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*)
- Temptation (*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
  \((\text{Walther, 94}), (\text{Giesl, 95}), (\text{Lee, Jones, Ben-Amram, 01})\)

- ensuring termination (e.g., by typing)
  \((\text{Telford & Turner, 00}), (\text{Xi, 02}), (\text{Abel, 04}), (\text{Barthe et al, 04}) \text{ etc.}\)

- outermost termination of untyped first-order rewriting
  \((\text{Fissore, Gnaedig, Kirchner, 02})\)

- automated technique for small \textsc{haskell}-like language
  \((\text{Panitz & Schmidt-Schauss, 97})\)

- new approach to use TRS-techniques for termination of \textsc{haskell}

- based on \((\text{Panitz & Schmidt-Schauss, 97})\), but:
  - works on full \textsc{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 → Nats**

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

Terms (well-typed)

- Variables: 
  - **x, y, ...**

- Function Symbols: 
  - constructors (Z, S, Nil, Cons) & defined (from, take)

- Applications (t₁ t₂)
  - **S Z** represents number 1
  - **Cons x Nil ≡ (Cons x) Nil** represents [x]
**Syntax of HASKELL**

**Data Structures**

```haskell
data Nats = Z | S Nats
```

- **type constructor**: `Nats` of arity 0
- **data constructors**:
  - `Z :: Nats`
  - `S :: Nats -> Nats`

```haskell
data List a = Nil | Cons a (List a)
```

- **type constructor**: `List a` of arity 1
- **data constructors**:
  - `Nil :: List a`
  - `Cons :: a -> (List a) -> (List a)`

**Types**

- **Type Variables**: `a, b, ...`
- **Applications of type constructors to types**: `List Nats, a -> (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)

\[
\begin{align*}
\text{from } x &= \text{Cons } x \left( \text{from } (S \ x) \right) \\
\text{take } Z \ xs &= \text{Nil} \\
\text{take } n \ \text{Nil} &= \text{Nil} \\
\text{take } (S \ n) \ (\text{Cons } x \ xs) &= \text{Cons } x \ (\text{take } n \ xs) \\
\text{from } :: \ \text{Nats} \rightarrow \text{List Nats} \\
\text{take } :: \ \text{Nats} \rightarrow (\text{List } a) \rightarrow (\text{List } a) \\
\text{from } x &\equiv [x, x + 1, x + 2, \ldots] \\
\text{take } n \ [x_1, \ldots, x_n, \ldots] &\equiv [x_1, \ldots, x_n]
\end{align*}
\]
Syntax of \textsc{Haskell}

Extension of our approach for

\begin{itemize}
  \item type classes
  \item built-in data structures
\end{itemize}

\textbf{All other \textsc{Haskell}-constructs:} eliminated by automatic transformation

\begin{itemize}
  \item \textbf{Lambda Abstractions}
  \begin{align*}
    \text{replace} & \quad m \to \text{take} \ u \ (\text{from} \ m) \\
    \text{by} & \quad f \ u \\
    \text{where} & \quad f \ u \ m = \ \text{take} \ u \ (\text{from} \ m)
  \end{align*}
\end{itemize}
Extension of our approach for

- type classes
- built-in data structures

**All other HASKELL-constructs:** eliminated by automatic transformation

- **Lambda Abstractions**
  replace \( t_1 \ldots t_n \rightarrow t \) with free variables \( x_1, \ldots, x_m \)
  by \( f \; x_1 \ldots x_m \)
  where \( f \; x_1 \ldots x_m \; t_1 \ldots t_n = t \)

- **Conditions**
- **Local Declarations**
- ...
Semantics and Termination of HASKELL

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

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

from \( Z \)
\( \rightarrow_H \, \text{Cons} \, Z \, (\text{from} \, (S \, Z)) \)
\( \rightarrow_H \, \text{Cons} \, Z \, (\text{Cons} \, (S \, Z) \, (\text{from} \, (S \, (S \, Z)))) \) \( \text{evaluation position} \)
\( \rightarrow_H \, \ldots \)
from $x = \text{Cons } x (\text{from } (\text{S } x))$

take $Z \hspace{1mm} x x = \text{Nil}$

take $n \hspace{1mm} \text{Nil} = \text{Nil}$

take $(\text{S } n) (\text{Cons } x \hspace{1mm} x x) = \text{Cons } x (\text{take } n \hspace{1mm} x x)$

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

from $m$

$\rightarrow_{H} \hspace{1mm} \text{Cons } m (\text{from } (\text{S } m))$

$\rightarrow_{H} \hspace{1mm} \text{Cons } m (\text{Cons } (\text{S } m) (\text{from } (\text{S } (\text{S } m))))$

$\rightarrow_{H} \hspace{1mm} \ldots$
Semantics and Termination of \textsc{Haskell}

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

\textbf{Evaluation Relation} \quad \rightarrow_{\text{H}}

\[
\begin{align*}
\text{take } (S \ Z) \ (\text{from } m) & \rightarrow_{\text{H}} \text{take } (S \ Z) \ (\text{Cons } m \ (\text{from } (S \ m))) \\
\rightarrow_{\text{H}} \text{Cons } m \ (\text{take } Z \ (\text{from } (S \ m))) & \quad \text{evaluation position} \\
\rightarrow_{\text{H}} \text{Cons } m \ \text{Nil}
\end{align*}
\]
**Semantics and Termination of HASKELL**

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

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

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

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

- “from” not \( H \)-terminating (“from \( Z \)” has infinite evaluation)  
  
  “take \( u \) (from \( m \))” is \( H \)-terminating
Proving Termination of HASKELL

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

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

Goal: Prove termination of start term “\( \text{take} \ u \ (\text{from} \ m) \)”

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

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

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

- 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

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

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

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

**Case**

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

- \([u/Z]\)
- \([u/(S \ n)]\)

---

**Main Property of Termination Graphs:**

A node is \( H \)-terminating if all its children are \( H \)-terminating.
From HASKELL to Termination Graphs

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

take \( Z \) \( xs = \) Nil

take \( n \) Nil = Nil

take (\( S \) \( n \)) (Cons \( x \) \( xs \)) = Cons \( x \) (take \( n \) \( xs \))

**Eval** rule:

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

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

\[
\text{Case} \quad \text{take } u \text{ (from } m) \\
\text{Eval} \quad \text{take } Z \text{ (from } m) \\
\text{Nil} \\
\]

\[
\text{Eval} \quad \text{take } (S \ n) \text{ (from } m) \\
\]

\[
\text{Eval} \quad \text{take } (S \ n) \text{ (Cons } m \text{ (from } (S \ m))) \\
\]

\[
\text{Cons } m \text{ (take } n \text{ (from } (S \ m)))
\]

Case and Eval rule perform narrowing w.r.t. HASKELL’s evaluation strategy and types
From HASKELL to Termination Graphs

from \( x = Cons \, x \, (from \, (S \, x)) \)

take \( Z \, xs = Nil \)

take \( n \, Nil = Nil \)

take \( (S \, n) \, (Cons \, x \, xs) = Cons \, x \, (take \, n \, xs) \)

\[ \text{Eval} \]
\[ \text{take} \, U \, (from \, m) \]
\[ \text{Case} \]
\[ \text{take} \, U \, (from \, m) \]
\[ \text{Eval} \]
\[ \text{take} \, (S \, n) \, (from \, m) \]

\[ \text{Eval} \]
\[ \text{take} \, (S \, n) \, (Cons \, m \, (from \, (S \, m))) \]

\[ \text{ParSplit} \]
\[ \text{Cons} \, m \, (take \, n \, (from \, (S \, m))) \]

\[ m \]
\[ \text{take} \, n \, (from \, (S \, m)) \]

- **ParSplit** rule:
  if head of term is a constructor like Cons or a variable,
  then continue with the parameters
From **HASKELL** to Termination Graphs

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

- **Case**
  - take \( u \) (from \( m \))
    - [\( u/Z \)]
    - Nil

- **Eval**
  - take \( Z \) (from \( m \))
    - Nil
  - take \( (S \, n) \) (from \( m \))
    - [\( u/(S \, n) \)]

- **Eval**
  - take \( (S \, n) \) (Cons \( m \) (from \( (S \, m) \)))

- **ParSplit**

\[
\begin{align*}
\text{Cons } m \, (\text{take } n \, (\text{from } (S \, m)))
\end{align*}
\]

- **Ins** rule to obtain finite graphs

- **Case, Eval, ParSplit** ⇒ infinite tree

- **Instead**: \textbf{Ins} rule to obtain finite graphs
From **HASKELL** to Termination Graphs

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

**Eval**

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

\( \text{Nil} \)

**Case**

\( \text{take} \; u \; (\text{from} \; m) \)

\( \text{[u/Z]} \)

**Eval**

\( \text{take} \; (S \; n) \; (\text{from} \; m) \)

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

\( \text{take} \; (S \; n) \; (\text{Cons} \; m \; (\text{from} \; (S \; m))) \)

**Eval**

\( \text{Cons} \; m \; (\text{take} \; n \; (\text{from} \; (S \; m))) \)

**ParSplit**

\( m \)

**Ins**

\( \text{take} \; n \; (\text{from} \; (S \; m)) \)

**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

from $x = \text{Cons } x \text{ (from } (S \, x))$  
\hspace{1cm} \text{take } Z \, xs = \text{Nil} 
\hspace{1cm} \text{take } n \, \text{Nil} = \text{Nil} 
\hspace{1cm} \text{take } (S \, n) \, (\text{Cons } x \, xs) = \text{Cons } x \, (\text{take } n \, xs)$

\textbf{Case}

\text{take } u \, (\text{from } m)$

\hspace{1cm} [u/Z]$

\text{Eval}$

\hspace{1cm} \text{Nil}$

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

\hspace{1cm} [u/(S \, n)]$

\text{Eval}$

\hspace{1cm} \text{take } n \, (\text{from } (S \, m))$

\textbf{ParSplit}$

\hspace{1cm} \text{Cons } m \, (\text{take } n \, (\text{from } (S \, m)))$

\textbf{Ins}$

\hspace{1cm} m$

\hspace{1cm} m/(S \, m)$

\hspace{1cm} n$

\textbf{Ins}$ rule:

- if leaf $t$ is instance of $t'$, then add instantiation edge from $t$ to $t'$
- since instantiation is $[u/n, m/(S \, m)]$, add child nodes $n$ and $(S \, m)$
From HASKELL to Termination Graphs

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

\begin{itemize}
  \item **ParSplit** rule:
    if head of term is a constructor like \( S \),
    then continue with the parameter
\end{itemize}
From HASKELL to Termination Graphs

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

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

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

- **Goal:** Prove $\eta$-termination of all terms in termination graph
From HASKELL to Termination Graphs

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

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

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

**Case**

- **Eval**
  - take \(Z\) \(\left(\text{from } m\right)\)
    - Nil
  
- **Eval**
  - take \(u\) \(\left(\text{from } m\right)\)
    - \([u/Z]\)
  
- **Eval**
  - take \((S \times n)\) \(\left(\text{from } m\right)\)
    - \([u/(S \times n)]\)
  
- **ParSplit**
  - Cons \(m\) \(\left(\text{take } n \left(\text{from } (S \times m)\right)\right)\)
  
- **Ins** rule:
  - if leaf \(t\) is instance of \(t'\), then add instantiation edge from \(t\) to \(t'\)
  - since instantiation is \([\times s/\text{from } (S \times m)]\), add child node from \(\text{from } (S \times m)\)
From HASKELL to Termination Graphs

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

**Case**

- **Eval**
  - \( \text{take } Z (\text{from } m) \) leads to Nil
  - \( \text{take } u (\text{from } m) \) with substitutions \([u/Z] \) and \([u/(S \, n)]\)

**Eval**

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

**Eval**

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

**ParSplit**

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

- \( m \) leading to \( \text{take } n (\text{from } (S \, m)) \)

- \( \text{take } n \, x s \) leading to \( \text{from } (S \, m) \)

**Ins** rule:

- if leaf \( t \) is instance of \( t' \), then add instantiation edge from \( t \) to \( t' \)
- proving \( \text{H-termination of all terms in termination graph fails!} \)
From **HASKELL** to Termination Graphs

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

**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 $\mathcal{H}$-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: \(\text{Proc}( (\mathcal{P}, \mathcal{R}) ) = \{ (\mathcal{P}_1, \mathcal{R}_1), \ldots, (\mathcal{P}_n, \mathcal{R}_n) \}\)

- When is a problem solved?
  - \((\mathcal{P}, \mathcal{R})\) is finite iff there is no infinite \((\mathcal{P}, \mathcal{R})\)-chain
    \[s_1 \sigma_1 \rightarrow_{\mathcal{P}} t_1 \sigma_1 \rightarrow^*_\mathcal{R} s_2 \sigma_2 \rightarrow_{\mathcal{P}} t_2 \sigma_2 \rightarrow^*_\mathcal{R} \ldots\] where \(s_i \rightarrow t_i \in \mathcal{P}\)
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
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

Start Term

Program → Termination Graph → DP Problems → Termination Tool (AProVE)
higher-order terms can be represented as applicative first-order terms

"x y" becomes "ap(x, y)"
**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** $\mathcal{P}$: $\text{take } (S\, n) \ (\text{from } m) \rightarrow \text{take } n \ (\text{from } (S\, m))$

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

- $\emptyset$
  - termination is easy to prove

**Dependency Pairs:**

if there is a DP path from $s$ to $t$ marked with $\mu$, then generate the dependency pair $s \ \mu \rightarrow t$
Generating infinite \((\mathcal{P}, \mathcal{R})\)-chains

Term in graph not terminating

\[ \sim \quad \text{not terminating} \]

DP path from \(s_1\) to \(t_1\) marked with \(\mu_1\)

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ \sim \quad \text{not terminating} \]

\[ s_1 \mu_1 \sigma_1 \rightarrow_{\mathcal{P}} t_1 \sigma_1 \]

\[ s_2 \mu_2 \sigma_2 \rightarrow_{\mathcal{P}} t_2 \sigma_2 \]

\[ s_2 \tau_2 \rightarrow_{\mathcal{R}}^* s_2 (\tau_2 \downarrow_H) \]

\[ \mathcal{R}: \text{rules for terms in matcher} \]

\[ \mathcal{R} = \emptyset \text{ if no defined symbol in matcher} \]
From Termination Graphs to DP Problems

$$\text{from } x = \text{Cons } x \text{ (from } (S \ x)) \quad \text{take } Z \hspace{1mm} x \hspace{1mm} s = \text{Nil}$$

$$\text{take } n \hspace{1mm} \text{Nil} = \text{Nil}$$

$$p \hspace{1mm} (S \ x) = x$$

$$\text{take } (S \ n) \text{ (Cons } x \hspace{1mm} x \hspace{1mm} s) = \text{Cons } x \text{ (take } (p \hspace{1mm} (S \ n)) \hspace{1mm} x \hspace{1mm} s)$$
Dependency Pair $\mathcal{P}$: $\text{take} \ (S \ n) \ (\text{from } m) \rightarrow \text{take} \ (p \ (S \ n)) \ (\text{from } (S \ m))$

$\mathcal{R}$: rules for terms in matcher
From Termination Graphs to DP Problems

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

Rule \( \mathcal{R} \):

- \(p \, (S \, n) \rightarrow n\)
- termination easy to prove

Rules

if there is a rule path from \(s\) to \(t\) marked with \(\mu\),
then generate the rule \(s \, \mu \rightarrow t\)
From Termination Graphs to DP Problems

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

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

- **Improvement**: evaluate rhs of DP as much as possible
  
  \[
  \text{ev}\,(\text{take}\,(p\,(S\,n))\,(\text{from}\,(S\,m))) = \text{take}\,\text{ev}\,(p\,(S\,n))\,(\text{from}\,\text{ev}(S\,m)) = \text{take}\,n\,(\text{from}\,(S\,m))}
  \]

- **$\text{ev}(t)$**: term reachable from $t$ by traversing $\textbf{Eval}$-nodes
  
  traverses subterms of $\textbf{ParSplit}$- and $\textbf{Ins}$-nodes
From Termination Graphs to DP Problems

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

- **Rule** $\mathcal{R}$: $\emptyset$

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

\[
ev(\ \text{take} \ (p\ (S\ n)) \ (\text{from} \ (S\ m))) \\
= \text{take} \ ev(\ p\ (S\ n)) \ (\text{from} \ ev(\ S\ m)) \\
= \text{take} \ n \ (\text{from} \ (S\ m))
\]

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

- **Rules**

  only needed for terms where computation of $\ev$ stopped
From Termination Graphs to DP Problems

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

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

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

$p \,(S \, Z) = Z$

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

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

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

- $p (SZ) \rightarrow Z$
- $p (S (S \, x)) \rightarrow S (p (S \, x))$

**Rules**

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

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

- **Rules** $\mathcal{R}$:
  - $p(SZ) \rightarrow Z$
  - $p(S(Sx)) \rightarrow S(p(Sx))$

- **Dependency Pair** $\mathcal{P}$: $p(S(Sx)) \rightarrow p(Sx)$

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

- **Diagram**

- **Observations**
  - 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

<table>
<thead>
<tr>
<th></th>
<th>FiniteMap</th>
<th>List</th>
<th>Maybe</th>
<th>Monad</th>
<th>Prelude</th>
<th>Queue</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>YES</strong></td>
<td>256</td>
<td>166</td>
<td>9</td>
<td>69</td>
<td>489</td>
<td>5</td>
<td>994</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>321</td>
<td>174</td>
<td>9</td>
<td>80</td>
<td>692</td>
<td>5</td>
<td>1281</td>
</tr>
</tbody>
</table>
Termination of **HASKELL-Programs**

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

  - **Start Term**
  - **HASKELL-Program** $\rightarrow$ **Termination Graph** $\rightarrow$ **DP Problems** $\rightarrow$ **Termination Tool (AProVE)**

- Implemented in **AProVE**

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

<table>
<thead>
<tr>
<th></th>
<th>FiniteMap</th>
<th>List</th>
<th>Maybe</th>
<th>Monad</th>
<th>Prelude</th>
<th>Queue</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>YES</strong></td>
<td>256</td>
<td>166</td>
<td>9</td>
<td>69</td>
<td>489</td>
<td>5</td>
<td>994</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td>321</td>
<td>174</td>
<td>9</td>
<td>80</td>
<td>692</td>
<td>5</td>
<td>1281</td>
</tr>
</tbody>
</table>
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
analyzing PROLOG is challenging due to cuts etc.

**New approach**

- **Frontend**
  - evaluate PROLOG a few steps ⇒ *symbolic evaluation graph*
    - graph captures evaluation strategy due to cuts etc.
  - transform *symbolic evaluation graph* ⇒ 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**

- PROLOG-Program → Symbolic Evaluation Graph → TRS → Rewrite Tool (AProVE)
  - Termination
  - Complexity
  - Determinacy

**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, []) :- !.  \hspace{1cm} (1)
star([], ZS) :- !, eq(ZS, []). \hspace{1cm} (2)
star(XS, ZS) :- app(XS, YS, ZS), star(XS, YS). \hspace{1cm} (3)
\hspace{1cm} app([], YS, YS). \hspace{1cm} (4)
app([X \mid XS], YS, [X \mid ZS]) :- app(XS, YS, ZS). \hspace{1cm} (5)
\hspace{1cm} eq(X, X). \hspace{1cm} (6)

\bullet \text{ star}(t_1, t_2) \text{ holds iff } t_2 \text{ results from concatenation of } t_1 \ (t_2 \in (t_1)^*)

\bullet \text{ star([1, 2], []) holds}

\bullet \text{ star([1, 2], [1, 2]) holds, since app([1, 2], [], [1, 2]), star([1, 2], []) hold}

\bullet \text{ star([1, 2], [1, 2, 1, 2]) holds, etc.}

\bullet \text{ cut in clause (2) needed for termination. Otherwise:}

\hspace{1cm} \text{ star([], t) would lead to}

\hspace{1cm} \text{ app([], YS, t), star([], YS) would lead to}

\hspace{1cm} \text{ star([], t)}
\[
\text{star}(XS, []) \leftarrow !. \quad (1)
\]
\[
\text{star}([], ZS) \leftarrow !, \text{eq}(ZS, []). \quad (2)
\]
\[
\text{star}(XS, ZS) \leftarrow \text{app}(XS, YS, ZS), \text{star}(XS, YS). \quad (3)
\]
\[
\text{app}([], YS, YS). \quad (4)
\]
\[
\text{app}([X \mid XS], YS, [X \mid ZS]) \leftarrow \text{app}(XS, YS, ZS). \quad (5)
\]
\[
\text{eq}(X, X). \quad (6)
\]

- **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**

\[
\begin{align*}
\text{Case} & \quad \vdash \text{star}([1, 2], []) \\
\text{Eval} & \quad \vdash \text{star}([1, 2], [])^{(1)} \mid \text{star}([1, 2], [])^{(2)} \mid \text{star}([1, 2], [])^{(3)} \\
\text{Back} & \quad \vdash !_1 \mid \text{star}([1, 2], [])^{(2)} \mid \text{star}([1, 2], [])^{(3)} \\
\text{Cut} & \quad \vdash \quad \vdash \text{star}([1, 2], [])^{(1)} \mid \text{star}([1, 2], [])^{(2)} \mid \text{star}([1, 2], [])^{(3)} \\
\text{Suc} & \quad \vdash \quad \vdash \text{star}([1, 2], [])^{(1)} \mid \text{star}([1, 2], [])^{(2)} \mid \text{star}([1, 2], [])^{(3)} \\
\end{align*}
\]
\[
\text{star}(XS, []) := !. \quad (1)
\]
\[
\text{star}([], ZS) := !, \text{eq}(ZS, []). \quad (2)
\]
\[
\text{star}(XS, ZS) := \text{app}(XS, YS, ZS), \text{star}(XS, YS). \quad (3)
\]
\[
\text{app}([], YS, YS). \quad (4)
\]
\[
\text{app}([X | XS], YS, [X | ZS]) := \text{app}(XS, YS, ZS). \quad (5)
\]
\[
\text{eq}(X, X). \quad (6)
\]

- **state:** \((G_1 | \ldots | G_n)\) 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

\[
\text{star}([1, 2], []) \quad \vdash \quad \text{CASE}
\]
\[
\text{star}([1, 2], []) \quad \vdash \quad \text{EVAL}
\]
\[
\text{star}([1, 2], []) \quad \vdash \quad \text{CUT}
\]
\[
\quad \vdash \quad \text{SUC}
\]
\[
\quad \vdash \quad \varepsilon
\]
Symbolic Evaluation Graphs and Term Rewriting

**General methodology for analyzing PROLOG programs**

1. **PROLOG-Program** $\rightarrow$ **Symbolic Evaluation Graph** $\rightarrow$ **TRS** $\rightarrow$ **Rewrite Tool (AProVE)**
   - **Termination**
   - **Complexity**
   - **Determinacy**

**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
symbolic evaluation graph: all evaluations for a class of queries

class of queries $Q^p_m$ described by predicate $p$ and moding $m$

Example: $Q^\text{star}_m = \{ \text{star}(t_1, t_2) \mid t_1, t_2 \text{ are ground} \}$.

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: $\text{star}(\overline{T_1}, \overline{T_2}) \sim \text{star}(XS, [])$
\[\text{star}(XS, []) : \neg !. \quad (1)\]
\[\text{star}([], ZS) : \neg !, \text{eq}(ZS, []). \quad (2)\]
\[\text{star}(XS, ZS) : \text{app}(XS, YS, ZS), \text{star}(XS, YS). \quad (3)\]

- **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: \(\text{star}(\overline{T_1}, \overline{T_2}) \sim \text{star}(XS, [])\)
\[
\text{star}(\text{XS}, []) \leftarrow !. \quad (1)
\text{star}([], \text{ZS}) \leftarrow !, \text{eq}(\text{ZS}, []). \quad (2)
\text{star}(\text{XS}, \text{ZS}) \leftarrow \text{app}(\text{XS}, \text{YS}, \text{ZS}), \text{star}(\text{XS}, \text{YS}). \quad (3)
\]
Symbolic Evaluation Graphs and Term Rewriting

**General methodology for analyzing** PROLOG programs

- PROLOG-Program
- Symbolic Evaluation Graph
- TRS
- Rewrite Tool (AProVE)

**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

**Termination**

**Complexity**

**Determinacy**
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 $f_{s}^{in}, f_{s}^{out}$
- **encode connection paths to rewrite rules**

**connection path:**

- **start state** = root, successor of \texttt{Inst}, or successor of \texttt{Split} but no \texttt{Inst} or \texttt{Split} node itself

- **end state** = \texttt{Inst}, \texttt{Split}, \texttt{Suc} node, or successor of \texttt{Inst} node

- connection path may not traverse end nodes except \texttt{Suc} nodes
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 $\sigma$:

\[ f_{s}^{in}(\ldots)\sigma \text{ evaluates to } f_{s}^{out}(\ldots)\sigma \text{ if } f_{s'}^{in}(\ldots) \text{ evaluates to } f_{s'}^{out}(\ldots) \]

\[ f_{A}^{in}(T_{1}, T_{2}) \text{ evaluates to } f_{A}^{out} \text{ if } f_{A}^{in}(T_{1}, T_{2}) \text{ evaluates to } f_{A}^{out}(T_{3}) \]

rewrite rules:

\[ f_{s}^{in}(\ldots)\sigma \rightarrow u_{s,s'}( f_{s'}^{in}(\ldots) ) \]

\[ u_{s,s'}( f_{s'}^{out}(\ldots) ) \rightarrow f_{s}^{out}(\ldots)\sigma \]

\[ f_{A}^{in}(T_{1}, T_{2}) \rightarrow u_{A}( f_{F}^{in}(T_{1}, T_{2}) ) \]

\[ u_{A,F}( f_{F}^{out}(T_{3}) ) \rightarrow \]

\[ f_{A}^{out}(T_{1}, T_{2}) \rightarrow u_{A,F}( f_{F}^{out}(T_{1}, T_{2}) ) \]
connection path from \( s \) ending in \( \text{Suc} \) node:

\[ f^\text{in}_s(\ldots)\sigma \text{ evaluates to } f^\text{out}_s(\ldots)\sigma \]

\[ f^\text{in}_A(T_1, [\ ])(\ ) \text{ evaluates to } f^\text{out}_A(\ ) \]

intuition:

\[ f^\text{in}_A(T_1, T_2) \text{ evaluates to } f^\text{out}_A(\ ) \text{ if } T_2 \in (T_1)^* \]

\[ f^\text{in}_F(T_1, T_2) \text{ evaluates to } f^\text{out}_F(T_3) \text{ if } T_1 \neq [\ ], T_2 \neq [\ ], T_3 \text{ is } T_2 \text{ without prefix } T_1, T_3 \in (T_1)^* \]
\[ f^{in}_A(T_1, T_2) \rightarrow u_{A,F}(f^{in}_F(T_1, T_2)) \]
\[ u_{A,F}(f^{out}_F(T_3)) \rightarrow f^{out}_A \]
\[ f^{in}_A(T_1, []) \rightarrow f^{out}_A \]
\[ f^{in}_F(T_1, T_2) \rightarrow u_{F,G}(f^{in}_G(T_1, T_2)) \]
\[ u_{F,G}(f^{out}_G(T_4)) \rightarrow u_{G,H}(f^{in}_A(T_1, T_4), T_4) \]
\[ u_{G,H}(T_4) \rightarrow f^{out}_F(T_4) \]

**Split node** \( s \) with successors \( s_1 \) and \( s_1' \):

- \( f^{in}_s(\ldots)\sigma \) evaluates to \( f^{out}_s(\ldots)\sigma \) if \( f^{in}_s(\ldots)\sigma \) evaluates to \( f^{out}_s(\ldots)\sigma \) and \( f^{in}(T_1, T_2) \) evaluates to \( f^{out}(T_4) \)
- \( f^{in}_s(\ldots)\sigma \) evaluates to \( f^{out}_s(\ldots)\sigma \) if \( f^{in}(T_1, T_2) \) evaluates to \( f^{out}(T_4) \) and \( f^{in}(T_1, T_4) \) evaluates to \( f^{out}(T_4) \)
\[
\begin{align*}
\text{intuition:} \\
\text{if } T_1 \neq [], T_2 \neq [], T_4 \text{ is } T_2 \text{ without prefix } T_1, T_4 \in (T_1)^* & \\
\text{if } T_1 \neq [], T_2 \neq [], T_4 \text{ is } T_2 \text{ without prefix } T_1 & \\
\text{if } T_4 \in (T_1)^* & \\
\end{align*}
\]
\begin{align*}
\text{star}(XS, []) & :- \bot. \\
\text{star}([], ZS) & :- \bot, \text{eq}(ZS, []). \\
\text{star}(XS, ZS) & :- \text{app}(XS, YS, ZS), \text{star}(XS, YS). \\
\text{app}([], YS, YS). \\
\text{app}([X | XS], YS, [X | ZS]) & :- \text{app}(XS, YS, ZS). \\
\text{eq}(X, X) & .
\end{align*}

\begin{align*}
\text{f}_{\text{in}}(T_1, T_2) & \rightarrow \text{u}_{A,F}(\text{f}_{\text{in}}(T_1, T_2)) \\
\text{u}_{A,F}(\text{f}_{\text{out}}(T_3)) & \rightarrow \text{f}_{\text{out}} \\
\text{f}_{\text{in}}(T_1, []) & \rightarrow \text{f}_{\text{out}} \\
\text{f}_{\text{in}}(T_1, T_2) & \rightarrow \text{u}_{F,G}(\text{f}_{\text{in}}(T_1, T_2)) \\
\text{u}_{F,G}(\text{f}_{\text{out}}(T_4)) & \rightarrow \text{u}_{G,H}(\text{f}_{\text{in}}(T_1, T_4), T_4) \\
\text{u}_{G,H}(\text{f}_{\text{out}}(T_4)) & \rightarrow \text{f}_{\text{out}}(T_4) \\
\text{f}_{\text{in}}([T_5 | T_6], [T_5 | T_7]) & \rightarrow \text{u}_{G,I}(\text{f}_{\text{in}}(T_6, T_7)) \\
\text{u}_{G,I}(\text{f}_{\text{out}}(T_3)) & \rightarrow \text{f}_{\text{out}}(T_3) \\
\text{f}_{\text{in}}([T_8 | T_9], [T_8 | T_{10}]) & \rightarrow \text{u}_{I,K}(\text{f}_{\text{in}}(T_9, T_{10})) \\
\text{u}_{I,K}(\text{f}_{\text{out}}(T_3)) & \rightarrow \text{f}_{\text{out}}(T_3) \\
\text{f}_{\text{in}}([], T_3) & \rightarrow \text{f}_{\text{out}}(T_3)
\end{align*}

- 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 $Q^p_m$

- $prc_{\mathcal{P},Q^p_m}$ maps $n \in \mathbb{N}$ to longest evaluation starting with $Q \in Q^p_m$, 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 $Q^p_m$ if $prc_{\mathcal{P},Q^p_m}(n) \in \mathcal{O}(n)$

- $\mathcal{R}$ has quadratic complexity for class $Q^p_m$ if $prc_{\mathcal{P},Q^p_m}(n) \in \mathcal{O}(n^2)$ etc.

- Example (star-program): has linear complexity

- Goal: Re-use existing methodology for termination analysis to analyze complexity as well
\[ P : \quad \text{star}(XS, []) \;::=\; !. \\
\quad \text{star}([], ZS) \;::=\; !, \text{eq}(ZS, []). \\
\quad \text{star}(XS, ZS) \;::=\; \text{app}(XS, YS, ZS), \text{star}(XS, YS). \]

**Diagram:**

- **A:** star(\( \overline{T_1} \), \( \overline{T_2} \))
- **B:** star(\( \overline{T_1} \), \( \overline{T_2} \)\(^{(1)} \)) \| star(\( \overline{T_1} \), \( \overline{T_2} \)\(^{(2)} \)) \| star(\( \overline{T_1} \), \( \overline{T_2} \)\(^{(3)} \))
- **C:** ! \| star(\( \overline{T_1} \), [])\(^{(2)} \) \| star(\( \overline{T_1} \), [])\(^{(3)} \)
- **D:** star(\( \overline{T_1} \), \( \overline{T_2} \)\(^{(2)} \)) \| star(\( \overline{T_1} \), \( \overline{T_2} \)\(^{(3)} \))
- **E:** Cut
- **F:** !, \text{eq}(\( \overline{T_2} \), []) \| star([], \( \overline{T_2} \)\(^{(3)} \))
- **G:** \text{app}(\( \overline{T_1}, T_3, \overline{T_2} \)), star(\( \overline{T_1}, T_3 \))
- **H:** star(\( \overline{T_1}, \overline{T_4} \))

**Inference:**

- Case
- Inst
- Eval
- Cut
- Eval

**Functions:**

- \( f_{in}^{A}(T_1, T_2) \rightarrow u_{A,F}(f_{in}^{F}(T_1, T_2)) \)
- \( u_{A,F}(f_{out}^{F}(T_3)) \rightarrow f_{out}^{A} \)
- \( f_{in}^{A}(T_1, []) \rightarrow f_{out}^{A} \)
- \( f_{in}^{in}(T_1, T_2) \rightarrow u_{F,G}(f_{in}^{in}(T_1, T_2)) \)
- \( u_{F,G}(f_{out}^{G}(T_4)) \rightarrow u_{G,H}(f_{in}^{in}(T_1, T_4), T_4) \)
- \( u_{G,H}(f_{out}^{A}, T_4) \rightarrow f_{out}^{F}(T_4) \)

**Notes:**

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

**Correct?**

depends on Split's successor

in P: repeat evaluation of h for every answer of g (backtracking)

Here: g is deterministic (has only one answer)
\[ P : \begin{align*}
\text{star}(XS, []) & \quad \text{!} . \\
\text{star}([], ZS) & \quad ! , \text{eq}(ZS, []). \\
\text{star}(XS, ZS) & \quad \text{app}(XS, YS, ZS), \text{star}(XS, YS).
\end{align*} \]

**Correct!**
- depends on `Split`'s successor `G`
- in `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)

\[ f_{A,F}^{in}(T_1, T_2) \rightarrow u_{A,F}(f_{F}^{in}(T_1, T_2)) \]
\[ u_{A,F}(f_{F}^{out}(T_3)) \rightarrow f_{A}^{out} \]
\[ f_{A}^{in}(T_1, []) \rightarrow f_{A}^{out} \]
\[ f_{A}^{in}(T_1, T_2) \rightarrow u_{F,G}(f_{G}^{in}(T_1, T_2)) \]
\[ u_{F,G}(f_{G}^{out}(T_4)) \rightarrow u_{G,H}(f_{A}^{in}(T_1, T_4), T_4) \]
\[ u_{G,H}(f_{A}^{out}, T_4) \rightarrow f_{F}^{out}(T_4) \]

**Case**
- `\text{split}(T_1, T_2)`
- `\text{star}(T_1, T_2)`

**Eval**
- `\text{star}(T_1, T_2)^{(1)} | \text{star}(T_1, T_2)^{(2)} | \text{star}(T_1, T_2)^{(3)}`

**Split**
- `\text{app}(T_1, T_3, T_2), \text{star}(T_1, T_3)`
- `\text{app}(T_1, T_3, T_2)`
- `\text{star}(T_1, T_4)`
- `\text{star}(T_1, T_2)^{(3)}`

**Correct!**
- generate symbolic evaluation graph
- generate TRS from graph
- determine complexity of TRS by existing tool
- infer that `P` has the same complexity
\[ P : \quad \text{sublist}(X, Y) \leftarrow \text{app}(P, U, Y), \text{app}(V, X, P). \quad (1) \]
\[ \text{app}([], YS, YS). \quad (2) \]
\[ \text{app}([X | XS], YS, [X | ZS]) \leftarrow \text{app}(XS, YS, ZS). \quad (3) \]

**Evaluation of sublist**

- \( Q_{m}^{\text{sublist}} = \{ \text{sublist}(t_1, t_2) \mid t_2 \text{ ground} \} \)
- computes all sublists of \( Y \)
  (by *backtracking* )
- \( P \):
  - linear many possibilities
    to split \( Y \) into \( P \) and \( U \)
  - for each possible \( P \),
    linear evaluation of \( \text{app}(V, X, P) \)
\[
P : \quad \text{sublist}(X, Y) \leftarrow \text{app}(P, U, Y),
\text{app}([], YS, YS).
\text{app}([X | XS], YS, [X | ZS]) \leftarrow \text{app}(XS, YS, ZS)
\]

Generate symbolic evaluation graph and TRS
Determine complexity of TRS by existing tool
Infer that \( P \) has the same complexity
\[ \mathcal{P} : \text{sublist}(X, Y) \leftarrow \text{app}(P, U, Y), \text{app}(V, X, P). \] (1)

\[ \text{app}([], YS, YS). \] (2)

\[ \text{app}([X | XS], YS, [X | ZS]) \leftarrow \text{app}(XS, YS, ZS) \] (3)

---

**Correctness of Complexity Analysis**

- depends on \( \text{Split} \)'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 \( \Rightarrow \) \( \text{Split} \) node \( B \) is multiplicative

**Symbolic Evaluation:**

\[ f_{B}^{\text{in}}(T_2) \rightarrow u_{B,C}(f_{D}^{\text{in}}(T_2)) \]

\[ u_{B,C}(f_{D}^{\text{out}}(\ldots)) \rightarrow u_{C,D}(f_{D}^{\text{in}}(\ldots)) \]

\[ u_{C,D}(f_{D}^{\text{out}}(\ldots)) \rightarrow f_{B}^{\text{out}}(\ldots) \]

---

- 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 $R_1, \ldots, R_5$ for parts up to multiplicative Split nodes (no multiplicative Split node may reach itself)
- determine $irc_{R_1, R}, \ldots, irc_{R_5, R}$ separately
  - maps $n \in \mathbb{N}$ to maximal number of $R_i$-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
  - $irc_{R_1, R} + irc_{R_2, R} \cdot (irc_{R_3, R} + irc_{R_4, R} \cdot irc_{R_5, R})$
**irc}_R_1, \mathcal{R}_2, \mathcal{R}_3 : \text{constant, linear, linear}\)

- complexity of \(P\): quadratic 

\[ \text{irc}_R_1, \mathcal{R}_2, \mathcal{R}_3 + \text{irc}_R_2, \mathcal{R}_3 \cdot \text{irc}_R_3, \mathcal{R} \]

\[ \text{sublist}(T_1, T_2) \]

\[ \text{app}(T_3, T_4, T_2), \text{app}(T_5, T_1, T_3) \]

\[ \text{inst}(\text{app}(T_5, T_1, T_6)) \]

\[ \text{eval}(\text{app}(T_5, T_1, T_6)) \]

\[ \text{eval}(\text{app}(T_5, T_1, T_6)) \]

\[ \text{split}(\text{app}(T_5, T_1, T_6)) \]

\[ \text{split}(\text{app}(T_5, T_1, T_6)) \]

\[ \text{eval}(\text{app}(T_8, T_1, T_9)) \]

\[ \text{eval}(\text{app}(T_8, T_1, T_9)) \]

- \(f_{in}(T_2) \rightarrow u_{A,B}(f_{in}(T_2))\)
- \(u_{A,B}(f_{out}(\ldots)) \rightarrow f_{out}(T_1)\)
- \(u_{B,C}(f_{out}(\ldots)) \rightarrow f_{B}(\ldots)\)
- \(u_{C,D}(f_{out}(\ldots)) \rightarrow f_{out}(\ldots)\)
- \(f_{in}(T_6) \rightarrow f_{D}(\ldots)\)
- \(f_{D}(T_6) \rightarrow u_{D,G}(f_{G}(T_6))\)
- \(u_{D,G}(f_{out}(\ldots)) \rightarrow f_{D}(T_5, T_1)\)
- \(f_{in}([T_7, T_9]) \rightarrow u_{G,H}(\ldots)\)
- \(u_{G,H}(\ldots) \rightarrow f_{G}(\ldots)\)

- generate graph and TRSs \(\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3\)
- determine \(\text{irc}_{\mathcal{R}_1, \mathcal{R}}, \text{irc}_{\mathcal{R}_2, \mathcal{R}}, \text{irc}_{\mathcal{R}_3, \mathcal{R}}\)
- infer complexity of \(P\)
Symbolic Evaluation Graphs and Term Rewriting

PROLOG-Program $\rightarrow$ Symbolic Evaluation Graph $\rightarrow$ TRS $\rightarrow$ Rewrite Tool (AProVE)↓

Complexity

implemented in tool AProVE

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

<table>
<thead>
<tr>
<th></th>
<th>$O(1)$</th>
<th>$O(n)$</th>
<th>$O(n^2)$</th>
<th>$O(n \cdot 2^n)$</th>
<th>bounds</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASLOG</td>
<td>1</td>
<td>21</td>
<td>4</td>
<td>3</td>
<td>29</td>
<td>14.8</td>
</tr>
<tr>
<td>CiaoPP</td>
<td>3</td>
<td>19</td>
<td>4</td>
<td>3</td>
<td>29</td>
<td>11.7</td>
</tr>
<tr>
<td>AProVE</td>
<td>54</td>
<td>117</td>
<td>37</td>
<td>0</td>
<td>208</td>
<td>10.6</td>
</tr>
</tbody>
</table>
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 $\text{Suc}$ node $s'$, then there is no path from $s'$ to a $\text{Suc}$ 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 $\Rightarrow$ path to $\text{Suc}$ node in symbolic evaluation graph

- C not deterministic $\Rightarrow$ 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.

- **Case A**: $\text{star}(\overline{T_1}, \overline{T_2})$
  - **Eval**: $\text{star}(\overline{T_1}, \overline{T_2})^{(1)} | \text{star}(\overline{T_1}, \overline{T_2})^{(2)} | \text{star}(\overline{T_1}, \overline{T_2})^{(3)}$
- **Case B**: $\text{star}(\overline{T_1}, \overline{T_2})^{(1)} | \text{star}(\overline{T_1}, \overline{T_2})^{(2)} | \text{star}(\overline{T_1}, \overline{T_2})^{(3)}$
  - **Eval**: $\text{star}(\overline{T_1}, \overline{T_2})^{(2)} | \text{star}(\overline{T_1}, \overline{T_2})^{(3)}$
  - **Eval**: $\text{star}(\overline{T_1}, \overline{T_2})^{(3)}$
  - **Eval**: $\text{star}(\overline{T_1}, \overline{T_2})^{(3)}$
- **Case C**: $! | \text{star}(\overline{T_1}, [])^{(2)} | \text{star}(\overline{T_1}, [])^{(3)}$
- **Cut**
- **Suc $\overline{E}$**
- **Cut**
- **Eval**: $\text{!}, \text{eq}(\overline{T_2}, []) | \text{star}([], \overline{T_2})^{(3)}$
- **Eval**: $\text{!}, \text{eq}(\overline{T_2}, []) | \text{star}([], \overline{T_2})^{(3)}$
- **Eval**: $\text{eq}(\overline{T_2}, [])$
- **Case**
- **…**
- **Eval**: $\text{app}(\overline{T_1}, \overline{T_3}, \overline{T_2}), \text{star}(\overline{T_1}, \overline{T_3})$
- **Split**
- **Eval**: $\text{app}(\overline{T_1}, \overline{T_3}, \overline{T_2})$
- **Case**
- **…**
- **Eval**: $\text{app}(\overline{T_1}, \overline{T_3}, \overline{T_2})$
- **Split**
- **Eval**: $\text{star}(\overline{T_1}, \overline{T_4})$
- **Eval**

- **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)
Direct Approaches

- Synthesis of Linear Ranking Functions
  \textit{(Colon & Sipma, 01), (Podelski & Rybalchenko, 04), \ldots}

- Terminator: Termination Analysis by Abstraction & Model Checking
  \textit{(Cook, Podelski, Rybalchenko et al., since 05)}

- Julia & COSTA: Termination Analysis of \textsc{java bytecode}
  \textit{(Spoto, Mesnard, Payet, 10),
  (Albert, Arenas, Codish, Genaim, Puebla, Zanardini, 08)}

- \ldots

- 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 $\Rightarrow$ termination graph
      termination graph captures side effects, sharing, cyclic data objects etc.
    - transform termination graph $\Rightarrow$ 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**

- **our technique:**
  abstract objects to **terms**
  
  - introduce function symbol for every class
  - IntList-object representing \([0, 1, 2]\)
  is abstracted to **term:** \(\text{IntList}(0, \text{IntList}(1, \text{IntList}(2, \text{null})))\)
  
  - 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

PROLOG-Program

HASKELL-Program

Termination Graph

TRS

Termination Tool (AProVE)
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

iffnull 8 | n:o₁, l:o₂ | o₁
o₁ = Int(val = i₁)  i₁ = (−∞, ∞)
o₂ = Int(?)

4 components

1. next program instruction
2. values of local variables
   (value of num is reference o₁)
3. values on the operand stack
4. information about the heap
   - object at address o₂ is null or of type Int
   - object at o₁ has type Int, val-field has value i₁
   - i₁ 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

State $A$:

- do all calls of count terminate?
- num and limit are arbitrary, but distinct Int-objects
From JBC to Termination Graphs

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)
States $E$ and $F$:

- evaluate “ifnull 8” in $C$ and $D$
- evaluation edges
From JBC to Termination Graphs

```
00: aload_0
01: ifnonnull 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)
**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

\[ i_3 = i_1 + \text{iconst}_1 \]
From JBC to Termination Graphs

State $M$:
- again reaches "if_icmpge"
- $M$ instance of $G$
- instantiation edge
Termination Graphs

- expand nodes until all leaves correspond to program ends
- by appropriate generalization steps, one always reaches a \textit{finite} termination graph
- state $s_1$ is \textit{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 \textit{computation path} in the termination graph
- termination graph is called \textit{terminating} iff it has no infinite computation path
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 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 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 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 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 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 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
- $\text{list}(o_1)$ and $\text{cur}(o_2)$ may be equal and may join
public class Flatten {
    public static IntListflatten(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 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 $S$
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 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 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

PROLOG-Program ➔ Termination Graph ➔ TRS ➔ Termination Tool (AProVE)

HASKELL-Program ➔ Termination Graph

JBC-Program ➔ Termination Graph
Transforming Objects to Terms

```plaintext
aload_1 | l:o_1, c:o_9, r:o_8 | ε

\[ o_9 = \text{TreeList}(\text{value} = o_6, \text{next} = o_2) \]
\[ o_2 = \text{TreeList}(\text{value} = o_7, \text{next} = o_5) \]
\[ o_8 = \text{IntList}(\text{value} = i_1, \text{next} = o_3) \]
\[ i_1 = (-\infty, \infty) \]
\[ o_6 = \text{Tree}(\_\_\_\_) \]
\[ o_7 = \text{Tree}(\_\_\_\_) \]
\[ o_1 = \text{TreeList}(\_\_\_) \]
\[ o_5 = \text{TreeList}(\_\_\_) \]
\[ o_3 = \text{IntList}(\_\_\_) \]

\[ o_1 =? o_5 \quad o_1 \lor o_2 \quad o_1 \lor o_5 \quad o_1 \lor o_6 \quad o_1 \lor o_7 \]
```

For every class \( C \) with \( n \) fields, introduce function symbol \( C \) with \( n \) arguments:

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

```java
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 \( x \): \( j\text{O}(A(\text{eoc}, 1)) \) (eoc for “end of class”)
- term for \( y \): \( j\text{O}(A(B(\text{eoc}, 3), 2)) \) (jO for “java.lang.Object”)
Transforming States to Tuples of Terms

Transforming $D$

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

Transforming $F$

$f_F(jlO(\text{Int}(\text{eoc}, i_1)), o_2)$
Transforming Edges to Rewrite Rules

\[ f_D(jlO(Int(eoc, i_1)), o_2, \]
\[ jlO(Int(eoc, i_1))) \]
\[ \rightarrow \]
\[ f_F(jlO(Int(eoc, i_1)), o_2) \]
Transforming Edges to Rewrite Rules

Transforming Evaluation Edges with Conditions

\[ f_H(j\text{O}(\text{Int}(\text{eoc}, i_1)), j\text{O}(\text{Int}(\text{eoc}, i_2)), j\text{O}(\text{Int}(\text{eoc}, i_1)), i_1, i_2) \]

\[ \rightarrow \]

\[ f_J(j\text{O}(\text{Int}(\text{eoc}, i_1)), j\text{O}(\text{Int}(\text{eoc}, i_2)), j\text{O}(\text{Int}(\text{eoc}, i_1))) \]

\[ | i_1 \geq i_2 \]
Transforming Edges to Rewrite Rules

Transforming Refinement Edges

\[ f_B(jlO(\text{Int}(eoc, i_1)), \quad o_2, \quad jlO(\text{Int}(eoc, i_1))) \rightarrow \]

\[ f_D(jlO(\text{Int}(eoc, i_1)), \quad o_2, \quad jlO(\text{Int}(eoc, i_1))) \]

\[ \text{aload} \_0 \mid n:o_1, l:o_2 \mid \varepsilon \]
\[ o_1 = \text{Int}(?) \quad o_2 = \text{Int}(?) \]

\[ \text{ifnull} 
\]
\[ 8 \mid n:o_1, l:o_2 \mid o_1 \]
\[ o_1 = \text{Int}(?) \quad o_2 = \text{Int}(?) \]

\[ \text{aload} \_1 \mid n:o_1, l:o_2 \mid \varepsilon \]
\[ o_1 = \text{Int}(?) \quad o_2 = \text{Int}(?) \]

\[ \text{return} \mid n:\text{null}, l:o_2 \mid \varepsilon \]
\[ o_2 = \text{Int}(?) \]

\[ i_1 \geq i_2 \]

\[ \text{if.icmpge} \ 35 \mid n:o_1, l:o_2, c:o_1 \mid i_1, i_2 \]
\[ o_1 = \text{Int}(\text{val} = i_1) \quad i_1 = (-\infty, \infty) \]
\[ o_2 = \text{Int}(\text{val} = i_2) \quad i_2 = (-\infty, \infty) \]

\[ \text{if.icmpge} \ 35 \mid n:o_1, l:o_2, c:o_1 \mid i_1, i_2 \]
\[ o_1 = \text{Int}(\text{val} = i_1) \quad i_1 = (-\infty, \infty) \]
\[ o_2 = \text{Int}(\text{val} = i_2) \quad i_2 = (-\infty, \infty) \]

\[ \text{if.icmpge} \ 35 \mid n:o_1, l:o_2, c:o_1 \mid i_1, i_2 \]
\[ o_1 = \text{Int}(\text{val} = i_1) \quad i_1 = (-\infty, \infty) \]
\[ o_2 = \text{Int}(\text{val} = i_2) \quad i_2 = (-\infty, \infty) \]

\[ \text{iadd} \mid n:o_1, l:o_2, c:o_1 \mid \text{val}, i_1, \text{iconst}_1 \]
\[ o_1 = \text{Int}(\text{val} = i_1) \quad i_1 = (-\infty, \infty) \]
\[ o_2 = \text{Int}(\text{val} = i_2) \quad i_2 = (-\infty, \infty) \]
\[ \text{iconst}_1 = [1, 1] \]

\[ i_3 = i_1 + \text{iconst}_1 \]
Transforming Edges to Rewrite Rules

Merging Rewrite Rules

\[ f_B(jlO(\text{Int}(eoc, i_1)), o_2, jlO(\text{Int}(eoc, i_1))) \]

\[ f_F(jlO(\text{Int}(eoc, i_1)), o_2) \]
Transforming Edges to Rewrite Rules

TRS for count

\[ f_G(jlO(Int(eoc, i_1))), \quad jlO(Int(eoc, i_2)), \quad jlO(Int(eoc, i_1 + 1)), \quad jlO(Int(eoc, i_2)), \quad jlO(Int(eoc, i_1 + 1)), \quad i_1, \quad i_2 \rightarrow f_G(jlO(Int(eoc, i_1 + 1))), \quad jlO(Int(eoc, i_2)), \quad jlO(Int(eoc, i_1 + 1)), \quad i_1 + 1, \quad i_2) \mid i_1 < i_2 \]

TRS is "natural"

termination easy
to prove
automatically
From Termination Graphs to TRSs

TRS for count

\[\begin{align*}
f_G(\text{jL}(\text{Int}(\text{eoc}, i_1)), \text{jL}(\text{Int}(\text{eoc}, i_2)), \text{jL}(\text{Int}(\text{eoc}, i_1)), i_1, i_2) & \rightarrow \text{jL}(\text{Int}(\text{eoc}, i_1 + 1)), \text{jL}(\text{Int}(\text{eoc}, i_2)), \text{jL}(\text{Int}(\text{eoc}, i_1 + 1)), i_1 + 1, i_2) & | & i_1 < i_2 \end{align*}\]

- 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
Rewrite Rules & Annotations

- when writing to a field of $o_2$ with $o_1 \triangleright o_2$:
  - $o_1$ on lhs, fresh variable $o'_1$ on rhs
- cyclic objects: fresh variable on rhs
From Termination Graphs to TRSs

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*

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
<th>Failure</th>
<th>Timeout</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>AProVE</td>
<td>267</td>
<td>81</td>
<td>11</td>
<td>28</td>
<td>9.5</td>
</tr>
<tr>
<td>Julia</td>
<td>191</td>
<td>22</td>
<td>174</td>
<td>0</td>
<td>4.7</td>
</tr>
<tr>
<td>COSTA</td>
<td>160</td>
<td>0</td>
<td>181</td>
<td>46</td>
<td>11.0</td>
</tr>
</tbody>
</table>

- 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