Proving Termination of Integer Term Rewriting*

C. Fuhs[1], J. Giesl[1], M. Plücker[1], P. Schneider-Kamp[2], and S. Falke[3]

[1] LuFG Informatik 2, RWTH Aachen University, Germany
[2] Dept. of Mathematics & CS, University of Southern Denmark, Odense, Denmark
[3] CS Department, University of New Mexico, Albuquerque, NM, USA

Abstract. When using rewrite techniques for termination analysis of programs, a main problem are pre-defined data types like integers. We extend term rewriting by built-in integers and adapt the dependency pair framework to prove termination of *integer term rewriting* automatically.

1 Introduction

Recently, techniques and tools from term rewriting have been successfully applied to prove termination automatically for different programming languages, cf. e.g. [14, 27]. The advantage of rewrite techniques is that they are very powerful for algorithms on user-defined data structures, since they can automatically generate suitable well-founded orders comparing arbitrary forms of terms. But in contrast to techniques for termination of imperative programs (e.g., [2–8, 24, 25]),[4] the drawback of rewrite techniques is that they do not support data structures like integer numbers which are pre-defined in almost all programming languages. Up to now, integers have to be represented as terms, e.g., using the symbols $\mathsf{0}$ for zero, s for the successor function, and pos and neg to convert natural to integer numbers. Then the integers 1 and -2 are represented by the terms $\mathsf{pos}(\mathsf{s}(\mathsf{0}))$ resp. $\mathsf{neg}(\mathsf{s}(\mathsf{s}(\mathsf{0})))$ and one has to add rules for pre-defined operations like $+$, $-$, $*$, $/$, $\%$ that operate on these terms. This representation leads to efficiency problems for large numbers and it makes termination proofs difficult. Therefore up to now, termination tools for term rewrite systems (TRSs) were not very powerful for algorithms on integers, cf. Sect. 6. Hence, an extension of TRS termination techniques to built-in data structures is one of the main challenges in the area [26].

To solve this challenge, we extend[5] TRSs by built-in integers in Sect. 2 and adapt the popular dependency pair (DP) framework for termination of TRSs to integers in Sect. 3. This combines the power of TRS techniques on user-defined data types with a powerful treatment of pre-defined integers. In Sect. 4, we improve the main *reduction pair processor* of the adapted DP framework by considering *conditions* and show how to simplify the resulting conditional constraints. Sect. 5 explains how to transform these conditional constraints into Diophantine

* In *Proc. RTA'09*, LNCS. Supported by the DFG grant GI 274/5-2 and G.I.F. grant 966-116.6.

[4] Moreover, integers were also studied in termination analysis for logic programs [28].

[5] First steps in this direction were done in [9], but [9] only integrated natural instead of integer numbers, which is substantially easier. Moreover, [9] imposed several restrictions (e.g., they did not integrate multiplication and division of numbers and disallowed conditions with mixtures of pre-defined and user-defined functions).

constraints in order to generate suitable orders for termination proofs of integer TRSs (ITRSs). Sect. 6 evaluates our implementation in the prover AProVE [15].

2 Integer Term Rewriting

To handle integers in rewriting, we now represent each integer by a pre-defined constant of the same name. So the signature is split into two disjoint subsets $\mathcal{F}$ and $\mathcal{F}_{int}$. $\mathcal{F}_{int}$ contains the integers $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$, the Boolean values $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$, and pre-defined operations. These operations are classified into *arithmetic operations* like $+$ which yield an integer when applied to integers, *relational operations* like $>$ which yield true or false when applied to integers, and *Boolean operations* like $\wedge$ which yield true or false when applied to Booleans.

Every ITRS implicitly contains an infinite set of pre-defined rules $\mathcal{PD}$ in order to evaluate the pre-defined operations on integers and Booleans. For example, the set $\mathcal{PD}$ contains the rules $2{*}21 \to 42$, $42 \geqslant 23 \to \mathsf{true}$, and $\mathsf{true}\wedge\mathsf{false} \to \mathsf{false}$.

These pre-defined operations can only be evaluated if both their arguments are integers resp. Booleans. So terms like $1 + x$ and $1 + \mathsf{true}$ are normal forms. Moreover, "$t/0$" and "$t\,\%\,0$" are also normal forms for all terms t. As in most programming languages, an ITRS $\mathcal{R}$ may not have rules $\ell \to r$ where ℓ contains pre-defined operations or where $\ell \in \mathbb{Z} \cup \mathbb{B}$. The rewrite relation for an ITRS $\mathcal{R}$ is defined by simply considering innermost[6] rewriting with the TRS $\mathcal{R} \cup \mathcal{PD}$.

Definition 1 (ITRS). *Let $\mathcal{A}rith\mathcal{O}p = \{+, -, *, /, \%\}$, $\mathcal{R}el\mathcal{O}p = \{>, \geqslant, <, \leqslant, ==, !=\}$, and $\mathcal{B}ool\mathcal{O}p = \{\wedge, \Rightarrow\}$.[7] Moreover, $\mathcal{F}_{int} = \mathbb{Z} \cup \mathbb{B} \cup \mathcal{A}rith\mathcal{O}p \cup \mathcal{R}el\mathcal{O}p \cup \mathcal{B}ool\mathcal{O}p$. An* ITRS *$\mathcal{R}$ is a (finite) TRS over $\mathcal{F} \uplus \mathcal{F}_{int}$ where for all rules $\ell \to r$, we have $\ell \in \mathcal{T}(\mathcal{F} \cup \mathbb{Z} \cup \mathbb{B}, \mathcal{V})$ and $\ell \notin \mathbb{Z} \cup \mathbb{B}$. As usual, $\mathcal{V}$ contains all variables. The rewrite relation $\hookrightarrow_{\mathcal{R}}$ of an ITRS $\mathcal{R}$ is defined as $\overset{\mathsf{i}}{\to}_{\mathcal{R}\cup\mathcal{PD}}$, where*

$$\begin{aligned}\mathcal{PD} = \quad & \{n \circ m \to q \mid n, m, q \in \mathbb{Z},\ n \circ m = q,\ \circ \in \mathcal{A}rith\mathcal{O}p\} \\ \cup\, & \{n \circ m \to q \mid n, m \in \mathbb{Z},\ q \in \mathbb{B},\ n \circ m = q,\ \circ \in \mathcal{R}el\mathcal{O}p\} \\ \cup\, & \{n \circ m \to q \mid n, m, q \in \mathbb{B},\ n \circ m = q,\ \circ \in \mathcal{B}ool\mathcal{O}p\}\end{aligned}$$

For example, consider the ITRSs $\mathcal{R}_1 = \{(1), (2), (3)\}$ and $\mathcal{R}_2 = \{(4), (5), (6)\}$. Here, $\mathsf{sum}(x, y)$ computes $\sum_{i=y}^{x} i$ and $\mathsf{log}(x, y)$ computes $\lfloor log_y(x) \rfloor$.

[6] In this paper, we restrict ourselves to innermost rewriting for simplicity. This is not a severe restriction as innermost termination is equivalent to full termination for non-overlapping TRSs and moreover, many programming languages already have an innermost evaluation strategy. Even for lazy languages like Haskell, with the translation of programs to TRSs in [14], it suffices to show innermost termination.

[7] Of course, one could easily include additional junctors like $\vee$ or $\neg$ in $\mathcal{B}ool\mathcal{O}p$. Moreover, one could also admit ITRSs with conditions and indeed, our implementation also works on *conditional* ITRSs. This is no additional difficulty, because conditional (I)TRSs can be automatically transformed into unconditional ones [23]. E.g., the ITRS $\mathcal{R}_1$ below could result from the transformation of this conditional ITRS:

$$\begin{array}{ll}\mathsf{sum}(x, y) \to y + \mathsf{sum}(x, y + 1) & \mid\ x \geqslant y\ \to^* \mathsf{true} \\ \mathsf{sum}(x, y) \to 0 & \mid\ x \geqslant y\ \to^* \mathsf{false}\end{array}$$

$$\begin{array}{lrlr}
\mathsf{sum}(x,y) \to \mathsf{sif}(x \geqslant y, x, y) & (1) & \mathsf{log}(x,y) \to \mathsf{lif}(x \geqslant y \wedge y > 1, x, y) & (4)\\
\mathsf{sif}(\mathsf{true},x,y) \to y + \mathsf{sum}(x, y+1) & (2) & \mathsf{lif}(\mathsf{true},x,y) \to 1 + \mathsf{log}(x/y, y) & (5)\\
\mathsf{sif}(\mathsf{false},x,y) \to 0 & (3) & \mathsf{lif}(\mathsf{false},x,y) \to 0 & (6)
\end{array}$$

The term $\mathsf{sum}(1,1)$ can be rewritten as follows (redexes are underlined):

$$\begin{aligned}
\underline{\mathsf{sum}(1,1)} &\hookrightarrow_{\mathcal{R}_1} \mathsf{sif}(\underline{1 \geqslant 1}, 1, 1) \hookrightarrow_{\mathcal{R}_1} \underline{\mathsf{sif}(\mathsf{true},1,1)} \hookrightarrow_{\mathcal{R}_1} 1 + \mathsf{sum}(1, \underline{1+1})\\
&\hookrightarrow_{\mathcal{R}_1} 1 + \underline{\mathsf{sum}(1,2)} \hookrightarrow_{\mathcal{R}_1} 1 + \mathsf{sif}(\underline{1 \geqslant 2}, 1, 2) \hookrightarrow_{\mathcal{R}_1} 1 + \underline{\mathsf{sif}(\mathsf{false},1,2)}\\
&\hookrightarrow_{\mathcal{R}_1} \underline{1+0} \hookrightarrow_{\mathcal{R}_1} 1
\end{aligned}$$

3 Integer Dependency Pair Framework

The *DP framework* [1, 12, 13, 16, 19] is one of the most powerful and popular methods for automated termination analysis of TRSs and the DP technique is implemented in almost all current TRS termination tools. Our goal is to extend the DP framework in order to handle ITRSs. The main problem is that proving innermost termination of $\mathcal{R} \cup \mathcal{PD}$ *automatically* is not straightforward, as the TRS $\mathcal{PD}$ is infinite. Therefore, we will not consider the rules $\mathcal{PD}$ explicitly, but integrate their handling in the different processors of the DP framework instead.

Of course, the resulting method should be as powerful as possible for term rewriting on integers, but at the same time it should have the full power of the original DP framework when dealing with other function symbols. In particular, if an ITRS does not contain any symbols from $\mathcal{F}_{int}$, then our new variant of the DP framework coincides with the existing DP framework for ordinary TRSs.

As usual, the *defined* symbols $\mathcal{D}$ are the root symbols of left-hand sides of rules. All other symbols are *constructors*. For an ITRS $\mathcal{R}$, we consider all rules in $\mathcal{R} \cup \mathcal{PD}$ to determine the defined symbols, i.e., here $\mathcal{D}$ also includes $\mathcal{ArithOp} \cup \mathcal{RelOp} \cup \mathcal{BoolOp}$. Nevertheless, we ignore these symbols when building DPs, since these DPs would never be the reason for non-termination.[8]

Definition 2 (DP). *For all $f \in \mathcal{D} \setminus \mathcal{F}_{int}$, we introduce a fresh* tuple symbol $f^\sharp$ *with the same arity, where we often write F instead of $f^\sharp$. If $t = f(t_1, ..., t_n)$, let $t^\sharp = f^\sharp(t_1, ..., t_n)$. If $\ell \to r \in \mathcal{R}$ for an ITRS $\mathcal{R}$ and t is a subterm of r with* $\mathrm{root}(t) \in \mathcal{D} \setminus \mathcal{F}_{int}$*, then $\ell^\sharp \to t^\sharp$ is a* dependency pair *of $\mathcal{R}$. $DP(\mathcal{R})$ is the set of all DPs.*

For example, we have $DP(\mathcal{R}_1) = \{(7), (8)\}$ and $DP(\mathcal{R}_2) = \{(9), (10)\}$, where

$$\begin{array}{lrlr}
\mathsf{SUM}(x,y) \to \mathsf{SIF}(x \geqslant y, x, y) & (7) & \mathsf{LOG}(x,y) \to \mathsf{LIF}(x \geqslant y \wedge y > 1, x, y) & (9)\\
\mathsf{SIF}(\mathsf{true},x,y) \to \mathsf{SUM}(x, y+1) & (8) & \mathsf{LIF}(\mathsf{true},x,y) \to \mathsf{LOG}(x/y, y) & (10)
\end{array}$$

The main result of the DP method for innermost termination states that a TRS $\mathcal{R}$ is innermost terminating iff there is no infinite innermost $DP(\mathcal{R})$-*chain*. This can be adapted to ITRSs in a straightforward way. For any TRS $\mathcal{P}$ and ITRS $\mathcal{R}$, a $\mathcal{P}$*-chain* is a sequence of variable renamed pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ from $\mathcal{P}$ such that there is a substitution σ (with possibly infinite domain) where $t_i\sigma \hookrightarrow_{\mathcal{R}}^* s_{i+1}\sigma$ and $s_i\sigma$ is in normal form w.r.t. $\hookrightarrow_{\mathcal{R}}$, for all i. Then we

[8] Formally, they would never occur in any infinite *chain* and could easily be removed by standard techniques like the so-called *dependency graph* [1, 12].

immediately get the following corollary from the standard results on DPs.[9]

Corollary 3 (Termination Criterion for ITRSs). *An ITRS $\mathcal{R}$ is terminating (w.r.t. $\hookrightarrow_{\mathcal{R}}$) iff there is no infinite $DP(\mathcal{R})$-chain.*

Termination techniques are now called *DP processors* and they operate on sets of DPs (called *DP problems*).[10] A DP processor *Proc* takes a DP problem as input and returns a set of new DP problems which have to be solved instead. *Proc* is *sound* if for all DP problems $\mathcal{P}$ with an infinite $\mathcal{P}$-chain there is also a $\mathcal{P}' \in Proc(\mathcal{P})$ with an infinite $\mathcal{P}'$-chain. Soundness of processors is required to prove termination and to conclude that there is no infinite $\mathcal{P}$-chain if $Proc(\mathcal{P}) = \varnothing$.

So termination proofs in the DP framework start with the initial DP problem $DP(\mathcal{R})$. Then the DP problem is simplified repeatedly by sound DP processors. If all resulting DP problems have been simplified to $\varnothing$, then termination is proved. Many processors (like the well-known *(estimated) dependency graph processor* [1, 12, 13], for example) do not rely on the rules of the TRS, but just on the DPs and on the defined symbols. Therefore, they can also be directly applied for ITRSs, since the sets of DPs and of defined symbols are finite and one does not have to consider the infinitely many rules in $\mathcal{PD}$. One just has to take into account that the defined symbols also include $\mathcal{A}rith\mathcal{O}p \cup \mathcal{R}el\mathcal{O}p \cup \mathcal{B}ool\mathcal{O}p$.

But an adaption is non-trivial for one of the most important processors, the *reduction pair processor*. Thus, the main contribution of the paper is to adapt this processor to obtain a powerful automated termination method for ITRSs.

For a DP problem $\mathcal{P}$, the reduction pair processor generates constraints which should be satisfied by a suitable order on terms. In this paper, we consider orders based on *integer*[11] *max-polynomial interpretations* [11, 17]. Such interpretations suffice for most algorithms typically occurring in practice. The set of *max-polynomials* is the smallest set containing the integers $\mathbb{Z}$, the variables, and $p + q$, $p * q$, and $\max(p, q)$ for all max-polynomials p and q. An *integer max-polynomial interpretation* $\mathcal{P}ol$ maps every[12] n-ary function symbol f to a max-polynomial $f_{\mathcal{P}ol}$ over n variables $x_1, \ldots, x_n$. This mapping is extended to terms by defining $[x]_{\mathcal{P}ol} = x$ for all variables x and by letting $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = f_{\mathcal{P}ol}([t_1]_{\mathcal{P}ol}, \ldots, [t_n]_{\mathcal{P}ol})$. One now defines $s \succ_{\mathcal{P}ol} t$ (resp. $s \succsim_{\mathcal{P}ol} t$) iff $[s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol}$ (resp. $[s]_{\mathcal{P}ol} \geqslant [t]_{\mathcal{P}ol}$) holds for all instantiations of the variables with integer numbers.

[9] For Cor. 3, it suffices to consider only *minimal* chains where all $t_i\sigma$ are $\hookrightarrow_{\mathcal{R}}$-terminating [13]. All results of this paper also hold for *minimal* instead of ordinary chains.

[10] To ease readability we use a simpler definition of *DP problems* than [13], since this simple definition suffices for the presentation of the new results of this paper.

[11] Interpretations into the *integers* instead of the naturals are often needed for algorithms like sum that *increase* an argument y until it reaches a *bound* x. In [17], we already presented an approach to prove termination by *bounded increase*. However, [17] did not consider built-in integers and pre-defined operations on them. Instead, [17] only handled natural numbers and all operations (like "$\geqslant$") had to be defined by rules of the TRS itself. Therefore, we now extend the approach of [17] to ITRSs.

[12] This is more general than related previous classes of interpretations: In [17], there was no "max" and only tuple symbols could be mapped to polynomials with integer coefficients, and in [11], all ground terms had to be mapped to natural numbers.

For example, consider the interpretation $\mathcal{P}ol$ where $\mathsf{SUM}_{\mathcal{P}ol} = x_1 - x_2$, $\mathsf{SIF}_{\mathcal{P}ol} = x_2 - x_3$, $+_{\mathcal{P}ol} = x_1 + x_2$, $n_{\mathcal{P}ol} = n$ for all $n \in \mathbb{Z}$, and $\geqslant_{\mathcal{P}ol} = \mathsf{true}_{\mathcal{P}ol} = \mathsf{false}_{\mathcal{P}ol} = 0$. For any term t and any position π in t, we say that t is $\succsim_{\mathcal{P}ol}$*-dependent* on π iff there exist terms u, v where $t[u]_\pi \not\approx_{\mathcal{P}ol} t[v]_\pi$. Here, $\approx_{\mathcal{P}ol} = \succsim_{\mathcal{P}ol} \cap \precsim_{\mathcal{P}ol}$. So in our example, $\mathsf{SIF}(b, x, y)$ is $\succsim_{\mathcal{P}ol}$-dependent on 2 and 3, but not on 1. We say that a term t is $\succsim_{\mathcal{P}ol}$*-increasing* on π iff $u \succsim_{\mathcal{P}ol} v$ implies $t[u]_\pi \succsim_{\mathcal{P}ol} t[v]_\pi$ for all terms u, v. So clearly, if t is $\succsim_{\mathcal{P}ol}$-independent on π, then t is also $\succsim_{\mathcal{P}ol}$-increasing on π. In our example, $\mathsf{SIF}(b, x, y)$ is $\succsim_{\mathcal{P}ol}$-increasing on 1 and 2, but not on 3.

The constraints generated by the reduction pair processor require that all DPs in $\mathcal{P}$ are strictly or weakly decreasing and all *usable rules* are weakly decreasing. Then one can delete all strictly decreasing DPs.

The *usable rules* [1, 16] include all rules that can reduce terms in $\succsim_{\mathcal{P}ol}$-dependent positions of $\mathcal{P}$'s right-hand sides when instantiating their variables with normal forms. Formally, for a term with f on a $\succsim_{\mathcal{P}ol}$-dependent position, all f-rules are usable. Moreover, if f's rules are usable and g occurs in the right-hand side of an f-rule on a $\succsim_{\mathcal{P}ol}$-dependent position, then g's rules are usable as well. For any symbol f with $\text{arity}(f) = n$, let $dep(f) = \{i \mid 1 \leqslant i \leqslant n$, there exists a term $f(t_1, ..., t_n)$ that is $\succsim_{\mathcal{P}ol}$-dependent on $i\}$. So $dep(\mathsf{SIF}) = \{2, 3\}$ for the interpretation $\mathcal{P}ol$ above. Moreover, as $\succsim_{\mathcal{P}ol}$ is not monotonic in general, one has to require that defined symbols only occur on $\succsim_{\mathcal{P}ol}$-increasing positions of right-hand sides.[13]

When using interpretations into the integers, then $\succ_{\mathcal{P}ol}$ is not well founded. However, $\succ_{\mathcal{P}ol}$ is still "non-infinitesimal", i.e., for any given bound, there is no infinite $\succ_{\mathcal{P}ol}$-decreasing sequence of terms that remains greater than the bound. Hence, the reduction pair processor transforms a DP problem into *two* new problems. As mentioned before, the first problem results from removing all strictly decreasing DPs. The second DP problem results from removing all DPs $s \to t$ from $\mathcal{P}$ that are *bounded from below*, i.e., DPs which satisfy the inequality $s \succsim \mathsf{c}$ for a fresh constant c. In Thm. 4, both TRSs and relations are seen as sets of pairs of terms. Thus, "$\mathcal{P} \setminus \succ_{\mathcal{P}ol}$" denotes $\{s \to t \in \mathcal{P} \mid s \not\succ_{\mathcal{P}ol} t\}$. Moreover, for any function symbol f and any TRS $\mathcal{S}$, let $Rls_{\mathcal{S}}(f) = \{\ell \to r \in \mathcal{S} \mid \text{root}(\ell) = f\}$.

Theorem 4 (Reduction Pair Processor [17]). *Let $\mathcal{R}$ be an ITRS, $\mathcal{P}ol$ be an integer max-polynomial interpretation, c be a fresh constant, and $\mathcal{P}_{bound} = \{s \to t \in \mathcal{P} \mid s \succsim_{\mathcal{P}ol} \mathsf{c}\}$. Then the following DP processor Proc is sound.*

$$Proc(\mathcal{P}) = \begin{cases} \{\, \mathcal{P} \setminus \succ_{\mathcal{P}ol},\ \mathcal{P} \setminus \mathcal{P}_{bound} \,\}, & \textit{if } \mathcal{P} \subseteq \succsim_{\mathcal{P}ol} \cup \succ_{\mathcal{P}ol},\ \mathcal{U}_{\mathcal{R} \cup \mathcal{PD}}(\mathcal{P}) \subseteq \succsim_{\mathcal{P}ol}, \\ & \textit{and defined symbols only occur on} \\ & \succsim_{\mathcal{P}ol}\textit{-increasing positions} \\ & \textit{in right-hand sides of } \mathcal{P} \cup \mathcal{U}_{\mathcal{R}}(\mathcal{P}) \\ \{\, \mathcal{P} \,\}, & \textit{otherwise} \end{cases}$$

[13] This is needed to ensure that $t\sigma \hookrightarrow_{\mathcal{R}}^* u$ implies $t\sigma \succsim_{\mathcal{P}ol} u$ whenever t's usable rules are weakly decreasing and σ instantiates variables by normal forms. Note that Thm. 4 is a simplified special case of the corresponding processor from [17]. In [17], we also introduced the possibility of reversing usable rules for function symbols occurring on *decreasing* positions. The approach of the present paper can also easily be extended accordingly and, indeed, our implementation makes use of this extension.

For any term t and TRS $\mathcal{S}$, the usable rules $\mathcal{U}_\mathcal{S}(t)$ *are the smallest set with*

- $\mathcal{U}_\mathcal{S}(x) = \varnothing$ *for every variable x and*
- $\mathcal{U}_\mathcal{S}(f(t_1,\ldots,t_n)) = Rls_\mathcal{S}(f) \;\cup\; \bigcup_{\ell\to r\in Rls_\mathcal{S}(f)} \mathcal{U}_\mathcal{S}(r) \;\cup\; \bigcup_{i\in dep(f)} \mathcal{U}_\mathcal{S}(t_i)$

For a set of dependency pairs $\mathcal{P}$, its usable rules *are* $\mathcal{U}_\mathcal{S}(\mathcal{P}) = \bigcup_{s\to t\in\mathcal{P}} \mathcal{U}_\mathcal{S}(t)$.

For $\mathcal{R}_1$, by Thm. 4 we search for an interpretation $\mathcal{P}ol$ with $s \;{}_{(}\!\succsim\!{}_{)\mathcal{P}ol}\; t$ for all $s\to t \in DP(\mathcal{R}_1) = \{(7),(8)\}$ and $\ell \succsim_{\mathcal{P}ol} r$ for all $\ell \to r \in \mathcal{U}_{\mathcal{R}_1\cup\mathcal{PD}}(DP(\mathcal{R}_1)) = \{0+1 \to 1,\; 1+1\to 2,\; -1+1\to 0,\; \ldots,\; 0 \geqslant 0 \to \mathsf{true},\; 1\geqslant 2 \to \mathsf{false},\; \ldots\}$. However, $\mathcal{U}_{\mathcal{R}_1\cup\mathcal{PD}}(DP(\mathcal{R}_1))$ is infinite and thus, this approach is not suitable for automation. When using the interpretation with $\mathsf{SIF}_{\mathcal{P}ol} = x_2 - x_3$, then the $\geqslant$-rules would not be usable, because $\geqslant$ only occurs on a $\succsim_{\mathcal{P}ol}$-independent position in the right-hand side of the DP (7). But the desired interpretation $\mathsf{SUM}_{\mathcal{P}ol} = x_1 - x_2$ cannot be used, because in DP (8), the defined symbol $+$ occurs in the second argument of SUM which is not a $\succsim_{\mathcal{P}ol}$-increasing position.[14]

To avoid the need for considering infinitely many rules in the reduction pair processor and in order to handle ITRSs where defined symbols like $+$ occur on non-increasing positions, we will now restrict ourselves to so-called *I-interpretations* where we fix the max-polynomials that are associated with the pre-defined symbols from $\mathbb{Z}\cup\mathcal{A}rith\mathcal{O}p$. The definition of I-interpretations guarantees that we have $\ell \approx_{\mathcal{P}ol} r$ for all rules $\ell\to r\in\mathcal{PD}$ where $\mathrm{root}(\ell)\in\{+,-,*\}$. For this reason, one can now also allow occurrences of $+$, $-$, and $*$ on non-increasing positions. Moreover, for I-interpretations we have $\ell \succsim_{\mathcal{P}ol} r$ for all rules $\ell\to r\in\mathcal{PD}$ where $\mathrm{root}(\ell)\in\{/,\%\}$. For these latter rules, obtaining $\ell \approx_{\mathcal{P}ol} r$ with a useful max-polynomial interpretation is impossible, since division and modulo are no max-polynomials.[15]

Definition 5 (I-interpretation). *An integer max-polynomial interpretation $\mathcal{P}ol$ is an* I-interpretation *iff $n_{\mathcal{P}ol} = n$ for all $n\in\mathbb{Z}$, $+_{\mathcal{P}ol} = x_1 + x_2$, $-_{\mathcal{P}ol} = x_1 - x_2$, $*_{\mathcal{P}ol} = x_1 * x_2$, $\%_{\mathcal{P}ol} = |x_1|$, and $/_{\mathcal{P}ol} = |x_1| - \min(|x_2|-1, |x_1|)$. Note that for any max-polynomial p, "$|p|$" is also a max-polynomial since this is just an abbreviation for* $\max(p,-p)$. *Similarly,* "$\min(p,q)$" *is an abbreviation for* $-\max(-p,-q)$. *We say that an I-interpretation is* proper *for a term t if all defined symbols except $+$, $-$, and $*$ only occur on $\succsim_{\mathcal{P}ol}$-increasing positions of t and if symbols from $\mathcal{R}el\mathcal{O}p$ only occur on $\succsim_{\mathcal{P}ol}$-independent positions of t.*

Now $[n/m]_{\mathcal{P}ol}$ is greater or equal to n/m for all $n,m\in\mathbb{Z}$ where $m\neq 0$ (and

[14] Nevertheless, Thm. 4 is helpful for ITRSs where the termination argument is not due to integer arithmetic. For example, consider the ITRS $\mathsf{g}(x,\mathsf{cons}(y,ys)) \to \mathsf{cons}(x+y,\mathsf{g}(x,ys))$. When using interpretations $\mathcal{P}ol$ with $f_{\mathcal{P}ol} = 0$ for all $f\in\mathcal{F}_{int}$, the rules $\ell\to r\in\mathcal{PD}$ are always weakly decreasing. Hence, then one only has to regard finitely many usable rules when automating Thm. 4. Moreover, if all $f_{\mathcal{P}ol}$ have just *natural* coefficients, then one does not have to generate the new DP problem $\mathcal{P}\setminus\mathcal{P}_{bound}$. In this case, one can define $s \;{}_{(}\!\succsim\!{}_{)\mathcal{P}ol}\; t$ iff $[s]_{\mathcal{P}ol} \;{}_{(}\!\geqslant\!{}_{)}\; [t]_{\mathcal{P}ol}$ holds for all instantiations of the variables by *natural* numbers. Thus, in the example above the termination proof is trivial by using the interpretation with $\mathsf{G}_{\mathcal{P}ol} = x_2$ and $\mathsf{cons}_{\mathcal{P}ol} = x_2 + 1$.

[15] In principle, one could also permit interpretations $f_{\mathcal{P}ol}$ containing divisions. But existing implementations to search for interpretations cannot handle division or modulo.

similar for $[n \,\%\, m]_{\mathcal{P}ol})$.[16] Hence, one can improve the processor of Thm. 4 by not regarding the infinitely many rules of $\mathcal{PD}$ anymore. The concept of *proper* I-interpretations ensures that we can disregard the (infinitely many) usable rules for the symbols from $\mathcal{R}el\mathcal{O}p$ and that the symbols "/" and "%" only have to be estimated "upwards". Then one can now replace the usable rules w.r.t. $\mathcal{R} \cup \mathcal{PD}$ in Thm. 4 by the usable rules w.r.t. $\mathcal{R} \cup \mathcal{BO}$. Here, $\mathcal{BO}$ are the (finitely many) rules for the symbols $\wedge$ and $\Rightarrow$ in $\mathcal{B}ool\mathcal{O}p$, i.e., $\mathcal{BO} = Rls_{\mathcal{PD}}(\wedge) \cup Rls_{\mathcal{PD}}(\Rightarrow)$.

Theorem 6 (Reduction Pair Processor for ITRSs). *Let $\mathcal{R}$ be an ITRS, $\mathcal{P}ol$ be an I-interpretation, and $\mathcal{P}_{bound}$ be as in Thm. 4. Then Proc is sound.*

$$Proc(\mathcal{P}) = \begin{cases} \{\, \mathcal{P} \setminus \succ_{\mathcal{P}ol},\ \mathcal{P} \setminus \mathcal{P}_{bound} \,\}, & \text{if } \mathcal{P} \subseteq \succsim_{\mathcal{P}ol} \cup \succ_{\mathcal{P}ol},\ \mathcal{U}_{\mathcal{R} \cup \mathcal{BO}}(\mathcal{P}) \subseteq \succsim_{\mathcal{P}ol}, \\ & \text{and } \mathcal{P}ol \text{ is proper for all right-hand} \\ & \text{sides of } \mathcal{P} \cup \mathcal{U}_{\mathcal{R}}(\mathcal{P}) \\ \{\, \mathcal{P} \,\}, & \text{otherwise} \end{cases}$$

Proof. We show that Thm. 6 follows from Thm. 4. In Thm. 6, we only require that usable rules from $\mathcal{R} \cup \mathcal{BO}$ are weakly decreasing, whereas Thm. 4 considers usable rules from $\mathcal{R} \cup \mathcal{PD}$. For any I-interpretation $\mathcal{P}ol$, we have $\ell \approx_{\mathcal{P}ol} r$ for all $\ell \to r \in \mathcal{PD}$ with $\mathrm{root}(\ell) \in \{+, -, *\}$. So these rules are even equivalent w.r.t. $\approx_{\mathcal{P}ol}$. Moreover, the rules with $\mathrm{root}(\ell) \in \{/, \%\}$ are weakly decreasing w.r.t. $\succsim_{\mathcal{P}ol}$. The rules with $\mathrm{root}(\ell) \in \mathcal{R}el\mathcal{O}p$ are never contained in $\mathcal{U}_{\mathcal{R} \cup \mathcal{PD}}(\mathcal{P})$, because by properness of $\mathcal{P}ol$, symbols from $\mathcal{R}el\mathcal{O}p$ only occur on $\succsim_{\mathcal{P}ol}$-independent positions in right-hand sides of $\mathcal{P} \cup \mathcal{U}_{\mathcal{R}}(\mathcal{P})$ and they do not occur at all in right-hand sides of $\mathcal{PD}$. Thus, $\mathcal{U}_{\mathcal{R} \cup \mathcal{PD}}(\mathcal{P}) \subseteq \succsim_{\mathcal{P}ol}$, as required in Thm. 4.

The other difference between Thm. 6 and 4 is that in Thm. 6, $+$, $-$, and $*$ may also occur on non-$\succsim_{\mathcal{P}ol}$-increasing positions. But as shown in [17, 20], this is possible since the rules for these symbols are equivalent w.r.t. $\approx_{\mathcal{P}ol}$. $\square$

To solve the DP problem $\mathcal{P} = \{(7), (8)\}$ of $\mathcal{R}_1$ with Thm. 6, we want to use an I-interpretation $\mathcal{P}ol$ where $\mathsf{SUM}_{\mathcal{P}ol} = x_1 - x_2$ and $\mathsf{SIF}_{\mathcal{P}ol} = x_2 - x_3$. Now there are no usable rules $\mathcal{U}_{\mathcal{R} \cup \mathcal{BO}}(\mathcal{P})$, since the $+$- and $\geqslant$-rules are not included in $\mathcal{R} \cup \mathcal{BO}$. The DP (8) is strictly decreasing, but none of the DPs (7) and (8) is bounded, since we have neither $\mathsf{SUM}(x, y) \succsim_{\mathcal{P}ol} \mathsf{c}$ nor $\mathsf{SIF}(\mathsf{true}, x, y) \succsim_{\mathcal{P}ol} \mathsf{c}$ for any possible value of $\mathsf{c}_{\mathcal{P}ol}$. Thus, the reduction pair processor would return the two DP problems $\{(7)\}$ and $\{(7), (8)\}$, i.e., it would not simplify $\mathcal{P}$.

4 Conditional Constraints

The solution to the problem above is to consider *conditions* for inequalities like $s \ {}_{(}\!\succsim_{)}\ t$ or $s \succsim \mathsf{c}$. For example, to include the DP (7) in $\mathcal{P}_{bound}$, we do not have to demand $\mathsf{SUM}(x, y) \succsim \mathsf{c}$ for *all* instantiations of x and y. Instead, it suffices to require the inequality only for those instantiations of x and y which can be used in chains. So we require $\mathsf{SUM}(x, y) \succsim \mathsf{c}$ only for instantiations σ where (7)'s instantiated right-hand side $\mathsf{SIF}(x \geqslant y, x, y)\sigma$ reduces to an instantiated left-hand

[16] Let $m \neq 0$. If $|m| = 1$ or $n = 0$, then we have $[n/m]_{\mathcal{P}ol} = |n|$. Otherwise, we obtain $[n/m]_{\mathcal{P}ol} < |n|$. The latter fact is needed for ITRSs like $\mathcal{R}_2$ which terminate because of divisions in their recursive arguments.

side $u\sigma$ for some DP $u \to v$ where $u\sigma$ is in normal form. Here, $u \to v$ should again be variable renamed. As our DP problem contains two DPs (7) and (8), we get the following two *conditional constraints* (by considering all possibilities $u \to v \in \{(7), (8)\}$). We include (7) in $\mathcal{P}_{bound}$ if both constraints are satisfied.

$$\mathsf{SIF}(x \geqslant y, x, y) = \mathsf{SUM}(x', y') \quad \Rightarrow \quad \mathsf{SUM}(x, y) \succsim \mathsf{c} \tag{11}$$
$$\mathsf{SIF}(x \geqslant y, x, y) = \mathsf{SIF}(\mathsf{true}, x', y') \quad \Rightarrow \quad \mathsf{SUM}(x, y) \succsim \mathsf{c} \tag{12}$$

Definition 7 (Syntax and Semantics of Conditional Constraints [17]). *The set $\mathcal{C}$ of conditional constraints is the smallest set with*[17]

- $\{TRUE,\ s \succsim t,\ s \succ t,\ s = t\} \subseteq \mathcal{C}$ *for all terms s and t*
- *if* $\{\varphi_1, \varphi_2\} \subseteq \mathcal{C}$, *then* $\varphi_1 \wedge \varphi_2 \in \mathcal{C}$ *and* $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{C}$

For an I-interpretation $\mathcal{P}ol$, we define which normal substitutions[18] *σ satisfy a constraint $\varphi \in \mathcal{C}$, denoted "$\sigma \models_{\mathcal{P}ol} \varphi$":*

- $\sigma \models_{\mathcal{P}ol} TRUE$ *for all normal substitutions* σ
- $\sigma \models_{\mathcal{P}ol} s \succsim t$ *iff* $s\sigma \succsim_{\mathcal{P}ol} t\sigma$ *and* $\sigma \models_{\mathcal{P}ol} s \succ t$ *iff* $s\sigma \succ_{\mathcal{P}ol} t\sigma$
- $\sigma \models_{\mathcal{P}ol} s = t$ *iff* $s\sigma \hookrightarrow^*_{\mathcal{R}} t\sigma$ *and $t\sigma$ is a normal form w.r.t.* $\hookrightarrow_{\mathcal{R}}$
- $\sigma \models_{\mathcal{P}ol} \varphi_1 \wedge \varphi_2$ *iff* $\sigma \models_{\mathcal{P}ol} \varphi_1$ *and* $\sigma \models_{\mathcal{P}ol} \varphi_2$
- $\sigma \models_{\mathcal{P}ol} \varphi_1 \Rightarrow \varphi_2$ *iff* $\sigma \not\models_{\mathcal{P}ol} \varphi_1$ *or* $\sigma \models_{\mathcal{P}ol} \varphi_2$

A constraint φ is valid *("$\models_{\mathcal{P}ol} \varphi$") iff $\sigma \models_{\mathcal{P}ol} \varphi$ for* all *normal substitutions σ.*

Now we refine the reduction pair processor by taking conditions into account. To this end, we modify the definition of $\mathcal{P}_{bound}$ and introduce $\mathcal{P}_{\succsim}$ and $\mathcal{P}_{\succ}$.

Theorem 8 (Conditional Reduction Pair Processor for ITRSs). *Let $\mathcal{R}$ be an ITRS, $\mathcal{P}ol$ be an I-interpretation, c be a fresh constant, and let*

$$\begin{aligned}
\mathcal{P}_{\succsim} &= \{\, s \to t \in \mathcal{P} \mid\ \models_{\mathcal{P}ol} \textstyle\bigwedge_{u \to v \in \mathcal{P}} (t = u' \ \Rightarrow\ s \succsim t) \,\}\\
\mathcal{P}_{\succ} &= \{\, s \to t \in \mathcal{P} \mid\ \models_{\mathcal{P}ol} \textstyle\bigwedge_{u \to v \in \mathcal{P}} (t = u' \ \Rightarrow\ s \succ t) \,\}\\
\mathcal{P}_{bound} &= \{\, s \to t \in \mathcal{P} \mid\ \models_{\mathcal{P}ol} \textstyle\bigwedge_{u \to v \in \mathcal{P}} (t = u' \ \Rightarrow\ s \succsim \mathsf{c}) \,\}
\end{aligned}$$

where u' results from u by renaming its variables. Then Proc is sound.

$$Proc(\mathcal{P}) = \begin{cases} \{\, \mathcal{P} \setminus \mathcal{P}_{\succ},\ \mathcal{P} \setminus \mathcal{P}_{bound} \,\}, & \text{if } \mathcal{P}_{\succsim} \cup \mathcal{P}_{\succ} = \mathcal{P},\ \mathcal{U}_{\mathcal{R} \cup \mathcal{BO}}(\mathcal{P}) \subseteq \succsim_{\mathcal{P}ol}, \\ & \text{and } \mathcal{P}ol \text{ is proper for all right-hand} \\ & \text{sides of } \mathcal{P} \cup \mathcal{U}_{\mathcal{R}}(\mathcal{P}) \\ \{\, \mathcal{P} \,\}, & \text{otherwise} \end{cases}$$

Proof. Thm. 8 immediately follows from Thm. 6 in the same way as [17, Thm. 11] follows from [17, Thm. 8]. □

To ease readability, in Thm. 8 we only consider conditions resulting from *two* DPs $s \to t$ and $u \to v$ which may follow each other in chains. In our implemen-

[17] To simplify the presentation, we neither regard conditional constraints with universally quantified subformulas nor the simplification of constraints by induction, cf. [17]. This technique of [17] could be integrated in our approach to also handle ITRSs where tests are not of the form "$s \geqslant t$" with the pre-defined symbol "$\geqslant$", but of the form "$\mathsf{ge}(s, t)$", where ge is given by user-defined rules in the ITRS. After such an integration, our approach would subsume the corresponding technique of [17].

[18] A *normal substitution* σ instantiates all variables by normal forms w.r.t. $\hookrightarrow_{\mathcal{R}}$.

tation, we extended this by also regarding conditions resulting from more than two adjacent DPs and by also regarding DPs *preceding* $s \to t$ in chains, cf. [17].

The question remains how to check whether conditional constraints are valid, since this requires reasoning about reachability w.r.t. TRSs with infinitely many rules. In [17], we introduced the rules (I)-(IV) to simplify conjunctions $\varphi_1 \wedge ... \wedge \varphi_n$ of conditional constraints. These rules can be used to replace a conjunct φ_i by a new formula φ_i'. The rules are sound, i.e., $\models_{\mathcal{P}ol} \varphi_i'$ implies $\models_{\mathcal{P}ol} \varphi_i$. Of course, *TRUE* $\wedge\, \varphi$ can always be simplified to φ. Eventually, we want to remove all equalities "$p = q$" from the constraints.

I. **Constructor and Different Function Symbol**			
$\dfrac{f(s_1,...,s_n) = g(t_1,...,t_m) \wedge \varphi \;\Rightarrow\; \psi}{TRUE}$	if f is a constructor and $f \neq g$		
II. **Same Constructors on Both Sides**			
$\dfrac{f(s_1,...,s_n) = f(t_1,...,t_n) \wedge \varphi \;\Rightarrow\; \psi}{s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge \varphi \;\Rightarrow\; \psi}$	if f is a constructor		
III. **Variable in Equation**			
$\dfrac{x{=}q \wedge \varphi \;\Rightarrow\; \psi}{\varphi\,\sigma \;\Rightarrow\; \psi\,\sigma}$	if $x \in \mathcal{V}$ and $\sigma = [x/q]$	$\dfrac{q{=}x \wedge \varphi \;\Rightarrow\; \psi}{\varphi\,\sigma \;\Rightarrow\; \psi\,\sigma}$	if $x \in \mathcal{V}$, q has no defined symbols, $\sigma{=}[x/q]$
IV. **Delete Conditions**			
$\dfrac{\varphi \;\Rightarrow\; \psi}{\varphi' \;\Rightarrow\; \psi}$	if $\varphi' \subseteq \varphi$		

For example, Rule (I) detects that the premise of constraint (11) is unsatisfiable: there is no substitution σ with $\sigma \models_{\mathcal{P}ol} \mathsf{SIF}(x \geqslant y, x, y) = \mathsf{SUM}(x', y')$, since SIF is not a defined function symbol (i.e., it is a *constructor*) and therefore, SIF-terms can only be reduced to SIF-terms.

Rule (II) handles conditions like $\mathsf{SIF}(x \geqslant y, x, y) = \mathsf{SIF}(\mathsf{true}, x', y')$ where both terms start with the same constructor SIF. So (12) is transformed into

$$x \geqslant y = \mathsf{true} \;\wedge\; x = x' \;\wedge\; y = y' \quad \Rightarrow \quad \mathsf{SUM}(x, y) \gtrsim \mathsf{c} \tag{13}$$

Rule (III) removes conditions of the form "$x = q$" or "$q = x$" by applying the substitution $[x/q]$ to the constraint. So (13) is transformed into

$$x \geqslant y = \mathsf{true} \quad \Rightarrow \quad \mathsf{SUM}(x, y) \gtrsim \mathsf{c} \tag{14}$$

Rule (IV) can omit arbitrary conjuncts from the premise of an implication. To ease notation, we regard a conjunction as a set of formulas. So their order is irrelevant and we write $\varphi' \subseteq \varphi$ iff all conjuncts of φ' are also conjuncts of φ. The empty conjunction is *TRUE* (i.e., *TRUE* $\Rightarrow \psi$ can always be simplified to ψ).

Since [17] did not handle pre-defined function symbols, we now extend the rules (I)-(IV) from [17] by new rules to "lift" pre-defined function symbols from $\mathcal{R}el\mathcal{O}p$ like $\geqslant$ to symbols like $\gtrsim$ that are used in conditional constraints. Similar rules are used for the other symbols from $\mathcal{R}el\mathcal{O}p$. The idea is to replace a conditional constraint like "$s \geqslant t = \mathsf{true}$" by the conditional constraint "$s \succsim t$". However, this is not necessarily sound, because s and t may contain defined symbols. Note that $\sigma \models_{\mathcal{P}ol} s \geqslant t = \mathsf{true}$ means that $s\sigma \hookrightarrow^*_{\mathcal{R}} n$ and $t\sigma \hookrightarrow^*_{\mathcal{R}} m$ for $n, m \in \mathbb{Z}$ with $n \geqslant m$. For any I-interpretation $\mathcal{P}ol$, we therefore have $n \succsim_{\mathcal{P}ol} m$, since $n_{\mathcal{P}ol} = n$ and $m_{\mathcal{P}ol} = m$. To guarantee that $\sigma \models_{\mathcal{P}ol} s \succsim t$ holds as well, we ensure that $s\sigma \succsim_{\mathcal{P}ol} n$ and $m \succsim_{\mathcal{P}ol} t\sigma$. To this end, we require that $\mathcal{U}_{\mathcal{R}\cup\mathcal{BO}}(s) \subseteq \succsim_{\mathcal{P}ol}$ and that t contains no defined symbols except $+$, $-$, and $*$. An analogous rule can also be formulated for constraints of the form "$s \geqslant t = \mathsf{false}$".[19]

V. **Lift Symbols from** $\mathcal{R}el\mathcal{O}p$

$$\frac{s \geqslant t = \mathsf{true} \;\wedge\; \varphi \;\Rightarrow\; \psi}{(s \succsim t \qquad\quad \wedge\; \varphi \;\Rightarrow\; \psi) \;\wedge \displaystyle\bigwedge_{\ell\to r\,\in\,\mathcal{U}_{\mathcal{R}\cup\mathcal{BO}}(s)} \ell \succsim r}$$

if t contains no defined symbols except $+, -, *$ and $\mathcal{P}ol$ is proper for s and for all right-hand sides of $\mathcal{U}_{\mathcal{R}}(s)$

By Rule (V), (14) is transformed into

$$x \succsim y \quad\Rightarrow\quad \mathsf{SUM}(x,y) \succsim \mathsf{c} \tag{15}$$

Similar to the lifting of the function symbols from $\mathcal{R}el\mathcal{O}p$, it is also possible to lift the function symbols from $\mathcal{B}ool\mathcal{O}p$. For reasons of space, we only present the corresponding rules for lifting "$\wedge$", but of course "$\Rightarrow$" can be lifted analogously.

VI. **Lift Symbols from** $\mathcal{B}ool\mathcal{O}p$

$$\frac{s \wedge t = \mathsf{true} \qquad \wedge\, \varphi \;\Rightarrow\; \psi}{s = \mathsf{true} \wedge t = \mathsf{true} \wedge \varphi \;\Rightarrow\; \psi} \qquad \frac{s \wedge t = \mathsf{false} \wedge \varphi \;\Rightarrow\; \psi}{(s = \mathsf{false} \wedge \varphi \;\Rightarrow\; \psi) \;\wedge\; (t = \mathsf{false} \wedge \varphi \;\Rightarrow\; \psi)}$$

To illustrate this rule, consider the constraint "$(x \geqslant y \wedge y > 1) = \mathsf{true} \;\Rightarrow\; \mathsf{LOG}(x,y) \succsim \mathsf{c}$" which results when trying to include the DP (9) of the ITRS $\mathcal{R}_2$ in $\mathcal{P}_{bound}$. Here, Rule (VI) gives "$x \geqslant y = \mathsf{true} \wedge y > 1 = \mathsf{true} \;\Rightarrow\; \mathsf{LOG}(x,y) \succsim \mathsf{c}$" which is transformed by Rule (V) into "$x \succsim y \,\wedge\, y \succ 1 \;\Rightarrow\; \mathsf{LOG}(x,y) \succsim \mathsf{c}$".

Let $\varphi \vdash \varphi'$ iff φ' results from φ by repeatedly applying the above inference rules. We can now refine the processor from Thm. 8.

Theorem 9 (Conditional Reduction Pair Processor with Inference). *Let $\mathcal{P}ol$ be an I-interpretation and c be a fresh constant. For all $s \to t \in \mathcal{P}$ and all $\psi \in \{\, s \succsim t,\; s \succ t,\; s \succsim \mathsf{c} \,\}$, let φ_ψ be a constraint with $\bigwedge_{u\to v\in\mathcal{P}}(t = u' \;\Rightarrow\; \psi) \vdash \varphi_\psi$. Here, u' results from u by renaming its variables. Then the processor Proc from Thm. 8 is still sound if we define $\mathcal{P}_{\succsim} = \{s \to t \in \mathcal{P} \mid \;\models_{\mathcal{P}ol} \varphi_{s\succsim t}\}$, $\mathcal{P}_{\succ} = \{s \to t \in \mathcal{P} \mid \;\models_{\mathcal{P}ol} \varphi_{s\succ t}\}$, and $\mathcal{P}_{bound} = \{s \to t \in \mathcal{P} \mid \;\models_{\mathcal{P}ol} \varphi_{s\succsim \mathsf{c}}\}$.*

Proof. It suffices to show the soundness of the rules (I)-(VI): If $\varphi \vdash \varphi'$, then $\models_{\mathcal{P}ol} \varphi'$ implies $\models_{\mathcal{P}ol} \varphi$. Then Thm. 9 immediately follows from Thm. 8.

Soundness of the rules (I)-(IV) was shown in [17, Thm. 14]. For Rule (V), let $\models_{\mathcal{P}ol} (s \succsim t \wedge \varphi \;\Rightarrow\; \psi) \wedge \bigwedge_{\ell\to r\,\in\,\mathcal{U}_{\mathcal{R}\cup\mathcal{BO}}(s)} \ell \succsim r$ and $\sigma \models_{\mathcal{P}ol} s \geqslant t = \mathsf{true} \wedge \varphi$. As explained above, this implies $\sigma \models_{\mathcal{P}ol} s \succsim t \wedge \varphi$ and hence, $\sigma \models_{\mathcal{P}ol} \psi$, as desired.

For the first variant of (VI), $\sigma \models_{\mathcal{P}ol} s \wedge t = \mathsf{true}$ iff $s\sigma \hookrightarrow^*_{\mathcal{R}} \mathsf{true}$ and $t\sigma \hookrightarrow^*_{\mathcal{R}}$

[19] In addition, one can also use rules to perform narrowing and rewriting on the terms in conditions, similar to the use of narrowing and rewriting in [16].

true, i.e., $\sigma \models_{\mathcal{P}ol} s = \mathsf{true} \wedge t = \mathsf{true}$. For the second variant, $\sigma \models_{\mathcal{P}ol} s \wedge t = \mathsf{false}$ implies $s\sigma \hookrightarrow^*_{\mathcal{R}} \mathsf{false}$ or $t\sigma \hookrightarrow^*_{\mathcal{R}} \mathsf{false}$, i.e., $\sigma \models_{\mathcal{P}ol} s = \mathsf{false}$ or $\sigma \models_{\mathcal{P}ol} t = \mathsf{false}$. □

5 Generating I-Interpretations

To automate the processor of Thm. 9, we show how to generate an I-interpretation that satisfies a given conditional constraint. This conditional constraint is a conjunction of formulas like $\varphi_{s \succsim t}$, $\varphi_{s \succ t}$, and $\varphi_{s \succsim \mathsf{c}}$ for DPs $s \to t$ as well as $\ell \succsim r$ for usable rules $\ell \to r$. Moreover, one has to ensure that the I-interpretation is chosen in such a way that $\mathcal{P}ol$ is proper for the right-hand sides of the DPs and the usable rules.[20] Compared to our earlier work in [11], the only additional difficulty is that now we really consider arbitrary max-polynomial interpretations over the integers where $[t]_{\mathcal{P}ol}$ can also be negative for any ground term t.

To find I-interpretations automatically, one starts with an *abstract* I-interpretation. It maps each function symbol to a max-polynomial with *abstract* coefficients. In other words, one has to determine the degree and the shape of the max-polynomial, but the actual coefficients are left open. For example, for the ITRS $\mathcal{R}_1$ we could use an abstract I-interpretation $\mathcal{P}ol$ where $\mathsf{SUM}_{\mathcal{P}ol} = a_0 + a_1\, x_1 + a_2\, x_2$, $\mathsf{SIF}_{\mathcal{P}ol} = b_0 + b_1\, x_1 + b_2\, x_2 + b_3\, x_3$, and $\mathsf{c}_{\mathcal{P}ol} = c_0$. Here, a_i, b_i, and c_0 are abstract coefficients. Of course, the interpretation for the symbols in $\mathbb{Z} \cup \mathcal{A}rith\mathcal{O}p$ is fixed as for any I-interpretation (i.e., $+_{\mathcal{P}ol} = x_1 + x_2$, etc.).

After application of the rules in Sect. 4, we have obtained a conditional constraint without the symbol "$=$". Now we transform the conditional constraint into a so-called *inequality constraint* by replacing all atomic constraints "$s \succsim t$" by "$[s]_{\mathcal{P}ol} \geqslant [t]_{\mathcal{P}ol}$" and all atomic constraints "$s \succ t$" by "$[s]_{\mathcal{P}ol} \geqslant [t]_{\mathcal{P}ol} + 1$". For instance, the atomic constraint "$\mathsf{SUM}(x, y) \succsim \mathsf{c}$" is transformed into "$a_0 + a_1\, x + a_2\, y \geqslant c_0$". Here, the abstract coefficients a_0, a_1, a_2, c_0 are implicitly existentially quantified and the variables $x, y \in \mathcal{V}$ are universally quantified. In other words, we search for values of the abstract coefficients such that the inequalities hold *for all* integer numbers x and y. To make this explicit, we add universal quantifiers for the variables from $\mathcal{V}$. More precisely, if our overall inequality constraint has the form $\varphi_1 \wedge \ldots \wedge \varphi_n$, then we now replace each φ_i by "$\forall x_1 \in \mathbb{Z}, \ldots, x_m \in \mathbb{Z}\ \varphi_i$" where $x_1, \ldots, x_m$ are the variables from $\mathcal{V}$ occurring in φ_i. So the conditional constraint (15) is transformed into the inequality constraint

$$\forall x \in \mathbb{Z}, y \in \mathbb{Z} \quad (\, x \geqslant y \quad \Rightarrow \quad a_0 + a_1\, x + a_2\, y \geqslant c_0 \,) \tag{16}$$

In general, inequality constraints have the following form where Num_i is $\mathbb{Z}$ or $\mathbb{N}$.

$$\forall x_1 \in Num_1, \ldots, x_m \in Num_m \quad p_1 \geqslant q_1 \wedge \ldots \wedge p_n \geqslant q_n \;\Rightarrow\; p \geqslant q$$

Now our goal is to transform such inequality constraints further into *Diophantine constraints* which do not contain any universally quantified variables $x_1, \ldots, x_m$ anymore. Then one can apply existing methods to search for values

[20] The set of usable rules and thus, the given conditional constraint depends on the I-interpretation (that determines which positions are increasing or dependent). Nevertheless, we showed in [11] how to encode such search problems into a single constraint.

of the abstract coefficients that satisfy the Diophantine constraints.

We already developed such transformation rules in [11]. But [11] was restricted to the case where all universally quantified variables range over $\mathbb{N}$, i.e., $Num_1 = ... = Num_m = \mathbb{N}$. Moreover, [11]'s essential rule to eliminate universally quantified variables only works if there are no conditions (i.e., $n = 0$), cf. Rule (C) below. Thus, we extend the transformation rules from [11][21] by the following rule which can be used whenever a condition can be transformed into "$x \geqslant p$" or "$p \geqslant x$" for a polynomial p not containing x. It does not only replace a variable ranging over $\mathbb{Z}$ by one over $\mathbb{N}$, but it also "applies" the condition "$x \geqslant p$" resp. "$p \geqslant x$" and removes it afterwards without increasing the number of constraints.

A. **Eliminating Conditions**

$$\frac{\forall x \in \mathbb{Z}, \dots \quad (x \geqslant p \,\wedge\, \varphi \Rightarrow \psi)}{\forall z \in \mathbb{N}, \dots \quad (\varphi[x/p+z] \Rightarrow \psi[x/p+z])} \qquad \frac{\forall x \in \mathbb{Z}, \dots \quad (p \geqslant x \,\wedge\, \varphi \Rightarrow \psi)}{\forall z \in \mathbb{N}, \dots \quad (\varphi[x/p-z] \Rightarrow \psi[x/p-z])}$$

if x does not occur in the polynomial p

By Rule (A), the inequality constraint (16) is therefore transformed into

$$\forall y \in \mathbb{Z}, z \in \mathbb{N} \quad a_0 + a_1\,(y+z) + a_2\,y \geqslant c_0 \tag{17}$$

To replace all remaining quantifiers over $\mathbb{Z}$ by quantifiers over $\mathbb{N}$, we add the following rule. It splits the remaining inequality constraint φ (which may have additional universal quantifiers) into the cases where y is positive resp. negative.

B. **Split**

$$\frac{\forall y \in \mathbb{Z} \quad \varphi}{\forall y \in \mathbb{N} \quad \varphi \qquad \wedge \qquad \forall y \in \mathbb{N} \quad \varphi[y/-y]}$$

Thus, Rule (B) transforms (17) into the conjunction of (18) and (19).

$$\forall y \in \mathbb{N}, z \in \mathbb{N} \quad a_0 + a_1\,(y+z) + a_2\,y \geqslant c_0 \tag{18}$$
$$\forall y \in \mathbb{N}, z \in \mathbb{N} \quad a_0 + a_1\,(-y+z) - a_2\,y \geqslant c_0 \tag{19}$$

If φ still has conditions, then a split by Rule (B) often results in unsatisfiable conditions. To detect them, we use SMT-solvers for linear integer arithmetic and additional sufficient criteria to detect also certain non-linear unsatisfiable conditions like $x^2 < 0$, etc. If a condition is found to be unsatisfiable, we delete the inequality constraint. Note that (18) can be reformulated as

$$\forall y \in \mathbb{N}, z \in \mathbb{N} \quad (a_1 + a_2)\,y + a_1\,z + (a_0 - c_0) \geqslant 0$$

So we now have to ensure non-negativeness of "polynomials" over variables like y and z that range over $\mathbb{N}$, where the "coefficients" are polynomials like "$a_1 + a_2$" over the abstract variables. To this end, it suffices to require that all these "coefficients" are $\geqslant 0$ [21]. In other words, now one can eliminate all universally

[21] For reasons of space, we do not present the remaining transformation rules of [11], which are applied in our implementation as well. These rules are used to delete "max" and to eliminate arbitrary conditions, e.g., conditions that are not removed by Rule (A). Similar transformation rules can for example also be found in [18].

quantified variables like y, z and transform (18) into the *Diophantine constraint*

$$a_1 + a_2 \geqslant 0 \qquad \wedge \qquad a_1 \geqslant 0 \qquad \wedge \qquad a_0 - c_0 \geqslant 0$$

C. **Eliminating Universally Quantified Variables**

$$\frac{\forall x_1 \in \mathbb{N}, \ldots, x_m \in \mathbb{N} \quad p_1\, x_1^{e_{11}} \ldots x_m^{e_{m1}} + \ldots + p_k\, x_1^{e_{1k}} \ldots x_m^{e_{mk}} \geqslant 0}{p_1 \geqslant 0 \;\wedge \ldots \wedge\; p_k \geqslant 0} \quad \text{if the } p_i \text{ do not contain variables from } \mathcal{V}$$

To search for values of the abstract coefficients that satisfy the resulting Diophantine constraints, one fixes upper and lower bounds for these values. Then we showed in [10] how to translate such Diophantine constraints into a satisfiability problem for propositional logic which can be handled by SAT solvers efficiently. The constraints resulting from the initial inequality constraint (16) are for example satisfied by $a_0 = 0$, $a_1 = 1$, $a_2 = -1$, and $c_0 = 0$.[22] With these values, the abstract interpretation $a_0 + a_1\, x_1 + a_2\, x_2$ for SUM is turned into the concrete interpretation $x_1 - x_2$. With the resulting concrete I-interpretation $\mathcal{P}ol$, we would have $\mathcal{P}_\succ = \{(8)\}$ and $\mathcal{P}_{bound} = \{(7)\}$. The reduction pair processor of Thm. 9 would therefore transform the initial DP problem $\mathcal{P} = \{(7), (8)\}$ into the two problems $\mathcal{P} \setminus \mathcal{P}_\succ = \{(7)\}$ and $\mathcal{P} \setminus \mathcal{P}_{bound} = \{(8)\}$. Both of them are easy to solve (e.g., by using $\mathcal{P}ol'$ with $\mathsf{SUM}_{\mathcal{P}ol'} = 1$, $\mathsf{SIF}_{\mathcal{P}ol'} = 0$ and $\mathcal{P}ol''$ with $\mathsf{SUM}_{\mathcal{P}ol''} = 0$, $\mathsf{SIF}_{\mathcal{P}ol''} = 1$ or by using other processors like the *dependency graph*).

Our approach also directly works for ITRSs with extra variables on right-hand sides of rules. Then the rewrite relation is defined as $s \hookrightarrow_{\mathcal{R}} t$ iff there is a rule $\ell \to r \in \mathcal{R} \cup \mathcal{PD}$ such that $s|_\pi = \ell\sigma$ and $t = s[r\sigma]_\pi$, $\ell\sigma$ does not contain redexes as proper subterms, and σ *is a normal substitution* (i.e., $\sigma(y)$ is in normal form also for variables y occurring only in r). Now we can also handle ITRSs with non-determinism like $\mathsf{f}(\mathsf{true}, x) \to \mathsf{f}(x > y \wedge x \geqslant \mathsf{0},\ y)$. Here, the argument x of f is replaced by an arbitrary smaller number y. Handling non-deterministic algorithms is often necessary for termination proofs of imperative programs when abstracting away "irrelevant" parts of the computation, cf. [4, 8, 24].

This also opens up a possibility to deal with algorithms that contain "large constants" computed by user-defined functions. For instance, consider an ITRS containing $\mathsf{f}(\mathsf{true},\ x) \ \to\ \mathsf{f}(\mathsf{ack}(\mathsf{10}, \mathsf{10}) \geqslant x,\ x + \mathsf{1})$ and ack-rules computing the *Ackermann* function. With our approach, the ack-rules would have to be weakly decreasing, cf. Rule (V). This implies $\mathsf{ack}_{\mathcal{P}ol}(n, m) \geqslant \mathit{Ackermann}(n, m)$, which does not hold for any max-polynomial $\mathsf{ack}_{\mathcal{P}ol}$. But such examples can be handled by automatically transforming the original ITRS to an ITRS with extra variables whose termination implies termination of the original ITRS. If s is a ground term

[22] Note that the abstract coefficient c_0 can only occur in atomic Diophantine constraints of the form "$p - c_0 \geqslant 0$" where the polynomial p does not contain c_0. These constraints are always satisfied when choosing c_0 small enough. Therefore, one does not have to consider constraints with c_0 anymore and one also does not have to determine the actual value of c_0. This is advantageous for ITRSs with "large constants" like $\mathsf{f}(\mathsf{true}, x) \to \mathsf{f}(\mathsf{1000} \geqslant x, x + \mathsf{1})$, since current Diophantine constraint solvers like [10] usually only consider small ranges for the abstract coefficients.

like $\mathsf{ack}(10, 10)$ on the right-hand side of a rule and all usable rules of s are non-overlapping, then one can replace s by a fresh variable y. This variable must then be added as an additional argument. In this way, one can transform the f-rule above into the following ones. Termination of the new ITRS is easy to show.

$$\mathsf{f}(\mathsf{true}, x) \to \mathsf{f}'(\mathsf{true}, x, y) \qquad\qquad \mathsf{f}'(\mathsf{true}, x, y) \;\to\; \mathsf{f}'(y \geqslant x, x + 1, y)$$

6 Experiments and Conclusion

We have adapted the DP framework in order to prove termination of ITRSs where integers are built-in. To evaluate our approach, we implemented it in our termination prover AProVE [15]. Of course, here we used appropriate strategies to control the application of the transformation rules from Sect. 4 and 5, since these are neither confluent nor equivalence-preserving. We tested our implementation on a data base of 117 ITRSs (including also *conditional* ITRSs, cf. Footnote 7). Our data base contains all 19 examples from the collection of [17] and all 29 examples from the collection[23] of [9] converted to integers, all 19 examples from the papers [2–8, 24, 25][24] on imperative programs converted to term rewriting, and several other "typical" algorithms on integers (including also some non-terminating ones). With a timeout of 1 minute for each example, the new version of AProVE with the contributions of this paper can prove termination of 104 examples (i.e., of 88.9 %). In particular, AProVE succeeds on all ITRSs mentioned in the current paper. In contrast, we also ran the previous version of AProVE (AProVE08) and the termination tool T$_\mathsf{T}$T$_2$ [22] that do not support built-in integers on this data base. Here, we converted integers into terms constructed with 0, s, pos, and neg and we added rules for the pre-defined operations on integers in this representation, cf. Sect. 1.[25] Although AProVE08 won the last *International Competition of Termination Provers* 2008 for term rewriting[26] and T$_\mathsf{T}$T$_2$ was second, both performed very poorly on the examples. AProVE08 could only prove termination of 24 of them (20.5 %) and T$_\mathsf{T}$T$_2$ proved termination of 6 examples (5.1 %). This clearly shows the enormous benefits of built-in integers in term rewriting. To access our collection of examples, for details on our experimental results, and to run the new version of AProVE via a web interface, we refer to `http://aprove.informatik.rwth-aachen.de/eval/Integer/`.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133-178, 2000.
2. A. R. Bradley, Z. Manna, and H. B. Sipma. Termination of polynomial programs. In *Proc. VMCAI'05*, LNCS 3385, pp. 113-129, 2005.
3. A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In

[23] In these examples, (multi)sets were replaced by lists and those examples where the use of (multi)sets was essential were omitted.

[24] We omitted 4 examples from these papers that contain parallel processes or pointers.

[25] In this way, one can always convert DP problems for ITRSs to ordinary DP problems.

[26] For more information, see `http://termcomp.uibk.ac.at/`.

Proc. CAV'05, LNCS 3576, pp. 491-504, 2005.
4. A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking abstractions. In *Proc. ESOP'08*, LNCS 4960, pp. 148-162, 2008.
5. M. Colón and H. B. Sipma. Synthesis of linear ranking functions. In *Proc. TACAS'01*, LNCS 2031, pp. 67-81, 2001.
6. M. Colón and H. B. Sipma. Practical methods for proving program termination. In *Proc. CAV'02*, LNCS 2034, pp. 442-454, 2002.
7. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *Proc. SAS'05*, LNCS 3672, pp. 87-101, 2005.
8. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI'06*, ACM Press, pp. 415-426, 2006.
9. S. Falke and D. Kapur. Dependency pairs for rewriting with built-in numbers and semantic data structures. In *Proc. RTA'08*, LNCS 5117, pp. 94-109, 2008.
10. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *Proc. SAT'07*, LNCS 4501, pp. 340-354, 2007.
11. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal termination. In *Proc. RTA'08*, LNCS 5117, pp. 110-125, 2008.
12. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation* 34(1):21-58, 2002.
13. J. Giesl, R. Thiemann, P. Schneider-Kamp. The DP framework: Combining techniques for automated termination proofs. *Pr. LPAR'04*, LNAI 3452, 301-331, 2005.
14. J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In *Proc. RTA'06*, LNCS 4098, pp. 297-312, 2006.
15. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. *Proc. IJCAR'06*, LNAI 4130, pp. 281-286, 2006.
16. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155-203, 2006.
17. J. Giesl, R. Thiemann, S. Swiderski, and P. Schneider-Kamp. Proving termination by bounded increase. *Proc. CADE'07*, LNAI 4603, pp. 443-459, 2007.
18. S. Gulwani and A. Tiwari. Constraint-based approach for analysis of hybrid systems. In *Proc. CAV'08*, LNCS 5123, pp. 190-203, 2008.
19. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172-199, 2005.
20. N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and features. *Information and Computation*, 205(4):474-511, 2007.
21. H. Hong and D. Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21(1):23-38, 1998.
22. M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. RTA'09*, LNCS, 2009. To appear.
23. E. Ohlebusch. Termination of logic programs: Transformational approaches revisited. *Appl. Algebra in Engineering, Comm. and Computing*, 12:73-116, 2001.
24. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. VMCAI'04*, LNCS 2937, pp. 239-251, 2004.
25. A. Podelski and A. Rybalchenko. Transition invariants. *LICS'04*, pp. 32-41, 2004.
26. A. Rubio. Present and future of proving termination of rewriting. Invited talk. `www.risc.uni-linz.ac.at/about/conferences/rta2008/slides/Slides_Rubio.pdf`
27. P. Schneider-Kamp, J. Giesl, A. Serebrenik, R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Tr. Comp. Log.*, 2009. To appear.
28. A. Serebrenik and D. De Schreye. Inference of termination conditions for numerical loops in Prolog. *Theory and Practice of Logic Programming*, 4 (5,6):719-751, 2004.