# Termination Proving for Rewrite Systems and Programs Solutions

Carsten Fuhs

Birkbeck, University of London

$15^{\text{th}}$ International School on Rewriting 2026

Advanced Track

Nijmegen, The Netherlands

15 & 16 July 2026

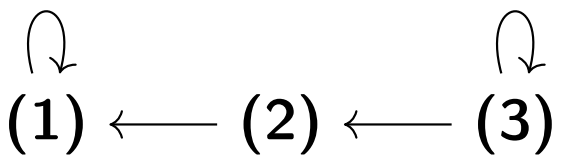
https://www.dcs.bbk.ac.uk/~carsten/isr2026/

# Exercise I.1 (1/2)

The initial set of DP problems is $D = \{(DP(\mathcal{R}), \mathcal{R})\}$ where $DP(\mathcal{R}) = \{\mathbf{(1)}, \mathbf{(2)}, \mathbf{(3)}\}$ with the following DPs:

$$\begin{aligned}
\mathsf{app}^\sharp(\mathsf{add}(n, x), y) &\longrightarrow \mathsf{app}^\sharp(x, y) && \mathbf{(1)}\\
\mathsf{reverse}^\sharp(\mathsf{add}(n, x)) &\longrightarrow \mathsf{app}^\sharp(\mathsf{reverse}(x), \mathsf{add}(n, \mathsf{nil})) && \mathbf{(2)}\\
\mathsf{reverse}^\sharp(\mathsf{add}(n, x)) &\longrightarrow \mathsf{reverse}^\sharp(x) && \mathbf{(3)}
\end{aligned}$$

The Dependency Graph over-approximation for the DP problem $(\{\mathbf{(1)}, \mathbf{(2)}, \mathbf{(3)}\}, \mathcal{R}) \in D$ computed by comparing root symbols of DPs is:




The Dependency Graph processor decomposes this over-approximation of the dependency graph into its non-trivial Strongly Connected Components (SCCs). We update $D$ to $D = \{\ (\{\mathbf{(1)}\}, \mathcal{R}),\ (\{\mathbf{(3)}\}, \mathcal{R})\ \}$.

# Exercise I.1 (2/2)

For the DP problem $(\{\mathbf{(1)}\}, \mathcal{R}) \in D$, we apply the Reduction Pair processor with Usable Rules with the polynomial interpretation $[\cdot]$ with $[\mathsf{app}^\sharp](x_1, x_2) = x_1$ and $[\mathsf{add}](x_1, x_2) = x_2 + 1$. Then $\mathbf{(1)} \in \succ$ holds. No usable rules need to be considered. We replace this DP problem in $D$ by the DP problem $(\emptyset, \mathcal{R})$. The Dependency Graph processor deletes this DP problem from $D$ as it has no non-trivial SCCs.

For the DP problem $(\{\mathbf{(3)}\}, \mathcal{R}) \in D$, we apply the Reduction Pair processor with Usable Rules with the polynomial interpretation $[\cdot]$ with $[\mathsf{reverse}^\sharp](x_1) = x_1$ and $[\mathsf{add}](x_1, x_2) = x_2 + 1$. Then $\mathbf{(3)} \in \succ$ holds. No usable rules need to be considered. We replace this DP problem in $D$ by the DP problem $(\emptyset, \mathcal{R})$. The Dependency Graph processor deletes this DP problem from $D$ as it has no non-trivial SCCs.

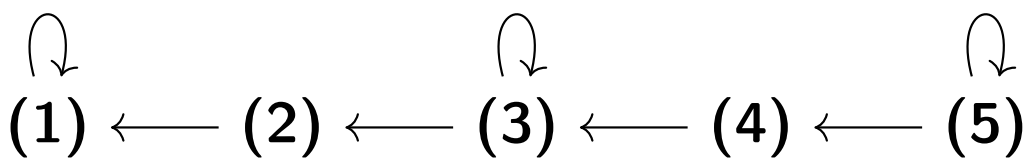
As now $D = \emptyset$, we conclude that $\mathcal{R}$ must be terminating.

# Exercise I.2 (Exercise I.1 continued) (1/2)

The initial set of DP problems is $D = \{(DP(\mathcal{R}), \mathcal{R})\}$ where $DP(\mathcal{R}) = \{\mathbf{(1)}, \mathbf{(2)}, \mathbf{(3)}, \mathbf{(4)}, \mathbf{(5)}\}$ with the DPs $\mathbf{(1)}, \mathbf{(2)}, \mathbf{(3)}$ taken from Exercise I.1 and:

$$\mathsf{shuffle}^\sharp(\mathsf{add}(n, x)) \longrightarrow \mathsf{reverse}^\sharp(x) \qquad \mathbf{(4)}$$
$$\mathsf{shuffle}^\sharp(\mathsf{add}(n, x)) \longrightarrow \mathsf{shuffle}^\sharp(\mathsf{reverse}(x)) \qquad \mathbf{(5)}$$

The Dependency Graph over-approximation for the DP problem $(\{\mathbf{(1)}, \mathbf{(2)}, \mathbf{(3)}, \mathbf{(4)}, \mathbf{(5)}\}, \mathcal{R}) \in D$ computed by comparing root symbols of DPs is:




The Dependency Graph processor decomposes this over-approximation of the dependency graph into its non-trivial Strongly Connected Components (SCCs). We update $D$ to $D = \{\ (\{\mathbf{(1)}\}, \mathcal{R}),\ (\{\mathbf{(3)}\}, \mathcal{R}),\ (\{\mathbf{(5)}\}, \mathcal{R})\ \}$.

# Exercise I.2 (Exercise I.1 continued) (2/2)

We deal with the DP problems $(\{\textbf{(1)}\}, \mathcal{R}) \in D$ and $(\{\textbf{(3)}\}, \mathcal{R}) \in D$ in an analogous way to Exercise I.1. Note that in the two exercises, $\mathcal{R}$ stands for different TRSs, but these two DP problems still have no usable rules.

For the DP problem $(\{\textbf{(5)}\}, \mathcal{R}) \in D$, we apply the Reduction Pair processor with Usable Rules with the polynomial interpretation $[\cdot]$ with $[\mathsf{shuffle}^\sharp](x_1) = [\mathsf{reverse}](x_1) = x_1$, $[\mathsf{app}](x_1, x_2) = x_1 + x_2$, $[\mathsf{nil}] = 0$, and $[\mathsf{add}](x_1, x_2) = x_2 + 1$. Then $\textbf{(5)} \in \succ$ holds. The usable rules are the rules for reverse and for app, and they are oriented with $\succsim$. We replace this DP problem in $D$ by the DP problem $(\emptyset, \mathcal{R})$. The Dependency Graph processor deletes this DP problem from $D$ as it has no non-trivial SCCs.

As now $D = \emptyset$, we conclude that $\mathcal{R}$ must be terminating.

# Exercise I.3

For use of the order $>$ for termination proving, the order would have to be well founded. But we have the infinite sequence

$$1 > \frac{1}{2} > \frac{1}{4} > \frac{1}{8} > \cdots > \frac{1}{2^k} > \frac{1}{2^{k+1}} > \cdots$$

contradicting well-foundedness of the order $>$.

As a remedy, we define $s \succ t$ as $[s] >_\delta [t]$, where $[s] >_\delta [t]$ iff $[s] \geq [t] + \delta$, and where $\delta > 0$ is fixed.

In practice, we do not need to search for $\delta$ explicitly. If we are trying to orient only finitely many term constraints $s_i \succ t_i$, we search for an interpretation $[\cdot]$ in the same way as for polynomial interpretations over $\mathbb{N}$, but allowing for solutions from $\mathbb{Q}_{\geq 0}$ for the parameters.

When we have found such an interpretation, we consider all cases of $[s_i] > [t_i]$. In each of these cases, we have $[s_i] = [t_i] + \delta_i$ for some $\delta_i > 0$. To define a well-founded order $\succ$ that orients the same $s_i \succ t_i$, we now set $\delta$ to the *minimum* of all of the (finitely many!) values $\delta_i$, and we are done.

# Exercise I.4

Our first ranking function is $[\cdot]$ with $[\ell_0](x, y) = 1$ and $[\ell_1](x, y) = 0$. Then, for all rules $l \longrightarrow r\ [\varphi]$, we have $[l] \geq [r]$, and for the second rule we additionally have $[l] > [r]$ and $[l] \geq 0$. Thus, we delete the second rule, and only the first rule remains to be analysed.

Our second ranking function is $[\cdot]$ with $[\ell_0](x, y) = y - x$. Then, for our remaining rule $l \longrightarrow r\ [\varphi]$, we have $[l] \geq [r]$, $[l] > [r]$, and $[l] \geq 0$. Thus, we delete the first rule, and no rules remain to be analysed.

This allows us to conclude that the given program is indeed terminating.

# Exercise I.5 (1/3)

1. The term $\ell_1(0,1)$ is non-terminating: $\ell_1(0,1) \longrightarrow \ell_1(0,1) \longrightarrow \cdots$
2. By looking very carefully at the given program, we notice that in the inner loop, the value of x is always strictly greater than 0. The reason is that since we entered the body of the outer loop, the check $x > 0$ must have succeeded. In the loop body of the outer loop up to the end of the inner loop, the value of x is not changed. We get the following rules:

$$
\begin{array}{rcll}
\ell_0(x) & \longrightarrow & \ell_1(x, x*x) & [x > 0] \quad \textbf{(1)} \\
\ell_1(x,y) & \longrightarrow & \ell_1(x, y-x) & [y > 0 \wedge x > 0] \quad \textbf{(2)} \\
\ell_1(x,y) & \longrightarrow & \ell_2(x,y) & [y \leq 0 \wedge x > 0] \quad \textbf{(3)} \\
\ell_2(x,y) & \longrightarrow & \ell_0(x-1) & [x > 0] \quad \textbf{(4)} \\
\ell_0(x) & \longrightarrow & \ell_3(x) & [x \leq 0] \quad \textbf{(5)}
\end{array}
$$

*Additional exercise after Day 2: Can you use the construction of the symbolic execution graph to find the invariant in a systematic way?*

# Exercise I.5 (2/3)

Now that we have found (hopefully suitable) invariants, we can start with the search for ranking functions.

Our first ranking function is $[\cdot]$ with $[\ell_0](x) = [\ell_1](x,y) = [\ell_2](x,y) = 1$ and $[\ell_3](x,y) = 0$. Then, for all rules $l \longrightarrow r\ [\varphi]$, we have $[l] \geq [r]$, and for rule **(5)** we additionally have $[l] > [r]$ and $[l] \geq 0$. Thus, we delete rule **(5)**, and only rules **(1)**, **(2)**, **(3)**, **(4)** remain to be analysed.

Our second ranking function is $[\cdot]$ with $[\ell_0](x) = [\ell_1](x,y) = [\ell_2](x,y) = x$. Then, for all rules $l \longrightarrow r\ [\varphi]$, we have $[l] \geq [r]$, and for rule **(4)** we additionally have $[l] > [r]$ and $[l] \geq 0$. Thus, we delete rule **(4)**, and only the rules **(1)**, **(2)**, **(3)** remain to be analysed. (Intuitively, we have now shown that there cannot be infinitely many iterations of the *outer* loop.)

Our third ranking function is $[\cdot]$ with $[\ell_0](x) = 2$, $[\ell_1](x,y) = 1$, $[\ell_2](x,y) = 0$. Then, for all rules $l \longrightarrow r\ [\varphi]$, we have $[l] \geq [r]$, and for rules **(1)** and **(3)** we additionally have $[l] > [r]$ and $[l] \geq 0$. Thus, we delete rules **(1)** and **(3)**, and only rule **(1)** remains to be analysed.

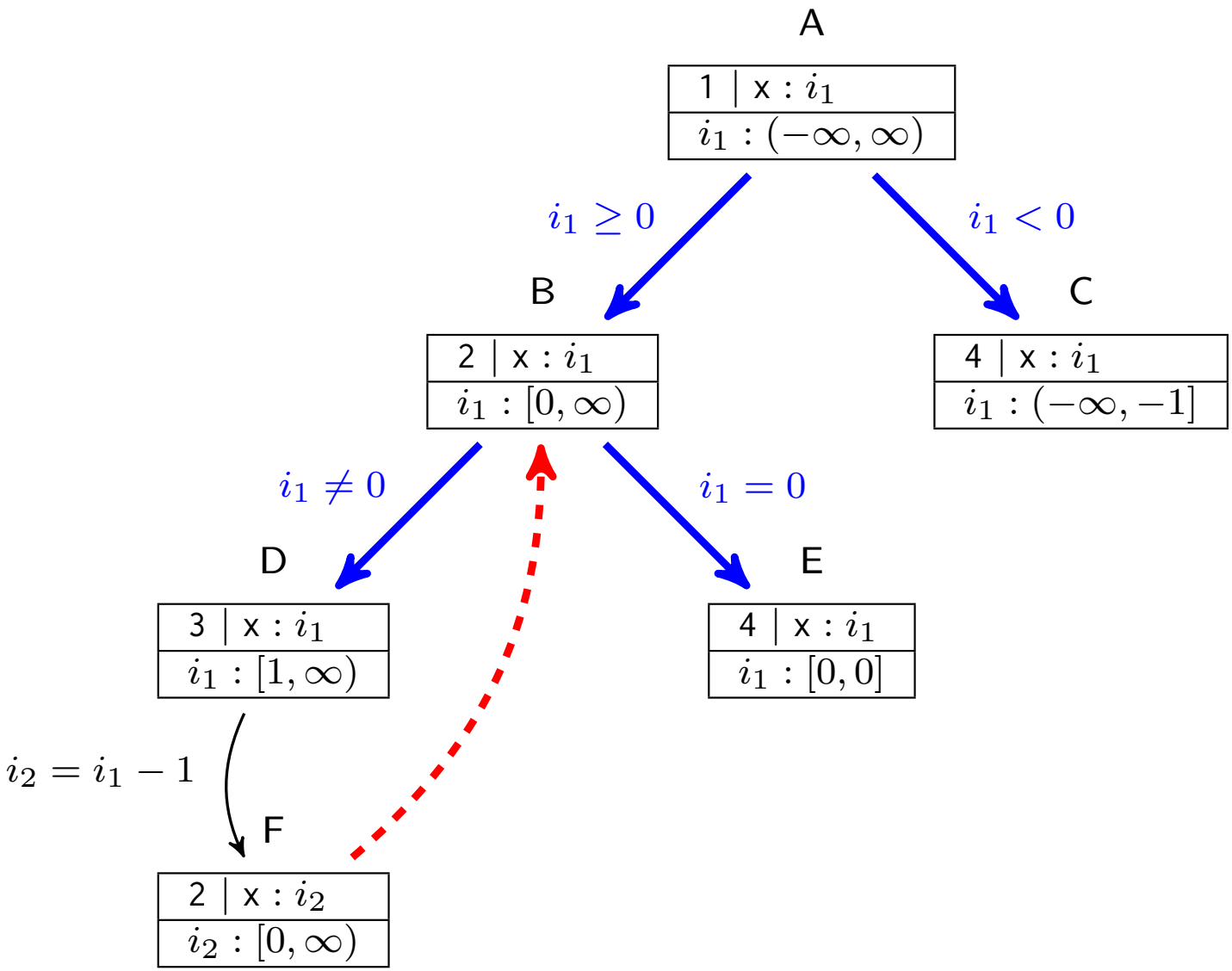
# Exercise I.5 (3/3)

Our fourth ranking function is $[\cdot]$ with $[\ell_1](x, y) = y$. Then, for the only rule **(2)**, we have $[l] \geq [r]$, $[l] > [r]$, and $[l] \geq 0$. Thus, we delete rule **(2)**, and no rules remain to be analysed. (Intuitively, we have now shown that there cannot be infinitely many iterations of the *inner* loop.)

This allows us to conclude that the given program is indeed terminating.

# Exercise II.1 (1/2)

We get the following termination graph for the program:



# Exercise II.1 (2/2)

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

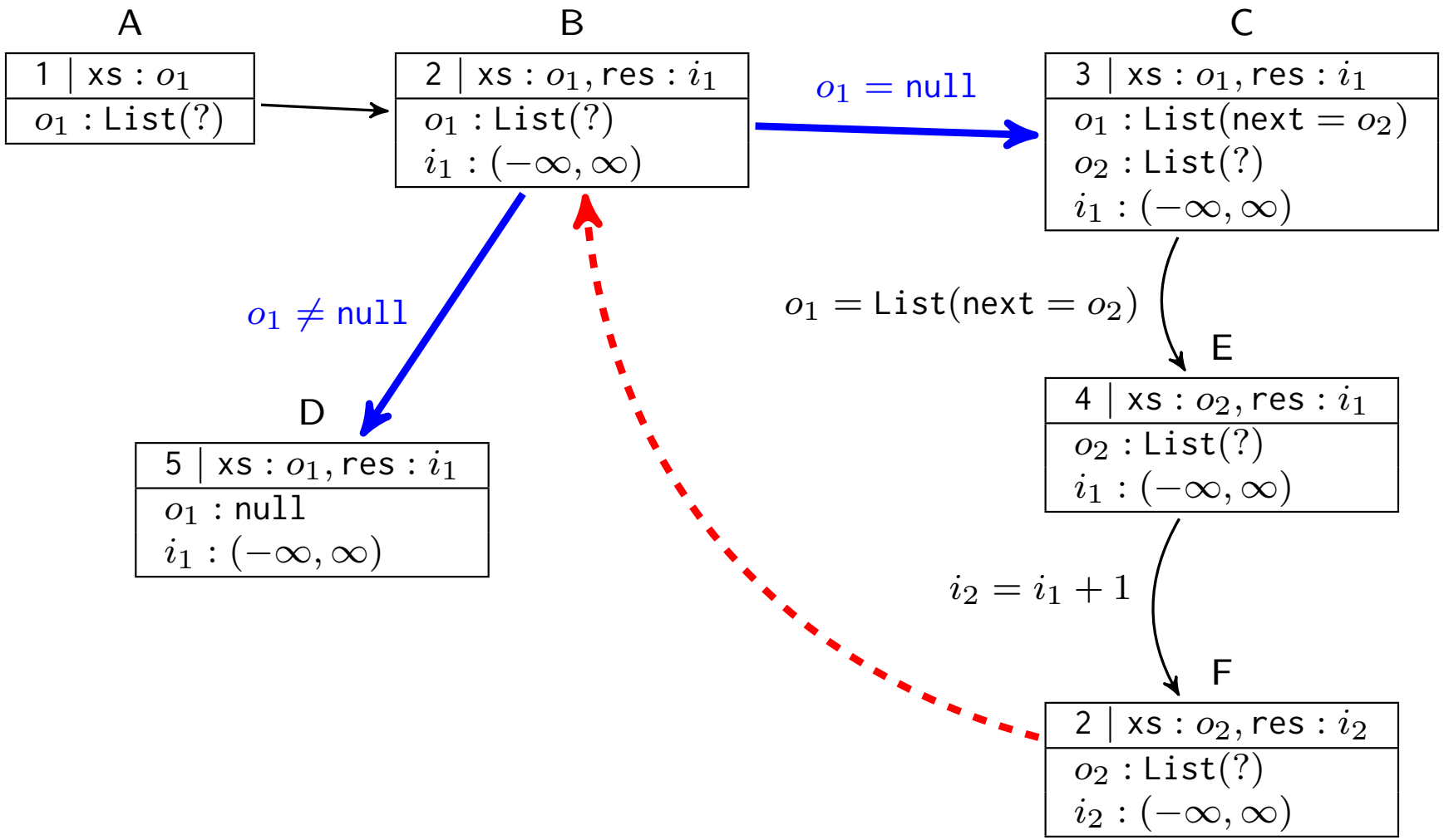
$$
\begin{array}{lcll}
\ell_{\mathsf{A}}(i_1) & \longrightarrow & \ell_{\mathsf{B}}(i_1) & [i_1 \geq 0] \\
\ell_{\mathsf{A}}(i_1) & \longrightarrow & \ell_{\mathsf{C}}(i_1) & [i_1 < 0] \\
\ell_{\mathsf{B}}(i_1) & \longrightarrow & \ell_{\mathsf{D}}(i_1) & [i_1 \geq 0 \wedge i_1 \neq 0] \\
\ell_{\mathsf{D}}(i_1)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

Apart from the different names for function symbols and variables, we get essentially the same result as on slide 29, with two differences:

- Instead of $\ell_3$, we now have the two different end-of-program symbols $\ell_{\mathsf{C}}$ and $\ell_{\mathsf{E}}$.
- The second-to-last rule has $i_1 \geq 1$ as its condition, which is stronger than $i_1 \geq 0$ (i.e., $i_1 \geq 1$ implies $i_1 \geq 0$).

# Exercise II.2 (1/3)

We get the following termination graph for the program:



# Exercise II.2 (2/3)

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

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

The dependency pairs for these rules are identical to the rules (except that we add a $\sharp$ to the defined function symbols). The Reduction Pair processor with Usable Rules with the polynomial interpretation $[\cdot]$ with

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

orients all DPs with $\succ$, and there are no usable rules that would need to be oriented. The Dependency graph processor deletes the resulting DP problem, and termination is proved.

# Exercise II.2 (3/3)

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

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

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

# Exercise II.3 (1/2)

1. Consider the following program:

   **while** $(\mathsf{x} \neq 0)$
   $\quad \mathsf{x} = \mathsf{x} - 1;$

   Over (signed) 32-bit machine integers, as used for Java's type `int`, the loop condition becomes false in at most $2^{32} - 1$ loop iterations. In this setting, the value $0$ for x *will* eventually be reached.

   However, if we consider this program over $\mathbb{Z}$, the program does not terminate from the initial value $-1$ for x.

2. Consider the following program, where $2147483647 = 2^{31} - 1$:

   **while** $(\mathsf{x} \leq 2147483647)$
   $\quad \mathsf{x} = \mathsf{x} + 1;$

   On $\mathbb{Z}$, the program terminates because x *will* eventually reach a value of (at least) $2^{31}$, and the loop condition becomes false.

   On Java's `int`, the program does not terminate since $\mathsf{x} \leq 2147483647$ holds for all `int` values x ($2147483647$ is the largest `int` value).

# Exercise II.3 (2/2)

3. There are (at least) two options:

   (a) Adapt the backend. The frontend generates the same constrained rewrite rules as in the current version. However, we consider the arithmetic for the generated rewrite rules for signed 32-bit bitvector arithmetic rather than for $\mathbb{Z}$. Bitvector arithmetic is another prominent theory supported by SMT solvers. Note that if all variables are bitvectors, termination becomes decidable as the state space is finite!

   (b) Adapt the frontend. Whenever an operation on `int` is performed, add constraints to ensure that the result of the operation is at least $-2^{31}$ and at most $2^{31} - 1$. This may involve suitable operations using Java's % ("modulo") operator. Then the frontend can still generate constrained rewrite rules over the integers, and the backend for constraints over $\mathbb{Z}$ continues to be applicable.

   This approach is pursued, e.g., by [Hensel et al, *JLAMP '18*].

# Exercise II.4

For runtime complexity (innermost or unrestricted), we consider only *basic terms* as start terms. For the given TRS (and its implicit signature), basic terms all have the shape $\mathsf{a}(\mathsf{b}^n(x))$ for some $n \in \mathbb{N}$. None of these terms can be rewritten using the provided rewrite rule, which requires at least two occurrences of the defined symbol a in a redex. Thus, all basic terms are normal forms! This means that the runtime complexity for this TRS is $0$.

The example illustrates that the notion of runtime complexity is geared towards functional programs, where all left-hand sides of rules correspond to basic terms (no defined symbols below the root of the term).

As an aside, let us talk about the derivational complexity of this TRS. Recall that in the course on tree automata, we have seen a termination proof for this TRS using match-bounds. This proof implies a *linear* upper bound on the derivational complexity of the TRS.

# Exercise II.5 (1/3)

We compute the set $\mathcal{S}$ of the following Dependency Tuples for this TRS:

$$\begin{aligned}
\mathsf{half}^\sharp(0) &\to \mathsf{Com}_0 && \textbf{(1)}\\
\mathsf{half}^\sharp(\mathsf{s}(\mathsf{s}(x))) &\to \mathsf{Com}_1(\mathsf{half}^\sharp(x)) && \textbf{(2)}\\
\mathsf{log}^\sharp(\mathsf{s}(0)) &\to \mathsf{Com}_0 && \textbf{(3)}\\
\mathsf{log}^\sharp(\mathsf{s}(\mathsf{s}(x))) &\to \mathsf{Com}_2(\mathsf{log}^\sharp(\mathsf{s}(\mathsf{half}(x))), \mathsf{half}^\sharp(x)) && \textbf{(4)}
\end{aligned}$$

We use the polynomial interpretation $[\cdot]$ with $[0] = 1$, $[\mathsf{s}](x) = x + 1$, $[\mathsf{half}](x) = [\mathsf{half}^\sharp](x) = x$, and $[\mathsf{half}^\sharp](x) = x^2$.

This gives us the following polynomial inequalities:

$$\begin{aligned}
1 &> 0 && \textbf{(1')}\\
x + 2 &> x && \textbf{(2')}\\
4 &> 0 && \textbf{(3')}\\
x^2 + 4x + 4 &> x^2 + 3x + 1 && \textbf{(4')}
\end{aligned}$$

All these inequalities hold over $\mathbb{N}$, and also the half-rules as the usable rules are oriented with $\succsim$, so we delete all DTs from $\mathcal{S}$.

The maximum degree of the $[f^\sharp]$ interpretations is 2, so the innermost runtime complexity of the given TRS is in $\mathcal{O}(n^2)$.

# Exercise II.5 (2/3)

To *find* this polynomial interpretation without using a constraint solver, we can use the following intuitive considerations:

- Only the rules for the function symbol half are usable. The rules for log are not called from the DTs, neither directly nor indirectly.
- The interpretation for the defined symbol half is intuitively an upper bound for the *size* of the result of the function half. We want this upper bound to be as tight as possible. This will help us to find also tight bounds for the runtime complexity of the TRS. If the function indeed computes half the value of its argument, we can try the smallest polynomial on $\mathbb{N}$ that is at least $\frac{1}{2} \cdot x$: the polynomial $x$.
- The symbol 0 occurs only on left-hand sides in **(1)** and **(3)**. We may try a positive value for its interpretation to help us orient these DTs.
- If we want to have a chance at orienting **(2)**, we need the interpretation of s to add a constant and to regard its argument.

# Exercise II.5 (3/3)

- The function $\mathsf{half}^\sharp$ should need only linear runtime: it removes two constructors in each recursive call, and no helper functions are used in the recursive call.
- The function $\mathsf{log}^\sharp$ calls itself recursively on a value for whose size we have a linear bound (from our earlier considerations for half), and it calls the function $\mathsf{half}^\sharp$ (which itself has linear cost) in each of these recursive calls. So $\mathsf{log}^\sharp$ calls $\mathsf{half}^\sharp$ linearly often, and *each* of these calls has linear cost.

  Together, this gives a quadratic runtime bound for $\mathsf{log}^\sharp$ ($x \cdot x = x^2$), and we may try to verify this using a quadratic interpretation.