

- every JBC-computation of concrete states
  corresponds to a *computation path* in the termination graph
- termination graph is called *terminating* iff
  it has no infinite computation path

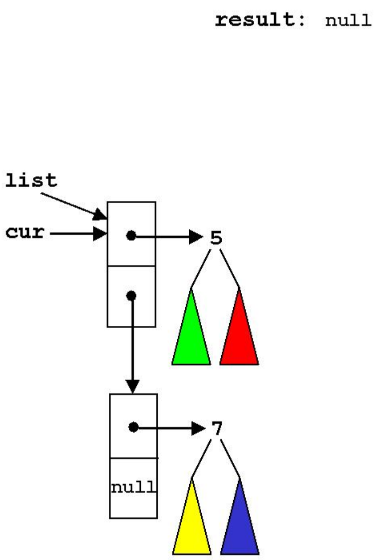
Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

```
public class Tree {
  int value;
  Tree left;
  Tree right;
}

public class TreeList {
  Tree value;
  TreeList next;
}

public class IntList {
  int value;
  IntList next;
}
```

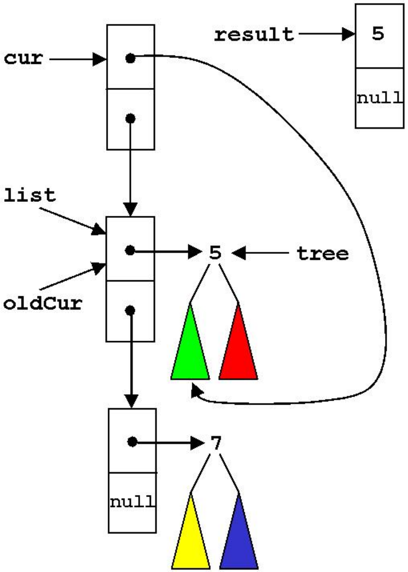
Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



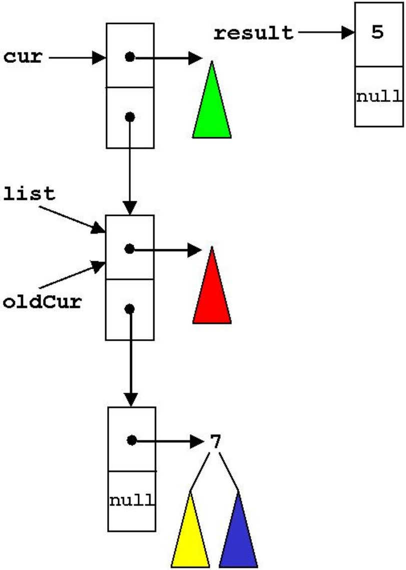
Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



no termination by *path length*

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

General state at beginning of loop body

$\texttt{aload\_1} \mid \texttt{l}:o_1, \texttt{c}:o_2, \texttt{r}:o_3 \mid \varepsilon$
$o_1 = \texttt{TreeList}(?)$ $o_2 = \texttt{TreeList}(?)$
$o_3 = \texttt{IntList}(?)$
$o_1 =^? o_2$ $\quad o_1 \searrow\!\!\swarrow o_2$

Annotations

- $o_1 =^? o_2$: o_1 and o_2 *may be equal*
- $o_1 \searrow\!\!\swarrow o_2$: o_1 and o_2 *may join*
  - $o \to o'$ iff object at address o has a field with value o'
  - $o_1 \searrow\!\!\swarrow o_2$: $o_1 \to^* \circ \leftarrow^+ o_2$ or $o_1 \to^+ \circ \leftarrow^* o_2$
- $o\,!$: *o does not have to be a tree*

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

A

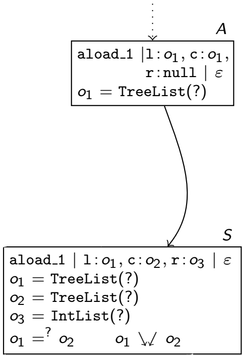
aload_1 | l : o_1, c : o_1, r : null | ε
o_1 = TreeList(?)

State *A*:

- reaches loop condition "`cur != null`" for the first time
- list and cur (o_1) are equal

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

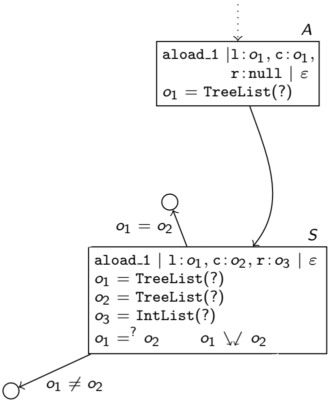


State *S*:

- *generalize A* to obtain finite termination graph
- `list` (o_1) and `cur` (o_2) may be equal and may join

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

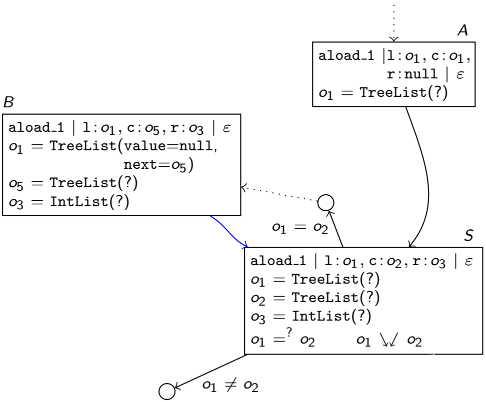


State S:

- *refinement* of annotation $o_1 =^? o_2$

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

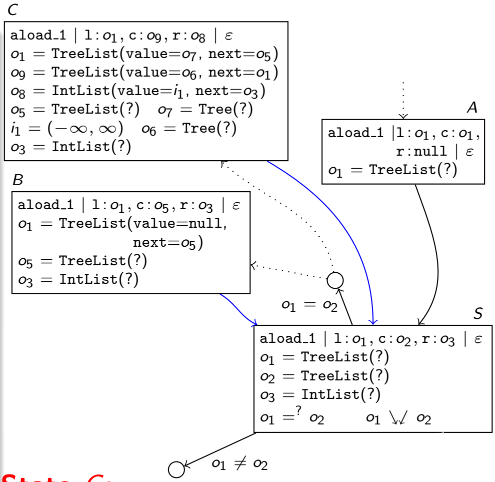


State B:

- reach loop condition if `tree == null`
- `list` $\rightarrow^{+} \circ \leftarrow^{*}$ `cur`
- B is *instance* of S

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

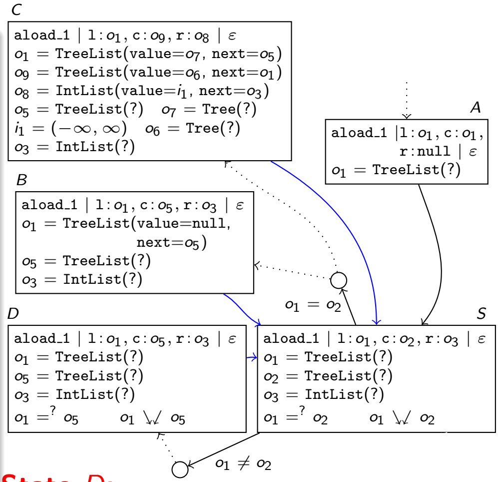


State C:

- Tree(value=i_1, left=o_6, right=o_7)
- list $\rightarrow^* \circ \leftarrow^+$ cur
- C is *instance* of S

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
         flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
         IntList oldIntList = result;
         result = new IntList();
         result.value = tree.value;
         result.next = oldIntList;
         TreeList oldCur = cur;
         cur = new TreeList();
         cur.value = tree.left;
         cur.next = oldCur;
         oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```

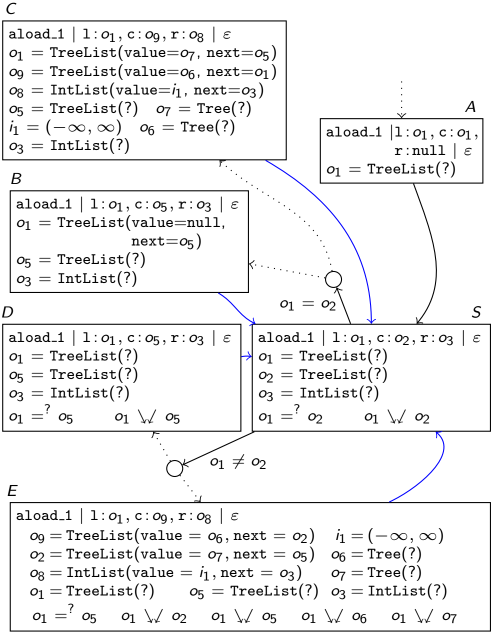


State *D*:

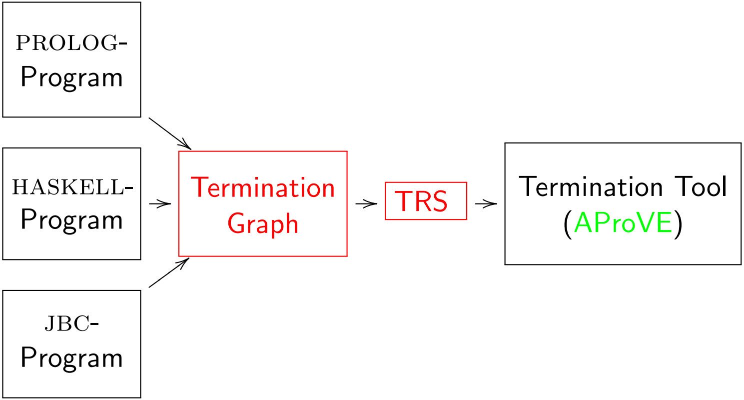
- o_2 = TreeList(value=o_4, next=o_5)
- tree (o_4) is null
- *D* is *instance* of *S*

Example with User-Defined Data Type

```
public class Flatten {
  public static IntList
        flatten(TreeList list) {
    TreeList cur = list;
    IntList result = null;

    while (cur != null) {
      Tree tree = cur.value;
      if (tree != null) {
        IntList oldIntList = result;
        result = new IntList();
        result.value = tree.value;
        result.next = oldIntList;
        TreeList oldCur = cur;
        cur = new TreeList();
        cur.value = tree.left;
        cur.next = oldCur;
        oldCur.value = tree.right;
      } else cur = cur.next;
    }
    return result;
  }
}
```



From Termination Graphs to TRSs



Transforming Objects to Terms

$\texttt{aload\_1} \mid \texttt{l}: o_1, \texttt{c}: o_9, \texttt{r}: o_8 \mid \varepsilon$

$o_9 = \texttt{TreeList}(\texttt{value} = o_6, \texttt{next} = o_2)$ $\quad i_1 = (-\infty, \infty)$

$o_2 = \texttt{TreeList}(\texttt{value} = o_7, \texttt{next} = o_5)$ $\quad o_6 = \texttt{Tree}(?)$

$o_8 = \texttt{IntList}(\texttt{value} = i_1, \texttt{next} = o_3)$ $\quad o_7 = \texttt{Tree}(?)$

$o_1 = \texttt{TreeList}(?)$ $\quad o_5 = \texttt{TreeList}(?)$ $\quad o_3 = \texttt{IntList}(?)$

$o_1 =^? o_5 \quad o_1 \searrow\!\!\!\swarrow o_2 \quad o_1 \searrow\!\!\!\swarrow o_5 \quad o_1 \searrow\!\!\!\swarrow o_6 \quad o_1 \searrow\!\!\!\swarrow o_7$

For every class C with n fields,
introduce function symbol C with n arguments

- term for o_1: o_1
- term for o_2: $\mathsf{TL}(o_7, o_5)$
- term for o_9: $\mathsf{TL}(o_6, \mathsf{TL}(o_7, o_5))$
- term for o_8: $\mathsf{IL}(i_1, o_3)$

Transforming Objects to Terms

Class Hierarchy

- for every class `C` with n fields,
  introduce function symbol C with $n+1$ arguments
- first argument: part of the object corresponding to subclasses of `C`

```
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;
```

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

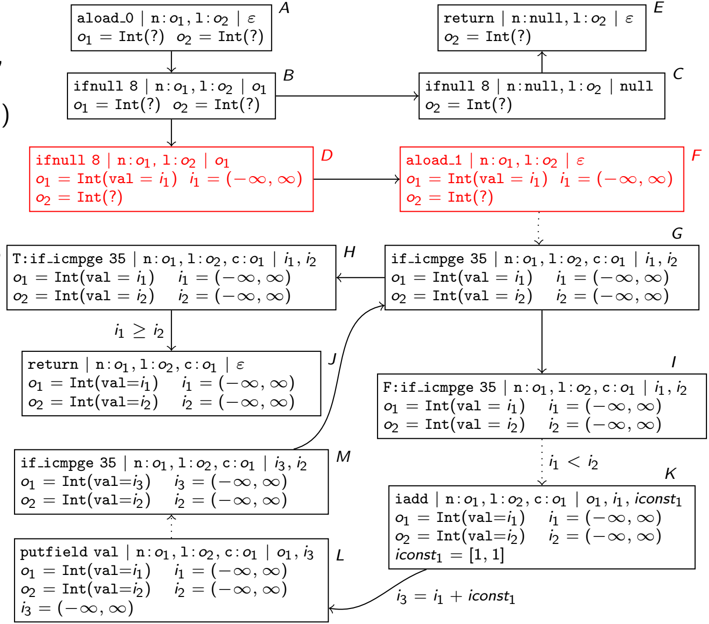
Transforming States to Tuples of Terms

Transforming D

$f_D(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),$
$\quad o_2,$
$\quad \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$

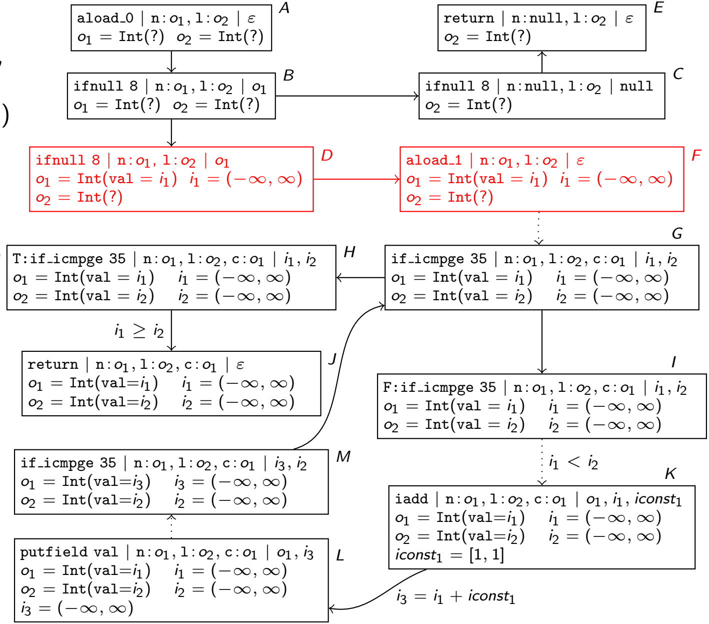
Transforming F

$f_F(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),$
$\quad o_2\,)$



Transforming Edges to Rewrite Rules

$$f_D(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)),\; o_2,\; \mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1))\,) \;\rightarrow\; f_F(\,\mathsf{jlO}(\mathsf{Int}(\mathsf{eoc}, i_1)),\; o_2\,)$$



Transforming Edges to Rewrite Rules

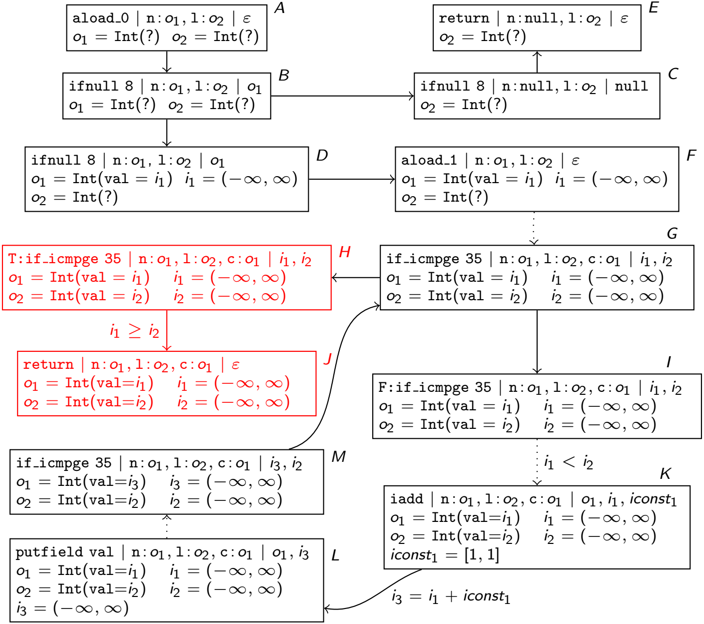
Transforming Evaluation Edges with Conditions

$$f_H(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\ \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_2)),\ \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\ i_1,\ i_2\,)$$

$$\rightarrow$$

$$f_J(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\ \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_2)),\ \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$

$$\mid\ i_1 \geq i_2$$



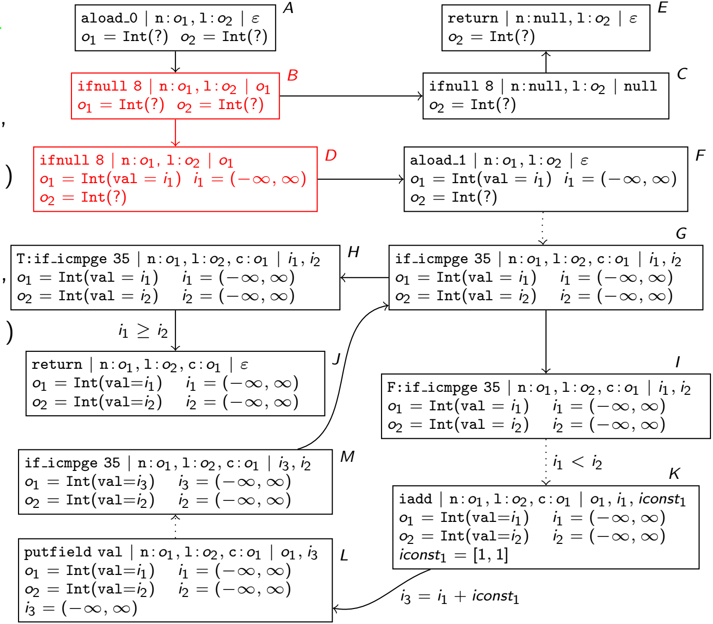
Transforming Edges to Rewrite Rules

Transforming Refinement Edges

$$f_B(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\; o_2,\; \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$

$$\to$$

$$f_D(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\; o_2,\; \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$



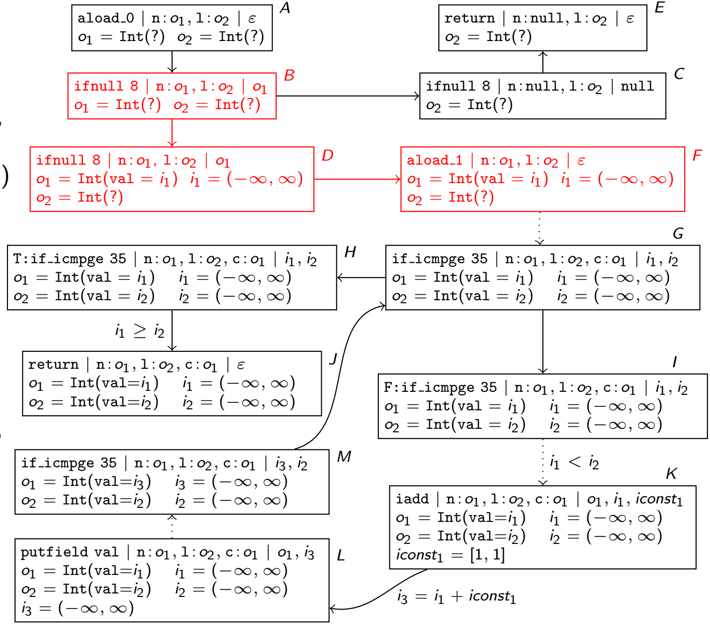
Transforming Edges to Rewrite Rules

Merging Rewrite Rules

$$f_B(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\; o_2,\; \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1))\,)$$

$$\to$$

$$f_F(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)),\; o_2\,)$$



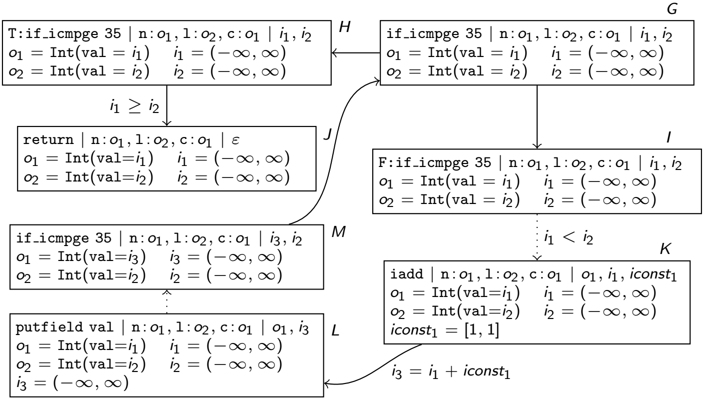
Transforming Edges to Rewrite Rules

TRS for count

$$\begin{array}{llllll} f_G(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)), & \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_2)), & \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1)), & i_1, & i_2) & \rightarrow \\ f_G(\,\mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1+1)), & \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_2)), & \mathrm{jlO}(\mathrm{Int}(\mathrm{eoc}, i_1+1)), & i_1+1, & i_2) & \mid \quad i_1 < i_2 \end{array}$$

TRS is “*natural*”

termination easy to prove automatically



From Termination Graphs to TRSs

TRS for count

$$\begin{array}{llllll} f_G(\text{jlO}(\text{Int}(\text{eoc}, i_1)), & \text{jlO}(\text{Int}(\text{eoc}, i_2)), & \text{jlO}(\text{Int}(\text{eoc}, i_1)), & i_1, & i_2) & \rightarrow \\ f_G(\text{jlO}(\text{Int}(\text{eoc}, i_1+1)), & \text{jlO}(\text{Int}(\text{eoc}, i_2)), & \text{jlO}(\text{Int}(\text{eoc}, i_1+1)), & i_1+1, & i_2) & \mid \quad i_1 < i_2 \end{array}$$

- every JBC-computation of concrete states corresponds to a *computation path* in the termination graph
- termination graph is called *terminating* iff it has no infinite computation path
- every computation path corresponds to rewrite sequence in TRS

Theorem

TRS corresponding to termination graph is terminating $\Rightarrow$

termination graph is terminating $\Rightarrow$

JBC-program terminating for all states represented in termination graph

From Termination Graphs to TRSs

$f_S(\mathsf{TL}(\mathsf{null}, o_5), \mathsf{TL}(\mathsf{null}, o_5), o_3) \rightarrow$
$f_S(\mathsf{TL}(\mathsf{null}, o_5), \quad o_5 \ , o_3)$

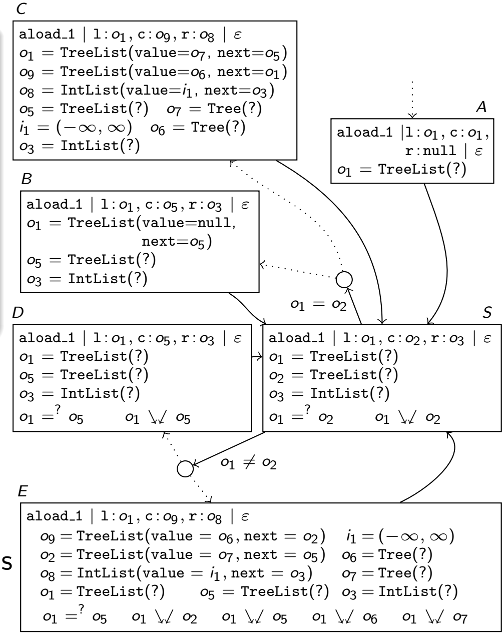
$f_S(\ldots, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), \quad o_3 \ ) \rightarrow$
$f_S(\ldots, \mathsf{TL}(\ o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$

$f_S(o_1, \mathsf{TL}(\mathsf{null}, o_5), o_3) \rightarrow$
$f_S(o_1, \quad o_5 \ , o_3)$

$f_S(o_1, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), \quad o_3 \ ) \rightarrow$
$f_S(o_1', \mathsf{TL}(\ o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$

Rewrite Rules & Annotations

- when writing to a field of o_2 with $o_1 \searrow\!\swarrow o_2$:
  o_1 on lhs, fresh variable o_1' on rhs
- cyclic objects: fresh variable on rhs



From Termination Graphs to TRSs

$f_S(\mathsf{TL}(\mathsf{null}, o_5), \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$

$f_S(\mathsf{TL}(\mathsf{null}, o_5), \quad o_5 \ , o_3)$

$f_S(\ldots, \mathsf{TL}(\mathsf{T}(i_1, o_6, o_7), o_5), \quad o_3 \ ) \to$

$f_S(\ldots, \mathsf{TL}(\ o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(i_1, o_3))$

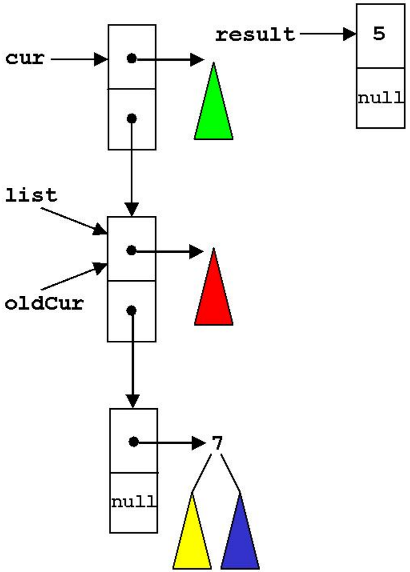
$f_S(o_1, \mathsf{TL}(\mathsf{null}, o_5), o_3) \to$

$f_S(o_1, \quad o_5 \ , o_3)$

$f_S(o_1, \mathsf{TL}(\ \mathsf{T}(5, o_6, o_7), o_5), \quad \mathsf{null} \ ) \to$
$f_S(o_1', \mathsf{TL}(\ o_6, \mathsf{TL}(o_7, o_5)), \mathsf{IL}(5, \mathsf{null}))$

TRS is "*natural*"

termination easy
to prove automatically



Automated Termination Analysis of JAVA BYTECODE by Term Rewriting

- implemented in AProVE and evaluated on collection of 387 JAVA-programs (including `java.util.LinkedList` and `HashMap`)
- extended for *recursion* and *cyclic data*
- adapted to detect *non-termination* and `NullPointerExceptions`

	Yes	No	Failure	Timeout	Runtime
AProVE	267	81	11	28	9.5
Julia	191	22	174	0	4.7
COSTA	160	0	181	46	11.0

- AProVE winner of the *International Termination Competition* for JAVA, HASKELL, PROLOG, term rewriting
- termination of "real" languages can be analyzed automatically, term rewriting is a suitable approach